Introduction to Mathematical Programming

Theory and Algorithms of Linear and Nonlinear Optimization

Michael Kupferschmid

```
unix[1] cat students.m

function students(whom)
   printf('%s students\n',whom)
   students('and their')
end

unix[2] octave
octave:1> printf('\nthis book is dedicated to '); students('my')

this book is dedicated to my students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
and their students
:
```

Contents

0

Introduction

This book is about formulating mathematical models for optimization problems, solving the models by analytic techniques and iterative numerical algorithms, implementing the algorithms in computer programs, and using computational experiments to study how the programs behave.

0.1 Optimization

Who among us has not wished for an idyllic marriage, a flawless gemstone, or a house with four southern exposures? Alas, our happiness is often tempered by tradeoffs and constraints, and then instead of demanding perfection we must do the best we can. Sometimes this **optimization** takes only common sense, but many problems can benefit from a more systematic and quantitative approach.

"For since the fabric of the universe is most perfect and the work of a most wise Creator, nothing at all takes place in the universe in which some rule of maximum or minimum does not appear." – Leonhard Euler

The approach that we will use is based on an algebraic description or **mathematical model** of the optimization problem. In trying to find a best course of action we will ignore certain details and construct a simplified idealization that is just realistic enough to predict how the outcomes we care about depend on the actions we take.

"I fail every day.
Yet to victory am I born."
– Ralph Waldo Emerson

In life we often approach success only gradually, by making a sequence of mistakes that miss the mark by less and less; trial and error are essential in learning how to play the piano or how to bake bread, and if perfection is ever achieved it is on the very last try. To solve a mathematical optimization model it is usually also necessary to use trial and error, in the form of an **iterative algorithm** or numerical procedure that (we hope) produces, from each wrong answer that it finds, an answer that is closer to being right.

Because of the iterative nature of optimization algorithms, it is a practical necessity that their steps of logic and arithmetic be carried out automatically by a computer program.

0.2 About This Book

In 1988, early in my career at Rensselaer Polytechnic Institute, I attended a faculty meeting about combining two of our optimization courses and persuaded my colleagues that we should use the opportunity to introduce some material about numerical methods. Then I spent many years as Scientific Programming Consultant, helping graduate students and research faculty

with numerical computing, supervising thesis projects on optimization, publishing my own research, and teaching courses including Mathematical Models of Operations Research.

It was not until 2014 that I began teaching the course I had helped to design, which was by then called Computational Optimization. We used three very good textbooks [1] [4] [5], but together they did not quite cover the syllabus and the students claimed to prefer my notes. In Operations Research I always used the linear programming chapters from successive editions of [3], but the course gradually evolved away from that text and again I found the students relying more on my notes.

In 2015 I began this book to give my students typeset classnotes, so I hope they like it as much as they did the handwritten version and that other instructors find it useful for their students too. Because the notes were designed mainly to provide an easy introduction to the more comprehensive texts cited above, this book should be read to accompany those works rather than to replace them.

0.2.1 Audience

The courses in which I have taught this material enroll mostly juniors, seniors and first-year graduate students in mathematics, engineering, computer science, and finance, but postdocs and precocious sophomores have also found them worthwhile. I have assumed that readers will have some prior knowledge of computer programming and numerical methods as well as undergraduate mathematics, as detailed in §28. However, the computer programs presented in the book advance gradually from very simple to only moderately complex and they are all explained in detail, so students who have had even a superficial exposure to MATLAB can easily learn the coding along with the mathematics.

0.2.2 Pedagogical Approach

"So much of science proceeds by telling stories ... Even the most distant and abstract subjects ... fall within the bounds of necessary narrative." – S. J. Gould

TELL A GOOD STORY. Of all the wondrous tellings in science it is the never-ending story of mathematics, at once awesome in majesty and familiar as breakfast, that is surely the most beguiling. You can learn *about* it without ever having fallen in love, but learning *the thing itself* requires enough enchantment that you will cherish the tale and remember how it goes. In this book I have tried very hard to enchant you by weaving words, pictures, equations, graphs, code, and computational results into a clear and simple narrative. This is the only way I know how to teach, and if by the end I have succeeded you will not only know the subject but also love it as I do.

LET THE READER DISCOVER THE IDEAS. You will learn from this book if and only if you actually *read it.* Many pages are needed to tell the story of mathematical optimization, partly because there are many ideas in the subject and partly because you will remember only the ones that I help you discover for yourself. It will be obvious in many places that I am

trying not to spoil the plot by prematurely revealing what happens next. Serious students typically enjoy reading a story that is told in this way, but if you always look at the last page of a mystery first you might be happier using the Index to read the book in fragments. If a need for instant gratification makes you abandon this book entirely in favor of the internet, please remember that humbug often passes for wisdom on the web.

Use few proofs. A good proof can deepen our understanding and lead to fresh insights and valuable discoveries, and even a bad proof can (within fundamental limitations [119, p98]) persuasively establish the claims of its theorem. Mathematics often seems to progress by proving things. It is therefore tempting to explain linear programming by starting at the foundations of linear algebra and proving a succession of theorems concerning row-reduced echelon form; after that, all of the results that are needed for practical application follow trivially. To explain nonlinear programming it is similarly tempting to begin with precise definitions for differentiable and twice-differentiable functions and then prove a succession of theorems to build up the magnificent edifice of the Karush-Kuhn-Tucker theory; after that, the results that are needed for practical applications follow trivially. I have known a few students for whom this austere and lofty approach actually seemed to work, though none have ever used the word "trivial" in telling me about their struggles with it.

Many other students have told me, after studying optimization in that way, "I understood all of the proofs, but I never knew what any of them had to do with solving problems." *Rigor* can unfortunately be accompanied by *mortis,* and *formal*ity by the sharp odor of *formal*dehyde. Our focus will be on the practical use of ideas that are mostly quite simple and intuitive, and which I would rather have you understand from a plausibility argument than be distracted from by the technical details of a formal proof. I have therefore tried to make the exposition in this book so compelling and transparent that each discovery will seem, by the time we make it, obvious enough that no formal proof is required. If you want to learn how to construct proofs you should study books such as [1], [148], [8], and [136]. There are unfortunately a few places where I was driven to the heavy machinery because I felt unable to make the case in any other way, so the book does formally prove these eight theorems; I apologize for this lapse.

§	theorem
3.5.1	The set $\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ is convex.
3.5.2	The set of points that are optimal for a linear program is convex.
13.4.3	The BFGS update maintains symmetry of $\mathbf{B}$.
13.4.3	The BFGS result satisfies the secant equation.
13.4.3	if $\mathbf{U}$ is nonsingular, then $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is PD if and only if $\mathbf{M}$ is PD.
13.4.3	The BFGS update maintains positive definiteness of $\mathbf{B}$.
16.4	The KKT points of a convex program are global minima.
16.6	Convex constraints $f_i(\mathbf{x}) \leq 0$ have a convex intersection.

I have stated another nine important results in the form of theorems, listed on the next page, but without formal proof.

§	theorem
10.7	First-order necessary conditions for optimality.
10.7	Second-order necessary conditions for optimality.
10.7	Strong second-order sufficient conditions.
10.7	Weak second-order sufficient conditions.
11.3	Global minimizers.
11.3	Unique global minimizer.
13.4.4	The Sherman-Morrison-Woodbury formula.
15.2	Existence of Lagrange multipliers.
16.4	Existence of KKT multipliers.

The algorithms discussed in this book can be proved to converge under certain conditions, at least if we assume that they are implemented in perfect arithmetic. But the conditions are seldom satisfied and most of the methods work well enough to be useful even when they are not, so I have omitted formal proofs of convergence and cited other books where they can be found. Theorems of the alternative are charming but far from our focus on practical methods for numerical optimization, so I have also refrained from discussing those.

Use many examples. A colleague of mine was once lecturing in an abstract way on some mathematical topic when he was interrupted by a student. "Professor," the student asked, "could you please show us a specific example of what you are talking about?" The lecturer replied, flustered and annoyed, "Oh, very well, if you really want to get specific then let x equal some constant a." I found it hard to blame the student for dropping that course! Every general theory has its origin in particular problems, so to investigate mathematical optimization we will generalize from concrete numerical examples rather than trying to learn the general theory first and apply it after.

Engage the reader in conversation. In this book I refer to myself as "I" or "me" and to the reader as "you," but the personal pronoun that appears most often is "we." I am neither a king nor a pope and I do not have a mouse in my pocket, so "we" will always refer to the two of us together. Thus, for example, when I say in §8.1 "As in formulating a linear program, *we* begin by summarizing the data." I mean to suggest that you imagine the two of us working on the data summary together.

Use actual code. The difference between a clever idea and a hare-brained scheme is often in the details of carrying it out, so in explaining optimization algorithms it is essential to discuss their practical implementation in computer programs. To clarify the theory and animate the algorithms, I have tried to implement every method in working code. Usually this code is not sufficiently robust to serve as a numerical recipe for solving every problem. However, you should try to become as familiar with the example programs as with the mathematics and the prose, because in optimization all three are co-equal tools of discourse. To meet the needs of readers having different cognitive styles I have often used multiple representations of an algorithm, including pseudocode and flowcharts as well as MATLAB code and in-line comments, in describing its implementation.

0.2.3 Computing

To learn how to write your own programs you must write programs of your own, and the Exercises provide many opportunities for you to do that. But you will learn faster and develop better coding habits if you start by imitating the correct programs discussed in the text. I have assumed that you know a little about computer programming in a procedural language such as Java or C, and that you have at least watched someone else use base MATLAB. A typical optimization class includes at least a few students who are facile programmers, so if you are not you can ask one who is for help getting started. This book demands only meager programming skill, and it does not include any algorithms that involve the explicit manipulation of tree data structures (which computer scientists think of as the start of real programming).

Why did I spoil this otherwise wonderful book by not using your favorite programming language? My statistician friends wish I had chosen R, the computer scientists wish I had chosen C++, and the big-data mechanics can't imagine computing without Python. There are probably even engineers who would have preferred that the examples be written in FORTRAN. I suspect that whatever language I had chosen when I began, half of the readers would now want something else. My personal preference runs toward assembler language, so I had no emotional investment in picking a high-level computing environment and ended up using several.

Unix. This is the operating system most ardently championed by developers of software for scientific and technical applications, and its command-line interface makes it possible to show how it was used. I have therefore assumed that it, rather than Windows or Mac OS-X, is where the user asks the computer to run programs, as in this example from §26.3.4.

```
unix[1] ftn eacyc.f ea.f matmpy.f cse.f ek1.f getcyc.c
unix[2] a.out > ek1.e
```

Unix is used only a few times (in §3, §26, and §27) and each interaction is explained in detail. Unix is worth knowing, but you do not need to know anything about it to read this book. The applications that are described below can be used on any machine, not just on computers running Unix, and they work the same on all of them.

MATLAB OR OCTAVE. These are high-level software environments that can be used, either interactively or by writing programs, to do numerical calculations. MATLAB optionally includes toolboxes for a wide variety of tasks (including optimization) and it can be licensed from The Math Works, Inc. for most computers and operating systems. Octave [50] is a program that works like MATLAB except that it lacks the MATLAB toolboxes, and it can be downloaded free for most computers and operating systems. Octave has all of the functionality required for this book, and although it provides extensions to MATLAB I have been careful to avoid using them. My students have used both Octave and MATLAB with equal success, so whenever I refer to MATLAB in this book I will mean either MATLAB or Octave. The MATLAB programming in this book is not difficult, it is usually extensively

commented, and it is often explained line-by-line in the body of the text. To keep the programs easy to understand, I have observed the coding standards outlined in §28.4.

THE `pivot` PROGRAM. This is a learning tool for manipulating linear programming tableaus. All of the tableau operations described in this book can be performed by hand for the small examples we will study, but the pivot operation involves enough arithmetic to be tedious and once you have learned how to do it little is to be gained from additional practice. Homework papers in which the pivoting has been done by hand are typically rife with numerical errors, sometimes to the extent that the whole point of a problem is lost. Trivial mistakes deserve only little penalties, so the grader must spend a long time figuring out which errors are new at each step and which were propagated from earlier in the solution process. For many years I have therefore encouraged my students to use *some* program to automate that particular calculation, and because `pivot` can do many other operations as well I have used it extensively in this book. As discussed in §2.7 you can understand the examples without having or using the program, but if you want the code you can download it for free from the publisher's web site and install it on your computer.

The `pivot` program is written in classical FORTRAN and available only in source, so if your computing environment does not already include a FORTRAN compiler you will need to install one first by following the instructions in §27.2. Although the program will accommodate problems having up to 30 rows and 40 columns, neither its data structures nor its algorithms are of industrial strength so it is not meant to serve as a production linear program solver. The bones of the program are very old, so I also do not offer it as a paragon of design. If you have an idea for improving either the program or its manual in §27.1, please tell me so that I can make corrections and improvements in a future release of the program or a future edition of this text.

Maple OR Mathematica. These amazing symbolic algebra programs can analytically solve equations and inequalities, evaluate derivatives and integrals, and do arbitrary-precision arithmetic. In §8.2.4 I will show you how Maple works, but I have made scant use of it elsewhere and none at all of Mathematica or of the symbolic computation features of MATLAB, because all three programs are proprietary closed-source products with high license fees. As I write this the **Sage Math** open-source mathematical software system (see `sagemath.org`) has recently become available, and its wide-ranging capabilities might soon make it the preferred free alternative to these commercial offerings.

AMPL AND NEOS. AMPL[61] is a modeling language in which you can describe an optimization problem for solution by one of the canned packages that are available on the NEOS web server. I will show you in §8.3.1 how to use these utilities to solve nonlinear programs but, because our focus is on constructing algorithms rather than simply solving problems, they will play no other role in this book.

`gnuplot`. This program draws graphs from data. It is available free for many computing environments and sometimes it works better than the corresponding functions of Octave, so I will show you how to use it in §3.6.1 and thereafter use it a few times to draw graphs in three dimensions.

FORTRAN. For teaching numerical algorithms and experimenting with their implementation MATLAB or Octave is the ideal platform, but writing a production solver in FORTRAN [100, §0.3] or some other compiled language usually produces much faster machine code. We will use FORTRAN in §26.3 for studying the performance of optimization algorithms, but you don't need to know the language to read this book.

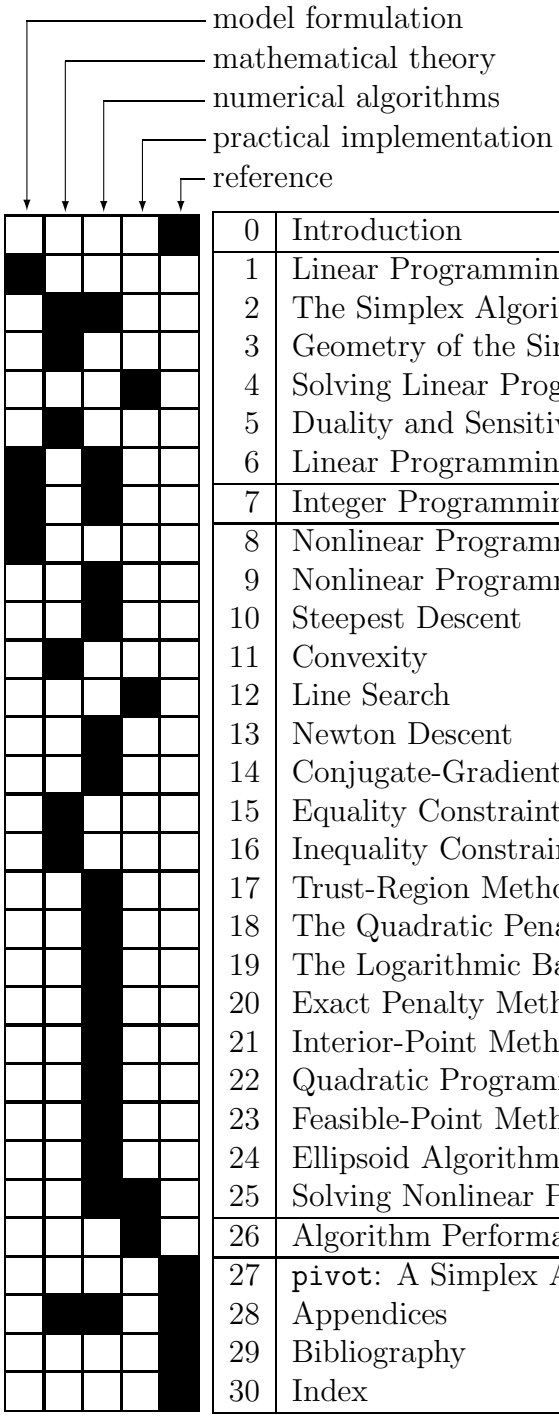
0.2.4 Coverage and Organization

According to its subtitle this book is about theory and algorithms of *linear optimization* and *nonlinear optimization,* so in the summary on the following page those two segments account for most of the Chapters. The one on nonlinear optimization can be subdivided into Chapters 10, 13, 14, and 17 on methods for unconstrained problems, Chapters 15 and 16 on the theory of constrained optimization, and Chapters 18–24 on methods for constrained problems. The Chapters on constrained problems can be further subdivided according to whether they describe methods for equality constraints (Chapter 18), inequality constraints (Chapters 19 and 21), or both (Chapters 20, 22, and 23). The checkerboard display shows how the material on model formulation, mathematical theory, numerical algorithms, and practical implementation is distributed through the Chapters.

According to its title this book is an *introduction,* so I have omitted some topics that are covered in some graduate courses, such as Lagrangian methods for integer programming and computing the rank-one update of a matrix by adjusting its triangular factors.

While many readers will be reassured by my focus on classical theory and methods, others might wish that I had written only about topics that have become fashionable much more recently. At the dawn of numerical optimization, computer memories were tiny and machine-readable data were scarce so the problems that people could actually solve did not have many variables. Little problems that are nice are not very interesting, so for many years the focus of research and algorithm development was on problems that are downright nasty. Much of what is known, and thus much of what you will learn from this book, has to do with solving models that are complex, unstructured, nonconvex, and nonsmooth, but not very large. As I finish this book in 2020, the problems that business and industry seem most eager to solve arise from the use of machine learning for data analytics. Most of these problems are theoretically very easy because they have a strictly convex objective and linear constraints, but they are practically very difficult because they have millions of variables. Unfortunately the techniques that work well for problems that are nasty mostly do not scale, because their storage requirements and running time grow quadratically with the number of variables. While most research in optimization was historically focused on developing sophisticated methods for solving small nasty models, it is now focused on the formulation of huge nice models tractable for very simple methods that scale linearly with problem size.

The techniques that are used for big-data problems are based on the classical methods, and many applications that are never mentioned on Fox News still give rise to problems that are of the traditional kind, so I have been loath to simply abandon the prior art in favor



					0	Introduction
■					1	Linear Programming Models
	■	■			2	The Simplex Algorithm
	■				3	Geometry of the Simplex Algorithm
			■		4	Solving Linear Programs
	■				5	Duality and Sensitivity Analysis
■		■			6	Linear Programming Models of Network Flow
■		■			7	Integer Programming
■					8	Nonlinear Programming Models
		■			9	Nonlinear Programming Algorithms
		■			10	Steepest Descent
	■				11	Convexity
			■		12	Line Search
		■			13	Newton Descent
		■			14	Conjugate-Gradient Methods
	■				15	Equality Constraints
	■				16	Inequality Constraints
		■			17	Trust-Region Methods
		■			18	The Quadratic Penalty Method
		■			19	The Logarithmic Barrier Method
		■			20	Exact Penalty Methods
		■			21	Interior-Point Methods
		■			22	Quadratic Programming
		■			23	Feasible-Point Methods
		■			24	Ellipsoid Algorithms
		■	■		25	Solving Nonlinear Programs
			■		26	Algorithm Performance Evaluation
				■	27	`pivot`: A Simplex Algorithm Workbench
	■	■		■	28	Appendices
				■	29	Bibliography
				■	30	Index

linear optimization: 1–6

nonlinear optimization: 8–25

of the new. The compromise that I have struck is to embed applications and algorithms that are essential to the big-data revolution into a conventional treatment of mathematical programming. The list below shows what these topics are and where they are discussed.

location	material most relevant to big-data problems
§1.5.1	minimizing the maximum
§1.5.2	minimizing the absolute value
§1.8	compressed sensing
§4.3	solving large linear programs
§8.6.5	regression on big data
§8.7.5	classification on big data
§16.6	convex programs
§16.9	duality in nonlinear programming
§20.2.4	the augmented Lagrangian method
§20.3	alternating direction methods of multipliers
§25.7	solving large nonlinear programs

I hope you will find that this book provides a useful introduction to techniques specifically useful in data analytics, along with a solid background in the mathematical theory and classical methods of optimization.

0.2.5 Typographical Conventions

PAGE HEADERS. Each right-hand (odd-numbered) page shows the title of the current Section or Subsection above its top rule, and the corresponding left-hand page shows the title of the Chapter or Section of which the Section or Subsection on the right-hand page is a part. For example, the header of this page shows the Subsection title **Typographical Conventions** while the header of the facing page shows the Section title **About This Book**. Although some parts come and go in the course of a page and thus never get mentioned at all, you might find the page headers (together with the Table of Contents) helpful in navigating through the book. In the text, "Section" can refer to either a Section or a Subsection.

KEY WORDS. An important word is printed **bold** on its first or defining appearance in the text but *slanted* in an Exercise, and it is an Index entry. Other Index entries are for ideas and concepts that might not be described in the text by a single key word.

REFERENCES. The literature citation [100, §4.6.1] is to section 4.6.1 in Bibliography reference 100, the book *Classical FORTRAN*. Context will often make it obvious whether a literature citation is given to suggest additional reading or to support a specific claim that is made in this book. The pages on which each citation appears are listed in §30.3.

EXERCISES. The final Section in each Chapter consists of questions on that Chapter, arranged in roughly the same order as the material to which they refer. Exercises marked `[E]` test only whether you recall what you have read, and can often be answered by quoting verbatim from the text; Exercises marked `[H]` test your comprehension of what you have read

and often require some hand calculation; Exercises marked `[P]` ask you to use a computer or write a program. Questions marked `[E]` are not always easy, and questions marked `[H]` are not always hard, but some of the `[H]` questions are much harder than others and a few are research problems to which I do not know the answer.

APPROXIMATE NUMBERS. Numbers that are stated as decimal fractions are sometimes imprecise. If a mathematical analysis yields an answer that is a formula and I round its exact value r to, say, 1.23 then I will write $r \approx 1.23$ to indicate that the decimal is not exact. If a computer calculation yields a value for the floating point variable `r` that rounds to `1.23`, I will write `r = 1.23` even though the value might be inexact because of the rounding or errors resulting from machine arithmetic or the infinitely-convergent nature of an algorithm. Outputs printed by computer programs will always be in `typewriter` font.

EXAMPLE PROBLEMS. This book includes many example optimization problems. I will give names to those that are referred to more than once, and collect all of the named problems in §28.5–§28.8. The page where each named problem is first mentioned is given in §30.1.

MATHEMATICAL SYMBOLS. Sometimes I will use $f(\alpha)$ to mean $f(\mathbf{x} + \alpha)$; otherwise the notation follows the prototypes in this table. The precise in-context meanings of variables are given in §30.2.

notation	meaning
s	a scalar
s_k	the value of s at iteration k
s^2	$s \times s$
$\mathbf{v}$	a column vector
$\mathbf{v}^\top$	a row vector
$\mathbf{v}^k$	the vector $\mathbf{v}$ at iteration k
v_j	the j'th element of $\mathbf{v}$ or of $\mathbf{v}^\top$
v_j^2	$v_j \times v_j$
$\mathbf{v}_i$	the i'th vector $\mathbf{v}$
$\mathbf{v}_i^k$	the vector $\mathbf{v}_i$ at iteration k
$[\mathbf{v}_i^k]_j$	the j'th element of $\mathbf{v}_i$ at iteration k
$\mathbf{0}$	a vector of all zeros
$\mathbf{1}$	a vector of all ones
$\mathbf{e}^j$	the j'th unit vector, zero except for 1 in element j
$\mathbf{M}$	an $m \times n$ matrix; a simplex tableau
$\mathbf{M}^\top$	the $n \times m$ transpose of $\mathbf{M}$
$\mathbf{M}^{-1}$	the inverse of a square matrix $\mathbf{M}$
$\mathbf{M}^{-\top}$	the transpose of the inverse of $\mathbf{M}$
$\mathbf{M}_k$	the matrix $\mathbf{M}$ at iteration k
M_i	the row vector that is the i'th row of the matrix $\mathbf{M}$

notation	meaning
$f(s)$	a scalar function of the scalar s
$f(\mathbf{v})$	a scalar function of the vector $\mathbf{v}$
$f(\mathbf{v}; p)$	a scalar function in which p is a fixed parameter
$\mathbf{f}(\mathbf{v})$	a vector function of the vector $\mathbf{v}$
$\mathbb{A}$	a set
$\lvert\mathbb{A}\rvert$	the cardinality of the set $\mathbb{A}$
$\mathbb{R}^n$	the space of n-vectors having real components
$\mathbb{R}^n_+$	the positive orthant of $\mathbb{R}^n$
$\mathbb{Z}^n$	the space of n-vectors having integer components
$\times$	scalar multiplication
⨳	a contradiction
✓	a confirmation
ℸ	the Hebrew letter resh
ס	the Hebrew letter samech
$\square$	the end of a proof or argument
$\longrightarrow$	the problem on the right is derived from the one on the left
$\longleftrightarrow$	the two optimization problems have the same optimal point

BOXES. Sometimes I will $\boxed{\text{box}}$ an important result in a complicated derivation for emphasis or so that I can refer to it (equations are not numbered). In line-by-line descriptions of computer programs, a boxed number such as $\boxed{\texttt{123}}$ refers to that line in the program's listing.

OTHER CONVENTIONS. Crosshatching in the graphical solution of an optimization problem indicates the feasible set. A "smooth" function is one that is sufficiently differentiable for the purpose at hand. A "function" can be either a mathematical function or a MATLAB subprogram. I will use "minimum" to refer, depending on context, to a minimizing point of an optimization problem or to the objective value at a minimizing point.

0.3 Teaching From This Book

A determined student can learn what this book has to teach by reading it and working the Exercises, but I hope that the book will also be required or recommended as a course text. The sample syllabi at the top of the next page are for the courses that gave rise to the book, and assume a 14-week semester with 2 class meetings per week all dedicated to instruction. These two courses, or two courses like them, are not big enough to cover all of the material in the book. Parts of the book can be used in other courses, serving different audiences and having different aims, as either a primary or an alternate text. Some possibilities are listed in the middle of the next page.

One approach to teaching this material is to recapitulate the book's exposition in class and expect the students to read the relevant Sections afterward. Another is to expect the students

Mathematical Models of Operations Research
mostly Juniors and Seniors

class	topics	reading
1	the idea of LP; graphical solution	1.1-1.2
2	static formulations	1.3
3	dynamic formulations	1.4
4	nonsmooth formulations	1.5-1.6
5	bilevel programs; compressed sensing	1.7-1.8
6	standard form and pivoting	2.1-2.3
7	canonical form and final forms	2.4-2.7
8	the subproblem technique	2.8.1
9	the method of artificial variables	2.8.2
10	getting standard form	2.9
11	graphical interpretation of pivoting	3.1-3.2
12	graphical interpretation of tableaus	3.3-3.4
13	convex sets	3.5
14	higher dimensions	3.6
15	implementing the simplex algorithm	4.1
16	the revised simplex method	4.2
17	large problems; software	4.3-4.4
18	convergence, degeneracy, and cycling	4.5
19	duality relations and shadow prices	5.1
20	finding duals; dual simplex method	5.2-5.3
21	sensitivity analysis	5.5
22	the transportation problem	6.1-6.2
23	transshipment; general network flows	6.3-6.4
24	explicit and implicit enumeration	7.1-7.2
25	branch-and bound for IP	7.3-7.4
26	zero-one programs	7.5
27	IP formulations; software	7.6-7.8
28	dynamic programming; complexity	7.8-7.9

Computational Optimization
mostly graduate students

class	topics	reading
1	nonlinear programming models	8.1-8.5
2	regression	8.6
3	classification; SVMs	8.7
4	NLP algorithms	9.1-9.6
5	steepest descent	10.1-10.8
6	convexity	11.1-11.6
7	bisection line search	12.1-12.2
8	Wolfe line search	12.3-12.4
9	Newton descent	13.1-13.3
10	quasi-Newton algorithms	13.4
11	the method of Lagrange	15.1-15.3
12	classifying Lagrange points	15.4-15.5
13	KKT; constraint qualifications	16.1-16.7
14	solution phenomena and duality	16.8-16.10
15	restricted steplength methods	17.1-17.2
16	trust-region algorithms	17.3-17.4
17	the quadratic penalty method	18.1-18.4
18	the logarithmic barrier method	19.1-19.4
19	exact penalty methods	20.1-20.2.3
20	augmented Lagrangian and ADMM	20.2.4-20.3
21	interior-point methods for LP	21.1-21.2
22	interior-point methods for NLP	21.3
23	feasible-point methods	23.1-23.2
24	space confinement	24.1-24.3
25	ellipsoid algorithms	24.4-24.8
26	solving nonlinear programs	25.1-25.5
27	approximating derivatives	25.6-25.7
28	algorithm performance	26.1-26.5

typical other course title	parts most likely to be of interest
Introduction to Optimization	1, 2, 6, 7, 8, 9, 10, 25, 26
Linear Programming	1, 2, 27, 3, 4, 5, 6, 21.1
Nonlinear Programming Fundamentals	8, 9, 10, 11, 13, 15, 16, 17
Nonlinear Programming Algorithms	8, 9, 11, 12, some from {13–24}, 25, 26
Network Optimization	1, 2, 3, 4, 5, 6
Data Analytics	1, 2, 4, 8, 15, 16, 20.3, 25.7
Numerical Methods	9, 10.6, 12, 18.4, 25, 26
Convex Analysis	3, 11, 15, 16, 24
Quadratic Programming	14, 18, 22
Integer Programming	1, 2, 3, 7
Analysis of Algorithms	4, 7, 26

to read the relevant Sections first and devote each class to a summary of the reading and a detailed study of one example (perhaps chosen from the Exercises so as to be different from those discussed in the text). The Exercises marked `[E]` can be used in short quizzes or graded homework to test whether a student has done the reading. Computing can (and ideally should) be made a part of the course by assigning Exercises marked `[P]`, or by assigning a term project, or by including hands-on programming in some classes.

0.4 About The Author

My professional life began in 1968 when I received a BS degree in electrical engineering from Rensselaer and went to work for Sikorsky Aircraft designing autopilots for military helicopters. After three years (and 53 test flights) I returned for an MEng degree in control systems engineering. Then I studied theatre engineering at the Yale School of Drama (Meryl Streep and Sigourney Weaver were also students at that time), became a licensed Professional Engineer, and designed controls for scenery-lifting winches at a little company that has since become part of the Wenger Corporation. There I also managed a group of technicians and drafters until 1978, when I returned yet again to Rensselaer set on a future in research and teaching (which in my innocence I imagined would *not* involve management). In 1980 I received an MS degree in operations research and statistics, and in 1981 the PhD for a thesis [98] about numerical optimization. Then I spent the next 34 years as a staff consultant and teacher of engineering and mathematics courses, eventually publishing 21 research articles in refereed journals. I also co-authored one textbook [3] first published in 1988 and wrote another on my own [100] first published in 2002, both of which are still in print. Now I hope to teach courses from this book, and to see it come into the widest possible use by students and by other instructors.

0.5 Acknowledgements

My gratitude begins with Don Schwendeman and Kristin Bennett, who made it possible for me finally to teach Computational Optimization a quarter of a century after I advocated for its introduction. I am grateful to Kristin and to Joe Ecker for sharing their classnotes, and to Joe for sharing the code he wrote for the course (though both the text and the code in this book ended up being quite different from either of theirs).

Next I must thank my Operations Research and Computational Optimization students for taking those courses and thereby helping me to perfect my own class notes, which as I have explained form the basis for this text. Drafts of the book have been used in those and other courses by Kristin Bennett, John Mitchell, Rong Ji Lai, and Yangyang Xu, eliciting valuable feedback from students including Joseph Hitchcock, Xiaoyan Lu, Miao Qi, Jonathan Reilly, and Yu Chen. John Mitchell suggested improvements to §1.8, §5.1.6, and several Exercises.

Some of the ideas in §26.2 and §26.3 came from work that was done by Steve Dziuban, David Covey, and Eric Johnson when they were my PhD students. The inspiration for the `pivot` command `Gnf` (see §27.1) was a class project by Scott Sacci, and a prototype of the `pivot` manual was a class project by Miranda Polin, Jen Karkoska, and Christine Goodrich. Dan Serino helped me with MATLAB.

Several friends who read parts of the book in draft pointed out errors or made other valuable suggestions, including Ken Miller, Matt Milone, Nancy Lawson, Hari Prasadh, Seth Lotts, and M. S. Krishnamoorthy.

Kevin Lewis worked many hardware miracles to keep my various antique laptop computers running for long enough to finish the project, and Erin Lynch emailed and printed many drafts.

While all of these people helped me and deserve a share of the credit for whatever you might like about this book, I must take the blame for any failures of judgement or other mistakes you find in it. I will of course be very happy to receive corrections or comments so that I can perfect the book in a later edition.

0.6 Disclaimers

Although I have tried very hard to ensure that everything in this book is correct, I cannot guarantee it (perhaps the author is the person *least* capable of issuing such a guarantee). I make no warranties, express or implied, that the mathematics, algorithms, or code contained in this book are free of error, or are consistent with any particular standard of merchantability, or that they will be suitable for any particular purpose. Both I and the publisher disclaim all liability for direct or consequential damages resulting from the use of anything you find in this book. The computer codes in particular are present only for instructional purposes and should not be relied upon for solving any problem whose incorrect solution could result in injury to a person, destruction of property, or loss of data. While you are welcome to all of the code, please be aware of its shortcomings and remember that you are using it *at your own risk.*

0.7 Exercises

0.7.1 [E] What is this book about?

0.7.2 [E] What is *optimization*?

0.7.3 [E] What is a *mathematical model* of an optimization problem?

0.7.4 [E] This book discusses two basic ways of solving optimization models. (a) What are they? (b) Can every problem be solved in both ways? Explain.

0.7.5 [E] When trial and error is used to solve an optimization model, what form does the process take? What makes a numerical algorithm *iterative*?

0.7.6 [E] Why is it usually necessary to use a computer program to perform the steps of an optimization algorithm?

0.7.7 [E] When I began writing this book, several very good texts about linear and nonlinear optimization were already in print. Why do I think this book might be a worthwhile supplement to them?

0.7.8 [E] Who are the audience for this book? How much computing background do you need in order to read it?

0.7.9 [E] List the main features of the pedagogical approach that I used in writing this book. Why do I try to help you discover the ideas for yourself?

0.7.10 [E] What is the role of proof in this book? How many theorems are formally stated, and how many are proved? Why was it necessary to include these theorems and proofs? Have I proved the convergence of the algorithms discussed in the book?

0.7.11 [E] Which usually comes first in this book, a general theory or a specific example?

0.7.12 [E] When the text says "we" to whom is it referring?

0.7.13 [E] Why are the algorithms in this book implemented in working code?

0.7.14 [E] In discussing optimization theory and algorithms I will use three basic forms of expression. What are they? What are the different representations for an algorithm that I will use in describing its implementation?

0.7.15 [E] What computing background have I assumed you will have as you begin reading this book? Where can you find help in getting started with the computer programming required by this book?

0.7.16 [E] List the computing environments used in this book. Why did I choose Unix, rather than Windows or Mac OS-X, as the operating system to assume in examples that involve using one?

0.7.17 [E] How do MATLAB and Octave differ?

0.7.18 [E] Does this book make any use of the MATLAB Optimization Toolbox? Does it use any of the extensions that Octave makes to MATLAB?

0.7.19 [E] Describe the `pivot` program. Where can you find instructions telling how to install the program if you want to have it? Do you need to install it on your computer in order to understand the examples in this book?

0.7.20 [E] How do Maple and Mathematica differ from Octave and base MATLAB?

0.7.21 [E] What are AMPL and NEOS, and why do they play only a small role in this book?

0.7.22 [E] How is `gnuplot` used in this book? Find out how to get it for your computer, and explain the procedure.

0.7.23 [E] Why is FORTRAN usually preferable to MATLAB or Octave as a language for writing production optimization software? Do you need to know FORTRAN to read this book?

0.7.24 [E] The content summary of §0.2.4 divides the Chapters of this book into 6 segments. What are they? Which Chapters include material relating to the practical implementation of optimization algorithms?

0.7.25 [E] Research in optimization used to be focused on developing sophisticated methods for small nasty models, but it is now focused on the formulation of huge nice models that are tractable for very simple methods. Why?

0.7.26 [E] Where does this book discuss topics that are of interest for the solution of optimization problems involving big data? Why does the book also discuss methods for solving traditional models that do not involve big data?

0.7.27 [E] Explain how to navigate through this book by using (a) the page headers; (b) the Table of Contents; (c) the Index.

0.7.28 [E] What does it mean when a word is printed in **bold** type?

0.7.29 [E] Each Exercise in this book is marked [E] or [H] or [P]. What do these designations mean? Which category consists of questions that might be included in a reading quiz to test a student's recall?

0.7.30 [E] If the text says $r = 1.23$, is the value given exactly? If the text says `r=1.23`, is the value given exactly? Explain.

0.7.31 [E] The optimal objective value of the `ek1` problem is given approximately as 614.2 in §24.2. Where can you find its value precise to machine precision?

0.7.32 [E] What does the symbol ⋘ denote?

0.7.33 [E] What does a boxed number such as $\boxed{123}$ denote?

0.7.34 [E] Do the mathematical results, algorithm descriptions, or computer code in this book come with any sort of warranty? Explain.

0.7.35 [H] If you find a mistake in the book, how can you report it to the author? Hint: read the verso on the back of the title page.

1

Linear Programming Models

We begin, as mathematics often begins, with a story.

Two of the courses in which David is enrolled have their first exams next week. He is already confident that he knows 2 of the 5 textbook sections to be covered by the Linear Programming exam, but in dark moments of terror and self-reproach he is forced to admit that he has so far learned nothing at all about Art History. He estimates that he can master the remaining Linear Programming sections if he spends 3 hours studying the book and 2 hours working problems, but to catch up in Art History he needs to devote 10 hours to learning his class notes and visiting the on-line gallery. He hopes to get the highest grades he possibly can, but to avoid having an alert sent to his advisor he must score at least 60% on each exam. Unfortunately, his family commitments and other courses leave him only 12 hours to prepare for these exams. What should he do?

1.1 Allocating a Limited Resource

David has already learned enough from his Linear Programming course to recognize his problem as an **optimization**. His goal, stated more precisely, is to maximize the *sum* of the two exam scores, but because his time for study is a limited resource there is a tradeoff *between* the two scores; the only way he can do better on one exam is by doing less well on the other.

He cannot directly control the scores he will get but he can control the allocation of his study time, so to describe the problem mathematically he identifies these **decision variables**.

$$\begin{aligned} x_1 &= \text{hours spent studying for Linear Programming} \\ x_2 &= \text{hours spent studying for Art History} \end{aligned}$$

If he already knows $\frac{2}{5}$ of the Linear Programming material he could score 40% on that exam without any further study at all, and if 5 hours are enough to learn the rest then studying for x_1 hours should allow him to achieve a score of

$$s_1 = 40 + 60 \times \tfrac{1}{5}x_1 = 40 + 12x_1.$$

If 10 hours are enough to learn all of the Art History that will be tested, then studying for x_2 hours should allow him to achieve a score of

$$s_2 = 100 \times \tfrac{1}{10}x_2 = 10x_2.$$

The scores s_1 and s_2 are **state variables**, because they depend on x_1 and x_2 and are useful in describing the problem but they are not themselves decision variables. In this problem what makes them important is that the quantity to be maximized is their total $T = s_1 + s_2$.

The statement of David's problem includes conditions that must be satisfied by any solution. They can be expressed in terms of the decision variables and state variables like this.

$$\left.\begin{array}{rcl} s_1 & \geq & 60 \\ s_2 & \geq & 60 \end{array}\right\} \text{ avoid unwanted attention from advisor}$$
$$x_1 + x_2 \leq 12 \qquad \text{meet other obligations}$$

Additional conditions, while not given explicitly in the problem statement, are implied by the story or demanded by common sense.

$$\left.\begin{array}{rcl} s_1 & \leq & 100 \\ s_2 & \leq & 100 \end{array}\right\} \text{ can't get better than a perfect score}$$
$$\left.\begin{array}{rcl} x_1 & \geq & 0 \\ x_2 & \geq & 0 \end{array}\right\} \text{ can't study for less than 0 hours}$$

Now David knows what to do: he should study Linear Programming for x_1 hours and Art History for x_2 hours, where x_1 and x_2 are chosen so that all of these conditions are satisfied and T is as high as possible. But how can he find those values of x_1 and x_2?

1.1.1 Formulating the Linear Program

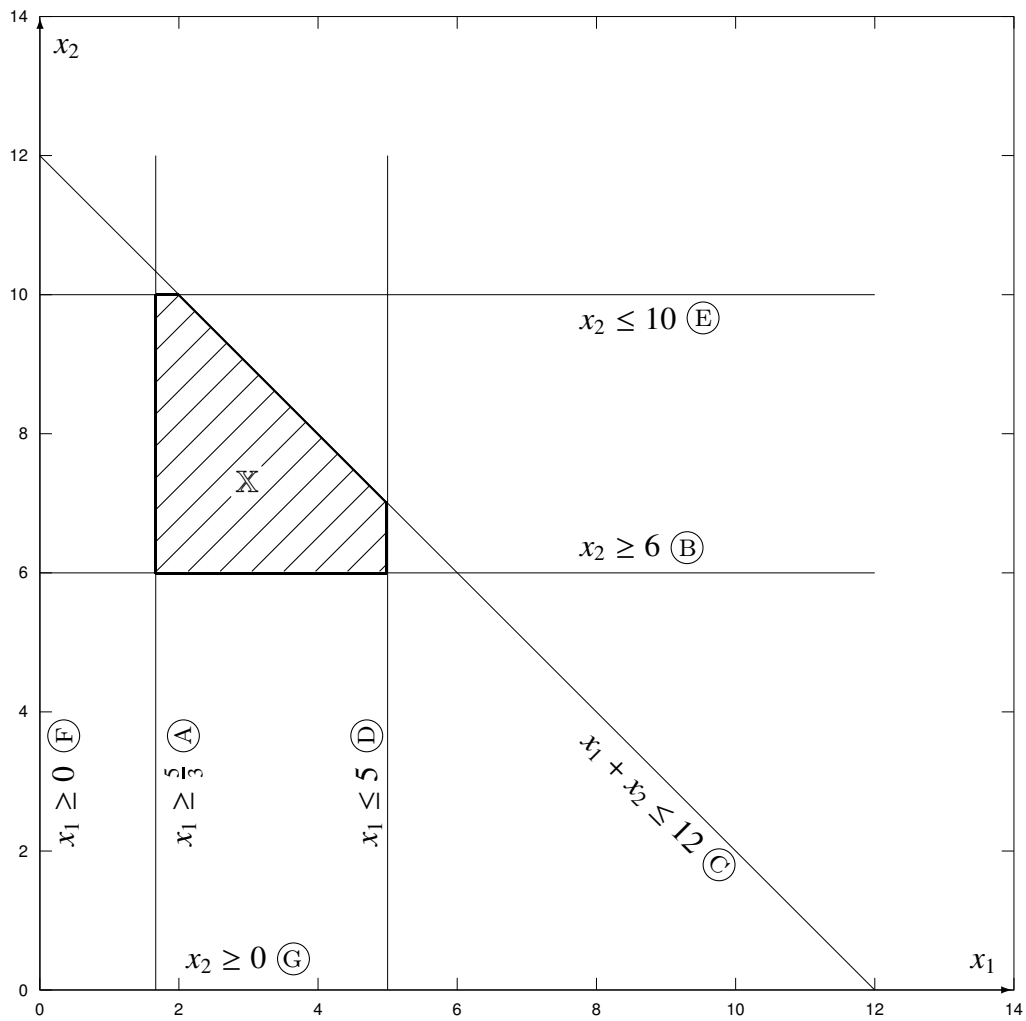
The analysis above can be summarized algebraically in the form of this **mathematical program**, which I will call the `twoexams` problem (see §28.5.1).

$$\begin{array}{lrcrcrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & 40 & + & 12x_1 & + & 10x_2 & = & T \\ \text{subject to} & 40 & + & 12x_1 & & & \geq & 60 \quad \text{(A)} \\ & & & & & 10x_2 & \geq & 60 \quad \text{(B)} \\ & & & x_1 & + & x_2 & \leq & 12 \quad \text{(C)} \\ & 40 & + & 12x_1 & & & \leq & 100 \quad \text{(D)} \\ & & & & & 10x_2 & \leq & 100 \quad \text{(E)} \\ & & & x_1 & & & \geq & 0 \quad \text{(F)} \\ & & & & & x_2 & \geq & 0 \quad \text{(G)} \end{array}$$

In a mathematical program an **objective function** is maximized or minimized subject to side conditions or **constraints**, which can be inequalities or equalities. Because the objective and constraint functions in this mathematical program are all linear in the decision variables, it is called a **linear program**.

1.1.2 Finding the Optimal Point

This linear program might seem daunting because it requires us to find values of x_1 and x_2 that satisfy the seven constraint inequalities (A) – (G) simultaneously. But because this problem has only two decision variables we can graph its **feasible set** $\mathbb{X}$, crosshatched below, which contains *all* such **feasible points**.

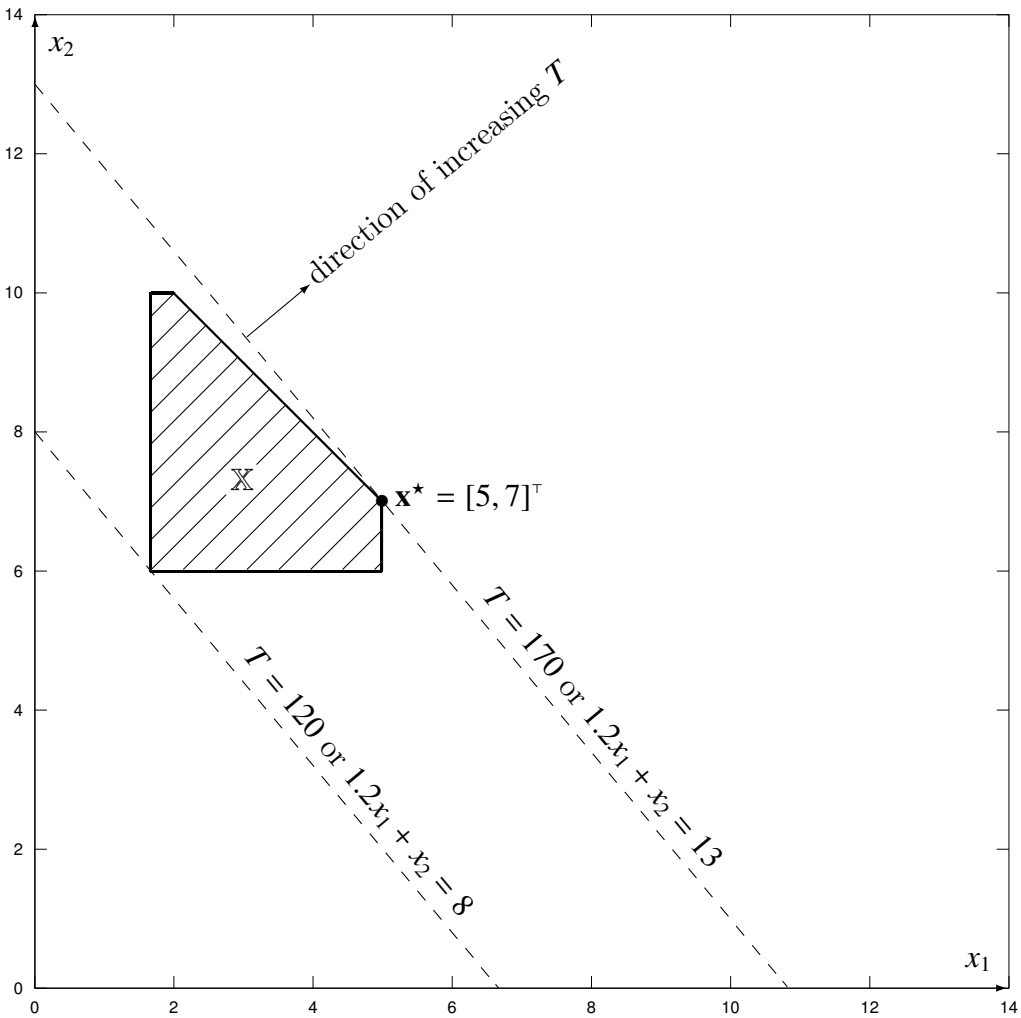


The **nonnegativity constraints** $x_1 \geq 0$ and $x_2 \geq 0$, represented respectively by the x_2 and x_1 coordinate axes in this graph, confine the feasible set to the first quadrant. The constraint on study time, $x_1 + x_2 \leq 12$, rules out points above the diagonal line. The vertical lines are the limits on x_1 that must be enforced to ensure that $60 \leq s_1 \leq 100$, and the horizontal lines are the limits on x_2 that must be enforced to ensure that $60 \leq s_2 \leq 100$. In this problem the nonnegativities are **redundant constraints** because they do not affect the feasible set.

Now to solve the linear program we need only select, from among all the points in $\mathbb{X}$, one that maximizes the objective function

$$T = s_1 + s_2 = 40 + 12x_1 + 10x_2.$$

For a given value of T, this equation describes an **objective contour** that we can plot along with the feasible set. In the picture below I have drawn one objective contour through the point $[\frac{5}{3}, 6]^\top$ where $T = 120$, and another through $[5, 7]^\top$ where $T = 170$.



The objective contours are parallel to one another and as we increase T they move up and to the right. The feasible point yielding the highest objective value is thus the corner of $\mathbb{X}$ marked $\mathbf{x}^\star$, and David's optimal test preparation program is to spend $x_1 = 5$ hours studying Linear Programming and $x_2 = 7$ hours studying Art History; this will allow him to earn exam scores of $s_1 = 100$ and $s_2 = 70$. He could do better in Art History by choosing a feasible point with a higher x_2, but only by decreasing x_1 and settling for lower values of s_1, and T.

1.1.3 Modeling Assumptions

In formulating his time allocation problem as a linear program, David made several important idealizing approximations. This is inevitable whenever we attempt a conceptually simple description of our inherently complicated world. Often the assumptions we find it necessary or convenient to make are also quite reasonable, and then they can lead to a realistic and useful **mathematical model**, but always it is prudent to remember what they were.

The most obvious assumptions underlying the `twoexams` linear programming model are David's estimates about how much of the Linear Programming material he already knows, how long it will take him to learn the rest, and how long it will take him to catch up in Art History. Experienced students often make good guesses about such things, but sometimes they guess wrong. In other settings the coefficients and constants in a linear programming model might be uncertain statistical estimates from data, arbitrary numbers specified by some authority, or the results of theoretical calculations concerning a natural phenomenon.

The objective and constraint functions of the `twoexams` model are linear in x_1 and x_2, and this implies strict proportionality of the output T to each of those inputs. Each minute spent on study is assumed to produce the same increment in knowledge and understanding, even though in reality comprehension grows more quickly in the middle of learning a topic than it does at either end and fatigue makes the first minute of study more effective than the last. The credit on each exam is assumed to be uniformly distributed over the material to be covered, so that knowing $p\%$ of it results in a grade of $p\%$, even though some topics typically carry more weight than others and instructors do not always accurately disclose exam content. Exam performance is assumed to depend only on student knowledge and understanding, but other factors such as anxiety and distraction can also play a role. The credit that will be given is assumed to be precisely proportional to the knowledge displayed, but in practice exams are organized into parts and the distribution of partial credit might not be smooth.

In a linear program $\mathbf{x}$ is a real variable, so we implicitly assumed that study time is infinitely divisible even though we know that David probably won't measure it with split-second precision. The optimal point we found for `twoexams` has components that happen to be whole numbers, but that was just a coincidence. In other settings the decision variables count discrete things rather than measuring something continuous, and then using linear programming entails the assumption that rounding the continuous solution gets close enough to the right count. This might be a good approximation if a decision variable represents the number of grains in a corn silo but a bad one if it represents the number of silos on a farm. *Insisting* that a mathematical program have whole number solution components turns it into a much more difficult **integer linear program** or **integer nonlinear program** (see §7).

If the numbers in the `twoexams` problem had been a little different, its feasible set $\mathbb{X}$ might have been empty so that the problem was **infeasible**. If this possibility did not cross David's mind as he wrote down the linear program, then feasibility was another thing he unwittingly assumed.

1.1.4 Solution Techniques

The solution to a mathematical program is an **optimal vector $\mathbf{x}^\star$** whose components are the best possible values of the variables. Together these numbers specify an ideal plan of action or optimal program of things to do, and that is the origin of the name "mathematical programming." Certain mathematical programs can be solved using analytical methods that were discovered long before the digital computer was invented, but others can be solved only by numerical methods implemented in computer programs. Thus, while the discipline of mathematical programming preceded that of computer programming, there is an intimate connection between the two and they have developed together [36]. This book is about mathematical programs, analytical and numerical methods for solving them, and computer programs that implement the numerical methods.

In §1.1.2 we solved the `twoexams` problem graphically, and throughout the book we will often study examples that have one, two, or three variables by drawing a graph (see the Index entry for "graphical solution"). This approach gives so much insight into linear programming that I have devoted the next Section and all of §3 to the construction and interpretation of graphical solutions.

Real mathematical programs typically have more than three variables, and then it is necessary to use analytic or numerical solution techniques. In §2 we will take up the simplex algorithm for solving linear programs, and we will write and begin using numerical software to implement it. As we explore the theory and methods of linear optimization the examples that we consider will often be divorced from the applications that gave rise to them, so before we leave the topic of linear programming models we will consider several formulation techniques in §1.3–§1.6, a survey of applications in §1.7, and in §1.8 one important application that is currently of great interest.

1.2 Solving a Linear Program Graphically

The procedure outlined below can be used to solve any linear program that has inequality constraints and two (or with obvious extensions three) variables. Several features of the graphical solution that are referred to here in an informal way will be given more precise definition in §3.

To begin the solution process you need an algebraic statement of the linear program, a sheet of graph paper, and a straightedge. If the variables are nonnegative the feasible set will be in the first quadrant, but for convenience in plotting constraints it might be useful to extend the axes to negative values. Experiment with the axis scales to find good ones.

Plot each **constraint contour** as the line where the constraint holds with equality; the inequality will be satisfied on one side and violated on the other. If $x_1 = 0$, what is x_2? If $x_2 = 0$, what is x_1? If the answers are not the origin, draw a line between the intercepts; if setting $x_1 = 0$ makes $x_2 = 0$ then write the constraint as $x_2 = mx_1$ and plot that line through the origin. Draw hash marks perpendicular to each inequality to show which side is feasible;

you can find out by picking a point (such as the origin) on one side or the other and asking "does this point satisfy the constraint?"

The constraint inequalities partition the x_1–x_2 plane into windowpanes, some of them extending off the page. Figure out which *one* windowpane is feasible for *all* of the inequalities, and outline or crosshatch it. This feasible set is the intersection of the constraint sides on which you drew hash marks. No constraints cross the interior of a feasible set. To verify that you have identified the feasible set, pick a point inside it (not a corner) and evaluate the constraint functions numerically to show that all of the inequalities are satisfied there.

Plot a trial contour of the objective function. To do this evaluate the objective at some corner of the feasible set; then plot a dashed line, passing through that corner, on which the objective has that value.

Find the optimal point. Translate the objective contour you drew parallel to itself in the direction that maximizes or minimizes the objective (whichever is required) until its intersection with the feasible set is a single point or an edge. That point or edge is optimal; label it. The point or edge obtained by translating the objective contour in the *other* direction will minimize the objective if you found its maximum, or maximize it if you found its minimum. You can check your work by evaluating the objective at both extreme corners, or at all corners, of the feasible set. Find the coordinates of the optimal point algebraically, by solving simultaneously the equations of the inequalities that intersect there.

Plot the optimal objective contour, if the trial contour you drew before does not happen to go through the optimal point. Evaluate the objective at the optimal point and plot a dashed line through it on which the objective has that value. The optimal objective contour cannot cross the interior of the feasible set.

If the linear program is infeasible ($\mathbb{X}$ is empty) or unbounded (which we will study in §2.5.2) then it has *no* solution, and this procedure will also reveal that fact.

1.3 Static Formulations

To construct a mathematical programming model for any optimization, we can proceed as we did in analyzing the `twoexams` problem.

1. Summarize the facts in a way that makes them easy to understand. If the problem is simple a concise statement in words might be good enough, but often it is helpful to organize the data in a table or diagram.

2. Identify decision variables. These always quantify the things we can directly control.

3. State the constraints mathematically. Remember to include **obvious constraints** such as nonnegativities and **natural constraints** such as that there are 24 hours in a day or that 100% of something is all of it.

4. State the objective mathematically. What is to be minimized or maximized?

1.3.1 Brewing Beer

When barley is allowed to partially germinate and is then dried, it becomes malt. When malt is crushed and mixed with water, boiled with hops, and fermented with yeast it becomes the delightful beverage we call beer. Sarah operates a local craft brewery that makes Porter, Stout, Lager, and India Pale Ale beer by using different amounts of pale malt, black malt, and hops. For example, to make 5 gallons of Porter requires 7 pounds of pale malt, 1 pound of black malt, and 2 ounces of hops, and the finished keg can be sold for $90. The **technology table** below summarizes the resource requirements and anticipated revenue for all four varieties, along with the stock on hand of each ingredient.

	Porter	Stout	Lager	IPA	stock
pale malt	7	10	8	12	160 lb
black malt	1	3	1	1	50 lb
hops	2	4	1	3	60 oz
revenue	$90	$50	$0	$70	

How much of each product should Sarah make to maximize her revenue?

1. The first step in the formulation procedure of §1.3.0 is to summarize the facts, and this has already been done in the technology table above.

2. What Sarah controls is how much of each product she will make, so the decision variables are

$$\begin{aligned} x_1 &= \text{kegs of Porter to make,} \\ x_2 &= \text{kegs of Stout to make,} \\ x_3 &= \text{kegs of Lager to make, and} \\ x_4 &= \text{kegs of IPA to make.} \end{aligned}$$

3. Sarah's revenue increases as she sells more beer so ideally $x_j = +\infty$ for $j = 1 \ldots 4$, but the limited stock of ingredients makes this plan infeasible. For example, a production program $[x_1, x_2, x_3, x_4]^\top$ requires $7x_1 + 10x_2 + 8x_3 + 12x_4$ pounds of pale malt, but only 160 pounds are in stock. To keep from using more supplies than she has, Sarah must choose x_1, x_2, x_3, and x_4 so that

$$\begin{array}{rcrcrcrcr} 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & \leq & 160 \\ 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & \leq & 50 \\ 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & \leq & 60. \end{array}$$

The amount of each beer variety produced can't be negative, so the obvious constraints $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$, $x_4 \geq 0$ must also be satisfied by an optimal production program.

4. Sarah's goal is to maximize her total revenue $90x_1 + 150x_2 + 60x_3 + 70x_4$.

Thus we can state the `brewery` problem (see §28.5.2) as the following linear program

$$
\begin{array}{lrcrcrcrcl}
\underset{\mathbf{x}\in\mathbb{R}^4}{\text{maximize}} & 90x_1 & + & 150x_2 & + & 60x_3 & + & 70x_4 & & \\
\text{subject to} & 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & \leq & 160 \\
 & 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & \leq & 50 \\
 & 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & \leq & 60 \\
 & x_1 & & & & & & & \geq & 0 \\
 & & & x_2 & & & & & \geq & 0 \\
 & & & & & x_3 & & & \geq & 0 \\
 & & & & & & & x_4 & \geq & 0
\end{array}
$$

Because x_1, x_2, x_3, and x_4 are real variables, this formulation assumes that fractional amounts of each variety can be made. Later we will find that the optimal solution to this problem is $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\intercal$, in which the amount of Stout to be made is not a whole number of kegs (see §7.1).

1.3.2 Coloring Paint

A chemical company has developed two batch processes for making pigments. Both processes use feedstocks designated A, B, and C, but each is based on a different sequence of reactions. The RB process produces a final product called RED, but at an intermediate stage it incidentally yields some BLUE as a byproduct. The BR process produces mostly BLUE, with RED as a byproduct. One batch of the RB process uses 5 liters of A, 7 liters of B, and 2 liters of C to produce 9 liters of RED and 5 liters of BLUE, while one batch of the BR process uses 3 liters of A, 9 liters of B, and 4 liters of C to produce 5 liters of RED and 11 liters of BLUE. A paint company has offered to buy as much product as the chemical company can make, at \$6 per liter of RED and \$12 per liter of BLUE, but it insists that at least half of the shipment be RED. The chemical company has on hand 1500 liters of A, 2520 liters of B, and 1200 liters of C. How should it use this inventory of feedstocks to maximize its revenue?

1. The problem description includes a welter of details, so we begin by organizing them in the technology table below.

feedstock type	feedstock used		feedstock available
	RB process	BR process	
A	5	3	1500
B	7	9	2520
C	2	4	1200
RED	9	5	\$6
BLUE	5	11	\$12
pigment color	RB process	BR process	revenue per liter
	product produced		

2. Unlike the brewery, the chemical company does not directly control how much of each product it makes; it only controls how many batches of the two products it makes by each process.

$$\begin{aligned} x_1 &= \text{runs of the RB process to make} \\ x_2 &= \text{runs of the BR process to make} \end{aligned}$$

3. Like the brewery, the chemical company cannot use more inputs than it has. For example, making x_1 runs of the RB process and x_2 runs of the BR process will use $5x_1 + 3x_2$ liters of feedstock A, but only 1500 liters are on hand. To keep from using more than its supply of each feedstock, the chemical company must choose x_1 and x_2 so that

$$\begin{array}{rcrcl} 5x_1 & + & 3x_2 & \leq & 1500 \\ 7x_1 & + & 9x_2 & \leq & 2520 \\ 2x_1 & + & 4x_2 & \leq & 1200. \end{array}$$

Making x_1 runs of the RB process and x_2 runs of the BR process will produce $r = 9x_1 + 5x_2$ liters of RED and $b = 5x_1 + 11x_2$ liters of BLUE. The customer's requirement that at least half the total product shipped be RED means that

$$\frac{r}{r+b} = \frac{9x_1 + 5x_2}{14x_1 + 16x_2} \geq \frac{1}{2}.$$

As it stands this **ratio constraint** is nonlinear, but unless $r + b = 0$ we can rewrite it as a linear inequality.

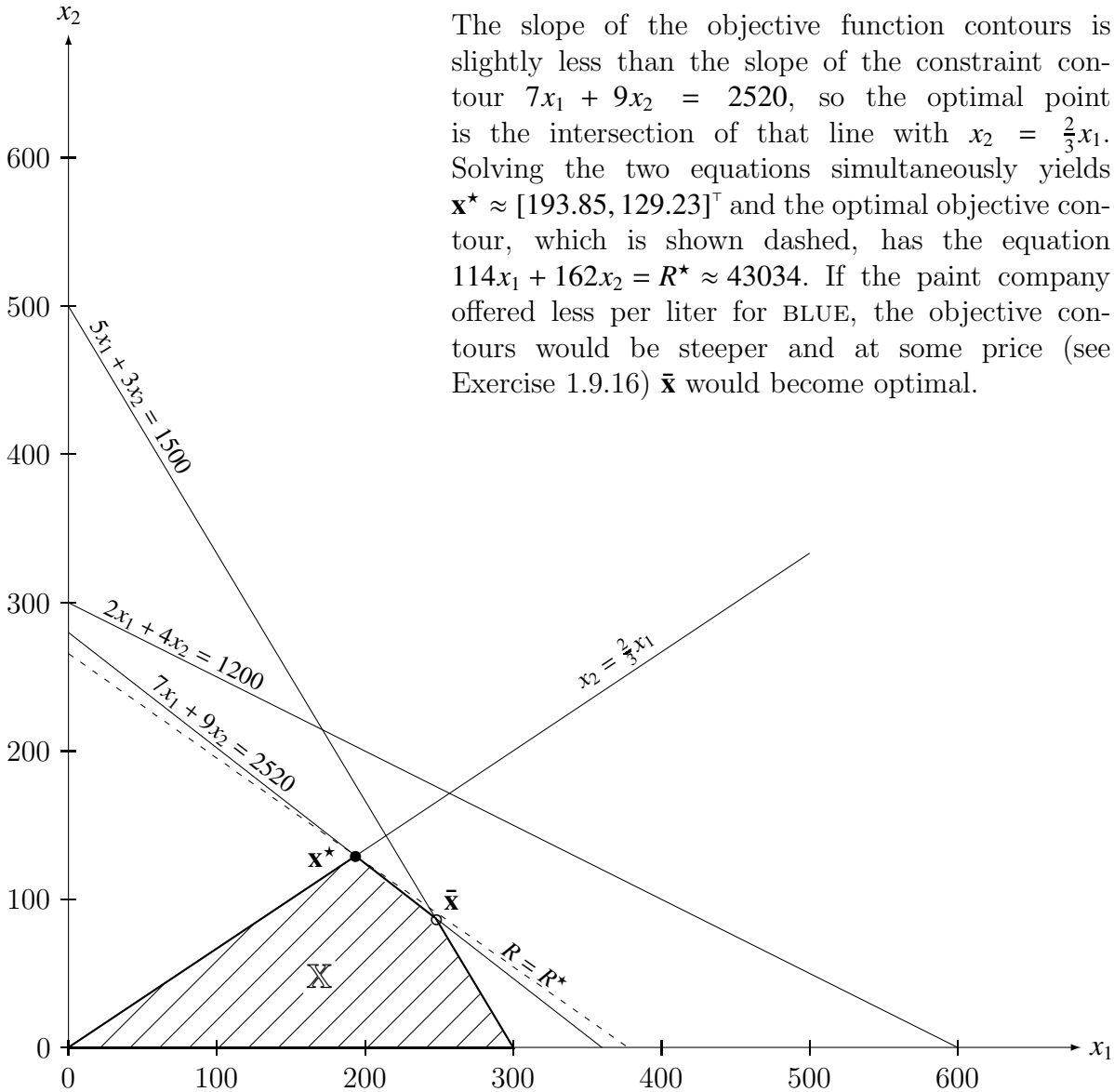
$$\begin{array}{rcl} 18x_1 + 10x_2 & \geq & 14x_1 + 16x_2 \\ 4x_1 & \geq & 6x_2 \end{array}$$

4. The chemical company wants to maximize its revenue $R = 6r + 12b = 114x_1 + 162x_2$.

Including nonnegativity constraints, we can state the `paint` problem (see §28.5.3) as this linear program.

$$\begin{array}{lrcrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & 114x_1 & + & 162x_2 & = & R \\ \text{subject to} & 5x_1 & + & 3x_2 & \leq & 1500 \\ & 7x_1 & + & 9x_2 & \leq & 2520 \\ & 2x_1 & + & 4x_2 & \leq & 1200 \\ & 2x_1 & - & 3x_2 & \geq & 0 \\ & x_1 & & & \geq & 0 \\ & & & x_2 & \geq & 0 \end{array}$$

This problem has only two variables so I solved it graphically by following the procedure given in §1.2, obtaining the picture on the next page.



The slope of the objective function contours is slightly less than the slope of the constraint contour $7x_1 + 9x_2 = 2520$, so the optimal point is the intersection of that line with $x_2 = \frac{2}{3}x_1$. Solving the two equations simultaneously yields $\mathbf{x}^\star \approx [193.85, 129.23]^\top$ and the optimal objective contour, which is shown dashed, has the equation $114x_1 + 162x_2 = R^\star \approx 43034$. If the paint company offered less per liter for BLUE, the objective contours would be steeper and at some price (see Exercise 1.9.16) $\bar{\mathbf{x}}$ would become optimal.

The third constraint $2x_1 + 4x_2 \leq 1200$ does not affect the feasible set, so it is redundant and could be removed from the problem without changing the answer.

The phrasing of the problem statement suggests that the number of batches run using each process should be a whole number, but both components of $\mathbf{x}^\star$ have fractional parts. Rounding each to the nearest integer yields $\hat{\mathbf{x}} = [194, 129]^\top$, which happens to be the optimal integer point for this problem. In general, rounding each component in the solution of a linear program to the nearest whole number can yield a point that is infeasible or that is feasible but *not* the optimal integer point. To be sure of finding the optimal integer point for a mathematical program it is necessary to use the techniques of §7.

1.4 Dynamic Formulations

Many optimization problems involve an ordered sequence of decisions each of which is somehow affected by those that came before it [151, §2.6]. The key to formulating such a problem as a mathematical program is often a **conservation law** that holds at the beginning of each stage in the process being modeled. Finding such a law can reveal precisely what it is that we control and hence what the decision variables ought to be.

1.4.1 Scheduling Shift Work

The number of airplanes that are in flight varies with the time of day, so the number of people who are needed to staff an air traffic control center varies by work period. If a center has the following daily staff requirements and each controller works for two consecutive periods, how can the schedule be covered with the minimum number of controllers?

work period		controllers needed
j	time interval	r_j
1	0000-0300	3
2	0300-0600	6
3	0600-0900	14
4	0900-1200	18
5	1200-1500	16
6	1500-1800	14
7	1800-2100	12
8	2100-2400	6

1. The number of workers present is governed the following conservation law.

$$\begin{array}{c}\text{number of controllers}\\ \text{working during period } j\end{array} = \begin{array}{c}\text{number of controllers}\\ \text{who start work at the}\\ \text{beginning of period } j\end{array} + \begin{array}{c}\text{number of controllers}\\ \text{who started work at the}\\ \text{beginning of the previous}\\ \text{period}\end{array}$$

Here the indexing of the periods is cyclic, so when $j = 1$ the previous period is $j = 8$. The table of requirements and the conservation law together summarize the facts of this problem.

2. The manager of the center cannot directly control how many people will be on duty during any given work period, because some will have started in the previous period and they cannot be sent home early. However, the conservation law makes it clear that what the manager does control is how many people *start* work at the beginning of each period, and those are the natural decision variables.

$$x_j = \text{number of controllers starting work at the beginning of period } j, \quad j = 1 \ldots 8$$

3. Using the conservation law and these decision variables we can express the staffing requirements like this.

$$\begin{aligned} x_1 + x_8 &\geq r_1 \\ x_j + x_{j-1} &\geq r_j \quad j = 2 \ldots 8 \end{aligned}$$

The number of people starting work in period j can never be negative, so an optimal solution must also have $x_j \geq 0$ for $j = 1 \ldots 8$.

4. Assuming that no controller works more than one 2-period shift, each begins work exactly once each day and the number needed to cover a day is the total number who start work. Thus we must minimize this sum.

$$N = \sum_{j=1}^{8} x_j$$

Now we can formulate the `shift` problem (see §28.5.4) as this linear program.

$$\begin{array}{lrcll} \underset{\mathbf{x}\in\mathbb{R}^8}{\text{minimize}} & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 & = & N & \\ \text{subject to} & x_1 \qquad\qquad\qquad\qquad\qquad\qquad + x_8 & \geq & 3 & \\ & x_1 + x_2 & \geq & 6 & \\ & x_2 \quad + x_3 & \geq & 14 & \\ & x_3 \qquad + x_4 & \geq & 18 & \\ & x_4 \qquad\quad + x_5 & \geq & 16 & \\ & x_5 \qquad\qquad + x_6 & \geq & 14 & \\ & x_6 \qquad\qquad\quad + x_7 & \geq & 12 & \\ & x_7 \qquad\qquad\qquad\quad + x_8 & \geq & 6 & \\ & x_j & \geq & 0 & j = 1 \ldots 8. \end{array}$$

The solution is $\mathbf{x}^\star = [3, 4, 10, 8, 8, 6, 6, 0]^\top$ so 45 people are required to cover the schedule. To satisfy the constraints it is necessary that some work periods are overstaffed even in this optimal program; for example, $x_1^\star + x_2^\star = 7 > 6 = r_2$.

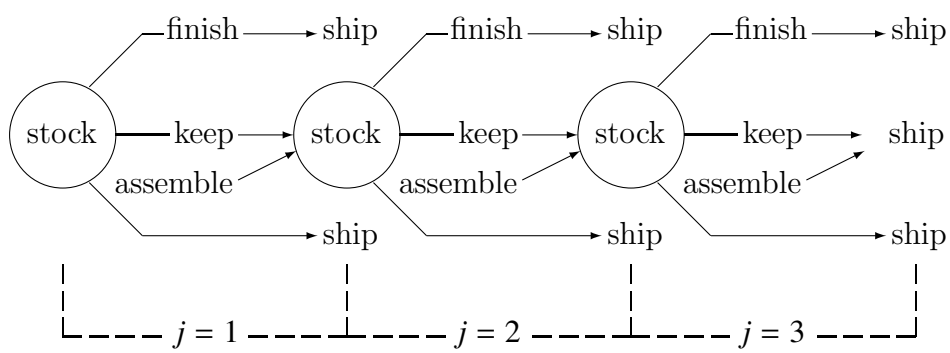
The x_j count people, so it is essential that their optimal values be whole numbers. It might seem to have been by lucky coincidence that the solution we found has components that are all integers, but the structure of this problem ensures that if the requirements are whole numbers then the $x_j^\star$ will be too (see Exercise 1.9.17).

The shift assignments we found are repeated each day, so this planning problem is said to have a **finite horizon**. Of course most people don't work all seven days of each week, so the 45 people in the daily schedule are probably not the *same* people each day.

1.4.2 Making Furniture

A specialty furniture company has a contract to manufacture wooden rocking chairs for a retail chain. The chairs are in great demand but their production is limited by the number of skilled artisans the furniture company can assign to make them. During each 2-week production period a worker can either assemble 50 chairs or stain and varnish 25. Finished chairs sell for \$300 each, but there is also a market for unfinished chairs at \$120. Each period's sales are delivered to the retailer in a single shipment at the end of the period. Up to 200 unfinished chairs can be stored from one period to the next, but no finished chair is ever packed into storage because that might damage the varnish. The furniture company's factory has enough space and staff to assign up to 12 workers to chair production during the next three periods. If there are currently 100 unfinished chairs in storage, what production schedule should the company follow to maximize its revenue over the next six weeks?

1. To summarize the facts of this problem it is helpful to make a **stage diagram** showing the flow of unfinished and finished chairs through the production process.



This picture suggests the following conservation law.

$$\begin{array}{c}\text{chairs in stock at}\\ \text{start of period } j\end{array} = \begin{array}{c}\text{chairs in stock at}\\ \text{start of period } j-1\end{array} + \begin{array}{c}\text{chairs assembled}\\ \text{during period } j-1\end{array} - \begin{array}{c}\text{chairs shipped at}\\ \text{end of period } j-1\end{array}$$

2. To express this relationship mathematically we can introduce variables to count for each period the chairs in stock at the beginning, the chairs assembled, the chairs finished and shipped, and the chairs that are left unfinished but shipped.

$$\begin{aligned} s_j &= \text{number of chairs in stock at start of period } j \\ a_j &= \text{number of chairs assembled in period } j \\ f_j &= \text{number of chairs finished and shipped in period } j \\ u_j &= \text{number of chairs shipped unfinished at end of period } j \end{aligned}$$

Then conservation of chairs requires that $s_j = s_{j-1} + a_{j-1} - (f_{j-1} + u_{j-1})$.

Of the quantities defined above the first three are state variables because the company does not control them directly. The company *does* control u_j and

$$\begin{aligned} x_j &= \text{number of workers assembling chairs in period } j \\ y_j &= \text{number of workers finishing chairs in period } j \end{aligned}$$

so they are the decision variables.

3. It makes no sense for any of the variables to be negative. The state variables and decision variables are related, according to the problem description, in the following ways.

$x_j + y_j$	$\leq$	12	up to 12 workers can be used, if there is enough work
a_j	$\leq$	$50x_j$	each assembler can make 50 chairs, if there is space to store them
f_j	$\leq$	$25y_j$	each finisher can finish 25 chairs, if there are enough unfinished
$f_j + u_j$	$\leq$	s_j	we can't ship more chairs than are in stock at the period start
s_2	$\leq$	200	there is only enough space
s_3	$\leq$	200	to store 200 unfinished chairs

To enforce the conservation law requires the following **state equation** constraints.

$$\begin{aligned} s_1 &= 100 \\ s_2 &= s_1 + a_1 - (f_1 + u_1) \\ s_3 &= s_2 + a_2 - (f_2 + u_2) \\ 0 &= s_3 + a_3 - (f_3 + u_3) \end{aligned}$$

According to the problem description the starting stock is 100 chairs; at the end of the third production period everything has been sold, so there is no ending stock.

4. At the ends of the production periods the furniture company realizes these revenues.

$$\begin{aligned} R_1 &= 300f_1 + 120u_1 \\ R_2 &= 300f_2 + 120u_2 \\ R_3 &= 300f_3 + 120(s_3 - f_3) + 120a_3 \end{aligned}$$

At the end of the third production period we sell the f_3 chairs that have been finished in that period, the entire remaining stock $(s_3 - f_3)$ of unfinished chairs, and the a_3 unfinished chairs that are assembled in period three. The objective to be maximized is thus

$$\begin{aligned} R &= R_1 + R_2 + R_3 \\ &= 300f_1 + 120u_1 + 300f_2 + 120u_2 + 180f_3 + 120s_3 + 120a_3 \end{aligned}$$

Now we can formulate the `chairs` problem (see §28.5.5) as the linear program below. This model has 18 variables, 4 equality constraints, and 14 inequality constraints in addition to the nonnegativities.

$$
\begin{array}{lrcl}
\underset{\mathbf{s}\mathbf{a}\mathbf{f}\mathbf{u}\mathbf{x}\mathbf{y}}{\text{maximize}} & 120s_3 + 120a_3 + 300f_1 + 300f_2 + 180f_3 + 120u_1 + 120u_2 & = & R \\
\text{subject to} & x_1 + y_1 & \leq & 12 \\
& x_2 + y_2 & \leq & 12 \\
& x_3 + y_3 & \leq & 12 \\
& a_1 - 50x_1 & \leq & 0 \\
& a_2 - 50x_2 & \leq & 0 \\
& a_3 - 50x_3 & \leq & 0 \\
& f_1 - 25y_1 & \leq & 0 \\
& f_2 - 25y_2 & \leq & 0 \\
& f_3 - 25y_3 & \leq & 0 \\
& f_1 + u_1 - s_1 & \leq & 0 \\
& f_2 + u_2 - s_2 & \leq & 0 \\
& f_3 + u_3 - s_3 & \leq & 0 \\
& s_2 & \leq & 200 \\
& s_3 & \leq & 200 \\
& s_1 & = & 100 \\
& s_2 - s_1 - a_1 + f_1 + u_1 & = & 0 \\
& s_3 - s_2 - a_2 + f_2 + u_2 & = & 0 \\
& s_3 + a_3 - f_3 - u_3 & = & 0 \\
& \mathbf{s} & \geq & \mathbf{0} \\
& \mathbf{a} & \geq & \mathbf{0} \\
& \mathbf{f} & \geq & \mathbf{0} \\
& \mathbf{u} & \geq & \mathbf{0} \\
& \mathbf{x} & \geq & \mathbf{0} \\
& \mathbf{y} & \geq & \mathbf{0}
\end{array}
$$

This linear program has the optimal solution

$$
\begin{array}{lcl}
\mathbf{x}^\star & = & [4, 4, 0]^\top \\
\mathbf{y}^\star & = & [4, 8, 8]^\top \\
\mathbf{u}^\star & = & [0, 0, 0]^\top \\
\mathbf{s}^\star & = & [100, 200, 200]^\top \\
\mathbf{a}^\star & = & [200, 200, 0]^\top \\
\mathbf{f}^\star & = & [100, 200, 200]^\top \\
R^\star & = & 150000.
\end{array}
$$

Notice that only 8 workers are needed in periods 1 and 3, and that no chairs are ever shipped unfinished. The optimal values of the decision variables x_j, y_j, and u_j tell the company what to do; the corresponding values of the state variables s_j, a_j and f_j, along with the objective value, describe the consequences of those actions.

The structure of the `shift` problem ensures that if the data are whole numbers then the optimal point will have integer components, but that is *not* true of this problem. If the data had been different the solution might have required that some workers divide their time between assembly and finishing or that fractional numbers of chairs be shipped. To ship whole chairs we would need to find a feasible rounded solution or solve the problem as an integer program.

If the furniture company's contract with the retail chain is for longer than the next six weeks, we could enlarge the model to include more production periods (each would add six variables, six nonnegativities, and six other constraints to the formulation). If the contract has no certain end date then the planning problem would have an **infinite horizon** and we would need to decide how many periods are enough. In this problem the production process achieves steady state in period 2, so if production is to continue past period 3 we could have 4 workers finish and 8 assemble in periods $2, 3, \ldots$ In other problems the startup transient lasts longer, or some input such as the number of workers available varies from one period to the next so that steady state is never achieved (see Exercise 1.9.22).

1.5 Nonsmooth Formulations

This Chapter is about formulating *linear* programs, in which the objective and constraints are linear functions of the decision and state variables. It is very desirable for an optimization to have this special form because, as we shall see beginning in §2, linear programs are easy to solve. Some optimization problems in which the functions are *not* linear can, by clever tricks, be recast as linear programs. In this Section we will consider two important kinds of nonlinear optimization that can be easily solved in this way.

1.5.1 Minimizing the Maximum

A disaster-recovery team is equipped with two gasoline-powered water pumps having different fuel-consumption and water-pumping rates as summarized below.

pump	fuel used [gal/hr]	water pumped [1000 ft^3/hr]
A	2	12
B	8	20

The team has been allocated 16 gallons of gasoline to use in pumping out a hospital basement that is flooded with 60000 ft^3 of water. If pumps A and B start at the same time, how long should each be run to drain the basement as soon as possible?

The decision variables in this problem are implicit in its statement.

$$\begin{aligned} x_A &= \text{hours pump A runs} \\ x_B &= \text{hours pump B runs} \end{aligned}$$

Using these variables and the data given in the table above we can state the constraints mathematically.

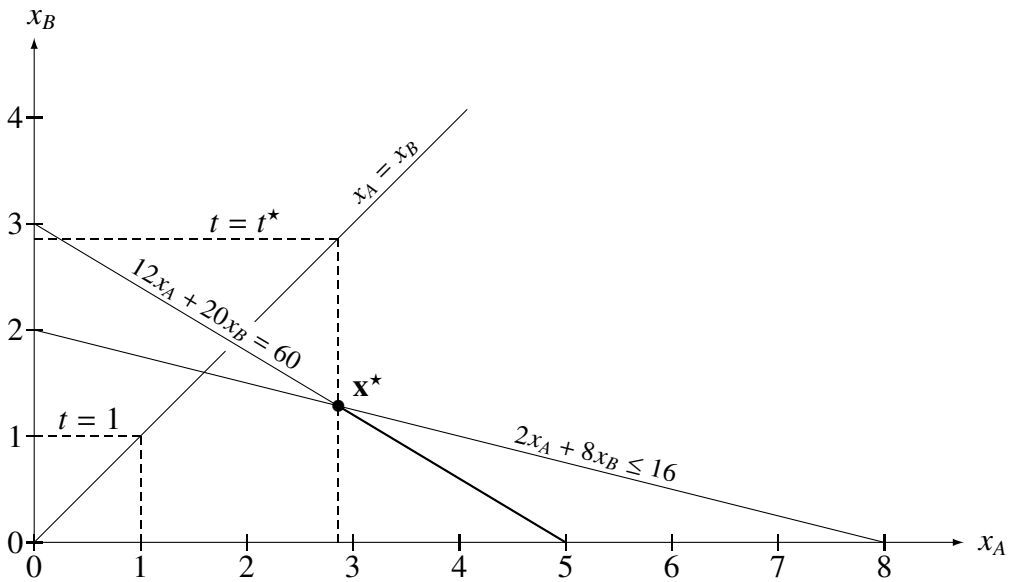
$$\begin{aligned} 2x_A + 8x_B &\leq 16 && \text{use no more gasoline than provided} \\ 12x_A + 20x_B &= 60 && \text{pump out all of the water} \\ x_A &\geq 0 && \text{pump A time can't be negative} \\ x_B &\geq 0 && \text{pump B time can't be negative} \end{aligned}$$

The pump that is running at the moment the basement becomes empty stops then, so the time it takes to pump out all of the water will be x_A if pump A is the last to stop or x_B if pump B is the last to stop. In other words the time t required is the larger of x_A and x_B, so the team wants to

$$\text{minimize} \quad t = \max(x_A, x_B).$$

This function is nonlinear, so it cannot be the objective in a linear program. It is also not smooth, which makes it hard to minimize using the techniques for nonlinear programming that we will take up starting in §8.

Because the problem has only two variables, we can solve it graphically as shown below. The contours of t are corners rather than straight lines, but they are not hard to draw. For example, if $t = 1$ that must be the value of x_B if $x_B \geq x_A$ (above the diagonal $x_A = x_B$). If $x_A \geq x_B$ (below the diagonal) then $t = 1$ must be the value of x_A.



Because the second constraint is an equality it is satisfied only *on* the line $12x_A + 20x_B = 60$, so in this picture the feasible set is the line segment that is drawn thick. The feasible point having the lowest objective value is the leftmost point on that line segment, which is marked $\mathbf{x}^\star$. Solving the two constraint equations simultaneously yields

$$\begin{aligned} \mathbf{x}^\star &= \left[\tfrac{20}{7}, \tfrac{9}{7}\right]^\top \\ t^\star &= \tfrac{20}{7}. \end{aligned}$$

Thus the optimal pumping schedule is to run both for $\frac{9}{7} = 1.29$ hours, then shut pump B off and let pump A continue to run for an additional $\frac{11}{7} = 1.57$ hours. This uses all of the gasoline and empties the basement in $\max(\frac{20}{7}, \frac{9}{7}) \approx 2.86$ hours.

Now notice that if $t = \max(x_A, x_B)$ then

$$\begin{aligned} t &\geq x_A \\ t &\geq x_B. \end{aligned}$$

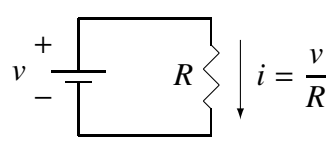
We can see this in the graph above, where at each point on the $t = 1$ contour $1 \geq x_A$ and $1 \geq x_B$. Minimizing t subject to these two constraints will push t down against whichever bound is higher so that constraint is satisfied with equality, making t equal to the larger of x_A and x_B. Using this idea we can formulate the optimization as the linear program shown at the top of the next page, which I will call the `pumps` problem (see §28.5.6).

$$\begin{array}{lrcl} \underset{x_A\, x_B\, t}{\text{minimize}} & t & & \\ \text{subject to} & t & \geq & x_A \\ & t & \geq & x_B \\ & 2x_A + 8x_B & \leq & 16 \\ & 12x_A + 20x_B & = & 60 \\ & x_A & \geq & 0 \\ & x_B & \geq & 0 \end{array}$$

This linear program has three variables so it is hard to solve graphically, but the simplex method that you will learn later yields the optimal point $x_A^\star = \frac{20}{7}$, $x_B^\star = \frac{9}{7}$, $t^\star = \frac{20}{7}$. This is the $\mathbf{x}^\star$ we found above by solving the two-variable nonlinear problem graphically. At this point the constraint $t \geq x_A$ is satisfied with equality while $t \geq x_B$ is satisfied as a strict inequality.

1.5.2 Minimizing the Absolute Value

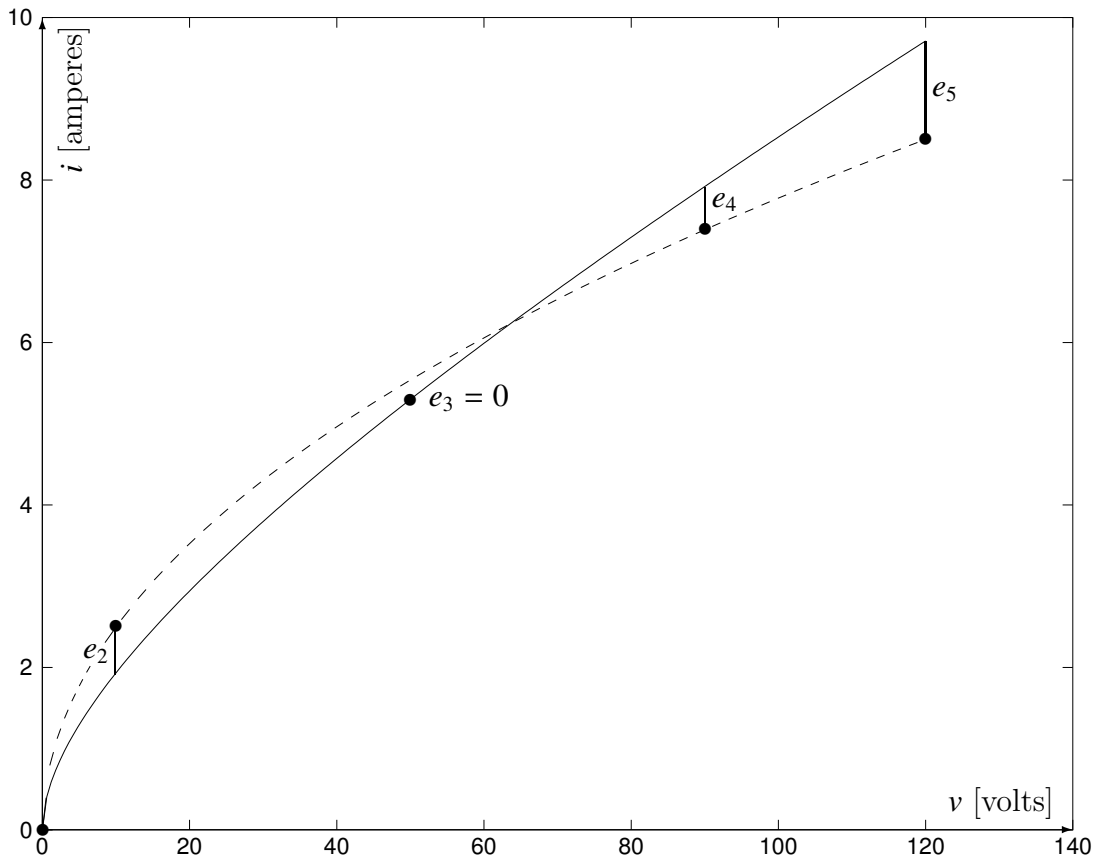
An incandescent lamp works by passing an electric current through a metal filament. Because the filament has resistance, the flow of current raises the temperature of the metal until it emits visible light in addition to waste heat. If the resistance of the filament is constant, then according to Ohm's law the current that flows through it is a linear function of the voltage across it. The circuit diagram below shows a battery of v volts connected to an ideal resistor of R ohms and the current flow of i amperes that results.



The resistance of a metal such as tungsten depends on its temperature. As the voltage applied to an incandescent lamp is increased the temperature of the filament increases and its resistance also increases, so Ohm's law does not apply and i is a nonlinear function of v. Once I had occasion to measure the current flowing in a large incandescent lamp at several different voltages, and five of my observations are given in the table below.

observation j	v [volts]	i [amperes]
1	0	0
2	10	2.5
3	50	5.3
4	90	7.4
5	120	8.5

These data are plotted in the graph on the next page. Can we deduce from them a formula describing the relationship between i and v?



To derive a simple model for predicting the current at voltages between these data points, I ignored the complicated physics of the light bulb and guessed that a function of the form

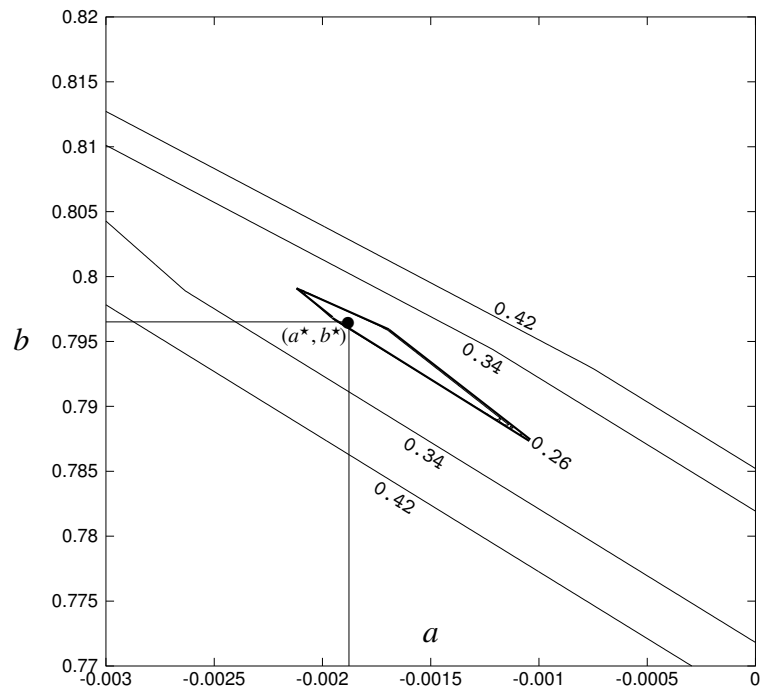
$$i(v) = av + b\sqrt{v}$$

might be made to fit the measurements by adjusting the parameters a and b. The solid line above plots $i(v)$ for $b = 0.5$, with $a = (i_3 - b\sqrt{v_3})/v_3 \approx 0.035$ chosen so that the curve passes through the point (v_3, i_3) exactly (every function of the assumed form passes through the origin). This trial function is clearly not a good fit to the data, because the estimate it provides is too low at v_2 yet too high at v_4 and v_5. One way of finding the values of a and b that yield the best fit is to minimize the sum of the absolute values of the errors,

$$\begin{aligned} E &= \sum_{j=2}^{5} |e_j| = \sum_{j=2}^{5} \left| i_j - i(v_j) \right| = \sum_{j=2}^{5} \left| i_j - av_j - b\sqrt{v_j} \right| \\ &= \left|2.5 - 10a - b\sqrt{10}\right| + \left|5.3 - 50a - b\sqrt{50}\right| + \left|7.4 - 90a - b\sqrt{90}\right| + \left|8.5 - 120a - b\sqrt{120}\right|. \end{aligned}$$

The absolute values make E nonlinear in a and b, so it cannot be the objective of a linear program. It is also not smooth, so it is hard to minimize using nonlinear programming.

Because the problem has only two variables we can solve it graphically in the same way that we solved the nonlinear version of the `pumps` problem. The contours of $E(a,b)$ are hard to plot by hand (even though they are polyhedra) so I used Octave, obtaining the picture below. Computer-generated contour plots will be an indispensable tool in our study of nonlinear programming, so I will have much more to say about their construction and interpretation in §9.1 and §19.5.



Here each curve is the locus of points where $E(a,b)$ has the value shown, so $(a^\star, b^\star)$ must be inside the central figure; it turns out to be the point marked with a dot.

It is possible [152] to write $E(a,b)$ in a way that does not involve absolute values, by using the following elementary property of real numbers.

> A real number y can always be written as $y = u - w$, where $u \geq 0$, $w \geq 0$, and one or the other is zero; then $|y| = u+w$.

A couple of examples might convince you that this is true. If $y = 10$ we can write it as $y = u - w$ where $u = 10$ and $w = 0$; then $u + w = 10 + 0 = 10 = |y|$. If $y = -10$ we can write it as $y = u - w$ where $u = 0$ and $w = 10$; then $u + w = 0 + 10 = |y|$. In our formula for $E(a,b)$, each term is of the form $|y_j|$ and can therefore be written as the sum of two variables u_j and w_j whose difference is y_j.

Doing that produces the following linear program for minimizing $E(a, b)$, which I will call the `bulb` problem (see §28.5.7).

$$
\begin{array}{lrcl}
\underset{ab\mathbf{uw}}{\text{minimize}} & E & = & (u_2 + w_2) + (u_3 + w_3) + (u_4 + w_4) + (u_5 + w_5) \\
\text{subject to} & u_2 - w_2 & = & 2.5 - 10a - b\sqrt{10} \\
 & u_3 - w_3 & = & 5.3 - 50a - b\sqrt{50} \\
 & u_4 - w_4 & = & 7.4 - 90a - b\sqrt{90} \\
 & u_5 - w_5 & = & 8.5 - 120a - b\sqrt{120} \\
 & u_2,\, u_3,\, u_4,\, u_5 & \geq & 0 \\
 & w_2,\, w_3,\, w_4,\, w_5 & \geq & 0 \\
 & a,\, b & & \text{free}
\end{array}
$$

The state variables u_j and w_j are nonnegative because that is required by the real number property that is boxed on the previous page, but a and b are unconstrained in sign so they are said to be **free variables**.

The optimal solution to this linear program has $a^\star = -0.00187741$ and $b^\star = 0.79650632$, resulting in a fit with total error $E^\star = 0.25092429$. These values of a and b are the ones marked in the contour diagram on the previous page, and when they are used in the model function it has the curve drawn dashed in the graph on the page before that. The very small value of $a^\star$ suggests that not much would be lost by simplifying the model to $i = b\sqrt{v}$.

The state variables corresponding to data points 2 and 5 have $u_j^\star = w_j^\star = 0$ because the dashed curve passes through them exactly. At point 4, $u_4^\star = 0.01264476$ and $w_4^\star = 0$ because the model underestimates the data by a small amount; at point 3, $w_3^\star = 0.23827953$ and $u_3^\star = 0$ because it overestimates by a larger amount.

The model function that I assumed does not describe the data precisely, so no combination of parameter values could make the dashed curve pass through all of the points. Minimizing the sum of the absolute values of the e_j selects the set of data points that yields the lowest error when the curve comes as close as possible to going through them. The other data points, in this case point 3, are essentially ignored, and are thus identified by the algorithm as **outliers**. The ability to reject outliers is an important virtue of this approach to fitting an equation to data.

1.5.3 Summary

In both linear and nonlinear programming we would almost always rather solve a smooth problem than one whose functions are not everywhere differentiable. Nondifferentiability can arise for reasons other than the ones we have studied, but it is so often the result of minimizing a maximum or an absolute value that the formulation techniques of this Section will be of use throughout the book. They are summarized in somewhat more general form in the table on the next page, where the smooth problem is a linear program only if $f_i(\mathbf{x})$ happens to be linear in $\mathbf{x}$. In the notation of this table, $\mathbf{x} = [x_A, x_B]^\top$ for the `pumps` problem

and $\mathbf{x} = [a, b]^\top$ for the `bulb` problem. If a nonsmooth problem includes constraints they must of course be carried over to the smooth reformulation. Some problems call for minimizing the maximum of terms that are absolute values, and then both reformulation techniques must be applied (see Exercise 1.9.37).

nonsmooth problem	smooth problem
$\underset{\mathbf{x}}{\text{minimize}} \quad t = \max\limits_{i=1\ldots m} \left\{ f_i(\mathbf{x}) \right\}$	$\begin{array}{lll} \underset{t\,\mathbf{x}}{\text{minimize}} & t & \\ \text{subject to} & t \;\geq\; f_i(\mathbf{x}) & i = 1 \ldots m \end{array}$
$\underset{\mathbf{x}}{\text{minimize}} \quad \sum\limits_{i=1}^{m} \left\lvert f_i(\mathbf{x}) \right\rvert$	$\underset{\mathbf{u}\,\mathbf{w}\,\mathbf{x}}{\text{minimize}} \quad \sum\limits_{i=1}^{m} (u_i + w_i)$ $\text{subject to} \quad \left.\begin{array}{rcl} u_i - w_i & = & f_i(\mathbf{x}) \\ u_i & \geq & 0 \\ w_i & \geq & 0 \end{array}\right\} \; i = 1 \ldots m$

1.6 Bilevel Programming

Crude oil, a complex mixture of hydrocarbons, is separated into products having different boiling points by a process called fractional distillation. Some fractions are then transformed, using heat and pressure in a process called cracking, into the lighter compounds that make up gasoline.

Every month a refinery distills enough crude oil to bring its stock of kerosene up to its storage capacity of 1000 barrels. It considers gasoline an important secondary product so it sends some of the kerosene stock to be cracked, but at the premium price point of \$100 per barrel it expects to sell no more than 300 barrels of gasoline in a month. It markets the remainder of the kerosene as jet fuel, which is the refinery's primary product (and for which it has a good reputation in the aviation industry) at \$50 per barrel.

As a separate business unit of the refinery, the cracking operation independently maximizes its production of gasoline based on the amount of kerosene that it has been allocated. To start up the process requires 50 barrels of kerosene, which are not cracked; after that each barrel that is cracked yields 0.8 barrel of gasoline. Any allocated kerosene that is not cracked is returned to the refinery and is not sold that month.

Kerosene and gasoline are shipped sequentially, partitioned by spacers, in a single pipeline. The pipeline company has contracted to ship up to 900 barrels each month, but it will not accept any partition of less than 100 barrels.

How much kerosene should the refinery crack into gasoline each month to maximize its revenue from selling jet fuel and gasoline?

The amount of gasoline produced depends on the amount of kerosene made available, but in a complicated way that makes it hard to formulate this problem as a single linear program. Because the cracking business decides for itself how much gasoline to make, it is more natural to model each part of the production process separately.

If the refinery sends x barrels of kerosene for cracking then $1000 - x$ remain to be sold as jet fuel. The x barrels of kerosene that are cracked produce y barrels of gasoline, so the total revenue to be maximized is $50(1000 - x) + 100y$. The jet fuel and gasoline that are shipped must each be more than the pipeline minimum but together be less than its capacity. Thus the refinery wants to

$$\begin{array}{llrcll}
\underset{xy}{\text{maximize}} & 50(1000-x)+100y & = & \text{revenue} & & \\
\text{subject to} & 1000-x & \geq & 100 & & \text{kerosene pipeline minimum} \\
 & 1000-x & \leq & 900 & & \text{pipeline capacity limit} \\
 & y & \leq & 300 & & \text{policy limit on gasoline sales} \\
 & y & & & & \text{maximizes the gasoline from } x \text{ barrels of kerosene.}
\end{array}$$

Meanwhile the cracking operation's optimization problem is

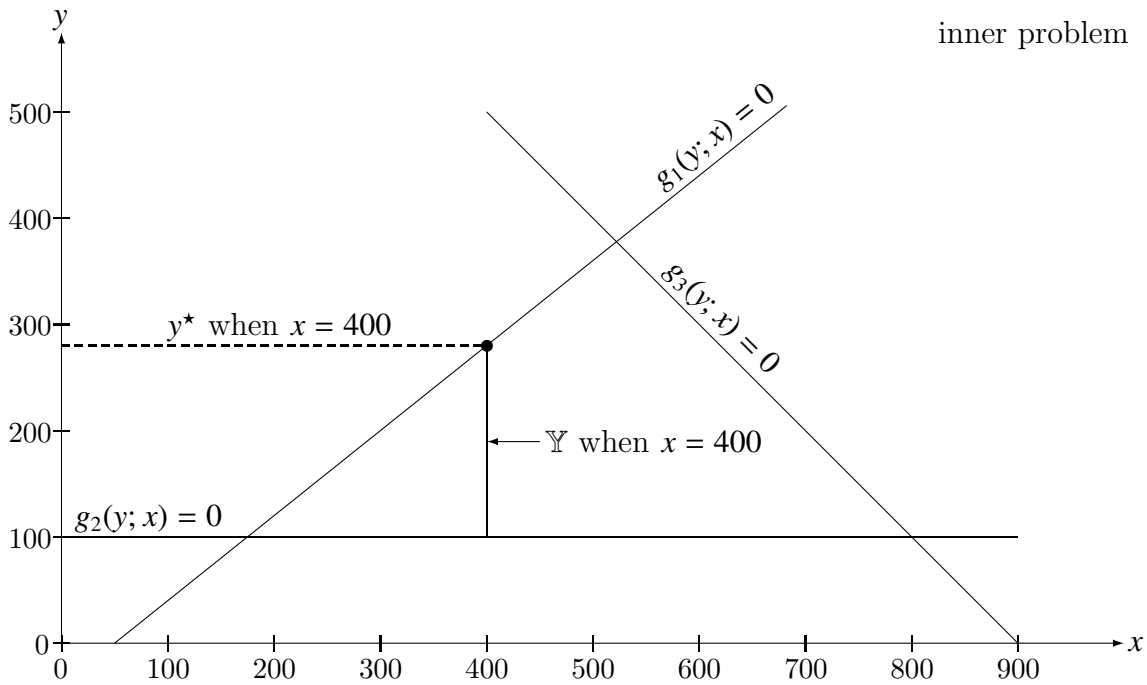
$$\begin{array}{llcll}
\underset{y}{\text{maximize}} & y & & & \text{gasoline produced} \\
\text{subject to} & y & \leq & 0.8(x-50) & \text{yield from cracking} \\
 & y & \geq & 100 & \text{gasoline pipeline minimum} \\
 & y & \leq & 900-x & \text{pipeline capacity limit.}
\end{array}$$

These linear programs are connected by the last constraint in the refinery model, which requires that $y(x)$ be the optimal point of the cracking optimization, so they can be combined into the following **bilevel program** [43].

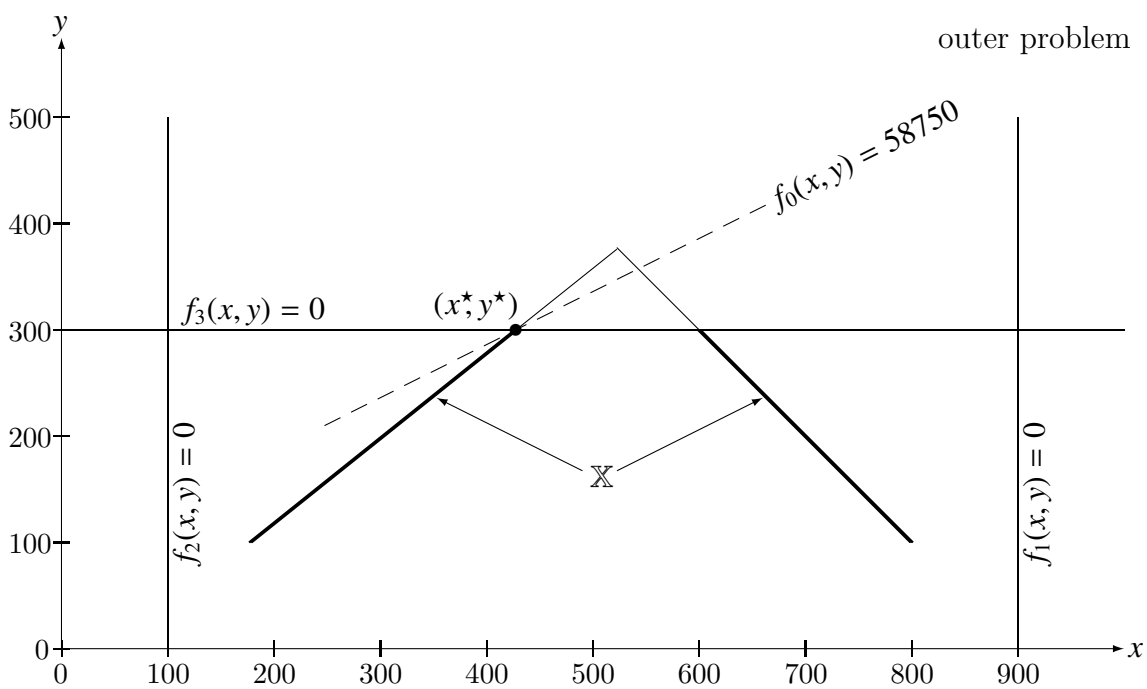
$$\underset{xy}{\text{maximize}} \quad f_0(x,y) = 50000 - 50x + 100y$$

$$\text{subject to} \begin{cases}
f_1(x,y) = x - 900 \leq 0 \\
f_2(x,y) = 100 - x \leq 0 \\
f_3(x,y) = y - 300 \leq 0 \\
y \text{ solves} \left[\begin{array}{l}
\underset{y}{\text{maximize}} \quad g_0(y;x) = y \\
\text{subject to} \begin{cases}
g_1(y;x) = y - 0.8(x-50) \leq 0 \\
g_2(y;x) = 100 - y \leq 0 \\
g_3(y;x) = y - 900 + x \leq 0
\end{cases}
\end{array} \right]
\end{cases}$$

The **outer problem** or overall optimization is solved by varying both x and y, but in the **inner problem**, shown here enclosed by square brackets, x is treated as a constant parameter and the optimization is performed by varying only y.



The graph above plots the constraints of the inner problem. For any given fixed value of x, the values of y that are feasible for the inner problem are points on the vertical line that is delimited by the inner constraints. For example, at $x = 400$ the feasible set $\mathbb{Y}$ of the inner problem is the line drawn there.



The objective of the inner problem is $g_0(y; x) = y$, so for a given value of x it is maximized at the top of the feasible line $\mathbb{Y}(x)$. In the picture, the optimal point of the inner problem when $x = 400$ is marked with a dot.

The locus of points $y^\star(x)$ is called the **inducible region** of the inner problem, and it is plotted as the bent line in the lower graph. In solving the outer problem, one of the constraints that must be satisfied is that y is on this bent line. The other constraints of this outer problem are simple bounds on x and y. The feasible set $\mathbb{X}$ of the outer problem is that part of the inducible region that is within these bounds, and it is drawn with thick lines. The outer constraints $f_1(x, y) \leq 0$ and $f_2(x, y) \leq 0$ do not affect the feasible set, but the outer constraint $f_3(x, y) \leq 0$ intersects the inducible region and results in a feasible set that is comprised of two disjoint line segments.

Having identified the feasible set $\mathbb{X}$ of the outer problem, we can easily find the feasible point having the highest objective value; this turns out to be $x^\star = 425$, $y^\star = 300$.

Often a situation involving decision makers who act independently but whose actions affect one another can be modeled as a bilevel program. A bus company that is optimizing improvements to its route map must anticipate that its riders will optimize their own travel choices in response; an automobile dealership that is negotiating to employ a salesperson on commission must anticipate the agent's personal objective and constraints [23].

The two-stage graphical approach illustrated above can be used only for tiny problems, so analytic and numerical methods, based on the theory and algorithms we will study, are essential. Even when the inner and outer problems are both linear programs the bilevel problem is decidedly *non*linear, and in many practical applications the functions f_i and g_i are themselves nonlinear. Bilevel programs are among the most difficult optimization problems, and they are an active area of research in nonlinear programming [86].

1.7 Applications Overview

The toy problems discussed above suggest only a few of the many uses that linear programming has in science, engineering, business, and government. Here are a few representative fields in which linear optimization models play an important role (as we shall see in §8.4 some of them are also fields in which *nonlinear* programming is widely used).

signal processing	supply-chain management
airline flight scheduling	natural gas transmission
arbitrage and investment banking	disaster response planning
machine learning	public health and nutrition [169]
pollution abatement	city planning
fulfillment and delivery operations	military logistics
renewable energy distribution	conservation of natural resources [65]

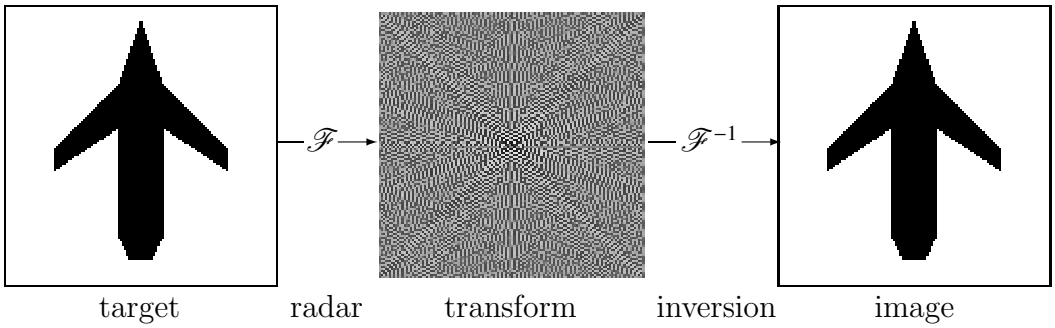
The references cited in the list above and described in the table below discuss the formulation of specific **application problems** from some of these fields. I have arranged the books in

decreasing order of their emphasis on problem formulation; useful general advice is also provided in [25, §I], [35, §3-1,3-2], [145, §2.1-2.2], [79, §5.1], and [151, §2.2].

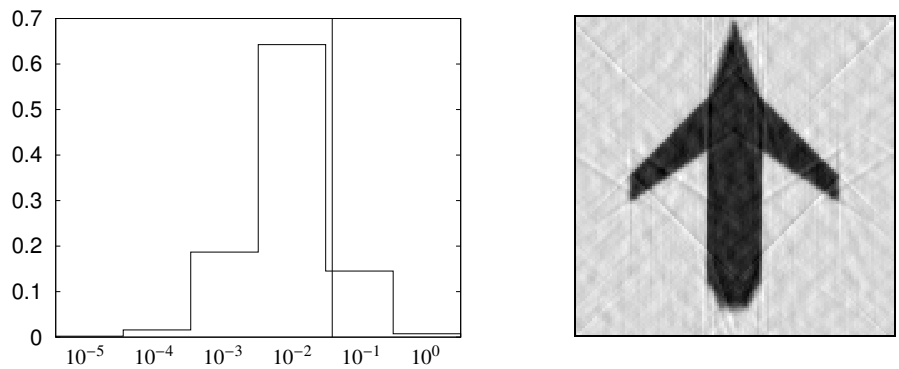
reference	modeling content
[145, §2]	The formulation examples concern gasoline blending, advertising media selection, investment portfolio design, transportation and assignment problems, production scheduling, make-or-buy decisions, the traveling salesman problem, the general diet problem, awarding contracts, and maintaining a "profitable ecological balance." The chapter includes 32 formulation exercises.
[151, §2]	The formulation examples are described as product mix selection, feed mix selection, fluid blending, arbitrage transactions, integrated production planning, and product allocation through a transportation network. The final two sections of the chapter pose 23 exercises.
[79]	Models are given in §2.4 for regional planning and controlling air pollution; in this chapter problems 1, 2, and 3 are formulations. Models are given in §4 for network, assignment, and multi-divisional planning problems; in this chapter all 30 problems are formulations. In §5, problems 1-21 are formulations.
[3, §2]	The formulation examples involve making furniture, brewing beer, mixing oil, warehousing peanuts, raising chickens, scheduling nurses, curve fitting, inconsistent systems of equations, and feasibility problems. Exercises 2.8, 2.10, 2.12-2.16, and 2.18-2.22 are formulations.
[35]	Sections 3-3 through 3-7 discuss a transportation problem, blending examples, a product mix problem, a warehouse problem, and an on-the-job training problem. In §3-9, problems 4 through 22 are formulations.

1.8 Compressed Sensing

The first application of linear programming listed in the survey of §1.7 is signal processing, and an important example of signal processing is radar imaging. A synthetic-aperture radar [27] emits pulses of microwave radiation. When the radio waves encounter a target they excite current flow in the object so that it emits radiation, and these pulses travel back to the radar where they are detected. The received signals are filtered by analog electronics, converted to numbers, and processed to construct a **Fourier transform** [101] of the scene, which can then be numerically inverted to obtain a picture of the object.



Each element of the transform is a complex number. The raster above shows the real part, with dark pixels correspond to high values and light ones correspond to low values. In a transform with enough points (this one has 128×128 pixels) many of them will be almost zero. I used a log transformation to make the pixels with low values visible; if I had not done so the raster would appear mostly blank with only a few dark pixels near the center.



Transform elements that are very small contribute little to the reconstructed image. The histogram on the left above shows the proportion of pixels whose real values have the indicated orders of magnitude. The lowest 90% of the values, below the threshold of 0.125, are to the left of the vertical line. Setting those values (and the corresponding imaginary parts) to zero and inverting the resulting **sparse transform** yields the reconstruction shown on the right above. This image contains some artifacts but it is still recognizable as a picture of the target. Lowering the threshold to include half of the pixels yields a transform whose inverse is hard to distinguish from the full reconstruction. Thus it is possible to more or less perform the radar imaging task by using only a fraction of the information captured in the transform, and in some applications the fraction that must be retained can be quite small.

The time that is needed to acquire a radar image would be greatly reduced if we could capture only the high-value pixels of the transform and not bother measuring the others. Unfortunately that is not possible, because each element of the transform depends on all of the input data. However, it *is* possible by using **compressed sensing** [75] to make a very good guess about what the good pixels are, based on a small number of measurements.

1.8.1 Perfect Data

Suppose we construct a vector $\mathbf{x}$ by stacking the columns of the unknown transform matrix vertically, with the leftmost column on top, the second column below that, and so on until the rightmost column is on the bottom. For our example this results in a vector $n = 128^2 = 16384$ elements long (to keep things simple we will assume that it contains only the real parts of the transform elements). Next suppose that our radar set has been designed to report, for each pulse that it sends and receives, only the value $b_i = A_i^\top \mathbf{x}$ of some linear combination of the transform elements. To be consistent with m such measurements, the vector $\mathbf{x}$ would

have to satisfy this system of linear algebraic equations in which the A_i are the rows of $\mathbf{A}$.

$$\mathbf{Ax} = \mathbf{b}$$

If $A_1 \dots A_m$ were the unit vectors then $\mathbf{A}$ would be an $n \times n$ identity matrix and we could get the transform exactly by solving the square system. In compressed sensing the a_{ij} are randomly generated coefficients and $m \ll n$, so there are too few equations to uniquely determine the solution and many vectors $\mathbf{x}$ satisfy the linear system. But we know that $\mathbf{x}$ is sparse, or that we can treat it as sparse and still recover the image, so the $\mathbf{x}$ we want is very likely to be one that has the fewest nonzero elements.

The number of nonzero elements in a vector $\mathbf{x}$ is not really a norm (see §10.6.3) but it is conventionally referred to [39, §1.2.1] as the **zero norm**, $\|\mathbf{x}\|_0$. Using this notation, the $\mathbf{x}$ we want is the one that solves the following mathematical program.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \|\mathbf{x}\|_0 & \\ \text{subject to} & \mathbf{Ax} & = \mathbf{b} \end{array}$$

Alas, to find it we might need to try all of the ways that there could be m nonzero elements among the n elements of $\mathbf{x}$, of which there are [116, Theorem 1.8]

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}.$$

For $n = 128^2$ and $m = 20$ this number is on the order of 10^{67}.

When we histogrammed the elements of the transform in our example we saw that the elements we want to consider nonzero are not too far from 1 and most of the elements we want to consider zero are much smaller, so it might be reasonable to approximate the number of nonzeros in $\mathbf{x}$ by the sum of the absolute values of its elements,

$$\|\mathbf{x}\|_1 = \sum_{j=1}^{n} |x_j|.$$

This is about 1354 for our dense transform, while the number of nonzeros in the sparse transform is 1594. The compressed sensing problem

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \|\mathbf{x}\|_1 & \\ \text{subject to} & \mathbf{Ax} & = \mathbf{b} \end{array}$$

thus approximates the solution of the zero norm problem, and [45, Theorem 8] if $\mathbf{x}$ is sparse enough it can be shown to solve it exactly. Using a formulation technique from §1.5.3 we can rewrite this optimization as the linear program at the top of the next page.

$$\begin{array}{lll} \underset{\mathbf{u}\,\mathbf{v}\,\mathbf{x}}{\text{minimize}} & \displaystyle\sum_{j=1}^{n}(u_j+v_j) & \\ \text{subject to} & \mathbf{A}(\mathbf{u}-\mathbf{v}) = \mathbf{b} & \\ & \left.\begin{array}{rcl} u_j - v_j & = & x_j \\ u_j & \geq & 0 \\ v_j & \geq & 0 \end{array}\right\} & j = 1\ldots n \end{array}$$

Even for $n = 128^2$ this optimization is non-trivial, and in a real radar application the image raster might be much bigger. Special techniques based on the interior point method of §21.1 can be used [24] to solve the resulting **big data problem**.

1.8.2 Regularization

Above I argued that the sparsest solution $\mathbf{x}^\star$ to $\mathbf{Ax} = \mathbf{b}$ is sparse because it is a Fourier transform. No physical measurement is perfect, so in practice our radar set reports for each pulse not b_i but $b_i+\eta$, where η is random noise. Then solving the compressed sensing problem actually yields the sparsest vector $\hat{\mathbf{x}}$ that satisfies $\mathbf{Ax} = \mathbf{b} + \boldsymbol{\eta}$. But $\mathbf{b} = \mathbf{Ax}^\star$ so

$$\mathbf{A}\hat{\mathbf{x}} = \mathbf{Ax}^\star + \boldsymbol{\eta}.$$

If $\mathbf{y}$ is any vector that makes $\mathbf{Ay} = \boldsymbol{\eta}$ then

$$\begin{array}{rcl} \mathbf{A}\hat{\mathbf{x}} & = & \mathbf{Ax}^\star + \mathbf{Ay} \\ \mathbf{A}\hat{\mathbf{x}} & = & \mathbf{A}(\mathbf{x}^\star + \mathbf{y}) \\ \hat{\mathbf{x}} & = & \mathbf{x}^\star + \mathbf{y}. \end{array}$$

The unknown noise vector $\boldsymbol{\eta}$ is dense so almost every possible unknown $\mathbf{y}$ is too, and that makes it unlikely that $\hat{\mathbf{x}}$ will be sparse. By insisting in our formulation of the compressed sensing problem that $\mathbf{Ax} = \mathbf{b}$ is satisfied exactly, we made it almost certain that the mathematical program will produce the wrong answer if the data come from the real world.

To keep noise from making it impossible to find a sparse $\mathbf{x}$ we can, instead of insisting that the constraint be satisfied exactly, **regularize** the objective by adding a term that penalizes constraint violations.

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad \|\mathbf{x}\|_1 + \mu(\mathbf{Ax}-\mathbf{b})^\top(\mathbf{Ax}-\mathbf{b})$$

Now by adjusting the positive penalty parameter μ we can control the tradeoff between sparseness of the optimal point, which is achieved by minimizing $\|\mathbf{x}\|_1$, and satisfaction of the constraints. Using the same formulation technique as before we can make this problem smooth, but because the penalty term involves the product $\mathbf{x}^\top\mathbf{x}$ the result is a quadratic rather than a linear program (see §22).

The regularized noisy compressed sensing problem has a closed-form semi-analytic solution that can be found by **soft thresholding** [17, §4.4.3]. Soft thresholding is unfortunately beyond the scope of this introductory text, but I will have more to say about semi-analytic results in §25.7.4.

1.8.3 Related Problems

Compressed sensing is used in transform-based imaging technologies other than radar, including magnetic resonance imaging, computer assisted tomography, and geophysical data analysis. It also plays a role in image compression, where the transforms that are used are based on wavelets other than sinusoids.

Basis pursuit [17, §6.2] is another name for compressed sensing; the **lasso technique** for identifying the best variables to use in a regression model [17, §6.4] gives rise to the same optimization problem as regularized compressed sensing.

1.9 Exercises

1.9.1 [E] In many optimization problems our goal is to find the best way to allocate limited resources. Are there optimization problems that do not fit this prototype? If yes, give an example; if no, explain how all optimizations can be thought of as resource allocation problems.

1.9.2 [E] In the mathematical formulation of an optimization model, variables represent the quantities that are being reasoned about. (a) What is a *decision variable*? (b) What is a *state variable*? (c) What is a *free variable*?

1.9.3 [H] Explain the formulas given in §1.1 for s_1 and s_2 of the `twoexams` problem. Why are s_1 and s_2 identified as state variables rather than as decision variables?

1.9.4 [E] What precisely is a mathematical program? Describe its form and identify its parts. What makes a mathematical program a *linear* program? What modeling assumptions underlie the formulation of an optimization as a linear program?

1.9.5 [E] The word "programming" can be a synonym for "planning." What sort of plan is specified by a computer program? What sort of plan is specified by the solution to a mathematical program? How does mathematical programming differ from the writing of a computer program to carry out mathematical calculations? Is there any connection between the two?

1.9.6 [H] In a typical resource allocation problem [3, p17-18] the decision variables measure the levels of different **production activities**, doing more of any activity increases the objective, and the amount we can do is limited only by the resources. To solve such a problem it might seem that we could just find a production program that uses up all of the

resources. (a) With the help of an example, explain why that is usually impossible. (b) If there is a production program that does use up all of the resources, is it necessarily optimal? If yes, explain why; if no, provide a counterexample.

1.9.7 [H] Show by evaluating the constraint functions of the `twoexams` problem that the point $[2, 8]^\top$ satisfies all of them. Why is this sufficient to establish that $\mathbb{X}$ is the feasible set?

1.9.8 [E] What is a *nonnegativity constraint*? What makes a constraint *redundant*? What is a *constraint contour*? Explain how a linear program can be *infeasible*.

1.9.9 [H] What is an *objective contour*? Why are the objective contours of a linear program parallel to each other? What is an *optimal vector*?

1.9.10 [H] Show that in the `twoexams` problem, reducing the total study time available reduces the size of the feasible set. For what values of total study time available is the feasible set empty?

1.9.11 [H] Why can't a constraint contour ever cross the interior of the feasible set of a linear program? Why can't the optimal objective contour ever cross the interior of the feasible set?

1.9.12 [E] In §1.3.0 I suggested a systematic procedure for formulating linear programs. (a) List the steps in that procedure. (b) When a problem is dynamic, an additional formulation step is often helpful; what is it? (c) What is an *obvious constraint*? What is a *natural constraint*? What is a *technology table*?

1.9.13 [H] What assumptions are implicit in the formulation of the `brewery` problem? You might find it helpful to consult `www.beerrecipes.org` or review the similar formulations suggested in [3, p16-17] and [145, p55-56]. How would the model need to change for Sarah to maximize profit rather than revenue?

1.9.14 [H] The optimal solution to the `brewery` problem is $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$, in which the amount of Stout to be made is not a whole number of kegs. (a) Can Sarah round up that solution component and make 13 kegs of Stout, along with the optimal 5 kegs of Porter? (b) Can she round down and make 12 kegs of Stout along with 5 kegs of Porter? Is this the optimal integer solution? (c) Stout fetches by far the highest price per keg. Why isn't the best strategy to simply make as much Stout as possible? (d) There is clearly a market for all four varieties of beer. Why not make some of each?

1.9.15 [H] The `paint` problem of §1.3.2 includes a ratio constraint that the total product shipped be at least half RED. Now suppose the paint company instead insists that $\frac{2}{3}$ of the total product shipped be RED. (a) Is it still possible for the chemical company to make money by using its available feedstock to produce product for the paint company? If no, explain why not; if yes, how does the formulation change? (b) Is it possible for a ratio constraint to render a linear programming problem infeasible? If not, explain why not; if so, provide an example.

1.9.16 [H] In the `paint` problem of §1.3.2, at what selling price for BLUE would $\mathbf{x}^\star$ and $\bar{\mathbf{x}}$ *both* be optimal production programs?

1.9.17 [H] The `shift` problem of §1.4.1 has an optimal point with integer components, if all of the requirements are integers. Explain how the structure of the problem ensures this. What makes this a *finite horizon* planning problem?

1.9.18 [H] How does the formulation of the `shift` problem change if each shift consists of three work periods (a total of 9 hours) rather than two?

1.9.19 [H] The `shift` formulation of §1.4.1 assumes that every day is the same. (a) Enlarge the formulation to determine the optimal deployment of controllers across the work week, Monday through Friday, assuming that the requirements r_{ij} for day i and work period j are not necessarily the same from day to day. (b) Enlarge the formulation to include weekends. How can you ensure that no controller works more than five days in each week?

1.9.20 [H] The `chairs` formulation of §1.4.2 involves three decision variables and three state variables. (a) Can this problem be formulated in a way that requires fewer than three decision variables? If yes explain how; if no explain why not. (b) Can this problem be formulated in a way that requires fewer than three state variables? If yes explain how; if no explain why not. (c) Is a formulation that involves the fewest possible variables always to be preferred to one that involves more?

1.9.21 [E] What is a *stage diagram*? What is a *state equation*?

1.9.22 [H] Under what circumstances would the `chairs` formulation be an *infinite-horizon* planning problem? If an infinite-horizon problem never reaches steady state but future inputs are always known for the upcoming k periods, how can mathematical programming be used to plan the *next* production period?

1.9.23 [H] A hardware supplier produces J-bolts of a single size, and nuts to go with them, for use in fastening steel cables to support posts for highway guard rails. Each bolt or nut must be processed on 3 different machines during its manufacture. The table to the right shows the time required on each machine to process one ton of each product, and the amount of time available on each machine during the next production period. The company can sell all the bolts it can make, but along with them it must also deliver nuts weighing at least as much and not more than twice as much. (a) Letting x_1 represent the tons of bolts made and x_2 the tons of nuts made, formulate a linear programming model whose solution will maximize the revenue from selling bolts (the nuts are given away). (b) Solve the problem graphically. On your graph crosshatch the feasible set, label the optimal point $\mathbf{x}^\star$, and draw a dashed line for the optimal objective function contour. Label each constraint hyperplane with the inequality that it represents. (c) Find $x_1^\star$ and $x_2^\star$ algebraically.

machine number	times/ton		time available
	bolts	nuts	
1	3	1	9
2	1	3	9
3	2	2	16

1.9.24 [H] A linear programming student hates exercise but wants to impress a certain person by walking for at least an hour each day. By choosing an appropriate closed path the student can adjust the number of minutes spent walking uphill, downhill, and on the level. The prospective significant-other tags along, and agrees to hold hands one minute for every 20 they walk on the flat or downhill, and the whole time they walk uphill. Meanwhile, the student thinks (but wisely does not say) "exercise is really tiring" for one minute out of every ten minutes they walk on the flat, all the time they walk uphill, and not at all when they are walking downhill. It takes three times as long to ascend a given height walking uphill as it does to descend that vertical distance walking downhill. The student wants to maximize the daily hand-holding time without wasting more than ten minutes on thoughts of exhaustion, and hopes to formulate an optimization problem whose solution will reveal how many minutes the two friends should spend walking uphill, downhill, and on the level. (a) What can the student directly control? Call these decision variables x_j, $j = 1\ldots$, and define them precisely. (b) In terms of your decision variables, what constraints are imposed by the statement of the problem? Express these requirements as equations or inequalities involving the decision variables. (c) How can the student's objective be stated mathematically in terms of the decision variables? (d) Use the graphical method to solve the linear program you have formulated, and report the optimal distribution of times, in minutes per day, spent walking uphill, downhill, and on the flat. On your graph crosshatch the feasible set, label the optimal point $\mathbf{x}^\star$, and draw a dashed line for the optimal objective function contour. (e) What practical considerations are ignored in the statement of the problem? What does this illustrate about the mathematical modeling of real situations?

1.9.25 [H] A foundry must decide how many tons x_1 of new steel and how many tons x_2 of scrap metal to mix in casting steel shot for one of its customers. The ratio of scrap to new metal in the mix cannot exceed 7:8. Producing the shot costs \$300 per ton of new steel included in the mix and \$500 per ton of scrap included. Thus, for example, using 4 tons of new steel and 1 ton of scrap metal would yield 5 tons of shot at a production cost of $4 \times \$300 + 1 \times \$500 = \$1700$. The customer requires at least 5 tons of shot, but will accept more. The foundry has 4 tons of new steel and 6 tons of scrap metal on hand. (a) Formulate a linear programming model whose solution $[x_1, x_2]^\top$ minimizes the foundry's cost of production, subject to the various constraints. (b) Show that the constraints imply $1 \le x_2 \le 3\frac{1}{2}$. (c) Solve the problem graphically. On your graph crosshatch the feasible set, label the optimal point $\mathbf{x}^\star$, and draw a dashed line for the optimal objective function contour. Label each constraint hyperplane with the inequality that it represents. (d) How much new steel and scrap metal are left over from the optimal production program?

1.9.26 [H] A college senior estimates that the probability he will find a job prior to graduation is zero if he does not search for work, even if he keeps his 3.0 grade-point average (out of 4.0). He can improve his chances by interviewing prospective employers, or by raising his grades, or by doing both. His probability of finding a job will increase by 0.05 for every hour-per-day that he spends interviewing, and will increase or decrease by 0.06 for each increase

or decrease of 0.1 in his average. To maintain his 3.0 he finds that he needs to attend class and do homework for 8 hours per day, and he expects his average to rise or fall by 0.125 for every hour-per-day he increases or decreases that time. (a) Formulate a linear programming model to find the hours x he should spend each day looking for work and the hours y he should spend each day on school, to maximize the probability z of finding a job. (b) Solve the problem (i.e., find the optimal values of x and y) by using the graphical method. On your graph crosshatch the feasible set, label the optimal point, and draw a dashed line for the optimal objective function contour. What is the student's probability of finding a job if he carries out the optimal program? What does his grade-point average become? How many hours does he get to spend on things *other* than school and job-hunting each day, such as eating and sleeping? (c) The job-search optimization model is unrealistically pessimistic, though students who have a hard time finding a job might not think so. Suggest some ways to make the model more realistic. Is your improved model still a linear program?

1.9.27 [H] A company uses 2 machines to manufacture 2 products. It wants to maximize the total units of product made in this production period, but the units of product B made must be at least one-third of the total. (a) Supply numerical values for t_{11}, t_{12}, t_{13}, a_{21}, a_{22}, and a_{23}, and a formula for b_3, to make the following technology table and linear program formulation consistent with one another. Should the question marks be replaced by $\leq$, $=$, or $\geq$? (b) What must be the meanings of x_1 and x_2?

machine	time/unit product A	time/unit product B	time available
lathe	t_{11}	t_{12}	t_{13}
sander	6	3	36

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & x_1 + x_2 & & \\ \text{subject to} & 12x_1 + 8x_2 & \leq & 96 \\ & a_{21}x_1 + a_{22}x_2 & ? & a_{23} \\ & x_1 & ? & \frac{1}{3}b_3 \\ & x_1 \text{ and } x_2 & \geq & 0 \end{array}$$

(c) Solve the problem graphically and report $\mathbf{x}^\star$. (d) What is the optimal *integer* solution?

1.9.28 [H] Upon his arrival at college, a student whose parents forced him to eat the healthiest possible diet decides that while he is away at school he will instead eat the *least* healthy diet he can design. He knows the two main constituents of this diet will be jelly donuts and atomic-hot chicken wings, but he needs to determine their ideal mixture. After years of exposure to Brussels sprouts and fresh mangoes, the student figures his body can initially tolerate only certain amounts of fat, sugar, and synthetic additives. On the other hand, he is determined to eat at least 3 dozen donuts and 2 buckets of wings every day. The table below shows the quota (in dozens or buckets) and nutritional content (in ounces per dozen or bucket) of the donuts and wings, the health hazard (in lost days of life per ounce consumed) presented by each kind of content, and the maximum daily content amounts (in ounces) the student thinks he can stand. (a) Formulate a linear programming model to maximize the health hazard of this diet subject to the student's constraints. Assume the student can eat any fraction of a dozen or bucket. (b) Solve the problem by using the graphical method. On your graph crosshatch the feasible set, label the optimal point, and draw a dashed line for

the optimal objective function contour. What is the worst possible diet? How much health hazard does it deliver?

		basic food group		
		fat	sugar	chemicals
health hazard		0.0010	0.0008	0.0005
maximum tolerated		156	80	42
donuts	3	12	10	2
wings	2	20	5	4
component	quota			

1.9.29 [H] Discuss the assumptions implicit in our formulation of the `pumps` problem. How might each pump practically be fueled with the correct amount of gasoline?

1.9.30 [H] The graphical solution of the `pumps` problem in §1.5.1 shows the contours of $t = \max(x_A, x_B)$ as corners in the first quadrant. If the contours of this function are extended into the other quadrants of the graph, are they squares centered at the origin? If yes, explain why; if no, construct a function whose contours *are* squares centered at the origin.

1.9.31 [E] If a linear program has two variables and its constraints include a single equality, what will the feasible set look like in a graphical solution of the problem?

1.9.32 [H] If $t = \max(x_A, x_B)$ then $t \geq x_A$ and $t \geq x_B$. (a) Show that this is true. (b) If t is minimized at $[x_A^\star, x_B^\star]^\top$ and $x_A^\star \neq x_B^\star$, show that one of the constraints must be satisfied as an equality and the other must be satisfied as an inequality. (c) Show that $\max(0, f) = (f + |f|)/2$.

1.9.33 [H] If $y = u - v$ where $u \geq 0$ and $v \geq 0$, how is the quantity $u + v$ related to y? Give an example to illustrate your answer.

1.9.34 [H] In §1.5.2 we found that fitting the model function $i = va + b\sqrt{v}$ to the given data yielded a very small value for the parameter a. (a) Revise the `bulb` formulation to derive a linear program that fits the model function $i = b\sqrt{v}$. (b) Graphically approximate the solution of the nonlinear problem.

1.9.35 [E] What is an *outlier*? Give the most precise definition you can.

1.9.36 [H] Use linear programming to find values of x_1 and x_2 that minimize

$$|x_1 + x_2 - 1| + |x_1 + x_2 - 3|.$$

1.9.37 [H] It is possible for a square system of linear algebraic equations $\mathbf{Ax} = \mathbf{b}$ to be **inconsistent**, and then no vector $\mathbf{x}$ satisfies them all. In that case we might be interested in finding the $\mathbf{x}^\star$ that comes *closest* to satisfying them, in the sense that it minimizes the largest absolute row deviation $|\boldsymbol{a}_i^\top \mathbf{x} - b_i|$ [3, p26-27]. Formulate a linear program whose solution yields $\mathbf{x}^\star$.

1.9.38 [E] What is the *inducible region* of a bilevel program, and how can it be found?

1.9.39 [E] The outer problem in a bilevel program must include a constraint that has a special form. What is this constraint?

1.9.40 [E] If both the outer problem and the inner problem of a bilevel program are linear programs, is the bilevel problem a linear program?

1.9.41 [H] Our bilevel formulation of the oil refinery problem makes many implicit assumptions about the situation being modeled. Write down all of them that you can think of. Hint: *when* in the production process do the various steps occur?

1.9.42 [H] Formulate the oil refinery problem of §1.6 as a one-level linear program, and compare its graphical solution to the answer we found using the bilevel formulation.

1.9.43 [H] In perfect-data compressed sensing the linear system $\mathbf{Ax} = \mathbf{b}$ containing the measurements is underdetermined, because the matrix $\mathbf{A}$ is $m \times n$ and $m \ll n$. What property of $\mathbf{x}$ is used to select, from among all the vectors satisfying this system, the one that is most likely to approximate the transform of the image?

1.9.44 [H] What is the *zero norm* of a vector, and what symbol is used to represent it? In what ways does the zero norm fail to meet the mathematical definition of a vector norm?

1.9.45 [H] In §1.8.1 we derived three optimization models for the perfect-data compressed sensing problem. (a) Give the formulation in terms of $\|\mathbf{x}\|_0$, and explain why it cannot be used in practice. (b) Give the formulation in terms of $\|\mathbf{x}\|_1$, and explain why it is hard to solve by the classical techniques of nonlinear programming. (c) Give the formulation as a linear program, and explain why it is challenging to solve when its data are of realistic size.

1.9.46 [H] How does measurement noise affect the optimal transform that is found by our perfect-data compressed sensing model? How can the formulation be changed to more gracefully accommodate noise?

1.9.47 [H] Rewrite the regularized noisy compressed sensing problem as a smooth quadratic program.

1.9.48 [H] The curve-fitting example of §1.5.2 and the bilevel program of §1.6 use, respectively, an incandescent lamp and an oil refinery to illustrate general ideas about mathematical programming. Those technologies are still important to our economy and everyday lives as I write these words, but they might have become quaint historical curiosities by the time you work this Exercise. However, if you have understood this Chapter you should be able to see optimization problems everywhere you look. (a) Make up a new example to illustrate curve-fitting by minimizing a sum of absolute values. (b) Make up a new example to illustrate bilevel linear programming.

1.9.49 [H] Like any technology, mathematical optimization can be used for good or evil purposes. Describe one application of linear programming that you would consider beneficial to humanity and one application that you would consider harmful. How will your ethical judgements affect your conduct as a practitioner of linear programming?

2

The Simplex Algorithm

In §1 you learned the graphical method for solving linear programs. When there are more than three variables it is necessary to use a method that does not depend on drawing a picture. We will study two, the interior point algorithm [89] in §21.1 and the **simplex algorithm** [35] in this Chapter.

2.1 Standard Form

The simplex algorithm solves linear programs that are stated in a special way called **standard form**. Here is the standard form that we will use.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & z(\mathbf{x}) & = & d + \mathbf{c}^\top\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

A linear program that is in standard form has these three distinguishing characteristics.

- It is a *minimization.* The `twoexams`, `brewery`, `paint`, `chairs`, and oil refinery problems of §1 were all naturally formulated as maximizations, so to put them into standard form we must reverse the sense of the optimization. Whenever I call the objective function of a mathematical program z, the optimization will always be a minimization.

- It has *equality constraints.* The `pumps` problem included a single equality constraint but it and all the other examples had inequality constraints, so to put them into standard form requires some reformulation. I will use m to denote the number of constraints that are *not* nonnegativities (these are called **functional constraints**) so in a standard-form problem the **constraint coefficient matrix A** and the **constant column b** will always have m rows. Sometimes I will refer to $\mathbf{b}$ as the **right-hand side vector**.

- It has *nonnegative variables.* In the `bulb` problem the model parameters a and b were free variables, so to put that problem into standard form requires some reformulation. I will use n to denote the number of variables, so $\mathbf{A}$ will always have n columns and the **solution vector x** and the **objective cost coefficient vector c** will always have n rows. The optimal point of a linear program is sure to be in the boundary of the feasible set, so it is essential that $\mathbb{X}$ include its boundary points and thus that $\mathbf{x}$ be greater than *or equal* to zero rather than strictly positive.

The linear program below is the `brewery` problem of §1.3.1 in standard form; in §2.9, I will explain how to put *any* linear program into standard form.

$$
\begin{array}{lrcrcrcrcrcrcrcr}
\underset{\mathbf{x}\in\mathbb{R}^7}{\text{minimize}} & -90x_1 & - & 150x_2 & - & 60x_3 & - & 70x_4 & + & 0x_5 & + & 0x_6 & + & 0x_7 & & \\
\text{subject to} & 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & + & 1x_5 & + & 0x_6 & + & 0x_7 & = & 160 \\
 & 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & + & 0x_5 & + & 1x_6 & + & 0x_7 & = & 50 \\
 & 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & + & 0x_5 & + & 0x_6 & + & 1x_7 & = & 60 \\
 & & & & & & & & & & & & & \mathbf{x} & \geq & \mathbf{0}
\end{array}
$$

This problem has $n = 7$ variables and $m = 3$ functional equality constraints. The vector inequality means that each variable is nonnegative.

$$\mathbf{x} \geq \mathbf{0} \quad \Leftrightarrow \quad x_j \geq 0, \quad j = 1 \ldots n$$

I will (almost) always use j to index the variables of a mathematical program and i to index the constraints.

The scalar constant d is often nonzero (as in, for example, `twoexams`) but in this problem it happens to be zero so the objective function value is

$$
\begin{aligned}
z(\mathbf{x}) \;=\; \mathbf{c}^\top\mathbf{x} &= \begin{bmatrix} -90 & -150 & -60 & -70 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \\
&= \sum_{j=1}^{n} c_j x_j \\
&= -90x_1 - 150x_2 - 60x_3 - 70x_4.
\end{aligned}
$$

Here $\mathbf{c}^\top$, the **transpose** of the cost vector $\mathbf{c}$, is a row vector, so the **inner product** (also called the **scalar product** or **dot product**) $\mathbf{c}^\top\mathbf{x}$ is **conformable**. If you need to brush up on matrix arithmetic you can consult §28.2, but I will also refresh your memory about the facts we need as we first need them.

The problem has this constraint coefficient matrix and constant column.

$$
\mathbf{A} = \begin{bmatrix} 7 & 10 & 8 & 12 & 1 & 0 & 0 \\ 1 & 3 & 1 & 1 & 0 & 1 & 0 \\ 2 & 4 & 1 & 3 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} A_1 \\ \vdots \\ A_m \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 160 \\ 50 \\ 60 \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}
$$

It will occasionally be convenient to refer to row i of a matrix $\mathbf{A}$ as A_i. All of the other vectors in this book, denoted by lower-case bold letters such as $\mathbf{x}$, are column vectors. Thus $A_i\mathbf{x}$ is the dot product of row i with the column $\mathbf{x}$ and the equality constraints can be written as shown at the top of the next page.

$$\mathbf{Ax} = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} A_1\mathbf{x} \\ A_2\mathbf{x} \\ A_3\mathbf{x} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}x_j \\ \sum_{j=1}^{n} a_{2j}x_j \\ \sum_{j=1}^{n} a_{3j}x_j \end{bmatrix} = \mathbf{b}$$

2.2 The Simplex Tableau

We will often *state* a linear program algebraically, as in the formulations of §1 and the description of standard form given in §2.1. To *solve* a problem with the simplex algorithm, it is convenient to represent its standard form more compactly in a **simplex tableau**. Rearranging the terms in our definition of standard form we get the algebraic statement below, in which the right hand side vector $\mathbf{b}$ actually appears on the *left* (this is more natural if tableaus with different numbers of columns are to be manipulated by a computer program). The objective and constraints are represented by the tableau on the right, in which each column of $\mathbf{c}^\top$ and $\mathbf{A}$ is to be thought of as multiplied by the variable x_j that appears in the corresponding column of the equations.

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & z - d = \mathbf{c}^\top\mathbf{x} \\ \text{subject to} & \mathbf{b} = \mathbf{Ax} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \qquad \begin{array}{c|c|c|l} z & -d & \mathbf{c}^\top & \text{objective row} \\ \hline & \mathbf{b} & \mathbf{A} & \text{constraint rows} \\ \hline & & & \text{nonnegativity of } \mathbf{x} \text{ is implicit} \end{array}$$

If you think of the vertical line inside the tableau as a column of = signs, and visualize (if it is not present) the z that I have printed to the left of the **objective row**, then you can read off the objective and constraint equations from the tableau. There are m **constraint rows** and n variables, so a simplex tableau always has $m + 1$ rows and $n + 1$ columns.

The nonnegativity constraints are *implied* in representing the problem by a tableau, rather than being stated explicitly. To be represented by a tableau a linear program must be in standard form, and that means all of the variables are nonnegative.

The standard form given above for the `brewery` problem has this tableau.

$$\mathbf{T}_0 = \begin{array}{c|rrrrrrr} & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \hline 0 & -90 & -150 & -60 & -70 & 0 & 0 & 0 \\ \hline 160 & 7 & 10 & 8 & 12 & 1 & 0 & 0 \\ 50 & 1 & 3 & 1 & 1 & 0 & 1 & 0 \\ 60 & 2 & 4 & 1 & 3 & 0 & 0 & 1 \end{array}$$

Often I will label the right-hand columns of a tableau with the names of the corresponding variables, but usually it is easy to tell from the problem context which variables go with which columns even if they are not labeled.

2.3 Pivoting

The third constraint row of the `brewery` tableau represents the equation

$$\begin{aligned}
60 &= 2x_1 + 4x_2 + x_3 + 3x_4 + x_7 \\
\text{or, dividing through by 4,} \quad 15 &= \tfrac{1}{2}x_1 + x_2 + \tfrac{1}{4}x_3 + \tfrac{3}{4}x_4 + \tfrac{1}{4}x_7 \qquad (4) \\
\text{or, arbitrarily solving for } x_2, \quad x_2 &= 15 - \tfrac{1}{2}x_1 - \tfrac{1}{4}x_3 - \tfrac{3}{4}x_4 - \tfrac{1}{4}x_7.
\end{aligned}$$

We could substitute this expression for x_2 into the objective and the other constraints to eliminate that variable from them, like this.

$$\begin{aligned}
0 &= -90x_1 - 150\left(15 - \tfrac{1}{2}x_1 - \tfrac{1}{4}x_3 - \tfrac{3}{4}x_4 - \tfrac{1}{4}x_7\right) - 60x_3 - 70x_4 \\
\Rightarrow \quad 2250 &= -15x_1 - 22\tfrac{1}{2}x_3 + 42\tfrac{1}{2}x_4 + 37\tfrac{1}{2}x_7 \qquad (1)
\end{aligned}$$

$$\begin{aligned}
160 &= 7x_1 + 10\left(15 - \tfrac{1}{2}x_1 - \tfrac{1}{4}x_3 - \tfrac{3}{4}x_4 - \tfrac{1}{4}x_7\right) + 8x_3 + 12x_4 + x_5 \\
\Rightarrow \quad 10 &= 2x_1 + 5\tfrac{1}{2}x_3 + 4\tfrac{1}{2}x_4 + x_5 - 2\tfrac{1}{2}x_7 \qquad (2)
\end{aligned}$$

$$\begin{aligned}
50 &= x_1 + 3\left(15 - \tfrac{1}{2}x_1 - \tfrac{1}{4}x_3 - \tfrac{3}{4}x_4 - \tfrac{1}{4}x_7\right) + x_3 + x_4 + x_6 \\
\Rightarrow \quad 5 &= -\tfrac{1}{2}x_1 + \tfrac{1}{4}x_3 - 1\tfrac{1}{4}x_4 + x_6 - \tfrac{3}{4}x_7 \qquad (3)
\end{aligned}$$

The new equations are algebraically equivalent to the old ones so we could use them to replace the rows of $\mathbf{T}_0$, obtaining this tableau.

$$\mathbf{T}_1 = \begin{array}{|r|rrrrrrr|c}
 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & \\
\hline
2250 & -15 & 0 & -22\tfrac{1}{2} & 42\tfrac{1}{2} & 0 & 0 & 37\tfrac{1}{2} & (1) \\
\hline
10 & 2 & 0 & 5\tfrac{1}{2} & 4\tfrac{1}{2} & 1 & 0 & -2\tfrac{1}{2} & (2) \\
5 & -\tfrac{1}{2} & 0 & \tfrac{1}{4} & -1\tfrac{1}{4} & 0 & 1 & -\tfrac{3}{4} & (3) \\
15 & \tfrac{1}{2} & 1 & \tfrac{1}{4} & \tfrac{3}{4} & 0 & 0 & \tfrac{1}{4} & (4) \\
\hline
\end{array}$$

$\mathbf{T}_0$ and $\mathbf{T}_1$ are **equivalent tableaus** in the sense that they represent two different standard forms of exactly the same linear program, and other tableaus equivalent to $\mathbf{T}_0$ and $\mathbf{T}_1$ could be produced in a similar way. The simplex algorithm generates equivalent tableaus until finding a standard form that reveals the solution.

It would have been much easier to transform $\mathbf{T}_0$ into $\mathbf{T}_1$ by using the elementary row operations of linear algebra [147, §1]. Unfortunately, not every sequence of elementary row

operations on a tableau yields an equivalent tableau (see Exercise 2.10.21). In generating a new tableau like $\mathbf{T}_1$, the easiest way to be certain that it represents the same linear program we started with is to perform the particular sequence of row operations that is called a **pivot**, as follows.

- Select a **pivot element** $a_{hp} \neq 0$, where $h \in \{1 \ldots m\}$ is the index in $\mathbf{A}$ of the **pivot row** and $p \in \{1 \ldots n\}$ is the index in $\mathbf{A}$ of the **pivot column**. A "pivot" in the constant column of a tableau (corresponding to $p = 0$) is never useful; a "pivot" in the objective row (corresponding to $h = 0$) produces a new tableau that is *not* equivalent to the starting tableau.
- Divide the pivot row of the tableau by the pivot element. This makes the pivot element equal to 1.
- Add multiples of the resulting pivot row to the other rows of the tableau to get zeros elsewhere in the pivot column.

The simplex algorithm is defined in terms of pivots, so we will consider the pivot to be *the fundamental operation* that we use in solving linear programs. We will never need or use any other row operations.

2.3.1 Performing a Pivot

The pivot operation is in fact so important to everything we will do between now and §8 that it deserves the following step-by-step illustration.

To obtain $\mathbf{T}_1$ from $\mathbf{T}_0$ by pivoting we can proceed as follows. First we select the pivot element $a_{hp} = a_{32}$, which I have circled in the tableau $\mathbf{T}_0$ below. In generating a sequence of tableaus by hand pivoting, it is helpful to circle each pivot element.

$\mathbf{T}_0 =$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	−90	−150	−60	−70	0	0	0
160	7	10	8	12	1	0	0
50	1	a_{32} 3	1	1	0	1	0
60	2	4	1	3	0	0	1

$\longleftarrow \mathbf{A}$

Next we divide the pivot row by the pivot element to obtain that row of the result tableau.

$\mathbf{T}_1 =$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
15	$\frac{1}{2}$	1	$\frac{1}{4}$	$\frac{3}{4}$	0	0	$\frac{1}{4}$

To zero out the objective function component in the pivot column we can add 150 times this new row 4 to row 1 of $\mathbf{T}_0$ and fill in the objective row as shown below.

$$\mathbf{T}_1 = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \hline 2250 & -15 & 0 & -22\frac{1}{2} & 42\frac{1}{2} & 0 & 0 & 37\frac{1}{2} \\ \hline \\ 15 & \frac{1}{2} & 1 & \frac{1}{4} & \frac{3}{4} & 0 & 0 & \frac{1}{4} \end{array}$$

To zero out the 10 in the pivot column we can subtract 10 times the new row 4 from row 2 of $\mathbf{T}_0$ and fill in the result.

$$\mathbf{T}_1 = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \hline 2250 & -15 & 0 & -22\frac{1}{2} & 42\frac{1}{2} & 0 & 0 & 37\frac{1}{2} \\ \hline 10 & 2 & 0 & 5\frac{1}{2} & 4\frac{1}{2} & 1 & 0 & -2\frac{1}{2} \\ \\ 15 & \frac{1}{2} & 1 & \frac{1}{4} & \frac{3}{4} & 0 & 0 & \frac{1}{4} \end{array}$$

Finally, to complete $\mathbf{T}_1$ we can zero out the 3 in the pivot column by subtracting 3 times the new row 4 from row 3 of $\mathbf{T}_0$.

$$\mathbf{T}_1 = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \hline 2250 & -15 & 0 & -22\frac{1}{2} & 42\frac{1}{2} & 0 & 0 & 37\frac{1}{2} \\ \hline 10 & 2 & 0 & 5\frac{1}{2} & 4\frac{1}{2} & 1 & 0 & -2\frac{1}{2} \\ 5 & -\frac{1}{2} & 0 & \frac{1}{4} & -1\frac{1}{4} & 0 & 1 & -\frac{3}{4} \\ 15 & \frac{1}{2} & 1 & \frac{1}{4} & \frac{3}{4} & 0 & 0 & \frac{1}{4} \end{array}$$

In performing a pivot by hand it is unnecessary to separately show or explain the intermediate steps as I have done here. Now that you know how to pivot you can simply look at $\mathbf{T}_0$, do the arithmetic in your head, and write down $\mathbf{T}_1$.

2.3.2 Describing Standard Forms

In $\mathbf{T}_0$ the constraint coefficient matrix, constant column, and cost vector are the $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ of our *initial* standard form for the `brewery` problem. To be fussy about this we could refer to them as $\mathbf{A}_0$, $\mathbf{b}^0$ and $\mathbf{c}^0$. Pivoting changes the numbers in the tableau, so the corresponding parts of $\mathbf{T}_1$ are different and should technically be called $\mathbf{A}_1$, $\mathbf{b}^1$ and $\mathbf{c}^1$ (the entries of $\mathbf{c}^1$ are called **reduced costs**). This precise notation is occasionally helpful, but usually we will be talking about these quantities in a generic way. From now on we will therefore think of $\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$, and also d and z, without subscripts or superscripts, as denoting their values in *any* standard form problem or tableau. This is similar to the use of a single variable name in a computer program to represent a quantity that changes as the iterations of an algorithm progress [100, §2.2,§2.6].

2.4 Canonical Form

Pivoting on the element circled in $\mathbf{T}_1$ on the previous page produces the tableau $\mathbf{T}_2$ shown below (you should verify some of the numbers to be sure that you understand the pivot).

$$\mathbf{T}_2 = \begin{array}{c|cccccccc} & & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \hline & 2290\frac{10}{11} & -6\frac{9}{11} & 0 & 0 & 60\frac{10}{11} & 4\frac{1}{11} & 0 & 27\frac{3}{11} \\ \hline & 1\frac{9}{11} & \frac{4}{11} & 0 & 1 & \frac{9}{11} & \frac{2}{11} & 0 & -\frac{5}{11} \\ & 4\frac{6}{11} & -\frac{13}{22} & 0 & 0 & -1\frac{5}{11} & -\frac{1}{22} & 1 & -\frac{7}{11} \\ \mathbf{b} & 14\frac{6}{11} & \frac{9}{22} & 1 & 0 & \frac{6}{11} & -\frac{1}{22} & 0 & \frac{4}{11} \end{array}$$

Many possible vectors satisfy $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$ and are therefore feasible for the linear program. Can you read off one of them from this tableau?

The tableau contains a system of 3 constraint equations in 7 variables. This system is underdetermined but not inconsistent, so we can find a solution by setting any 4 of the variables to zero and solving for the others. If we pick the variables having coefficients of 1 to be those that are nonzero then they will be easy to solve for. Setting the others to zero, $x_1 = x_4 = x_5 = x_7 = 0$ and the constraint equations read like this.

$$\begin{aligned} 1\tfrac{9}{11} &= \tfrac{4}{11}(0) + 0x_2 + 1x_3 + \tfrac{9}{11}(0) + \tfrac{2}{11}(0) + 0x_6 - \tfrac{5}{11}(0) \\ 4\tfrac{6}{11} &= -\tfrac{13}{22}(0) + 0x_2 + 0x_3 - 1\tfrac{5}{11}(0) - \tfrac{1}{22}(0) + 1x_6 - \tfrac{7}{11}(0) \\ 14\tfrac{6}{11} &= \tfrac{9}{22}(0) + 1x_2 + 0x_3 + \tfrac{6}{11}(0) - \tfrac{1}{22}(0) + 0x_6 + \tfrac{4}{11}(0) \end{aligned}$$

Except for $1x_2$, $1x_3$, and $1x_6$, every term to the right of the equals signs is zero because either the coefficient or the variable is zero. But there is no need to write out the equations; if while looking at $\mathbf{T}_2$ we think of those variables whose tableau columns are not identity columns as being zero, then we can simply read off the others as

$$\begin{aligned} x_2 &= b_3 = 14\tfrac{6}{11} \\ x_3 &= b_1 = 1\tfrac{9}{11} \\ x_6 &= b_2 = 4\tfrac{6}{11}. \end{aligned}$$

Because $\mathbf{b} \geq \mathbf{0}$ in $\mathbf{T}_2$ this solution satisfies $\mathbf{x} \geq \mathbf{0}$ as well as $\mathbf{Ax} = \mathbf{b}$.

What makes it possible to find a feasible point in this way is that $\mathbf{T}_2$ (like $\mathbf{T}_0$ and $\mathbf{T}_1$) is in **canonical form**. A canonical form tableau has these three distinguishing characteristics.

- The $\mathbf{A}$ part of the tableau contains all the columns of the $m \times m$ identity matrix.
- The reduced cost entries c_j over those identity columns are zero.
- The constant column is nonnegative: $\mathbf{b} \geq \mathbf{0}$.

The identity columns in a canonical-form tableau are called **basis columns** and their order in the $m \times m$ identity matrix, here $S = (x_3, x_6, x_2)$, is the **basic sequence** of the tableau; the tableau is said to be in canonical form **with respect to** this basic sequence. The variables in the basic sequence are the **basic variables**, while the others, here x_1, x_4, x_5, and x_7, are the **nonbasic variables**.

Pivoting in a canonical-form tableau makes the **entering variable** corresponding to the pivot column basic, and it makes the **leaving variable** whose column had a 1 in the pivot row nonbasic. The pivot from $\mathbf{T}_1$ to $\mathbf{T}_2$ made the x_3 column basic while the x_5 column, which had its 1 in the pivot row, became a nonbasic column.

2.4.1 Basic Feasible Solutions

By assuming the nonbasic variables are zero in $\mathbf{T}_2$, we were able to read off the feasible point

$$\mathbf{x}^2 = \left[0, 14\tfrac{6}{11}, 1\tfrac{9}{11}, 0, 0, 4\tfrac{6}{11}, 0\right]^\top.$$

This is called the **basic feasible solution** that is **associated with** that canonical-form tableau. In $\mathbf{T}_2$ the reduced cost vector is

$$\mathbf{c}^\top = \left[-6\tfrac{9}{11}, 0, 0, 60\tfrac{10}{11}, 4\tfrac{1}{11}, 0, 27\tfrac{3}{11}\right]^\top$$

so the dot product that appears in the objective function row of $\mathbf{T}_2$ is

$$\mathbf{c}^\top\mathbf{x}^2 = -6\tfrac{9}{11}(0) + 0(14\tfrac{6}{11}) + 0(1\tfrac{9}{11}) + 60\tfrac{10}{11}(0) + 4\tfrac{1}{11}(0) + 0(4\tfrac{6}{11}) + 27\tfrac{3}{11}(0) = 0.$$

At the basic feasible solution $\bar{\mathbf{x}}$ associated with any canonical form tableau, $\bar{x}_j = 0$ for nonbasic variables and $c_j = 0$ for basic variables, so $\mathbf{c}^\top\bar{\mathbf{x}} = 0$.

At the basic feasible solution $\mathbf{x}^2$, the objective row of $\mathbf{T}_2$ looks like the picture on the left below and represents the equation on the right

z	$2290\frac{10}{11}$	$\mathbf{c}^\top\mathbf{x}^2$

$$\begin{aligned} z + 2290\tfrac{10}{11} &= \mathbf{c}^\top\mathbf{x}^2 = 0 \\ z &= -2290\tfrac{10}{11} \end{aligned}$$

Because $\mathbf{c}^\top\bar{\mathbf{x}} = 0$ in any canonical-form tableau, the element $\mathbf{T}(1,1)$ in its upper left corner is the *negative* of the objective value at the associated basic feasible solution $\bar{\mathbf{x}}$.

$-z$	$\mathbf{c}^\top\bar{\mathbf{x}} \equiv 0$
	canonical-form tableau

Of course the value of $\mathbf{c}^\top\mathbf{x}$ at an arbitrary point $\mathbf{x}$ that is *not* the basic feasible solution is *not* zero. In $\mathbf{T}_2$ the objective row says

$$\begin{aligned} z + 2290\tfrac{10}{11} &= -6\tfrac{9}{11}x_1 + 60\tfrac{10}{11}x_4 + 4\tfrac{1}{11}x_5 + 27\tfrac{3}{11}x_7 \\ z &= -2290\tfrac{10}{11} - 6\tfrac{9}{11}x_1 + 60\tfrac{10}{11}x_4 + 4\tfrac{1}{11}x_5 + 27\tfrac{3}{11}x_7. \end{aligned}$$

The basic feasible solution $\mathbf{x}^2$ associated with $\mathbf{T}_2$ has $x_1 = x_4 = x_5 = x_7 = 0$, so $z(\mathbf{x}^2) = -2290\frac{10}{11}$. But we are trying to minimize z, so we would like to make it lower while keeping all of the $x_j \geq 0$. Is there some way to do that, according to the formula above?

Yes! Because the reduced cost for x_1 is negative, we could decrease z by letting x_1 be positive instead of zero. To make x_1 positive we can introduce it as a basic variable by pivoting on some element a_{h1} in the x_1 column of $\mathbf{T}_2$.

2.4.2 The pivot.m Routine

In pivoting from $\mathbf{T}_0$ to $\mathbf{T}_1$ to $\mathbf{T}_2$ I did exact arithmetic, so that you could obtain the same results by hand and thereby confirm that you understand the process. When the entries of an initial tableau are integers, successive pivots often produce fractions having progressively larger denominators and this makes hand calculation increasingly tedious. Practical applications usually involve data that are arbitrary real numbers, and then hand pivoting is nearly impossible. Using a computer program to perform pivots will spare us much labor as we continue our study of the `brewery` problem in §2.4.3, so this seems an opportune moment to introduce `pivot.m`, which is listed on the next page. MATLAB calls a code segment like this a `function` because it has inputs and outputs, but I will call one that we write a **routine** to distinguish it from mathematical functions and from code functions like `sqrt()` that are built into MATLAB. The line numbers [1] through [40] on the left are not part of the code.

The input parameters [1] are `T`, a tableau that might or might not be in canonical form; `mm` $= m + 1$, the number of rows in the tableau; `nn` $= n + 1$, the number of columns in the tableau; `ip`, the index in `T` of the pivot row; `jp`, the index in `T` of the pivot column; and `S`, a vector describing the basic sequence. If the pivot element is a_{hp} then `ip` $= h + 1$ and `jp` $= p + 1$. Each element of `S` is 0 if the corresponding variable column is nonbasic, or if it is basic the index in `T` of the row containing its identity 1.

The output parameters [1] are `Tnew`, the tableau resulting from the pivot; `Snew`, the basic sequence of the new tableau; and a return code `rc` that signals success if it is 0 [39] or, if it is 1, failure because the specified pivot element `T(ip,jp)` is zero [5].

If the pivot element is nonzero the routine [9-15] computes the elements of the new tableau, except for those in the [10] pivot row and [12] pivot column, by a process equivalent to what we have been doing by hand. The quantity `T(ii,jp)/T(ip,jp)` [13] is the fraction of the *original* pivot row that must be subtracted from the row `ii` being updated to zero out the element in the pivot column of that row. Next [17-20] we update the elements in the pivot row, except [18] for the pivot element. This [19] is where the pivot row gets divided by the pivot


```
 1 function [Tnew,Snew,rc]=pivot(T,mm,nn,ip,jp,S)
 2 %  perform a pivot at T(ip,jp)
 3 %
 4    if(T(ip,jp) == 0)                  % check for a zero pivot
 5       rc=1;                           % signal the error
 6       return                          % and give up
 7     end                               % finished checking
 8
 9    for ii=1:mm                        % update tableau rows
10        if(ii == ip) continue; end     % except for pivot row
11        for jj=1:nn                    % update non-pivot columns
12            if(jj == jp) continue; end
13            Tnew(ii,jj)=T(ii,jj)-T(ip,jj)*T(ii,jp)/T(ip,jp);
14        end
15    end                                % advance to the next row
16
17    for jj=1:nn                        % update pivot row
18        if(jj == jp) continue; end     % except for pivot column
19        Tnew(ip,jj)=T(ip,jj)/T(ip,jp); % divide by the pivot element
20    end                                % advance to next column
21
22    for ii=1:mm                        % update pivot column
23        if(ii == ip)                   % making the pivot element
24           Tnew(ii,jp)=1;              % exactly 1
25        else                           % and the other elements
26           Tnew(ii,jp)=0;              % exactly 0
27        end                            % finished testing
28    end                                % finished with the column
29
30    for jj=2:nn                        % update the basis
31        if(S(jj-1) == ip)              % mark outgoing column
32           Snew(jj-1)=0;               % nonbasic
33        else                           % while keeping
34           Snew(jj-1)=S(jj-1);         % the other columns unchanged
35        end                            % finished testing
36    end                                % finished removing outgoing
37    Snew(jp-1)=ip;                     % mark incoming column basic
38
39    rc=0;                              % signal success
40 end                                   % and return to the caller
```

element. Then 22-28 we update the pivot column, making the pivot element 1 and the others 0; this is to prevent roundoff errors from making the elements of the new identity column slightly different from 1 and 0. If the pivot element is negative, zeros in that row of the other identity columns remain zero but acquire a minus sign (see Exercise 2.10.25). Finally 30-37 the vector describing the basic sequence is revised to mark the column whose identity 1 was in the pivot row as nonbasic 32 and the pivot column as basic 37.

I tested this routine by using it to perform the pivots we earlier did by hand, as shown in the Octave session on the next page. First 1> I gave the variable `T0` the contents of tableau $\mathbf{T}_0$. Then 2> I set `S0` to describe the basic sequence of `T0` according to the scheme described above: the first four variable columns are nonbasic, then the basic columns have their 1 entries in rows 2, 3, and 4 of the tableau. Next 3> I set Octave's output format so that the result tableaus will fit on the screen. The first invocation of `pivot.m` 4> produces `T1`, which has basic sequence `S1`, and the second invocation 5> produces `T2` and `S2`.


```
octave:1> T0=[0,-90,-150,-60,-70,0,0,0;
>             160,7,10,8,12,1,0,0;
>             50,1,3,1,1,0,1,0;
>             60,2,4,1,3,0,0,1]
T0 =

     0   -90  -150   -60   -70     0     0     0
   160     7    10     8    12     1     0     0
    50     1     3     1     1     0     1     0
    60     2     4     1     3     0     0     1

octave:2> S0=[0,0,0,0,2,3,4];
octave:3> format bank
octave:4> [T1,S1,rc]=pivot(T0,4,8,4,3,S0)
T1 =

  2250.00   -15.00     0.00   -22.50    42.50     0.00     0.00    37.50
    10.00     2.00     0.00     5.50     4.50     1.00     0.00    -2.50
     5.00    -0.50     0.00     0.25    -1.25     0.00     1.00    -0.75
    15.00     0.50     1.00     0.25     0.75     0.00     0.00     0.25

S1 =

  0.00  4.00  0.00  0.00  2.00  3.00  0.00

rc = 0.00
octave:5> [T2,S2,rc]=pivot(T1,4,8,2,4,S1)
T2 =

  2290.91    -6.82     0.00     0.00    60.91     4.09     0.00    27.27
     1.82     0.36     0.00     1.00     0.82     0.18     0.00    -0.45
     4.55    -0.59     0.00     0.00    -1.45    -0.05     1.00    -0.64
    14.55     0.41     1.00     0.00     0.55    -0.05     0.00     0.36

S2 =

  0.00  4.00  2.00  0.00  0.00  3.00  0.00

rc = 0.00
octave:6>
```

2.4.3 Finding a Better Solution

In §2.4.1 we reasoned that pivoting in the x_1 column of $\mathbf{T}_2$ would yield a new basic feasible solution having an objective value lower than z = `-2290.91`. There are three possible pivot positions in that column so I tried them all, obtaining the results shown on the next page.

Pivoting at `T2(3,2)=-0.59` [7>] yields tableau `T3a`, which has b_2 = `-7.69` < 0 and is therefore not in canonical form. Pivoting on a negative a_{hp} in a canonical-form tableau always makes a positive b_h negative and thereby destroys canonical form. The basic solution represented by this tableau is $\mathbf{x}^{3a}$ = `[-7.69,17.69,4.62,0,0,0,0]`$^\top$, which violates the nonnegativity constraint $x_1 \geq 0$ and therefore cannot be a basic *feasible* solution.

Pivoting at `T2(4,2)=0.41` [8>] yields tableau `T3b`, which has b_1 = `-11.11` < 0 and is therefore also not in canonical form. To zero out a_{12} = `0.36` it was necessary to subtract


```
octave:6> T2
T2 =
```

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
2290.91	-6.82	0.00	0.00	60.91	4.09	0.00	27.27
1.82	0.36	0.00	1.00	0.82	0.18	0.00	-0.45
4.55	-0.59	0.00	0.00	-1.45	-0.05	1.00	-0.64
14.55	0.41	1.00	0.00	0.55	-0.05	0.00	0.36

```
octave:7> [T3a]=pivot(T2,4,8,3,2,S2)
T3a =
```

2238.46	0.00	0.00	0.00	77.69	4.62	-11.54	34.62
4.62	0.00	0.00	1.00	-0.08	0.15	0.62	-0.85
-7.69	1.00	-0.00	-0.00	2.46	0.08	-1.69	1.08
17.69	0.00	1.00	0.00	-0.46	-0.08	0.69	-0.08

```
octave:8> [T3b]=pivot(T2,4,8,4,2,S2)
T3b =
```

2533.33	0.00	16.67	0.00	70.00	3.33	0.00	33.33
-11.11	0.00	-0.89	1.00	0.33	0.22	0.00	-0.78
25.56	0.00	1.44	0.00	-0.67	-0.11	1.00	-0.11
35.56	1.00	2.44	0.00	1.33	-0.11	0.00	0.89

```
octave:9> [T3c]=pivot(T2,4,8,2,2,S2)
T3c =
```

2325.00	0.00	0.00	18.75	76.25	7.50	0.00	18.75
5.00	1.00	0.00	2.75	2.25	0.50	0.00	-1.25
7.50	0.00	0.00	1.62	-0.12	0.25	1.00	-1.38
12.50	0.00	1.00	-1.12	-0.38	-0.25	0.00	0.88

```
octave:10> quit
```

`0.36/0.41` ≈ 0.88 times the pivot row from the second row of the tableau, making b_1 negative. The basic solution represented by this tableau is $\mathbf{x}^{3b}$ = `[35.56,0,-11.11,0,0,25.56,0]`$^\top$, which violates the nonnegativity $x_3 \geq 0$, so it is not feasible either.

Pivoting at `T2(2,2)=0.36` `9>` yields `T3c`. This tableau has $\mathbf{b} \geq \mathbf{0}$ and the three identity columns with zero costs over them, so it is in canonical form with the basic feasible solution $\mathbf{x}^{3c} = \mathbf{x}^\star$ = `[5,12.5,0,0,0,7.5,0]`$^\top$ and objective value $z^\star$ = `-2325` < `-2290.91`.

This example has shown that if we pick a suitable pivot position it is possible to reduce the objective value by pivoting from one canonical-form tableau to another canonical-form tableau. The simplex algorithm does this repeatedly, eventually generating a tableau whose basic feasible solution is the optimal point. The pivots we performed to get from `T0` to `T1` to `T2` to `T3c` are in fact simplex algorithm pivots for solving the `brewery` problem.

2.4.4 The Simplex Pivot Rule

Could we have found the right pivot location in tableau `T2` without trying every possible a_{h1}? Our goal was to decrease the objective by making x_1 positive, so instead of pivoting we could think of increasing x_1 gradually by setting it equal to some number $t \geq 0$ while keeping

the other nonbasic variables $x_4 = x_5 = x_7 = 0$. How must the basic variables x_2, x_3, and x_6 change to keep the constraints $\mathbf{Ax} = \mathbf{b}$ satisfied? The constraint rows of $\mathbf{T}_2$ require that

$$\begin{array}{rcrcrcr} 1.82 & = & 0.36x_1 + x_3 & \Rightarrow & x_3 & = & 1.82 - 0.36t \\ 4.55 & = & -0.59x_1 + x_6 & \Rightarrow & x_6 & = & 4.55 + 0.59t \\ 14.55 & = & 0.41x_1 + x_2 & \Rightarrow & x_2 & = & 14.55 - 0.41t \end{array}$$

so to remain feasible we must make

$$\mathbf{x}(t) = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} t \\ 14.55 - 0.41t \\ 1.82 - 0.36t \\ 0 \\ 0 \\ 4.55 + 0.59t \\ 0 \end{bmatrix}.$$

When $t = 0$ we have $\mathbf{x}(t) = [0,14.55,1.82,0,0,4.55,0]^\top$, which is the basic feasible solution $\mathbf{x}^2$ corresponding to T2; when $t = 5$ we have $\mathbf{x}(t) = [5,12.5,0,0,0,7.5,0]^\top$, which is the basic feasible solution $\mathbf{x}^{3c}$ corresponding to T3c.

To reduce z as much as possible we want to make t as high as possible while keeping $\mathbf{x}(t) \geq \mathbf{0}$, so t must satisfy these inequalities.

$$\left.\begin{array}{rcl} t & \geq & 0 \quad \checkmark \\ 14.55 - 0.41t & \geq & 0 \quad \Rightarrow \quad t \leq 14.55/0.41 = b_3/a_{31} = 35.49 \\ 1.82 - 0.36t & \geq & 0 \quad \Rightarrow \quad t \leq 1.82/0.36 = b_1/a_{11} = 5.00 \\ 0 & \geq & 0 \quad \checkmark \\ 0 & \geq & 0 \quad \checkmark \\ 4.55 + 0.59t & \geq & 0 \quad \Rightarrow \quad t \geq 4.55/(-0.59) = b_2/a_{22} = -7.71 \\ 0 & \geq & 0 \quad \checkmark \end{array}\right\} \Rightarrow t \leq 5$$

It was the pivot on a_{11} that produced the canonical-form tableau T3c, and now we can see why: among the positive a_{h1} in the pivot column, a_{11} has the lowest ratio b_h/a_{h1}.

We can also see where to pivot in the x_1 column of $\mathbf{T}_2$ by noticing that in the $\mathbf{x}(t)$ we found above, as t is increased from zero both x_2 and x_3 decrease. The first to become zero is x_3, so that variable leaves the basis as x_1 enters the basis. In $\mathbf{T}_2$ the x_3 identity column has its 1 in the first row, so that must be the pivot row.

We chose $p = 1$ as the pivot *column* because $c_1 = -1.82$ is negative and z will therefore be decreased by introducing x_1 into the basis. We chose $h = 1$ as the pivot *row* because pivoting there keeps $\mathbf{b} \geq \mathbf{0}$ and thereby preserves canonical form. These two ideas are combined in the summary given at the top of the next page.

In a canonical-form tableau having one or more $c_j < 0$, to decrease z perform a pivot on a_{hp} according to this **simplex pivot rule**:

- choose the pivot column p so that $c_p < 0$;
- choose the pivot row h so that

$$\frac{b_h}{a_{hp}} = \min_i \left\{ \frac{b_i}{a_{ip}} \,\middle|\, a_{ip} > 0 \right\}.$$

This choice of h pivots on the positive a_{ip} where the ratio b_i/a_{ip} is smallest, so the pivot is called the **minimum-ratio pivot** in column p.

2.5 Final Forms

We begin solving any linear programming problem in the fervent hope that some sequence of simplex-rule pivots will lead to an optimal basic feasible solution, but that future is only one of several that might possibly come to pass.

2.5.1 Optimal Form

When we solved the `brewery` problem in §2.4 a sequence of simplex-rule pivots led to `T3c`. It has the associated basic feasible solution $\mathbf{x}^\star = \texttt{[5,12.5,0,0,0,7.5,0]}^\top$, in which x_3, x_4, x_5, and x_7 are nonbasic. The objective row of that tableau represents this equation.

$$z + 2325 = 18.75x_3 + 76.25x_4 + 7.5x_5 + 18.75x_7$$

Because these reduced costs are all positive, increasing any of the nonbasic variables from zero could only *increase* the objective value.

$$z = -2325 + \underbrace{[18.75x_3 + 76.25x_4 + 7.5x_5 + 18.75x_7]}_{\geq 0}$$

Thus $z(\mathbf{x}) \geq z(\mathbf{x}^\star)$ for all $\mathbf{x} \geq \mathbf{0}$, and $\mathbf{x}^\star$ must be optimal. A canonical-form tableau having $\mathbf{c} \geq \mathbf{0}$ is in **optimal form**.

	≥ 0 ≥ 0 $\cdots$ ≥ 0
	canonical-form tableau

2.5.2 Unbounded Form

Now consider a new example, which I will call the `unbd` problem (see §28.5.8).

	x_1	x_2	x_3	x_4	x_5
-9	0	0	-2	1	0
3	0	0	-1	2	1
1	1	0	0	1	0
5	0	1	-4	1	0

This tableau is in canonical form and has $c_3 < 0$, so we can reduce the objective by increasing x_3 while we keep x_4 nonbasic. No a_{h3} is positive so we cannot do this by pivoting according to the simplex rule, but we can let $x_3 = t \geq 0$ and study what happens as we increase t gradually. To remain feasible for $\mathbf{Ax} = \mathbf{b}$ we must simultaneously adjust the basic variables to satisfy the constraint equations.

$$\begin{array}{rclcrcl} 3 & = & -t + x_5 & \Rightarrow & x_5 & = & 3 + t \\ 1 & = & x_1 & \Rightarrow & x_1 & = & 1 \\ 5 & = & x_2 - 4t & \Rightarrow & x_2 & = & 5 + 4t \end{array}$$

To remain feasible for $\mathbf{x} \geq \mathbf{0}$ requires that

$$\begin{array}{rclcl} x_1 & = & 1 & \geq 0 & \checkmark \\ x_2 & = & 5 + 4t & \geq 0 & \Rightarrow \quad t \geq -\frac{5}{4} \\ x_3 & = & t & \geq 0 & \checkmark \\ x_4 & = & 0 & \geq 0 & \checkmark \\ x_5 & = & 3 + t & \geq 0 & \Rightarrow \quad t \geq -3 \end{array}$$

but these conditions are satisfied for or *all* $t \geq 0$. The objective row of the tableau says that $z - 9 = -2t$ so $z = 9 - 2t$ and by increasing t indefinitely we can make z as low as we like. This problem has *no* optimal vector, and informally we will say that $z^\star = -\infty$.

When a canonical-form tableau has some $c_j < 0$ but $a_{ij} \leq 0$ for all $i \in \{1 \ldots m\}$ it is in **unbounded form**.

	< 0
canonical-form tableau	≤ 0 ≤ 0 $\vdots$ ≤ 0 ≤ 0

2.5.3 Infeasible Forms

If a tableau is in canonical form then the linear program it represents has at least one feasible point, namely the basic feasible solution associated with the tableau. In §2.8 you will learn how to put any tableau into canonical form if the problem has one.

But not every problem does, because not every linear program is feasible. If there is no $\mathbf{x} \geq \mathbf{0}$ that simultaneously satisfies all of the constraints $\mathbf{Ax} = \mathbf{b}$, then the search for an initial canonical form is sure to produce a tableau having a constraint row like either the second or the third constraint row in this tableau, which I will call the `infea` problem (see §28.5.9).

	x_1	x_2	x_3	x_4
2	0	0	−3	8
1	0	1	5	−1
4	0	0	0	0
−7	1	0	2	6

$$
\begin{array}{rcl}
z+2 & = & - 3x_3 + 8x_4 \\
1 & = & x_2 + 5x_3 - x_4 \\
4 & = & 0 \qquad\qquad\qquad\qquad\qquad\qquad \text{※} \\
-7 & = & x_1 + + 2x_3 + 6x_4 \qquad \text{※}
\end{array}
$$

The equations represented by this tableau are shown on the right. Nonnegative values of x_2, x_3, and x_4 can be found to satisfy the first constraint, but *no* $\mathbf{x}$ can satisfy the second constraint and no *nonnegative* $\mathbf{x}$ can satisfy the third. It is occasionally useful to distinguish between these two ways in which a linear program can be infeasible, so we will identify them as follows [3, p49-50].

infeasible form 1

$\neq 0$	$0 \quad 0 \quad \cdots \quad 0 \quad 0$

infeasible form 2

< 0	$\geq 0 \quad \geq 0 \quad \cdots \quad \geq 0 \quad \geq 0$

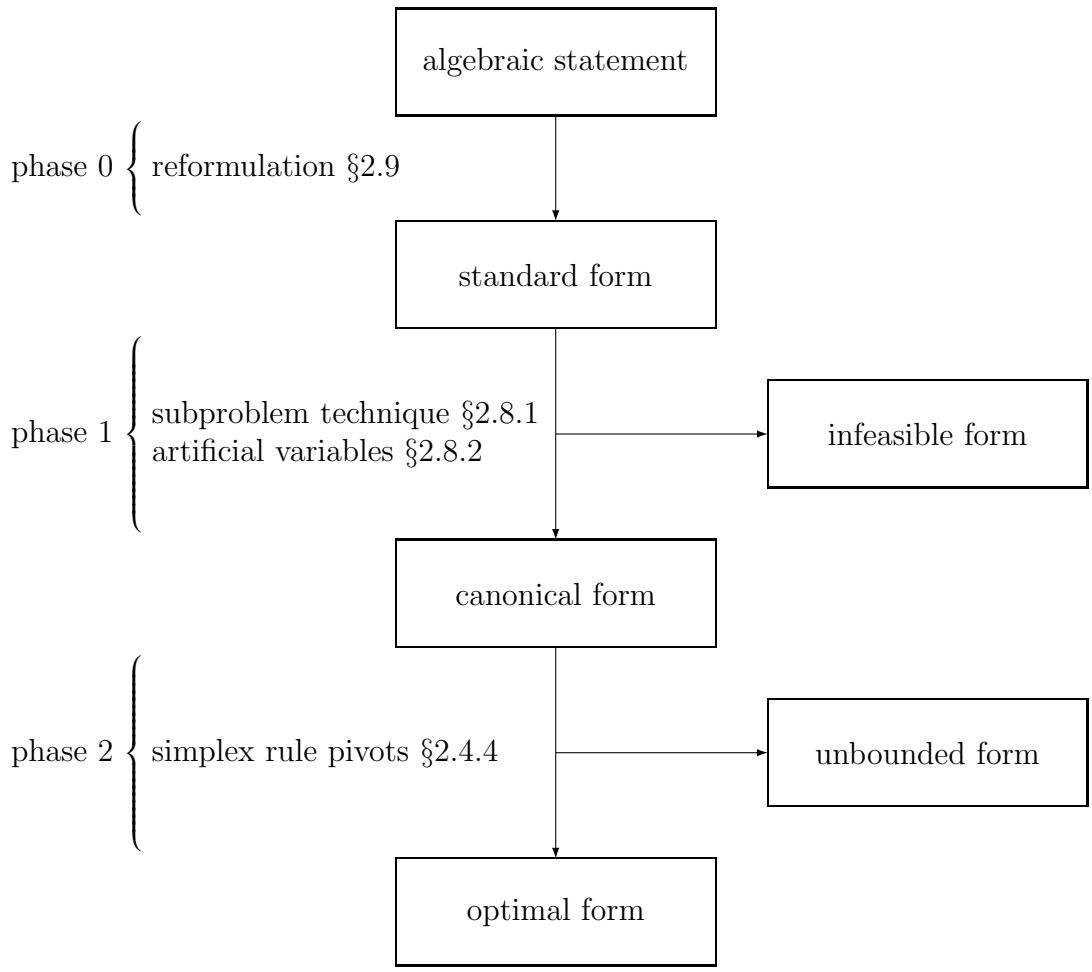
2.6 The Solution Process

In this book to "solve" a linear program means to

- find an optimal vector $\mathbf{x}^\star$ OR
- show that the problem has an unbounded objective and thus *no* optimal point OR
- show that the problem is infeasible and thus has no optimal point.

Some authors call this ***re***solving the problem, in the sense of deciding which of the three possible outcomes it has, but we will consider a linear program to have been solved if we get to any of the final forms described in §2.5.

The solution of a linear program by the simplex algorithm is traditionally [35, §5-2] divided into two phases. **Phase 1** finds an initial canonical form tableau for a problem that is already in standard form, or discovers that the problem is infeasible. **Phase 2** pivots an initial canonical form tableau to optimal form, or discovers that the problem is unbounded. **Phase 0** is what we will call the reformulation of an arbitrary linear program into standard form. The whole solution process is pictured below [3, p50].



You already know how to transform an initial canonical form tableau into either optimal or unbounded form, by repeatedly applying the simplex rule as we did in §2.4. There we adopted the `pivot.m` MATLAB function to automate the arithmetic of pivoting. In §2.8 and §2.9 we will take up phase 1 and phase 0 of the solution process. There it will be convenient to automate other tableau manipulations in addition to pivoting, so first we will pause to consider a much more powerful computational utility.

2.7 The `pivot` Program

In his magnum opus *The Art of Computer Programming,* Donald Knuth described a hypothetical computer and invented a machine language for it which he called `MIX` [94, p x-xi]. He then used this imaginary language to illustrate the algorithms and programming ideas that are the subject of his book, in the process making them independent of any particular computing environment and thus relevant to all of them. Since then several `MIX` simulators have been written in various real programming languages, but few of the many students who have learned from his book ever used one to actually run the programs.

Imitating his approach I have provided, by means of the user's manual in §27.1, the abstract definition of a hypothetical computer program named `pivot`. This imaginary utility automates pivoting and many other tableau manipulations, and from now on I will talk about it as though it were real. You can download my implementation of `pivot` by following the directions given in §27.2 or (far better) write your own, but you do not need to be able to run the program in order to understand the examples in which we will use it.

The `pivot` program (like `pivot.m`) refers to a tableau element by its row `i` and column `j` in the tableau rather than by its row h and column p in the $\mathbf{A}$ matrix. As an introduction to the program, I have used it below to solve the `brewery` problem by a different sequence of simplex-rule pivots. More of the program's features will become evident in future examples.

```
> This is PIVOT, Unix version 4.2
> For a list of commands, enter HELP.
>
< tableau 4 8
< insert
T( 1, 1)... = 0 -90 -150 -60 -70 0 0 0
T( 2, 1)... = 160 7 10 8 12 1 0 0
T( 3, 1)... = 50 1 3 1 1 0 1 0
T( 4, 1)... = 60 2 4 1 3 0 0 1

   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< pivot 4 3

 2250. -15.0  0. -22.50  42.50  0.  0.  37.50
   10.   2.0  0.   5.50   4.50  1.  0.  -2.50
    5.  -0.5  0.   0.25  -1.25  0.  1.  -0.75
   15.   0.5  1.   0.25   0.75  0.  0.   0.25

< p 2 2

 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875

< quit
> STOP
```


2.8 Getting Canonical Form

This tableau, representing a linear program that I will call `sf1` (see §28.5.10), is not in canonical form. Its **b** part has negative components, its **A** part contains only one column of the 5×5 identity, and the cost over that column is not zero.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	−8	6	2	0	−7	5	0
−1	0	−3	0	8	6	−4	3
−2	−9	7	0	−5	0	0	−9
3	−6	0	1	−7	4	−6	5
4	9	−5	0	0	3	9	4
1	0	−1	0	3	9	5	−2

There are two approaches to putting a tableau like this into canonical form: either get identity columns with zero costs first and then make **b** nonnegative, or make **b** nonnegative first and then get identity columns with zero costs.

2.8.1 The Subproblem Technique

The approach of getting the identity columns first is called the **subproblem technique** [145, §3.7] [3, §3.5]. Pivoting on any nonzero a_{hp} makes that element 1 and zeroes out the other elements in the pivot column. We will adopt a systematic way of doing this to generate the m identity columns, as follows.

pivoting-in a basis
let $h \leftarrow 1$
`1` find any nonzero element in row h of **A**
  if each $a_{hp} = 0$ and $b_h = 0$ the row is redundant; delete it, let $m \leftarrow m - 1$, and `GO TO 1`
  if each $a_{hp} = 0$ and $b_h \neq 0$, `STOP` with infeasible form 1
pivot on a_{hp}
if $h < m$ let $h \leftarrow h + 1$ and `GO TO 1`
`STOP` with m identity columns having zero costs.

Using this algorithm I found a basis for `sf1`, as shown in the `pivot` session excerpt on the next page. The first command reads the starting tableau from the text file `sf1.tab`, which is listed to the right. This algorithm pays no attention to the signs of the pivot elements or to the ratios a_{hp}/b_h, so it often produces negative **b** elements even if $\mathbf{b} \geq \mathbf{0}$ in the starting tableau (which here it is not). Pivoting-in a basis revealed a redundant constraint, so at the end I deleted the zero row 6.

```
6 8
     x1 x2 x3 x4 x5 x6 x7
  0  -8  6  2  0 -7  5  0
 -1   0 -3  0  8  6 -4  3
 -2  -9  7  0 -5  0  0 -9
  3  -6  0  1 -7  4 -6  5
  4   9 -5  0  0  3  9  4
  1   0 -1  0  3  9  5 -2
```



```
< read sf1.tab
Reading the tableau...
...done.

     x1  x2  x3  x4  x5  x6  x7
 0. -8.  6.  2.  0. -7.  5.  0.
-1.  0. -3.  0.  8.  6. -4.  3.
-2. -9.  7.  0. -5.  0.  0. -9.
 3. -6.  0.  1. -7.  4. -6.  5.
 4.  9. -5.  0.  0.  3.  9.  4.
 1.  0. -1.  0.  3.  9.  5. -2.

< p 2 5

        x1  x2     x3  x4  x5    x6   x7
 0.000 -8.  6.000  2.  0. -7.00  5.0  0.000
-0.125  0. -0.375  0.  1.  0.75 -0.5  0.375
-2.625 -9.  5.125  0.  0.  3.75 -2.5 -7.125
 2.125 -6. -2.625  1.  0.  9.25 -9.5  7.625
 4.000  9. -5.000  0.  0.  3.00  9.0  4.000
 1.375  0.  0.125  0.  0.  6.75  6.5 -3.125

< p 3 3

            x1         x2  x3  x4  x5         x6         x7
 3.0731707   2.536585  0.  2.  0. -11.390244   7.926829  8.3414634
-0.3170732  -0.658537  0.  0.  1.   1.024390  -0.682927 -0.1463415
-0.5121951  -1.756098  1.  0.  0.   0.731707  -0.487805 -1.3902439
 0.7804878 -10.609756  0.  1.  0.  11.170732 -10.780488  3.9756098
 1.4390244   0.219512  0.  0.  0.   6.658537   6.560976 -2.9512195
 1.4390244   0.219512  0.  0.  0.   6.658537   6.560976 -2.9512195

< p 4 4

            x1         x2  x3  x4  x5         x6         x7
 1.5121951  23.756098  0.  0.  0. -33.731707  29.487805  0.3902439
-0.3170732  -0.658537  0.  0.  1.   1.024390  -0.682927 -0.1463415
-0.5121951  -1.756098  1.  0.  0.   0.731707  -0.487805 -1.3902439
 0.7804878 -10.609756  0.  1.  0.  11.170732 -10.780488  3.9756098
 1.4390244   0.219512  0.  0.  0.   6.658537   6.560976 -2.9512195
 1.4390244   0.219512  0.  0.  0.   6.658537   6.560976 -2.9512195

< p 5 6

            x1         x2  x3  x4  x5  x6         x7
 8.8021978  24.868132  0.  0.  0.  0.  62.725275 -14.560440
-0.5384615  -0.692308  0.  0.  1.  0.  -1.692308   0.307692
-0.6703297  -1.780220  1.  0.  0.  0.  -1.208791  -1.065934
-1.6336996 -10.978022  0.  1.  0.  0. -21.787546   8.926740
 0.2161172   0.032967  0.  0.  0.  1.   0.985348  -0.443223
 0.0000000   0.000000  0.  0.  0.  0.   0.000000   0.000000

< delete 6 0

            x1         x2  x3  x4  x5  x6         x7
 8.8021978  24.868132  0.  0.  0.  0.  62.725275 -14.560440
-0.5384615  -0.692308  0.  0.  1.  0.  -1.692308   0.307692
-0.6703297  -1.780220  1.  0.  0.  0.  -1.208791  -1.065934
-1.6336996 -10.978022  0.  1.  0.  0. -21.787546   8.926740
 0.2161172   0.032967  0.  0.  0.  1.   0.985348  -0.443223
```


The final tableau on the previous page has $b_1 < 0$, $b_2 < 0$, and $b_3 = -1.6336996 < 0$. Ignoring b_1 and b_2 for the moment, how might we make b_3 less negative than it is, while keeping b_4 nonnegative? Recall from §2.4 that the number in the upper left corner of a canonical-form tableau is always $-z$, and that pivoting by the simplex rule minimizes z which increases $-z$.

If the third constraint row were the objective row of a linear program, then b_3 would be the negative of that problem's objective value and we could increase it by pivoting that linear program toward optimality. To keep b_4 nonnegative while we did that we could include that row as a constraint. Below I have outlined a **subproblem** in which b_3 is the $-z$ element of a tableau whose only constraint is the original row having $b_4 > 0$.

```
< digits 4
> Display precision is set to  4 digits.
< list
```

		x1	x2	x3	x4	x5	x6	x7
	8.802	24.87	0.	0.	0.	0.	62.73	-14.56
$b_1 =$	-0.538	-0.69	0.	0.	1.	0.	-1.69	0.31
$b_2 =$	-0.670	-1.78	1.	0.	0.	0.	-1.21	-1.07
$b_3 =$	-1.634	-10.98	0.	1.	0.	0.	-21.79	8.93
$b_4 =$	0.216	0.03	0.	0.	0.	1.	0.99	-0.44

subproblem

The subproblem tableau is in canonical form, because its single b_h is $0.216 > 0$ and it has one identity column with a zero cost (the x_5 column). There are two possible simplex-rule pivot elements in the subproblem, the 0.03 and the 0.99; I arbitrarily picked the 0.99 because it has the most negative reduced cost, and pivoted *the whole tableau.* Although our choice of a pivot position is guided by the subproblem, the other rows must also be included in the pivot to preserve $m = 4$ identity columns and keep the new tableau equivalent to the original.

```
< p 5 7
```

		x1	x2	x3	x4	x5	x6	x7
	-4.955	22.77	0.	0.	0.	-63.66	0.	13.65
$b_1 =$	-0.167	-0.64	0.	0.	1.	1.72	0.	-0.45
$b_2 =$	-0.405	-1.74	1.	0.	0.	1.23	0.	-1.61
$b_3 =$	3.145	-10.25	0.	1.	0.	22.11	0.	-0.87
$b_4 =$	0.219	0.03	0.	0.	0.	1.01	1.	-0.45

The pivot made $b_3 > 0$ but left $b_2 < 0$, so we can form a new subproblem having that element as its upper left corner. Now both b_3 and b_4 are nonnegative, and to ensure that they stay that way those constraint rows must be included in the new subproblem. Unfortunately, this subproblem is unbounded (in the x_7 column) so we cannot increase b_2 by pivoting the subproblem toward optimality.

Fortunately, pivoting on the negative subproblem objective entry in the unbounded column will make b_2 positive while keeping b_3 and b_4 nonnegative. To see why this happens it is helpful to examine the details of the pivot operation. First we divide the pivot row by the pivot element a_{27} = `-1.61`, which makes that element 1 and b_2 positive. Then we add multiples of this new pivot row to the constraint rows of the subproblem, to make the other elements in the subproblem pivot column zero. Because a_{37} = `-0.87` and a_{47} = `-0.45` are both negative, the needed multiples are positive. But b_2 is now positive, so adding positive multiples of it to b_3 and b_4 will keep them positive.

```
< p 3 8

                x1      x2      x3  x4   x5      x6  x7
     -8.393  8.012   8.483   0.  0. -53.25   0.  0.
b1 = -0.053 -0.145  -0.282   0.  1.   1.37   0.  0.
b2 =  0.252  1.081  -0.621   0.  0.  -0.76   0.  1.
b3 =  3.365 -9.305  -0.543   1.  0.  21.45   0.  0.
b4 =  0.333  0.520  -0.279   0.  0.   0.67   1.  0.
```

Notice that in addition to making b_2 positive, the pivot increased b_3 and b_4 as we predicted. It left b_1 negative so we form a final subproblem, which also happens to be unbounded (in the x_2 column). Pivoting on that subproblem objective element yields this canonical form.

```
< p 2 3

          x1      x2  x3  x4     x5     x6  x7
 -9.992  3.631  0.  0.  30.11 -11.95  0.  0.
  0.189  0.516  1.  0.  -3.55  -4.87  0.  0.
  0.369  1.402  0.  0.  -2.20  -3.79  0.  1.
  3.467 -9.025  0.  1.  -1.93  18.80  0.  0.
  0.385  0.664  0.  0.  -0.99  -0.69  1.  0.
```

Can the subproblem technique be used if the starting tableau has *every* $b_h < 0$? In this example, which I will call `sf2` (see §28.5.11), we cannot form a subproblem in the usual way.

```
< read sf2.tab
Reading the tableau...
...done.

      x1  x2  x3  x4  x5  x6
  0.  0.  0.  4. -1.  2.  0.
-15.  0.  0. -1.  1. -1.  1.
 -8.  1.  0.  0. -1.  0.  0.
 -5.  0.  1. -1.  3. -2.  0.
```


But if the first constraint row is considered the objective in a subproblem that has *no* constraints and is thus unbounded, then we can pivot on either a_{13} or a_{15}.

```
< p 2 6

      x1  x2  x3  x4  x5  x6
 -30.  0.  0.  2.  1.  0.  2.
  15.  0.  0.  1. -1.  1. -1.
  -8.  1.  0.  0. -1.  0.  0.
  25.  0.  1.  1.  1.  0. -2.
```

The pivot on a_{15} resulted in $b_1 > 0$ and $b_3 > 0$, so both of those constraints must be included in a subproblem to increase b_2. Rearranging the rows makes this subproblem easy to visualize (it is always prudent to do this when solving a problem by hand) and then one pivot achieves canonical form.

```
< swap 2 3

      x1  x2  x3  x4  x5  x6
 -30.  0.  0.  2.  1.  0.  2.
  -8.  1.  0.  0. -1.  0.  0.
  15.  0.  0.  1. -1.  1. -1.
  25.  0.  1.  1.  1.  0. -2.

< p 4 5

      x1  x2  x3  x4  x5  x6
 -55.  0. -1.  1.  0.  0.  4.
  17.  1.  1.  1.  0.  0. -2.
  40.  0.  1.  2.  0.  1. -3.
  25.  0.  1.  1.  1.  0. -2.
```

In §4.1 we will implement the subproblem technique in MATLAB, and then instead of swapping rows we will maintain a list of the indices of the rows that are in each subproblem. That will let us use the same code to solve the subproblems and the canonical-form tableau that is discovered by the subproblem technique.

Subproblems are in canonical form by construction. It might take more than one pivot to get a subproblem's $-z$ entry nonnegative, but once that is achieved we construct the next larger subproblem rather than solving the current one to optimality. Sometimes solving a subproblem to make one b_h nonnegative also makes others nonnegative. Pivoting a subproblem toward optimality might reveal that it is unbounded, in which case we pivot in its objective row and that makes its $-z$ entry positive. If a subproblem reaches optimal form with its $-z$ entry still negative, the original problem is in infeasible form 2.

Now we can summarize the procedure we have developed for making the b_h nonnegative in a tableau that has a basis.

getting b nonnegative
if every $b_h < 0$ then
  if any constraint row h has $a_{hp} \geq 0$ for all $p \in \{1 \ldots n\}$, `STOP` with infeasible form 2
  otherwise pivot on any negative entry in the first constraint row
`1` if every $b_h \geq 0$, `STOP` with canonical form
if some $b_h < 0$ then
  form a subproblem with that row as objective and all rows with $b_h \geq 0$ as constraints
  if the subproblem is unbounded pivot in that column of its objective and `GO TO 1`
  otherwise pivot the subproblem towards optimality by simplex rule pivots
    if the subproblem's optimal $-z$ entry is negative, `STOP` with infeasible form 2
    otherwise when the subproblem's $-z$ entry becomes nonnegative, `GO TO 1`

When we write MATLAB code for the simplex method in §4.1 the subproblem technique will consist of two routines, one for pivoting-in a basis and one that implements this algorithm.

2.8.2 The Method of Artificial Variables

The other approach to getting canonical form [145, §3.6] [3, §3.8] begins by multiplying every constraint row that has a negative b_h through by -1, to make $\mathbf{b} \geq \mathbf{0}$. In this form the linear program is called the **original problem**.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & z & = & \mathbf{c}^\top\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \geq \mathbf{0} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

The $\mathbf{A}$ matrix in the original problem does not necessarily contain any basis columns, so we append the identity columns to the tableau and do some pivots to move them into the $\mathbf{A}$ part as basis columns. To accomplish that we form and solve this **artificial problem**.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n\ \mathbf{y}\in\mathbb{R}^m}{\text{minimize}} & \mathbf{1}^\top\mathbf{y} & = & y_1 + \cdots + y_m \\ \text{subject to} & \mathbf{A}\mathbf{x} + \mathbf{I}\mathbf{y} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

Here $\mathbf{1}$ is a column vector of m 1's and the y_i are called **artificial variables**. Because $\mathbf{b}$ is nonnegative the constraints of the artificial problem can be satisfied by letting $\mathbf{x} = \mathbf{0}$ and $\mathbf{y} = \mathbf{b}$, so every artificial problem is feasible. Because $\mathbf{y} \geq \mathbf{0}$ the **artificial objective** $\mathbf{1}^\top\mathbf{y}$ is always nonnegative, so the minimization of $\mathbf{1}^\top\mathbf{y}$ subject to $\mathbf{y} \geq \mathbf{0}$ will try to make it zero.

If $(\mathbf{x}^\star, \mathbf{y}^\star)$ is optimal for the artificial problem and the artificial objective has an optimal value of zero, then

$$\left.\begin{array}{rcl} \mathbf{1}^\top\mathbf{y}^\star & = & 0 \\ \mathbf{y}^\star & \geq & \mathbf{0} \end{array}\right\} \Rightarrow \mathbf{y}^\star = \mathbf{0} \qquad \text{but} \qquad \left.\begin{array}{rcl} \mathbf{y}^\star & = & \mathbf{0} \\ \mathbf{A}\mathbf{x}^\star + \mathbf{I}\mathbf{y}^\star & = & \mathbf{b} \end{array}\right\} \Rightarrow \mathbf{A}\mathbf{x}^\star = \mathbf{b}$$

so $\mathbf{x}^\star$ is feasible for the original problem.

Conversely, if $\mathbf{x}^\star$ is feasible for the original problem then it satisfies $\mathbf{A}\mathbf{x}^\star = \mathbf{b}$ and

$$\left.\begin{array}{rcl} \mathbf{A}\mathbf{x}^\star & = & \mathbf{b} \\ \mathbf{A}\mathbf{x}^\star + \mathbf{I}\mathbf{y}^\star & = & \mathbf{b} \end{array}\right\} \Rightarrow \mathbf{b} + \mathbf{I}\mathbf{y}^\star = \mathbf{b} \Rightarrow \mathbf{y}^\star = \mathbf{0} \Rightarrow \mathbf{1}^\top\mathbf{y}^\star = 0$$

so the optimal value of the artificial objective is zero.

Thus the original problem is feasible if and only if the artificial problem has an optimal value of zero. In that case we can get an initial canonical form for the original problem from the $\mathbf{x}$ part of the optimal tableau for the artificial problem, as shown by this example.

```
> This is PIVOT, Unix version 4.2
> For a list of commands, enter HELP.
>
< read sf1.tab
Reading the tableau...
...done.

     x1  x2  x3  x4  x5  x6  x7
 0. -8.  6.  2.  0. -7.  5.  0.
-1.  0. -3.  0.  8.  6. -4.  3.
-2. -9.  7.  0. -5.  0.  0. -9.
 3. -6.  0.  1. -7.  4. -6.  5.
 4.  9. -5.  0.  0.  3.  9.  4.
 1.  0. -1.  0.  3.  9.  5. -2.

< * first we multiply rows with negative b's through by -1
< scale 2 0 -1;
< scale 3 0 -1

     x1  x2  x3  x4  x5  x6  x7
 0. -8.  6.  2.  0. -7.  5.  0.
 1.  0.  3.  0. -8. -6.  4. -3.
 2.  9. -7.  0.  5.  0.  0.  9.
 3. -6.  0.  1. -7.  4. -6.  5.
 4.  9. -5.  0.  0.  3.  9.  4.
 1.  0. -1.  0.  3.  9.  5. -2.

< * this is the "original" problem; now we form the artificial
< append 0 5;   * append 5 columns of zeros
< insert 2 9;   * and make them the identity columns
T( 2, 9) = 1    * by putting 1's
< insert 3 10;  * on the diagonal
T( 3,10) = 1
< insert 4 11;
T( 4,11) = 1
< insert 5 12;
T( 5,12) = 1
< insert 6 13;
T( 6,13) = 1
```



```
< names x1 x2 x3 x4 x5 x6 x7 y1 y2 y3 y4 y5

      x1  x2  x3  x4  x5  x6  x7  y1  y2  y3  y4  y5
  0. -8.  6.  2.  0. -7.  5.  0.  0.  0.  0.  0.  0.
  1.  0.  3.  0. -8. -6.  4. -3.  1.  0.  0.  0.  0.
  2.  9. -7.  0.  5.  0.  0.  9.  0.  1.  0.  0.  0.
  3. -6.  0.  1. -7.  4. -6.  5.  0.  0.  1.  0.  0.
  4.  9. -5.  0.  0.  3.  9.  4.  0.  0.  0.  1.  0.
  1.  0. -1.  0.  3.  9.  5. -2.  0.  0.  0.  0.  1.

< * next we replace the objective by the artificial objective
< insert 1 0
T( 1, 1)... = 0 0 0 0 0 0 0 0 1 1 1 1 1

      x1  x2  x3  x4  x5  x6  x7  y1  y2  y3  y4  y5
  0.  0.  0.  0.  0.  0.  0.  0.  1.  1.  1.  1.  1.
  1.  0.  3.  0. -8. -6.  4. -3.  1.  0.  0.  0.  0.
  2.  9. -7.  0.  5.  0.  0.  9.  0.  1.  0.  0.  0.
  3. -6.  0.  1. -7.  4. -6.  5.  0.  0.  1.  0.  0.
  4.  9. -5.  0.  0.  3.  9.  4.  0.  0.  0.  1.  0.
  1.  0. -1.  0.  3.  9.  5. -2.  0.  0.  0.  0.  1.

< * pivoting on the identity column 1's makes those costs zero
< p 2 9;
< p 3 10;
< p 4 11;
< p 5 12;
< p 6 13

       x1   x2   x3  x4  x5   x6   x7   y1  y2  y3  y4  y5
 -11. -12.  10. -1.  7. -10. -12. -13.  0.  0.  0.  0.  0.
   1.   0.   3.  0. -8.  -6.   4.  -3.  1.  0.  0.  0.  0.
   2.   9.  -7.  0.  5.   0.   0.   9.  0.  1.  0.  0.  0.
   3.  -6.   0.  1. -7.   4.  -6.   5.  0.  0.  1.  0.  0.
   4.   9.  -5.  0.  0.   3.   9.   4.  0.  0.  0.  1.  0.
   1.   0.  -1.  0.  3.   9.   5.  -2.  0.  0.  0.  0.  1.

< * now the artificial problem is in canonical form
< digits 4
> Display precision is set to  4 digits.
< solve

        x1    x2  x3  x4    x5     x6  x7  y1    y2     y3  y4  y5
 -0.00 +0.00  0.  0. -0.00  -0.00  0.  0.  2.00  2.000  1.  0.  2.000
 +0.00 +0.00  0.  0. -0.00  -0.00  0.  0. -1.00 -1.000  0.  1. -1.000
  0.37  1.40  0.  0. -2.20  -3.79  0.  1.  0.29  0.156  0.  0. -0.230
  3.47 -9.02  0.  1. -1.93  18.80  0.  0. -0.30 -0.336  1.  0.  1.443
  0.39  0.66  0.  0. -0.99  -0.69  1.  0.  0.19  0.074  0.  0.  0.049
  0.19  0.52  1.  0. -3.55  -4.87  0.  0.  0.37  0.057  0.  0. -0.295

< * optimal objective value is zero so original problem is feasible
< * row 2 is zeros in the x part so that constraint is redundant
< delete 2 0;
< delete 0 12

         x1     x2  x3  x4     x5     x6  x7  y1     y2     y3  y5
 -0.000 +0.000  0.  0. -0.000  -0.00  0.  0.  2.000  2.000  1.  2.000
  0.369  1.402  0.  0. -2.205  -3.79  0.  1.  0.287  0.156  0. -0.230
  3.467 -9.025  0.  1. -1.926  18.80  0.  0. -0.303 -0.336  1.  1.443
  0.385  0.664  0.  0. -0.992  -0.69  1.  0.  0.189  0.074  0.  0.049
  0.189  0.516  1.  0. -3.549  -4.87  0.  0.  0.369  0.057  0. -0.295
```



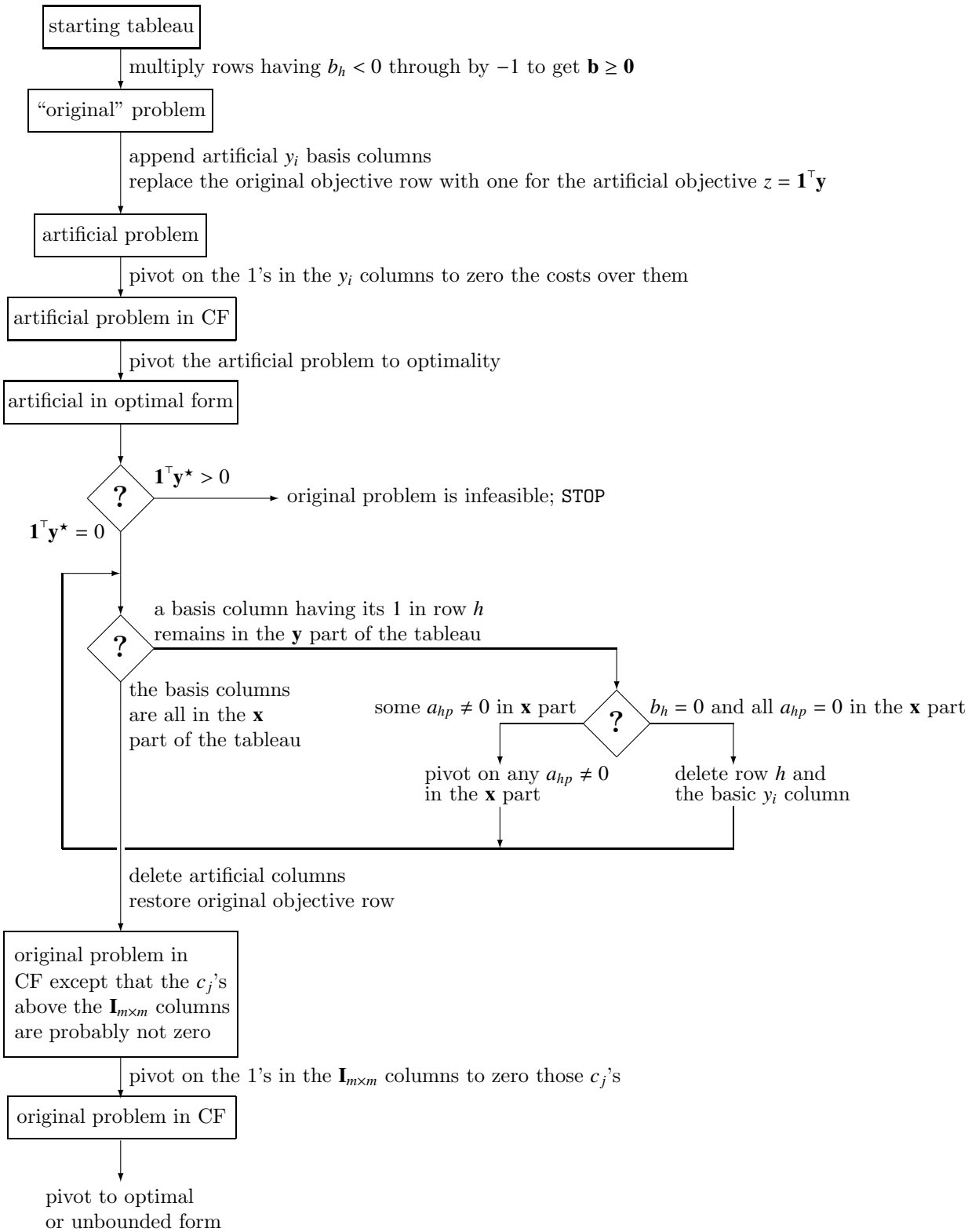
```
< * the basic columns are all in the x part
< * delete artificial columns and restore original objective row
< delete 0 9;
< delete 0 9;
< delete 0 9;
< delete 0 9;
< insert 1 0
T( 1, 1)... =   0  -8  6  2  0 -7  5  0

         x1     x2  x3  x4     x5     x6  x7
  0.000 -8.000  6.  2.  0.000  -7.00  5.  0.
  0.369  1.402  0.  0. -2.205  -3.79  0.  1.
  3.467 -9.025  0.  1. -1.926  18.80  0.  0.
  0.385  0.664  0.  0. -0.992  -0.69  1.  0.
  0.189  0.516  1.  0. -3.549  -4.87  0.  0.

< * pivoting on the identity 1's makes those costs zero
< p 5 3;
< p 3 4;
< p 4 7;
< p 2 8

          x1     x2  x3  x4     x5     x6  x7
  -9.992  3.631  0.  0.  30.11 -11.95  0.  0.
   0.369  1.402  0.  0.  -2.20  -3.79  0.  1.
   3.467 -9.025  0.  1.  -1.93  18.80  0.  0.
   0.385  0.664  0.  0.  -0.99  -0.69  1.  0.
   0.189  0.516  1.  0.  -3.55  -4.87  0.  0.

< * this is a canonical form for the original problem
```

Most linear programs have many possible canonical forms, so the artificial problem typically has multiple optimal solutions (see Exercise 2.10.67) and the one we find might leave some y_i basic. If the optimal value of the artificial problem is not zero ($\mathbf{1}^\top \mathbf{y}^\star > 0$) then the original problem is infeasible and it is not possible to make all of the y_i basic. If some y_i remain in the basis but the artificial problem has an optimal value of zero, then those y_i can and must be made nonbasic to complete the construction of a basis for the original problem. There are two possible cases.

1. The basic y_i column has its 1 in row h, $b_h = 0$ and $a_{hp} = 0$ for $p \in \{1 \ldots n\}$ (there are zeros in the $\mathbf{x}$ columns of row h). Then that row of the original problem is redundant, so we can delete the row and the basic y_i column from the optimal-form tableau of the artificial problem. This happened in the example above, when we deleted row 2 and column 12 of the artificial problem's optimal tableau.

2. The basic y_i column has its 1 in row h and some a_{hp} in the $\mathbf{x}$ part of the tableau is nonzero. Then we can pivot on that element to make column p basic and the y_i column nonbasic.

These complications are included in the flowchart on the next page, which summarizes the method of artificial variables. The abbreviation CF means canonical form.



The method of artificial variables is a way of manipulating the *constraints* of a linear program. Once the constraints are expressed in the form we want, with basis columns in the **x** part of the tableau, we can discard the artificial columns and objective and replace the constraints of the original problem by the reworked ones under the original objective.

2.9 Getting Standard Form

Recall from §2.1 that a linear program in standard form is a minimization with equality constraints and nonnegative variables. Many formulations, including those we considered in §1, lead to problems which *do not* fit that description but can be rewritten so that they do.

2.9.1 Inequality Constraints

The second constraint of the `brewery` problem, "don't use more black malt than you have," is formulated in §1.3.1 as this inequality.

$$1x_1 + 3x_2 + 1x_3 + 1x_4 \leq 50$$

The optimal production program $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$ uses

$$1{\times}5 \;+\; 3{\times}12\tfrac{1}{2} \;+\; 1{\times}0 \;+\; 1{\times}0 = 42\tfrac{1}{2}$$

pounds of black malt, so the constraint is satisfied as an *in*equality with $50 - 42\frac{1}{2} = 7\frac{1}{2}$ pounds of black malt left over. At $\mathbf{x}^\star$ this inequality is said to be **slack** or **inactive**; if it were satisfied as an equality it would be **tight** or **active**. We can rewrite the inequality as an equation by introducing a **slack variable** x_6 to represent the amount of black malt that is not used:

$$1x_1 + 3x_2 + 1x_3 + 1x_4 + 1x_6 = 50.$$

When we solved the standard-form `brewery` problem the optimal value of x_6 came out $7\frac{1}{2}$, as you can confirm by inspecting the optimal tableau `T3c` of §2.4.3.

Unused resources generate no revenue, so the objective function cost coefficient of a slack variable is zero. If we introduce additional slacks x_5 and x_7 to represent the unused amounts of pale malt and hops we get this reformulation of the `brewery` problem, which has equality constraints but is still a maximization and thus not yet in canonical form.

$$\begin{array}{lrcrcrcrcrcrcrcr}
\underset{\mathbf{x}\in\mathbb{R}^7}{\text{maximize}} & 90x_1 & + & 150x_2 & + & 60x_3 & + & 70x_4 & + & 0x_5 & + & 0x_6 & + & 0x_7 & & \\
\text{subject to} & 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & + & 1x_5 & + & 0x_6 & + & 0x_7 & = & 160 \\
 & 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & + & 0x_5 & + & 1x_6 & + & 0x_7 & = & 50 \\
 & 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & + & 0x_5 & + & 0x_6 & + & 1x_7 & = & 60 \\
 & & & & & & & & & & & & & \mathbf{x} & \geq & \mathbf{0}
\end{array}$$

Notice that the added columns for x_5, x_6, and x_7 are the columns of $\mathbf{I}_{3\times 3}$ with zero costs above them, so they constitute an **all-slack basis**. Here I have used the name x_{4+i} for the slack variable associated with inequality constraint i, but in the future I will often call it s_i to distinguish it from the variables that are not slacks.

To rewrite a greater-than-or-equal-to inequality as an equation by this approach we must first multiply through by -1 to change the sense of the inequality. Here is how we would rewrite the first constraint in our formulation of the `shift` problem as an equation.

$$\begin{aligned} x_1 + x_8 &\geq 3 \\ -x_1 - x_8 &\leq -3 \\ -x_1 - x_8 + s_1 &= -3 \end{aligned}$$

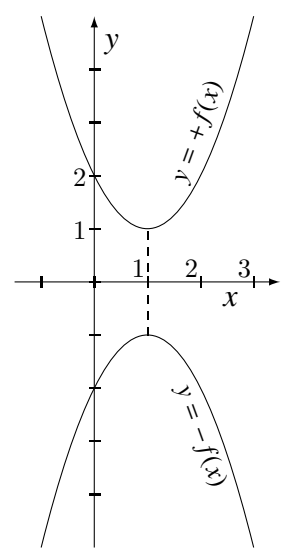
In order for the slack variable that we *add* to the left-hand side to be nonnegative the inequality must already be turned around, so it is important to do the multiplication through by -1 *first.*

2.9.2 Maximization Problems

If $f(x) = x^2 - 2x + 2$ then $y = +f(x)$, the quadratic graphed on top, has its minimum value of 1 at $x = 1$ while $y = -f(x)$, which is graphed on the bottom, has its maximum value of -1 at $x = 1$. It is true in general (and in particular when $f(x)$ is a linear function) that

$$\underset{\mathbf{x}}{\text{maximum}}\,[-f(\mathbf{x})] = -\underset{\mathbf{x}}{\text{minimum}}\,[+f(\mathbf{x})]$$

and these extreme values are attained at the same point $\mathbf{x}^\star$. Thus to maximize a given objective we need only minimize its negative. It is necessary to remember this sign change when reporting the optimal value of the original maximization, but it does not affect the optimal point. To convert the `brewery` problem from the maximization of §1.3.1 to the minimization of §2.1, I changed the sign of each per-keg revenue from positive to negative; minimizing the negative of the total revenue maximizes the total revenue.



$$\begin{array}{lrcrcrcrcrcrcrcr} \underset{\mathbf{x}\in\mathbb{R}^7}{\text{minimize}} & -90x_1 & - & 150x_2 & - & 60x_3 & - & 70x_4 & + & 0x_5 & + & 0x_6 & + & 0x_7 & & \\ \text{subject to} & 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & + & 1x_5 & + & 0x_6 & + & 0x_7 & = & 160 \\ & 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & + & 0x_5 & + & 1x_6 & + & 0x_7 & = & 50 \\ & 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & + & 0x_5 & + & 0x_6 & + & 1x_7 & = & 60 \\ & & & & & & & & & & & & & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

Now the problem is in standard form, and because it has a basis and $\mathbf{b} \geq \mathbf{0}$ it happens also to be in canonical form.

2.9.3 Free Variables

Some linear programs, such as our `bulb` problem, are naturally formulated in terms of variables that are unconstrained in algebraic sign. Here is a simpler example with a single free variable.

$$\begin{array}{lrcl} \underset{y\in\mathbb{R}^1}{\text{minimize}} & z = & y & \\ \text{subject to} & & y & \geq -10 \\ & & y & \text{free} \end{array}$$

The lowest value of y that is greater than or equal to -10 is obviously $y^\star = -10$. The logic of our simplex algorithm depends on the variables being nonnegative (but see [71, Myth 13]) so we cannot use it to solve the problem when it is stated like this. However, from §1.5.2,

> any real number y can be written as $y = u - w$, where $u \geq 0$ and $w \geq 0$.

Using this fact any free variable can be replaced by the difference between two nonnegative ones, so we could reformulate the above example like this.

$$\begin{array}{lrcrcr} \underset{u\in\mathbb{R}^1\ w\in\mathbb{R}^1}{\text{minimize}} & z = & u & -w & & \\ \text{subject to} & & u & -w & \geq & -10 \\ & & u & & \geq & 0 \\ & & & w & \geq & 0 \end{array}$$

This problem can be solved by inspection too. To minimize z we want u to be as low as possible and w to be as high as possible. Because u is nonnegative it can go no lower than 0, and if $u = 0$ then $-w \geq -10$ or $w \leq 10$. Thus $u^\star = 0$ and $w^\star = 10$.

Now, however, we can rewrite the functional inequality constraint as an equation in nonnegative variables and solve the problem by the simplex method. First we multiply through by -1 to reverse the inequality, and then we add a slack to get this standard form.

$$\begin{array}{lrcrcrcr} \underset{u\in\mathbb{R}^1\ w\in\mathbb{R}^1\ s\in\mathbb{R}^1}{\text{minimize}} & z = & u & -w & & & & \\ \text{subject to} & & -u & +w & +s & = & 10 \\ & & u & & & \geq & 0 \\ & & & w & & \geq & 0 \\ & & & & s & \geq & 0 \end{array}$$

The corresponding tableau is already in canonical form and only one phase-2 pivot is needed to obtain optimal form.

$$\begin{array}{c|ccc|} & u & w & s \\ \hline 0 & 1 & -1 & 0 \\ \hline 10 & -1 & 1 & 1 \\ \hline \end{array} \longrightarrow \begin{array}{c|ccc|} & u & w & s \\ \hline 10 & 0 & 0 & 1 \\ \hline 10 & -1 & 1 & 1 \\ \hline \end{array}$$

If we substitute $y_j = u_j - w_j$ to replace a free variable by the difference between nonnegative ones, it will turn out that in the simplex solution either u_j or w_j is zero and the other is $|y_j^\star|$. If there are r free variables and we replace each of them in this way, we end up with $2r$ nonnegative variables. But the fact about real scalars that is boxed on the previous page can be generalized to vectors as follows.

> Any vector $\mathbf{y}$ of r real numbers can be written as $\mathbf{y} = \mathbf{u} - w\mathbf{1}$, where $\mathbf{u} \geq \mathbf{0}$, $w \geq 0$, and $\mathbf{1}$ is a column vector of r 1's.

For example,

$$\mathbf{y} = \begin{bmatrix} 22 \\ -8 \\ -3 \end{bmatrix} = \begin{bmatrix} 30 \\ 0 \\ 5 \end{bmatrix} - \begin{bmatrix} 8 \\ 8 \\ 8 \end{bmatrix} = \begin{bmatrix} 30 \\ 0 \\ 5 \end{bmatrix} - 8 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{u} - w\mathbf{1}$$

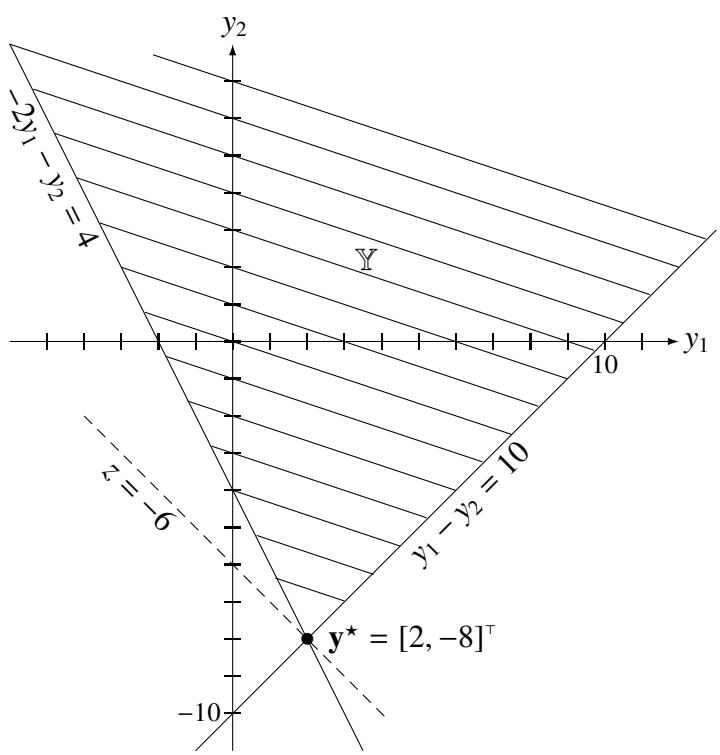
where $r = 3$,

$$w = \max_{j \in \{1 \ldots r\}} \Big(|y_j| \;\big|\; y_j < 0 \Big) = |y_2| = 8,$$

and that (second) element of $\mathbf{u}$ is zero. Using this idea we can replace the r free variables in $\mathbf{y}$ by r nonnegative variables in $\mathbf{u}$ and the single nonnegative scalar w.

In solving a linear program with free variables, we need not (and usually cannot) figure out $\mathbf{u}^\star$ and $w^\star$ ahead of time. But if we substitute $y_j = u_j - w$ for each of the free variables and require that $\mathbf{u} \geq \mathbf{0}$ and $w \geq 0$, then the optimization will discover $\mathbf{u}^\star$ and $w^\star$ having the properties we observed in the example above. To show how the idea works we will use it to get standard form for the problem below, which has two free variables and the graphical solution on the right.

$$\begin{array}{lrcr} \underset{\mathbf{y} \in \mathbb{R}^2}{\text{minimize}} & z = y_1 + y_2 & & \\ \text{subject to} & -2y_1 - y_2 & \leq & 4 \\ & y_1 - y_2 & \leq & 10 \\ & \mathbf{y} & & \textbf{free} \end{array}$$



Substituting $y_1 = u_1 - w$ and $y_2 = u_2 - w$ we get this problem with nonnegative variables.

$$\begin{array}{lrcl} \underset{\mathbf{u}\in\mathbb{R}^2\, w\in\mathbb{R}^1}{\text{minimize}} & (u_1 - w) + (u_2 - w) & & \\ \text{subject to} & -2(u_1 - w) - (u_2 - w) & \leq & 4 \\ & (u_1 - w) - (u_2 - w) & \leq & 10 \\ & \mathbf{u} & \geq & \mathbf{0} \\ & w & \geq & 0 \end{array}$$

```
> This is PIVOT, Unix version 4.2
> For a list of commands, enter HELP.
>
< tableau 3 6
< names u1 u2 w s1 s2;
< insert
T( 1, 1)... = 0 1 1 -2 0 0
T( 2, 1)... = 4 -2 -1 3 1 0
T( 3, 1)... = 10 1 -1 0 0 1

      u1  u2  w   s1  s2
  0.  1.  1. -2.  0.  0.
  4. -2. -1.  3.  1.  0.
 10.  1. -1.  0.  0.  1.

< solve

      u1  u2  w   s1         s2
  6.  0.  0.  0.  0.6666667  0.3333333
 10.  1. -1.  0.  0.0000000  1.0000000
  8.  0. -1.  1.  0.3333333  0.6666667

< quit
> STOP
```

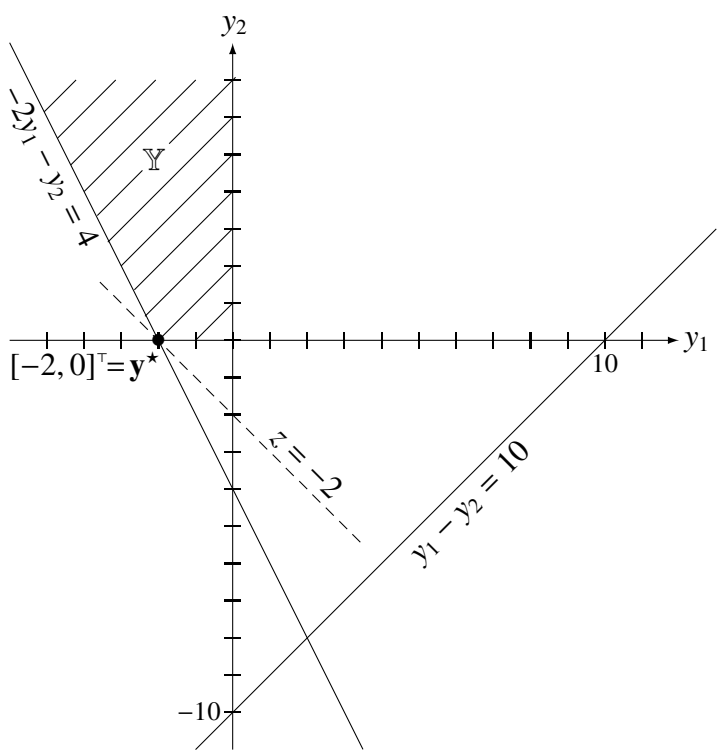
Simplifying and adding slacks yields canonical form.

$$\begin{array}{lrcl} \underset{\mathbf{u}\in\mathbb{R}^2\, w\in\mathbb{R}^1}{\text{minimize}} & u_1 + u_2 - 2w & & \\ \text{subject to} & -2u_1 - u_2 + 3w + s_1 & = & 4 \\ & u_1 - u_2 \qquad + s_2 & = & 10 \\ & \mathbf{u} & \geq & \mathbf{0} \\ & w & \geq & 0 \\ & \mathbf{s} & \geq & \mathbf{0} \end{array}$$

The `pivot` session finds $\mathbf{u}^\star = [10, 0]^\top$ and $w^\star = 8$. Then $\mathbf{u}^\star - w^\star\mathbf{1} = [10, 0]^\top - [8, 8]^\top = [2, -8]^\top = \mathbf{y}^\star$✓

2.9.4 Nonpositive Variables

Some linear programs are naturally stated using variables that are less than or equal to zero. For example, in an engineering problem a design variable might be a fraction less than or equal to 1 so an optimization variable that is its logarithm is nonpositive. In the example below, solved graphically to the right, y_2 is nonnegative but y_1 is nonpositive. The functional constraints are the same as those in the free-variables example.



$$\begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^2}{\text{minimize}} & z = y_1 + y_2 & & \\ \text{subject to} & -2y_1 - y_2 & \leq & 4 \\ & y_1 - y_2 & \leq & 10 \\ & y_1 & \leq & 0 \\ & y_2 & \geq & 0 \end{array}$$

To reformulate the problem so that all of the variables are nonnegative we can let $y_1 = -x_1$; then adding slacks we get standard form.

$$\begin{array}{lrrrcr}
\underset{x_1\in\mathbb{R}^1\, y_2\in\mathbb{R}^1\, \mathbf{s}\in\mathbb{R}^2}{\text{minimize}} & z = -x_1 & +y_2 & & & \\
\text{subject to} & 2x_1 & -y_2 & +s_1 & = & 4 \\
 & -x_1 & -y_2 & +s_2 & = & 10 \\
 & x_1 & & & \geq & 0 \\
 & & y_2 & & \geq & 0 \\
 & & & \mathbf{s} & \geq & \mathbf{0}
\end{array}$$

	x_1	y_2	s_1	s_2
0	-1	1	0	0
4	2	-1	1	0
10	-1	-1	0	1

$\downarrow$

	x_1	y_1	s_1	s_2
2	0	$\frac{1}{2}$	$\frac{1}{2}$	0
2	1	$-\frac{1}{2}$	$\frac{1}{2}$	0
12	0	$-\frac{3}{2}$	$\frac{1}{2}$	1

One pivot finds $x_1^\star = 2$ and $y_2^\star = 0$, so $\mathbf{y}^\star = [-x_1^\star, y_2^\star]^\top = [-2, 0]^\top$✓

2.9.5 Variables Bounded Away from Zero

A problem in standard form has nonnegative variables, but many linear programs (such as `twoexams` and `chairs`) include functional constraints that impose additional simple bounds. In this problem x_1 and x_2 are both bounded away from zero.

$$\begin{array}{lrcrrcr}
\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & z & = & x_1 & -x_2 & & \\
\text{subject to} & & & x_1 & & \geq & 2 \\
 & & & & x_2 & \leq & -3
\end{array}$$

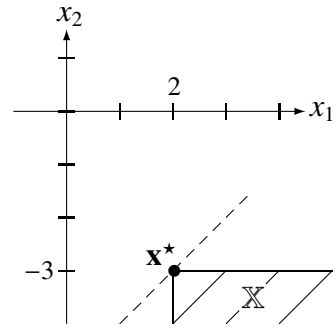
To restate the problem with all of the variables nonnegative we could let $y_2 = -x_2$ and rewrite the second constraint as $-y_2 \leq -3$. Then, multiplying the first constraint through by -1 to reverse the inequality and adding slacks, we get this standard form

$$\begin{array}{lrcrrrrcr}
\underset{x_1\in\mathbb{R}^1\, y_2\in\mathbb{R}^1\, \mathbf{s}\in\mathbb{R}^2}{\text{minimize}} & z & = & x_1 & +y_2 & & & & \\
\text{subject to} & & & -x_1 & & +s_1 & & = & -2 \\
 & & & & -y_2 & & +s_2 & = & -3 \\
 & & & x_1 & & & & \geq & 0 \\
 & & & & y_2 & & & \geq & 0 \\
 & & & & & & \mathbf{s} & \geq & \mathbf{0}
\end{array}$$

Then we can form a tableau and solve the problem; the `pivot` session on the right finds $x_1^\star = 2$ and $y_2^\star = 3$, so $\mathbf{x}^\star = [2, -3]^\top$.

```
> This is PIVOT, Unix version 4.2
> For a list of commands, enter HELP.
>
< t 3 5
< names x1 y2 s1 s2;
< in
T( 1, 1)... = 0 1 1 0 0
T( 2, 1)... = -2 -1 0 1 0
T( 3, 1)... = -3 0 -1 0 1

     x1  y2  s1  s2
 0.  1.  1.  0.  0.
-2. -1.  0.  1.  0.
-3.  0. -1.  0.  1.

< solve

     x1  y2  s1  s2
-5.  0.  0.  1.  1.
 2.  1.  0. -1.  0.
 3.  0.  1.  0. -1.

< quit
> STOP
```


The top picture shows some of the feasible set $\mathbb{X}$ for this problem, along with the objective function contour through its optimal point.



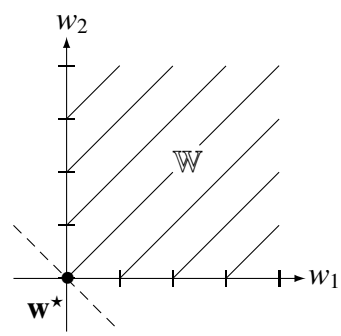
A simpler reformulation is based on these observations.

$$\begin{array}{rcrcrcrcl} x_1 & \geq & 2 & \Rightarrow & w_1 & = & x_1 - 2 & \geq & 0 \\ x_2 & \leq & -3 & \Rightarrow & w_2 & = & -x_2 - 3 & \geq & 0 \end{array}$$

In terms of $\mathbf{w}$ our problem becomes

$$\begin{array}{rl} \underset{\mathbf{w}\in\mathbb{R}^2}{\text{minimize}} & z = (w_1 + 2) - (-w_2 - 3) = w_1 + w_2 + 5 \\ \text{subject to} & \mathbf{w} \geq \mathbf{0} \end{array}$$

which by inspection has the optimal point $\mathbf{w}^\star = [0, 0]^\top$. The bottom picture shows the objective function contour through the optimal point along with some of the feasible set $\mathbb{W}$, which is now the whole first quadrant; the reformulation has eliminated the functional constraints entirely. From $\mathbf{w}^\star$ we find $\mathbf{x}^\star = [2, -3]^\top$.



2.9.6 Summary

Here are some prototypes for the reformulations discussed above.

§	not standard form	substitute	and require
2.9.1	$x_1 + x_2 + 2x_3 \leq 10$ $2x_1 - x_2 - x_3 \geq 8$	$x_1 + x_2 + 2x_3 + s_1 = 10$ $-2x_1 + x_2 + x_3 + s_2 = -8$	$s_1 \geq 0$ $s_2 \geq 0$
2.9.2	$\max\ -3x_1 + x_2 - 4x_3$	$\min\ 3x_1 - x_2 + 4x_3$	sign change for optimal value
2.9.3	x_1 free x_1, x_2 free	$x_1 = u - w$ $x_1 = u_1 - w,\ x_2 = u_2 - w$	$u \geq 0,\ \ w \geq 0$ $u_1 \geq 0,\ \ u_2 \geq 0,\ \ w \geq 0$
2.9.4	$x_1 \leq 0$	$y_1 = -x_1$	$y_1 \geq 0$
2.9.5	$x_1 \geq 2$ $x_2 \leq -3$	$w_1 = x_1 - 2$ $w_2 = -x_2 - 3$	$w_1 \geq 0$ $w_2 \geq 0$

2.10 Exercises

2.10.1 [E] List two numerical methods for solving linear programs, and tell where they are discussed in this book.

2.10.2 [E] What are the distinguishing characteristics of a linear program that is in *standard form*?

2.10.3 [E] What is a *functional constraint*? Describe the constraints a mathematical program might have that are *not* functional constraints.

2.10.4 [E] The definition of standard form given in §2.1 involves quantities named d, $\mathbf{c}$, $\mathbf{x}$, $\mathbf{A}$, and $\mathbf{b}$. (a) What English phrase is used in this book to refer to each of these quantities? (b) What variable names are almost always used in this book to denote the number of functional constraints, the number of variables, the index of a particular functional constraint, and the index of a particular variable? (c) In terms of those numbers, what are the dimensions of d, $\mathbf{c}$, $\mathbf{x}$, $\mathbf{A}$, and $\mathbf{b}$? (d) When an objective function in this book is named z, is it to be maximized or minimized? (e) Most vectors in this book are denoted by lower-case bold letters. Are they column vectors, or row vectors? If $\mathbf{M}$ is a matrix, what does M_i denote?

2.10.5 [H] Why are the sign constraints in our standard form for a linear program nonnegativities, rather than requiring $\mathbf{x}$ to be strictly positive?

2.10.6 [E] The simplex tableau that we use to represent a standard form linear program is defined in §2.2. (a) Describe its structure and contents. (b) In terms of the number of variables n and the number of constraints m, how big is a simplex tableau? (c) Where in a simplex tableau are the constraints $\mathbf{x} \geq \mathbf{0}$?

2.10.7 [H] The first two rows of a tableau look like this.

	x_1	x_2	x_3	x_4
1	2	3	4	5
6	7	8	9	0

(a) What equation is represented by the first row? (b) What equation is represented by the second row?

2.10.8 [H] How can you tell if two simplex tableaus are *equivalent*?

2.10.9 [E] What is the fundamental operation that we use in solving linear programs by the simplex method?

2.10.10 [E] There are three steps to performing a pivot. What are they?

2.10.11 [E] Why is it necessary for a pivot element to be nonzero?

2.10.12 [P] The following tableau [3, p47] can be transformed into an equivalent tableau by substitution or by pivoting.

$$\mathbf{T}_0 = \begin{array}{|r|rrrr|} \hline & x_1 & x_2 & x_3 & x_4 \\ \hline 0 & -2 & -1 & 0 & 0 \\ \hline 5 & 2 & -1 & 0 & 1 \\ 10 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(a) Solve the first constraint equation to obtain an expression for x_1 in terms of the other variables, and substitute to eliminate that variable from the other rows. (b) By hand, perform

a pivot that produces the same result. (c) Perform the pivot by using the `pivot` program. (d) Perform the pivot by using the `pivot.m` routine.

2.10.13 [H] Performing a "pivot" in the objective row of a simplex tableau yields a new tableau that is not equivalent to the original one. To see that this is true, consider the following tableau.

$$\mathbf{T}_0 = \begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline 0 & -2 & -1 & 0 & 0 \\ \hline 5 & 2 & -1 & 0 & 1 \\ 10 & 1 & 1 & 1 & 0 \end{array}$$

(a) What linear program does $\mathbf{T}_0$ represent? (b) Perform the arithmetic of a pivot using as the pivot element the -2 in the objective row, and show that the resulting tableau describes a linear program that is *not* equivalent to the one we began with. (c) In the subproblem technique of §2.8.1 we sometimes pivot in the objective row of a subproblem. How can the resulting tableau still describe the original linear program?

2.10.14 [E] What is a *reduced cost*?

2.10.15 [E] What are the distinguishing characteristics of a tableau that is in canonical form?

2.10.16 [E] Which columns of a canonical-form tableau are the basis columns? Which variables are basic and which are nonbasic?

2.10.17 [E] Explain how to read off the basic feasible solution associated with a canonical form tableau.

2.10.18 [H] The following tableau is in canonical form.

$$\mathbf{T}_0 = \begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline 0 & -2 & -1 & 0 & 0 \\ \hline 5 & 2 & -1 & 0 & 1 \\ 10 & 1 & 1 & 1 & 0 \end{array}$$

(a) What is its associated basic feasible solution? (b) What is the tableau's basic sequence S? (c) If a pivot is performed on the 2, which variable will enter the basis and which will leave?

2.10.19 [H] In §2.2 the quantity shown in the upper left corner of the simplex tableau is $-d$, the negative of the constant d in our standard form. In §2.4.1 the quantity shown in the upper left corner of the canonical-form tableau $\mathbf{T}_2$ is $-z$, which is $2290\frac{10}{11}$. Explain how both of these pictures can be correct.

2.10.20 [H] In a canonical-form tableau, what is $\mathbf{c}^\top\bar{\mathbf{x}}$ if $\bar{\mathbf{x}}$ is the basic feasible solution associated with the tableau? How can we minimize $\mathbf{c}^\top\mathbf{x}$ by moving away from the basic feasible solution? Explain.

2.10.21 [H] The linear program on the left has the tableau on the right.

$$\begin{array}{lrcl} \text{minimize} & -2x_1 + x_2 - x_3 & & \\ \text{subject to} & x_1 - 2x_2 + 2x_3 + x_4 & = & -5 \\ & x_2 - x_3 + x_5 & = & 5 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad \mathbf{T}_0 = \begin{array}{c|ccccc} & x_1 & x_2 & x_3 & x_4 & x_5 \\ \hline 0 & -2 & 1 & -1 & 0 & 0 \\ \hline -5 & 1 & -2 & 2 & 1 & 0 \\ 5 & 0 & 1 & -1 & 0 & 1 \end{array}$$

(a) Pivot on the a_{22} element of $\mathbf{T}_0$ to produce tableau $\mathbf{T}_1$, in which $S = (x_4, x_2)$ and $\mathbf{b}$ is nonnegative. Confirm that the basic feasible solution corresponding to $\mathbf{T}_1$ satisfies the constraints of the original problem. Is $\mathbf{T}_1$ equivalent to $\mathbf{T}_0$? (b) Perform the following sequence of row operations on $\mathbf{T}_0$ to produce tableau $\mathbf{T}_2$:

$$\begin{array}{lcl} r_2 & \leftarrow & r_2 + r_3 \\ r_2 & \leftarrow & r_2 + r_1 \\ r_1 & \leftarrow & r_1 - r_3 \end{array}$$

Confirm that $\mathbf{T}_2$ also has $S = (x_4, x_2)$ and $\mathbf{b} \geq \mathbf{0}$. Does its basic feasible solution satisfy the constraints of the original problem? Is $\mathbf{T}_2$ equivalent to $\mathbf{T}_0$? (c) Every pivot is a sequence of row operations. Is every sequence of row operations a pivot? (d) Why can't the objective row of a simplex tableau be treated like a constraint row?

2.10.22 [H] In the `pivot.m` routine of §2.4.2, `ip` and `jp` are the indices of the pivot row and column in `T`. To what indices h and p in the constraint coefficient matrix $\mathbf{A}$ do these correspond?

2.10.23 [E] In the `pivot.m` routine of §2.4.2, why are the elements in the pivot row and column computed separately from the other elements in the result tableau?

2.10.24 [P] In the `pivot.m` routine of §2.4.2, why is the pivot row divided by the pivot element only *after* the tableau elements that are not in the pivot row or column have been updated? Hint: what happens if the routine is invoked with the same matrix for `T` and `Tnew`?

2.10.25 [P] The `pivot.m` routine of §2.4.2 constructs the entering basis column by assigning the value 1 to its element in the pivot row and the value 0 to its elements not in the pivot row, but it computes the other elements of the new tableau by performing floating-point arithmetic. Revise the routine to construct all of the basis columns that are in the new tableau (of which there might be fewer than m) by assignment rather than by doing arithmetic.

2.10.26 [E] This Chapter introduces two different ways to describe the basic sequence of a tableau. The basic sequence of tableau $\mathbf{T}_2$ is given in one place as $S = (x_3, x_6, x_2)$ but in `pivot.m` it is `S=(0,4,2,0,0,3,0)`. Explain how to get each characterization from the other. When is each most useful?

2.10.27 [H] Suppose that we pivot in column p of a canonical-form tableau having $c_p < 0$. (a) What happens if the pivot element a_{hp} is negative? (b) What happens if h, the pivot row in $\mathbf{A}$, is chosen so that the ratio b_h/a_{hp} is *not* the minimum ratio in column p?

2.10.28 [H] Consider the following linear program.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & \leq & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

Show [3, Exercise 3.22] that if $\mathbf{c} \geq \mathbf{0}$ and $\mathbf{b} \geq \mathbf{0}$ then $\mathbf{x}^\star = \mathbf{0}$.

2.10.29 [H] In the t–analysis of §2.4.4, increasing t from 0 to 5 moves $\mathbf{x}(t)$ from the basic feasible solution corresponding to `T2` to the basic feasible solution corresponding to `T3c`. (a) Where does the value $t = 5$ come from? (b) What is $\mathbf{x}(t)$ when $t = 2\frac{1}{2}$? (c) Is $\mathbf{x}(2\frac{1}{2})$ feasible? (d) Is $\mathbf{x}(2\frac{1}{2})$ a basic solution? Explain.

2.10.30 [E] What is the *simplex pivot rule*? Why does the simplex algorithm use it in pivoting a canonical-form tableau toward optimality?

2.10.31 [H] If we pivot by the minimum-ratio rule in a tableau that is not in canonical form, does $\mathbf{x}$ move toward $\mathbf{x}^\star$? Explain.

2.10.32 [E] What final forms can the simplex algorithm produce?

2.10.33 [H] In §1 we formulated linear programming models for several practical applications. Unfortunately, not every linear program has an optimal point. (a) Describe three ways in which a linear program can fail to have an optimal point. (b) When a linear program is **defective** in one of these ways, does it mean that there is something wrong with the formulation? Does it mean that there is something wrong with the underlying application problem? Explain.

2.10.34 [H] The pictures in §2.5 show what a tableau looks like in each of the possible final forms. A linear program that is feasible and not unbounded typically has many canonical forms, and it is only by solving the problem that we find an optimal one. (a) Can a linear program that is unbounded have canonical form tableaus that do not reveal its unboundedness? If not, explain why not; if so, provide an example. (b) Can a linear program that is infeasible have tableaus that do not reveal its infeasibility? If not, explain why not; if so, provide an example.

2.10.35 [E] What does an optimal form tableau look like? Why can't its objective value be further reduced?

2.10.36 [H] What is indicated by a tableau that is not in canonical form but has $\mathbf{c} \geq \mathbf{0}$?

2.10.37 [H] In the `brewery` problem discussed in §2.1, $b_1 = 160$. (a) Change its value to 150 and show that the resulting linear program has multiple optimal solutions. (b) What vectors $\mathbf{x}$ are optimal for the revised problem?

2.10.38 [E] What does an unbounded form tableau look like? Why can its objective value be reduced without limit?

2.10.39 [H] Is the following tableau [3, p48-49] in unbounded form?

−9	0	0	−2	−1	0
3	0	0	−1	2	1
−1	1	0	0	1	0
5	0	1	−4	1	0

If so, explain why; if not, obtain a final form that is not unbounded.

2.10.40 [E] What does an infeasible form tableau look like?

2.10.41 [E] Can pivoting in a canonical-form tableau ever yield infeasible form? Explain.

2.10.42 [H] Suppose that each of the constraint rows in a tableau has the property that some vector $\mathbf{x}$ satisfies the equation it represents. Is it necessarily true that the linear program is feasible? If so, explain why; if not, provide a counterexample.

2.10.43 [H] Consider the following tableau.

	x_1	x_2	x_3	x_4	x_5
9	0	$-a$	c	0	0
a	1	$-a$	1	0	0
2	0	b	−1	1	0
4	0	−1	d	0	1

Give general conditions, if any, on a, b, c, d, and e (not just particular values) so that the tableau is in (a) optimal form; (b) unbounded form; (c) infeasible form.

2.10.44 [E] What does it mean to "solve" a linear program? Describe the three phases of the solution process.

2.10.45 [E] In studying the simplex algorithm, why might it be helpful to have a utility program capable of manipulating tableaus? Describe three different manipulations of a simplex tableau that can be performed by the `pivot` program discussed in §2.7.

2.10.46 [E] Where in this book are the `pivot` program's commands explained in detail? Can any of its commands be abbreviated?

2.10.47 [E] To put a simplex tableau into canonical form it is necessary to transform it so that it has basis columns and to make its $\mathbf{b}$ part nonnegative. In what order are these tasks performed (a) by the subproblem technique; (b) by the method of artificial variables?

2.10.48 [E] If in a tableau that does not have a basis we perform pivots to obtain canonical form, can some sequence of pivots be performed to restore the original tableau? Explain.

2.10.49 [E] The subproblem technique begins by pivoting-in a basis. (a) Explain how it does that. (b) If $\mathbf{b} \geq \mathbf{0}$ at the start of this process, is $\mathbf{b}$ necessarily nonnegative at the end?

2.10.50 [E] After the subproblem technique has pivoted-in a basis, it gets $\mathbf{b}$ nonnegative. Explain how it does that.

2.10.51 [E] In forming a subproblem it is necessary to include all of the rows having $b_h \geq 0$. (a) Why is that? (b) What if, at the beginning of the process, there are no such rows? (c) Why is it necessary to pivot the entire tableau when solving a subproblem? (d) How can a subproblem solution be completed if the subproblem is unbounded in column p?

2.10.52 [H] Does pivoting in the objective row of an unbounded subproblem ever leave the b_h that is its upper-left corner negative? What does pivoting in the objective row of an unbounded subproblem do to the b_h of its constraint rows?

2.10.53 [E] Solving a subproblem makes its upper-left entry go up. (a) Does that entry always become nonnegative? Explain. (b) Is it necessary to solve a subproblem all the way to optimality? Explain. (c) Can negative b_h that are not in a subproblem ever become nonnegative in the process of solving the subproblem?

2.10.54 [E] Is a subproblem always in canonical form? If so, explain why; if not, present a counterexample.

2.10.55 [E] If a linear program has redundant constraints, at what stage of the subproblem technique is that fact discovered?

2.10.56 [E] If a linear program is infeasible, at what stage in the subproblem technique is that discovered?

2.10.57 [H] The method of artificial variables is flowcharted at the end of §2.8.2. Draw a similar flowchart for the subproblem technique, including enough detail to show how it detects redundant and inconsistent constraints.

2.10.58 [H] Often in performing a step of the subproblem technique several possible pivot positions can be used. This latitude leads some students to assume (incorrectly) that *any* pivot producing a desirable result constitutes using this technique. (a) Present an example illustrating how it is possible for the technique to require a choice between possible pivot positions. (b) In your example, identify a pivot position that does *not* conform to the algorithm. (c) Explain why "I feel lucky" pivoting is not a practical strategy in general.

2.10.59 [H] In §2.8.1 we found an initial canonical form for the `sf1` problem. (a) Pivot that tableau to optimality. (b) Change the sign of the objective and solve the revised problem.

2.10.60 [E] How does the method of artificial variables make $\mathbf{b}$ nonnegative? How does it supply basis columns to the resulting tableau?

2.10.61 [E] Describe the form that a linear program must be in if it is to serve as the original problem in the method of artificial variables.

2.10.62 [E] If a linear program has redundant constraints, when in the method of artificial variables is that fact discovered?

2.10.63 [E] If a linear program is infeasible, when in the method of artificial variables is that discovered?

2.10.64 [H] If the artificial variables are all nonbasic in the solution of an artificial problem, how can we construct an initial canonical form tableau for the original problem?

2.10.65 [H] The method of artificial variables solves an artificial problem. (a) Describe this problem, identifying the artificial variables. (b) Is every artificial problem feasible? If so, write down a feasible solution. If not, present a counterexample. (c) What is the algebraic sign of an artificial objective? (d) If the optimal value of an artificial problem is zero, what can we deduce about the corresponding original problem?

2.10.66 [H] If an artificial variable remains basic in the solution of an artificial problem and the corresponding constraint is not redundant, how can we move that basis column into the $\mathbf{x}$ part of the tableau?

2.10.67 [H] In §2.8.2 we used the method of artificial variables to find an initial canonical form for the `sf1` problem. The solution that we found to the artificial problem, with basic sequence $S = (x_7, x_3, x_6, x_2)$, is not unique. (a) Pivot by the minimum-ratio rule in the x_1 column of the optimal tableau for the artificial problem. Does this change the objective value? Explain. (b) What canonical form for the original problem do we obtain from this optimal solution to the artificial problem?

2.10.68 [H] This tableau already has $\mathbf{b} \geq \mathbf{0}$ and one basis column.

	x_1	x_2	x_3	x_4	x_5
0	1	0	2	−1	4
6	1	0	−1	−3	1
5	−1	1	0	3	−3

(a) Use the method of artificial variables with one artificial variable to find an initial canonical form. (b) Pivot the canonical-form tableau to optimality.

2.10.69 [H] Can *every* linear program be put into standard form? If yes, explain why; if no, give a counterexample.

2.10.70 [E] In a resource allocation problem, some resource might not be used up by a given production program. (a) What do we call a variable that is introduced to represent the amount of the resource that is not used up? (b) What objective cost coefficient is associated with such a variable?

2.10.71 [E] How much slack is there in an active inequality constraint?

2.10.72 [H] Consider the following linear program.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -x_1 + x_2 & & \\ \text{subject to} & x_2 & \geq & \frac{1}{2}x_1 - \frac{1}{2} \\ & x_1 + x_2 & \leq & 4 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

(a) Reformulate the problem into standard form, using y_j for the variables in the standard-form problem. (b) Show that the standard-form problem is equivalent to the original problem in the sense that if $\hat{\mathbf{x}}$ is a feasible point for the original problem and its optimal value is $\hat{z}$ then there is a feasible point $\hat{\mathbf{y}}$ for the standard-form problem that yields the objective value $\hat{z}$. Explain how to construct $\hat{\mathbf{x}}$ from $\hat{\mathbf{y}}$. (c) If any two linear programs are equivalent in this sense and one has an optimal value of $z^\star$, why must it be true that the other has an optimal value of $z^\star$? (d) If any two linear programs are equivalent in this sense and one is feasible but unbounded, why must the other also be feasible but unbounded? (e) If any two linear programs are equivalent in this sense and one is infeasible, why must the other also be infeasible?

2.10.73 [P] Consider the following linear program.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & 2x_1 + x_2 & & \\ \text{subject to} & x_1 - x_2 & \leq & -1 \\ & x_1 - x_2 & \geq & +1 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

(a) Reformulate the problem to have equality constraints. (b) Construct an original problem for the method of artificial variables. (c) Construct an artificial problem, and pivot on the appended identity-column 1's to zero those costs. The resulting canonical-form tableau should be in optimal form with both artificial variables still basic. (d) Pivot in the $\mathbf{x}$ part of the tableau to move the artificial basis columns there, or explain why that cannot be done.

2.10.74 [H] Reformulate this linear program into standard form, and solve it.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & -x_1 + x_2 & & \\ \text{subject to} & \frac{1}{2}x_1 - x_2 & \leq & \frac{1}{2} \\ & -x_1 - x_2 & \geq & -4 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

2.10.75 [H] Consider the following linear program, which is similar to Exercise 3.4 of [3].

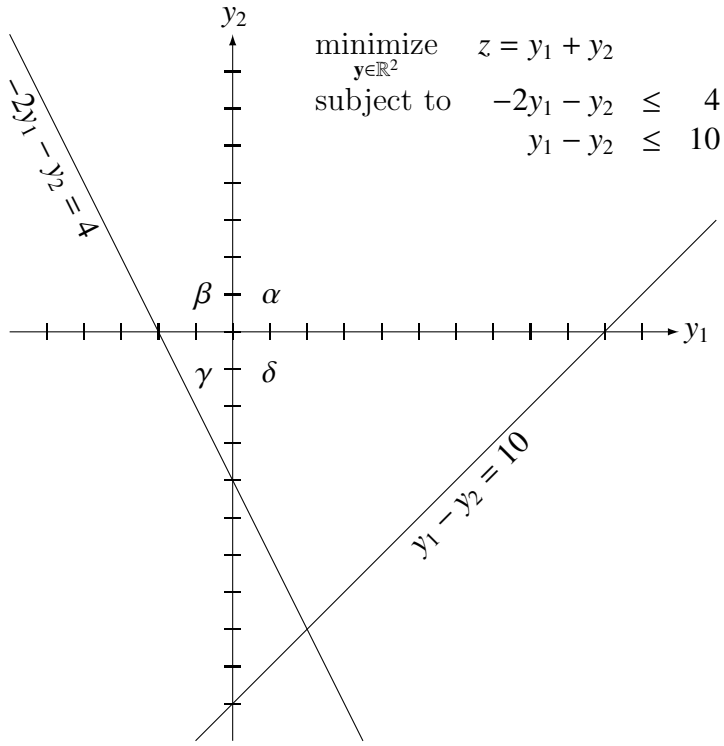
$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^6}{\text{maximize}} & 2x_1 + 6x_2 - 1x_3 + 5x_4 - 4x_5 + 3x_6 & & \\ \text{subject to} & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 & = & 1 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

(a) Reformulate the problem into standard form and construct a simplex tableau that represents it. (b) Perform a single pivot to obtain optimal form. (c) Give a rule for writing down the solution to any linear program of the form

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{maximize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \sum_{j=1}^n x_j & = & 1 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

Such a rule is called a semi-analytic result (see §25.7.4).

2.10.76 [H] Examples in §2.9.3 and §2.9.4 have the same objective and functional constraints but impose different requirements on the signs of the variables. For each set of sign requirements given below, say which regions of $\mathbb{R}^2$ marked α, β, γ, and δ in the picture on the right are included in the feasible set of the problem, and give the coordinates of the resulting optimal point.



$$\begin{array}{lrcr} \underset{\mathbf{y}\in\mathbb{R}^2}{\text{minimize}} & z = y_1 + y_2 & & \\ \text{subject to} & -2y_1 - y_2 & \leq & 4 \\ & y_1 - y_2 & \leq & 10 \end{array}$$

(a) $y_1 \geq 0,\ y_2 \geq 0$;
(b) $y_1 \geq 0,\ y_2$ free;
(c) y_1 free, $y_2 \geq 0$;
(d) y_1 free, y_2 free;
(e) $y_1 \leq 0,\ y_2$ free;
(f) y_1 free, $y_2 \leq 0$;
(g) $y_1 \leq 0,\ y_2 \leq 0$.

2.10.77 [H] For each set of sign requirements in Exercise 2.10.76, reformulate the problem into standard form and solve it to confirm the optimal points that you predicted.

2.10.78 [H] Solve the following linear program.

$$\begin{array}{lrcr} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & z = -y_1 & & \\ \text{subject to} & \frac{1}{2}y_1 - y_2 & \leq & -\frac{3}{2} \\ & \frac{1}{2}y_1 + y_2 & \leq & \frac{1}{2} \\ & y_1,\ \ y_2 & & \text{free} \end{array}$$

2.10.79 [H] Reformulate the following problem to eliminate the functional constraints that bound the variables away from zero. Solve it graphically and by using the simplex method.

$$\begin{array}{lrcr} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & x_1 - x_2 & & \\ \text{subject to} & x_1 & \leq & -1 \\ & x_2 & \geq & 1 \\ & x_1 + x_2 & \geq & 1 \end{array}$$

2.10.80 [H] In the first example of §2.9.3, $u = 1$ and $w = 11$ also solves the linear program. Why did the simplex algorithm find $u = 0$ and $w = 10$, making one of the nonnegative variables zero?

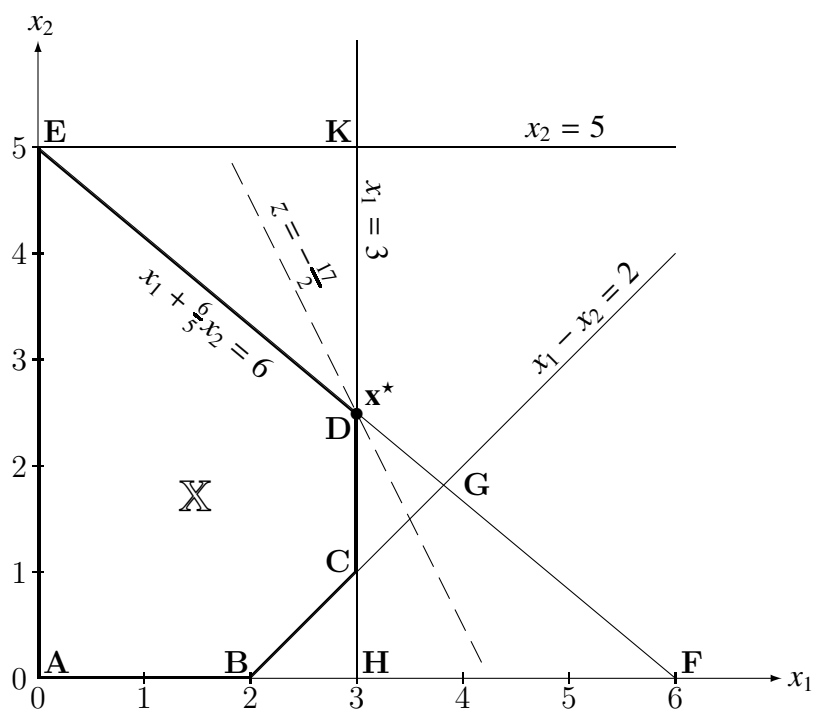
3

Geometry of the Simplex Algorithm

In §2 you learned how to solve a linear program by pivoting in a tableau according to the simplex algorithm. A problem having two or three variables can also be solved graphically by following the procedure described in §1.2, as we did for the `twoexams`, `paint`, `pumps`, `bulb`, and oil refinery problems. This Chapter is about the many connections between the simplex and graphical solutions. Most practical problems have many variables so they cannot be solved graphically, but valuable insights about linear programming in general can be gained from the study of low-dimensional examples.

3.1 A Graphical Solution in Detail

In the `graph` problem (see §28.5.12) given below the inequality constraints of the algebraic formulation on the right are graphed on the left along with the optimal objective contour. The feasible set $\mathbb{X}$ is outlined with thick lines, and the optimal point is marked $\mathbf{x}^\star$.



$$
\begin{array}{lrcl}
\text{minimize} & -2x_1 - x_2 & = & z \\
\text{subject to} & x_1 + \frac{6}{5}x_2 & \leq & 6 \\
 & x_1 - x_2 & \leq & 2 \\
 & x_1 & \leq & 3 \\
 & x_2 & \leq & 5 \\
 & x_1 & \geq & 0 \\
 & x_2 & \geq & 0
\end{array}
$$

An inequality constraint divides $\mathbb{R}^n$ into two **halfspaces**. In the graph above, the constraint $x_1 + \frac{6}{5}x_2 \leq 6$ has the associated halfspaces

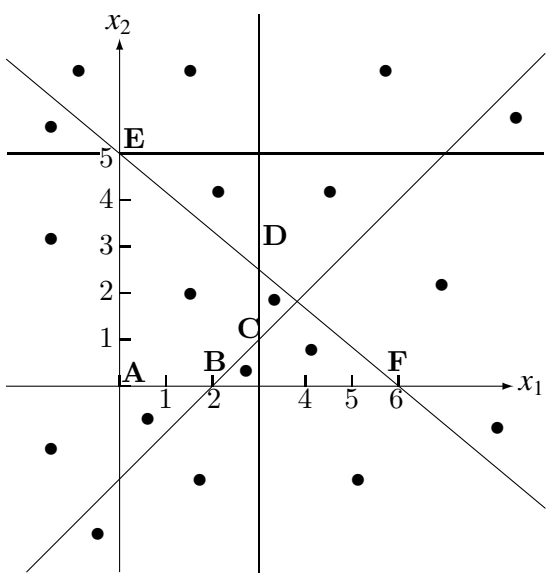
$$\underbrace{\left\{\mathbf{x}\,\middle|\,x_1 + \tfrac{6}{5}x_2 \le 6\right\}}_{\text{feasible}} \quad \cup \quad \underbrace{\left\{\mathbf{x}\,\middle|\,x_1 + \tfrac{6}{5}x_2 > 6\right\}}_{\text{infeasible}} \quad = \quad \mathbb{R}^2$$

The set of points where a $\le$ or $\ge$ constraint is satisfied with equality is called the constraint's **hyperplane**. In our example, $x_1 + \frac{6}{5}x_2 \le 6$ has the associated hyperplane

$$\left\{\mathbf{x}\,\middle|\,x_1 + \tfrac{6}{5}x_2 = 6\right\}$$

which belongs to the constraint's feasible halfspace. To represent an inequality constraint we plot its hyperplane. The constraint is satisfied on the line and on one side (the feasible halfspace) and violated on the other side of the line (the infeasible halfspace).

The constraint hyperplanes partition $\mathbb{R}^n$ into disjoint regions. In our example we see 19 distinct "windowpanes," each of which is marked with a dot • in the graph below.



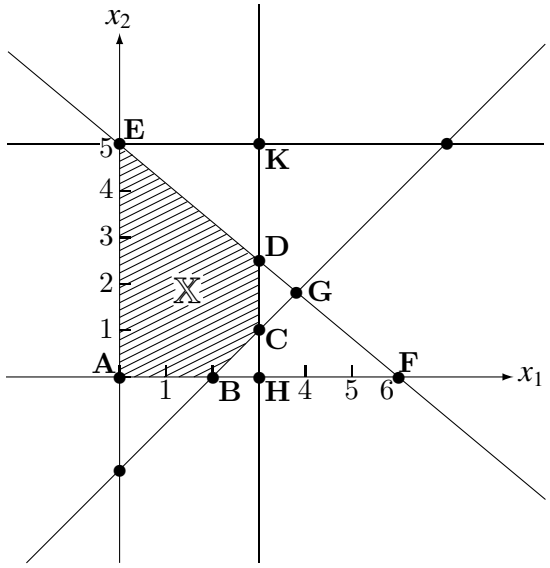
The union of these regions is $\mathbb{R}^2$ and the intersection of any two is empty. Each disjoint region is a convex polyhedron whose interior is not crossed by any constraint hyperplane. (AEF is a convex polyhedron but it is *not* disjoint from ABCDE because AEF $\cap$ ABCDE $\neq \varnothing$.)

In every partitioning of $\mathbb{R}^n$ some of the disjoint regions are unbounded; the 12 border regions in the picture are unbounded.

Exactly *one* of the disjoint regions contains all the points that satisfy all of the inequalities; it is called the **feasible set**. The feasible set is thus the intersection of the feasible halfspaces associated with the constraints. The feasible set $\mathbb{X}$ for our example, crosshatched in the graph on the next page, is this intersection of halfspaces:

$$\mathbb{X} = \left\{\mathbf{x}\,\middle|\,x_1 + \tfrac{6}{5}x_2 \le 6\right\} \cap \left\{\mathbf{x}\,\middle|\,x_1 - x_2 \le 2\right\} \cap \left\{\mathbf{x}\,\middle|\,x_1 \le 3\right\} \cap \left\{\mathbf{x}\,\middle|\,x_2 \le 5\right\} \cap \left\{\mathbf{x}\,\middle|\,x_1 \ge 0\right\} \cap \left\{\mathbf{x}\,\middle|\,x_2 \ge 0\right\}$$

The intersection of two or more constraint hyperplanes is called a **vertex** of the constraints. In our example there are 11 vertices, each of which is marked with a dot • in the graph on the next page. The vertices of the feasible set are called **extreme points**. An extreme point is a feasible point that is not the midpoint of any line segment contained in the feasible set. The point $[1, 0]^\top$ is not an extreme point, even though it is in the boundary of $\mathbb{X}$, because it is the midpoint of the feasible line segment $[\mathbf{A},\mathbf{B}]$. The point $[2, 0]^\top$, which is point $\mathbf{B}$, is an extreme point because it is not the midpoint of any line segment in $\mathbb{X}$.



An **edge** is a line segment between two vertices which lies on a constraint hyperplane and contains no other vertex; [**B**,**H**] is an edge but [**B**,**F**] and [**B**,**K**] are not. An edge of the feasible set is a line segment between two extreme points such that no point on the line segment is the midpoint of two distinct feasible points that are not on the line segment. [**D**,**E**] is an edge of $\mathbb{X}$, but [**A**,**D**] is not.

The **boundary** of the feasible set is the union of its edges and its rays. Rays are discussed in §3.3.3 below. In this example the feasible set has no rays, so its boundary is the union of its edges:

$$\partial\mathbb{X} = [\mathbf{A},\mathbf{E}] \cup [\mathbf{E},\mathbf{D}] \cup [\mathbf{D},\mathbf{C}] \cup [\mathbf{C},\mathbf{B}] \cup [\mathbf{B},\mathbf{A}]$$

3.2 Graphical Interpretation of Pivoting

We can put our example into standard form by adding slack variables, obtaining this algebraic formulation.

$$\begin{array}{lrcrrrrrcl}
\text{minimize} & -2x_1 & - & x_2 & & & & & & \\
\text{subject to} & x_1 & + & \frac{6}{5}x_2 & +s_1 & & & & = & 6 \\
 & x_1 & - & x_2 & & +s_2 & & & = & 2 \\
 & x_1 & & & & & +s_3 & & = & 3 \\
 & & & x_2 & & & & +s_4 & = & 5 \\
 & & & & & & & \mathbf{x} & \geq & \mathbf{0} \\
 & & & & & & & \mathbf{s} & \geq & \mathbf{0}
\end{array}$$

The tableau below representing this linear program has the basic solution $[0, 0, 6, 2, 3, 5]^\top$ in which $\mathbf{x} = \mathbf{0}$. In the graphs above $\mathbf{x} = \mathbf{0}$ is the origin, so the tableau is said to **correspond** to the origin in the graph, and to show this both are labeled **A**. In §3.3.1 we shall see how the basic variables (**s** in tableau **A**) can sometimes also be found in the graph.

$$\mathbf{A} = \begin{array}{c|cccccc}
 & x_1 & x_2 & s_1 & s_2 & s_3 & s_4 \\
\hline
0 & -2 & -1 & 0 & 0 & 0 & 0 \\
\hline
6 & 1 & \frac{6}{5} & 1 & 0 & 0 & 0 \\
2 & 1 & -1 & 0 & 1 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 1 & 0 & 0 & 0 & 1
\end{array}$$

3.2.1 Pivoting in Slow Motion

Suppose that in tableau **A** we let $x_1 = t \geq 0$ and keep $x_2 = 0$. Then to remain feasible we must adjust s_1, s_2, s_3, and s_4. The constraint rows require that

$$\begin{aligned} 6 = t + s_1 &\quad\Rightarrow\quad s_1 = 6 - t \\ 2 = t + s_2 &\quad\Rightarrow\quad s_2 = 2 - t \\ 3 = t + s_3 &\quad\Rightarrow\quad s_3 = 3 - t \\ 5 = s_4 &\quad\Rightarrow\quad s_4 = 5 \end{aligned}$$

Using these expressions for s_1, s_2, s_3, and s_4, we can write the basic solution represented by the tableau as a function of t, and from it we deduce that $t \leq 2$ to keep $\mathbf{x} \geq \mathbf{0}$.

$$\begin{bmatrix} x(t) \\ \hdashline s(t) \end{bmatrix} = \begin{bmatrix} t \\ 0 \\ \hdashline 6-t \\ 2-t \\ 3-t \\ 5 \end{bmatrix} \qquad \begin{array}{rcl} t & \geq & 0 \quad \checkmark \\ 0 & \geq & 0 \quad \checkmark \\ \left.\begin{array}{rcl} 6-t & \geq & 0 \;\Rightarrow\; t \leq 6 \\ 2-t & \geq & 0 \;\Rightarrow\; t \leq 2 \\ 3-t & \geq & 0 \;\Rightarrow\; t \leq 3 \end{array}\right\} & \Rightarrow t & \leq 2 \\ 5 & \geq & 0 \quad \checkmark \end{array}$$

This t–analysis should be familiar from §2.4.4; setting $t = 2$ corresponds to pivoting on the circled element of tableau **A** to the new basic solution $[2, 0, 4, 0, 1, 5]^\top$, which corresponds to the vertex marked **B** in the graphs. If $0 < t < 2$, however, the point represented by the tableau is interior to the line segment [**A**,**B**]. For example, if $t = 1$ the (nonbasic) point is $[1, 0, 5, 1, 2, 5]^\top$ which in the graph is halfway between **A** and **B**. Thus, if t increases gradually from 0 to 2, the point represented by the tableau slides gradually from **A** to **B** in the picture.

3.2.2 A Guided Tour in $\mathbb{R}^2$

A pivot moves the basic solution represented by the tableau from one vertex to another along (and only along) a constraint hyperplane. The `pivot` session below shows the trajectory of basic solutions resulting from a sequence of pivots, including some pivots that do not follow the simplex rule. As you read this Section it will be helpful to refer to the graph in §3.1. The file `tour.tab` contains the tableau labeled **A** above.

```
> This is PIVOT, Unix version 4.2
> For a list of commands, enter HELP.
>
< read tour.tab
Reading the tableau...
...done.
```



```
    x1  x2   s1  s2  s3  s4
0. -2. -1.0  0.  0.  0.  0.
6.  1.  1.2  1.  0.  0.  0.
2.  1. -1.0  0.  1.  0.  0.
3.  1.  0.0  0.  0.  1.  0.
5.  0.  1.0  0.  0.  0.  1.

< * This is tableau A, corresponding to point A = [0,0] in the
< * picture.  When we put our example into standard form that
< * happened to also put it into canonical form.  Using phase 2 of
< * the simplex algorithm we can pivot to optimality like this.
<
< pivot 3 2

    x1  x2   s1  s2  s3  s4
4.  0. -3.0  0.  2.  0.  0.
4.  0.  2.2  1. -1.  0.  0.
2.  1. -1.0  0.  1.  0.  0.
1.  0.  1.0  0. -1.  1.  0.
5.  0.  1.0  0.  0.  0.  1.

< * This is tableau B, corresponding to point B = [2,0].
<
< pivot 4 3

     x1  x2  s1  s2   s3   s4
7.0  0.  0.  0. -1.0  3.0  0.
1.8  0.  0.  1.  1.2 -2.2  0.
3.0  1.  0.  0.  0.0  1.0  0.
1.0  0.  1.  0. -1.0  1.0  0.
4.0  0.  0.  0.  1.0 -1.0  1.

< * This is tableau C.
<
< pivot 2 5

     x1  x2  s1         s2  s3         s4
8.5  0.  0.  0.8333333  0.  1.1666667  0.
1.5  0.  0.  0.8333333  1. -1.8333333  0.
3.0  1.  0.  0.0000000  0.  1.0000000  0.
2.5  0.  1.  0.8333333  0. -0.8333333  0.
2.5  0.  0. -0.8333333  0.  0.8333333  1.

< * This is tableau D.%
< * We have found the optimal point; now let's pivot back to the
< * starting tableau.
<
< pivot 2 4
```



```
     x1  x2  s1  s2   s3   s4
 7.0  0.  0.  0. -1.0  3.0  0.
 1.8  0.  0.  1.  1.2 -2.2  0.
 3.0  1.  0.  0.  0.0  1.0  0.
 1.0  0.  1.  0. -1.0  1.0  0.
 4.0  0.  0.  0.  1.0 -1.0  1.

 < * This is tableau C.
 <
 < pivot 4 6

    x1  x2   s1  s2  s3  s4
 4.  0. -3.0  0.  2.  0.  0.
 4.  0.  2.2  1. -1.  0.  0.
 2.  1. -1.0  0.  1.  0.  0.
 1.  0.  1.0  0. -1.  1.  0.
 5.  0.  1.0  0.  0.  0.  1.

 < * This is tableau B.
 <
 < pivot 3 5

    x1  x2   s1  s2  s3  s4
 0. -2. -1.0  0.  0.  0.  0.
 6.  1.  1.2  1.  0.  0.  0.
 2.  1. -1.0  0.  1.  0.  0.
 3.  1.  0.0  0.  0.  1.  0.
 5.  0.  1.0  0.  0.  0.  1.

 < * This is tableau A.
 < * We are back where we began.  The cost coefficient of x2 is also
 < * negative, so there is another path to the optimal point.  In
 < * the x2 column there is a tie for the minimum ratio,
 < * so there are two possible pivots.
 <
 < pivot 5 3

    x1  x2  s1  s2  s3  s4
 5. -2.  0.  0.  0.  0.  1.0
-0.  1.  0.  1.  0.  0. -1.2
 7.  1.  0.  0.  1.  0.  1.0
 3.  1.  0.  0.  0.  1.  0.0
 5.  0.  1.  0.  0.  0.  1.0

 < * This is tableau E1, with x1 and s4 nonbasic.
 < * Because there was a tie in the minimum ratio
 < * in tableau A, this tableau has a zero constant column entry b1=0
 < * (the minus sign is due to roundoff in the numerical calculations).
 <
```



```
< pivot 2 2

    x1  x2  s1  s2  s3  s4
 5.  0.  0.  2.  0.  0. -1.4
-0.  1.  0.  1.  0.  0. -1.2
 7.  0.  0. -1.  1.  0.  2.2
 3.  0.  0. -1.  0.  1.  1.2
 5.  0.  1.  0.  0.  0.  1.0

< * This is tableau E2, with s1 and s4 nonbasic.  E is said to be a
< * degenerate vertex, because 3 constraint hyperplanes
< * intersect there but only 2 are needed to determine the point in
< * R^2 (x2 <= 5 is redundant).  Because b1=0 the pivot we did
< * at (2,2) is called a degenerate pivot.  The objective
< * did not change, and this tableau corresponds to the same point
< * E as the previous one; only the basic sequence changed.
<
< pivot 2 7

    x1         x2  s1         s2  s3  s4
 5. -1.1666667  0.  0.8333333  0.  0.  0.
+0. -0.8333333  0. -0.8333333  0.  0.  1.
 7.  1.8333333  0.  0.8333333  1.  0.  0.
 3.  1.0000000  0.  0.0000000  0.  1.  0.
 5.  0.8333333  1.  0.8333333  0.  0.  0.

< * This is tableau E3, with x1 and s1 nonbasic.
< * This tableau corresponds to the vertex E in the final way that
< * is possible.  Because the pivot at (2,7) was once again
< * degenerate it changed neither the objective nor the point.
<
< pivot 4 2

     x1  x2  s1         s2  s3         s4
 8.5  0.  0.  0.8333333  0.  1.1666667  0.
 2.5  0.  0. -0.8333333  0.  0.8333333  1.
 1.5  0.  0.  0.8333333  1. -1.8333333  0.
 3.0  1.  0.  0.0000000  0.  1.0000000  0.
 2.5  0.  1.  0.8333333  0. -0.8333333  0.

< * This is tableau D.
< * It is equivalent to the first optimal tableau we found
< * but has the constraint rows permuted.
<
< * Pivoting not by the simplex rule leads to infeasible points.
< * Such pivots are called exterior pivots and by performing them
< * we can visit other vertices.
<
< pivot 3 6
```



```
           x1  x2  s1         s2         s3  s4
 9.4545455  0.  0.  1.3636364 0.63636364  0.  0.
 3.1818182  0.  0. -0.4545455 0.45454545  0.  1.
-0.8181818  0.  0. -0.4545455 -.54545455  1.  0.
 3.8181818  1.  0.  0.4545455 0.54545455  0.  0.
 1.8181818  0.  1.  0.4545455 -.45454545  0.  0.

 < * This is tableau G.
 < pivot 5 5

      x1  x2   s1  s2  s3  s4
 12.  0.  1.4  2.  0.  0.  0.
  5.  0.  1.0  0.  0.  0.  1.
 -3.  0. -1.2 -1.  0.  1.  0.
  6.  1.  1.2  1.  0.  0.  0.
 -4.  0. -2.2 -1.  1.  0.  0.

 < * This is tableau F.
 < pivot 4 3

     x1         x2  s1         s2  s3  s4
 5. -1.1666667  0.  0.8333333  0.  0.  0.
+0. -0.8333333  0. -0.8333333  0.  0.  1.
 3.  1.0000000  0.  0.0000000  0.  1.  0.
 5.  0.8333333  1.  0.8333333  0.  0.  0.
 7.  1.8333333  0.  0.8333333  1.  0.  0.

 < * This is tableau E3 with its constraint rows permuted.
 < * Notice that with a single pivot we jumped over two vertices.
 < undo

      x1  x2   s1  s2  s3  s4
 12.  0.  1.4  2.  0.  0.  0.
  5.  0.  1.0  0.  0.  0.  1.
 -3.  0. -1.2 -1.  0.  1.  0.
  6.  1.  1.2  1.  0.  0.  0.
 -4.  0. -2.2 -1.  1.  0.  0.

 < * This is tableau F.
 < * By choosing a different pivot we can jump to a different
 < * tableau representing the E vertex.
 < pivot 2 3

     x1  x2  s1  s2  s3  s4
 5.  0.  0.  2.  0.  0. -1.4
 5.  0.  1.  0.  0.  0.  1.0
 3.  0.  0. -1.  0.  1.  1.2
-0.  1.  0.  1.  0.  0. -1.2
 7.  0.  0. -1.  1.  0.  2.2
```



```
< * This is the E2 tableau with its rows permuted.
< * The E1 tableau can't be reached from point F in one pivot
< * because it differs from the point F tableau in two basis
< * columns, not just one; to reach it we would need to "turn the
< * corner" (even though it's the same point) by performing a
< * second pivot.
<
< * Instead let's visit the remaining vertices shown in the
< * picture (there are two other vertices that are not shown).
< pivot 3 4

     x1  x2  s1  s2  s3  s4
 11.  0.  0.  0.  0.  2.  1.0
  5.  0.  1.  0.  0.  0.  1.0
 -3.  0.  0.  1.  0. -1. -1.2
  3.  1.  0.  0.  0.  1.  0.0
  4.  0.  0.  0.  1. -1.  1.0

< * This is the K tableau.
< pivot 2 7

    x1  x2   s1  s2  s3  s4
 6.  0. -1.0  0.  0.  2.  0.
 5.  0.  1.0  0.  0.  0.  1.
 3.  0.  1.2  1.  0. -1.  0.
 3.  1.  0.0  0.  0.  1.  0.
-1.  0. -1.0  0.  1. -1.  0.

< * This is the H tableau.
< quit
> STOP
```

3.2.3 Observations From the Guided Tour

Having explored our example problem by pivoting, we can now say some more about the relationships between its graph and its tableaus.

Two tableaus that are connected by a single pivot (e.g., tableaus **A** and **B** or tableaus **E1** and **E2**) are called **adjacent tableaus**; two vertices that are connected by a single edge (e.g., vertices **A** and **B**) are called **adjacent vertices**.

Each tableau or basic solution of the constraint equations corresponds to exactly one vertex or intersection of constraint hyperplanes [3, p96] (see Exercise 3.7.7). A single pivot can move from any vertex on a hyperplane to any other vertex on the hyperplane (or to the same vertex if it is degenerate) because only a single basis column swap is needed. However, a single vertex can correspond to several different tableaus. Whether or not the vertex is degenerate, the constraint rows can be permuted, yielding different tableaus that correspond

to the same point (and hence the same tableau letter in the example) but have different basic sequences. If the vertex is nondegenerate (exactly n constraint hyperplanes intersect there) then the same variables are basic in each tableau and the basic feasible solution is the same in each tableau. In the example only one tableau letter corresponds to each nondegenerate vertex.

If the vertex is degenerate (more than n hyperplanes cross) then the same variables are zero in each tableau and the basic feasible solution is the same in each tableau, but some zero variables are basic with $b_i = 0$ while others are zero because they are nonbasic. If a vertex is degenerate some of its tableaus might be more than one pivot away from a tableau corresponding to an adjacent vertex (in our example, tableau **E1** is two pivots from tableau **D**, even though vertex **E** is only one edge away from vertex **D**).

If a vertex in $\mathbb{R}^n$ is the intersection of r constraint hyperplanes then [153, §1.4] there are

$$\binom{r}{n} = \frac{r!}{n!(r-n)!}$$

different sets of basic variables corresponding to the point. In our example vertex **E** is degenerate with $r = 3$, so we found

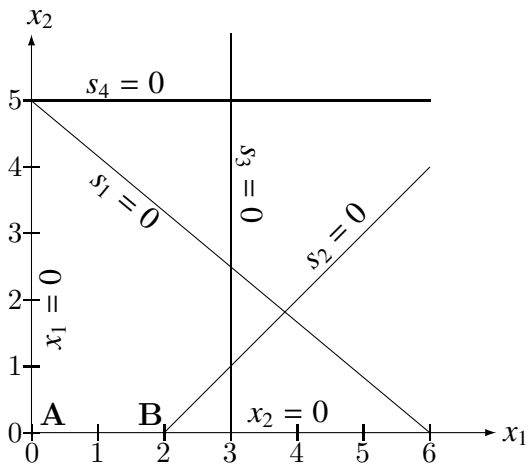
$$\binom{3}{2} = \frac{3!}{2!(3-2)!} = \frac{3 \times 2 \times 1}{(2 \times 1)(1)} = 3$$

tableaus **E1**, **E2**, and **E3**, with different basic variables, all corresponding to that vertex.

Nondegenerate phase-2 simplex pivots yield adjacent tableaus corresponding to adjacent extreme points, because each pivot turns a corner of the feasible set. However, if we start at a feasible point and pivot not by the simplex rule, we move along the hyperplane to a vertex that is not adjacent to the starting point and is thus not an extreme point (in our example if we start at point **A** and do a pivot that is in the x_1 column but not in the minimum-ratio row, we move to vertex **H** or **F** rather than to vertex **B**).

3.3 Graphical Interpretation of Tableaus

We have described a constraint's hyperplane as the set of points where that inequality is satisfied with equality, but each hyperplane is also the set of points where that constraint's slack variable is zero. In our example, on the hyperplane $\{\mathbf{x} | x_1 + \frac{6}{5}x_2 = 6\}$ we have $s_1 = 0$. Similarly, each coordinate axis is the set of points where the other coordinates are zero; on the x_1 axis we have $x_2 = 0$. Our example is graphed again on the next page with each constraint identified by which variable is zero on its hyperplane. Pivoting from **A** to **B** decreases s_1, s_2, and s_3 in the tableau because in the picture the point **B** is closer than point **A** is to the hyperplanes where those variables are zero. A vertex is the intersection of hyperplanes where a variable (slack or coordinate) is zero, so it is possible to move to any vertex by pivoting to make those variables nonbasic. At **A**, $x_1 = x_2 = 0$; at **B**, $x_2 = s_2 = 0$.



$\mathbf{A} =$

	x_1	x_2	s_1	s_2	s_3	s_4
0	−2	−1	0	0	0	0
6	1	$\frac{6}{5}$	1	0	0	0
2	1	−1	0	1	0	0
3	1	0	0	0	1	0
5	0	1	0	0	0	1

$\mathbf{B} =$

	x_1	x_2	s_1	s_2	s_3	s_4
4	0	−3	0	2	0	0
4	0	$\frac{11}{5}$	1	−1	0	0
2	1	−1	0	1	0	0
1	0	1	0	−1	1	0
5	0	1	0	0	0	1

3.3.1 Slack Variables in the Graph

Sometimes it is also possible to read off the values of the basic variables from the graph. At the point $\mathbf{x} = \mathbf{0}$, to which tableau **A** corresponds, $\mathbf{s} = [6, 2, 3, 5]^\top$ and these values can be found in the graph as the x_1, x_1, x_1, and x_2 intercepts of the hyperplanes on which the slacks are zero. In each case, if the coefficient of x_p in an equality constraint (i.e., in the tableau constraint row) is 1, then it is the x_p intercept of the corresponding inequality's hyperplane that tells the value of the slack variable associated with that constraint.

If we rewrite the first inequality as $\frac{5}{6}x_1 + x_2 \leq 5$ *before* adding slacks to get standard form, the picture remains unchanged but corresponding to point **A** we get the tableau on the left below.

	x_1	x_2	s_1	s_2	s_3	s_4
0	−2	−1	0	0	0	0
5	$\frac{5}{6}$	1	1	0	0	0
2	1	−1	0	1	0	0
3	1	0	0	0	1	0
5	0	1	0	0	0	1

	x_1	x_2	s_1	s_2	s_3	s_4
0	−2	−1	0	0	0	0
30	5	6	1	0	0	0
2	1	−1	0	1	0	0
3	1	0	0	0	1	0
6	0	1	0	0	0	1

Now it is x_2 that has a coefficient of 1 in the first constraint, so to read off the value $s_1 = 5$ from the graph we must use the x_2 intercept of the $s_1 = 0$ constraint hyperplane.

Of course we could write the first inequality like this instead: $5x_1 + 6x_2 \leq 30$. Then the tableau corresponding to the origin, shown on the right above, has $s_1 = 30$. Now neither x_1 nor x_2 has a coefficient of 1 in that constraint, so we cannot read the tableau's value of $s_1 = 30$ from the graph directly on either coordinate axis. However, by looking at the tableau we can see that the x_1 intercept of that constraint hyperplane will be $30/5 = 6$ and the x_2 intercept will be $30/6 = 5$.

3.3.2 Alternate Views of a Linear Program

If we are given only a graph of the constraint and objective contours for a linear program with inequality constraints, we can easily write down an algebraic statement of the problem. Then, using the techniques of §2.9 and §2.8, we can put the problem into standard form and pivot to obtain a canonical-form tableau.

If we are instead given only a canonical-form tableau that someone obtained in that way, can we figure out what inequality-constrained linear program they must have started with? To study this question, consider the canonical-form tableau on the left below (it happens to be in optimal form but that is not a requirement for what we are about to do).

	x_1	x_2	x_3	x_4	x_5	x_6
$\frac{17}{2}$	0	0	$\frac{5}{6}$	0	$\frac{7}{6}$	0
3	1	0	0	0	1	0
$\frac{5}{2}$	0	1	$\frac{5}{6}$	0	$-\frac{5}{6}$	0
$\frac{3}{2}$	0	0	$\frac{5}{6}$	1	$-\frac{11}{6}$	0
$\frac{5}{2}$	0	0	$-\frac{5}{6}$	0	$\frac{5}{6}$	1

$$\begin{aligned} z + \tfrac{17}{2} &= \tfrac{5}{6}x_3 + \tfrac{7}{6}x_5 \\ 3 &= x_1 + x_5 \\ \tfrac{5}{2} &= x_2 + \tfrac{5}{6}x_3 - \tfrac{5}{6}x_5 \\ \tfrac{3}{2} &= \tfrac{5}{6}x_3 + x_4 - \tfrac{11}{6}x_5 \\ \tfrac{5}{2} &= -\tfrac{5}{6}x_3 + \tfrac{5}{6}x_5 + x_6 \end{aligned}$$

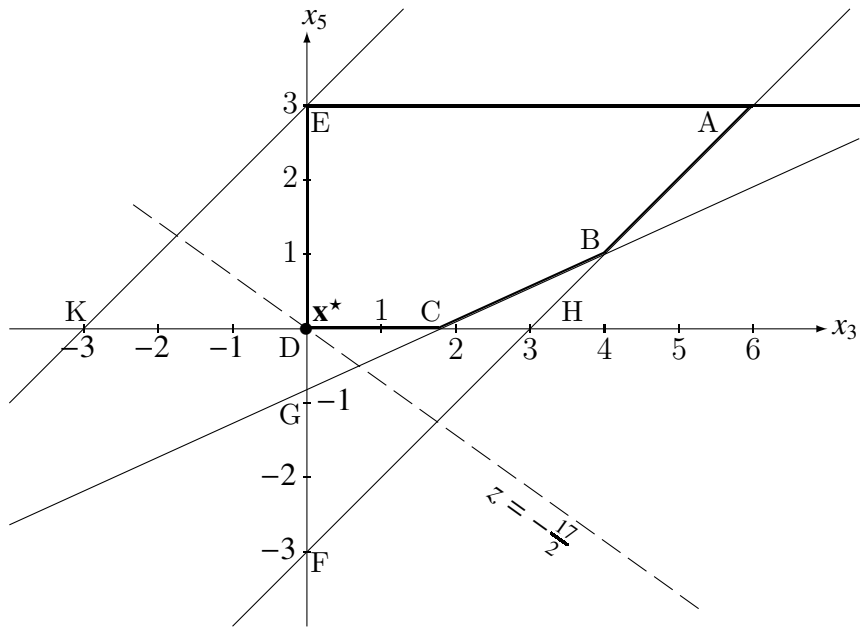
The equations represented by the tableau are given to its right. If we rearrange them as shown below, so that the nonbasic variables x_3 and x_5 come first, then the basic variables x_1, x_2, x_4, and x_6 look like slacks that were added to turn $\leq$ constraints into equalities.

$$\begin{aligned} z + \tfrac{17}{2} &= \tfrac{5}{6}x_3 + \tfrac{7}{6}x_5 \\ 3 &= x_5 + x_1 \\ \tfrac{5}{2} &= \tfrac{5}{6}x_3 - \tfrac{5}{6}x_5 + x_2 \\ \tfrac{3}{2} &= \tfrac{5}{6}x_3 - \tfrac{11}{6}x_5 + x_4 \\ \tfrac{5}{2} &= -\tfrac{5}{6}x_3 + \tfrac{5}{6}x_5 + x_6 \end{aligned}$$

Looked at in this way, the tableau must have come from the inequality-constrained linear program below.

$$\begin{array}{lrcl} \text{minimize} & -\tfrac{17}{2} + \tfrac{5}{6}x_3 + \tfrac{7}{6}x_5 & = & z \\ \text{subject to} & x_5 & \leq & 3 \\ & \tfrac{5}{6}x_3 - \tfrac{5}{6}x_5 & \leq & \tfrac{5}{2} \\ & \tfrac{5}{6}x_3 - \tfrac{11}{6}x_5 & \leq & \tfrac{3}{2} \\ & -\tfrac{5}{6}x_3 + \tfrac{5}{6}x_5 & \leq & \tfrac{5}{2} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

Using this statement of the problem we can graph the constraint and objective contours as usual, obtaining the picture on the next page.



The tableau we began with is actually the optimal tableau of the example problem we have been using all along, as you should verify by finding tableau **D** in the Guided Tour of §3.2.2. (To emphasize that we could have started from any canonical-form tableau, without knowing where it came from, I disguised tableau **D** here by using rational rather than decimal fractions and by replacing the variable names s_1 through s_4 with x_3 through x_6.)

This graph and the one in §3.1 both describe the same linear program, but they look quite different because they were drawn from different tableaus. The graph in §3.1 is a **view** of the problem from tableau **A**, whereas the graph above is a view of the problem from tableau **D**. The nonbasic variables in the tableau from which a view is drawn are always the axes of that view's graph, so the origin of the graph corresponds to the basic feasible solution in the tableau and the dimension of the feasible set is the number of nonbasic variables. Because the coordinates of the graph are the nonbasic variables, the values of the basic variables can (perhaps) be read from the graph only by thinking of them as slacks and using the approach discussed in §3.3.1.

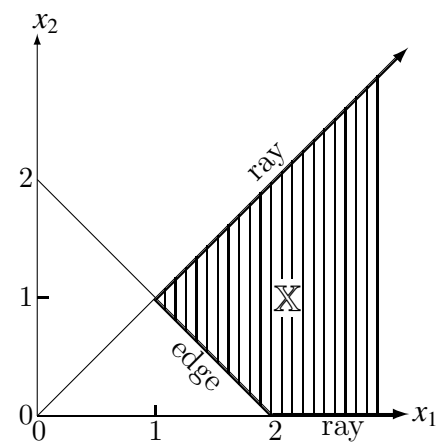
Because tableau **D** is in optimal form, $\mathbf{x}^\star$ is where the nonbasic variables x_3 and x_5 are zero, which is the origin of the graph in this view. In the view from tableau **A**, the optimal point is still at vertex **D** but that vertex is not at the origin.

Notice that the feasible set in the view from tableau **D**, outlined above in thick lines, has the same vertices, in the same order, as the feasible set in the view from tableau **A**. At each iteration the simplex algorithm sees the problem from the perspective of the current basic feasible solution; it uses only local information. The solution process can be thought of as generating a sequence of views, each pivot moving from the origin in the current view to the vertex that will become the origin in the next.

3.3.3 Unbounded Feasible Sets

It is possible for a linear program to have an **unbounded feasible set**, as shown by this example (we will consider several possible objective functions).

$$\begin{array}{lrcrcl} \text{minimize} & & & & & z \\ \text{subject to} & x_1 & - & x_2 & \geq & 0 \\ & x_1 & + & x_2 & \geq & 2 \\ & & & \mathbf{x} & \geq & 0 \end{array}$$



If a feasible set is unbounded it includes **feasible rays**. In this problem the boundary of $\mathbb{X}$ is the one edge and two rays shown in the graph.

$$\begin{array}{rcll} \partial\mathbb{X} & = & \{\mathbf{x} \mid x_1 + x_2 = 2\} \cap \{\mathbf{x} \mid 1 \leq x_1 \leq 2\} & \text{edge} \\ & \cup & \{\mathbf{x} \mid x_1 = x_2 \geq 1\} \qquad \text{diagonal} & \text{ray} \\ & \cup & \{\mathbf{x} \mid x_2 = 0,\ x_1 \geq 2\} \qquad \text{horizontal} & \text{ray} \end{array}$$

Unbounded optimal value. If a linear program has an unbounded optimal value, like the `unbd` problem of §2.5.2, then its feasible set must be unbounded too. The linear program above is unbounded if, for example, $z = -x_1 - 2x_2$. Then it has these starting and final tableaus.

$$\begin{array}{c|cccc} & x_1 & x_2 & s_1 & s_2 \\ \hline 0 & -1 & -2 & 0 & 0 \\ \hline 0 & -1 & 1 & 1 & 0 \\ -2 & -1 & -1 & 0 & 1 \\ \hline \end{array} \quad \longrightarrow \quad \begin{array}{c|cccc} & x_1 & x_2 & s_1 & s_2 \\ \hline 3 & 0 & 0 & \frac{1}{2} & -\frac{3}{2} \\ \hline 1 & 1 & 0 & -\frac{1}{2} & -\frac{1}{2} \\ 1 & 0 & 1 & \frac{1}{2} & -\frac{1}{2} \\ \hline \end{array}$$

The final tableau's s_2 column reveals unbounded form, because $c_4 < 0$ and $a_{i4} \leq 0$ for all i. If we let $s_2 = t \geq 0$ and keep $s_1 = 0$ then its constraint rows require

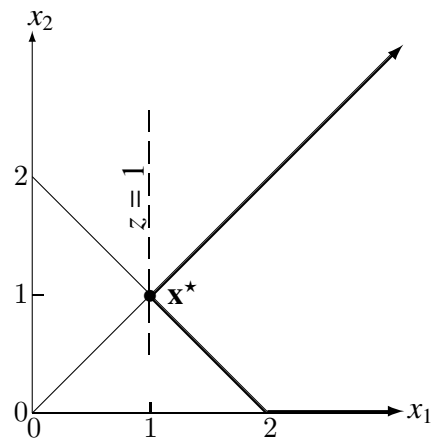
$$\begin{array}{rcl} 1 & = & x_1 - \frac{1}{2}t \quad \Rightarrow \quad x_1 = 1 + \frac{1}{2}t \\ 1 & = & x_2 - \frac{1}{2}t \quad \Rightarrow \quad x_2 = 1 + \frac{1}{2}t \end{array}$$

so $x_1 = x_2$ and both remain nonnegative no matter how high we make t. From the objective row we see that $z = -3 - \frac{3}{2}t$, so

$$\lim_{t \to \infty} z = -\infty.$$

Starting from the point $[1, 1]^\top$ corresponding to the final tableau, we can imagine sliding $\mathbf{x}$ to the right and up along the diagonal ray forever, decreasing the objective as we go.

Unique optimal point. But a linear program with an unbounded feasible set need not have an unbounded optimal value. The linear program above has a unique optimal point if, for example, $z = x_1$. Then it has the optimal point shown in the graph to the right, and these are the starting and final tableaus. This final tableau is in optimal form.



	x_1	x_2	s_1	s_2
0	1	0	0	0
0	−1	1	1	0
−2	−1	−1	0	1

$\longrightarrow$

	x_1	x_2	s_1	s_2
−1	0	0	$\frac{1}{2}$	$\frac{1}{2}$
1	0	1	$\frac{1}{2}$	$-\frac{1}{2}$
1	1	0	$-\frac{1}{2}$	$-\frac{1}{2}$

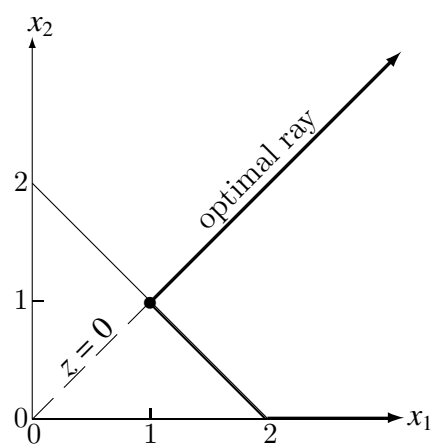
The optimal tableau's s_2 column still indicates a feasible ray, because $a_{i4} \leq 0$ for all i. If we again let $s_2 = t \geq 0$ and keep $s_1 = 0$, we find as before that $x_1 = x_2 = \frac{1}{2}t$, so $\mathbf{x}$ remains feasible no matter how high we make t. Now, however, $z = 1 + \frac{1}{2}t$ so only the point $[1, 1]^\top$ where $t = 0$ is optimal. The signal of unbounded form that we identified in §2.5.2 is actually a tableau column indicating a ray that happens also to have a negative c_j.

3.4 Multiple Optimal Solutions

If the objective of a linear program has its optimal value at two different feasible points, then those points are **multiple optimal solutions**. If the objective contours are parallel to an edge or ray of the feasible set, that whole edge or ray can be optimal. If the feasible set is bounded then any multiple optima must be on an edge, but a problem with an unbounded feasible set can have multiple optima either on an edge or on a ray.

3.4.1 Optimal Rays

If in the example of §3.3.3 we let $z = x_1 - x_2$ then the optimal set is the whole ray from $[1, 1]^\top$ (including that point). In the final tableau the ray that is indicated by the s_2 column (because $a_{i4} \leq 0$ for all i) is now optimal, because $c_4 = 0$. If we let $s_2 = t \geq 0$ and keep $s_1 = 0$ we still find that $x_1 = x_2 = 1 + \frac{1}{2}t$, so that $\mathbf{x}$ remains feasible no matter how high we make t.



	x_1	x_2	s_1	s_2
0	1	−1	0	0
0	−1	1	1	0
−2	−1	−1	0	1

$\longrightarrow$

	x_1	x_2	s_1	s_2
0	0	0	1	0
1	0	1	$\frac{1}{2}$	$-\frac{1}{2}$
1	1	0	$-\frac{1}{2}$	$-\frac{1}{2}$

Now, however, $z = 0$ independent of t, so every point on the ray is optimal.

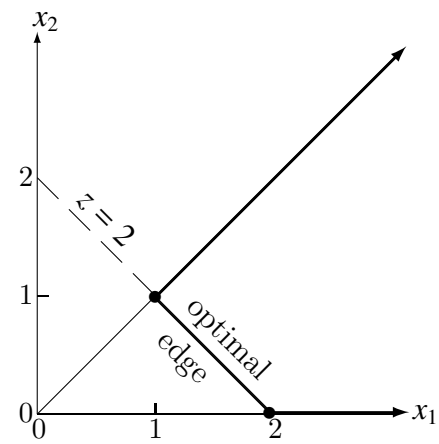
3.4.2 Optimal Edges

If in the example of §3.3.3 we let $z = x_1 + x_2$ then we can solve the problem by pivoting as shown below.

$$\begin{array}{c|cccc} & x_1 & x_2 & s_1 & s_2 \\ \hline 0 & 1 & 1 & 0 & 0 \\ \hline 0 & -1 & 1 & 1 & 0 \\ -2 & \textcircled{-1} & -1 & 0 & 1 \end{array} \longrightarrow \begin{array}{c|cccc} & x_1 & x_2 & s_1 & s_2 \\ \hline -2 & 0 & 0 & 0 & 1 \\ \hline 2 & 0 & \textcircled{2} & 1 & -1 \\ 2 & 1 & 1 & 0 & -1 \end{array} \longrightarrow \begin{array}{c|cccc} & x_1 & x_2 & s_1 & s_2 \\ \hline -2 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & \frac{1}{2} & -\frac{1}{2} \\ 1 & 1 & 0 & -\frac{1}{2} & -\frac{1}{2} \end{array}$$

The starting tableau for this problem, on the left, corresponds to the origin. Pivoting on the circled element yields the middle tableau, which is in optimal form and corresponds to the point $[2, 0]$ in the picture.

The x_2 column in the middle tableau is nonbasic but it has $c_2 = 0$, so if we pivot anywhere in that column the multiple of the pivot row that gets added to the objective row is zero. That means the $(1, 1)$ element of the tableau won't change, so the objective value will remain the same. The right tableau, resulting from the minimum-ratio pivot, corresponds to the point $[1, 1]^\top$ in the picture and is also in optimal form.



It is clear from the graph that the whole edge between $[1, 1]^\top$ and $[2, 0]^\top$ is optimal, but by pivoting we can find only the endpoints because they correspond to basic solutions of the constraint equations. Of course we can find the interior points of the line segment by pivoting in slow motion as in §3.2.1.

3.4.3 Signal Tableau Columns

We have seen, in this Chapter and in §2, that certain properties of a linear program are indicated by the signs of the entries in its tableau. In the summary of these patterns given below, when multiple signs are shown for the a_{ip} that means those entries can have a mixture of the signs shown. Most of these sign patterns occur in only some of a linear program's canonical form tableaus, and none of them necessarily mean anything in a tableau that is *not* in canonical form. Recall from §2.5.3 that infeasibility is signalled by sign patterns in a tableau's *rows,* and is discovered in the process of trying to get canonical form.

$$\begin{array}{c} x_p \\ \hline \begin{array}{c} - \\ \hline - \\ 0 \\ + \end{array} \end{array}$$

The tableau is not yet in optimal form, and this column is a candidate pivot column. In the simplex rule, we pivot on a positive a_{ip} for which b_i/a_{ip} is the smallest.

x_p
0
−
0
+

A pivot in this column will not change the objective value, so if the tableau is in optimal form (which depends on the other c_j) and pivoting in this column by the simplex rule yields a new basic feasible solution, that point is an alternate optimum.

x_p
+
−
0
+

A simplex pivot in this column would make the objective value worse, so this is not a candidate pivot column if we are solving the linear program. If the other c_j are also nonnegative then the tableau is in optimal form.

x_p
−
−
0
−

The feasible set is unbounded, and the linear program has an unbounded optimal value. This is the "unbounded" final form of §2.5.2.

x_p
0
−
0
−

The feasible set is unbounded, and an optimal ray emanates from the basic feasible solution represented by the tableau.

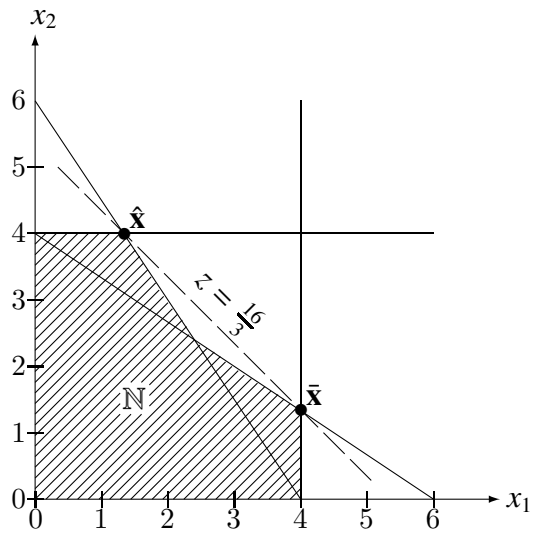
x_p
+
−
0
−

The feasible set is unbounded, and a non-optimal feasible ray emanates from the basic feasible solution represented by the tableau.

3.5 Convex Sets

The linear program we studied in §3.4.2 had two optimal vertices, and from the graphical solution we could see that the line segment connecting them was also optimal. The interior points of that line segment are not basic, but they can be discovered by slow-motion pivoting as described in §3.2.1. Is it *always* true that the line segment between two optimal points is also optimal, and that it can be traced out by slow-motion pivoting?

Consider the problem whose graphical solution is shown at the top of the next page. Here there are also two optimal points, $\hat{\mathbf{x}}$ and $\bar{\mathbf{x}}$, but the line segment between them falls *outside* of the feasible set. Because the interior points of that line segment are infeasible, they cannot be optimal.



$$\begin{array}{lrcl} \text{minimize} & -x_1 - x_2 & = & z \\ \text{subject to} & x_2 & \le & \max\left\{6 - \frac{3}{2}x_1,\ 4 - \frac{2}{3}x_1\right\} \\ & x_1 & \le & 4 \\ & x_2 & \le & 4 \\ & \mathbf{x} & \ge & 0 \end{array}$$

The picture describes the optimization problem stated above, which is of course not a linear program as defined in §1.1.1. The objective and constraint functions of a *linear* program must be *linear* functions, but here the first constraint has a kink in its graph. This *non*linear program really belongs later in the book, but it is useful here to illustrate a **nonconvex** feasible set.

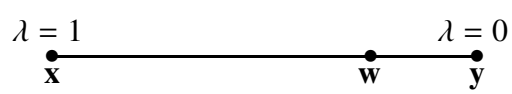
A set $\mathbb{S}$ is **convex** if and only if

$$\left.\begin{array}{l} \mathbf{x} \in \mathbb{S} \\ \mathbf{y} \in \mathbb{S} \end{array}\right\} \Rightarrow [\mathbf{x}, \mathbf{y}] \subseteq \mathbb{S}$$

The empty set, a single point, a line segment, a circle, an ellipse, the regular polygons, a halfspace, and $\mathbb{R}^n$ all satisfy this definition of a convex set [110, §4.1]. In the problem of §3.4.2, the line segment connecting the multiple optimal points is itself optimal because the feasible set of that problem is convex, but in the above problem $\mathbb{N}$ is nonconvex because $\hat{\mathbf{x}} \in \mathbb{N}$ and $\bar{\mathbf{x}} \in \mathbb{N}$ but the line segment $[\hat{\mathbf{x}}, \bar{\mathbf{x}}] \notin \mathbb{N}$. An equivalent but more often useful characterization is that a set $\mathbb{S}$ is convex if and only if

$$\left.\begin{array}{l} \mathbf{x} \in \mathbb{S} \\ \mathbf{y} \in \mathbb{S} \end{array}\right\} \Rightarrow \lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in \mathbb{S} \quad \text{for all } \lambda \in [0, 1].$$

The point $\mathbf{w} = \lambda\mathbf{x} + (1-\lambda)\mathbf{y},\ 0 \le \lambda \le 1$, is called a **convex combination** of $\mathbf{x}$ and $\mathbf{y}$ and is on the line between $\mathbf{x}$ and $\mathbf{y}$.



If the line above happens to be an edge of a linear program's feasible set, and if pivoting in slow motion slides $\mathbf{w}$ from $\mathbf{y}$ to $\mathbf{x}$, then the parameter t of §3.2.1 is zero at $\lambda = 0$ and equal to the minimum ratio at $\lambda = 1$.

3.5.1 Convexity of the Feasible Set

Using the second definition of convexity given above, we can prove that the feasible set of a linear program is always convex [3, §4.2].

Theorem: The set $\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ is convex.

Proof: Suppose that $\mathbf{x}^0 \in \mathbb{X}$ and $\mathbf{x}^1 \in \mathbb{X}$. Then to prove that $\mathbb{X}$ is a convex set it suffices to show that $\mathbf{w} = \lambda\mathbf{x}^0 + (1-\lambda)\mathbf{x}^1 \in \mathbb{X}$ for all $\lambda \in [0,1]$.

$$\begin{aligned}
\mathbf{x}^0 \in \mathbb{X} &\Rightarrow \mathbf{x}^{\mathbf{0}} \geq \mathbf{0} \\
\mathbf{x}^1 \in \mathbb{X} &\Rightarrow \mathbf{x}^{\mathbf{1}} \geq \mathbf{0} \\
\lambda \in [0,1] &\Rightarrow \lambda \geq 0 \text{ and } (1-\lambda) \geq 0 \\
\text{Thus } \lambda\mathbf{x}^0 + (1-\lambda)\mathbf{x}^1 &\geq \mathbf{0} \\
\text{so } \mathbf{w} &\geq \mathbf{0}.
\end{aligned}$$

$$\begin{aligned}
\mathbf{A}\mathbf{w} &= \mathbf{A}(\lambda\mathbf{x}^0 + (1-\lambda)\mathbf{x}^1) \\
\mathbf{A}\mathbf{w} &= \mathbf{A}\lambda\mathbf{x}^0 + \mathbf{A}(1-\lambda)\mathbf{x}^1 \\
\mathbf{A}\mathbf{w} &= \lambda\mathbf{A}\mathbf{x}^0 + (1-\lambda)\mathbf{A}\mathbf{x}^1 \\
\text{but } \mathbf{x}^0 \in \mathbb{X} &\Rightarrow \mathbf{A}\mathbf{x}^{\mathbf{0}} = \mathbf{b} \\
\text{and } \mathbf{x}^1 \in \mathbb{X} &\Rightarrow \mathbf{A}\mathbf{x}^{\mathbf{1}} = \mathbf{b}. \\
\text{Thus } \mathbf{A}\mathbf{w} &= \lambda\mathbf{b} + (1-\lambda)\mathbf{b} \\
\text{so } \mathbf{A}\mathbf{w} &= \mathbf{b}.
\end{aligned}$$

We have shown that $\mathbf{w} \geq \mathbf{0}$ and $\mathbf{A}\mathbf{w} = \mathbf{b}$, so $\mathbf{w} \in \mathbb{X}$ and $\mathbb{X}$ is convex. □

3.5.2 Convexity of the Optimal Set

In §3.4.2 the optimal set is a line segment, which is convex, but when $n > 2$ the optimal set can be of higher dimension. Is it still a convex set? Using the convexity of the feasible set, we can prove that it is [3, §4.2].

Theorem: The set of points that are optimal for a linear program is convex.

Proof: If $\mathbf{x}^\star$ is unique, it is convex because a point is convex. Otherwise suppose that $\mathbf{x}^0$ and $\mathbf{x}^1$ are distinct optimal vectors in $\mathbb{R}^n$. Then to prove that the optimal set is convex it suffices to show that $\mathbf{w} = \lambda\mathbf{x}^0 + (1-\lambda)\mathbf{x}^1$ is optimal for all $\lambda \in [0,1]$. For $\mathbf{w}$ to be optimal it must be feasible and have the optimal objective value.

$$\begin{aligned}
\mathbf{x}^0 \text{ optimal} &\Rightarrow \mathbf{x}^0 \in \mathbb{X} \\
\mathbf{x}^1 \text{ optimal} &\Rightarrow \mathbf{x}^1 \in \mathbb{X}
\end{aligned}$$

$$\left.\begin{array}{l}
\mathbf{x}^0 \in \mathbb{X} \\
\mathbf{x}^1 \in \mathbb{X} \\
\mathbb{X} \text{ is convex}
\end{array}\right\} \Rightarrow \mathbf{w} = \lambda\mathbf{x}^0 + (1-\lambda)\mathbf{x}^1 \in \mathbb{X} \quad \text{from §3.5.1}$$

$$\begin{aligned} \mathbf{x}^0 \text{ optimal} &\Rightarrow \mathbf{c}^\top\mathbf{x}^0 = z^\star \\ \mathbf{x}^1 \text{ optimal} &\Rightarrow \mathbf{c}^\top\mathbf{x}^1 = z^\star \\ \mathbf{w} = \lambda\mathbf{x}^0 + (1-\lambda)\mathbf{x}^1 &\Rightarrow \mathbf{c}^\top\mathbf{w} = \lambda\mathbf{c}^\top\mathbf{x}^0 + (1-\lambda)\mathbf{c}^\top\mathbf{x}^1 \end{aligned}$$

$$\left.\begin{aligned} &\mathbf{c}^\top\mathbf{x}^0 = z^\star \\ &\mathbf{c}^\top\mathbf{x}^1 = z^\star \\ &\mathbf{c}^\top\mathbf{w} = \lambda\mathbf{c}^\top\mathbf{x}^0 + (1-\lambda)\mathbf{c}^\top\mathbf{x}^1 \end{aligned}\right\} \Rightarrow \mathbf{c}^\top\mathbf{w} = \lambda z^\star + (1-\lambda)z^\star = z^\star$$

We have shown that $\mathbf{w} \in \mathbb{X}$ and $\mathbf{c}^\top\mathbf{w} = z^\star$ for all $\lambda \in [0,1]$, so any convex combination of optimal points is optimal and the optimal set of a linear program is convex. □

Convexity makes linear programming relatively easy, both in practice and in the theory of computational complexity (see §7.9). The example above illustrates that in a *non*linear program neither the feasible set nor the optimal set need be convex. We will revisit the subject of convexity from the standpoint of nonlinear programming in §11.

3.6 Higher Dimensions

In the preceding Sections of this Chapter we have discussed many amazing and delightful things about linear programming in $\mathbb{R}^2$, but how do they generalize to $\mathbb{R}^n$?

In $\mathbb{R}^3$ constraint and objective contours are planes instead of lines, and feasible sets look like faceted gemstones. In higher dimensions those geometrical objects are called hyperplanes and n-dimensional polyhedra, but giving them technical names does not help us much to imagine what they "look" like. Instead of pictures we must put our trust in linear algebra and the formal operations you learned in §2. Yet it is still true that the feasible set of a linear program is the intersection of its feasible halfspaces, that tableaus correspond to vertices, that a pivot moves from one vertex to another along a constraint hyperplane, that unbounded feasible sets have rays, that multiple optima are possible when the objective contours are parallel to a constraint, and so on. In fact, except for the pictures nothing we have done with our two-dimensional examples works *only* in two dimensions. The fundamental ideas are true in general, so they can inform your mathematical intuition and maybe help you to visualize some things that you can't actually see.

In this Section we consider two important problems which, although they are in more than two dimensions, can still be understood, or understood better, by thinking about the geometry of the simplex algorithm.

3.6.1 Finding All Optimal Solutions

In §3.4.2 we found an optimal edge, and it was easy to see from the picture that it was the *entire* optimal set. In higher dimensions, it can take more work to be sure that every optimal point has been accounted for.

Although it is not obvious from the starting tableau **A** below, this linear program [3, p103-105] has multiple optimal solutions. To find all of the optimal tableaus it is necessary to consider every possible simplex-rule pivot connecting them. This is also sufficient, because the convexity of the optimal set guarantees that if there are multiple optimal tableaus each will be adjacent to at least one of the others. To show that we have found them all, an arrow is drawn from each pivot position to the tableau that results from the pivot. The basic feasible solution corresponding to each tableau is given to its right.

	x_1	x_2	x_3	x_4	x_5
0	−1	−1	1	0	0
2	1	1	−1	1	0
4	−1	1	0	0	1

$\mathbf{A} = [0, 0, 0, 2, 4]^\top$

Tableau **A** is in canonical form, and its columns reveal that it is not yet in optimal form. There are two possible phase-2 simplex pivots.

	x_1	x_2	x_3	x_4	x_5
2	0	0	0	1	0
2	1	1	−1	1	0
6	0	2	−1	1	1

$\mathbf{B} = [2, 0, 0, 0, 6]^\top$

Tableau **B** is in optimal form, and its x_2 column reveals that there is another optimal point.

	x_1	x_2	x_3	x_4	x_5
2	0	0	0	1	0
2	1	1	−1	1	0
2	−2	0	1	−1	1

$\mathbf{C} = [0, 2, 0, 0, 2]^\top$

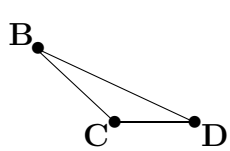
Tableau **C** is also optimal. Its x_1 column reveals an alternate optimum, but the circled pivot returns to **B**. The x_3 column indicates a different optimum.

	x_1	x_2	x_3	x_4	x_5
2	0	0	0	1	0
4	−1	1	0	0	1
2	−2	0	1	−1	1

$\mathbf{D} = [0, 4, 2, 0, 0]^\top$

Tableau **D** is in optimal form too. Its x_5 column reveals another optimum, but the circled pivot returns to tableau **C**.

We have identified the optimal basic solutions **B**, **C**, and **D**, but there are other optimal points that we can't find by pivoting. From §3.5.2 we know that every convex combination of the three optimal vertices is also optimal. This linear program has 5 variables, so its optimal vertices define a two-dimensional figure in $\mathbb{R}^5$, which is pictured on the left below. The side lengths $\|\mathbf{B} - \mathbf{C}\|_2$, $\|\mathbf{B} - \mathbf{D}\|_2$, and $\|\mathbf{C} - \mathbf{D}\|_2$ are drawn in correct proportions.



This triangle is called the **convex hull** $\mathbb{H}$ of the optimal vertices, and it contains all of their convex combinations [1, §2.1.3].

$$\mathbb{H} = \{\mathbf{x} \in \mathbb{R}^5 \mid \mathbf{x} = \alpha\mathbf{B} + \beta\mathbf{C} + \gamma\mathbf{D}, \alpha \geq 0, \beta \geq 0, \gamma \geq 0, \alpha + \beta + \gamma = 1\}$$

But $\mathbb{H}$ is not the whole optimal set, either. The x_3 column of tableau $\mathbf{B}$ and the x_1 column of tableau $\mathbf{D}$ each indicate an optimal ray. In tableau $\mathbf{B}$, we can't pivot in the x_3 column but we could increase x_3 and remain feasible. If we let $x_3 = t$ and keep x_2 and x_4 nonbasic, then the constraints require that

$$\mathbf{x}(t) = \begin{bmatrix} 2+t \\ 0 \\ t \\ 0 \\ 6+t \end{bmatrix} = \mathbf{B} + t \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \mathbf{B} + t\mathbf{u}.$$

Thus there is an optimal ray $\mathbf{u}$ that emanates from the optimal point $\mathbf{B}$ and goes in the direction $[1, 0, 1, 0, 1]^\top$ forever. In tableau $\mathbf{D}$, we can't pivot in the x_1 column but we could increase x_1 and still remain feasible. If we let $x_1 = t$ and keep x_4 and x_5 nonbasic, then the constraints require that

$$\mathbf{x}(t) = \begin{bmatrix} t \\ 4+t \\ 2+2t \\ 0 \\ 0 \end{bmatrix} = \mathbf{D} + t \begin{bmatrix} 1 \\ 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \mathbf{D} + t\mathbf{v}.$$

Thus there is an optimal ray $\mathbf{v}$ that emanates from the optimal point $\mathbf{D}$ and goes in the direction $[1, 1, 2, 0, 0]^\top$ forever. Of course it is not only $\mathbf{u}$ and $\mathbf{v}$ that belong to the optimal set, but all of their convex combinations as well. The convex hull of two rays in $\mathbb{R}^5$ is once again a two-dimensional figure, but this one is unbounded.

Formally we can say that the optimal set for this problem is that unbounded face of the feasible set which includes $\mathbb{H}$ and all convex combinations of the points on the rays $\mathbf{u}$ and $\mathbf{v}$ emanating from two vertices of $\mathbb{H}$. Perhaps we can imagine this geometry, and if the dimension of the feasible set were higher that is *all* we could do. But this problem has 3 nonbasic variables, so its feasible set is in $\mathbb{R}^3$ and we can actually draw a graph.

Because we found optimal rays it must be that the feasible set is unbounded, so to complete its characterization we must check whether it includes other rays. The x_3 column in tableau $\mathbf{A}$ and the x_4 column in tableau $\mathbf{D}$ indicate non-optimal rays, and by the same method we used above they are

$$\mathbf{p} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \text{ from } \mathbf{A} \qquad \mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \text{ from } \mathbf{D}$$

The feasible rays $\mathbf{p}$ and $\mathbf{q}$ happen to be parallel.

It is easy to sketch one plane in three dimensions but hard to sketch several with their intersections, so making an accurate picture requires a systematic approach and graphing software. The simplest procedure is to specify the coordinates of the corners of each face of the feasible set. Then the MATLAB `plot3()` function or the `gnuplot` command `splot` can be used to render the planes.

To begin it is necessary to select a view as we did in §3.3.2. For this problem the view that is easiest to interpret is the one from tableau **C**, in which the nonbasic variables are x_1, x_3, and x_4. These will be the coordinate axes in the graph, so in this view the coordinates of each vertex of the feasible set will be those elements of the corresponding basic solution, as follows.

$$\begin{aligned}
\hat{\mathbf{A}} &= [0,0,2]^\top \\
\hat{\mathbf{B}} &= [2,0,0]^\top \\
\hat{\mathbf{C}} &= [0,0,0]^\top \\
\hat{\mathbf{D}} &= [0,2,0]^\top
\end{aligned}$$

Because we will specify the corners of each face, we must pick a point on each ray at which to cut the unbounded feasible set. Each ray that appears in this view will, like each vertex, have for its components the x_1, x_3, and x_4 elements of the vectors we found above. For example, the ray **u** becomes in this view $\bar{\mathbf{u}} = [1,1,0]^\top$. Arbitrarily choosing $t = 10$, we find these points on the rays to specify as corners of the faces in which they lie.

$$\begin{aligned}
\hat{\mathbf{B}} + 10\bar{\mathbf{u}} = [2,0,0]^\top + 10[1,1,0]^\top = [12,10,0]^\top &= \hat{\mathbf{u}} \\
\hat{\mathbf{D}} + 10\bar{\mathbf{v}} = [0,2,0]^\top + 10[1,2,0]^\top = [10,22,0]^\top &= \hat{\mathbf{v}} \\
\hat{\mathbf{A}} + 10\bar{\mathbf{p}} = [0,0,2]^\top + 10[0,1,1]^\top = [0,10,12]^\top &= \hat{\mathbf{p}} \\
\hat{\mathbf{D}} + 10\bar{\mathbf{q}} = [0,2,0]^\top + 10[0,1,1]^\top = [0,12,10]^\top &= \hat{\mathbf{q}}
\end{aligned}$$

Above we saw that the optimal vertices **B**, **C**, and **D** lie in the same plane; because the rays **u** and **v** are also optimal and the optimal set is convex, they and all of their convex combinations must lie in that plane too, so the optimal face of the feasible set (cut off at $t = 10$) is outlined by the sequence of points $\hat{\mathbf{B}}$, $\hat{\mathbf{C}}$, $\hat{\mathbf{D}}$, $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, $\hat{\mathbf{B}}$. The tableaus **A**, **B**, and **C** are adjacent, so those vertices must also be adjacent and lie in the same plane; no other tableau is adjacent to more than one of them, so the triangle outlined by $\hat{\mathbf{B}}$, $\hat{\mathbf{C}}$, $\hat{\mathbf{A}}$, $\hat{\mathbf{B}}$ is another face of the feasible set. The rays **p** and **q** are feasible, and the feasible set is convex, so all convex combinations of **p** and **q** are also feasible; thus another face of the feasible set must be outlined by the points $\hat{\mathbf{A}}$, $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$, $\hat{\mathbf{D}}$, $\hat{\mathbf{C}}$, $\hat{\mathbf{A}}$. These faces partially bound a solid figure which, because the feasible set is convex, must be completed by two other faces.

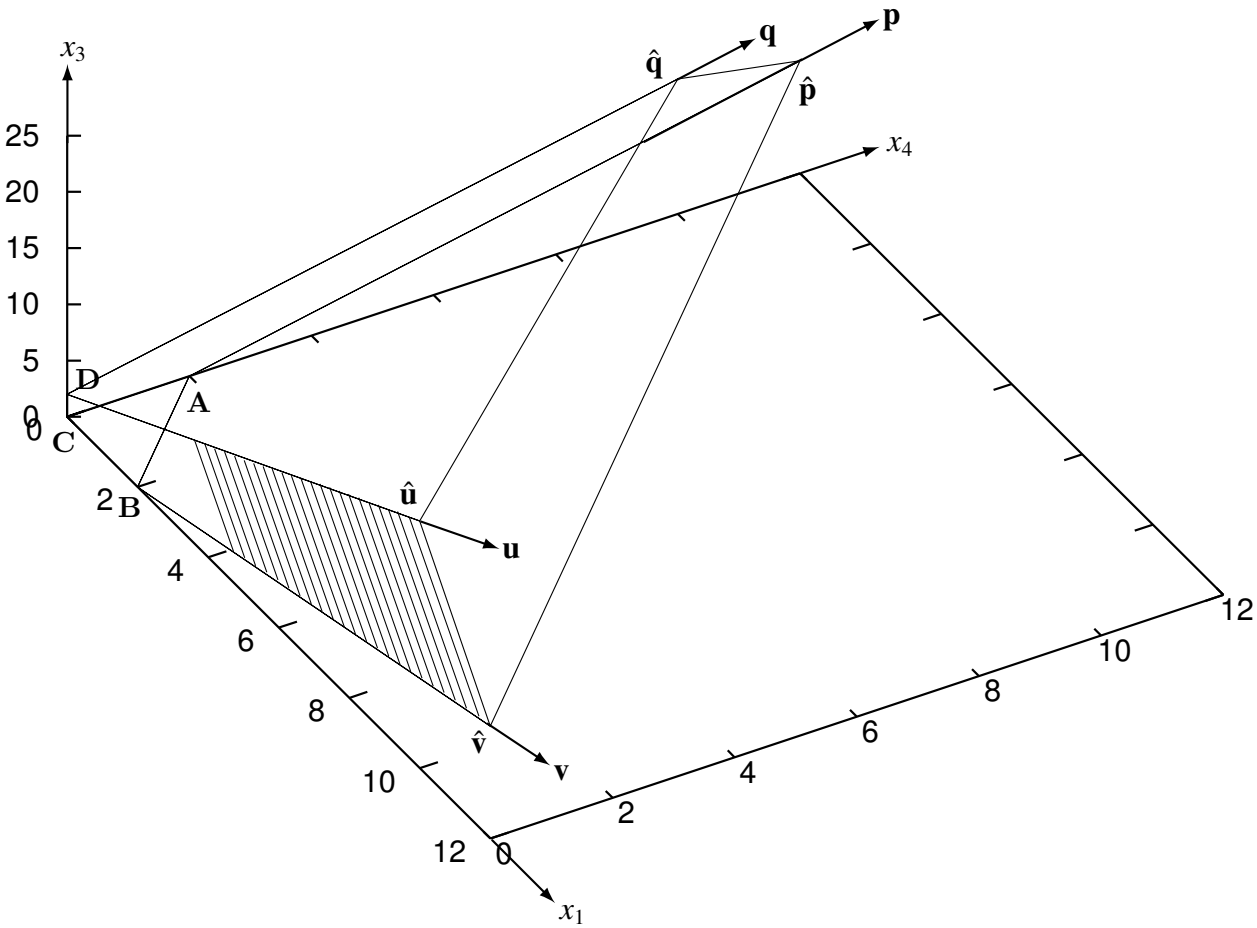
To specify the numerical coordinates of the points for plotting it is necessary to decide in what order to give them. Using the order (x_1, x_4, x_3) orients the axes in such a way that the optimal face is in front, with none of it hidden by other faces of the feasible set, so that is the order I used in the data file listed below on the left.

```
# this is file rays.dat
# front (optimal) face
2 0 0    # B
0 0 0    # C
0 0 2    # D
10 0 22  # u
12 0 10  # v
2 0 0    # B

# bottom face in x1-x4 plane
2 0 0    # B
0 0 0    # C
0 2 0    # A
2 0 0    # B

# back face
0 2 0    # A
0 12 10  # p
0 10 12  # q
0 0 2    # D
0 0 0    # C
0 2 0    # A

# top face
0 0 2    # D
0 10 12  # q
10 0 22  # u
0 0 2    # D

# bottom face tilted up
2 0 0    # B
12 0 10  # v
0 12 10  # p
0 2 0    # A
2 0 0    # B
```

```
# this is file rays.gnu
set xrange [0:12]
set yrange [0:12]
set zrange [0:25]
set view 30,60
set xyplane at 0
set nokey
set terminal postscript eps
set output "rays.eps"
splot "rays.dat" with lines
```

```
unix[1] echo 'load "rays.gnu"' | gnuplot
```

The `gnuplot` input file listed above on the right configures the plot, and when it is loaded as shown in the Unix command the program produces the picture on the next page (except for the annotations, which I added later).



Part of the optimal face (which should be seen as vertical and above the $x_1 - x_4$ plane) is crosshatched. The boundary of the feasible set can be seen to resemble a sheet-metal air duct that flares out from the origin and has a trapezoidal cross section (of course its interior points are also feasible).

3.6.2 Finding All Extreme Points

It is an article of faith in operations research that linear programming is an aid to decision making, but [151, §1.3] many an analyst has heard an executive say something that, when translated from business jargon into optimization jargon, decodes like this:

> "The course of action you recommend, while optimal in a technical sense, would be inconvenient to actually follow in this particular case. I want more options, so that I can pick one based on factors that are too subjective to be included in your mathematical model. Are there other production programs that are almost as good as the one you found?"

Geometrically, this question is about how much the objective function changes as we pivot from the optimal vertex of the feasible set to each adjacent vertex, then from each of those to its neighboring vertices, and so on. Exploring the feasible set in this way might yield useful insights about its geometry even if it has too many dimensions to picture. For example, if there are several vertices that are only slightly suboptimal but differ quite a bit in their coordinates, then the surface of the jewel must be relatively flat near $\mathbf{x}^\star$.

Analytically, the question is about finding the second-best basic feasible solution, or the third-best, or the hundredth-best. Why not enumerate *all* vertices of the feasible set, in order of increasing objective value? Then we could provide our decision-maker with an exhaustive list of suboptimal alternatives. We would also systematically find all of the optimal basic solutions if there are several (which we worried about in §3.6.1).

To see how such an enumeration is possible, consider the following optimal tableau, which solves the `brewery` problem of §1.3.1.

	x_1	x_2	x_3	x_4	s_1	s_2	s_3
2325.0	0	0	18.750	76.250	7.50	0	18.750
5.0	1	0	2.750	2.250	0.50	0	-1.250
7.5	0	0	1.625	-0.125	0.25	1	-1.375
12.5	0	1	-1.125	-0.375	-0.25	0	0.875

To find the next-best basic feasible solution we must pivot away from optimality while staying feasible and while increasing the objective (decreasing the $(1,1)$ entry of the tableau) as little as possible. In which nonbasic column does the minimum-ratio pivot increase the objective the least?

In the x_3 column the minimum-ratio pivot is at $a_{13} = 2.750$, and that would increase the objective by $(c_3/a_{13})b_1 = 34\frac{1}{11}$. In the x_4 column the only possible pivot is on the 2.250, and that would increase the objective by $169\frac{4}{9}$. In the s_1 column the minimum-ratio pivot is on the 0.50, and that would increase the objective by 75. In the s_3 column the only possible pivot is on the 0.875, and that would increase the objective by $267\frac{6}{7}$. Thus it is the pivot at a_{13} that yields the next-best tableau, which we could then analyze in the same way to find the next-best one after that.

The MATLAB program on the next page automates this process, generating all of the basic feasible solutions in objective-value order. Its first stanza could be modified to read the starting data from a file, and then it could be used for any problem.

The output of the program, which is shown on the page after the listing, reveals that the Brewery Problem has 6 basic feasible solutions. The dimension of the feasible set is 4 so we can't graph it, but from the objective values we can see that there are two production programs "almost as good" as the optimal one. The second-best vertex, at $\mathbf{x}$ = `[0,14.55,1.82,0]`$^\top$, has z = `-2290.91` which is within 2% of the optimal value, and the third-best vertex, which is at $\mathbf{x}$ = `[0,15,0,0]`$^\top$, has z = `-2250` which is within 4% of the optimal value. The next alternative is within 7% of optimal, but the others are much worse.


```
% subopt.m: list all basic feasible solutions in objective order

% define the problem
T=[2325.0,0,0,18.750,76.250, 7.50,0,18.750;  % optimal tableau
      5.0,1,0, 2.750, 2.250, 0.50,0,-1.250;
      7.5,0,0, 1.625,-0.125, 0.25,1,-1.375;
     12.5,0,1,-1.125,-0.375,-0.25,0, 0.875];
S=[2,4,0,0,0,3,0];                          % its basic sequence
n=7;                                        % number of variables
m=3;                                        % number of functional constraints

maxpiv=factorial(n)/factorial(n-m)-1;       % only maxpiv pivots are possible
for npiv=1:maxpiv                           % so do no more than that
    T                                       % report the current tableau

    pos=0;                                  % count
    for j=2:n+1                             % the
        if(T(1,j) > 0); pos=pos+1; end      % positive
    end                                     % costs
    if(pos == 0); break; end                % pivot only until there are none left

%   find the next pivot away from optimality
    dzmin=realmax;
    jzmin=0;
    izmin=0;
    for j=1:n                               % examine each variable column
        if(S(j) == 0 && T(1,1+j) > 0)       % try nonbasic columns with positive cost

           rmin=realmax;
           for i=1:m                        % find
               if(T(1+i,1+j) > 0)           % the
                  r=T(i+1,1)/T(1+i,1+j);    % minimum
                  if(r < rmin)              % ratio
                     rmin=r;                % pivot
                     imin=i;                % position
                  end                       % in
               end                          % this
           end                              % column

%          pivoting there would increase the objective by this much
           dz=T(1+imin,1)*T(1,1+j)/T(1+imin,1+j);

           if(dz < dzmin)                   % we want
              dzmin=dz;                     % to change
              jzmin=j;                      % the objective
              izmin=imin;                   % as little
           end                              % as possible
        end
    end

%   perform the pivot yielding smallest dz
    mp=m+1;                                 % number of rows in tableau
    np=n+1;                                 % number of columns in tableau
    ip=izmin+1;                             % tableau row of pivot
    jp=jzmin+1;                             % tableau column of pivot
    [Tnew,Snew,rc]=pivot(T,mp,np,ip,jp,S);  % perform the pivot
    if(rc ~= 0); exit; end                  % quit if pivot failed
    T=Tnew;                                 % update the tableau
    S=Snew;                                 % update the basic sequence
end
```



```
octave:1> format bank
octave:2> subopt
T =

  2325.00     0.00     0.00    18.75    76.25     7.50     0.00    18.75
     5.00     1.00     0.00     2.75     2.25     0.50     0.00    -1.25
     7.50     0.00     0.00     1.62    -0.12     0.25     1.00    -1.38
    12.50     0.00     1.00    -1.12    -0.38    -0.25     0.00     0.88

T =

  2290.91    -6.82     0.00     0.00    60.91     4.09     0.00    27.27
     1.82     0.36     0.00     1.00     0.82     0.18     0.00    -0.45
     4.55    -0.59     0.00     0.00    -1.45    -0.05     1.00    -0.64
    14.55     0.41     1.00     0.00     0.55    -0.05     0.00     0.36

T =

  2250.00   -15.00     0.00   -22.50    42.50     0.00     0.00    37.50
    10.00     2.00     0.00     5.50     4.50     1.00     0.00    -2.50
     5.00    -0.50     0.00     0.25    -1.25     0.00     1.00    -0.75
    15.00     0.50     1.00     0.25     0.75     0.00     0.00     0.25

T =

  2155.56   -33.89     0.00   -74.44     0.00    -9.44     0.00    61.11
     2.22     0.44     0.00     1.22     1.00     0.22     0.00    -0.56
     7.78     0.06     0.00     1.78     0.00     0.28     1.00    -1.44
    13.33     0.17     1.00    -0.67     0.00    -0.17     0.00     0.67

T =

  933.33  -49.17  -91.67  -13.33    0.00    5.83    0.00    0.00
   13.33    0.58    0.83    0.67    1.00    0.08    0.00    0.00
   36.67    0.42    2.17    0.33    0.00   -0.08    1.00    0.00
   20.00    0.25    1.50   -1.00    0.00   -0.25    0.00    1.00

T =

    0.00  -90.00  -150.00  -60.00  -70.00    0.00    0.00    0.00
  160.00    7.00   10.00    8.00   12.00    1.00    0.00    0.00
   50.00    1.00    3.00    1.00    1.00    0.00    1.00    0.00
   60.00    2.00    4.00    1.00    3.00    0.00    0.00    1.00

octave:3> quit
```

Notice that the final and most-suboptimal tableau discovered by the program is the initial canonical form for the problem.

Linear programs typically encountered in practice have basic feasible solutions whose number grows very fast with problem size, so in studying a realistic application it might not be practical to rank-order all of them. But for a large problem most of the basic feasible solutions will be too suboptimal to be of interest anyway, and it might still be useful to generate the first few nearly-optimal ones.

In §5.4 we will take up sensitivity analysis, which is useful for answering other questions about a linear programming model. Some of the techniques we study there will also involve pivoting from an optimal tableau to a suboptimal one.

3.7 Exercises

3.7.1 [E] Explain one insight about linear programming in general that you have gained from our study of low-dimensional examples in this Chapter.

3.7.2 [E] What halfspaces are associated with the constraint $4x_1 - 3x_2 + 5x_3 \leq 9$? What is the constraint's associated hyperplane? To which halfspace does the hyperplane belong?

3.7.3 [E] Each constraint hyperplane of a linear program divides $\mathbb{R}^n$ into two halfspaces, one feasible and the other infeasible. Together the constraint hyperplanes divide $\mathbb{R}^n$ into disjoint regions. The feasible set is the region that is the intersection of all the feasible halfspaces. In the example of §3.1, pick a region that is *not* the feasible set and explain how it is also the intersection of halfspaces.

3.7.4 [E] When is a vertex an extreme point? How many vertices can belong to an edge? Is the boundary of a feasible set always the union of its edges?

3.7.5 [H] In the `graph` problem of §3.1, the point $[1, 0]^\top$ is the midpoint of the edge $[\mathbf{A}, \mathbf{B}]$, and it is also the midpoint of other line segments in $\mathbb{X}$. Describe the set $\mathbb{L}$ of all line segments in $\mathbb{X}$ of which $[1, 0]^\top$ is the midpoint.

3.7.6 [E] The tableaus of a linear program correspond to vertices in its graph. What is necessary for a tableau to correspond to a given vertex?

3.7.7 [H] In the Guided Tour of §3.2.2 each basic feasible solution corresponds to one extreme point of the feasible set. (a) Could a linear program ever have a basic feasible solution that corresponded to some point other than an extreme point of its feasible set? Could a linear program ever have a feasible set with an extreme point that did not correspond to one of its basic feasible solutions? Make a convincing argument based on what you know about the geometry of the simplex algorithm. (b) Use linear algebra to construct a formal proof that every basic feasible solution of any canonical form linear program is an extreme point of the feasible set defined by $\{\mathbf{x} \mid \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$.

3.7.8 [E] In §3.2.1 we saw how, as t is increased from 0 to the minimum ratio for a pivot, the point represented by a tableau slides from one vertex to another along a constraint hyperplane. What happens to the objective value z as this is happening? For the example of that Section, derive an expression for $z(t)$ and confirm that $z(0)$ is the objective value at vertex $\mathbf{A}$ and $z(2)$ is the objective value at vertex $\mathbf{B}$.

3.7.9 [E] When there is a tie for the minimum ratio in pivoting from a given tableau $\mathbf{T}_1$ to a next tableau $\mathbf{T}_2$, what does the resulting pattern of entries in some row of $\mathbf{T}_2$ signal about the linear program? Which row of $\mathbf{T}_2$ is it that shows this?

3.7.10 [E] What makes a vertex degenerate? What makes a pivot degenerate? Does a degenerate pivot always result in a decrease in the objective function? Does a pivot always move the solution from one vertex to an adjacent vertex? Explain.

3.7.11 [E] Is a pivot by the simplex rule ever an exterior pivot? Explain. Does a pivot by the simplex rule always move from one extreme point to an adjacent extreme point?

3.7.12 [E] If two tableaus are the same except that their constraint rows are permuted, do they have the same basic sequence? Do they have the same basic variables?

3.7.13 [H] If we use the simplex algorithm to solve a linear program that has an optimal solution, does choosing each pivot column as one with the most negative cost always lead to optimal form in the fewest pivots? If yes, explain why; if no, provide a counterexample.

3.7.14 [E] In the example of §3.1 there are two paths from vertex **A** to the optimal point at vertex **D**. (a) If a linear program has a feasible set of dimension 2, can there ever be *more* than two paths from a starting point to the optimal point? (b) If a linear program has a feasible set of dimension 3, how many paths might there be from a starting point to the optimal point? In answering this question it might be helpful to imagine what a convex polyhedron looks like in $\mathbb{R}^3$.

3.7.15 [E] In §3.3 I claimed that because a vertex can be viewed as the intersection of n hyperplanes on which a variable is zero, it is possible to move to any vertex by pivoting to make those variables zero. Use this advice to pivot from **A** to **K** in the example.

3.7.16 [H] In §3.3.1 we read the values of the basic variables s_1, s_2, s_3, and s_4 from tableau **A** and then were able to find them in the graph, which shows the view from that tableau. (a) Read the values of the basic variables from tableau **B** and find those values in the view from tableau **B**. (b) Can you find the values of the tableau **B** basic variables in the view from tableau **A**?

3.7.17 [E] In the view of §3.3.2, can you read off the values of the slack variables x_1, x_2, x_3, and x_4 from the graph? If yes, what are their values? If not, why not?

3.7.18 [H] Draw views of the §3.3.2 example from tableau (a) **B**; (b) **C**; (c) **E**.

3.7.19 [H] From the following tableau draw a view of the linear program and solve the problem graphically. What is the dimension of the feasible set?

	x_1	x_2	x_3	x_4	x_5
0	0	1	−1	1	0
10	1	1	1	1	0
5	0	1	1	$\frac{1}{5}$	1

3.7.20 [E] If a linear program has a feasible ray, can it have a finite optimal value? If it has an unbounded optimal value, can it have a feasible ray? If a tableau has a column whose a_{ij} indicate a ray, what is sufficient to ensure that the linear program has an unbounded optimal value?

3.7.21 [E] If the unique optimal vertex of a linear program is degenerate, does the linear program have multiple optima? If the objective function contours of a linear program are parallel to a constraint hyperplane, does the linear program have multiple optimal solutions?

3.7.22 [H] Construct a linear program having an optimal-form tableau in which $c_j = 0$ over a nonbasic column but there is only one optimal point.

3.7.23 [E] Does a ray include the point from which it emanates? If a tableau has a column whose a_{ij} indicate a ray, what is sufficient to ensure that the ray is optimal?

3.7.24 [E] Can all of the points on an optimal edge be found by pivoting? In the example of §3.4.2, show how to find the optimal point $[\frac{3}{2}, \frac{1}{2}]^\top$.

3.7.25 [E] Describe the sign pattern of entries in a canonical-form tableau that indicates the linear program has the following properties: (a) infeasible form 1; (b) infeasible form 2; (c) degeneracy; (d) suboptimality; (e) optimal form; (f) multiple bounded optimal solutions; (g) an optimal ray; (h) a non-optimal feasible ray; (i) unbounded form.

3.7.26 [H] Prove that the intersection of two convex sets is a convex set.

3.7.27 [E] Sketch the convex hull of the feasible set $\mathbb{N}$ in the example of §3.5. Is the set $\mathbb{N}$ the intersection of halfspaces?

3.7.28 [H] How does the proof of §3.5.2 *fail* if $\mathbb{X}$ is not known to be convex?

3.7.29 [E] A linear program can have more than one optimal vertex. What other points might belong to the optimal set?

3.7.30 [E] If a linear program has two optimal vertices, why must the tableaus corresponding to them be adjacent tableaus?

3.7.31 [H] The convex hull of an equilateral triangle is the triangle itself. Write a formula for the convex combination of the triangle's vertices and show that by adjusting the parameters the formula can produce any point in the triangle (and no points outside of it).

3.7.32 [E] In the example of §3.6.1, three edges of the feasible set are incident to vertex **D**. Explain how this can be discovered by inspecting tableau **D**.

3.7.33 [H] Draw a view of the example in §3.6.1 from tableau **A**, and use it to solve the problem graphically.

3.7.34 [P] Modify the MATLAB program of §3.6.2 to find all of the basic feasible solutions to the example of §3.1.

3.7.35 [P] Our statement of the simplex pivot rule in §2.4.4 just says to pick a pivot column with a negative c_j. In practice we have usually chosen the most negative c_j, but computer implementations sometimes use the first negative c_j or the most negative c_j from a **candidate list** of the first p columns having $c_j < 0$. It is also possible to select the pivot column as one (or one from a candidate list) whose minimum-ratio pivot yields the biggest decrease in z. More work is required to select the pivot column in this way, but if the greatest possible decrease in z is achieved at each iteration it might be possible to reach optimal form with fewer pivots. (a) Modify the MATLAB code given in §4.1 to use this "best-z" strategy, and test it on some examples. (b) Add code to count the numbers of arithmetic operations used,

and compare the total numbers required by this strategy to the total numbers required by the strategy of picking the first negative c_j. (c) Why do you think this Exercise is located in this Chapter rather than in §4?

3.7.36 [H] In the `graph` problem of §3.1, the constraint $x_2 \leq 5$ is redundant because it could be removed without changing the feasible set. (a) When the procedure outlined in §2.8.1 for pivoting-in a basis is applied to this problem, does it discover the redundant constraint? (b) Does the method of artificial variables outlined in §2.8.2 discover the redundant constraint? (c) How can we ensure that the feasible set of a linear programming problem will have no degenerate vertices?

4

Solving Linear Programs

The process outlined in §2.6 for solving a linear program consists of reformulation into standard form, putting the resulting tableau into canonical form by the subproblem technique or the method of artificial variables, and pivoting by the simplex rule until one of the final forms is obtained. Reformulation or phase 0 is, as we saw in §2.9, essentially algebraic and thus not easily automated. In contrast the simplex algorithm, which transforms a standard-form tableau into canonical form via phase 1 and then into a final form via phase 2, is essentially numerical, and to be practical it *must* be automated. This Chapter is about using the simplex method to solve real problems.

4.1 Implementing the Simplex Algorithm

As illustrated in §2.9.3 and §2.9.5 the `pivot` program's `SOLVE` command can be used to perform the simplex algorithm, but sometimes we will wish to solve a linear program as part of a larger calculation and then it will be convenient to have an implementation in MATLAB. The code presented in this Section combines ideas that were introduced in §2.4 and §2.8.1 and identifies infeasible and unbounded as well as optimal form, so its details illuminate the whole algorithm.

The top-level routine of this implementation is `simplex.m`, listed on the next page. It receives [1] the tableau `T` of a standard-form problem, the number of equality constraints `m` and the number of variables `n`, and returns [1] the solution vector `xstar`, the final tableau `Tnew`, and a return code `rc` whose value signals success if `rc=0`, infeasibility if `rc=1`, or unboundedness in column `rc` of the tableau if `rc>1`.

The vector `tr` contains the indices in the tableau of the `mr` rows that make up the problem. To begin [6-9] all of the rows are included, but if pivoting-in a basis reveals a row to be redundant the list will be modified to exclude that row.

This routine invokes [11] `newseq.m` to pivot-in an identity, [19] `phase1.m` to implement the subproblem technique, and [28] `phase2.m` to obtain a final form. If infeasible form is discovered by `newseq` [12] or `phase1` [20] this routine returns with [13,21] `rc=1`. If unbounded form is discovered by `phase2` [29] this routine returns [30-31] in `rc` the number of the tableau column that reveals the unboundedness. Otherwise [32-41] the basic solution is extracted from the optimal-form tableau and returned in `xstar`. Here, as in the `pivot.m` routine of §2.4.2, each element of the basic-sequence vector `S` or `Snew` corresponds to a variable column of the tableau and contains `0` if that variable is nonbasic or the row number in the tableau of the identity-column `1` if the variable is basic.


```
 1 function [xstar,rc,Tnew]=simplex(T,m,n)
 2 %  solve a linear program in standard form
 3
 4    nn=n+1;        % tableau columns = variables+1
 5    mm=m+1;        % tableau rows = constraints+1
 6    for ii=1:mm    % to start include them all
 7        tr(ii)=ii; % in the list of rows
 8    end            % that are in the problem
 9    mr=mm;         % there are mr of those
10
11    [Tnew,S,trnew,mrnew,rc0]=newseq(T,mm,nn,tr,mr); % get identity
12    if(rc0 != 0)   % on failure
13       rc=1;       % report infeasible
14       return      % and give up
15    else           % otherwise
16       T=Tnew;     % update the tableau
17    end            % and continue
18
19    [Tnew,Snew,rc1]=phase1(T,S,mm,nn,tr,mr); % get b nonnegative
20    if(rc1 != 0)   % on failure
21       rc=1;       % report infeasible
22       return      % and give up
23    else           % otherwise
24       T=Tnew;     % update the tableau
25       S=Snew;     % update the basic sequence
26    end            % and continue
27
28    [Tnew,Snew,rc2]=phase2(T,S,mm,nn,tr,mr); % get c nonnegative
29    if(rc2 != 0)               % on failure
30       rc=rc2;                 % report unbounded form in column rc
31       return                  % and give up
32    else                       % otherwise
33       rc=0;                   % report optimal form
34       for j=1:n               % for each j
35           ii=Snew(j);         % find the row of the basic column 1
36           if(ii == 0)         % if this column is nonbasic
37              xstar(j)=0;      % return zero
38           else                    % otherwise
39              xstar(j)=Tnew(ii,1); % return the basic variable value
40           end                     % finished retrieving this variable
41       end                     % finished constructing x* vector
42    end                        % end of simplex algorithm
43 end
```

The `newseq.m` routine, listed on the next page, receives [1] a tableau `T` having `mm` rows and `nn` columns, the list `tr` of tableau rows in the problem, and the number `mr` of tableau rows in the problem. To begin [6-8] it initializes the basic sequence vector `S` to zeros, which marks all of the variable columns as nonbasic. Then [11-40] it considers the constraint rows one at a time. First it [14-20] searches the row for an entry that is big enough to pivot on, and if it finds one [17-18] it remembers the column number `jp` and [23] pivots there using the `pivot.m` routine. The tableau and its basic sequence are updated [24-25] and [26] the next row is considered. This process continues until a pivot has been performed in each constraint row so that a basis is present. Then [42-43] the list and number of active rows are updated and the routine returns [1] the updated tableau `Tnew`, the basic sequence `S`, the new row list `trnew` and count `mrnew`, and `rc0=0` [5] to signal that a basis is present.


```
 1 function [Tnew,S,trnew,mrnew,rc0]=newseq(T,mm,nn,tr,mr)
 2 %  get the identity columns with zero costs above
 3
 4    ztol=1e-6;       % set zero tolerance
 5    rc0=0;           % assume this routine will succeed
 6    for j=2:nn       % start
 7        S(j-1)=0;    % with
 8    end              % no basis
 9    ir=1;            % point to the objective row
10
11    while(ir < mr)   % are any constraint rows left to consider?
12      ir=ir+1;       % yes; advance to the next one
13      ip=tr(ir);                      % in row ip
14        jp=0;                         % find
15        for jj=2:nn                   % the first
16            if(abs(T(ip,jj)) > ztol) % nonzero entry
17               jp=jj;                 % at column jp
18               break                  % and use it
19            end                       % if not yet
20        end                           % keep looking
21
22        if(jp > 0)                    % if we found a nonzero entry
23           [Tnew,Snew,rc]=pivot(T,mm,nn,ip,jp,S); % pivot on it
24           T=Tnew;                    % update the tableau
25           S=Snew;                    % and the basic sequence
26           continue                   % go to do the next row
27        end                           % otherwise fall through
28
29        if(abs(T(ip,1)) <= ztol)      % A row is zero; check the b
30           for iir=ir:mr-1            % this tableau row is redundant
31               tr(iir)=tr(iir+1);     % copy the row pointers up
32           end                        % to squeeze out redundant row
33           tr(mr)=0;                  % zero last pointer now repeated
34           mr=mr-1;                   % one less row in the problem
35           ir=ir-1;                   % account for the deletion
36        else         % we have discovered infeasible form 1
37           rc0=1;    % set the return code to indicate that
38           break     % and return
39        end          % finished processing the zero A row
40    end              % finished with constraint rows in the problem
41
42    trnew=tr;        % return updated list of active rows
43    mrnew=mr;        % and updated number of active rows
44 end
```

If some row of the tableau has zeros in its **A** part, then the search for a pivot position [15-20] leaves `jp=0` [14]. The second stanza in the `while` loop [22-27] is skipped, and the last stanza [29-39] is executed instead. It checks [29] whether $|b_{\mathtt{ip}}| \approx 0$. If it is, then to remove the redundant constraint from the problem the indices of the remaining constraint rows (if `ir < mr` so that there are any) are copied up [30-32]; the index of the last row, now unused, is set to zero [33]; and the number of rows in the problem is reduced by one [34]. So that `ir` will point to the next constraint row after it is incremented [12], it is [35] reduced by one here. If $b_{\mathtt{ip}} \neq 0$ then [37] the return code is set to show infeasible form 1 and [38] the `while` loop is interrupted. The list [42] and number [43] of active rows are updated, and the routine returns with `rc0=1` to signal that no basis is present. I used the MATLAB `while` construct instead of a `for` loop (see §28.4.1) because both `ir` and `mr` are changed inside it.


```
 1 function [Tnew,Snew,rc1]=phase1(T,S,mm,nn,tr,mr)
 2 % get constant column nonnegative, or find problem infeasible
 3
 4   ztol=1e-6;                        % set zero tolerance
 5   Tnew=T;                           % return T on failure
 6   Snew=S;                           % return S on failure
 7
 8   ii=0;                             % assume every b is negative
 9   for ir=2:mr                       % search the constant column
10       ic=tr(ir);                    % constraint rows in the problem
11       if(T(ic,1) >= 0)              % is this b nonnegative?
12          ii=ic;                     % yes; remember the tableau row
13          break                      % we found one
14       end                           % so stop
15   end                               % searching
16
17   if(ii == 0)                       % every b is negative
18      jp=0;                          % search
19      for jj=2:nn                    % the first A row
20          if(T(tr(2),jj) < 0)        % for a negative entry
21             jp=jj;                  % and remember where it was
22             break                   % found one
23          end                        % finished testing
24      end                            % finished searching row
25      if(jp == 0)                    % if no A row entry is negative
26         rc1=2;                      % signal infeasible form 2
27         return                      % and give up
28      else                                          % otherwise
29         [Tnew,Snew,rc]=pivot(T,mm,nn,tr(2),jp,S); % pivot there
30         T=Tnew;                                   % update T
31         S=Snew;                     % update S
32      end                            % pivot made b1 nonnegative
33   end                               % now ready for subproblems
```

The `phase1.m` routine, listed above and on the next page, implements the subproblem technique. The method of artificial variables could be used instead of `newseq.m` and `phase1.m` (see Exercise 4.6.12) but this code is brief and requires no additional array storage. The routine begins by 8-15 finding a nonnegative b_i. If there are none it 18-24 finds a negative entry in the first constraint row and 28-32 pivots on it to make $b_1 > 0$. If there is no negative entry in the row it 25-27 sets `rc1=2` to indicate infeasible form 2 and resigns.

When there is at least one nonnegative b_i, subproblems are solved 35-73 to make the others nonnegative. There are `mm-1` constraint rows in the tableau, so the process of searching for a negative b_i and solving a subproblem to make it nonnegative will be repeated no more than 35 that number of times. The process begins by 36-48 constructing the next subproblem. The vector `sr` will list the `ms` tableau rows included in the subproblem, starting with the subproblem objective. To begin 36-37 the code sets `sr(1)=0` to show that no subproblem objective has been found yet. Then 38-48 it examines each constraint row in the problem, makes the first one with a negative b_i 40-43 the subproblem objective, and makes all of the rows with nonnegative b_i 44-47 the subproblem constraints. If no negative b_i remain 49-52 it returns `rc1=0` to show that canonical form has been achieved.

The `phase2.m` routine is invoked 54 to solve the subproblem, which is sure to be in canonical form, and 55-56 the tableau and basic sequence are updated. If the optimal


```
35   for p=1:mm-1                      % need no more than m subprobs
36       ms=1;                         % construct the next subproblem
37       sr(ms)=0;                     % row ms is to be selected
38       for ir=2:mr                   % search constraint rows
39           ii=tr(ir);                % that are in the problem
40           if(T(ii,1) < 0)           % for a negative b
41              if(sr(1) == 0)         % if it is the first
42                 sr(1)=ii;           % make it the subprob obj
43              end;                   % finished making subprob obj
44           else                      % this b is nonnegative
45              ms=ms+1;               % enlarge the subproblem
46              sr(ms)=ii;             % and add this row to it
47           end                       % done testing this row
48       end                           % done constructing subproblem
49       if(sr(1) == 0)                % if no subproblem objective
50          rc1=0;                     % signal canonical form
51          return                     % and return
52       end                           % done testing completion
53
54       [Tnew,Snew,rc2]=phase2(T,S,mm,nn,sr,ms); % solve subproblem
55       T=Tnew;                                  % update T
56       S=Snew;                       % update S
57       if(abs(T(sr(1),1)) < ztol)    % if final b is tiny
58          T(sr(1),1)=0;              % make it zero exactly
59       end                           % finished checking b
60
61       if(rc2 == 0 && T(sr(1),1) < 0) % if final b is negative
62          rc1=2;                       % mark infeasible form 2
63          return                     % and give up
64       end                           % finished with infeasible
65
66       if(rc2 > 0 && T(sr(1),1) < 0) % if subproblem unbounded
67          jp=rc2;                     % pivot in unbounded column
68          ip=sr(1);                              % in objective
69          [Tnew,Snew,rc]=pivot(T,mm,nn,ip,jp,S); % do the pivot
70          T=Tnew;                                % update T
71          S=Snew;                    % update S
72       end                           % finished with unbounded
73   end                               % finished with subproblems
74   rc1=0;                            % signal success
75 end
```

subproblem objective value is small enough that it might be numerical noise [57–59] it is set to zero; this prevents roundoff errors from making a feasible problem appear infeasible. If `phase2.m` reports success but the optimal subproblem objective value is negative [61] the routine [62–63] sets `rc1=2` to indicate infeasible form 2 and resigns. If [66] the subproblem is unbounded and its objective is still negative [67–71] a pivot is performed in column `rc2` of the subproblem objective row. Then [73] we repeat the process until $\mathbf{b} \geq \mathbf{0}$.

The `phase2.m` routine, listed on the next page, cannot require more than

$$n!/(n-m)!$$

iterations [8] to reach a final form (see §4.5). If this number is [9] greater than the highest integer allowed in the MATLAB range expression `1:kmax`, that integer `2147483645` is used for

```
 1 function [Tnew,Snew,rc2]=phase2(T,S,mm,nn,tr,mr)
 2 %  optimize a tableau in canonical form, or find it unbounded
 3
 4    ztol=1e-6;                      % set zero tolerance
 5    Tnew=T;                         % return T on failure
 6    Snew=S;                         % return S on failure
 7
 8    kmax=factorial(nn-1)/factorial(nn-mm); % theoretical maximum
 9    if(kmax > intmax-2) kmax=intmax-2; end % integer iteration limit
10
11    for k=1:kmax                    % do up to kmax pivots
12        cmin=0;                     % find
13        jp=0;                       % in
14        ii=tr(1);                   % the objective row
15        for jj=2:nn                 % the column
16            if(T(ii,jj) < cmin)     % with the lowest
17               cmin=T(ii,jj);       % negative cost entry
18               jp=jj;               % and remember the column number
19            end                     % finish testing cost entry
20        end                         % finish finding least cost entry
21
22        if(jp == 0 || cmin > -ztol) % no (sufficiently) negative cost
23           rc2=0;                   % signal optimal form
24           return                   % and return to the caller
25        end                         % finished testing for optimality
26
27        ip=minr(T,tr,mr,jp);        % find min ratio row in column jp
28        if(ip == 0)                 % if there is none
29           rc2=jp;                  % signal unbounded in column jp
30           return                   % and return to the caller
31        end                         % finished finding pivot row
32
33        [Tnew,Snew,rc]=pivot(T,mm,nn,ip,jp,S); % pivot at T(ip,jp)
34        T=Tnew;                                % update the tableau
35        S=Snew;                     % and the basic sequence
36    end                             % for the next iteration
37 end
```

`kmax` instead. Each iteration begins [12-20] by finding the variable column having the lowest adjusted cost. If [22] no negative costs remain, the routine [23] sets `rc2=0` to indicate convergence and [24] returns.

Once a pivot column is chosen [27] `minr.m` is invoked to find the minimum-ratio row. If no $a_{i,\mathtt{jp}}$ in the pivot column `jp` is positive, `minr.m` returns zero for the pivot row; then [28-31] this routine sets `rc2` to the index of the unbounded column and resigns. If a pivot row was found then [33] `pivot.m` is invoked to perform the pivot, the tableau and basic sequence are [34-35] updated, and [36] the iterations continue. If `kmax` iterations are performed without finding a final form the routine returns [37] the current tableau and basic sequence.

The `minr.m` routine, listed on the next page, starts by [6] setting `minr` = $+\infty$. Then [8-17] it examines the constraint rows of the pivot column `jp`, skipping [9-11] elements too small to be a pivot, in search of the positive one with the lowest value of [12] $b_i/a_{i,\mathtt{jp}}$. When a ratio is found that is [13] lower than the lowest one found previously, `rmin` is updated [14] along with [15] the corresponding tableau row. On return `ip` is the minimum ratio row or zero if the problem is unbounded.


```
function ip=minr(T,tr,mr,jp);
% find the minimum ratio row ip in pivot column jp of T

  ztol=1.e-6;                      % zero tolerance
  ip=0;                            % return index zero on failure
  rmin=realmax;                    % rmin = +infinity

  for ii=2:mr                      % check each constraint row
      if(T(tr(ii),jp) <= ztol)     % is this pivot negative or too small?
         continue                  % yes; skip it
      end                          % and continue down the column
      r=T(tr(ii),1)/T(tr(ii),jp);  % find this row ratio
      if(r < rmin)                 % is it lower than best so far?
         rmin=r;                   % yes; update best so far
         ip=tr(ii);                % and the row where it happens
      end                          % and continue
  end                              % until all rows are checked
end
```

In the Octave session below I used `simplex.m` to solve the `brewery` problem. The optimal tableau is the one we found in §2.4.3 except that the constraint rows are permuted.

```
octave:1> % brewery
octave:2> T=[  0,-90,-150,-60,-70,0,0,0;
>            160,  7,  10,  8, 12,1,0,0;
>             50,  1,   3,  1,  1,0,1,0;
>             60,  2    4,  1,  3,0,0,1];
octave:3> format bank
octave:4> [xstar,rc,Tstar]=simplex(T,3,7)
xstar =

   5.00  12.50   0.00   0.00   0.00   7.50   0.00

rc = 0.00
Tstar =

  2325.00     0.00     0.00    18.75    76.25     7.50     0.00    18.75
     5.00     1.00     0.00     2.75     2.25     0.50     0.00    -1.25
    12.50     0.00     1.00    -1.12    -0.37    -0.25     0.00     0.88
     7.50     0.00     0.00     1.62    -0.13     0.25     1.00    -1.37

octave:5> quit
```

You can confirm that `simplex.m` returns `rc=4` for the `unbd` problem of §2.5.2, which is unbounded in tableau column 4, and `rc=1` for the `infea` problem of §2.5.3, which is in both infeasible forms. Solving `sf1` shows that the routine leaves redundant rows in the tableau even though it ignores them in solving the problem.

4.2 The Revised Simplex Method

When we pivot by the simplex algorithm to solve a linear program, whether we do the calculations by hand or with a computer every element of each tableau gets filled in. To find the multipliers for the non-pivot rows we do m divisions. Then each of the $1 + n - m$

constant and nonbasic columns requires a multiplication in every row and a subtraction in the m non-pivot rows. Is all of this arithmetic really necessary?

In carrying out the algorithm there are two reasons why we need each tableau: to determine the position of the next pivot, and to find the elements of the tableau resulting from that pivot so that we can do it all again. It would be less work to compute only enough of the current tableau to determine the position of the next pivot, while keeping track of the pivots we have already done so that when a final form is reached we can extract the optimal point or report that there is none. This is the idea of the **revised simplex method** [3, §3.9] [145, §4.3].

4.2.1 Pivot Matrices

Pivoting in a tableau yields a new tableau. It is an interesting fact of matrix arithmetic that premultiplying the original tableau by an appropriate square matrix also yields the new tableau, as illustrated by the example below. I will call this linear program `pm` (see §28.5.13).

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline -3 & 0 & 1 & 0 & -2 \\ \hline 3 & 1 & 1 & 0 & 1 \\ 2 & 0 & -4 & 1 & 2 \end{array} = \begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline -1 & 0 & -3 & 1 & 0 \\ \hline 2 & 1 & 3 & -\frac{1}{2} & 0 \\ 1 & 0 & -2 & \frac{1}{2} & 1 \end{array}$$

$$\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline -1 & 0 & -3 & 1 & 0 \\ \hline 2 & 1 & 3 & -\frac{1}{2} & 0 \\ 1 & 0 & -2 & \frac{1}{2} & 1 \end{array}$$

same result tableau

In order for this to work, the last m columns of the **pivot matrix** must be the columns of the new tableau corresponding to the basic-sequence columns in the original tableau. Here $m = 2$, and in the original tableau the basic sequence is $S = (x_1, x_3)$. In the new tableau the x_1 column becomes the second column of the pivot matrix and the x_3 column becomes the third column of the pivot matrix. How this happens is more obvious if we consider the multiplication of the x_3 column in the original tableau by the pivot matrix.

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0\times 1 + 0\times 0 + 1\times 1 \\ 0\times 0 + 0\times 1 + 1\times -\frac{1}{2} \\ 0\times 0 + 0\times 0 + 1\times \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 1 \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

Because the x_3 column of the original tableau is a basis column, multiplying it by the pivot matrix copies out the pivot-matrix column corresponding to the row of the identity-column `1`. If the new x_1 and x_3 columns produced by the matrix multiplication equal those resulting from the pivot, then so do the others.

Recall from §2.3 that a pivot is a particular sequence of row operations. Performing those row operations on the $(m+1)\times(m+1)$ identity matrix yields a pivot matrix which, when it premultiplies an $(m+1)\times(n+1)$ tableau, performs that pivot in the tableau. To do the pivot circled in the example above, we divide row 3 by the pivot element 2. Then we subtract the new row 3 from row 2 to zero out the 1 in the pivot column, and add twice the new row 3 to row 1 to zero out the -2. Performing these operations on the 3×3 identity matrix we find

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \to r_3/2 \to \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} \to r_2 - r_3 \to \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix} \to r_1 + 2r_3 \to \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix}.$$

To find the pivot matrix that performs a given pivot, it is only necessary to "do to the identity whatever you would like to do to the tableau" [3, p75]. Because we never pivot in the objective row of a tableau, the first column of a pivot matrix is always the first identity column.

4.2.2 Not Doing Unnecessary Work

In the matrix multiplication of §4.2.1 we found all the elements of the result tableau, but only a few of them are needed to pick the next pivot position. That element is circled in the tableau $\mathbf{T}_1$ on the right below.

$$\underset{\mathbf{Q}_1}{\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix}} \underset{\mathbf{T}_0}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline -3 & 0 & 1 & 0 & -2 \\ \hline 3 & 1 & 1 & 0 & 1 \\ 2 & 0 & -4 & 1 & 2 \end{array}} = \underset{\mathbf{T}_1}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline & & -3 & & \\ \hline & & 3 & & \\ & & -2 & & \end{array}}$$

To find the simplex pivot column in $\mathbf{T}_1$ we need the objective function cost coefficients. Some of these we know without having to calculate them, because they are the zero costs of the new basis columns. The basic sequence of $\mathbf{T}_0$ is $S_0 = (x_1, x_3)$ and there we pivot in the second constraint row of the x_4 column, so we know even before performing the pivot that the basic sequence of $\mathbf{T}_1$ is going to be $S_1 = (x_1, x_4)$. In general if we pivot on a_{hp} then element h of S gets replaced by x_p. Thus $c_1 = c_4 = 0$ in $\mathbf{T}_1$ and we can begin checking cost entries with c_2. That is the dot product of the x_2 column in $\mathbf{T}_0$ with the first row of $\mathbf{Q}_1$, which turns out to be -3. If we are willing to use the *first* negative cost rather than the *most* negative cost (they are the same in this case) then we can take $p = 2$ as the pivot column.

To find the pivot row in $\mathbf{T}_1$ we need a_{12} and a_{22} and, if both are positive, the corresponding constant-column values b_1 and b_2 so that we can compare the ratios b_1/a_{12} and b_2/a_{22}. In this case there is only one positive constraint coefficient so that must be the pivot element and there is no need to find the minimum ratio.

To perform the pivot in $\mathbf{T}_1$ we would divide row 2 by 3, add three times the new row 2 to row 1, and add 2 times the new row 2 to row 3. Doing these things to the identity matrix we get

$$\mathbf{Q}_2 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix}.$$

To perform the marked pivot in $\mathbf{T}_1$ we can compute $\mathbf{T}_2 = \mathbf{Q}_2\mathbf{T}_1$, but we found $\mathbf{T}_1 = \mathbf{Q}_1\mathbf{T}_0$ so $\mathbf{T}_2 = \mathbf{Q}_2\,[\mathbf{Q}_1\mathbf{T}_0] = \mathbf{P}_2\mathbf{T}_0$ where

$$\mathbf{P}_2 = \mathbf{Q}_2\mathbf{Q}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \frac{1}{2} \\ 0 & \frac{1}{3} & -\frac{1}{6} \\ 0 & \frac{2}{3} & \frac{1}{6} \end{bmatrix}.$$

Performing the pivot circled in $\mathbf{T}_1$ will make $S_2 = (x_2, x_4)$. Using this information about the basic sequence of $\mathbf{T}_2$ and the pivot matrix $\mathbf{P}_2$ we can continue the solution process like this.

$$\underset{\mathbf{P}_2}{\begin{bmatrix} 1 & 1 & \frac{1}{2} \\ 0 & \frac{1}{3} & -\frac{1}{6} \\ 0 & \frac{2}{3} & \frac{1}{6} \end{bmatrix}} \underset{\mathbf{T}_0}{\begin{array}{c|ccccc} & x_1 & x_2 & x_3 & x_4 & \\ \hline -3 & 0 & 1 & 0 & -2 & \\ \hline 3 & 1 & 1 & 0 & 1 & \\ 2 & 0 & -4 & 1 & 2 & \end{array}} = \underset{\mathbf{T}_2}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline & & & \frac{1}{2} & \\ \hline \frac{2}{3} & & & & \\ \frac{7}{3} & & & & \end{array}}$$

We know without calculating them that c_2 and c_4 are zero in $\mathbf{T}_2$, because x_2 and x_4 are basic variables in S_2.

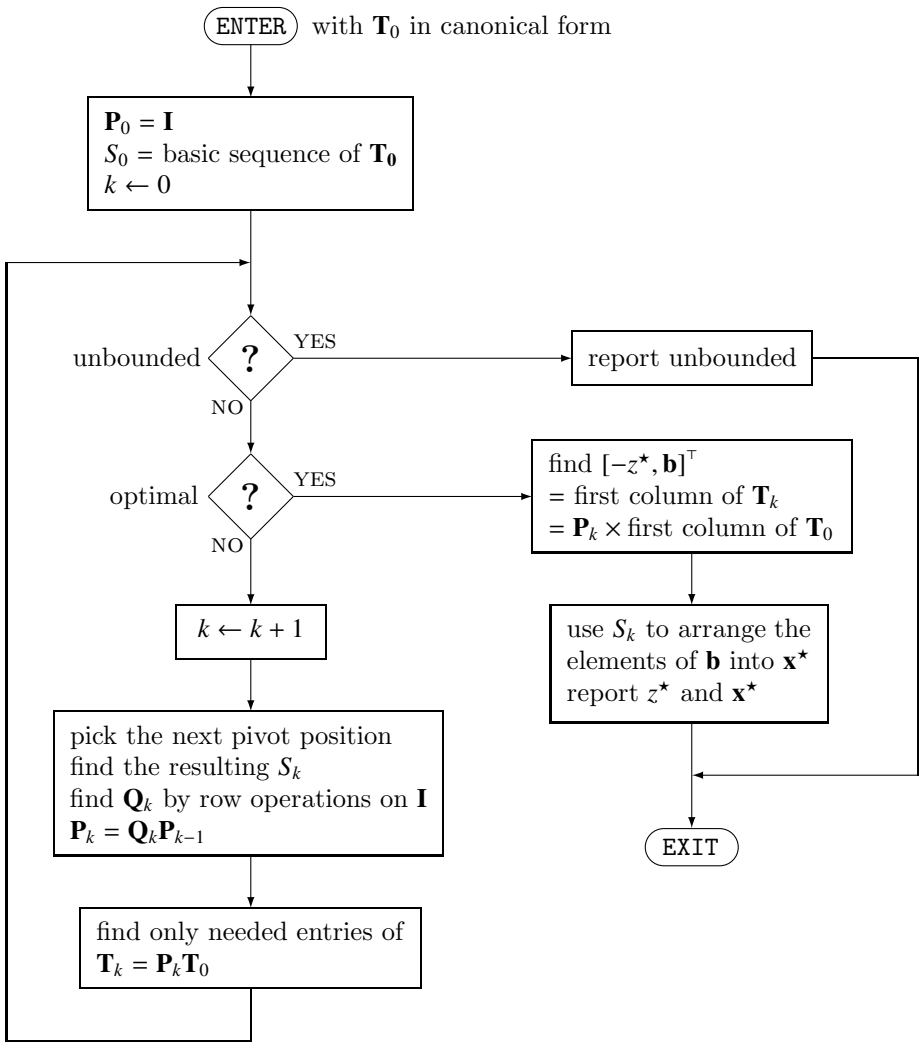
We also know without calculating it that $c_1 > 0$ in $\mathbf{T}_2$, because the x_1 column was basic in $\mathbf{T}_1$, where $S_1 = (x_1, x_4)$, and became nonbasic in $\mathbf{T}_2$, where $S_2 = (x_2, x_4)$. In $\mathbf{T}_1$ the basic x_1 column had a cost coefficient of zero, and its identity-column $\mathtt{1}$ must have been in the pivot row because that is how x_1 came to be nonbasic in $\mathbf{T}_2$. Every simplex-rule pivot is in a column with $c_j < 0$ on an $a_{hp} > 0$, so the multiple of the pivot row that gets added to the objective row is positive. Thus c_1 became in $\mathbf{T}_2$ that positive multiple of its identity-column $\mathtt{1}$ in $\mathbf{T}_1$. (In pivoting from $\mathbf{T}_0$ to $\mathbf{T}_1$ the basic variable x_3 likewise became nonbasic so its cost coefficient c_3 became positive in $\mathbf{T}_1$, as you should confirm.)

Tableau $\mathbf{T}_2$ has $\mathbf{c} \geq \mathbf{0}$ so it is in optimal form. To recover the optimal point we compute $\mathbf{b}$. Then using the basic sequence S_2 it must be that $x_2^\star = b_1$ and $x_4^\star = b_2$, so $\mathbf{x}^\star = [0, \frac{2}{3}, 0, \frac{7}{3}]^\top$. If the optimal value is of interest the $-z$ entry of $\mathbf{T}_2$ can be found by computing one more dot product.

By using pivot matrices, updating the basic sequence S, and thinking carefully about what happens as we pivot from each canonical-form tableau to the next, we were able to solve this problem without finding most of the elements in $\mathbf{T}_1$ and $\mathbf{T}_2$. In solving a problem with $n \gg m$, as is typical of real applications, this can save a lot of work.

4.2.3 The Phase-2 Algorithm

In solving the pm example we began with a tableau already in canonical form, so the process we used had the effect of carrying out phase 2 of the simplex algorithm. It is summarized by the flowchart below, in which k counts the iterations or pivots.



The "needed entries of $\mathbf{T}_k$" are those c_j that might be negative up to the first one that is, the a_{ip} in that column, and if more than one a_{ip} is positive the corresponding b_i. However, when this algorithm is implemented in a computer program it might turn out that it is less work to calculate some tableau entries that are not needed than it would be to perform the tests required to avoid calculating them.

4.2.4 Phase-1 Algorithms

The modified-simplex approach can also be used to find an initial canonical form, in either of the two ways that we considered in §2.8. One way is to construct an artificial problem and use the phase-2 algorithm of 4.2.3 to solve it. The other is to use pivot matrices to pivot-in a basis and to solve subproblems, calculating at each step only those tableau entries that are needed.

In the tableau $\mathbf{T}_{-2}$ below only x_3 is basic, so $S_{-2} = (\square, x_3)$ is incomplete. To pivot-in a basis I performed the circled pivot by premultiplying with $\mathbf{Q}_{-1}$ to obtain $\mathbf{T}_{-1}$. A tableau that results from pivoting-in a basis can have some b_i negative, so I began computing the elements of $\mathbf{T}_{-1}$ by finding b_1 and b_2. Because b_2 is negative I formed a subproblem to increase it. Computing the cost entries in the subproblem objective row revealed $a_{21} < 0$ so the subproblem pivot must be on a_{11}.

$$\underset{\mathbf{Q}_{-1}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}} \underset{\mathbf{T}_{-2} \text{ with } S_{-2} = (\square, x_3)}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline 3 & 2 & 3 & 0 & 0 \\ \hline 3 & 1 & 1 & 0 & 1 \\ -1 & -1 & -5 & 1 & 1 \end{array}} = \underset{\mathbf{T}_{-1} \text{ with } S_{-1} = (x_4, x_3)}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline & & & & \\ \hline 3 & 1 & & & \\ -4 & -2 & & & \end{array}}$$

The pivot matrix that performs the pivot on a_{11} in $\mathbf{T}_{-1}$ is

$$\mathbf{Q}_0 = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \quad \text{so} \quad \mathbf{P}_0 = \mathbf{Q}_0\mathbf{Q}_{-1} = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

and we can find $\mathbf{T}_0$ as follows.

$$\underset{\mathbf{P}_0}{\begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}} \underset{\mathbf{T}_{-2} \text{ with } S_{-2} = (\square, x_3)}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline 3 & 2 & 3 & 0 & 0 \\ \hline 3 & 1 & 1 & 0 & 1 \\ -1 & -1 & -5 & 1 & 1 \end{array}} = \underset{\mathbf{T}_0 \text{ with } S_0 = (x_1, x_3)}{\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline & & & & \\ \hline 3 & & & & \\ 2 & & & & \end{array}}$$

Now $b_1 > 0$ and $b_2 > 0$ and S_0 contains a complete basis, so it must be that $\mathbf{T}_0$ is in canonical form. You can fill in the remaining entries to verify that this is the starting tableau given in §4.2.1 for the `pm` problem, but in solving that problem by the revised simplex algorithm from this point we would find only the c_j whose values we do not already know, then the a_{ip} in the pivot column, and continue as we did in §4.2.2.

4.2.5 Not Using Unnecessary Space

In solving a small linear program by the simplex algorithm it is convenient to manipulate its $(m+1)\times(n+1)$ tableau [107, p58]. A tableau that is in canonical form includes the identity columns, which makes its basic feasible solution obvious at a glance. But in solving a linear program by the revised simplex method we update the basic sequence S separately, and this allows the algorithm to be described in terms of a data structure that is only $m \times m$. In solving a problem with $n \gg m$, as is typical of real applications, this can save a lot of space.

In §2.2 we formed this initial tableau for the `brewery` problem, in which the all-slack basis has the sequence $S_0 = (x_5, x_6, x_7)$.

$$
\mathbf{T}_0 = \left[\begin{array}{c|c|c}
 & \mathbf{c}_N^{0\top} & \mathbf{c}_B^{0\top} \\
 & x_1 \quad x_2 \quad x_3 \quad x_4 & x_5 \quad x_6 \quad x_7 \\
\hline
0 & [-90 \quad -150 \quad -60 \quad -70] & [0 \quad 0 \quad 0] \\
\hline
\begin{bmatrix}160\\50\\60\end{bmatrix} & \begin{bmatrix}7 & 10 & 8 & 12\\1 & 3 & 1 & 1\\2 & 4 & 1 & 3\end{bmatrix} & \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}
\end{array}\right]
\begin{array}{l}
z = \mathbf{c}_N^\top \mathbf{x}_N + \mathbf{c}_B^\top \mathbf{x}_B \\
\left.\begin{array}{l}\\ \\ \end{array}\right\} \mathbf{b} = \mathbf{N}\mathbf{x}_N + \mathbf{B}\mathbf{x}_B
\end{array}
$$

$\mathbf{b}^0$ $\mathbf{N}_0$ $\mathbf{B}_0$

If we collect the variables that are basic into $\mathbf{x}_B^0 = [x_5, x_6, x_7]^\top$ and those that are nonbasic into $\mathbf{x}_N^0 = [x_1, x_2, x_3, x_4]^\top$, that also partitions the cost and constraint coefficients in this tableau as shown. In general the rows of any tableau with a basis can be thought of as representing the equations given to the right, in which the $m \times m$ matrix $\mathbf{B}$ is called the **basis matrix**.

Solving the constraint equation for the basic variables we find

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N.$$

At a basic feasible solution $\mathbf{x}_N = \mathbf{0}$ so $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$; in $\mathbf{T}_0$, for example,

$$\mathbf{x}_B^0 = \mathbf{B}_0^{-1}\mathbf{b}_0 = \mathbf{I}\mathbf{b}_0 = \begin{bmatrix}160\\50\\60\end{bmatrix}.$$

If we increase some nonbasic variable from zero, the formula for $\mathbf{x}_B$ tells how the basic variables must change to maintain feasibility. Substituting it into the equation for the objective and letting $\mathbf{y}^\top = \mathbf{c}_B^\top \mathbf{B}^{-1}$ we find

$$
\begin{aligned}
z &= \mathbf{c}_N^\top \mathbf{x}_N + \mathbf{c}_B^\top \left(\mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N\right) \\
&= \mathbf{c}_B^\top \mathbf{B}^{-1}\mathbf{b} + \left(\mathbf{c}_N^\top - \mathbf{c}_B^\top \mathbf{B}^{-1}\mathbf{N}\right)\mathbf{x}_N \\
&= \mathbf{y}^\top \mathbf{b} + (\mathbf{c}_N^\top - \mathbf{y}^\top \mathbf{N})\,\mathbf{x}_N.
\end{aligned}
$$

At a basic feasible solution $\mathbf{x}_N = \mathbf{0}$ so $z = \mathbf{y}^\top \mathbf{b}$; in $\mathbf{T}_0$, for example, $\mathbf{y}^\top = [0,0,0]\mathbf{B}^{-1} = [0,0,0]$ and $z = \mathbf{y}^\top \mathbf{b} = 0$.

If we increase some nonbasic variable from zero, the formula

$$z = \mathbf{y}^\top \mathbf{b} + \underbrace{(\mathbf{c}_N^\top - \mathbf{y}^\top \mathbf{N})}_{\text{reduced costs}} \mathbf{x}_N$$

shows that the objective will change by an amount that depends on $\mathbf{x}_N$ and the **nonbasic reduced cost vector** in parentheses. For $\mathbf{T}_0$ we found that $\mathbf{y}^\top = [0, 0, 0]$ so its nonbasic reduced cost vector is just $\mathbf{c}_N^{0\top} = [-90, -150, -60, -70]$.

Now suppose that we store the original problem data

$$\mathbf{A} = \begin{array}{c} \begin{array}{ccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{array} \\ \begin{bmatrix} 7 & 10 & 8 & 12 & 1 & 0 & 0 \\ 1 & 3 & 1 & 1 & 0 & 1 & 0 \\ 2 & 4 & 1 & 3 & 0 & 0 & 1 \end{bmatrix} \end{array} \qquad \mathbf{b} = \begin{bmatrix} 160 \\ 50 \\ 60 \end{bmatrix}$$

as fixed constants but treat $\mathbf{B}$, $\mathbf{x}_B$, $\mathbf{x}_N$, $\mathbf{c}_B$ and $\mathbf{c}_N$ as variables with these initial values.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{array}{ll} \mathbf{x}_B = [x_5, x_6, x_7]^\top = [160, 50, 60]^\top & \mathbf{c}_B = [0, 0, 0]^\top \\ \mathbf{x}_N = [x_1, x_2, x_3, x_4]^\top = [0, 0, 0, 0]^\top & \mathbf{c}_N = [-90, -150, -60, -70]^\top \end{array}$$

Can we solve the `brewery` problem by manipulating only these variables?

Because the reduced cost vector $\mathbf{c}_N$ has negative entries the current solution must not be optimal. We can find a better point by increasing the variable that corresponds to any negative entry in $\mathbf{c}_N$, so let $x_1 = t$ or $\mathbf{x}_N = [t, 0, 0, 0]^\top$. To stay feasible we must adjust $\mathbf{x}_B$ to

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N.$$

The matrix-vector product

$$\mathbf{N}\mathbf{x}_N = \begin{bmatrix} 7 & 10 & 8 & 12 \\ 1 & 3 & 1 & 1 \\ 2 & 4 & 1 & 3 \end{bmatrix} \begin{bmatrix} t \\ 0 \\ 0 \\ 0 \end{bmatrix} = t \begin{bmatrix} 7 \\ 1 \\ 2 \end{bmatrix}$$

is always just t times the column of $\mathbf{A}$ that corresponds to the nonbasic variable being increased, so it is never actually necessary to write down $\mathbf{N}$. The current basis matrix $\mathbf{B}$ is the identity so $\mathbf{B}^{-1}$ is too, and

$$\mathbf{x}_B = \begin{bmatrix} 160 \\ 50 \\ 60 \end{bmatrix} - t \begin{bmatrix} 7 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 160 - 7t \\ 50 - t \\ 60 - 2t \end{bmatrix}. \qquad \begin{array}{rcl} 160 - 7t & \geq & 0 \;\Rightarrow\; t \leq \frac{160}{7} \approx 22.9 \\ 50 - t & \geq & 0 \;\Rightarrow\; t \leq 50 \\ 60 - 2t & \geq & 0 \;\Rightarrow\; t \leq 30 \end{array}$$

The highest value of t that keeps $\mathbf{x}_B \geq \mathbf{0}$ is $t = \frac{160}{7}$, and it yields

$$\begin{bmatrix} x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 160 - 7t \\ 50 - t \\ 60 - 2t \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{190}{7} \\ \frac{100}{7} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} t \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{160}{7} \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The pivot has made x_5 nonbasic and x_1 basic, changing the basic sequence to $S_1 = (x_1, x_6, x_7)$.

$$\mathbf{x}_B = \begin{bmatrix} x_1 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} \frac{160}{7} \\ \frac{190}{7} \\ \frac{100}{7} \end{bmatrix} \qquad \mathbf{x}_N = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This basic sequence specifies the columns of the original data that make up the new $\mathbf{B}$, $\mathbf{c}_B$, and $\mathbf{c}_N$.

$$\mathbf{B} = \begin{bmatrix} 7 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \quad \mathbf{B}^{-1} = \begin{bmatrix} \frac{1}{7} & 0 & 0 \\ -\frac{1}{7} & 1 & 0 \\ -\frac{2}{7} & 0 & 1 \end{bmatrix} \qquad \begin{aligned} \mathbf{c}_B &= [-90, 0, 0]^\top \\ \mathbf{c}_N &= [-150, -60, -70, 0]^\top \end{aligned}$$

Using these quantities we can compute reduced costs for the new nonbasic columns.

$$\begin{aligned} \mathbf{y}^\top = \mathbf{c}_B^\top \mathbf{B}^{-1} &= [-90, 0, 0] \begin{bmatrix} \frac{1}{7} & 0 & 0 \\ -\frac{1}{7} & 1 & 0 \\ -\frac{2}{7} & 0 & 1 \end{bmatrix} = \left[-\tfrac{90}{7}, 0, 0\right] \\ \mathbf{y}^\top \mathbf{N} &= [-\tfrac{90}{7}, 0, 0] \begin{bmatrix} 10 & 8 & 12 & 1 \\ 3 & 1 & 1 & 0 \\ 4 & 1 & 3 & 0 \end{bmatrix} = \left[-\tfrac{900}{7}, -\tfrac{720}{7}, -\tfrac{1080}{7}, -\tfrac{90}{7}\right] \\ \mathbf{c}_N^\top - \mathbf{y}^\top \mathbf{N} &= [-150, -60, -70, 0] - \left[-\tfrac{900}{7}, -\tfrac{720}{7}, -\tfrac{1080}{7}, -\tfrac{90}{7}\right] = \left[-\tfrac{150}{7}, \tfrac{300}{7}, \tfrac{590}{7}, \tfrac{90}{7}\right] \end{aligned}$$

Here as usual $\mathbf{N}$ is not a separate matrix but merely a shorthand way of referring to those columns of $\mathbf{A}$ that correspond to the current set of nonbasic variables. In **pricing out** the nonbasic columns we can compute the elements of $\mathbf{y}^\top \mathbf{N}$ and $\mathbf{c}_N^\top - \mathbf{y}^\top \mathbf{N}$ one at a time until finding the *first* reduced cost that is negative. In a real problem $\mathbf{A}$ might have a great many columns, so it is important for efficiency to refrain from finding unneeded elements of $\mathbf{y}^\top \mathbf{N}$.

It is the first nonbasic variable, now x_2, that has a negative reduced cost, so we let $x_2 = t$ or $\mathbf{x}_N = [t, 0, 0, 0]$ and write down, by inspection of $\mathbf{A}$,

$$\mathbf{N}\mathbf{x}_N = t \begin{bmatrix} 10 \\ 3 \\ 4 \end{bmatrix}.$$

Then we can find the basic variables in terms of t,

$$\mathbf{x}_B = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{N}\mathbf{x}_N) = \begin{bmatrix} \frac{1}{7} & 0 & 0 \\ -\frac{1}{7} & 1 & 0 \\ -\frac{2}{7} & 0 & 1 \end{bmatrix} \begin{bmatrix} 160 & - & 10t \\ 50 & - & 3t \\ 60 & - & 4t \end{bmatrix} = \begin{bmatrix} \frac{160}{7} - \frac{10}{7}t \\ \frac{190}{7} - \frac{11}{7}t \\ \frac{100}{7} - \frac{8}{7}t \end{bmatrix}$$

and the minimum-ratio row.

$$\begin{array}{rcl} \frac{160}{7} - \frac{10}{7}t & \geq & 0 \;\Rightarrow\; t \leq \frac{160}{10} = 16 \\ \frac{190}{7} - \frac{11}{7}t & \geq & 0 \;\Rightarrow\; t \leq \frac{190}{11} \approx 17.27 \\ \frac{100}{7} - \frac{8}{7}t & \geq & 0 \;\Rightarrow\; t \leq \frac{100}{8} = 12.5 \end{array}$$

The minimum ratio pivot that increases x_2 makes the third basic variable, x_7, nonbasic, changing the basic sequence to $S_2 = (x_1, x_6, x_2)$ and yielding

$$\mathbf{x}_B = \begin{bmatrix} x_1 \\ x_6 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{160}{7} - \frac{10}{7}\frac{100}{8} \\ \frac{190}{7} - \frac{11}{7}\frac{100}{8} \\ \frac{100}{8} \end{bmatrix} = \begin{bmatrix} 5 \\ 7.5 \\ 12.5 \end{bmatrix} \qquad \mathbf{x}_N = \begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_7 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Pricing out this solution reveals that the reduced costs corresponding to its nonbasic variables are all positive, so $\mathbf{x}^\star = [5, 12.5, 0, 0, 0, 7.5, 0]^\top$. This is the optimal point we found in §2.4.3 for the `brewery` problem.

Although the algorithm flowcharted in §4.2.3 requires less arithmetic than this one it requires about twice as much space, so the **matrix simplex method** [107, §3.7] illustrated by this example is always used in production linear programming codes based on pivoting, and it is the one that most authors (e.g., [4, §5.2.1], [5, §3.3], [79, §17.4]) refer to as the revised simplex method.

Stating the problem in matrix form reveals that solving a linear program consists simply of finding the best set of $\mathbf{A}$ columns to have in the basis, or the best m of the n variables to allow to be nonzero. The basis matrix enters into the revised simplex calculations in such a way that each step uses the *original* problem data. The canonical form at iteration k is represented by the square linear system $\mathbf{B}_k\mathbf{x} = \mathbf{b}$, where the columns of $\mathbf{B}_k$ are the columns of $\mathbf{A}$ that are in S_k. Each phase-2 pivot exchanges one of the columns of $\mathbf{A}$ that is in $\mathbf{B}$ for another column of $\mathbf{A}$, and each basic feasible solution is $\mathbf{x}^k = \mathbf{B}_k^{-1}\mathbf{b}$.

4.3 Large Problems

Our naïve implementation of the tableau simplex method in `simplex.m` is straightforward and easy to understand, but it is practical for solving only small linear programs. The matrix version of the revised simplex method uses less processor time and memory, but it too is unsuitable for large problems unless implemented in a more subtle way than suggested above.

Since the discovery of the simplex algorithm in July of 1947 [35, p15] several generations of very smart people have refined its software realization, in the process generating a vast literature whose details are well beyond the scope of this text. Here I will describe only a few of their clever ideas, which you can find out more about by consulting the cited references.

4.3.1 Representing the Basis Inverse

Whenever we needed $\mathbf{B}^{-1}$ in §4.2.5 I just wrote it down as though finding it were effortless, but a revised simplex code that uses this **basis inverse matrix** must somehow calculate it at each step. Explicitly inverting $\mathbf{B}$ with a direct method requires a number of arithmetic operations that is proportional to m^3 [20, p282], which is ruinous if m is large. The first practical implementations of the revised simplex method found $\mathbf{B}_k^{-1}$ through a less-expensive process of updating $\mathbf{B}_{k-1}^{-1}$, either by pivoting an augmented matrix [103, §1.2.2] [107, p60-63] or by using a **product-form inverse** [4, §7.5.1] [103, §6.2] in which $\mathbf{B}^{-1}$ is represented as a product of elementary matrices.

If $\mathbf{B}$ has an inverse it is convenient in matrix algebra to denote the solution to $\mathbf{Br} = \mathbf{s}$ as $\mathbf{r} = \mathbf{B}^{-1}\mathbf{s}$, and that is what we did in §4.2.5. But to solve the linear system numerically it is better to begin by finding a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$ such that $\mathbf{B} = \mathbf{LU}$. Then $\mathbf{LUr} = \mathbf{s}$, and if we let $\mathbf{Ur} = \mathbf{v}$ we can solve $\mathbf{Lv} = \mathbf{s}$ for $\mathbf{v}$ very easily by doing simple substitutions. Once $\mathbf{v}$ has been found we can solve $\mathbf{Ur} = \mathbf{v}$ for $\mathbf{r}$ in the same easy way. If this approach of **matrix factorization** followed by forward- and back-substitutions is used for solving the linear systems in the revised simplex algorithm, the factors $\mathbf{L}_k$ and $\mathbf{U}_k$ can be found by updating $\mathbf{L}_{k-1}$, and $\mathbf{U}_{k-1}$ [4, §7.5.2] [5, §13.4]. Even if the product-form inverse is used to update $\mathbf{B}^{-1}$, calculating $\mathbf{r} = \mathbf{B}^{-1}\mathbf{s}$ turns out to be slower and less accurate, so modern codes update $\mathbf{L}$ and $\mathbf{U}$ and solve the triangular systems $\mathbf{Lv} = \mathbf{s}$ and $\mathbf{Ur} = \mathbf{v}$ instead.

4.3.2 Exploiting Problem Structure

Large linear programs almost always [103, page v] have special **structure**: if we were to put the standard-form problem into a tableau its entries (perhaps after some rearrangement of rows and columns) would have a regular pattern. Often, as in the case of the transportation problem that we will study in §6, it is possible to develop a special-purpose algorithm that exploits the particular pattern that is present, to reduce the amount of work or space needed to solve the problem. It might be impractical to solve a very large problem *except* by using an algorithm that takes advantage of its structure. The most broadly useful exploitations of special structure are **upper bounding** and **column generation**.

UPPER BOUNDING Many linear programs have the special structure that some constraints are upper bounds on the variables (see §2.9.5). For example, the branch-and-bound algorithm for solving integer linear programs, which we will study in §7, generates linear programming

subproblems that include upper bound constraints. A bound such as $x_1 \leq 3$ can be handled like any other inequality, by adding a slack variable and a row to $\mathbf{A}$ and $\mathbf{b}$. But it is also possible to modify the revised simplex algorithm [4, §7.2] [103, §6.3] [145, §10.6] in such a way that upper bounds on the variables are handled in the same way as their (usually zero) lower bounds *without* enlarging the basis matrix $\mathbf{B}$. The algorithm becomes significantly more complicated, but if many variables have upper bounds this strategy can save both work and space.

COLUMN GENERATION Some linear programs have a special structure that permits a column of $\mathbf{A}$ with negative cost to be produced when needed by solving an auxiliary problem within each iteration of the revised simplex method. This permits the simplex iterations to continue until the auxiliary problem's solution reveals that optimality has been achieved, and it can make possible the solution of problems in which there are too many variables to find or store all of $\mathbf{A}$.

4.3.3 Decomposition

The most important instance in which column generation can be used is when the nonzero constraint coefficients of a large linear program can be arranged into a **block-angular** structure so that

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1p} \\ \mathbf{A}_{21} & & & \\ & \mathbf{A}_{32} & & \\ & & \ddots & \\ & & & \mathbf{A}_{(p+1)p} \end{bmatrix}.$$

Each block $\mathbf{A}_{(j+1)j}$ contains the coefficients in a set of constraints that involve only a subset of the variables, but the **coupling equations** in the first row involve all of the variables. If it were not for the coupling equations each linear (sub)program represented by an $\mathbf{A}_{(j+1)j}$ block could be solved independently to find the optimal values of its variables.

For simplicity we will consider the case when $p = 2$, so that the linear program having constraint coefficient matrix $\mathbf{A}$ can be written as follows,

$$\begin{array}{lrcrcrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & z & = & \mathbf{c}^{1\top}\mathbf{x}^1 & + & \mathbf{c}^{2\top}\mathbf{x}^2 & & \\ \text{subject to} & & & \mathbf{A}_{11}\mathbf{x}^1 & + & \mathbf{A}_{12}\mathbf{x}^2 & = & \mathbf{b}^1 \\ & & & \mathbf{A}_{21}\mathbf{x}^1 & & & = & \mathbf{b}^2 \\ & & & & & \mathbf{A}_{32}\mathbf{x}^2 & = & \mathbf{b}^3 \\ & & & \mathbf{x}^1, & & \mathbf{x}^2 & \geq & \mathbf{0} \end{array}$$

where $\mathbf{x}^1 = [x_1 \ldots x_{n_1}]^\top$, $\mathbf{x}^2 = [x_{n_1+1} \ldots x_{n_1+n_2}]^\top$, and $n_1 + n_2 = n$. If there are m_1 coupling equations and the block constraints have a total of m_2 rows then this problem has $m = m_1 + m_2$ equality constraints and n variables.

In any optimal solution, $\mathbf{x}^1$ and $\mathbf{x}^2$ must each satisfy its block and nonnegativity constraints, which define these polyhedra.

$$\mathbb{X}_1 = \left\{ \mathbf{x}^1 \in \mathbb{R}^{n_1} \;\middle|\; \mathbf{A}_{21}\mathbf{x}^1 = \mathbf{b}^2,\ \mathbf{x}^1 \geq \mathbf{0} \right\} \qquad \mathbb{X}_2 = \left\{ \mathbf{x}^2 \in \mathbb{R}^{n_2} \;\middle|\; \mathbf{A}_{32}\mathbf{x}^2 = \mathbf{b}^3,\ \mathbf{x}^2 \geq \mathbf{0} \right\}$$

If $\mathbb{X}_1$ is bounded and has extreme points $\mathbf{u}^1 \ldots \mathbf{u}^{L_1}$ then [103, §3.2] any point $\mathbf{x}^1 \in \mathbb{X}_1$ can be written (see §3.5) as the convex combination

$$\mathbf{x}^1 = \sum_{l=1}^{L_1} \alpha_l \mathbf{u}^l \quad \text{where} \quad \sum_{l=1}^{L_1} \alpha_l = 1 \quad \text{and} \quad \alpha_l \geq 0,\ \ l = 1 \ldots L_1.$$

If $\mathbb{X}_2$ is bounded and has extreme points $\mathbf{v}^1 \ldots \mathbf{v}^{L_2}$ then any point $\mathbf{x}^2 \in \mathbb{X}_2$ can be written as the convex combination

$$\mathbf{x}^2 = \sum_{l=1}^{L_2} \beta_l \mathbf{v}^l \quad \text{where} \quad \sum_{l=1}^{L_2} \beta_l = 1 \quad \text{and} \quad \beta_l \geq 0,\ \ l = 1 \ldots L_2.$$

Here L_1 and L_2 are the numbers of extreme points of the polyhedra $\mathbb{X}_1$ and $\mathbb{X}_2$. A polyhedron in $\mathbb{R}^n$ can have many more than n extreme points, so typically $L_1 \gg n_1$ and $L_2 \gg n_2$.

By substituting these representations of $\mathbf{x}^1$ and $\mathbf{x}^2$ we can rewrite the original linear program in terms of the extreme points as this **master problem**.

$$\begin{array}{lrcl}
\underset{\alpha \in \mathbb{R}^{L_1}\ \beta \in \mathbb{R}^{L_2}}{\text{minimize}} & z & = & c^{1\top} \displaystyle\sum_{l=1}^{L_1} \alpha_l \mathbf{u}^l \;+\; c^{2\top} \displaystyle\sum_{l=1}^{L_2} \beta_l \mathbf{v}^l \\
\text{subject to} & \mathbf{A}_{11} \displaystyle\sum_{l=1}^{L_1} \alpha_l \mathbf{u}^l \;+\; \mathbf{A}_{12} \displaystyle\sum_{l=1}^{L_2} \beta_l \mathbf{v}^l & = & \mathbf{b}^1 \\
& \displaystyle\sum_{l=1}^{L_1} \alpha_l & = & 1 \\
& \displaystyle\sum_{l=1}^{L_2} \beta_l & = & 1 \\
& \boldsymbol{\alpha} & \geq & \mathbf{0} \\
& \boldsymbol{\beta} & \geq & \mathbf{0}
\end{array}$$

Now there are a huge number of variables but only $m_1 + 2$ constraints, so the basis matrix $\mathbf{B}$ is small. This problem is therefore easy, if only we can figure out where to pivot at every iteration of the revised simplex algorithm. It turns out [4, §7.4] [103, §3.3] that there is an auxiliary problem, also an easy linear program, that can be used on each constraint block to generate an extreme-point column having a negative cost or to determine that there are none.

To solve the master problem by the revised simplex method it is convenient to rewrite it as follows.

$$\begin{array}{lll} \underset{\mathbf{w}\in\mathbb{R}^{L_1+L_2}}{\text{minimize}} & \mathbf{c}_M^\top\mathbf{w} & \\ \text{subject to} & \mathbf{A}_M\mathbf{w} & = \mathbf{b}_M \\ & \mathbf{w} & \geq \mathbf{0} \end{array} \quad \text{where} \quad \begin{cases} \mathbf{c}_M^\top = \left[\mathbf{c}^{1\top}\mathbf{u}^1 \cdots \mathbf{c}^{1\top}\mathbf{u}^{L_1}, \mathbf{c}^{2\top}\mathbf{v}^1 \cdots \mathbf{c}^{2\top}\mathbf{v}^{L_2}\right] \\ \mathbf{w}^\top = [\boldsymbol{\alpha}^\top, \boldsymbol{\beta}^\top] \\ \mathbf{A}_M = \begin{bmatrix} \mathbf{A}_{11}\mathbf{u}^1 & \cdots & \mathbf{A}_{11}\mathbf{u}^{L_1} & \mathbf{A}_{12}\mathbf{v}^1 & \cdots & \mathbf{A}_{12}\mathbf{v}^{L_2} \\ 1 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & \cdots & 1 \end{bmatrix} \\ \mathbf{b}_M = \begin{bmatrix} \mathbf{b}^1 \\ 1 \\ 1 \end{bmatrix} \end{cases}$$

If at some stage the basis inverse matrix is $\mathbf{B}^{-1}$ and the original costs corresponding to the basic columns are $[\mathbf{c}_M]_B^\top$ then we can find $\mathbf{y}^\top = [\mathbf{c}_M]_B^\top\mathbf{B}^{-1} = \left[\bar{\mathbf{y}}, y_\alpha, y_\beta\right]$. This vector is $m+2$ elements long, with its last two elements corresponding to the sum constraints on $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Then the reduced costs are the elements of this vector.

$$\mathbf{c}_M^\top - \mathbf{y}^\top\mathbf{A}_M = \left[(\mathbf{c}^{1\top} - \bar{\mathbf{y}}^\top\mathbf{A}_{11})\mathbf{u}^1 \cdots (\mathbf{c}^{1\top} - \bar{\mathbf{y}}^\top\mathbf{A}_{11})\mathbf{u}^{L_1}, (\mathbf{c}^{2\top} - \bar{\mathbf{y}}^\top\mathbf{A}_{12})\mathbf{v}^1 \cdots (\mathbf{c}^{2\top} - \bar{\mathbf{y}}^\top\mathbf{A}_{12})\mathbf{v}^{L_1}\right]$$

Because [4, p232] the $\mathbf{u}^l$ and $\mathbf{v}^l$ are extreme points of $\mathbb{X}_1$ and $\mathbb{X}_2$ we can find the lowest reduced cost among the terms involving the $\mathbf{u}^l$ by solving the auxiliary problem on the left below and the lowest reduced cost among the terms involving the $\mathbf{v}^1$ by solving the auxiliary problem on the right.

$$\begin{array}{llll} \underset{\mathbf{x}^1\in\mathbb{R}^{n_1}}{\text{minimize}} & q_1 & = & \left(\mathbf{c}^{1\top} - \mathbf{A}_{11}^\top\bar{\mathbf{y}}\right)^\top\mathbf{x}^1 - y_\alpha \\ \text{subject to} & \mathbf{x}^1 & \in & \mathbb{X}_1 \end{array} \qquad \begin{array}{llll} \underset{\mathbf{x}^2\in\mathbb{R}^{n_2}}{\text{minimize}} & q_2 & = & \left(\mathbf{c}^{2\top} - \mathbf{A}_{12}^\top\bar{\mathbf{y}}\right)^\top\mathbf{x}^2 - y_\beta \\ \text{subject to} & \mathbf{x}^2 & \in & \mathbb{X}_2 \end{array}$$

If either problem has an objective value that is negative then its optimal vector is one of the extreme points $\mathbf{u}^l$ or $\mathbf{v}^l$ and that column of $\mathbf{A}_M$ can be chosen to enter the basis; if $q_1^\star \geq 0$ and $q_2^\star \geq 0$ then the current basis is optimal for the master problem.

When the original problem has $p \geq 2$ blocks of constraints we get p subproblems, and because they involve disjoint sets of variables they could be solved simultaneously on a computer with p processors.

If the master problem has the optimal solution $(\boldsymbol{\alpha}^\star, \boldsymbol{\beta}^\star)$, the solution to the original linear program is

$$\begin{aligned} \mathbf{x}_1^\star &= \mathbf{U}\boldsymbol{\alpha}^\star \\ \mathbf{x}_2^\star &= \mathbf{V}\boldsymbol{\beta}^\star \end{aligned}$$

where $\mathbf{U}$ and $\mathbf{V}$ are matrices whose columns are respectively the $\mathbf{u}$ and the $\mathbf{v}$ columns that are basic in the solution to the master problem.

To simplify the exposition above I assumed that $\mathbb{X}_1$ and $\mathbb{X}_2$ are bounded sets, but [103, §3.2] the decomposition algorithm also works if there are rays.

4.4 Linear Programming Software

The *algorithm* improvements described in §4.2 and §4.3 are mathematical results and thus largely independent of how the simplex method calculations are carried out. Refinements can also be made in the *implementation* of the algorithm [4, §7.6] [5, §13.5,13.7] and computer programs that are considered to be of industrial strength do that too.

4.4.1 Picking a Good Pivot Column

To solve a canonical-form linear program by the simplex algorithm we can pivot in any column having $c_j < 0$, so in the revised simplex methods described above we avoided some work by picking the *first* such column; I will call this the **first-negative pricing rule**. Might a heuristic that requires more columns to be **priced out** for each pivot nonetheless speed convergence, by allowing the algorithm to reach optimal form in fewer iterations? To study this question recall the `graph` problem of §3.1, whose starting tableau is shown on the left below.

	x_1	x_2	s_1	s_2	s_3	s_4
0	−2	−1	0	0	0	0
6	1	$\frac{6}{5}$	1	0	0	0
2	1	−1	0	1	0	0
3	1	0	0	0	1	0
5	0	1	0	0	0	1

	x_1	x_2	s_1	s_2	s_3	s_4
4	0	−3	0	2	0	0
4	0	$\frac{11}{5}$	1	−1	0	0
2	1	−1	0	1	0	0
1	0	1	0	−1	1	0
5	0	1	0	0	0	1

	x_1	x_2	s_1	s_2	s_3	s_4
5	−2	0	0	0	0	1
0	1	0	1	0	0	$-\frac{6}{5}$
7	1	0	0	1	0	1
3	1	0	0	0	1	0
5	0	1	0	0	0	1

In this problem $z = -2x_1 - x_2$ so a unit increase in x_1 improves the objective by 2 while a unit increase in x_2 improves the objective by only 1. This suggests that we should pivot in the column having the most negative c_j, which I will call the **most-negative** pricing rule. Following it yields the top tableau on the right, with an objective value of $z = -4$.

Alas, the most-negative pricing rule does not always result in the biggest improvement to the objective value; pivoting in the x_2 column above produces the bottom tableau on the right, with an objective value of $z = -5$. Rather than picking the column with the most negative c_j we could calculate for each column having a negative c_j what the objective would change to if the pivot were performed in that column,

$$z^{k+1} = z^k + b_h\left(\frac{c_j}{a_{hj}}\right).$$

Then we could pick the column whose pivot would result in the lowest z^{k+1}. Unfortunately this **optimal** pricing rule requires finding the pivot row h in each possible pivot column. To do this using the MATLAB code we wrote in §4.1 would require an invocation of `minr.m` for each column we consider, and this is likely to take more work than we could save by picking better pivots.

A strategy that is cheaper than optimal pricing but yields faster convergence than first-negative pricing is the **steepest-edge** pricing rule [4, §7.6.1]. In the matrix simplex method of §4.2.5 we derived this formula telling how the basic variables must be related to the nonbasic ones in order for $\mathbf{x}^\top = [\mathbf{x}_N^\top, \mathbf{x}_B^\top]$ to be feasible.

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N$$

If $\mathbf{x}^k$ is a basic feasible solution and we change $\mathbf{x}_N$ from $\mathbf{0}$ by increasing some nonbasic variable, then $\mathbf{x}_B$ must also change to remain feasible and we will move to this point.

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_N \\ \mathbf{x}_B \end{bmatrix} = \begin{bmatrix} \mathbf{x}_N \\ \mathbf{B}_k^{-1}\mathbf{b} - \mathbf{B}_k^{-1}\mathbf{N}\mathbf{x}_N \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{B}_k^{-1}\mathbf{b} \end{bmatrix} + \begin{bmatrix} \mathbf{x}_N \\ -\mathbf{B}_k^{-1}\mathbf{N}\mathbf{x}_N \end{bmatrix} = \mathbf{x}^k + \begin{bmatrix} \mathbf{I} \\ -\mathbf{B}_k^{-1}\mathbf{N} \end{bmatrix}\mathbf{x}_N = \mathbf{x}^k + \mathbf{Z}_k\mathbf{x}_N$$

For example, in solving the `brewery` problem our first pivot increased x_1, changing $\mathbf{x}_N$ from $[0,0,0,0]^\top$ to $[t > 0,0,0,0]^\top$ and moving the solution to

$$\mathbf{x}(t) = \mathbf{x}^0 + \mathbf{Z}_0\mathbf{x}_N = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 160 \\ 50 \\ 60 \end{bmatrix} + \begin{array}{c} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -7 & -10 & -8 & -12 \\ -1 & -3 & -1 & -1 \\ -2 & -4 & -1 & -3 \end{bmatrix} \\ \begin{array}{cccc} \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_4 \end{array} \end{array} \begin{bmatrix} t \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 160 \\ 50 \\ 60 \end{bmatrix} + t \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ -7 \\ -1 \\ -2 \end{bmatrix} = \mathbf{x}^0 + t\mathbf{v}_1.$$

The columns of $\mathbf{Z}$, which I have labeled $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, and $\mathbf{v}_4$, are the **edge directions** in which $\mathbf{x}(t)$ moves if we pivot by the simplex rule in the first, second, third, or fourth nonbasic column of $\mathbf{A}$, and the distance t that we move is the minimum ratio for that column.

Recall that the costs associated with the nonbasic variables are given by $\hat{\mathbf{c}}^\top = \mathbf{c}_N^\top - \mathbf{c}_B^\top\mathbf{B}^{-1}\mathbf{N}$. At the first pivot in our example, $\mathbf{c}_B^\top = [0,0,0]^\top$ so

$$\hat{c}_1 = c_1 - [0,0,0]\begin{bmatrix} -7 \\ -1 \\ -2 \end{bmatrix} = -90, \quad \hat{c}_2 = c_2 - [0,0,0]\begin{bmatrix} -10 \\ -3 \\ -4 \end{bmatrix} = -150, \ldots$$

and in general $\hat{c}_j = [\mathbf{c}_N^\top, \mathbf{c}_B^\top]\mathbf{v}_j = \mathbf{c}^\top\mathbf{v}_j$. In this dot product the identity-column part of $\mathbf{v}_j$ picks the appropriate nonbasic cost c_j out of $\mathbf{c}$ and the part of $\mathbf{v}_j$ that is a column of $-\mathbf{B}^{-1}\mathbf{N}$ is used in calculating the second term in the formula for $\hat{c}_j$.

We have shown that the reduced costs associated with the nonbasic variables can be found one at a time in each iteration k of revised simplex by constructing $\mathbf{Z}_k$ and computing $\hat{c}_j = \mathbf{c}^\top \mathbf{v}_j$ for each column $\mathbf{v}_j$ of $\mathbf{Z}_k$ (that is, for each j in the current index set of nonbasic variables). Thinking about the pricing-out operation in this way reveals that each reduced cost $\hat{c}_j$ is a weighted sum of the c_j in which the weights are the elements of the edge direction vector $\mathbf{v}_j$. If some of the $\mathbf{v}_j$ are longer than others this calculation can yield $\hat{c}_j$ values that do not fairly represent the relative importance of the nonbasic variables. Steepest-edge pricing removes this bias by normalizing each edge direction vector to compute

$$\bar{c}_j = \frac{1}{\|\mathbf{v}_j\|}\mathbf{c}^\top \mathbf{v}_j$$

and pivoting in the column for which $\bar{c}_j$ is most negative. To avoid the work of explicitly computing $\mathbf{Z}_k$ and normalizing its columns, in practice a rather complicated updating scheme [4, p261-264] is used to find the normalized edge directions.

In a problem that has many columns it might be expensive to apply the most-negative or steepest-edge pricing rule to all of them. This is called **full** pricing. Instead many codes do **partial** pricing, by finding the most negative $\hat{c}_j$ or $\bar{c}_j$ among a smaller **candidate list** of nonbasic columns. Thus the most-negative and steepest-edge pricing rules can each be either full or partial.

4.4.2 Tolerances and Scaling

Our MATLAB implementation of the tableau simplex algorithm in §4.1 must avoid pivoting on a zero a_{hp}, delete a constraint row that is all zeros, accept a subproblem solution if its objective value is close enough to zero, and identify optimal form when the reduced costs are all positive or zero. In each context the numbers that ought to be zero would be if we used exact arithmetic but usually come out slightly different in floating point. To decide if a real value can be assumed zero I compare its absolute value to `ztol` $= 10^{-6}$. This **zero tolerance** works for the examples we have studied, in which the coefficients are neither much bigger nor much smaller than 1, but it would cause the algorithm to malfunction in solving a problem whose data are all tiny numbers or all huge ones.

If a problem has data that span many orders of magnitude it is likely that at least some of its basis matrices will be **ill-conditioned**, and this accelerates the accumulation of roundoff errors (see §10.6.2).

To mitigate these tolerance and conditioning effects many authors (e.g. [4, §7.6.4]) recommend scaling the constraint rows or variable columns of a linear program, or both, to make the element largest in absolute value have magnitude near 1. To keep the scaling calculations themselves from introducing roundoff errors [77, p60] the scale factor can be made a power of 2. Although [87, §4.8-4.9] scaling often fails to ensure the accuracy of computed results, linear programming packages commonly provide scaling options and also allow the user to set the various tolerances that are used (which might not all have the same value).

4.4.3 Preprocessing

If the `newseq.m` routine of §4.1 discovers a zero **A** row it either removes the redundant constraint if $b_i = 0$ or reports infeasible form 1 if $b_i \neq 0$. Because this happens before entering `phase1.m` it can be thought of as simplifying the problem (or, if infeasibility is discovered, solving it) before the simplex algorithm even begins. In our MATLAB implementation this **preprocessing** is an accidental byproduct of pivoting-in a basis, but many production codes explicitly analyze a linear program for these and other ways of making the problem smaller or easier, before applying the simplex algorithm [5, §13.7] [4, §7.6.5].

An equality constraint that involves a single variable (this is called a **row singleton**) fixes the value of that variable. By substituting this value wherever the variable appears, both the variable and the constraint can be eliminated from the problem.

In a code that uses upper-bounding as described in §4.3.2, a general constraint that is really just an upper bound on a variable can be treated that way instead. Variable bounds that happen to be known can also sometimes be used to simplify other constraints. In this example

$$\begin{aligned} x_1 + x_2 &= 10 \\ x_1 &\geq 10 \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

it must be that $x_1 = 10$ and $x_2 = 0$, so we can fix those values and remove both constraints.

One pass of preprocessing might simplify the problem in such a way that a second pass can make further simplifications. In this example

$$\begin{aligned} x_3 &= 1 \\ x_3 + 2x_4 &= 5 \end{aligned}$$

the first pass could substitute for x_3 its value of 1, removing that variable and the first constraint. The resulting second constraint

$$1 + 2x_4 = 5$$

then implies that $x_4 = 2$ so a second pass of preprocessing could replace that variable by its value, eliminating x_4 and this constraint.

Some preprocessors can detect and exploit more complicated relationships between constraints. Simplifying a problem might dramatically reduce the number of pivots required to solve it, but preprocessing also takes work and the more sophisticated the preprocessing is the more work it takes. For a given linear program some optimal level of preprocessing will minimize the total time to solution, but unfortunately that level is hard to guess beforehand.

4.4.4 Black-Box Solvers

Some books on applied operations research introduce linear programming by showing the student how a particular canned computer program or **package** can be used to solve typical problems, and this practical knowledge might be all an analyst needs to get useful answers out of well-behaved optimization models. Of course some formulations are infeasible or unbounded or badly-scaled, or have optimal rays or multiple optimal vertices or some other peculiarity, and then it can be hard to interpret the output from a linear programming package without having some idea how it works inside.

Other books refrain even from naming particular packages in light of how many have come and gone, waxing and waning in popularity, over the long history of linear programming (a web search will turn up dozens). Yet linear programming is, in theory and largely also in practice, a solved problem, and a few simplex-method codes have persisted for so many years that it seems likely they will still be in use as you read these words [117, §5]. Surely these deserve to be mentioned, even though this book is mainly about the mathematical and algorithmic foundations of numerical optimization rather than production software.

For most small problems, most any **solver** will do. Our MATLAB routine `simplex.m` has the virtue that you know all about it. At any given moment in history there are other free open-source solvers, of varying capabilities and quality, that can be downloaded from the internet. **Excel** can solve linear programs exactly by representing the data as fractions. Both **Maple** and **Mathematica** can solve linear programs symbolically as well as numerically. **Lingo** has both a venerable heritage and a modern interface for web applications.

For the largest problems, only purpose-written code will do. Models that involve vast amounts of data always have special structure, and any effective approach to solving them must exploit it. Often interior-point methods (see §21.3 and [5, §14.4]) work better than the simplex algorithm in this context.

For linear programs of intermediate size, two widely respected packages are **CPLEX** (which implements interior-point as well as simplex algorithms) and **MINOS** (which can handle nonlinear as well as linear programs). Both of these solvers are proprietary, but you can avoid paying a license fee if you use them via the NEOS web server discussed in §8.3.1.

4.5 Degeneracy

The right tableau is an optimal form for the left tableau [3, p52-53] [145, p91] [11].

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
3	0	0	0	$-\frac{3}{4}$	20	$-\frac{1}{2}$	6
0	1	0	0	$\frac{1}{4}$	-8	-1	9
0	0	1	0	$\frac{1}{2}$	-12	$-\frac{1}{2}$	3
1	0	0	1	0	0	1	0

$\xrightarrow{\text{pivots}}$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$\frac{17}{4}$	0	$\frac{3}{2}$	$\frac{5}{4}$	0	2	0	$\frac{21}{2}$
$\frac{3}{4}$	1	$-\frac{1}{2}$	$\frac{3}{4}$	0	-2	0	$\frac{15}{2}$
1	0	2	1	1	-24	0	6
1	0	0	1	0	0	1	0


```
> This is PIVOT, Unix version 4.3
> For a list of commands, enter HELP.
>
< read cycle.tab
Reading the tableau...
...done.

     x1  x2  x3  x4    x5    x6   x7
 3.  0.  0.  0. -0.75   20. -0.5  6.
 0.  1.  0.  0.  0.25   -8. -1.0  9.
 0.  0.  1.  0.  0.50  -12. -0.5  3.
 1.  0.  0.  1.  0.00    0.  1.0  0.

< p 2 5

     x1  x2  x3  x4  x5   x6   x7
 3.  3.  0.  0.  0.  -4. -3.5  33.
 0.  4.  0.  0.  1. -32. -4.0  36.
 0. -2.  1.  0.  0.   4.  1.5 -15.
 1.  0.  0.  1.  0.   0.  1.0   0.

< p 3 6

     x1    x2    x3  x4  x5  x6     x7
 3.   1.0  1.00  0.  0.  0. -2.000  18.00
 0. -12.0  8.00  0.  1.  0.  8.000 -84.00
 0.  -0.5  0.25  0.  0.  1.  0.375  -3.75
 1.   0.0  0.00  1.  0.  0.  1.000   0.00

< p 2 7

     x1      x2     x3  x4       x5  x6  x7
 3. -2.0000  3.000  0. 0.250000  0.  0.  -3.0000
 0. -1.5000  1.000  0. 0.125000  0.  1. -10.5000
 0.  0.0625 -0.125  0. -.046875  1.  0.   0.1875
 1.  1.5000 -1.000  1. -.125000  0.  0.  10.5000

< p 3 8

     x1         x2         x3  x4    x5         x6  x7
 3. -1.0000000  1.0000000  0. -0.50  16.000000  0.  0.
 0.  2.0000000 -6.0000000  0. -2.50  56.000000  1.  0.
 0.  0.3333333 -0.6666667  0. -0.25   5.333333  0.  1.
 1. -2.0000000  6.0000000  1.  2.50 -56.000000  0.  0.

< p 2 2

     x1  x2         x3  x4         x5   x6         x7
 3.  0. -2.0000000  0. -1.7500000  44.  0.5000000  0.
 0.  1. -3.0000000  0. -1.2500000  28.  0.5000000  0.
 0.  0.  0.3333333  0.  0.1666667  -4. -0.1666667  1.
 1.  0.  0.0000000  1.  0.0000000   0.  1.0000000  0.

< p 3 3

     x1  x2  x3  x4    x5    x6   x7
 3.  0.  0.  0. -0.75   20. -0.5  6.
 0.  1.  0.  0.  0.25   -8. -1.0  9.
 0.  0.  1.  0.  0.50  -12. -0.5  3.
 1.  0.  0.  1.  0.00    0.  1.0  0.
```

This problem, which I will call `cycle` (see §28.5.14), is more interesting than it might appear, because there is a sequence of simplex-rule pivots that leads from its first tableau back to the same tableau without ever producing a final form. In this `pivot` session the columns are chosen by the most-negative pricing rule and when there is a tie for the minimum ratio the minimum ratio row having the smallest row index is chosen. We have used these rules to solve other problems, but following them here makes the algorithm **cycle** endlessly through the same six tableaus, none of which is the optimal form given on the previous page. This is not a result of roundoff error, and it happens even if we use exact rather than floating-point arithmetic.

To understand why the simplex algorithm fails to converge on this problem we need to investigate the circumstances in which it succeeds.

4.5.1 Simplex Algorithm Convergence

When we solve a linear program by the simplex algorithm, each pivot transforms one canonical-form tableau into another. Each canonical form can be uniquely identified by its basic sequence S. The number of possible basic sequences is the number of ways in which the columns of $\mathbf{I}_{m\times m}$ can be placed among the n variable columns of the tableau. That number is [3, p55]

$$q = \underbrace{\binom{n}{m}}_{\text{ways to choose } m \text{ columns from among } n \text{ [153, p13]}} \underbrace{m!}_{\text{ways to order a given set of } m \text{ columns [153, p10]}} = \frac{n!}{(n-m)!\,m!}\,m! = \frac{n!}{(n-m)!} = n(n-1)\cdots(n-[m-1]).$$

For example, if $n = 5$ and $m = 3$ then there are at most

$$q = \frac{5!}{(5-3)!} = \frac{(5)(5-1)(5-2)\cancel{(5-3)}\cancel{(5-4)}}{\cancel{(5-3)}\cancel{(5-4)}} = 5\times 4\times 3 = 60$$

possible basic sequences.

Each basic sequence determines a basic feasible solution and its objective value z. If each phase-2 pivot decreases z, then each must generate a different basic feasible solution and no basic sequence can repeat. If no basic sequence repeats, then because there are no more than q of them the simplex algorithm must converge in no more than q phase-2 pivots.

When can we be sure that z decreases with each phase-2 pivot performed by the simplex algorithm? This pivot in $\mathbf{T}_k$ yields the entries in $\mathbf{T}_{k+1}$ (I have assumed that numbers not shown on the left are appropriate for that tableau to be in canonical form, and they are of course also updated by the pivot). After dividing the pivot row by the pivot element we must add 3 times the new pivot row to the objective row to zero out the cost coefficient,

$$\mathbf{T}_k = \begin{array}{|c|c|}\hline 5 & -3 \\ \hline 2 & 4 \\ \hline \end{array} \xrightarrow[\text{pivot}]{} \begin{array}{|c|c|}\hline 5+\frac{2}{4}\times 3 & 0 \\ \hline \frac{2}{4} & 1 \\ \hline \end{array} = \mathbf{T}_{k+1}$$

The value of the upper-left entry in tableau $\mathbf{T}_k$ is $-z^k$ and after pivoting on a_{hp} the upper-left entry of tableau $\mathbf{T}_{k+1}$ is $-z^{k+1} = -z^k + \Delta z$ where

$$\Delta z = \frac{b_h}{a_{hp}} c_p.$$

For $\mathbf{T}_k$ to be in canonical form it must be that $b_h \geq 0$. For column p to be chosen as the pivot column it must be that $c_p < 0$. For row h to be chosen as the pivot row it must be that $a_{hp} > 0$. Thus $\Delta z \leq 0$ and the pivot reduces the objective provided that $b_h \neq 0$.

In our cycling example, $b_1 = b_2 = 0$ in every tableau, and the simplex algorithm pivots we performed *never* made the objective go down. If a problem has even one canonical form in which even one $b_i = 0$, it is said to be a **degenerate linear program**. The `graph` problem of §3.1 is degenerate because 3 hyperplanes intersect at vertex $\mathbf{E}$ of its feasible set, overdetermining the point in $\mathbb{R}^2$. In the guided tour of §3.2.2 we found 3 canonical-form tableaus representing that extreme point, each having $b_1 = 0$. A linear program in which every $b_i > 0$ in every canonical form tableau is said to be **nondegenerate**. If the most-negative pricing rule is used to select the pivot column and the smallest-row-index rule is used to break ties in the selection of the minimum-ratio pivot row, then the simplex algorithm is sure to converge only on problems that are nondegenerate.

4.5.2 Ways to Prevent Cycling

Cycling can be prevented by using more complicated rules to pick the pivot element at each phase-2 iteration of the simplex algorithm.

The **smallest-leaving-index rule** [16, §1] [107, Exercise 3.12.35] uses the first-negative pricing rule to pick the pivot column p. When the smallest b_i/a_{ip} with $a_{ip} > 0$ is unique, it picks that row as the pivot row h. If the minimum ratio occurs for more than one row, it selects for the pivot row the minimum ratio row for which the corresponding basic variable x_j (the variable that will leave the basis) has the lowest index j. The `pivot` session in the left column on the next page uses this rule to solve the `cycle` problem. The first three pivot positions determined by this rule are the same ones we used in §4.5.0, but now to resolve the tie in the fourth tableau we get to decide between row 2, for which the identity-column `1` is in the x_6 column and row 3, for which the identity-column `1` is in the x_5 column, and therefore pick row 3. The remaining pivots are uniquely determined. The optimal tableau is that given in §4.5.0 but with its constraint rows permuted.

The **successive-ratio rule** [3, p55-58] [145, §3.4] [38] permits any column having $c_k < 0$ (including one chosen by steepest-edge pricing) to be used as the pivot column p. When the smallest b_i/a_{ip} with $a_{ip} > 0$ is unique, it picks that row as the pivot row h. If the minimum ratio occurs for more than one row, it computes for each such row the successive ratios

$$\frac{b_i}{a_{ip}} \quad \frac{a_{i1}}{a_{ip}} \quad \frac{a_{i2}}{a_{ip}} \cdots \frac{a_{ip}}{a_{ip}} \cdots \frac{a_{in}}{a_{ip}}.$$

Then it compares the rows of successive ratios one column at a time from left to right, until a column is reached for which the successive ratio in one row is smallest. That **minimum successive-ratio row** is chosen as the pivot row. The `pivot` session in the right column on the next page uses this rule solve the `cycle` problem. The optimal tableau is that given in §4.5.0


```
> This is PIVOT, Unix version 4.3
> For a list of commands, enter HELP.
>
< read cycle.tab
Reading the tableau...
...done.

     x1  x2  x3  x4    x5    x6   x7
 3.  0.  0.  0. -0.75   20. -0.5  6.
 0.  1.  0.  0.  0.25   -8. -1.0  9.
 0.  0.  1.  0.  0.50  -12. -0.5  3.
 1.  0.  0.  1.  0.00    0.  1.0  0.

< p 2 5

     x1  x2  x3  x4  x5   x6   x7
 3.  3.  0.  0.  0.  -4. -3.5  33.
 0.  4.  0.  0.  1. -32. -4.0  36.
 0. -2.  1.  0.  0.   4.  1.5 -15.
 1.  0.  0.  1.  0.   0.  1.0   0.

< p 3 6

     x1    x2    x3  x4  x5  x6     x7
 3.   1.0  1.00  0.  0.  0. -2.000  18.00
 0. -12.0  8.00  0.  1.  0.  8.000 -84.00
 0.  -0.5  0.25  0.  0.  1.  0.375  -3.75
 1.   0.0  0.00  1.  0.  0.  1.000   0.00

< p 2 7

     x1      x2     x3  x4       x5  x6  x7
 3. -2.0000  3.000  0. 0.250000  0.  0.  -3.0000
 0. -1.5000  1.000  0. 0.125000  0.  1. -10.5000
 0.  0.0625 -0.125  0. -.046875  1.  0.   0.1875
 1.  1.5000 -1.000  1. -.125000  0.  0.  10.5000

< p 3 2

     x1  x2  x3  x4    x5   x6  x7
 3.  0. -1.  0. -1.25  32.  0.  3.
 0.  0. -2.  0. -1.00  24.  1. -6.
 0.  1. -2.  0. -0.75  16.  0.  3.
 1.  0.  2.  1.  1.00 -24.  0.  6.

< p 4 3

      x1  x2  x3   x4    x5    x6  x7
 3.5  0.  0.  0.5 -0.75   20.  0.  6.
 1.0  0.  0.  1.0  0.00    0.  1.  0.
 1.0  1.  0.  1.0  0.25   -8.  0.  9.
 0.5  0.  1.  0.5  0.50  -12.  0.  3.

< p 4 5

       x1  x2   x3    x4  x5   x6  x7
 4.25  0.  1.5  1.25  0.   2.  0.  10.5
 1.00  0.  0.0  1.00  0.   0.  1.   0.0
 0.75  1. -0.5  0.75  0.  -2.  0.   7.5
 1.00  0.  2.0  1.00  1. -24.  0.   6.0
```

```
> This is PIVOT, Unix version 4.3
> For a list of commands, enter HELP.
>
< read cycle.tab
Reading the tableau...
...done.

     x1  x2  x3  x4    x5    x6   x7
 3.  0.  0.  0. -0.75   20. -0.5  6.
 0.  1.  0.  0.  0.25   -8. -1.0  9.
 0.  0.  1.  0.  0.50  -12. -0.5  3.
 1.  0.  0.  1.  0.00    0.  1.0  0.

< p 3 5

     x1  x2   x3  x4  x5   x6    x7
 3.  0.  1.5  0.  0.   2. -1.25  10.5
 0.  1. -0.5  0.  0.  -2. -0.75   7.5
 0.  0.  2.0  0.  1. -24. -1.00   6.0
 1.  0.  0.0  1.  0.   0.  1.00   0.0

< p 4 7

       x1  x2   x3    x4  x5   x6  x7
 4.25  0.  1.5  1.25  0.   2.  0.  10.5
 0.75  1. -0.5  0.75  0.  -2.  0.   7.5
 1.00  0.  2.0  1.00  1. -24.  0.   6.0
 1.00  0.  0.0  1.00  0.   0.  1.   0.0
```

In the first tableau above I arbitrarily chose x_4 as the pivot column, so the successive ratios for rows 2 and 3 are

$$\begin{array}{cccccccc} \frac{0}{0.25} & \frac{1}{0.25} & \frac{0}{0.25} & \frac{0}{0.25} & \frac{0.25}{0.25} & \frac{-8}{0.25} & \frac{-1}{0.25} & \frac{9}{0.25} \\ \frac{0}{0.50} & \frac{0}{0.50} & \frac{1}{0.50} & \frac{0}{0.50} & \frac{0.50}{0.50} & \frac{-12}{0.50} & \frac{-0.5}{0.50} & \frac{3}{0.50} \end{array}$$

The entries in the first column of successive ratios are both 0, but in the second column $\frac{0}{0.50} < \frac{1}{0.25}$, so the pivot row is row 3. The second pivot is uniquely determined.

4.5.3 Degeneracy and Convergence in Practice

In the simplex algorithm implementation of §4.1, `phase2.m` picks 12-20 the variable column with the most-negative cost as the pivot column. Then for the pivot row `minr.m` picks 12-16 the minimum-ratio constraint row, using 13 the row with the smallest row index if there is a tie. As we have seen, these rules permit cycling.

To implement either the smallest-leaving-index or the successive-ratio anti-cycling rule it is necessary to identify the constraint rows that are tied for the minimum ratio. We might list them explicitly by calculating the row ratios for all possible pivot rows, sorting them into ascending order, and searching the sorted list for the first value greater than the preceding one; the rows corresponding to the identical ratios that appear before that first greater one would then be the tied minimum-ratio rows. This approach is conceptually simple, but it is expensive because the work of sorting m numbers grows at least as fast as $m \log_2(m)$ [95, §5.3.1] and as m^2 for naïve methods like bubble sort.

It is faster to calculate the row ratios for all possible pivot rows, search for their minimum, and simply rule out the rows that have ratios higher than that. The routines listed on the next two pages both take this approach, setting `flag(i)=0` to indicate that a constraint row has been ruled out or leaving `flag(i)=1` if the row is still a candidate to be chosen for the pivot. Each of these routines is meant to replace `minr.m` in the `phase2.m` subroutine of `simplex.m`, so it is necessary to address the rows of the tableau `T` indirectly using `tr`. Recall from §4.1 that this vector contains the indices of the rows of `T` that are in the current problem or subproblem. It will be easier to understand how `smind.m` and `srr.m` work if you assume for now that `tr(i)=i` and that `mr` is the number $m+1$ of rows in `T`.

First consider `smind.m`, which implements the smallest-leaving-index rule. The code begins by 4 setting the zero tolerance `ztol` and 5 initializing `flag` to a vector of m ones. This makes all of the constraint rows candidates for selection as the pivot row. The second stanza 7-20 finds the row ratios `r(i)` for all possible pivot rows, and their minimum `rmin`. In the process it 11-14 rules out rows that cannot be the pivot row because the pivot-column element is not positive, and 18 sets `ip` to the index of the first minimum-ratio row. The third stanza 22-31 rules out any remaining row whose ratio is greater than `rmin` 26-27, or counts the tie 28-30 if the ratio is equal to `rmin`. The fourth stanza 33-34 returns the pivot row `ip` that was set earlier 18 if that row alone has the minimum ratio.

The final stanza 36-50 finds the minimum-ratio row whose identity-column `1` has the lowest column index `idxmin`. It initializes `idxmin` to `nn` $= n+1$, which is greater than the highest column index in $\mathbf{A}$. Then it 38-50 examines each constraint row. If the row has been excluded previously 39 it is skipped; otherwise it is one of the tied rows. Recall that `S(j)` is zero if x_j is nonbasic or the row index in $\mathbf{A}$ of the identity-column `1` if x_j is basic. The loop over `jj` 40-45 searches the basis vector `S` to determine the column index `idx` in $\mathbf{A}$ of the identity-column `1` that is in this row. That index `idx` is 46-49 compared to the lowest index `idxmin` found so far; if it is lower `idxmin` is replaced and the pivot row `ip` is set to the current row of `T`. At the end of this process the routine returns the last value set for `ip`.


```
 1 function ip=smind(T,tr,mr,jp,nn,S)
 2 % find the pivot row using the smallest-leaving-index rule
 3
 4   ztol=1e-6;
 5   flag=ones(1,mr-1);
 6
 7 % find the row ratios and their minimum
 8   rmin=realmax;
 9   ip=0;
10   for i=1:mr-1
11       if(T(tr(i+1),jp) <= ztol)
12          flag(i)=0;
13          continue
14       end
15       r(i)=T(tr(i+1),1)/T(tr(i+1),jp);
16       if(r(i) < rmin)
17          rmin=r(i);
18          ip=tr(i+1);
19       end
20   end
21
22 % rule out non-min-ratio rows and count min-ratio ties
23   tied=0;
24   for i=1:mr-1
25       if(flag(i) == 0) continue; end
26       if(abs(r(i)-rmin) > ztol)
27          flag(i)=0;
28       else
29          tied=tied+1;
30       end
31   end
32
33 % accept the minimum ratio row if it is unique
34   if(tied == 1) return; end
35
36 % among min ratio rows pick the one whose 1 has lowest col index
37   idxmin=nn;
38   for i=1:mr-1
39       if(flag(i) == 0) continue; end
40       for jj=2:nn
41           if(S(jj-1) == i+1)
42             idx=jj-1;
43             break
44           end
45       end
46       if(idx < idxmin)
47          idxmin=idx;
48          ip=tr(i+1);
49       end
50   end
51
52 end
```

Next consider `srr.m`, which implements the successive-ratio rule. The first stanza of this code 4-5 is identical to that of `smind.m`. Then comes an outer loop 7-39 over the columns `jr` of `T`, containing three stanzas that are almost identical to the second, third, and fourth stanzas of `smind.m`. In the first pass of this loop `jr=1`, so $T_{2,1} = b_1$, $T_{3,1} = b_2 \ldots T_{(m+1),1} = b_m$ are used 19 in computing the row ratios $b_i/a_{i,\mathrm{jp}}$. If only one row has the minimum ratio

```
 1 function ip=srr(T,tr,mr,jp,nn)
 2 % find the pivot row using the successive-ratio rule
 3
 4   ztol=1e-6;
 5   flag=ones(1,mr-1);
 6
 7 % use successive columns to form the row ratios
 8   for jr=1:nn
 9
10 %     find the row ratios and their minimum
11       rmin=realmax;
12       ip=0;
13       for i=1:mr-1
14           if(flag(i) == 0) continue; end
15           if(T(tr(i+1),jp) <= ztol)
16              flag(i)=0;
17              continue
18           end
19           r(i)=T(tr(i+1),jr)/T(tr(i+1),jp);
20           if(r(i) < rmin)
21              rmin=r(i);
22              ip=tr(i+1);
23           end
24       end
25
26 %     rule out non-min-ratio rows and count ties
27       tied=0;
28       for i=1:mr-1
29           if(flag(i) == 0) continue; end
30           if(abs(r(i)-rmin) > ztol)
31              flag(i)=0;
32           else
33              tied=tied+1;
34           end
35       end
36
37 %     accept the minimum ratio row if it is unique
38       if(tied == 1) return; end
39   end
40 end
```

the routine 38 returns with `ip` set 22 to the index of that row. Otherwise the outer loop advances `jr` to 2 and the process is repeated using $T_{2,2} = a_{1,1}$, $T_{3,2} = a_{2,1} \ldots T_{(m+1),2} = a_{m,1}$ in computing the row ratios $a_{i,1}/a_{i,\mathtt{jp}}$. If there is still no unique minimum ratio, `jr` is increased again and again, stepping across the columns of $\mathbf{A}$, until there is.

When there are ties in the minimum ratio, `smind.m` does the extra work of its final stanza while `srr.m` does the extra work of repeating the stanzas in the body of its `jr` loop. Which takes more processor cycles, and which choice of pivot row yields faster convergence of the simplex algorithm [4, p166] depends on the particulars of the problem being solved. But both `smind.m` and `srr.m` clearly do more work than `minr.m` even when the minimum ratio is unique. How necessary is it for a production code to defend against the possibility of cycling, and is there some less-expensive way to do that?

Almost all real linear programming models are degenerate, but for many years only a few had been discovered that cycle [11] [82] [159]. Even if several vertices of the feasible

polyhedron are degenerate, the simplex algorithm might never encounter one in pivoting from an initial vertex to an optimal vertex. If a degenerate vertex is encountered, the worst consequence is usually that a few degenerate pivots are needed before the algorithm can move on (to avoid such **stalling** is one reason that some preprocessors try to remove redundant constraints). Unfortunately, linear programming relaxations of integer programs (see §7.3) *are* frequently observed to cause cycling [5, p381] so to accommodate this important special class of problems most production linear programming codes do somehow guard against it. Several strategies can be used to minimize the cost of this prudence.

The smallest-leaving-index or successive-ratio rule can be used, instead of the ordinary minimum-ratio rule, just when the current tableau has some $b_i = 0$ so that its basic feasible solution corresponds to a degenerate vertex [4, p167].

When the current tableau has some $b_i = 0$, the constant-column entries corresponding to the constraints that intersect there can be perturbed slightly to make the vertex nondegenerate [5, p381-382]. The unique minimum ratio row can then be used for the pivot row, and a postprocessing step can remove the perturbation to ensure that the reported $\mathbf{x}^\star$ is optimal for the original problem. The nonzero value of `ztol` [4, §7.6.3] or unintentional roundoff errors [63, p182] can through perturbation render nondegenerate a problem that would be degenerate in perfect arithmetic or, much less likely, render degenerate a problem that would be nondegenerate in perfect arithmetic.

After tied rows have been identified by using code like the first four stanzas of `smind.m`, one of the tied rows can be chosen at random [145, p93]; this is less expensive than either the full smallest-leaving-index rule or the successive-ratio rule, and often prevents cycling.

The crudest strategy is to fix an upper limit on phase-2 iterations and simply resign with an error message in the unlikely event that a problem exceeds that limit because of cycling.

What is a practical limit to set on the iterations used by the simplex algorithm? In §4.5.1 we found that in solving a nondegenerate problem it must converge in no more than $q = n!/(n-m)!$ iterations, and the `phase2.m` routine of §4.1 tries to use that theoretical maximum for its `kmax`. But even for small problems this is an enormous number. For example, if $n = 20$ and $m = 10$ then $q \approx 6.7 \times 10^{11}$ pivots are needed in the worst case. For a problem of that size `phase2.m` has to settle for making `kmax=2147483645`, the highest integer allowed by MATLAB as a `for`-loop limit. If the algorithm actually needed all of those iterations to solve real problems it would not be a practical computational tool.

In the worst case the number of phase-2 pivots needed by the simplex *algorithm* grows exponentially with the size of the problem (see §7.9) and examples have been contrived [93] to exhibit this by forcing it to visit every vertex of the feasible set. However, the linear programming *problem* can be solved using other methods that require an amount of work that grows only polynomially with the size of the problem [92]. Because linear programming is easy in this sense, the simplex method almost always exhibits much better performance than it does in the worst case. In solving real problems the number of phase-2 iterations needed is [107, p59] on the order of $1.5m$, independent of n. (The `pivot` program allows up to 30 constraints so to be generous its `SOLVE` command sets a default iteration limit of 60.)

The interior-point methods for linear programming that we will study in §21.1 cannot cycle because of degeneracy and do not have the exponential worst-case time complexity of the simplex algorithm, but degeneracy causes them other difficulties and they are faster in practice only for very large problems. Thus the simplex method is widely used despite its theoretical shortcomings.

4.6 Exercises

4.6.1[E] Outline the process described in §2.6 for solving a linear program. What parts of the process can be automated? What parts *must* be automated in order for the solution process to be practical?

4.6.2[P] We implemented the simplex algorithm in MATLAB in the routine `simplex.m` and its subroutines, which are described in §4.1. (a) Why is it useful to understand this code? (b) Draw a block diagram showing the main components of `simplex.m`, how they are connected, and what they do. (c) List the possible values of the return code `rc` from `simplex.m` and explain the meaning of each.

4.6.3[P] If `simplex.m` delivers a return code of `rc=4`, what do we know about the optimal objective value of the linear program it is being used to solve?

4.6.4[P] The `simplex.m` routine described in §4.1 uses a vector named `tr`. (a) What do the numbers in this vector indicate? (b) Why is it useful to introduce this vector? (c) How many components does `tr` have?

4.6.5[P] The `simplex.m` routine described in §4.1 uses a vector named `S`. (a) How many components does this vector have? (b) What do the numbers in `S` mean? (c) What does it indicate if all of the entries in `S` are zero? (d) Can an element of `S` ever be 1? Explain.

4.6.6[P] Where in `simplex.m` are redundant rows excluded from the problem? How is that done? What happens to the redundant rows?

4.6.7[P] When `simplex.m` is used to solve a linear program that is infeasible, where is the infeasibility detected? Explain.

4.6.8[P] Explain the role of the *zero tolerance* `ztol` in `simplex.m` and its subroutines.

4.6.9[P] The `newseq.m` routine of §4.1 pivots-in a basis. (a) Explain how the routine works. (b) What does it do if it is invoked with a basis already present in `T`? Explain. (c) Under what circumstances does the routine return a nonzero return code `rc0`? (d) Why is it convenient in this routine to process the rows of the tableau sequentially with a MATLAB `while` construct, rather than with a `for` loop?

4.6.10[P] In the `phase1.m` routine of §4.1, how are the subproblems solved?

4.6.11 [P] For simplicity the `phase1.m` routine refrains from exploiting every possible efficiency in the subproblem technique. (a) Does the loop over `p` [35-73] ever need to be performed `mm-1` times? (b) Does that loop ever exit through its [73] `end` statement? (c) Is it necessary to solve every subproblem [54] all the way to optimality? (d) Modify `phase1.m` to make it faster in all the ways that you can think of, and test the resulting code.

4.6.12 [P] Write a `phase1.m` routine that has the same calling sequence as the `newseq` routine of §4.1 but uses the method of artificial variables instead of the subproblem technique. (a) How much additional array storage is required to use this approach? (b) Revise `simplex.m` to use it, and show that the resulting code still solves the `brewery` problem.

4.6.13 [P] What *pricing rule* does `phase2.m` use? If two columns have the same negative reduced cost, which one is chosen as the pivot column?

4.6.14 [P] What does `phase2.m` do if `kmax` pivots are performed without discovering a final form? How is the value of `kmax` determined? Explain.

4.6.15 [P] If two tableau rows have the same minimum ratio $b_i/a_{i,\mathtt{jp}}$, which row's index does `minr.m` return for `ip`? How does the routine signal to its caller that it has discovered the problem is unbounded?

4.6.16 [P] How many additions, subtractions, multiplications, and divisions are performed by the `pivot.m` function of §2.4.2 in carrying out one pivot in a canonical-form tableau that has $m+1$ rows and $n+1$ columns?

4.6.17 [E] Explain the basic idea of the *revised simplex method.*

4.6.18 [H] The tableaus shown below are one pivot apart. Write down a *pivot matrix* $\mathbf{Q}$ such that the matrix product $\mathbf{QT_1}$ produces $\mathbf{T}_2$. Circle the pivot element in $\mathbf{T}_1$.

$$\mathbf{T}_1 = \left[\begin{array}{r|rrrr} -3 & 0 & 1 & 0 & -2 \\ \hline 3 & 1 & 1 & 0 & 1 \\ 2 & 0 & -4 & 1 & 2 \end{array}\right] \xrightarrow[\text{1 pivot}]{} \left[\begin{array}{r|rrrr} -6 & -1 & 0 & 0 & -3 \\ \hline 3 & 1 & 1 & 0 & 1 \\ 14 & 4 & 0 & 1 & 6 \end{array}\right] = \mathbf{T}_2$$

4.6.19 [H] Construct a pivot matrix to pivot on the circled element in the following tableau, use it to calculate the next tableau, and confirm that the result is the same as you get by performing the pivot.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	−90	−150	−60	−70	0	0	0
160	7	10	8	12	1	0	0
50	1	3	1	1	0	1	0
60	2	4	1	3	0	0	1

4.6.20 [H] Starting with tableau $\mathbf{T}$, pivot matrices $\mathbf{Q}_1$, $\mathbf{Q}_2$, and $\mathbf{Q}_3$ are used in that order to carry out three pivots. (a) Does the order affect the result? (b) What matrix $\mathbf{P}$ would $\mathbf{T}$ have to be premultiplied by to produce the tableau resulting from the three pivots?

4.6.21 [E] If a tableau $\mathbf{T}$ has basic sequence $S = (x_4, x_2, x_6)$ and we pivot on a_{27}, what will be the new basic sequence? Construct an example to illustrate your answer.

4.6.22 [E] A linear program has the initial and optimal tableaus shown below. If the pivot matrix $\mathbf{P}$ performs pivots so that $\mathbf{T}^\star = \mathbf{PT}$, explain how to write down $\mathbf{P}$ by inspection of the two tableaus.

$$\mathbf{T}_0 = \left[\begin{array}{c|ccccc} 0 & -6 & -5 & -3 & 0 & 0 \\ \hline 50 & 1 & 1 & 0 & 1 & 0 \\ 150 & 2 & 1 & 2 & 0 & 1 \end{array}\right]$$

$$\mathbf{T}^\star = \left[\begin{array}{c|ccccc} 400 & \frac{1}{2} & 0 & 0 & \frac{7}{2} & \frac{3}{2} \\ \hline 50 & 1 & 1 & 0 & 1 & 0 \\ 50 & \frac{1}{2} & 0 & 1 & -\frac{1}{2} & \frac{1}{2} \end{array}\right]$$

4.6.23 [E] If $\mathbf{Q}$ is a pivot matrix and $\mathbf{T}$ is a tableau, then the product $\mathbf{QT}$ is a new tableau that results from performing the pivot on $\mathbf{T}$. If $\mathbf{T}$ has a basis and the pivot is in a nonbasic column, the matrix multiplication changes the basic sequence. What happens if $\mathbf{T}$ does *not* have a basis? Explain.

4.6.24 [H] A linear program has the following canonical-form tableau.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	0	0	−2	7	2	5	0
80	0	0	4	4	1	−1	1
110	0	1	−1	1	3	1	0
20	1	0	2	3	−4	2	0

Use the phase-2 algorithm of §4.2.3 to solve the problem, filling in only those elements of each tableau that are necessary to determine the next pivot position.

4.6.25 [E] There are two ways in which the modified-simplex approach described in §4.2.2 and §4.2.3 can be used to obtain an initial canonical form. What are they?

4.6.26 [H] In the *matrix simplex method* the constraints of the linear program are expressed in the form $\mathbf{b} = \mathbf{N}\mathbf{x}_N + \mathbf{B}\mathbf{x}_B$. (a) What is the matrix $\mathbf{B}$ called, and what are its dimensions? (b) Write down a formula for the basic variables $\mathbf{x}_B$ at a basic feasible solution. (b) If a nonbasic variable is increased from zero, how must the basic variables change in order to maintain feasibility?

4.6.27 [E] What is the main computational advantage that the matrix simplex method has over the tableau simplex method?

4.6.28 [H] In the matrix simplex method the objective of the linear program is written in the form $z = \mathbf{c}_N^\top \mathbf{x}_N + \mathbf{c}_B^\top \mathbf{x}_B$. (a) Suppose a nonbasic variable is increased from zero and the basic variables are adjusted to maintain feasibility. Write down a formula for z in terms of only $\mathbf{x}_N$ and the data of the problem. (b) How can the formula for z in terms of $\mathbf{x}_N$ be used to select a variable to enter the basis? Explain.

4.6.29 [H] In §4.2.5 the matrix simplex method is used to solve the `brewery` problem by pivoting in the x_1 column first. Use the matrix simplex method to solve the `brewery` problem by pivoting in the x_2 column first.

4.6.30 [H] Solving a linear program consists of finding the best set of $\mathbf{A}$ columns to have in the basis, or the best m of the n variables to allow to be nonzero. The `subopt.m` program of §3.6.2 lists all of the basic feasible solutions of the `brewery` problem, in each of which 3 of the 7 variables are basic. (a) How many ways are there to select 3 of the 7 variables to be basic? (b) Why are there fewer basic feasible solutions than that?

4.6.31 [E] Describe three refinements of the matrix simplex method that are commonly used to facilitate the solution of large problems.

4.6.32 [H] When the nonzero constraint coefficients of a large linear program can be arranged in a *block-angular structure* with p blocks, it is possible to decompose the problem into a master problem and p subproblems. (a) What do the variables in the master problem represent? (b) How big is the master problem? (c) How are the subproblems used in solving the master problem by the matrix simplex algorithm?

4.6.33 [E] Describe three different *pricing rules* that can be used in selecting a variable to enter the basis in the matrix simplex algorithm.

4.6.34 [E] Explain the difference between *full* and *partial* pricing. What is a *candidate list*?

4.6.35 [H] What is the relationship between the zero tolerance used in a simplex algorithm implementation and the scaling of the rows and columns in a linear program?

4.6.36 [E] Many optimization codes offer some sort of *preprocessing*. (a) What benefits can result from preprocessing a linear program before attempting its solution by the simplex method? (b) Describe one kind of transformation that a preprocessor can do. (c) If preprocessing an original linear program LP0 results in the transformed linear program LP1, might it be worthwhile to preprocess LP1 and produce LP2 before solving LP2? Explain.

4.6.37 [E] In §4.4.4 some folklore is collected about using *black-box solvers*. (a) Summarize this advice. (b) Use a commercial solver of your choice to solve the `brewery` problem, and describe your experience.

4.6.38 [E] Does the simplex algorithm described in §2 always converge? Explain.

4.6.39 [H] Every linear program has a finite number of basic sequences. (a) How many basic sequences q can there be if the linear program has n variables and m constraints? (b) Does every linear program with n variables and m constraints have q basic sequences? If so, prove it; if not, present a counterexample.

4.6.40 [H] Show that

$$\frac{n!}{(n-m)!} = n(n-1)\cdots(n-[m-1]).$$

4.6.41 [E] What makes a linear program *nondegenerate*? Are most linear programs that result from real applications nondegenerate?

4.6.42 [H] If a linear program is nondegenerate, how do we know that the simplex algorithm of §2 will solve it without *cycling*?

4.6.43 [H] Show that the linear program

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

is nondegenerate if and only if (a) every set of m columns chosen from the matrix $[\mathbf{A}, \mathbf{b}]$ is linearly independent; (b) in every basic feasible solution no basic variable is zero.

4.6.44 [E] If a linear program has multiple optimal points, a pivot can leave z unchanged. (a) Does this indicate that the problem is degenerate? (b) Why does this not affect the convergence of the simplex algorithm?

4.6.45 [H] The `graph` problem of §3.1 is degenerate at vertex $\mathbf{E}$ of its feasible set, where 3 constraint hyperplanes intersect. (a) Slightly change the right-hand sides of those constraints so that the problem is not degenerate. How does this affect the optimal point? (b) Explain how the true $\mathbf{x}^\star$ can be recovered from the perturbed solution.

4.6.46 [H] Find the next pivot position in the following tableau by using (a) the *smallest-leaving-index rule;* (b) the *successive-ratio rule.*

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
3	$-\frac{1}{2}$	0	$-\frac{3}{4}$	0	6	0	20	0
0	-1	0	$\frac{1}{4}$	1	9	0	-8	0
1	3	0	2	0	5	0	6	1
1	1	0	0	0	0	1	0	0
0	$-\frac{1}{2}$	1	$\frac{1}{2}$	0	3	0	-12	0

4.6.47 [H] In applying the successive-ratio rule [3, p54] can it ever happen that all of the successive ratios for two tied candidate pivot rows come out the same? If so, provide an example; if not; explain why not.

4.6.48 [P] When the degenerate linear program of §4.5.0 is solved in §4.5.2 by pivoting according to the successive-ratio rule, the first pivot is arbitrarily chosen to be in the x_4 column. Solve the problem by pivoting according to the successive-ratio rule and making the first pivot in the x_6 column.

4.6.49 [H] If we perform phase 1 of the simplex algorithm by pivoting-in a basis and then solving subproblems to make $\mathbf{b} \geq \mathbf{0}$, might the simplex algorithm cycle in solving a subproblem? If not, explain why not; if so, how can such cycling be prevented?

4.6.50 [P] The `smind.m` and `srr.m` routines of §4.5.3 test for the equality of `r(i)` and `rmin` by comparing `abs(r(i)-rmin)` to `ztol`. Why is this subterfuge necessary?

4.6.51 [E] The `smind.m` and `srr.m` routines of §4.5.3 use the same approach to identify constraint rows that are tied for the lowest ratio. (a) Describe the three steps that comprise this approach. (b) What role does the vector `flag` play? (c) Why is `flag` initialized 5 to a vector of all `1`s? (d) Why is it necessary for these routines to address the rows of `T` indirectly by means of the vector `tr`?

4.6.52 [P] Explain how this code excerpt from `smind.m` finds, for each minimum-ratio row in `T`, the column index in **A** of the row's identity-column `1`.

```
38   for i=1:mr-1
39       if(flag(i) == 0) continue; end
40       for jj=2:nn
41           if(S(jj-1) == i+1)
42              idx=jj-1;
43              break
44           end
45       end
         :
50   end
```

4.6.53 [P] In `srr.m` the three stanzas that constitute the body of the `jr` loop 8-39 get executed repeatedly until the second stanza 26-35 produces `tied=1`. (a) What role does the index `jr` play in this process? (b) Why does `tied` eventually become `1`? (c) Why can `flag(i)` elements that are set to `0` in one iteration of the `jr` loop be kept at `0` for subsequent iterations?

4.6.54 [P] When the minimum ratio is unique, the same pivot row `ip` is returned by `minr.m`, `smind.m`, and `srr.m`, but `minr.m` does less work. (a) Count the elementary operations performed by each routine in finding `ip`, in terms of the tableau dimensions `mr` and `nn`. (b) What performance penalty is incurred by using `smind.m` or `srr.m` to solve problems that are nondegenerate?

4.6.55 [E] What is *stalling* and how can it be prevented?

4.6.56 [E] If most real linear programming models are degenerate, why do so few of them make the simplex algorithm of §2 cycle? Name one class of models that is more likely than others to do that.

4.6.57 [E] Describe four strategies that prevent cycling or make it less likely but impose less of a performance penalty than using the smallest-leaving-index or successive-ratio rule at each phase 2 iteration. What are the drawbacks of these strategies?

4.6.58 [E] If the simplex algorithm's worst-case time complexity is exponential, why is its average-case time complexity polynomial? How many iterations does it typically use in solving a problem that has n variables and m constraints? Why is the simplex method widely used if there are interior-point methods that do not suffer from its theoretical shortcomings?

4.6.59 [P] The `simplex.m` implementation of §4.1 assumes that a final form will be found by `phase2.m` in fewer than `kmax` iterations, but this ignores the possibility of cycling. Try `simplex.m` on the `cycle` problem of §4.5. This should elicit the message

```
warning: phase2: some elements in list of return values are undefined
```

and print a final tableau that is the same as the initial one. (a) What element in the list of return values from `phase2.m` is undefined, and why? (b) How many iterations does `phase2.m` perform to produce this final tableau? Hint: 210/6=35. (c) Revise `phase2.m` to set `rc2=1` if `kmax` iterations are used because the routine did not execute either `return` $\boxed{\texttt{24,30}}$. Now `rc2` is zero if optimal form is obtained, or `jp>1` if the problem is unbounded in variable column `jp`, or `1` if the allowed iterations were exhausted. (d) Revise `phase1.m` and `simplex.m` to deal in some sensible way with the new return code `rc2=1`. What should these routines do if `phase2.m` fails to converge? (e) Test your revised code on the `cycle` problem. Does it somehow alert you to the presence of cycling? (f) Would this be a practical way to detect cycling in a problem that had, say, $n = 20$ variables and $m = 10$ constraints? Explain. (g) Why do you suppose I ignored the possibility of cycling in the version of the code that is presented in §4.1? (h) Revise `phase2.m` to invoke `smind.m` instead of `minr.m`, and use the resulting code to solve the `cycle` problem. Remember that the smallest-leaving-index rule requires first-negative pricing to select the pivot column. (i) Revise `phase2.m` to invoke `srr.m` instead of `minr.m`, and use the resulting code to solve the `cycle` problem.

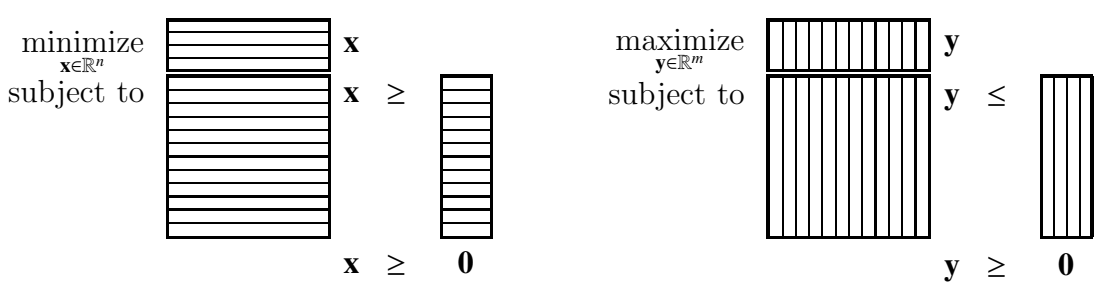
5

Duality and Sensitivity Analysis

Of the many enchantments that suffuse the theory of linear programming, perhaps the most powerful is the deep connection between problems that are **duals** of one another. In a pair like the one shown below, which I will call `dp1`, the problems have a structural relationship because the same coefficients appear in different roles. As pictured in the **standard dual pair** at the bottom, the cost vector in the minimization or **x problem** is the constant vector in the maximization or **y problem**, the constant vector in the **x** problem is the cost vector in the **y** problem, and the constraint coefficient matrices are transposes of one another.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & 6x_1 + 3x_2 + 2x_3 & & \\ \text{subject to} & x_1 \qquad + \ x_3 & \geq & 2 \\ & 2x_1 + 2x_2 - 2x_3 & \geq & 1 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad \begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^2}{\text{maximize}} & 2y_1 + y_2 & & \\ \text{subject to} & y_1 + 2y_2 & \leq & 6 \\ & 2y_2 & \leq & 3 \\ & y_1 - 2y_2 & \leq & 2 \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & \begin{bmatrix} 6 & 3 & 2 \end{bmatrix}\mathbf{x} & & \\ \text{subject to} & \begin{bmatrix} 1 & 0 & 1 \\ 2 & 2 & -2 \end{bmatrix}\mathbf{x} & \geq & \begin{bmatrix} 2 \\ 1 \end{bmatrix} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad \begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^2}{\text{maximize}} & \begin{bmatrix} 2 & 1 \end{bmatrix}\mathbf{y} & & \\ \text{subject to} & \begin{bmatrix} 1 & 2 \\ 0 & 2 \\ 1 & -2 \end{bmatrix}\mathbf{y} & \leq & \begin{bmatrix} 6 \\ 3 \\ 2 \end{bmatrix} \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$



Later we will derive the **y** problems that correspond to **x** problems stated using various names for the cost vector, constant vector, and constraint coefficient matrix, so it is best to remember the relationship between the problems in this pictorial way. Often it will be convenient to call one of the problems in a dual pair the **primal problem** $\mathscr{P}$ and the other the **dual problem** $\mathscr{D}$, but since each is the dual of the other the choice of which to call what is purely aesthetic.

5.1 Algebraic Duality Relations

The structural relationship between the problems of a dual pair gives rise to mathematical connections between them. To explore these it is convenient to consider this particular pair [3, §5.1], but because any dual pair can be written in this way our discoveries will apply (see Exercise 5.5.33) to all of them.

$$\begin{array}{rrcl} \mathscr{P}: & \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & \\ & \text{subject to} & \mathbf{A}\mathbf{x} & \geq \mathbf{b} \\ & & \mathbf{x} & \geq \mathbf{0} \end{array} \qquad \begin{array}{rrcl} \mathscr{D}: & \underset{\mathbf{y}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{b}^\top\mathbf{y} & \\ & \text{subject to} & \mathbf{A}^\top\mathbf{y} & \leq \mathbf{c} \\ & & \mathbf{y} & \geq \mathbf{0} \end{array}$$

5.1.1 Both Problems Infeasible

If $\mathbf{c}^\top = [-1]$, $\mathbf{b} = [1]$, and $\mathbf{A} = [0]$ then the problems are these.

$$\begin{array}{rrcl} \mathscr{P}: & \underset{x\in\mathbb{R}^1}{\text{minimize}} & -1x & \\ & \text{subject to} & 0x & \geq 1 \\ & & x & \geq 0 \end{array} \qquad \begin{array}{rrcl} \mathscr{D}: & \underset{y\in\mathbb{R}^1}{\text{maximize}} & 1y & \\ & \text{subject to} & 0y & \leq -1 \\ & & y & \geq 0 \end{array}$$

No value of x can make $0x \geq 1$ and no value of y can make $0y \leq -1$, so in a dual pair

$$\boxed{\text{it is possible for both problems to be infeasible.}}$$

5.1.2 Both Problems Feasible

If $\bar{\mathbf{x}}$ is feasible for the minimization and $\bar{\mathbf{y}}$ is feasible for the maximization then from the constraints of the two problems

$$\begin{array}{rcl} \left.\begin{array}{rcl} \mathbf{c} & \geq & \mathbf{A}^\top\bar{\mathbf{y}} \\ \bar{\mathbf{x}} & \geq & \mathbf{0} \end{array}\right\} \Rightarrow \bar{\mathbf{x}}^\top\mathbf{c} & \geq & \bar{\mathbf{x}}^\top(\mathbf{A}^\top\bar{\mathbf{y}}) \\ \bar{\mathbf{x}}^\top\mathbf{c} & \geq & (\mathbf{A}\bar{\mathbf{x}})^\top\bar{\mathbf{y}} \\ \bar{\mathbf{x}}^\top\mathbf{c} & \geq & \bar{\mathbf{y}}^\top\mathbf{A}\bar{\mathbf{x}} \\ \left.\begin{array}{rcl} \mathbf{A}\bar{\mathbf{x}} & \geq & \mathbf{b} \\ \bar{\mathbf{y}} & \geq & \mathbf{0} \end{array}\right\} \Rightarrow \bar{\mathbf{y}}^\top(\mathbf{A}\bar{\mathbf{x}}) & \geq & \bar{\mathbf{y}}^\top\mathbf{b}. \end{array}$$

Thus $\bar{\mathbf{x}}^\top\mathbf{c} \geq \bar{\mathbf{y}}^\top\mathbf{A}\bar{\mathbf{x}} \geq \bar{\mathbf{y}}^\top\mathbf{b}$ so

$$\boxed{\text{if } \bar{\mathbf{x}} \text{ is feasible for the min problem and } \bar{\mathbf{y}} \text{ is feasible for the max problem then } \mathbf{c}^\top\bar{\mathbf{x}} \geq \mathbf{b}^\top\bar{\mathbf{y}}.}$$

This means that $\mathbf{c}^\top\bar{\mathbf{x}}$ is an upper bound on $\mathbf{b}^\top\mathbf{y}$ for any $\mathbf{y}$ that is feasible for the max problem, and $\mathbf{b}^\top\bar{\mathbf{y}}$ is a lower bound on $\mathbf{c}^\top\mathbf{x}$ for any $\mathbf{x}$ that is feasible for the min problem. Therefore

$$\boxed{\text{if both problems are feasible then neither is unbounded.}}$$

In the `dp1` example we began with, both problems are feasible so neither is unbounded. We can find their optimal vectors by reformulating them into standard form, constructing an initial tableau for each, and pivoting by the simplex algorithm.

$$\begin{array}{rrcl} \mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & 6x_1 + 3x_2 + 2x_3 & = & z_{\mathbf{x}} \\ \text{subject to} & x_1 \qquad + x_3 & \geq & 2 \\ & 2x_1 + 2x_2 - 2x_3 & \geq & 1 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

$$\begin{array}{rrcl} \mathscr{D}: \underset{\mathbf{y}\in\mathbb{R}^2}{\text{maximize}} & 2y_1 + y_2 & = & z_{\mathbf{y}} \\ \text{subject to} & y_1 + 2y_2 & \leq & 6 \\ & 2y_2 & \leq & 3 \\ & y_1 - 2y_2 & \leq & 2 \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3\ \mathbf{s}\in\mathbb{R}^2}{\text{minimize}} & 6x_1 + 3x_2 + 2x_3 & & \\ \text{subject to} & -x_1 \qquad - x_3 + s_1 & = & -2 \\ & -2x_1 - 2x_2 + 2x_3 + s_2 & = & -1 \\ & \mathbf{x}, \mathbf{s} & \geq & \mathbf{0} \end{array}$$

$$\begin{array}{rrcl} \underset{\mathbf{y}\in\mathbb{R}^2\ \mathbf{w}\in\mathbb{R}^3}{\text{minimize}} & -2y_1 - y_2 & & \\ \text{subject to} & y_1 + 2y_2 + w_1 & = & 6 \\ & 2y_2 + w_2 & = & 3 \\ & y_1 - 2y_2 + w_3 & = & 2 \\ & \mathbf{y}, \mathbf{w} & \geq & \mathbf{0} \end{array}$$

```
< read primal.tab
Reading the tableau...
...done.

     x1  x2  x3  s1  s2
 0.  6.  3.  2.  0.  0.
-2. -1.  0. -1.  1.  0.
-1. -2. -2.  2.  0.  1.

< * get canonical form
< pivot 2 2

      x1  x2  x3  s1  s2
-12.  0.  3. -4.  6.  0.
  2.  1.  0.  1. -1.  0.
  3.  0. -2.  4. -2.  1.

< pivot 3 4

       x1  x2   x3  s1   s2
-9.00  0.  1.0  0.  4.0  1.00
 1.25  1.  0.5  0. -0.5 -0.25
 0.75  0. -0.5  1. -0.5  0.25
```

```
< read dual.tab
Reading the tableau...
...done.

     y1  y2  w1  w2  w3
 0. -2. -1.  0.  0.  0.
 6.  1.  2.  1.  0.  0.
 3.  0.  2.  0.  1.  0.
 2.  1. -2.  0.  0.  1.

< pivot 4 2

     y1  y2  w1  w2  w3
 4.  0. -5.  0.  0.  2.
 4.  0.  4.  1.  0. -1.
 3.  0.  2.  0.  1.  0.
 2.  1. -2.  0.  0.  1.

< pivot 2 3

     y1  y2  w1    w2  w3
 9.  0.  0.  1.25  0. 0.75
 1.  0.  1.  0.25  0. -.25
 1.  0.  0. -0.50  1. 0.50
 4.  1.  0.  0.50  0. 0.50
```

$\mathbf{x}^\star = [1.25, 0, 0.75]^\top$

$z_{\mathbf{x}} = 6 \times 1.25 + 3 \times 0 + 2 \times 0.75 = 9$

$\mathbf{y}^\star = [4, 1]^\top$

$z_{\mathbf{y}} = 2 \times 4 + 1 \times 1 = 9$

Notice that in the optimal tableau for $\mathscr{D}$ the cost coefficients of the slack variables w_j are the elements of the optimal vector for $\mathscr{P}$, and in the optimal tableau for $\mathscr{P}$ the cost coefficients of the slack variables s_i are the elements of the optimal vector for $\mathscr{D}$. The primal and dual

solutions can both be read off from either optimal tableau because

> the optimal vector for each problem is the transpose of the cost coefficients for the slack variables in the optimal tableau for the other problem.

Also notice from the optimal tableaus in the example that $\mathbf{c}^\top\mathbf{x}^\star = \mathbf{b}^\top\mathbf{y}^\star$. Earlier we found that if $\mathbf{x}$ and $\mathbf{y}$ are feasible vectors then $\mathbf{c}^\top\mathbf{x} \geq \mathbf{b}^\top\mathbf{y}$; now we see that the **duality gap** between these objective values is zero when $\mathbf{x} = \mathbf{x}^\star$ and $\mathbf{y} = \mathbf{y}^\star$ so

> if one problem has an optimal vector then so does the other, and the objective values are equal.

To show that these two propositions are true in general we can recapitulate the above solution of the `dp1` min problem symbolically [3, p113-115] assuming that its data make the problem feasible and bounded but are otherwise arbitrary.

$$\mathscr{P}: \begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \xrightarrow[\text{standard form}]{\text{reformulate to}} \begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n\ \mathbf{s}\in\mathbb{R}^m}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} + \mathbf{0}^\top\mathbf{s} \\ \text{subject to} & -\mathbf{A}\mathbf{x} + \mathbf{I}\mathbf{s} = -\mathbf{b} \\ & \mathbf{x}, \mathbf{s} \geq \mathbf{0} \end{array}$$

$$\Big\downarrow \text{form tableau}$$

$$\mathbf{T}^\star = \begin{array}{c|c|c|} & \mathbf{x} & \mathbf{s} \\ \hline & & \mathbf{y}^{\star\top} \\ \hline & & \\ & & \mathbf{M} \\ & & \end{array} \xleftarrow[\text{optimality}]{\text{pivot to}} \begin{array}{c|c|c|} & \mathbf{x} & \mathbf{s} \\ \hline 0 & \mathbf{c}^\top & \mathbf{0}^\top \\ \hline \mathbf{-b} & \mathbf{-A} & \mathbf{I} \end{array} = \mathbf{T}$$

Recall from §4.2.1 that to construct a pivot matrix $\mathbf{Q}$ that will make $\mathbf{QT} = \mathbf{T}^\star$ we need only do to the identity what we would like to do to $\mathbf{T}$. But solving the problem did precisely those things to the identity in the all-slack basis columns of $\mathbf{T}$, yielding the $\mathbf{s}$ columns of $\mathbf{T}^\star$. Thus we can write down $\mathbf{Q}$ by inspection and fill in the rest of $\mathbf{T}^\star$ by computing this matrix product.

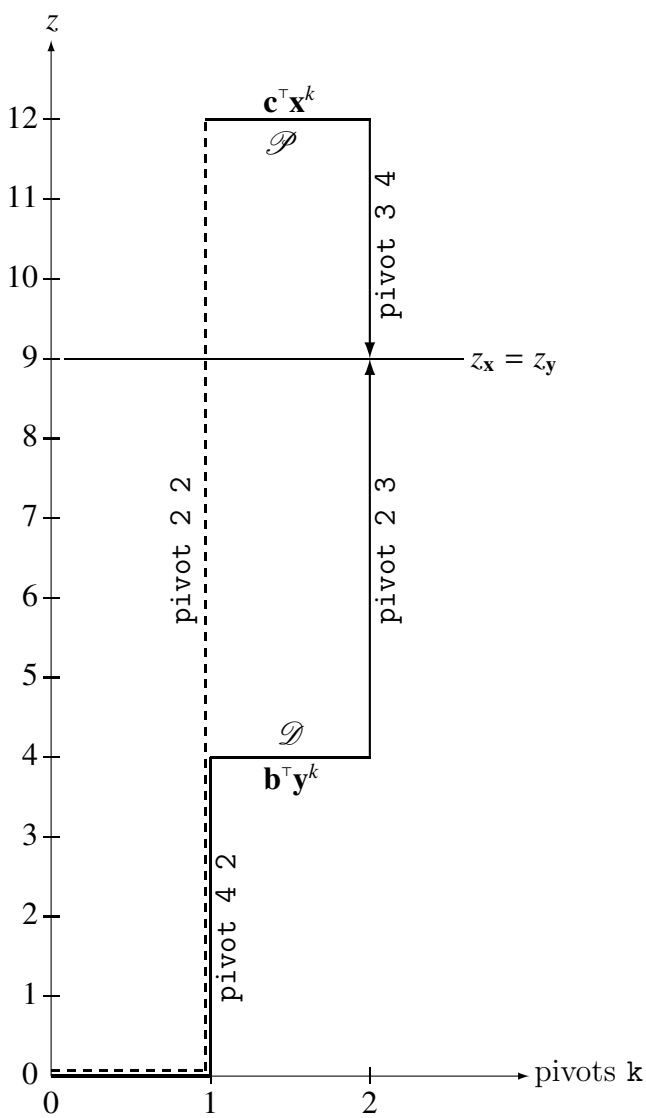
$$\mathbf{T}^\star = \mathbf{QT} = \left[\begin{array}{c|c} 1 & \mathbf{y}^{\star\top} \\ \hline 0 & \\ \vdots & \mathbf{M} \\ 0 & \end{array}\right] \begin{array}{c|c|c|} & \mathbf{x} & \mathbf{s} \\ \hline 0 & \mathbf{c}^\top & \mathbf{0}^\top \\ \hline \mathbf{-b} & \mathbf{-A} & \mathbf{I} \end{array} = \begin{array}{c|c|c|} & \mathbf{x} & \mathbf{s} \\ \hline -\mathbf{y}^{\star\top}\mathbf{b} & (\mathbf{c}^\top - \mathbf{y}^{\star\top}\mathbf{A}) & \mathbf{y}^{\star\top} \\ \hline -M_1\mathbf{b} & & \\ \vdots & \mathbf{-MA} & \mathbf{M} \\ -M_m\mathbf{b} & & \end{array}$$

Because $\mathbf{T}^\star$ is in optimal form its cost coefficients are nonnegative, so

$$\begin{array}{rcl} \mathbf{c}^\top - \mathbf{y}^{\star\top}\mathbf{A} & \geq & \mathbf{0}^\top \\ \mathbf{c} - \mathbf{A}^\top\mathbf{y}^\star & \geq & \mathbf{0} \\ \mathbf{A}^\top\mathbf{y}^\star & \leq & \mathbf{c} \end{array} \quad \text{and} \quad \begin{array}{rcl} \mathbf{y}^{\star\top} & \geq & \mathbf{0}^\top \\ \mathbf{y}^\star & \geq & \mathbf{0}. \end{array}$$

Thus $\mathbf{y}^\star$ is feasible for the max problem. The optimal value of the min problem is $\mathbf{c}^\top\mathbf{x}^\star$ so it must be that $\mathbf{c}^\top\mathbf{x}^\star = -(-\mathbf{y}^{\star\top}\mathbf{b}) = \mathbf{b}^\top\mathbf{y}^\star$ and $\mathbf{y}^\star$ is optimal for the max problem □.

A similar construction can be used to show that $\mathbf{x}^\star$ is the transpose of the cost coefficients for the slack variables in the optimal tableau for the max problem.



Plotting the $\mathbf{c}^\top\mathbf{x}^k$ and $\mathbf{b}^\top\mathbf{y}^k$ values generated by the simplex algorithm in solving the primal and dual problems of the `dp1` example yields the picture on the left. The initial tableau for each problem has $z = 0$, so both curves begin at the origin of this graph.

The primal problem starts infeasible so a subproblem pivot is required to obtain canonical form, and this is shown as a dashed line. At the end of the first pivot the primal is in canonical form with $\mathbf{c}^\top\mathbf{x} = 12$ (the upper left entry in the tableau is $-z_\mathbf{x}$ for the minimization). Then one phase 2 pivot reduces $\mathbf{c}^\top\mathbf{x}$ to its optimal value of 9.

The minimization corresponding to the dual problem has an initial tableau that is in canonical form. The first phase 2 pivot increases $\mathbf{b}^\top\mathbf{y}$ to 4 and the second increases it to its optimal value of 9 (the upper left entry in the tableau is $+z_\mathbf{y}$ for the maximization problem, because we had to change the sign of the objective to put the problem into standard form).

It is clear from this picture that $\mathbf{c}^\top\mathbf{x} \geq \mathbf{b}^\top\mathbf{y}$ when $\mathbf{x}$ and $\mathbf{y}$ are feasible for their respective problems, and that the duality gap between them is zero when both vectors are optimal.

5.1.3 One Problem Feasible

In the dual pair below, which I will call `dp2`, only one of the problems is feasible. Putting each into standard form results in the initial tableaus shown.

$$\mathscr{P}: \begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -2x_1 - x_2 & & \\ \text{subject to} & x_1 - 2x_2 & \geq & -3 \\ & -x_2 & \geq & -1 \\ & 4x_1 - x_2 & \geq & -5 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

$$\mathscr{D}: \begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^3}{\text{maximize}} & -3y_1 - y_2 - 5y_3 & & \\ \text{subject to} & y_1 \quad + 4y_3 & \leq & -2 \\ & -2y_1 - y_2 - \ y_3 & \leq & -1 \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2\ \mathbf{s}\in\mathbb{R}^3}{\text{minimize}} & -2x_1 - x_2 & & \\ \text{subject to} & -x_1 + 2x_2 + s_1 & = & 3 \\ & x_2 + s_2 & = & 1 \\ & -4x_1 + x_2 + s_3 & = & 5 \\ & \mathbf{x},\ \mathbf{s} & \geq & \mathbf{0} \end{array}$$

$$\begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^3\ \mathbf{w}\in\mathbb{R}^2}{\text{minimize}} & 3y_1 + y_2 + 5y_3 & & \\ \text{subject to} & y_1 \quad + 4y_3 + w_1 & = & -2 \\ & -2y_1 - y_2 - \ y_3 + w_2 & = & -1 \\ & \mathbf{y},\ \mathbf{w} & \geq & \mathbf{0} \end{array}$$

	x_1	x_2	s_1	s_2	s_3
0	$\boxed{-2}$	−1	0	0	0
3	$\boxed{-1}$	2	1	0	0
1	$\boxed{0}$	1	0	1	0
5	$\boxed{-4}$	1	0	0	1

	y_1	y_2	y_3	w_1	w_2
0	3	1	5	0	0
−2	1	0	4	1	0
−1	−2	−1	−1	0	1

maximization problem is infeasible
minimization problem is unbounded

The minimization on the left is feasible, but the x_1 column of its tableau shows that it is unbounded. Because of the structural relationship between duals, this column becomes (with sign changes in the a_{i1}) the boxed part of the second *row* in the tableau on the right, from which it is obvious that the maximization problem is infeasible. Algebraically, if $\bar{\mathbf{y}}$ were feasible for the max problem then $\mathbf{b}^\top\bar{\mathbf{y}}$ would be a lower bound on $\mathbf{c}^\top\bar{\mathbf{x}}$, but because the min problem is unbounded $\mathbf{c}^\top\bar{\mathbf{x}}$ has no lower bound. If it were the max problem that was unbounded then no $\bar{\mathbf{x}}$ could exist to provide an upper bound $\mathbf{c}^\top\bar{\mathbf{x}}$ on $\mathbf{b}^\top\bar{\mathbf{y}}$, and the min problem would have to be infeasible. Thus the argument works both ways, and

$$\boxed{\text{if either problem is feasible but unbounded then the other problem is infeasible.}}$$

Both problems can be feasible and bounded as we saw in §5.1.2, or both can be infeasible as we saw in §5.1.1, but if only one is infeasible then the other must be unbounded and if one is unbounded the other must be infeasible.

5.1.4 Shadow Prices

Recall from §1.3.1 that the constraints of the `brewery` problem keep Sarah from using more of any ingredient than she has on hand.

$$\mathbf{T}_0 = \begin{array}{c|cccccccc|l} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 & \\ \hline 0 & -90 & -150 & -60 & -70 & 0 & 0 & 0 & \\ \hline 160 & 7 & 10 & 8 & 12 & 1 & 0 & 0 & \text{pale malt} \\ 50 & 1 & 3 & 1 & 1 & 0 & 1 & 0 & \text{black malt} \\ 60 & 2 & 4 & 1 & 3 & 0 & 0 & 1 & \text{hops} \end{array}$$

The production program $\mathbf{x}^0 = [0, 0, 0, 0]^\top$ uses none of the resources, so in $\mathbf{T}_0$ each slack is equal to the total supply of the ingredient it measures and the revenue from selling finished products is zero.

$$\mathbf{T}^\star = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 2325 & 0 & 0 & 18\frac{3}{4} & 76\frac{1}{4} & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ \hline 5 & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 12\frac{1}{2} & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & -\frac{1}{4} & 0 & \frac{7}{8} \\ 7\frac{1}{2} & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & \frac{1}{4} & 1 & -1\frac{3}{8} \end{array}$$

In $\mathbf{T}^\star$ the slack variables are $\mathbf{s}^\star = [0, 7\frac{1}{2}, 0]^\top$, so to produce a revenue of 2325 the optimal production program $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$ uses all of the pale malt and all of the hops, with $7\frac{1}{2}$ pounds of black malt left over (s_2 is still the slack in black malt, even though pivots have moved its identity column 1 to a different row).

If one of Sarah's fellow brewers needed some black malt she could give him up to $7\frac{1}{2}$ pounds from her stock for free, since she cannot use it anyway. Giving away some pale malt or some hops, however, would change $\mathbf{x}^\star$ and decrease the revenue she realizes from producing beer. In order to have 1 pound of pale malt left over to give away, she would have to change the production program in such a way that $s_1^\star = 1$. As discussed in §2.2, each row of $\mathbf{T}^\star$ corresponds to an equation whose = sign is represented by the vertical line inside the tableau. To investigate the consequences of requiring that s_1 have a particular value, we can rewrite $\mathbf{T}^\star$ by moving its s_1 column from the right side of the equals signs to the left, like this.

$$\overline{\mathbf{T}} = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & s_2 & s_3 \\ \hline 2325 - 7\frac{1}{2}s_1 & 0 & 0 & 18\frac{3}{4} & 76\frac{1}{4} & 0 & 18\frac{3}{4} \\ \hline 5 - \frac{1}{2}s_1 & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & 0 & -1\frac{1}{4} \\ 12\frac{1}{2} + \frac{1}{4}s_1 & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & 0 & \frac{7}{8} \\ 7\frac{1}{2} - \frac{1}{4}s_1 & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & 1 & -1\frac{3}{8} \end{array}$$

If $s_1 = 0$ this tableau represents the same production program as $\mathbf{T}^\star$. Increasing s_1 changes the optimal solution to $\mathbf{x}^\star = [5 - \frac{1}{2}s_1, 12\frac{1}{2} + \frac{1}{4}s_1, 0, 0]^\top$, decreasing the revenue from production

by \$7.50 for each pound of pale malt that Sarah insists on having left over. Of course $\overline{\mathbf{T}}$ is in optimal form only if its $\mathbf{b}$ part is nonnegative, which requires that

$$\left.\begin{array}{rcccrcr} 5-\frac{1}{2}s_1 & \geq & 0 & \Rightarrow & s_1 & \leq & 10 \\ 12\frac{1}{2}+\frac{1}{4}s_1 & \geq & 0 & \Rightarrow & s_1 & \geq & -50 \\ 7\frac{1}{2}-\frac{1}{4}s_1 & \geq & 0 & \Rightarrow & s_1 & \leq & 30 \end{array}\right\} \Rightarrow s_1 \leq 10$$

If a buyer wanted some pale malt Sarah could sell him up to 10 pounds of her stock, and if she charged \$7.50 per pound for it her total revenue would remain unchanged.

$$\begin{array}{rclcll} [\text{total revenue}] & = & [\text{revenue from making beer}] & + & [\text{revenue from selling pale malt}] & \\ & = & [2325-7\frac{1}{2}s_1] & + & [7\frac{1}{2}s_1] & = \quad 2325 \end{array}$$

The \$7.50 per pound that Sarah needs to charge in order not to loose money by selling some of her pale malt is called the **shadow price** of the resource. It is the amount by which the objective is spoiled in $\overline{\mathbf{T}}$ for each pound she sells, which we got from the cost coefficient in the s_1 column of $\mathbf{T}^\star$.

Using a result from §5.1.2, the optimal vector $\mathbf{y}^\star$ of the `brewery` problem's dual is the transpose of the cost coefficients for the $\mathbf{s}$ variables in the $\mathbf{T}^\star$ tableau given above,

$$\mathbf{y}^\star = \begin{bmatrix} 7\frac{1}{2} \\ 0 \\ 18\frac{3}{4} \end{bmatrix}.$$

Thus the shadow price of pale malt is also the optimal value of the dual variable y_1 corresponding to the first constraint. We could construct optimal-form tableaus as we did $\overline{\mathbf{T}}$, by moving the s_2 column and then the s_3 column of $\mathbf{T}^\star$ to the left of the equals signs, and those tableaus would reveal that the shadow price of black malt is $0 = y_2^\star$ (recall that Sarah could give some away for nothing) and that the shadow price of hops is $18\frac{3}{4} = y_3^\star$.

Using another result from §5.1.2,

$$\begin{array}{rcccl} \text{the optimal objective value} = \mathbf{c}^\top\mathbf{x}^\star = \mathbf{b}^\top\mathbf{y}^\star & = & z^\star & = & b_1y_1^\star + \ldots + b_my_m^\star \\ \text{so the [shadow price of resource } i\text{]} & = & \dfrac{\partial z^\star}{\partial b_i} & = & y_i^\star \end{array}$$

and it is true in general that

the shadow price of a resource in one problem of a dual pair is the optimal value of the corresponding variable in the other.

Useful insights can be gained into a resource allocation problem by considering the economic interpretation of its dual. Here I have written the `brewery` problem in the form of the primal in the dual pair that we adopted in §5.1.0, then reversed the sense of the optimization and the directions of the functional inequalities to obtain the original formulation of §1.3.1.

$$
\begin{array}{lrrcr}
\mathscr{P}: & \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & -90x_1 - 150x_2 - 60x_3 - 70x_4 & = & z_{\mathbf{x}} \\
& \text{subject to} & -7x_1 - 10x_2 - 8x_3 - 12x_4 & \geq & -160 \\
& & -1x_1 - 3x_2 - 1x_3 - 1x_4 & \geq & -50 \\
& & -2x_1 - 4x_2 - 1x_3 - 3x_4 & \geq & -60 \\
& & \mathbf{x} & \geq & \mathbf{0}
\end{array}
$$

$$\downarrow$$

$$
\begin{array}{lrcr}
\underset{\mathbf{x}\in\mathbb{R}^4}{\text{maximize}} & 90x_1 + 150x_2 + 60x_3 + 70x_4 & & \\
\text{subject to} & 7x_1 + 10x_2 + 8x_3 + 12x_4 & \leq & 160 \\
& 1x_1 + 3x_2 + 1x_3 + 1x_4 & \leq & 50 \\
& 2x_1 + 4x_2 + 1x_3 + 3x_4 & \leq & 60 \\
& \mathbf{x} & \geq & \mathbf{0}
\end{array}
$$

In solving this problem we try to maximize the revenue from selling products by setting their production levels x_j, while not using more of each ingredient than the amount on hand.

Here is the dual of problem $\mathscr{P}$, also rewritten to reverse the sense of the optimization and the directions of the functional inequalities.

$$
\begin{array}{lrrcr}
\mathscr{D}: & \underset{\mathbf{y}\in\mathbb{R}^3}{\text{maximize}} & -160y_1 - 50y_2 - 60y_3 & = & z_{\mathbf{y}} \\
& \text{subject to} & -7y_1 - 1y_2 - 2y_3 & \leq & -90 \\
& & -10y_1 - 3y_2 - 4y_3 & \leq & -150 \\
& & -8y_1 - 1y_2 - 1y_3 & \leq & -60 \\
& & -12y_1 - 1y_2 - 3y_3 & \leq & -70 \\
& & \mathbf{y} & \geq & \mathbf{0}
\end{array}
$$

$$\downarrow$$

$$
\begin{array}{lrcr}
\underset{\mathbf{y}\in\mathbb{R}^3}{\text{minimize}} & 160y_1 + 50y_2 + 60y_3 & & \\
\text{subject to} & 7y_1 + 1y_2 + 2y_3 & \geq & 90 \\
& 10y_1 + 3y_2 + 4y_3 & \geq & 150 \\
& 8y_1 + 1y_2 + 1y_3 & \geq & 60 \\
& 12y_1 + 1y_2 + 3y_3 & \geq & 70 \\
& \mathbf{y} & \geq & \mathbf{0}
\end{array}
$$

In solving this problem we try to minimize the total value to us of the ingredients that we must use by setting their prices y_i, while ensuring that the value we place on the ingredients that go into one keg of each variety of beer is at least equal to the revenue we get from selling that keg of product.

We have seen that $y_i^\star$, the shadow price for resource i, tells how much $z_{\mathbf{x}}$ goes up per unit reduction in the supply of that resource. In a symmetric (or, more precisely, dual) way, $x_j^\star$ tells how much $z_{\mathbf{y}}$ goes down per unit reduction in the selling price of product j.

5.1.5 Complementary Slackness

In §5.1.2 we found that if $\bar{\mathbf{x}}$ is feasible for the min problem in a dual pair and $\bar{\mathbf{y}}$ is feasible for the max problem, then $\mathbf{c}^\top\bar{\mathbf{x}} \geq \bar{\mathbf{y}}^\top\mathbf{A}\bar{\mathbf{x}} \geq \mathbf{b}^\top\bar{\mathbf{y}}$. By solving the min problem symbolically we showed that at optimality these three quantities are equal. Thus we have

$$\begin{aligned} \mathbf{c}^\top\mathbf{x}^\star &= \mathbf{y}^{\star\top}\mathbf{A}\mathbf{x}^\star & \qquad \mathbf{y}^{\star\top}\mathbf{A}\mathbf{x}^\star &= \mathbf{b}^\top\mathbf{y}^\star \\ \mathbf{c}^\top\mathbf{x}^\star - \mathbf{y}^{\star\top}\mathbf{A}\mathbf{x}^\star &= 0 & \mathbf{y}^{\star\top}\mathbf{A}\mathbf{x}^\star - \mathbf{b}^\top\mathbf{y}^\star &= 0 \end{aligned}$$

$$\boxed{\begin{aligned} \mathbf{x}^{\star\top}(\mathbf{c} - \mathbf{A}^\top\mathbf{y}^\star) &= 0 & \qquad \mathbf{y}^{\star\top}(\mathbf{A}\mathbf{x}^\star - \mathbf{b}) &= 0 \end{aligned}}$$

The boxed equations are called the **complementary slackness conditions**. They hold only at optimality, so

> if $\bar{\mathbf{x}}$ is feasible for the min problem and $\bar{\mathbf{y}}$ is feasible for the max problem and together they satisfy the complementary slackness conditions, then $\bar{\mathbf{x}} = \mathbf{x}^\star$ and $\bar{\mathbf{y}} = \mathbf{y}^\star$.

For the `brewery` problem these are the complementary slackness conditions.

$$\begin{aligned} x_1(-90 + 7y_1 + 1y_2 + 2y_3) &\quad + \\ x_2(-150 + 10y_1 + 3y_2 + 4y_3) &\quad + \\ x_3(-60 + 8y_1 + 1y_2 + 1y_3) &\quad + \\ x_4(-70 + 12y_1 + 1y_2 + 3y_3) &\quad = \quad 0 \end{aligned} \qquad \begin{aligned} y_1(-7x_1 - 10x_2 - 8x_3 - 12x_4 + 160) &\quad + \\ y_2(-1x_1 - \ \ 3x_2 - 1x_3 - \ \ 1x_4 + \ \ 50) &\quad + \\ y_3(-2x_1 - \ \ 4x_2 - 1x_3 - \ \ 3x_4 + \ \ 60) &\quad = \quad 0 \end{aligned}$$

Because $\mathbf{x}^\star$ is feasible for $\mathscr{P}$ we know that $\mathbf{x}^\star \geq \mathbf{0}$ and $\mathbf{A}\mathbf{x}^\star \geq \mathbf{b}$ or $(\mathbf{A}\mathbf{x}^\star - \mathbf{b}) \geq \mathbf{0}$. Because $\mathbf{y}$ is feasible for $\mathscr{D}$ we know that $\mathbf{y}^\star \geq \mathbf{0}$ and $\mathbf{A}^\top\mathbf{y}^\star \leq \mathbf{c}$ or $(\mathbf{c} - \mathbf{A}^\top\mathbf{y}^\star) \geq \mathbf{0}$. Thus all of the terms in these equations are nonnegative and the only way each sum can equal zero is if each term is zero. It is easy to verify that this is the case for $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$ and $\mathbf{y}^\star = [7\frac{1}{2}, 0, 18\frac{3}{4}]^\top$. For example,

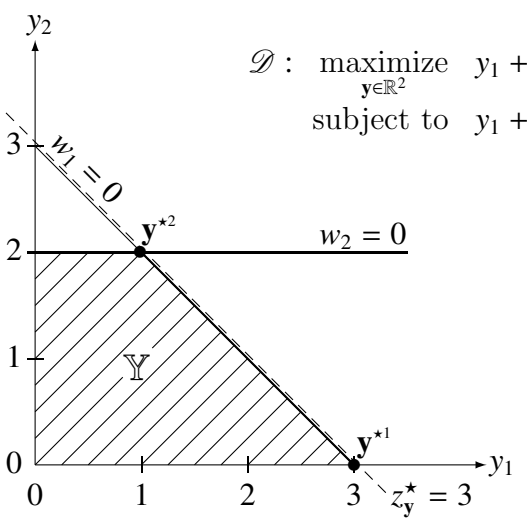
$$\begin{aligned} x_1^\star(7y_1^\star + 1y_2^\star + 2y_3^\star - 90) &= 5(7 \times 7\tfrac{1}{2} + 1 \times 0 + 2 \times 18\tfrac{3}{4} - 90) \\ &= 5(0) = 0 \\ y_2^\star(-1x_1^\star - 3x_2^\star - 1x_3^\star - 1x_4^\star + 50) &= 0(-1 \times 5 - 3 \times 12\tfrac{1}{2} - 1 \times 0 - 1 \times 0 + 50) \\ &= 0(7.5) = 0. \end{aligned}$$

In §5.1.4 we observed that if a resource is not used up its constraint is satisfied as a strict inequality, its slack variable is positive, and its shadow price (the dual variable corresponding to the constraint) is zero. Only when a resource is used up, so that its constraint is satisfied with equality and its slack variable is zero, can its shadow price be positive. The complementary slackness conditions show it is true in general that

> at optimality, if a constraint in one problem is slack the corresponding variable in the other is zero, and if a variable in one problem is positive the corresponding constraint in the other is satisfied with equality.

5.1.6 Multiple Optima and Degeneracy

At optimality a positive variable in one problem of a dual pair implies that the corresponding constraint in the other problem is tight. It might seem that the converse would also be true; after all, if a constraint is satisfied with equality then tightening it will move the optimal solution to a different point. But if the problem has *multiple* optimal solutions, changing the optimal point need not change the objective. This `dp3` problem, whose graphical and simplex solutions are shown, has the form of the dual in the pair we adopted in §5.1.0.



$$\mathscr{D}: \quad \underset{\mathbf{y}\in\mathbb{R}^2}{\text{maximize}} \quad y_1 + y_2 = z_{\mathbf{y}}$$
$$\text{subject to} \quad y_1 + y_2 \le 3$$
$$y_2 \le 2$$
$$\mathbf{y} \ge \mathbf{0}$$

$$\mathbf{D}_0 = \begin{array}{c|cccc} & y_1 & y_2 & w_1 & w_2 \\ \hline 0 & -1 & -1 & 0 & 0 \\ \hline 3 & 1 & 1 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \end{array}$$

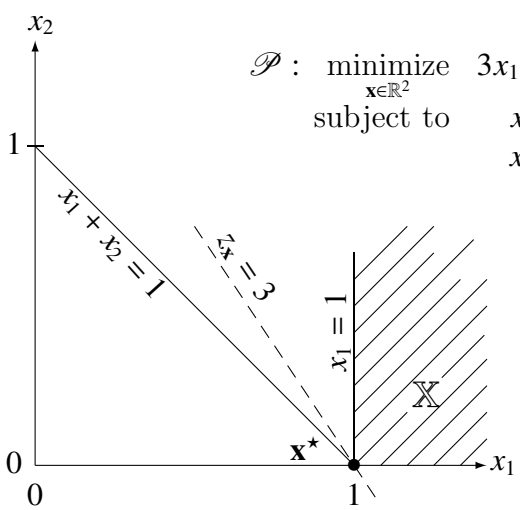
$$\mathbf{D}_1^{\star} = \begin{array}{c|cccc} & y_1 & y_2 & w_1 & w_2 \\ \hline 3 & 0 & 0 & \boxed{1} & \boxed{0} \\ \hline 3 & 1 & 1 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \end{array} \longleftarrow \mathbf{x}^{\star\top}$$

$$\mathbf{D}_2^{\star} = \begin{array}{c|cccc} & y_1 & y_2 & w_1 & w_2 \\ \hline 3 & 0 & 0 & \boxed{1} & \boxed{0} \\ \hline 1 & 1 & 0 & 1 & -1 \\ 2 & 0 & 1 & 0 & 1 \end{array} \longleftarrow \mathbf{x}^{\star\top}$$

The optimal set consists of $\mathbf{y}^{\star 1}$, $\mathbf{y}^{\star 2}$, and the edge between them. At $\mathbf{y}^{\star 2} = [1, 2]^{\top}$ the second constraint is satisfied with equality, but in the optimal tableaus $\mathbf{D}_1^{\star}$ and $\mathbf{D}_2^{\star}$ we see that the cost coefficient of w_2, which is the shadow price $x_2^{\star}$ of the second constraint, is zero. Although increasing w_2 from zero would move the contour down and push the optimal point diagonally along the optimal edge, that would not change $z_{\mathbf{y}}$.

The first constraint has the positive shadow price $x_1^{\star} = 1$, because increasing w_1 from zero would move that contour toward the origin and spoil the objective by an equal amount.

The dual of the max problem we solved above is given below along with its graphical and simplex solutions.



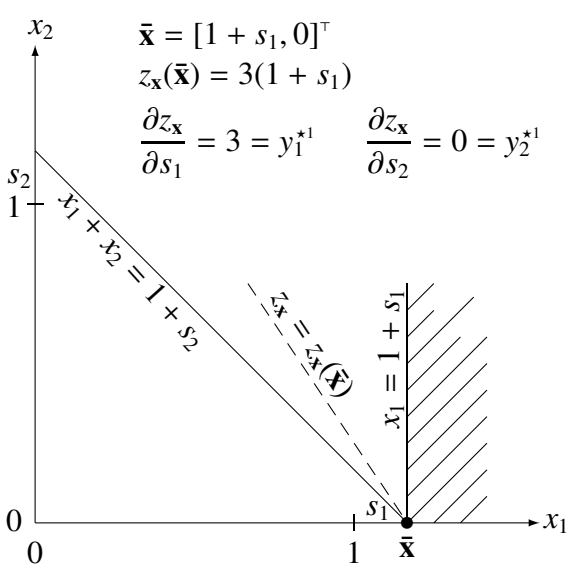
$$\mathscr{P}: \quad \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad 3x_1 + 2x_2 = z_{\mathbf{x}}$$
$$\text{subject to} \quad x_1 \geq 1, \quad x_1 + x_2 \geq 1, \quad \mathbf{x} \geq \mathbf{0}$$

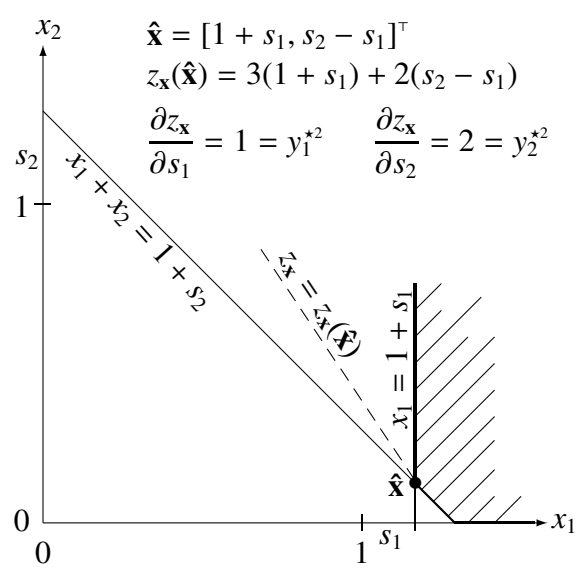
$$\mathbf{P}_0 = \begin{array}{c|ccccc} & x_1 & x_2 & s_1 & s_2 \\ \hline 0 & 3 & 2 & 0 & 0 \\ \hline -1 & -1 & 0 & 1 & 0 \\ -1 & -1 & -1 & 0 & 1 \end{array}$$

$$\mathbf{P}_1^\star = \begin{array}{c|ccccc} & x_1 & x_2 & s_1 & s_2 \\ \hline -3 & 0 & 2 & \boxed{3} & \boxed{0} \\ \hline 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & -1 & 1 \end{array} \longleftarrow \mathbf{y}^{\star 1\top}$$

$$\mathbf{P}_2^\star = \begin{array}{c|ccccc} & x_1 & x_2 & s_1 & s_2 \\ \hline -3 & 0 & 0 & \boxed{1} & \boxed{2} \\ \hline 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 1 & -1 \end{array} \longleftarrow \mathbf{y}^{\star 2\top}$$

The optimal vertex $\mathbf{x}^\star = [1, 0]^\top$ is overdetermined by the intersection of 3 constraint hyperplanes in $\mathbb{R}^2$, so it is represented by two different basic sequences and the pivot from $\mathbf{P}_1^\star$ to $\mathbf{P}_2^\star$ is degenerate. In both optimal tableaus, $s_1 = 0$ and $s_2 = 0$ because both functional constraints are active. We can find their shadow prices graphically by considering the two cases pictured below.





The graph on the left describes what happens if we perturb the solution represented by $\mathbf{P}_1^\star$ by making $s_1 > 0$. The optimal point is displaced to $\bar{\mathbf{x}}$ and the shadow prices we derive from the resulting objective are the cost coefficients of the slack variables in that tableau. We

could also deduce these numbers using the approach we took in §5.1.4, by moving the s_1 column of the tableau to the left of the equals signs like this.

	x_1	x_2	s_2
$-3-3s_1$	0	2	0
$1+1s_1$	1	0	0
$0+1s_1$	0	-1	1

In this basic feasible solution $x_1 = 1 + s_1$ and $-z_{\mathbf{x}} = -3 - 3s_1$, as we found in the graphical analysis, so the shadow price for the first constraint is again 3. Increasing s_1 in this tableau also increases s_2 because it is basic. This is the *only* way to increase a basic variable without changing the basis: change a nonbasic variable and thus the b_i that is the value of the basic variable. It would not make sense to study the effect of changing s_2 by moving its basic column to the other side of the line, because that would destroy canonical (and hence optimal) form. Because $s_2 = s_1$ in this basis, the motion of the first constraint contour that results from increasing s_1 is enough by itself to change $\bar{\mathbf{x}}$, so the shadow price of the second constraint is zero even though its contour gets dragged along too.

The graph on the right above describes what happens if we perturb the solution represented by $\mathbf{P}_2^\star$. Now $s_2 > s_1$ and the optimal point is displaced to $\hat{\mathbf{x}}$, which I found by solving for the intersection of the hyperplanes. The shadow prices we derive from the resulting objective are the cost coefficients of the slack variables in the tableau. Because the s_1 and s_2 columns are nonbasic, we could also deduce the shadow prices by moving those columns one at a time to the other side of the line.

Even when a linear program is degenerate the cost coefficients of the slack variables in each of its optimal tableaus can be interpreted as the shadow prices of the constraints in that tableau. These vectors are also optimal points of the dual problem, and in this example they are different so the dual has distinct multiple optima.

To further explore the connection between multiple optima in one problem of a dual pair and degeneracy in the other, recall that our §5.1.2 symbolic solution of the min problem yielded the optimal tableau on the left.

$$\begin{array}{c|c:c}
 & \mathbf{x} & \mathbf{s} \\
\hline
-\mathbf{y}^{\star\top}\mathbf{b} & (\mathbf{c}^\top - \mathbf{y}^{\star\top}\mathbf{A}) & \mathbf{y}^{\star\top} \\
\hline
-M_1\mathbf{b} & & \\
\vdots & -\mathbf{MA} & \mathbf{M} \\
-M_m\mathbf{b} & & \\
\end{array}
=
\begin{array}{c|cc:cc}
 & x_1 & x_2 & s_1 & s_2 \\
\hline
-3 & 0 & 2 & 3 & 0 \\
\hline
1 & 1 & 0 & -1 & 0 \\
0 & 0 & -1 & -1 & 1
\end{array}
= \mathbf{P}_1^\star$$

Using this result we argued that $\mathbf{y}^\star$ is optimal for the max problem and that $\mathbf{c}^\top\mathbf{x}^\star = \mathbf{b}^\top\mathbf{y}^\star$. For our `dp3` example, the min problem has

$$\mathbf{M} = \begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad \mathbf{y}^\star = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

so $\mathbf{P}_1^\star$ is the tableau on the right above.

To solve the max problem of the standard dual pair we put it into standard form like this, adding the vector of slack variables $\mathbf{w}$.

$$\mathscr{D}: \begin{array}{ll} \underset{\mathbf{y}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{b}^\top\mathbf{y} \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} \le \mathbf{c} \\ & \mathbf{y} \ge \mathbf{0} \end{array} \quad \xrightarrow[\text{add slacks}]{\text{max to min}} \quad \begin{array}{ll} \underset{\mathbf{y}\in\mathbb{R}^m\ \mathbf{w}\in\mathbb{R}^n}{\text{minimize}} & -\mathbf{b}^\top\mathbf{y} \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} + \mathbf{w} = \mathbf{c} \\ & \mathbf{y},\ \mathbf{w} \ge \mathbf{0} \end{array}$$

Feasible points for the problem on the right satisfy $\mathbf{w} = \mathbf{c} - \mathbf{A}^\top\mathbf{y}$, so at the optimal point $\mathbf{w}^{\star\top} = (\mathbf{c}^\top - \mathbf{y}^{\star\top}\mathbf{A})$, and these are just the cost coefficients of the $\mathbf{x}$ columns in $\mathbf{P}_1^\star$. Similarly, the cost coefficients of the $\mathbf{y}$ variables in the optimal tableau for the problem on the right are the optimal values of the slacks $\mathbf{s}$ in our reformulation of the min problem of the standard dual pair. Here again are the final tableaus we found above for the `dp3` problems.

$$\mathbf{P}_1^\star = \begin{array}{c|cccc} & x_1 & x_2 & s_1 & s_2 \\ \hline -3 & \boxed{0 \quad 2} & & \boxed{3 \quad 0} & \\ 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & -1 & 1 \end{array} \qquad \mathbf{D}_1^\star = \begin{array}{c|cccc} & y_1 & y_2 & w_1 & w_2 \\ \hline 3 & \boxed{0 \quad 0} & & \boxed{1 \quad 0} & \\ 3 & 1 & 1 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \end{array}$$

$\mathbf{s}^\star = [0,0]^\top$ $\qquad$ $\mathbf{y}^\star = [3,0]^\top$

$\mathbf{x}^\star = [1,0]^\top$ $\qquad$ $\mathbf{w}^\star = [0,2]^\top$

It is true in general that

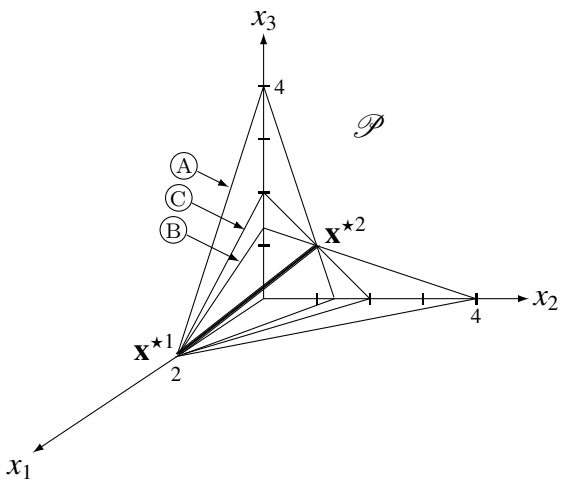
> the optimal slack vector for each problem is the transpose of the cost coefficients for the non-slack variables in the optimal tableau for the other problem.

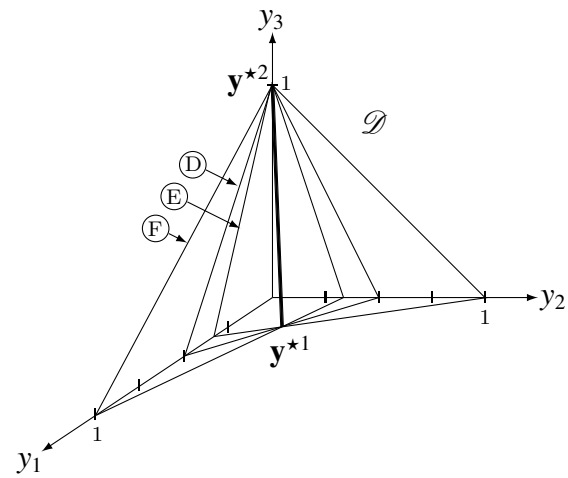
In §5.1.2 we found that the vector of optimal non-slack variables for each problem is the transpose of the cost coefficients for the slack variables in the optimal tableau for the other, so *all* of the components in the solution $[\mathbf{x}^{\star\top}, \mathbf{s}^{\star\top}]$ to the min problem appear as cost coefficients in the optimal tableau for the max problem and *all* of the components in $[\mathbf{y}^{\star\top}, \mathbf{w}^{\star\top}]$ appear as cost coefficients in the optimal tableau for the min problem.

If one problem has an alternate optimal solution some nonbasic column in its optimal tableau must have a zero cost coefficient, and that means the corresponding constant-column entry in the optimal tableau of the other problem is zero. Thus

> if one problem in a dual pair has multiple optimal vectors then the other problem is degenerate.

If one problem is degenerate and the slack variable cost coefficients in the different tableaus representing its optimal point are different, as in the `dp3` primal, then the other problem has multiple optimal vertices. In the dual pair `dp4` on the next page [114] both problems are degenerate and each has two optimal vertices.





$$\mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} \quad x_1 + x_2 + x_3$$
$$\text{subject to} \quad \begin{aligned} 2x_1 + 3x_2 + x_3 &\geq 4 \quad \text{(A)} \\ 2x_1 + x_2 + 3x_3 &\geq 4 \quad \text{(B)} \\ x_1 + x_2 + x_3 &\geq 2 \quad \text{(C)} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

	x_1	x_2	x_3	s_1	s_2	s_3
0	1	1	1	0	0	0
−4	−2	−3	−1	1	0	0
−4	−2	−1	−3	0	1	0
−2	−1	−1	−1	0	0	1

solve

	x_1	x_2	x_3	s_1	s_2	s_3
−2	0	0	0	0	0	1
2	1	0	2	0	−1	1
0	0	1	−1	0	1	−2
0	0	0	0	1	1	−4

$\mathbf{x}^{\star 1} = [2, 0, 0]^\top$

	x_1	x_2	x_3	s_1	s_2	s_3
−2	0	0	0	0	0	1
1	$\frac{1}{2}$	0	1	0	$-\frac{1}{2}$	$\frac{1}{2}$
1	$\frac{1}{2}$	1	0	0	$\frac{1}{2}$	$-1\frac{1}{2}$
0	0	0	0	1	1	−4

$\mathbf{x}^{\star 2} = [0, 1, 1]^\top$

$$\mathscr{D}: \underset{\mathbf{y}\in\mathbb{R}^3}{\text{maximize}} \quad 4y_1 + 4y_2 + 2y_3$$
$$\text{subject to} \quad \begin{aligned} 2y_1 + 2y_2 + y_3 &\leq 1 \quad \text{(D)} \\ 3y_1 + y_2 + y_3 &\leq 1 \quad \text{(E)} \\ y_1 + 3y_2 + y_3 &\leq 1 \quad \text{(F)} \\ \mathbf{y} &\geq \mathbf{0} \end{aligned}$$

	y_1	y_2	y_3	w_1	w_2	w_3
0	−4	−4	−2	0	0	0
1	2	2	1	1	0	0
1	3	1	1	0	1	0
1	1	3	1	0	0	1

solve

	y_1	y_2	y_3	w_1	w_2	w_3
2	0	0	0	0	1	1
$\frac{1}{4}$	1	0	$\frac{1}{4}$	0	$\frac{3}{8}$	$-\frac{1}{8}$
$\frac{1}{4}$	0	1	$\frac{1}{4}$	0	$-\frac{1}{8}$	$\frac{3}{8}$
0	0	0	0	1	$-\frac{1}{2}$	$-\frac{1}{2}$

$\mathbf{y}^{\star 1} = [\frac{1}{4}, \frac{1}{4}, 0]^\top$

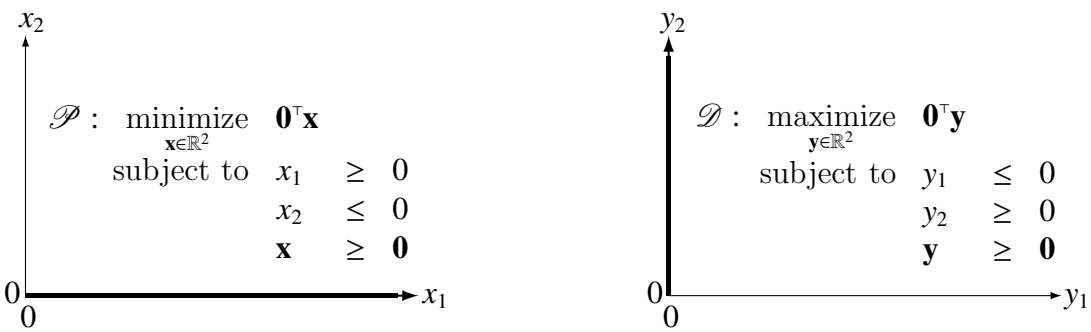
	y_1	y_2	y_3	w_1	w_2	w_3
2	0	0	0	0	1	1
1	4	0	1	0	$1\frac{1}{2}$	$-\frac{1}{2}$
0	−1	1	0	0	$-\frac{1}{2}$	$\frac{1}{2}$
0	0	0	0	1	$-\frac{1}{2}$	$-\frac{1}{2}$

$\mathbf{y}^{\star 2} = [0, 0, 1]^\top$

The optimal set for the primal consists of the vertices $\mathbf{x}^{\star 1}$ and $\mathbf{x}^{\star 2}$ and the line connecting them; the optimal set for the dual consists of the vertices $\mathbf{y}^{\star 1}$ and $\mathbf{y}^{\star 2}$ and the line connecting them. These optimal tableaus for the primal have slack variable cost coefficients $[0, 0, 1]^\top = \mathbf{y}^{\star 2}$ and these optimal tableaus for the dual have slack variable cost coefficients $[0, 1, 1]^\top = \mathbf{x}^{\star 2}$; in each optimal tableau a degenerate pivot can be performed to represent the point by a different basic sequence, and the slack variable cost coefficients in those tableaus correspond to $\mathbf{x}^{\star 1}$ and $\mathbf{y}^{\star 1}$ (see Exercise 5.5.27). This example shows that

$$\boxed{\text{it is possible for both problems to be degenerate,}}$$

and in that case both can have multiple optimal vertices. If both problems in a dual pair are degenerate it is also possible that neither has multiple optimal vertices, as shown by the example below [71, Myth 12].



$$\mathscr{P}: \quad \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad \mathbf{0}^\top\mathbf{x}$$
$$\text{subject to} \quad x_1 \geq 0, \quad x_2 \leq 0, \quad \mathbf{x} \geq \mathbf{0}$$

$$\mathscr{D}: \quad \underset{\mathbf{y}\in\mathbb{R}^2}{\text{maximize}} \quad \mathbf{0}^\top\mathbf{y}$$
$$\text{subject to} \quad y_1 \leq 0, \quad y_2 \geq 0, \quad \mathbf{y} \geq \mathbf{0}$$

	x_1	x_2	s_1	s_2
0	0	0	0	0
0	−1	0	1	0
0	0	1	0	1

	x_1	x_2	s_1	s_2
0	0	0	0	0
0	1	0	−1	0
0	0	1	0	1

	y_1	y_2	w_1	w_2
0	0	0	0	0
0	1	0	1	0
0	0	−1	0	1

	y_1	y_2	w_1	w_2
0	0	0	0	0
0	1	0	1	0
0	0	1	0	−1

The primal tableaus on the left are separated by a degenerate pivot, so both represent the same point $\mathbf{x}^\star = [0, 0]^\top$. The cost coefficients of the $\mathbf{s}$ variables are the same in both tableaus, so the dual does not have multiple optimal vertices. The tableaus on the right are also separated by a degenerate pivot, so both of them represent the same point $\mathbf{y}^\star = [0, 0]^\top$. The cost coefficients of the $\mathbf{w}$ variables are the same in both tableaus, so the primal does not have multiple optimal vertices either.

In 21.1.3 we will see that the convergence and numerics of the primal-dual interior point method for linear programming are affected by the presence of multiple optimal solutions in either problem and the resulting degeneracy of the other.

5.2 Finding Duals

The dual of a max problem is a min problem and the dual of a min problem is a max problem, but finding the dual of a given linear program is more than just a complicated way of changing the direction of the optimization. The dual of a linear program must have the structural relationship to its primal discussed in §5.0, so that the two problems will have the algebraic relationships to each other discussed in §5.1.

The easiest way to find a dual is to rewrite the given linear program in the form of one of the problems in the standard dual pair. Then the dual of the given linear program is the *other* problem in the standard dual pair, which can be rewritten if necessary to put it in a convenient form. In rewriting the given linear program or the dual, it is often helpful to

- replace an equality constraint by opposing inequality constraints, or replace opposing inequality constraints by an equality constraint;
- combine vectors or matrices into one, or partition the elements of a single vector or matrix into different ones;
- replace a free variable by the difference between nonnegative variables as in §2.9.3, or replace the difference between nonnegative variables by a free variable.

The other reformulation techniques discussed in §2.9 are also sometimes useful in this context.

We have been using $\mathbf{x}$ as the variable, $\mathbf{c}$ as the cost vector, $\mathbf{b}$ as the constant vector, and $\mathbf{A}$ as the constraint coefficient matrix in the min or primal problem of the standard dual pair. In finding the duals of arbitrary linear programs, which might use those variable names in other ways, it is better to keep in mind the pictorial representation of the standard dual pair that was suggested in the introduction to the Chapter.

5.2.1 The Standard Form Linear Program

Recall from §2.1 that a linear program is in standard form when it is written like this.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

Both problems in the standard dual pair have inequality constraints, so to make this problem resemble either of them we must replace the equality by opposing inequalities.

$$\mathbf{A}\mathbf{x} = \mathbf{b} \Leftrightarrow \mathbf{A}\mathbf{x} \leq \mathbf{b} \text{ and } \mathbf{A}\mathbf{x} \geq \mathbf{b}$$

Then the original problem can be rewritten as at the top of the next page.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & \geq & \mathbf{b} \\ & -\mathbf{A}\mathbf{x} & \geq & -\mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \quad \longrightarrow \quad \begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \begin{bmatrix} \mathbf{A} \\ -\mathbf{A} \end{bmatrix}\mathbf{x} & \geq & \begin{bmatrix} \mathbf{b} \\ -\mathbf{b} \end{bmatrix} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

The problem on the right is in the form of the min problem in the standard dual pair.

If we introduce dual variables $\mathbf{u}$ and $\mathbf{v}$ each the length of $\mathbf{b}$, we can write the max problem of the standard dual pair like this [3, p120-121].

$$\begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^m}{\text{maximize}} & [\mathbf{b}^\top, -\mathbf{b}^\top]\begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} & & \\ \text{subject to} & [\mathbf{A}^\top, -\mathbf{A}^\top]\begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} & \leq & \mathbf{c} \\ & \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} & \geq & \mathbf{0} \end{array} \quad \longrightarrow \quad \begin{array}{lrcl} \underset{\mathbf{u}\in\mathbb{R}^m\ \mathbf{v}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{b}^\top(\mathbf{u}-\mathbf{v}) & & \\ \text{subject to} & \mathbf{A}^\top(\mathbf{u}-\mathbf{v}) & \leq & \mathbf{c} \\ & \mathbf{u},\ \mathbf{v} & \geq & \mathbf{0} \end{array}$$

Letting the difference $\mathbf{u}-\mathbf{v}$ between nonnegative variables be a free variable $\mathbf{y}$ yields the simpler dual on the right below.

$$\mathscr{P}:\ \begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad \mathscr{D}:\ \begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{b}^\top\mathbf{y} & & \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} & \leq & \mathbf{c} \\ & \mathbf{y} & & \text{free} \end{array}$$

Having established that these linear programs are a dual pair, they can from now on be used like the dual pair that we earlier identified as standard. In particular, we can use them to easily write down the dual of any linear program that is in standard form.

5.2.2 The Transportation Problem

In §6 we will take up linear programming models of network flows. The simplest of them is this **transportation problem**, in which s_i is a supply, d_j is a demand, and c_{ij} and x_{ij} are the unit cost of shipping and the amount shipped from source i to destination j.

$$\begin{array}{lrcll} \underset{\mathbf{x}\in\mathbb{R}^{pq}}{\text{minimize}} & \displaystyle\sum_{j\in\mathbb{D}}\sum_{i\in\mathbb{S}} c_{ij}x_{ij} & = & \alpha(\mathbf{x}) & p=|\mathbb{S}|,\ q=|\mathbb{D}| \\ \text{subject to} & \displaystyle\sum_{j\in\mathbb{D}} x_{ij} & = & s_i & i\in\mathbb{S} \\ & \displaystyle\sum_{i\in\mathbb{S}} x_{ij} & = & d_j & j\in\mathbb{D} \\ & \mathbf{x} & \geq & \mathbf{0} & \end{array}$$

In developing an algorithm to solve this problem we will make use of its dual. To find that dual we begin by putting this primal into the form of one of the problems in some dual pair. Because the problem is already in standard form it is convenient to rewrite it as the min problem in the dual pair we derived in §5.2.1.

It will be easy to write the general transportation problem in that form if we first consider a specific instance. This one [3, p123] has sources $\mathbb{S} = \{1, 2\}$ and destinations $\mathbb{D} = \{3, 4, 5\}$.

$$\begin{array}{lrcrcrcrcrcrcl}
\underset{\mathbf{x}\in\mathbb{R}^6}{\text{minimize}} & c_{13}x_{13} & + & c_{14}x_{14} & + & c_{15}x_{15} & + & c_{23}x_{23} & + & c_{24}x_{24} & + & c_{25}x_{25} & & \\
\text{subject to} & x_{13} & + & x_{14} & + & x_{15} & & & & & & & = & s_1 \\
 & & & & & & & x_{23} & + & x_{24} & + & x_{25} & = & s_2 \\
 & x_{13} & & & & & + & x_{23} & & & & & = & d_3 \\
 & & & x_{14} & & & & & + & x_{24} & & & = & d_4 \\
 & & & & & x_{15} & & & & & + & x_{25} & = & d_5 \\
 & & & & & & & & & & & \mathbf{x} & \geq & \mathbf{0}
\end{array}$$

If we put the x_{ij} and c_{ij} into vectors in the order they appear above, we can write the objective as $\mathbf{c}^\top\mathbf{x}$. If we put the right-hand side values into $\mathbf{b} = [s_1, s_2, d_3, d_4, d_5]^\top$ and repeat in $\mathbf{A}$ the pattern of 1 and 0 coefficients evident in the $p = 2$ source constraints and $q = 3$ demand constraints,

$$\mathbf{A} = \underbrace{\left[\begin{array}{ccc:ccc}
1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
\hdashline
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1
\end{array}\right]}_{pq \text{ columns}}
\begin{array}{l}
\left.\vphantom{\begin{array}{c}1\\1\end{array}}\right\} p \text{ rows} \\
\left.\vphantom{\begin{array}{c}1\\1\\1\end{array}}\right\} q \text{ rows}
\end{array}$$

q columns

we can write the constraints as $\mathbf{Ax} = \mathbf{b}$.

To form the dual we must introduce a variable corresponding to each constraint, and because there are two sets of those it is natural to let $\mathbf{u} = [u_1 \ldots u_p]$ correspond to the supply constraints and $\mathbf{v} = [v_{p+1} \ldots v_{p+q}]$ correspond to the demand constraints. Then $\mathbf{y}^\top = [\mathbf{u}^\top, \mathbf{v}^\top]$ and we can write the dual as at the top left on the next page. In the dual of our example, on the top right, notice that each inequality is of the form $u_i + v_j \leq c_{ij}$. From that it is apparent this must be the dual of the general transportation problem.

$$\begin{array}{lll}
\underset{\mathbf{u}\in\mathbb{R}^p\ \mathbf{v}\in\mathbb{R}^q}{\text{maximize}} & \displaystyle\sum_{i\in\mathbb{S}} s_i u_i + \sum_{j\in\mathbb{D}} d_j v_j & \\
\text{subject to} & u_i + v_j \leq c_{ij} & i \in \mathbb{S},\ j \in \mathbb{D} \\
 & \mathbf{u}, \mathbf{v} \quad \text{free} &
\end{array}$$

$$\begin{array}{lll} \underset{\mathbf{y}\in\mathbb{R}^{p+q}}{\text{maximize}} & \mathbf{b}^\top\mathbf{y} & \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} & \le \mathbf{c} \\ & \mathbf{y} & \text{free} \end{array}$$

$$\begin{array}{ll} \underset{\mathbf{u}\in\mathbb{R}^2\ \mathbf{v}\in\mathbb{R}^3}{\text{maximize}} & [s_1, s_2, d_3, d_4, d_5]\begin{bmatrix} u_1 \\ u_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \\ \text{subject to} & \left[\begin{array}{cc:ccc} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \hdashline 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right]\begin{bmatrix} u_1 \\ u_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \le \begin{bmatrix} c_{13} \\ c_{14} \\ c_{15} \\ c_{23} \\ c_{24} \\ c_{25} \end{bmatrix} \\ & \mathbf{u},\ \mathbf{v} \qquad\qquad \text{free} \end{array}$$

5.2.3 Finding Duals Numerically

Putting the problems of the standard dual pair from §5.1 into standard form yields the initial tableaus `Tp` and `Td` below.

$$\mathscr{P}:\ \begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & \\ \text{subject to} & \mathbf{A}\mathbf{x} & \ge \mathbf{b} \\ & \mathbf{x} & \ge \mathbf{0} \end{array} \qquad \mathscr{D}:\ \begin{array}{lll} \underset{\mathbf{y}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{b}^\top\mathbf{y} & \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} & \le \mathbf{c} \\ & \mathbf{y} & \ge \mathbf{0} \end{array}$$

$$\mathtt{Tp} = \begin{array}{c:c} & \begin{array}{cc:ccc} & \mathbf{x} & & \mathbf{s} & \end{array} \\ \hdashline \begin{array}{c:c} 0 & \mathbf{c}^\top \\ \hdashline \mathbf{-b} & \mathbf{-A} \end{array} & \left[\begin{array}{c:ccc} 0 & \cdots & 0 \\ \hdashline 1 & & 0 \\ 0 & \ddots & 0 \\ 0 & & 1 \end{array}\right] \end{array} \begin{array}{c} \\ \\ m \\ \\ \end{array}$$

$$\begin{array}{cc} n & m \end{array}$$

$$\mathtt{Td} = \begin{array}{c:c} & \begin{array}{cc:ccc} & \mathbf{y} & & \mathbf{w} & \end{array} \\ \hdashline \begin{array}{c:c} 0 & \mathbf{-b}^\top \\ \hdashline \mathbf{c} & \mathbf{A}^\top \end{array} & \left[\begin{array}{c:ccc} 0 & \cdots & 0 \\ \hdashline 1 & & 0 \\ 0 & \ddots & 0 \\ 0 & & 1 \end{array}\right] \end{array} \begin{array}{c} \\ \\ n \\ \\ \end{array}$$

$$\begin{array}{cc} m & n \end{array}$$

Much can be learned about linear programming duality by studying numerical examples, so to facilitate experimentation I wrote the `duals.m` routine listed below.

```
function [Tp,Tpstar,Td,Tdstar]=duals(A,b,c)
% construct and solve both problems in the standard dual pair

m=size(A,1);
n=size(A,2);

Tp=[ 0, c',zeros(1,m);
    -b,-A, eye(m)    ];
[xs,rc,Tpstar]=simplex(Tp,m,n+m);

Td=[0,-b',zeros(1,n);
    c, A',eye(n)     ];
[yw,rc,Tdstar]=simplex(Td,n,m+n);
```

This code uses the built-in MATLAB functions `eye`, which returns an identity matrix, and `zeros`, which returns a matrix of all zeros.

From given data this routine constructs initial tableaus for the primal 7-8 and dual 11-12 and 9,13 uses `simplex.m` to pivot them to final form. In the Octave session below I used `duals.m` to solve the `brewery` problem and its dual, after giving `A`, `b`, and `c` values that put the problems in the form of $\mathscr{P}$ and $\mathscr{D}$ above.

```
octave:1> A=[-7,-10,-8,-12;-1,-3,-1,-1;-2,-4,-1,-3]
A =

   -7  -10   -8  -12
   -1   -3   -1   -1
   -2   -4   -1   -3

octave:2> b=[-160;-50;-60]
b =

  -160
   -50
   -60

octave:3> c=[-90;-150;-60;-70]
c =

   -90
  -150
   -60
   -70

octave:4> format bank
octave:5> [Tp,Tpstar,Td,Tdstar]=duals(A,b,c)
Tp =

    0.00  -90.00  -150.00  -60.00  -70.00    0.00    0.00    0.00
  160.00    7.00   10.00    8.00   12.00    1.00    0.00    0.00
   50.00    1.00    3.00    1.00    1.00    0.00    1.00    0.00
   60.00    2.00    4.00    1.00    3.00    0.00    0.00    1.00

Tpstar =

  2325.00     0.00     0.00    18.75    76.25     7.50     0.00    18.75
     5.00     1.00     0.00     2.75     2.25     0.50     0.00    -1.25
    12.50     0.00     1.00    -1.12    -0.37    -0.25     0.00     0.88
     7.50     0.00     0.00     1.62    -0.13     0.25     1.00    -1.37

Td =

    0.00  160.00   50.00   60.00    0.00    0.00    0.00    0.00
  -90.00   -7.00   -1.00   -2.00    1.00    0.00    0.00    0.00
  -150.00  -10.00   -3.00   -4.00    0.00    1.00    0.00    0.00
  -60.00   -8.00   -1.00   -1.00    0.00    0.00    1.00    0.00
  -70.00  -12.00   -1.00   -3.00    0.00    0.00    0.00    1.00

Tdstar =

  -2325.00     0.00     7.50     0.00     5.00    12.50     0.00     0.00
     18.75     0.00    -1.62     0.00    -2.75     1.12     1.00     0.00
     76.25     0.00     0.13     0.00    -2.25     0.37     0.00     1.00
     18.75     0.00     1.37     1.00     1.25    -0.87     0.00     0.00
      7.50     1.00    -0.25     0.00    -0.50     0.25     0.00     0.00
```


Tableaus `Tp` and `Tpstar` are recognizable from §2.2 and §4.1; we derived the dual of the `brewery` problem in §5.1.4. Notice that $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$ and $\mathbf{y}^\star = [7\frac{1}{2}, 0, 18\frac{3}{4}]^\top$ are each in both optimal tableaus. From a tableau in the form of either `Tp` or `Td` it is easy to extract `A`, `b`, and `c`, which can then be used to construct the other tableau.

5.3 Efficiency Considerations

In §5.2.3 we constructed tableaus for the problems in our standard dual pair. Because of the slack variables they each have $n+m$ columns, but `Tp` has m constraint rows while `Td` has n. The constraint coefficient matrices are negative transposes of each other and are seldom square, so typically $m \neq n$ and one problem has more constraints than the other. We saw in §5.1.2 how to find $\mathbf{x}^\star$ and $\mathbf{y}^\star$ in both optimal tableaus, so we can solve either problem. But as I mentioned in §4.5.3, the number of phase-2 pivots required by the simplex algorithm is observed in practice to depend on the number of constraints, and this suggests that one problem in a dual pair might be easier to solve than the other.

5.3.1 Tall & Thin vs Short & Fat

To investigate this idea consider the following `dp5` pair [3, §5.6] in which $\mathbf{A}$ is tall and thin so that $\mathbf{A}^\top$ is short and fat.

$$\mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \begin{bmatrix} -1 & -1 \end{bmatrix}\mathbf{x}$$

$$\text{subject to} \underbrace{\begin{bmatrix} 3 & -1 \\ 2 & -1 \\ 1 & -1 \\ 0 & -1 \\ -1 & 3 \\ -1 & 2 \\ -1 & 1 \\ -1 & 0 \end{bmatrix}}_{\mathbf{A}} \mathbf{x} \geq \begin{bmatrix} 0 \\ -1 \\ -3 \\ -6 \\ 0 \\ -1 \\ -3 \\ -6 \end{bmatrix}$$

$$\mathbf{x} \geq \mathbf{0}$$

$$\mathscr{D}: \underset{\mathbf{y}\in\mathbb{R}^8}{\text{maximize}} \begin{bmatrix} 0 & -1 & -3 & -6 & 0 & -1 & -3 & -6 \end{bmatrix}\mathbf{y}$$

$$\text{subject to} \underbrace{\begin{bmatrix} 3 & 2 & 1 & 0 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & 3 & 2 & 1 & 0 \end{bmatrix}}_{\mathbf{A}^\top} \mathbf{y} \leq \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

$$\mathbf{y} \geq \mathbf{0}$$

I put each of these problems into standard form and solved it by following the pivot rules of the simplex algorithm that we developed in §2, as shown on the next page. In the right-hand `pivot` session I read the tableau for the primal problem and then used the `DUAL` command

to find the dual. The initial tableau for $\mathscr{P}$ is in canonical form, so only phase-2 pivots are required to reach optimal form. Because the primal tableau has negative cost coefficients, the initial tableau for $\mathscr{D}$ has negative constant-column entries and subproblem pivots are needed to put it into canonical form; then a single phase-2 pivot reaches optimality.

```
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
< read thin.tab
Reading the tableau...
...done.

     x1  x2  s1  s2  s3  s4  s5  s6  s7  s8
 0. -1. -1.  0.  0.  0.  0.  0.  0.  0.  0.
 0. -3.  1.  1.  0.  0.  0.  0.  0.  0.  0.
 1. -2.  1.  0.  1.  0.  0.  0.  0.  0.  0.
 3. -1.  1.  0.  0.  1.  0.  0.  0.  0.  0.
 6.  0.  1.  0.  0.  0.  1.  0.  0.  0.  0.
 0.  1. -3.  0.  0.  0.  0.  1.  0.  0.  0.
 1.  1. -2.  0.  0.  0.  0.  0.  1.  0.  0.
 3.  1. -1.  0.  0.  0.  0.  0.  0.  1.  0.
 6.  1.  0.  0.  0.  0.  0.  0.  0.  0.  1.

< * phase-2 simplex algorithm pivots
< pivot 6 2;
< pivot 7 3;
< pivot 8 8;
< pivot 9 9;
< pivot 5 10

      x1  x2  s1  s2  s3  s4  s5  s6  s7  s8
 12.  0.  0.  0.  0.  0.  1.  0.  0.  0.  1.
 12.  0.  0.  1.  0.  0. -1.  0.  0.  0.  3.
  7.  0.  0.  0.  1.  0. -1.  0.  0.  0.  2.
  3.  0.  0.  0.  0.  1. -1.  0.  0.  0.  1.
  3.  0.  0.  0.  0.  0.  1.  0.  0.  1. -1.
  6.  1.  0.  0.  0.  0.  0.  0.  0.  0.  1.
  6.  0.  1.  0.  0.  0.  1.  0.  0.  0.  0.
 12.  0.  0.  0.  0.  0.  3.  1.  0.  0. -1.
  7.  0.  0.  0.  0.  0.  2.  0.  1.  0. -1.

< * optimal form achieved in 5 pivots
```

```
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
< read thin.tab;
Reading the tableau...
...done.
< dual

     y1  y2  y3  y4  y5  y6  y7  y8  w1  w2
 0.  0.  1.  3.  6.  0.  1.  3.  6.  0.  0.
-1.  3.  2.  1.  0. -1. -1. -1. -1.  1.  0.
-1. -1. -1. -1. -1.  3.  2.  1.  0.  0.  1.

< * make b1 positive
< p 2 6

     y1  y2  y3  y4  y5  y6  y7  y8  w1  w2
 0.  0.  1.  3.  6.  0.  1.  3.  6.  0.  0.
 1. -3. -2. -1.  0.  1.  1.  1.  1. -1.  0.
-4.  8.  5.  2. -1.  0. -1. -2. -3.  3.  1.

< * pivot b2 subproblem toward optimality
< p 2 9

     y1   y2   y3  y4  y5  y6  y7  y8  w1  w2
-6.  18.  13.  9.  6. -6. -5. -3.  0.  6.  0.
 1.  -3.  -2. -1.  0.  1.  1.  1.  1. -1.  0.
-1.  -1.  -1. -1. -1.  3.  2.  1.  0.  0.  1.

< * unbounded subproblem; pivot in objective row
< p 3 2

      y1  y2  y3  y4   y5   y6   y7   y8  w1  w2
-24.  0. -5. -9. -12.  48.  31.  15.  0.  6.  18.
  4.  0.  1.  2.   3.  -8.  -5.  -2.  1. -1.  -3.
  1.  1.  1.  1.   1.  -3.  -2.  -1.  0.  0.  -1.

< * phase 2 simplex algorithm pivot
< p 3 5

      y1   y2  y3  y4  y5   y6  y7  y8  w1  w2
-12.  12.  7.  3.  0.  12.  7.  3.  0.  6.  6.
  1.  -3. -2. -1.  0.   1.  1.  1.  1. -1.  0.
  1.   1.  1.  1.  1.  -3. -2. -1.  0.  0. -1.

< * optimal form achieved in 4 pivots
```

floating point operation	$\mathscr{P}$		$\mathscr{D}$	
	1 pivot	5 pivots	1 pivot	4 pivots
/	8	40	2	8
*	27	135	27	108
-	24	120	18	72

As discussed in §4.2, one pivot in a tableau having $m+1$ rows and $n+1$ columns requires m divisions, $(1+n-m)(m+1)$ multiplications, and $(1+n-m)m$ subtractions (see Exercise 4.6.16). From the table of operation counts given above for this example it is clear that solving the short fat problem takes less work than solving the tall thin one.

5.3.2 The Dual Simplex Method

To solve the dual in §5.3.1, I first had to construct a tableau for that problem. Then I could use phase 1 and phase 2 of the simplex method to pivot the dual tableau to optimal form. The dual simplex method instead solves the dual by pivoting in the primal tableau.

In §5.2.3 we constructed these tableaus to represent the problems in our standard dual pair. If $\mathbf{c} \geq \mathbf{0}$ then tableau $\mathbf{D}$ is in canonical form, and to put it into optimal form we would perform minimum-ratio pivots to make $-\mathbf{b}^\top \geq \mathbf{0}^\top$ while keeping $\mathbf{c} \geq \mathbf{0}$.

$$\mathbf{D} = \begin{array}{c|c:c} & \mathbf{y} & \mathbf{w} \\ \hline 0 & -\mathbf{b}^\top & \mathbf{0}^\top \\ \hline \mathbf{c} & \mathbf{A}^\top & \mathbf{I}_{n\times n} \end{array} \qquad \mathbf{P} = \begin{array}{c|c:c} & \mathbf{x} & \mathbf{s} \\ \hline 0 & \mathbf{c}^\top & \mathbf{0}^\top \\ \hline -\mathbf{b} & -\mathbf{A} & \mathbf{I}_{m\times m} \end{array}$$

Because $\mathbf{c}^\top \geq \mathbf{0}$, tableau $\mathbf{P}$ would be in optimal form if $-\mathbf{b}$ were nonnegative. We can make $-\mathbf{b} \geq \mathbf{0}$ while keeping $\mathbf{c}^\top \geq \mathbf{0}^\top$ by performing **dual simplex pivots** in tableau $\mathbf{P}$.

To see how a dual simplex pivot works consider the `dp6` example below [3, §5.5] in which our goal is to solve the problem described by tableau $\mathbf{P}_0$. Comparing this tableau to the template on the right above we can recover the values of $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{c}$; then using them in the template on the left above yields $\mathbf{D}_0$. Tableaus $\mathbf{P}_0$ and $\mathbf{D}_0$, because they describe problems that are duals of each other, are said to be **dual tableaus**. One consequence of the fact that these are dual tableaus is that the entries in the nonbasic columns of the first constraint row in $\mathbf{P}_0$ appear with signs changed in the constraint rows of the y_1 column in $\mathbf{D}_0$.

$$\mathbf{D}_0 = \begin{array}{c|c:ccc:cccc} & y_1 & y_2 & y_3 & w_1 & w_2 & w_3 & w_4 \\ \hline 0 & 50 & 5 & -10 & 0 & 0 & 0 & 0 \\ \hline 2 & -1 & -1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & -1 & 1 & 0 & 1 & 0 & 0 \\ 5 & 1 & -3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 2 & 1 & 0 & 0 & 0 & 0 & 1 \end{array} \qquad \mathbf{P}_0 = \begin{array}{c|cccc:ccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 0 & 2 & 1 & 5 & 4 & 0 & 0 & 0 \\ \hline 50 & 1 & -2 & -1 & -2 & 1 & 0 & 0 \\ 5 & 1 & 1 & 3 & -1 & 0 & 1 & 0 \\ -10 & -1 & -1 & 0 & 0 & 0 & 0 & 1 \end{array}$$

$\mathbf{A}^\top$ $\qquad$ $-\mathbf{A}$

$(2,3)$ element of $\mathbf{A}^\top$ corresponds to $(3,2)$ element of $-\mathbf{A}$
simplex pivot in $\mathbf{D}$ corresponds to dual simplex pivot in $\mathbf{P}$

$$\mathbf{D}_1 = \begin{array}{c|c:ccc:cccc} & y_1 & y_2 & y_3 & w_1 & w_2 & w_3 & w_4 \\ \hline 10 & 70 & -5 & 0 & 0 & 10 & 0 & 0 \\ \hline 1 & -3 & 0 & 0 & 1 & -1 & 0 & 0 \\ 1 & 2 & -1 & 1 & 0 & 1 & 0 & 0 \\ 5 & 1 & -3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 2 & 1 & 0 & 0 & 0 & 0 & 1 \end{array} \qquad \mathbf{P}_1 = \begin{array}{c|cccc:ccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline -10 & 1 & 0 & 5 & 4 & 0 & 0 & 1 \\ \hline 70 & 3 & 0 & -1 & -2 & 1 & 0 & -2 \\ -5 & 0 & 0 & 3 & -1 & 0 & 1 & 1 \\ 10 & 1 & 1 & 0 & 0 & 0 & 0 & -1 \end{array}$$

In $\mathbf{D}_0$ it is easy to see that the next step in solving the dual is to pivot on the circled element $(2,3)$ of $\mathbf{A}^\top$. That element of $\mathbf{A}^\top$ in $\mathbf{D}$ corresponds to the $(3,2)$ element of $-\mathbf{A}$ in $\mathbf{P}$.

Performing the pivots yields $\mathbf{D}_1$ and $\mathbf{P}_1$, and these are also dual tableaus. The entries in the nonbasic columns of the first constraint row in $\mathbf{P}_1$ again appear with signs changed in the constraint rows of the y_1 column in $\mathbf{D}_1$, though in a different order due to the pivots.

Performing a simplex pivot in $\mathbf{D}$ and the corresponding dual-simplex pivot in $\mathbf{P}$ yield tableaus that are duals of each other, so the pivots are equivalent. Just as simplex-rule pivots in a canonical-form tableau $\mathbf{D}$ lead to either optimal or unbounded form, dual-simplex-rule pivots in $\mathbf{P}$ lead to either optimal or infeasible form (assuming neither problem cycles). Because $\mathbf{D}$ is feasible (it is in canonical form) $\mathbf{P}$ cannot be unbounded.

To perform the simplex pivot in $\mathbf{D}_0$ we used the rule we derived in §2.4.4, which can be restated in terms of the variables in the $\mathbf{D}$ template like this.

- choose h so that $-b_h < 0$;
- choose p so that

$$\frac{c_p}{a^\top_{ph}} = \min_j \left\{ \frac{c_j}{a^\top_{jh}} \,\middle|\, a^\top_{jh} > 0 \right\} \qquad \text{or} \qquad \frac{c_p}{a_{hp}} = \min_j \left\{ \frac{c_j}{a_{hj}} \,\middle|\, a_{hj} > 0 \right\}$$

Here the (j, i) element of $\mathbf{A}^\top$ is $a^\top_{ji} = a_{ij}$. Applying this rule to $\mathbf{P}_0$ we find that only the third element of $-\mathbf{b}$ is negative, $-b_3 = -10$, so the pivot row is $h = 3$. To find the pivot column we must compute the ratios of the c_j to the positive a_{3j}. The numbers appearing in the $-\mathbf{A}$ part of tableau $\mathbf{P}_0$ are the *negatives* of the a_{ij} so the columns we want are those having *negative* entries in row 3. But the ratios involve $+a_{ij}$ so in calculating them we must use the negatives of those entries to find

$$\begin{aligned} \frac{c_1}{a_{31}} &= \frac{2}{-(-a_{31})} = \frac{2}{-(-1)} = \frac{2}{1} = 2 \\ \frac{c_2}{a_{32}} &= \frac{1}{-(-a_{32})} = \frac{1}{-(-1)} = \frac{1}{1} = 1 \end{aligned}$$

and pick the minimum-ratio column $p = 2$.

It is easy to miss a sign change in this process, so you might find it simpler to remember the dual-simplex pivot rule in terms of the primal tableau entries.

- choose a pivot row h that has a negative constant-column entry;
- in that row, for each column j that has a negative entry T_{hj} find the ratio $c_j/(-T_{hj})$;
- choose as the pivot column one that has the minimum ratio.

The `pivot` session on the next page uses this pivot rule to solve the problem described by tableau $\mathbf{P}_0$.

We assumed at the beginning that $\mathbf{c}^\top \geq \mathbf{0}^\top$ so the process illustrated above can be viewed as phase 2 of the dual simplex method. If pivoting-in a basis leaves some costs negative, a dual version of the subproblem technique can be used to make $\mathbf{c}^\top \geq \mathbf{0}$; thus the whole primal simplex algorithm can be performed on the dual (without ever writing it down) by pivoting in the primal. There are also primal-dual algorithms [107, §4.6] [162, §3.4] that combine aspects of both. These topics are, unfortunately, beyond the scope of this introduction.


```
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
< read P.tab
Reading the tableau...
...done.

      x1  x2  x3  x4  s1  s2  s3
  0.  2.  1.  5.  4.  0.  0.  0.
 50.  1. -2. -1. -2.  1.  0.  0.
  5.  1.  1.  3. -1.  0.  1.  0.
-10. -1. -1.  0.  0.  0.  0.  1.

< * the most-negative constant-column entry is -10 at i=4
< * that row has negative constraint coefficients -1 and -1
< * the column ratios are 2/[-(-1)]=2 and 1/[-(-1)]=1
< * so the pivot column is the x2 column and j=3
< p 4 3

      x1  x2  x3  x4  s1  s2  s3
-10.  1.  0.  5.  4.  0.  0.  1.
 70.  3.  0. -1. -2.  1.  0. -2.
 -5.  0.  0.  3. -1.  0.  1.  1.
 10.  1.  1.  0.  0.  0.  0. -1.

< * the most-negative constant-column entry is -5 at i=3
< * that row has a single negative entry at j=5
< p 3 5

      x1  x2  x3   x4  s1  s2  s3
-30.  1.  0.  17.  0.  0.  4.  5.
 80.  3.  0.  -7.  0.  1. -2. -4.
  5.  0.  0.  -3.  1.  0. -1. -1.
 10.  1.  1.   0.  0.  0.  0. -1.

< * optimal form
```

5.4 Sensitivity Analysis

In §5.1.4 we asked the following questions about the `brewery` model; we took the answer to the first as the answer to the second, but they are actually not quite the same.

> If Sarah decreases her supply of pale malt by exactly 1 pound, what will happen to her revenue from selling beer?
>
> How much should Sarah charge per pound of pale malt in order to keep her total revenue, from selling both beer and malt, constant?

To answer the first question we might simply change the available resource in the starting tableau and solve the modified problem. The original `brewery` model yields an optimal revenue of $-z^\star = 2325$ while the modified version yields $-\bar{z}^\star = 2317\frac{1}{2}$, so if Sarah sells a pound of pale malt she will make $2325 - 2317\frac{1}{2} = \$7.50$ less from selling beer.

The answer to the second question, how much Sarah should charge per pound of pale malt, depends on how much she sells. We found that she can sell up to 10 pounds at \$7.50 per pound without changing her total revenue, but for every pound she sells beyond that she will need to charge more. One could approximate a price-versus-quantity curve for pale malt by brute computation, but that would require the solution of many models each assuming that she sells a different quantity. To study how shadow price depends on quantity it is easier to use algebraic manipulations of the optimal tableau as we did in §5.1.4.

5.4.1 Changes to Problem Data

We begin our study of sensitivity analysis by taking up questions of the first kind, which are about specific changes to the numbers in a model. In practical applications of linear programming it is often useful to know what happens to the optimal solution of a resource allocation problem when changes are made to the available resources, the selling prices of the products, or the technology coefficients that appear in the constraint equations. The effect of all these changes, taken singly or in combination, can be discovered by revising the initial tableau and solving the modified problem from scratch. But if we know the optimal tableau for the unmodified problem, we can usually find the optimal tableau for the modified problem with much less work, especially if the perturbations to the data are small.

We know these starting and optimal tableaus for the unmodified `brewery` problem, so by inspection we can write down the pivot matrix $\mathbf{P}$ that makes $\mathbf{P}\mathbf{T}_0 = \mathbf{T}^\star$.

$$\mathbf{T}_0 = \begin{array}{c|ccccccc|l} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 & \\ \hline 0 & -90 & -150 & -60 & -70 & 0 & 0 & 0 & \\ \hline 160 & 7 & 10 & 8 & 12 & 1 & 0 & 0 & \text{pale malt} \\ 50 & 1 & 3 & 1 & 1 & 0 & 1 & 0 & \text{black malt} \\ 60 & 2 & 4 & 1 & 3 & 0 & 0 & 1 & \text{hops} \end{array}$$

variable	kegs of
x_1	Porter
x_2	Stout
x_3	Lager
x_4	IPA

$$\mathbf{T}^\star = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 2325 & 0 & 0 & 18\tfrac{3}{4} & 76\tfrac{1}{4} & 7\tfrac{1}{2} & 0 & 18\tfrac{3}{4} \\ \hline 5 & 1 & 0 & 2\tfrac{3}{4} & 2\tfrac{1}{4} & \tfrac{1}{2} & 0 & -1\tfrac{1}{4} \\ 12\tfrac{1}{2} & 0 & 1 & -1\tfrac{1}{8} & -\tfrac{3}{8} & -\tfrac{1}{4} & 0 & \tfrac{7}{8} \\ 7\tfrac{1}{2} & 0 & 0 & 1\tfrac{5}{8} & -\tfrac{1}{8} & \tfrac{1}{4} & 1 & -1\tfrac{3}{8} \end{array}$$

$$\mathbf{P} = \begin{bmatrix} 1 & 7\tfrac{1}{2} & 0 & 18\tfrac{3}{4} \\ 0 & \tfrac{1}{2} & 0 & -1\tfrac{1}{4} \\ 0 & -\tfrac{1}{4} & 0 & \tfrac{7}{8} \\ 0 & \tfrac{1}{4} & 1 & -1\tfrac{3}{8} \end{bmatrix}$$

To solve the problem described by a modified initial tableau $\overline{\mathbf{T}}_0$, we can begin by computing $\mathbf{P}\overline{\mathbf{T}}_0$. If that tableau happens to be in optimal form then it is the modified optimal tableau $\overline{\mathbf{T}}^\star$; if not it provides a **hot start** for completing the solution of the modified problem. If in $\mathbf{P}\overline{\mathbf{T}}_0$ some b_i became negative but $\mathbf{c}^\top \geq \mathbf{0}^\top$, then dual simplex pivots can be used to restore canonical (and hence optimal) form; if some c_j became negative but $\mathbf{b} \geq \mathbf{0}$ then primal simplex pivots can be used to restore optimal form.

If Sarah sold one pound of pale malt that would change b_1 from 160 to 159 in the starting tableau for the `brewery` problem. To get close to the optimal tableau for the modified problem we can perform the same pivots that solved the unmodified problem, by computing this matrix product.

$$\mathbf{P}\overline{\mathbf{T}}_0 = \begin{bmatrix} 1 & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ 0 & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 0 & -\frac{1}{4} & 0 & \frac{7}{8} \\ 0 & \frac{1}{4} & 1 & -1\frac{3}{8} \end{bmatrix} \begin{array}{|r|rrrrrrr|} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 0 & -90 & -150 & -60 & -70 & 0 & 0 & 0 \\ \hline 159 & 7 & 10 & 8 & 12 & 1 & 0 & 0 \\ 50 & 1 & 3 & 1 & 1 & 0 & 1 & 0 \\ 60 & 2 & 4 & 1 & 3 & 0 & 0 & 1 \\ \hline \end{array}$$

decrease pale malt by 1 pound

$$= \begin{array}{|r|rrrrrrr|} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 2317\frac{1}{2} & 0 & 0 & 18\frac{3}{4} & 76\frac{1}{4} & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ \hline 4\frac{1}{2} & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 12\frac{3}{4} & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & -\frac{1}{4} & 0 & \frac{7}{8} \\ 7\frac{1}{4} & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & \frac{1}{4} & 1 & -1\frac{3}{8} \\ \hline \end{array} = \overline{\mathbf{P}}^\star$$

The resulting tableau is in optimal form, so it is the optimal tableau for the modified problem and we can read off $-\bar{z}^\star = 2317\frac{1}{2}$. If we really care only about this one number, I could have saved some work by finding only the first row and column of this tableau to confirm that it is in optimal form. As it turned out the prospective buyer in the story of §5.1.4 wasn't willing to spend \$7.50 for the pound of pale malt, so Sarah kept her stock at 160 pounds.

Another local brewer who makes only India Pale Ale told Sarah that he might go out of business, and in that case he would give her the 10 ounces of hops he had in stock. This resource is used up in $\mathbf{T}^\star$ ($s_3^\star = 0$) so having more of it might let Sarah brew more beer, and with one less competitor selling IPA she thought she could increase her price for that product to \$75 per keg. What would her new optimal production program be in that scenario?

$$\mathbf{P}\hat{\mathbf{T}}_0 = \begin{bmatrix} 1 & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ 0 & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 0 & -\frac{1}{4} & 0 & \frac{7}{8} \\ 0 & \frac{1}{4} & 1 & -1\frac{3}{8} \end{bmatrix} \begin{array}{|r|rrrrrrr|} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 0 & -90 & -150 & -60 & -75 & 0 & 0 & 0 \\ \hline 160 & 7 & 10 & 8 & 12 & 1 & 0 & 0 \\ 50 & 1 & 3 & 1 & 1 & 0 & 1 & 0 \\ 70 & 2 & 4 & 1 & 3 & 0 & 0 & 1 \\ \hline \end{array}$$

increase price by \$5

increase hops by 10 ounces

$$= \begin{array}{|r|rrrrrrr|} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 2512\frac{1}{2} & 0 & 0 & 18\frac{3}{4} & 71\frac{1}{4} & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ \hline -7\frac{1}{2} & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 21\frac{1}{4} & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & -\frac{1}{4} & 0 & \frac{7}{8} \\ -6\frac{1}{4} & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & \frac{1}{4} & 1 & -1\frac{3}{8} \\ \hline \end{array}$$

This time the pivots that solved the unmodified problem produce a tableau that is *not* in optimal form, because two of its constant-column entries are negative. But doing a dual-simplex pivot in the row of the most negative one yields this optimal form.

$$
\begin{array}{r|rrrrrrr}
 & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\
\hline
2400 & 15 & 0 & 60 & 105 & 15 & 0 & 0 \\
\hline
6 & -\frac{8}{10} & 0 & -2\frac{2}{10} & -1\frac{8}{10} & -\frac{4}{10} & 0 & 1 \\
16 & \frac{7}{10} & 1 & \frac{8}{10} & 1\frac{2}{10} & \frac{1}{10} & 0 & 0 \\
2 & -1\frac{1}{10} & 0 & -1\frac{4}{10} & -2\frac{6}{10} & -\frac{3}{10} & 1 & 0
\end{array}
= \hat{\mathbf{P}}^\star
$$

Sarah's new optimal production program would thus be $\hat{\mathbf{x}}^\star = [0, 16, 0, 0]^\top$. As a result of the IPA maker going out of business she would produce only Stout, even though she could now charge more for IPA if she made any. Sarah worried about marketing only one product, but fortunately for IPA lovers this competitor decided not to go out of business after all.

A single change in a linear programming model might affect more than one number in $\mathbf{T}_0$. For example, if in the `twoexams` problem of §1.1.1 the grade that triggers an advisor alert is increased to 65, constraints (A) and (B) are both affected.

5.4.2 Inserting or Deleting Columns

Every autumn Sarah gets inquires about an Oktoberfest beer, so she wants to consider adding that variety to her production program. An internet search leads her to a recipe that includes 5 pounds of pale malt, 2 pounds of black malt, and 2 ounces of hops. To earn the good will of her customers she would be content to sell this specialty product for only \$80 per keg. Would making it be worthwhile?

Letting x_5 represent the kegs of Oktoberfest to make, Sarah inserts the product column into her starting tableau and proceeds as usual.

new product

$$
\mathbf{P}\overline{\mathbf{T}}_0 = \begin{bmatrix} 1 & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ 0 & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 0 & -\frac{1}{4} & 0 & \frac{7}{8} \\ 0 & \frac{1}{4} & 1 & -1\frac{3}{8} \end{bmatrix}
\begin{array}{r|rrrrrrrr}
 & x_1 & x_2 & x_3 & x_4 & x_5 & s_1 & s_2 & s_3 \\
\hline
0 & -90 & -150 & -60 & -70 & -80 & 0 & 0 & 0 \\
\hline
160 & 7 & 10 & 8 & 12 & 5 & 1 & 0 & 0 \\
50 & 1 & 3 & 1 & 1 & 2 & 0 & 1 & 0 \\
60 & 2 & 4 & 1 & 3 & 2 & 0 & 0 & 1
\end{array}
$$

$$
=
\begin{array}{r|rrrrrrrr}
 & x_1 & x_2 & x_3 & x_4 & x_5 & s_1 & s_2 & s_3 \\
\hline
2325 & 0 & 0 & 18\frac{3}{4} & 76\frac{1}{4} & -5 & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\
\hline
5 & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & 0 & \frac{1}{2} & 0 & -1\frac{1}{4} \\
12\frac{1}{2} & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & \frac{1}{2} & -\frac{1}{4} & 0 & \frac{7}{8} \\
7\frac{1}{2} & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & \frac{1}{2} & \frac{1}{4} & 1 & -1\frac{3}{8}
\end{array}
$$

If in $\mathbf{P}\overline{\mathbf{T}}_0$ the reduced cost over the x_5 column had turned out to be positive then Sarah's original production program would have remained optimal and it would not be worthwhile to make Oktoberfest. If Sarah wanted to know only that, I could have saved some work in finding the matrix product by calculating only c_5; if she wants to know the other consequences of adding the new product we can do a primal simplex pivot in the x_5 column to get this optimal form.

$$
\begin{array}{r|cccccccc}
 & x_1 & x_2 & x_3 & x_4 & x_5 & s_1 & s_2 & s_3 \\
\hline
2400 & 0 & 0 & 35 & 75 & 0 & 10 & 10 & 5 \\
\hline
5 & 1 & 0 & 2\tfrac{3}{4} & 2\tfrac{1}{4} & 0 & \tfrac{1}{2} & 0 & -1\tfrac{1}{4} \\
5 & 0 & 1 & -2\tfrac{3}{4} & -\tfrac{1}{4} & 0 & -\tfrac{1}{2} & -1 & 2\tfrac{1}{4} \\
15 & 0 & 0 & 3\tfrac{1}{4} & -\tfrac{1}{4} & 1 & \tfrac{1}{2} & 2 & -2\tfrac{3}{4}
\end{array}
= \overline{\mathbf{T}}^\star
$$

Deleting from $\mathbf{T}_0$ a column that is nonbasic in $\mathbf{T}^\star$ is trivial, because if the product is not being made it can be removed from both tableaus without changing the optimal program. Deleting from $\mathbf{T}_0$ a column that is basic in $\mathbf{T}^\star$ is trickier, because in that case $\mathbf{PT}_0$ will lack a basis. If, instead of adding Oktoberfest, Sarah stopped making Stout then we would get this guess at a new optimal tableau.

deleted product

$$
\mathbf{P}\hat{\mathbf{T}}_0 = \begin{bmatrix} 1 & 7\tfrac{1}{2} & 0 & 18\tfrac{3}{4} \\ 0 & \tfrac{1}{2} & 0 & -1\tfrac{1}{4} \\ 0 & -\tfrac{1}{4} & 0 & \tfrac{7}{8} \\ 0 & \tfrac{1}{4} & 1 & -1\tfrac{3}{8} \end{bmatrix}
\begin{array}{r|cccccc}
 & x_1 & x_3 & x_4 & s_1 & s_2 & s_3 \\
\hline
0 & -90 & -60 & -70 & 0 & 0 & 0 \\
\hline
160 & 7 & 8 & 12 & 1 & 0 & 0 \\
50 & 1 & 1 & 1 & 0 & 1 & 0 \\
60 & 2 & 1 & 3 & 0 & 0 & 1
\end{array}
$$

$$
= \begin{array}{r|cccccc}
 & x_1 & x_3 & x_4 & s_1 & s_2 & s_3 \\
\hline
2325 & 0 & 18\tfrac{3}{4} & 76\tfrac{1}{4} & 7\tfrac{1}{2} & 0 & 18\tfrac{3}{4} \\
\hline
5 & 1 & 2\tfrac{3}{4} & 2\tfrac{1}{4} & \tfrac{1}{2} & 0 & -1\tfrac{1}{4} \\
12\tfrac{1}{2} & 0 & -1\tfrac{1}{8} & -\tfrac{3}{8} & -\tfrac{1}{4} & 0 & \tfrac{7}{8} \\
7\tfrac{1}{2} & 0 & 1\tfrac{5}{8} & -\tfrac{1}{8} & \tfrac{1}{4} & 1 & -1\tfrac{3}{8}
\end{array}
$$

Now there is no identity column whose 1 is in the second constraint row, so I pivoted on the positive entry in that row (if there were more than one, picking an entry having the minimum ratio c_j/a_{hj} would keep the pivot from making some cost coefficient negative).

$$
\begin{array}{r|cccccc}
 & x_1 & x_3 & x_4 & s_1 & s_2 & s_3 \\
\hline
2057\tfrac{1}{7} & 0 & 42\tfrac{6}{7} & 84\tfrac{2}{7} & 12\tfrac{6}{7} & 0 & 0 \\
\hline
22\tfrac{6}{7} & 1 & 1\tfrac{1}{7} & 1\tfrac{5}{7} & \tfrac{1}{7} & 0 & 0 \\
14\tfrac{2}{7} & 0 & -1\tfrac{2}{7} & -\tfrac{3}{7} & -\tfrac{2}{7} & 0 & 1 \\
27\tfrac{1}{7} & 0 & -\tfrac{1}{7} & -\tfrac{5}{7} & -\tfrac{1}{7} & 1 & 0
\end{array}
= \hat{\mathbf{T}}^\star
$$

5.4.3 Inserting or Deleting Rows

Our original formulation of the `brewery` problem in §1.3.1 did not require Sarah to produce a certain amount of any product, and her unmodified optimal production program includes no Lager or IPA. A tavern that buys her beer might find this inconvenient and request that she supply at least 1 keg of Lager. The simplest way to enforce that condition is by appending the constraint $x_3 \geq 1$ or $-x_3 + s_4 = -1$ to the *optimal* tableau; then one simplex pivot restores optimal form (in general some dual simplex pivots might also be needed).

```
< read brewopt.tab
Reading the tableau...
...done.

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375

< * this is T* for the unmodified problem
< append 1 1

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750  0.
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250  0.
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875  0.
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375  0.
    0.0  0.  0.   0.000   0.000  0.00  0.   0.000  0.

< * add constraint -1=-x3+s4
< insert 5 0
T( 5, 1)... = -1 0 0 -1 0 0 0 0 1

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750  0.
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250  0.
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875  0.
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375  0.
   -1.0  0.  0.  -1.000   0.000  0.00  0.   0.000  1.

< * pivot in the added row to make x3 basic
< pivot 5 4

           x1  x2  x3  x4      s1    s2  s3
 2306.250  0.  0.  0.  76.250  7.50  0.  18.750  18.750
    2.250  1.  0.  0.   2.250  0.50  0.  -1.250   2.750
   13.625  0.  1.  0.  -0.375 -0.25  0.   0.875  -1.125
    5.875  0.  0.  0.  -0.125  0.25  1.  -1.375   1.625
    1.000  0.  0.  1.   0.000  0.00  0.   0.000  -1.000
```

Production requirements can also be enforced [3, §6.2] by moving columns as we did in §5.1.4 but that approach, natural for hand calculation, is much harder to implement in code.

The technique illustrated above can also be used to add constraints that are not bounds [3, p156]. The operations required to restore optimal form are then case-specific and more complicated, but might still be easier than solving a modified problem from scratch.

To remove a constraint it is necessary to delete both its tableau row and its slack variable column. If the slack is positive at optimality this is trivial, because a constraint that is not active does not affect the optimal point. If the slack is nonbasic in $\mathbf{T}^\star$ it is necessary to first make it basic, as in this example of removing the first constraint from the `brewery` model.

```
< read brewopt.tab
Reading the tableau...
...done.

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375

< * make s1 basic so it is not in the other equations
< pivot 2 6

        x1    x2  x3     x4     s1  s2  s3
 2250. -15.0  0. -22.50  42.50  0.  0.  37.50
   10.   2.0  0.   5.50   4.50  1.  0.  -2.50
   15.   0.5  1.   0.25   0.75  0.  0.   0.25
    5.  -0.5  0.   0.25  -1.25  0.  1.  -0.75

< * then remove the first constraint row
< delete 2 0

        x1    x2  x3     x4     s1  s2  s3
 2250. -15.0  0. -22.50  42.50  0.  0.  37.50
   15.   0.5  1.   0.25   0.75  0.  0.   0.25
    5.  -0.5  0.   0.25  -1.25  0.  1.  -0.75

< * and remove the s1 column
< delete 0 6

        x1    x2  x3     x4     s2  s3
 2250. -15.0  0. -22.50  42.50  0.  37.50
   15.   0.5  1.   0.25   0.75  0.   0.25
    5.  -0.5  0.   0.25  -1.25  1.  -0.75

< * now use primal simplex pivots to get optimal form
< pivot 3 4

        x1   x2  x3  x4   s2   s3
 2700. -60.  0.  0. -70.  90. -30.
   10.   1.  1.  0.   2.  -1.   1.
   20.  -2.  0.  1.  -5.   4.  -3.

< pivot 2 2

        x1  x2   x3  x4   s2   s3
 3300.  0.  60.  0.  50.  30.  30.
   10.  1.   1.  0.   2.  -1.   1.
   40.  0.   2.  1.  -1.   2.  -1.
```

The deletions preserve canonical form, so this might be faster than removing the constraint from the original tableau and solving the modified problem from scratch.

5.4.4 Shadow-Price Curves

Finally, we return to the second question of §5.4.0 and find the shadow price of pale malt as a function of how much Sarah sells. This involves repeatedly moving a tableau column to the left of the line, writing inequalities that must be satisfied to maintain canonical form, increasing the value of a nonbasic variable, and pivoting if the variable reaches the minimum row-ratio. As I mentioned in §5.4.3 these algebraic manipulations can also be used [3, §6.2] to study changes in production requirements without adding constraints.

Here again on the left is the optimal tableau for the unmodified `brewery` model, in which s_1 is the amount of pale malt that is left over. The equations represented by this tableau are still satisfied if we move the s_1 column to the other side of the line, as in $\mathbf{T}_1$.

$$\mathbf{T}^\star = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 2325 & 0 & 0 & 18\frac{3}{4} & 76\frac{1}{4} & 7\frac{1}{2} & 0 & 18\frac{3}{4} \\ \hline 5 & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & \frac{1}{2} & 0 & -1\frac{1}{4} \\ 12\frac{1}{2} & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & -\frac{1}{4} & 0 & \frac{7}{8} \\ 7\frac{1}{2} & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & \frac{1}{4} & 1 & -1\frac{3}{8} \end{array} \qquad \mathbf{T}_1 = \begin{array}{c|cccccc} & x_1 & x_2 & x_3 & x_4 & s_2 & s_3 \\ \hline 2325 - 7\frac{1}{2}s_1 & 0 & 0 & 18\frac{3}{4} & 76\frac{1}{4} & 0 & 18\frac{3}{4} \\ \hline 5 - \frac{1}{2}s_1 & 1 & 0 & 2\frac{3}{4} & 2\frac{1}{4} & 0 & -1\frac{1}{4} \\ 12\frac{1}{2} + \frac{1}{4}s_1 & 0 & 1 & -1\frac{1}{8} & -\frac{3}{8} & 0 & \frac{7}{8} \\ 7\frac{1}{2} - \frac{1}{4}s_1 & 0 & 0 & 1\frac{5}{8} & -\frac{1}{8} & 1 & -1\frac{3}{8} \end{array}$$

Now for every unit that we increase s_1 the objective is spoiled by $7\frac{1}{2}$, so the shadow price of pale malt is $y_1 = \$7.50$ per pound. This is true only while $\mathbf{T}_1$ remains in canonical (and thus optimal) form, which is while

$$\left.\begin{array}{rcl} 5 - \frac{1}{2}s_1 & \geq & 0 \;\Rightarrow\; s_1 \leq 10 \\ 12\frac{1}{2} + \frac{1}{4}s_1 & \geq & 0 \;\Rightarrow\; s_1 \geq -50 \\ 7\frac{1}{2} - \frac{1}{4}s_1 & \geq & 0 \;\Rightarrow\; s_1 \leq 30 \end{array}\right\} \Rightarrow s_1 \leq 10.$$

When s_1 reaches 10, $x_1 = 5 - \frac{1}{2}s_1$ reaches zero. Making $s_1 = 10$ and $x_1 = 0$ amounts to a pivot on the circled element of $\mathbf{T}^\star$, yielding the tableau on the left below.

$$\mathbf{T}_2 = \begin{array}{c|ccccccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 & s_3 \\ \hline 2250 & -15 & 0 & -22\frac{1}{2} & 42\frac{1}{2} & 0 & 0 & 37\frac{1}{2} \\ \hline 10 & 2 & 0 & 5\frac{1}{2} & 4\frac{1}{2} & 1 & 0 & -2\frac{1}{2} \\ 15 & \frac{1}{2} & 1 & \frac{1}{4} & \frac{3}{4} & 0 & 0 & \frac{1}{4} \\ 5 & -\frac{1}{2} & 0 & \frac{1}{4} & -1\frac{1}{4} & 0 & 1 & -\frac{3}{4} \end{array} \qquad \mathbf{T}_3 = \begin{array}{c|cccccc} & x_1 & x_2 & x_3 & x_4 & s_1 & s_2 \\ \hline 2250 - 37\frac{1}{2}s_3 & -15 & 0 & -22\frac{1}{2} & 42\frac{1}{2} & 0 & 0 \\ \hline 10 + 2\frac{1}{2}s_3 & 2 & 0 & 5\frac{1}{2} & 4\frac{1}{2} & 1 & 0 \\ 15 - \frac{1}{4}s_3 & \frac{1}{2} & 1 & \frac{1}{4} & \frac{3}{4} & 0 & 0 \\ 5 + \frac{3}{4}s_3 & -\frac{1}{2} & 0 & \frac{1}{4} & -1\frac{1}{4} & 0 & 1 \end{array}$$

We pivoted away from optimality, so $\mathbf{T}_2$ is a suboptimal tableau. In it $s_1 = b_1$ is basic, so the only way to increase s_1 further is to change b_1. The equations represented by this tableau are still satisfied if we move the s_3 column to the other side of the line, as in $\mathbf{T}_3$ (if there were more than one $a_{1j} < 0$ we would move the column having the highest ratio c_j/a_{1j} so as to spoil the objective the least). Now we can increase s_1 further by increasing s_3.

Increasing s_3 by one unit increases s_1 by $\partial s_1/\partial s_3 = 2\frac{1}{2}$ units and decreases the revenue z that Sarah realizes from making beer by $\partial z/\partial s_3 = 37\frac{1}{2}$ units. The shadow price of pale malt is therefore

$$y_1 = \frac{\partial z}{\partial s_1} = \frac{\partial z}{\partial s_3}\frac{\partial s_3}{\partial s_1} = 37\tfrac{1}{2} \times \frac{1}{2\frac{1}{2}} = \$15.00 \text{ per pound.}$$

Tableau $\mathbf{T}_3$ remains in canonical form while

$$\left.\begin{array}{rcl} 10 + 2\frac{1}{2}s_3 & \geq & 0 \;\Rightarrow\; s_3 \geq -4 \\ 15 - \frac{1}{4}s_3 & \geq & 0 \;\Rightarrow\; s_3 \leq 60 \\ 5 + \frac{3}{4}s_3 & \geq & 0 \;\Rightarrow\; s_3 \geq -6\frac{2}{3} \end{array}\right\} \Rightarrow s_3 \leq 60,$$

but when s_3 reaches 60, $x_2 = 15 - \frac{1}{4}s_3$ reaches zero. Making $s_3 = 60$ and $x_2 = 0$ amounts to a pivot on the circled element of $\mathbf{T}_2$, yielding the tableau below.

$\mathbf{T}_0 =$

	x_1	x_2	x_3	x_4	s_1	s_2	s_3
0	−90	−150	−60	−70	0	0	0
160	7	10	8	12	1	0	0
60	2	4	1	3	0	0	1
50	1	3	1	1	0	1	0

Except for a row permutation this is the starting tableau, so I have labeled it $\mathbf{T}_0$. In this canonical form none of the pale malt is used so all of it can be sold; Sarah has given up making beer and is now in the business of selling pale malt.

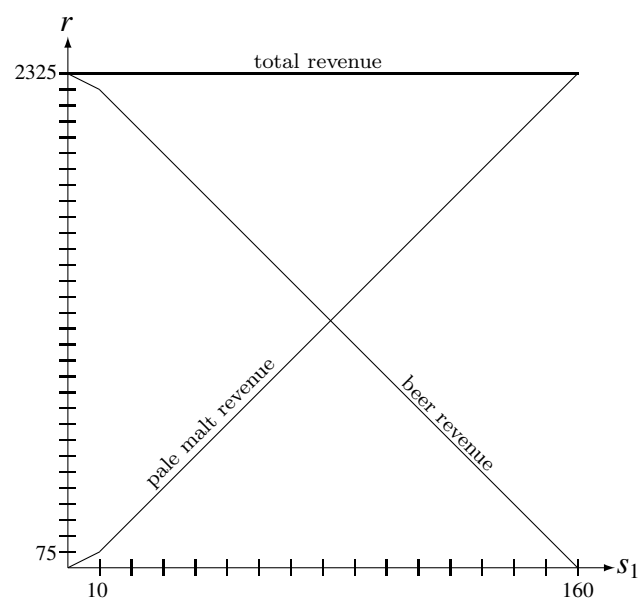
We found that the shadow price of pale malt is

$$y_1 = \begin{cases} 7\frac{1}{2} & \text{for} \quad 0 \leq s_1 \leq 10 \\ 15 & \text{for} \quad 10 \leq s_1 \leq 160 \end{cases}$$

so its sale generates this revenue.

$$r = \begin{cases} 7\frac{1}{2}s_1 & \text{for} \quad 0 \leq s_1 \leq 10 \\ 75 + 15(s_1 - 10) & \text{for} \quad 10 \leq s_1 \leq 160 \end{cases}$$

The graph shows r and the optimal revenue from producing beer as functions of the amount s_1 of pale malt that is sold. These curves have one kink at $s_1 = 10$; in general there are as many segments as there are pivots between $\mathbf{T}^\star$ and $\mathbf{T}_0$.



5.5 Exercises

5.5.1 [E] If one linear program is the *dual* of another, there are certain structural relationships between them. Explain what those structural relationships are (a) in words; (b) by using a diagram.

5.5.2 [E] In the standard dual pair described in the Chapter introduction, the unknown vector is called $\mathbf{x}$ in one problem and $\mathbf{y}$ in the other. (a) Which problem is the minimization problem, and which the maximization? (b) Might you encounter a dual pair in which the variable names are switched? (c) How can you tell which problem in a dual pair is the primal $\mathscr{P}$ and which is the dual $\mathscr{D}$?

5.5.3 [E] In §5.1 we arbitrarily adopted the variable names $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{c}$ for the data arrays of the standard dual pair and we arbitrarily identified one problem as the primal and the other as the dual. (a) Write down the resulting algebraic statement of the standard dual pair. (b) Explain in what sense the algebraic duality relations discussed in §5.1 apply to *all* dual pairs rather than only to this particular one.

5.5.4 [E] Say whether it is possible for both problems in a dual pair to be (a) infeasible; (b) feasible and bounded; (c) feasible but unbounded.

5.5.5 [H] If in our standard dual pair $\bar{\mathbf{x}}$ is feasible for the min problem and $\bar{\mathbf{y}}$ is feasible for the max problem, why must it be true that $\mathbf{c}^\top\bar{\mathbf{x}} \geq \mathbf{b}^\top\bar{\mathbf{y}}$? How does this ensure that neither problem is unbounded?

5.5.6 [H] If $\mathbf{P}$ is an optimal tableau for a primal problem in our standard dual pair and $\mathbf{D}$ is an optimal tableau for the dual problem, $\mathbf{x}^\star$ and $\mathbf{y}^\star$ can both be found in each tableau. (a) Explain where. (b) Why does this happen?

5.5.7 [E] What is a *duality gap,* and why is it zero when $\mathbf{x} = \mathbf{x}^\star$ and $\mathbf{y} = \mathbf{y}^\star$?

5.5.8 [H] If one problem in a dual pair has an optimal vector then so does the other; why? If both have an optimal vector the objective values are equal; why?

5.5.9 [H] In §5.1 we used matrix algebra to derive a formula for the optimal tableau of the primal problem in our standard dual pair. The pivot matrix $\mathbf{Q}$ in this derivation contains the slack-variable or $\mathbf{s}$ columns of the optimal tableau $\mathbf{T}^\star$. (a) If the primal problem is the one in the `dp1` dual pair, what are the numerical values of the elements in $\mathbf{Q}$? (b) Show numerically that $\mathbf{T}^\star = \mathbf{QT}$ for that problem.

5.5.10 [E] If one problem in a dual pair is unbounded, the other is infeasible. (a) Explain how the structural relationship between the problems ensures this. (b) Explain how the relationship between the objective values of the two problems ensures this.

5.5.11 [E] If one problem in a dual pair is infeasible but the other is feasible, what can we say about the optimal value of the problem that is feasible?

5.5.12 [E] If one problem in a dual pair is infeasible, is it necessarily true that the other problem is unbounded? Explain.

5.5.13 [H] As explained in §2.5, a linear program that is solved by the simplex algorithm must end in optimal form, unbounded form, infeasible form 1, or infeasible form 2. (a) Write the `brewery` problem of §1.3.1 in the form of the minimization problem in our standard dual pair, and construct its dual. Solve both problems to optimality and describe the connections between the optimal tableaus. (b) Write the `unbd` problem of §2.5.2 in the form of the minimization problem in our standard dual pair, and construct its dual. Apply the simplex algorithm to both problems and describe the connections between the final-form tableaus. (c) Modify the `infea` problem of §2.5.3 to be in only infeasible form 1. Write the resulting problem in the form of the minimization problem in our standard dual pair, and construct its dual. Apply the simplex algorithm to both problems and describe the connections between the final-form tableaus. (d) Modify the `infea` problem of §2.5.3 to be in only infeasible form 2. Write the resulting problem in the form of the minimization problem in our standard dual pair, and construct its dual. Apply the simplex algorithm to both problems and describe the connections between the final-form tableaus.

5.5.14 [H] The `unbd` problem of §2.5.2 has a feasible ray $\mathbf{r}(t) = [1, 5, 0, 0, 3]^\top + t[0, 4, 1, 0, 1]^\top$, where $t \geq 0$. (a) Draw a view of the problem from the tableau given there, in which x_3 and x_4 are nonbasic. Crosshatch the feasible set and draw an arrow to show the feasible ray. (b) Write the problem in the form of the minimization $\mathscr{P}$ in our standard dual pair, and state the numerical values of $\mathbf{c}$, $\mathbf{A}$, and $\mathbf{b}$. (c) Confirm by numerical calculation that points $\mathbf{x} = \mathbf{r}(t)$ satisfy $\mathbf{Ax} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$ for all $t \geq 0$ and are thus feasible for $\mathscr{P}$. (d) Construct the dual $\mathscr{D}$ of the primal problem $\mathscr{P}$. (e) Explain how it is possible to see by inspection of the dual constraints that $\mathscr{D}$ is infeasible. Would this still be easy if $\mathbf{A}^\top$ had many rows? (f) Show how the primal ray $\mathbf{r}(t)$ can be used to compute a linear combination of the constraint rows $\mathbf{A}^\top\mathbf{y} \leq \mathbf{c}$ and thereby make the infeasibility of $\mathscr{D}$ obvious. Would this still be easy to do if $\mathbf{A}^\top$ had many rows?

5.5.15 [E] What is a *shadow price*? How are shadow prices related to the values of dual variables? What is the shadow price of a resource that is slack at optimality?

5.5.16 [H] Use the approach of §5.1.4 to deduce the shadow price of (a) black malt; (b) hops.

5.5.17 [H] In solving the primal of the `brewery` problem we try to maximize revenue from selling products by setting their production levels x_j, while not using more of each ingredient than the amount on hand. (a) Give a similar economic interpretation for the dual of the `brewery` problem. What does its objective function represent, and what do its constraints require? (b) In view of this economic interpretation, explain how solving the dual implicitly solves the primal. (c) If $\mathbf{y}_i^\star$ is the shadow price for resource i, of what is $\mathbf{x}_j^\star$ the shadow price?

5.5.18 [E] Write down the *complementary slackness conditions* in terms of the variables in our standard dual pair.

5.5.19 [E] If $\bar{\mathbf{x}}$ is feasible for the min problem in a dual pair and $\bar{\mathbf{y}}$ is feasible for the max problem, and if together they satisfy the complementary slackness conditions, what can we say about $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$?

5.5.20 [H] If $\mathscr{P}$ is a linear program in the form of the min problem in the standard dual pair of §5.1.0 and $\mathscr{D}$ is its dual, what must be true of the tableaus representing the problems if they are both in canonical form? Explain.

5.5.21 [H] The `brewery` problem has $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$ and $\mathbf{y}^\star = [7\frac{1}{2}, 0, 18\frac{3}{4}]^\top$. Show that these vectors satisfy the complementary slackness conditions.

5.5.22 [H] At the optimal solutions to the problems in a dual pair, if a constraint in one problem is slack the corresponding variable in the other problem is zero. Is it also true that if a variable in one problem is zero the corresponding constraint in the other is slack? Explain.

5.5.23 [H] At the optimal solutions to the problems of a dual pair, if a variable in one problem is positive the corresponding constraint in the other is satisfied with equality. Is it also true that if a constraint in one problem is satisfied with equality the corresponding variable in the other is positive? Explain.

5.5.24 [E] What must be true of a primal problem $\mathscr{P}$ if at optimality it has a constraint that is satisfied with equality but the corresponding optimal variable in its dual $\mathscr{D}$ is zero?

5.5.25 [H] This problem has its minimizing point at a degenerate vertex of its feasible set.

$$\begin{array}{rlrcl} \mathscr{P}: & \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1 - x_2 & & \\ & \text{subject to} & -x_1 - x_2 & \geq & -1 \\ & & -x_2 & \geq & -1 \\ & & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

(a) Solve the problem graphically. (b) Put the problem into standard form and construct a tableau. (c) There is a tie for the minimum ratio so there are two possible pivot positions. Solve the problem by pivoting at each. (d) In each optimal tableau identify the optimal values of the dual variables, and show by a graphical argument that they are the shadow prices of the constraints. (e) Write down the dual $\mathscr{D}$ and solve it graphically. (f) Put the dual problem into standard form and construct a tableau. (g) Solve the dual, finding both of its optimal tableaus. (h) In each optimal tableau for the dual identify the optimal values of the primal variables, and show that they are the shadow prices of the dual constraints.

5.5.26 [H] If $\mathbf{P}$ is an optimal tableau for a primal problem in our standard dual pair and $\mathbf{D}$ is an optimal tableau for its dual, the slack variables $\mathbf{s}^\star$ and $\mathbf{w}^\star$ can both be found in each tableau. (a) Explain where. (b) Why does this happen?

5.5.27 [H] The `dp4` example in §5.1.6 is a dual pair in which both problems are degenerate and each has two optimal vertices. Perform degenerate pivots in the optimal tableaus to find a different optimal basis for each problem in which the slack variable cost coefficients correspond to $\mathbf{x}^{\star 1}$ and $\mathbf{y}^{\star 1}$.

5.5.28 [H] The dp3 and dp4 examples of §5.1.6 show that if one problem in a dual pair is degenerate the other can have multiple optimal vertices and if both problems are degenerate both can have multiple optimal vertices. (a) If exactly one problem is degenerate at its optimal point, can the other have a unique optimal vertex? If not, explain why; if so, devise an example. (b) If each problem is degenerate at an optimal point, can each have a unique optimal vertex? If not, explain why; if so, present an example.

5.5.29 [E] The structural relationships between the problems in a dual pair give rise to various algebraic relationships, which we studied in §5.1. (a) List all of the relationships that are boxed in that Section. (b) Give an example to illustrate each.

5.5.30 [H] Use the duality relations discussed in §5.1 to establish **Farkas' theorem**: for any $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{b} \in \mathbb{R}^m$, exactly one of these systems has a solution.

$$\begin{array}{rcl} \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad\qquad \begin{array}{rcl} \mathbf{A}^\top\mathbf{y} & \leq & \mathbf{0} \\ \mathbf{b}^\top\mathbf{y} & > & 0 \end{array}$$

Hint: the final-form tableau of what linear program would answer the question "does the left system have a solution?" Farkas' result is the most famous **theorem of the alternative**, of which many have been discovered [108, §2].

5.5.31 [H] Construct a dual of the following linear program.

$$\begin{array}{lrcl} \underset{\mathbf{b}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{a}^\top\mathbf{b} & & \\ \text{subject to} & \mathbf{C}\mathbf{b} & \leq & \mathbf{y} \\ & \mathbf{b} & \geq & \mathbf{0} \end{array}$$

5.5.32 [H] In this linear program y_1 is unconstrained in sign.

$$\begin{array}{lrcr} \underset{\mathbf{y}\in\mathbb{R}^4}{\text{maximize}} & y_1 + 2y_2 + 3y_3 + 4y_4 & & \\ \text{subject to} & y_1 + y_2 + y_3 + y_4 & \leq & 5 \\ & y_1 - y_2 & \geq & -3 \\ & -y_3 + y_4 & \leq & 6 \\ & y_2,\ y_3,\ y_4 & \geq & 0 \end{array}$$

(a) Reformulate this problem into standard form, construct an initial tableau, pivot to optimality, and from the optimal tableau read off $\mathbf{y}^\star$. (b) Form the dual, solve it, and from its optimal tableau read off $\mathbf{y}^\star$. (c) Are the two problems equally easy to solve?

5.5.33 [H] In §5.2.1 we derived this dual pair, in which the optimal tableau for $\mathscr{P}$ has no slack variable columns whose cost coefficients could be the elements of $\mathbf{y}^\star$.

$$\begin{array}{lrcl} \mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array} \qquad\qquad \begin{array}{lrcl} \mathscr{D}: \underset{\mathbf{y}\in\mathbb{R}^m}{\text{maximize}} & \mathbf{b}^\top\mathbf{y} & & \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} & \leq & \mathbf{c} \\ & \mathbf{y} & & \text{free} \end{array}$$

(a) In §5.1.0, I glibly claimed that the algebraic duality relations apply to all dual pairs because any dual pair *can* be written in the form of our standard dual pair. Explain how

this claim must be interpreted in order for it to be true in this case. Should I have worded it more precisely? (b) Which algebraic duality relations of §5.1 hold for this $\mathscr{P}$ and $\mathscr{D}$?

5.5.34 [H] The coefficients in these constraint equations have a pattern you might recognize.

$$
\begin{array}{lrcl}
\mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^6}{\text{minimize}} & 4x_{11} + 1x_{12} + 2x_{13} + 3x_{21} + 2x_{22} + 1x_{23} & & \\
\text{subject to} & x_{11} + x_{12} + x_{13} & = & 30 \\
& x_{21} + x_{22} + x_{23} & = & 10 \\
& x_{11} + x_{21} & = & 20 \\
& x_{12} + x_{22} & = & 15 \\
& x_{13} + x_{23} & = & 5 \\
& \mathbf{x} & \geq & \mathbf{0}
\end{array}
$$

(a) Construct a dual $\mathscr{D}$ for this problem. (b) Put $\mathscr{D}$ into standard form. (c) Form an initial tableau for $\mathscr{D}$ and pivot it to optimal form. (d) Can you deduce $\mathbf{x}^\star$ from the optimal tableau for $\mathscr{D}$? Explain. (e) Form an initial tableau for $\mathscr{P}$. Why is it not already in optimal form? (f) Pivot the initial tableau for $\mathscr{P}$ to optimal form. (g) Can you deduce the optimal values of the dual variables from the optimal tableau for $\mathscr{P}$? Explain.

5.5.35 [E] Putting the primal and dual problems of our standard dual pair into standard form lead in §5.2.3 to tableaus that we called `Tp` and `Td`. If their coefficient matrices $-\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ are transposes (with a sign change) and $\mathbf{A}$ is usually not square, why do these tableaus always have the same number of columns?

5.5.36 [P] The `duals.m` routine of §5.2.3 can be used to construct and solve both problems in the standard dual pair. (a) Use `duals.m` to solve the primal and dual problems of the `dp1` pair discussed in the Chapter introduction. (b) Use `duals.m` to solve the primal and dual problems of the `dp5` pair discussed in §5.3.1. (c) Deduce $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ from the dual you found in Exercise 5.5.32(b) and use `duals.m` to solve that problem and the dual the function constructs for it. Confirm that the primal and dual solutions agree with those you found in solving the Exercise. (c) Use `duals.m` to solve the transportation problem of Exercise 5.5.34 and the dual that the function constructs for it.

5.5.37 [H] In §5.2.3, I claimed that from a tableau in the form of either `Tp` or `Td` it is easy to extract `A`, `b`, and `c`. (a) Show how to obtain these arrays from `Tp` for the `brewery` problem. (b) Show how to obtain them from `Td` for that problem.

5.5.38 [E] One problem in a primal-dual pair might be easier to solve than the other. (a) Precisely what does it mean to say that one linear program is "easier to solve" than another? (b) Why might the problems in a primal-dual pair differ in their ease of solution?

5.5.39 [E] If the constraint coefficient matrix is tall and thin in one problem of a dual pair but short and fat in the other, which problem is likely to be easier to solve? Why?

5.5.40 [E] Explain in words the basic idea of the *dual simplex method.* How does a *dual simplex pivot* work?

5.5.41 [E] What makes two tableaus *dual tableaus*?

5.5.42 [H] In §5.3.2, I claimed that $\mathbf{D}_1$ and $\mathbf{P}_1$ are dual tableaus. Prove that this claim is true by (a) showing the structural relationships between the two tableaus; (b) showing that the two tableaus describe linear programs that are duals of each other.

5.5.43 [E] In §5.3.2, I wrote down dual tableaus $\mathbf{D}_0$ and $\mathbf{D}_1$ along with $\mathbf{P}_0$ and $\mathbf{P}_1$ to explain how a dual simplex pivot works. In applying the dual simplex algorithm, are the pivots performed in the primal tableau or in the dual one? Is it necessary to write down both? Explain.

5.5.44 [H] In §5.3.1 we solved the short & fat problem of the `dp5` pair by doing subproblem pivots to obtain canonical form and then a single phase-2 simplex-rule pivot to get optimal form. (a) Solve the problem by using the dual simplex algorithm instead. (b) For each pivot you perform in the short & fat problem, identify the corresponding pivot that is implicitly performed in the tall & thin problem, explicitly perform that pivot in the primal tableau, and show that at each step the resulting tableaus are duals of each other.

5.5.45 [H] If one problem in a primal-dual pair is feasible and the dual simplex algorithm is used to pivot the tableau for the *other* problem to a final form, what final forms are possible?

5.5.46 [E] Write down the steps of the dual simplex algorithm in terms of the entries T_{ij} in the primal tableau (where the pivots are performed).

5.5.47 [H] Use the dual simplex algorithm of §5.3.2 to solve this problem [3, p127-128].

	x_1	x_2	x_3	x_4	x_5	x_6
−10	0	0	3	1	2	0
−5	1	0	−1	0	−1	0
2	0	0	2	3	0	1
−7	0	1	2	−1	−1	0

5.5.48 [E] On which of the elements in the following tableau [3, Exercise 5.18c] could we perform (a) a subproblem pivot; (b) a dual-simplex pivot?

	x_1	x_2	x_3	x_4	x_5
0	0	5	3	2	0
−2	1	−1	−1	0	0
−3	0	1	−1	−1	1

(c) Solve the problem using the primal simplex algorithm. (d) Solve the problem using the dual simplex algorithm.

5.5.49 [H] Devise a dual version of the subproblem technique for getting canonical form, and illustrate how it works.

5.5.50 [E] Explain in words the basic idea of sensitivity analysis.

5.5.51 [E] Sensitivity analysis is sometimes called **postoptimality analysis** [145, §5] [151, §5] but most of the techniques described in §5.4 involve making changes to the *initial* tableau. Why is it necessary to solve a linear program before studying the sensitivity of the model to changes in its data?

5.5.52 [E] A single approach is used in §5.4.1 to study changes in resource availabilities, selling prices, and technology coefficients, singly or in combination. What is it?

5.5.53 [E] In §5.4.1, I wrote down the pivot matrix $\mathbf{P}$ by looking at the initial and optimal tableaus for the `brewery` problem. Explain how I did that.

5.5.54 [H] Suppose that a particular sequence of pivots leads from the initial tableau $\mathbf{T}_0$ of a linear program to an optimal tableau $\mathbf{T}^\star$, that $\mathbf{P}$ is a pivot matrix such that $\mathbf{P}\mathbf{T}_0 = \mathbf{T}^\star$, and that the initial tableau is then modified to $\overline{\mathbf{T}}_0$. (a) If the same sequence of pivots that solved the original problem can be performed starting from $\overline{\mathbf{T}}_0$, the result tableau is given by the matrix product $\mathbf{P}\overline{\mathbf{T}}_0$. If this result tableau is in optimal form, how do we know that it solves the modified problem? (b) If the modification of the initial tableau is such that the same sequence of pivots that solved the unmodified problem *cannot* be performed, how is the new optimal tableau $\overline{\mathbf{T}}^\star$ related to $\mathbf{P}\overline{\mathbf{T}}_0$? Present an example to illustrate your answer.

5.5.55 [H] The optimal tableau $\mathbf{T}^\star$ that I used in §5.4.1 is the one that we found with `simplex.m` in §4.1, so it results from strictly following the steps of the algorithm that we developed in §2. (a) Why are the constraint rows in this tableau permuted from those in the optimal tableau that we found by hand-pivoting in §2.4.3? (b) Does it matter which optimal tableau we use to find a pivot matrix $\mathbf{P}$ for sensitivity analysis? Explain. (c) In §5.4.1, each constraint row in $\mathbf{T}_0$ is labeled to show the resource whose consumption it constrains. In $\mathbf{T}^\star$ the slack variable for black malt, s_2, has its identity-column 1 in the third row so $s_2 = 7\frac{1}{2}$. Does it make sense to therefore think of this row as still representing the constraint on black malt? If so, which constraints are represented by the other rows, now that s_1 and s_3 are nonbasic? Explain.

5.5.56 [E] What is a *hot start* for solving a linear program?

5.5.57 [H] When it seemed likely that her IPA-making competitor might go out of business, Sarah investigated the consequences for her optimal production program of simultaneously increasing her hops on hand to 70 ounces and increasing her price for IPA to \$75 per keg. What would happen if, in addition to these changes, she also reduced the black malt in the IPA recipe from 12 pounds to 9 pounds?

5.5.58 [H] If $\mathbf{P}$ is the pivot matrix that solves the unmodified `brewery` problem, propose a change to the starting tableau $\mathbf{T}_0$ that will make $\mathbf{P}\overline{\mathbf{T}}_0$ have a negative cost coefficient.

5.5.59 [H] If in the `brewery` model the amounts of pale malt, black malt, and hops are [3, Exercise 6.8] increased simultaneously in the proportions $p : p : 2p$, how big can p get before the optimal basic sequence changes from $S = (x_1, x_2, s_2)$?

5.5.60 [H] Suppose that in the `brewery` problem the initial tableau is modified to increase the prices for Porter, Stout, Lager, and IPA by ρ, σ, λ, and α dollars respectively. Write a system of inequalities in ρ, σ, λ, and α which if it is satisfied ensures that the optimal production program $\mathbf{x}^\star$ does not change.

5.5.61 [H] The `twoexams` problem of §1.1 is a resource allocation problem. (a) Construct an initial tableau $\mathbf{T}_0$ for the problem. (b) Pivot to optimal form and call the optimal tableau $\mathbf{T}^\star$. (c) Find a pivot matrix $\mathbf{P}$ such that $\mathbf{PT}_0 = \mathbf{T}^\star$. (d) Use sensitivity analysis to find the new optimal point if the grade that triggers an advisor alert is increased to 65.

5.5.62 [H] In §5.4.2 we found that it would be profitable for Sarah to make Oktoberfest beer if she can sell it for \$80 per keg. What is the lowest price she could accept per keg if x_5 is to remain in the optimal basic sequence?

5.5.63 [H] Use sensitivity analysis to study, by removing the x_1 column from the `brewery` model, what happens to the optimal solution if Sarah decides to stop making Porter.

5.5.64 [H] In §5.4.3 we found, after appending a lower bound constraint on x_3 to $\mathbf{T}^\star$, that one pivot on $a_{4,3}$ was sufficient to restore optimal form. (a) Explain why, if a lower bound constraint on x_p is appended to $\mathbf{T}^\star$, a single pivot on $a_{m+1,p}$ restores optimal form if the appended row $m+1$ is the minimum-ratio row in the x_p column. (b) Explain why, if the appended row is *not* the minimum-ratio row in the x_p column, one or more dual simplex pivots are also required.

5.5.65 [H] Suppose that Sarah acquires an unlimited supply of pale malt, so that it is no longer necessary for her to constrain the amount she uses. Determine by sensitivity analysis how the optimal solution changes if the first constraint row is removed from the `brewery` model.

5.5.66 [H] In §5.4.4 we studied how the sale of pale malt affects Sarah's revenue from selling beer. Repeat that analysis for (a) the sale of black malt; (b) the sale of hops.

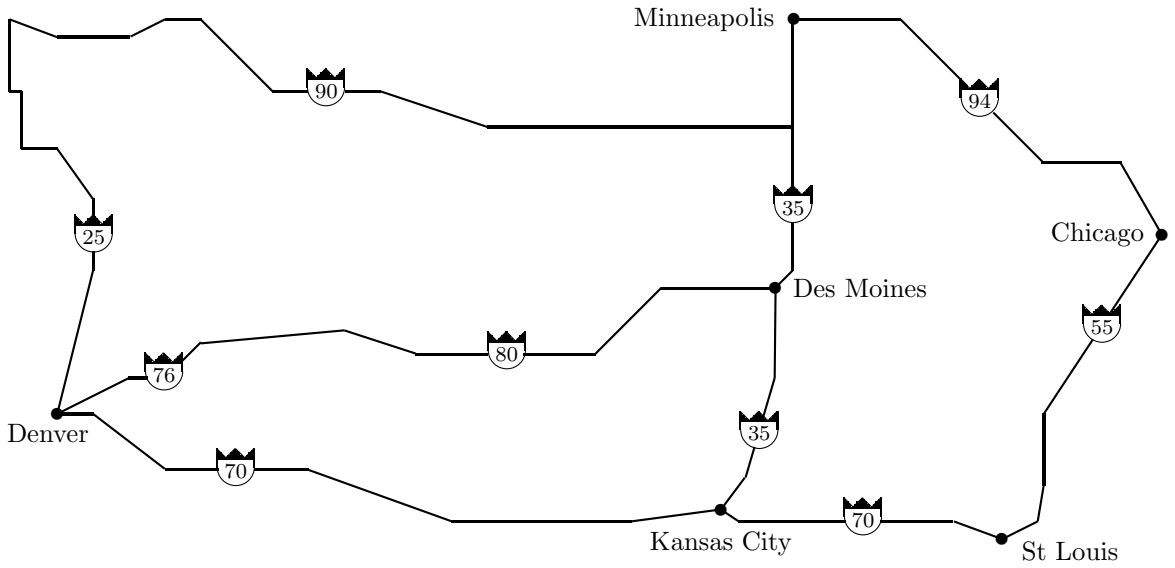
5.5.67 [H] In §5.4.4 we found the shadow price of pale malt as a function of how much Sarah sells, by gradually increasing one nonbasic variable after another and pivoting whenever the minimum row-ratio was reached. Then we could draw a curve showing the revenue realized as a function of the quantity sold. Can you suggest a more direct way of determining the breakpoints on that curve?

5.5.68 [H] Why, apart from its profound and mystical character, do you suppose anyone bothers to study linear programming duality? Now that you have read this whole Chapter, list all of the ways you can think of in which duality theory is of practical use in linear programming.

6

Linear Programming Models of Network Flow

A meat processing company with plants in Des Moines and Chicago provisions restaurant suppliers in those cities, and also rents refrigerator trucks to operate on the interstate highways shown below for shipping product to Minneapolis, Saint Louis, and Denver.



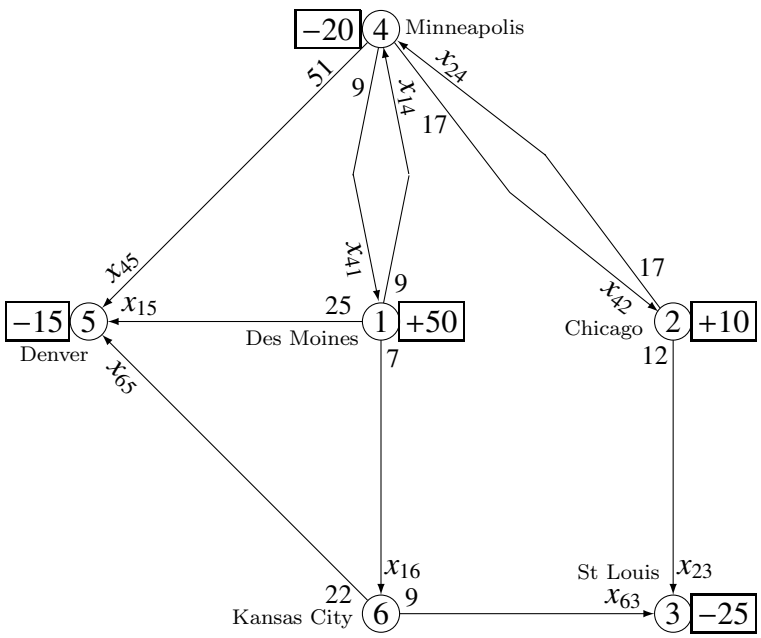
The company predicts that during the next year it will sell 20 truckloads of product to customers in Minneapolis, 25 truckloads to customers in Saint Louis, and 15 truckloads to customers in Denver. To meet these requirements it will produce 50 truckloads more than the local demand in Des Moines and 10 truckloads more than the local demand in Chicago.

Many possible routes can be used to move product from the processing plants to the out-of-town customers. For example, the demand in Saint Louis could be met with a shipment from Des Moines by sending it either through Minneapolis and Chicago or through Kansas City. This route map is very simple and drawn to scale so you might be able to guess the optimal shipping schedule, but more complicated problems are hard to solve by inspection so we will formulate an optimization model to minimize the total expense of shipping.

To operate a truck costs the company $2 per mile for fuel and rent, plus $80 per hour for the driver's salary and benefits. From the distances and driving times between the cities we can compute the cost for a truck to make each trip, as shown in the table on the next page. To keep the numbers in the model simple I have in the last column rounded off each trip cost to the nearest multiple of $100.

trip between				distance	time	cost	c_{ij}
i	city i	j	city j	[miles]	[hr:min]	[\$]	[\$ × 100]
1	Des Moines	4	Minneapolis	245	5:25	923	9
1	Des Moines	5	Denver	677	14:05	2481	25
1	Des Moines	6	Kansas City	197	4:10	727	7
2	Chicago	3	Saint Louis	324	7:15	1228	12
2	Chicago	4	Minneapolis	442	9:35	1651	17
4	Minneapolis	5	Denver	1374	29:55	5141	51
6	Kansas City	3	Saint Louis	257	5:25	947	9
6	Kansas City	5	Denver	600	12:25	2193	22

The first step in constructing our optimization model is to idealize the map on the previous page by the **network diagram** below. The circles are **nodes** corresponding to the $m = 6$ cities, each with its supply minus demand or **net stock** shown in an adjacent box. The $n = 10$ **links** connecting the nodes correspond to the highways, but each link is **directed** (even though the highways are not) because it represents shipments in just one direction. Node 6 has zero net stock but it can be used as a **transshipment point** for trucks from node 1 to pass through on their way to node 5 or node 3. Company policy also allows transshipments through nodes 1, 2, and 4, and to make that possible the diagram includes links for flow in both directions between nodes 1 and 4 and between nodes 2 and 4. Every truck that goes to node 3 or node 5 delivers its cargo rather than driving on to another city.



This network diagram summarizes not only the relevant geography but also the net stocks, per-truck shipping costs, and admissible routes.

Each **link cost** c_{ij} that we found in the table is shown in the network diagram near the *tail* of the arrow representing trips from node i to node j, and the number of trucks or **flow** on that link is represented by the variable x_{ij} shown near the *head* of the arrow.

A **shipping schedule** consists of a vector of flows $x_{ij} \geq 0$ for the links (i, j) that are in the transportation network. For the meat processor's network,

$$(i, j) \in \mathbb{N} = \{(1,4)\ (1,5)\ (1,6)\ (2,3)\ (2,4)\ (4,1)\ (4,2)\ (4,5)\ (6,3)\ (6,5)\}.$$

There are $n = |\mathbb{N}| = 10$ elements in this set so there are n link flows in the shipping schedule and n elements in the cost vector.

$$\begin{aligned} \mathbf{x} &= [x_{14}, x_{15}, x_{16}, x_{23}, x_{24}, x_{41}, x_{42}, x_{45}, x_{63}, x_{65}]^\top \\ \mathbf{c} &= [\ \ 9,\ 25,\ \ 7,\ 12,\ 17,\ \ 9,\ 17,\ 51,\ \ 9,\ 22]^\top \end{aligned}$$

Here $c_{14} = c_{41} = 9$ and $c_{24} = c_{42} = 17$, so we have assumed that those link costs do not depend on the direction of travel. With these definitions the total cost of shipments is $\mathbf{c}^\top\mathbf{x}$.

To be feasible, a shipping schedule must move the supplies to meet the demands. At node 4, for example, after trucks have arrived from nodes 1 and 2 and departed for nodes 1, 2, and 5, node 4's demand must have been met so that its net stock ends up zero.

$$\underset{\text{initial net stock}}{(-20)} + \underset{\text{trucks in}}{(x_{14} + x_{24})} - \underset{\text{trucks out}}{(x_{41} + x_{42} + x_{45})} = \underset{\text{final net stock}}{0}$$

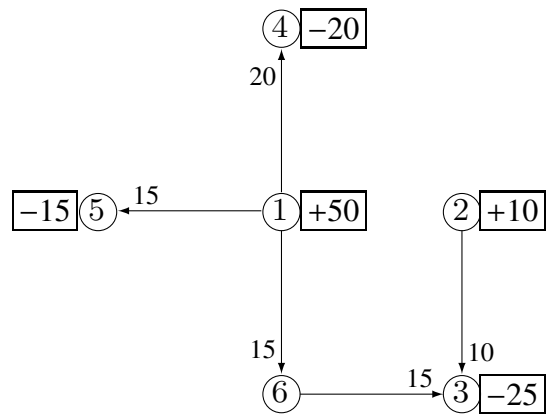
This **node equilibrium equation** expresses a conservation law like those discussed in §1.4. Because shipping product costs money, in any optimal solution it will turn out that either x_{14} or x_{41} is zero (or both) and that either x_{42} or x_{24} is zero (or both), but to allow flow in either direction between nodes 1 and 4 and between nodes 2 and 4, all four variables must be included in the model. Node 4 has a demand of 20, so its initial net stock is -20.

Minimizing the total cost of shipments subject to all $m = 6$ of the node equilibrium constraints yields this linear program, which I will call `nf1` (see §28.5.16).

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ & z(\mathbf{x}) = 9x_{14} + 25x_{15} + 7x_{16} + 12x_{23} + 17x_{24} + 9x_{41} + 17x_{42} + 51x_{45} + 9x_{63} + 22x_{65} \\ \text{subject to}\ & x_{41} - x_{14} - x_{15} - x_{16} = -50 && (1) \\ & x_{42} - x_{23} - x_{24} = -10 && (2) \\ & x_{23} + x_{63} = 25 && (3) \\ & x_{14} + x_{24} - x_{41} - x_{42} - x_{45} = 20 && (4) \\ & x_{15} + x_{45} + x_{65} = 15 && (5) \\ & x_{16} - x_{63} - x_{65} = 0 && (6) \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

The `pivot` session and optimal network diagram on the next page show that the solution of this problem is $\mathbf{x}^\star = [20, 15, 15, 10, 0, 0, 0, 0, 15, 0]^\top$.


```
unix[1] pivot
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
< read nf1.tab
Reading the tableau...
...done.

       x14 x15  x16 x23  x24  x41 x42  x45  x63 x65
   0.   9.  25.  7.  12.  17.  9.  17.  51.  9.  22.
 -50. -1.  -1. -1.   0.   0.  1.   0.   0.  0.   0.
 -10.  0.   0.  0.  -1.  -1.  0.   1.   0.  0.   0.
  25.  0.   0.  0.   1.   0.  0.   0.   0.  1.   0.
  20.  1.   0.  0.   0.   1. -1.  -1.  -1.  0.   0.
  15.  0.   1.  0.   0.   0.  0.   0.   1.  0.   1.
   0.  0.   0.  1.   0.   0.  0.   0.   0. -1.  -1.

< solve

        x14 x15 x16 x23 x24  x41  x42  x45  x63 x65
 -915.  0.  0.  0.  0.  12.  18.  22.  35.  0.  4.
   20.  1.  0.  0.  0.   1.  -1.  -1.  -1.  0.  0.
   10.  0.  0.  0.  1.   1.   0.  -1.   0.  0.  0.
   15.  0.  0.  0.  0.  -1.   0.   1.   0.  1.  0.
   15.  0.  1.  0.  0.   0.   0.   0.   1.  0.  1.
   15.  0.  0.  1.  0.  -1.   0.   1.   0.  0. -1.
    0.  0.  0.  0.  0.   0.   0.   0.   0.  0.  0.
```



In this example we formulated a **general network flow model** as a linear program and solved it using the tableau simplex method. Some problems that have little or nothing to do with trucking and highways can also be cast as general network flow models (see for example Exercises 6.6.31 and 6.6.32) and solved in the same way.

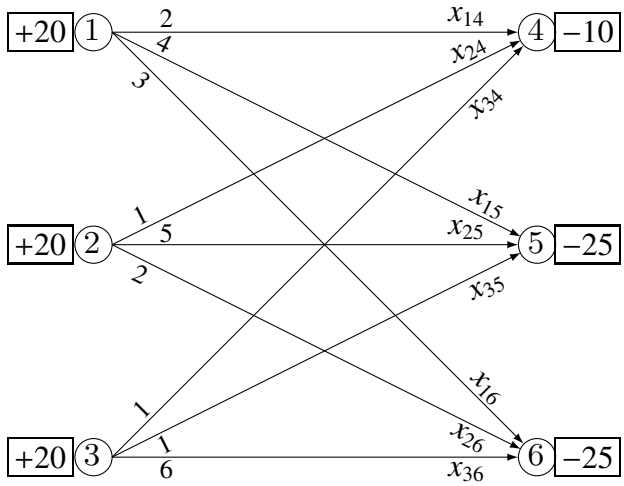
Unfortunately, the size of the simplex tableau grows very fast as the network gets bigger. If there are m nodes and flows are allowed in either direction between each node and every other node then there are $n = m(m-1)$ directed links, leading to a tableau with $(m^2 - m + 1) \times (m + 1) = m^3 + 1$ elements. Most networks are not fully connected (in our example $n = \frac{1}{3}m(m-1)$) but [151, §6.1] real problems are often too big to solve with the tableau simplex algorithm.

Fortunately, the problem has a special structure which can be exploited by a **network simplex algorithm** that requires computer memory in an amount proportional to m^2 rather than m^3. Developing such an algorithm is worthwhile for several reasons.

- It is a practical necessity if we are to find minimum-cost shipping schedules for networks of realistic size.
- The exploitation of the special structure in this problem illustrates techniques that can be used in constructing special-purpose algorithms for other problems.
- By analyzing successively more complicated problems, the development of the algorithm will offer deeper insight into all network flow models.

6.1 The Transportation Problem

We begin our development of a compact algorithm for the general network flow problem by considering its simplest instance, the **transportation problem** [3, §7.1] [151, §6.2] [79, §4.1] [107, §5.1]. In the network diagram below, the **supply nodes** $i = 1, 2, 3$ are connected only to the **demand nodes** $j = 4, 5, 6$.



Here $\mathbf{x} = [x_{14}, x_{15}, x_{16}, x_{24}, x_{25}, x_{26}, x_{34}, x_{35}, x_{36}]^\top$, $\mathbf{c} = [2, 4, 3, 1, 5, 2, 1, 1, 6]^\top$, and we want to find a shipping schedule $\mathbf{x}$ that minimizes $\mathbf{c}^\top\mathbf{x}$ subject to equilibrium constraints at the $m = 6$ nodes. Because there are no transshipments, the constraints have this simple and regular form.

node	initial stock	+	flow in	−	flow out	=	final stock
$i = 1$	+20	+	0	−	$(x_{14} + x_{15} + x_{16})$	=	0
$i = 2$	+20	+	0	−	$(x_{24} + x_{25} + x_{26})$	=	0
$i = 3$	+20	+	0	−	$(x_{34} + x_{35} + x_{36})$	=	0
$j = 4$	−10	+	$(x_{14} + x_{24} + x_{34})$	−	0	=	0
$j = 5$	−25	+	$(x_{15} + x_{25} + x_{35})$	−	0	=	0
$j = 6$	−25	+	$(x_{16} + x_{26} + x_{36})$	−	0	=	0

I multiplied the supply-node constraints through by -1 and then moved all initial stocks to the constant column in constructing the simplex tableau $\mathbf{T}_0$ on the next page.

6.1.1 Finding a Basic Feasible Solution

Tableau $\mathbf{T}_0$ has no basis, so to solve the problem we start with phase 1. In §2.8.1 we began the subproblem technique for phase 1 by pivoting-in a basis. In doing that for an arbitrary linear program it is usually not possible to select each pivot according to the minimum-ratio rule, so even if originally $\mathbf{b} \geq \mathbf{0}$ some of its entries end up negative. But in a transportation problem it always *is* possible to pick a minimum-ratio row when pivoting-in a basis.


```
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
< read nf2.tab
Reading the tableau...
...done.

       x14 x15 x16 x24 x25 x26 x34 x35 x36
   0.   2.  4.  3.  1.  5.  2.  1.  1.  6.
  20.   1.  1.  1.  0.  0.  0.  0.  0.  0.
  20.   0.  0.  0.  1.  1.  1.  0.  0.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T0
  10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
  25.   0.  1.  0.  0.  1.  0.  0.  1.  0.
  25.   0.  0.  1.  0.  0.  1.  0.  0.  1.

< p 5 2

       x14 x15 x16 x24 x25 x26 x34 x35 x36
 -20.   0.  4.  3. -1.  5.  2. -1.  1.  6.
  10.   0.  1.  1. -1.  0.  0. -1.  0.  0.
  20.   0.  0.  0.  1.  1.  1.  0.  0.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T1
  10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
  25.   0.  1.  0.  0.  1.  0.  0.  1.  0.
  25.   0.  0.  1.  0.  0.  1.  0.  0.  1.

< p 2 3

       x14 x15 x16 x24 x25 x26 x34 x35 x36
 -60.   0.  0. -1.  3.  5.  2.  3.  1.  6.
  10.   0.  1. [1.] -1.  0.  0. -1.  0.  0.
  20.   0.  0.  0.  1.  1.  1.  0.  0.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T2
  10.   1.  0.  0. [1.]  0.  0.  1.  0.  0.
  15.   0.  0. -1.  1.  1.  0.  1.  1.  0.
  25.   0.  0.  1.  0.  0.  1.  0.  0.  1.

< p 6 6

       x14 x15 x16 x24 x25 x26 x34 x35 x36
-135.   0.  0.  4. -2.  0.  2. -2. -4.  6.
  10.   0.  1.  1. -1.  0.  0. -1.  0.  0.
   5.   0.  0.  1.  0.  0.  1. -1. -1.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T3
  10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
  15.   0.  0. -1.  1.  1.  0.  1.  1.  0.
  25.   0.  0.  1.  0.  0.  1.  0.  0.  1.

< p 3 7

       x14 x15 x16 x24 x25 x26 x34 x35 x36
-145.   0.  0.  2. -2.  0.  0.  0. -2.  6.
  10.   0.  1.  1. -1.  0.  0. -1.  0.  0.
   5.   0.  0.  1.  0.  0.  1. -1. -1.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T4
  10.   1.  0.  0.  1.  0.  0. [1.]  0.  0.
  15.   0.  0. -1.  1.  1.  0.  1. [1.]  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.

< p 4 10

       x14 x15 x16 x24 x25 x26 x34 x35 x36
-265.   0.  0.  2. -2.  0.  0. -6. -8.  0.
  10.   0.  1.  1. -1.  0.  0. -1.  0.  0.
   5.   0.  0.  1.  0.  0.  1. -1. -1.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T5
  10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
  15.   0.  0. -1.  1.  1.  0.  1.  1.  0.
   0.   0.  0.  0.  0.  0.  0.  0.  0.  0.

< delete 7 0

       x14 x15 x16 x24 x25 x26 x34 x35 x36
-265.   0.  0.  2. -2.  0.  0. -6. -8.  0.
  10.   0.  1.  1. -1.  0.  0. -1.  0.  0.
   5.   0.  0.  1.  0.  0.  1. -1. -1.  0.
  20.   0.  0.  0.  0.  0.  0.  1.  1.  1.   T6
  10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
  15.   0.  0. -1.  1.  1.  0.  1.  1.  0.
```

In the $\mathbf{T}_0$ and $\mathbf{T}_1$ tableaus, I made x_{14} and x_{15} basic by pivoting in the minimum ratio rows of those columns.

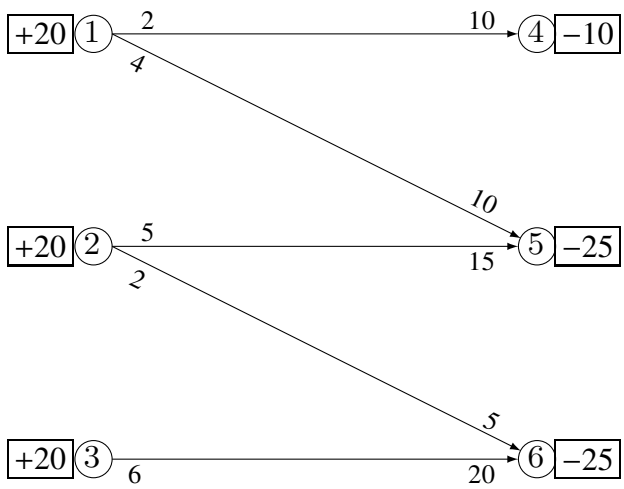
A minimum-ratio pivot on either boxed element in $\mathbf{T}_2$ would change a variable that is already basic, so I skipped those columns. The minimum-ratio pivot on the circled element in the x_{25} column of $\mathbf{T}_2$ does not change any variable that is already basic, so I made that pivot. The minimum-ratio pivot in the x_{26} column of $\mathbf{T}_3$ does not change any variable that is already basic, so I made that pivot.

A minimum-ratio pivot on either boxed element in tableau $\mathbf{T}_4$ would change a basic variable, so I pivoted in the x_{36} column instead (the bottom row is tied for the minimum ratio so I could have pivoted there.)

Tableau $\mathbf{T}_5$ is in canonical form, except for the redundant row which I deleted. This always happens, because the sum of the supplies equals the sum of the demands.

The special structure of the transportation problem guarantees [3, §7.1] that it will always be possible to perform phase 1 of the simplex algorithm in this simple way, to get canonical form in exactly $m - 1$ pivots.

In the network diagram below the costs are from $\mathbf{T}_0$, the flows are those in the basic feasible solution of $\mathbf{T}_6$, and for clarity I have omitted the nonbasic links. Remember that on the arrow representing link (i, j) the cost c_{ij} is always shown near the tail and the flow x_{ij} is always shown near the head.



A picture like this makes it easy to visualize the flows, but it takes up a lot of space and requires some drawing skill. Usually we will find it more convenient to represent the current state of a network in a **transportation tableau**. This one corresponds to the network diagram above, and it also represents the basic feasible solution in $\mathbf{T}_6$.

		demands		
		4 10	5 25	6 25
supplies	1 20	2^{10}	4^{10}	3
	2 20	1	5^{15}	2^{5}
	3 20	1	1	6^{20}

$c_{26}^{x_{26}}$

The uncircled numbers down the left side of a transportation tableau are always the supplies in node-number order and the uncircled numbers across the top are always the demands in node-number order, even if they are not labeled as such and even if the circled node numbers are not provided. The (i, j)th entry in the tableau is c_{ij}, and if that link is basic the flow x_{ij} is shown as a superscript (these numbers are *not* exponents). The flows in each row add up to the row's supply, and the flows in each column add up to the column's demand.

We found the basic feasible solution that is shown in this transportation tableau by pivoting in the simplex tableau, but it can be constructed much more easily by using the **northwest corner rule**.The steps in this procedure are illustrated for our example by the sequence of transportation tableaus on the next page, but in performing it you can annotate a single tableau by filling in the flows as you assign them.

The process begins by assigning as much flow as possible to the link, in our case $(1,4)$, whose reduced cost appears in the upper left or northwest corner of the tableau.

	10	25	25
20	2^{10}	4	3
20	1	5	2
20	1	1	6

The most we can ship on the 2 in the northwest corner is 10, because that meets the column's demand; cross off the column.

	10	25	25
20	2^{10}	4^{10}	3
20	1	5	2
20	1	1	6

The new northwest corner element is the 4. The most we can ship on it is 10, because that uses up the first row's supply; cross off the row.

	10	25	25
20	2^{10}	4^{10}	3
20	1	5^{15}	2
20	1	1	6

The new northwest corner element is the 5. The most we can ship on it is 15, because that meets the column's demand; cross off the column.

	10	25	25
20	2^{10}	4^{10}	3
20	1	5^{15}	2^{5}
20	1	1	6

The new northwest corner element is the 2. The most we can ship on it is 5, because that uses up the second row's supply; cross off the row.

	10	25	25
20	2^{10}	4^{10}	3
20	1	5^{15}	2^{5}
20	1	1	6^{20}

The new northwest corner element is the 6. Because total supply equals total demand, shipping all of the row 3 supply simultaneously uses it up and meets the column 3 demand.

	10	25	25
20	2^{10}	4^{10}	3
20	1	5^{15}	2^{5}
20	1	1	6^{20}

The resulting transportation tableau has the same initial basic feasible solution we obtained by pivoting in the simplex tableau.

6.1.2 Finding a Better Solution

In §6.1.0, to construct the initial simplex tableau $\mathbf{T}_0$ I multiplied the supply-node equilibrium constraints through by -1. This has the effect of making the formulation look like this.

$$\begin{array}{llcll}
\mathscr{P}: & \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad \displaystyle\sum_{j\in\mathbb{D}}\sum_{i\in\mathbb{S}} c_{ij}x_{ij} & = & \alpha(\mathbf{x}) & \\
& \text{subject to} \quad \displaystyle\sum_{j\in\mathbb{D}} x_{ij} & = & s_i & i\in\mathbb{S} \\
& \qquad\qquad \displaystyle\sum_{i\in\mathbb{S}} x_{ij} & = & d_j & j\in\mathbb{D} \\
& \qquad\qquad\qquad \mathbf{x} & \geq & \mathbf{0} &
\end{array}$$

In our example transportation problem, which I will call `nf2` when it is written in this form (see §28.5.17), $\mathbf{s} = [20, 20, 20]^\top$ are the supplies at nodes $i \in \mathbb{S} = \{1, 2, 3\}$ and $\mathbf{d} = [10, 25, 25]^\top$ are the demands at nodes $j \in \mathbb{D} = \{4, 5, 6\}$. Each demand is the negative of a negative net stock and hence a positive number. There are $p = |\mathbb{S}| = 3$ source nodes and $q = |\mathbb{D}| = 3$ demand nodes so there are $m = p + q = 6$ constraints. The set of links is the set product $\mathbb{S} \times \mathbb{D} = \mathbb{N} = \{(1,4)\ (1,5)\ (1,6)\ (2,4)\ (2,5)\ (2,6)\ (3,4)\ (3,5)\ (3,6)\}$, and the number of link flows x_{ij} is $n = |\mathbb{N}| = p \times q = 9$.

In §6.1.1 we found the initial basic feasible solution $\bar{\mathbf{x}} = [10, 10, 0, 0, 15, 5, 0, 0, 20]^\top$ and observed that because $\mathbf{T}_6$ has negative reduced costs this point is not optimal. Then we used the northwest corner rule to find the same assignment of flows in the transportation tableau. Does it also somehow reveal that $\bar{\mathbf{x}}$ is not optimal?

Recall from §5.1.5 that if $\bar{\mathbf{x}}$ is feasible for a linear program and $\bar{\mathbf{y}}$ is feasible for its dual, and if the objective values are equal, then $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are optimal. In §5.2.2 we found (with slight changes in notation) this dual of the transportation problem.

$$\begin{array}{llcll}
\mathscr{D}: & \underset{\mathbf{u}\in\mathbb{R}^p\ \mathbf{v}\in\mathbb{R}^q}{\text{maximize}} \quad \displaystyle\sum_{i\in\mathbb{S}} s_i u_i + \sum_{j\in\mathbb{D}} d_j v_j & = & \beta(\mathbf{u},\mathbf{v}) & \\
& \text{subject to} \quad u_i + v_j & \leq & c_{ij} & i\in\mathbb{S},\ j\in\mathbb{D} \\
& \qquad\qquad \mathbf{u},\ \mathbf{v} & & \text{free} &
\end{array}$$

If some vector $\bar{\mathbf{y}} = [\mathbf{u}^\top, \mathbf{v}^\top]^\top$ that makes the two objectives equal is also feasible for $\mathscr{D}$, then we can conclude that $\bar{\mathbf{x}}$ is optimal. The difference between the objectives is

$$\begin{aligned}
\alpha(\mathbf{x}) - \beta(\mathbf{u},\mathbf{v}) &= \left[\sum_{j\in\mathbb{D}}\sum_{i\in\mathbb{S}} c_{ij}x_{ij}\right] - \left[\sum_{i\in\mathbb{S}} s_i u_i + \sum_{j\in\mathbb{D}} d_j v_j\right] \\
&= \left[\sum_{i\in\mathbb{S}}\sum_{j\in\mathbb{D}} (c_{ij} - u_i - v_j)x_{ij} + \sum_{i\in\mathbb{S}}\sum_{j\in\mathbb{D}} u_i x_{ij} + \sum_{j\in\mathbb{D}}\sum_{i\in\mathbb{S}} v_j x_{ij}\right] - \left[\sum_{i\in\mathbb{S}} s_i u_i + \sum_{j\in\mathbb{D}} d_j v_j\right].
\end{aligned}$$

Because we assumed that $\mathbf{x}$ is feasible,

$$\sum_{j\in\mathbb{D}} x_{ij} = s_i \quad \text{and} \quad \sum_{i\in\mathbb{S}} x_{ij} = d_j$$

so

$$\sum_{i\in\mathbb{S}}\sum_{j\in\mathbb{D}} u_i x_{ij} = \sum_{i\in\mathbb{S}}\left(\sum_{j\in\mathbb{D}} x_{ij}\right)u_i = \sum_{i\in\mathbb{S}} s_i u_i \quad \text{and} \quad \sum_{j\in\mathbb{D}}\sum_{i\in\mathbb{S}} v_j x_{ij} = \sum_{j\in\mathbb{D}}\left(\sum_{i\in\mathbb{S}} x_{ij}\right)v_j = \sum_{j\in\mathbb{D}} d_j v_j.$$

Substituting in the last equation on the previous page,

$$\begin{aligned}\alpha(\mathbf{x}) - \beta(\mathbf{u},\mathbf{v}) &= \left[\sum_{i\in\mathbb{S}}\sum_{j\in\mathbb{D}}(c_{ij} - u_i - v_j)x_{ij} + \sum_{i\in\mathbb{S}} s_i u_i + \sum_{j\in\mathbb{D}} d_j v_j\right] - \left[\sum_{i\in\mathbb{S}} s_i u_i + \sum_{j\in\mathbb{D}} d_j v_j\right] \\ &= \sum_{i\in\mathbb{S}}\sum_{j\in\mathbb{D}}(c_{ij} - u_i - v_j)x_{ij} = 0.\end{aligned}$$

The difference between the objectives will be zero if each term in the final sum is zero. If x_{ij} is nonbasic then it is zero, so to make sure each term is zero we need only require that

$$c_{ij} - u_i - v_j = 0 \quad \text{for each } (i,j) \text{ where } x_{ij} \text{ is basic.}$$

To find, for a given basic feasible solution $\bar{\mathbf{x}}$, dual vectors that make $\beta(\mathbf{u},\mathbf{v}) = \alpha(\bar{\mathbf{x}})$, we need to determine the p components of $\mathbf{u}$ and the q components of $\mathbf{v}$, or $m = p + q$ numbers altogether. As we saw when we pivoted-in a basis for `nf2` in §6.1.1, there are only $m-1$ basic variables because there is always one redundant constraint, so there are $m-1$ equations in the above system and they can be satisfied by many choices of $\mathbf{u}$ and $\mathbf{v}$.

We found this initial assignment of flows. Writing the equation above for each basic **spot** (i,j) in the tableau yields the system of 5 equations in 6 unknowns on the right.

		$j=4$	$j=5$	$j=6$
		10	25	25
$i=1$	20	2^{10}	4^{10}	3
$i=2$	20	1	5^{15}	2^{5}
$i=3$	20	1	1	6^{20}

$$\begin{aligned} 2 - u_1 - v_4 &= 0 \\ 4 - u_1 - v_5 &= 0 \\ 5 - u_2 - v_5 &= 0 \\ 2 - u_2 - v_6 &= 0 \\ 6 - u_3 - v_6 &= 0 \end{aligned}$$

For small systems of linear equations having this special form it is easy to find a **chain-reaction solution** [3, p170] by hand, or even by inspection if we write each u_i to the right of tableau row i and each v_j below tableau column j. If we arbitrarily let $u_1 = 0$ then

$$u_1 = 0 \Rightarrow \begin{cases} v_4 = 2 \\ v_5 = 4 \Rightarrow u_2 = 1 \Rightarrow v_6 = 1 \Rightarrow u_3 = 5. \end{cases}$$

To solve for $\mathbf{u}$ and $\mathbf{v}$ in a computer program we can append $u_1 = 0$ and solve the resulting set of linear equations $\mathbf{My} = \mathbf{c}$.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_1 \\ u_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 5 \\ 2 \\ 6 \\ 0 \end{bmatrix}$$

```
octave:1> M=[1,0,0,1,0,0;
>           1,0,0,0,1,0;
>           0,1,0,0,1,0;
>           0,1,0,0,0,1;
>           0,0,1,0,0,1;
>           1,0,0,0,0,0];
octave:2> c=[2;4;5;2;6;0];
octave:3> y=M\c; y'
ans =

   0   1   5   2   4   1
```

Here I have used Octave's backslash operator, but code could be written to exploit the pattern of 1's and 0's in $\mathbf{M}$ (such as by rearranging its rows and columns to permit the use of banded-matrix techniques [67, §4.3]).

We have now found for `nf2` that setting $\mathbf{u} = [0, 1, 5]^\top$ and $\mathbf{v} = [2, 4, 1]^\top$ makes $\alpha(\bar{\mathbf{x}}) = \beta(\mathbf{u}, \mathbf{v})$ because $c_{ij} - u_i - v_j = 0$ for each (i, j) where x_{ij} is basic. But we can conclude that $\bar{\mathbf{x}}$ is optimal only if $\bar{\mathbf{y}} = [\mathbf{u}^\top, \mathbf{v}^\top]^\top$ is feasible for $\mathscr{D}$, and that also requires $c_{ij} - u_i - v_j \geq 0$ for each (i, j) where x_{ij} is *non*basic. To find out whether that is the case we can price out the transportation tableau by updating all of its reduced cost entries like this.

$$\begin{array}{c|ccc|c} & 10 & 25 & 25 & \mathbf{u} \\ \hline 20 & 2^{10} & 4^{10} & 3 & 0 \\ 20 & 1 & 5^{15} & 2^{5} & 1 \\ 20 & 1 & 1 & 6^{20} & 5 \\ \hline \mathbf{v} & 2 & 4 & 1 & \end{array} \xrightarrow{\quad c_{ij} \leftarrow c_{ij} - u_i - v_j \quad} \begin{array}{c|ccc|} & 10 & 25 & 25 \\ \hline 20 & 0^{10} & 0^{10} & 2 \\ 20 & -2 & 0^{15} & 0^{5} \\ 20 & -6 & -8 & 0^{20} \\ \hline \end{array}$$

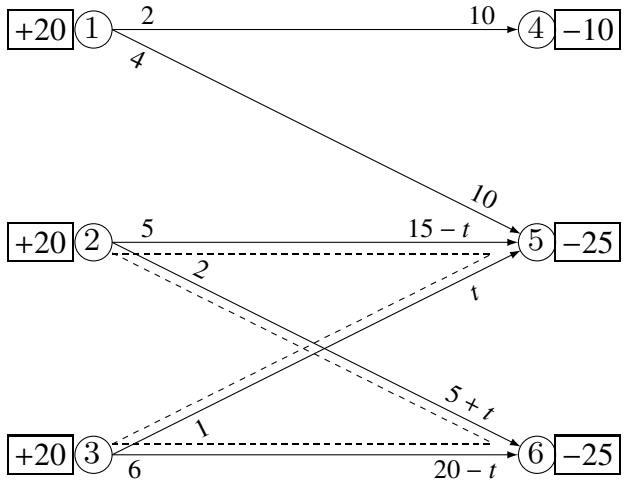
Notice in the new tableau that $c_{ij} = 0$ on the basic spots because we chose $\mathbf{u}$ and $\mathbf{v}$ to make that happen.

Unfortunately, three of the other reduced costs are negative so $\bar{\mathbf{y}}$ is not feasible for $\mathscr{D}$ and $\bar{\mathbf{x}}$ is therefore *not* optimal for $\mathscr{P}$. The reduced costs in the new transportation tableau are the same as those in this initial canonical form simplex tableau, which we found in §6.1.1 by pivoting-in a basis.

$$\mathbf{T}_6 = \begin{array}{c|ccccccccc} & x_{14} & x_{15} & x_{16} & x_{24} & x_{25} & x_{26} & x_{34} & x_{35} & x_{36} \\ \hline -265 & 0 & 0 & 2 & -2 & 0 & 0 & -6 & -8 & 0 \\ \hline 10 & 0 & 1 & 1 & -1 & 0 & 0 & -1 & 0 & 0 \\ 5 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & -1 & 0 \\ 20 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 10 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 15 & 0 & 0 & -1 & 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Our first phase-2 pivot in $\mathbf{T}_6$, on the circled element, would increase x_{35} to 15 and make x_{25} nonbasic. Might we somehow perform this pivot in the transportation tableau?

Increasing x_{35} in the simplex tableau from 0 to t has the effect of introducing that much flow from node 3 to node 5 in the network.



The supply at node 3 is still 20, but instead of shipping all of it to node 6 we can now ship only $20 - t$. To keep the sum of the flows into node 6 equal to its demand, we must increase x_{26} from 5 to $5 + t$. But node 2 can ship only 20, so x_{25} must decrease from 15 to $15 - t$. Together these changes amount to **shifting** t units of flow around the loop that is shown dashed, alternately increasing and decreasing the flow on those links. The new flows still use up the supplies and satisfy the demands, but now the total cost is

$$\begin{aligned}
\alpha(\mathbf{x}) &= 2x_{14} + 4x_{15} + 5x_{25} + 2x_{26} + 1x_{35} + 6x_{36} \\
&= 2(10) + 4(10) + 5(15 - t) + 2(5 + t) + 1(t) + 6(20 - t) \\
&= 265 - 8t.
\end{aligned}$$

Each unit of flow we put on link $(3, 5)$ changes $\alpha(\mathbf{x})$ by -8, the reduced cost that is in the x_{35} column of $\mathbf{T}_6$ and in the $(3, 5)$ spot of the priced-out transportation tableau.

To minimize $\alpha(\mathbf{x})$ we would like to make t as high as possible, but the flows must remain nonnegative so

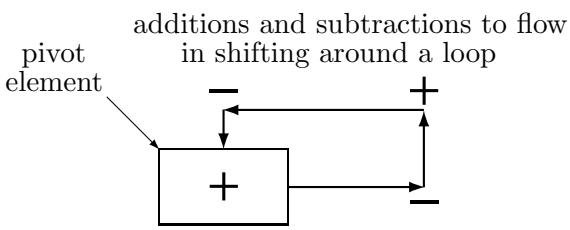
$$\left.\begin{array}{rcrcl}
x_{35} & = & t & \geq & 0 \\
x_{36} & = & 20 - t & \geq & 0 \\
x_{26} & = & 5 + t & \geq & 0 \\
x_{25} & = & 15 - t & \geq & 0
\end{array}\right\} \Rightarrow t \leq 15,$$

which is the minimum ratio in the x_{35} column of $\mathbf{T}_6$. Shifting that amount of flow around the loop makes x_{35} basic and x_{25} nonbasic while adjusting x_{26} and x_{36} to maintain feasibility, and it corresponds exactly to performing the circled pivot in $\mathbf{T}_6$. It also corresponds to shifting flow around a loop in the transportation tableau, as shown on the next page.

A **loop** in a transportation tableau is an even number of 4 or more spots connected by lines that are alternately horizontal and vertical. To perform a simplex-rule pivot, exactly one of the spots should be nonbasic with a negative reduced cost and the others basic and thus with zero reduced cost. The loop that includes a given nonbasic spot is unique [3, p173].

$$
\begin{array}{cc|ccc|}
 & & j=4 & j=5 & j=6 \\
 & & 10 & 25 & 25 \\
\hline
i=1 & 20 & 0^{10} & 0^{10} & 2 \\
i=2 & 20 & -2 & 0^{15} & 0^{5} \\
i=3 & 20 & -6 & \boxed{-8} & 0^{20} \\
\hline
\end{array}
\quad \xrightarrow{\text{shift } t=15} \quad
\begin{array}{cc|ccc|}
 & & 10 & 25 & 25 \\
\hline
20 & & 0^{10} & 0^{10} & 2 \\
20 & & -2 & 0 & 0^{20} \\
20 & & -6 & -8^{15} & 0^{5} \\
\hline
\end{array}
$$

On the left above we can shift flow in either direction around the loop, alternately adding and subtracting the amount of the shift to maintain feasibility. Going counterclockwise, we increase the flow on the -8 in the $(3,5)$ spot by $t = 15$, decrease the flow in the $(3,6)$ spot by 15, increase the flow in the $(2,6)$ spot by 15, and decrease the flow in the $(2,5)$ spot by 15, yielding the tableau on the right in which x_{35} has become basic and x_{25} nonbasic. The amount of the shift is the smallest of the flows from which we *subtract* in doing the shift.



$$t = \min\{x_{36}, x_{25}\} = \min\{20, 15\} = 15$$

As in the network diagram, the shift amount t is the minimum ratio in the x_{35} column of $\mathbf{T}_6$.

Below I priced out the shifted tableau above, found that it is not optimal, constructed another loop, performed another shift, and priced out that result to obtain optimal form. From the optimal tableau, $\mathbf{x}^\star = [10, 5, 5, 0, 0, 20, 0, 20, 0]^\top$ which yields $\alpha(\mathbf{x}^\star) = 115$.

$$
\begin{array}{c|ccc|c}
 & 10 & 25 & 25 & \mathbf{u} \\
\hline
20 & 0^{10} & 0^{10} & 2 & 0 \\
20 & -2 & 0 & 0^{20} & -8 \\
20 & -6 & -8^{15} & 0^{5} & -8 \\
\hline
\mathbf{v} & 0 & 0 & 8 &
\end{array}
\quad \xrightarrow{\text{price}} \quad
\begin{array}{c|ccc|}
 & 10 & 25 & 25 \\
\hline
20 & 0^{10} & 0^{10} & -6 \\
20 & 6 & 8 & 0^{20} \\
20 & 2 & 0^{15} & 0^{5} \\
\hline
\end{array}
\quad \xrightarrow{\text{shift } 5}
$$

$$
\begin{array}{c|ccc|c}
 & 10 & 25 & 25 & \mathbf{u} \\
\hline
20 & 0^{10} & 0^{5} & -6^{5} & 0 \\
20 & 6 & 8 & 0^{20} & 6 \\
20 & 2 & 0^{20} & 0 & 0 \\
\hline
\mathbf{v} & 0 & 0 & -6 &
\end{array}
\quad \xrightarrow{\text{price}} \quad
\begin{array}{c|ccc|}
 & 10 & 25 & 25 \\
\hline
20 & 0^{10} & 0^{5} & 0^{5} \\
20 & 0 & 2 & 0^{20} \\
20 & 2 & 0^{20} & 6 \\
\hline
\end{array}
$$

6.1.3 Degeneracy

In the new transportation tableau below I made a feasible initial assignment of flows by starting in the northwest corner as we did in §6.1.1.

		$j=4$	$j=5$	$j=6$
		10	5	20
$i=1$	10	9^{10}	3	1
$i=2$	15	2	3^{5}	7^{10}
$i=3$	10	3	1	1^{10}

$$\begin{aligned} 9-u_1-v_4 &= 0 \\ 3-u_2-v_5 &= 0 \\ 7-u_2-v_6 &= 0 \\ 1-u_3-v_6 &= 0 \end{aligned}$$

To price out this tableau we need to find $\mathbf{u}$ and $\mathbf{v}$, so on the right I wrote the equations $c_{ij}-u_i-v_j=0$ for the spots having positive flow. Letting $u_1=0$ as usual, I attempted a chain-reaction solution,

$$u_1 = 0 \Rightarrow v_4 = 9,$$

but this is as far as it gets because none of the other equations involves u_1 or v_4. What has gone wrong is that there are 6 unknowns but with $u_1=0$ only 5 equations. That is because the assignment of flows made only 4 variables basic while we need $m-1=5$. The reason there are too few basic spots in the tableau is that making the first assignment of 10 units on link $(1,4)$ simultaneously used up the supply at node 1 and satisfied the demand at node 4, something that normally happens only when making the *final* assignment.

To study the phenomenon in more detail I constructed an initial simplex tableau $\mathbf{D}_0$ for this problem, which I will call `nf3` (see §28.5.18), and did these minimum-ratio pivots.

```
< read nf3.tab

       x14 x15 x16 x24 x25 x26 x34 x35 x36
   0.   9.  3.  1.  2.  3.  7.  3.  1.  1.
  10.   1.  1.  1.  0.  0.  0.  0.  0.  0.
  15.   0.  0.  0.  1.  1.  1.  0.  0.  0.
  10.   0.  0.  0.  0.  0.  0.  1.  1.  1.   D0
  10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
   5.   0.  1.  0.  0.  1.  0.  0.  1.  0.
  20.   0.  0.  1.  0.  0.  1.  0.  0.  1.

< p 2 2;
< p 6 6;
< p 3 7;
< p 4 10

        x14 x15 x16 x24 x25 x26 x34 x35 x36
 -185.   0. -2. -8. -5.  0.  0.  2.  4.  0.
   10.   1.  1.  1.  0.  0.  0.  0.  0.  0.
   10.   0. -1.  0.  1.  0.  1.  0. -1.  0.
   10.   0.  0.  0.  0.  0.  0.  1.  1.  1.   D4
    0.   0. -1. -1.  1.  0.  0.  1.  0.  0.
    5.   0.  1.  0.  0.  1.  0.  0.  1.  0.
    0.   0.  1.  1. -1.  0.  0. -1.  0.  0.
```

```
< p 7 3

        x14 x15 x16 x24 x25 x26 x34 x35 x36
 -185.   0.  0. -6. -7.  0.  0.  0.  4.  0.
   10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
   10.   0.  0.  1.  0.  0.  1. -1. -1.  0.
   10.   0.  0.  0.  0.  0.  0.  1.  1.  1.   D5
    0.   0.  0.  0.  0.  0.  0.  0.  0.  0.
    5.   0.  0. -1.  1.  1.  0.  1.  1.  0.
    0.   0.  1.  1. -1.  0.  0. -1.  0.  0.

< delete 5 0

        x14 x15 x16 x24 x25 x26 x34 x35 x36
 -185.   0.  0. -6. -7.  0.  0.  0.  4.  0.
   10.   1.  0.  0.  1.  0.  0.  1.  0.  0.
   10.   0.  0.  1.  0.  0.  1. -1. -1.  0.   D6
   10.   0.  0.  0.  0.  0.  0.  1.  1.  1.
    5.   0.  0. -1.  1.  1.  0.  1.  1.  0.
    0.   0.  1.  1. -1.  0.  0. -1.  0.  0.
```


Tableau $\mathbf{D}_4$ has $\bar{\mathbf{x}} = [10, 0, 0, 0, 5, 10, 0, 0, 10]^\top$, the same assignment of flows as in the transportation tableau, with the incomplete basic sequence $S = (x_{14}, x_{26}, x_{36}, x_{25}, \square)$. To complete a basis one more pivot is required, and to be minimum-ratio that pivot must be in a row having $b_i = 0$; I chose the alternative that makes x_{15} basic at 0. Tableau $\mathbf{D}_5$ still represents $\bar{\mathbf{x}}$, but now that point is a basic feasible solution with $S = (x_{14}, x_{26}, x_{36}, x_{25}, x_{15})$. Row 5 is redundant because supply equals demand, so I deleted it. In the final tableau $b_5 = 0$ so the problem is degenerate, and for the basis in this tableau to be complete one of its basic variables must be zero.

Making the variable x_{15} basic at zero by performing a minimum-ratio pivot in the simplex tableau is equivalent to assigning a flow of zero on the $(1, 5)$ spot in the transportation tableau. If we do that we get the additional equation $c_{15} - u_1 - v_5 = 3 - u_1 - v_5 = 0$, and the chain-reaction solution that failed before succeeds like this.

$$u_1 = 0 \Rightarrow \begin{cases} v_4 = 9 \\ v_5 = 3 \Rightarrow u_2 = 0 \Rightarrow v_6 = 7 \Rightarrow u_3 = -6. \end{cases}$$

The northwest corner rule, which we can now state precisely, introduces a basic flow of zero when degeneracy is discovered, so that $m - 1$ flows are always made basic.

the northwest corner rule
initialize the row index $i \leftarrow 1$
initialize the column index $j \leftarrow p + 1$
`1` ship as much as possible on link (i, j)
if the row i supply is used up *and* the column j demand is met
  but this is not the *final* assignment
  ship 0 on link $(i, j + 1)$ and consider $x_{i,j+1} = 0$ basic
if the row i supply is used up
  cross off row i and let $i \leftarrow i + 1$
if the column j demand is satisfied
  cross off column j and let $j \leftarrow j + 1$
if any row or column is not yet crossed off, `GO TO 1`

This rule assigns one shipment for each of the p supplies that are used up and one shipment for each of the q demands that are met, except for the last assignment which does both. Recall that there are only $p + q - 1 = m - 1$ basic variables to assign, because the equality of supply and demand always makes one constraint redundant. In §6.1.5 we will consider some other methods of finding an initial feasible assignment of flows, and each of them will deal with degeneracy in a way similar to that used here.

Degeneracy also manifests itself in phase 2 of the transportation simplex algorithm, as we shall see in the next Section, and there it can result in cycling. Refinements to prevent cycling are possible as in the tableau simplex algorithm, but they are beyond the scope of this introduction.

6.1.4 The Transportation Simplex Algorithm

The solution process [3, §7.1-7.4] [79, §4.2] that we developed in §6.1.1, §6.1.2, and §6.1.3 is summarized in the following formal statement of the transportation simplex algorithm. The complications that arise from degeneracy are explained in [square brackets].

0. **initialize**
   - Construct a transportation tableau for the problem, using the original per-unit shipping costs.
   - Find an initial basic feasible solution $\mathbf{x}$, by using the northwest corner rule or another start method. [If the basis is degenerate, some basic x_{ij} will be zero.]
1. **find dual vectors that make $\alpha(\mathbf{x}) = \beta(\mathbf{u}, \mathbf{v})$**
   - Identify the tableau spots that are basic [including any that are basic with $x_{ij} = 0$]. If there are p source rows in the transportation tableau and q destination columns, there should be $p + q - 1$ basic variables.
   - Using the current cost coefficients c_{ij}, find $\mathbf{u}$ and $\mathbf{v}$ such that
     $$u_i + v_j = c_{ij} \qquad \text{for every } (i, j) \text{ where } x_{ij} \text{ is basic.}$$
2. **test for dual feasibility**
   - Replace the per-unit shipping costs by the reduced costs
     $$c_{ij} \leftarrow c_{ij} - u_i - v_j \qquad \text{for every } (i, j).$$
     The reduced cost on each basic spot should come out zero.
   - If each reduced cost is nonnegative, the current $\mathbf{x}$ is optimal; `STOP`.
3. **update the flows**
   - Find a loop.
     Pick a spot having a negative reduced cost and find the unique loop starting at that spot with all other spots in the loop being basic [some might have zero flow].
   - Shift as much assigned flow as possible around the loop.
     The amount to shift is the minimum flow assigned to any spot in the loop from which you must subtract in performing the shift [and might be zero].
   - Update the transportation tableau with the new flows.
     If the shift makes one x_{ij} in the loop zero, it becomes nonbasic. [If more than one x_{ij} in the loop becomes zero, pick one arbitrarily to be nonbasic and mark the others basic with zero flow.]
4. **continue**
   `GO TO` 1.

To illustrate the algorithm we will use it to solve the degenerate problem `nf3`.

	10	5	20
10	$9^{x_{14}}$	$3^{x_{15}}$	$1^{x_{16}}$
15	$2^{x_{24}}$	$3^{x_{25}}$	$7^{x_{26}}$
10	$3^{x_{34}}$	$1^{x_{35}}$	$1^{x_{36}}$

northwest

	10	5	20	**u**
10	9^{10}	3^{0}	1	0
15	2	3^{5}	7^{10}	0
10	3	1	1^{10}	−6
v	9	3	7	

price

The initial tableau includes the x_{ij} only as a reminder of which one goes where, so that I can refer to them in this description; in performing the algorithm it is not necessary to remember the names of the variables.

The northwest corner rule produces an initial assignment of flows in which $x_{15} = 0$, to make $p + q - 1 = 3 + 3 - 1 = 5$ variables basic. In performing the algorithm by hand it is prudent to check after each shift that the tableau still has the right number of basic spots and that $\mathbf{x}$ is still feasible.

I found the dual vectors at each pricing step by inspection, choosing $u_1 = 0$ and following the chain reaction. That each shift is of 5 units is only a coincidence. Because the problem is degenerate the final shift makes both x_{15} and x_{26} zero, so I arbitrarily chose x_{26} to be the one that is basic.

	10	5	20
10	0^{10}	0^{0}	−6
15	$\boxed{-7}$	0^{5}	0^{10}
10	0	4	0^{10}

shift 5 →

	10	5	20	**u**
10	0^{5}	0^{5}	−6	0
15	-7^{5}	0	0^{10}	−7
10	0	4	0^{10}	−7
v	0	0	7	

price

	10	5	20
10	0^{5}	0^{5}	$\boxed{-13}$
15	0^{5}	7	0^{10}
10	7	11	0^{10}

shift 5 →

	10	5	20	**u**
10	0	0^{5}	-13^{5}	0
15	0^{10}	7	0^{5}	13
10	7	11	0^{10}	13
v	−13	0	−13	

price

	10	5	20
10	13	0^{5}	0^{5}
15	0^{10}	$\boxed{-6}$	0^{5}
10	7	−2	0^{10}

shift 5 →

	10	5	20	**u**
10	13	0	0^{10}	0
15	0^{10}	-6^{5}	0^{0}	0
10	7	−2	0^{10}	0
v	0	−6	0	

price

	10	5	20
10	13	6	0^{10}
15	0^{10}	0^{5}	0^{0}
10	7	4	0^{10}

optimal form
$\mathbf{x}^\star = [0, 0, 10, 10, 5, 0, 0, 0, 10]^\top$

6.1.5 Other Starting Methods

The northwest corner rule is easy to apply, but it usually produces an initial basic feasible solution that is far from optimal. The three methods described on the next page pay attention to the link costs, so they often find better starting points. The tableaus below show initial flow assignments for the `nf3` problem using all of the rules.

	10	5	20
10	9^{10}	3^{0}	1
15	2	3^{5}	7^{10}
10	3	1	1^{10}

northwest corner
$\alpha(\mathbf{x}^0) = 185$

	10	5	20
10	9	3	1^{10}
15	2^{10}	3	7^{5}
10	3	1^{5}	1^{5}

smallest cost
$\alpha(\mathbf{x}^0) = 75$

	10	5	20
10	9	3	1^{10}
15	2^{10}	3^{5}	7^{0}
10	3	1	1^{10}

Vogel and Russell
$\alpha(\mathbf{x}^\star) = 55$

In this case Vogel's rule and Russell's rule both yield the optimal tableau. When the rules are ranked by the quality of the starting point $\mathbf{x}^0$ that they typically produce, as measured by $\alpha(\mathbf{x}^0)$, Russell's rule > Vogel's rule > the smallest-cost rule > the northwest corner rule. Unfortunately, this is also their ranking by the amount of work they require.

Russell's rule is the most laborious because of the pricing calculation that it uses to find the Δ_{ij}. Applying it to `nf3`, we find $\bar{\mathbf{u}}$, $\bar{\mathbf{v}}$, and then $\mathbf{\Delta}_1 = c_{ij} - \bar{u}_i - \bar{v}_j$. The most negative element of $\mathbf{\Delta}_1$ is the -15, so our first assignment of flow is on the upper right spot in the tableau. This uses up the first supply, so I crossed off that row. Now we find new vectors $\bar{\mathbf{u}}$ and $\bar{\mathbf{v}}$, and $\mathbf{\Delta}_2$. Its most negative element is the -9 so we assign flow on the lower right spot in the tableau. This simultaneously uses up the third supply and satisfies the third demand, so we must assign a flow of 0 on the spot corresponding to the next most negative element in that row or column of $\mathbf{\Delta}_2$, which is the -7. I assigned the remaining supply to the elements in the one remaining row of the tableau.

	10	5	20	$\bar{\mathbf{u}}$
10	9	3	1	9
15	2	3	7	7
10	3	1	1	3
$\bar{\mathbf{v}}$	9	3	7	

$$\mathbf{\Delta}_1 = \begin{bmatrix} -9 & -9 & -15 \\ -14 & -7 & -7 \\ -9 & -5 & -9 \end{bmatrix}$$

	10	5	20	$\bar{\mathbf{u}}$
~~10~~	~~9~~	~~3~~	~~1^{10}~~	
15	2	3	7	7
10	3	1	1	3
$\bar{\mathbf{v}}$	3	3	7	

$$\mathbf{\Delta}_2 = \begin{bmatrix} & & \\ -8 & -7 & -7 \\ -3 & -5 & -9 \end{bmatrix}$$

	10	5	20
~~10~~	~~9~~	~~3~~	~~1^{10}~~
15	2	3	7^{0}
~~10~~	~~3~~	~~1~~	~~1^{10}~~

$\longrightarrow$

	10	5	20
10	9	3	1^{10}
15	2^{10}	3^{5}	7^{0}
10	3	1	1^{10}

In using each rule, when there is a tie between links where flow can be assigned the choice can be made at random (but see [133, p62-69]). The more work we do in phase 1 to find a good starting point the less work we need to do in phase 2 simplex iterations, and the best tradeoff [3, p181] is usually to get the best possible $\mathbf{x}^0$.

smallest-cost rule [3, p178-179]

```
1 ship as much as possible on a link (i, j) having the smallest cost
  if the row i supply is used up and the column j demand is met
     but this is not the final assignment
     ship 0 on a link in column j having the next smallest cost
  if the row i supply is used up cross off row i
  if the column j demand is satisfied cross off column j
  if any row or column is not yet crossed off, GO TO 1
```

Vogel's rule [3, p180-181] [79, p134-137] [133, §4]

```
1 if only one row or one column remains under consideration no choice remains
     assign flows in that row or column to use up the supplies and satisfy the demands
     EXIT with an initial basic feasible assignment of flows
  for each row and column
     find the difference between the two smallest remaining cost entries
  pick a row or column having the largest difference
  ship as much as possible on a link having the smallest cost
     in a row or column that had the largest difference
  if this simultaneously uses up a supply and meets a demand
     but it is not the final assignment
     ship 0 on a link having the next smallest cost
     in the row or column where the nonzero assignment was made
  if a supply is used up cross off that row
  if a demand is satisfied cross off that column
  GO TO 1
```

Russell's rule [79, p137-138] [138]

```
1 if only one row or one column remains under consideration no choice remains
     assign flows in that row or column to use up the supplies and satisfy the demands
     EXIT with an initial basic feasible assignment of flows
  for each row i find ū_i, the maximum cost entry among columns still under consideration
  for each column j find v̄_j, the maximum cost entry among rows still under consideration
  for each link (i, j) not yet assigned a flow, compute Δ_ij = c_ij − ū_i − v̄_j
  ship as much as possible on a link having the most negative Δ_ij
  if this simultaneously uses up a supply and meets a demand
     but it is not the final assignment
     ship 0 on a link having the next most negative Δ_ij
     in the row or column where the nonzero assignment was made
  if a supply is used up cross off that row
  if a demand is satisfied cross off that column
  GO TO 1
```


6.1.6 Multiple Optimal Solutions

In an optimal-form simplex tableau a nonbasic column whose cost coefficient is zero indicates (see §3.4) the presence of an alternate optimum, which can be found if the optimal set is bounded by performing a minimum-ratio pivot in that column. In an optimal-form transportation tableau alternate optima are indicated by nonbasic spots with zero reduced cost, and they can be found by shifting flow onto those spots.

The tableau on the left represents the optimal basic feasible solution $\mathbf{x}^{\star 1}$, but the problem is alleged [3, p186-187] to have three other distinct optimal basic feasible solutions. To reveal $\mathbf{x}^{\star 2}$ we can choose a nonbasic spot with a zero reduced cost, find the unique loop containing it and other spots that are all basic, and shift as much flow as possible around the loop.

$$
\begin{array}{c|cccc|}
 & 15 & 30 & 15 & 30 \\
\hline
20 & 0 & 2 & 3 & 0^{20} \\
15 & 6 & 0^{15} & 1 & \boxed{0} \\
10 & 0^{10} & 0^{0} & 9 & 7 \\
15 & 3 & 0^{15} & 4 & 2 \\
30 & 0^{5} & 5 & 0^{15} & 0^{10} \\
\hline
\end{array}
\quad \xrightarrow{\text{shift } t = 10} \quad
\begin{array}{c|cccc|}
 & 15 & 30 & 15 & 30 \\
\hline
20 & 0 & 2 & 3 & 0^{20} \\
15 & 6 & 0^{5} & 1 & 0^{10} \\
10 & 0 & 0^{10} & 9 & 7 \\
15 & 3 & 0^{15} & 4 & 2 \\
30 & 0^{15} & 5 & 0^{15} & 0^{0} \\
\hline
\end{array}
$$

$$\mathbf{x}^{\star 1} = [0,0,0,20,0,15,0,\ \ 0,10,\ \ 0,0,0,0,15,0,0,\ \ 5,0,15,10]^{\mathsf{T}}$$

$$\mathbf{x}^{\star 2} = [0,0,0,20,0,\ \ 5,0,10,\ \ 0,10,0,0,0,15,0,0,15,0,15,\ \ 0]^{\mathsf{T}}$$

The maximum flow t that we can shift around this loop is the smallest of the flows assigned to the spots from which we must subtract,

$$t = \min\{15, 10, 10\} = 10.$$

Shifting this amount yields the optimal-form tableau on the right. Two of the assigned flows in the loop are simultaneously reduced to zero by the shift, so to keep $p+q-1 = 5+4-1 = 8$ variables basic I arbitrarily chose the link in the lower right corner of the new tableau to be basic with a flow of zero.

6.2 Unequal Supply and Demand

Our formulation of the transportation problem in §6.1 assumes that total supply is equal to total demand,

$$\sum_{i \in \mathbb{S}} s_i = \sum_{j \in \mathbb{D}} d_j,$$

but there are practical situations in which that is not true. For example, a hardware manufacturer might intentionally keep more bolts in stock than it expects to ship so that it can respond promptly to customer demands. Can our linear programming model still somehow be used to find an optimal shipping schedule for meeting those demands?

6.2.1 More Supply Than Demand

Suppose that in our `nf2` example the supply nodes are factories, the demand nodes are hardware stores, and we are shipping boxes of bolts. We found above that $\alpha(\mathbf{x}^\star) = 115$ for this problem. If each factory produces 5 boxes more, so that supply exceeds demand by a total of 15 boxes, we could modify the formulation as shown on the left below. Each supply is increased by 5 boxes, and a new node 7 is included with a demand for the 15 extra boxes. Unlike the other nodes in the model, this fictitious demand point does not correspond to a physical location because it represents the unsold inventories x_{17}, x_{27}, and x_{37} that are held at the three factories.

factories	stores: 4 (10)	5 (25)	6 (25)	unshipped: 7 (15)
1 (20+5)	2	4	3	0
2 (20+5)	1	5	2	0
3 (20+5)	1	1	6	0

— solve →

	10	25	25	15
25	0^{10}	2	0	0^{15}
25	0^{0}	4	0^{25}	1
25	0^{0}	0^{25}	4	1

In this new, larger problem total supply again equals total demand. Assuming that it costs nothing to leave unsold bolts where they are, the cost to ship from each factory into its own inventory is zero.

The optimal tableau on the right shows that increasing the supplies at nodes 2 and 3 changed the other flows so that all 15 units of excess supply, comprising the 5 units of extra production at factory 1 and 10 units of its original production, are retained in inventory there. Now $\alpha(\mathbf{x}^\star) = 95$, and the entry $c_{16} = 0$ indicates an alternate optimum that was not present in the original problem. That problem was not degenerate but this one is, with basic variables $x_{24} = 0$ and $x_{34} = 0$.

6.2.2 Less Supply Than Demand

If we have too little supply, no reformulation of any mathematical model will let us satisfy the demand. We can, however, modify the transportation problem to find the least expensive way of shipping the inadequate supplies we do have. Suppose that in our `nf2` example each factory now produces 5 boxes fewer, so that demand exceeds supply by a total of 15 boxes. To make up this deficit we can, as shown on the left at the top of the next page, include a fictitious supply of 15 boxes that can be shipped at zero cost to each of the stores. Throwing away the fictitious-source row of the optimal tableau, on the right at the top of the next page, leaves a shipping schedule for the supplies that we have. If there were some way of altering this schedule to reduce its cost, then we could adjust the flows in the fictitious-source row to make the enlarged problem feasible and it would also have the lower cost. But we already found a lowest-cost solution to the enlarged problem, so it must be that the part of the tableau above the dashed line is optimal for the original problem.

		stores						
		④ 10	⑤ 25	⑥ 25			10	25	25
factories	① 20 − 5	2	4	3		15	0^{10}	1	0^{5}
	② 20 − 5	1	5	2		15	0	3	0^{15}
	③ 20 − 5	1	1	6	solve →	15	1	0^{15}	5
fictitious source	⑦ 15	0	0	0		15	1	0^{10}	0^{5}

The optimal tableau on the right does not reveal degeneracy, but $c_{24} = 0$ so again there is an alternate optimum that was not present in the original problem.

6.2.3 "At Least This Much" Demands

The bolt manufacturer of §6.2.1 had 25 boxes at each of its three factories when a change in corporate tax law suddenly made it undesirable to keep unsold inventory. Fortunately, the store at node 5 has agreed to accept more bolts than its minimum demand of 25 boxes. How can the total supply be shipped at least cost?

Now instead of shipping the excess supply into inventory at the three factories, which costs nothing, it must be shipped to node 5 at those per-unit costs, like this.

		stores			extra to node 5					
		④ 10	⑤ 25	⑥ 25	⑦ 15		10	25	25	15
factories	① 25	2	4	3	4	25	0	0^{15}	0^{10}	0
	② 25	1	5	2	5	25	0^{10}	2	0^{15}	2
	③ 25	1	1	6	1	25	2	0^{10}	6	0^{15}

In this scenario the excess supply all comes from factory 3, and the store at node 5 receives those 15 boxes in addition to its minimum demand of 25.

If the store at node 4 also agrees to accept more than its minimum demand, the formulation must allow for the excess supply to go to either node 4 or node 5. Here node 7 will absorb any extra shipments for node 4 and node 8 will absorb those for node 5.

		stores			extra shipments	
		④ 10	⑤ 25	⑥ 25	⑦ 15	⑧ 15
factories	① 25	2	4	3	2	4
	② 25	1	5	2	1	5
	③ 25	1	1	6	1	1
fictitious source	⑨ 15				0	0

Now the indicated demand is 90, so to make the indicated supply equal to that number we must add a fictitious source with a supply of 15 boxes. This supply will make up the difference between the 15 boxes of excess supply at the factories and the 30 boxes now demanded at nodes 7 and 8, so it must ship only to those nodes and with zero cost. To show that node 9 cannot ship to the stores, I have left those cost coefficients blank.

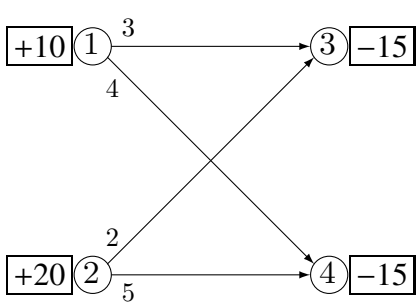
I solved the problem by using our transportation algorithm as usual but skipping the tableau spots that are blank, obtaining this optimal tableau.

		stores			extra shipments	
		(4) 10	(5) 25	(6) 25	(7) 15	(8) 15
factories	(1) 25	0	2	0^{25}	0	2
	(2) 25	0^{10}	4	0^{0}	0^{15}	4
	(3) 25	0^{0}	0^{25}	4	0	0
fictitious source	(9) 15				0^{0}	0^{15}

In this scenario the excess supply all goes from factory 2 through node 7 to the store at node 4, so that store receives those 15 boxes in addition to its minimum demand of 10. The 15 boxes that are shipped from node 9 through node 8 to the store at node 5 are only a mathematical fiction, so the store at node 5 actually receives only its minimum demand of 25.

6.3 Transshipment

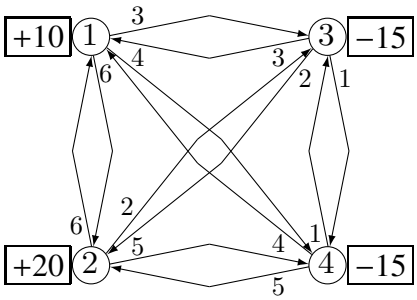
Our formulation of the transportation problem assumes that each node is either a supply or a demand, that only supply nodes ship, and that they ship only to demand nodes. The network on the left has $p = 2$ supply nodes and $q = 2$ demand nodes, and the transportation tableau on the right shows its supplies, demands, per-unit shipping costs, and optimal flows.



	(3) 15	(4) 15
(1) 10	3	4^{10}
(2) 20	2^{15}	5^{5}

The total cost of these shipments is 95.

Now suppose that directed links are added to to make the network fully connected, permitting flow at the same cost in either direction between any two nodes.



	1 0	2 0	3 15	4 15
1 10	0	6	3	4
2 20	6	0	2	5
3 0	3	2	0	1
4 0	4	5	1	0

The resulting network is shown on the left with each link's per-unit shipping cost at the tail of its arrow. The added connections between nodes 1 and 2 and between nodes 3 and 4 have costs of 6 and 1 respectively, but the connections between the other nodes have the same costs as in the previous network. Now transshipments are allowed everywhere, so the supply at node 1 or node 2 can take any path to node 3 or node 4.

We can formulate the problem of finding the least-cost assignment of flows as the transportation problem on the right, in which each node is both a source *and* a destination. The diagonal elements of this **transshipment tableau** are zero because it costs nothing to ship from a node to itself. Because of the ordering of the rows and columns, the submatrix in the lower left partition of the tableau is the transpose of the submatrix in the upper right.

There is no supply at node 3 or node 4 and there is no demand at node 1 or node 2, so the optimal solution to this enlarged transportation problem is the one given on the previous page for the network without transshipments. Our transportation algorithm will not ship anything to a node where the demand is zero, and it cannot ship anything from a node whose supply is zero. To make transshipments possible it is necessary to add a fictitious **buffer stock**, equal to the total demand, to *each* supply and *each* demand, like this.

	1 0+30	2 0+30	3 15+30	4 15+30
1 10+30	0^{30}	6	3	4^{10}
2 20+30	6	0^{30}	2^{15}	5^{5}
3 0+30	3	2	0^{30}	1
4 0+30	4	5	1	0^{30}

The buffer stock is a mathematical fiction, because adding it to both the supply and the demand at a node does not change the net amount of stuff to be shipped or received. I have modeled this fact in the transportation tableau by assigning a flow equal to the buffer stock on each of the $p + q = 4$ diagonal zeros. This shows each node shipping its extra supply to itself and thereby satisfying its own extra demand. To have a basic assignment of flows the transshipment tableau needs a total of $2(p+q)-1 = 7$, so in the upper right partition I have made the assignment of $p + q - 1 = 3$ flows that we already know is optimal for the original network. Now we can use our transportation algorithm to solve the problem.

	30	30	45	45	**u**
40	0^{30}	6	3	4^{10}	0
50	6	0^{30}	2^{15}	5^{5}	1
30	3	2	0^{30}	1	−1
30	4	5	1	0^{30}	−4
v	0	−1	1	4	

price →

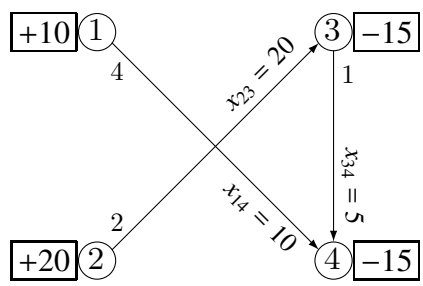
	30	30	45	45
40	0^{30}	7	2	0^{10}
50	5	0^{30}	0^{15}	0^{5}
30	4	4	0^{30}	$\boxed{-2}$
30	8	10	4	0^{30}

← shift 5

	30	30	45	45	**u**
40	0^{30}	7	2	0^{10}	0
50	5	0^{30}	0^{20}	0	−2
30	4	4	0^{25}	-2^{5}	−2
30	8	10	4	0^{30}	0
v	0	2	2	0	

price ↓

	30	30	45	45
40	0^{30}	5	0	0^{10}
50	7	0^{30}	0^{20}	2
30	6	4	0^{25}	0^{5}
30	8	8	2	0^{30}

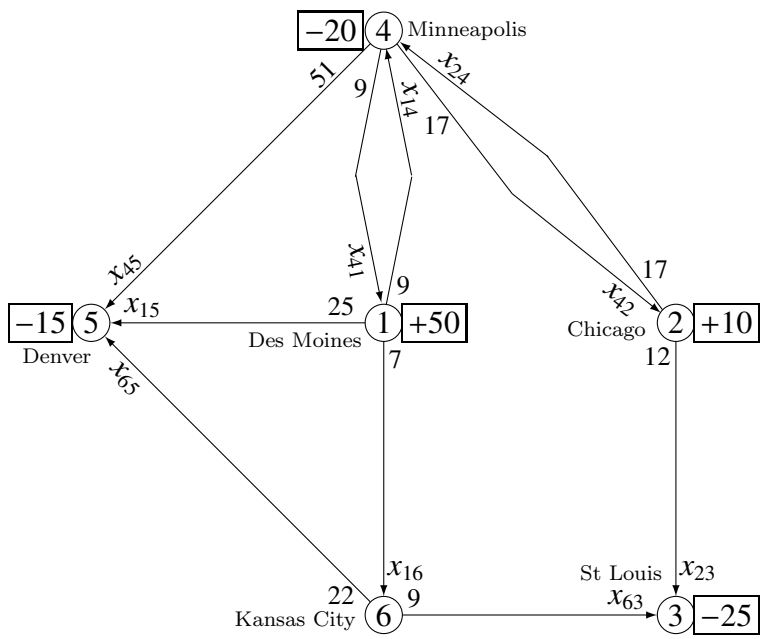


The flows on the off-diagonal spots in the optimal transshipment tableau solve the network problem, as pictured to the right. Of the 20 units flowing from node 2 to node 3, 15 satisfy the demand at node 3 and the other 5 are redirected to node 4. In the accounting of the tableau this moves 5 units of buffer stock from node 3 to node 4, but they are made up for by the 5-unit excess of x_{23} over the demand at node 3.

Allowing transshipments can never increase the optimal shipping cost. For this example the original network had an optimal cost of 95, but the cost of the transshipment solution is 85. The nonbasic spot $(1, 3)$ has zero reduced cost, so there is at least one alternate optimum.

6.4 General Network Flows

A network in which transshipments are allowed at only certain nodes or from which some links are missing can be described by a **sparse transshipment tableau** as shown on the next page for the `nf1` problem of §6.0. The nodes are ordered so that those with supplies come first, those having zero net stock come next, and those having demands come last.



	① 0+60	② 0+60	⑥ 0+60	③ 25+60	④ 20+60	⑤ 15+60	
① 50+60	0^{60}		7^{25}		9^{10}	25^{15}	supplies
② 10+60		0^{60}		12	17^{10}		supplies
⑥ 0+60			0^{35}	9^{25}		22	transshipments
③ 0+60				0^{60}			demands
④ 0+60	9	17			0^{60}	51	demands
⑤ 0+60						0^{60}	demands

The 10 nonzero shipping costs are for the links that are in the network, and the zero shipping costs on the diagonal make it free to ship from a node to itself. The supply at each supply or transshipment node and the demand at each transshipment or demand node is increased by a buffer stock of 60, equal to the total supply. This transshipment tableau has 6 rows and 6 columns, so every basic feasible solution to the problem it describes must have $6+6-1=11$ basic variables even though many of the possible links are missing.

The simplex tableau that we constructed for this problem in §6.0 has a constraint row for each node, but one row is redundant because supply equals demand so each canonical form has 5 basic variables. I found a feasible shipping schedule with the 5 basic variables $x_{14}=10$, $x_{15}=15$, $x_{16}=25$, $x_{24}=10$, $x_{63}=25$ and assigned those flows in the transshipment tableau. Then I assigned flows on the 6 diagonal zeros to make each row and column sum correct, so that a total of $5+6=11$ spots are basic as required.

With this initial basic feasible assignment of flows I solved the problem using our transportation algorithm as shown on the next page, obtaining the same optimal point $x_{14}=20$, $x_{15}=15$, $x_{16}=15$, $x_{23}=10$, $x_{63}=15$ that we found using the tableau simplex method.

	60	60	60	85	80	75	**u**
110	0^{60}		7^{25}		9^{10}	25^{15}	0
70		0^{60}		12	17^{10}		8
60			0^{35}	9^{25}		22	−7
60				0^{60}			−16
60	9	17			0^{60}	51	−9
60						0^{60}	−25
v	0	−8	7	16	9	25	

price →

	60	60	60	85	80	75
110	0^{60}		0^{25}		0^{10}	0^{15}
70		0^{60}		-12	0^{10}	
60			0^{35}	0^{25}		4
60				0^{60}		
60	18	34			0^{60}	35
60						0^{60}

← shift 10

	60	60	60	85	80	75	**u**
110	0^{60}		0^{15}		0^{20}	0^{15}	0
70		0^{60}		-12^{10}	0		−12
60			0^{45}	0^{15}		4	0
60				0^{60}			0
60	18	34			0^{60}	35	0
60						0^{60}	0
v	0	12	0	0	0	0	

price →

	60	60	60	85	80	75
110	0^{60}		0^{15}		0^{20}	0^{15}
70		0^{60}		0^{10}	12	
60			0^{45}	0^{15}		4
60				0^{60}		
60	18	22			0^{60}	35
60						0^{60}

6.4.1 Finding a Basic Feasible Solution

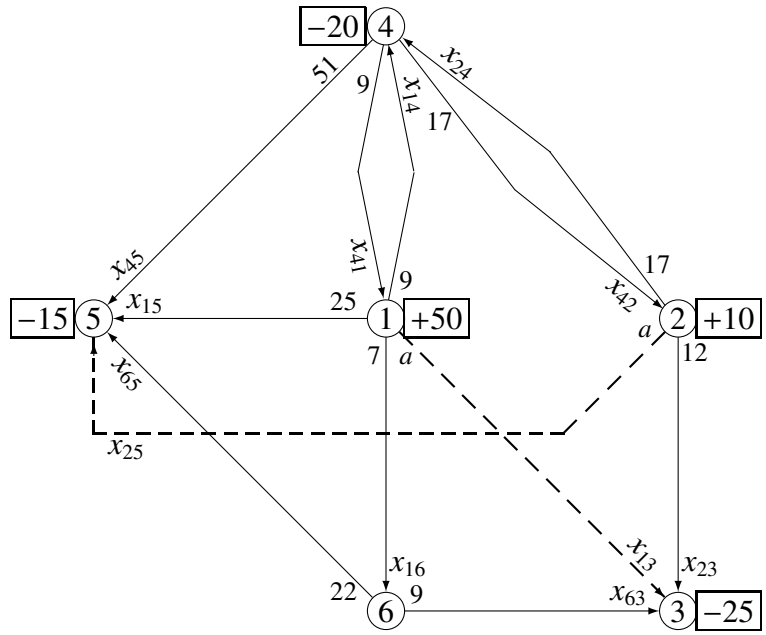
In solving `nf1` as a sparse transshipment problem we used the basic feasible shipping schedule $x_{14} = 10$, $x_{15} = 15$, $x_{16} = 25$, $x_{24} = 10$, $x_{63} = 25$ to construct an initial assignment of flows. This schedule can be found by trying different assignments of flow in the network diagram, but that is possible only for toy problems. A shipping schedule can also be found by pivoting in the initial simplex tableau for a general network problem in the same way that we pivoted in the initial simplex tableau for the transportation problem of §6.1.1, but as discussed earlier using a simplex tableau at all is impractical for network problems of realistic size.

In solving the transshipment problem of §6.3, I used a shipping schedule that was optimal for the transportation problem in the upper right partition of the transshipment tableau. That was possible because every source node could ship to every demand node, which might not be true in a general network problem. In the `nf1` problem there is no link from supply node 1 to demand node 3 or from supply node 2 to demand node 5, so the transportation problem represented by the upper right partition of its transshipment tableau is

	③ 25	④ 20	⑤ 15
① 50		9	25
② 10	12	17	

in which there is no way to make a feasible assignment of flows. But suppose that we add **artificial links** to make the missing connections, as shown dashed in the network diagram

below. If we make the shipping cost a on each artificial link arbitrarily high then any optimal solution we find will surely assign zero flow there (a least-cost solution would include flow on such a link only if the original network problem were infeasible).



When the artificial links are included in the model the transportation tableau in the upper right partition of the transshipment tableau no longer has any empty cells.

	1 0+60	2 0+60	6 0+60	3 25+60	4 20+60	5 15+60	
1 50+60	0^{60}		7^{0}	a^{25}	9^{20}	25^{5}	supplies
2 10+60		0^{60}		12	17	a^{10}	
6 0+60			0^{60}	9		22	transshipments
3 0+60				0^{60}			demands
4 0+60	9	17			0^{60}	51	
5 0+60						0^{60}	

Now we can ship the entire buffer stock on each diagonal zero and use the northwest corner rule in the transportation part of the transshipment tableau. This assignment of flows is feasible but it makes only 10 variables basic and we need 11. The transportation-problem links form a tree connecting nodes 1, 2, 3, 4, and 5, but the directed links in a basic feasible solution to a transshipment problem must form a tree connecting *all* the nodes [3, §7.7] and therefore must include a link from some supply to each transshipment-only node. To provide this connection I assigned $x_{16} = 0$. Starting from this initial basic feasible assignment of flows, I used our transportation algorithm to solve the problem as shown on the next page.

	60	60	60	85	80	75	**u**
110	0^{60}		7^{0}	a^{25}	9^{20}	25^{5}	0
70		0^{60}		12	17	a^{10}	$a-25$
60			0^{60}	9		22	-7
60				0^{60}			$-a$
60	9	17			0^{60}	51	-9
60						0^{60}	-25
v	0	$-(a-25)$	7	a	9	25	

— price →

	60	60	60	85	80	75
	0^{60}		0^{0}	0^{25}	0^{20}	0^{5}
		0^{60}		$37-2a$	$33-a$	0^{10}
			0^{60}	$16-a$		4
				0^{60}		
	18	$a+1$			0^{60}	35
						0^{60}

shift 10

	60	60	60	85	80	75	**u**
110	0^{60}		0^{0}	0^{15}	0^{20}	0^{15}	0
70		0^{60}		$(37-2a)^{10}$	$33-a$	0	$(37-2a)$
60			0^{60}	$16-a$		4	0
60				0^{60}			0
60	18	$a+1$			0^{60}	35	0
60						0^{60}	0
v	0	$-(37-2a)$	0	0	0	0	

→

	60	60	60	85	80	75
	0^{60}		0^{0}	0^{15}	0^{20}	0^{15}
		0^{60}		0^{10}	$a-4$	$2a-37$
			0^{60}	$16-a$		4
				0^{60}		
	18	$38-a$			0^{60}	35
						0^{60}

shift 15

	60	60	60	85	80	75	**u**
110	0^{60}		0^{15}	0	0^{20}	0^{15}	0
70		0^{60}		0^{10}	$a-4$	$2a-37$	$-(16-a)$
60			0^{45}	$(16-a)^{15}$		4	0
60				0^{60}			$-(16-a)$
60	18	$38-a$			0^{60}	35	0
60						0^{60}	0
v	0	$(16-a)$	0	$(16-a)$	0	0	

→

	60	60	60	85	80	75
	0^{60}		0^{15}	$a-16$	0^{20}	0^{15}
		0^{60}		0^{10}	12	$a-21$
			0^{45}	0^{15}		4
				0^{60}		
	18	22			0^{60}	35
						0^{60}

The solution process turns some of the u_i, v_j, and c_{ij} into expressions involving a, the shipping cost on the artificial links. In the second tableau the reduced costs $37-2a$, $33-a$, and $16-a$ are all *negative* because a is positive and arbitrarily high; in the final tableau the reduced costs $c_{13} = a-16$ and $c_{25} = a-21$ of the artificial links are both *positive* for the same reason. The final tableau shows the shipping schedule $x_{14} = 20$, $x_{15} = 15$, $x_{16} = 15$, $x_{23} = 10$, and $x_{63} = 15$, which we earlier found to be optimal. The artificial links make it possible to find an initial basic feasible assignment of flows, but because the original problem is feasible they are nonbasic in the final tableau and do not enter into the optimal solution.

6.4.2 The General Network Flow Algorithm

The solution process [127] [151, §6.3] that we developed in §6.4.0 and §6.4.1 is summarized in the following formal statement of the general network flow algorithm. The complications that arise from degeneracy in the transportation part of the sparse transshipment tableau are explained in [square brackets].

0. **initialize**
   - Construct a sparse transshipment tableau for the problem, ordering the nodes so that those with supplies come first, those having zero net stock come next, and those having demands come last. Add a buffer stock, equal to the total demand, to each supply and each demand. In each empty cell of the transportation part of the tableau, insert the shipping cost a of an artificial link. The diagonal costs should be zero, and there should be as many off-diagonal cost entries as there are links in the network.
   - Find an initial basic feasible solution $\mathbf{x}$ by assigning the buffer stock on each diagonal zero, assigning a flow of zero on some off-diagonal element in each pure-transshipment column, and using the northwest corner rule or some other starting rule in the transportation part of the tableau. [If the basis is degenerate, some x_{ij} in this part will also be zero.] If there are m nodes in the network, $2m - 1$ spots in the sparse transshipment tableau should be basic.
1. **solve the sparse transshipment problem**
   - Apply steps 1-4 of the transportation simplex algorithm in §6.1.4 to the sparse transshipment tableau, assuming that the artificial link cost a is arbitrarily high. If an artificial link remains in the optimal solution, the original problem was infeasible. Otherwise, the optimal shipping schedule consists of the off-diagonal flows [some of which might be zero].

6.5 Solving Network Models

The algorithms we have developed can be used to solve small transportation, transshipment, and general network flow examples by hand, but the devil turns out to be in the details when they are used for practical applications.

6.5.1 Computer Implementation

To solve problems of realistic size the calculations must be performed by a computer program. The transportation simplex algorithm of §6.1.4 is at the heart of all three algorithms, and it has three main steps. Step 2, updating the reduced costs, requires that the same arithmetic

be performed on every element of the tableau and is thus simple to automate. Steps 1 and 3, however, involve some operations that are much harder to write code for than they are to do by hand. To find the dual vectors it is easy to solve a linear system for the u_i and v_j, but first the system must be *constructed* by finding each basic spot, making its cost the right-hand side of its equation, and filling in the coefficient matrix 1s corresponding to u_i and v_j. To shift flow it is easy to alternately add and subtract the same number on successive spots in a loop, but first the loop must be *discovered.* Having chosen a nonbasic spot with a negative c_{ij} we might search row i for basic spots (i,k), then for each such spot search column k for basic spots (l,k), and then for each such spot search row l for a basic spot in column j to close the loop. But of course only the simplest loops have just four links; to find any possible loop requires a more sophisticated approach.

How best to code these operations depends entirely on the data structures that are chosen to represent the problem data and the state of the solution. A sparse transshipment tableau that we draw by hand includes cells that contain reduced costs and many others that are blank. Some of the nonblank spots are basic, with link flows that are either positive or zero, while others are nonbasic with link flows that are also zero. On some tableaus we sketch a polygon that indicates a loop. In a language like MATLAB we naturally think of representing all of these objects by matrices and vectors, perhaps using sparse-matrix techniques [50, §22] [100, §11.6.2] to compress the sparse transshipment tableau. But because the underlying data structure of a network problem is the network rather than an array it might be much simpler to implement the operations we need in a programming environment like C++, which supports user-defined data structures and operations.

In a practical implementation of the general network flow algorithm it might be desirable also to automate step 0, sparing the analyst the tedium of constructing a large sparse transshipment tableau from problem data and of finding an initial basic feasible solution.

Implementing the algorithms in this Chapter is, unfortunately, far beyond the scope of this book. A few network optimization codes are available through the NEOS web server [117, §9] but I have found a detailed algorithm description for only one, `RELAX-IV` [15]. This is perhaps unsurprising given the technical challenge of producing industrial-strength code and the commercial value of keeping it proprietary.

6.5.2 Capacity Constraints

Anyone who has been stuck in traffic has encountered an active **link capacity constraint**. These are simple upper bounds like $x_{ij} \le w_{ij}$, so they are trivial to incorporate in a tableau simplex formulation and can even be exploited in solving the linear program by the matrix simplex method (see §4.3.2). Unfortunately, it is more complicated to accommodate capacity constraints in the general network flow algorithm [151, p228]. To see why, consider adding a capacity constraint to our original linear programming formulation of the `nf1` problem. At the top of the next page I have insisted that $x_{15} \le 10$ and introduced the slack variable s_{15} to make the added constraint an equality.

$$\begin{array}{lll}
\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & z(\mathbf{x}) = 9x_{14} + 25x_{15} + 7x_{16} + 12x_{23} + 17x_{24} + 9x_{41} + 17x_{42} + 51x_{45} + 9x_{63} + 22x_{65} & \\
\text{subject to} & x_{41} - x_{14} - x_{15} - x_{16} = -50 & (1) \\
& x_{42} - x_{23} - x_{24} = -10 & (2) \\
& x_{23} + x_{63} = 25 & (3) \\
& x_{14} + x_{24} - x_{41} - x_{42} - x_{45} = 20 & (4) \\
& x_{15} + x_{45} + x_{65} = 15 & (5) \\
& x_{16} - x_{63} - x_{65} = 0 & (6) \\
& x_{15} + s_{15} = 10 & \text{link capacity constraint} \\
& \mathbf{x} \geq \mathbf{0} &
\end{array}$$

For this problem the set of links is

$$\mathbb{N} = \{(1,4)\ (1,5)\ (1,6)\ (2,3)\ (2,4)\ (4,1)\ (4,2)\ (4,5)\ (6,3)\ (6,5)\}.$$

If $y_1 \ldots y_7$ are dual variables corresponding to the constraints and $\mathbf{x}$ is a feasible assignment of flows, then recapitulating our analysis in §6.1.2 we find for this problem that

$$\alpha(\mathbf{x}) - \beta(\mathbf{y}) = \sum_{(i,j)\in\mathbb{N}} (c_{ij} - y_i - y_j)x_{ij} - 10y_7.$$

Reasoning as we did there, we derive a slightly different algorithm for solving the sparse transshipment problem. To find a $\mathbf{y}$ that makes the primal and dual objectives equal we must now solve one or the other of these systems, depending on the value of x_{15}.

$$\left.\begin{array}{rcl} c_{ij} - y_i - y_j & = & 0 \text{ where } x_{ij} \text{ is basic} \\ y_7 & = & 0 \end{array}\right\} \quad x_{15} < 10$$

$$\left.\begin{array}{rcl} c_{ij} - y_i - y_j & = & 0 \text{ where } x_{ij} \text{ is basic and } (i,j) \neq (1,5) \\ (c_{15} - y_1 - y_5)x_{15} - 10y_7 & = & 0 \end{array}\right\} \quad x_{15} = 10$$

Also, in shifting flows we must keep $x_{15} \leq 10$.

If two of the link flows have upper bounds then both must be observed in shifting flow and we have four cases to consider in finding $\mathbf{y}$: the first flow at its upper bound but not the second, the second but not the first, neither at its upper bound, or both at their upper bounds. This naïve approach soon becomes unwieldy as the number of variable bounds increases. Fortunately, there is a more sophisticated approach [145, §7] to using the transportation problem dual in this context, which yields a more practical algorithm for capacitated flow problems. Unfortunately, it too is far beyond the scope of this book.

6.5.3 Related Problems

The minimum-cost flow problem that we have studied is just one of many network optimization problems. We will take a glance at three others here, and revisit two of them in §7.

In the **assignment problem** [3, §7.6] [151, §6.4] [79, §4.4] task $i \in \{1 \dots m\}$ can be performed by agent $j \in \{1 \dots m\}$ at cost c_{ij}. If task i is assigned to agent j then $x_{ij} = 1$, otherwise $x_{ij} = 0$. We seek an assignment $\mathbf{x}$ of tasks to agents that minimizes total cost.

$$\begin{aligned}
\underset{\mathbf{x}\in\mathbb{Z}^{m\times m}}{\text{minimize}} \quad & \sum_{i=1}^{m}\sum_{j=1}^{m} c_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{i=1}^{m} x_{ij} = 1 && j = 1 \dots m \\
& \sum_{j=1}^{m} x_{ij} = 1 && i = 1 \dots m \\
& x_{ij} \in \{0, 1\} && \text{for all } (i, j)
\end{aligned}$$

The first set of constraints ensure that exactly one task is assigned to each agent, and the second set ensure that exactly one agent is assigned to each task. Because each x_{ij} can be only zero or one, this is an **integer programming problem**. In the linear programs we have studied so far, $\mathbf{x}$ has always been a *real* variable (see §1.1.3) and we have seen that the optimal point then need not necessarily have integer components even if the problem data are all whole numbers. But because of the special structure of a transportation problem, if each x_{ij} is a whole number in the initial basic feasible assignment of flows then each $x^\star_{ij}$ will be too [3, p177] (see Exercise 6.6.24). We can therefore replace $x_{ij} \in \{0, 1\}$ by the constraint $x_{ij} \geq 0$ in the formulation above and solve the assignment problem as a transportation problem in which $p = q = m$ and each $s_i = d_j = 1$.

In the **shortest-path problem** [151, §6.5] [79, §6.3] each link $(i, j) \in \mathbb{N}$ of a network has a length $c_{ij} \geq 0$ and a **path** is a sequence of directed links leading from an origin node to a destination node. If link (i, j) is included in the path then $x_{ij} = 1$, otherwise $x_{ij} = 0$. We seek a vector $\mathbf{x}$ specifying a path that has the lowest total length.

$$\begin{aligned}
\underset{\mathbf{x}\in\mathbb{Z}^{m\times m}}{\text{minimize}} \quad & \sum_{i=1}^{m}\sum_{j=1}^{m} c_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{(k,j)\in\mathbb{N}} x_{kj} - \sum_{(i,k)\in\mathbb{N}} x_{ik} = \begin{cases} +1 & \text{if } k \text{ is the origin node} \\ -1 & \text{if } k \text{ is the destination node} \\ 0 & \text{otherwise} \end{cases} \\
& x_{ij} \in \{0, 1\} \qquad \text{for all } (i, j) \in \mathbb{N}
\end{aligned}$$

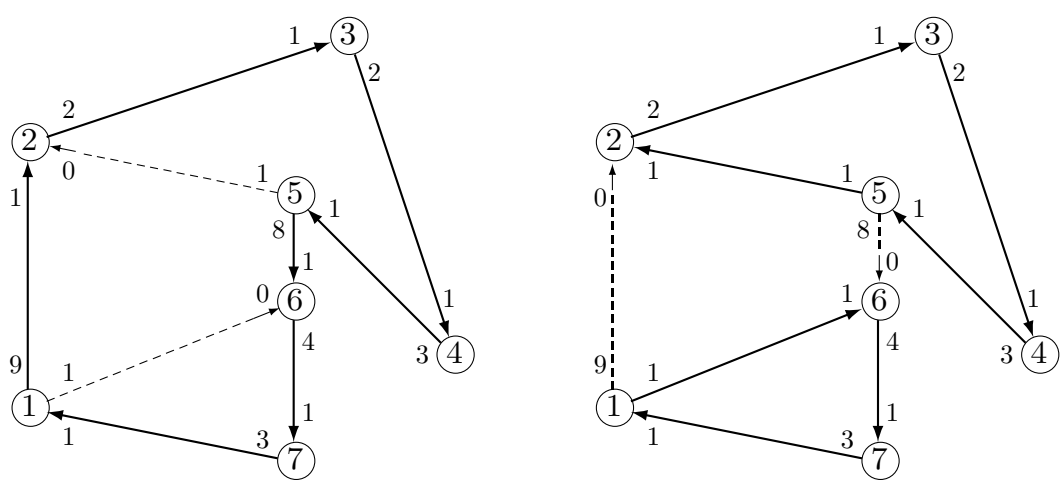
The constraints ensure that the origin node is exited one more time than it is entered, the destination node is entered one more time than it is exited, and every other node is exited as many times as it is entered. Because traversing any link incurs a cost, the minimization ensures that no node is entered or exited more than once. This problem is equivalent [151, p179] to the assignment problem, and can also be solved as a transportation problem.

These models have even simpler structures than the transportation problem of which they are special cases, and algorithms have been discovered for solving the assignment [10] and shortest path [151, §7.6-7.7] problems that are even more efficient than the transportation simplex method.

In the **traveling salesman problem** [3, p246-247] [151, §6.5] a salesperson (who is just as likely to be a woman) must depart from a city of origin, visit each of $m-1$ other cities exactly once, and return to the city of origin. Each c_{ij} is the cost of traveling from city i to city j. If link $(i,j) \in \mathbb{N}$ is included in the salesperson's **tour** then $x_{ij} = 1$, otherwise $x_{ij} = 0$. We seek a vector $\mathbf{x}$ specifying a tour of minimum total cost.

$$
\begin{array}{lll}
\underset{\mathbf{x}\in\mathbb{Z}^m\ \mathbf{w}\in\mathbb{Z}^{m-1}}{\text{minimize}} & \displaystyle\sum_{(i,j)\in\mathbb{N}} c_{ij}x_{ij} & \\
\text{subject to} & \displaystyle\sum_{i=1}^{m} x_{ij} = 1 & \text{for all } j \text{ such that } (i,j)\in\mathbb{N} \\
 & \displaystyle\sum_{j=1}^{m} x_{ij} = 1 & \text{for all } i \text{ such that } (i,j)\in\mathbb{N} \\
 & w_i - w_j + m x_{ij} \le m-1 & \text{for } (i,j)\in\mathbb{N},\ i\neq 1,\ j\neq 1 \\
 & x_{ij} \in \{0,1\} & \text{for } (i,j)\in\mathbb{N}
\end{array}
$$

The first set of constraints ensure that each city will be entered exactly once, and the second set ensure that each city will be exited exactly once. If those constraints were sufficient this would be an assignment problem, but they do not ensure that the chosen links form a tour. The network below has links $\mathbb{N} = \{(1,2)\ (1,6)\ (2,3)\ (3,4)\ (4,5)\ (5,2)\ (5,6)\ (6,7)\ (7,1)\}$ connecting its $m = 7$ nodes. The unique tour $\mathbf{x}^\star = [1,0,1,1,1,0,1,1,1]^\top$ on the left has a cost of 31 but the unique pair of **subtours** $\bar{\mathbf{x}} = [0,1,1,1,1,1,0,1,1]^\top$ on the right, which also satisfy the first two constraints, have a total cost of only 16 and would therefore be found by the minimization.



To prevent the minimization from finding the subtour solution we could introduce these constraints.

$$\begin{aligned} x_{23} + x_{34} + x_{45} + x_{52} &\leq 3 \\ x_{16} + x_{67} + x_{71} &\leq 2 \end{aligned}$$

In any subtour all of the $x_{ij} = 1$, so the left-hand sides in these constraints add up to $4 > 3$ and $3 > 2$ disallowing both subtours. Finding all of the possible subtours in a larger network is usually very hard to do, but the third constraint in the problem statement rules out all of them [151, p455-456]. Writing out this anti-subtour constraint for the example, we get these inequalities.

$$\begin{aligned} w_2 - w_3 + 7x_{23} &\leq 6 \\ w_3 - w_4 + 7x_{34} &\leq 6 \\ w_4 - w_5 + 7x_{45} &\leq 6 \\ w_5 - w_2 + 7x_{52} &\leq 6 \\ w_5 - w_6 + 7x_{56} &\leq 6 \\ w_6 - w_7 + 7x_{67} &\leq 6 \end{aligned}$$

Summing the first four yields $7(x_{23} + x_{34} + x_{45} + x_{52}) \leq 24$ or $x_{23} + x_{34} + x_{45} + x_{52} \leq \frac{24}{7} < 4$, ruling out the first subtour; doing that makes the second subtour impossible as well. These inequalities do not involve node 1, so they do not exclude the tour we found (proving in general that the anti-subtour constraint does not rule out any tour takes more work).

The anti-subtour constraint makes the traveling salesman problem *not* equivalent to the assignment problem, so it *cannot* be solved by using our transportation algorithm. When we take up integer programming in §7 you will learn an algorithm for solving it, and also another approach to solving the shortest-path problem.

6.6 Exercises

6.6.1 [E] A *network diagram* is often helpful in the formulation of a network flow problem. (a) In what ways does it idealize the underlying problem? (b) What are its constituent parts? (c) What makes a link *directed*? (d) What is a *transshipment point*? Does a transshipment point necessarily have zero supply and zero demand? (e) What is the *net stock* at a node? (f) Where in a network diagram is a *link cost* c_{ij} shown? (g) Where in a network diagram is a *link flow* x_{ij} shown? (h) What is a *shipping schedule*?

6.6.2 [E] What makes a shipping schedule feasible?

6.6.3 [E] Suppose that (i, j) and (j, i) are directed links connecting node i with node j. (a) Is it necessarily true that $c_{ij} = c_{ji}$? If so, explain why; if not, suggest a reason why the

link costs might be different. (b) Can it ever happen that in an optimal shipping schedule both $x_{ij} > 0$ and $x_{ji} > 0$? Explain your answer.

6.6.4 [E] What conservation law is expressed by a *node equilibrium equation*? In a network of m nodes in which total supply equals total demand, how many of the node equilibrium equations are linearly independent?

6.6.5 [H] Major repairs are being planned to route 70 in the highway network used by the meat processing company of §6.0, and construction delays are expected to increase the travel time on the link from Kansas City to St Louis by 20%. How does this affect the optimal shipping schedule?

6.6.6 [H] Suppose a network has m nodes and flow is possible in either direction between any two of them. (a) How many directed links n must there be? Show that your answer is correct for networks having $m \in \{3, 4, 5\}$ nodes, and give a convincing argument that it is correct in general. (b) If the general network flow problem with m nodes and n links is formulated as a simplex tableau, derive expressions for the number of rows and the number of columns in the tableau. (c) If $n = m(m-1)$, plot the number of elements in the simplex tableau for $m = 1 \ldots 1000$.

6.6.7 [E] What is the main advantage of the *network simplex algorithm* over the tableau simplex algorithm? List three reasons for studying the development of the network simplex algorithm.

6.6.8 [E] What makes a network flow problem a *transportation problem*?

6.6.9 [H] In §6.1 we formulated a transportation problem for solution by the tableau simplex method in a way that made the tableau's constant column $\mathbf{b}$ nonnegative. When we pivoted-in a basis we always picked a minimum-ratio row, so that $\mathbf{b}$ remained nonnegative. (a) What about the special structure of the transportation problem makes it always possible to do that? (b) How many pivots are required, and why?

6.6.10 [H] This transportation tableau describes a basic feasible solution to a network flow problem.

	9	10	11	12
20	1^{9}	3^{10}	5^{1}	7
22	2	4	6^{10}	8^{12}

(a) Draw the corresponding network diagram, showing the net stock at each node and the link costs and flows. (b) Write down the shipping schedule $\mathbf{x}^0$ that is given in the tableau, and show that it is feasible. To do this you might find it convenient to introduce node numbers. (c) Construct a simplex tableau for this problem and perform minimum-ratio pivots to obtain the basic feasible solution $\mathbf{x}^0$. (d) Explain how the flows shown in the transportation tableau get assigned by the *northwest corner rule*.

6.6.11 [H] Is it always possible to make an initial assignment of flows in a transportation tableau by using the northwest corner rule? Explain.

6.6.12 [E] If the supply-node equilibrium constraints of a transportation problem are multiplied through by -1 we get the linear program $\mathscr{P}$ given in §6.1.2. (a) Write down this algebraic formulation of the problem. (b) In a network having source nodes $1, 2 \ldots p$ and demand nodes $p+1, p+2 \ldots p+q$, what is the set $\mathbb{S}$? What is the set $\mathbb{D}$? (c) What do i and j index? (d) Explain the formula for $\alpha(\mathbf{x})$. (e) Explain the functional constraints. (f) Why is it necessary for $\mathbf{x}$ to be nonnegative?

6.6.13 [H] If you worked Exercise 5.5.34 you discovered that using the tableau simplex method to solve the dual of the transportation problem is even harder than using it to solve the primal. Is there some other way in which the dual is useful in solving the primal? Explain.

6.6.14 [E] In §5.2.2 we derived the transportation problem dual $\mathscr{D}$ that is used in §6.1.2. (a) Write down $\mathscr{D}$. (b) In a transportation network having p source nodes and q demand nodes, what is the set $\mathbb{S}$? What is the set $\mathbb{D}$? (c) What do i and j index? (d) What do the variables $\mathbf{u}$ and $\mathbf{v}$ represent? Hint: they are row multipliers. (d) Explain the formula for $\beta(\mathbf{u}, \mathbf{v})$. (e) Explain the functional constraints. (f) Why is it necessary for $\mathbf{u}$ and $\mathbf{v}$ to be free variables?

6.6.15 [H] In §6.1.2 we used the primal $\mathscr{P}$ and dual $\mathscr{D}$ of the transportation problem to derive a way of determining whether a particular assignment of flows in a transportation tableau is optimal. (a) What is the way that we derived? (b) Why does it work?

6.6.16 [H] If $\hat{\mathbf{x}}$ is feasible for the primal of a transportation problem, how can we choose dual variables $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ so that $\alpha(\hat{\mathbf{x}}) = \beta(\hat{\mathbf{u}}, \hat{\mathbf{v}})$?

6.6.17 [P] To make $\alpha(\mathbf{x}) = \beta(\mathbf{u}, \mathbf{v})$ we find u_i and v_j such that $c_{ij} - u_i - v_j = 0$ for each (i, j) where x_{ij} is basic. (a) If there are p supply nodes and q demand nodes in the transportation problem, how many equations are there in this system? How many u_i and v_j are there to find? (b) Describe the *chain reaction solution* method for finding vectors $\mathbf{u}$ and $\mathbf{v}$ that satisfy the system. In using this method we have arbitrarily set $u_1 = 0$; what happens if we set $u_1 = -7$ instead? (c) How can MATLAB be used to obtain the chain reaction solution? (d) Are the vectors $\mathbf{u}$ and $\mathbf{v}$ uniquely determined?

6.6.18 [H] If $\hat{\mathbf{x}}$ is feasible for the primal of a transportation problem and we have chosen $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ so that $\alpha(\hat{\mathbf{x}}) = \beta(\hat{\mathbf{u}}, \hat{\mathbf{v}})$, what must be true in order for us to conclude that $\hat{\mathbf{x}}$ is optimal?

6.6.19 [E] In the simplex tableau for a transportation problem, pivoting-in a basis produces a vector $\mathbf{c}^\top$ of reduced costs. (a) If we make the same basic feasible assignment of flows in the transportation tableau for the problem, how do we find those same reduced costs? (b) Why is it necessary to find the reduced costs? Explain.

6.6.20 [H] A pivot in the simplex tableau for a transportation problem has the effect of increasing a flow that was nonbasic (and hence zero) while decreasing a flow that was basic to zero so that it becomes nonbasic. (a) Explain how this can be accomplished in a network diagram by *shifting* flow around a loop. What determines the maximum amount of flow that

can be shifted? (b) Draw the new network diagram that results from completing the shift that is indicated in the network diagram of §6.1.2.

6.6.21 [H] In a network diagram, a flow can be increased from zero by introducing that link to form a loop and then shifting flow onto it. In a transportation tableau the flow can be increased from zero by choosing the nonbasic spot that represents that link and forming a loop which includes it. (a) What properties must the loop in the transportation tableau have? (b) Describe a systematic procedure for finding the loop. (c) Why is the loop unique? Hint: What do the spots in a loop correspond to in the simplex tableau for the transportation problem?

6.6.22 [E] To perform a simplex-rule pivot in a transportation tableau we shift flow around a loop. (a) Does the direction of the shift matter? Explain. (b) What determines the largest amount that can be shifted?

6.6.23 [E] In performing an iteration of the transportation simplex algorithm we shift t units of flow around a loop, where t is the smallest of the flows assigned to the spots from which we subtract in doing the shift. (a) What happens if we shift *less* than t units of flow? Can the resulting status of the network be described by a simplex tableau? Explain. (b) What happens if we shift *more* than t units of flow? Can the resulting status of the network be described by a simplex tableau? Explain.

6.6.24 [H] If in the initial basic feasible assignment of flows for a transportation problem each element of $\mathbf{x}^0$ is a whole number, then so will be the elements of the optimal shipping schedule $\mathbf{x}^\star$. Explain why.

6.6.25 [H] Given an optimal simplex tableau we can pivot to all of the other basic feasible solutions, and from each tableau we can read off the objective value corresponding to that canonical form (see §3.2.2). Now suppose that we are instead given an optimal transportation tableau, such as the one we found for `nf3` in §6.1.4 in which $\mathbf{x}^\star = [0, 0, 10, 10, 5, 0, 0, 0, 10]^\intercal$. (a) Is it possible to find all of the other basic feasible solutions by operating on the transportation tableau? Explain. (b) Find $\alpha(\mathbf{x}^\star)$ for the `nf3` problem. Can this value be deduced from the optimal transportation tableau for the problem? Explain.

6.6.26 [E] An arbitrary linear program can be infeasible or unbounded, some of its vertices can be degenerate, and it can have multiple optimal points. Which of these properties can a transportation problem have? Explain.

6.6.27 [E] The northwest corner rule can be used to make an initial assignment of flows. (a) In using the rule what is evident if a flow assignment simultaneously uses up a supply and satisfies a demand, but it is *not* the final assignment? What does the northwest corner rule do then? (b) In a transportation tableau having p supply rows and q demand columns, how many spots must be basic in a basic feasible assignment of flows? (c) In performing the transportation simplex algorithm, what happens if a mistake leads to having too few basic flows assigned? (d) Can degeneracy in a transportation problem lead to cycling?

6.6.28 [H] In §6.1.3 we saw that degeneracy of a transportation problem can be revealed in the process of making an initial assignment of flows. (a) Does that always happen? (b) How else might degeneracy become evident in the transportation simplex solution process?

6.6.29 [P] In §6.1.3 the first assignment of flows that I proposed had only 4 basic variables, so with $u_1 = 0$ appended the system had 5 equations in 6 unknowns and the chain-reaction solution failed. (a) Write the underdetermined system in the form $\mathbf{My} = \mathbf{c}$, as we did for the full-rank example in §6.1.2, and solve it using MATLAB (I got `y'=[0,4,-2,9,-1,3]`). (b) Does this result have any meaning for the transportation problem? (c) How does MATLAB "solve" an underdetermined linear system?

6.6.30 [E] The transportation simplex algorithm is stated in §6.1.4. (a) List its major steps. (b) When does it stop? (c) If in performing the algorithm the flow becomes zero simultaneously on two previously-basic tableau spots, what does that indicate?

6.6.31 [H] A cinema fan can watch new movies at two theatres, both of which are advertising reduced prices next week. Theatre A offers 2 movies he wants to watch, each costing \$16.50 this week or \$9.00 next week. Theatre B offers 5 movies he wants to watch, each costing \$16.00 this week or \$13.50 next week. The fan wants to watch 3 movies this week and 4 movies next week but doesn't care which movies he watches in which week. (a) Formulate a transportation problem whose solution will tell the fan how many movies he should watch at each theatre each week. (b) Solve the problem using the transportation simplex algorithm.

6.6.32 [H] A factory supplies customers with product. The factory produces 10 units of product each month, but customer demand and the per-unit cost of shipping vary with the month of delivery as shown for the first quarter in the table below.

month of delivery	demand	shipping cost
January	5	1
February	10	2
March	15	1

Product left over in January can be stored for delivery in February or March, and product left over in February can be stored for delivery in March. However, it is company policy to begin each calendar quarter with zero inventory, so the total production for the first quarter equals the total demand and no first-quarter product can be stored for delivery after March. The warehouse charges \$2 to store 1 unit of product from January until February, but \$1 to store 1 unit of product from February until March. (a) Draw the network diagram for a transportation problem whose solution will tell how to meet the first-quarter demands at lowest total cost. (b) Write down the transportation tableau corresponding to the network diagram. (c) Use the northwest-corner rule to make an initial feasible assignment of shipments. Show that this assignment of shipments is optimal, and draw a network diagram illustrating the solution. (d) Find an alternate optimal shipping schedule, and draw a network diagram illustrating it.

6.6.33 [E] Why does the northwest corner rule often produce an initial basic feasible solution that is far from optimal? Why is it worthwhile to use a phase-1 procedure that produces a better starting point, even if it takes more work?

6.6.34 [H] To make an initial assignment of flows in the transportation tableau for the `nf2` problem in §6.1.1, we used the northwest corner rule. Instead use (a) the smallest-cost rule; (b) Vogel's rule; (c) Russell's rule. In each case report whether the initial assignment is optimal.

6.6.35 [H] Make an initial assignment of flows in the following transportation tableau [3, §7.4] by using (a) the smallest-cost rule; (b) Vogel's rule; (c) Russell's rule.

	15	10	10	5	30
10	3	6	8	11	5
15	1	9	3	2	7
30	4	2	8	25	15
5	9	1	4	9	8
10	2	4	2	11	1

(d) Solve the transportation problem.

6.6.36 [H] The transportation tableaus in §6.1.6 represent optimal solutions $\mathbf{x}^{\star 1}$ and $\mathbf{x}^{\star 2}$ of a transportation problem. Find all of the other alternate optima.

6.6.37 [H] In §6.2.1 we assumed that it costs nothing to ship extra production from a factory into its own inventory, but this might not be realistic. (a) If factories 1, 2, and 3 incur inventory stocking costs of 3, 2, and 1 respectively for each box of bolts retained there, how does the formulation change? (b) Solve the modified problem.

6.6.38 [E] If a transportation problem has too little supply to meet the demand, what does an optimal solution tell us?

6.6.39 [E] Explain the role of a *fictitious source* in a transportation problem. How do we know that the flows in the non-fictitious part of the optimal tableau are optimal for the original problem?

6.6.40 [E] A fully-connected network that has p supply nodes and q demand nodes is modeled in §6.3 as a *transshipment* problem. (a) How many rows are in a transshipment tableau? (b) How many columns are in a transshipment tableau? (c) Why are the diagonal elements of a transshipment tableau zero? (d) Why is the cost coefficient matrix in the bottom left partition the transpose of the cost coefficient matrix in the top right partition? (e) What is the purpose of a *buffer stock,* and what should be its value?

6.6.41 [E] What makes a general network flow problem different from a dense transshipment problem? Describe the construction of a *sparse transshipment tableau.* Which off-diagonal entries are nonzero?

6.6.42 [H] In §6.4 we used $x_{14} = 10$, $x_{15} = 15$, $x_{16} = 25$, $x_{24} = 10$, and $x_{63} = 25$ to make an initial assignment of flows in the sparse transshipment tableau for the `nf1` problem. (a) Perform minimum-ratio pivots to get this basis in the initial simplex tableau given in §6.0 for the `nf1` problem. (b) How did we complete the initial basic feasible assignment of flows in the sparse transshipment tableau?

6.6.43 [H] A basic feasible assignment of flows in a sparse transshipment tableau must include directed links connecting all of the nodes in the network, forming a basic feasible **spanning tree** [3, §7.7]. By performing minimum-ratio pivots in the initial simplex tableau given in §6.0 for the `nf1` problem, try to find a basis in which x_{16}, x_{63}, and x_{65} are all nonbasic, so that there is no flow to or from node 6. Why is this impossible?

6.6.44 [H] In §6.4.1 we made a feasible assignment of flows in the transportation part of the transshipment tableau by adding *artificial links* and using the northwest corner rule. (a) What ensures that these links will be nonbasic in an optimal solution to the general network problem? (b) How did we assign the other flows that are needed to make an initial basic feasible assignment for the sparse transshipment tableau? (c) Why is it necessary to assign a zero flow somewhere off the diagonal in each column that corresponds to a pure-transshipment point? (d) Could one of the other rules described in §6.1.5 be used to make the initial assignment of flows in the transportation part of the transshipment tableau even though it contains artificial links?

6.6.45 [H] Revise the simplex tableau formulation of `nf1` to include the artificial links x_{13} and x_{25} (set $a = 1000$ for numerical calculations). Perform minimum-ratio pivots to get the initial basic feasible solution that we found by doing a northwest-corner assignment of flows in the transportation part of the transshipment tableau. Pivot the simplex tableau to optimality. Are x_{13} and x_{25} zero in the optimal solution?

6.6.46 [P] Step 2 of the transportation simplex algorithm updates the c_{ij} in the tableau. Assume that the (i, j) entry of a matrix `C(p,q)` stores c_{ij} and that the (i, j) entry of matrix `F(p,q)` stores x_{ij} or -1 if the spot is nonbasic or -2 if link (i, j) is missing. If the dual variables are stored in vectors `u` and `v`, write MATLAB code to perform the update.

6.6.47 [P] Step 1 of the transportation simplex algorithm finds the dual vectors $\mathbf{u}$ and $\mathbf{v}$. Assume that the (i, j) entry of a matrix `C(p,q)` stores c_{ij} and that the (i, j) entry of matrix `F(p,q)` stores x_{ij} or -1 if the spot is nonbasic or -2 if link (i, j) is missing. If the augmented linear system that must be solved to find the dual vectors is `M*y=c` as in §6.1.2, write MATLAB code to construct the coefficient matrix `M` and the right-hand side vector `c`, solve for `y`, and extract `u` and `v` from the solution vector.

6.6.48 [P] Step 3 of the transportation simplex algorithm finds a loop and shifts flow around it. Finding a loop involves searching the transportation tableau for basic spots that are in the appropriate rows and columns to be vertices of the loop. (a) Describe an algorithm for constructing a tree of vertices that might be consecutive in a closed loop, and explain how

such a tree could be used to find the loop. (b) Using a linked list to represent the tree, write code in a programming language of your choice to implement the algorithm you described in part a.

6.6.49 [E] Search the internet for computer codes that can be used to solve the network optimization problems considered in this Chapter.

6.6.50 [H] In §6.5.2 I revised the linear programming formulation of the `nf1` problem to include a capacity constraint. (a) Solve the revised problem by pivoting in the simplex tableau. (b) Using the two-case rule we derived for finding $\mathbf{y}$, solve the capacitated sparse transshipment problem by the general network flow algorithm.

6.6.51 [E] Name three network optimization problems other than finding a minimum-cost shipping schedule.

6.6.52 [E] What is an *integer program*, and how does it differ from linear programs such as the `brewery` problem?

6.6.53 [E] Why is it possible to solve the assignment problem, which is an integer program, by using the transportation algorithm, which is based on the simplex method?

6.6.54 [H] Show that the shortest-path problem can be written as an assignment problem.

6.6.55 [E] The traveling salesman problem is very similar to an assignment problem except for the presence of anti-subtour constraints. (a) Why are these constraints necessary? (b) How do they work? (c) Why is this problem much more difficult than the assignment problem?

7

Integer Programming

In quantifying our experience of the world most of us count bagels but measure cream cheese, without ever pausing to contemplate how different one operation is from the other. When we reason about the analog world of measurement we are free to use algebra and calculus, but in the digital world of integers nothing is smooth and the exquisite machinery of real analysis gets stuck at the discontinuities. Optimization problems in which some or all of the variables are restricted to take on only whole-number values are called **integer programs** [62] [3, §8] [151, §13] [79, §18]. They are not only more difficult than smooth optimizations but fundamentally different in kind, because they are the archetype for a class of problems requiring an amount of work that is an exponential function of problem size.

Unfortunately, in many practical applications of mathematical programming the optimal vector we seek has components that naturally ought to be integers. In §1 the brewery would prefer to sell whole kegs, the chemical factory would prefer to make a whole number of process runs, the air traffic control center must assign whole people to start each shift, and the furniture factory would prefer to have workers either assemble or finish chairs rather than divide their time. Sometimes the answer to a smooth optimization happens to come out integers, or a variable is so big that its fractional part doesn't matter, or an artful interpretation makes a non-integer result useful anyway, but there are other times when an integer programming formulation cannot be avoided. This Chapter is about what to do then.

7.1 Explicit Enumeration

In §1.3.1 we found for the `brewery` problem that $\mathbf{x}^\star = [5, 12\frac{1}{2}, 0, 0]^\top$ in which an odd half-keg of Stout gets made. If all of Sarah's customers insist on buying only full kegs, she suffers a $150 \times \frac{1}{2} = \$75$ decrease in revenue, from \$2325 to \$2250. Might she do better than that by repeating the optimization subject to the additional **integer constraint** that $\mathbf{x}^\star$ have whole-number components? This is that problem, which I will name `brewip` (see §28.6.1).

$$
\begin{array}{lrcrcrcrcrl}
\underset{\mathbf{x}\in\mathbb{Z}^4}{\text{minimize}} & -90x_1 & - & 150x_2 & - & 60x_3 & - & 70x_4 & = & z(\mathbf{x}) & \\
\text{subject to} & 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & \leq & 160 & \\
 & 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & \leq & 50 & \\
 & 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & \leq & 60 & \\
 & & & & & & & x_j & \geq & 0 & \text{and integer, } j = 1\ldots4
\end{array}
$$

The original problem, without the integer constraint, is called the **linear programming relaxation** of `brewip`.


```
 1 % brewip.m: solve integer brewery problem by exhaustive enumeration
 2
 3 A=[7,10,8,12;
 4    1, 3,1, 1;
 5    2, 4,1, 3];
 6 b=[160;50;60];
 7 c=[-90;-150;-60;-70];
 8
 9 % find the maximum possible value of each variable
10 for j=1:4
11     xmax(j)=intmax;
12     for i=1:3
13         xmax(j)=min(xmax(j),fix(b(i)/A(i,j)));
14     end
15 end
16
17 % examine all integer points that might be feasible
18 zstar=0;
19 for x1=0:xmax(1)
20     for x2=0:xmax(2)
21         for x3=0:xmax(3)
22             for x4=0:xmax(4)
23                 x=[x1;x2;x3;x4];
24
25 %               is the point feasible?
26                 s=b-A*x;
27                 if(min(s) < 0)
28                    continue
29                 end
30 %
31 %               yes; update the optimal point
32                 z=c'*x;
33                 if(z < zstar)
34                    zstar=z;
35                    xstar=x;
36                    continue
37                 end
38
39             end
40         end
41     end
42 end
43
44 xstar
45 zstar
```

```
octave:1> brewip
xstar =

    4
   13
    0
    0

zstar = -2310
```

The functional constraints require $x_1 \leq \lfloor 160/7 \rfloor = 22$, $x_1 \leq \lfloor 50/1 \rfloor = 50$, and $x_1 \leq \lfloor 60/2 \rfloor = 30$, so every feasible integer point has $x_1 \in [0, 22]$; similarly $x_2 \in [0, 15]$, $x_3 \in [0, 20]$, and $x_4 \in [0, 13]$. Thus there are only $23 \times 16 \times 21 \times 14 = 108192$ **lattice points** that might be feasible. I wrote the MATLAB program listed above to 19-23 generate these points, 25-29 check each for feasibility, and 31-37 remember the feasible one having the lowest objective. The Octave session on the right shows the program finding $\mathbf{x}^\star_{\text{IP}} = [4, 13, 0, 0]^\top$ for a revenue of \$2310, which is indeed better than brewing but not selling that extra half-keg of Stout.

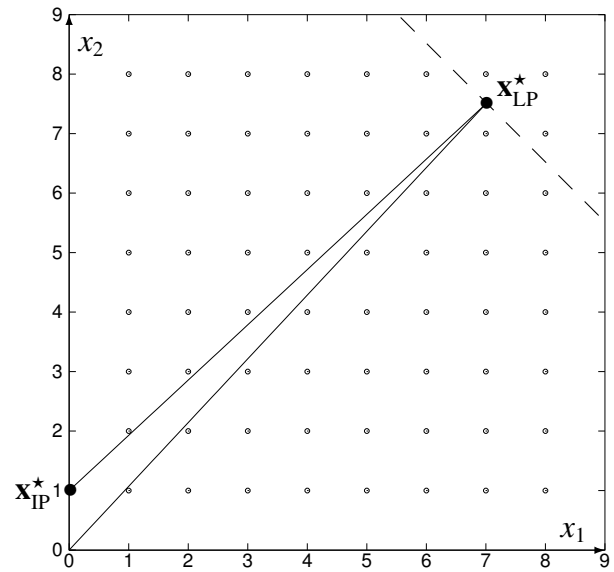
The number of lattice points that must be considered in an **exhaustive enumeration** like this grows exponentially with the number of variables in the problem, and generating

all of them is possible only if the feasible set is bounded. This makes exhaustive enumeration practical only for a subset of very small integer programs.

A **partial enumeration** generates only some of the lattice points. The simplest strategy is to round the non-integer components in the solution $\mathbf{x}^\star_{\text{LP}}$ of the linear programming relaxation. For the `brewery` problem we found $x^\star_2 = 12\frac{1}{2}$, and rounding this component down by not selling the odd half-keg of Stout yielded the point $[5, 12, 0, 0]^\top$ which we found is feasible but suboptimal; rounding up instead yields $[5, 13, 0, 0]^\top$, which violates the first and third constraints. If $\mathbf{x}^\star_{\text{LP}}$ has p non-integer components to be rounded up or down we get 2^p lattice points to check, and there is no guarantee that any of them will turn out to be $\mathbf{x}^\star_{\text{IP}}$.

Considering more of the lattice points that are near $\mathbf{x}^\star_{\text{LP}}$ makes the partial enumeration heuristic more robust, but it can still fail if not enough points are included. The problem below, which I will call `spear` (see §28.6.2), has two feasible lattice points, $[0, 0]^\top$ and $[0, 1]^\top$, and both are far enough from $\mathbf{x}^\star_{\text{LP}}$ that only exhaustive enumeration would find them.

$$\begin{array}{lrcrcl} \underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 & - & x_2 & & \\ \text{subject to} & -13x_1 & + & 14x_2 & \leq & 14 \\ & 15x_1 & - & 14x_2 & \leq & 0 \\ & & & \mathbf{x} \geq \mathbf{0} \text{ and integer} & & \end{array}$$



In **random enumeration** [29, Part 2] we select a sample of the lattice points by chance, find the objective value at each one that is feasible, and declare the point having the lowest objective value to be optimal (this resembles the pure random search algorithm for nonlinear programming discussed in §9.1). If the objective values are histogrammed, the resulting sample probability density function can be used to estimate how close the objective at the declared optimal point is to the true value $z(\mathbf{x}^\star_{\text{IP}})$. Refinements to this scheme include examining adjacent lattice points to confirm the declared optimum or find a better one, and repeating the random sample over a smaller region centered on the declared optimum.

Explicit enumeration is hard to use at all for an unbounded feasible set, takes too much work if it includes all the lattice points, and might not find the right answer if it doesn't.

7.2 Implicit Enumeration

If all the components of $\mathbf{x}^\star_{\text{LP}}$ happen to be whole numbers, then $\mathbf{x}^\star_{\text{IP}} = \mathbf{x}^\star_{\text{LP}}$. Adding constraints to a minimization problem can never decrease its optimal value, so $z(\mathbf{x}^\star_{\text{IP}}) \geq z(\mathbf{x}^\star_{\text{LP}})$. By using these two facts it is possible to deduce that whole sets of lattice points cannot include $\mathbf{x}^\star_{\text{IP}}$.

To see how, we will solve the `bb1` problem (see §28.6.3) given below. The integer program is labeled IP, its linear programming relaxation is labeled LP, and $\mathbb{F}$ is the set of points satisfying the linear program's constraints.

$$\left.\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 - 3x_2 & = & z(\mathbf{x}) \\ \text{subject to} & \left.\begin{array}{rcl} -x_1 + x_2 & \leq & 2 \\ x_1 + x_2 & \leq & 6\frac{1}{2} \\ \mathbf{x} & \geq & \mathbf{0} \end{array}\right\}\mathbb{F} & & \\ & x_1 \text{ and } x_2 \text{ are integers} & & \end{array}\right\}\text{LP}\ \Bigg\}\text{IP}$$

The top graph on the right shows the solution of the linear programming relaxation LP

$$\underset{\mathbf{x}\in\mathbb{F}}{\text{minimize}}\ z(\mathbf{x})$$

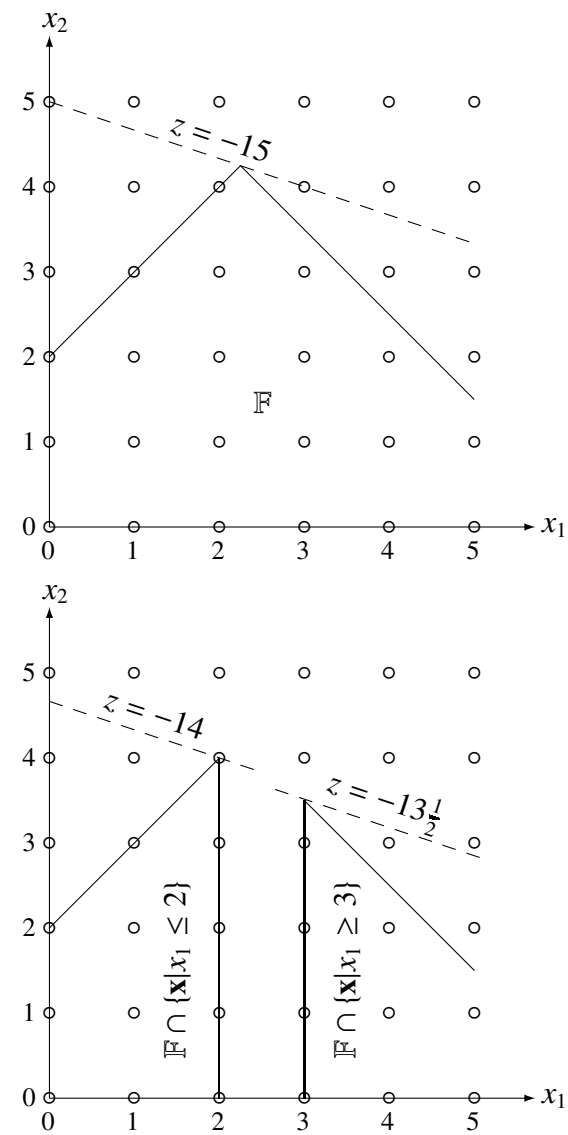
which is $\mathbf{x}^\star_{\text{LP}} = [2\frac{1}{4}, 4\frac{1}{4}]^\top$. In any solution to IP, x_1 cannot have a fractional part so it must satisfy either $x_1 \leq 2$ or $x_1 \geq 3$. In other words, the solution to IP must be in either $\mathbb{F} \cap \{\mathbf{x}|x_1 \leq 2\}$ or $\mathbb{F} \cap \{\mathbf{x}|x_1 \geq 3\}$. To find it we can examine both possibilities by **branching on** x_1 to form these two linear programs.

$$\underset{\mathbf{x}\in\mathbb{F},\, x_1\leq 2}{\text{minimize}}\ z(\mathbf{x}) \qquad\qquad \underset{\mathbf{x}\in\mathbb{F},\, x_1\geq 3}{\text{minimize}}\ z(\mathbf{x})$$

$$\mathbf{x}^\star = [2, 4]^\top \qquad\qquad \mathbf{x}^\star = [3, 3\tfrac{1}{2}]^\top$$

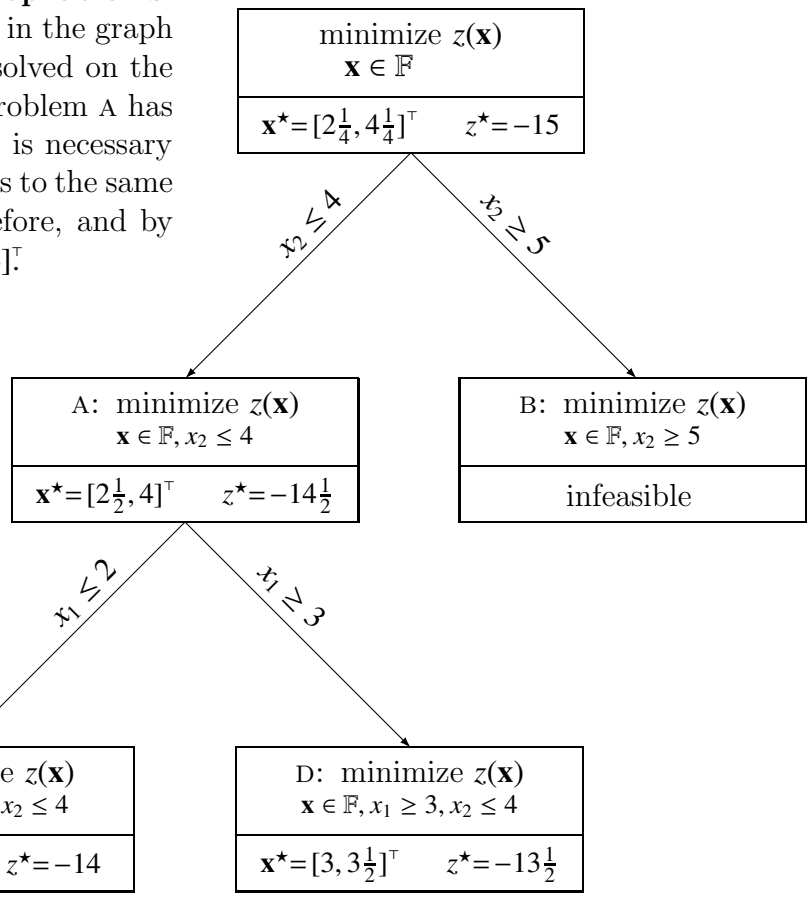
$$z^\star = -14 \qquad\qquad z^\star = -13\tfrac{1}{2}$$

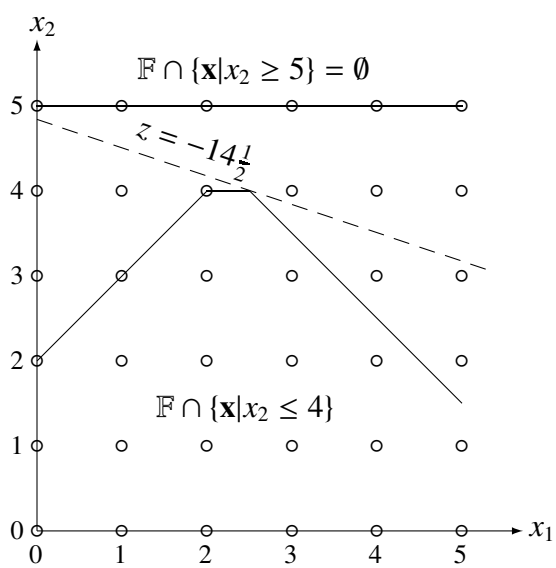
The solutions to these problems are shown in the bottom graph on the right. The left problem has its optimum at a point with integer components, so that is the best lattice point in $\mathbb{F} \cap \{x_1 \leq 2\}$. The optimal point for the right problem is not a lattice point, so the solution does not tell us what the best lattice point is in $\mathbb{F} \cap \{x_1 \geq 3\}$. However, since the objective of the right problem is worse than that of the left problem we know that the best lattice point in $\mathbb{F}\cap\{x_1 \geq 3\}$ is not as good as the one we found on the left. Therefore, it must be that $\mathbf{x}^\star_{\text{IP}} = [2, 4]^\top$.

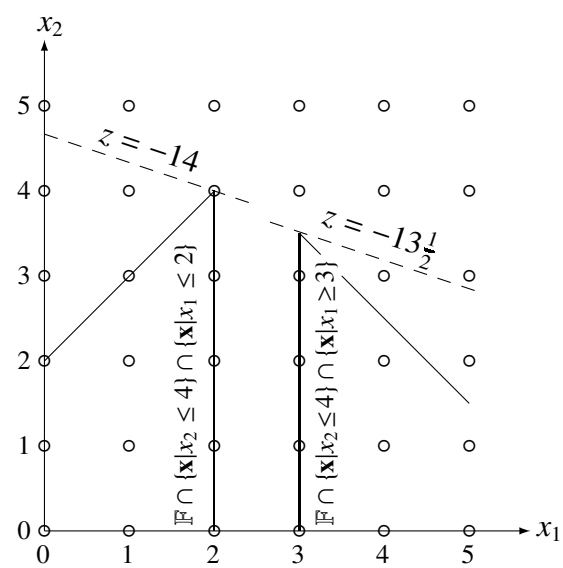


Adding constraints on x_2 instead produces this **branching diagram**, in which the original linear programming relaxation at the top is called the **master problem** and the branchings produce a tree of **subproblems**. Subproblems A and B are solved in the graph on the left below; C and D are solved on the right. Because the solution to problem A has a non-integer first component it is necessary to branch again, on x_1. This leads to the same subproblem solutions we got before, and by the same reasoning to $\mathbf{x}^\star_{\text{IP}} = [2, 4]^\top$.

We ruled out all of the many suboptimal lattice points without examining any of them, so the enumeration performed by this process is said to be **implicit**. The algorithm [64] [62, §4.3] [70, §8] is stated precisely in §7.3 on the next page.







7.3 Branch-and-Bound for Integer Programs

0. **initialize**
   - Construct the linear programming relaxation LP and solve it. If the solution to this master problem satisfies the integer constraints, it is optimal for IP; `STOP`.
   - Find an upper bound $\overline{z}$ on the objective, equal to its value at some **incumbent solution** that is feasible for IP; if no such point is known set $\overline{z} = +\infty$.

1. **branch**
   - Select a subproblem whose subset of $\mathbb{F}$ is **unfathomed**. On the first iteration this is the master problem; after that the bounding and fathoming steps must have been completed for both subproblems that resulted from a given branching before you branch again from either of them.
   - Choose a noninteger component of the subproblem solution, and construct two new subproblems. To one add a constraint to keep that variable no lower than the next higher integer; to the other add a constraint to keep that variable no higher than the next lower integer. Each new subproblem must also include all of the bound constraints inherited from earlier branchings.

2. **bound**
   Solve both new subproblems to obtain a lower bound $\underline{z}_p$ on the objective over the subset of $\mathbb{F}$ that is feasible for each of them.

3. **fathom**
   Exclude subproblem p (and thus its subset of $\mathbb{F}$) from further consideration if any of these conditions is satisfied:

   (a) the subproblem is infeasible, so its subset of $\mathbb{F}$ is empty

   (b) $\underline{z}_p \geq \overline{z}$ so some lattice point that is not in the subset is at least as good as every point that is in the subset

   (c) $\underline{z}_p < \overline{z}$ is attained at a lattice point in the subset

   In case (c),
   - declare the subproblem solution the incumbent solution to IP
   - let $\overline{z} = \underline{z}_p$
   - if unfathomed subsets remain `GO TO` 3 and check them against the new $\overline{z}$

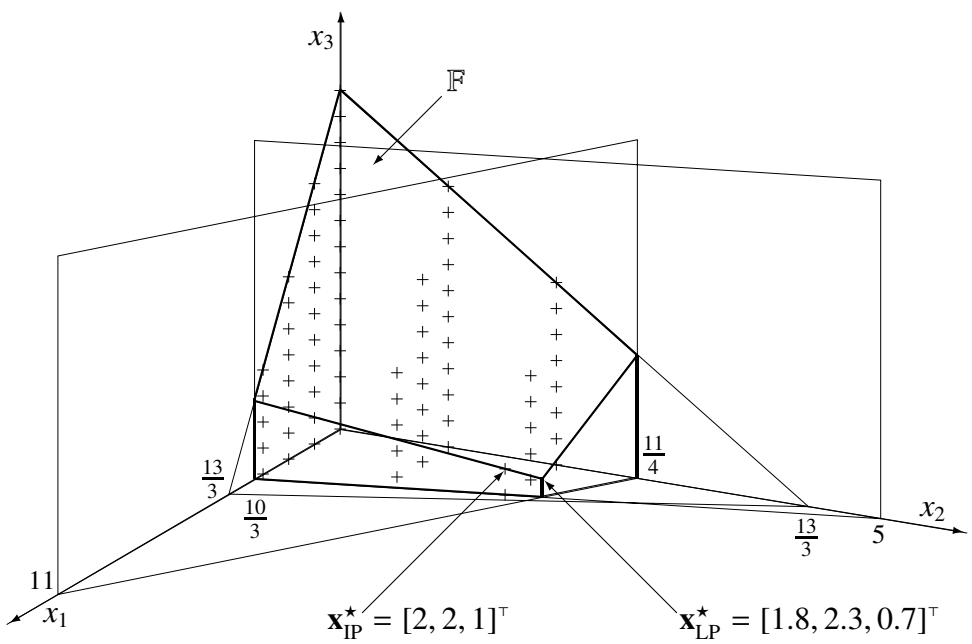
4. **test**
   If no unfathomed subsets remain, the incumbent solution is optimal for IP; `STOP`. Otherwise, `GO TO` 1.

The algorithm generates a binary tree in which the subproblems at the nodes differ only in the bounds on the variables. When we exclude from further consideration a subset of $\mathbb{F}$ that cannot contain $\mathbf{x}^\star_{\text{IP}}$ we say that the subset (and hence its node) is **fathomed**, because we have sounded its depths and either discovered a new incumbent solution or determined that even its best lattice point is not as good as the incumbent solution we already know.

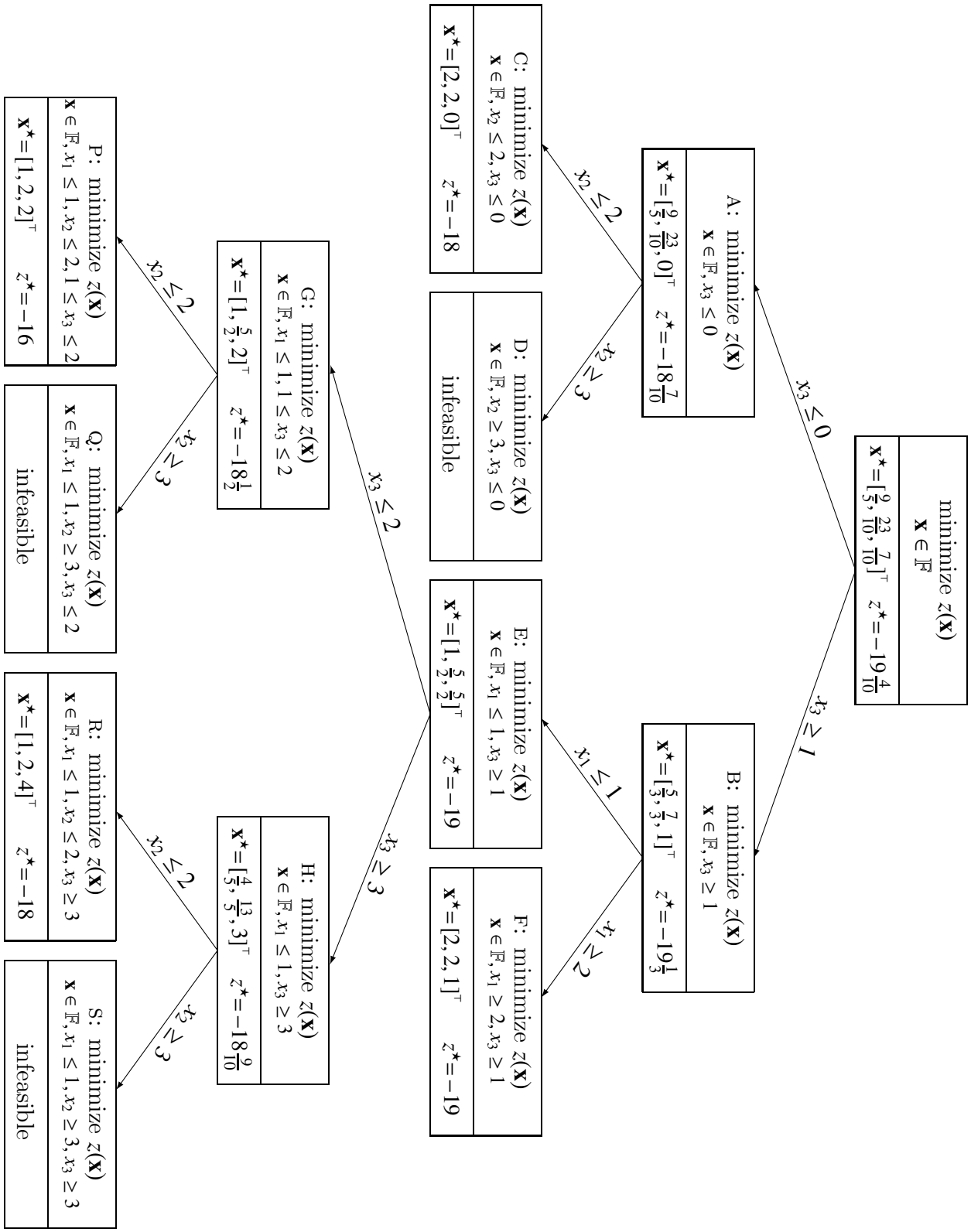
To illustrate the algorithm we will use it to solve this larger problem [62, Exercise 5.19.2], which I will call `bb2` (see §28.6.4).

$$\left.\begin{array}{lrcl}
\underset{\mathbf{x}\in\mathbb{Z}^3}{\text{minimize}} & -4x_1 - 5x_2 - x_3 & = & z \\
\text{subject to} & \left.\begin{array}{rcl} 3x_1 + 2x_2 & \leq & 10 \\ x_1 + 4x_2 & \leq & 11 \\ 3x_1 + 3x_2 + x_3 & \leq & 13 \\ \mathbf{x} & \geq & \mathbf{0} \end{array}\right\}\mathbb{F} & & \left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\}\text{LP} \\
 & x_1, x_2, \text{ and } x_3 & \text{are} & \text{integers} \qquad \left.\begin{array}{l} \\ \end{array}\right\}\text{IP}
\end{array}\right.$$

The picture below shows the constraint hyperplanes, the feasible set $\mathbb{F}$, and all of the lattice points + that are in $\mathbb{F}$. The optimal solution to LP is the indicated vertex and the optimal solution to IP happens to be the closest lattice point (recall from §7.1 that this does not always happen). For clarity `gnuplot` chose a different scaling for each axis.



The diagram on the next page shows the entire tree of subproblems that results from the branching decisions shown; different trees would result from picking other variables to branch on (see Exercise 7.10.16). In carrying out the branch-and-bound algorithm on this tree only part of the tree might actually be constructed.



The master problem's solution is not a lattice point so it cannot be $\mathbf{x}^\star_{\mathrm{IP}}$. The origin is a lattice point that is obviously in $\mathbb{F}$ so we can make $\mathbf{x} = [0, 0, 0]^\top$ the incumbent solution; the objective there is zero so we set $\overline{z} = 0$. I arbitrarily chose x_3 for the first branch, generating subproblems A and B. Then I used the `solve` command of the `pivot` program to solve each linear program. Neither subproblem solution satisfies any of the fathoming conditions of algorithm step 3. Whenever more than one subset of $\mathbb{F}$ remains unfathomed the algorithm allows us to select which subproblem to solve next, so what happens depends on the sequence of choices we make.

In the **breadth-first** strategy we generate all of the nodes at the current depth of the tree before any that are farther down. In this problem we branch first on subproblem A to generate subproblems C and D, and solve them. Subproblem D is infeasible, so we fathom that node of the branching diagram by condition (a). The solution to C is a lattice point having $\underline{z} = -18 < 0 = \overline{z}$, so fathoming condition (c) is satisfied. We declare $\mathbf{x} = [2, 2, 0]^\top$ to be the incumbent solution and let $\overline{z} = -18$. Next we branch on subproblem B to generate subproblems E and F, and solve them. The solution to F is a lattice point having $\underline{z} = -19$ so we update the incumbent solution to $\mathbf{x} = [2, 2, 1]^\top$ and let $\overline{z} = -19$. Node E has $\underline{z} = \overline{z}$ so it is fathomed by condition (b) and we never generate subproblem G, H, P, Q, R, or S. No subsets of $\mathbb{F}$ remain unfathomed, so the incumbent solution is optimal and $\mathbf{x}^\star_{\mathrm{IP}} = [2, 2, 1]^\top$.

In the **depth-first** strategy we extend the branching diagram as far down as possible before considering nodes to the left or right. In this problem we might pick subproblem B to branch from first, generating subproblems E and F. The solution to F is a lattice point, so we declare it the incumbent solution and let $\overline{z} = -19$. This updated value of $\overline{z}$ is lower than the optimal value for subproblem A, so we fathom that node and never generate subproblem C or D. Node E has $\underline{z} = \overline{z}$ so it is fathomed by condition (b) and we never generate subproblem G, H, P, Q, R, or S. No subsets of $\mathbb{F}$ remain unfathomed, so the incumbent solution is optimal and $\mathbf{x}^\star_{\mathrm{IP}} = [2, 2, 1]^\top$.

For this example the breadth-first strategy required the solution of 7 subproblems while the depth-first strategy required the solution of only 5, but that is just because we decided to start the depth-first solution by branching at node B rather than at node A. In practice [117, p60] the optimal solution often occurs deep in the tree, and then the depth-first strategy can have a bigger advantage over breadth-first. Our algorithm permits the selection of *any* unfathomed node to branch from next, so it is also possible to use a deliberate strategy that is neither breadth-first nor depth-first [3, p225], or even to make the selection at random.

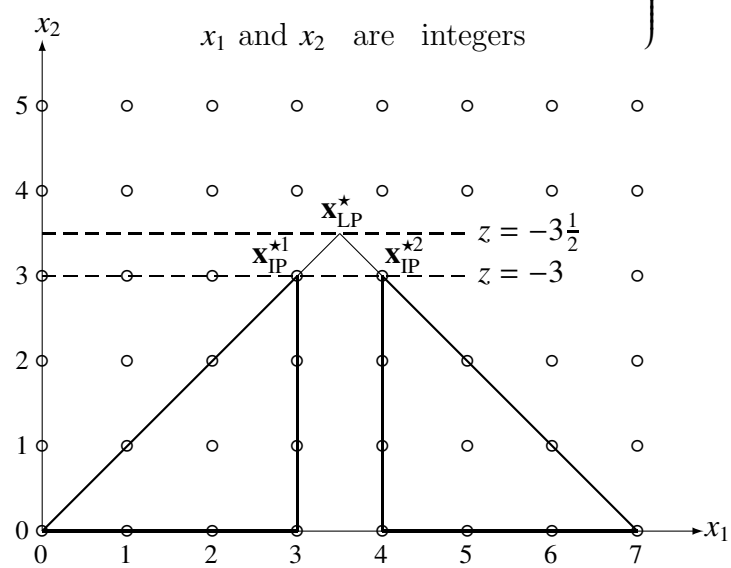
7.4 Multiple Optimal Points

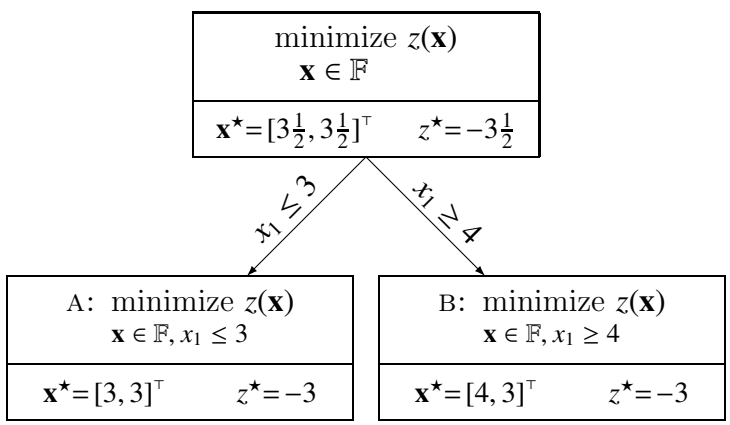
The problem at the top of the next page, which I will call `bb3` (see §28.6.5), has the *two* optimal points shown in its graphical solution. Both points are discovered by our branch-and-bound algorithm if we slightly modify its fathoming conditions (see Exercise 7.10.19) because each is the solution to a subproblem.

Following the steps of the algorithm as stated in §7.3 I solved the master problem and found that $\mathbf{x}^\star_{\text{LP}}$ is not integer feasible. The origin is a lattice point in $\mathbb{F}$ so I declared it to be the incumbent solution and set $\overline{z} = 0$. Then I branched on x_1 and solved the two resulting subproblems. Subproblem A has $\underline{z} = -3 < \overline{z} = 0$ and is therefore fathomed by condition (c), so I updated the incumbent solution to $\mathbf{x}^{\star 1}_{\text{IP}} = [3, 3]^\top$ and let $\overline{z} = -3$. Then I considered subproblem B, which has $\underline{z} = -3 = \overline{z}$ and is therefore fathomed by condition (b). But by then its integer-feasible optimal point $\mathbf{x}^{\star 2}_{\text{IP}} = [4, 3]^\top$ had already been revealed.

$$\left.\begin{array}{lrcl}
\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -x_2 & = & z(\mathbf{x}) \\
\text{subject to} & \left.\begin{array}{rcl} -x_1 + x_2 & \leq & 0 \\ x_1 + x_2 & \leq & 7 \\ \mathbf{x} & \geq & \mathbf{0} \end{array}\right\}\mathbb{F} & & \\
\end{array}\right\}\text{LP}$$

$$\left.\begin{array}{l} \text{LP} \\ x_1 \text{ and } x_2 \text{ are integers} \end{array}\right\}\text{IP}$$

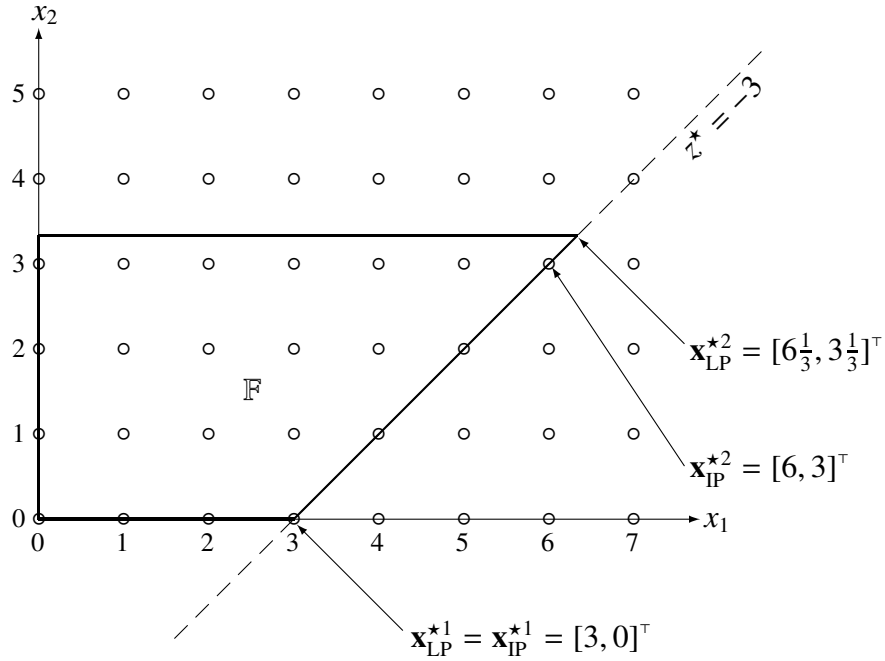




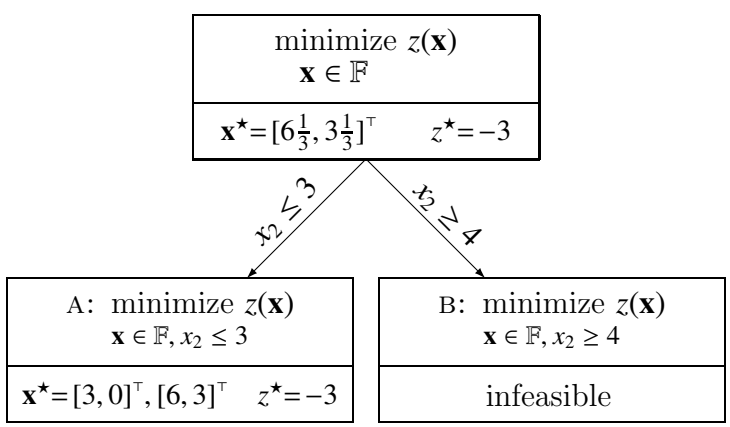
If a *subproblem* has multiple optimal solutions, the situation can be somewhat more complicated. This problem, which I will call `bb4` (see §28.6.6) is solved on the next page.

$$\left.\begin{array}{lrcl}
\underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 + x_2 & = & z(\mathbf{x}) \\
\text{subject to} & \left.\begin{array}{rcl} x_1 - x_2 & \leq & 3 \\ x_2 & \leq & 3\frac{1}{3} \\ \mathbf{x} & \geq & \mathbf{0} \end{array}\right\}\mathbb{F} & & \\
\end{array}\right\}\text{LP}$$

$$\left.\begin{array}{l} \text{LP} \\ x_1 \text{ and } x_2 \text{ are integers} \end{array}\right\}\text{IP}$$



The relaxation LP has two optima at vertices of $\mathbb{F}$, $\mathbf{x}^{\star 1}_{\mathrm{LP}} = [3, 0]^\top$ and $\mathbf{x}^{\star 2}_{\mathrm{LP}} = [6\frac{1}{3}, 3\frac{1}{3}]^\top$. One of these is a lattice point, and we can see from the picture that others lurk in the optimal edge. To search for them we might branch on a non-integer component of $\mathbf{x}^{\star 2}_{\mathrm{LP}}$ like this.



Subproblem A has two vertex optima, at $[3, 0]^\top$ and $[6, 3]^\top$, so branch-and-bound (if it does not give up too soon) can discover $\mathbf{x}^{\star 2}_{\mathrm{IP}}$. Unfortunately, the integer optima at $[4, 1]^\top$ and $[5, 2]^\top$ are beyond its view. To be sure of finding all of the optimal points when solving an integer program by branch-and-bound we must, whenever a subproblem has multiple optima, find all of the lattice points in its optimal set (which is in general of higher dimension than a line). The details of such a hybrid algorithm are beyond the scope of this introduction.

7.5 Zero-One Programs

Most of the work in the branch-and-bound algorithm of §7.3 is in step 2, when we solve both subproblems to get a lower bound $\underline{z}$ on the objective over each new subset of $\mathbb{F}$. For problems like the ones we have studied, in which the variables are nonnegative integers of arbitrary magnitude, that usually requires two invocations of the simplex method or of an interior-point method for linear programming.

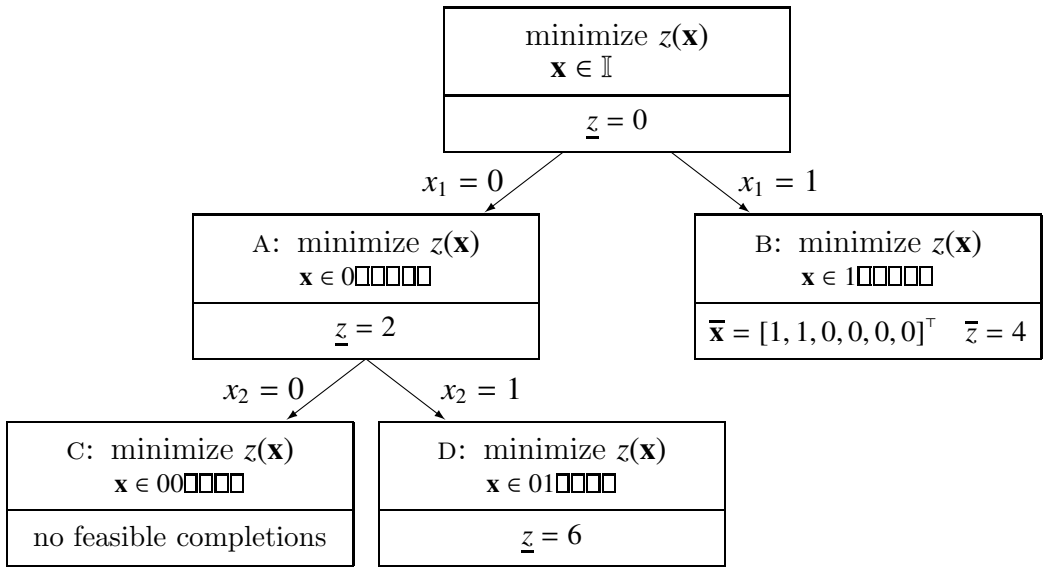
If instead each x_j is restricted to be 0 or 1, we can think of the integer program in an entirely different way that makes it very easy to find lower bounds on the objective. In the problem below [3, §8.4] which I will call `bb5` (see §28.6.7), $\mathbb{I}$ denotes the set of $2^6 = 64$ vectors $[x_1, x_2, x_3, x_4, x_5, x_6]^\top$ in which each x_j is either 0 or 1.

$$\left.\begin{array}{lrcr} \underset{\mathbf{x}\in\mathbb{Z}^6}{\text{minimize}} & 2x_1 + 2x_2 + 4x_3 + 7x_4 + 8x_5 + 9x_6 & = & z(\mathbf{x}) \\ \text{subject to} & \left.\begin{array}{rcr} -5x_1 + 3x_2 - 2x_3 + 3x_4 + x_5 - 2x_6 & \leq & 5 \\ x_1 - 2x_3 - x_4 - 3x_5 + 3x_6 & \leq & 1 \\ -x_1 - 2x_2 + x_3 - x_4 + 5x_5 + x_6 & \leq & -3 \end{array}\right\}\mathbb{F} & & \\ & x_1, x_2, x_3, x_4, x_5, x_6 \in \{0, 1\} \quad \}\mathbb{I} & & \end{array}\right\}\text{IP}$$

Because all of the coefficients in the objective function are nonnegative and each $x_j \in \{0, 1\}$ the lowest value that $z(\mathbf{x})$ could possibly have is $\underline{z} = 0$, at $\underline{\mathbf{x}} = [0, 0, 0, 0, 0, 0]^\top$. If that point were feasible for the inequalities then it would be optimal. Unfortunately it violates the third constraint because $0 \not\leq -3$, but that does not rule out the possibility that other lattice points are in $\mathbb{F}$.

If there is an optimal point it must have either $x_1 = 0$ or $x_1 = 1$. A systematic procedure for investigating these alternatives is described by the branching diagram on the next page, which is reminiscent of those we have drawn before but different from them in important ways. Now to form a subproblem, rather than ignoring the integer constraints and minimizing the objective over a subset of $\mathbb{F}$, we ignore the inequality constraints and minimize the objective over a subset of $\mathbb{I}$. Above we minimized the objective over all of $\mathbb{I}$ and found for the master problem that $\underline{\mathbf{x}} = [0, 0, 0, 0, 0, 0]^\top$, which is not feasible for $\mathbb{F}$. This leads us to branch on x_1, generating subproblems A and B.

In subproblem B the minimization is over those elements of $\mathbb{I}$ having $x_1 = 1$, and the notation $\mathbf{x} \in 1\square\square\square\square\square$ means that $\mathbf{x}$ belongs to the set of binary vectors having the form $[1, x_2, x_3, x_4, x_5, x_6]^\top$. When the value of x_1 is fixed at 0 or at 1 it is called a **partial solution**, and the $2^5 = 32$ possible vectors $[x_2, x_3, x_4, x_5, x_6]^\top$ are called its **completions**. Thus each trial solution consists of a partial solution and one of its completions. The trial solution $[1, 0, 0, 0, 0, 0]^\top$ violates the third constraint, so the lowest value that $z(\mathbf{x})$ could have over this subset of $\mathbb{I}$ is $\underline{z} = 4$ at $\underline{\mathbf{x}} = [1, 1, 0, 0, 0, 0]^\top$. That point happens to be in $\mathbb{F}$ so it becomes the incumbent solution $\overline{\mathbf{x}}$, and now we know that we can make $z(\mathbf{x})$ at least as low as $\overline{z} = 4$. We have found the best point in this subset of $\mathbb{I}$, so the node is fathomed.



In subproblem A the minimization is over those elements of $\mathbb{I}$ having $x_1 = 0$, or $\mathbf{x} \in 0\square\square\square\square\square$. We already found that the **zero completion**, yielding $\mathbf{x} = [0, 0, 0, 0, 0, 0]^\top$, is infeasible, so the lowest value that $z(\mathbf{x})$ can have over this subset of $\mathbb{I}$ is $\underline{z} = 2$, at $\underline{\mathbf{x}} = [0, 1, 0, 0, 0, 0]^\top$. Unfortunately this $\underline{\mathbf{x}}$ is also infeasible, but because $\underline{z} = 2 < 4 = \overline{z}$ the subset still might contain a lattice point better than the incumbent solution. To search for one I branched on x_2, generating subproblems C and D.

In subproblem C, $x_1 = x_2 = 0$ so the third constraint becomes

$$x_3 - x_4 + 5x_5 + x_6 \leq -3$$

and its left-hand side can never be less than -1. Thus the partial solution $\mathbf{x} \in 00\square\square\square\square$ has *no* feasible completions, and the node is fathomed.

In subproblem D, the zero completion yields $\mathbf{x} = [0, 1, 0, 0, 0, 0]^\top$ but that violates the third constraint, so the lowest $z(\mathbf{x})$ can be is $\underline{z} = 6$, at $\underline{\mathbf{x}} = [0, 1, 1, 0, 0, 0]^\top$. But the lattice point we found at node B has an objective lower than 6, so this node is fathomed because $\underline{z} > \overline{z}$. No unfathomed nodes remain, so the incumbent solution is optimal and $\mathbf{x}^\star = [1, 1, 0, 0, 0, 0]^\top$.

This procedure depends on the objective function cost coefficients being nonnegative and arranged in nondescending order, but that can always be achieved by using a substitution of variables. For example,

$$z(\mathbf{y}) = -10y_1 + 2y_2 - 3y_3 \longrightarrow \begin{bmatrix} y_1 & = & 1 - x_3 \\ y_2 & = & x_1 \\ y_3 & = & 1 - x_2 \end{bmatrix} \longrightarrow z(\mathbf{x}) = 2x_1 + 3x_2 + 10x_3 - 13.$$

The algorithm illustrated above is stated precisely in §7.5.1 on the next page. The enumerations performed by this algorithm and by the §7.3 branch-and-bound algorithm for integer programs are both implicit rather than explicit, but some authors [62, §4.5] [151, §13.7] use the term implicit enumeration to refer exclusively to the zero-one algorithm.

7.5.1 Branch-and-Bound for Zero-One Programs

0. **initialize**
   - Reformulate, if necessary, to make the objective function coefficients of the master problem nonnegative and nondecreasing.
   - If $\mathbf{x} = \mathbf{0} \in \mathbb{F}$ then it is optimal for IP; `STOP`.
   - Set $\overline{z}$ to the sum of the objective coefficients.
   - Set $\underline{\mathbf{x}} = \mathbf{0}$ and $\underline{z} = z(\underline{\mathbf{x}}) = 0$.

1. **branch**
   - Select a subproblem whose subset of $\mathbb{I}$ is unfathomed. On the first iteration this is the master problem; after that the bounding and fathoming steps must have been completed for both subproblems that resulted from a given branching before you branch again from either of them.
   - Construct two new subproblems by assigning 0 and 1 to the first variable that was not yet fixed in the previous partial solution. Each of the two new partial solutions is thus an extension by one variable of the previous partial solution, in which the earlier assignments of variables to be 0 or 1 are retained.

2. **bound**
   For each new subproblem p obtain a lower bound on the objective value over that subset of $\mathbb{I}$, by setting $\underline{\mathbf{x}}$ equal to the previous partial solution completed by $[1, 0 \ldots 0]^\top$ and $\underline{z}_p = z(\underline{\mathbf{x}})$.

3. **fathom**
   Exclude subproblem p (and thus its subset of $\mathbb{I}$) from further consideration if any of these conditions is satisfied:

   (a) there are no feasible completions in the subset

   (b) $\underline{z}_p \geq \overline{z}$ so some lattice point that is not in the subset is at least as good as every point in the subset

   (c) $\underline{z}_p < \overline{z}$ is attained at a point in the subset that is feasible for $\mathbb{F}$

   In case (c),

   - declare $\underline{\mathbf{x}}$ the incumbent solution to IP
   - let $\overline{z} = \underline{z}_p$
   - if unfathomed subsets remain `GO TO 3` and check them against the new $\overline{z}$

4. **test**
   If no unfathomed subsets remain, the incumbent solution is optimal for IP; `STOP`. Otherwise, `GO TO 1`.

In reformulating an ordinary linear program to have nonnegative and nondecreasing objective coefficients we might use variable substitutions of the form $y_1 = -x_3$, but in a zero-one program having $y_1 \in \{0, 1\}$ that would make $x_3 \in \{0, -1\}$. The algorithm requires that each $x_j \in \{0, 1\}$, so it is important to change the signs of negative objective coefficients by using variable substitutions of the form $y_1 = 1 - x_3$ as illustrated earlier. That way $x_3 = 0$ makes $y_1 = 1$ and $x_3 = 1$ makes $y_1 = 0$.

Because the objective coefficients are nonnegative $\mathbf{x} = \mathbf{0}$ yields the lowest possible value of $z(\mathbf{x})$, so that is the trial solution we try first; only if it does not satisfy the inequalities must we branch and bound. That process is based on the upper bound $\overline{z}$, which is highest when $\mathbf{x} = [1, 1 \ldots 1]^\top$ and is then just the sum of the objective coefficients.

The partial solution that constrains each subproblem defines the subset of $\mathbb{I}$ over which its minimization is performed, so each partial solution must inherit the variable assignments that were made in the branching decisions that preceded its subproblem in the tree.

In the bounding step the zero completion of the previous partial solution is always infeasible, because otherwise the node that is parent to this one would have been fathomed by condition (c) and there would have been no branch. Because the objective has its coefficients in nondecreasing order its lower bound can always be found by using the previous partial solution completed by $[1, 0 \ldots 0]^\top$. In [3, p233-238] this is referred to as **looking ahead**.

7.5.2 Checking Feasible Completions

Most of the work in zero-one branch-and-bound is in step 3(a), where we are obliged to say if the partial solution constraining subproblem p is certain to have no feasible completions.

One way to answer this question would be to search for a completion that satisfies all of the inequalities. We can generate the possible completions one at a time and for each evaluate all of the constraints at the corresponding trial solution. As soon as we find a completion that is feasible we can stop searching; there is at least one feasible completion, which means that the fathoming condition fails. If we test all possible completions without finding a feasible one, then the fathoming condition succeeds.

Another way to answer the question would be to search for an inequality that is violated by all of the possible completions. If we find one then we can say for sure that there are no feasible completions, and the fathoming condition succeeds. In our solution of `bb5` subproblem C had the partial solution $\mathbf{x} \in 00\square\square\square\square$, making the third constraint look like this.

$$-1(0) - 2(0) + x_3 - x_4 + 5x_5 + x_6 \leq -3$$

Because $x_j \in \{0, 1\}$ the left-hand side has its minimum value of -1 when $\mathbf{x} = [0, 0, 0, 1, 0, 0]^\top$, so above I argued that no completion is feasible and the fathoming condition succeeds.

Even if no single constraint is violated by all possible completions, it is of course still possible that every possible completion violates some constraint. In subproblem D of `bb5`,

the partial solution $\mathbf{x} \in 01\square\square\square\square$ makes the constraints look like this.

$$\begin{array}{rcl} -5(0) + 3(1) - 2x_3 + 3x_4 + x_5 - 2x_6 & \leq & 5 \\ 1(0) - 2(1) \qquad\qquad - x_4 - 3x_5 + 3x_6 & \leq & 1 \\ -1(0) - 2(1) + x_3 - x_4 + 5x_5 + x_6 & \leq & -3 \end{array}$$

The only completion that satisfies the third constraint is the one that makes the trial solution $\mathbf{x} = [0, 1, 0, 1, 0, 0]^\top$ but that point violates the first constraint. Thus, although each constraint is feasible for some completion of $\mathbf{x} = 01\square\square\square\square$, no completion satisfies all of the constraints. In solving `bb5` I did not fathom node D by condition (a), but there are in fact no feasible completions in that subset of $\mathbb{I}$. If in carrying out the steps of the zero-one algorithm we refrain from fathoming some node because no constraint is violated by every completion, but the subset contains no feasible completions, that fact will be discovered at a later iteration. If in the example node D had not been fathomed for a different reason before that could happen, we would have branched from it and found the infeasibility.

Because of its simplicity you might prefer the first strategy described above, but searching every node for a completion that satisfies all of the inequalities can be even more expensive than solving the master problem by exhaustive enumeration. The second strategy is much less work, as illustrated by its implementation in the `fathoma.m` routine listed below. The inputs to this routine are a matrix `A` of constraint coefficients, a vector `b` of right-hand-side values, and a vector `x` containing a partial solution followed by elements set equal to -1. These special values correspond to the boxes $\square$ that we have used to represent the possible completions of a partial solution. The Octave session on the next page shows how the routine can be used to decide whether fathoming condition (a) is satisfied by the partial solution at each of the nodes in the `bb5` branching diagram. Only for node C does the routine find that every completion violates some constraint, and its return value `row=3` shows it is the third constraint that is always violated (as we found above).

```
function row=fathoma(A,b,x)
% return index of first constraint in Ax <= b violated by all completions

  m=size(A,1);          % find out how many rows are in A and b
  row=0;                % assume no such constraint will be found
  ip=(x' ==  1);        % indices in partial solution where x(j)=1
  ic=(x' == -1);        % indices in trial solution to be completed
  for i=1:m             % check the constraints one at a time
      ap=sum(A(i,ip));  % value of the partial solution
      im=(A(i,:) < 0);  % indices where coefficient A(i,j)<0
      id=bitand(ic,im);          % indices in completion where A(i,j)<0
      ac=sum(A(i,logical(id)));  % value of most negative completion
      ax=ap+ac;                  % value of constraint
      if(ax > b(i));             % if inequality is violated
         row=i;         % get the number of the offending row
         return         % and return it
      end
  end
end
```



```
octave:1> A=[-5, 3,-2, 3, 1,-2;
>             1, 0,-2,-1,-3, 3;
>            -1,-2, 1,-1, 5, 1];
octave:2> b=[5;1;-3];
octave:3> % master problem
octave:3> x=[-1;-1;-1;-1;-1;-1];
octave:4> row=fathoma(A,b,x)
row = 0
octave:5> % node A
octave:5> x=[0;-1;-1;-1;-1;-1];
octave:6> row=fathoma(A,b,x)
row = 0
octave:7> % node B
octave:7> x=[1;-1;-1;-1;-1;-1];
octave:8> row=fathoma(A,b,x)
row = 0
octave:9> % node C
octave:9> x=[0;0;-1;-1;-1;-1]
octave:10> row=fathoma(A,b,x)
row =  3
octave:11> % node D
octave:11> x=[0;1;-1;-1;-1;-1];
octave:12> row=fathoma(A,b,x)
row = 0
```

If in a trial solution $\mathbf{x} \in \mathbb{Z}^n$ the first s elements are a partial solution and the final $n-s$ are a completion, then the left-hand side of constraint i can be found as the sum of two terms.

$$A_i\mathbf{x} = \underbrace{\sum_{j=1}^{s} a_{ij}x_j}_{\texttt{ap}} + \underbrace{\sum_{j=s+1}^{n} a_{ij}x_j}_{\texttt{ac}}$$

The value of `ap` is fixed by the partial solution, but the value of `ac` depends on which of the possible completions is used. The completion yielding the lowest value of `ac` will have $x_j = 1$ where $a_{ij} < 0$ and $x_j = 0$ elsewhere. If $A_i\mathbf{x} > b_i$ for this completion, then no completion is feasible for constraint i so no completion is feasible for $\mathbb{F}$.

The routine [4] finds out how many inequalities there are and [5] prepares to return a zero result in case a constraint is not found for which the partial solution has no feasible completions. Then [6] it uses a MATLAB construct [50, §4.6] to make `ip` a row vector of logical values in which `ip(j)=T` if $x_j = 1$ or `ip(j)=F` otherwise. In a similar way [7] it makes `ic` a row vector of logical values in which `ic(j)=T` if $x_j = -1$ or `ic(j)=F` otherwise. Next it enters a loop [8-18] over the constraints. For constraint `i` it first [1] computes `ap` as the sum of those constraint coefficients in row `i` corresponding to the elements of the partial solution that are 1. Then [10] it uses the MATLAB construct to make `im` a row vector of logical values in which `im(j)=T` if $a_{ij} < 0$ or `im(j)=F` otherwise, and [11] makes `id(j)=T` if x_j is an element of the completion *and* $a_{ij} < 0$. Then [12] it computes `ac` as the sum of those constraint coefficients in row `i`. Finally [13] it finds `ax`, the lowest possible left-hand-side of constraint `i`, and [14-17] compares it to the right-hand side of constraint `i`. If the inequality is violated it [15-16] returns the index of the constraint. If the loop completes without finding a violated inequality, the routine returns [5] `row=0`.

7.6 Integer Programming Formulations

Often an integer program is just a linear program to which we have appended the restriction that the variables have integer values, such as when we required that Sarah's brewery produce only whole kegs of beer. But the same discontinuities that make integer programs hard to solve also permit the formulation of models [3, §8.6] [79, §18.5] [151, §13.2] that select from among discrete alternatives or enforce logical conditions.

7.6.1 Techniques

Changing to zero-one variables. Sometimes it is easier to solve an integer program with bounded variables if it is written as a zero-one program. To see how this is possible recall the `bb1` problem, which is reproduced below.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 - 3x_2 & = & z(\mathbf{x}) \\ \text{subject to} & -x_1 + \ \ x_2 & \leq & 2 \\ & x_1 + \ \ x_2 & \leq & 6\tfrac{1}{2} \\ & \mathbf{x} & \geq & \mathbf{0} \\ & x_1 \text{ and } x_2 & \text{are} & \text{integers} \end{array}$$

binary	decimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

We can see from the second inequality that $x_1 \in \{0, 1, 2, 3, 4, 5, 6\}$ and $x_2 \in \{0, 1, 2, 3, 4, 5, 6\}$. With 3-bit binary numbers we can count up to 7, as shown on the right, so we could make the substitutions

$$\begin{aligned} x_1 &= u_1 + 2u_2 + 4u_3 \\ x_2 &= v_1 + 2v_2 + 4v_3 \end{aligned}$$

where $u_j \in \{0, 1\}$ and $v_j \in \{0, 1\}$ to rewrite the problem as this zero-one program.

$$\begin{array}{lrcl} \underset{\mathbf{u}\in\mathbb{Z}^3\ \mathbf{v}\in\mathbb{Z}^3}{\text{minimize}} & -u_1 - 2u_2 - 4u_3 - 3v_1 - 6v_2 - 12v_3 & = & z(\mathbf{u}, \mathbf{v}) \\ \text{subject to} & -u_1 - 2u_2 - 4u_3 + v_1 + 2v_2 + 4v_3 & \leq & 2 \\ & u_1 + 2u_2 + 4u_3 + v_1 + 2v_2 + 4v_3 & \leq & 6\tfrac{1}{2} \\ & u_j \in \{0, 1\} \text{ and } v_j \in \{0, 1\} & & \end{array}$$

Selecting from a list. Sometimes what makes a (linear or nonlinear) optimization problem into an integer program is that one or more real variables can take on only certain values. For example, optimizing the design of an electronic circuit might involve choosing the best value for a resistor from a list of standard values. To ensure that a variable r takes on one of the values in the vector $R = [2.2, 2.7, 3.3, 3.9, 4.7, 5.6, 6.8, 8.2]^\top$ we could introduce zero-one variables $y_1 \ldots y_8$ and enforce these constraints.

$$r = \sum_{j=1}^{8} y_j R_j \qquad \text{and} \qquad \sum_{j=1}^{8} y_j = 1$$

The right constraint ensures that exactly one of the y_j will be 1, and then the left constraint selects that single element of R for the value of r.

Enforcing logical conditions. Many optimization problems involve a selection from discrete alternative courses of action. If the choice whether or not to pursue each alternative is represented by the value of a zero-one variable,

$$x_j = \begin{cases} 0 & \text{reject alternative } j \\ 1 & \text{accept alternative } j \end{cases}$$

then constraints like these can be imposed to model relationships between the actions.

$x_1 = 1$	alternative 1 must be chosen
$x_1 + x_2 = 1$	exactly one of the two alternatives must be chosen
$x_1 + x_2 \geq 1$	at least one of the two alternatives must be chosen
$x_1 + x_2 \leq 1$	at most one of the two alternatives can be chosen
$x_1 + \cdots + x_p \geq k$	at least k of the p alternatives must be chosen
$x_1 \leq x_2$	alternative 1 can be chosen only if alternative 2 is also chosen
$(1 - x_1) \leq (1 - x_2)$	alternative 1 must be chosen if alternative 2 is chosen

Switching constraints on or off. Above we noticed that the second constraint of the `bb1` problem ensures $x_1 \leq 6$ and $x_2 \leq 6$. That means that the first constraint function

$$f_1(x) = -x_1 + x_2$$

takes on its highest value of +6 when $\mathbf{x} = [0, 6]^\top$ so $f_1(x) \leq 6$ would always be satisfied. If we introduce a zero-one variable y and rewrite the first constraint as

$$-x_1 + x_2 \leq 2 + 4y$$

then when $y = 0$ it is the original constraint but when $y = 1$ it becomes $-x_1 + x_2 \leq 6$ and is always satisfied. Thus, making $y = 1$ effectively removes this constraint from the problem.

If several inequalities have switches of this sort then relationships between them can be imposed by enforcing logical conditions on the y_j as described above.

7.6.2 Applications

In §6.5.3 we formulated the assignment, shortest-path, and traveling salesman problems as integer programs. Three other integer programming models are also encountered in practice often enough to be instantly recognizable and therefore known by name.

The knapsack problem. Jacob, age 15, had a terrible fight with his older brother and has decided to run away from home. Unfortunately, the n possessions he would like to bring weigh more than the W pounds he can carry. If item j has value v_j and weight w_j pounds, which items should he choose to maximize their total value without making his backpack too heavy? Having read this Chapter, he identifies the decision variables

$$x_j = \begin{cases} 0 & \text{if item } j \text{ is left at home} \\ 1 & \text{if item } j \text{ is brought along} \end{cases}$$

and states the problem like this.

$$\begin{array}{rrcll} \underset{\mathbf{x}\in\mathbb{Z}^n}{\text{maximize}} & \mathbf{v}^\top\mathbf{x} & = & z(\mathbf{x}) & \\ \text{subject to} & \mathbf{w}^\top\mathbf{x} & \leq & W & \\ & x_j & \in & \{0,1\} & j = 1\ldots n \end{array}$$

The research literature discusses many variations on this problem, the most famous of which involve cutting smaller pieces from stock sizes of sheet metal, fabric, or other materials.

The capital budgeting problem. A large corporation has m different kinds of resources (such as cash, land, equipment, and workers) at its disposal and contemplates deploying them to some or all of n possible new projects (such as buying back stock, building new factories, and introducing new products). Project j is expected to generate a revenue r_j, resource i is available in quantity b_i, and the amount of resource i needed for project j is a_{ij}. Which projects should be undertaken? The question suggests these decision variables

$$x_j = \begin{cases} 0 & \text{if project } j \text{ is rejected} \\ 1 & \text{if project } j \text{ is undertaken} \end{cases}$$

and they lead to this formulation.

$$\begin{array}{rrcll} \underset{\mathbf{x}\in\mathbb{Z}^n}{\text{maximize}} & \mathbf{r}^\top\mathbf{x} & = & z(\mathbf{x}) & \\ \text{subject to} & \mathbf{A}\mathbf{x} & \leq & \mathbf{b} & \\ & x_j & \in & \{0,1\} & j = 1\ldots n \end{array}$$

This is a generalization of the knapsack problem from one resource (weight) to several.

The facility location problem. An international aid organization plans to deliver relief supplies to n established refugee camps, by shipping from warehouse tents that it will erect in places chosen from m possible locations. If t_i is the cost of erecting a tent at site i, d_j is the demand at camp j, and c_{ij} is the per-unit shipping cost from site i to camp j, which sites should get a tent and how much should each site ship to each camp? Now there are both zero-one and real decision variables.

$$y_i = \begin{cases} 0 & \text{if site } i \text{ is rejected} \\ 1 & \text{if site } i \text{ gets a tent} \end{cases} \qquad x_{ij} = \text{shipment from site } i \text{ to camp } j$$

The aid organization, always strapped for funds, seeks to minimize the total cost of the operation.

$$z(\mathbf{x}, \mathbf{y}) = \underbrace{\sum_{i=1}^{m} t_i y_i}_{\text{tents}} + \underbrace{\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}}_{\text{shipments}}$$

The shipments must meet the demands,

$$\sum_{i=1}^{m} x_{ij} = d_j \quad j = 1 \ldots n$$

but they are further constrained because a site without a tent can ship nothing. A site with a tent would never ship more than the total demand of all the camps, so we can summarize the two possibilities like this

$$\sum_{j=1}^{n} x_{ij} \leq \begin{cases} 0 & y_i = 0 \\ \sum_{j=1}^{n} d_j & y_i = 1 \end{cases} \qquad i = 1 \ldots m$$

or by the linear constraints

$$\sum_{j=1}^{n} x_{ij} \leq y_i \sum_{j=1}^{n} d_j \qquad i = 1 \ldots m.$$

If the warehouses, once constructed, are used repeatedly for periodic shipments to the camps, then the demands and hence the optimal shipping schedule might change from period to period. If shipments continue far into the future it might be realistic to discount their costs to present value. The transportation network connecting the warehouses to the camps might have missing links or capacity constraints, and the c_{ij} might change over time. Thus the basic facility location model can be complicated in various ways [151, §13.2].

7.7 Solving Integer Programs

Integer programming is a vast subject that we have so far barely glimpsed, but enough space remains in this introduction only to touch on some practical considerations that arise in solving real problems.

7.7.1 Mixed-Integer Programs

A problem having both integer and real variables, such as the facility location problem of §7.6.2, is called a **mixed-integer program**. To solve it we could use the branch-and-bound algorithm of §7.3 (even though the integer variables are zero-one) but branch only on the

integer variables. The linear programming relaxations involve all of the variables, so the subproblem whose solution yields $\mathbf{y}^\star$ also yields $\mathbf{x}^\star$. Algorithms have also been devised [62, §4.10] specifically for solving mixed-zero-one programs.

7.7.2 Other Methods

The branch-and-bound algorithms that we have studied are easy to understand and they find optimal points exactly, but they are not always fastest. Other ways of solving integer programs include **cutting-plane** methods [62, §5] [151, §13.4], **branch-and-cut** methods [113], **Lagrangian relaxation** [58], variations of the simplex algorithm that produce integer solutions [71, §3], **simulated annealing** [132, §10.9] and other approximate heuristics [62, §9] [74, §8-3,8-4], and dynamic programming (see §7.8).

7.7.3 Integer Programming Software

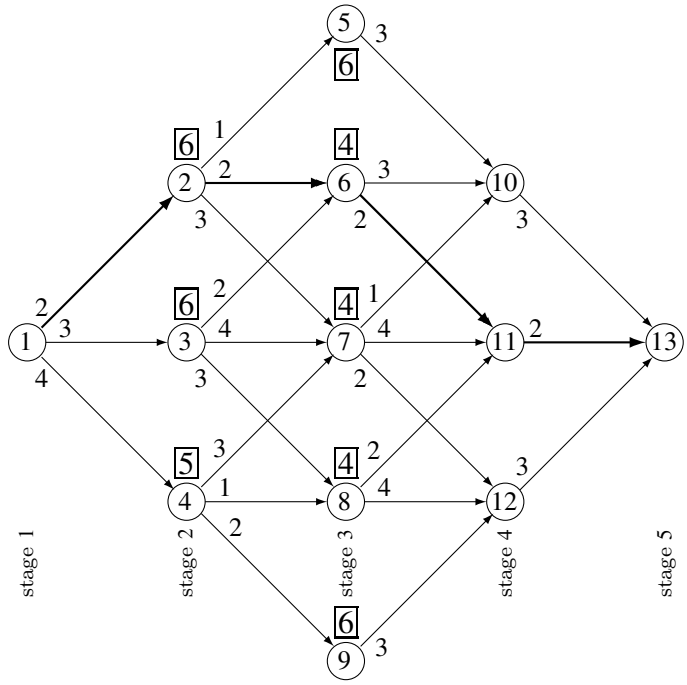
A computer program that implements either of the algorithms we have studied must somehow store the branching tree. The representation and manipulation of trees is a fundamental topic in data structures [94, §2.3] [83, §5] but it is beyond the modest programming experience that I have assumed readers of this book will have (see §0.2.3). In §6.5.1 we also encountered a tree, and there also I was forced to stop short of showing you MATLAB code for the algorithm under discussion. But in case you someday write your own code for solving integer programs I can pass on the observation [117, p60] that if the dual simplex algorithm is used to solve the subproblems in a depth-first strategy, then the solution of each subproblem can be found by an inexpensive update to the basis matrix factorization. Then it is also possible [62, p119-121] to find sharper bounds $\underline{\mathbf{x}}$ and to branch in a way that leads to the early fathoming of new nodes. Production software for integer programming might incorporate these and other algorithmic refinements, or permit the user to specify a branching order.

The CPLEX and Lingo packages mentioned in §4.4.4 can both solve integer linear programs and [117, §10] both use branch-and-bound.

7.8 Dynamic Programming

Many optimization problems can be modeled as a sequence of decisions, each of which changes a state variable which in turn affects the alternatives that are possible at subsequent decisions. For example, declaring an academic major affects the set of courses from which a student can select, and choosing a particular sequence of those courses affects the set of careers that will be open to the student upon graduation. Thus, to choose the right major one must try to anticipate the whole series of future decisions that would ensue if each alternative were chosen. **Dynamic programming** [13] [151] [3, §10] [79, §7] [74, §10-11] is a computational strategy that can be used to study problems of this type.

7.8.1 The Shortest-Path Problem



Simpler than choosing an academic major is the problem of finding the shortest path between two nodes in a network. If in the network pictured to the right [3, Exercise 10.4] the number at the tail of each arrow is the length of that link, what path from node 1 to node 13 has the smallest total length? We could write this problem as an integer program in the manner of §6.5.3 and then use the §7.5.1 zero-one algorithm to solve it, but because of its special structure there is a much easier way to answer the question.

From node 10 there is only one path, of length 3, to node 13; if we somehow find ourselves at node 10, that path is the one we must take.

Likewise there is a unique path, of length 2, from node 11 to node 13, and there is a unique path, having length 3, from node 12 to node 13. From node 5 there is only one possible path to node 10, so if we find ourselves at node 5 we should take that path. Similarly there is a unique path from node 9 to node 12, so if we find ourselves at node 9 we should take the path to node 12.

From node 6 we can go to either node 10 or node 11. We already found that the shortest path from node 10 to node 13 has length 3, so the path $6 \to 10 \to 13$ has a total length of $3+3=6$ units. We already found that the shortest path from node 11 to node 13 has length 2, so the path $6 \to 11 \to 13$ has a total length of $2+2=4$. Thus if we find ourselves at node 6 we should go next to node 11 at the minimum length of $\boxed{4}$. As a reminder of this minimum length to node 13, I have shown it in the rectangle near node 6.

From node 7 we can take the path $7 \to 10 \to 13$ for a length of $1+3=4$, or $7 \to 11 \to 13$ for a length of $4+2=6$, or $7 \to 12 \to 13$ for a length of $2+3=5$. Thus if we find ourselves at node 7 we should go next to node 10 at the minimum length of $\boxed{4}$. By similar reasoning, if we find ourselves at node 8 we should go to node 11 at a length of $\boxed{4}$.

The length of the optimal path from node 2 to node 13 is $1+\boxed{6}=7$ if we go to node 5, $2+\boxed{4}=6$ if we go to node 6, and $3+\boxed{4}=7$ if we go to node 7, so the best choice is to go from node 2 to node 6. Similarly, from node 3 we should go to node 6 and from node 4 we should go to node 8. Now it is easy to see that from node 1 we should go to node 2, with a minimum length for the whole path of $2+\boxed{6}=8$. Starting from node 1 and moving from each node to the optimal next node yields the shortest path $1 \to 2 \to 6 \to 11 \to 13$.

In the picture on the previous page the nodes in each column or **stage** of the problem have directed links entering only from their left and exiting only to their right. Using the picture we solved the problem by finding the length of the shortest path to the destination from each of the nodes in stage 4, then from each of the nodes in stage 3, then from each of the nodes in stage 2, and finally from the single node in stage 1. Then, starting at node 1 and progressing from one stage to the next, we included in the shortest path that next node which yielded the smallest remaining path length.

Suppose that in our example we index the stages by $s = 1 \ldots 5$ and give the set of nodes in stage s the name $\mathbb{N}_s$, so that $\mathbb{N}_1 = \{1\}$, $\mathbb{N}_2 = \{2, 3, 4\}$, $\mathbb{N}_3 = \{5, 6, 7, 8, 9\}$, $\mathbb{N}_4 = \{10, 11, 12\}$, and $\mathbb{N}_5 = \{13\}$. If $p \in \mathbb{N}_s$ and $q \in \mathbb{N}_{s+1}$ and there is a link between node p and node q, then we will call the length of that link L_{pq}; if there is no link then $L_{pq} = \infty$. At the destination node, $s = 5$ and because there is no next stage $\mathbb{N}_6 = \emptyset$ and $f(6, q) = 0$ for any q. With these conventions the calculations above can be described by this recursion.

$$\begin{aligned} f(s,p) &= \text{length of shortest path to destination from node } p \text{ of stage } s \\ &= \min_{q \in \mathbb{N}_{s+1}} [L_{pq} + f(s+1,\ q)] \end{aligned}$$

We can use this formula repeatedly to work backwards from the last stage to the first, so it is called a **backward recursive relation** [3, p350]. For example, once all of the $f(3, q)$ have been found we can compute $f(2, 3)$ like this.

$$\begin{aligned} f(2,3) &= \min[L_{35}+f(3,5),\ L_{36}+f(3,6),\ L_{37}+f(3,7),\ L_{38}+f(3,8),\ L_{39}+f(3,9)] \\ &= \min[\infty+6,\ 2+4,\ 4+4,\ 3+4,\ \infty+6] \\ &= 6 \text{ achieved by going from node } p = 3 \text{ to node } q = 6 \end{aligned}$$

Using backward recursive relations we can solve the problem like this.

$$\begin{aligned}
f(4,10) &= \text{length of shortest path to destination from node 10 of stage 4} &&= 3 \\
f(4,11) &= \text{length of shortest path to destination from node 11 of stage 4} &&= 2 \\
f(4,12) &= \text{length of shortest path to destination from node 12 of stage 4} &&= 3 \\
f(3,5) &= 3 + f(4,10) &&= 6 \\
f(3,6) &= \min[3 + f(4,10),\ 2 + f(4,11)] &&= 4 \\
f(3,7) &= \min[1 + f(4,10),\ 4 + f(4,11),\ 2 + f(4,12)] &&= 4 \\
f(3,8) &= \min[2 + f(4,11),\ 4 + f(4,12)] &&= 4 \\
f(3,9) &= 3 + f(4,12) &&= 6 \\
f(2,2) &= \min[1 + f(3,5),\ 2 + f(3,6),\ 3 + f(3,7)] &&= 6 \\
f(2,3) &= \min[2 + f(3,6),\ 4 + f(3,7),\ 3 + f(3,8)] &&= 6 \\
f(2,4) &= \min[3 + f(3,7),\ 1 + f(3,8),\ 2 + f(3,9)] &&= 5 \\
f(1,1) &= \min[2 + f(2,2),\ 3 + f(2,3),\ 4 + f(2,4)] &&= 8
\end{aligned}$$

It is helpful to organize these calculations in a table; one that is suitable for hand computation is shown on the following page.

s	p	q	$L_{pq} + f(s{+}1,\ q)$	$f(s,p)$
4	10	13	$3+0$	3
4	11	13	$2+0$	2
4	12	13	$3+0$	3
3	5	10	$3+3$	6
3	6	10	$3+3$	
3	6	11	$2+2$	4
3	7	10	$1+3$	4
3	7	11	$4+2$	
3	7	12	$2+3$	
3	8	11	$2+2$	4
3	8	12	$4+3$	
3	9	12	$3+3$	6
2	2	5	$1+6$	
2	2	6	$2+4$	6
2	2	7	$3+4$	
2	3	6	$2+4$	6
2	3	7	$4+4$	
2	3	8	$3+4$	
2	4	7	$3+4$	
2	4	8	$1+4$	5
2	4	9	$2+6$	
1	1	2	$2+6$	8
1	1	3	$3+6$	
1	1	4	$4+5$	

The table must be constructed from top to bottom, because the values we find for $f(s,p)$ in each stage become the $f(s+1,q)$ in the previous stage (which is *below* it in the table). The downward arrows show where the values of $f(4,p)$ end up in the calculations for stage 3. Each $f(s,p)$ value in the rightmost column is the minimum over q of the entries in the previous column for that (s,p).

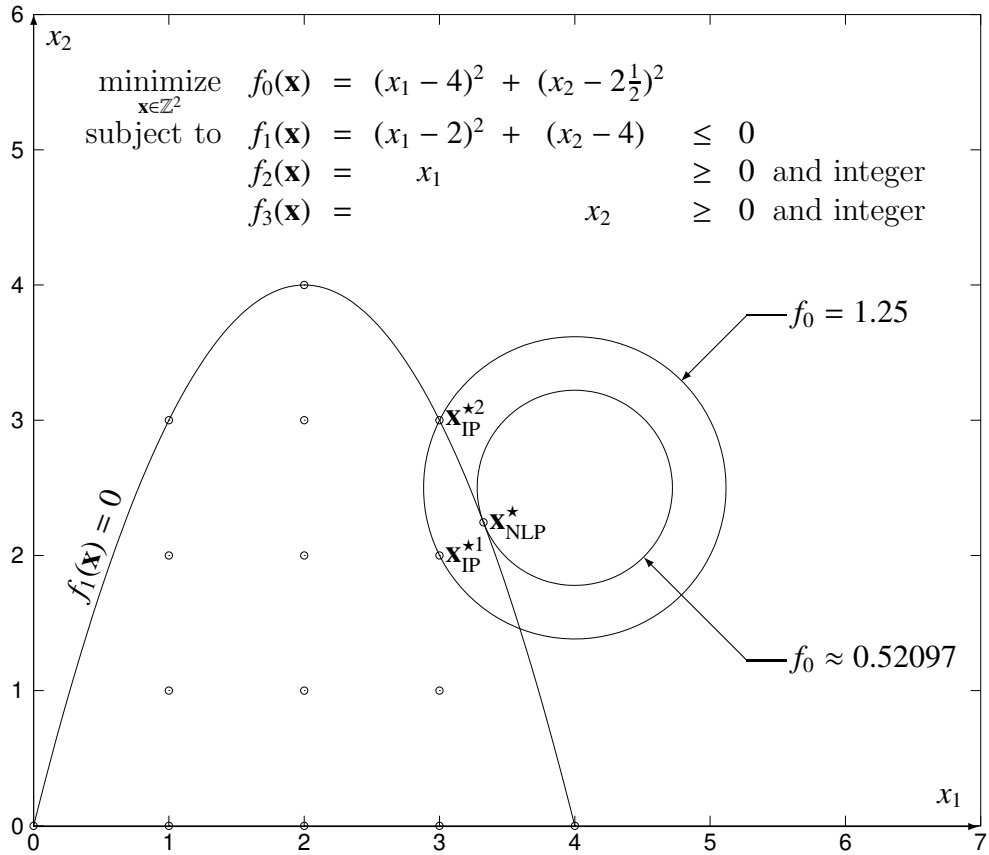
To unwind the recursion we start with the first-stage (p,q) yielding the lowest path length, in this case $(1,2)$ with a length of 8. Next in stage 2 we find the link from node 2 yielding the lowest path length, $(2,6)$. Then in stage 3 we find the link from node 6 yielding the lowest path length, $(6,11)$. Finally in stage 4 we find the link from node 11 yielding the lowest path length, $(11,13)$. The upward arrows show how these links assemble into the shortest path $1 \to 2 \to 6 \to 11 \to 13$.

The method [13] illustrated by this example was first discovered by Richard Bellman [74, p350], but it and several variants are sometimes referred to as **Dijkstra's algorithm**. A program to perform these calculations might use data structures quite different from this table.

7.8.2 Integer Nonlinear Programming

The branch-and-bound approach that we used in §7.3 to solve integer linear programs can also be used to solve integer or zero-one *non*linear programs [151, Exercise 13.42]. Each subproblem is once again a smooth relaxation of the integer-constrained master problem, but now it is a nonlinear program and therefore must be solved using techniques such as those discussed in Chapters 8–25 of this text.

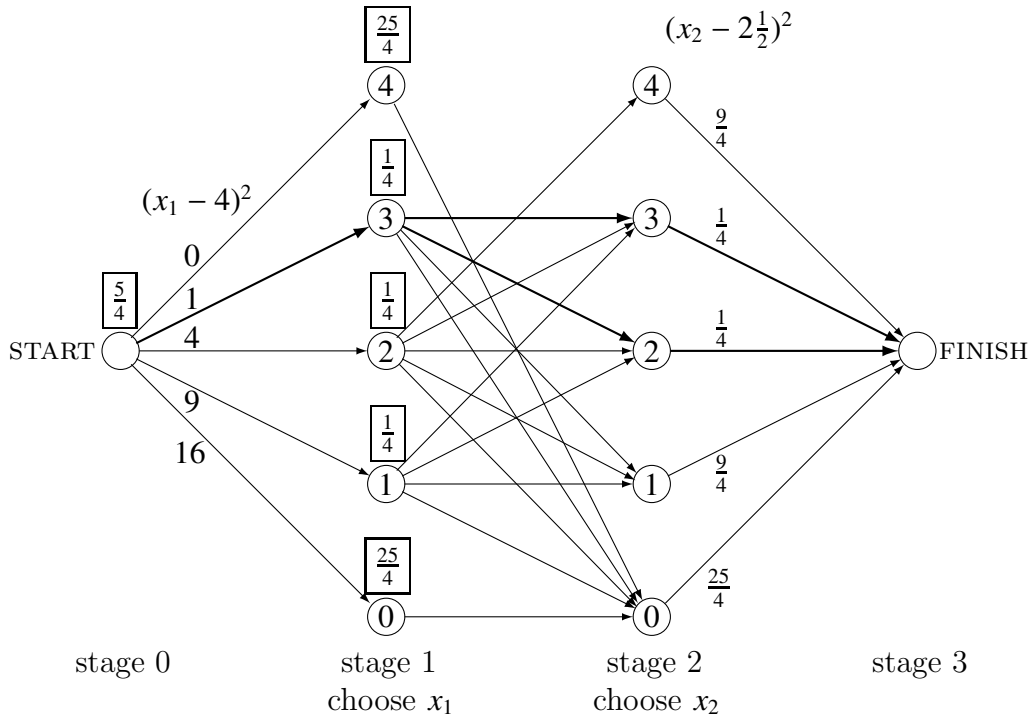
When the objective of an integer nonlinear program is **separable** in the sense that we can evaluate it in stages each involving a single variable, then it might be easier to solve the problem using a dynamic programming approach [3, Exercises 10.12, 10.13, 10.14, 10.15]. The example on the next page, which I will call `inlp` (see §28.8.1) has only $n = 2$ variables, so we can find its optimal lattice points $\mathbf{x}_{\text{IP}}^{\star 1} = [3,2]^\top$ and $\mathbf{x}_{\text{IP}}^{\star 2} = [3,3]^\top$ graphically. How could we find these points without drawing a picture?



From the constraints we can deduce which lattice points are feasible, like this.

$$
\begin{aligned}
-x_2 \ \ge\ (x_1-2)^2-4 &\le 0\\
(x_1-2)^2 &\le 4\\
x_1-2 &\le 2\\
x_1 &\le 4 \quad \text{and } x_1 \ge 0 \text{ so } x_1 \in \{0,1,2,3,4\}\\
\text{but}\quad x_2 &\le 4-(x_1-2)^2 \quad \text{so}\\
x_1 = 0 &\Rightarrow x_2 \in \{0\}\\
x_1 = 1 &\Rightarrow x_2 \in \{0,1,2,3\}\\
x_1 = 2 &\Rightarrow x_2 \in \{0,1,2,3,4\}\\
x_1 = 3 &\Rightarrow x_2 \in \{0,1,2,3\}\\
x_1 = 4 &\Rightarrow x_2 \in \{0\}.
\end{aligned}
$$

Now suppose that in one stage of the solution process we choose a value of x_1 from among the possibilities. Then, in the next stage we choose a value of x_2 from among the possibilities for each value of x_1. This process can be described by the directed graph on the next page, in which the objective contribution of each assignment is shown on the corresponding link. Now we can minimize $f_0(\mathbf{x})$ simply by finding the shortest path from START to FINISH.



Here I have used the same procedure as in §7.8.1. For example, if we choose $x_1 = 3$ then we can complete the evaluation of the objective by choosing x_2 to be 0 (at an additional cost of $\frac{25}{4}$) or 1 (at an additional cost of $\frac{9}{4}$) or 2 or 3 (each of which increases the objective by $\frac{1}{4}$). Thus the shortest path from the $x_1 = 3$ node to FINISH has a cost of

$$\min\left\{\tfrac{25}{4},\ \tfrac{9}{4},\ \tfrac{1}{4},\ \tfrac{1}{4}\right\} = \tfrac{1}{4}$$

as shown in the box above the $x_1 = 3$ node. The two optimal paths from START to FINISH, drawn with thick lines, reveal that $\mathbf{x}_{\text{IP}}^{\star 1} = [3, 2]^\top$ and $\mathbf{x}_{\text{IP}}^{\star 2} = [3, 3]^\top$. The backward recursive relation that we derived in §7.8.1,

$$\begin{aligned} f(s,p) &= \text{length of shortest path to FINISH from node } p \text{ of stage } s \\ &= \min_{q \in \mathbb{N}_{s+1}} [L_{pq} + f(s{+}1,\ q)] \end{aligned}$$

also works here if we label the stages as shown above and let

$$L_{pq} = \begin{cases} (q-4)^2 & p = \text{START}, \quad q \in \mathbb{N}_1 \\ 0 & p \in \mathbb{N}_1, \quad q \in \mathbb{N}_2 \\ (p - 2\frac{1}{2})^2 & p \in \mathbb{N}_2, \quad q = \text{FINISH}. \end{cases}$$

Then we can recurse as we did in the §7.8.1 problem to find $f(1, \text{START})$, and as in that example these calculations could be organized in a table (see Exercise 7.10.53).

Dynamic programming can also be used to solve nonlinear programs having a separable objective function and variables that are continuous rather than being restricted to integer values [3, §10.6] [74, §10.7]. However, that approach is unwieldy if there are more than a few variables, and better methods for smooth nonlinear programs are discussed in Chapters 8–25. Dynamic programs having multiple state variables [3, §10.5] are also beyond the scope of this introduction.

7.9 Computational Complexity

Branch-and-bound and dynamic programming are more efficient than exhaustive enumeration, but both require an amount of computation that increases dramatically with problem size. If we implement these algorithms in computer programs we can include code to count the elementary arithmetic and logical operations they do in solving particular problems (see §26.3) but to derive an analytic model that predicts in general how much work they require it is necessary to consider a simpler and more idealized scenario.

In solving some integer linear programs (such as the $n = 3$ example of §7.3) the branch-and-bound algorithm generates a binary tree having more than n layers, but to make our analysis easy suppose there are exactly n. If each node in one layer produced two nodes in the next there would be $1+2+4+\cdots+2^{n-1} = 2^n - 1$ nodes altogether. In practice some nodes are fathomed during the solution process, so to be more realistic suppose that instead of multiplying the number of nodes in each layer of the tree by 2 to get the number in the next layer, the multiplier is $r \in (1, 2]$. Then the total number of nodes that must be considered is

$$N = 1 + r + r^2 + \cdots + r^{n-1} = \frac{r^n - 1}{r - 1}.$$

This node count N, and hence the work required to perform the algorithm, grows exponentially with n, so in the worst case branch-and-bound has **exponential algorithmic complexity**. No known algorithm capable of exactly solving integer linear programs requires an amount of work that grows slower than that, so the integer linear programming problem is said to have **exponential problem complexity**.

I mentioned in §4.5.3 that although the simplex method has exponential worst-case algorithmic complexity, the smooth linear programming problem can be solved by other algorithms requiring an amount of work that is only a polynomial function of problem size. The complexity of a *problem* is the infimum of the complexities of the *algorithms* that can solve it, so the smooth linear programming problem has **polynomial problem complexity**.

Problems that have exponential complexity are fundamentally harder than those that have polynomial complexity [144, Part Three] because an exponential function always eventually grows faster than a polynomial function. The table on the next page shows that 2^n catches up with n^2 at $n = 4$ and thereafter grows faster, and it is not hard to show that 2^n eventually gets to be bigger than an^r for any $a > 0$ and positive integer r.

n	2^n	n^2
1	1	1
2	4	4
3	8	9
4	16	16
5	32	25
6	64	36

By repeatedly applying L'Hospital's rule [149, §4.5] we find that

$$\lim_{n\to\infty} \frac{2^n}{an^r} = \lim_{n\to\infty} \frac{2^n[\ln(2)]}{arn^{r-1}} = \lim_{n\to\infty} \frac{2^n[\ln(2)]^2}{ar(r-1)n^{r-2}} = \cdots = \lim_{n\to\infty} \frac{[\ln(2)]^r}{ar!}2^n = +\infty.$$

If n is big enough then we can ignore any lower-order terms in a polynomial whose first term is an^r, so this result is enough to show that 2^n gets to be bigger than *any* polynomial function of n.

Problems having polynomial complexity are considered **formally tractable** because they are relatively easy to solve, although some that are of practical importance are too large to be solved even with a polynomial-time algorithm. Problems having exponential complexity are considered **formally intractable** because they get to be so much harder as n increases, even though, as we have seen, many integer linear programs that are of practical importance can be solved.

I have been talking about the computer *time* needed to solve a problem, but the branch-and-bound and dynamic programming algorithms also use an amount of *memory* that grows exponentially with n, and that can also limit their practical utility.

7.10 Exercises

7.10.1 [E] Write down all of the ways you can think of in which counting is different from measuring. How are the two processes related?

7.10.2 [E] Are integer linear programs usually easier to solve or harder to solve than smooth linear programs? Explain.

7.10.3 [H] An argument can be made that all optimization problems involving the physical world are really integer programs. (a) Make the argument. What about the physical world is inherently grainy? (b) Present an exception or counter-argument.

7.10.4 [E] What is the *linear programming relaxation* of an integer program?

7.10.5 [P] The `brewip.m` program includes code for computing bounds on the variables. (a) Which lines of the program perform this calculation? (b) How do they work? (c) What bounds are deduced by this code?

7.10.6 [P] An integer program with a bounded feasible set can be solved by exhaustive enumeration, if we are prepared to wait long enough. For each of the following problems, use the constraints to deduce bounds on the variables, report the total number of lattice points to be considered, and write a MATLAB program that solves the problem by exhaustive enumeration: (a) the `spear` problem of §7.1; (b) the `bb1` problem of §7.2; (c) the `bb2` problem of §7.3. (d) Show how exhaustive enumeration can be used to find both optimal points in the `bb3` problem of §7.4.

7.10.7 [E] In exhaustive enumeration, how does the number of lattice points to check depend on n, the number of variables in the problem?

7.10.8 [H] In §7.1 we considered the possibility of examining all lattice points adjacent to $\mathbf{x}^\star_{\text{LP}}$ in search of $\mathbf{x}^\star_{\text{IP}}$. If the n variables in an integer programming problem are each restricted to be either 0 or 1, how many lattice points are adjacent to a point having $x_j \in \{0, 1\}$? Explain.

7.10.9 [E] Explain the differences between *exhaustive enumeration, partial enumeration, random enumeration*, and *implicit enumeration*. Which of these methods are sure to find an optimal point of an integer program?

7.10.10 [E] The implicit enumeration scheme described in §7.2 is based on two key facts about integer programs. What are those facts, and how are they used?

7.10.11 [E] The implicit enumeration scheme of §7.2 involves *branching*. (a) When does the algorithm branch on a variable? (b) How is branching accomplished? (c) What effect does branching have on the branching diagram? (d) What effect does branching have in the graphical solution of an integer program?

7.10.12 [E] A branching diagram has the shape of an inverted tree. (a) What do the nodes of the tree represent? (b) Where in the tree is the master problem? (c) How many subproblems are generated by each branching?

7.10.13 [E] In the branch-and-bound algorithm of §7.3, what is an *incumbent solution*? Why is it necessary to set an upper bound $\overline{z}$ on the optimal objective value?

7.10.14 [E] In the branch-and-bound algorithm of §7.3, how do we obtain for each subproblem a lower bound $\underline{z}$ on the objective over that subset of $\mathbb{F}$?

7.10.15 [H] What does it mean to say that a node in a branching diagram has been *fathomed*? What fathoming conditions are given in the branch-and-bound algorithm of §7.3? Explain for each fathoming condition why its satisfaction means that the node is fathomed.

7.10.16 [H] In §7.3 we used the branch-and-bound algorithm to solve the `bb2` problem by branching first on x_3. Use the algorithm to solve the problem (a) by branching first on x_1; (b) by branching first on x_2.

7.10.17 [H] Use the branch-and-bound algorithm of §7.3 to solve the `spear` problem of §7.1.

7.10.18 [E] In a branch-and-bound algorithm for solving integer programs, how does the breadth-first strategy differ from the depth-first strategy? Which usually works best in practice?

7.10.19 [H] Modify the branch-and-bound fathoming conditions to account for the possibility that more than one subproblem solution is an optimal point for the integer program.

7.10.20 [H] Use the branch and bound algorithm of §7.3 to find all solutions of the following integer program

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{Z}^3}{\text{minimize}} & -4x_1 - 5x_2 & = & z \\ \text{subject to} & 3x_1 + 2x_2 & \leq & 10 \\ & x_1 + 4x_2 & \leq & 11 \\ & 3x_1 + 3x_3 + x_3 & \leq & 13 \\ & \mathbf{x} & \geq & \mathbf{0} \\ & x_1, x_2, x_3 & \text{are} & \text{integers} \end{array}$$

7.10.21 [H] In our study of `bb3` in §7.4 we branched on x_2 to find a second integer optimum. Show how it can be found by branching on x_1 instead.

7.10.22 [H] If an integer program has multiple optima and each is the solution of a subproblem, then the branch-and-bound algorithm can find them all. Is there any other way in which an integer program can have multiple optima? Explain.

7.10.23 [H] Solve `bb5` by using the branch-and-bound algorithm of §7.3.

7.10.24 [E] The §7.5.1 algorithm for zero-one integer programs and the §7.3 algorithm for general integer programs both use branch-and-bound. How do they differ? Write down all of the ways you can think of.

7.10.25 [E] What does the notation $\mathbf{x} \in 110\square\square\square\square\square$ mean? In it what is the *partial solution*? What are its possible *completions*? What is its *zero completion*? Write down one of its possible trial solutions.

7.10.26 [H] Our zero-one algorithm assumes that the objective coefficients are nonnegative and arranged in nondecreasing order. Use a substitution of variables to put the objective $z(\mathbf{y}) = 10y_1 - 11y_2 + 1y_3 - 7y_4 + 5y_5$ into the required form in terms of $x_j,\ j = 1 \ldots 5$. Are your $x_j \in \{0, 1\}$? Why doesn't the constant matter?

7.10.27 [E] In the zero-one algorithm of §7.5.1, (a) why is $\overline{z}$ initially set to the sum of the objective coefficients? (b) Why does the bounding step use the completion $[1, 0 \ldots 0]^\top$ rather than the zero completion? (c) If the bounding step used the zero completion, would the algorithm still work? Explain. (d) What is *looking ahead*?

7.10.28 [H] In the bounding step of the §7.5.1 zero-one algorithm we look ahead by using the completion $[1, 0 \ldots 0]^\top$ rather than zero completion. (a) Why is the zero completion sure to be infeasible? (b) Modify the algorithm to look *farther* ahead. Would it be worth the effort to do this?

7.10.29 [E] The algorithms of §7.3 and §7.5.1 each have one step that accounts for most of the work. In each algorithm, which step is that? Which of these hard steps is easier?

7.10.30 [P] In §7.5.2, I claimed that in performing "fathoming test (a)" in the zero-one algorithm it is faster to search for an inequality that is violated by all possible completions than it is to verify that none of the possible completions satisfy all of the inequalities. (a) Write a MATLAB function `findfc(A,b,x)` that generates the possible completions to a partial solution one at a time, and for each evaluates all of the constraints at the corresponding trial solution. (b) Show that your code reports there are no feasible completions for node C in the solution of `bb5` but that there are feasible completions for all of the other nodes. (c) Time `fathoma.m` and `findfc.m` (see §26.3.3) to determine which is faster. Does which is faster depend on n? Does it depend on m?

7.10.31 [H] In §7.5.2, I pointed out that even if no single constraint is violated by all possible completions it is still possible that every possible completion violates some constraint. Explain why this claim is true.

7.10.32 [H] Construct a zero-one program in which searching every node for a completion that satisfies all of the inequalities is even more expensive than solving the master problem by exhaustive enumeration.

7.10.33 [P] The `fathoma.m` routine of §7.5.2 uses the MATLAB command `ip=(x' == 1)`. (a) What does this command do? (b) Why did I use `x'` rather than `x`? (c) Explain the behavior of the MATLAB `sum`, `bitand`, and `logical` functions. (d) What return value from `fathoma.m` means that "fathoming condition (a)" fails?

7.10.34 [E] The integrality constraint of an integer program ensures that the optimal vector will have whole-number components, but it also permits the modeling of situations that cannot be described by a smooth linear program. Name two such situations.

7.10.35 [H] In §7.6.1, I reformulated the `bb1` problem as a zero-one program. (a) Use the algorithm of §7.5.1 to solve it. (b) Use $\mathbf{u}^\star$ and $\mathbf{v}^\star$ to compute $\mathbf{x}^\star$, and show that it is the optimal point we found for the original problem. (c) In the reformulation the binary representation of x_1 can represent values from 0 to 7, yet we determined that x_1 can take on values only from 0 to 6. What effect, if any, does this have on the zero-one model and the process of solving it?

7.10.36 [E] In §7.6.1, I discussed an example of selecting an element from a list. What must the vector $\mathbf{y}$ be in order to select the entry 4.7 from the list?

7.10.37 [H] If $x_j \in \{0, 1\}$, write a constraint to enforce the logical condition that x_1 can be 1 only if both $x_2 = 1$ and $x_3 = 1$ (in other words, if either $x_2 = 0$ or $x_3 = 0$ then $x_1 = 0$ but if both $x_2 = 1$ and $x_3 = 1$ then x_1 can be either 0 or 1). Is your constraint linear in the x_j?

7.10.38 [H] In §7.6.1, we added a switch to the first constraint of the `bb1` problem. Can a switch be added to the second constraint? If yes, rewrite the second constraint and explain how the switch works. If no, explain why not.

7.10.39 [H] If two five-digit integers are composed of the unique digits from 0 through 9, their difference can be of either sign.

$$\begin{array}{r} 51627 \\ -\underline{38490} \\ 13137 \end{array} \qquad \begin{array}{r} 09483 \\ -\underline{72615} \\ -63132 \end{array}$$

(a) Formulate an integer linear program whose solution will be the digits of the two five-digit integers whose difference is as small as possible. (b) Solve the integer program.

7.10.40 [H] Consider this nonconvex optimization [74, §8-6].

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{X}}{\text{minimize}} & x_1 + x_2 & = & z(\mathbf{x}) \\ \text{where} & \mathbf{x} & \in & \mathbb{X} = \left\{\mathbf{x} \in \mathbb{R}^2 \;\middle|\; x_1 + x_2 \le 4 \cap \left[(x_2 \ge 2 \cap x_1 \ge 0) \cup (x_1 \ge 2 \cap x_2 \ge 0)\right]\right\} \end{array}$$

(a) Solve the problem graphically. (b) By introducing switch variables y_1 and y_2, formulate the problem as a mixed-zero-one integer program. (c) Solve the problem by using the zero-one algorithm of §7.5.1.

7.10.41 [E] Explain the difference between a *knapsack problem* and a *capital budgeting problem.*

7.10.42 [H] In the formulation of the *facility location problem* in §7.6.2 we replaced the constraint

$$\sum_{j=1}^{n} x_{ij} \le \begin{cases} 0 & y_i = 0 \\ \sum_{j=1}^{n} d_j & y_i = 1 \end{cases} \qquad i = 1 \ldots m,$$

which is nonlinear, by the linear constraint

$$\sum_{j=1}^{n} x_{ij} \le y_i \sum_{j=1}^{n} d_j \qquad i = 1 \ldots m.$$

Show that these constraints are equivalent.

7.10.43 [H] Suppose that in the `brewery` problem of §1.3.1 a **setup cost** is incurred to make any amount greater than zero of each product. Making zero kegs of Porter incurs no setup cost, but if $x_1 > 0$ then the fixed cost of setting up to make that variety is \$3, and this must be deducted from the revenue produced by selling Porter. The setup costs for Stout, Lager, and IPA are respectively \$4, \$5, and \$6. These fixed charges obviously affect $z^\star$, and they might also change $\mathbf{x}^\star$. Formulate this **fixed-charge problem** [74, §4-10] as a mixed-zero-one program.

7.10.44 [E] Describe one way of solving a mixed-integer linear program.

7.10.45 [H] Solve the fixed-charge problem of Exercise 7.10.43 by using the algorithm of §7.3 but branching only on the zero-one variables.

7.10.46 [E] List four methods other than branch-and-bound for solving integer linear programs.

7.10.47 [E] List two commercial software packages that can solve integer linear programs.

7.10.48 [E] What characteristics must an optimization problem have in order for it to be a candidate for solution by *dynamic programming*?

7.10.49 [E] In §7.8.1 we used dynamic programming to solve a *shortest-path problem*. (a) Explain in words the basic idea of this algorithm. (b) What do we mean by a *stage* of the problem? (c) How are the calculations specified by a *backward recursive relation*? (d) We used a table to organize the calculations. Explain how each of the $f(s, p)$ values in that table is obtained. (e) Explain how the shortest path can be deduced from the results in the table of calculations.

7.10.50 [H] In §7.8.1 we solved a shortest-path problem by dynamic programming. (a) Write a MATLAB program to solve this problem by exhaustively enumerating the lengths of all the possible paths from node 1 to node 13. (b) Write the problem as an integer program in the manner of §6.5.3 and then use the §7.5.1 zero-one algorithm to solve it. (c) Which approach requires the least work to find the shortest path?

7.10.51 [H] Suppose that a link is added from node 7 to node 6 in the shortest-path problem of §7.8.1. Can dynamic programming still be used to solve the problem? If not, explain why not; if so, show how.

7.10.52 [H] In the example of §7.8.1, is it possible to change which path is shortest by changing the length of a single link? If not, explain why not; if so, specify a link-length change that changes the shortest path, and report the new shortest path.

7.10.53 [H] In §7.8.2 we used the dynamic programming approach to solve `inlp`. (a) Explain how it is possible to deduce from the constraints of the problem which lattice points are feasible. (b) Devise a table that can be used to organize the evaluation of the backward recursive relations. (c) Evaluate the backward recursive relations to complete your table, and show how the results can be used to determine $\mathbf{x}^\star_{\text{IP}}$. (c) Can the backward recursive relations for this problem be used to find the solution analytically? Explain.

7.10.54 [E] An integer nonlinear program can in principle be solved by using the branch-and-bound approach of §7.3 to generate a tree of subproblems that are nonlinear programs. (a) Why might this approach be difficult to use in practice? (b) What must be true of the problem in order for it to be amenable to solution by dynamic programming instead?

7.10.55 [H] In §7.8.2 we used dynamic programming to solve an integer nonlinear program. (a) Show how the approach can also be used to solve the integer *linear* programs (a) `bb1` of §7.2; (b) `bb2` of §7.3. (c) How does the amount of computation required to perform the algorithm increase with the size of the integer linear program? (d) Would this be an efficient way of solving smooth linear programs?

7.10.56 [H] Consider the following integer nonlinear program [3, Exercise 10.14].

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{Z}^3}{\text{maximize}} & x_1x_2 + x_2 + 2x_1 & = \; z(\mathbf{x}) \\ \text{subject to} & x_1 + 2x_2 & \leq \; 25 \\ & x_1, x_2 & \geq \; 0 \text{ and integer} \end{array}$$

(a) Solve the problem graphically. (b) Solve the problem by using the dynamic programming approach, explaining the backward recursive relations that you use. Hint: rewrite the objective so that it is separable.

7.10.57 [E] Explain the difference between *algorithm complexity* and *problem complexity*. Why is the complexity of an algorithm always an upper bound on the complexity of the problem that it solves?

7.10.58 [H] Suppose that a problem can be solved by either an algorithm that has polynomial complexity or an algorithm that has exponential complexity. (a) What must be the complexity of the problem? (b) Explain why the polynomial algorithm is usually preferable. (c) Describe a class of problem for which the exponential algorithm might be preferable.

7.10.59 [E] Why are problems that have exponential complexity considered *formally intractable* while those that have polynomial complexity are considered *formally tractable*? Why have I used the qualifier "formally" in these terms of art?

7.10.60 [P] For n sufficiently positive 2^n is greater than any polynomial function of n. (a) Write a MATLAB program to compare the values of 2^n and n^5. For what values of n is $2^n > n^5$? (b) Give an analytic argument that there is some n for which $2^n > an^r$ for any a and r. (c) Write a MATLAB program to find, for given values of a and r, the smallest value of n for which $2^n > an^r$. What does it report for $a = 1$ and $r = 5$?

7.10.61 [H] According to L'Hospital's rule, if $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$ then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{df/dn}{dg/dn}$$

provided $dg/dn \neq 0$. (a) If $\lim_{n\to\infty} df/dn = \infty$ and $\lim_{n\to\infty} dg/dn = \infty$, how can the limit on the right be evaluated? (b) If $f(n) = 2^n$, compute df/dn. (c) If $g(n) = an^r$, compute dg/dn. (d) Explain why

$$\lim_{n\to\infty} \frac{[\ln(2)^r]}{ar!} 2^n = +\infty.$$

if r is a positive integer and $a > 0$.

7.10.62 [H] If the master problem in the first layer of a branch-and-bound tree has t constraints then each subproblem in the second layer will have $t + 1$ because of the bound constraints we add to perform the branch. By the time we get to layer p each subproblem will have $m = t + p - 1$ constraints, if no redundancies are eliminated along the way. As I

mentioned in §4.5.3 the simplex algorithm typically uses about $\frac{3}{2}m$ pivots, so to solve each subproblem in layer p we can expect to use $\frac{3}{2}(t + p - 1)$ pivots. In §4.2, I argued that if a linear program has m constraints and n variables then to perform a single pivot takes m divisions, $(1 + n - m)m$ multiplications, and $(1 + n - m)(m - 1)$ subtractions. Assuming that each node in one layer of the tree produces r nodes in the next, derive a formula in terms of n, t, and r for the number of elementary operations required to solve an integer linear program. What is the complexity of the algorithm if the work it does is taken to be the total number of elementary operations?

8

Nonlinear Programming Models

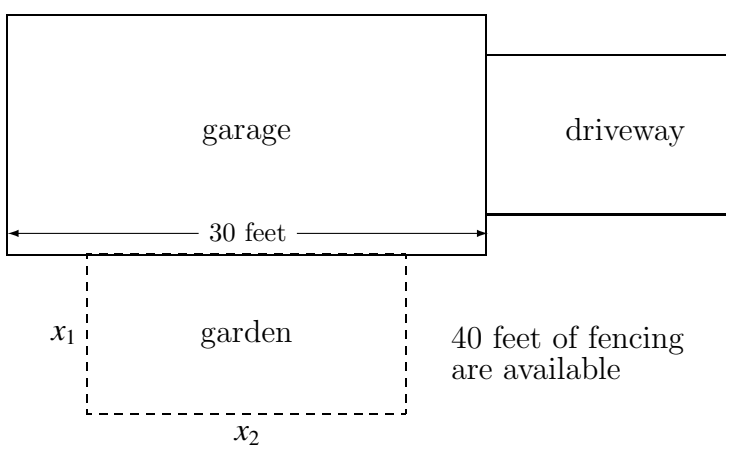
In §1 we studied linear programming models by considering several representative examples. The formulation process and technical vocabulary that you learned then mostly apply to nonlinear models as well, but now both the objective function and the constraint functions can be nonlinear. Many problems that are commonly formulated as linear programs are really nonlinear, and the simplifying approximation of linearity might not be justified when effects such as economies of scale cause departures from strict proportionality between inputs and outputs [3, §9.1]. Other models are **essentially nonlinear**, in that they cannot be linearized without fundamentally changing their character.

We will begin by considering a model that is essentially nonlinear even though it is *almost* a linear program.

8.1 Fencing the Garden

This summer Sarah's vegetable garden was eaten mainly by the local wildlife, so next year she plans to fence the critters out. To make the rectangular garden as big as possible she will use a side of her garage as one side of the enclosure. The garage is 30 feet long, and she has 40 feet of fencing on hand. What should be the dimensions of the garden to maximize its area?

As in formulating a linear program, we begin by summarizing the data. In this problem the easiest way to do that is in the diagram below.



The next step is to select decision variables by answering the question "what can Sarah control?" The answer is that she gets to pick the garden's side lengths, labeled x_1 and x_2.

Then we look for constraints. If the garage wall is going to serve as one side of the enclosure then x_2 can't be more than 30 feet, and if Sarah is not going to need extra fencing then $2x_1 + x_2$ can't be more than the 40 feet she has on hand. It also doesn't make sense for either garden side to have a negative length. Finally we come to the objective, which is to make the area $x_1 \times x_2$ as big as possible. If we express these thoughts mathematically we get this **nonlinear program**.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & x_1 x_2 & & \\ \text{subject to} & 2x_1 + x_2 & \leq & 40 \\ & x_2 & \leq & 30 \\ & x_1 & \geq & 0 \\ & x_2 & \geq & 0 \end{array}$$

Using the transformation you learned in §2.9.2 and rearranging yields the minimization below, which I will refer to from now on as the `garden` problem (it is cataloged in §28.7.1).

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}}\ f_0(\mathbf{x}) = & -x_1 x_2 & = & z \\ \text{subject to}\ f_1(\mathbf{x}) = & 2x_1 + x_2 - 40 & \leq & 0 \\ f_2(\mathbf{x}) = & x_2 - 30 & \leq & 0 \\ f_3(\mathbf{x}) = & -x_1 & \leq & 0 \\ f_4(\mathbf{x}) = & -x_2 & \leq & 0 \end{array}$$

Stated this way, the `garden` problem is in the **standard form** that we will use for nonlinear programs.

$$\boxed{\begin{array}{rcll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x}) & = & z & \\ \text{subject to}\ f_i(\mathbf{x}) & \leq & 0, & i = 1\ldots m \end{array}}$$

This standard form also describes problems having equality constraints, because $g(\mathbf{x}) = 0$ can always be replaced by the two constraints $g(\mathbf{x}) \leq 0$ *and* $g(\mathbf{x}) \geq 0$. The simplex method implicitly enforces $\mathbf{x} \geq \mathbf{0}$ but algorithms for nonlinear programming *do not,* so this standard form *does not* specify that the variables are nonnegative. If variables must be nonnegative, as in the `garden` problem, explicit constraints must be included to ensure that. As in the Chapters about linear programming, I will always use z for the value of a function that is being *minimized.*

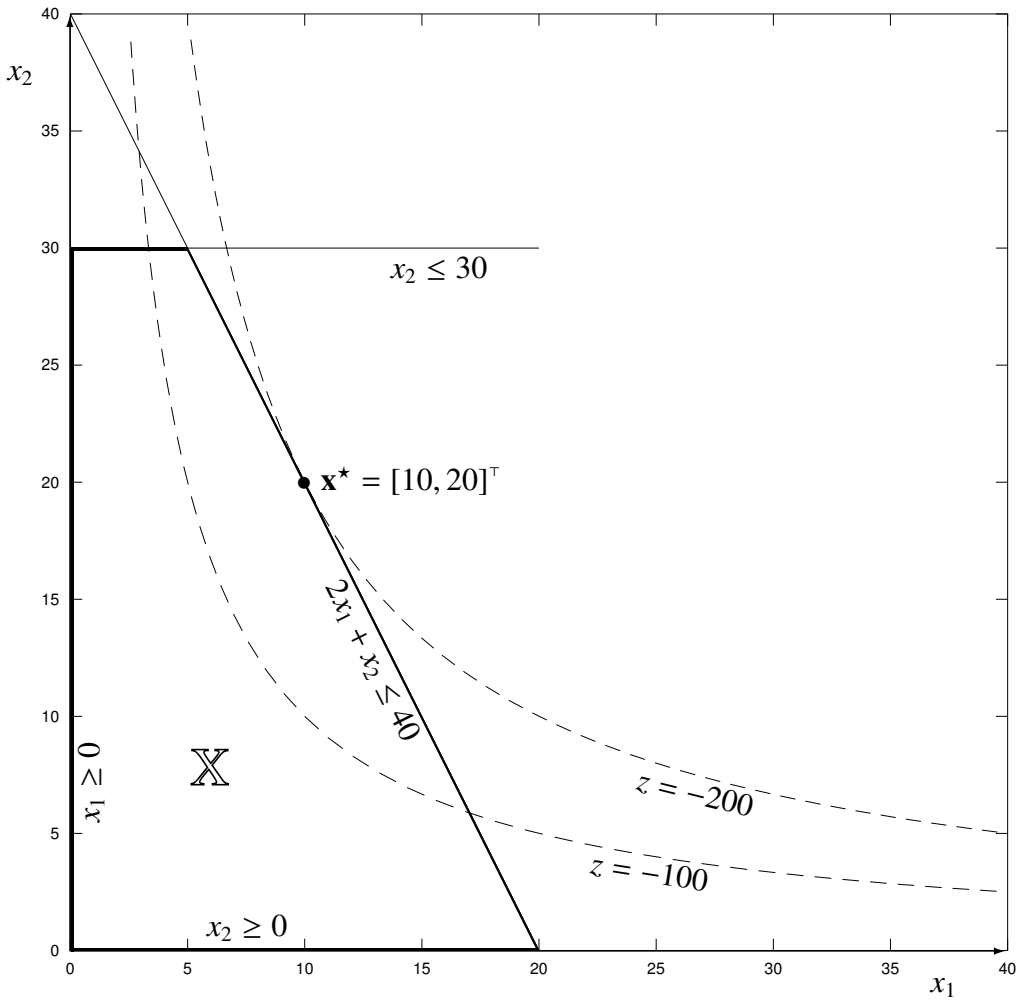
8.2 Analytic Solution Techniques

The nonlinear programming problem in general is really very simple to state: find a feasible $\mathbf{x}$, by any means you like, that yields the lowest possible value of the objective function.

"Any means" includes reading tea leaves or asking random passers-by, but other techniques have been discovered that usually work better and this Section introduces some of them. You should already have some idea how to solve the `garden` problem by graphing it or by using calculus, but please don't be alarmed if the other approaches are unfamiliar or if their exhibition here doesn't teach you how to use them, because we will cover them in detail later.

8.2.1 Graphing

If $n = 2$, and maybe even if $n = 3$, we can solve a nonlinear program graphically in a way similar to the way we have solved linear programs graphically (though the detailed procedure of §1.2 is of limited help here). The constraints of the `garden` problem require that $0 \le x_1 \le 20$ and $0 \le x_2 \le 30$, and using these bounds we can pick good scales for axes and plot the graph below (I used MATLAB but this picture is also easy to sketch by hand).



The feasible set of a nonlinear program has boundaries that are **hypersurfaces**, which can be either flat or curved, and it includes its boundary points so it is a **closed set** [1, §A.3]. Because the constraints of the `garden` problem are linear their zero contours or hypersurfaces are hyper*planes,* and the intersection of their feasible halfspaces is the convex polyhedron that is outlined in thick lines. This feasible set $\mathbb{X}$ has interior points and is bounded, but in general the feasible set of a nonlinear program can be a single point or unbounded, or it can be empty (in which case the nonlinear program is infeasible). It might or might not be a convex set (see §3.5), and it might or might not even be a **connected set** [148, §9.3.3].

Because the objective is nonlinear its contours are curves. Two contours of $f_0(\mathbf{x})$ are drawn above, for $z = -100$ and $z = -200$.

From the picture we see that the optimal point is where $f_0(\mathbf{x}^\star) = -200$ and $f_1(\mathbf{x}^\star) = 0$, because making z lower than -200 would move the objective contour up and to the right and then it would no longer touch $\mathbb{X}$. In a nonlinear program the optimal point need not be at an intersection of constraint contours, and might even be interior to the feasible set. At the optimal point of this problem $f_1(\mathbf{x}^\star) = 0$ so that constraint is tight, while the other constraints are slack. We can read off the coordinates of $\mathbf{x}^\star$ from the graph or find them algebraically by solving these simultaneous equations.

$$\left.\begin{array}{rcl} -x_1x_2 & = & -200 \\ 2x_1 + x_2 - 40 & = & 0 \end{array}\right\} \Rightarrow x_1 = 10,\ x_2 = 20$$

A linear program that is feasible has either a finite optimal value that is attained at an optimal point, or an unbounded optimal value and no optimal point. In addition to those outcomes a feasible nonlinear program can have an infimum [148, §3.1.1] instead of a minimum value. For example, minimize $1/x$ subject to $x \geq 0$ is a feasible nonlinear program and its objective is not unbounded, but its infimum of 0 is never attained so it has no minimizing point (we might say informally that $x^\star = +\infty$).

8.2.2 Calculus

If a nonlinear program has $m = 0$ constraints, or if the only ones we need to worry about are equalities, we might be able to find the minimizing point using calculus. From the statement of the `garden` problem we could guess that the constraint $2x_1 + x_2 - 40 \leq 0$ will be tight or active at optimality and the other constraints will be slack or inactive. Why use less fencing than available, or give the garden implausible dimensions? In that case we can use the tight constraint to eliminate x_2 in the objective and get an unconstrained optimization in only x_1.

$$\begin{array}{rcl} 2x_1 + x_2 - 40 & = & 0 \\ x_2 & = & 40 - 2x_1 \\ z = -x_1x_2 & = & -x_1(40 - 2x_1) = -40x_1 + 2x_1^2 \end{array}$$

Now we can treat minimizing z like an ordinary max-min problem (see §28.1.1).

Setting the first derivative of z to zero we can find x_1, and then we can find x_2 from the constraint equation. In a problem having a more interesting objective function we might not be able to solve the equation $dz/dx = 0$ analytically, or it might have multiple solutions. Here there is only one, but we should still use the second-derivative test to verify that $\mathbf{x}^\star$ actually minimizes z.

$$\begin{aligned}
\frac{dz}{dx_1} &= -40 + 4x_1 = 0 \\
4x_1 &= 40 \\
x_1 &= 10 \\
x_2 &= 40 - 2x_1 = 40 - 2(10) = 20 \\
\frac{d^2z}{dx_1^2} &= +4 > 0 \quad \Rightarrow \quad \mathbf{x}^\star = [10, 20]^\top \quad \text{is a minimizing point } \checkmark
\end{aligned}$$

In §15.0 and §16.8.2 I will have more to say about using equality constraints (or tight inequality constraints) to eliminate variables in nonlinear programs.

8.2.3 The Method of Lagrange

Another way to use the tight inequality constraint is to form the **Lagrangian**

$$\mathcal{L}(x_1, x_2, u) = -x_1x_2 + u(2x_1 + x_2 - 40)$$

and minimize it with respect to both $\mathbf{x}$ and the **Lagrange multiplier** u.

$$\left.\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_1} &= -x_2 + 2u = 0 \\
\frac{\partial \mathcal{L}}{\partial x_2} &= -x_1 + u = 0 \\
\frac{\partial \mathcal{L}}{\partial u} &= 2x_1 + x_2 - 40 = 0
\end{aligned}\right\} \Rightarrow x_1^\star = 10,\ x_2^\star = 20,\ u^\star = 10$$

The equations above are called the **Lagrange conditions** and depending on the problem it is possible that they will have no analytic solution. They can also have multiple solutions, and in that case it will be necessary to sort out the ones that are minimizing points. We will make a serious study of this approach in §15.3.

8.2.4 The KKT Method

If we were presented with the `garden` problem in mathematical form, without the story and the picture, it might not be so obvious which constraints are tight at optimality. In that case we could try an extension of the method of Lagrange called the **KKT method**,

which automatically figures out which constraints are tight (in §16.3 we will meet the people for whom this method is named). In the KKT method the Lagrangian includes all of the constraints, so for the `garden` problem we get

$$\mathcal{L}(\mathbf{x}, \mathbf{u}) = -x_1 x_2 + u_1(2x_1 + x_2 - 40) + u_2(x_2 - 30) + u_3(-x_1) + u_4(-x_2)$$

Then we write down the **KKT conditions** as follows.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_1} &= -x_2 + 2u_1 - u_3 = 0 \\
\frac{\partial \mathcal{L}}{\partial x_2} &= -x_1 + u_1 + u_2 - u_4 = 0 \\
\frac{\partial \mathcal{L}}{\partial u_1} &= 2x_1 + x_2 - 40 \le 0 \\
\frac{\partial \mathcal{L}}{\partial u_2} &= x_2 - 30 \le 0 \\
\frac{\partial \mathcal{L}}{\partial u_3} &= -x_1 \le 0 \\
\frac{\partial \mathcal{L}}{\partial u_4} &= -x_2 \le 0 \\
u_1(2x_1 + x_2 - 40) &= 0 \\
u_2(x_2 - 30) &= 0 \\
u_3(-x_1) &= 0 \\
u_4(-x_2) &= 0 \\
u_1 &\ge 0 \\
u_2 &\ge 0 \\
u_3 &\ge 0 \\
u_4 &\ge 0
\end{aligned}$$

Now we just need to find all solutions to this large system of nonlinear equations and inequalities, and sort out the ones we want. That is a tedious chore by hand, but Maple is very good at it as the conversation on the next page illustrates. Maple finds two solutions, but it is easy to see which of them yields the lower objective value and is therefore the minimizing point.

Nonlinear programs that are only slightly more complicated than the `garden` problem can have KKT conditions that are *much* more difficult to solve, and then a computer algebra system such as Maple is indispensable.


```
> eq1 := -x2+2*u1-u3 = 0;
                              -x2 + 2 u1 - u3 = 0

> eq2 := -x1+u1+u2-u4 = 0;
                            -x1 + u1 + u2 - u4 = 0

> eq3 := 2*x1+x2-40 <= 0;
                                2 x1 + x2 <= 40

> eq4 := x2-30 <= 0;
                                    x2 <= 30

> eq5 := -x1 <= 0;
                                    -x1 <= 0

> eq6 := -x2 <= 0;
                                    -x2 <= 0

> eq7 := u1*(2*x1+x2-40) = 0;
                            u1 (2 x1 + x2 - 40) = 0

> eq8 := u2*(x2-30) = 0;
                               u2 (x2 - 30) = 0

> eq9 := -u3*x1 = 0;
                                   -u3 x1 = 0

> eq10 := -u4*x2 = 0;
                                   -u4 x2 = 0

> eq11 := u1 >= 0;
                                    0 <= u1

> eq12 := u2 >= 0;
                                    0 <= u2

> eq13 := u3 >= 0;
                                    0 <= u3

> eq14 := u4 >= 0;
                                    0 <= u4

> solve(
  eq1,eq2,eq3,eq4,eq5,eq6,eq7,eq8,eq9,eq10,eq11,eq12,eq13,eq14,
  x1,x2,u1,u2,u3,u4
      );

            u1 =  0, u2 = 0, u3 = 0, u4 = 0, x1 =  0, x2 =  0,
            u1 = 10, u2 = 0, u3 = 0, u4 = 0, x1 = 10, x2 = 20
```

Of course it might turn out that the KKT conditions, like the Lagrange conditions or the simple equation $dz/dx = 0$, have no closed-form solution at all. Even more disappointing, some problems do not satisfy the conditions that are necessary for the KKT conditions or the Lagrange conditions or even the derivative condition to yield the optimal point as a solution. Must we abandon hope of ever solving such problems?

8.3 Numerical Solution Techniques

Valuable insights into nonlinear programming can be gained by using the mathematical theory of optimization to study toy examples, and we will do that routinely in future Chapters. However, as our experience in the previous Section suggests, the techniques described there are hard to use for the analytic solution of problems much larger or more complicated than the `garden` example. Fortunately, that limited theory has informed the development of methods that are effective for the *numerical* solution of many nonlinear programs arising in practical applications. So if we can tolerate answers that are numbers instead of formulas, we need not abandon the hope of solving real problems.

8.3.1 Black-Box Solvers

Often it is possible to carry out an optimization by using a computer program that someone else already wrote. If your main interest is in *applications* of nonlinear programming, so that getting the answer is more important to you than understanding how, then you should definitely take advantage of prepackaged or **black-box** software [117]. Of course, writing some toy programs of your own first will prepare you to make intelligent use of the professionally-written codes that are available. Selecting industrial-strength software, describing your problem to it, and interpreting the solution it reports should all be possible once you know the theory and methods that are covered in the remainder of this book.

NEOS. The easiest way to get an answer, if you have access to the internet, is to use one of the programs available on the NEOS Server. Navigating your web browser to

`www.neos-server.org/neos/solvers/index.html`

will display a list of programs capable of solving problems in several categories; the most general are those intended for "Nonlinearly Constrained Optimization" and the most famous program listed there is MINOS. Clicking on your selection will display a new page that includes spaces where you can enter your email address and pathnames to files describing your problem. Then you can click on a box to submit your job to the server, which will run it and email you the results.

You specify your problem to NEOS in a **modeling language**, the most widely-used of which is **AMPL**. This package has an excellent manual [61], which is indispensable and downloadable for free. AMPL is itself a high-level programming language, and it can be used to concisely describe a wide variety of optimization models along with the data sets to which you want them applied. To solve the `garden` problem I needed to prepare only a **model file**, which I called `garden.mod`, and a **command file**, which I called `garden.cmd`, both of which are listed on the next page. In the model I maximized x_1x_2 rather than minimizing $-x_1x_2$ only so that I could name the objective function `area`.


```
# this is AMPL input file garden.mod

var x1:=1;
var x2:=1;
maximize area:      x1*x2;
subject to fence: 2*x1+x2 <= 40;
subject to wall:       x2 <= 30;
subject to plusx:      x1 >=  0;
subject to plusy:      x2 >=  0;
```

```
# this is NEOS input file garden.cmd

# model garden.mod; <-- omit for NEOS
solve;
display x1,x2;
quit;
```

I submitted these files to NEOS as described earlier, and after a few minutes received via email the results shown below. To make sense of the output stanza beginning "Presolve" we would need to know some technical details about MINOS (but see §4.4.3).

```
************************************************************

   NEOS Server Version 5.0
   Job#     : 966706
   Password : QyJlpbSn
   Solver   : nco:MINOS:AMPL
   Start    : 2013-09-09 10:22:22
   End      : 2013-09-09 10:22:22
   Host     : neos-4.neos-server.org

   Disclaimer:

   This information is provided without any express or
   implied warranty. In particular, there is no warranty
   of any kind concerning the fitness of this
   information  for any particular purpose.

************************************************************

Job 966706 sent to neos-4.neos-server.org
password: QyJlpbSn

---------- Begin Solver Output -----------
Executing /opt/neos/Drivers/minos-ampl/minos-driver.py
  at time: 2013-09-09 10:22:22.539775
File exists
You are using the solver minos.
Executing AMPL.
processing data.
processing commands.

Presolve eliminates 3 constraints.
Adjusted problem:
2 variables, all nonlinear
1 constraint, all linear; 2 nonzeros
1 inequality constraint
1 nonlinear objective; 2 nonzeros.

MINOS 5.51: optimal solution found.
2 iterations, objective 200
Nonlin evals: obj = 7, grad = 6.
x1 = 10
x2 = 20
```


The NEOS programs are very sophisticated and powerful, and accessing them through the server ensures that they are, unlike other software you might find on the internet, safe to use. Some are open-source [175] but others are proprietary, which means you can't examine their source code. Although some of the programs command a hefty fee if they are licensed for stand-alone use, all of them can be used for free through the server (subject to a quite generous limit on the CPU time you consume).

MATLAB. If you can afford this program, it is only a little more difficult to install it on your computer than it is to use NEOS. MATLAB's **optimization toolbox** contains functions capable of solving a wide variety of mathematical programming problems [117, p105-106].

The free open-source work-alike Octave (see §0.2.3) lacks the optimization toolbox but does have a built-in function `sqp(xzero,f,g,h)` for solving nonlinear programs. When used in the simplest way it invokes these routines: `f` to compute values of the objective function, `g` to compute values of the equality constraint functions, and `h` to compute values of the inequality constraint functions, assuming the inequality constraints are written in the form $h(\mathbf{x}) \geq 0$. To solve the `garden` problem I prepared these MATLAB functions.

```
% gdnobj.m: garden problem objective

function f=gdnobj(x)
  f=-x(1)*x(2);
end
```

```
% gdngeq.m: garden problem inequality constraints

function h=gdngeq(x)
  h=[ 40-2*x(1)-x(2)
      30-x(2)
      x(1)
      x(2) ];
end
```

Then I was able to invoke `sqp()` to solve the problem, as shown below. Because the `garden` example has no equality constraints, I passed a null array for the `g` parameter. The solver made no progress from the starting point $\mathbf{x}^0 = [0, 0]^\top$ but it found the right answer from $\mathbf{x}^0 = [1, 1]^\top$ (that is also the starting point we used for MINOS).

```
octave:1> xzero=[0;0];
octave:2> xstar=sqp(xzero,@gdnobj,[],@gdngeq)
xstar =

   0
   0

octave:3> xzero=[1;1];
octave:4> xstar=sqp(xzero,@gdnobj,[],@gdngeq)
xstar =

   10.000
   20.000

octave:5> quit
```

MATLAB and Octave are both high-quality professional software, and Octave is open-source so you can examine its workings if that is really necessary to investigate unexpected behavior.

The user interface to `sqp()` is MATLAB functions, which are either easier to use than the AMPL interface to NEOS or more difficult, depending on your prior experience and what you need to do. By writing more complex `f`, `g`, and `h` routines it is possible to provide `sqp()` with derivatives of the objective and constraint functions, and by using extra calling parameters it is possible to impose lower and upper bounds on the variables, control the number of iterations performed, and set a convergence tolerance. By adding return parameters it is also possible to learn the optimal objective value, how many iterations were used, and other information about the solution. Invocations of `sqp()` are easy to include in a larger MATLAB program if, as sometimes happens, the optimization is just one step in a larger calculation.

8.3.2 Custom Software

If many nonlinear programs that arise in practice can be solved by simply using software that has already been written and perfected by experts, and if much of that software can be used for free, why would anyone go to the trouble of writing a new solver?

One answer, which I mentioned above, is that firsthand experience actually implementing optimization methods, and in the process using the theory on which they are based, will help you make effective use of those venerated black-box programs. This is the same argument I made in §4.4.4 for learning linear programming rather than just learning about how to use the excellent packages that are available for solving those problems, and it applies with extra force in the case of nonlinear programming because more things can go wrong.

Indeed, some difficulties can arise from depending on prepackaged software even if you know enough to make expert use of it.

The most obvious drawback of a black box program is that you either can't look inside or, if the source code is public, can't readily understand what you see. Journal editors [170] [176], referees, and the readers of scientific papers are often (justifiably) skeptical that a calculation performed in secret is really the one that is wanted or that its results are correct. If you write your own code you will know how it works, and that it works. You will also incidentally avoid license charges and internet security exposures.

It is also possible that all of the extant programs will fail outright or run too long on the one particular problem you desperately need to solve. In that case your only recourse might be writing a special-purpose code, based on the theory and classical methods of nonlinear programming, that precisely fits your project.

Finally, the fact that you are reading this book in the first place suggests you might be someone who would enjoy writing a production-quality code of your own. The programs that are available today were all written by people just like you, and they leave plenty of room for improvement. The perfect solver has yet to be devised for either general nonlinear programs or those falling in the other categories listed on NEOS. As I write these words, big data problems (see §8.6 and §8.7) are of great and growing interest, and the development of methods for solving them is an active area of research. This book uses MATLAB, but a production code is typically written in a compiled language such as C++ or FORTRAN [100].

8.4 Applications Overview

Besides being helpful in the fencing of vegetable gardens, nonlinear programming has many uses in science, engineering, business, and government. Here are a few representative fields in which nonlinear optimization models play an important role (some of them are also recognizable as fields in which *linear* programming is widely used).

composite beam design	supply-chain management
option pricing	disaster response planning
electronic circuit synthesis	protein folding
machine learning	genetic sequence alignment
radar signal processing	molecular structure prediction
electoral redistricting	drug design
electrical generator dispatching	city planning
cancer radiotherapy	pollution control
hospital operating-room scheduling	military logistics
chemical synthesis	aircraft design
design of experiments	stellarator design
renewable energy	feedback control

The references described below discuss the formulation of specific application problems from some of these fields. I have arranged these books in decreasing order of their emphasis on problem formulation; useful general advice is also provided in [2, §2.7] and [1, §1.3].

reference	modeling content
[18]	The chapter topics are weapons assignment, bid evaluation, alkylation process optimization, chemical equilibrium, structural optimization, launch vehicle design, parameter estimation and curve fitting, stochastic programming, and optimal sample sizes.
[4]	Problems from scheduling, portfolio optimization, radiation therapy, image reconstruction, and shape optimization are discussed in §1.7 and its exercises; §1.7.2 is about support vector machines.
[46]	Problems involving solar energy and the design of transformers are discussed in §V; a simpler problem is discussed in §I.5.
[1]	Problems from optimal control, structural design, mechanical design, electrical networks, water resources management, stochastic resource allocation, and facility location are discussed in §1.2, and Exercises 1.2–1.14 are nonlinear programming formulations.
[12]	Problems involving economic order quantity, queueing systems, chemical reactors, box beams, and material processing are discussed in §4; a simpler problem is discussed in §1.
[156]	The design of a chemical plant is discussed in §2-01–§2-05.
[161]	Problems involving regression, container design, and optimal control are discussed in §1.3–§1.5, and other formulations are requested in exercises 1.4, 1.5, 1.6, 1.12, 1.13, and 1.14.
[74]	The optimization of a manufacturing process beset by random flaws is discussed in §3-3.
[59]	The design of a distillation column is discussed in §1.1.
[3]	A nonlinear program is formulated in §9.1, and Exercises 9.1–9.3 are nonlinear programming formulations.
[151]	Exercises 14.47–14.51 are nonlinear programming formulations.
[80]	Exercise 2.8 has four parts that are nonlinear programming formulations.

Many **synthetic problems** have also been made up just to illustrate the theory of nonlinear optimization or how numerical methods work. Since mathematical programming became a recognized subject in the 1940s, researchers and practitioners have collected small problems of both the application and synthetic varieties for use in software testing, and I will describe several well-known collections of such standard test problems in §26.2.1.

Some applications of nonlinear optimization give rise to problems that have many variables and in which the function values can depend on vast quantities of data. Practical models for these big data problems are often constructed along with special-purpose methods for solving them.

As we explore the theory and methods of nonlinear optimization, the examples that we consider will be synthetic problems having only a few variables and functions that are specified by simple formulas. Before we leave the topic of nonlinear programming models we will therefore consider an important application problem in each of the next three Sections. To study them in detail it will be necessary to use simple instances, but hopefully you will be able to imagine more realistic (and more challenging) versions of these problems. Here, as in §8.2, you should be able to follow the development even if a few details happen to be things you don't know yet.

8.5 Parameter Estimation

Dynamical systems can often be described by differential equations whose form is determined by physical laws. For example, the height y of an object of mass m falling under the influence of gravity can be predicted from Newton's second law (force = mass×acceleration) by solving the following initial-value problem.

$$-mg = m\frac{d^2y}{dt^2}, \quad y(0) = y_0,\ y'(0) = 0.$$

Integrating this equation to obtain $y(t)$ is called the **forward problem** and yields

$$y(t) = y_0 - \tfrac{1}{2}gt^2.$$

In a physics course you might have used this result to predict the itinerary of an object as it falls to Earth, near which g is about 32.17 ft/sec^2. Now suppose that the experiment is instead conducted near the surface of another planet, where the local value of g is unknown. Using measurements of y at several values of t to estimate the constant parameter g is called the **inverse problem** [132, §18.4] [106, §1.5].

We can estimate g by finding the value that makes the predictions $y(t_l; g)$ of the solution to the differential equation agree as closely as possible with observations $\hat{y}_l$ taken at times $t_l,\ l = 1\ldots L$ after the object is released. A direct way of doing this is to minimize, by varying g, the sum of the squares of the differences between the $\hat{y}_l$ and the $y(t_l; g)$, like this.

$$\underset{g}{\text{minimize}}\ \ R(g) = \sum_{l=1}^{L} [\hat{y}_l - y(t_l; g)]^2$$

The objective R is called the **residual** of the fit between the model and the data. If the data are the measurements in this table

l	time t_l (sec)	height $\hat{y}_l$ (ft)
0	0	$5000 = y_0$
1	5	4750
2	10	4037
3	15	2828

we can evaluate the sum to obtain the nonlinear program solved below.

$$\begin{aligned}
\underset{g}{\text{minimize}}\ R(g) &= [\hat{y}_1 - y(t_1; g)]^2 + [\hat{y}_2 - y(t_2; g)]^2 + [\hat{y}_3 - y(t_3; g)]^2 \\
&= \quad [4750 - (5000 - \tfrac{1}{2}g5^2\ \ )]^2 \\
&\quad + [4037 - (5000 - \tfrac{1}{2}g10^2)]^2 \\
&\quad + [2828 - (5000 - \tfrac{1}{2}g15^2)]^2 \\
\frac{dR}{dg} &= \quad 2[4750 - (5000 - 12.5g)]^1(12.5) \\
&\quad + 2[4037 - (5000 - 50g]^1(50) \\
&\quad + 2[2828 - (5000 - 112.5g]^1(112.5) \\
&= 30625g - 591250 = 0 \\
g^\star &= 19.31 \text{ ft/sec}^2
\end{aligned}$$

This is a minimizing point because $d^2R/dg^2 = 30625 > 0$, so $g^\star$ is the best **least-squares estimate** of g. Apparently this planet has about 60% of Earth's gravity.

Another way to measure gravitational acceleration is by using a pendulum. If a point mass m that is suspended from a frictionless pivot by a rigid, straight, weightless rod of fixed length r is displaced from the vertical by an angle θ_0 and released, its motion can be predicted (also from Newton's second law) by solving this initial value problem.

$$-mg\sin(\theta) = mr\frac{d^2\theta}{dt^2}, \quad \theta(0) = \theta_0,\ \theta'(0) = 0$$

It is possible by using perturbation series [105, p48-53] to approximate $\theta(t)$ as accurately as desired, but this problem has no closed-form analytic solution. If we make observations $\hat{\theta}_l$ at times t_l, $l = 1 \ldots L$ after the pendulum is released, we can estimate g as we did before by solving this nonlinear program.

$$\underset{g}{\text{minimize}}\ R(g) = \sum_{l=1}^{L} [\hat{\theta}_l - \theta(t_l; g)]^2$$

Now, however, we cannot simply substitute an algebraic expression for $\theta(t)$ into the formula for $R(g)$, simplify, and use calculus to find $g^\star$. In this case it is necessary to do the optimization numerically, solving the initial value problem numerically whenever a value of $\theta(t_l; g)$ is needed by the nonlinear program solver (see Exercise 8.8.23 of reference [100]).

Until now we have considered only **type–1** nonlinear programs, in which the function values and derivatives could be calculated using formulas. The parameter estimation problem is usually a **type–2** nonlinear program [49] like this one, in which the value and the derivatives of the objective function (or of a constraint function, if there are any) must be approximated numerically [115, §3].

Inverse problems are ubiquitous in science and engineering, making parameter estimation probably the most common single application of nonlinear programming. Often the differential equation model is much more complicated than the ones we have considered. It might involve multiple variables and several constant parameters, be a boundary-value problem rather than an initial-value problem or have side conditions that are algebraic equations, and make use of a large number of experimental measurements. If the errors in the observations do not follow the **normal** or **Gaussian probability distribution**, we might prefer to minimize the sum of the absolute values of the errors rather than the sum of their squares, and then the objective function is not everywhere differentiable. In some problems it is also necessary to constrain the parameters to have a particular sign or to have some relationship to one another. Thus, in addition to being of great practical importance, the estimation of parameters in differential equation models often gives rise to nonlinear programs that are among the hardest to solve.

8.6 Regression

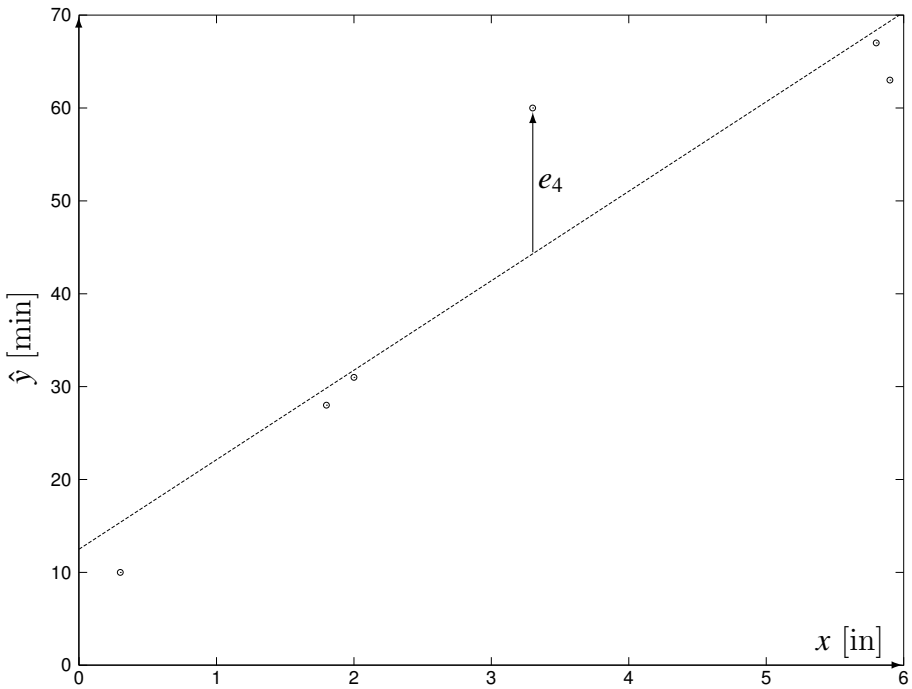
David knows from experience that if the weather is good he can wake up at 7:00 and get to work on time. If snow is forecast, however, he must set his alarm early to allow for shoveling the driveway, and how many minutes that takes varies from storm to storm. Although he can imagine several things that might affect his shoveling time, he is sure that the depth of the snow is the most important factor. Last winter he gathered this data.

storm i	snow inches x_i	shoveling minutes $\hat{y}_i$
1	0.3	10
2	5.8	67
3	2.0	31
4	3.3	60
5	5.9	63
6	1.8	28

Is there some way that David can use this information to predict his shoveling time when each storm is forecast this year?

8.6.1 One Predictor Variable

To investigate this question David plots the data, obtaining the graph below. From the picture he conjectures that his shoveling time increases as a linear function of the snowfall, with some variation resulting from the random effects of factors he didn't measure. He draws a straight line interpolating the data points, but other lines $y = ax+b$ seem equally plausible. Because the single **predictor variable** x affects the **response variable** y in a way that can be described by an equation that is linear in the coefficients a and b, the problem of finding the *best* straight line is called **simple linear regression**.



For each snow depth x_i the straight line predicts a shoveling time of $y_i = ax_i + b$. This prediction is in error by an amount $e_i = \hat{y}_i - y_i$, which can have either sign. The graph shows e_4, which happens to be positive. We could find the best least-squares fit of the line to the data by minimizing the sum of the squares of these errors.

$$\underset{a,b}{\text{minimize}} \;\; E(a,b) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} (\hat{y}_i - [ax_i + b])^2$$

Here the variables in the optimization problem, which are the slope and intercept we want to estimate, are given the names a and b, while the data of the problem are in vectors named $\mathbf{x}$ and $\mathbf{y}$. I have also used i for the index on observations and n for the number of observations. These departures from the notational conventions introduced in §8.1, which are used only here and in §8.7, are a concession to the usage that is standard in the literature on regression [123] and classification [14].

This formulation should be reminiscent of the parameter estimation problem in §8.5, because here too we are estimating the constant parameters of a model. The difference is that the parameter estimation model is a differential equation that is usually impossible to solve, while this regression model is a linear algebraic equation that is trivial to solve.

Setting the derivatives of E with respect to a and b equal to zero yields the **normal equations** boxed below, which are linear in a and b and have coefficients that are quantities we can compute from the data. In the final step I used the fact that $\sum_{i=1}^{n} 1 = n$. The limits on the summations are always the same so for simplicity I have left them out.

$$\begin{aligned}
\frac{\partial E}{\partial a} &= \frac{\partial}{\partial a}\sum(\hat{y}_i - [ax_i + b])^2 \\
&= \sum \frac{\partial}{\partial a}(\hat{y}_i - ax_i - b)^2 \\
&= \sum 2(\hat{y}_i - ax_i - b)^1(-x_i) \qquad = \quad 0
\end{aligned}$$

$$\boxed{-\sum x_i\hat{y}_i + a\sum x_i^2 + b\sum x_i \quad = \quad 0}$$

$$\begin{aligned}
\frac{\partial E}{\partial b} &= \frac{\partial}{\partial b}\sum(\hat{y}_i - [ax_i + b])^2 \\
&= \sum \frac{\partial}{\partial b}(\hat{y}_i - ax_i - b)^2 \\
&= \sum 2(\hat{y}_i - ax_i - b)^1(-1) \qquad = \quad 0
\end{aligned}$$

$$\boxed{-\sum \hat{y}_i + a\sum x_i + bn \qquad = \quad 0}$$

Solving the normal equations simultaneously yields these formulas for a and b.

$$a = \frac{\sum x_i\hat{y}_i - \frac{1}{n}\sum x_i \sum \hat{y}_i}{\sum x_i^2 - \frac{1}{n}\sum x_i \sum x_i}$$

$$b = \frac{\sum \hat{y}_i - a\sum x_i}{n}$$

David finds, using the data given at the beginning of the Section, that

$$y(x) \approx 9.634x + 12.498,$$

which is the line plotted in the graph. Thus the least-squares regression problem turns out to have a closed-form analytic solution, and the only numerical calculation it requires is evaluating the formulas for a and b.

Simple linear regression can be described in a more compact way [123, §6.10] by arranging the data and the unknown coefficients in matrices, like this.

$$\mathbf{Y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \qquad \beta = \begin{bmatrix} b \\ a \end{bmatrix}$$

Then the errors e_i are elements of the vector

$$\mathbf{e} = \mathbf{Y} - \mathbf{X}\beta = \begin{bmatrix} \hat{y}_1 - b - ax_1 \\ \hat{y}_2 - b - ax_2 \\ \vdots \\ \hat{y}_n - b - ax_n \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - (ax_1 + b) \\ \hat{y}_2 - (ax_2 + b) \\ \vdots \\ \hat{y}_n - (ax_n + b) \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \hat{y}_2 - y_2 \\ \vdots \\ \hat{y}_n - y_n \end{bmatrix}$$

and the sum-of-squares error is

$$\begin{aligned} E = \sum(\hat{y}_i - y_i)^2 &= [(\hat{y}_1 - y_1), (\hat{y}_2 - y_2) \cdots (\hat{y}_n - y_n)] \begin{bmatrix} (\hat{y}_1 - y_1) \\ (\hat{y}_2 - y_2) \\ \vdots \\ (\hat{y}_n - y_n) \end{bmatrix} \\ &= (\mathbf{Y} - \mathbf{X}\beta)^\top(\mathbf{Y} - \mathbf{X}\beta) \\ &= \mathbf{Y}^\top\mathbf{Y} - 2\beta^\top(\mathbf{X}^\top\mathbf{Y}) + (\mathbf{X}\beta)^\top(\mathbf{X}\beta). \end{aligned}$$

Setting the derivative with respect to β equal to zero we find the **matrix normal equations**, which are boxed below.

$$\begin{aligned} \nabla_\beta E = -2\mathbf{X}^\top\mathbf{Y} + 2\mathbf{X}^\top(\mathbf{X}\beta) &= \mathbf{0} \\ \boxed{\mathbf{X}^\top\mathbf{Y} - (\mathbf{X}^\top\mathbf{X})\beta = \mathbf{0}} \end{aligned}$$

Now, provided $\mathbf{X}^\top\mathbf{X}$ is nonsingular, we can find the regression coefficients like this.

$$\begin{aligned} (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{Y}) - (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{X})\beta &= \mathbf{0} \\ \beta = (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{Y}) &= \mathbf{X}^+\mathbf{Y} \end{aligned}$$

where $\mathbf{X}^+ = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ is called the **pseudoinverse** of the (non-square) matrix $\mathbf{X}$ [150, p81-82].

Once again it is clear that least-squares regression is conceptually easy, because to find the unknown parameters we only need to evaluate a formula. However, expressing the calculation in matrix form reveals that finding β entails computing a matrix inverse, either explicitly (as indicated above) or in effect (as in our algebraic solution of the scalar normal equations). That requires many arithmetic operations, which take time and introduce roundoff errors [60, p31] [30, p166-167], so in practice and especially if $\mathbf{X}^\top\mathbf{X}$ is large we might prefer to solve the boxed normal equations using Gauss elimination instead.

8.6.2 Multiple Predictor Variables

This winter our friend David has made some use of the formula we derived for predicting shoveling time based on snowfall, but his experience with blowing and drifting snow now leads him to suspect that his regression model might be improved by considering wind speed too. Some research into last winter's meteorology turned up the extra column of data in the table below.

storm i	snow inches x_{i1}	wind mph x_{i2}	shoveling minutes $\hat{y}_i$
1	0.3	0.7	10
2	5.8	11.8	67
3	2.0	4.1	31
4	3.3	6.7	60
5	5.9	11.9	63
6	1.8	3.7	28

In many practical applications of regression the response variable y depends on $p > 1$ predictor variables so the model function involves constant parameters $\beta_0 \ldots \beta_p$. To accommodate multiple predictor variables in our matrix formulation requires [123, §7] only that we adjust the parameter vector β and the matrix $\mathbf{X}$, as follows.

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1p} \\ 1 & x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \ldots & x_{np} \end{bmatrix}$$

For David's new problem $p = 2$ and we have

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} 1 & 0.3 & 0.7 \\ 1 & 5.8 & 11.8 \\ 1 & 2.0 & 4.1 \\ 1 & 3.3 & 6.7 \\ 1 & 5.9 & 11.9 \\ 1 & 1.8 & 3.7 \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} 10 \\ 67 \\ 31 \\ 60 \\ 63 \\ 28 \end{bmatrix}$$

To find $\beta^\star$, I wrote the `smneq.m` program on the next page. In MATLAB it is easy to compute the inverse of a matrix by using the `inv()` function, but as I mentioned above it is faster and more accurate to solve the matrix normal equations by using Gauss elimination. To do that I used `chol()` to perform the matrix factorization $\mathbf{X}^\top\mathbf{X} = \mathbf{U}^\top\mathbf{U}$, where $\mathbf{U}$ is upper-triangular. Then the equation $(\mathbf{U}^\top\mathbf{U})\beta = (\mathbf{X}^\top\mathbf{Y})$ can be solved in two steps, by first solving the triangular system $\mathbf{U}^\top\mathbf{z} = \mathbf{X}^\top\mathbf{Y}$ for $\mathbf{z}$ and then solving the triangular system $\mathbf{U}\beta = \mathbf{z}$ for β. I used the MATLAB backslash operator so, for example, `bta=U\z` solves $\mathbf{U}\beta = \mathbf{z}$ for β. This program uses the variable name `bta` for β to avoid confusion with the MATLAB built-in function `beta`.


```
% smneq.m: solve the matrix normal equations
clear

load -ascii snowind.dat        % read David's new data
n=size(snowind,1);             % find out how many data points
X=[ones(n,1),snowind(:,1:2)];  % construct the X matrix
Y=snowind(:,3);                % construct the Y vector

U=chol(X'*X);                  % matrix factorization
z=U'\(X'*Y);                   % forward substitution to find z
bta=U\z                        % back substitution to find bta
```

```
octave:1> smneq
bta =

   14.542
   58.265
  -24.193

octave:2> quit
```

The Octave session on the right shows the optimal regression coefficients, which yield the **multiple regression model**

$$y = 14.542 + 58.265 \times \text{inches of snow} - 24.193 \times \text{mph of wind}.$$

8.6.3 Ridge Regression

The multiple regression model we found in §8.6.2 is a good fit to the data, in that y predicts $\hat{y}$ accurately (see Exercise 8.8.32). But does it make any sense? It claims that about 58 minutes of shoveling are required to clear each inch of snow, which contradicts the data in the table on the previous page. Even worse, it says that shoveling time dramatically *decreases* with increasing wind speed while the data show exactly the opposite!

This phenomenon, which is called **multicollinearity** [123, §10.1], is unfortunately quite common in multiple regression models. It results in coefficients having extreme values that do not indicate the relative importance of the predictor variables. The β_j also have large **sampling variance**, so that next year's data might yield wildly different values. The cause of multicollinearity is a high correlation between predictor variables, which makes $\mathbf{X}^\top\mathbf{X}$ almost singular and the normal equations therefore hard to solve precisely (we will make a careful study of matrix conditioning in §18.4.2). In David's problem, $x_{i2} = 2x_{i1} + 0.1$ in every case except one, so snowfall and wind speed are almost perfectly correlated.

Statisticians know all about multicollinearity and try in constructing their regression models to avoid including predictor variables that are highly correlated. Unfortunately, when many factors are obviously important some might be mutually correlated in complicated ways that are difficult to anticipate. Eternal vigilance, while a prudent policy, is therefore not a sure cure for multicollinearity. Fortunately, **ridge regression** can help.

In **ordinary least squares** or **OLS** regression, we solve the following nonlinear program.

$$\min_{\beta} \; E = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} (\hat{y}_i - \mathbf{X}_i\beta)^2$$

in which $\mathbf{X}_i = [1, x_{i1}, \ldots, x_{ip}]$ is the i'th row of $\mathbf{X}$. If the errors in the observations y_i are independent identically-distributed random variables with mean zero, then [123, p38] the Gauss-Markov Theorem guarantees that the estimates β are **unbiased** (not systematically

over- or under-estimating the true population values) and have minimum variance among all unbiased estimators. Unfortunately, when the prediction variables are highly correlated that minimum variance can be inconveniently large.

The ridge regression formulation [26, §8.5-8.9] [153, §9.9] assumes more realistically that each observation adds to x_{ij} some error v_{ij}, where the v_{ij} are independent identically-distributed random variables with mean 0 and variance λ. To use this model we solve the nonlinear program

$$\min_{\beta} E = \mathscr{E}\left\{\sum_{i=1}^{n} (\hat{y}_i - [\mathbf{X}_i + \mathbf{V}_i]\beta)^2\right\}$$

where $\mathbf{V}_i = [0, v_{i1}, \ldots, v_{ip}]$ is a row vector of random errors and $\mathscr{E}$ denotes the expected value operator. Expanding the argument of the sum we find

$$(\hat{y}_i - [\mathbf{X}_i + \mathbf{V}_i]\beta)^2 = ([\mathbf{X}_i\beta - \hat{y}_i] + \mathbf{V}_i\beta)^2 = (\mathbf{X}_i\beta - \hat{y}_i)^2 + 2(\mathbf{X}_i\beta - \hat{y}_i)(\mathbf{V}_i\beta) + (\mathbf{V}_i\beta)^2.$$

The expectation of a sum is the sum of the expectations of the terms [153, §2.7] so

$$E = \sum_{i=1}^{n} \mathscr{E}\left\{(\mathbf{X}_i\beta - \hat{y}_i)^2\right\} + 2\sum_{i=1}^{n} \mathscr{E}\left\{(\mathbf{X}_i\beta - \hat{y}_i)(\mathbf{V}_i\beta)\right\} + \sum_{i=1}^{n} \mathscr{E}\left\{(\mathbf{V}_i\beta)^2\right\}.$$

The quantity $(\mathbf{X}_i\beta - \hat{y}_i)^2$ does not depend on the random variables v_{ij} so it is its own expectation. The v_{ij} have zero mean, so $\mathscr{E}\{\mathbf{V}_i\} = 0$ and thus $\mathscr{E}\{(\mathbf{X}_i\beta - \hat{y}_i)(\mathbf{V}_i\beta)\} = 0$. The expectation of a square is the square of the expectation plus the variance $\mathscr{V}$ [153, §2.8] so

$$\begin{aligned}\mathscr{E}\left\{(\mathbf{V}_i\beta)^2\right\} &= \mathscr{E}\left\{(0 \cdot \beta_0 + v_{i1}\beta_1 + \cdots + v_{ip}\beta_p)^2\right\} \\ &= \left[\mathscr{E}\left\{(v_{i1}\beta_1 + \cdots + v_{ip}\beta_p)\right\}\right]^2 + \mathscr{V}\left\{(v_{i1}\beta_1 + \cdots + v_{ip}\beta_p)\right\}.\end{aligned}$$

But $\mathscr{E}\left\{v_{ij}\right\} = 0$ and the variance of a constant times a random variable is the square of the constant times the variance of the random variable [153, §2.9] so

$$\mathscr{E}\left\{(\mathbf{V}_i\beta)^2\right\} = [0]^2 + \mathscr{V}\left\{v_{ij}\right\}(\beta_1^2 + \cdots + \beta_p^2) = \lambda\sum_{j=1}^{p}\beta_j^2.$$

Thus the ridge regression nonlinear program reduces to this.

$$\min_{\beta} E = \sum_{i=1}^{n} (\mathbf{X}_i\beta - \hat{y}_i)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

The regression coefficients β that solve this problem have lower variance than those produced by OLS regression, but because of the second summation or **bias term** in the objective they are no longer unbiased. Accepting some bias in exchange for a reduction in the sampling variance of β is often a worthwhile tradeoff. Because the assumed variance of the v_{ij} is seldom

actually known, and because its size affects the amount of the bias, λ is referred to as the **bias parameter**. Notice that when $\lambda = 0$ ridge regression reduces to OLS regression.

Solving the ridge regression NLP analytically we get the following $p+1$ normal equations, which can be solved for the $p+1$ regression coefficients $\beta_0, \beta_1, \ldots, \beta_p$.

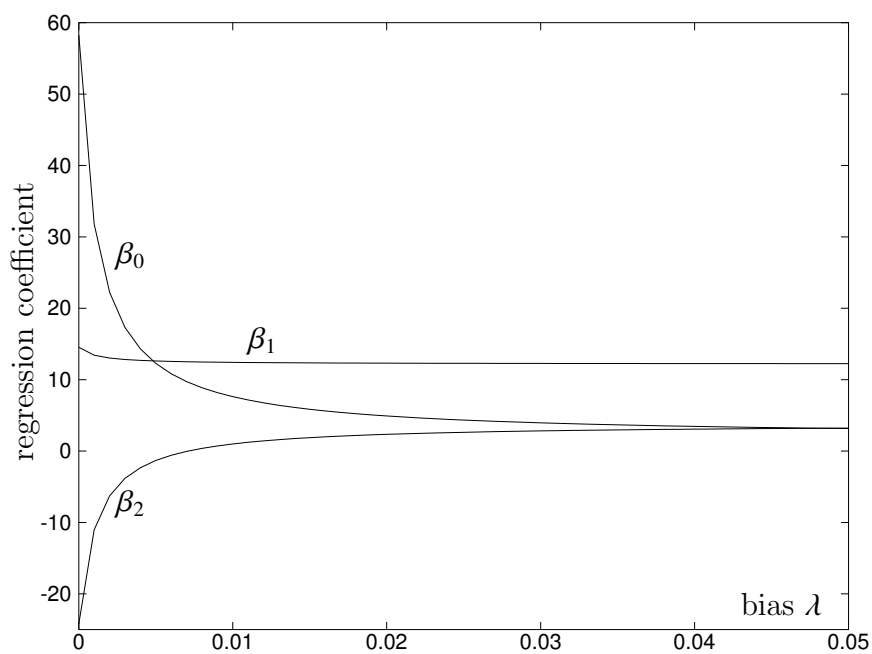
$$\begin{aligned}
\frac{\partial E}{\partial \beta_0} &= \sum_{i=1}^{n} 2(\hat{y}_i - \mathbf{X}_i\beta)^1 \cdot 1 = 0 \\
\frac{\partial E}{\partial \beta_j} &= \sum_{i=1}^{n} 2(\hat{y}_i - \mathbf{X}_i\beta)^1 \cdot x_{ij} + \lambda(2\beta_j) = 0, \quad j = 1\ldots p
\end{aligned}$$

These normal equations can be written in matrix form like this.

$$(\mathbf{X}^\top\mathbf{X} + \lambda\bar{\mathbf{I}})\beta = \mathbf{X}^\top\mathbf{Y}$$

Here $\bar{\mathbf{I}}$ is like the $(p+1) \times (p+1)$ identity matrix, except that the (1,1) element is zero because β_0 is not included in the bias term. Adding a multiple of $\mathbf{I}$ to $\mathbf{X}^\top\mathbf{X}$ also improves its conditioning so some people do that instead, thus including β_0 in the bias term even though that is not justified by the statistical argument presented above; in that case the bias term is referred to as a **regularization**.

As λ is increased from zero, the ridge regression coefficients become less extreme and converge to estimates of their true values. Increasing λ also increases the bias in those estimates, so we want to use the smallest value of λ that makes the parameter values stop changing. This subjective judgement can be guided by a **ridge trace**, which plots the β_j as functions of λ. The `ridge.m` program on the next page solves the normal equations for different values of λ and produces the graph below.




```
% ridge.m: plot the ridge trace
clear; clf; set(gca,'FontSize',25)

load -ascii snowind.dat           % read David's new data
n=size(snowind,1);                % find out how many data points
X=[ones(n,1),snowind(:,1:2)];     % construct the X matrix
Y=snowind(:,3);                   % construct the Y vector

Ibar=eye(3);                      % construct the identity
Ibar(1,1)=0;                      % zero the upper left element

for p=1:51                        % consider 51 values of lambda
    lambda(p)=0.001*(p-1);        % going from 0 to 0.05
    U=chol(X'*X+lambda(p)*Ibar);  % matrix factorization
    z=U'\(X'*Y);                  % forward substitution to find z
    bta=U\z;                      % back substitution to find bta
    b0(p)=bta(1);                 % capture the
    b1(p)=bta(2);                 % coefficient estimates
    b2(p)=bta(3);                 % to plot later
end

hold on                           % prepare to plot 3 curves
axis([0,0.05,-25,60])             % set graph axes
plot(lambda,b0)                   % plot the
plot(lambda,b1)                   % coefficient estimates
plot(lambda,b2)                   % saved earlier
hold off
print -deps -solid ridge.eps      % print the graph
```

From the ridge trace it appears that $\lambda = 0.04$ is big enough to produce reliable estimates of the coefficients, which yield this multiple regression model.

$$y = 12.256 + 3.4559 \times \text{inches of snow} + 3.0709 \times \text{mph of wind}$$

Now it takes about $3\frac{1}{2}$ minutes to shovel an inch of snow and that time is increased by wind, findings that are both plausible given the data. The fact that snowfall and wind speed are both important and affect shoveling time in the same direction makes sense because they are correlated.

8.6.4 Least-Absolute-Value Regression

In §1.5.2 we fitted a nonlinear model function to experimental data by minimizing the sum of the absolute values of the e_i, and we found that this strategy ignores outliers. The same approach can also be used to reject outliers when fitting a linear regression model, and if we apply the same transformations we get another linear program.

The data plotted in §8.6.1 contain an outlier, which pulls the least-squares regression line up so that it is above every other data point. If we reformulate that problem as a least-absolute-value or **LAV regression**, we get the standard-form linear program on the next page. Here the free regression coefficients are each written as the difference between nonnegative variables so that $a = a^+ - w$ and $b = b^+ - w$. Recall that the optimization will

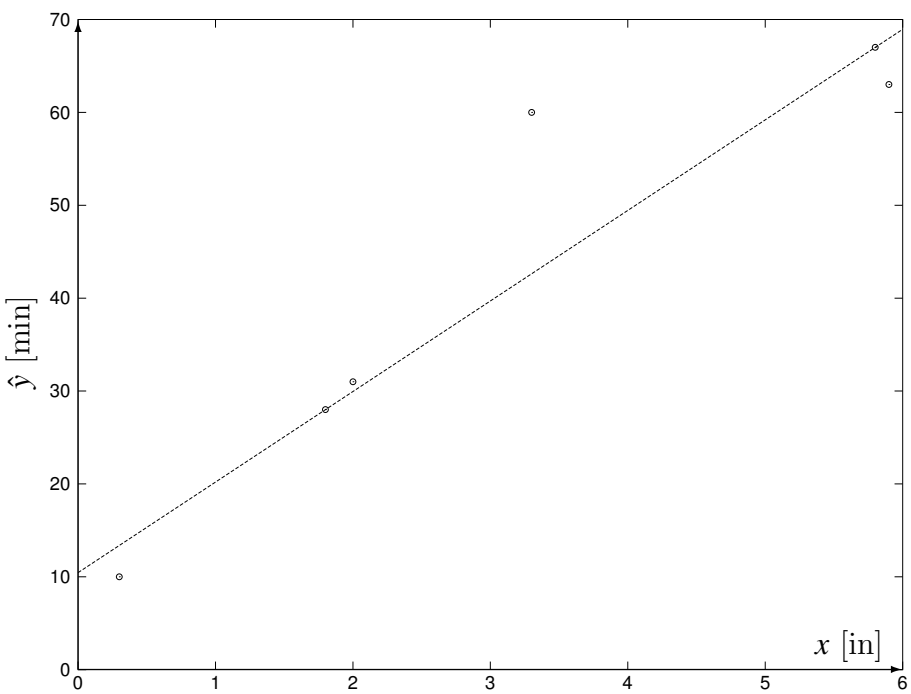
then force w to be zero if $a^+ > 0$ and $b^+ > 0$ or the absolute value of the most negative if one or both are negative. As in §1.5.2, we write each $|e_i| = u_i - v_i$ where u_i and v_i are nonnegative and the minimization will force one or the other of them to be zero.

$$\begin{array}{lrcl} \underset{a^+\ b^+\ w\ u\ v}{\text{minimize}} & E = \displaystyle\sum_{i=1}^{n} (u_i + v_i) & & \\ \text{subject to} & \left.\begin{array}{rcl} u_i - v_i & = & (a^+ - w)x_i + (b^+ - w) - \hat{y}_i \\ a^+, b^+, w, u_i, v_i & \geq & 0 \end{array}\right\} & i = 1 \ldots n \end{array}$$

I substituted the data $(x_i, \hat{y}_i)$ from the table of §8.6.0 into this formulation and used the `pivot` program to solve the linear program, obtaining the simple regression model

$$y(x) \approx 9.75x + 10.45,$$

which is plotted over the data in the graph below. This model has $p = 2$ parameters a and b, so minimizing the sum of the absolute deviations automatically selects the best two data points (here the second and fifth observations) to use in determining the LAV regression line.



Ignoring the outlier yields a fit that is probably more useful to David than the least-squares one for estimating his snow-shoveling time.

LAV regression generalizes to multiple predictor variables as follows, where $\mathbf{1}$ is a vector of n 1's.

$$\begin{array}{lll} \underset{\beta\, w\, u\, v}{\text{minimize}} & E = \sum_{i=1}^{n} (u_i + v_i) & \\ \text{subject to} & \left.\begin{array}{rcl} u_i - v_i & = & \mathbf{X}_i(\beta - w\mathbf{1}) - \hat{y}_i \\ \beta_i, w, u_i, v_i & \geq & 0 \end{array}\right\} & i = 1 \ldots n \end{array}$$

Bad conditioning of the $\mathbf{X}$ matrix due to multicollinearity is problematic in LAV regression just as it is in the least-squares formulation, and it is often dealt with in the same way by adding a regularization term. Depending on the regularization that is used, the resulting optimization problem might still be a linear program.

8.6.5 Regression on Big Data

We have seen that in a purely mathematical sense the regression problem is easy, because the OLS formulation has an explicit solution and the LAV formulation yields a linear program we can solve in a finite number of pivots.

Unfortunately there are important applications (e.g., in bioinformatics) where a response variable might depend on not just one or two predictor variables but on 1000 or 10000 or 100000. Then the $(p+1)\times(p+1)$ $\mathbf{X}$ matrix, which must be inverted or factored in solving the normal equations or linear program, contains 10^6 or 10^8 or 10^{10} elements (typically most of them zero). Even for linear systems that are well-conditioned, the growth in computing time, the fill-in of sparse matrices, and the need to manage roundoff error make direct methods such as Gauss elimination impractical when the number of rows and columns gets too big [150, §32; p325].

To solve large sparse systems of linear equations it is necessary to resort to **iterative methods**, which provide neither a formula for $\beta^\star$ nor even an exact numerical result in a finite number of iterations. These methods are classified [87, §6] as **stationary methods** such as Jacobi iteration, or **gradient methods** such as the **conjugate gradient algorithm** [4, §13.2] [5, §5] (see §14).

Gradient methods for linear systems work by minimizing some measure of the error in a trial solution, and this suggests instead simply minimizing one of our error measures E by means of any nonlinear program solver. In practice that is the approach usually taken, often using an algorithm tailor-made for the purpose (see §25.7).

8.7 Classification

Sarah wants to take Computational Optimization. She passed the one course that is an official prerequisite, so the instructor has given her permission to enroll even though she is only a junior. Now she is having second thoughts, because she wonders if the other six undergraduate math courses she has passed provide enough background for her to get a

good grade in the graduate course. To help her decide, she interviews all of the people she knows who have already taken Computational Optimization and asks each of them these two questions.

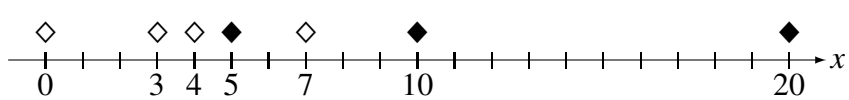
> Did you get at least a B in Computational Optimization?
> Besides the prerequisite, how many math courses had you passed before?

She arranges the results of her survey in increasing order of prior courses passed, obtaining the table below. The data justify Sarah's indecision, because one student with even less background than she has got a good grade while another with more background did not; the students who got good grades are not **separable**, based on prior experience, from those who got less than a B.

student i	prior math courses x_i	grade $\geq$ B?
1	0	no
2	3	no
3	4	no
4	5	yes
5	7	no
6	10	yes
7	20	yes

Trying to find some way to **classify** herself as belonging to one group or the other based on x, she plots the data along a line, representing a "no" response by an open $\diamond$ diamond and a "yes" response by a filled $\blacklozenge$ one, and reasons as follows.

> I have passed $\bar{x} = 6$ courses. Suppose that there is some number b such that if $\bar{x} \geq b$ I am likely to get at least a B but if $\bar{x} < b$ I am likely to get less than a B. Then b must fall between $x = 0$ and $x = 20$. Nobody has taken fewer than zero math courses beyond the prerequisite, and none of the graduate students I know have taken more than twenty. In fact, b probably falls between the highest value of x below which all the students failed ($x = 4$), and the lowest value of x above which they all succeeded ($x = 10$). It might be reasonable to set b midway between those limits, at $b = 7$. Because $\bar{x} < 7$ I fall in the "no" category, so even though I have the prerequisite I should wait until I have more math background before taking Computational Optimization.



Sarah is satisfied with this argument, but being at heart a mathematician she wonders if some formulation of the problem as an optimization might permit a more certain conclusion.

8.7.1 Measuring Classification Error

Given a trial value of b, the distance from any point x on the line to b is $f(x;b) = x - b$. Positive values of $f(x;b)$ predict success ($x > b$) while negative values mean that x is too low to ensure a good grade. For example, if b is set at 9, then Sarah's experience $\bar{x} = 6$ yields $f(\bar{x};9) = 6 - 9 = -3$, predicting that she will not succeed in getting at least a B.

The survey results can be coded as follows.

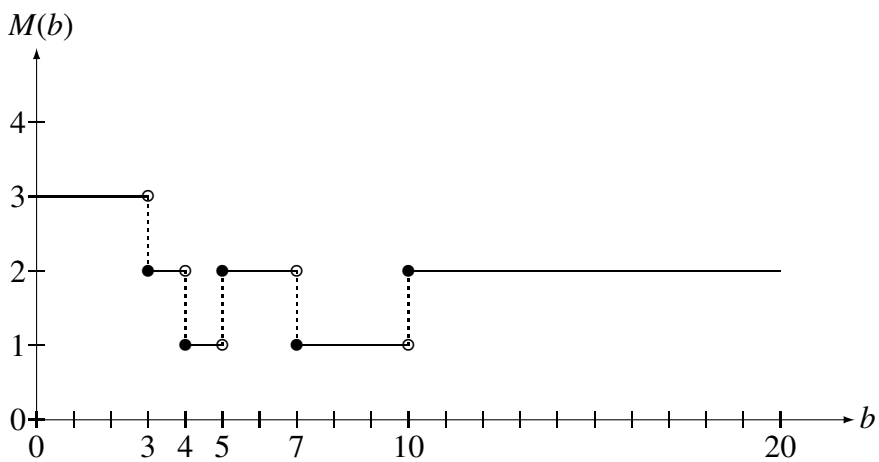
$$y_i = \begin{cases} +1 & \text{for "yes"} \\ -1 & \text{for "no"} \end{cases}$$

Then the quantity $y_i f(x_i;b)$ is nonnegative if x_i is classified correctly for that choice of b or negative if x_i is classified incorrectly. For example, if we pick $b = 9$ then $y_4 f(x_4;9) = (+1) \times (5 - 9) = -4$ meaning the "yes" point 4 is classified incorrectly; it falls on the "no" side of the **classifier** $b = 9$. On the other hand $y_7 f(x_7;9) = (+1) \times (20 - 9) = +11$ so the "yes" point 7, which falls on the "yes" side of $b = 9$, is classified correctly.

For a given value of b the total number of misclassified points can then be found as

$$M(b) = \sum_{i=1}^{n} \operatorname{sgn}(\max(0, -y_i f(x_i;b))) \quad \text{where} \quad \operatorname{sgn}(\mathrm{r}) = \begin{cases} +1 & \text{if } r > 0 \\ 0 & \text{if } r = 0 \\ -1 & \text{if } r < 0 \end{cases}$$

is **signum function**. If x_i is misclassified for a given value of b then $y_i f(x_i;b) < 0$ so $\max(0, -y_i f(x_i;b)) > 0$ and 1 gets added to the sum. If x_i is correctly classified, then $y_i f(x_i;b) \geq 0$ so $\max(0, -y_i f(x_i;b)) = 0$ and 0 gets added to the sum. Sarah computes $M(b)$ using the data from the table and gets the graph below.



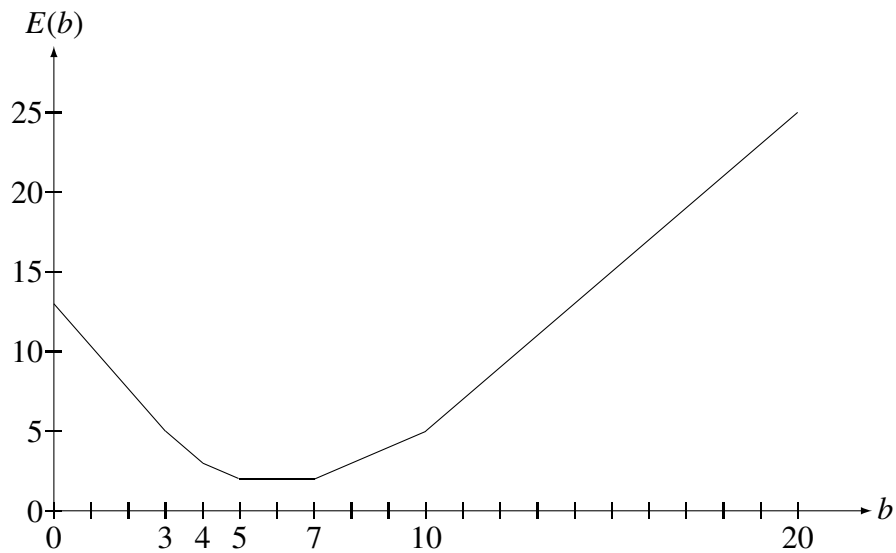
This function is piecewise constant, so it has jump discontinuities. To minimize the number of points that are misclassified, b must be chosen either between 4 and 5, in which case Sarah's $\bar{x} = 6$ falls in the "yes" region, or between 7 and 10, in which case it falls in the "no"

region. This way of looking at the problem does not make Sarah feel any more confident about what her decision should be.

Of course $M(b)$ is just a count, and does not measure the *amount* of each misclassification. Sarah's next thought is that it might be more telling to minimize the sum of the classification errors,

$$E(b) = \sum_{i=1}^{n} \max(0, -y_i f(x_i; b)).$$

The graph of $E(b)$ below is piecewise linear, so it is continuous but at the data points not differentiable. It shows that to minimize the total classification error, b should be chosen between 5 and 7. Since Sarah's experience score is $\bar{x} = 6$, looking at the problem like this does not reassure her about taking the course either.



8.7.2 Two Predictor Variables

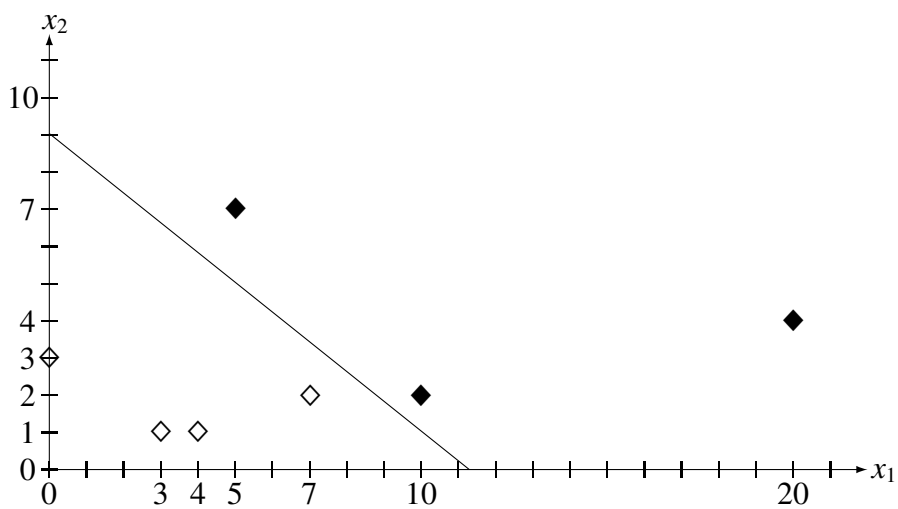
Dejected, Sarah explains to her friend David how she came to the conclusion that she should not take Computational Optimization yet. David immediately suggests that she has ignored some important factors in her analysis. "How hard did your friends work?" he wonders. Returning to the students she surveyed earlier, Sarah asks one additional question.

> "How many hours did you spend studying Computational Optimization outside of class each week?"

Including the responses in her summary, she gets the revised table on the next page.

student i	background x_{i1}	effort x_{i2}	y_i	symbol
1	0	3	−1	◇
2	3	1	−1	◇
3	4	1	−1	◇
4	5	7	+1	◆
5	7	2	−1	◇
6	10	2	+1	◆
7	20	4	+1	◆

Now a two-dimensional graph is required to represent the survey data. Lo and behold, it turns out to be possible in this space to draw many straight lines that separate the ◇ symbols from the ◆ symbols.



The classifier shown has the equation $x_2 = 9 - \frac{4}{5}x_1$ or $f(\mathbf{x}) = \frac{4}{5}x_1 + x_2 - 9 = 0$, and we can use this function to find out on which side of the hyperplane a given point falls. For example, $\mathbf{x}_4 = [5, 7]^\top$ yields $f(\mathbf{x}_4) = \frac{4}{5}\times 5 + 7 - 9 = 2 > 0$ and is therefore on the ◆ side of the hyperplane, while the point $\mathbf{x}_5 = [7, 2]^\top$ yields $f(\mathbf{x}_5) = \frac{4}{5} \times 7 + 2 - 9 = -\frac{7}{5} < 0$ so it is on the ◇ side.

If the equation of the separating hyperplane is $ax_1 + x_2 - b = 0$ then $f(\mathbf{x}; a, b) = ax_1 + x_2 - b$ measures the amount by which a point is on one side or the other, and the classifier that minimizes the total error solves this optimization problem.

$$\underset{a\,b}{\text{minimize}} \;\; E(a, b) = \sum_{i=1}^{n} \max\left(0, -y_i f(\mathbf{x}_i; a, b)\right)$$

Recall from §1.5.1 that minimizing the maximum of two linear expressions can be recast as a linear program. If we introduce variables $e_i = \max\left(0, -y_i f(\mathbf{x}_i; a, b)\right)$ then we can rewrite the problem as shown on the next page.

$$\begin{array}{lrcll} \underset{a\,b\,\mathbf{e}}{\text{minimize}} & E(a,b) & = & \displaystyle\sum_{i=1}^{n} e_i & \\ \text{subject to} & e_i & \geq & -y_i f(\mathbf{x}_i; a, b) & i = 1 \ldots n \\ & e_i & \geq & 0 & i = 1 \ldots n \end{array}$$

The minimization will ensure that at optimality each e_i is equal to the larger of $-y_k f(\mathbf{x}_i; a, b)$ and zero. For Sarah's problem we have

$$y_i f(\mathbf{x}_i; a, b) = y_i(a x_{i1} + x_{i2} - b) = (y_i x_{i1})a + (-y_i)b + (y_i x_{i2}).$$

Then each functional constraint can be rewritten as an equality by adding a slack variable s_i.

$$-e_i + s_i \quad = \quad (y_i x_{i1})a + (-y_i)b + (y_i x_{i2})$$

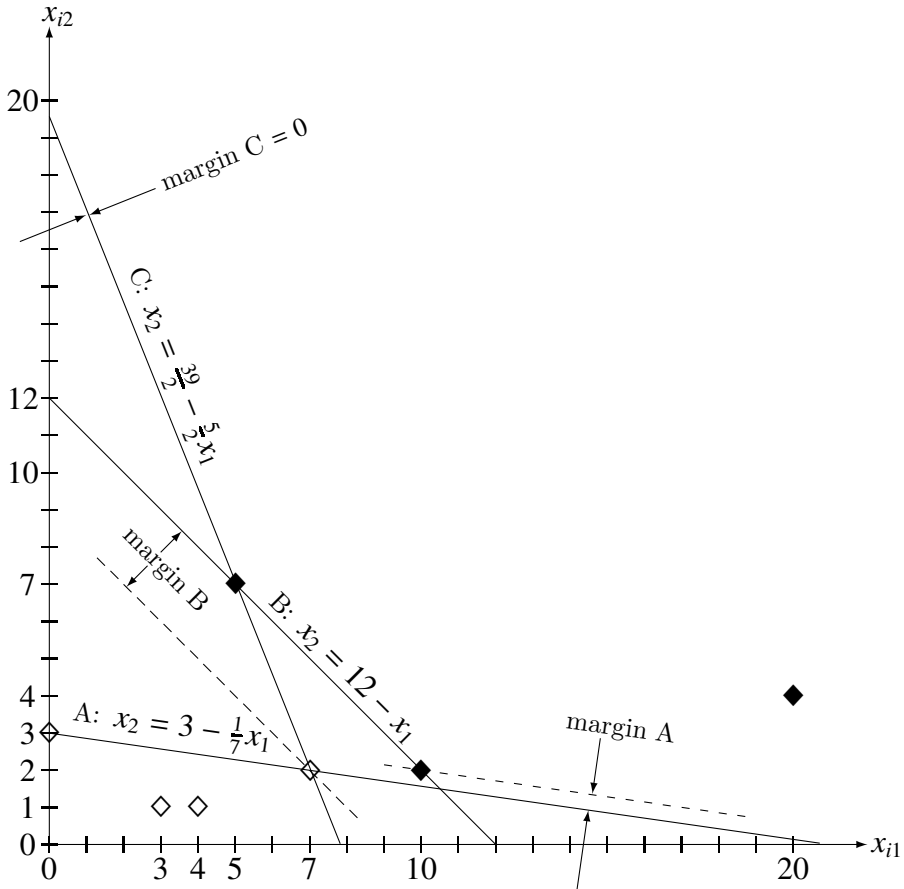
Using this result and the data from the enlarged table, the optimization becomes the standard-form linear program

$$\begin{array}{lrcl} \underset{a\,b\,w\,\mathbf{e}\,\mathbf{s}}{\text{minimize}} & e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7 & = & z \\ \text{subject to} & -e_1 + s_1 - (0)a - (+1)b & = & -3 \\ & -e_2 + s_2 - (-3)a - (+1)b & = & -1 \\ & -e_3 + s_3 - (-4)a - (+1)b & = & -1 \\ & -e_4 + s_4 - (+5)a - (-1)b & = & +7 \\ & -e_5 + s_5 - (-7)a - (+1)b & = & -2 \\ & -e_6 + s_6 - (+10)a - (-1)b & = & +2 \\ & -e_7 + s_7 - (+20)a - (-1)b & = & +4 \\ & a \geq 0,\ b \geq 0,\ \mathbf{e} \geq \mathbf{0},\ \mathbf{s} \geq \mathbf{0} & & \end{array}$$

with the following initial tableau. In general a and b must be treated as free variables, but in this problem they will be nonnegative so for simplicity this formulation assumes that.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	s_1	s_2	s_3	s_4	s_5	s_6	s_7	a	b
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
−3	−1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	−1
−1	0	−1	0	0	0	0	0	0	1	0	0	0	0	0	3	−1
−1	0	0	−1	0	0	0	0	0	0	1	0	0	0	0	4	−1
7	0	0	0	−1	0	0	0	0	0	0	1	0	0	0	−5	+1
−2	0	0	0	0	−1	0	0	0	0	0	0	1	0	0	7	−1
2	0	0	0	0	0	−1	0	0	0	0	0	0	1	0	−10	+1
4	0	0	0	0	0	0	−1	0	0	0	0	0	0	1	−20	+1

I used the `pivot` program to solve the problem and found three alternate optimal solutions, corresponding to the hyperplanes plotted on the next page.



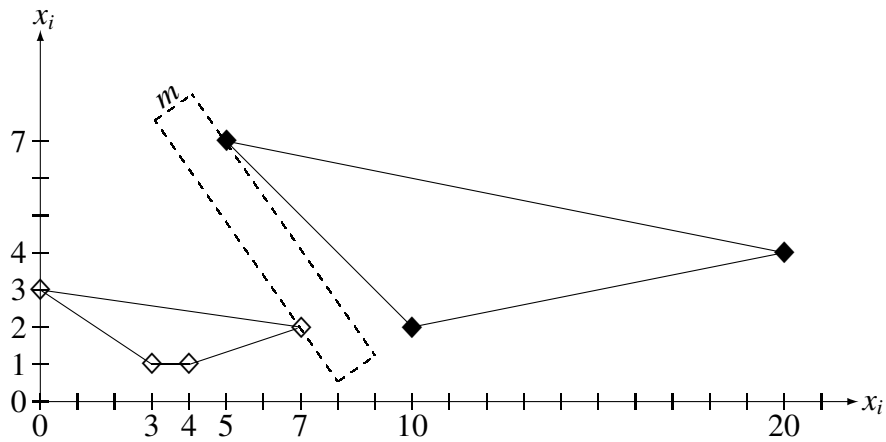
For hyperplane A, points on and below the line are ◇ while those above it are ◆. For hyperplanes B and C, points on and above the line are ◆ while those below it are ◇. In the technical sense of our formulation, each of these hyperplanes achieves a perfect separation between the ◇ and ◆ points because each solution of the linear program has $z^\star = 0$.

Sarah is reluctant to use any of the three hyperplanes as a classifier, however. Because each of them goes through two of the data points, they afford no margin for error in the classification of *new* points. Parallel to line B she draws a dashed line through $\mathbf{x}_5$ to show how far apart the two sets of points really are in that direction. This distance is called a **margin** between the convex hulls (see §3.6.1) of the two sets of points. Parallel to line A she draws a dashed line through $\mathbf{x}_6$ to show the margin in that direction. The margin for line C is zero. "The classifier I will actually use," she decides, "is a line that bisects the widest margin. That way it will be possible to classify a new point, such as one representing me, with confidence even if its coordinates are not known precisely, provided it doesn't fall exactly on the margin bisector."

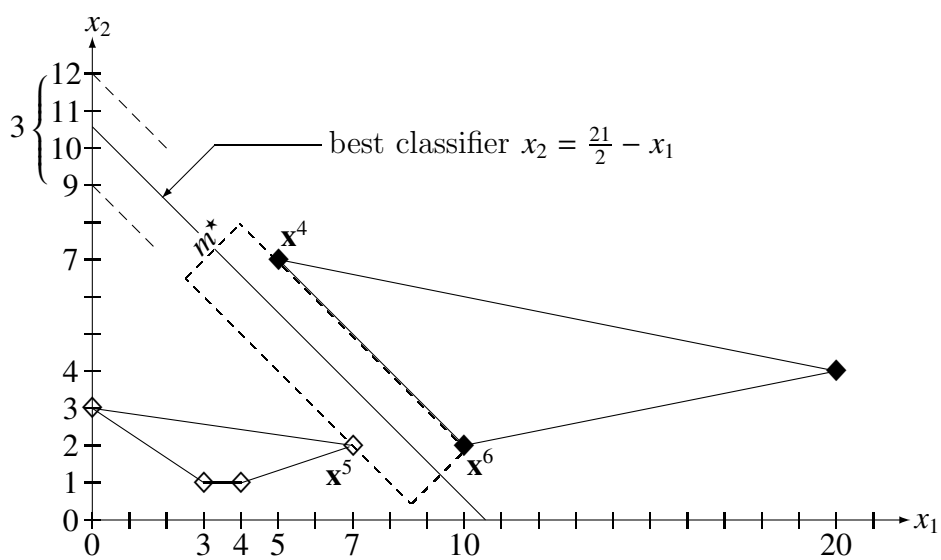
Drawing the bisector of margin B, Sarah realizes that she can be confident of getting at least a B in Computational Optimization if she is willing to study the subject outside of class for enough hours each week to locate the new data point corresponding to her on the ◆ side of that bisector (see Exercise 8.8.43).

8.7.3 Support Vector Machines

In the end Sarah chose as a classifier the hyperplane bisecting the widest margin between the two convex hulls of points. This suggests a different way of formulating classification as an optimization problem [4, §1.7.2]. Consider the graph below, in which the convex hull of the ◇ points is separated from the convex hull of the ◆ points by the dashed box whose width m is the margin of separation.

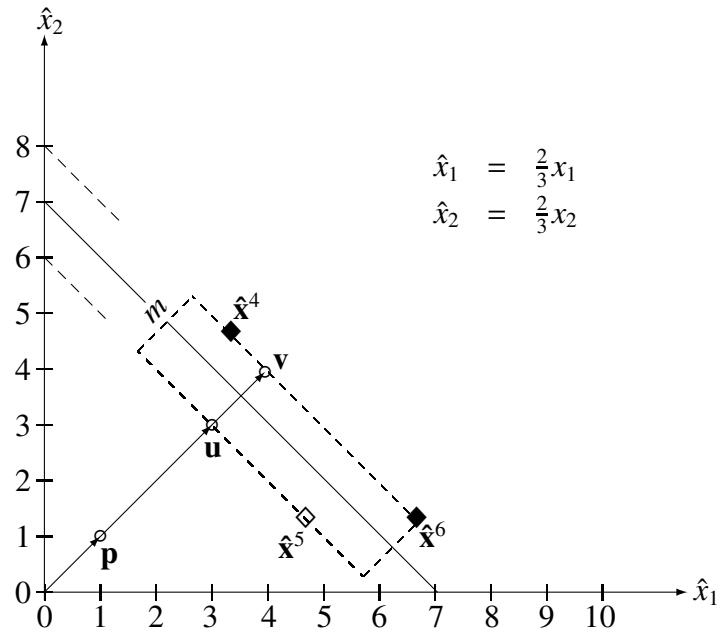


If we maximize m subject to the requirement that the box stay between the convex hulls, that will force the box to pivot into the optimal position shown below, and the bisector of the optimal margin will be the same classifier Sarah found before.



In this optimal configuration the points $\mathbf{x}^5$, $\mathbf{x}^4$, and $\mathbf{x}^6$ are called **support vectors** because the box is tangent to them. Unlike the other data points, none of the support vectors can be removed without changing the solution.

To maximize m we need to know how it depends on the coefficients of the classifier hyperplane. To derive that relationship in a general way it will be convenient to **rescale** our problem so that the $\hat{x}_2$-intercepts of the margin boundaries are separated by 2 units (in $x_1 - x_2$ space they are separated by 3 units, as shown in the bottom graph on the previous page). If we make the substitution of variables shown on the right below, only the axis labels and tic-mark values change.



Recall from §3.1 that we can describe a hyperplane by the equation

$$\mathbf{p}^\top \hat{\mathbf{x}} + q = 0.$$

If we let $\mathbf{p} = [1, 1]^\top$ and $q = -7$ then $f(\hat{\mathbf{x}}; \mathbf{p}, q) = \mathbf{p}^\top \hat{\mathbf{x}} + q = \hat{x}_1 + \hat{x}_2 - 7 = 0$ is the equation of the classifier hyperplane pictured above, and that line is orthogonal to the vector $\mathbf{p}$. These are the equations of the hyperplanes bounding the margin below and above.

$$\begin{array}{rcl} \hat{x}_1 + \hat{x}_2 - 6 &=& 0 \quad \text{or} \quad \mathbf{p}^\top \hat{\mathbf{x}} + q = -1 \\ \hat{x}_1 + \hat{x}_2 - 8 &=& 0 \quad \text{or} \quad \mathbf{p}^\top \hat{\mathbf{x}} + q = +1 \end{array}$$

In the picture I have extended $\mathbf{p}$ to intersect these hyperplanes at $\mathbf{u} = \alpha\mathbf{p}$ and $\mathbf{v} = \beta\mathbf{p}$, where

$$\begin{array}{rclclcrcl} \mathbf{p}^\top \mathbf{u} + q &=& \mathbf{p}^\top(\alpha\mathbf{p}) + q &=& -1 &\Rightarrow& \alpha &=& (-1-q)/(\mathbf{p}^\top\mathbf{p}) \\ \mathbf{p}^\top \mathbf{v} + q &=& \mathbf{p}^\top(\beta\mathbf{p}) + q &=& +1 &\Rightarrow& \beta &=& (+1-q)/(\mathbf{p}^\top\mathbf{p}) \end{array}$$

Then

$$\mathbf{v} - \mathbf{u} = \beta\mathbf{p} - \alpha\mathbf{p} = \frac{\mathbf{p}}{\mathbf{p}^\top\mathbf{p}}[(+1-q) - (-1-q)] = \frac{2\mathbf{p}}{\mathbf{p}^\top\mathbf{p}}.$$

Finally, [14] the margin is

$$m = \|\mathbf{v} - \mathbf{u}\| = \frac{2\|\mathbf{p}\|}{\|\mathbf{p}\|^2} = \frac{2}{\|\mathbf{p}\|}.$$

To maximize m, we should minimize $\|\mathbf{p}\|$, subject to the requirement that all of the points end up correctly classified. The following **support vector machine** or **SVM** does that.

$$\begin{array}{lll} \underset{\mathbf{p}\,q}{\text{minimize}} & \mathbf{p}^\top\mathbf{p} & \\ \text{subject to} & y_i(\mathbf{p}^\top\hat{\mathbf{x}}_i + q) \;\geq\; 1 & i = 1\ldots n \end{array}$$

If $y_i = +1$ and $(\mathbf{p}^\top\hat{\mathbf{x}} + q) \geq 1$, then point i is correctly classified and falls on the ◆ side of the margin; if $y_i = -1$ and $(\mathbf{p}^\top\hat{\mathbf{x}} + q) \leq 1$, then it is also correctly classified and falls on the ◇ side of the margin. The SVM finds the widest margin and returns the $\mathbf{p}$ and q that define its bisector.

For our example the SVM simplifies to this.

$$\begin{array}{lll} \underset{p_1\,p_2\,q}{\text{minimize}} & p_1^2 + p_2^2 \;=\; z & \\ \text{subject to} & y_i(p_1\hat{x}_{i1} + p_2\hat{x}_{i2} + q) \;\geq\; 1 & i = 1\ldots 7 \end{array}$$

To solve Sarah's problem numerically I used the MATLAB program `cfyrun.m` listed on the top left below, which reads the file `cfy.dat` of unscaled data listed on the bottom left. (I chose these file names because `class` and `classify` are reserved words in MATLAB.) The program invokes the built-in function `sqp()` that we used in §8.3.1, and `sqp()` in turn invokes the routines `cfyobj.m` and `cfygeq.m` that are listed on the right.

```
% cfyrun.m: classify using SVM
clear
global X Y

% read and scale the problem data
load -ascii cfy.dat
X=(2/3)*cfy(:,1:2);
Y=cfy(:,3);

% solve the SVM problem
pqzero=[0;0;0];
pqstar=sqp(pqzero,@cfyobj,[],@cfygeq)
```

```
% cfy.dat: unscaled classification data
0  3 -1
3  1 -1
4  1 -1
5  7  1
7  2 -1
10 2  1
20 4  1
```

```
% cfyobj.m: SVM objective function
function z=cfyobj(pq)
  p=pq(1:2);
  z=p'*p;
end
```

```
% cfygeq.m: SVM inequality constraints
function h=cfygeq(pq)
  global X Y
  p=pq(1:2);
  q=pq(3);
  h=[ Y(1)*(p(1)*X(1,1)+p(2)*X(1,2)+q)-1
      Y(2)*(p(1)*X(2,1)+p(2)*X(2,2)+q)-1
      Y(3)*(p(1)*X(3,1)+p(2)*X(3,2)+q)-1
      Y(4)*(p(1)*X(4,1)+p(2)*X(4,2)+q)-1
      Y(5)*(p(1)*X(5,1)+p(2)*X(5,2)+q)-1
      Y(6)*(p(1)*X(6,1)+p(2)*X(6,2)+q)-1
      Y(7)*(p(1)*X(7,1)+p(2)*X(7,2)+q)-1 ];
end
```



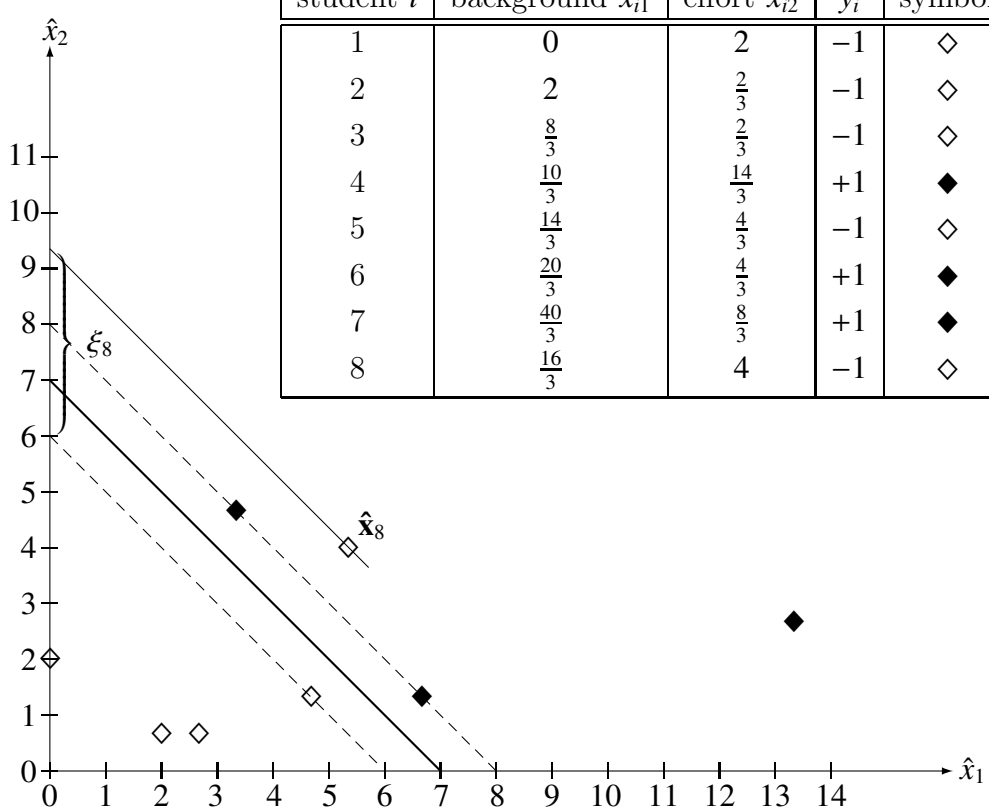
```
octave:1> cfyrun

pqstar =

   1.0000
   1.0000
  -7.0000

octave:2> quit
```

This Octave session reports $\mathbf{p}^\star = [1, 1]^\top$ and $q^\star = -7$, which are the parameters of the optimal classifier in $\hat{x}_1 - \hat{x}_2$ space. The corresponding hyperplane in $x_1 - x_2$ space has $q = \frac{3}{2} \times -7 = -\frac{21}{2}$, as we found using the linear programming formulation, but it has the *same* $\mathbf{p}$.

8.7.4 Nonseparable Data

One of the students Sarah queried is late in responding but does finally send her the data $x_{81} = 8$, $x_{82} = 6$, $y_8 = -1$. When this point is included and all of the data are scaled as described in §8.7.3, we get the enlarged table and new graph below.

student i	background $\hat{x}_{i1}$	effort $\hat{x}_{i2}$	y_i	symbol
1	0	2	-1	◇
2	2	$\frac{2}{3}$	-1	◇
3	$\frac{8}{3}$	$\frac{2}{3}$	-1	◇
4	$\frac{10}{3}$	$\frac{14}{3}$	$+1$	◆
5	$\frac{14}{3}$	$\frac{4}{3}$	-1	◇
6	$\frac{20}{3}$	$\frac{4}{3}$	$+1$	◆
7	$\frac{40}{3}$	$\frac{8}{3}$	$+1$	◆
8	$\frac{16}{3}$	4	-1	◇



Now the ◇ points are no longer linearly separable from the ◆ ones, so no matter what hyperplane we draw some of the points will be misclassified. For the classifier and margin

that we determined earlier, the new point has the classification error ξ_8 shown, and in general we can let ξ_i denote the amount by which point i violates that side of the margin on which it would fall if it were correctly classified. The **soft-margin SVM** generalizes our earlier formulation to accommodate data like this that are not perfectly separable.

$$\begin{array}{lrcll} \underset{\mathbf{p}\,q\,\boldsymbol{\xi}}{\text{minimize}} & \mathbf{p}^\top\mathbf{p} + c\sum_{i=1}^{n}\xi_i & & & \\ \text{subject to} & y_i(\mathbf{p}^\top\hat{\mathbf{x}}_i + q) & \geq & 1-\xi_i & i = 1\ldots n \\ & \xi_i & \geq & 0 & i = 1\ldots n \end{array}$$

The **compromise parameter** $c > 0$ expresses the weight that we attach to minimizing misclassifications, relative to the conflicting goal of achieving the widest possible margin. The minimization will make each classification error ξ_i just big enough to satisfy the constraints, and how big that is will depend on $\mathbf{p}^\star$ and hence on the value of c.

For our example the soft-margin SVM simplifies to this nonlinear program.

$$\begin{array}{lrcll} \underset{p_1\,p_2\,q\,\boldsymbol{\xi}}{\text{minimize}} & p_1^2 + p_2^2 + c\sum_{i=1}^{8}\xi_i & = & z & \\ \text{subject to} & y_i(p_1\hat{x}_{i1} + p_2\hat{x}_{i2} + q) & \geq & 1-\xi_i & i = 1\ldots 8 \\ & \xi_i & \geq & 0 & i = 1\ldots 8 \end{array}$$

To experiment with this model I wrote the MATLAB program `cfysrun.m` listed on the next page. It reads the `cfys.dat` file listed below and invokes `sqp()` with pointers to the routines `cfysobj.m` and `cfysgeq.m` listed below, producing the output shown at the bottom of the next page.

```
% cfysobj.m: soft-margin SVM objective
function z=cfysobj(pqxi)
  global c
  p=pqxi(1:2);
  q=pqxi(3);
  sxi=0;
  for i=1:8
      sxi=sxi+pqxi(3+i);
  end
  z=p'*p + c*sxi;
end
```

```
% cfys.dat
0  3 -1
3  1 -1
4  1 -1
5  7  1
7  2 -1
10 2  1
20 4  1
8  6 -1
```

```
% cfysgeq.m: soft-margin SVM constraints
function h=cfysgeq(pqxi)
  global X Y
  p=pqxi(1:2);
  q=pqxi(3);
  xi=pqxi(4:11);
  h=[ Y(1)*(p(1)*X(1,1)+p(2)*X(1,2)+q)-1+xi(1)
      Y(2)*(p(1)*X(2,1)+p(2)*X(2,2)+q)-1+xi(2)
      Y(3)*(p(1)*X(3,1)+p(2)*X(3,2)+q)-1+xi(3)
      Y(4)*(p(1)*X(4,1)+p(2)*X(4,2)+q)-1+xi(4)
      Y(5)*(p(1)*X(5,1)+p(2)*X(5,2)+q)-1+xi(5)
      Y(6)*(p(1)*X(6,1)+p(2)*X(6,2)+q)-1+xi(6)
      Y(7)*(p(1)*X(7,1)+p(2)*X(7,2)+q)-1+xi(7)
      Y(8)*(p(1)*X(8,1)+p(2)*X(8,2)+q)-1+xi(8)
      xi(1)
      xi(2)
      xi(3)
      xi(4)
      xi(5)
      xi(6)
      xi(7)
      xi(8) ];
end
```

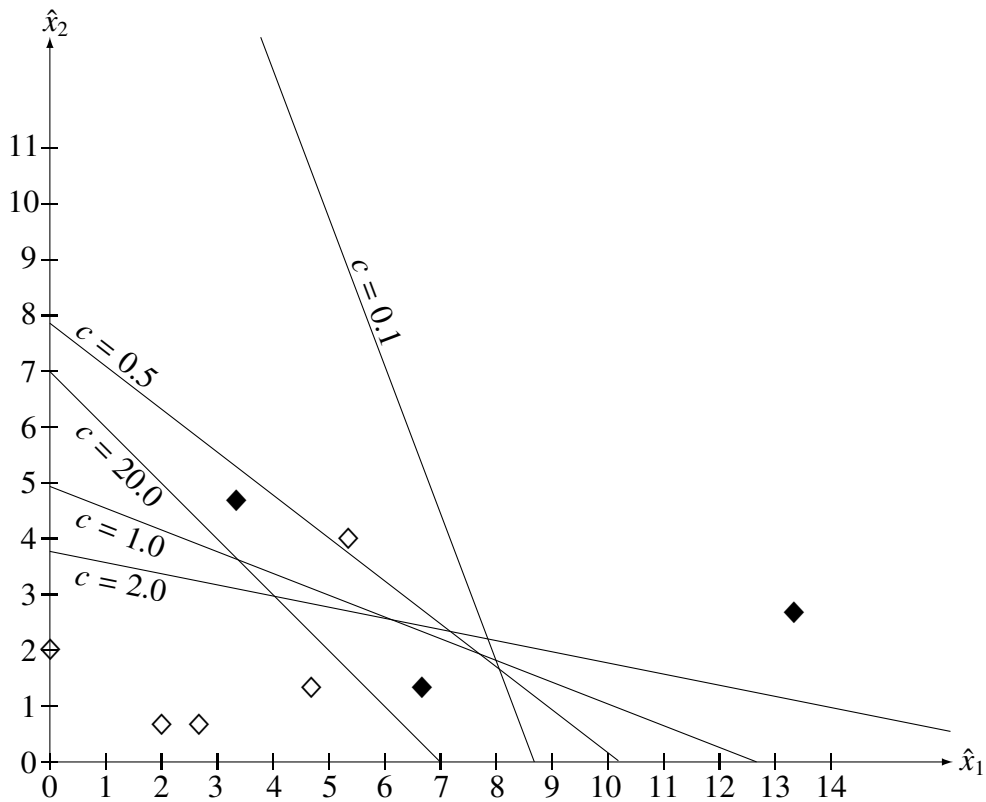


```
% cfysrun.m: classify using soft-margin SVM
clear; clf; set(gca,'FontSize',20)
global X Y c

% read and scale the enlarged problem data
load -ascii cfys.dat
X=(2/3)*cfys(:,1:2);
Y=cfys(:,3);

% use these values of the compromise parameter
cs(1)=0.1;
cs(2)=0.5;
cs(3)=1.0;
for k=4:49
  cs(k)=0.4*(k+1);
end

% solve the soft-margin SVM for the tabled values of c
printf('  c    intercepts   margin xi\n')
for k=1:49
  c=cs(k);                                      % set c
  pqxizero=zeros(11,1);                         % starting point
  pqxistar=sqp(pqxizero,@cfysobj,[],@cfysgeq);  % solve the NLP
  p=pqxistar(1:2);                              % extract p*
  xi=pqxistar(4:11);                            % extract xi*
  m(k)=2/norm(p);                               % save margin
  tce(k)=0;                                     % save
  for i=1:8                                     %   total
      tce(k)=tce(k)+xi(i);                      %   classification
  end                                           %   error

% print some of the results
  if(k <= 4 || k == 49)
     q=pqxistar(3);                             % extract q*
     x1=-q/p(1);                                % x1-intercept
     x2=-q/p(2);                                % x2-intercept
     printf('%5.1f  %5.2f %5.2f  %5.2f',c,x1,x2,m(k))
     for i=1:8
         printf('%6.2f',xi(i))
     end
     printf('\n')
  end
end

% plot error and margin as functions of c
hold on
plot(cs,tce)
plot(cs,m)
hold off
print -deps -solid cfys.eps
```

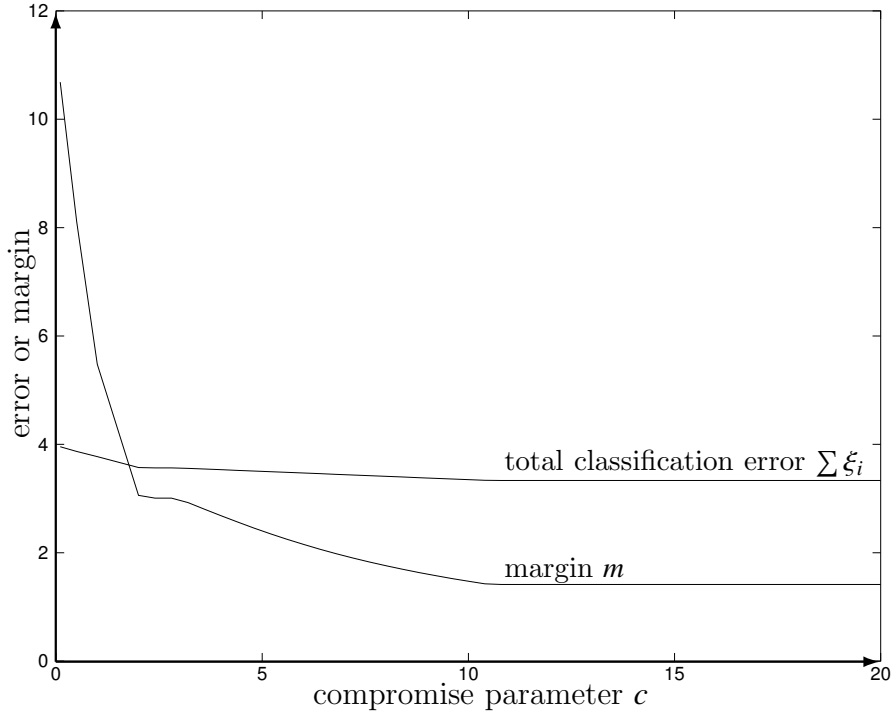
```
octave:1> cfysrun
  c    intercepts   margin xi
  0.1   8.63 22.84  10.69  0.00  0.00  0.00  1.62  0.39  1.26  0.00  0.69
  0.5  10.15  7.86   8.11  0.00 -0.00  0.00  1.12  0.43  1.27  0.00  1.05
  1.0  12.63  4.94   5.48  0.00  0.00 -0.00  0.65  0.39  1.34  0.00  1.39
  2.0  18.87  3.77   3.06  0.00  0.00  0.00  0.00  0.03  1.71  0.00  1.83
 20.0   7.00  7.00   1.41  0.00  0.00  0.00  0.00  0.00  0.00  0.00  3.33
octave:2> quit
```


The printed output lists, for each of five values of c, the x_1- and x_2-intercepts of the classifying hyperplane, the margin, and the resulting classification errors of the eight data points. If $c = 20$ (we insist on minimizing the total classification error) we get the classifier we found for the separable case, which ignores the extra point with an error of $\xi_8 = 3\frac{1}{3}$ as pictured in the graph at the beginning of this Subsection. As c is reduced it becomes possible to obtain successively wider margins, but at the cost of misclassifying more points. The extra point, which made the data nonseparable, is misclassified in each of these solutions.

The hyperplanes are plotted below to show that quite different classifiers result from using the different values of c. For clarity the corresponding margins are not shown, but if they were we could confirm graphically the misclassifications indicated by values of $\xi_i > 0$ in the printed output (see Exercise 8.8.46).



The program `cfysrun.m` also produced the graph on the next page, which shows how the margin and the total classification error both decrease with increasing c, up to a critical value (of about 11) above which the classifier does not change. For this example a large increase in margin can be had in exchange for a small increase in the total classification error, but deciding what value of c yields the classifier that is most useful in practice is ultimately a subjective judgment that depends on the particular application.



8.7.5 Classification on Big Data

In a purely mathematical sense the perfect-separation and soft-margin SVM models are both, like the regression models we studied in §8.6, easy. Although they have quadratic objectives and inequality constraints, their feasible sets are polyhedra and under assumptions that are usually satisfied they have unique solutions; see §22.

Unfortunately, there are important applications (e.g., in data mining) where the number n of points to be classified is not seven or eight but 1000 or 10000 or 100000. Because there are either n or $2n$ constraints, and in the case of soft-margin SVM n error variables ξ_i in addition to the classification variables $\mathbf{p}$ and b, the nonlinear program quickly becomes daunting as the size of the classification problem increases. Most real problems also have more than the two predictor variables $\mathbf{x}$ we considered, and in big data applications there might be many. Sometimes data that are not linearly separable are *nonlinearly* separable by the use of **kernel methods** [4, §14.8.5]. Practical algorithms for these problems are often based on the theory of nonlinear programming duality (see §16.9) and their development is an active area of research.

8.8 Exercises

8.8.1 [E] Give a concise statement of the *nonlinear programming problem.*

8.8.2 [E] Where in this textbook are example nonlinear programs, such as the `garden` problem, cataloged? What characteristics of each problem are described in its catalog entry?

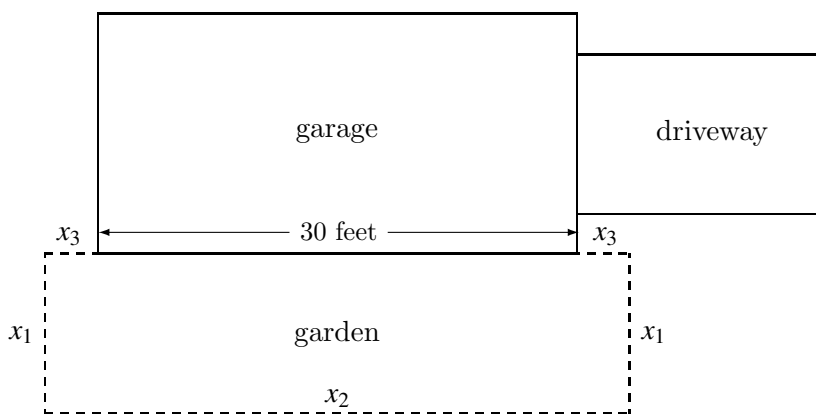
8.8.3 [H] The `garden` problem is essentially nonlinear, in that it cannot be linearized without fundamentally changing its character. Give an example of an optimization model that is nonlinear but can reasonably be approximated by a linear program over some range of parameter values.

8.8.4 [E] State the standard form that this book uses for a nonlinear program. Give an example to show that a problem including equality constraints can be stated in this standard form. Can this standard form be used to describe a problem in which some of the variables are required to be nonpositive? If so, explain how.

8.8.5 [E] Prove that $g(\mathbf{x}) = 0$ if and only if $g(\mathbf{x}) \leq 0$ and $g(\mathbf{x}) \geq 0$.

8.8.6 [E] The simplex method assumes and implicitly enforces the requirement that each variable be nonnegative. Is this also true of numerical algorithms for nonlinear programming?

8.8.7 [H] The statement of the `garden` problem requires that one side of the enclosure be provided by the garage wall. If more fencing were available, might it be possible to enclose a larger area by relaxing that constraint and making the garden look like this?



(a) Formulate a new nonlinear program that assumes the fencing extends a distance x_3 feet on each side of the garage wall, as shown, and that 200 feet of fencing are available. (b) Find a feasible $\mathbf{x}$, by any means you like, that yields the largest area for this configuration. Does $x_3^\star$ turn out to be zero?

8.8.8 [E] Show how the bounds $0 \leq x_1 \leq 20$ and $0 \leq x_2 \leq 30$ can be deduced from the constraints of the `garden` problem.

8.8.9 [H] In our graphical solution of the `garden` problem we found that the nonnegativity constraints are slack at $\mathbf{x}^\star$. (a) Show that if these constraints are removed from the problem the optimal value of the objective function is $+\infty$. (b) Is it ever true in a *linear* program that removing a constraint that is inactive at $\mathbf{x}^\star$ allows a different point to become optimal? If so, give an example; if not, explain why not.

8.8.10 [H] Revise the §1.2 procedure so that it works for *non*linear programs having $n = 2$.

8.8.11 [E] State all of the ways you can think of in which the feasible set of a nonlinear program can be different from that of a linear program. Is the feasible set of a linear program also a closed set? Where in the feasible set must the optimal point of a nonlinear program be?

8.8.12 [E] State a nonlinear program that is feasible and bounded but does not have an optimal point.

8.8.13 [H] A nonlinear programming model includes either (a) the constraints on the left [151, p514] or (b) the constraints on the right [1, p26].

$$\begin{aligned} -x &\le 0 \\ x-1 &\le 0 \\ x(1-x) &\le 0 \end{aligned} \qquad\qquad x(x-1)(x-2) = 0$$

What effect do these constraints have on the optimal value that will be found for x? (c) Can the conditions that they place on x be handled in a different or better way? Explain.

8.8.14 [E] What is a *Lagrangian*, and where is it used in nonlinear programming? State one advantage the KKT method has over the method of Lagrange.

8.8.15 [E] Name three non-graphical analytic methods for solving nonlinear programs. Which of these are guaranteed always to discover an optimal solution? Which of them can yield points that are *not* optimal? What role can computer algebra systems such as Maple play in the use of these methods?

8.8.16 [E] What is *black-box software*? Explain its virtues and drawbacks. State two possible ways of accessing black-box software for nonlinear programming, and describe the mechanism that each uses for specifying the problem to be solved.

8.8.17 [E] Name one stand-alone industrial-strength program for solving nonlinear optimization problems. Name one Octave function for solving nonlinear optimization problems.

8.8.18 [E] What effect does the starting point have on the behavior of black-box nonlinear program solvers?

8.8.19 [E] If an optimization is just one step in a larger calculation, would it be easier to solve it by using NEOS or by using MATLAB?

8.8.20 [E] Why is it sometimes advantageous to write custom software for nonlinear programming, rather than relying on black-box software? What computer programming languages are typically used for writing custom nonlinear program solvers?

8.8.21 [E] State one field in which nonlinear programming plays a role, and describe a likely application of nonlinear programming in that field.

8.8.22 [E] What is a *synthetic* test problem, and how do synthetic problems differ from application problems? Where in this book can you find a list of nonlinear programming test problem collections?

8.8.23 [E] If a forward problem is integrating a differential equation that contains a fixed parameter, what is the inverse problem?

8.8.24 [H] In solving a parameter estimation problem, why is it customary to define the residual as the sum of the squares of the errors, rather than as the sum of the errors or the sum of their absolute values?

8.8.25 [H] In §8.5 we considered the problem of estimating the gravitational acceleration g from measurements of the angle of a pendulum at several times, and we found that this yields a type-2 nonlinear program. But if θ is sufficiently small, then $\sin(\theta) \approx \theta$. (a) Use this approximation to simplify the initial value problem, and show that the simplified problem is satisfied by $\theta(t) = \theta_0 \cos(\omega t)$ where $\omega = \sqrt{g/r}$. (b) Use this result to construct a type-1 nonlinear program whose solution would approximate $g^\star$.

l	time t_l (sec)	angle $\hat{\theta}_l$ (radians)
0.0	0	$0.150 = \theta_0$
0.1	5	0.028
0.2	10	−0.135
0.3	15	−0.077

(c) Using the data given in the table above and the pendulum length $r = 10$ feet, solve the nonlinear program by one of the solution techniques exhibited in §8.2.

8.8.26 [E] Explain the difference between a type-1 and a type-2 nonlinear program.

8.8.27 [E] How does linear regression differ from the problem of estimating the parameters in a differential-equation model?

8.8.28 [H] What are *normal equations*? By using the definitions of $\mathbf{Y}$, $\mathbf{X}$, and β given in §8.6.1, show that the matrix normal equations are equivalent to the scalar normal equations.

8.8.29 [E] What is the *pseudoinverse* of the nonsquare matrix $\mathbf{X}$? Why might it be preferable to use Gauss elimination to solve a least-squares system, rather than explicitly computing the pseudoinverse and then premultiplying by it?

8.8.30 [E] In the §8.6.2 matrix formulation of the multiple regression problem, why is the first column of the $\mathbf{X}$ matrix all 1's?

8.8.31 [P] Write a MATLAB program that uses the `chol()` function to factor a matrix of your choice, and confirm that the product of the factors yields the original matrix. Does the factorization work for *every* matrix?

8.8.32 [P] Modify the `smneq.m` program of §8.6.2 to compute the shoveling time predicted by the multiple regression model for each data point, and compare these numbers to the measured $\hat{y}$ values. Is the model a good representation of the data?

8.8.33 [E] What is *multicollinearity,* what are its causes, and how can its pernicious effects be mitigated?

8.8.34 [E] The coefficients β produced by OLS regression are unbiased, while those produced by ridge regression are biased. Why would we ever prefer biased estimates?

8.8.35 [E] What value of the bias parameter λ makes ridge regression equivalent to OLS regression? What value should be used in practice? Explain the function of a ridge trace.

8.8.36 [E] How can *outliers* be rejected in fitting a linear regression model? How do LAV and OLS regression differ?

8.8.37 [H] Multicollinearity can be dealt with in LAV multiple regression by adding a regularization term as in ridge regression. Propose a regularization that permits the model to still be stated as a linear program, and state the linear program.

8.8.38 [P] Write a MATLAB program to plot the signum function $\operatorname{sgn}(x)$ for $-2 \le x \le 2$. Use the built-in function, then write your own, and show that they produce the same results.

8.8.39 [H] Give an algebraic condition that must be satisfied in order for two sets of points to be linearly separable. What is a *classifier*? If two sets of points are not linearly separable on the basis of one predictor variable, might they be separable on the basis of two? Must they be?

8.8.40 [H] Explain how minimizing the maximum of 0 and $f(x)$ is equivalent to minimizing e subject to the constraints that $e \ge f(x)$ and $e \ge 0$.

8.8.41 [P] Starting from the tableau given in §8.7.2, use `pivot` or some other program of your choice to solve the linear program, and show that you find the three alternate optima discussed there. Why are these hyperplanes not ideal for use as classifiers?

8.8.42 [E] What do we mean by the *margin* between two sets of points? Does its width depend on the direction in which we look?

8.8.43 [E] In §8.7.2 Sarah decided that she could take the course Computational Optimization if she is willing to study the subject for a certain number of hours each week outside of class. How many hours is that? Does that seem enough in view of the analysis in §8.7.3?

8.8.44 [E] What are *support vectors*? What is a *support vector machine*?

8.8.45 [H] A classification problem can always be rescaled so that its margin in the direction $\mathbf{p}$ is $2/\|\mathbf{p}\|$. Is it necessary to actually perform this rescaling in order to solve the problem using a support vector machine? Modify `cfysrun.m` to solve the example problem without scaling the data, and explain how the results change.

8.8.46 [P] In §8.7.4 we plotted soft-margin SVM classifiers corresponding to five different values of the compromise parameter c. For the hyperplane corresponding to $c = 0.5$ plot dashed lines bounding the margin and use the resulting picture to confirm that points $\mathbf{x}^4$, $\mathbf{x}^5$, $\mathbf{x}^6$, and $\mathbf{x}^8$ are misclassified.

8.8.47 [E] In a soft-margin SVM, what happens if the compromise parameter c is made very big? What happens if c is made very small? Does it make sense for c to be zero?

8.8.48 [H] The optimization theory, algorithms, and software discussed in this book are, in the abstract, of purely intellectual interest, but like every technology mathematical programming has applications that are profoundly value-laden (see, e.g., [3, p1-2] [151, p9]). In particular, unethical uses of big data by business and government have been widely, and rightly, condemned (see, e.g., [171] [172] [165]). Discuss the moral implications of using optimization techniques to extract actionable information from large sets of personal data such as credit card transactions, medical records, and the geographical locations from which cellphone calls are made. Are there noble and worthy uses for the information extracted from such personal data? List some venal and destructive uses. Is there some way to permit the good uses while preventing the bad ones?

9

Nonlinear Programming Algorithms

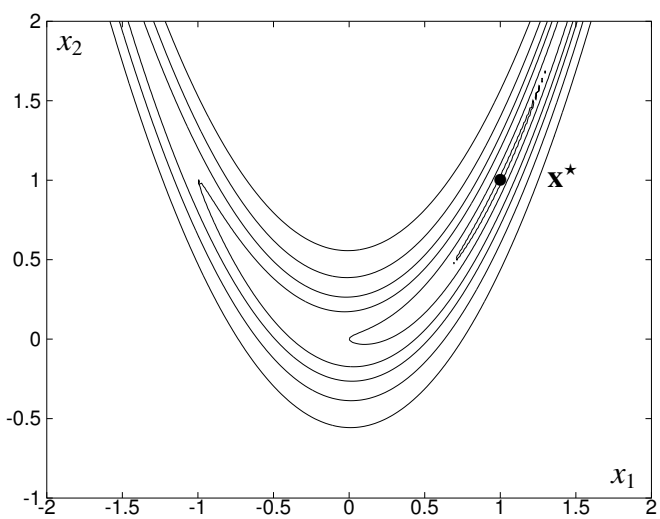
In §8, I used the `garden` problem to illustrate several different ways of solving nonlinear programs. The analytic techniques mentioned there are seldom useful in practice, but they will occupy us in §9.3, §15, and §16 because they provide the motivation and conceptual basis for the **numerical methods** that will be our main focus. A numerical method is an iterative **algorithm** [94, §1.1] [161, §4.3] or mechanical procedure that approximates the solution to a mathematical problem by performing only arithmetic and logical operations. This Chapter is about certain properties that are shared by all numerical optimization methods.

9.1 Pure Random Search

The most obvious numerical methods for unconstrained optimization are based on evaluating the objective at points that are chosen arbitrarily. To see how this idea works, consider the Rosenbrock problem (see §28.7.2) which I will refer to from now on as `rb`.

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

This classic is easy to state, trivial to solve analytically, and notoriously troublesome for numerical methods that do *not* chose points arbitrarily. Both terms in the objective are squares so $f(\mathbf{x})$ is never negative, and it's easy to see that $f(\mathbf{x})$ is zero only at $\mathbf{x}^\star = [1, 1]^\top$. The contour plot below, which was produced by the `plotrb.m` program listed on the next page, reveals why the `rb` objective is sometimes referred to as the "banana function."




```
 1 % plotrb.m: plot contours of the rb objective
 2 clear; clf; set(gca,'FontSize',30)
 3
 4 % compute the function value on a grid of points
 5 xl=[-2;-1];
 6 xh=[2;2];
 7 ng=200;
 8 [xc,yc,zc,zmin,zmax]=gridcntr(@rb,xl,xh,ng);
 9
10 % plot some contours
11 hold on
12 axis([-2,2,-1,2]);
13 vc=[0.1,1,4,8,16,32];
14 contour(xc,yc,zc,vc)
15
16 % print the resulting graph
17 print -deps -solid rb.eps
```

The `plotrb.m` program invokes [8] `gridcntr.m` to compute the value of $f(\mathbf{x})$ at $\mathtt{ng} \times \mathtt{ng} = 40000$ points in a box bounded by the lower and upper bounds [5] `xl` and [6] `xh`. Then it [13] sets some contour levels and [14] invokes the MATLAB `contour` function to draw the contour diagram. Finally it [17] prints the graph so that I could include it on the previous page.

```
 1 function [xc,yc,zc,zmin,zmax]=gridcntr(fcn,xl,xh,ng)
 2 % evaluate fcn(x) at grid points equally spaced in [xl,xh]
 3   zmax=-realmax;
 4   zmin=+realmax;
 5   for i=1:ng
 6       xc(i)=xl(1)+(xh(1)-xl(1))*((i-1)/(ng-1));
 7       for j=1:ng
 8           yc(j)=xl(2)+(xh(2)-xl(2))*((j-1)/(ng-1));
 9           x=[xc(i);yc(j)];
10           zc(j,i)=fcn(x);
11           zmax=max(zmax,zc(j,i));
12           zmin=min(zmin,zc(j,i));
13       end
14   end
15 end
```

The `gridcntr` routine used in `plotrb.m` figures out the x_1-coordinates `xc` [6] and the x_2-coordinates `yc` [8] of the grid points, and invokes `fcn` [10] (here `rb`) to compute the corresponding values `zc` of the objective. In addition to those vectors, `gridcntr.m` returns [1] the extreme values `zmin` and `zmax` that the objective takes on at the grid points. We will be drawing many contour diagrams and will make extensive use of this routine.

```
 1 % compute one value of the Rosenbrock function
 2 function f=rb(x)
 3   f=100*(x(2)-x(1)^2)^2+(1-x(1))^2;
 4 end
```

The `rb` routine computes the value of $f(\mathbf{x})$ at a single point by evaluating the formula given earlier. The `rb` problem will be of continuing interest, so this function will also be used again.

If the optimal point of this problem had not been obvious from the formula, we could have found it in the plot of the contours or by sorting through the grid of function values that Octave used to draw them. This brute-force algorithm is called a **grid search**. There are more sophisticated variants of grid search called **pattern search** methods [155, p145-157] [5, §9.3] [4, §12.5.2], but a *less* sophisticated variant is actually of more interest to us now.

Instead of evaluating the objective at every point on a grid, or at a succession of points each chosen based on previous function values, we could simply choose points $\mathbf{x}^k$ *at random,* compute each $f(\mathbf{x}^k)$, and declare the $\mathbf{x}^k$ yielding the lowest objective value to be optimal. This simplest of all optimization algorithms is called **pure random search**. "Let's just have a go at it" is the favorite heuristic of all those people who seem eager to tell you they were never good at math, so pure random search is used on a grand scale in business, government, and everyday life. As implemented in the MATLAB program `prs.m` listed below, it is used on a more modest scale to solve the `rb` problem.

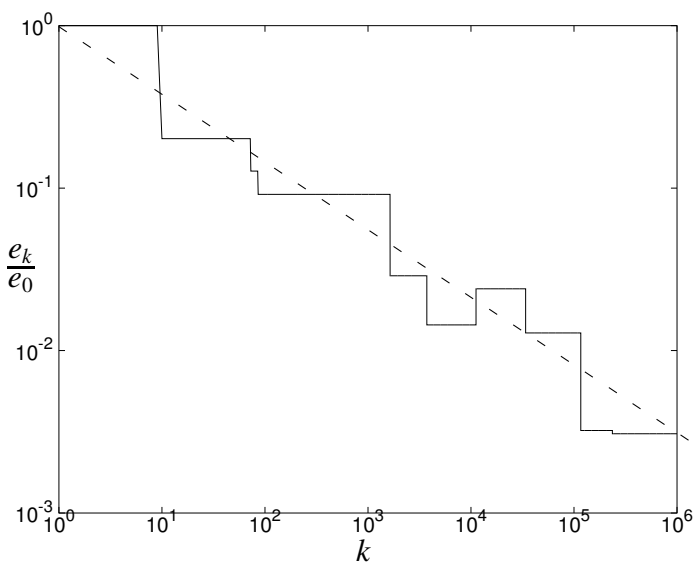
```
 1 % prs.m: solve the rb problem by pure random search
 2 clear; clf; set(gca,'FontSize',30)
 3 format long
 4
 5 xl=[-2;-1];                              % lower left corner of box
 6 xh=[ 2; 2];                              % upper right corner of box
 7
 8 xzero=[-1.2;1];                          % starting point
 9 xstar=[1;1];                             % optimal point
10 ezero=norm(xzero-xstar);                 % error at starting point
11
12 fr=+realmax;                             % record value = +infinity
13 xk=xzero;                                % current iterate = starting point
14 for k=1:1000000                          % try a million points
15     fk=rb(xk);                           % objective at current point
16     if(fk < fr)                          % better than record value?
17         fr=fk;                           % yes; remember it
18         xr=xk;                           % and where it happened
19     end
20     xerr(k)=norm(xr-xstar)/ezero;        % remember error at record point
21     it(k)=k;                             % remember current iteration
22     u=rand(2,1);                         % random vector uniform on (0,1)
23     for j=1:2                            % in each coordinate direction
24         xk(j)=xl(j)+u(j)*(xh(j)-xl(j));  % find value between bounds
25     end
26 end
27
28 xr                                       % report best point found
29 fr                                       % report the objective there
30 xerrend=xerr(1000000)                    % report final relative error
31 loglog(it,xerr)                          % plot log error versus log k
32 print -deps -solid prs.eps               % print the plot
```

The program begins by [5-6] defining the box in which random points will be examined and [8-10] computing the error $e_0 = \|\mathbf{x}^0 - \mathbf{x}^\star\|$ at the **catalog starting point** given for `rb` in §28.7.2. The variable `fr`, representing the lowest objective value found so far or **record value**, is initialized [12] to $+\infty$. Then the `for` loop [14-26] examines 1000000 points randomly positioned within the box.

Whenever a point is discovered 16 to have an objective value less than `fr`, the record value is updated 17 and the point is 18 declared the **record point** `xr`. The **relative error** of the current record point is then calculated 20 as `xerr(k)` $= e_k/e_0 = \|\mathbf{x}^r - \mathbf{x}^\star\|/e_0$, and the iteration number 21 is saved for plotting the error later.

Then the next trial point is generated. The statement `u=rand(2,1)` 22 makes `u` a 2-element column vector each of whose elements is a pseudorandom number uniformly distributed on the interval (0,1). Each of these random numbers u_j is 23-25 mapped onto the interval $[x_j^{\mathrm{L}}, x_j^{\mathrm{H}}]$ to 24 produce `xk(j)`, and 26 the loop over trial points continues.

At the end of the million trials, the record point and value are reported 28-29 along with 30 the final error `xerrend`, and a graph is produced 31-32 of $\log_{10}(e_k/e_0)$ versus $\log_{10}(k)$. That **error curve** and an Octave session showing the program's printed outputs are shown below.




```
octave:1> prs
xr =

   1.00328430564206
   1.00591651081019

fr =  5.47285992540590e-05
xerrend =  0.00307589144901223
octave:2> quit
```

At $k = 0$ we have $e_k = e_0$, so the relative error $e_k/e_0 = 1 = 10^0$ and the error curve begins at $(0, 10^0)$. Here the iterations are plotted on a log scale so the first point we see is the one at $k = 10^0 = 1$, but by then no improvement had yet been made in the objective value. Each transition from one error level to the next occurs when an iterate $\mathbf{x}^k$ is generated that has a lower objective value than the current record value f^r. The curve goes up and down because the measure of solution error that we are using is $\|\mathbf{x}^r - \mathbf{x}^\star\|/e_0$ and in this problem it is possible for the error in $\mathbf{x}$ to increase in moving from one record point to the next even though the error in $f(\mathbf{x})$ decreases.

The graph and the other outputs produced by `prs.m` change from one run to another, because the values returned by `rand` do not repeat. However, the results shown above are typical: $\mathbf{x}^r \approx \mathbf{x}^\star = [1, 1]^\top$ and $f^r \approx f(\mathbf{x}^\star) = 0$. The final relative error has $\log_{10}(\texttt{xerrend}) = -2.512$ so $\texttt{xerrend} = 10^{-2.512}$ and that is the final error level in the graph. To run `prs.m` on my computer, which has a clock speed of 1GHz, took about three minutes.

9.2 Rates of Convergence

Algorithms for nonlinear programming are **infinitely convergent**, in contrast to the simplex method which converges in a finite number of steps if it does not cycle. A given nonlinear programming algorithm applied to a given problem might not even get close to the answer, but if it does it will be in the limit as $k \to \infty$. If an algorithm for finding $\mathbf{x}^\star$ starts from $\mathbf{x}^0$ and generates iterates $\mathbf{x}^k$, we define the **error** of the k'th iterate as $e_k = \|\mathbf{x}^k - \mathbf{x}^\star\|$. Then if

$$\lim_{k\to\infty} e_k = 0$$

we say the algorithm **converges** to the solution $\mathbf{x}^\star$.

Pure random search converges to the solution of `rb`, and its error curve shows $\log_{10}(e_k/e_0)$ decreasing in a roughly linear fashion as $\log_{10}(k)$ increases. I modeled this behavior by drawing the dashed line from the point $(1, 10^0)$ to the final point in our experiment, $(10^6, \texttt{xerrend})$. Using the definition of relative error and the equation of this straight line we find

$$\log_{10}(e_k/e_0) = \begin{cases} 0 & \text{for} \quad k = 0 \\ \dfrac{\log_{10}(\texttt{xerrend})}{\log_{10}(10^6 - 1)} \times \log_{10}(k) & \text{for} \quad k \geq 1 \end{cases}$$

where the slope of the line is

$$\log_{10}(c) = \frac{\log_{10}(\texttt{xerrend})}{\log_{10}(10^6 - 1)} \approx \frac{-2.512}{6} = -0.419 \quad \text{so that} \quad c \approx 10^{-0.419} \approx 0.381$$

Then we have

$$\log_{10}(e_k/e_0) = \begin{cases} 0 & \text{for} \quad k = 0 \\ \log_{10}(k)\log_{10}(c) & \text{for} \quad k \geq 1 \end{cases}$$

or, for $k \geq 1$,

$$\frac{e_k}{e_0} = c^{\log_{10}(k)} \quad \text{so that} \quad e_k = e_0 c^{\log_{10}(k)}.$$

This is called **sublinear convergence**.

The convergence of the other algorithms we will study is described (when they converge at all) by a different model [4, p58-61]. If the errors of successive iterates satisfy

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k^r} = c \quad \text{where} \quad 0 \leq c < \infty.$$

the algorithm is said to have **rate** or **order** of convergence r with **convergence constant** c. For example, if $x_0 = -10$ this recurrence on $\mathbf{x} \in \mathbb{R}^1$

$$x^{k+1} = \frac{x^k}{2} + \frac{2}{x^k}$$

generates iterates $-10, -5.2, -2.9846, \ldots$ that converge to $x^\star = -2$. We can deduce the order of convergence and the convergence constant of this sequence as follows.

$$\begin{aligned}
e_k &= \|x^k - x^\star\| = |x^k + 2| \\
e_{k+1} &= \|x^{k+1} - x^\star\| = \left|\frac{x^k}{2} + \frac{2}{x^k} + 2\right| \\
&= \left|\frac{1}{2x^k}([x^k]^2 + 4 + 4x^k)\right| \\
&= \frac{1}{|2x^k|}|x^k + 2|^2 \\
&= \frac{1}{|2x^k|}e_k^2
\end{aligned}$$

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k^2} = \frac{1}{|2x^\star|} = \frac{1}{4} \quad \text{so this algorithm converges with} \quad r = 2 \quad \text{and} \quad c = \tfrac{1}{4}.$$

Most optimization algorithms are more complicated than this simple recurrence, so it is seldom possible to find the order and constant of convergence analytically as in this example. However, we can derive a general formula for e_k as a function of k by assuming (somewhat unrealistically) that the iterates $\mathbf{x}^k$ obey exactly the recurrence

$$e_{k+1} = ce_k^r$$

for all k rather than just as $k \to \infty$. Starting from $e_0 = \|\mathbf{x}^0 - \mathbf{x}^\star\|$ we find

$$\begin{aligned}
e_1 &= ce_0^r \\
e_2 &= ce_1^r = c(ce_0^r)^r = c(c^r e_0^{r^2}) = c^{1+r}e_0^{r^2} \\
e_3 &= ce_2^r = c(c^{1+r}e_0^{r^2})^r = c^{1+r+r^2}e_0^{r^3} \\
e_4 &= ce_3^r = c(c^{1+r+r^2}e_0^{r^3})^r = c^{1+r+r^2+r^3}e_0^{r^4} \\
&\vdots \\
e_k &= c^{\sum_{j=0}^{k-1} r^j} e_0^{r^k}.
\end{aligned}$$

But the sum of a geometric series is

$$\sum_{j=0}^{k-1} r^j = \begin{cases} \dfrac{1-r^k}{1-r} & \text{if } r \neq 1 \\ k & \text{if } r = 1 \end{cases}$$

so

$$e_k = \begin{cases} c^{(1-r^k)/(1-r)} e_0^{r^k} & \text{if } r \neq 1 \\ c^k e_0 & \text{if } r = 1 \end{cases}$$

Because r can be bigger than 1 it is possible for e_{k+1} to be less than e_k even if $c > 1$, but the values that r and c can take on are restricted by the convergence requirement that

$$\lim_{k\to\infty} e_k = 0.$$

We know that $c \geq 0$ because it is the ratio of norms and a norm is never negative. If r were negative the e_k would alternate in sign, which is impossible because e_k is a norm, so it must be that $r \geq 0$. If $e_{k+1} < e_k$ for all k, the algorithm will surely converge, and that will happen if

$$\begin{aligned} ce_k^r &< e_k \\ ce_k^{r-1} &< 1 \\ c &< 1/e_k^{r-1}. \end{aligned}$$

If $r \geq 1$ we can require that $c < 1/e_k^{r-1}$ for the largest e_k, which we just assumed is e_0. If $r < 1$, the inequality requires that $c < e_k^{1-r}$ for the smallest e_k, which is 0, but the algorithm will certainly (and suddenly!) converge if $c = 0$. Thus, our formula for e_k makes sense if

$$\begin{aligned} c < 1/e_0^{r-1} &\quad \text{for} \quad r \geq 1 \\ c = 0 &\quad \text{for} \quad 0 \leq r < 1. \end{aligned}$$

For $r = 1$ the formula predicts $e_k = c^k e_0$, and this is called **linear** or **first-order** convergence. If $e_0 = 1$ and $c = 0.1$ a linearly-convergent algorithm generates a sequence of iterates with relative errors of $1, 0.1, 0.01, 0.001, \ldots$ in which one additional correct digit is obtained for each iteration.

For $r = 2$ the formula predicts $e_k = c^{2^k-1} e_0^{2^k}$, and this is called **quadratic** or **second-order** convergence. If $e_0 = 1$ and $c = 0.1$ a quadratically-convergent algorithm generates a sequence of iterates with relative errors of $1, 0.1, 0.001, 0.0000001, \ldots$ in which the number of correct digits doubles for each iteration after the second.

An algorithm having $r > 1$ is said to have **superlinear convergence**. Quadratic convergence is superlinear, but often the term is used when $1 < r < 2$.

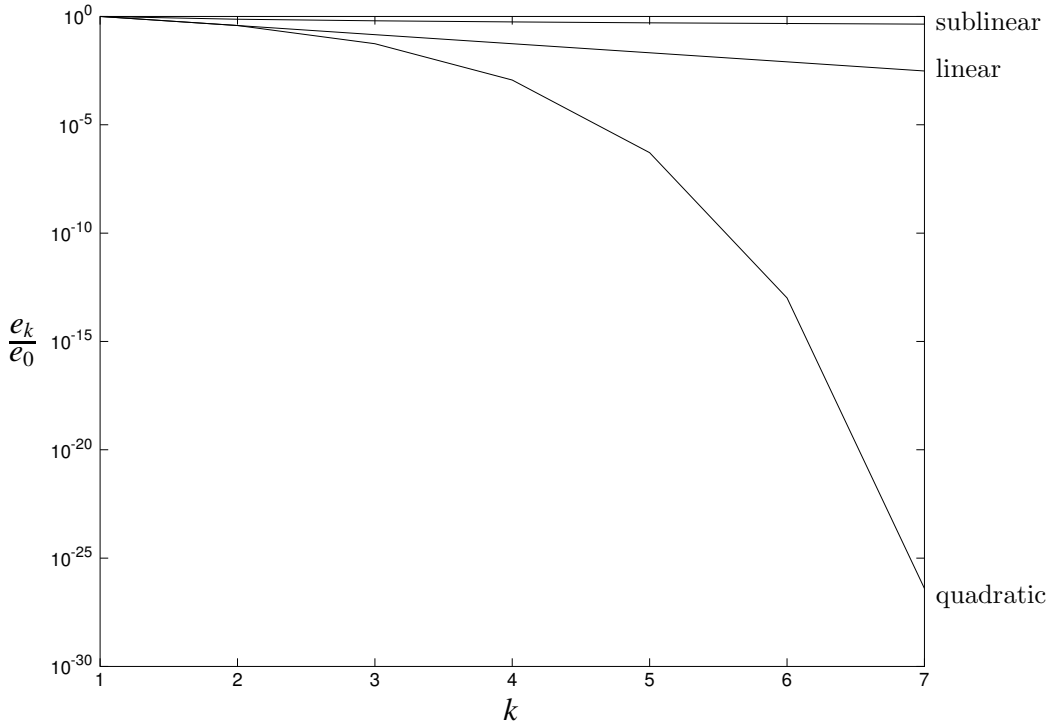
In studying the convergence of an algorithm empirically we usually plot e_k/e_0 versus k, so it is convenient to know when interpreting such a plot that the model predicts for $k \geq 1$

$$e_k/e_0 = \begin{cases} c^k & \text{for } r = 1 \\ (ce_0)^{2^k-1} & \text{for } r = 2 \end{cases}$$

or

$$\log_{10}(e_k/e_0) = \begin{cases} k \log_{10} c & \text{for } r = 1 \\ (2^k - 1)(\log_{10} c + \log_{10} e_0) & \text{for } r = 2 \end{cases}$$

Error curves for algorithms having particular orders of convergence have characteristic shapes, as shown by the graph on the next page. Here the horizontal axis uses a linear rather than a logarithmic scale, and only the first 7 iterations are plotted.



The picture was produced by the MATLAB program listed below, which uses the formula for e_k that we derived for sublinear convergence, the recurrence $e_{k+1} = ce_k$ for linear convergence, and the recurrence $e_{k+1} = ce_k^2$ for quadratic convergence. All three error curves assume the same value of $c = 0.381$ that we measured from the pure random search solution of `rb`. The log error plot has the shape of a quadratic for quadratic convergence, a straight line for linear convergence, and a line that barely descends for sublinear convergence.

```
% cvrg.m: plot a particular set of ideal error curves
set(gca,'FontSize',20)

c=0.381
ezero=1;
quad=ezero; linr=ezero;

for k=1:7
    y(k,1)=quad;
    quad=c*quad^2;
    y(k,2)=linr;
    linr=c*linr;
    y(k,3)=(c^log10(k))*ezero;
    it(k)=k;
end

semilogy(it,y)
print -deps -solid cvrg.eps
```

To facilitate experimentation with different convergence characteristics (other than sublinear) I wrote the MATLAB function listed on the next page. It computes the `kmax`'th iterate

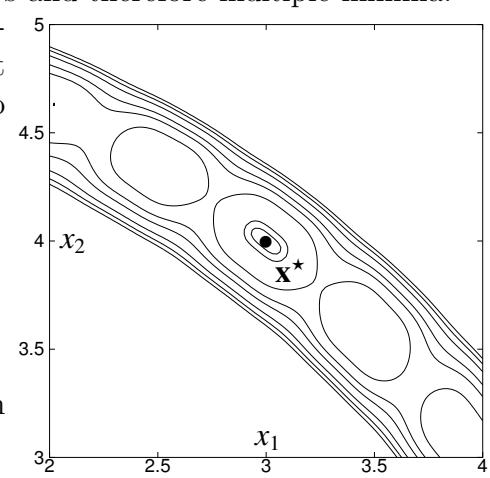
from the formulas we derived and also by performing the iterations (so that the results can be compared) and plots the ideal error curve. Experimenting with it will help you understand how the convergence behavior of an algorithm depends on its rate r and constant c.

```
function converge(r,c,ezero,kmax)
% plot a given arbitrary set of ideal error curves
  if(r == 1)                    % find ending error
     form=ezero*c^kmax          % from the formulas we derived
  else
     form=ezero^(r^kmax)*c^((1-r^kmax)/(1-r))
  end

  ek=ezero                      % current error = starting error
  kk(1)=0;                      % at zero iterations
  err(1)=ek/ezero;              % starting relative error = 1
  for k=2:kmax+1                % plot for k=0...kmax
      ek=c*ek^r                 % recursion for next error
      if(ek == 0) break; end    % if zero no point in more
      err(k)=ek/ezero;          % current relative error
      kk(k)=k-1;                % at current iteration
  end
  semilogy(kk,err)              % plot the iterated error curve
end
```

9.3 Local Minima

The `rb` problem has a single minimizing point at $\mathbf{x}^\star = [1,1]^\top$, but the objective of a nonlinear program can have a graph with multiple hills and valleys and therefore multiple minima.

The `gpr` problem (see §28.7.3), whose contour diagram is shown on the right, has a single optimal point $\mathbf{x}^\star = [3,4]^\top$ at the bottom of its deepest valley, but also many shallower valleys.



$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f(\mathbf{x}) & = & e^{u^2} + \sin^4(v) + \frac{1}{2}w^2 \\ \text{where} & u & = & \frac{1}{2}(x_1^2 + x_2^2 - 25) \\ & v & = & 4x_1 - 3x_2 \\ & w & = & 2x_1 + x_2 - 10 \end{array}$$

To distinguish the various kinds of minima that can occur we will use the following taxonomy [4, p45-46].

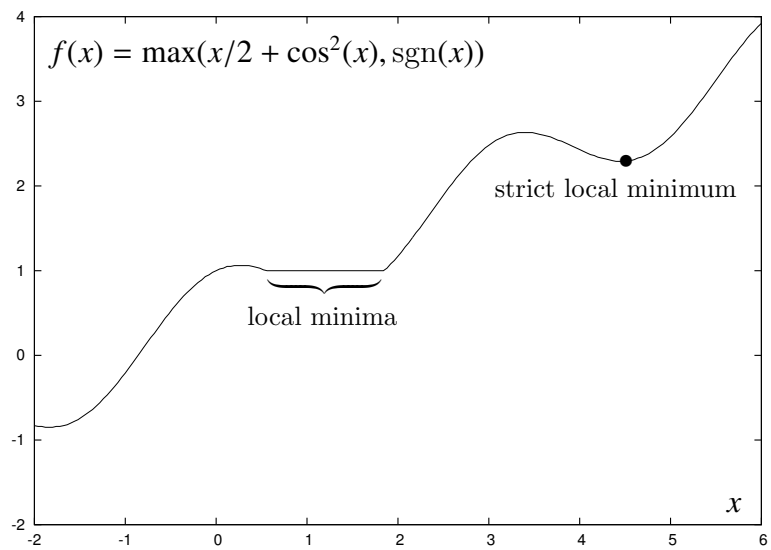
$\bar{\mathbf{x}}$ is a	if and only if	
strict global minimum	$f(\bar{\mathbf{x}}) < f(\mathbf{x})$	for all $\mathbf{x} \neq \bar{\mathbf{x}}$
global minimum	$f(\bar{\mathbf{x}}) \leq f(\mathbf{x})$	for all $\mathbf{x}$
strict local minimum	$f(\bar{\mathbf{x}}) < f(\mathbf{x})$	for all $\mathbf{x} \in \mathcal{N}_\varepsilon(\bar{\mathbf{x}}) \setminus \bar{\mathbf{x}}$
local minimum	$f(\bar{\mathbf{x}}) \leq f(\mathbf{x})$	for all $\mathbf{x} \in \mathcal{N}_\varepsilon(\bar{\mathbf{x}})$

In these definitions $\mathcal{N}_\varepsilon(\bar{\mathbf{x}}) = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x} - \bar{\mathbf{x}}\| < \varepsilon\}$, where $\varepsilon > 0$, denotes an **epsilon-neighborhood** of $\bar{\mathbf{x}}$ [136, p32]. If the norm is the 2-norm, this neighborhood is an open ball centered at $\bar{\mathbf{x}}$. Thus if $\bar{\mathbf{x}}$ is a local minimum then $f(\bar{\mathbf{x}}) \le f(\mathbf{x})$ for all points within some positive radius ε of $\bar{\mathbf{x}}$. The symbol $\setminus$ is "set minus" so $\mathcal{N}_\varepsilon(\bar{\mathbf{x}}) \setminus \bar{\mathbf{x}}$ means the neighborhood without the point at its center. If $\bar{\mathbf{x}}$ is a strict local minimum then $f(\bar{\mathbf{x}})$ is strictly less than $f(\mathbf{x})$ at every other point within some positive radius ε of $\bar{\mathbf{x}}$.

In the case of a strict global minimum, $\bar{\mathbf{x}} = \mathbf{x}^\star$ is the *unique* point at which $f(\mathbf{x})$ takes on its lowest value. The point $[3, 4]^\top$ is the strict global minimizing point of the `gpr` problem pictured above, and the point $[1, 1]^\top$ is the strict global minimizing point of the `rb` problem.

In the case of a global minimum that is not strict, $\bar{\mathbf{x}} = \mathbf{x}^\star$ is one point, but maybe not the only point, at which $f(\mathbf{x})$ takes on its lowest value. If $\mathbf{x} \in \mathbb{R}^2$ the function $f(\mathbf{x}) = x_1^2$ has its lowest value of zero at every point on the x_2 axis, so they are all nonstrict global minima.

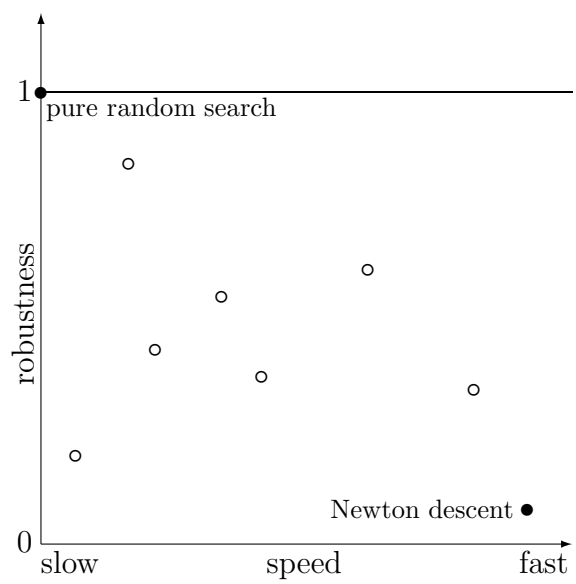
The distinction between strict and non-strict local minima is illustrated in the graph below.



9.4 Robustness versus Speed

The convergence behavior of real algorithms is seldom predicted exactly by the theory we developed in §9.2, because our analytical model is just an approximation and we never actually let k reach ∞. The error curve we measured for pure random search doesn't look much like the theoretical one, and the experimental error curves that we draw for other algorithms will often depart somewhat from the ideal. Actual performance must be measured empirically. But the predictions of the model are at least qualitatively correct, and from them we can conclude that linear convergence is good but quadratic convergence is dramatically better. Sublinear convergence, especially for problems having $n > 2$, is practically useless; unfortunately, even the best algorithms for some large problems can do no better [160, §4].

Algorithms that achieve first-order convergence typically make use of first derivatives in addition to function values, while those that achieve second-order convergence typically require second derivatives as well. For this and other reasons fancy algorithms usually use more CPU time per iteration than simple ones, but they need fewer iterations so they run faster overall. Unfortunately, they also more often fail to converge, or get trapped at a local minimum that is **suboptimal** (i.e., not as good as the *global* minimum). Yogi Berra could have been thinking of this behavior when he famously remarked "We're lost, but we're making good time." Pure random search is very **robust** in that it finds a global minimizing point almost no matter what the problem is like. It plods along using only function values, too stupid not to work. Newton descent, which we will take up in §13, is by comparison elegant and clever, and when it works it has breathtaking second-order convergence, but it fails catastrophically on many problems. Of course this need not concern us if Newton descent happens to work well on the one problem we want to solve. Special-purpose algorithms have also been contrived to solve certain limited classes of problem very fast. But if our aim is to design a general-purpose method, the goals of robustness and speed are always in competition [2, §2.7]. The tradeoff between them is depicted graphically below, where each point represents a different algorithm.



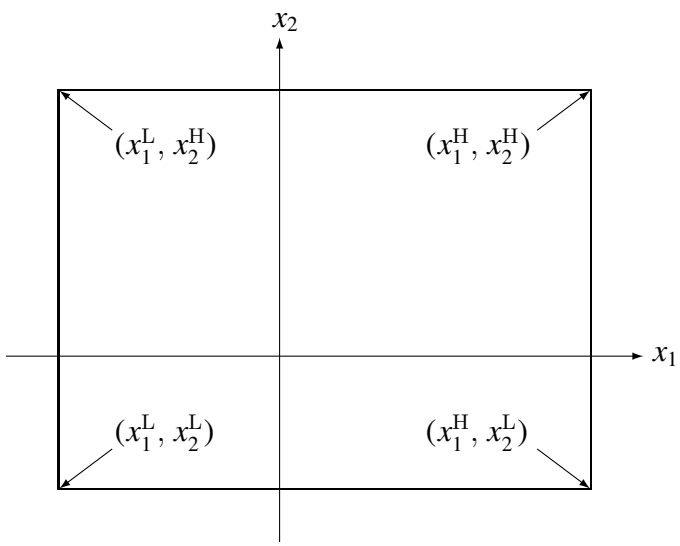
In this picture robustness can be thought of as the likelihood of solving a problem chosen at random from some universe of all possible nonlinear programs, while speed measures the computational effort required to achieve some suitable level of accuracy in the reported $\mathbf{x}^\star$. Both of these notions will be made more precise and quantitative in §26.

Algorithms that fall in the lower left corner of this graph deserve only the scorn and derision they receive. One that fell in the upper-right corner, a single method that could be used to resolve any nonlinear program just as the simplex algorithm is used to resolve any

linear program, has been a prize avidly sought since the foundations of numerical optimization were laid. The story you will learn in future Chapters is therefore largely the tale of heroic efforts to find some Northeast Passage into that (still vacant) corner of this graph.

9.5 Variable Bounds

To draw the contours of the `rb` objective and to solve the problem by pure random search, we evaluated the function at points within a box defined by bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ on the variables. Here is a picture of the box, showing the coordinates of its corners.



We will use variable bounds in §12.2.2 to limit the range of a line search, in §24.3.1 to construct a starting ellipsoid for the ellipsoid algorithm, and in several places to determine a starting point $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}})$ from which to begin the solution of a problem. Variable bounds can also be used to limit the radius of the trust region in the trust-region algorithm of §17.3, to keep slack variables nonnegative in the §20.2.5 augmented Lagrangian algorithm extension for inequality constraints, and to avoid regions of $\mathbb{R}^n$ where an objective or constraint function is undefined. But the best reason for fixing bounds on the variables of a nonlinear program before attempting a solution, whether analytic or numeric, is to ensure that you really understand the formulation; having no idea where to look for the optimal point suggests that the problem requires further preliminary study [1, p29]. Each of the example nonlinear programs cataloged in §28.7 includes as part of the statement of the problem a specification of the variable bounds that are to be respected in its solution.

The variable bounds that we use in solving a problem express our deductions about the region of $\mathbb{R}^n$ where the optimal point must be found, or our expectations about where it is likely to be found, rather than conditions that must be enforced. Therefore, while bounds on the variables can be among the constraints usually they are *not* formal constraints.

Bounds certain to contain $\mathbf{x}^\star$ can often be established when an optimization problem is formulated, based on laws of nature or on standard practice in the field of application. In the `garden` problem of §8.1 the width of the garage and the length of the fence determined variable bounds that we used in scaling our graph of the feasible region. Even synthetic problems with no practical application often include inequality constraints from which bounds on the variables can be deduced. In the rare case when it is necessary to *guess* bounds in the initial investigation of a problem, convergence to a suboptimal point that is at or outside the bounds is evidence that those bounds were chosen too narrow. On the other hand, many problems can be solved from bounds that are generous, so if you really must guess it might not hurt to guess wide.

9.6 The Prototypical Algorithm

> **pro·to·typ·i·cal** *adj.* Representing an original model or type after which other similar things are patterned.

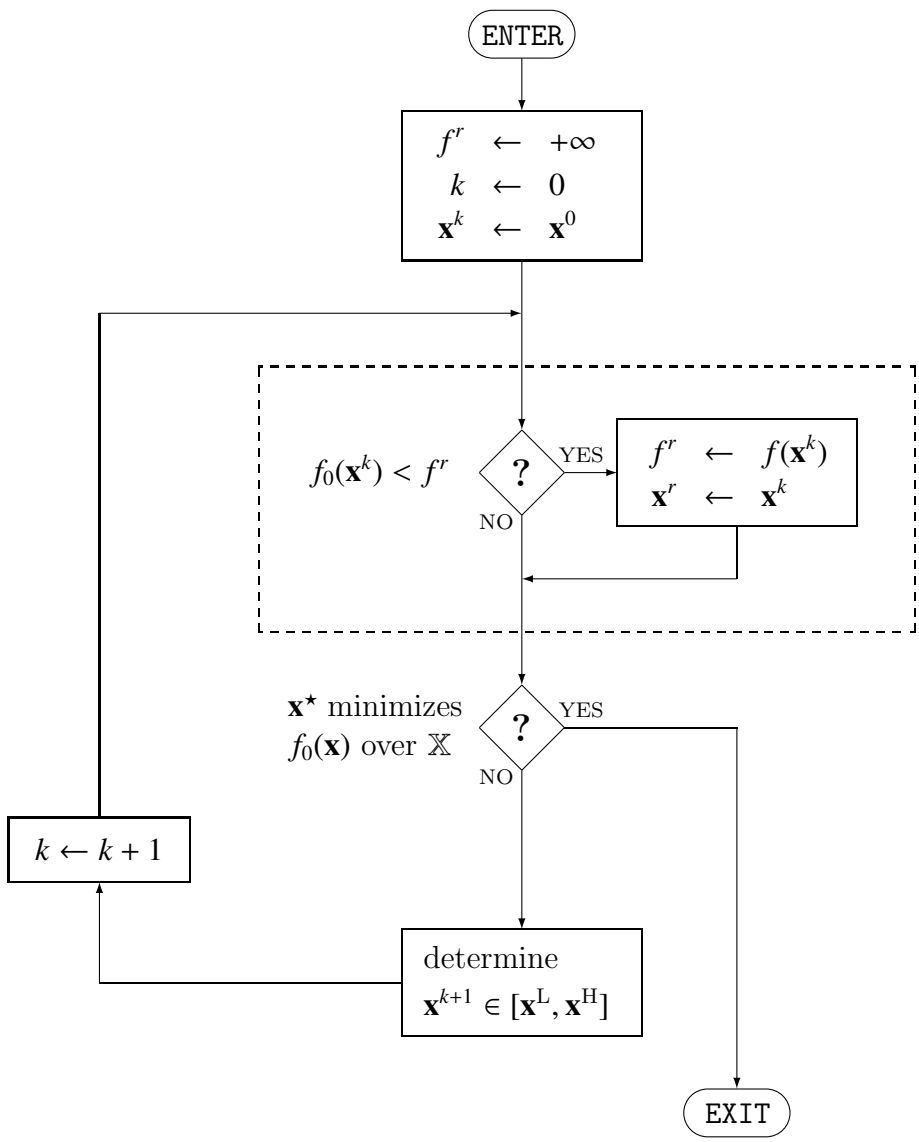
All of the nonlinear programming algorithms we will study can be represented by the flowchart on the next page, and I will occasionally refer to it in explaining how they work.

We begin by initializing the record value f^r, the iteration counter k, and the current estimate of the optimal point $\mathbf{x}^k$. Then the record value and the record point $\mathbf{x}^r$ are updated. A dashed box is drawn around these steps because, except when they are essential (as in the case of pure random search and the ellipsoid algorithm of §24) I will routinely omit them to simplify the explanation of the algorithms we will study. When you are *learning* how an algorithm works it is instructive to watch the $\mathbf{x}^k$ that are generated, rather than concealing any missteps that might occur behind a record point. When you are *using* a nonlinear programming algorithm to solve a practical problem, however, it is always prudent to keep a record value and record point as shown in the flowchart, and to accept the record point, rather than the final iterate, as the optimal vector.

Next comes the convergence test, which mentions the feasible set $\mathbb{X}$. The `rb` and `gpr` examples we used in this Chapter have no constraints, and for the next five Chapters we will consider only unconstrained problems. Of course most nonlinear programs (like the `garden` problem of §8.1) *do* have constraints, and if this flowchart is going to describe the methods that solve them the convergence test must not return for $\mathbf{x}^\star$ a point that is infeasible. As we shall see in §10, the convergence criterion for a nonlinear programming algorithm is usually based on whether a minimizing point has been (at least approximately) found, rather than on an arbitrary iteration limit like the one that `prs.m` uses.

How $\mathbf{x}^{k+1}$ is determined is what characterizes each of the algorithms we will study, so another way to view the rest of this book is that it is about what goes inside that box of

the flowchart. For pure random search it is "pick $\mathbf{x}^{k+1}$ at random" but for more effective algorithms the prescription can be much more complicated.



In the next Chapter we begin our study of more effective algorithms with the method of steepest descent, which uses first derivatives of the objective in determining $\mathbf{x}^{k+1}$ and thereby achieves linear convergence. It is only a little more complicated than pure random search, but to understand how it works you might find it helpful to return to this flowchart.

9.7 Exercises

9.7.1 [E] An *algorithm* is a mechanical procedure that can be performed by rote. Describe, as precisely as you can, an algorithm (not necessarily involving mathematics) that you carry out routinely in the course of your everyday life.

9.7.2 [E] A numerical method is an iterative algorithm that approximates the solution to a mathematical problem by performing only arithmetic and logical operations. (a) What is meant by an *iterative* algorithm? (b) Give examples of some arithmetic and logical operations. (c) Describe, as precisely as you can, a numerical method for solving some mathematical problem *other than* optimization.

9.7.3 [E] Prove that $\mathbf{x}^\star = [1, 1]^\top$ is optimal for the `rb` problem of §28.7.2.

9.7.4 [E] State the purpose of the MATLAB `gridcntr` function described in §9.1, and explain how it works. In `gridcntr.m`, the name of the routine that calculates a value of the function being contoured is `fcn`. In the example, how did we get `gridcntr` to use `rb` as that routine?

9.7.5 [H] Label each contour of the `rb` objective with the value the function has at every point on the contour.

9.7.6 [H] Suppose a grid search with `ng=100` points is used to approximate the minimizing point of a function of $\mathbf{x} \in \mathbb{R}^1$ on the interval $[x^{\mathrm{L}}, x^{\mathrm{H}}] = [0, 1]$. (a) How much error might there be in the estimate of $x^\star$? (b) Now suppose that $\mathbf{x} \in \mathbb{R}^n$ where $n > 1$, and that $\mathbf{x}^{\mathrm{L}} = \mathbf{0}$ (the origin) and $\mathbf{x}^{\mathrm{H}} = \mathbf{1}$ (a vector of all 1's). How many function evaluations must be used, as a function of n, to achieve the same level of error in $\mathbf{x}^\star$?

9.7.7 [P] A refinement of grid search shrinks the variable bounds after each sweep through the grid, by bisecting the distance from x_j^{L} to x_j^{H} in each coordinate direction $j = 1 \ldots n$ to throw away the half that does not appear to contain the minimizing point. Write a MATLAB program to implement this idea, and use it to solve the `rb` problem.

9.7.8 [P] Pure random search can easily be generalized to solve problems having constraints. Modify `prs.m` to enforce constraints, and use your program to solve the `garden` problem of §8.1.

9.7.9 [E] If a starting point $\mathbf{x}^0$ is identified as the *catalog* starting point, what does that mean?

9.7.10 [E] What is a *record value*? A *record point*? Why might it be helpful to update these in the course of solving a nonlinear program?

9.7.11 [E] In monitoring the convergence of a numerical method, why do we typically plot the relative error $\log_{10}(e_k/e_0)$, so that the error curve begins at 0, rather than the absolute error $\log_{10}(e_k)$?

9.7.12 [H] How is it possible for $\mathbf{x}^{k+1}$ to be farther from $\mathbf{x}^\star$ than $\mathbf{x}^k$ is, even though $f(\mathbf{x}^{k+1})$ is closer to $f(\mathbf{x}^\star)$ than $f(\mathbf{x}^k)$ is? Give an example in which this happens.

9.7.13 [E] How does an infinitely-convergent algorithm differ from one having finite convergence? If a degenerate linear program cycles, does that make the simplex algorithm infinitely convergent? What technical definition of convergence is adopted in this text? Does the pure random search algorithm converge in that sense?

9.7.14 [E] What must be true in order for an algorithm to have order of convergence r with convergence constant c? For the algorithm to converge, is it necessary that $c < 1$? Explain.

9.7.15 [E] If the solution error at $\mathbf{x}^0$ is e_0, what does the sublinear convergence model of §9.2 predict the solution error will be at $\mathbf{x}^1$? What is it predicted to be if the convergence is linear with $c = \frac{1}{2}$?

9.7.16 [P] The recurrence used as an example in §9.2 converges to $\mathbf{x}^\star = -2$ if $x^0 = -10$. To what point does it converge if $x^0 = +10$? Write a MATLAB program to illustrate your answer, and plot output from the program to illustrate the convergence of this algorithm. Is the convergence still second-order? Is the convergence constant still $\frac{1}{4}$?

9.7.17 [E] The model of algorithm convergence that we developed in §9.2 predicts what the solution error e_k will be after k iterations, given the rate of convergence r and the convergence constant c. What value of r corresponds to *quadratic* convergence? What values of c are possible for a convergent algorithm that has $r = 1$?

9.7.18 [H] The convergence model of §9.2 predicts the appearance of error curves that plot $\log_{10}(e_k/e_0)$ versus k. (a) What are the slope and intercept of the straight line predicted by the model for $r = 1$? (b) How does the convergence constant c affect the appearance of the curve when $r = 2$?

9.7.19 [E] Use `converge.m` to investigate what happens if $r = \frac{1}{2}$ and $x^0 = 1$. (a) Is convergence achieved with $c = 0.1$? (b) Is convergence achieved with $c = 0$?

9.7.20 [E] Many algorithms have superlinear convergence with $1 < r < 2$. Use `converge.m` to plot an error curve for 10 iterations if $r = 1.1$, $c = 0.1$, and $e_0 = 1$.

9.7.21 [P] Write a program that reproduces the contour diagram of the `gpr` objective shown in §9.3 by using the `gridcntr` function of §9.1 to compute grid points and the MATLAB `contour` function to draw the contours.

9.7.22 [E] What do we mean by $\mathcal{N}_\varepsilon(\bar{\mathbf{x}})$? How big is ε?

9.7.23 [H] Prove that a strict global minimum is also a global minimum, a strict local minimum, and a local minimum.

9.7.24 [H] In §9.3 the case of a non-strict global minimum is illustrated by the example of $f(\mathbf{x}) = x_1^2$, where $\mathbf{x} \in \mathbb{R}^2$. Sketch a graph of $f(\mathbf{x})$ showing which points are its global minima.

9.7.25 [E] Write down the formula for a function that has more than one global minimizing point.

9.7.26 [E] Why is the convergence behavior of real algorithms seldom predicted exactly by the theory we developed in §9.2? Is a quadratically-convergent algorithm always to be preferred over one that has only linear convergence? Explain.

9.7.27 [E] If a function has several local minima of different depths it is possible for a nonlinear programming algorithm to get stuck at a suboptimal one. Does "*sub*optimal" mean that the objective value there is *less* than at the global optimum? Explain.

9.7.28 [E] Explain what is meant by the *robustness* of an algorithm, and how it typically relates to the method's speed.

9.7.29 [E] In world history, what was the Northwest Passage? Why does §9.4 refer to a Northeast Passage?

9.7.30 [E] In this book, bounds on the variables are part of the specification of every nonlinear program. Why is that? Can bounds on the variables also be constraints? Must they be constraints?

9.7.31 [E] If you hope to eat lunch in a kosher deli, where might you focus your search for one? (1) On a ranch in Wyoming; (2) at the bottom of the Marianas trench; (3) on Manhattan Island in New York City; (4) on planet Earth; (5) it would be necessary to search the entire universe. What does this question have to do with stating bounds on the variables in a nonlinear programming problem?

9.7.32 [E] Use the prescription $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}})$ to find an alternative (i.e., non-catalog) starting point for the `rb` problem. How does using this $\mathbf{x}^0$ affect the error curve drawn by `prs.m`?

9.7.33 [E] In a certain nonlinear program involving the design of a whisky distillery, x_3 represents the inside diameter of a glass tube. What does this fact suggest about the values that x_3 could plausibly take on?

9.7.34 [E] From the statement of the `garden` problem in §8.1, deduce bounds on the variables. Do you need to know $\mathbf{x}^\star$ in order to do this? Do you need to know what the objective function is?

9.7.35 [H] Consider the problem

$$\text{minimize} \quad f(x) = \frac{1}{\sqrt{x-1}} + 3\sqrt{x-1}.$$

(a) Graph $f(x)$ on the interval from $x = 0$ to $x = 2$. (b) Show analytically that $x^\star = \frac{4}{3}$. (c) What lower bound could you impose on x to prevent a numerical method from trying to evaluate $f(x)$ where it is not defined? When would it be necessary to enforce this bound as an explicit constraint?

9.7.36 [E] The prototypical algorithm of §9.6 calls for keeping a record point and value, but I will often omit those steps from the algorithms we study. Why? Is it a good idea to omit them from an algorithm implementation that you expect to use for solving real problems? What must be true about an algorithm for it to be *unnecessary* to keep a record value and point?

9.7.37 [E] The flowchart given in §9.6 is for a generic nonlinear programming algorithm, but the details of what happens in one block of the flowchart will vary with the specific algorithm being represented. (a) Which block is that? (b) What detailed description does that block contain for pure random search? (c) What does it mean that for the algorithm to converge "$\mathbf{x}^\star$ minimizes $f_0(\mathbf{x})$ *over* $\mathbb{X}$"?

10

Steepest Descent

In §9 we found that although pure random search is very robust, its sublinear convergence makes it too slow to be practical even for problems having only $n = 2$ variables. To get linear or quadratic convergence a minimization algorithm must actually try to go downhill. A function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^1$ descends from a point $\bar{\mathbf{x}}$ most rapidly in the direction of its negative gradient there. In this Chapter we will derive that result and use it to construct a minimization algorithm that is far more useful than pure random search.

10.1 The Taylor Series in $\mathbb{R}^n$

If the function $f(\mathbf{x})$ is sufficiently differentiable, information about its slope and curvature at a point $\bar{\mathbf{x}}$ are captured in its **Taylor series expansion** [1, §3.3.5] about that point.

$$\text{for } n = 1, \qquad f(x) \approx f(\bar{x}) + f'(\bar{x})(x - \bar{x}) + \tfrac{1}{2} f''(\bar{x})(x - \bar{x})^2$$

$$\text{for } n > 1, \qquad \boxed{f(\mathbf{x}) \approx f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top (\mathbf{x} - \bar{\mathbf{x}}) + \tfrac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^\top \mathbf{H}(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}})}$$

The formula for $n = 1$, in which f' denotes the first derivative and f'' the second derivative, might be familiar from a calculus course (if not see §28.1.2). For a function of $n > 1$ variables the analog of $f'(x)$ is the **gradient vector** $\nabla f(\mathbf{x})$ and the analog of $f''(x)$ is the **Hessian matrix** $\mathbf{H}(\mathbf{x})$. The gradient vector and Hessian matrix are made up of partial derivatives of the function, like this.

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{bmatrix} \qquad \mathbf{H}(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}$$

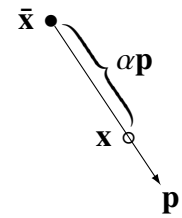
The Hessian matrix is square, and if the mixed partials are continuous then [110, §6.2]

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$$

so $\mathbf{H}$ is symmetric. We will be concerned with other properties of the Hessian matrix in §11, and we will make use of the Taylor series expansion for $n > 1$ on many occasions throughout the rest of the book.

10.2 The Steepest Descent Direction

If we take a step α away from $\bar{\mathbf{x}}$ in the direction $\mathbf{p}$, to $\mathbf{x} = \bar{\mathbf{x}} + \alpha\mathbf{p}$ as pictured below, then the Taylor series expansion of $f(\mathbf{x})$ yields



$$f(\alpha) \equiv f(\bar{\mathbf{x}} + \alpha\mathbf{p}) \approx f(\bar{\mathbf{x}}) + \alpha\mathbf{p}^\top\nabla f(\bar{\mathbf{x}}) + \tfrac{1}{2}\alpha^2\mathbf{p}^\top\mathbf{H}(\bar{\mathbf{x}})\mathbf{p}.$$

Taking the derivative of this approximation with respect to α, we find

$$\frac{df}{d\alpha} = 0 + \mathbf{p}^\top\nabla f(\bar{\mathbf{x}}) + \alpha\mathbf{p}^\top\mathbf{H}(\bar{\mathbf{x}})\mathbf{p} + \text{terms of higher order in } \alpha.$$

At $\bar{\mathbf{x}}$ we have $\alpha = 0$, so at that point $df/d\alpha = \mathbf{p}^\top\nabla f(\bar{\mathbf{x}})$ is the rate of increase of the function. The direction $\mathbf{p}$ of unit norm resulting in the most rapid *decrease* in $f(\mathbf{x})$ makes $df/d\alpha$ at $\alpha = 0$ as *negative* as possible and must therefore be the vector that solves this optimization problem [5, §2.2].

$$\underset{\mathbf{p}}{\text{minimize}} \;\; \mathbf{p}^\top\nabla f(\bar{\mathbf{x}}) \quad \text{subject to} \;\; \|\mathbf{p}\| = 1$$

We can write the dot product $\mathbf{p}^\top\nabla f(\bar{\mathbf{x}}) = \|\mathbf{p}\| \times \|\nabla f(\bar{\mathbf{x}})\| \times \cos(\theta)$ where θ is the angle between the vectors measured on the hyperplane that contains them both (see §28.2.3). A norm is never negative, so this quantity is minimized when $\cos(\theta) = -1$; then the vectors are collinear and point in opposite directions. We required $\|\mathbf{p}\| = 1$, so the direction of steepest descent is the unit vector $\mathbf{p}$ that solves $\mathbf{p}^\top\nabla f(\bar{\mathbf{x}}) = 1 \times \|\nabla f(\bar{\mathbf{x}})\| \times (-1)$. If some direction is downhill from $\bar{\mathbf{x}}$, then $\|\nabla f(\bar{\mathbf{x}})\| \neq 0$ and we can divide to obtain

$$\mathbf{p}^\top\left(\frac{-\nabla f(\bar{\mathbf{x}})}{\|\nabla f(\bar{\mathbf{x}})\|}\right) = 1.$$

Because $\mathbf{p}^\top\mathbf{p} = \|\mathbf{p}\|^2 = 1$, the equation above is satisfied by

$$\mathbf{p} = \frac{-\nabla f(\bar{\mathbf{x}})}{\|\nabla f(\bar{\mathbf{x}})\|}.$$

Thus $f(\mathbf{x})$ descends most steeply from a point $\bar{\mathbf{x}}$ in the direction opposite to its gradient vector at that point.

10.3 The Optimal Step Length

We have shown that if $\bar{\mathbf{x}}$ is not already a minimizing point then $f(\mathbf{x})$ can be reduced by moving in the direction $-\nabla f(\bar{\mathbf{x}})$. To see how this idea can be used consider the nonlinear program at the top of the next page, which is the `gns` problem (see §28.7.4).

$$\text{minimize} \quad f(\mathbf{x}) = 4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1 \quad \text{from} \quad \mathbf{x}^0 = [2,2]^\top$$

Using the definition of the gradient from §10.1 we find

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 8x_1 + 4x_2 - 3 \\ 4x_2 + 4x_1 \end{bmatrix} \qquad \text{so} \qquad \nabla f(\mathbf{x}^0) = \begin{bmatrix} 21 \\ 16 \end{bmatrix}.$$

Thus from the point $\mathbf{x}^0 = [2,2]^\top$ the direction of steepest descent is

$$\mathbf{d}^0 = -\nabla f(\mathbf{x}^0) = \begin{bmatrix} -21 \\ -16 \end{bmatrix}.$$

Moving a distance α in that direction takes us to the point

$$\mathbf{x}^0 + \alpha\mathbf{d}^0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \alpha \begin{bmatrix} -21 \\ -16 \end{bmatrix} = \begin{bmatrix} 2 - 21\alpha \\ 2 - 16\alpha \end{bmatrix} = \mathbf{x}^1,$$

and we want to choose α so that

$$\begin{aligned} f(\alpha) \equiv f(\mathbf{x}^1) &= 4(2-21\alpha)^2 + 2(2-16\alpha)^2 + 4(2-21\alpha)(2-16\alpha) - 3(2-21\alpha) \\ &= 3620\alpha^2 - 697\alpha + 34 \end{aligned}$$

is minimized. Setting the derivative equal to zero and solving for α we find

$$\frac{df}{d\alpha} = 7240\alpha - 697 = 0 \qquad \text{so} \qquad \alpha^\star = 697/7240 \approx 0.096271 = \alpha_0$$

and this is a minimizing point of $f(\alpha)$ because

$$\frac{d^2 f}{d\alpha^2} = 7240 > 0.$$

Moving from $\mathbf{x}^0$ a distance α_0 in the steepest-descent direction $\mathbf{d}^0$ takes us to the point

$$\mathbf{x}^1 = \mathbf{x}^0 + \alpha_0\mathbf{d}^0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \frac{697}{7240} \begin{bmatrix} -21 \\ -16 \end{bmatrix} \approx \begin{bmatrix} -0.021685 \\ 0.459669 \end{bmatrix}$$

where the objective function is $f(\mathbf{x}^1) \approx 0.449655$, a big reduction from $f(\mathbf{x}^0) = 34$. Unfortunately $\mathbf{x}^1$ is not the optimal point, because

$$\nabla f(\mathbf{x}^1) \approx \begin{bmatrix} -1.33480 \\ 1.75194 \end{bmatrix} \neq \mathbf{0}.$$

However, we can use $\nabla f(\mathbf{x}^1)$ to continue the process of moving downhill.

10.4 The Steepest Descent Algorithm

The calculations we did in §10.3 constitute one step of the **steepest-descent algorithm**, first described by Cauchy [168] and formalized in the pseudocode below.

	$k = 0$	start from $\mathbf{x}^0$
1	$\mathbf{g}^k = \nabla f(\mathbf{x}^k)$	find the uphill direction
	`if(`$\|\mathbf{g}^k\| < \epsilon$`) STOP`	if flat there is no uphill direction
	$\mathbf{d}^k = -\mathbf{g}^k$	go downhill
	$\alpha^\star = \underset{\alpha}{\operatorname{argmin}}\, f(\mathbf{x}^k + \alpha\mathbf{d}^k)$	as far as you can
	$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^\star\mathbf{d}^k$	move to that point
	$k = k + 1$	count the iteration
	`GO TO 1`	and repeat

The general optimization algorithm of §9.6 includes flowchart boxes for keeping a record value and record point, in case (as in pure random search) the function values $f(\mathbf{x}^k)$ do not always decrease. In the steepest-descent algorithm it is reasonable to expect that $f(\mathbf{x}^{k+1})$ will never be greater than $f(\mathbf{x}^k)$, so for simplicity I have not provided in this description for keeping a record value or a record point. However, a skeptic could reasonably argue that roundoff errors or the nonzero value of ϵ might result in the objective *not* decreasing at every step. Except for using a tiny amount of processing time and memory, keeping a record value and record point to guard against that would not hurt (see Exercise 10.9.6).

The **argmin operator** used in this pseudocode returns the value $\alpha^\star$ at which the minimum is found, in contrast to the min operator, which would return the value of the function there.

$$\begin{aligned} \min_\alpha f(\alpha) &= \text{value of } f \text{ where } f(\alpha) \text{ is minimized} &= f(\alpha^\star) \\ \underset{\alpha}{\operatorname{argmin}}\, f(\alpha) &= \text{value of } \alpha \text{ where } f(\alpha) \text{ is minimized} &= \alpha^\star \end{aligned}$$

We will use both the min operator and the argmin operator in describing optimization algorithms.

The hard part of the steepest-descent algorithm, whether we execute it by hand or by running a computer program, is finding $\alpha^\star$ for each new point $\mathbf{x}^k$ and direction $\mathbf{d}^k$. The task of finding $\alpha^\star$ for an arbitrary problem will occupy our whole attention in §12, but for this particular problem we can find $\alpha^\star(\mathbf{x};\mathbf{d})$ in general, analytically, as follows.

$$\begin{aligned} f(\mathbf{x} + \alpha\mathbf{d}) &= 4(x_1 + \alpha d_1)^2 + 2(x_2 + \alpha d_2)^2 + 4(x_1 + \alpha d_1)(x_2 + \alpha d_2) - 3(x_1 + \alpha d_1) \\ &= \alpha^2(4d_1^2 + 2d_2^2 + 4d_1d_2) \quad + \quad \alpha(8x_1d_1 + 4x_2d_2 + 4x_1d_2 + 4x_2d_1 - 3d_1) \\ &\qquad\qquad\qquad\qquad\qquad\qquad + \quad (4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1) \end{aligned}$$

$$\begin{aligned}\frac{df}{d\alpha} &= 2\alpha(4d_1^2 + 2d_2^2 + 4d_1d_2) + (8x_1d_1 + 4x_2d_2 + 4x_1d_2 + 4x_2d_1 - 3d_1) = 0\\ \alpha^\star &= \frac{-(8x_1d_1 + 4x_2d_2 + 4x_1d_2 + 4x_2d_1 - 3d_1)}{(8d_1^2 + 4d_2^2 + 8d_1d_2)}\end{aligned}$$

Getting from the first expression for $f(\mathbf{x}+\alpha\mathbf{d})$ to the second (on the previous page) is a little complicated, so I checked all of this work using Maple as shown below. Here I differentiated before simplifying rather than after, so all of the work is in the `solve`.

```
> f := 4*x1^2+2*x2^2+4*x1*x2-3*x1;

                    2       2
                4 x1  + 2 x2  + 4 x1 x2 - 3 x1

> ff := subs(x1 = y1+alpha*d1, x2 = y2+alpha*d2, f);

                      2                     2
     4 (y1 + alpha d1)  + 2 (y2 + alpha d2)

        + 4 (y1 + alpha d1)(y2 + alpha d2) - 3 y1 - 3 alpha d1

> fp := diff(ff, alpha);

     8 (y1 + alpha d1) d1 + 4 (y2 + alpha d2) d2

        + 4 (y2 + alpha d2) d1 + 4 (y1 + alpha d1) d2 - 3 d1

> solve(fp = 0, alpha);

           8 d1 y1 + 4 d1 y2 + 4 d2 y1 + 4 d2 y2 - 3 d1
         - ---------------------------------------------
                     /    2                 2\
                   4 \2 d1  + 2 d1 d2 + d2 /
```

Using the first formula for $\alpha^\star$, I wrote the MATLAB program on the next page. It invokes `gns.m` to find $f(\mathbf{x})$ and `gnsg.m` to find $\nabla f(\mathbf{x})$ (`gnsh.m` returns $\mathbf{H}(\mathbf{x})$ and is used later).

The first stanza of the program [1-17] implements the solution process described in the pseudocode above. Twenty iterates are allowed [4] but 12 are enough to satisfy the convergence condition [10] (`epz` is used for ϵ because `eps` is a reserved word in MATLAB). The formula for $\alpha^\star(\mathbf{x}^k;\mathbf{d}^k)$ is evaluated in three steps [13-15]. The vectors `xk` [5] and `yk` [6] save the `kused` [7] iterates produced by the algorithm so that they can be plotted later. The final approximations to $\mathbf{x}^\star$ [18], $\nabla f(\mathbf{x}^\star)$ [19], and $f(\mathbf{x}^\star)$ [20] are reported along with `kused` [21].

The second stanza uses [26] the `gridcntr.m` routine of §9.1 to compute the objective at points equally spaced between the variable bounds [24,25]. Then [27-29] it finds contour levels equal to the objective value at each of the iterations generated by the algorithm and [32] plots those contours. To show the shape of the function, three more contour levels are plotted [33-36]. Finally the $\mathbf{x}^k$ that were saved earlier [5-6] are plotted [37] to show the algorithm's **convergence trajectory**.


```
% steep.m: use steepest descent to solve the gns problem
epz=1.e-06;
x=[2;2];
for kp=1:20
    xk(kp)=x(1);
    yk(kp)=x(2);
    kused=kp;

    g=gnsg(x);
    if(norm(g) <= epz); break; end

    d=-g;
    numer=-(8*x(1)*d(1)+4*x(2)*d(2)+4*x(1)*d(2)+4*x(2)*d(1)-3*d(1));
    denom= (8*d(1)^2+4*d(2)^2+8*d(1)*d(2));
    alpha=numer/denom;
    x=x+alpha*d;
end
x
g
f=gns(x)
kused

% plot convergence trajectory over contours
xl=[-2;-2];
xh=[3;3];
[xc,yc,zc]=gridcntr(@gns,xl,xh,200);
for kp=1:kused
    vu(kp)=gns([xk(kp);yk(kp)]);
end
hold on
axis('equal')
contour(xc,yc,zc,vu)
vn(1)=20;
vn(2)=10;
vn(3)=5;
contour(xc,yc,zc,vn)
plot(xk,yk)
hold off
print -deps -solid steep.eps
```

```
function f=gns(x)
  f=4*x(1)^2+2*x(2)^2+4*x(1)*x(2)-3*x(1);
end
```

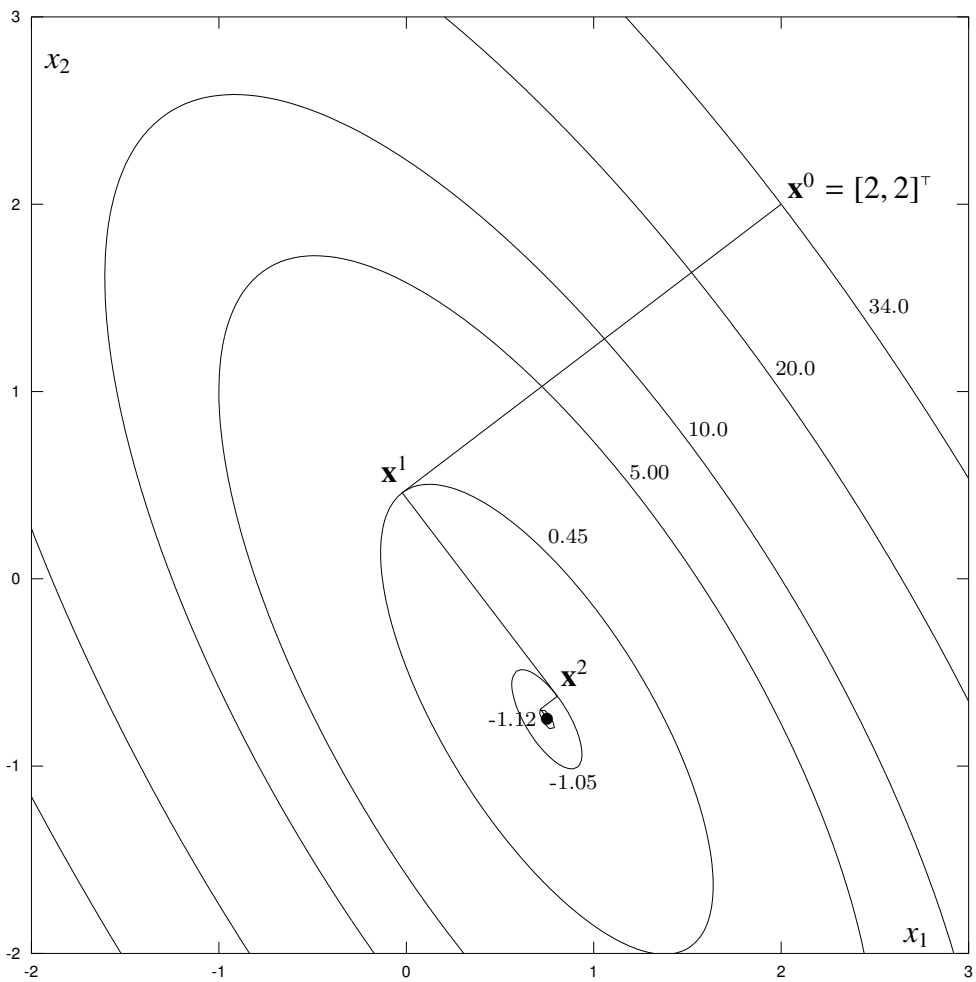
```
function g=gnsg(x)
  g=[8*x(1)+4*x(2)-3; 4*x(2)+4*x(1)];
end
```

```
function h=gnsh(x)
  h=[8,4;4,4];
end
```



```
octave:1> steep
x =

   0.75000
  -0.75000

g =

  -2.4169e-07
   3.1722e-07

f = -1.1250
kused =  12
octave:2> quit
```

Running the program produces the output above and the graph below, which show that the `gns` problem has $f(\mathbf{x}^\star) = -\frac{9}{8}$ at $\mathbf{x}^\star = [\frac{3}{4}, -\frac{3}{4}]^\top$. Notice that each step in the steepest-descent convergence trajectory is orthogonal to the preceding one; this is called **zigzagging**. At the scale of this picture only four of the twelve iterates (three steps) can be seen clearly.



10.5 The Full Step Length

Many optimization techniques approximate $f(\mathbf{x})$ near $\mathbf{x}^k$ by the **quadratic model function**

$$q(\mathbf{x}) = f(\mathbf{x}^k) + \nabla f(\mathbf{x}^k)^\top(\mathbf{x} - \mathbf{x}^k) + \tfrac{1}{2}(\mathbf{x} - \mathbf{x}^k)^\top \mathbf{H}(\mathbf{x}^k)(\mathbf{x} - \mathbf{x}^k)$$

given by the first three terms in the Taylor series expansion for $f(\mathbf{x})$. Another formula for $\alpha^\star$ can be obtained by minimizing this function along the direction of its steepest descent, which is also $-\nabla f(\mathbf{x}^k)$, as follows.

$$\begin{aligned}
\mathbf{x} &= \mathbf{x}^k - \alpha \nabla f(\mathbf{x}^k) \\
q(\mathbf{x}) &= f(\mathbf{x}^k) + \nabla f(\mathbf{x}^k)^\top(-\alpha \nabla f(\mathbf{x}^k)) + \tfrac{1}{2}(-\alpha \nabla f(\mathbf{x}^k))^\top \mathbf{H}(\mathbf{x}^k)(-\alpha \nabla f(\mathbf{x}^k)) \\
\frac{dq(\mathbf{x})}{d\alpha} &= -\nabla f(\mathbf{x}^k)^\top \nabla f(\mathbf{x}^k) + \alpha \nabla f(\mathbf{x}^k)^\top \mathbf{H}(\mathbf{x}^k) \nabla f(\mathbf{x}^k) = 0 \\
\alpha^\star &= \frac{\nabla f(\mathbf{x}^k)^\top \nabla f(\mathbf{x}^k)}{\nabla f(\mathbf{x}^k)^\top \mathbf{H}(\mathbf{x}^k) \nabla f(\mathbf{x}^k)} \\
\mathbf{d}^{\mathrm{S}} &= -\alpha^\star \nabla f(\mathbf{x}^k) = -\frac{\nabla f(\mathbf{x}^k)^\top \nabla f(\mathbf{x}^k)}{\nabla f(\mathbf{x}^k)^\top \mathbf{H}(\mathbf{x}^k) \nabla f(\mathbf{x}^k)} \nabla f(\mathbf{x}^k)
\end{aligned}$$

The vector $\mathbf{d}^{\mathrm{S}}$ is called the **full steepest-descent step**. Despite this name, the $\alpha^\star$ yielding it is usually *not* equal to 1. If $f(\mathbf{x})$ happens to be a quadratic function then $q(\mathbf{x}) = f(\mathbf{x})$ and the analysis above is equivalent to the one we did in §10.4, but if not the full step is usually different from the optimal step we get by minimizing $f(\alpha)$.

For the `gns` problem $f(\mathbf{x})$ is quadratic, and we can compute the full steepest-descent $\alpha^\star$ from the starting point $\mathbf{x}^0 = [2, 2]^\top$ like this.

$$\begin{aligned}
\nabla f(\mathbf{x}^0) &= \begin{bmatrix} 21 \\ 16 \end{bmatrix} \quad \text{from §10.3, so} \quad \nabla f(\mathbf{x}^0)^\top \nabla f(\mathbf{x}^0) = \begin{bmatrix} 21 & 16 \end{bmatrix} \begin{bmatrix} 21 \\ 16 \end{bmatrix} = 697 \\
\mathbf{H}(\mathbf{x}) &= \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1 \partial x_1} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2 \partial x_2} \end{bmatrix} = \begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix} \quad \text{independent of } \mathbf{x} \\
\nabla f(\mathbf{x}^0)^\top \mathbf{H} \nabla f(\mathbf{x}^0) &= \begin{bmatrix} 21 & 16 \end{bmatrix} \begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} 21 \\ 16 \end{bmatrix} = \begin{bmatrix} 21 & 16 \end{bmatrix} \begin{bmatrix} 232 \\ 148 \end{bmatrix} = 7240 \\
\alpha^\star &= \frac{697}{7240}
\end{aligned}$$

This is exactly the result we obtained by minimizing $f(\alpha)$.

Because the formula for $\mathbf{d}^{\mathrm{S}}$ is not specific to a particular problem, we can encapsulate the **full-step steepest-descent algorithm** in the general-purpose routine `sdfs.m` listed on the next page.


```
function [xstar,kp]=sdfs(xzero,kmax,epz,grd,hsn)
  xk=xzero;
  for kp=1:kmax
%     find the uphill direction
      g=grd(xk);
      if(norm(g) <= epz) break; end

%     find the full steepest-descent step downhill
      H=hsn(xk);
      astar=(g'*g)/(g'*H*g);
      d=-astar*g;

%     take the full step
      xk=xk+d;
  end
  xstar=xk;
end
```

Using this routine we can apply the algorithm to any problem for which we have MATLAB functions that compute the gradient and Hessian. For the `gns` problem those are `gnsg.m` and `gnsh.m`, listed in §10.4. Here `kp` numbers steps, of which there are 12 (see §28.4.3).

```
octave:1> [xstar,kp]=sdfs([2;2],20,1e-6,@gnsg,@gnsh)
xstar =

   0.75000
  -0.75000

kp =  12
octave:2> quit
```

10.6 Convergence

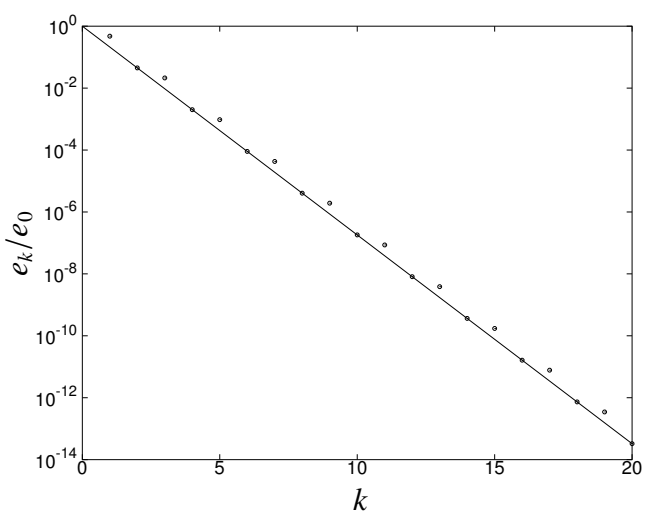
Steepest descent is clearly faster than pure random search, but just how fast is it? Because the `gns` problem is quadratic, the full-step and optimal-step versions of steepest descent generate the same sequence of points $\mathbf{x}^k$ and we can use either to measure the algorithm's order of convergence.

10.6.1 Error Curve

The program on the next page uses `sdfs` in such a way that each iterate in the solution process can be captured. It begins [4-6] by defining the starting and optimal points for the problem and setting a tolerance too small to be achieved in the allowed iterations. Then [9] it sets `x` to the starting point and [10-14] invokes `sdfs` 20 times, each time for a single iteration. An important property of `sdfs` is that it is **serially reusable** [21, p47]; its local variables are not saved from one invocation to the next, so it has no memory. Each invocation of `sdfs` just continues the solution process from the current `xzero` for `kmax` iterations or until convergence is achieved, so in `sdconv.m` each pass through the loop [10-14] replaces $\mathbf{x}^k$ by $\mathbf{x}^{k+1}$. This is a programming strategy that we will use throughout the book to study the behavior of an optimization algorithm that is implemented in a MATLAB routine.

```
 1 % sdconv.m: plot error in gns solution by steepest descent
 2 clear; clf; set(gca,'FontSize',30)
 3
 4 xzero=[2;2];
 5 xstar=[0.75;-0.75];
 6 epz=1e-15;
 7
 8 % generate the iterates and compute the errors
 9 x=xzero;
10 for k=1:20
11     x=sdfs(x,1,epz,@gnsg,@gnsh);
12     error(k)=norm(x-xstar)/norm(xzero-xstar);
13     iters(k)=k;
14 end
15
16 % plot log error versus iterations
17 hold on
18 semilogy(iters,error,'o')
19 semilogy([0,20],[1,error(20)])
20 hold off
21 print -deps -solid sdconv.eps
22 c=10^(log10(error(20))/20)
```

The solution error e_k/e_0 is saved [12] at each iteration along with the iteration count `k` [13]. Then [18] the log relative error $\log_{10}(e_k/e_0)$ is plotted as a function of `k`. When `sdconv.m` is run it produces the graph below. These data clearly fall on a straight line, which has the equation $\log_{10}(e_k/e_0) = k\log_{10} c$ that we derived in §9.2 (also see Exercise 10.9.20).



Thus the steepest-descent algorithm has order of convergence $r = 1$, also called first-order or linear convergence. The left end of the line in the graph above is at $(0, 10^0)$ because of the definition of the log relative error, and its other end, at $(20, \texttt{error(20)})$, determines the convergence constant c for the `gns` problem.

$$c = 10^{\log_{10}(\texttt{error(20)})/20} \approx 0.21173$$

This number, which is printed $\boxed{22}$ by `sdconv.m`, corresponds to the log relative error of -13.484 achieved at `k=20`. The equation of the error curve is therefore

$$\log_{10}(e_k/e_0) \approx k \log_{10}(0.21173) \approx -0.67422k$$

or $e_k \approx e_0 \times 0.21173^k$. This is far better than the sublinear convergence we observed for pure random search.

10.6.2 Bad Conditioning

Now that we have `sdfs.m` we might hope to solve the `rb` problem of §9.1 quickly too. Here is what happens when we try.

```
octave:2> xstar=sdfs([-1.2;1],2,10,1e-6,@rbg,@rbh)
xstar =

  -1.0111
   1.0283

octave:3> xstar=sdfs([-1.2;1],2,100,1e-6,@rbg,@rbh)
xstar =

  -0.80701
   0.65171

octave:4> xstar=sdfs([-1.2;1],2,1000,1e-6,@rbg,@rbh)
xstar =

  -1.5210
   2.3004

octave:5> xstar=sdfs([-1.2;1],2,10000,1e-6,@rbg,@rbh)
xstar =

   1.00000
   1.00000

octave:6> quit
```

The full-step version of steepest descent can solve the `rb` problem, but only if it is permitted to use a huge number of iterations. It can be shown [4, p407] [2, §1.3.2] that when an exact line search is used the convergence constant for steepest descent has the upper bound

$$c \leq \left[\frac{\kappa - 1}{\kappa + 1}\right]^2$$

where κ is the **condition number** of the Hessian matrix at the optimal point,

$$\kappa = \left\| \mathbf{H}(\mathbf{x}^\star) \right\| \left\| \left[\mathbf{H}(\mathbf{x}^\star)\right]^{-1} \right\| \geq 1.$$

The condition number [20, §8.3] tells how close to singular a matrix is (I will have much more to say about matrix conditioning in §18.4.2). If $\kappa(\mathbf{H})$ is close to 1 then $\mathbf{H}$ is said to

be **well-conditioned**. For example, if $\mathbf{H} = \mathbf{I}$ then $\kappa(\mathbf{H}) = 1$, $c = 0$, and steepest descent converges in one iteration. Unfortunately, $\kappa(\mathbf{H})$ is often much bigger than 1, and then c might be only a little less than 1 so that steepest descent converges very slowly. In fact, the algorithm can converge so slowly that $\|\mathbf{x}^{k+1} - \mathbf{x}^k\|$ becomes numerically zero, so that the $\mathbf{x}^k$ stop changing long before they get close to $\mathbf{x}^\star$. One of the things that makes the `rb` problem useful for testing is that $\mathbf{H}(\mathbf{x})$ is **badly conditioned** at $\mathbf{x}^\star$ (see Exercise 10.9.21) and that accounts for the poor performance of steepest descent on this problem. Our experiment used the full step rather than an exact line search, so the convergence constant of the algorithm might have been even worse (i.e., higher) than the bound stated above.

The bad conditioning of the `rb` problem's Hessian near $\mathbf{x}^\star$ corresponds geometrically to the placement of that point in a long thin valley, which might therefore be regarded as a "valley of the shadow of death" for steepest descent and, as we shall see, for other algorithms.

10.6.3 Vector and Matrix Norms

Ever since §3 I have used the notation $\|\mathbf{x}\|$ to denote the length of a vector. More generally, a **norm** is a function that maps each element of a vector space to a scalar and has these properties.

$$\begin{aligned} \|\mathbf{x}\| &\geq 0 \quad \text{with equality if and only if } \mathbf{x} = \mathbf{0} \\ \|a\mathbf{x}\| &= |a|\,\|\mathbf{x}\| \quad \text{for any scalar } a \\ \|\mathbf{x} + \mathbf{y}\| &\leq \|\mathbf{x}\| + \|\mathbf{y}\| \quad \text{triangle inequality} \end{aligned}$$

I will always use $|\bullet|$ to denote absolute value and $\|\bullet\|$ to denote a norm.

Norms of vectors. For $\mathbf{x} \in \mathbb{R}^n$ the norms that are most frequently useful in optimization are these.

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = +\sqrt{\sum_{j=1}^{n} x_j^2} = +\sqrt{\mathbf{x}^\top\mathbf{x}} \qquad \|\mathbf{x}\|_1 = \sum_{j=1}^{n} |x_j| \qquad \|\mathbf{x}\|_\infty = \max_j \{|x_j|\}$$

The subscript 2 denotes the **Euclidean norm** or **inner-product norm**. If $x \in \mathbb{R}^1$ and $f(x)$ is Lebesgue-integrable on an interval I, or $f \in L(I)$, and if also $f^2 \in L(I)$ then $<f, f>= \int_I [f(x)]^2\, dx$ is the inner product of $f(x)$ with itself and $\|f\|_2 = \sqrt{<f, f>}$ is called the L^2 norm of f. Analogously if $\mathbf{x} \in \mathbb{R}^n$ and $f(\mathbf{x}) = \mathbf{x}$, then $<f, f>= \mathbf{x}^\top\mathbf{x}$ is the inner product of $\mathbf{x}$ with itself, and $\sqrt{\mathbf{x}^\top\mathbf{x}}$ is also called the $\mathbf{L^2}$ **norm** or just the **2-norm** of $\mathbf{x}$ [8, §10.21].

Following this terminology, the sum of absolute values is often called the $\mathbf{L^1}$ **norm** or the **1-norm** and the **max-norm** is also called the $\mathbf{L^\infty}$ **norm** or the **infinity-norm**. When no subscript appears on a norm, it is assumed to be the 2-norm.

In addition to the properties listed above as characteristic of any norm, the 2-norm has several others [148, §9.1.2] given at the top of the next page. These assume that $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^n$, and $\mathbf{A} \in \mathbb{R}^{m\times n}$, and that $\mathbf{1} \in \mathbb{R}^n$ is a vector of 1's.

$$
\begin{aligned}
||\mathbf{x}||^2 &= \sum_{j=1}^{n} x_j^2 = \mathbf{x}^\top\mathbf{x} \\
||\mathbf{x} \pm \mathbf{y}||^2 &= ||\mathbf{x}||^2 + ||\mathbf{y}||^2 \pm 2\mathbf{x}^\top\mathbf{y} \\
||\mathbf{A}\mathbf{x}||^2 &= (\mathbf{A}\mathbf{x})^\top\mathbf{A}\mathbf{x} = \mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} \\
|\mathbf{x}^\top\mathbf{y}| &\leq ||\mathbf{x}||\,||\mathbf{y}|| \quad \text{Cauchy-Schwartz inequality} \\
\nabla_{\mathbf{x}}||\mathbf{x}|| &= \mathbf{x}/||\mathbf{x}|| \quad \text{if } ||\mathbf{x}|| \neq 0 \\
||a|| &= |a| \quad \text{for any scalar } a \\
||\mathbf{1}|| &= +\sqrt{n}
\end{aligned}
$$

If $||\mathbf{x}|| = 1$ then $\mathbf{x}$ is a **unit vector**. The three norms listed above are related by the following inequalities [67, §2.2-2.3] which hold for all vectors $\mathbf{x} \in \mathbb{R}^n$.

$$
\begin{array}{rclcr}
||\mathbf{x}||_2 & \leq & ||\mathbf{x}||_1 & \leq & \sqrt{n}\,||\mathbf{x}||_2 \\
||\mathbf{x}||_\infty & \leq & ||\mathbf{x}||_2 & \leq & \sqrt{n}\,||\mathbf{x}||_\infty \\
||\mathbf{x}||_\infty & \leq & ||\mathbf{x}||_1 & \leq & n\,||\mathbf{x}||_\infty
\end{array}
$$

To find $||\mathbf{x}||$ with MATLAB or Octave use `norm(x)` or `norm(x,2)`.

NORMS OF MATRICES. When $\mathbf{A} \in \mathbb{R}^{m\times n}$ the matrix norm that is most frequently useful in optimization is [147, §7.2]

$$||\mathbf{A}|| = ||\mathbf{A}||_2 = \max_{\mathbf{x}\neq\mathbf{0}} \frac{||\mathbf{A}\mathbf{x}||_2}{||\mathbf{x}||_2} = +\sqrt{\lambda_{\max}}$$

where $\lambda_{\max}$ is the maximum eigenvalue of $\mathbf{A}^\top\mathbf{A}$. (The matrix $\mathbf{A}^\top\mathbf{A}$ is symmetric, so $\lambda_{\max}$ is always real and $\sqrt{\lambda_{\max}}$ is the largest singular value of $\mathbf{A}$.) From this definition we have the inequality

$$||\mathbf{A}\mathbf{x}|| \leq ||\mathbf{A}||\,||\mathbf{x}||$$

for all $\mathbf{x} \in \mathbb{R}^n$, with equality holding for at least one nonzero $\mathbf{x}$. If $\mathbf{B} \in \mathbb{R}^{n\times q}$ (i.e., if the matrix product $\mathbf{AB}$ is conformable) then

$$||\mathbf{AB}|| \leq ||\mathbf{A}||\,||\mathbf{B}||$$

and if $\mathbf{C} \in \mathbb{R}^{m\times n}$ (i.e., if $\mathbf{C}$ has the same size as $\mathbf{A}$) then

$$||(\mathbf{A} + \mathbf{C})|| \leq ||\mathbf{A}|| + ||\mathbf{C}||.$$

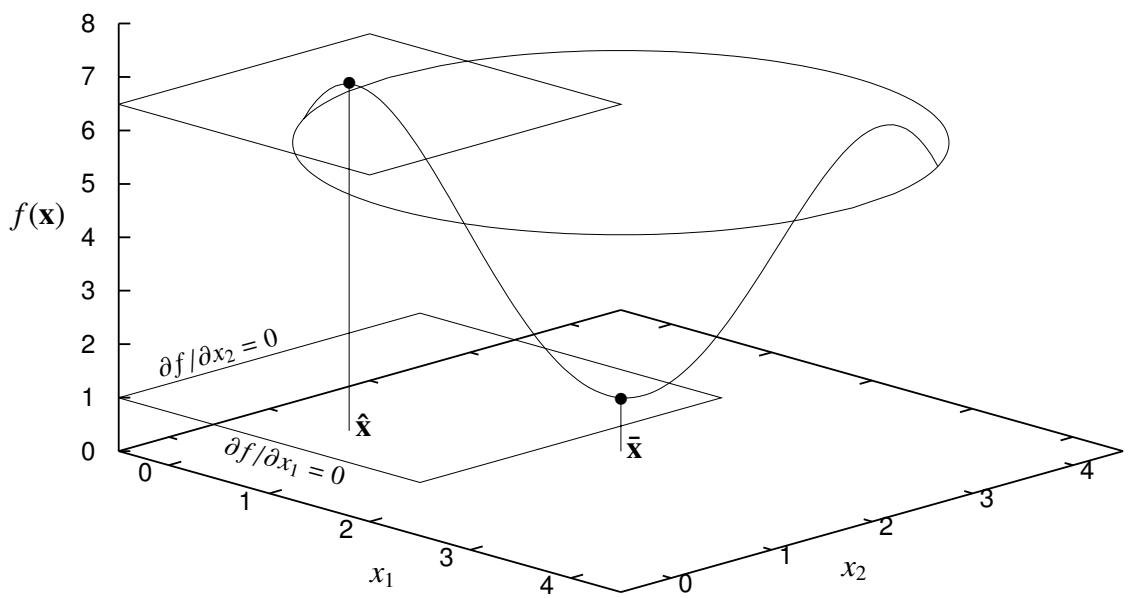
To find $||\mathbf{A}||$ with MATLAB or Octave use `norm(A)` or `norm(A,2)`.

10.7 Local Minima

The `rb` problem and the `gns` problem have $n = 2$, so for each we were able to draw a contour diagram and know that the point we identified as $\mathbf{x}^\star$ is the global minimum. If $n > 2$, how can we tell whether a given point $\bar{\mathbf{x}}$ is any kind of minimum?

Since §8 we have made use of the fact that if $f(x)$ is smooth and $\bar{x} \in \mathbb{R}^1$ is a minimizing point, then df/dx at that point is zero. In higher dimensions, if $\bar{\mathbf{x}} \in \mathbb{R}^n$ is a minimizing point then for $j = 1 \ldots n$ each of the partial derivatives $\partial f/\partial x_j$ must be zero there. In a graph of $f(\mathbf{x})$ this makes the **tangent hyperplane** to the function at $\bar{\mathbf{x}}$ horizontal.

The graph plotted below, which is of $f(r) = \frac{14}{5}r^2 - \frac{5}{14}r^4 + 1$ where $r = \sqrt{(x_1-2)^2 + (x_2-2)^2}$, looks like an inverted sombrero. For clarity only a single cross section is drawn, but rotated about a vertical axis through $\bar{\mathbf{x}}$ it describes a ridge running around the top of the figure.



The hyperplane that is tangent (at the lower dot) to the graph over the minimizing point $\bar{\mathbf{x}}$ intersects the x_1—$f(\mathbf{x})$ coordinate plane in a straight line whose slope is $\partial f/\partial x_1$ and the x_2—$f(\mathbf{x})$ coordinate plane in a straight line whose slope is $\partial f/\partial x_2$. Because these lines are horizontal, both partial derivatives are zero. This observation generalizes to $\mathbb{R}^n$ as follows [1, p167] [5, p14] [4, p359].

Theorem: first-order necessary conditions

if $f(\mathbf{x})$ is differentiable at $\bar{\mathbf{x}}$
$\bar{\mathbf{x}}$ is a local minimum

then $\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$

Any point $\bar{\mathbf{x}}$ where $\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$ is called a **stationary point**. Minima are stationary, but so are maxima such as $\hat{\mathbf{x}}$ (and all of the other points around the ridge) in the figure. Depending on the function it is also possible for the gradient to be zero at points that are neither maxima nor minima (e.g., saddle points [161, p45-46]). Thus,

$$\begin{aligned} \bar{\mathbf{x}} \text{ is a local minimum} \quad &\Rightarrow \quad \nabla f(\bar{\mathbf{x}}) = \mathbf{0} \\ \text{but } \nabla f(\bar{\mathbf{x}}) = \mathbf{0} \quad &\not\Rightarrow \quad \bar{\mathbf{x}} \text{ is a local minimum.} \end{aligned}$$

Since §8 we have also made use of the fact that if $f(x)$ is smooth and $\bar{x} \in \mathbb{R}^1$ is a point where $df/dx = 0$, then whether $\bar{x}$ is a minimizing point depends on the sign of d^2f/dx^2 there. In higher dimensions, if $\bar{\mathbf{x}} \in \mathbb{R}^n$ is a stationary point then whether it is a minimum depends on the **definiteness** of the Hessian matrix at that point. A matrix $\mathbf{M}$ is [67, §4.2]

$$\begin{aligned} \textbf{positive definite} \quad &\Leftrightarrow \quad \mathbf{w}^\top \mathbf{M}\mathbf{w} > 0 \text{ for all } \mathbf{w} \neq \mathbf{0} \\ \textbf{positive semidefinite} \quad &\Leftrightarrow \quad \mathbf{w}^\top \mathbf{M}\mathbf{w} \geq 0 \text{ for all } \mathbf{w}. \end{aligned}$$

The results below [1, p168-169] [5, p15-16] [4, p359-360] summarize the classification of stationary points based on the definiteness of the Hessian matrix.

Theorem: second-order necessary conditions

if $f(\mathbf{x})$ is twice differentiable at $\bar{\mathbf{x}}$
$\bar{\mathbf{x}}$ is a local minimum

then $\mathbf{H}(\bar{\mathbf{x}})$ is positive semidefinite

Theorem: strong second-order sufficient conditions

if $f(\mathbf{x})$ is twice differentiable at $\bar{\mathbf{x}}$
$\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$
$\mathbf{H}(\bar{\mathbf{x}})$ is positive definite

then $\bar{\mathbf{x}}$ is a strict local minimum

The implications in these theorems go in only one direction, as illustrated by the classic example of $f(x) = x^4$. This function obviously has a strict local minimum at $\bar{x} = 0$, but $\mathbf{H}(\bar{x}) = [d^2f/dx^2] = 12\bar{x}^2 = 0$ so its Hessian matrix is only positive *semi*definite there. Thus,

$$\begin{aligned} \nabla f(\bar{\mathbf{x}}) = \mathbf{0} \text{ and } \mathbf{H}(\bar{\mathbf{x}}) \text{ positive definite} \quad &\Rightarrow \quad \bar{\mathbf{x}} \text{ is a strict local minimum} \\ \text{but } \bar{\mathbf{x}} \text{ a strict local minimum} \quad &\not\Rightarrow \quad \mathbf{H}(\bar{\mathbf{x}}) \text{ is positive definite.} \end{aligned}$$

If $\mathbf{H}(\mathbf{x})$ is only positive semidefinite it might still be possible to deduce that $\bar{\mathbf{x}}$ is a local minimum, though not necessarily a strict one, by using the following result [3, p271] (also see Exercise 10.9.37).

Theorem: weak second-order sufficient conditions

if $f(\mathbf{x})$ is twice differentiable
$\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$
$\mathbf{H}(\mathbf{x})$ is positive semidefinite for all $\mathbf{x} \in \mathcal{N}_\varepsilon(\bar{\mathbf{x}})$

then $\bar{\mathbf{x}}$ is a local minimum

These results show by their one-directional and equivocal character that the theory of nonlinear programming has rather limited power. This impression will only be confirmed when we study constrained optimization in §15 and §16, and might help to explain the practical importance of numerical methods. However, the points that we identified graphically as global minima for the `gns` and `rb` problems can at least be confirmed analytically to be local minima by using the theorems stated above.

For the `gns` problem, $f(\mathbf{x}) = 4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1$ and $\mathbf{x}^\star = [\frac{3}{4}, -\frac{3}{4}]^\top$.

$$\nabla f(\mathbf{x}) = \begin{bmatrix} 8x_1 + 4x_2 - 3 \\ 4x_2 + 4x_1 \end{bmatrix} \quad \text{so} \quad \nabla f(\mathbf{x}^\star) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \mathbf{H} = \begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix}$$

This Hessian is independent of $\mathbf{x}$, and using the definition above we can show that it is positive definite.

$$\mathbf{w}^\top\mathbf{H}\mathbf{w} = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = 8w_1^2 + 8w_1w_2 + 4w_2^2 = 4w_1^2 + (2w_1 + 2w_2)^2$$

The final expression is a sum of squares so it can't be negative. The only way it could be zero is if $w_1 = 0$ and $w_2 = 0$, but that is impossible if $\mathbf{w} \neq \mathbf{0}$. Thus $\mathbf{w}^\top\mathbf{H}\mathbf{w} > 0$ for all $\mathbf{w} \neq \mathbf{0}$. We found that $f(x)$ is twice differentiable, that $\nabla f(\mathbf{x}^\star) = \mathbf{0}$, and that $\mathbf{H}$ is positive definite, so the strong second-order sufficient conditions are satisfied and $\mathbf{x}^\star$ is a strict local minimum.

For the `rb` problem, $f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ and $\mathbf{x}^\star = [1, 1]^\top$.

$$\nabla f(\mathbf{x}) = \begin{bmatrix} -400x_1(x_2 - x_1^2) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix} \quad \text{so} \quad \nabla f(\mathbf{x}^\star) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} -400x_2 + 1200x_1^2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix} \quad \text{so} \quad \mathbf{H}(\mathbf{x}^\star) = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}$$

This Hessian depends on $\mathbf{x}$ and is *not* positive definite everywhere (see §13.1). It *is* positive definite at $\mathbf{x}^\star$ (that is hard to show by using the definition, but easy using other techniques you will learn in §11). The strong second-order sufficient conditions are therefore satisfied at $\mathbf{x}^\star$ for this problem too, so its $\mathbf{x}^\star$ is also a strict local minimum.

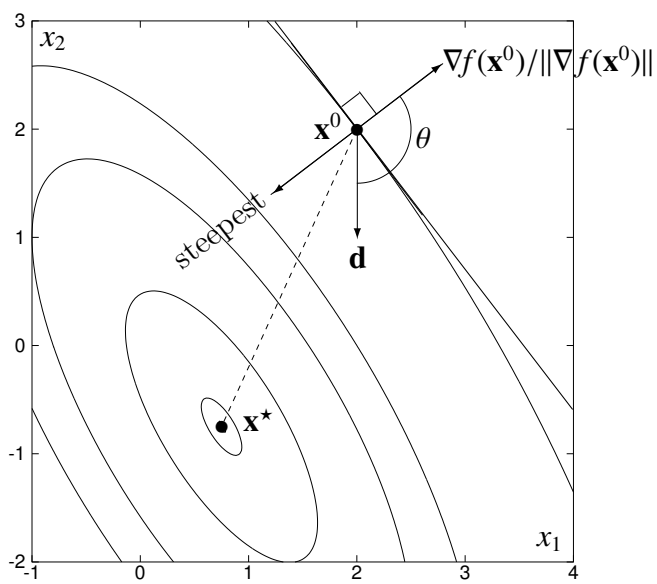
10.8 Open Questions

In this Chapter we developed our first practical *algorithm* for numerical optimization, discovering along the way some important ideas about the *theory* of nonlinear programming. I hope that you are curious rather than satisfied, because we have raised several questions that remain to be answered.

- When n is bigger than 2 or 3, so that we cannot draw a contour diagram, can we ever be sure that we have found a *global* minimizing point? If so, can we ever establish that the global minimum is unique? The conditions you have learned so far, when they hold at all, let us conclude only that a point is a *local* minimum.

- In using the steepest-descent algorithm, how can we find the optimal step length $\alpha^\star$ if we can't solve $df/d\alpha = 0$ analytically? We can of course use the full steepest-descent step instead, but usually the optimal step is different and results in better performance.

- Might it be possible to avoid zigzagging or to get quadratic convergence by moving from each $\mathbf{x}^k$ in some direction other than that of the negative gradient? The picture below shows some contours of the `gns` problem along with the normalized gradient at $\mathbf{x}^0$ and the hyperplane to which $\nabla f(\mathbf{x}^0)$ is orthogonal.



Any vector $\mathbf{d}$ in the halfspace where $90° < \theta < 270°$, so that $\nabla f(\mathbf{x})^\top \mathbf{d} < 0$, is a **descent direction**. Some descent directions result in a more direct path to $\mathbf{x}^\star$ than others, and for this problem the direction of the dashed line would take us there in just one step.

Each of the next three Chapters will take up one of these important questions.

10.9 Exercises

10.9.1 [E] A function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^1$ descends most rapidly in what direction?

10.9.2 [E] The quadratic Taylor series approximation to a function is discussed in §10.1. (a) Write down the quadratic Taylor series approximation to $f(x) = e^x$ about the point $x = 1$. (b) Write down the quadratic Taylor series approximation to $f(\mathbf{x}) = e^{x_1 x_2}$ about the point $\mathbf{x} = [1, -1]^\top$.

10.9.3 [E] What is necessary in order for the Hessian matrix of a function to be symmetric?

10.9.4 [H] Consider the vectors $\mathbf{x} = [1, 2, 3]^\top$ and $\mathbf{y} = [-1, 0, 2]^\top$. (a) Compute $\mathbf{x}^\top\mathbf{y} = \sum x_j y_j$. (b) Find θ, the angle between $\mathbf{x}$ and $\mathbf{y}$ measured in the plane containing them both, and use it to compute $\mathbf{x}^\top\mathbf{y} = \|x\| \times \|y\| \times \cos(\theta)$.

10.9.5 [E] Starting from the $\mathbf{x}^1$ found in §10.3, continue the steepest-descent process by hand, finding $\mathbf{d}^1$, α_1, and $\mathbf{x}^2$. Is $f(\mathbf{x}^2) < f(\mathbf{x}^1)$?

10.9.6 [P] Modify the pseudocode given in §10.4 for the steepest-descent algorithm to keep a record value and a record point.

10.9.7 [E] Explain the difference between the min operator and the argmin operator. What is argmin(min($f(\alpha)$))?

10.9.8 [H] From the first expression given in §10.4 for $f(\mathbf{x} + \alpha\mathbf{d})$, derive the second.

10.9.9 [P] What do we mean by an algorithm's *convergence trajectory*? Write a MATLAB program that draws contours of the `rb` problem and plots over them the convergence trajectory of record points generated by the pure random search algorithm when it is used to solve that problem.

10.9.10 [P] Revise the `steep.m` program of §10.4 to solve the `gns` problem from the starting point $[-1, 1]^\top$ and use it to produce a graph showing the convergence trajectory. Are successive steepest-descent steps still orthogonal?

10.9.11 [P] Derive an algebraic formula for $\alpha^\star(\mathbf{x}; \mathbf{d})$ for the `rb` problem, and modify `steep.m` to use it. Hint: use Maple or Mathematica. Does the optimal-step steepest-descent algorithm converge to $\mathbf{x}^\star = [1, 1]^\top$?

10.9.12 [E] What is *zigzagging*, and why does it happen?

10.9.13 [H] In the example of §10.4.0, the convergence trajectory of the optimal-step steepest-descent algorithm is made up of steps each of which is orthogonal to the previous one. (a) Why does that happen? (b) Does it happen even if $f(\mathbf{x})$ is not quadratic? (c) Are successive steps of the full-step steepest-descent algorithm also orthogonal?

10.9.14 [H] Using the definition of the quadratic model function given in §10.5, find the $q(\mathbf{x})$ that approximates the objective function $f(\mathbf{x}) = 4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1$ of the `gns` problem. Show that $q(\mathbf{x}) = f(\mathbf{x})$. Why are these functions equal?

10.9.15 [E] The full step length $\alpha^\star$ derived in §10.5 is usually not equal to 1. What must be true of $\mathbf{H}$ in order for $\alpha^\star$ to equal 1 exactly? What does that mean about $f(\mathbf{x})$? How many iterations of the full-step steepest descent algorithm are required to minimize $f(\mathbf{x})$?

10.9.16 [E] How does the convergence trajectory of the optimal-step steepest-descent algorithm differ from that of the full-step steepest-descent algorithm when both are used to solve the `gns` problem? Explain.

10.9.17 [P] Use `sdfs.m` to solve the Himmelblau 28 problem [80, p428],

$$\text{minimize } f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2.$$

Start from $\mathbf{x}^0 = [1, 1]^\top$ and show that $f(\mathbf{x}^\star) = 0$. Is the optimal point you found the only point that yields $f(\mathbf{x}) = 0$?

10.9.18 [H] In §10.6.1 a programming strategy is described for testing an optimization method that is implemented as a MATLAB function. (a) What is the purpose of using the strategy that is described? (b) Explain the properties that the optimization routine must have in order for the strategy to be used.

10.9.19 [H] When the steepest-descent algorithm converges, it typically generates iterates that yield an error curve having the formula $e_k = e_0 \times c^k$. (a) What is the algorithm's order of convergence? (b) Explain how to find the convergence constant c from experimental measurements of the e_k.

10.9.20 [P] In §10.6.1 we drew a straight line through the data of log relative error versus `k`, but half of the experimental points lie above the line. (a) Why is that? Experimenting with steepest descent for minimizing some other quadratic test functions might shed light on this question. (b) Why would it not make sense to displace the straight line to the right slightly so that it passes between the data points, leaving half below and half above? (c) Does the model that we proposed in §9.2 for explaining algorithm convergence make predictions that are quantitatively perfect in every instance? If not, why not? Are its predictions useful anyway? Explain.

10.9.21 [P] In §10.6.2 we saw that steepest descent converges very slowly in solving the `rb` problem; none of the digits in $\mathbf{x}_{1000}$ were correct, but in $\mathbf{x}_{10000}$ all six of the digits displayed were correct. (a) Conduct your own experiments to determine the smallest number of iterations $k^\star$ between 1000 and 10000 for which $\mathbf{x}_{k^\star}$ is correct to six digits. (b) Assuming linear convergence, use your value of $k^\star$ to estimate the convergence constant c for this problem. (c) Find the condition number κ of $\mathbf{H}(\mathbf{x}^\star)$. (d) Compute an upper bound on the value of c based on κ. Is the convergence constant you estimated experimentally less than or equal to this upper bound? If not, suggest a possible reason why.

10.9.22 [H] Show that steepest descent minimizes $f(\mathbf{x}) = \mathbf{x}^\top\mathbf{x}$ in one step. Explain how this result follows from $\mathbf{H}(\mathbf{x}^\star)$ for this problem.

10.9.23 [E] State the three properties that characterize every norm of a vector. State one additional property that characterizes the Euclidean norm of a vector.

10.9.24 [E] In §8.6.4 we studied LAV regression. Why is LAV regression sometimes referred to as L^1 regression? In LAV regression, how is the sum of the absolute values of the deviations related to the square root of the sum of their squares?

10.9.25 [P] Find the Euclidean norm of this matrix

$$\mathbf{A} = \begin{bmatrix} 7 & 5 \\ 5 & 3 \end{bmatrix}$$

(a) as $\sqrt{\lambda_{max}}$, where λ_{max} is the maximum eigenvalue of $\mathbf{A}^\top\mathbf{A}$; (b) by using the MATLAB `norm()` function.

10.9.26 [H] Using the definition of a matrix norm, prove the inequality $\|\mathbf{Ax}\| \leq \|\mathbf{A}\| \times \|\mathbf{x}\|$.

10.9.27 [E] Find matrices $\mathbf{A}$ and $\mathbf{B}$ such that $\|\mathbf{AB}\| \leq \|\mathbf{A}\| \times \|\mathbf{B}\|$.

10.9.28 [E] What is true of a hyperplane that is tangent to the graph of a function at a minimizing point? How is this related to the gradient of the function at that point?

10.9.29 [E] What must be true at a stationary point?

10.9.30 [E] If a matrix is positive definite, must it be positive semidefinite? If so, prove that by using the definitions given in §10.7; if not, find a counterexample.

10.9.31 [E] Prove that the identity matrix is positive definite, and that the zero matrix is positive semidefinite; then write down a matrix that is neither.

10.9.32 [E] Prove that if $\mathbf{A}$ and $\mathbf{B}$ are square matrices of the same size and both are positive definite, then the matrix $\mathbf{A} + \mathbf{B}$ is positive definite.

10.9.33 [H] In §10.7 it is shown for the `rb` problem that

$$\mathbf{H}(\mathbf{x}^\star) = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}.$$

Use the definition of a positive-definite matrix to prove that this Hessian matrix is positive definite.

10.9.34 [E] This Exercise asks you to recall the four theorems that are stated in §10.7, ideally from memory without looking them up. (a) State the first-order necessary conditions. (b) State the second-order necessary conditions. (c) State the strong second-order sufficient conditions. (d) State the weak second-order sufficient conditions.

10.9.35 [E] Provide a counterexample to show that if $\bar{\mathbf{x}}$ is a strict local minimum, $\mathbf{H}(\bar{\mathbf{x}})$ need not be positive definite. Then construct a function that has a strict local minimum where the Hessian matrix *is* positive definite.

10.9.36 [E] If $\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$, is it possible that $\bar{\mathbf{x}}$ is a local minimum? Is it certain? If $\bar{\mathbf{x}}$ is a strict local minimum, is it possible that $\mathbf{H}(\bar{\mathbf{x}})$ is positive definite? Is it certain? Explain the difference between necessary conditions and sufficient conditions.

10.9.37 [H] A truncated Taylor's series is introduced in §10.1 to approximate $f(\mathbf{x})$ near $\bar{\mathbf{x}}$. **Taylor's theorem** [110, p224-225] says that there exists a point between $\mathbf{x}$ and $\bar{\mathbf{x}}$, say $\bar{\mathbf{x}} + \theta(\mathbf{x} - \bar{\mathbf{x}})$ with $\theta \in [0, 1]$, such that if $\mathbf{H}(\mathbf{x})$ is evaluated there instead of at $\bar{\mathbf{x}}$, the quadratic approximation to f is *exact* at $\mathbf{x}$. Use Taylor's theorem to prove the theorem of §10.7 about the weak second-order sufficient conditions. Hint: suppose that $\mathbf{H}(\mathbf{x})$ is positive semidefinite for all $\mathbf{x} \in \mathcal{N}_\varepsilon(\bar{\mathbf{x}})$ and pick the point $\mathbf{w} \in \mathcal{N}_\varepsilon(\bar{\mathbf{x}})$. Then use Taylor's theorem and the definition of a positive semidefinite matrix to show that $f(\mathbf{w}) \geq f(\bar{\mathbf{x}})$.

10.9.38 [E] What is necessary in order for a vector $\mathbf{p}$ to be a descent direction? Show that if θ is defined as in the graph of §10.8, $\nabla f(\mathbf{x})^\top \mathbf{d} < 0$ if and only if $90° < \theta < 270°$.

11

Convexity

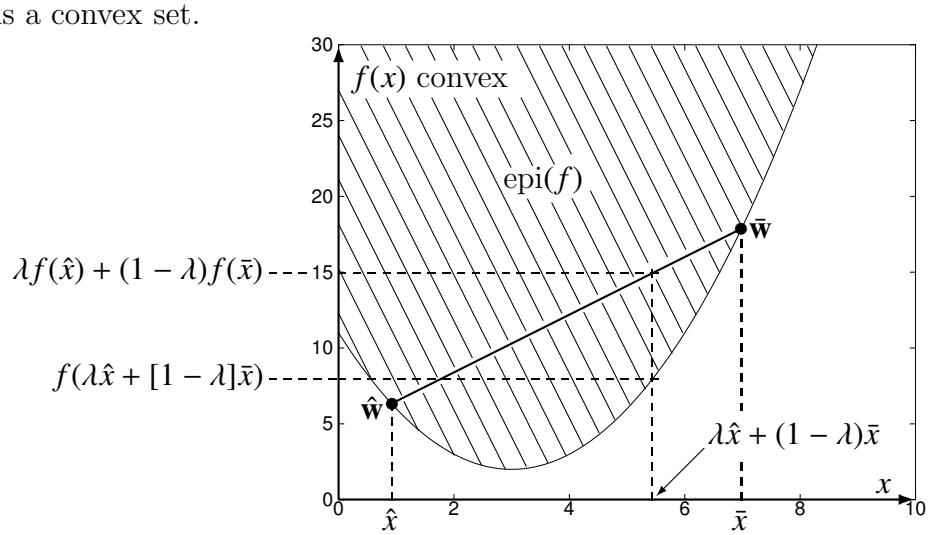
In §10.7 we saw that it is sometimes possible by using the second-order sufficient conditions to establish analytically that a given $\mathbf{x}^\star$ is a *local* minimizing point for an unconstrained optimization problem. But in §9.3 we saw that it is possible for a nonlinear program to have multiple local minima, some of which are not *global* minima. In this Chapter we will see that if the objective has the global property of being a convex function then every minimizing point $\mathbf{x}^\star$ is a global minimum.

11.1 Convex Functions

Recall from §3.5 that a set $\mathbb{S}$ is convex if and only if for all $\hat{\mathbf{w}}$ and $\bar{\mathbf{w}}$

$$\left.\begin{array}{l}\hat{\mathbf{w}} \in \mathbb{S} \\ \bar{\mathbf{w}} \in \mathbb{S}\end{array}\right\} \Rightarrow \lambda\hat{\mathbf{w}} + (1-\lambda)\bar{\mathbf{w}} \in \mathbb{S} \quad \text{for all } \lambda \in [0,1].$$

In the figure below, for any distinct $\hat{x}$ and $\bar{x}$ the points $\hat{\mathbf{w}} = [\hat{x}, f(\hat{x})]^\top$ and $\bar{\mathbf{w}} = [\bar{x}, f(\bar{x})]^\top$ are in epi(f) and so is the chord between them. Thus $\lambda\hat{\mathbf{w}} + (1-\lambda)\bar{\mathbf{w}} \in \text{epi(f)}$ for all $\lambda \in [0,1]$ and epi(f) is a convex set.

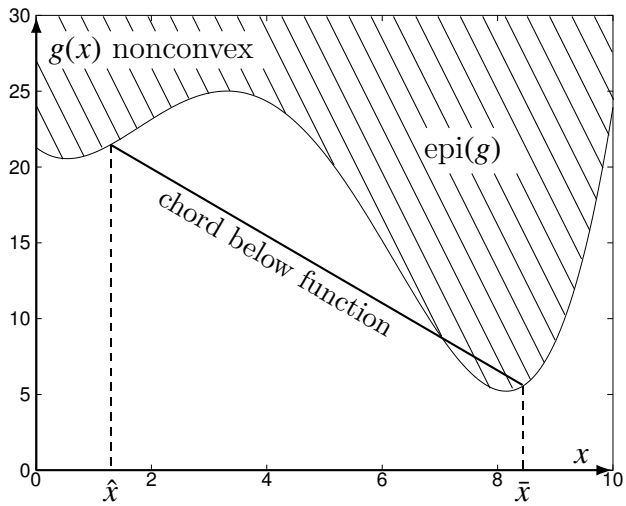


In general the set

$$\text{epi}(f) = \left\{[\mathbf{x}, y]^\top \in \mathbb{R}^{n+1} \;\middle|\; y \geq f(\mathbf{x})\right\}$$

is called the **epigraph** of $f(\mathbf{x})$, and [1, Theorem 3.2.2] it is a convex set if and only if $f(\mathbf{x})$ is a convex function. Thus the function $f(x) = (x-3)^2 + 2$ graphed above is a convex function.

The epigraph of $g(x) = \frac{1}{100}(\frac{3}{2}x - 6)^4 - \frac{2}{3}(\frac{3}{2}x - 5)^2 + 25$ pictured below is not a convex set, so $g(x)$ is not a convex function.



For all points on a chord between $(\hat{x}, f(\hat{x}))$ and $(\bar{x}, f(\bar{x}))$ to be in epi(f), the graph of the function must be below (or on) the chord, as in the graph on the previous page. In other words, the function value at any convex combination of the points must be no greater than the same convex combination of the function values at the points, or

$$\boxed{\underbrace{f(\lambda\hat{\mathbf{x}} + [1-\lambda]\bar{\mathbf{x}})}_{\text{value of function}} \le \underbrace{\lambda f(\hat{\mathbf{x}}) + (1-\lambda) f(\bar{\mathbf{x}})}_{\text{height of chord}} \quad \text{for all} \quad \hat{\mathbf{x}}, \bar{\mathbf{x}}, \text{ and } \lambda \in [0,1].}$$

We will take this as the definition of a **convex function**. The chord between $(\hat{x}, g(\hat{x}))$ and $(\bar{x}, g(\bar{x}))$ in the picture above has some points below the graph of the function, so using this definition we see once again that $g(x)$ is nonconvex.

If the boxed inequality is satisfied strictly for $\hat{\mathbf{x}} \neq \bar{\mathbf{x}}$ then $f(x)$ is **strictly convex**. From this definition, a function that is strictly convex is also convex. If $f(x)$ is a convex function then $-f(x)$ is a **concave function**; if $f(x)$ is strictly convex then $-f(x)$ is **strictly concave**. Most functions are neither convex nor concave, but a linear function is both.

First-year calculus textbooks (e.g., [146, p275]) call convex functions "concave up" and concave functions "concave down," but this terminology is seldom used anywhere else so I will avoid it. We will likewise have no use for the notion that a set might be concave like a mirror or a lens, so our sets will be either convex or nonconvex.

11.2 The Support Inequality

Our definition of convexity says that the graph of the function is not above any chord, but it is also not below any tangent [4, §2.3.1] [1, Theorem 3.3.3]. If a convex function is

smooth, this means that its graph is not below any first-order Taylor series approximation. By algebraically rearranging the §11.1 definition of a convex function we find

$$\begin{aligned} f(\lambda\hat{\mathbf{x}} + [1-\lambda]\bar{\mathbf{x}}) &\leq \lambda f(\hat{\mathbf{x}}) + (1-\lambda) f(\bar{\mathbf{x}}) \\ f(\bar{\mathbf{x}} + \lambda[\hat{\mathbf{x}} - \bar{\mathbf{x}}]) &\leq \lambda f(\hat{\mathbf{x}}) + f(\bar{\mathbf{x}}) - \lambda f(\bar{\mathbf{x}}) \\ f(\bar{\mathbf{x}} + \lambda[\hat{\mathbf{x}} - \bar{\mathbf{x}}]) - f(\bar{\mathbf{x}}) &\leq \lambda[f(\hat{\mathbf{x}}) - f(\bar{\mathbf{x}})] \\ f(\bar{\mathbf{x}} + \lambda\mathbf{a}) - f(\bar{\mathbf{x}}) &\leq \lambda[f(\hat{\mathbf{x}}) - f(\bar{\mathbf{x}})] \end{aligned}$$

where $\mathbf{a} = [\hat{\mathbf{x}} - \bar{\mathbf{x}}] \neq \mathbf{0}$. Expanding the first term in the last line by Taylor's series,

$$f(\bar{\mathbf{x}} + \lambda\mathbf{a}) = f(\bar{\mathbf{x}}) + \lambda\mathbf{a}^\top\nabla f(\bar{\mathbf{x}}) + \text{higher order terms}$$

so

$$f(\bar{\mathbf{x}}) + \lambda\mathbf{a}^\top\nabla f(\bar{\mathbf{x}}) + \text{higher order terms} - f(\bar{\mathbf{x}}) \leq \lambda[f(\hat{\mathbf{x}}) - f(\bar{\mathbf{x}})].$$

or, for $\lambda > 0$,

$$\mathbf{a}^\top\nabla f(\bar{\mathbf{x}}) + \text{terms of order } \lambda \text{ and higher} \leq f(\hat{\mathbf{x}}) - f(\bar{\mathbf{x}}).$$

Now in the limit as $\lambda \to 0$ we find at $\bar{\mathbf{x}}$ that

$$f(\hat{\mathbf{x}}) \geq f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\hat{\mathbf{x}} - \bar{\mathbf{x}}).$$

If $f(\mathbf{x})$ is a convex function this inequality must be satisfied for *all* $\hat{\mathbf{x}}$ and $\bar{\mathbf{x}}$. Conversely, if $f(\mathbf{x})$ satisfies this inequality for all $\hat{\mathbf{x}}$ and $\bar{\mathbf{x}}$ then it must be a convex function. To see this, let $\mathbf{y} = \lambda\hat{\mathbf{x}} + (1-\lambda)\bar{\mathbf{x}}$. Then

$$\begin{aligned} f(\hat{\mathbf{x}}) &\geq f(\mathbf{y}) + \nabla f(\mathbf{y})^\top(\hat{\mathbf{x}} - \mathbf{y}) \\ f(\bar{\mathbf{x}}) &\geq f(\mathbf{y}) + \nabla f(\mathbf{y})^\top(\bar{\mathbf{x}} - \mathbf{y}). \end{aligned}$$

Multiplying the first inequality through by λ and the second through by $(1-\lambda)$ and adding them together we get

$$\lambda f(\hat{\mathbf{x}}) + (1-\lambda) f(\bar{\mathbf{x}}) \geq f(\mathbf{y}) + \nabla f(\mathbf{y})^\top(\lambda\hat{\mathbf{x}} + (1-\lambda)\bar{\mathbf{x}} - \mathbf{y}) = f(\mathbf{y}) = f(\lambda\hat{\mathbf{x}} + [1-\lambda]\bar{\mathbf{x}})$$

which is the definition we began with. Thus a function $f(\mathbf{x})$ is convex if and only if

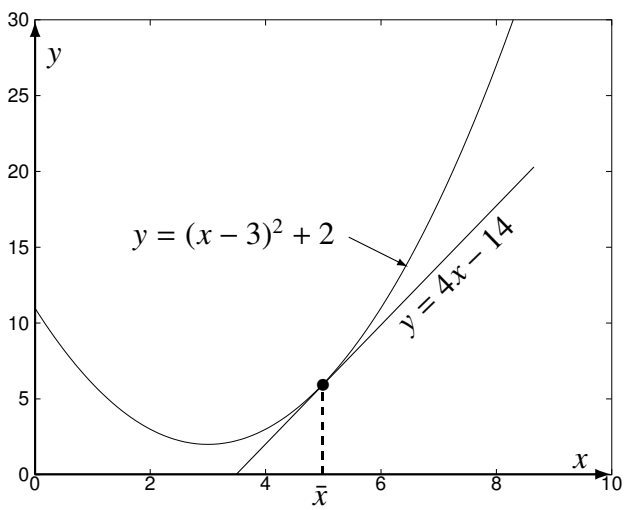
$$\boxed{\underset{\text{value of function}}{f(\mathbf{x})} \geq \underset{\text{height of tangent}}{f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x} - \bar{\mathbf{x}})} \quad \text{for all } \mathbf{x}, \bar{\mathbf{x}}}$$

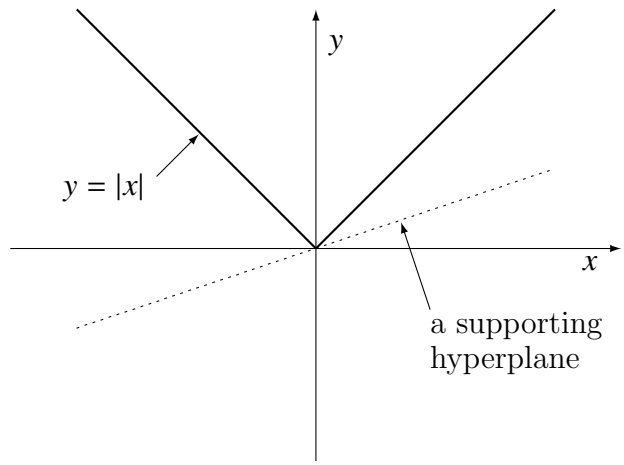
This **support inequality** plays an important role in the theory of nonlinear programming as another characterization of convex functions. If $f(\mathbf{x})$ is strictly convex then the support inequality holds strictly for all $\mathbf{x} \neq \bar{\mathbf{x}}$.

For the convex function $f(x) = (x-3)^2 + 2$ we find $\nabla f(\bar{x}) = 2\bar{x} - 6$, so the equation of a line tangent to the graph of the function at $\bar{x}$ is

$$y = (\bar{x} - 3)^2 + 2 + (2\bar{x} - 6)(x - \bar{x}).$$

For example, at $\bar{x} = 5$ the tangent line is $y = 4x - 14$ as shown on the left below.





Every hyperplane tangent to the graph of a convex function is a **supporting hyperplane** to the epigraph of the function. By using the support inequality it can also be shown (see Exercise 11.7.8) that if $f(\mathbf{x})$ is smooth then it is convex if and only if

$$[\nabla f(\mathbf{x}^2) - \nabla f(\mathbf{x}^1)]^\top (\mathbf{x}^2 - \mathbf{x}^1) \geq 0.$$

Every convex function is continuous on the interior of its domain [1, Theorem 3.1.3]. In the graph on the right, $y = |x|$ is convex so it is continuous on $\mathbb{R}^1$, but it is not differentiable at the origin. However, it still has supporting hyperplanes at that point (one is shown) for which $f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \boldsymbol{\xi}^\top(\mathbf{x} - \bar{\mathbf{x}})$. Each such vector $\boldsymbol{\xi}$ is called a **subgradient** of $f(\mathbf{x})$ [1, §3.2.3].

Most optimization algorithms can be proved to converge only if it is assumed that the objective and constraint functions of the nonlinear program are all convex. While some important applications yield such **convex programs**, many others unfortunately do not.

11.3 Global Minima

At a local minimum $\mathbf{x}^\star$, $\nabla f(\mathbf{x}^\star) = \mathbf{0}$ by the first-order necessary conditions of §9.3, so the supporting hyperplane at $\mathbf{x}^\star$ is horizontal. If $f(\mathbf{x})$ is a convex function then by the support inequality we have $f(\mathbf{x}) \geq f(\mathbf{x}^\star)$ for all $\mathbf{x}$, so $\mathbf{x}^\star$ is also a *global* minimum. If $f(\mathbf{x})$ is a strictly

convex function, then by the strict version of the support inequality $\mathbf{x}^\star$ is the unique global minimum. These results are summarized in the following theorems [1, Theorem 3.4.2].

Theorem: global minimizers

if $\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$
$f(\mathbf{x})$ is a convex function
then $\bar{\mathbf{x}}$ is a global minimum

Theorem: unique global minimizer

if $\nabla f(\bar{\mathbf{x}}) = \mathbf{0}$
$f(\mathbf{x})$ is a strictly convex function
then $\bar{\mathbf{x}}$ is the unique global minimum

In the graph of the convex function $f(x) = (x-3)^2+2$, the slope of a tangent line increases as x increases so

$$\frac{d^2f}{dx^2} \geq 0 \quad \text{for all } x.$$

In general, if $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^1$ has a positive semidefinite Hessian matrix then it is a convex function [1, Theorem 3.3.7], and if its Hessian matrix is positive definite then it is a strictly convex function. The first of these implications also works in the other direction, but the second does not; $f(x) = x^4$ is strictly convex, but $H(x) = \partial^2 f/\partial x^2 = 12x^2 = 0$ for $x = 0$ so it is only positive *semi*definite (we first encountered this counterexample in §10.7). Thus

$$\begin{aligned} \mathbf{H}(\mathbf{x}) \text{ is positive semidefinite for all } \mathbf{x} &\Leftrightarrow f(\mathbf{x}) \text{ is convex} \\ \mathbf{H}(\mathbf{x}) \text{ is positive definite for all } \mathbf{x} &\Rightarrow f(\mathbf{x}) \text{ is strictly convex} \\ f(\mathbf{x}) \text{ is strictly convex} &\not\Rightarrow \mathbf{H}(\mathbf{x}) \text{ is positive definite for all } \mathbf{x}. \end{aligned}$$

11.4 Testing Convexity Using Hessian Submatrices

In §11.1 we found that $f(x)$ is a convex function if epi(f) is a convex set; then $f(x)$ satisfies the defining inequality that requires every chord to be above or on the graph. In §11.2 we saw that $f(x)$ is a convex function if it satisfies either form of the support inequality. Each of these characterizations can sometimes be used to show that a given function is convex, but often it is easier to find out by checking the definiteness of the function's Hessian matrix. There are several ways to do that.

Recall from §10.7 that $\mathbf{H}$ is

$$\begin{aligned} \text{positive semidefinite} &\Leftrightarrow \mathbf{w}^\top\mathbf{H}\mathbf{w} \geq 0 \text{ for all } \mathbf{w} \\ \text{positive definite} &\Leftrightarrow \mathbf{w}^\top\mathbf{H}\mathbf{w} > 0 \text{ for all } \mathbf{w} \neq \mathbf{0}. \end{aligned}$$

It is also true that if the second partials that make up the Hessian are continuous then the matrix is symmetric, its eigenvalues are real, and $\mathbf{H}$ is

$$\begin{aligned} \text{positive semidefinite} &\Leftrightarrow \text{every eigenvalue is} \geq 0 \\ \text{positive definite} &\Leftrightarrow \text{every eigenvalue is} > 0. \end{aligned}$$

A third test that is usually easier to perform by hand is based on the determinants of submatrices [3, §9.5] [110, §2.2]. A **principal submatrix** of an $n \times n$ matrix is obtained by removing $r \in [0, n-1]$ of the rows along with the columns having the same indices as those rows. Notice that a principal submatrix need not be comprised of elements from *adjacent* rows and columns of the original matrix. A **leading principal submatrix** is obtained by removing the last r rows and columns of the original matrix, so that the $(1,1)$ element of the submatrix is the $(1,1)$ element of the original matrix and the submatrix *is* comprised of elements from adjacent rows and columns of the original matrix. The original matrix is itself a principal submatrix and a leading principal submatrix (corresponding to $r = 0$). A **minor** of a square matrix is the determinant of a square submatrix. By computing minors we can make use of the fact that if $\mathbf{H}$ is symmetric then it is

$$\begin{aligned} \text{positive semidefinite} &\Leftrightarrow \text{all of its principal minors are} \geq 0 \\ \text{positive definite} &\Leftrightarrow \text{all of its leading principal minors are} > 0. \end{aligned}$$

Consider the example of determining whether the function $f(\mathbf{x}) = 2x_1^4 + 3x_2^2 + x_3^2 - 2x_1 - 2x_2x_3$ is convex. Computing partial derivatives we find that

$$\frac{\partial f}{\partial x_1} = 8x_1^3 - 2 \qquad \frac{\partial f}{\partial x_2} = 6x_2 - 2x_3 \qquad \frac{\partial f}{\partial x_3} = 2x_3 - 2x_2$$

so

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} 24x_1^2 & 0 & 0 \\ 0 & 6 & -2 \\ 0 & -2 & 2 \end{bmatrix}.$$

If the leading principal minors are all positive then $\mathbf{H}$ is positive definite and also positive semidefinite; if any of them are negative then $\mathbf{H}$ is certainly *not* positive semidefinite. Thus it makes sense to check those minors first.

To avoid confusion with the absolute value function, I will use the MATLAB notation `det()` to denote the determinant of a scalar.

$$\texttt{det}(24x_1^2) = 24x_1^2 \geq 0 \qquad \begin{vmatrix} 24x_1^2 & 0 \\ 0 & 6 \end{vmatrix} = 144x_1^2 \geq 0 \qquad \begin{vmatrix} 24x_1^2 & 0 & 0 \\ 0 & 6 & -2 \\ 0 & -2 & 2 \end{vmatrix} = 192x_1^2 \geq 0$$

The leading principal minors are all nonnegative, so to decide about $\mathbf{H}$ we must compute the other principal minors, of which there are four. The rightmost principal minor listed on the next page is made up of the corner elements (1,1), (1,3), (3,1), and (3,3) of the full matrix.

$$\texttt{det}(6) = 6 > 0 \qquad \texttt{det}(2) = 2 > 0 \qquad \begin{vmatrix} 6 & -2 \\ -2 & 2 \end{vmatrix} = 8 > 0 \qquad \begin{vmatrix} 24x_1^2 & 0 \\ 0 & 2 \end{vmatrix} = 48x_1^2 \geq 0$$

All of the principal minors are nonnegative, so $\mathbf{H}(\mathbf{x})$ is positive semidefinite for all $\mathbf{x}$, and $f(\mathbf{x})$ is convex but not strictly convex.

11.4.1 Finding the Determinant of a Matrix

To compute the determinants in the example above I used an algorithm called **expansion by minors**. The smallest possible submatrix is a single element, so the smallest minor is the determinant of a scalar and that is just the scalar.

```
octave:1> d=det(5)
d =  5
octave:2> d=det(-5)
d = -5
```

The determinant of a 2×2 matrix $\mathbf{A}$ is $a_{11}a_{22} - a_{21}a_{12}$:

$$|\mathbf{A}| = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = 1 \times 4 - 3 \times 2 = -2$$

```
octave:3> A=[1,2;3,4];
octave:4> d=det(A)
d = -2
```

The determinant of a 3×3 matrix $\mathbf{B}$ can be found by evaluating three 2×2 minors.

$$|\mathbf{B}| = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} = 1 \times \begin{vmatrix} 5 & 6 \\ 8 & 9 \end{vmatrix} - 4 \times \begin{vmatrix} 2 & 3 \\ 8 & 9 \end{vmatrix} + 7 \times \begin{vmatrix} 2 & 3 \\ 5 & 6 \end{vmatrix} = 1(-3) - 4(-6) + 7(-3) = 0$$

```
octave:4> B=[1,2,3;4,5,6;7,8,9];
octave:5> det(B)
ans = -1.3326e-15
```

Here I formed submatrices by deleting the first column and each row in turn, multiplied each minor by the first-column element in the deleted row, and alternately added and subtracted the resulting terms. Now each of the 2×2 determinants can be found as described above.

This approach can be used to reduce the problem of finding an $n \times n$ determinant to the problem of finding n determinants each $n - 1$ elements square. Here is a 4×4 example.

$$|\mathbf{C}| = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{vmatrix} = 1 \times \begin{vmatrix} 6 & 7 & 8 \\ 10 & 11 & 12 \\ 14 & 15 & 16 \end{vmatrix} - 5 \times \begin{vmatrix} 2 & 3 & 4 \\ 10 & 11 & 12 \\ 14 & 15 & 16 \end{vmatrix} + 9 \times \begin{vmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \\ 14 & 15 & 16 \end{vmatrix} - 13 \times \begin{vmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \\ 10 & 11 & 12 \end{vmatrix}$$

Notice the alternation of + and − signs in the combination of the 3×3 determinants.

Expansion by minors is a practical way to find the determinants of small matrices, and it can be used even if the matrix elements are functions of $\mathbf{x}$ as in our first example. But

the bookkeeping and arithmetic grow combinatorially as n increases. Computer programs such as MATLAB compute the determinant of a matrix $\mathbf{M}$ by finding its lower and upper triangular factors $\mathbf{L}$ and $\mathbf{U}$ and then using

$$|\mathbf{M}| = |\mathbf{LU}| = |\mathbf{L}||\mathbf{U}|.$$

The determinant of a triangular matrix is just the product of its diagonals, so the work required by this approach is mainly in the matrix factorization and therefore grows as n^3 rather than $n!$ [67, p96]. Whether the matrix $\mathbf{M}$ is symmetric does not matter for expansion by minors but does determine what algorithm is used to compute the factors $\mathbf{L}$ and $\mathbf{U}$.

11.4.2 Finding the Principal Minors of a Matrix

We saw above that we can determine whether a matrix is positive *definite* by evaluating only its *leading* principal minors, of which there are just n. The `lpm.m` routine on the left performs this calculation for a matrix, assumed symmetric, whose entries are numbers.

```
function v=lpm(M)
% find the leading principal minors of a matrix

  n=size(M,1);
  for r=0:n-1
    v(n-r)=det(M(1:n-r,1:n-r));
  end
end
```

```
octave:1> M=[10,5,0;5,15,5;0,5,2]
M =

   10    5    0
    5   15    5
    0    5    2

octave:2> lpm(M)
ans =

   1.0000e+01   1.2500e+02   1.3553e-17

octave:3> quit
```

The Octave session on the right shows the leading principal minors found by `lpm` for the matrix `M`. The determinant of the whole matrix, reported here as a tiny number, is actually zero, so this test does not resolve the definiteness of `M`.

To test *all* of the principal minors of an $n \times n$ matrix we must generate the submatrices obtained by removing all possible combinations of $r \in [0, n-1]$ rows and the same columns. Our example matrix `M` with $n = 3$ has these $2^n - 1 = 7$ principal minors.

$$\begin{vmatrix} 10 & 5 & 0 \\ 5 & 15 & 5 \\ 0 & 5 & 2 \end{vmatrix} \quad \begin{vmatrix} 10 & 5 \\ 5 & 15 \end{vmatrix} \quad \begin{vmatrix} 15 & 5 \\ 5 & 2 \end{vmatrix} \quad \begin{vmatrix} 10 & 0 \\ 0 & 2 \end{vmatrix} \qquad \texttt{det(10)} \quad \texttt{det(15)} \quad \texttt{det(2)}$$

Suppose we denote a matrix row (and matrix column with the same index) that is retained in a submatrix by marking it with a one, and a row (and column with the same index) that is removed by marking it with a zero. Using this scheme the submatrices in the minors above could be specified by the following 3-bit strings.

`111` `110` `011` `101` `100` `010` `001`

These are all of the possible 3-bit binary numbers except `000`, or the numbers `i=1:(2^n)-1`. The `apm.m` routine below generates the $2^n - 1$ bit strings representing the principal submatrices of a given $n \times n$ matrix `M`, deletes the appropriate rows and columns to generate each submatrix, and finds the determinant of each submatrix.

```
 1 function v=apm(M)
 2 % find ALL principal minors of a matrix
 3
 4 % consider each principal submatrix
 5   n=size(M,1);
 6   for i=1:(2^n)-1
 7       A=M;
 8       s=n;
 9       j=uint32(i);
10
11 %     delete the rows and columns specified by the bit pattern
12       for k=1:n
13           p=bitget(j,1);
14           if(p == 0)
15 %            decrement the size of the submatrix
16              s=s-1;
17
18 %            delete row n-k+1 by copying rows up
19              for r=n-k+1:s
20                  A(r,[1:n])=A(r+1,[1:n]);
21              end
22 %            and zeroing out the bottom nonzero row
23              A(s+1,[1:n])=0;
24
25 %            delete column n-k+1 by copying columns left
26              for c=n-k+1:s
27                  A([1:n],c)=A([1:n],c+1);
28              end
29 %            and zeroing out the rightmost nonzero column
30              A([1:n],s+1)=0;
31           end
32           j=bitshift(j,-1);
33       end
34
35 %     the minor is the determinant of the submatrix
36       v(i)=det(A([1:s],[1:s]));
37   end
38 end
```

The built-in function `uint32` [9] converts its argument to an unsigned 32-bit integer; `bitget` [13] returns the value (0 or 1) of the rightmost bit of its 32-bit unsigned integer argument; and `bitshift` [32] shifts its argument bitstring (here to the right by 1 bit). By using these functions the routine examines the bits of the bit string that represents each submatrix. If a row (and the corresponding column) are not included in the submatrix, it copies rows below that row up [18-21] and columns to the right of that column left [25-28] overwriting and thus removing the omitted row and column. Each such copying leaves a duplicate row at the bottom [23] or column at the right [30] which is then set to zero.

The Octave session on the next page shows the principal minors for our $n = 3$ example, which are found and reported by `apm.m` in the order `001`, `010`, `011`, `100`, `101`, `111`.


```
octave:1> M=[10,5,0;5,15,5;0,5,2]
M =

   10    5    0
    5   15    5
    0    5    2

octave:2> apm(M)
ans =

   2.0000e+00   1.5000e+01   5.0000e+00   1.0000e+01   2.0000e+01   1.2500e+02   1.3553e-17

octave:3> quit
```

Thus, for example, the third value reported is the determinant of the submatrix composed of rows and columns 2 and 3 of `M`,

$$\begin{vmatrix} 15 & 5 \\ 5 & 2 \end{vmatrix} = 30 - 25 = 5.$$

Our scheme for representing which rows and columns are included in a given submatrix works only for n up to 32, at which size there are $2^{32} - 1 \approx 4.3 \times 10^9$ principal submatrices to check. Evaluating that number of determinants (many of them large) and reporting their values would not be very practical. While checking minors is easier than computing eigenvalues if the matrix is small, the opposite is true if the matrix is large, even though finding the eigenvalues of a large matrix also takes a lot of work.

11.5 Testing Convexity Using Hessian Eigenvalues

Recall from §11.4 that a symmetric matrix $\mathbf{H}$ is positive semidefinite if and only if its eigenvalues are all nonnegative, and positive definite if and only if they are strictly positive. The eigenvalues $\lambda_1 \ldots \lambda_n$ of a square matrix $\mathbf{A}$ are [147, §5] the solutions of its **characteristic equation**

$$|\mathbf{A} - \lambda\mathbf{I}| = 0.$$

The matrix on the left below has the characteristic equation on the right. The roots of the quadratic are $\lambda_1 \approx 6.8$ and $\lambda_2 \approx 1.2$, so this matrix is positive definite.

$$\mathbf{A} = \begin{bmatrix} 6 & -2 \\ -2 & 2 \end{bmatrix} \qquad \begin{vmatrix} 6-\lambda & -2 \\ -2 & 2-\lambda \end{vmatrix} = (6-\lambda)(2-\lambda) - 4 = 0$$

$$\begin{aligned} 12 - 8\lambda + \lambda^2 - 4 &= 0 \\ \lambda^2 - 8\lambda + 8 &= 0 \\ \lambda &= \tfrac{1}{2}(8 \pm 4\sqrt{2}) \end{aligned}$$

To solve the characteristic equation of a matrix that is $n \times n$ we need to find the roots of a polynomial of order n, and that cannot in general be done in closed form for $n > 4$.

Unfortunately, finding all of the zeros of a high-order polynomial numerically by naïvely using an algorithm such as bisection or Newton's method is notoriously difficult [60, p169].

Finding the eigenvalues of even a small matrix can be awkward if its elements are not numbers. If $f(\mathbf{x})$ is quadratic then its Hessian is constant, but in general $\mathbf{H}(\mathbf{x})$ really does depend on $\mathbf{x}$. The function on the left below is a **posynomial** [3, §9.8] and therefore convex for $\mathbf{x} > \mathbf{0}$, but the characteristic equation of its Hessian, given on the right, is unwieldy.

$$f(\mathbf{x}) = x_1^{-1} x_2^{-\frac{1}{2}} \qquad \begin{vmatrix} 2x_1^{-3}x_2^{-\frac{1}{2}} - \lambda & \frac{1}{2}x_1^{-2}x_2^{-\frac{3}{2}} \\ \frac{1}{2}x_1^{-2}x_2^{-\frac{3}{2}} & \frac{3}{4}x_2^{-\frac{5}{2}}x_1^{-1} - \lambda \end{vmatrix} = 0$$

If we found expressions for $\lambda_1(\mathbf{x})$ and $\lambda_2(\mathbf{x})$, it would be necessary to show that they are nonnegative for all $\mathbf{x} > \mathbf{0}$ in order to prove that $f(\mathbf{x})$ is convex there, a feat of algebra worthy of Maple. Of course the objective of a nonlinear program can have a Hessian that is both large *and* comprised of algebraic expressions.

It should be clear from this discussion that using eigenvalues to test the convexity of a function often calls for a certain amount of finesse. Fortunately there are some methods that can be used to investigate the definiteness of large matrices whether they contain numbers or formulas.

11.5.1 When the Hessian is Numbers

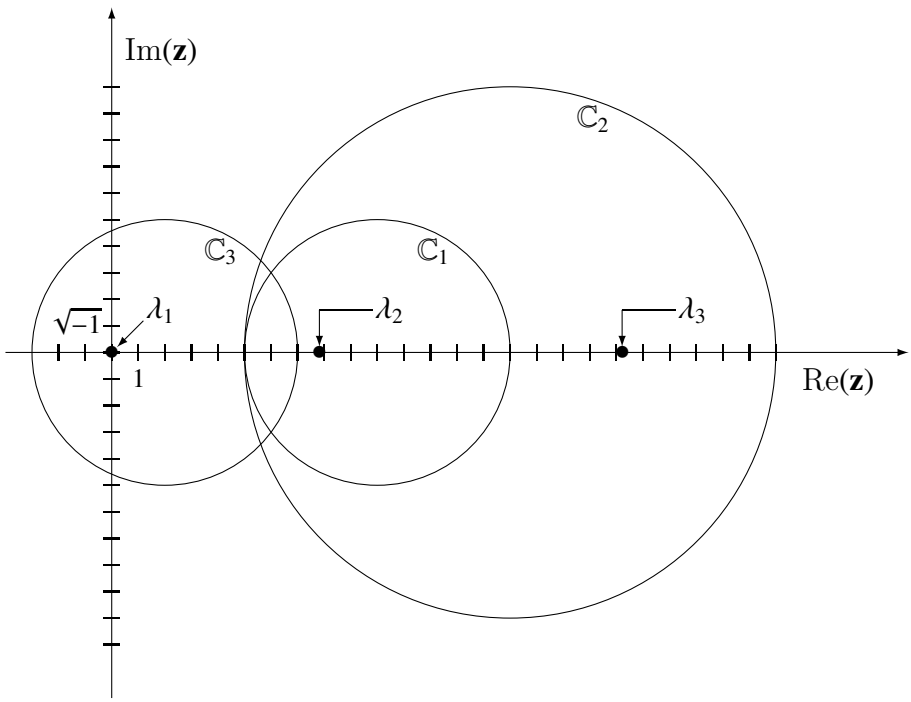
If $\mathbf{H}$ has elements that are numbers, a practical way to find its eigenvalues is with a numerical method that is custom-made for the task. MATLAB, for example, uses Hessenberg and Shur decompositions [150, §25] that avoid the characteristic equation altogether.

```
octave:1> M=[10,5,0;5,15,5;0,5,2]
M =

   10    5    0
    5   15    5
    0    5    2

octave:2> lambda=eig(M)
lambda =

   5.7988e-20
   7.8211e+00
   1.9179e+01
```

This is the same matrix we studied in §11.4.2, and here we find once again that it is positive semidefinite. Two of the eigenvalues are positive and the third is, within roundoff error, zero.

When we decided on the definiteness of `M` just now we paid attention only to the *signs* of the eigenvalues, not to their *values*. The **Gerschgorin circle theorem** [147, p289]

states that every eigenvalue of $\mathbf{H}$ lies in a union of circles $\mathbb{C}_1 \ldots \mathbb{C}_n$ in the complex plane (a nonsymmetric matrix can have complex eigenvalues). Circle $\mathbb{C}_i$ is centered at $\mathbf{z} = h_{ii} + 0\sqrt{-1}$ and its radius is the sum of the absolute values of the other elements in row i. The Gerschgorin circles for the above matrix `M` are shown below. $\mathbb{C}_1$ is centered at 10 with a radius of 5, $\mathbb{C}_2$ is centered at 15 with a radius of 10, and is $\mathbb{C}_3$ centered at 2 with a radius of 5. The eigenvalues reported by MATLAB, $\lambda_1 \approx 0$, $\lambda_2 \approx 7.82$, and $\lambda_3 \approx 19.2$, are marked with • dots and can be seen to lie along the real axis within the union of the Gerschgorin circles.



If each row of $\mathbf{H}$ has $h_{ii} > \sum_{j \neq i} |h_{ij}|$, so that the matrix is **diagonally dominant**, then every eigenvalue is positive and $\mathbf{H}$ is positive definite; if each row of $\mathbf{H}$ has $h_{ii} < \sum_{j \neq i} |h_{ij}|$, then all of the eigenvalues must be negative and $\mathbf{H}$ is surely *not* positive definite. In these cases the definiteness of $\mathbf{H}$ can be determined simply by checking for diagonal dominance. Diagonal dominance requires $h_{ii} > 0$ so it makes sense to check that condition before bothering to add up the absolute values of the off-diagonal elements.

If in some row h_{ii} is *equal* to the sum of the absolute values of the off-diagonal elements, then one of the circles is tangent to the imaginary axis and one of the eigenvalues *might* be zero. If the circles lie otherwise in the right half-plane the matrix is positive semidefinite and might be positive definite; if the circles lie otherwise in the left half-plane the matrix might be positive semidefinite but is certainly not positive definite.

If a circle overlaps the imaginary axis, as in the picture above, then the Gerschgorin test is equivocal so if we want to use eigenvalues we can't avoid computing them.

11.5.2 When the Hessian is Formulas

If the elements of the Hessian matrix are functions of $\mathbf{x}$ rather than numbers, it might still be possible to determine the definiteness of $\mathbf{H}$ by computing eigenvalues even if the characteristic equation $|\mathbf{H} - \lambda\mathbf{I}| = 0$ can't be solved analytically for $\lambda(\mathbf{x})$.

The `convcheck.m` routine listed below selects points at random within the variable bounds 8-11 as in pure random search (see `prs.m` in §9.1). At each random point the routine 12 invokes `hsn` to compute the Hessian matrix there and 13 finds its eigenvalues. If an eigenvalue is 15 numerically zero the return parameter `flag` is 16 set to zero and the checking of the eigenvalues continues. If an eigenvalue is 19 numerically negative `flag` is 20 set to `-1` and there is no need to check further. On return `flag=1` 6 if no point was found where the Hessian was not positive definite, `flag=0` if the Hessian was positive semidefinite at `xbad`, and `flag=-1` if the Hessian was not even positive semidefinite at `xbad`.

```
 1 % convcheck.m: search for a point where H(x) is not positive definite
 2 function [flag,xbad]=convcheck(n,xl,xh,hsn)
 3
 4 x=zeros(n,1);                          % make x a column vector
 5 xbad=x;                                % return xbad=0 if none found
 6 flag=+1;                               % assume positive definite
 7 for k=1:10^(n+1)                       % inspect many points
 8     u=rand(n,1);                       % generate a random n-vector
 9     for j=1:n                          % select
10         x(j)=xl(j)+u(j)*(xh(j)-xl(j)); % a random point
11     end                                % within the bounds
12     H=hsn(x);                          % find the Hessian there
13     ev=eig(H);                         % find its eigenvalues
14     for j=1:n                          % check them all
15         if(abs(ev(j)) < 1e-8)          % if small assume zero
16           flag=0;                      % which makes H psd
17           xbad=x;                      % at this x
18         end
19         if(ev(j) < 1e-8)               % if negative
20           flag=-1;                     % that makes H not psd
21           xbad=x;                      % at this x
22           return                       % and we are done
23         end
24     end
25 end
```

I tested the routine on the Hessian of the posynomial function we encountered in §11.5.0, with the following result.

```
octave:1> flag=convcheck(2,[0;0],[10;10],@gph)
flag =  1
octave:2> quit
```

```
function h=gph(x)
  h=zeros(2,2);
  h(1,1)=2*x(1)^(-3)*x(2)^(-1/2);
  h(1,2)=(1/2)*x(1)^(-2)*x(2)^(-3/2);
  h(2,1)=h(1,2);
  h(2,2)=(3/4)*x(2)^(-5/2)*x(1)^(-1);
end
```

Finding an eigenvalue that is negative proves that $\mathbf{H}$ is not positive semidefinite. Failing to find an eigenvalue that is zero or negative, while short of proof that $\mathbf{H}$ is positive definite, suggests that it is at least positive semidefinite.

11.6 Generalizations of Convexity

In the theory of nonlinear programming it is sometimes useful to consider functions that are almost but not quite convex. An elaborate taxonomy [1, p144] has been developed to distinguish between the strictly convex and convex functions we have studied so far, and those that are nonconvex in various ways. Here I will mention only two of the categories. A **quasiconvex function** satisfies the inequality

$$f(\lambda \mathbf{x}^1 + [1-\lambda]\mathbf{x}^2) \le \max\left\{f(\mathbf{x}^1), f(\mathbf{x}^2)\right\} \quad \text{for all } \mathbf{x}^1, \mathbf{x}^2, \text{ and } \lambda \in [0,1]$$

and has the interesting property that all of its level sets (see Exercise 11.7.3) are convex sets. A **pseudoconvex function** is defined by the property, also interesting, that

$$\nabla f(\mathbf{x}^1)^\top(\mathbf{x}^2 - \mathbf{x}^1) \ge 0 \Rightarrow f(\mathbf{x}^1) \ge f(\mathbf{x}^2).$$

Some authors [2, p787] also distinguish **strongly convex** functions, which satisfy

$$f(\mathbf{x}) \ge f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x} - \bar{\mathbf{x}}) + \tfrac{k}{2}\|\mathbf{x} - \bar{\mathbf{x}}\|^2 \quad \text{for all } \mathbf{x}, \bar{\mathbf{x}} \quad \text{and some } k > 0$$

and are thus in a sense *more* convex than those that satisfy the ordinary support inequality. You should be aware of this cottage industry of variations on the idea of a convex function, but we will have scant use for them. The focus of this text is on algorithms, and most nonconvex functions that are encountered in practice aren't quasiconvex or pseudoconvex either.

A generalization that we *will* use later in the book is the idea of local convexity. Throughout this Chapter we have treated convexity as a *global* property that a function can have, but in the discussion of methods it is often useful to describe what happens near a local minimizing point. A function is **locally convex** if it satisfies the definition of a convex function, or the support inequality, within some epsilon-neighborhood of a given point. If a locally convex function is smooth, its Hessian will be positive semidefinite at points within that neighborhood but perhaps not elsewhere.

11.7 Exercises

11.7.1 [E] In solving an unconstrained nonlinear program, why do we care whether the objective function is convex?

11.7.2 [E] What is the *epigraph* of a function, and what property must it have for the function to be convex? Does the epigraph have any special properties if the function is strictly convex? If so, draw a picture to illustrate your answer.

11.7.3 [H] The set $\mathbb{S}(\alpha) = \{\mathbf{x} \mid f(\mathbf{x}) \leq \alpha\}$, where α is a real number, is called the α **level set** of $f(\mathbf{x})$. (a) Use the definition of convexity to prove that if $f(\mathbf{x})$ is a convex function then $\mathbb{S}(\alpha)$ is a convex set for all values of α. (b) If $\mathbb{S}(\alpha)$ is a convex set for all values of α, is $f(\mathbf{x})$ necessarily a convex function? If not, sketch the graph of a counterexample. (c) If $f(\mathbf{x})$ is a nonconvex function, are all of its level sets necessarily nonconvex? If not, sketch a counterexample. (c) How are a function's level sets related to its epigraph?

11.7.4 [E] In §11.1 we derived an inequality that we take as the definition of a convex function. (a) Write it down from memory. (b) Give a graphical interpretation. (c) Explain how the definition changes to describe a function that is strictly convex.

11.7.5 [E] In §11.1, the convex function f has a unique minimum while the nonconvex function g has multiple minima. Does a convex function always have a unique minimum? If not, provide a counterexample. Does a nonconvex function always have multiple minima? If not, provide a counterexample.

11.7.6 [H] Use the definition of a convex function to prove that a linear function is both convex and concave.

11.7.7 [E] Write down the support inequality of §11.2 from memory, and give a graphical interpretation. Explain how it changes to describe a function that is strictly convex.

11.7.8 [H] Prove that if $f(\mathbf{x})$ is smooth then it is convex if and only if

$$[\nabla f(\mathbf{x}^2) - \nabla f(\mathbf{x}^1)]^\top(\mathbf{x}^2 - \mathbf{x}^1) \geq 0.$$

Hint: to show $\Rightarrow$ use the support inequality twice and add; to show $\Leftarrow$ use the mean value theorem, $f(\mathbf{x}^2) - f(\mathbf{x}^1) = \nabla f(\mathbf{x})^\top(\mathbf{x}^2 - \mathbf{x}^1)$ where $\mathbf{x} = \lambda\mathbf{x}^1 + (1-\lambda)\mathbf{x}^2$ for some $\lambda \in [0, 1]$.

11.7.9 [E] What is a *supporting hyperplane*? If a convex function is not differentiable at $\bar{\mathbf{x}}$, can it have a supporting hyperplane there? Explain.

11.7.10 [H] A convex function is continuous on the interior of its domain. (a) Give an example of a convex function that is discontinuous at a boundary of its domain. (b) Using a picture, show how a function having a jump discontinuity is nonconvex. (c) Using a picture, show how a function having a point discontinuity is nonconvex.

11.7.11 [E] What is a *subgradient*? What is the subgradient of a smooth convex function?

11.7.12 [H] In §11.2 the function $y = |x|$ is graphed to illustrate that it has no derivative at $x = 0$, and to show one of its supporting hyperplanes at that point. (a) Does this function have a subgradient at $x = 2$? If not explain why not; if so give the equation of its supporting hyperplane there. (b) What subgradients does the function have at $x = 0$? Write down an algebraic description of the set of subgradients, and show on the graph the cone containing the gradient vectors of all the hyperplanes in that set. (c) If a convex function $f(\mathbf{x})$ is not differentiable at $\bar{\mathbf{x}}$, can the cone of subgradients at that point ever include vectors that are not in the epigraph of the function?

11.7.13 [E] What is a *convex program*?

11.7.14 [E] Use the strict version of the support inequality to prove the unique global minimizer theorem of §11.3.

11.7.15 [E] True or false? (a) If $f(\mathbf{x})$ is convex then its Hessian matrix $\mathbf{H}(\mathbf{x})$ is positive semidefinite for all $\mathbf{x}$. (b) If $\mathbf{H}(\mathbf{x})$ is positive definite for all $\mathbf{x}$ then $f(\mathbf{x})$ is convex. (c) If $f(\mathbf{x})$ is strictly convex then $f(\mathbf{x})$ is convex.

11.7.16 [E] Give an example of a strictly convex function whose Hessian matrix is not everywhere positive definite. Give an example of a strictly convex function whose Hessian matrix *is* everywhere positive definite.

11.7.17 [E] List all of the ways mentioned in this Chapter for determining whether a given function $f(\mathbf{x})$ is convex.

11.7.18 [H] The definition of positive definiteness given in §10.7 assumes nothing about the symmetry of the matrix, but the principal-minor test described in §11.4 is meaningless if the matrix is nonsymmetric. (a) Use the definition to show that

$$\mathbf{A} = \begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix}$$

is positive definite. (b) Prove that the matrix $\mathbf{M}+\mathbf{M}^\top$ is symmetric even if $\mathbf{M}$ is not. (c) Prove that $\mathbf{M}$ is positive definite if and only if $\mathbf{M}+\mathbf{M}^\top$ is positive definite. (d) Devise a method that uses this fact along with the principal-minor test to establish the definiteness of a nonsymmetric matrix. Use your method and the `apm.m` routine to confirm that $\mathbf{A}$ is positive definite. (e) Use MATLAB to find the real eigenvalues of $\mathbf{A} + \mathbf{A}^\top$, and conclude from them that $\mathbf{A}$ is positive definite. (f) If $\mathbf{M}$ is nonsymmetric we can still use the Gerschgorin circle theorem because [147, Exercise 6.2.8a] $\mathbf{M}$ is positive definite if and only if the real parts of its complex eigenvalues are positive. Use this approach to show that $\mathbf{A}$ is positive definite.

11.7.19 [P] A numerical measure of the **asymmetry** of a matrix $\mathbf{A}$ is given by

$$\text{asym}(\mathbf{A}) = \|(\mathbf{A} + \mathbf{A}^\top)/2 - \mathbf{A}\|.$$

(a) Write a MATLAB routine `asym.m` to compute the asymmetry of a matrix using this formula. (b) Revise the `lpm.m` and `apm.m` routines of §11.4.2 to use `asym.m` and test whether `M` is symmetric as assumed.

11.7.20 [H] Write down a function of two variables whose Hessian matrix is *not* symmetric.

11.7.21 [H] The objective of the `garden` problem is $f(\mathbf{x}) = x_1 x_2$. Is this a convex function? Use techniques discussed in this Chapter to support your answer.

11.7.22 [H] The `rb` problem has objective $f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$. (a) Find the Hessian matrix $\mathbf{H}(\mathbf{x})$ of this function. (b) Determine the definiteness of $f(\mathbf{x})$ based on minors.

11.7.23 [H] Determine whether each of the following functions is or is not convex, and explain how you decided: (a) $f(x) = e^x$; (b) $f(\mathbf{x}) = e^{x_1 x_2}$; (c) $f(x) = -\ln(x)$; (d) $f(x) = 1/x,\ x > 0$; (e) $f(\mathbf{x}) = -2x_1 - 6x_2 + 2x_1^2 + 3x_2^2 - 4x_1x_2$; (f) $f(\mathbf{x}) = 2x_1^2 + x_1x_2 + x_2^2 + x_2x_3 + x_3^2 - 6x_1 - 7x_2 - 8x_3 + 9$.

11.7.24 [H] Prove that the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

is positive definite by each of the following methods. (a) Use the definition of a positive-definite matrix given in §10.7. (b) Solve $|\mathbf{A} - \lambda\mathbf{I}| = 0$ and use an argument based on the eigenvalues of $\mathbf{A}$. (c) Use the leading principal minors test of §11.4. (d) Use the Gerschgorin circle theorem.

11.7.25 [E] In the matrix below, all of the principal submatrices are boxed except one. What is it?

$$\begin{bmatrix} \boxed{1} & 2 & 3 \\ 4 & \boxed{5} & 6 \\ 7 & 8 & \boxed{9} \end{bmatrix}$$

11.7.26 [E] Consider the following symmetric matrix.

$$\mathbf{A} = \begin{bmatrix} 6 & 2 & 1 & -1 \\ 2 & 4 & 1 & 0 \\ 1 & 1 & 4 & -1 \\ -1 & 0 & -1 & 3 \end{bmatrix}$$

(a) Write down all of the principal submatrices, and find all of the principal minors. Compute the 1×1 and 2×2 minors by hand, but use MATLAB for the larger ones. (b) Check your calculations by using the `apm.m` routine of §11.4.2 to compute the minors. (c) Identify those principal submatrices that are *leading* principal submatrices. (d) Determine the definiteness of the matrix based on your calculations. (e) Is there an easier way to establish the definiteness of this particular matrix? If so, explain what it is.

11.7.27 [E] If expansion by minors is used to compute the determinant of an $n \times n$ matrix, how many 2×2 minors must be evaluated? Another method can be used to compute a determinant, in which the number of arithmetic operations grows only polynomially with the size of the matrix. What is it?

11.7.28 [H] Solve the characteristic equation of this matrix to find its eigenvalues λ_1 and λ_2 as functions of p, q, r, and s.

$$\mathbf{A} = \begin{bmatrix} p & q \\ r & s \end{bmatrix}$$

State conditions on p, q, r, and s to ensure that (a) the eigenvalues are a complex conjugate pair in which the imaginary parts are nonzero; (b) the eigenvalues are real and equal; (c) the

eigenvalues are real and distinct. (d) What must be true of p, q, r, and s in order for $\mathbf{A}$ to be positive definite by the eigenvalues test? Show that if those conditions are satisfied then the matrix is also positive definite by the determinants test.

11.7.29 [H] If a square symmetric matrix $\mathbf{A}$ is positive definite then its eigenvalues λ_i are all positive. Show that the eigenvalues of $\mathbf{A}^{-1}$ are $\mu_i = 1/\lambda_i$ and hence $\mathbf{A}^{-1}$ is also positive definite.

11.7.30 [E] Explain why it is hard to find the eigenvalues of a large matrix by solving its characteristic equation.

11.7.31 [P] In §11.5.0 I proposed testing the convexity of the posynomial $f(\mathbf{x}) = x_1^{-1}x_2^{-\frac{1}{2}}$ by finding the eigenvalues of its Hessian matrix. (a) Evaluate the determinant stated there and solve the resulting quadratic equation to obtain expressions for $\lambda_1(\mathbf{x})$ and $\lambda_2(\mathbf{x})$. (b) Show that for $\mathbf{x} > \mathbf{0}$ the eigenvalues are positive. (c) Write a MATLAB program to draw some contours of this function. You might find it helpful to use the `gridcntr.m` routine of §9.1.

11.7.32 [P] According to the Gerschgorin circle theorem, where in the complex plane must the eigenvalues of the following matrix lie?

$$\mathbf{A} = \begin{bmatrix} 0 & -2 & 1 & -1 \\ -1 & 5 & 2 & 0 \\ 1 & -1 & 2 & -3 \\ -1 & 0 & -1 & 1 \end{bmatrix}$$

Use MATLAB to find the eigenvalues, and confirm that they all lie in the union of the Gerschgorin circles. If a matrix is symmetric, where do its eigenvalues lie? For a matrix to be positive semidefinite, where must its eigenvalues lie?

11.7.33 [E] What can be deduced about the definiteness of a matrix if one or more of its Gerschgorin circles contains points on *both* sides of the imaginary axis?

11.7.34 [E] What is a *diagonally dominant* matrix? Is a positive definite matrix always diagonally dominant? If so, prove it; if not, provide a counterexample.

11.7.35 [P] The `convcheck.m` routine of §11.5.2 can be used to investigate the positive definiteness of a Hessian matrix $\mathbf{H}(\mathbf{x})$. (a) Can `convcheck.m` be used if $\mathbf{H}(\mathbf{x})$ is constant rather than varying with $\mathbf{x}$? Explain. (b) Use `convcheck.m` to assess the convexity of $f(\mathbf{x}) = e^{x_1x_2} + x_1x_2$. Are the results conclusive for this function? If not, explain why not. If so, support your claim by evaluating $\mathbf{H}(\mathbf{x})$ at one point and using MATLAB to compute its eigenvalues there.

11.7.36 [E] The `convcheck.m` routine of §11.5.2 examines an $n \times n$ Hessian at 10^{n+1} points. If the variable bounds are $[-\mathbf{1}, +\mathbf{1}]$, how far apart (in Euclidean norm) would the points be, as a function of n, if they were equally spaced?

11.7.37 [H] Quasiconvex and pseudoconvex functions are described in §11.6. (a) Are convex functions quasiconvex? Are they pseudoconvex? (b) Sketch the graph of a nonconvex quasiconvex function. (c) Sketch the graph of a nonconvex pseudoconvex function.

11.7.38 [H] Show that the level sets of a quasiconvex function are convex sets.

11.7.39 [H] Is the function $f(x) = e^x$ *strongly convex*?

11.7.40 [H] Find any intervals of x over which the function $g(x) = \frac{1}{100}(\frac{3}{2}x-6)^4 - \frac{2}{3}(\frac{3}{2}x-5)^2 + 25$ of §11.1 is locally convex.

11.7.41 [E] If $f(x)$ is a convex function, is its derivative $f'(x) = df/dx$ necessarily a convex function? If yes, prove it; if no, provide a counterexample.

11.7.42 [E] Once upon a time, in a certain university mathematics department, there were two professors who both studied optimization. One posted on her office door the slogan "Life is nice when things are linear." In response the other posted on his office door the slogan "Linearity is nice, but convexity is enough!" Were these people completely crazy? If not, how do you interpret the two slogans?

12

Line Search

In §10 we considered steepest descent, the simplest gradient-based member of a large class of optimization algorithms called **descent methods**. Descent methods work by finding a downhill direction, performing a univariate minimization of the objective function in that direction, and repeating the process until it generates a point $\mathbf{x}^\star$ from which no direction is downhill. The univariate minimization problem of finding

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}}\ f(\mathbf{x}^k + \alpha\mathbf{d}^k) \equiv \underset{\alpha}{\operatorname{argmin}}\ f(\alpha),$$

which must be solved at each iteration k of a descent method, is called a **line search**. This Chapter is about algorithms for searching a line in an arbitrary descent direction $\mathbf{d}$ that need not be the direction of steepest descent.

12.1 Exact and Approximate Line Searches

The `gns` problem we studied in §10.4 is simple enough that calculus can be used to derive an algebraic formula for $\alpha_k^\star(\mathbf{x}^k; \mathbf{d}^k)$. Using such a formula to find $\alpha_k^\star$ is called an **exact analytic line search**. Numerically finding an $\alpha_k^\star$ that minimizes $f(\alpha)$ to near machine precision, by using a method such as those discussed in this Chapter, is called an **exact numerical line search**. We will frequently use an exact line search of one kind or the other in our *study* of descent methods, just to make it easy to understand what is happening.

However, in the *use* of a descent method it is rarely possible to do an exact line search analytically and it is seldom desirable to do one numerically. It is only the final $\mathbf{d}^k$ that leads to $\mathbf{x}^\star$, so finding all of the $\mathbf{x}^k$ precisely is a waste of effort. It is necessary to find each $\alpha_k^\star$ accurate only to within some positive **line search tolerance** t, which is chosen just small enough that the descent method converges to $\mathbf{x}^\star$ within its tolerance ϵ. In the unusual situation when we need to find $\mathbf{x}^\star$ exactly, we can start with a loose line search tolerance and tighten it as we approach the optimal point (we will make use of this refinement in §12.4.2 and again in §13 and §21).

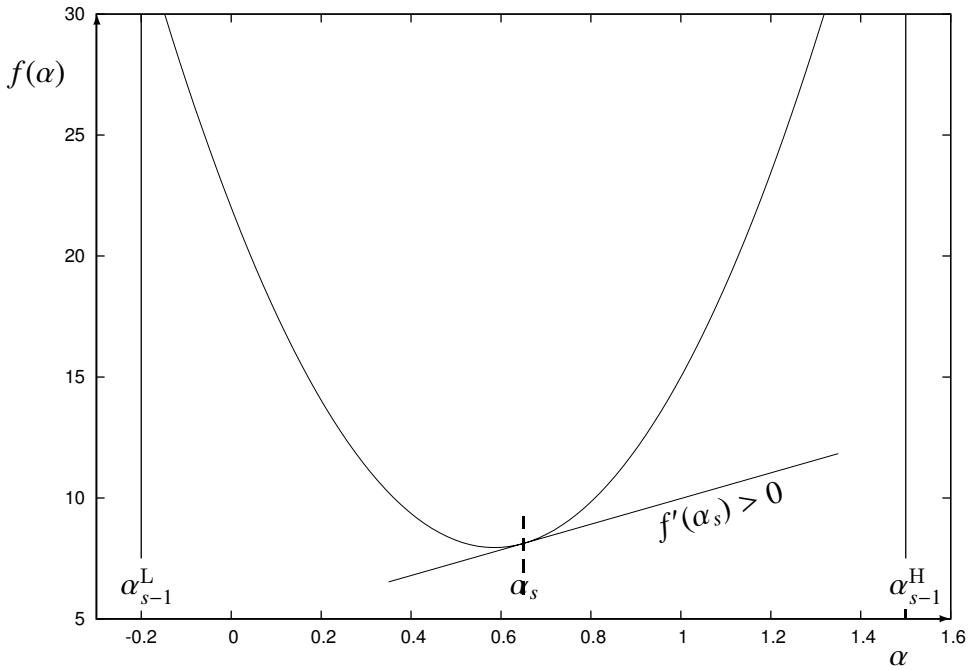
A numerical line search begins with an **interval of uncertainty** $[\alpha^{\mathrm{L}}, \alpha^{\mathrm{H}}]$, known to contain $\alpha^\star$, and its goal is to reduce that interval's width. The table on the next page describes several algorithms for reducing the interval of uncertainty. Methods that use only function values [1, §8.1] [155, §2], listed in the top part of the table, try to find an α where $f(\alpha)$ has its lowest value; those that use derivatives [1, §8.2], listed in the bottom part of the table, try to find an α where $f(\alpha)$ has zero slope.

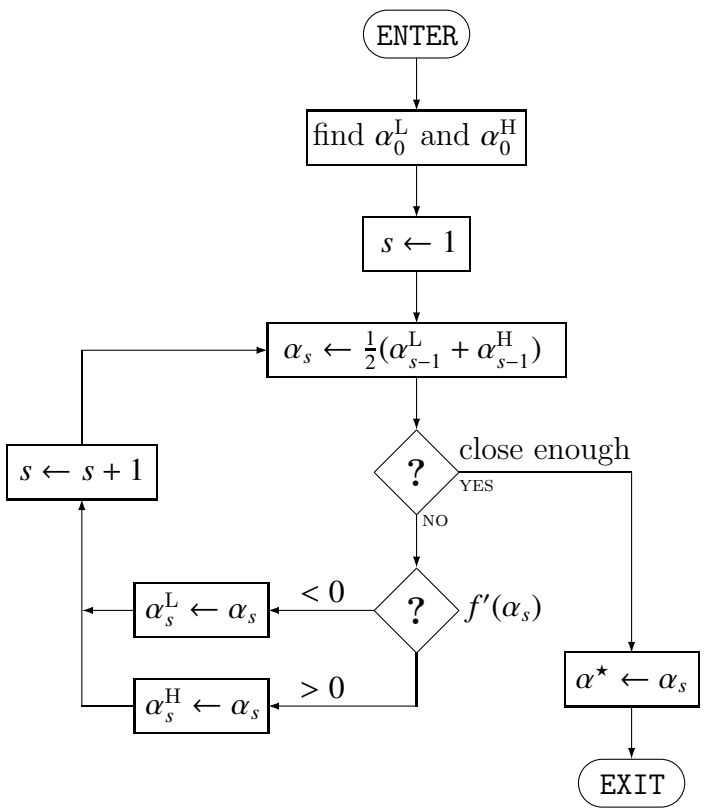
line search method	vague description
grid search	Find $f(\alpha)$ at several values of α evenly spaced in the interval of uncertainty, and pick the α yielding the lowest $f(\alpha)$.
dichotomous search	Repeatedly use finite differencing to bisect the remaining interval of uncertainty, discarding at each line search iteration the interval half that does not contain $\alpha^\star$.
golden section	Choose two values of α in a clever way and use the values of $f(\alpha)$ to reduce the interval of uncertainty; thereafter at each line search iteration choose one new value of α in a way that lets the process be repeated (see Exercise 12.5.4).
Fibonacci	Choose two values of α in an even more clever way and use the values of $f(\alpha)$ to reduce the interval of uncertainty; then for each of a fixed number of iterations choose one new value of α in a way that lets the interval of uncertainty be reduced further (see Exercise 12.5.5).
quadratic interpolation	Choose three values of α in the interval of uncertainty, interpolate a quadratic through the points $(\alpha, f(\alpha))$, and minimize the quadratic analytically [2, §C.2] [107, §7.2].
bisection	Use bisection to approximately solve $df/d\alpha = 0$.
Newton's method	Use Newton's method to approximately solve $df/d\alpha = 0$.
cubic interpolation	Fit a clamped cubic spline [20, §3.6] to $f(\alpha)$ and minimize that.

Derivative-based methods require that $f(\alpha)$ be differentiable (which is often not the case for type-2 problems) or that $df/d\alpha$ be approximated by finite differencing, and they might find a stationary point of $f(\alpha)$ that is not a minimum. Methods that use only function values typically require lots of them and are therefore comparatively slow; thus in line searching we encounter the usual tradeoff between robustness and speed. Of these algorithms the most mathematically intriguing are the golden section search and the Fibonacci search, but the idea that will prove most fruitful in our study of descent methods is the simplest derivative-based one, bisection [3, p306-307].

12.2 Bisection

The **bisection line search** finds a zero of $f'(\alpha)$ by using the familiar algorithm for finding a zero of any scalar function of one variable (see §28.3.1). To see how it works consider the example in the graph on the next page. If the current interval of uncertainty is $[\alpha^{\mathrm{L}}_{s-1}, \alpha^{\mathrm{H}}_{s-1}]$ we compute α_s as the midpoint of that interval and evaluate $df(\alpha)/d\alpha = f'(\alpha)$ there. If the slope is positive as shown then we assume the minimizing point is in the left half and shrink the interval by making $\alpha^{\mathrm{H}}_s = \alpha_s$ while keeping $\alpha^{\mathrm{L}}_s = \alpha^{\mathrm{L}}_{s-1}$. If the slope is negative we assume the minimizing point is in the right half and shrink the interval by making $\alpha^{\mathrm{L}}_s = \alpha_s$ while keeping $\alpha^{\mathrm{H}}_s = \alpha^{\mathrm{H}}_{s-1}$. Then we can repeat the process starting from the new interval $[\alpha^{\mathrm{L}}_s, \alpha^{\mathrm{H}}_s]$. This algorithm is formalized in the flowchart at the bottom of the page.





Line search iterations are indexed using s to distinguish them from iterations of the descent method, which are indexed using k. To perform one iteration of the descent method we do one line search that might require several line search iterations.

The first decision block in the flowchart is the convergence test. Often the stopping condition is chosen to be $|f'(\alpha_s)| < t$, so that t means how close to stationary we want $\alpha^\star$ to be, but some versions of the algorithm use $\alpha^{\mathrm{H}}_{s-1} - \alpha^{\mathrm{L}}_{s-1} < t$ instead or test both conditions. If $\alpha^{\mathrm{H}} = \alpha^{\mathrm{L}}$ further bisections would be pointless even if $|f'(\alpha)| > t$. In this example $f'(\alpha_s)$ is quite small (α_s is close to $\alpha^\star$) even though the interval of uncertainty is very big.

12.2.1 The Directional Derivative

How can we find $f'(\alpha)$, which is required for the bisection line search, when the overall optimization problem is instead defined in terms of $f(\mathbf{x})$ and its derivatives?

The graph on the next page shows one iteration of a descent method being used to minimize $f(\mathbf{x}) = (x_1-3)^2 + (x_2-4)^2 + 5$. In this picture it is easy to imagine a vertical pane of glass, bordered by $\mathbf{d}^k$ and the $f(\alpha)$ axis, slicing through the graph of the objective function. It is in this plane that the objective is $f(\alpha)$ and the line search takes place; this parabola is the same one graphed above. The line search yields the next iterate in the descent method, $\mathbf{x}^{k+1}$, at $\alpha = \alpha^\star$. (If in this example $\mathbf{d}^k$ were the direction of *steepest* descent then it would pass through $\mathbf{x}^\star$ and the line search would yield $\mathbf{x}^{k+1} = \mathbf{x}^\star$.)

The derivative of $f(\mathbf{x})$ in the plane of the cut, $f'(\alpha)$, is called the **directional derivative** of $f(\mathbf{x})$ at the point $\mathbf{x}^k + \alpha\mathbf{d}^k$, and we can find it using the definition of a derivative.

$$\begin{aligned} f'(\alpha) &= \lim_{h\to 0} \frac{f(\alpha+h) - f(\alpha)}{h} \\ &= \lim_{h\to 0} \frac{f(\mathbf{x}^k + (\alpha+h)\mathbf{d}) - f(\mathbf{x}^k + \alpha\mathbf{d})}{h} = \lim_{h\to 0} \frac{f(\mathbf{x}^k + \alpha\mathbf{d} + h\mathbf{d}) - f(\mathbf{x}^k + \alpha\mathbf{d})}{h} \end{aligned}$$

Expanding the first term in the numerator by Taylor's series,

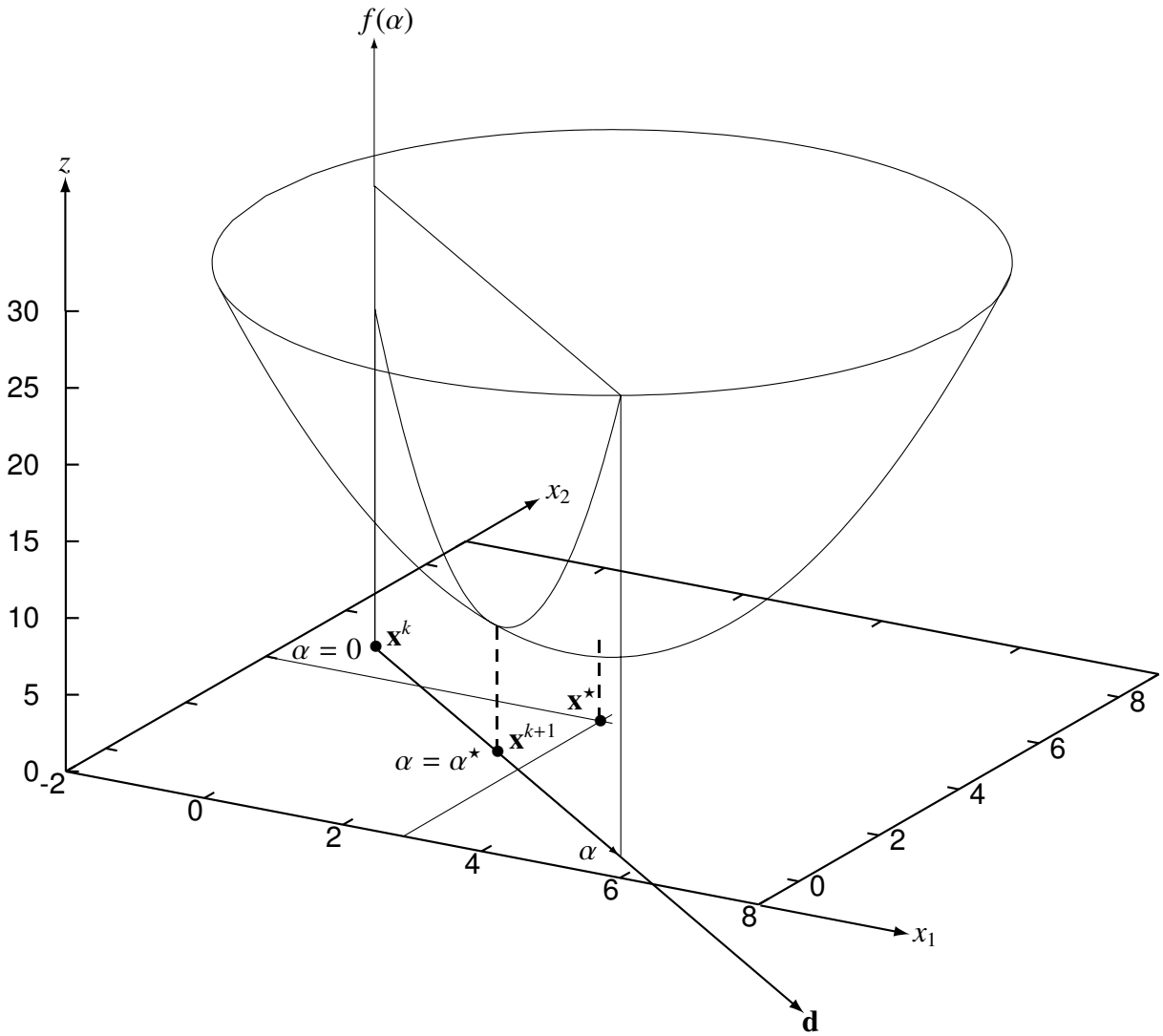
$$f(\mathbf{x}^k + \alpha\mathbf{d} + h\mathbf{d}) = f(\mathbf{x}^k + \alpha\mathbf{d}) + h\mathbf{d}^\top\nabla f(\mathbf{x}^k + \alpha\mathbf{d}) + \text{higher order terms.}$$

Then

$$\begin{aligned} f'(\alpha) &= \lim_{h\to 0}\left[\frac{f(\mathbf{x}^k + \alpha\mathbf{d}) + h\mathbf{d}^\top\nabla f(\mathbf{x}^k + \alpha\mathbf{d}) + \text{higher order terms}}{h} - \frac{f(\mathbf{x}^k + \alpha\mathbf{d})}{h}\right] \\ &= \lim_{h\to 0}\Big[\mathbf{d}^\top\nabla f(\mathbf{x}^k + \alpha\mathbf{d}) + \text{terms of order } h \text{ and higher}\Big]. \end{aligned}$$

Thus

$$\boxed{f'(\alpha) = \mathbf{d}^\top\nabla f(\mathbf{x}^k + \alpha\mathbf{d})}$$



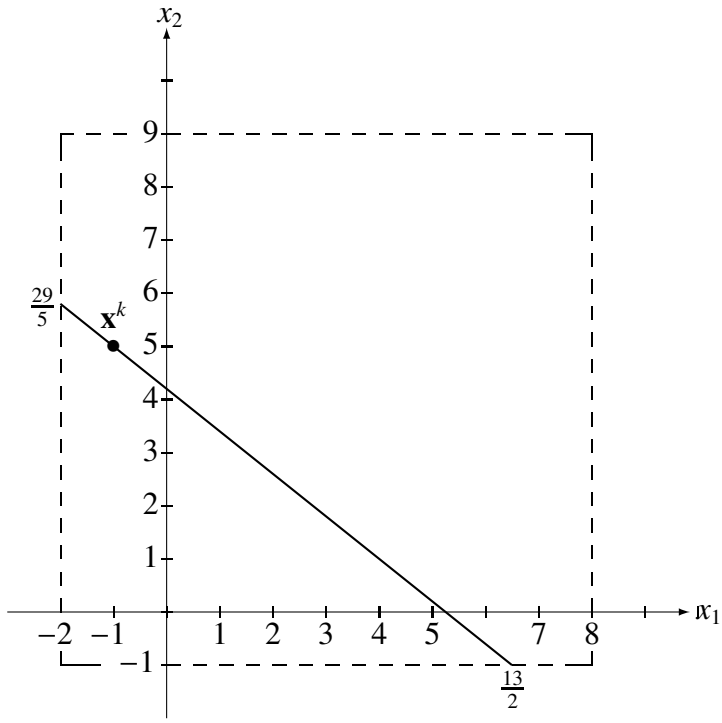
12.2.2 Staying Within Variable Bounds

How can we find α_0^{L} and α_0^{H}, which are required for the bisection line search, when the overall optimization problem instead includes bounds on $\mathbf{x}$?

In the graph above, $\mathbf{x}^k = [-1, 5]^\top$ and the search direction $\mathbf{d}^k = [5, -4]^\top$, so any point on $\mathbf{d}$ is described by

$$\mathbf{x}^k + \alpha \mathbf{d}^k = \begin{bmatrix} -1 \\ 5 \end{bmatrix} + \alpha \begin{bmatrix} 5 \\ -4 \end{bmatrix}.$$

Now suppose the box outlined by the axes in the x_1—x_2 plane describes the given bounds on $\mathbf{x}$, so that $\mathbf{x}^{\mathrm{L}} = [-2, -1]^\top$ and $\mathbf{x}^{\mathrm{H}} = [8, 9]^\top$. This situation is shown more clearly in the graph on the next page, in which the bounds on the variables are drawn as a dashed box.



At its left end the line to be searched intersects the dashed box where $x_1 = x_1^{\mathrm{L}} = -2$ or $-1 + 5\alpha = -2$, so $\alpha = -\frac{1}{5}$ (that makes $x_2 = 5 - 4(-\frac{1}{5}) = \frac{29}{5}$). At its right end the intersection is where $x_2 = x_2^{\mathrm{L}} = -1$ or $5 - 4\alpha = -1$, so $\alpha = \frac{3}{2}$ (that makes $x_1 = -1 + 5(\frac{3}{2}) = \frac{13}{2}$). Thus, for this example the lowest value of α is $\alpha_0^{\mathrm{L}} = -\frac{1}{5}$ and the highest is $\alpha_0^{\mathrm{H}} = \frac{3}{2}$. Of course it might turn out in other cases, depending on the orientation of the line relative to the variable bounds, that one or both intersection points are with the upper bounds on the variables.

We could have found the limits α^{L} and α^{H} algebraically, without drawing a picture, just by requiring that $\mathbf{x}^{\mathrm{L}} \le \mathbf{x}^k + \alpha\mathbf{d}^k \le \mathbf{x}^{\mathrm{H}}$, or

$$\begin{bmatrix} -2 \\ -1 \end{bmatrix} \le \begin{bmatrix} -1 \\ 5 \end{bmatrix} + \alpha \begin{bmatrix} 5 \\ -4 \end{bmatrix} \le \begin{bmatrix} 8 \\ 9 \end{bmatrix}.$$

This represents four scalar inequalities.

$$\begin{aligned} -2 \le -1 + 5\alpha \quad &\Rightarrow \quad \alpha \ge -\tfrac{1}{5} \\ -1 \le 5 - 4\alpha \quad &\Rightarrow \quad \alpha \le \tfrac{3}{2} \end{aligned} \qquad\qquad \begin{aligned} -1 + 5\alpha \le 8 \quad &\Rightarrow \quad \alpha \le \tfrac{9}{5} \\ 5 - 4\alpha \le 9 \quad &\Rightarrow \quad \alpha \ge -1 \end{aligned}$$

The bounds on the left correspond to the graph intersections we found above, and those on the right correspond to the intersections that the line would make, if it were extended, with

the upper limits on x_1 and x_2. From these four bounds on α we conclude that $\alpha \le \frac{3}{2}$ and $\alpha \ge -\frac{1}{5}$, as we found above graphically. In general for $\mathbf{x} \in \mathbb{R}^n$ we have, for $j = 1 \ldots n$,

$$\begin{aligned} x_j + \alpha d_j &\ge x_j^{\mathrm{L}} \\ \alpha &\ge \frac{x_j^{\mathrm{L}} - x_j}{d_j} \quad \text{if} \quad d_j > 0 \\ \alpha &\le \frac{x_j^{\mathrm{L}} - x_j}{d_j} \quad \text{if} \quad d_j < 0 \end{aligned} \qquad \begin{aligned} x_j + \alpha d_j &\le x_j^{\mathrm{H}} \\ \alpha &\le \frac{x_j^{\mathrm{H}} - x_j}{d_j} \quad \text{if} \quad d_j > 0 \\ \alpha &\ge \frac{x_j^{\mathrm{H}} - x_j}{d_j} \quad \text{if} \quad d_j < 0 \end{aligned}$$

Thus

$$\alpha^{\mathrm{L}} = \max_j \left\{ \left.\frac{x_j^{\mathrm{L}} - x_j}{d_j}\right|_{d_j>0}, \left.\frac{x_j^{\mathrm{H}} - x_j}{d_j}\right|_{d_j<0} \right\} \qquad \alpha^{\mathrm{H}} = \min_j \left\{ \left.\frac{x_j^{\mathrm{H}} - x_j}{d_j}\right|_{d_j>0}, \left.\frac{x_j^{\mathrm{L}} - x_j}{d_j}\right|_{d_j<0} \right\}$$

If we are doing a line search then $\mathbf{d} \ne \mathbf{0}$, but it is possible for some particular d_j to be zero; then α is not constrained by motion in that coordinate direction, and the corresponding term is omitted from the max and min over j.

The calculation of α^{L} and α^{H} described above is implemented in the `arange.m` routine listed below. It receives the current point `x` $= \mathbf{x}^k$, the direction of search `d` $= \mathbf{d}^k$, the upper and lower bounds `xh` $= \mathbf{x}^{\mathrm{H}}$ and `xl` $= \mathbf{x}^{\mathrm{L}}$ (assumed to contain `x`) and the number of variables `n`; it returns `al` and `ah`, the corresponding lower and upper limits on α.

```
 1 function [al,ah]=arange(x,d,xl,xh,n)
 2   al=-realmax;
 3   ah=+realmax;
 4   for j=1:n
 5       if(d(j) == 0) continue; end
 6       tl=(xl(j)-x(j))/d(j);
 7       th=(xh(j)-x(j))/d(j);
 8       if [ d(j) < 0 ]
 9          al=max(al,th);
10          ah=min(ah,tl);
11       else
12          al=max(al,tl);
13          ah=min(ah,th);
14       end
15   end
16 end
```

The function begins by $\boxed{\texttt{2-3}}$ initializing `al` $= -\infty$ and `ah` $= +\infty$, so that there are *no* bounds on α. These values are not useful for starting a line search, but they get replaced as the bounds on the `x(j)` are considered in the loop $\boxed{\texttt{4-15}}$. If $\boxed{\texttt{5}}$ `d(j)=0`, that `j` is skipped and the loop continues to the next coordinate direction. The terms involving x_j^{L} and x_j^{H} appearing in the formulas are computed as $\boxed{\texttt{6}}$ `tl` and $\boxed{\texttt{7}}$ `th` respectively. Then $\boxed{\texttt{8-14}}$ depending on the sign of `d(j)`, `al` and `ah` are updated so that when the loop is finished they have the values given above.

I used `arange.m` to compute the limits on α that we found by hand earlier, as shown in this Octave session excerpt.

```
octave:1> x=[-1;5];
octave:2> d=[5;-4];
octave:3> xl=[-2;-1];
octave:4> xh=[8;9];
octave:5> [al,ah]=arange(x,d,xl,xh,2)
al = -0.20000
ah =  1.5000
octave:6> quit
```

This result, $[\alpha^{\mathrm{L}}, \alpha^{\mathrm{H}}] = [-\frac{1}{5}, \frac{3}{2}]$, agrees with the interval we deduced.

We will routinely use `arange.m` to establish the starting interval $[\alpha_0^{\mathrm{L}}, \alpha_0^{\mathrm{H}}]$ over which to conduct any line search, so as to avoid points outside the known variable bounds for the nonlinear program. It might seem that the interval determined by `arange` is unnecessarily wide, because it can encompass negative values of α. If $\mathbf{d}$ really is a descent direction it should not be necessary in seeking a minimum to go the *opposite* way, so we might save work by ignoring the `al` returned by `arange` and always using $\alpha^{\mathrm{L}} = 0$. Alas, in solving real nonlinear programs nonconvexity can confuse even the cleverest of descent methods, and then the likelihood of missing a minimum can be reduced somewhat by searching the whole line. Whenever a line search fails, debugging should begin with checking whether the variable bounds $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ actually contain a minimizing point along the direction $\mathbf{d}$.

12.2.3 A Simple Bisection Line Search

Armed with a formula for the directional derivative and a routine to compute the starting interval of uncertainty, we can now implement the bisection line search algorithm flowcharted earlier. The `bls.m` routine listed at the top of the next page receives the current point `xk` $= \mathbf{x}^k$, the search direction `dk` $= \mathbf{d}^k$, the lower and upper bounds `xl` and `xh` on $\mathbf{x}$, the number of variables `n`, a pointer `grd` to a function that returns the gradient of the objective at a given point, and a (stationarity) convergence tolerance `tol` $= t$, and it returns `astar` $\approx \alpha^\star$.

The calculation begins [2] with an invocation of `arange` to find `al` $= \alpha^{\mathrm{L}}$ and `ah` $= \alpha^{\mathrm{H}}$. Then [3] up to 52 bisections are performed, enough to reduce the interval of uncertainty by a factor of 2^{52} or more than 10^{15} (see §17.5). If before convergence is achieved the width of the interval becomes numerically zero [5] or (much less likely) the iteration limit is met, the routine returns [16] the current `alpha`. Otherwise it finds [6-8] the trial point `x`, the gradient `g` there, and the directional derivative `fp`. Here the MATLAB statement `fp=dk'*g` evaluates the formula $f'(\alpha) = \mathbf{d}^\top \nabla f(\mathbf{x}^k + \alpha \mathbf{d})$ that we found in §12.2.1. If [9] `fp` is less in absolute value than the line search convergence tolerance `tol`, the routine returns the current `alpha`. If convergence has not yet been achieved, the sign of `fp` is used [10-14] to adjust the interval of uncertainty and the iterations continue.


```
function astar=bls(xk,dk,xl,xh,n,grd,tol)
  [al,ah]=arange(xk,dk,xl,xh,n);
  for s=1:52
      alpha=(al+ah)/2;
      if(al == ah) break; end
      x=xk+alpha*dk;
      g=grd(x);
      fp=dk'*g;
      if(abs(fp) < tol) break; end
      if(fp < 0)
         al=alpha;
      else
         ah=alpha;
      end
  end
  astar=alpha;
end
```

I tested `bls.m` by using it to perform the line search that we did analytically in §10.4, obtaining these results.

```
octave:1> format long
octave:2> xk=[2;2];
octave:3> dk=[-21;-16];
octave:4> xh=[3;3];
octave:5> xl=[-2;-2];
octave:6> astar=bls(xk,dk,xl,xh,2,@gnsg,0.01)
astar =  0.0962713332403274
octave:7> astar=bls(xk,dk,xl,xh,2,@gnsg,1e-8)
astar =  0.0962707182313482
octave:8> quit
```

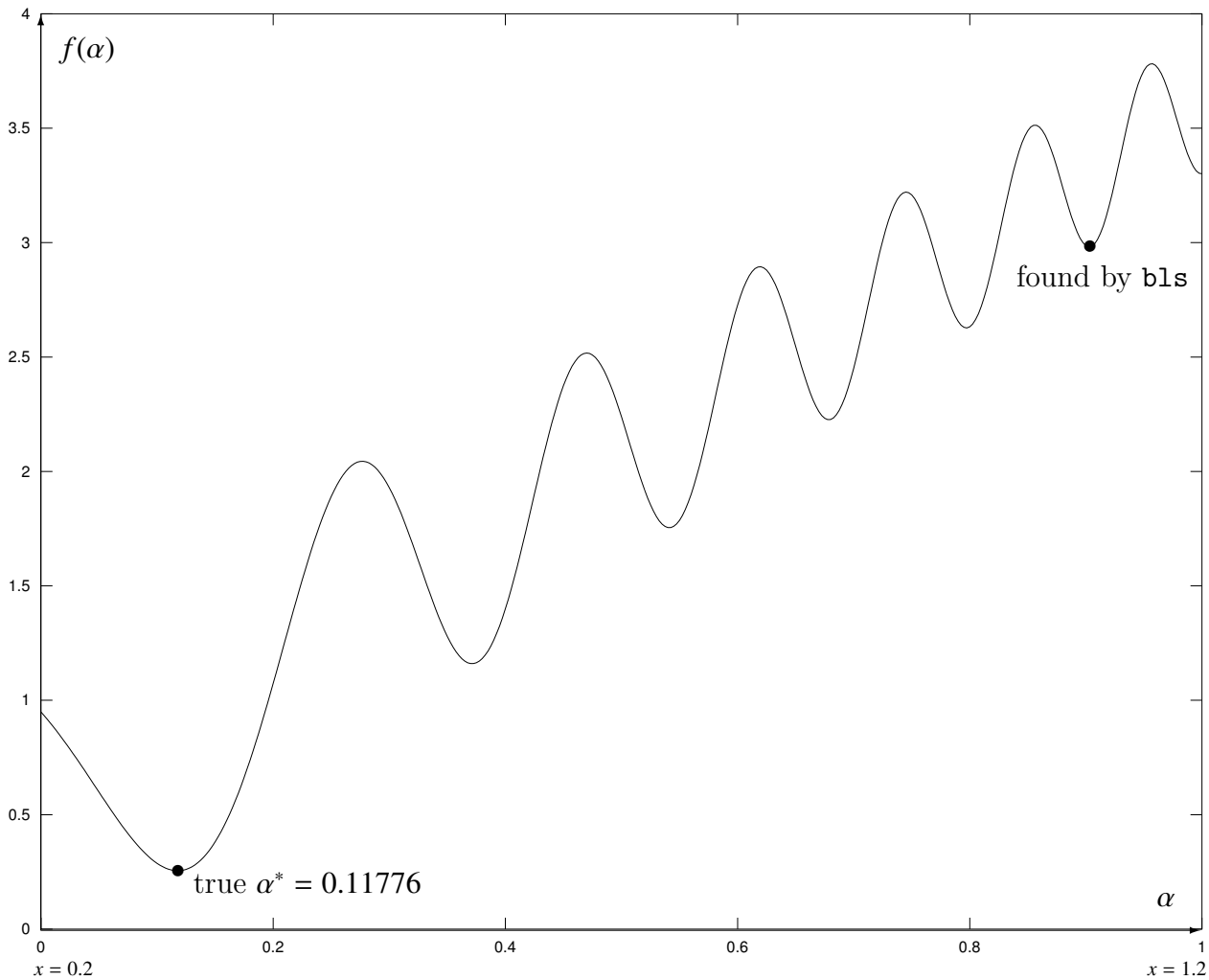
The answer we found by hand was $\alpha^\star = 697/7240 \approx$ `.0962707182320442`, so even with a line search tolerance as large as `0.01` the numerical approximation is quite good. We could now invoke `bls.m` in our steepest-descent solution of the `gns` problem instead of evaluating, or even deriving, the formula for $\alpha^\star$ (see Exercise 12.5.19).

12.3 Robustness Against Nonconvexity

In the line search examples of §10.4 and §12.2, $f(\mathbf{x})$ was convex so $f(\alpha)$ was unimodal. A function $f(\alpha)$ being minimized is **unimodal** if and only if it has a single local minimum [107, §7.1] [1, Exercise 8.10]. The logic of the bisection line search algorithm (and of several of the other methods tabled in §12.1) depends on $f(\alpha)$ having this property, so problems that are not unimodal, including most nonconvex problems, are much harder than those that are.

Suppose we want to minimize the wiggly function $f(x) = 3x + e^{-x}\cos(9\pi x^2)$ on the interval $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}] = [\frac{1}{5}, \frac{6}{5}]$. The next page shows a graph of $f(\mathbf{x}^{\mathrm{L}} + \alpha) \equiv f(\alpha)$ on the interval $[\alpha^{\mathrm{L}}, \alpha^{\mathrm{H}}] = [0, 1]$ along with an Octave session in which `bls.m` finds the wrong local minimum. The bisection line search algorithm first tries $\alpha_1 = \frac{1}{2}$ and finds the derivative negative, so it throws away the left half of the interval. The next trial point is $\alpha_2 = \frac{3}{4}$, where the derivative

is also negative, so it throws away the left half of the interval again. The trial point after that is $\alpha_3 = \frac{7}{8}$, where the derivative is again negative, so once more it throws away the left half of the interval. At $\alpha_4 = \frac{15}{16}$ the derivative is positive, but by then the remaining interval of uncertainty brackets a local minimum that is far from the global minimum in α and $\mathbf{x}$. Adding insult to injury, the objective value is *higher* at this point than where we started! This sort of disaster is unfortunately not confined to specially-contrived toy problems like this one, nor to the bisection line search.



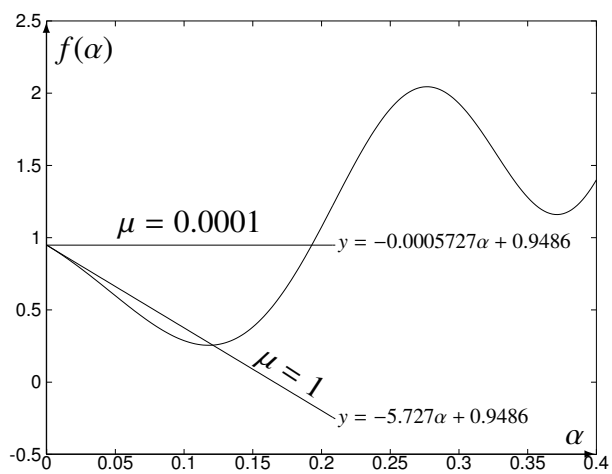

```
octave:1> xk=0.2;
octave:2> dk=1;
octave:3> xl=0.2;
octave:4> xh=1.2;
octave:5> astar=bls(xk,dk,xl,xh,1,@wiglg,1e-8)
astar =  0.90296
octave:6> quit
```


12.3.1 The Wolfe Conditions

The failure of the bisection line search on the wiggly function could have been averted by not looking so far from the starting point. The idea of restricting a line search to values of α that are at least not obviously wrong is embodied in the **Wolfe conditions** [157]. These conditions on $\alpha > 0$ can be stated in terms of $\nabla f(\mathbf{x})$ and $\mathbf{d}$ in the space of the overall optimization [4, §11.5] [5, p34] as on the left below, or in terms of α in the space of the line search subproblem as on the right. It is this second perspective that we will adopt here.

$$\begin{aligned} f(\mathbf{x}^k + \alpha\mathbf{d}^k) &\leq f(\mathbf{x}^k) + [\mu\nabla f(\mathbf{x}^k)^\top\mathbf{d}^k]\alpha \\ |\nabla f(\mathbf{x}^k + \alpha\mathbf{d}^k)^\top\mathbf{d}^k| &\leq \eta|\nabla f(\mathbf{x}^k)^\top\mathbf{d}^k| \end{aligned} \qquad \text{or} \qquad \begin{aligned} f(\alpha) &\leq f(0) + [\mu f'(0)]\alpha \\ |f'(\alpha)| &\leq \eta|f'(0)| \end{aligned}$$

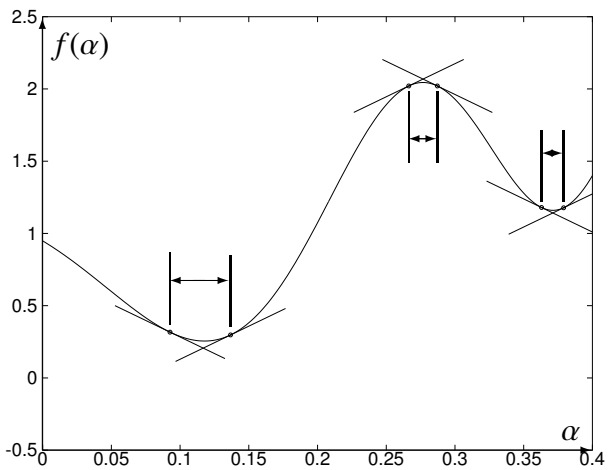
The first or **sufficient decrease condition** (also called the **Armijo condition**) requires that the function value $f(\alpha)$ go down by at least a little. This is a reasonable request, since we are trying to minimize $f(\alpha)$. The scalars $f(0)$ and $f'(0)$ on the right-hand side of the inequality are constants in a first-order Taylor series approximation $f(\alpha) \approx f(0) + f'(0)\,\alpha$ to $f(\alpha)$ at $\alpha = 0$. If $\mathbf{d}^k$ is a descent direction then $f'(\alpha) < 0$ and the straight line goes down as α increases from 0. Thus the inequality requires a decrease in the function at $\alpha^\star$ that is at least some fraction of that predicted by its linear approximation at $\alpha = 0$. That fraction is the parameter $\mu \in (0, 1)$, which is typically chosen to be on the order of 0.0001 so that only a small decrease is required.



This figure shows the first part of the wiggly function, along with its first-order Taylor series approximation (corresponding to $\mu = 1$) and the straight line describing the sufficient decrease condition for $\mu = 0.0001$. Here the sufficient decrease condition rules out all values of α greater than about 0.2.

If $\mathbf{d}^k$ is *not* a descent direction this condition only limits the amount by which the function can *increase,* but if μ is small that might still improve the robustness of the line search.

The second or **curvature condition** requires that $|f'(\alpha)|$ decrease by at least a little. This is also a reasonable request, since we are trying to find a point where $f'(\alpha) = 0$. If $\mathbf{d}^k$ is a descent direction then $f'(0) < 0$ and $|f'(0)| = -f'(0)$, so the condition reduces to $|f'(\alpha)| \leq -\eta f'(0)$. This inequality says that the directional derivative $f'(\alpha)$ can be of either sign at $\alpha^\star$ but no greater in absolute value than some fraction of its value at $\alpha = 0$. That fraction is the parameter $\eta \in [0, 1)$. If $\eta = 0$ this condition specifies an exact line search, but since that is not usually possible in numerical calculations the range of permissible η values is in practice the open interval $(0, 1)$.



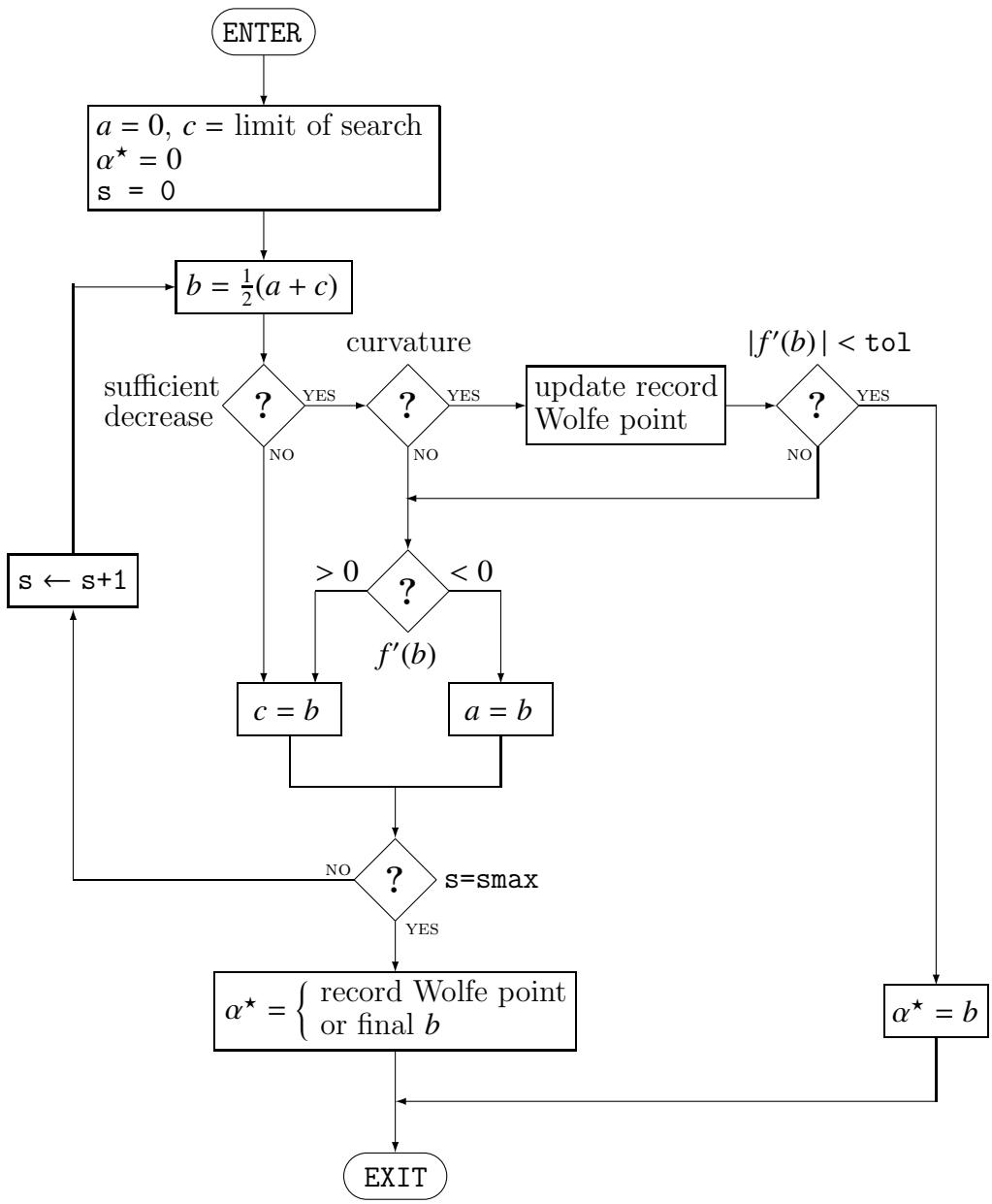
This figure shows tangent lines having slopes of $\pm\eta f'(0)$ with $\eta = 0.8$, defining three intervals over which the second Wolfe condition is satisfied. Because the first Wolfe condition excludes the rightmost two of these intervals, only the left one, where α is between about 0.08 and 0.14, satisfies both Wolfe conditions.

Convergence proofs for the DFP and BFGS algorithms, which we will encounter in §13.4, and for the Fletcher-Reeves algorithm of §14.5, require that line search results satisfy the Wolfe conditions. No particular values are prescribed for the Wolfe parameters μ and η, but the DFP and BFGS algorithms require $\mu > 0$ and $\eta < 1$, while the Fletcher-Reeves algorithm requires $\mu > 0$ and $\eta < \frac{1}{2}$ [5, p122,125-126]. Increasing μ or decreasing η makes it harder to find an α that satisfies the Wolfe conditions, but if $0 < \mu < \eta < 1$ and $f(\alpha)$ is smooth and bounded below then [5, Lemma 3.1] some α is sure to satisfy them both.

12.3.2 A Simple Wolfe Line Search

The flowchart on the next page outlines a naïve algorithm that can be viewed as a bisection line search in which certain restrictions are imposed in an attempt to satisfy the Wolfe conditions. It assumes that a minimum exists between $\alpha = 0$ and the positive value of α where a variable bound is first encountered in the given descent direction. At each stage in the search, an interval $[a, c]$ is assumed to contain a stationary point of $f(\alpha)$, so the flowchart

begins by setting a to zero and c to the upper bound on α. Before the search the starting point, corresponding to $\alpha = 0$, is the lowest point known, so $\alpha^\star$ is initialized to zero. This algorithm enforces an iteration limit, and to begin with no iterations have been performed so the iteration counter `s` is initialized to `0`.



Each iteration begins by finding the midpoint $b = \frac{1}{2}(a + c)$ of the current interval. If $f(b)$ is not sufficiently lower than $f(0)$, the first Wolfe condition is violated. But if $\mathbf{d}^k$ is a descent direction and f' is a continuous function of α, then $f(\alpha)$ must be less than $f(0)$ for *some* $\alpha > 0$

[148, §5] so if that is not true at b we must have stepped too far. The interval is shortened by moving c left to b, the iteration counter is incremented, and we bisect again. This process might be repeated several times until the sufficient decrease condition is satisfied.

If $f(b)$ is sufficiently lower than $f(0)$, so that the first Wolfe condition is satisfied, the algorithm checks whether $|f'(b)|$ is sufficiently less than $|f'(0)|$. If it is, the point $\alpha = b$ satisfies the second Wolfe condition, which we referred to above as the curvature condition, and a **Wolfe point** has been found. The box labeled "update record Wolfe point" remembers the Wolfe point having the lowest function value. If the current point $\alpha = b$ is the best one found so far, the convergence test is performed and if $|f'(b)| <$ `tol` that point is returned as $\alpha^\star$. In this case $\alpha^\star$ satisfies both the Wolfe conditions and the stationarity tolerance.

If the curvature condition is not satisfied, or if it is but the convergence tolerance is not met, then the sign of $f'(b)$ is used to discard half of the current interval. If $f'(b) > 0$ then the minimizing point is to the left of b, so c is moved left to b; if $f'(b) < 0$ then the minimizing point is to the right of b, so a is moved right to b. Then `s` is incremented, and the next iteration begins.

If the iteration limit is met before finding a Wolfe point that satisfies the convergence tolerance, the algorithm returns for $\alpha^\star$ either the best Wolfe point found so far or, if no Wolfe point has yet been found, the most recent interval midpoint b.

12.3.3 MATLAB Implementation

The source code of `wolfe.m` is listed in three parts beginning on the next page. The function header `1-2` shows the input and return parameters, which are summarized in the table below.

variable	meaning
`astar`	$\alpha^\star$ approximation returned
`rc`	return code, described later
`s`	last iteration used by the line search algorithm
`xk`	current point $\mathbf{x}^k$ in the overall optimization
`dk`	direction $\mathbf{d}^k$ of the line to be searched
`xl`	column vector of `n` lower bounds on the variables
`xh`	column vector of `n` upper bounds on the variables
`n`	number of variables in the overall optimization
`fcn`	pointer to MATLAB routine that returns $f(\mathbf{x})$
`grd`	pointer to MATLAB routine that returns $\nabla f(\mathbf{x})$
`mu`	Wolfe sufficient decrease parameter μ
`eta`	Wolfe curvature condition parameter η
`tol`	line search convergence tolerance t
`smax`	line search iteration limit


```
 1 % naive Wolfe line search based on bisection
 2 function [astar,rc,s]=wolfe(xk,dk,xl,xh,n,fcn,grd,mu,eta,tol,smax)
 3
 4 % initialize, and check for sensible inputs
 5   s=0;         % before searching we have done no iterations
 6   astar=0;     % and xk is the best point we know
 7   fk=fcn(xk);  % this is the function value there
 8   gk=grd(xk);  % this is the gradient vector there
 9   dfk=gk'*dk;  % this is the directional derivative in direction dk
10   if(dfk == 0) % does the function descend in either direction?
11      if(norm(gk) == 0)
12         rc=0;  % no because xk is a stationary point
13         return
14      else
15         rc=5;  % no because dk is orthogonal to gk
16         return
17      end
18   end
19   [amin,amax]=arange(xk,dk,xl,xh,n);
20   if(amin > 0)
21      rc=6;     % xk is not in [xl,xh] so no interval to search
22      return
23   end
24   if(mu <= 0 || mu >= 1 || eta <= 0 || eta >= 1)
25      rc=7;     % at least one Wolfe parameter has an illegal value
26      return
27   end
28   a=0;         % the line search will start from xk
29   xa=xk;       % that is the left end of the search interval
30   fa=fk;       % this the function value there
31   fr=fa;       % before searching it is the best value we know
32   if(dfk < 0)  % which direction is downhill?
33      c=amax;   % descend towards amax
34   else
35      c=amin;   % descend towards amin
36   end
37   rc=4;        % prepare to report failure
38
```

The variables `s` and `astar` are given initial values 5-6 so that they will be defined in the event of an early return (`rc=0`, `rc=5`, `rc=6`, or `rc=7`). Then 7-9 the function value $\texttt{fk} = f(0)$, gradient $\texttt{gk} = \nabla f(\mathbf{x}^k)$, and directional derivative $\texttt{dfk} = f'(0)$ are found at the starting point `xk`. If the directional derivative is zero 10 then no descent is possible, either because the gradient is zero ($\mathbf{x}^k$ is already a stationary point) or the direction vector is orthogonal to the gradient (maybe because $\mathbf{d}^k$ is zero); these cases are distinguished 11-17 and the routine returns without doing anything. Next `arange.m` is used to find limits `amin` and `amax` on α based on the variable bounds. If the lower limit returned by `arange` is positive then $\mathbf{x}^k \notin [\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ or $\mathbf{x}^{\mathrm{L}} \not< \mathbf{x}^{\mathrm{H}}$, so there is no interval to search and the routine returns without doing anything. Then a check is performed to ensure that the Wolfe parameters are in range, and if not the routine also returns without doing anything. The meanings of the various return codes are summarized in the table on the next page.

After these sanity checks are passed, we finish initializing 28-37 for the line search. The sufficient decrease condition depends on $\mathbf{d}^k$ actually being a descent direction, so the routine

checks the sign of $f'(0)$ [32] to determine which direction is downhill, and sets the right end c of the search interval accordingly. We intend that every direction $\mathbf{d}^k$ we search in will be a descent direction, but in the case of a nonconvex function it is possible (e.g., in §13.1 when `inv(H)` can be found in Newton descent even though $\mathbf{H}$ is not positive definite) to generate an uphill direction instead. In that case the most we can ask of a line search routine is that it minimize $f(\alpha)$ along the line whose direction is $\mathbf{d}^k$ within the specified bounds `[xl,xh]`. This routine does so, even if that involves moving "backwards" along $\mathbf{d}^k$.

The part of the program listed on the next page consists of one long loop [39-81] over the iterations `s`, implementing the logic outlined in the flowchart of §12.3.2. Each iteration begins by bisecting the current interval to find b [41], computing $f'(b)$ [42-44], and checking whether it has become zero [45]; if so, that point is stationary so no further iterations are possible, and it is returned [46-48] as the answer. Then the sufficient decrease condition is checked [52-53] and if it is not satisfied the search interval is shrunk towards the starting point [67-69]. If the sufficient decrease condition is satisfied, the curvature condition is checked [54], and if it is not satisfied control falls through the `end` [70] of the first `if` [53]. If the curvature condition is satisfied, a Wolfe point has been found [55] so the record function value `fr` is updated [56-58]. If the current point is a new record point, the convergence test is performed [59] and if it succeeds the current `astar` is returned as the answer [60-61]. If the sufficient decrease condition is satisfied but either the curvature condition is not satisfied or the convergence tolerance is not met, then the sign of $f'(b)$ is used to throw away one half of the current search interval [73-79]. Because of the finite precision of floating-point numbers it is possible that this process will result in an interval of zero width, and in that case [80] or if the iteration limit `smax` has been met, control falls through the `end` of the loop [81]. Otherwise the iteration counter is advanced and the next iteration begins.

`rc`	meaning
0	$\mathbf{x}^k$ is a stationary point so no descent is possible
1	$\alpha^\star$ satisfies the Wolfe conditions and $\lvert f'(\alpha^\star)\rvert <$ `tol`
2	$\alpha^\star$ satisfies the Wolfe conditions but $\lvert f'(\alpha^\star)\rvert \not< $ `tol`
3	$\lvert f'(\alpha^\star)\rvert <$ `tol` but $\alpha^\star$ does not satisfy the Wolfe conditions
4	$\lvert f'(\alpha^\star)\rvert \not< $ `tol` and $\alpha^\star$ does not satisfy the Wolfe conditions
5	$\mathbf{d}^k$ is orthogonal to $\nabla f(\mathbf{x}^k)$ so no descent is possible
6	$\mathbf{x}^k \notin [\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$
7	$\mu \notin (0,1)$ or $\eta \notin (0,1)$

If `wolfe.m` is used in a context where it is important to find an $\alpha^\star$ that satisfies the Wolfe conditions but unimportant whether that point is stationary, the routine can be invoked with `tol` set to a large number and the returned `astar` accepted only if `rc` is `0`, `1`, or `2`. If instead it is desired to perform an accurate bisection line search that uses the Armijo condition to reject unwanted local minima (as in the wiggly function) the routine can be invoked with `tol` set to a small number and the returned `astar` accepted if `rc` is `0`, `1`, or `3`.


```
% search the interval [a,c] for the minimizing point
  for s=1:smax
      b=(a+c)/2;             % find the midpoint value of alpha
      xb=xk+b*dk;            % find the midpoint value of x
      gb=grd(xb);            % find the gradient there
      dfb=gb'*dk;            % find the directional derivative there
      if(dfb == 0)           % is this point exactly stationary?
        rc=0;                % yes; inform the caller
        astar=b;             % and return it
        return
      end

%     check the Wolfe conditions
      fb=fcn(xb);            % find the function value at midpoint
      if(fb <= fk+mu*dfk*b)  % check sufficient decrease condition
         if(abs(dfb) <= eta*abs(dfk))  % check curvature condition
            rc=2;            % this is a Wolfe point
            if(fb < fr)      % is this the best point found so far?
               fr=fb;        % yes; update the record value
               astar=b;      % save the record point for return
               if(abs(dfb) < tol)      % is it stationary enough?
                  rc=1;      % the Wolfe point also satisfies tol
                  return     % return it
               end
            end
         end
      else
%        the function did not decrease enough; halve the step
         c=b;
         fc=fb;
         continue
      end

%     decide which half to keep and bisect the interval
      if(dfb < 0)
         a=b;                % the minimum is between b and c
         fa=fb;
      else
         c=b;                % the minimum is between a and b
         fc=fb;
      end
      if(a == c) break; end
  end

```

If the routine returns with `rc=4` and `s=smax`, a better result might be achieved by trying again with a larger value of `smax` (because this line search is based on bisection, it makes sense to set `smax=52` as in `bls.m` unless there is some reason to use a lower limit). If the routine returns with `rc=4` but `s` < `smax`, the search interval must have shrunk to zero and $f(\alpha^\star)$ is probably the minimum of the function along the line $\mathbf{x}^k + \alpha\mathbf{d}^k$ within the specified variable bounds. In that case the formulation of the problem should be reviewed to ensure that the variable bounds actually encompass the optimal point. The other return codes `rc=5`, `rc=6`, and `rc=7` suggest a programming error in the routine that invokes `wolfe.m`. Thus, all three of the return parameters from `wolfe.m` can be useful for figuring out what happened during the line search.

The final part of the routine, listed below, ensures that appropriate values are returned for `astar` and `rc` in the event that convergence is not achieved. If a Wolfe point was found then `rc` got set to 2 [55], and in that case the `astar` that was set then [58] is returned [86] as the answer. Otherwise $\alpha^\star$ is taken [88] to be the final point b resulting from the bisection line search. If it satisfies the convergence tolerance [89] then `rc=3` is returned [90-91]; otherwise `rc` is still set to its initial value of 4 [37] and that value is returned [93].

```
83 % out of iterations or search interval has shrunk to zero
84   if(rc == 2)
85 %    astar is the best Wolfe point but |f'(astar)| >= tol
86      return
87   else
88      astar=b;                % return the final non-Wolfe point
89      if(abs(dfb) < tol)      % is it at least stationary enough?
90         rc=3;                % yes; report that
91         return
92      else
93         return               % no; return with rc=4 set above
94      end
95   end
96 end
```

To test `wolfe.m`, I used it on the wiggly function as follows.

```
octave:1> xk=0.2;
octave:2> dk=1;
octave:3> xl=0.2;
octave:4> xh=1.2;
octave:5> [astar,rc,s]=wolfe(xk,dk,xl,xh,1,@wigl,@wiglg,0.0001,0.8,1e-8,50)
astar =  0.11776
rc =  1
s =  34
octave:6> quit
```

Now we get the true $\alpha^\star$, so enforcing the Wolfe conditions did keep this line search from finding the wrong local minimum of at least this wiggly test function.

12.4 Line Search in Steepest Descent

In §10 we studied two versions of the steepest-descent algorithm. The first version used an exact analytic line search based on a formula for $\alpha^\star(\mathbf{x};\mathbf{d})$ that we derived for the `gns` problem. The second version used the full steepest-descent step and can be applied to any problem; we implemented the full-step algorithm in the `sdfs.m` routine and used it to solve `gns` and `rb`.

Now we can write two other implementations, using the bisection and Wolfe line searches, to complete the following set of steepest-descent routines.

routine synopsis [xstar,k]=	algorithm for $\alpha^\star$
sd(xzero,xl,xh,n,kmax,epz,grd)	optimal step from bls
sdw(xzero,xl,xh,n,kmax,epz,fcn,grd)	Wolfe step from wolfe
sdfs(xzero,kmax,epz,grd,hsn)	full step from formula

Many of the nonlinear programming algorithms that we will study in subsequent Chapters make use of full steps in some descent direction, but when a line search is required we will usually use wolfe.m to perform it. Unless the nonlinear program that we are trying to solve is known for certain to be strictly convex, the robustness of any descent method based on a line search depends on enforcing the Wolfe conditions.

On the other hand, wolfe.m is complicated enough that it might be hard to follow the details of what is happening inside a descent algorithm that uses it. In studying the behavior of a method such as steepest descent it might therefore be more informative to use bls.m instead.

12.4.1 Steepest Descent Using bls.m

The routine below is similar to sdfs.m, but instead of using the full-step formula for $\alpha^\star$ it [8] invokes bls. For simplicity I have used the same tolerance [3] for both the descent method and the line search, but there might be situations in which it would be better if they were different.

```
 1 function [xstar,k]=sd(xzero,xl,xh,n,kmax,epz,grd)
 2   xk=xzero;
 3   tol=epz;
 4   for k=1:kmax
 5       g=grd(xk);
 6       if(norm(g) <= epz) break; end
 7       dk=-g;
 8       astar=bls(xk,dk,xl,xh,n,grd,tol);
 9       xk=xk+astar*dk;
10   end
11   xstar=xk;
12 end
```

The Octave session on the next page shows sd.m [5>] successfully solving the gns problem but failing [9>,10>,11>] to solve the rb problem. The routine detects optimality when [13>] rb is started from its optimal point $\mathbf{x}^\star = [1,1]^\top$, but when its published starting point $\mathbf{x}^0 = [-1.2,1]^\top$ is used, the point to which it converges has a gradient that is [12>] far from zero.


```
octave:1> format long
octave:2> xzero=[2;2];
octave:3> xl=[-2;-2];
octave:4> xh=[3;3];
octave:5> [xsd,ksd]=sd(xzero,xl,xh,2,20,1e-16,@gnsg)
xsd =

   0.750000000756451
  -0.749999999601187

ksd=  20
octave:6> xzero=[-1.2;1];
octave:7> xl=[-2;-1];
octave:8> xh=[2;2];
octave:9> [xsd,ksd]=sd(xzero,xl,xh,2,20,1e-16,@rbg)
xsd =

  -0.554727115497666
   0.296124455145839

ksd =  20
octave:10> [xsd,ksd]=sd(xzero,xl,xh,2,100,1e-16,@rbg)
xsd =

   0.459758038760584
   0.209774132099343

ksd =  100
octave:11> [xsd,ksd]=sd(xzero,xl,xh,2,1000,1e-16,@rbg)
xsd =

   0.458457195908287
   0.208574575848300

ksd =  1000
octave:12> rbg(xsd)
ans =

  -0.788128069575381
  -0.321684926357935

octave:13> [xsd,ksd]=sd([1;1],xl,xh,2,1000,1e-16,@rbg)
xsd =

   1
   1

ksd =  1
octave:14> quit
```

12.4.2 Steepest Descent Using `wolfe.m`

The routine on the next page is similar to `sd.m`, but instead of using `bls` it invokes `wolfe` to find $\alpha^\star$. To limit the number of arguments that must be passed to `sdw.m` I have fixed $\boxed{3}$ $\mu = 0.0001$ and $\boxed{4}$ $\eta = 0.4$, which meet the requirements for the DFP, BFGS, and Fletcher-

Reeves algorithms mentioned in §12.3.1. You might encounter situations in which it would make sense to use different numbers or to make them arguments of the function after all.

```
 1 function [xstar,k]=sdw(xzero,xl,xh,n,kmax,epz,fcn,grd)
 2   xk=xzero;
 3   mu=.0001;
 4   eta=0.4;
 5   smax=52;
 6   for k=1:kmax
 7       g=grd(xk);
 8       if(norm(g) <= epz) break; end
 9       dk=-g;
10       tol=1000*epz*norm(g);
11       [astar,rc,kw]=wolfe(xk,dk,xl,xh,n,fcn,grd,mu,eta,tol,smax);
12       if(rc > 3) break; end
13       xk=xk+astar*dk;
14   end
15   xstar=xk;
16 end
```

I set [5] the line search iteration limit `smax=52` as in `bls.m` because `wolfe.m` either halves the step or bisects the interval of uncertainty. Here the tolerance `tol` depends [10] on `epz` and $\|\nabla f(\mathbf{x}^k)\|$ so that the line search gets more precise as $\mathbf{x}^\star$ is approached, but depending on the problem some other heuristic might work better to reduce the number of descent iterations needed, or `tol=0.01` might work well enough. If `wolfe.m` fails [12] this routine gives up and [15] returns the best point it has found so far.

The Octave session below shows `sdw.m` successfully solving both the `gns` problem and the `rb` problem. Enforcing the Wolfe conditions does make steepest descent robust against nonconvexity [5, Theorem 3.2], but notice [9>] that `sdw.m` requires *many* iterations to get close to the solution of `rb`. In the next Chapter we will see that using a better descent direction can dramatically improve the speed with which we solve `rb` and other problems.

```
octave:1> format long
octave:2> xzero=[2;2];
octave:3> xl=[-2;-2];
octave:4> xh=[3;3];
octave:5> [xsdw,ksdw]=sdw(xzero,xl,xh,2,20,1e-8,@gns,@gnsg)
xsd =

   0.749999998476375
  -0.749999998560381

ksdw =  16
octave:6> xzero=[-1.2;1];
octave:7> xl=[-2;-1];
octave:8> xh=[2;2];
octave:9> [xsdw,ksdw]=sdw(xzero,xl,xh,2,10000,1e-8,@rb,@rbg)
xsdw =

   1.00000013041525
   1.00000026156901

ksdw =  10000
octave:10> quit
```


12.5 Exercises

12.5.1 [E] Is the simplex algorithm for linear programming a descent method, according to the description at the beginning of this Chapter? Is pure random search?

12.5.2 [E] What is the one-dimensional minimization problem solved by a line search? Is a line search necessarily in the direction of steepest descent?

12.5.3 [E] When is an exact line search appropriate? Explain. What is the goal of a numerical line search? Name one advantage of a derivative-free line search method, and one drawback.

12.5.4 [P] In §12.1 the golden section line search is described as "mathematically intriguing." Here is an outline of the algorithm, based on [1, p350].

0. Let $\lambda_0 = \alpha_0^{\mathrm{L}} + (1-r)(\alpha_0^{\mathrm{H}} - \alpha_0^{\mathrm{L}})$ and $\mu_0 = \alpha_0^{\mathrm{L}} + r(\alpha_0^{\mathrm{H}} - \alpha_0^{\mathrm{L}})$, where $r = \frac{1}{2}(\sqrt{5}-1) \approx 0.618$. Evaluate $f(\lambda_0)$ and $f(\mu_0)$, let $s = 0$, and go to step 1.

1. If $\alpha_s^{\mathrm{H}} - \alpha_s^{\mathrm{L}} < t$ STOP with $\alpha^\star \in [\alpha_s^{\mathrm{L}}, \alpha_s^{\mathrm{H}}]$. Otherwise, if $f(\lambda_s) > f(\mu_s)$, go to step 2, or if $f(\lambda_s) \le f(\mu_s)$, go to step 3.

2. Let $\alpha_{s+1}^{\mathrm{L}} = \lambda_s$ and $\alpha_{s+1}^{\mathrm{H}} = \mu_s$; then let $\lambda_{s+1} = \mu_s$, and $\mu_{s+1} = \alpha_{s+1}^{\mathrm{L}} + r(\alpha_{s+1}^{\mathrm{H}} - \alpha_{s+1}^{\mathrm{L}})$. Evaluate $f(\mu_{s+1})$, and go to step 4.

3. Let $\alpha_{s+1}^{\mathrm{L}} = \alpha_s^{\mathrm{L}}$ and $\alpha_{s+1}^{\mathrm{H}} = \mu_s$; then let $\mu_{s+1} = \lambda_s$, and $\lambda_{s+1} = \alpha_{s+1}^{\mathrm{L}} + (1-r)(\alpha_{s+1}^{\mathrm{H}} - \alpha_{s+1}^{\mathrm{L}})$. Evaluate $f(\lambda_{s+1})$, and go to step 4.

4. Replace s by $s+1$ and go to step 1.

(a) Flowchart this algorithm. (b) Implement the algorithm in a MATLAB function `[astar]=golden(f,x,d,xl,xh,t)` where `f` is a pointer to the objective function, `x` is the current point $\mathbf{x}^k$, `d` is the direction vector $\mathbf{d}^k$, `xl` $= \mathbf{x}^{\mathrm{L}}$ and `xh` $= \mathbf{x}^{\mathrm{H}}$ are the bounds on $\mathbf{x}$, and `t` $= t$ is the line search convergence tolerance. (c) Test your code on the wiggly function of §12.3. (d) Explain what makes this algorithm so clever.

12.5.5 [H] In §12.1 the Fibonacci line search is described as "mathematically intriguing." Study the detailed description given in [1, p351-354] and discuss the advantages and drawbacks of this particular derivative-free method.

12.5.6 [E] The logic of the bisection line search is that if the slope of $f(\alpha)$ is negative (positive) at the current point then the minimizing point is to the right (left) of that point. (a) Use a graph to explain this assumption. (b) What property is required of $f(\alpha)$ in order for the assumption to come true?

12.5.7 [E] If a bisection line search begins with an interval of uncertainty of $[0, 1]$, derive a formula for the width of the interval as a function of `s`, the number of line search iterations that are performed.

12.5.8 [H] In the example of §12.2.0, α_s is close to $\alpha^\star$. If the line search convergence tolerance t is small enough that this is considered not sufficiently close, the algorithm outlined in the flowchart takes another step. How many steps l does it take for α_{s+l} to get closer to $\alpha^\star$ than α_s was?

12.5.9 [E] In the bisection line search, if at some iteration $\alpha^{\mathrm{H}} = \alpha^{\mathrm{L}}$, why is it pointless to do further bisections? How could it happen that $\alpha^{\mathrm{H}} = \alpha^{\mathrm{L}}$ but $|f'(\alpha)| > t > 0$?

12.5.10 [E] How is $f'(\alpha)$ related to $f(\mathbf{x}^k + \alpha\mathbf{d}^k)$ and the derivatives of f with respect to $\mathbf{x}$?

12.5.11 [H] In the picture of §12.2.1, sketch the gradient vector $\nabla f(\mathbf{x}^k + \alpha\mathbf{d})$ at a few points on $\mathbf{d}$. Analytically calculate the dot product $\mathbf{d}^\top\nabla f(\mathbf{x}^k + \alpha\mathbf{d})$ as a function of α, and confirm the location of $\alpha^\star$ indicated in the graphs.

12.5.12 [E] By comparing points in the two graphs, confirm that the parabola graphed in §12.2.0 is the same one graphed in §12.2.1.

12.5.13 [H] If we know the starting point $\mathbf{x}^k$ of a line search, the direction of search $\mathbf{d}^k$, and bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ on the variables, how can we find upper and lower bounds $[\alpha^{\mathrm{L}}, \alpha^{\mathrm{H}}]$ on α? What happens if $d_j = 0$ for some value of j? Explain the geometrical basis of your answers.

12.5.14 [E] Why do we use the `arange.m` routine to establish the starting interval $[\alpha_0^{\mathrm{L}}, \alpha_0^{\mathrm{H}}]$ over which to conduct a line search? Does it ever make sense to accept an $\alpha_0^{\mathrm{L}} < 0$? Explain.

12.5.15 [E] Our derivation of the formulas for α^{L} and α^{H}, on which `arange.m` is based, assumed that $\mathbf{x}^k \in [\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$. What values are returned for `al` and `ah` if that is not true?

12.5.16 [H] In computing $[\alpha^{\mathrm{L}}, \alpha^{\mathrm{H}}]$ from $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ it is proposed to use the following formulas in place of those we derived in §12.2.2:

$$\alpha^{\mathrm{L}} = \max_j \min\left\{\frac{x_j^{\mathrm{L}} - x_j}{d_j}, \frac{x_j^{\mathrm{H}} - x_j}{d_j}\right\} \qquad \alpha^{\mathrm{H}} = \min_j \max\left\{\frac{x_j^{\mathrm{L}} - x_j}{d_j}, \frac{x_j^{\mathrm{H}} - x_j}{d_j}\right\}.$$

Do these formulas yield the same values as the formulas we derived? Give a convincing algebraic argument to support your claim.

12.5.17 [E] What does it mean to say that a quantity is "numerically zero"? Is such a quantity necessarily equal to zero exactly? If not, how different from zero can it be?

12.5.18 [P] The Octave session reproduced in §12.2.3 shows output from `bls.m` when it is used to do a line search. Modify the code to print the values of `s`, `alpha`, and `fp` that are generated, and repeat the calculations. How many iterations are used? How many correct digits can you produce in `astar` by reducing `tol`?

12.5.19 [P] Revise the `steep.m` program of §10.4 to use `bls.m` rather than the formula we derived for $\alpha^\star$. How do the results compare with what we found before?

12.5.20 [P] The prototypical optimization algorithm of §9.6 specifies that $\mathbf{x}^{k+1} \in [\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$, but for simplicity the `sdfs.m` routine of §10.5 ignores this requirement. Revise `sdfs.m` to take less than the full steepest-descent step if that is necessary in order to remain within the variable bounds.

12.5.21 [E] Define a unimodal function. Are all convex functions unimodal? If not, draw a convex function that is not unimodal. Are all unimodal functions convex? If not, draw a unimodal function that is not convex.

12.5.22 [H] Is the `rb` objective function unimodal? Present a convincing algebraic argument to support your claim.

12.5.23 [E] What property of $f(\alpha)$ can cause the bisection line search to find the wrong local minimum?

12.5.24 [E] Describe in words what the Wolfe conditions require.

12.5.25 [E] Why do the Wolfe conditions apply only to $\alpha > 0$? Why is μ usually chosen to be close to 0 while η is usually chosen to be close to 1? Explain how $\eta = 0$ corresponds to an exact line search. Does increasing μ and decreasing η make it easier or harder to find an α satisfying the Wolfe conditions? Explain why.

12.5.26 [H] The Wolfe conditions stated in §12.3.1 are sometimes referred to as the **strong Wolfe conditions** to distinguish them from the **weak Wolfe conditions** [4, Exercise 5.15] [5, p34]. The only difference between the strong and weak Wolfe conditions is that the weak curvature condition is $f'(\alpha) \geq \eta f'(0)$. Draw a graph to illustrate how the weak and strong curvature conditions differ in the values of α that they allow.

12.5.27 [P] The algorithm presented in §12.3.2 is described there as "a bisection line search in which certain restrictions are imposed in an attempt to satisfy the Wolfe conditions." How would the algorithm change if a more sophisticated technique than bisection were used to find each new trial point? Suppose in particular that instead of bisecting the interval on which $f'(\alpha)$ changes sign, linear interpolation is used to find the next trial α. (a) With the aid of a graph, explain how linear interpolation can be used to find the next guess at a zero of $f'(\alpha)$. Sometimes this is called the **secant method** [60, §7.1] [20, §2.3] for finding a zero of a function. (b) Revise the flowchart, and explain how the new algorithm works. (c) Revise the code, and test it to show that it works. Is it faster than the version using bisection?

12.5.28 [E] In the Wolfe line search algorithm of §12.3.2, why do we keep a record Wolfe point? When does it get updated? What is true of every record Wolfe point?

12.5.29 [H] What properties must $f(\alpha)$ have in order for it to be true that $f(\alpha)$ actually decreases if we take a small enough step in a descent direction?

12.5.30 [H] In attempting to perform a certain line search `wolfe.m` reports that no descent is possible from $\mathbf{x}^k$. What are the two possible reasons why this could have happened? How can you find out which of them occurred?

12.5.31 [H] In `wolfe.m` if `amin` > 0 we conclude that there is no interval to search. Why?

12.5.32 [E] Can `wolfe.m` be used to perform a very precise line search? Why might it be desirable to perform an imprecise line search instead?

12.5.33 [P] Revise the `steep.m` program of §10.4 to use `wolfe.m` rather than the formula we found for $\alpha^\star$. How do the results compare with what we found before?

12.5.34 [P] Construct a function with a global minimum that `wolfe.m` does not find.

12.5.35 [H] If an exact analytic line search is performed by evaluating a formula for $\alpha^\star(\mathbf{x};\mathbf{d})$, can we be confident that the $\alpha_k^\star$ we generate will satisfy the Wolfe conditions? If your answer is no, construct a counterexample based on the wiggly function of §12.3.0. If your answer is yes, why does an exact *numerical* line search generate points that do not necessarily satisfy the Wolfe conditions?

12.5.36 [E] Does the full-step steepest-descent algorithm generate step lengths α_k that satisfy the Wolfe conditions? If so, prove it. If not, explain why not.

12.5.37 [P] Exactly how does `sd.m` fail on the `rb` problem? Write a MATLAB program that invokes `sd` in a loop to perform one iteration at a time, as described in §10.6.1, and plots its (non)convergence trajectory over contours of the objective function. How can the algorithm stall at a point where the gradient is far from zero?

12.5.38 [P] Why does `sdw.m` take so long to solve the `rb` problem? Write a MATLAB program that invokes `sdw` in a loop to perform one iteration at a time, as described in §10.6.1, and plots its convergence trajectory over contours of the objective function. Does the picture suggest some change in the way `wolfe` is being used that might accelerate the convergence of `sdw.m`?

13

Newton Descent

In §10 we developed the steepest descent algorithm, which is much faster than pure random search yet still quite robust. Unfortunately it has only linear convergence, and bad conditioning of the Hessian matrix can sometimes make it, like pure random search, too slow to be useful. To be fast a minimization algorithm must somehow use second-order information about the function. **Newton descent** uses the Hessian as well as the gradient and thereby achieves quadratic convergence when it works at all, independent of Hessian conditioning [107, p225] [5, p27,44-45]. In this Chapter we will study Newton descent and its character flaws, along with several variants that are more robust than plain Newton descent but still [4, Theorem 11.3] have superlinear convergence.

13.1 The Full-Step Newton Algorithm

In §10.1 we used the Taylor's series expansion for a function of n variables,

$$f(\mathbf{x}) \approx q(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x} - \bar{\mathbf{x}}) + \tfrac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^\top \mathbf{H}(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}),$$

to derive the direction of steepest descent. Instead we could minimize the quadratic model function $q(\mathbf{x})$ and move to (or towards) that point. Setting the gradient equal to zero we find

$$\begin{aligned}
\nabla q(\mathbf{x}) = \nabla f(\bar{\mathbf{x}}) + \mathbf{H}(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) &= 0 \\
\mathbf{H}(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) &= -\nabla f(\bar{\mathbf{x}}) \\
(\mathbf{x} - \bar{\mathbf{x}}) &= [\mathbf{H}(\bar{\mathbf{x}})]^{-1}\,(-\nabla f(\bar{\mathbf{x}})) \\
\mathbf{x} &= \bar{\mathbf{x}} - [\mathbf{H}(\bar{\mathbf{x}})]^{-1}\,\nabla f(\bar{\mathbf{x}})
\end{aligned}$$

as the minimizing point of $q(\mathbf{x})$. Thus we could move from $\mathbf{x}^k = \bar{\mathbf{x}}$ to $\mathbf{x}^{k+1}$ by letting

$$\mathbf{x}^{k+1} = \mathbf{x}^k - [\mathbf{H}(\mathbf{x}^k)]^{-1}\nabla f(\mathbf{x}^k) = \mathbf{x}^k + \mathbf{d}^k$$

where the vector

$$\mathbf{d}^k = -[\mathbf{H}(\mathbf{x}^k)]^{-1}\nabla f(\mathbf{x}^k)$$

is called the **full Newton step**.

The `ntplain.m` routine listed at the top of the next page uses this update formula to implement the **full-step Newton descent algorithm**. Comparing this update to the one we found for steepest descent in §10.4 reveals that steepest descent is a special case of Newton descent when $\mathbf{H} = \mathbf{I}$.


```
function [xstar,kp]=ntplain(xzero,kmax,epz,grd,hsn)
  xk=xzero;
  for kp=1:kmax
%     find the uphill direction
      g=grd(xk);
      if(norm(g) <= epz) break; end

%     find the full Newton step downhill
      H=hsn(xk);
      d=-inv(H)*g;

%     take the step
      xk=xk+d;
  end
  xstar=xk;
end
```

This routine is identical to the `sdfs.m` routine of §10.5 except for the calculation of the descent direction `d`. I tested `ntplain.m` on the `gns` problem and got the answer in one iteration (when convergence is attained `kp` $= k + 1$ so here $\mathbf{x}^\star = \mathbf{x}^1$; see §28.4.3).

```
octave:1> [xstar,kp]=ntplain([2;2],10,1e-6,@gnsg,@gnsh)
xstar =

   0.75000
  -0.75000

kp =  2
octave:2> quit
```

At its starting point $\bar{\mathbf{x}} = [2, 2]^\top$, the `gns` problem has the gradient and Hessian that we found in §10.5. Using that data, its objective can be written like this.

$$\begin{aligned} f(\mathbf{x}) &= 4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1 \\ &= 34 + [21, 16]\begin{bmatrix} x_1 - 2 \\ x_2 - 2 \end{bmatrix} + \tfrac{1}{2}[x_1 - 2, x_2 - 2]\begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix}\begin{bmatrix} x_1 - 2 \\ x_2 - 2 \end{bmatrix} \\ &= f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x} - \bar{\mathbf{x}}) + \tfrac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^\top\mathbf{H}(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) = q(\mathbf{x}) \end{aligned}$$

so it is equal to its own quadratic model function (see Exercise 10.9.14). We derived the full Newton step to minimize $q(\mathbf{x})$, so for this problem it minimizes the objective. The Hessian of this function is positive definite, so $f(\mathbf{x})$ is strictly convex and $\mathbf{x}^\star = [\frac{3}{4}, -\frac{3}{4}]^\top$ is its unique global minimizing point. In general, Newton descent finds $\mathbf{x}^\star$ in a single step whenever $f(\mathbf{x})$ is quadratic and $\mathbf{H}$ is positive definite, just as steepest descent finds $\mathbf{x}^\star$ in one step if $\mathbf{H} = \mathbf{I}$.

Next I tried `ntplain.m` on the `rb` problem, with the mixed results shown at the top of the next page. From the catalog starting point $\mathbf{x}^0 = [-1.2, 1]^\top$ it finds the optimal point in 6 iterations, but starting from $\bar{\mathbf{x}} = [-1.2, 1.445]^\top$ it fails with some nasty messages about $\mathbf{H}$ being a singular matrix.


```
octave:3> [xstar,kp]=ntplain([-1.2;1],10,1e-6,@rbg,@rbh)
xstar =

   1.00000
   1.00000

kp =  7
octave:2> [xstar,kp]=ntplain([-1.2;1.445],6,1e-6,@rbg,@rbh)
warning: inverse: matrix singular to machine precision, rcond = 0
warning: inverse: matrix singular to machine precision, rcond = 0
warning: inverse: matrix singular to machine precision, rcond = 0
warning: inverse: matrix singular to machine precision, rcond = 0
warning: inverse: matrix singular to machine precision, rcond = 0
warning: inverse: matrix singular to machine precision, rcond = 0
xstar =

   NaN
   NaN

kp =  6
octave:3> quit
```

The trouble must be in computing `inv(H)`, so I abandoned that way of getting `d` and tried solving the linear system $\mathbf{Hd} = -\mathbf{g}$ by Gauss elimination instead. Recall from §8.6.2 that to do that we begin by finding an upper triangular matrix $\mathbf{U}$ such that $\mathbf{H} = \mathbf{U}^\top\mathbf{U}$. Then the linear system can be written $\mathbf{U}^\top\mathbf{Ud} = -\mathbf{g}$ or $\mathbf{U}^\top[\mathbf{Ud}] = -\mathbf{g}$. If we let $\mathbf{y} = \mathbf{Ud}$ then we can find $\mathbf{d}$ in two steps. First we solve $\mathbf{U}^\top\mathbf{y} = -\mathbf{g}$ for $\mathbf{y}$ and then we solve $\mathbf{Ud} = \mathbf{y}$ for $\mathbf{d}$. In MATLAB this process looks like statements 10-12 in the routine `ntchol.m` listed below

```
 1 function [xstar,kp]=ntchol(xzero,kmax,epz,grd,hsn)
 2   xk=xzero;
 3   for kp=1:kmax
 4 %     find the uphill direction
 5       g=grd(xk);
 6       if(norm(g) <= epz) break; end
 7
 8 %     find the full Newton step downhill
 9       H=hsn(xk);
10       U=chol(H);
11       y=U'\(-g);
12       d=U\y;
13
14 %     take the step
15       xk=xk+d;
16   end
17   xstar=xk;
18 end
```

Output from `ntchol.m` is shown at the top of the next page. The MATLAB `chol()` function can factor a matrix only if it is positive definite, so that step fails with a different error message. It turns out that `H` has a determinant of zero, so neither approach allows us to use plain Newton descent if we start from $[-1.2, 1.445]^\top$ or if the algorithm happens to encounter a point where $\mathbf{H}(\mathbf{x})$ is singular. How likely is that to happen?


```
octave:1> xzero=[-1.2,1.445];
octave:2> [xstar,k]=ntchol(xzero,10,1e-6,@rbg,@rbh)
error: chol: matrix not positive definite
error: called from ntchol.m at line 10, column 8
octave:3> H=rbh(xzero)
H =

   1152    480
    480    200

octave:4> det(H)
ans = 0
octave:5> quit
```

In §10.7 we found that for the `rb` problem

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} -400x_2 + 1200x_1^2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}.$$

This matrix is positive definite if its leading principal minors are positive, or

$$\begin{aligned} 200(1200x_1^2 - 400x_2 + 2) - 160000x_1^2 &> 0 \\ \text{and} \quad 1200x_1^2 - 400x_2 + 2 &> 0. \end{aligned}$$

These inequalities are both satisfied wherever $x_2 < x_1^2 + \frac{1}{200}$ (see Exercise 13.5.6).

Along the line $x_2 = x_1^2 + \frac{1}{200}$ the determinant of $\mathbf{H}(\mathbf{x})$ (the expression on the left in the first inequality) is zero, and the Hessian is singular only there; everywhere else, whether it is positive definite or not, it has an inverse. Why not somehow avoid the points where it is singular and go back to using `d=-inv(H)*g`? Unfortunately, the solution of $\mathbf{Hd} = -\mathbf{g}$ is sure to be a descent direction only if $\mathbf{H}$ is positive definite. To see this recall from §10.8 that $\mathbf{d}$ is a descent direction only if

$$\begin{aligned} \nabla f(\mathbf{x})^\top \mathbf{d} &< 0 \\ \mathbf{g}^\top(-[\mathbf{H}]^{-1}\mathbf{g}) &< 0 \\ -\mathbf{g}^\top([\mathbf{H}]^{-1}\mathbf{g}) &< 0 \\ \mathbf{g}^\top[\mathbf{H}]^{-1}\mathbf{g} &> 0 \end{aligned}$$

and this is sure to be true only if $\mathbf{H}$ is positive definite. So to solve the `rb` problem using plain Newton descent we must start at a point where $x_2 < x_1^2 + \frac{1}{200}$ and hope that no iterate $\mathbf{x}^k$ generated by the algorithm violates that inequality.

13.2 The Modified Newton Algorithm

Because plain Newton descent requires that the Hessian of the objective be positive definite, the algorithm is poorly suited to problems that are not strictly convex. Although some important applications give rise to strictly convex programs, a practical general-purpose

method must be more robust. Steepest descent is quite robust, which suggests the following strategy. If at some iterate $\mathbf{H}(\mathbf{x}^k)$ is not positive definite, modify it to be closer to the identity so that $\mathbf{d}$ turns out to be closer to the steepest-descent direction. This **modified Newton** algorithm is implemented in the code listed below.

```
 1 function [xstar,kp,nm,rc]=ntfs(xzero,kmax,epz,grd,hsn,gama)
 2 % modified Newton taking full step
 3   n=size(xzero,1);                 % get number of variables
 4   xk=xzero;                        % set starting point
 5   nm=0;                            % no modifications yet
 6   for kp=1:kmax                    % do up to kmax iterations
 7       g=grd(xk);                   % find uphill direction
 8       if(norm(g) <= epz)           % is xk stationary?
 9          xstar=xk;                   % yes; declare xk optimal
10          rc=0;                       % flag convergence
11          return                      % and return
12       end                          % no; continue iterations
13       H=hsn(xk);                   % get current Hessian matrix
14       [U,p]=chol(H);               % try to factor it
15       while(p ~= 0)                % does H need modification?
16         if(gama >= 1 || gama < 0)  % yes; can it be modified?
17            xstar=xk;                 % no; gama value prevents that
18            rc=2;                     % flag nonconvergence
19            return                    % and return
20         end                        % yes; modification possible
21         H=gama*H+(1-gama)*eye(n);    % average with identity
22         [U,p]=chol(H);               % and try again
23         nm=nm+1;                     % count the modification
24       end                          % now Hd=U'Ud=-g
25       y=U'\(-g);                   % solve U'y=-g for y
26       dk=U\y;                      % solve Ud=y for d
27       xk=xk+dk;                    % take the full step
28   end                              % and continue
29   xstar=xk;                        % out of iterations
30   rc=1;                            % so no convergence yet
31 end
```

This code resembles `ntchol.m` in that it uses Gauss elimination to solve $\mathbf{Hd} = -\mathbf{g}$ for the descent direction $\mathbf{d}$. Now, however, the MATLAB function `chol()` is invoked [14] with an additional return parameter `p` that signals whether `H` was positive definite ($\mathtt{p} = 0$) or not ($\mathtt{p} \neq 0$). If the matrix factorization failed because `H` was not positive definite [15] then `H` is modified [21] to be the weighted average

$$\mathbf{H} \leftarrow \gamma\mathbf{H} + (1-\gamma)\mathbf{I}$$

of its previous value and the identity matrix. The weighting is specified by the parameter `gama` (`gamma` is a reserved word in MATLAB) which can be given any value between 0 (if `H` is not positive definite use steepest descent for this iteration) and 1 (if `H` is not positive definite resign with `rc=2`). After `H` is modified another attempt is made [22] to factor it, updating the flag `p`, and [15] the process continues until `H` is close enough to the identity that it is positive definite. Then its factors are used in the usual way [25-27] to compute $\mathbf{x}^{k+1}$.

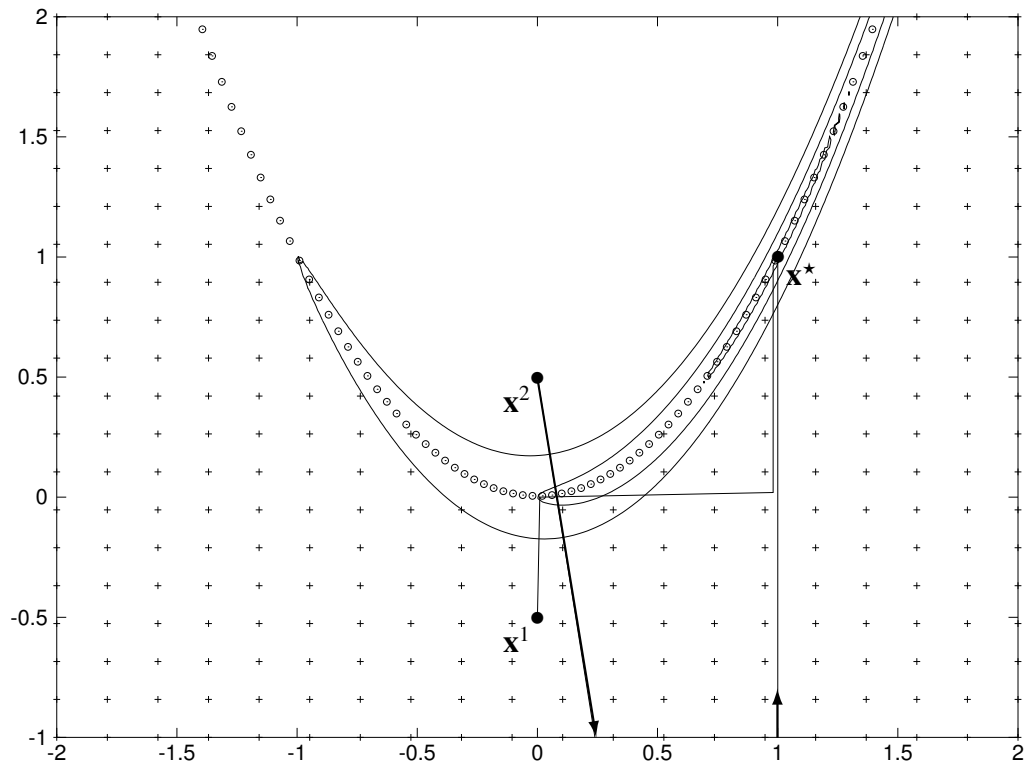
In addition to `xstar` and `kp`, this routine returns [1] the total number of Hessian modifications `nm` that it made and a return code `rc` to indicate whether (`rc=0`) the convergence tolerance `epz` was satisfied, (`rc=1`) the specified iteration limit `kmax` was met, or (`rc=2`) it was necessary to modify `H` but `gama` had a value which made that impossible.

I tried `ntfs.m` on the `rb` problem starting from $[-1.2, 1.445]^\top$, where we found that $\mathbf{H}(\mathbf{x})$ is singular, with the following results.

```
octave:1> [xstar,kp,nm,rc]=ntfs([-1.2;1.445],10,1e-6,@rbg,@rbh,0.5)
xstar =

   1.00000
   1.00000

kp =  7
nm =  1
rc = 0
octave:2> quit
```

Convergence was attained in `kp-1=6` iterations and only one Hessian modification was required, so 5 of the iterations used were full Newton steps ensuring superlinear convergence. To study the behavior of this algorithm in more detail, I wrote the MATLAB program `rbntfs.m` listed below to produce the picture on the next page.

```
%rbntfs.m: study the solution of rb using modified Newton descent
clear; clf; set(gca,'FontSize',20)
xl=[-2;-1];                                   % catalog lower bounds
xh=[2;2];                                     % catalog upper bounds
axis([xl(1),xh(1),xl(2),xh(2)],'equal')       % set graph axes
hold on                                       % start the graph
[xc,yc,zc,zmin,zmax]=gridcntr(@rb,xl,xh,200); % grid the objective
vc=[0.1,1,4];                                 % set contour levels
contour(xc,yc,zc,vc)                          % draw the contours

for p=1:100                                   % find 100 points on
    x(p)=xl(1)+(xh(1)-xl(1))*((p-1)/99);      % the curve x2=x1^2+1/200
    y(p)=x(p)^2+0.005;                        % where the Hessian
end                                           % is singular
plot(x,y,'o')                                 % and plot them
plotpd(xl,xh,20,@rbh)                         % show where H is pd

yks=[-0.5,+0.5];                              % define two starting points
for L=1:2                                     % for each
    xk=[0;yks(L)];                            % start there
    for k=1:10                                        % do up to 10 iterations
        [xkp,kp,nm,rc]=ntfs(xk,1,1e-6,@rbg,@rbh,0.5); % of modifed Newton
        xt(k)=xk(1);                                  % capture
        yt(k)=xk(2);                          % the iterate
        if(rc == 0) break; end                % quit if tolerance met
        xk=xkp;                               % otherwise update iterate
    end                                       % and continue
    plot(xt,yt)                               % plot convergence trajectory
end
hold off
print -deps -solid rbntfs.eps
```

The program [7-9] draws three contours of the `rb` problem. Then [11-15] it plots as small circles 100 points on the curve where $\mathbf{H}(\mathbf{x})$ is singular, and [16] plots as plus signs the points on a 20×20 grid where $\mathbf{H}(\mathbf{x})$ is positive definite. Finally [18-29] it plots the convergence trajectory of `ntfs.m` from two different starting points, with the weighting factor $\gamma = \frac{1}{2}$.

This analysis shows that $\mathbf{H}(\mathbf{x})$ is positive definite only below the curve where it is singular. From $\mathbf{x}^1 = [0, -\frac{1}{2}]^\top$ the algorithm takes 5 full Newton steps, only 3 of which are clearly visible, to reach $\mathbf{x}^\star$. From $\mathbf{x}^2 = [0, +\frac{1}{2}]^\top$ it takes 6 steps, modifying the Hessian 8 times and making an excursion far outside the frame of the picture before also reaching $\mathbf{x}^\star$.

To produce the field of plus signs showing where $\mathbf{H}(\mathbf{x})$ is positive definite, I used the `plotpd.m` routine listed on the next page. It computes the coordinates [5-7] of each point on an `npt` $\times$ `npt` grid within the variable bounds `[xl,xh]`, evaluates [8] the Hessian there, and finds [9-10] the leading principal minors. If the Hessian is positive definite [11-12] it plots a plus sign; if it is positive semidefinite [14-15] it plots a small circle. We will use `plotpd()` again in §18.

As I mentioned at the beginning of this Chapter, the convergence rate of Newton descent is $r = 2$ independent of the condition number of $\mathbf{H}(\mathbf{x}^\star)$. Modified Newton descent converges the same way if all of the $\mathbf{H}(\mathbf{x}^k)$ are positive definite, and with $r > 1$ if at least some of them are. However, in both algorithms the condition number $\kappa(\mathbf{H}(\mathbf{x}^k))$ *does* affect the numerical accuracy with which each $\mathbf{d}^k$ is found, as we shall see in §18.4.2, so bad conditioning of the Hessian might limit the precision with which $\mathbf{x}^\star$ can be determined (see Exercise 13.5.16).


```
function plotpd(xl,xh,npt,hsn)
% plot + where H is pd, o where it is psd
  for i=1:npt;
      for j=1:npt;
          xi(i)=xl(1)+(xh(1)-xl(1))*((i-1)/(npt-1));
          yi(j)=xl(2)+(xh(2)-xl(2))*((j-1)/(npt-1));
          x=[xi(i);yi(j)];
          H=hsn(x);
          lpm1=H(1,1);
          lpm2=H(1,1)*H(2,2)-H(1,2)*H(2,1);
          if(lpm1 > 0 && lpm2 > 0)
             plot(xi(i),yi(j),'+');
          else
             if(lpm1 >= 0 && lpm2 >= 0)
                plot(xi(i),yi(j),'o');
             end
          end
      end
  end
end
```

13.3 Line Search in Newton Descent

Instead of taking the full step in the modified Newton algorithm we could perform a line search in each descent direction `dk`, and depending on the problem that might reduce the number of descent iterations that are needed. I wrote two functions, using the bisection and Wolfe line searches, to complete the following set of Newton descent routines.

routine synopsis `[xstar,kp,nm,rc]=`	algorithm for $\alpha^\star$
`nt(xzero,xl,xh,kmax,epz,grd,hsn,gama)`	optimal step from `bls`
`ntw(xzero,xl,xh,kmax,epz,fcn,grd,hsn,gama)`	Wolfe step from `wolfe`
`ntfs(xzero,kmax,epz,grd,hsn,gama)`	full step from formula

We want our line searches to be in downhill directions, so it is still necessary to ensure that each `H` is positive definite.

13.3.1 Modified Newton Using `bls.m`

The `nt.m` routine listed on the next page differs from `ntfs.m` in only three particulars. Because `bls.m` requires the variable bounds `xl` and `xh`, these must be included in the `nt.m` calling sequence [1]. As in `sd.m`, the line search tolerance `tol` is [7] set equal to the descent method tolerance `epz`. Finally, instead of taking a full step the new point is found [30] as `xk+astar*dk`.


```
function [xstar,kp,nm,rc]=nt(xzero,xl,xh,kmax,epz,grd,hsn,gama)
% modified Newton using bisection line search
  n=size(xzero,1);                 % get number of variables
  xk=xzero;                        % set starting point
  nm=0;                            % no modifications yet
  rc=0;                            % assume it will converge
  tol=epz;                         % set line search tolerance
  for kp=1:kmax                    % do up to kmax descents
      g=grd(xk);                   % find uphill direction
      if(norm(g) <= epz)           % is xk stationary?
         xstar=xk;                   % yes; declare xk optimal
         rc=0;                       % flag convergence
         return                      % and return
      end                          % no; continue iterations
      H=hsn(xk);                   % get current Hessian matrix
      [U,p]=chol(H);               % try to factor it
      while(p ~= 0)                % does H need modification?
        if(gama >= 1 || gama < 0)  % yes; can it be modified?
           xstar=xk;                 % no; gama value prevents that
           rc=2;                     % flag nonconvergence
           return                    % and return
        end                        % yes; modification possible
        H=gama*H+(1-gama)*eye(n);    % average with identity
        [U,p]=chol(H);               % and try again
        nm=nm+1;                     % count the modification
      end                          % now Hd=U'Ud=-g
      y=U'\(-g);                   % solve U'y=-g for y
      dk=U\y;                      % solve Ud=y for d
      astar=bls(xk,dk,xl,xh,n,grd,tol);
      xk=xk+astar*dk;              % take the optimal step
  end                              % and continue
  xstar=xk;                        % out of iterations
  rc=1;                            % so no convergence yet
end
```

This routine works better than `sd.m` on the `gns` and `rb` problems (compare the results below and on the next page with those in §12.4.1) and it finds both minimizing points accurately. However, its convergence tolerance is never satisfied when solving the `rb` problem, so it always returns `rc=1` (see Exercise 13.5.18).

```
octave:1> format long
octave:2> xzero=[2;2];
octave:3> xl=[-2;-2];
octave:4> xh=[3;3];
octave:5> [xstar,kp,nm,rc]=nt(xzero,xl,xh,20,1e-16,@gnsg,@gnsh,0.5)
xstar =

   0.750000000000000
  -0.750000000000000

kp =  2
nm = 0
rc = 0
octave:6> quit
```



```
octave:1> format long
octave:2> xl=[-2;-1];
octave:3> xh=[2;2];
octave:4> [xstar,kp,nm,rc]=nt([0;-0.5],xl,xh,20,1e-16,@rbg,@rbh,0.5)
xstar =

   0.999999984277081
   0.999999970197678

kp =  20
nm = 0
rc =  1
octave:5> [xstar,kp,nm,rc]=nt([0;0.5],xl,xh,20,1e-16,@rbg,@rbh,0.5)
xstar =

   1.00000011204406
   1.00000023841858

kp =  20
nm =  8
rc =  1
octave:6> quit
```

Although `nt.m` uses the allowed iterations in these experiments, increasing the iteration limit does not significantly change either reported `xstar`.

13.3.2 Modified Newton Using `wolfe.m`

The `ntw.m` routine listed on the next page differs from `ntfs.m` in only five particulars. Like `nt.m`, it includes the variable bounds `xl` and `xh` in its calling sequence [1] so that they can be passed on to the line search. It sets the Wolfe parameters [7-8] and line search iteration limit `smax` [9] as in `sdw.m`, and it uses the same approach as in `sdw.m` [31] to make the line search tolerance get smaller as the optimal point is approached. It tests [33] the return code from `wolfe.m` and gives up if the line search failed. Finally, instead of taking a full step the new point is found [34] as `xk+astar*dk`.

```
octave:1> format long
octave:2> xzero=[2;2];
octave:3> xl=[-2;-2];
octave:4> xh=[3;3];
octave:5> [xstar,kp,nm,rc]=ntw(xzero,xl,xh,100,1e-14,@gns,@gnsg,@gnsh,0.5)
xstar =

   0.750000000000000
  -0.750000000000000

kp =  2
nm = 0
rc = 0
octave:6> quit
```

The output shown above and on the next page demonstrates that `ntw.m` can solve the `gns` and `rb` problems exactly.


```
function [xstar,kp,nm,rc]=ntw(xzero,xl,xh,kmax,epz,fcn,grd,hsn,gama)
% modified Newton using Wolfe line search
  n=size(xzero,1);                 % get number of variables
  xk=xzero;                        % set starting point
  nm=0;                            % no modifications yet
  rc=0;                            % assume it will converge
  mu=0.0001;                       % Wolfe sufficient decrease
  eta=0.4;                         % Wolfe curvature
  smax=52;                         % line search iteration limit
  for kp=1:kmax                    % do up to kmax descents
      g=grd(xk);                   % find uphill direction
      if(norm(g) <= epz)           % is xk stationary?
         xstar=xk;                   % yes; declare xk optimal
         rc=0;                       % flag convergence
         return                      % and return
      end                          % no; continue iterations
      H=hsn(xk);                   % get current Hessian matrix
      [U,p]=chol(H);               % try to factor it
      while(p ~= 0)                % does H need modification?
        if(gama >= 1 || gama < 0)  % yes; can it be modified?
           xstar=xk;                 % no; gama value prevents that
           rc=2;                     % flag nonconvergence
           return                    % and return
        end                        % yes; modification possible
        H=gama*H+(1-gama)*eye(n);    % average with identity
        [U,p]=chol(H);               % and try again
        nm=nm+1;                     % count the modification
      end                          % now Hd=U'Ud=-g
      y=U'\(-g);                   % solve U'y=-g for y
      dk=U\y;                      % solve Ud=y for d
      tol=1000*epz*norm(g);        % adapt line search tolerance
      [astar,rcw,kw]=wolfe(xk,dk,xl,xh,n,fcn,grd,mu,eta,tol,smax);
      if(rcw > 3) break; end       % resign if line search failed
      xk=xk+astar*dk;              % take the optimal step
  end                              % and continue
  xstar=xk;                        % out of iterations
  rc=1;                            % so no convergence yet
end
```

```
octave:1> format long
octave:2> xl=[-2;-1];
octave:3> xh=[2;2];
octave:4> [xstar,kp,nm,rc]=ntw([0;-0.5],xl,xh,100,1e-16,@rb,@rbg,@rbh,0.5)
xstar =

   1
   1

kp =  16
nm = 0
rc = 0
octave:5> [xstar,kp,nm,rc]=ntw([0;0.5],xl,xh,100,1e-16,@rb,@rbg,@rbh,0.5)
xstar =

   1
   1

kp =  14
nm =  8
rc = 0
octave:6> quit
```


13.4 Quasi-Newton Algorithms

The modified Newton algorithm has superlinear convergence and works from any starting point, but it uses a lot of CPU time. Computing $\mathbf{H}(\mathbf{x}^k)$ might involve evaluating complicated expressions for the matrix elements, and once they are all known factoring the result takes on the order of n^3 additional arithmetic operations. The time required to solve a problem also includes the labor of deriving a formula for the Hessian, which can be a tedious and tricky process even with the help of a computer algebra system such as Maple.

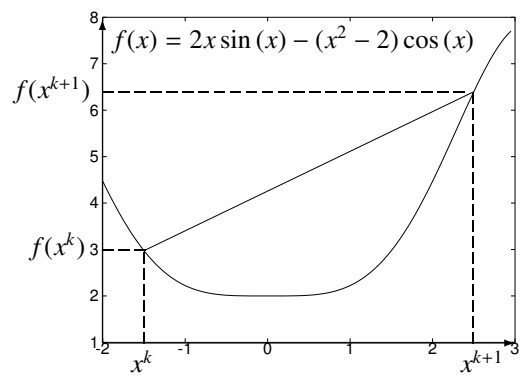
These drawbacks motivated a search for ways to *approximate* $\mathbf{H}(\mathbf{x})$ by using function and gradient values that have to be computed anyway in performing Newton descent. Several such approximations have been discovered that take only on the order of n^2 arithmetic operations, and the **variable metric** or **quasi-Newton algorithms** that use them still have superlinear convergence. The DFP and BFGS methods that we will take up in this Section were a "dramatic advance" that "transformed nonlinear optimization overnight" [5, p135-136], and they have played an important role in practical optimization ever since.

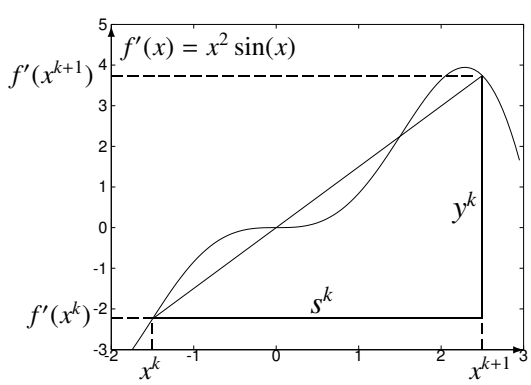
13.4.1 The Secant Equation

If $f(x)$ is a function of one variable and we know its value at two distinct points x^k and x^{k+1}, we can approximate its first derivative at x^{k+1} as

$$f'(x^{k+1}) \approx \frac{f(x^{k+1}) - f(x^k)}{x^{k+1} - x^k}.$$

If the points happen to be far apart, as in the example shown to the right, this approximation might not be very accurate.





Similarly, we can approximate the second derivative of $f(x)$ at x^{k+1} as

$$f''(x^{k+1}) \approx \frac{f'(x^{k+1}) - f'(x^k)}{x^{k+1} - x^k}.$$

In other words,

$$f''(x^{k+1})(x^{k+1} - x^k) \approx f'(x^{k+1}) - f'(x^k)$$

or, letting $s^k = x^{k+1} - x^k$ and $y^k = f'(x^{k+1}) - f'(x^k)$,

$$f''(x^{k+1})s^k \approx y^k.$$

Here the slope of the chord that approximates $f'(x)$ between x^k and x^{k+1} is y^k/s^k.

If $\mathbf{x} \in \mathbb{R}^n$, this way of approximating the second derivative of $f(\mathbf{x})$ yields

$$\mathbf{H}(\mathbf{x}^{k+1})\mathbf{s}^k \approx \mathbf{y}^k.$$

where now $\mathbf{s}^k = \mathbf{x}^{k+1} - \mathbf{x}^k$ and $\mathbf{y}^k = \nabla f(\mathbf{x}^{k+1}) - \nabla f(\mathbf{x}^k)$ are vectors. If we can find a matrix $\mathbf{B}^{k+1}$ such that

$$\mathbf{B}^{k+1}\mathbf{s}^k = \mathbf{y}^k$$

exactly, then $\mathbf{B}^{k+1}$ will approximate $\mathbf{H}(\mathbf{x}^{k+1})$. This is called the **secant equation**.

We assume that the Hessian is symmetric ($f(\mathbf{x})$ has continuous second partials) so we want $\mathbf{B}^{k+1}$ to be symmetric as well, and that means it has $\frac{1}{2}n(n+1)$ different elements. To determine them uniquely we would need that many independent conditions. Given values for $\mathbf{s}^k$ and $\mathbf{y}^k$, the secant equation $\mathbf{B}^{k+1}\mathbf{s}^k = \mathbf{y}^k$ is a linear system of n scalar equations in the elements of $\mathbf{B}^{k+1}$, so it provides n conditions. We want the Hessian to be positive definite to ensure the descent directions we find actually go downhill, so $\mathbf{B}^{k+1}$ should be positive definite too. That means its leading principal minors are all positive and there are n of those, so we have another n conditions. The table below summarizes, for several values of n, the number $2n$ of conditions that must be satisfied and the number $\frac{1}{2}n(n+1)$ of $\mathbf{B}^{k+1}$ elements to be determined.

n	$2n$	$\frac{1}{2}n(n+1)$	
1	2	1	not enough elements to ensure $\mathbf{B}^{k+1}$ meets all of the conditions
2	4	3	
3	6	6	
4	8	10	not enough conditions to determine $\mathbf{B}^{k+1}$ uniquely
$\vdots$	$\vdots$	$\vdots$	

For $n > 3$ there are many possible choices for $\mathbf{B}^{k+1}$.

13.4.2 Iterative Approximation of the Hessian

A quasi-Newton method starts with a matrix $\mathbf{B}^0$, typically set to $\mathbf{I}$ (which yields a steepest-descent step), and then applies an **update formula** involving function and gradient values to transform each $\mathbf{B}^k$ into $\mathbf{B}^{k+1}$. From a $\mathbf{B}^k$ that is symmetric and positive definite, the update formula must produce a $\mathbf{B}^{k+1}$ having these three properties:

- it is also symmetric, like $\mathbf{H}(\mathbf{x})$;
- it is also positive definite, so that $\mathbf{d} = -\mathbf{B}^{-1}\nabla f(\mathbf{x}^k)$ is a descent direction;
- it satisfies the secant equation so that it approximates $\mathbf{H}(\mathbf{x})$.

Many different update formulas can produce a $\mathbf{B}^{k+1}$ having these properties [5, §6.3]. The first effective one was found by Davidon, Fletcher, and Powell [40] and it is therefore known as the **DFP algorithm**; the one most often used today was found by Broyden, Fletcher, Goldfarb, and Shanno [131, p53-72] and is therefore known as the **BFGS algorithm**. If we let

$$\rho_k = \frac{1}{\mathbf{y}^{k\top}\mathbf{s}^k}$$

then the two updates can be written as follows.

$$\begin{aligned} \text{DFP:} \qquad \mathbf{B}^{k+1} &= (\mathbf{I} - \rho_k \mathbf{y}^k \mathbf{s}^{k\top})\mathbf{B}^k(\mathbf{I} - \rho_k \mathbf{s}^k \mathbf{y}^{k\top}) + \rho_k \mathbf{y}^k \mathbf{y}^{k\top} \\ \text{BFGS:} \qquad \mathbf{B}^{k+1} &= \mathbf{B}^k - \frac{\mathbf{B}^k \mathbf{s}^k \mathbf{s}^{k\top} \mathbf{B}^k}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \frac{\mathbf{y}^k \mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k} \end{aligned}$$

These formulas involve both inner and outer products of vectors. The inner or dot product yields a scalar while the outer product yields a matrix. For example, if

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \text{and} \qquad \mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

then

$$\mathbf{u}^\top \mathbf{v} = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = 1 \times 3 + 2 \times 4 = 11$$

but

$$\mathbf{u}\mathbf{v}^\top = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 \times 3 & 1 \times 4 \\ 2 \times 3 & 2 \times 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}.$$

Remembering this, it is easy to confirm that all of the indicated products are conformable and that the denominators in the BFGS update are scalars.

Either update produces (as we shall prove in §13.4.3) a $\mathbf{B}^{k+1}$ that is symmetric and satisfies the secant equation, and that is positive definite if the Wolfe curvature condition is satisfied. Recall from §12.3.1 that the Wolfe curvature condition requires

$$|\nabla f(\mathbf{x}^k + \alpha \mathbf{d}^k)^\top \mathbf{d}^k| \le \eta |\nabla f(\mathbf{x}^k)^\top \mathbf{d}^k|$$

If $\mathbf{d}^k$ is a descent direction then $|\nabla f(\mathbf{x}^k)^\top \mathbf{d}^k| = -\nabla f(\mathbf{x}^k)^\top \mathbf{d}^k > 0$. If also $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha \mathbf{d}^k$ then, because $\eta < 1$, we have

$$|\nabla f(\mathbf{x}^{k+1})^\top \mathbf{d}^k| < -\nabla f(\mathbf{x}^k)^\top \mathbf{d}$$

But if that is true then both of these inequalities must hold.

$$\begin{aligned} \nabla f(\mathbf{x}^{k+1})^\top \mathbf{d}^k &< -\nabla f(\mathbf{x}^k)^\top \mathbf{d}^k \\ -\nabla f(\mathbf{x}^{k+1})^\top \mathbf{d}^k &< -\nabla f(\mathbf{x}^k)^\top \mathbf{d}^k \end{aligned}$$

Rearranging the last inequality, we get

$$\begin{aligned}\nabla f(\mathbf{x}^{k+1})^\top \mathbf{d}^k - \nabla f(\mathbf{x}^k)^\top \mathbf{d}^k &> 0\\ [\nabla f(\mathbf{x}^{k+1}) - \nabla f(\mathbf{x}^k)]^\top \mathbf{d}^k &> 0\end{aligned}$$

But $[\nabla f(\mathbf{x}^{k+1}) - \nabla f(\mathbf{x}^k)] = \mathbf{y}^k$ and $\mathbf{d}^k = (\mathbf{x}^{k+1} - \mathbf{x}^k)/\alpha = \mathbf{s}^k/\alpha$, so $\mathbf{y}^{k\top}\mathbf{s}^k/\alpha > 0$. Because $\alpha > 0$, $\mathbf{y}^{k\top}\mathbf{s}^k > 0$. Thus, if the Wolfe curvature condition is satisfied then $\mathbf{s}^{k\top}\mathbf{y}^k > 0$, and this is the characterization that we will use in the next Section.

The DFP and BFGS updates require a lot of arithmetic but they still might be cheaper than finding $\mathbf{H}(\mathbf{x}^k)$, and because they are guaranteed to produce a positive-definite result if $\mathbf{s}^{k\top}\mathbf{y}^k > 0$ we need never modify $\mathbf{B}$ to ensure that $\mathbf{d} = -\mathbf{B}^{-1}\nabla f(\mathbf{x}^k)$ is a descent direction.

13.4.3 The BFGS Update Formula

Extraordinary claims demand compelling evidence, so four theorems [53] are proved below to establish that the BFGS update really does produce a matrix $\mathbf{B}^{k+1}$ having the properties listed in §13.4.2. Similar results can be obtained for the DFP update (see Exercise 13.5.27).

Theorem: the BFGS update maintains symmetry of $\mathbf{B}$.
if $\mathbf{B}^k$ is symmetric and

$$\mathbf{B}^{k+1} = \mathbf{B}^k - \frac{\mathbf{B}^k\mathbf{s}^k\mathbf{s}^{k\top}\mathbf{B}^k}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \frac{\mathbf{y}^k\mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k}$$

then $\mathbf{B}^{k+1}$ is symmetric

Proof:
The transpose of a scalar is the scalar, and the transpose of a product is the product of the transposes in opposite order, so

$$[\mathbf{B}^{k+1}]^\top = [\mathbf{B}^k]^\top - \frac{[(\mathbf{B}^k\mathbf{s}^k)(\mathbf{B}^{k\top}\mathbf{s}^k)^\top]^\top}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \frac{[\mathbf{y}^k\mathbf{y}^{k\top}]^\top}{\mathbf{y}^{k\top}\mathbf{s}^k}$$

$\mathbf{B}^k$ is symmetric by assumption, and the transpose of a product is the product of the transposes in opposite order, so

$$[\mathbf{B}^{k+1}]^\top = \mathbf{B}^k - \frac{(\mathbf{B}^{k\top}\mathbf{s}^k)(\mathbf{B}^k\mathbf{s}^k)^\top}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \frac{\mathbf{y}^k\mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k}$$

The transpose of a product is the product of the transposes in opposite order, so

$$[\mathbf{B}^{k+1}]^\top = \mathbf{B}^k - \frac{\mathbf{B}^{k\top}\mathbf{s}^k\mathbf{s}^{k\top}\mathbf{B}^k}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \frac{\mathbf{y}^k\mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k}$$

This is the formula for $\mathbf{B}^{k+1}$ so $[\mathbf{B}^{k+1}]^\top = \mathbf{B}^{k+1}$ and $\mathbf{B}^{k+1}$ is symmetric as was to be shown. □

Theorem: the BFGS result satisfies the secant equation.

if $\quad \mathbf{B}^{k+1} = \mathbf{B}^k - \dfrac{\mathbf{B}^k\mathbf{s}^k\mathbf{s}^{k\top}\mathbf{B}^k}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \dfrac{\mathbf{y}^k\mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k}$

then $\quad \mathbf{B}^{k+1}\mathbf{s}^k = \mathbf{y}^k$

Proof:
Using the update formula we compute

$$\mathbf{B}^{k+1}\mathbf{s}^k = \mathbf{B}^k\mathbf{s}^k - \frac{\mathbf{B}^k\mathbf{s}^k(\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k)}{(\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k)} + \frac{\mathbf{y}^k(\mathbf{y}^{k\top}\mathbf{s}^k)}{(\mathbf{y}^{k\top}\mathbf{s}^k)}$$

Each of the quantities in parentheses is a scalar, so in each fraction the parenthesized quantity in the numerator cancels out with the one in the denominator and we are left with

$$\mathbf{B}^{k+1}\mathbf{s}^k = \mathbf{B}^k\mathbf{s}^k - \mathbf{B}^k\mathbf{s}^k + \mathbf{y}^k.$$

Thus $\mathbf{B}^{k+1}\mathbf{s}^k = \mathbf{y}^k$ as was to be shown. □

In proving that the BFGS update preserves the positive-definiteness of $\mathbf{B}$ (on the next two pages) it will be convenient to use the following general result.

Theorem: Let $\mathbf{U}$ and $\mathbf{M}$ be square matrices the same size, with $\mathbf{U}$ upper triangular and nonsingular. Then $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is positive definite if and only if $\mathbf{M}$ is positive definite.

Proof:
First suppose that $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is positive definite. We assumed $\mathbf{U}$ is nonsingular, so we can let $\mathbf{w} = \mathbf{U}^{-1}\mathbf{z}$ so that $\mathbf{z} = \mathbf{U}\mathbf{w}$. Then $\mathbf{z}^\top\mathbf{M}\mathbf{z} = (\mathbf{U}\mathbf{w})^\top\mathbf{M}(\mathbf{U}\mathbf{w}) = \mathbf{w}^\top(\mathbf{U}^\top\mathbf{M}\mathbf{U})\mathbf{w}$. But we assumed that $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is positive definite, so $\mathbf{w}^\top(\mathbf{U}^\top\mathbf{M}\mathbf{U})\mathbf{w} > 0$ for all $\mathbf{w} \neq \mathbf{0}$. Because $\mathbf{U}$ is upper triangular, $\mathbf{z} = \mathbf{0} \Leftrightarrow \mathbf{w} = \mathbf{0}$ so $\mathbf{w} \neq \mathbf{0} \Rightarrow \mathbf{z} \neq \mathbf{0}$. Thus $\mathbf{z}^\top\mathbf{M}\mathbf{z} > 0$ for all $\mathbf{z} \neq \mathbf{0}$, so $\mathbf{M}$ is positive definite.

Next suppose that $\mathbf{M}$ is positive definite. We assumed $\mathbf{U}$ is nonsingular, so we can let $\mathbf{w} = \mathbf{U}\mathbf{v}$. Then $\mathbf{w}^\top\mathbf{M}\mathbf{w} = (\mathbf{U}\mathbf{v})^\top\mathbf{M}(\mathbf{U}\mathbf{v}) = \mathbf{v}^\top(\mathbf{U}^\top\mathbf{M}\mathbf{U})\mathbf{v}$. But we assumed that $\mathbf{M}$ is positive definite, so $\mathbf{w}^\top\mathbf{M}\mathbf{w} > 0$ for all $\mathbf{w} \neq \mathbf{0}$. Because $\mathbf{U}$ is upper triangular, $\mathbf{v} = \mathbf{0} \Leftrightarrow \mathbf{w} = \mathbf{0}$ so $\mathbf{w} \neq \mathbf{0} \Rightarrow \mathbf{v} \neq \mathbf{0}$. Thus $\mathbf{v}^\top(\mathbf{U}^\top\mathbf{M}\mathbf{U})\mathbf{v} > 0$ for all $\mathbf{v} \neq \mathbf{0}$, so $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is positive definite.

In summary, if $\mathbf{U}$ is nonsingular then

$$\begin{aligned} \mathbf{U}^\top\mathbf{M}\mathbf{U} \text{ positive definite} &\Rightarrow \mathbf{M} \text{ is positive definite} \\ \mathbf{M} \text{ positive definite} &\Rightarrow \mathbf{U}^\top\mathbf{M}\mathbf{U} \text{ is positive definite} \end{aligned}$$

In other words, $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is positive definite $\Leftrightarrow$ $\mathbf{M}$ is positive definite, as was to be shown. □

Theorem: the BFGS update maintains positive definiteness of $\mathbf{B}$.

if $\mathbf{B}^k$ is positive definite and

the Wolfe curvature condition is satisfied so $\mathbf{s}^{k\top}\mathbf{y}^k > 0$ and

$$\mathbf{B}^{k+1} = \mathbf{B}^k - \frac{\mathbf{B}^k\mathbf{s}^k\mathbf{s}^{k\top}\mathbf{B}^k}{\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k} + \frac{\mathbf{y}^k\mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k}$$

then $\mathbf{B}^{k+1}$ is positive definite

Proof:
Write $\mathbf{B}^k = \mathbf{U}^\top\mathbf{U}$, its Cholesky factorization [150, §23]. This is possible because $\mathbf{B}^k$ is positive definite. Substituting, the update formula becomes

$$\mathbf{B}^{k+1} = \mathbf{U}^\top\mathbf{U} - \frac{\mathbf{U}^\top\mathbf{U}\mathbf{s}^k\mathbf{s}^{k\top}\mathbf{U}^\top\mathbf{U}}{\mathbf{s}^{k\top}\mathbf{U}^\top\mathbf{U}\mathbf{s}^k} + \frac{\mathbf{y}^k\mathbf{y}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k}.$$

Now let

$$\begin{array}{rclcrcl} \bar{\mathbf{y}} & = & \mathbf{U}^{-\top}\mathbf{y}^k & \text{so} & \mathbf{y}^k & = & \mathbf{U}^\top\bar{\mathbf{y}} \\ \bar{\mathbf{s}} & = & \mathbf{U}\mathbf{s}^k & \text{so} & \mathbf{s}^k & = & \mathbf{U}^{-1}\bar{\mathbf{s}}. \end{array}$$

The triangular factor $\mathbf{U}$ has positive elements on its diagonal, so it is nonsingular. The notation $\mathbf{U}^{-\top}$ denotes the transpose of the inverse of $\mathbf{U}$. Substituting, we get

$$\begin{aligned} \mathbf{B}^{k+1} &= \mathbf{U}^\top\mathbf{U} - \mathbf{U}^\top\left(\frac{(\mathbf{U}\mathbf{s}^k)(\mathbf{U}\mathbf{s}^k)^\top}{(\mathbf{U}\mathbf{s}^k)^\top(\mathbf{U}\mathbf{s}^k)}\right)\mathbf{U} + \frac{\mathbf{U}^\top\bar{\mathbf{y}}\bar{\mathbf{y}}^\top\mathbf{U}}{\bar{\mathbf{y}}^\top\mathbf{U}\mathbf{U}^{-1}\bar{\mathbf{s}}} \\ &= \mathbf{U}^\top\mathbf{U} - \mathbf{U}^\top\left(\frac{\bar{\mathbf{s}}\bar{\mathbf{s}}^\top}{\bar{\mathbf{s}}^\top\bar{\mathbf{s}}}\right)\mathbf{U} + \mathbf{U}^\top\left(\frac{\bar{\mathbf{y}}\bar{\mathbf{y}}^\top}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}}\right)\mathbf{U} \\ &= \mathbf{U}^\top\left(\mathbf{I} - \frac{\bar{\mathbf{s}}\bar{\mathbf{s}}^\top}{\bar{\mathbf{s}}^\top\bar{\mathbf{s}}} + \frac{\bar{\mathbf{y}}\bar{\mathbf{y}}^\top}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}}\right)\mathbf{U}. \end{aligned}$$

Because $\mathbf{U}$ is nonsingular, $\mathbf{U}^\top\mathbf{M}\mathbf{U}$ is positive definite if and only if the matrix $\mathbf{M}$ is positive definite. Thus, to show that $\mathbf{B}^{k+1}$ is positive definite we need to show that

$$\mathbf{M} = \left(\mathbf{I} - \frac{\bar{\mathbf{s}}\bar{\mathbf{s}}^\top}{\bar{\mathbf{s}}^\top\bar{\mathbf{s}}} + \frac{\bar{\mathbf{y}}\bar{\mathbf{y}}^\top}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}}\right)$$

is positive definite, or $\mathbf{z}^\top\mathbf{M}\mathbf{z} > 0$ for all vectors $\mathbf{z} \neq \mathbf{0}$. The remainder of the argument is devoted to establishing that fact.

We found that

$$\mathbf{M} = \left(\mathbf{I} - \frac{\bar{\mathbf{s}}\bar{\mathbf{s}}^\top}{\bar{\mathbf{s}}^\top\bar{\mathbf{s}}} + \frac{\bar{\mathbf{y}}\bar{\mathbf{y}}^\top}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}}\right)$$

so

$$\begin{aligned}
\mathbf{z}^\top\mathbf{M}\mathbf{z} &= \mathbf{z}^\top\mathbf{z} - \frac{(\mathbf{z}^\top\bar{\mathbf{s}})(\bar{\mathbf{s}}^\top\mathbf{z})}{\bar{\mathbf{s}}^\top\bar{\mathbf{s}}} + \frac{(\mathbf{z}^\top\bar{\mathbf{y}})(\bar{\mathbf{y}}^\top\mathbf{z})}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}} \\
&= \mathbf{z}^\top\mathbf{z} - \frac{(\bar{\mathbf{s}}^\top\mathbf{z})^2}{\bar{\mathbf{s}}^\top\bar{\mathbf{s}}} + \frac{(\bar{\mathbf{y}}^\top\mathbf{z})^2}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}}.
\end{aligned}$$

Now suppose that the angle between $\bar{\mathbf{s}}$ and $\mathbf{z}$ is θ. Then

$$\begin{aligned}
\bar{\mathbf{s}}^\top\mathbf{z} &= \|\bar{\mathbf{s}}\|\,\|\mathbf{z}\|\,\cos\theta \\
(\bar{\mathbf{s}}^\top\mathbf{z})^2 &= (\|\bar{\mathbf{s}}\|\,\|\mathbf{z}\|\,\cos\theta)^2
\end{aligned}$$

If the angle between $\bar{\mathbf{y}}$ and $\mathbf{z}$ is ϕ then we find similarly that

$$\begin{aligned}
\bar{\mathbf{y}}^\top\mathbf{z} &= \|\bar{\mathbf{y}}\|\,\|\mathbf{z}\|\,\cos\phi \\
(\bar{\mathbf{y}}^\top\mathbf{z})^2 &= (\|\bar{\mathbf{y}}\|\,\|\mathbf{z}\|\,\cos\phi)^2
\end{aligned}$$

Then

$$\begin{aligned}
\mathbf{z}^\top\mathbf{M}\mathbf{z} &= \|\mathbf{z}\|^2 - \frac{\|\bar{\mathbf{s}}\|^2\|\mathbf{z}\|^2\cos^2\theta}{\|\bar{\mathbf{s}}\|^2} + \frac{\|\bar{\mathbf{y}}\|^2\|\mathbf{z}\|^2\cos^2\phi}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}} \\
\mathbf{z}^\top\mathbf{M}\mathbf{z} &= \|\mathbf{z}\|^2(1-\cos^2\theta) + \|\mathbf{z}\|^2\left(\frac{\|\bar{\mathbf{y}}\|^2\cos^2\phi}{\bar{\mathbf{y}}^\top\bar{\mathbf{s}}}\right)
\end{aligned}$$

But

$$\bar{\mathbf{y}}^\top\bar{\mathbf{s}} = (\mathbf{U}^{-\top}\mathbf{y}^k)^\top(\mathbf{U}\mathbf{s}^k) = \mathbf{y}^{k\top}\mathbf{U}^{-1}\mathbf{U}\mathbf{s}^k = \mathbf{y}^{k\top}\mathbf{s}^k > 0$$

because the Wolfe curvature condition is satisfied. Thus $\mathbf{z}^\top\mathbf{M}\mathbf{z} \geq 0$ for all vectors $\mathbf{z} \neq \mathbf{0}$, and it can be equal to zero only if $\cos^2\theta = 1$ ($\mathbf{z}$ and $\bar{\mathbf{s}}$ are collinear, $\mathbf{z} = \gamma\bar{\mathbf{s}}$) and $\cos^2\phi = 0$ ($\bar{\mathbf{y}}$ and $\mathbf{z}$ are orthogonal, $\bar{\mathbf{y}}^\top\mathbf{z} = 0$). To show that those things cannot both be true, suppose to the contrary that they are both true. Then we would have

$$\begin{aligned}
\mathbf{z} &= \gamma\bar{\mathbf{s}} = \gamma\mathbf{U}\mathbf{s}^k \\
\bar{\mathbf{y}}^\top\mathbf{z} &= (\mathbf{U}^{-\top}\mathbf{y}^k)^\top\mathbf{z} = (\mathbf{U}^{-\top}\mathbf{y}^k)^\top\gamma\mathbf{U}\mathbf{s}^k = \gamma\mathbf{y}^{k\top}\mathbf{U}^{-1}\mathbf{U}\mathbf{s}^k = \gamma\mathbf{y}^{k\top}\mathbf{s}^k = 0
\end{aligned}$$

But $\mathbf{y}^{k\top}\mathbf{s}^k > 0$ because the Wolfe curvature condition is satisfied, so it cannot be true that both $\cos^2\theta = 1$ and $\cos^2\phi = 0$. Thus $\mathbf{z}^\top\mathbf{M}\mathbf{z} > 0$ for all $\mathbf{z} \neq \mathbf{0}$, $\mathbf{M}$ is positive definite, and $\mathbf{B}^{k+1}$ is also positive definite as was to be shown. □

13.4.4 Updating the Inverse Matrix

Now we have a way to get superlinear convergence without computing the Hessian matrix. Unfortunately, we still need to solve the linear system

$$\mathbf{B}^k\mathbf{d}^k = -\nabla f(\mathbf{x}^k)$$

for each descent direction $\mathbf{d}^k$, and this accounts for the majority of the computational effort in each iteration. If we could somehow approximate $\mathbf{G} = \mathbf{B}^{-1} \approx \mathbf{H}^{-1}(\mathbf{x}^k)$ instead of $\mathbf{B}$, then each $\mathbf{d}^k$ could be found by this much faster matrix multiply.

$$\mathbf{d}^k = -\mathbf{G}^k\nabla f(\mathbf{x}^k)$$

That turns out to be possible, thanks to the following miraculous gift from linear algebra [5, p612]. This theorem is about a **rank-one update** to a matrix, which results from adding the outer product of two vectors (and which we will encounter again in §24).

Theorem: the Sherman-Morrison-Woodbury formula

if $\bar{\mathbf{A}} = \mathbf{A} + \mathbf{a}\mathbf{b}^\top$
$\mathbf{A}$ is nonsingular
$\bar{\mathbf{A}}$ is nonsingular

then $\bar{\mathbf{A}}^{-1} = \mathbf{A}^{-1} - \dfrac{\mathbf{A}^{-1}\mathbf{a}\mathbf{b}^\top\mathbf{A}^{-1}}{1 + \mathbf{b}^\top\mathbf{A}^{-1}\mathbf{a}}$

Using this result we can derive the following updates for $\mathbf{G}$.

$$\begin{aligned} \text{DFP:} \qquad \mathbf{G}^{k+1} &= \mathbf{G}^k - \frac{\mathbf{G}^k\mathbf{y}^k\mathbf{y}^{k\top}\mathbf{G}^k}{\mathbf{y}^{k\top}\mathbf{G}^k\mathbf{y}^k} + \frac{\mathbf{s}^k\mathbf{s}^{k\top}}{\mathbf{y}^{k\top}\mathbf{s}^k} \\ \text{BFGS:} \qquad \mathbf{G}^{k+1} &= (\mathbf{I} - \rho_k\mathbf{s}^k\mathbf{y}^{k\top})\mathbf{G}^k(\mathbf{I} - \rho_k\mathbf{y}^k\mathbf{s}^{k\top}) + \rho_k\mathbf{s}^k\mathbf{s}^{k\top} \end{aligned}$$

Each update for $\mathbf{G}$ resembles the other update for $\mathbf{B}$, revealing a deep connection between the DFP and BFGS schemes. Surprisingly, they can perform differently in practice.

13.4.5 The DFP and BFGS Algorithms

Algorithms based on the DFP and BFGS updates are more complicated than plain Newton descent because they use a Wolfe line search rather than taking a full step, but they are simpler than modified Newton descent because it is never necessary to factor $\mathbf{G}$. The MATLAB routines `dfp.m` and `bfgs.m` listed on the next page differ only in their update formulas. Their calling sequences do not include a routine to compute the Hessian; however, so that they will be serially reusable they do include an initial value `Gzero` for the Hessian-inverse approximation and return its final value in `Gstar`.


```
 1 % Davidon-Fletcher-Powell algorithm
 2
 3 function [xstar,Gstar,kp,rc]=dfp(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)
 4 n=size(xzero,1);
 5 mu=0.0001;
 6 eta=0.9;
 7 tol=0.01;
 8 smax=52;
 9 xk=xzero;
10 g=grd(xk);
11 G=Gzero;
12 for kp=1:kmax
13     dk=G*(-g);
14     [astar,rc]=wolfe(xk,dk,xl,xh,n,fcn,grd,mu,eta,tol,smax);
15     if(rc > 2) break; end
16     sk=astar*dk;
17     xk=xk+sk;
18     gnew=grd(xk);
19     yk=gnew-g;
20     g=gnew;
21     if(norm(g) <= epz) break; end
22     G=G-((G*yk)*(yk'*G))/(yk'*G*yk)+(sk*sk')/(yk'*sk);
23 end
24 xstar=xk;
25 Gstar=G;
```

```
 1 % Broyden-Fletcher-Goldfarb-Shanno algorithm
 2
 3 function [xstar,Gstar,kp,rc]=bfgs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)
 4 n=size(xzero,1);
 5 mu=0.0001;
 6 eta=0.9;
 7 tol=0.01;
 8 smax=52;
 9 xk=xzero;
10 g=grd(xk);
11 G=Gzero;
12 for kp=1:kmax
13     dk=G*(-g);
14     [astar,rc]=wolfe(xk,dk,xl,xh,n,fcn,grd,mu,eta,tol,smax);
15     if(rc > 2) break; end
16     sk=astar*dk;
17     xk=xk+sk;
18     gnew=grd(xk);
19     yk=gnew-g;
20     g=gnew;
21     if(norm(g) <= epz) break; end
22     rho=1/(yk'*sk);
23     G=(eye(n)-rho*sk*yk')*G* ...
24       (eye(n)-rho*yk*sk')+rho*sk*sk';
25 end
26 xstar=xk;
27 Gstar=G;
```

The line search can be imprecise [7] but as usual (see §12.2.3) I have allowed it `smax=52` iterations [8]. The Wolfe parameter values [5,6] are chosen deliberately [5, p142] for quasi-Newton algorithms, and we insist [15] that `astar` satisfy the Wolfe conditions; the return code from `wolfe.m` is passed back [3] so that the calling routine can determine whether that

happened. The update formulas are easy to code but lengthy; in `bfgs.m` I used the MATLAB ellipsis "..." to continue the formula 23-24 from one line to the next.

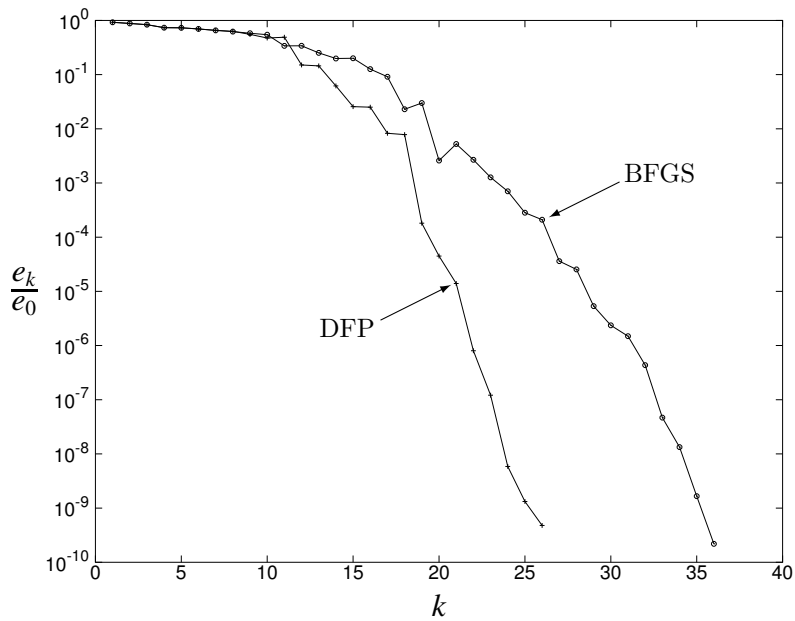
To test these routines I wrote the `tryqn.m` program listed below. It exercises `dfp.m` and `bfgs.m` on the `rb` problem and plots their error curves.

```
 1 % tryqn.m: compare DFP to BFGS on the rb problem
 2 clear; clf; set(gca,'FontSize',25)
 3
 4 xl=[-2;-1];
 5 xh=[2;2];
 6 xzero=[-1.2;1];
 7 xstar=[1;1];
 8 ezero=norm(xzero-xstar);
 9 kmax=100;
10 epz=1e-9;
11
12 % solve the problem using DFP
13 xk=xzero;
14 Gk=eye(2);
15 for kp=1:kmax
16     x=xk;
17     G=Gk;
18     [xk,Gk,kused,rc]=dfp(x,G,xl,xh,1,epz,@rb,@rbg);
19     kdfp(kp)=kp;
20     edfp(kp)=norm(xk-xstar)/ezero;
21     if(edfp(kp) < epz) break; end
22 end
23 printf('DFP:  x= %17.15f %17.15f at kp=%i3\n',xk(1),xk(2),kp)
24
25 % solve the problem using BFGS
26 xk=xzero;
27 Gk=eye(2);
28 for kp=1:kmax
29     x=xk;
30     G=Gk;
31     [xk,Gk,kused,rc]=bfgs(x,G,xl,xh,1,epz,@rb,@rbg);
32     kbfgs(kp)=kp;
33     ebfgs(kp)=norm(xk-xstar)/ezero;
34     if(ebfgs(kp) < epz) break; end
35 end
36 printf('BFGS: x= %17.15f %17.15f at kp=%i3\n',xk(1),xk(2),kp)
37
38 % plot error versus iteration for the two methods
39 hold on
40 semilogy(kdfp,edfp);
41 semilogy(kdfp,edfp,'+');
42 semilogy(kbfgs,ebfgs);
43 semilogy(kbfgs,ebfgs,'o');
44 hold off
45 print -deps -solid tryqn.eps
```

The Hessian-inverse approximation `Gk` is initialized to the identity for both `dfp.m` 14 and `bfgs.m` 27. The loops 15-22 and 28-35 invoke 18 `dfp.m` and 31 `bfgs.m` to perform one iteration at a time, so that the relative error of each iterate can be captured in `edfp` 20 and `ebfgs` 33 for plotting 40-43.

The `tryqn.m` program produces the output and error curves below.

```
octave:1> tryqn
DFP:  x= 1.000000000356373 1.000000000988120 at kp=263
BFGS: x= 0.999999999792015 0.999999999566663 at kp=363
octave:2> quit
```



Both algorithms accurately solve the `rb` problem from its catalog starting point with almost-quadratic convergence, but `dfp.m` requires significantly fewer iterations.

13.4.6 The Full BFGS Step

In §13.1.0 we found that the full-step Newton algorithm fails when the Hessian $\mathbf{H}$ is non-positive-definite at some iterate $\mathbf{x}^k$. But quasi-Newton methods approximate $\mathbf{H}^{-1}$ by a matrix $\mathbf{G}^k$ that is positive definite for all k. Why not skip the Wolfe line search and just take a full ($\alpha = 1$) step in the descent direction $\mathbf{d}^k = -\mathbf{G}^k \nabla f(\mathbf{x}^k)$?

The trouble with this idea is that to ensure each $\mathbf{B}^k$ is positive definite, so that $\mathbf{G}^k$ is positive definite and $\mathbf{d}^k$ actually is a descent direction, we found it necessary to assume in §13.4.3 that the Wolfe curvature condition is satisfied, and the full DFP or BFGS step might not do that. On the other hand, it might! In fact, as the $\mathbf{x}^k$ approach $\mathbf{x}^\star$ this becomes increasingly likely [5, p142]. In a quasi-Newton algorithm the line search can be safely avoided altogether if $\alpha = 1$ happens to satisfy the Wolfe conditions, even if that step length is far from $\alpha^\star$.

The `chkwlf.m` routine listed at the top of the next page returns `rc=0` if a proposed step length `alpha` satisfies both Wolfe conditions, `rc=1` if it violates the sufficient decrease condition, `rc=2` if it violates the curvature condition, or `rc=3` if it violates both. I used it in `bfgsfs.m`, which is listed at the bottom of the next page.


```
function [rc]=chkwlf(xk,dk,alpha,mu,eta,fcn,grd)
% check the Wolfe conditions
  gk=grd(xk);                 % gradient at current point
  dfk=gk'*dk;                 % directional derivative there
  fk=fcn(xk);                 % function value at current point
  x=xk+alpha*dk;              % proposed next point
  g=grd(x);                   % gradient there
  df=g'*dk;                   % directional derivative there
  f=fcn(x);                   % function value there
  rc=0;                       % assume both conditions satisfied
  if(f > fk+mu*dfk*alpha)     % sufficient decrease?
     rc=rc+1;                 % no; violated
  end
  if(abs(df) > eta*abs(dfk))  % curvature?
     rc=rc+2;                 % no; violated
  end
end


% Broyden-Fletcher-Goldfarb-Shanno taking a full step if possible
function [xstar,Gstar,kp,rc]=bfgsfs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)
n=size(xzero,1);
mu=0.0001;
eta=0.9;
tol=0.01;
smax=52;
xk=xzero;
g=grd(xk);
G=Gzero;
for kp=1:kmax
    dk=G*(-g);
    [rcchk]=chkwlf(xk,dk,1,mu,eta,fcn,grd);  % is a full step OK?
    if(rcchk == 0)                           % if so,
       astar=1;                              % use it
       rc=8;                                 % and tell the caller
    else
      [astar,rc]=wolfe(xk,dk,xl,xh,n,fcn,grd,mu,eta,tol,smax);
      if(rc > 2) break; end
    end
    sk=astar*dk;
    xk=xk+sk;
    gnew=grd(xk);
    yk=gnew-g;
    g=gnew;
    if(norm(g) <= epz) break; end
    rho=1/(yk'*sk);
    G=(eye(n)-rho*sk*yk')*G* ...
      (eye(n)-rho*yk*sk')+rho*sk*sk';
end
xstar=xk;
Gstar=G;
```

If the final iteration of `bfgsfs.m` uses a full step rather than a line search, it returns `rc=8` (this value differs from the return codes that can be passed back from `wolfe.m`). From the results on the next page it is clear that the full BFGS step can sometimes be taken.

It might happen that $\alpha = 1$ falls outside the line search limits $[\alpha^{\mathrm{L}}, \alpha^{\mathrm{H}}]$ determined by the variable bounds, so `bfgsfs.m`, like our other full-step routines `sdfs.m` and `ntfs.m`, can return an optimal point that is not in $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ (but see Exercises 12.5.20 and 13.5.19).


```
octave:1> format long
octave:2> Gzero=eye(2);
octave:3> xzero=[2;2];
octave:4> xl=[-2;-2];
octave:5> xh=[3;3];
octave:6> [xstar,Gstar,kp,rc]=bfgsfs(xzero,Gzero,xl,xh,100,1e-16,@gns,@gnsg)
xstar =

   0.750000000000000
  -0.750000000000000

Gstar =

   0.250000000022406  -0.249999999982929
  -0.249999999982929   0.500000000013006

kp =  4
rc =  8
octave:7> xl=[-2;-1];
octave:8> xh=[2;2];
octave:9> [xstar,Gstar,kp,rc]=bfgsfs([0;-0.5],Gzero,xl,xh,100,1e-16,@rb,@rbg)
xstar =

   1
   1

Gstar =

   0.499800081614917   0.999569749238111
   0.999569749238111   2.004081926521275

kp =  24
rc =  8
octave:10> [xstar,Gstar,kp,rc]=bfgsfs([0;0.5],Gzero,xl,xh,100,1e-16,@rb,@rbg)
xstar =

   1
   1

Gstar =

   0.499427207759372   0.998853413502508
   0.998853413502508   2.002704124873238

kp =  24
rc =  8
octave:11> quit
```

A version of `dfp.m` can be constructed that uses `chkwlf.m` and takes a full step if that satisfies the Wolfe conditions (see Exercise 13.5.32) and it will complete our set of four routines implementing quasi-Newton algorithms.

routine synopsis `[xstar,Gstar,kp,rc]=`	algorithm for $\alpha^\star$
`dfp(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)`	DFP update
`bfgs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)`	BFGS update
`dfpfs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)`	full DFP step if safe
`bfgsfs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)`	full BFGS step if safe

13.5 Exercises

13.5.1 [H] In §13.1 claimed that

$$\begin{array}{rrcl} \text{the quadratic approximation} & q(\mathbf{x}) & = & f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x}-\bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}})^\top\mathbf{H}(\bar{\mathbf{x}})(\mathbf{x}-\bar{\mathbf{x}}) \\ \text{has gradient} & \nabla q(\mathbf{x}) & = & \nabla f(\bar{\mathbf{x}}) + \mathbf{H}(\bar{\mathbf{x}})(\mathbf{x}-\bar{\mathbf{x}}). \end{array}$$

Show that this claim is true. Hint: the gradient of a constant is zero.

13.5.2 [E] Steepest descent is a simple and robust algorithm for unconstrained nonlinear optimization. What drawbacks does it have that motivate the search for better methods? How do Newton descent and its variants achieve superlinear convergence?

13.5.3 [E] When does taking one full Newton step minimize a function? When does taking one full steepest descent step minimize a function? When is a full Newton step *the same* as a full steepest descent step?

13.5.4 [H] Consider the system of linear equations $\mathbf{Hd} = -\mathbf{g}$ in which

$$\mathbf{H} = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4\frac{1}{4} & 2\frac{3}{4} \\ 1 & 2\frac{3}{4} & 3\frac{1}{2} \end{bmatrix} \qquad \text{and} \qquad \mathbf{g} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

(a) Show that $\mathbf{H} = \mathbf{U}^\top\mathbf{U}$, where

$$\mathbf{U} = \begin{bmatrix} 2 & -\frac{1}{2} & \frac{1}{2} \\ 0 & 2 & 1\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}.$$

(b) Demonstrate using hand calculations how $\mathbf{d}$ can be found by simple forward- and back-substitutions. (c) Use MATLAB to repeat parts (a) and (b) .

13.5.5 [E] Under what circumstances does plain Newton descent fail? *How* does it fail?

13.5.6 [H] In §13.1 we found two inequalities that must be satisfied if the leading principal minors of the `rb` Hessian matrix are positive, and I claimed that they are both satisfied where $x_2 < x_1^2 + \frac{1}{200}$. (a) Prove that claim. (b) Explain why `ntplain.m` fails to solve the `rb` problem from $\mathbf{x}^0 = [-1.2, 1.445]^\top$.

13.5.7 [H] If $\mathbf{H}$ is positive definite, is it sure to have an inverse? If yes, prove it; if no, provide a counterexample. If $\mathbf{H}$ has an inverse, is it sure to be positive definite? If yes, prove it; if no, provide a counterexample.

13.5.8 [H] Suppose we are minimizing a function $f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^2$, and that at a particular point $\bar{\mathbf{x}}$ its gradient vector is $\mathbf{g} = \nabla f(\bar{\mathbf{x}})$ and its Hessian matrix is $\mathbf{H}(\bar{\mathbf{x}})$. (a) Find values for the elements of the Hessian matrix that make it symmetric and nonsingular but *not* positive definite. (b) Find values for the elements of the gradient vector that make $\mathbf{d} = -\mathbf{H}^{-1}\mathbf{g}$ *not*

a descent direction. (c) What must be true of an unconstrained optimization in order for plain Newton descent to be a suitable algorithm?

13.5.9 [E] Name one important application that gives rise to a strictly convex unconstrained nonlinear program.

13.5.10 [E] Explain in words the basic idea of modified Newton descent.

13.5.11 [E] In modified Newton descent, what happens to the Hessian matrix when it becomes non-positive-definite if the weighting factor γ is (a) 0; (b) 0.5; (c) 1? (d) What does the algorithm do if the Hessian *never* becomes non-positive-definite?

13.5.12 [E] When does modified Newton descent have quadratic convergence? Can it ever have only linear convergence?

13.5.13 [P] Over a contour diagram of the `rb` problem like that in §13.2, plot the convergence trajectory of the DFP algorithm from the two starting points $\mathbf{x}^1 = [0, -\frac{1}{2}]^\top$ and $\mathbf{x}^2 = [0, +\frac{1}{2}]^\top$. Does either trajectory include an excursion far outside the frame of the picture?

13.5.14 [P] The Himmelblau 28 problem [80, p428],

$$\text{minimize } f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2,$$

has optimal points near $[0.29, 0.28]^\top$ and $[-21, -36.7]^\top$. (a) Write down two inequalities that must be satisfied at points where the Hessian matrix is positive definite. (b) Analytically characterize the region(s) where the Hessian matrix is positive definite. (c) Use the MATLAB function `plotpd.m` to show graphically where the Hessian matrix is positive definite. (d) Use `ntfs.m` to solve this problem from the starting point $\mathbf{x}^0 = [1, 1]^\top$.

13.5.15 [E] The condition number of the Hessian matrix does not affect the convergence rate of plain Newton descent. Does it have *any* effect on the behavior of the algorithm?

13.5.16 [P] In §13.2, I mentioned that bad conditioning of the Hessian might limit the precision with which $\mathbf{x}^\star$ can be determined. (a) Use MATLAB to find the condition number κ of $\mathbf{H}(\mathbf{x}^\star)$ for the `rb` problem. Recall from §10.6.2 that $\kappa = 1$ is perfect conditioning. (b) Use `format long` in MATLAB to find out how precisely `ntfs.m` can solve the `rb` problem. Is this Hessian badly enough conditioned to limit the accuracy with which you can find $\mathbf{x}^\star$?

13.5.17 [P] In §13.3.0, I claimed that using a line search in the modified Newton algorithm might result in fewer descent iterations than using the full Newton step. Using `format long` in MATLAB, compare the solutions found by `ntfs.m` to those found by `ntw.m` and `nt.m`. (a) On the `gns` and `rb` problems, do the line-search methods use fewer or more descent iterations than the full-step method to achieve roughly the same level of accuracy? (b) On the `gns` and `rb` problems, are the line-search methods capable of greater accuracy than the full-step method? (c) Name one reason unrelated to speed or accuracy why it is sometimes preferable to use `nt.m` or `ntw.m` rather than `ntfs.m`.

13.5.18 [P] For comparison with the `sd.m` routine of §12.4.1, the `nt.m` routine of §13.3.1 uses the same tolerance for the descent method and the line search. Because of this the optimal step length `astar` returned by `bls` for the `rb` problem is never quite precise enough to allow the descent method convergence test to succeed, and although accurate solutions are returned they are always accompanied by `rc=1`. (a) Find a better way of setting `tol` that enables `nt.m` to satisfy some convergence tolerance `epz` on this problem. Is your solution likely to work for *all* nonconvex problems? (b) Modify `nt.m` to receive the line search tolerance `tol` as a separate parameter. Can you find values of `epz` and `tol` that allow your code to return `rc=0` on the `rb` problem? What is the smallest value of `epz` that you can use?

13.5.19 [P] The prototypical optimization algorithm of §9.6 specifies that $\mathbf{x}^{k+1} \in [\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$, but for simplicity the `ntfs.m` routine of §13.2 and the `bfgsfs.m` routine of §13.4.6 ignore this requirement. (a) Revise `ntfs.m` to take less than the full Newton step if that is necessary in order to remain within the variable bounds. (b) Revise `bfgsfs.m` to take less than the full BFGS step if that is necessary in order to remain within the variable bounds.

13.5.20 [E] Explain in words the basic idea of quasi-Newton algorithms. Name two particular quasi-Newton algorithms.

13.5.21 [H] Quasi-Newton methods approximate Newton descent for minimizing $f(\mathbf{x})$ just as the secant method for minimizing $f(x)$ [4, §12.3] approximates Newton's method for finding a zero of $f'(x)$ when $x \in \mathbb{R}^1$. The secant method of minimization uses the approximation

$$f''(x^k) \approx \frac{f'(x^k) - f'(x^{k-1})}{x^k - x^{k-1}}.$$

of §13.4.1 in the Newton zero-finding formula (see §28.3.2)

$$x^{k+1} = x^k - \frac{f'(x^k)}{f''(x^k)}.$$

(a) Derive a formula for x^{k+1} in terms of x^k, $f'(x^k)$, x^{k-1}, and $f'(x^{k-1})$. (b) Use your recursion to minimize $f(x) = (x-1)^2$ starting from $x^0 = 10$ and $x^1 = 7$.

13.5.22 [E] When does more than one matrix $\mathbf{B}^{k+1}$ satisfy the secant equation? What other properties must $\mathbf{B}^{k+1}$ have if it is to approximate the Hessian? How do quasi-Newton methods find a suitable $\mathbf{B}^{k+1}$?

13.5.23 [E] In the BFGS update formula for $\mathbf{B}^{k+1}$, why is it important that $\mathbf{s}^{k\top}\mathbf{B}^k\mathbf{s}^k$ and $\mathbf{y}^{k\top}\mathbf{s}^k$ be scalars? Show that they are scalars.

13.5.24 [E] How can we express the Wolfe curvature condition in terms of $\mathbf{s}^k = \mathbf{x}^{k+1} - \mathbf{x}^k$ and $\mathbf{y}^k = \nabla f(\mathbf{x}^{k+1}) - \nabla f(\mathbf{x}^k)$?

13.5.25 [E] In the modified Newton algorithm, $\mathbf{H}(\mathbf{x}^k)$ begins as the Hessian at $\mathbf{x}^k$, but it might get averaged with the identity matrix. In a quasi-Newton method, is it ever necessary to modify the matrix $\mathbf{B}$ that approximates the Hessian? Explain.

13.5.26 [E] State the four theorems of §13.4.3. For each, briefly outline the argument used in the proof.

13.5.27 [H] The theorems of §13.4.3 establish that the BFGS update formula produces a matrix $\mathbf{B}^{k+1}$ having the properties listed in §13.4.2. State and prove similar theorems to establish that the matrix $\mathbf{B}^{k+1}$ produced by the DFP update formula also has those properties.

13.5.28 [E] In a quasi-Newton method, why is it useful to approximate $\mathbf{G} \approx \mathbf{H}^{-1}$ rather than $\mathbf{B} \approx \mathbf{H}$?

13.5.29 [E] Explain in words what the Sherman-Morrison-Woodbury formula allows us to compute.

13.5.30 [H] The introductory example of §13.4.1 shows that if $\mathbf{x}^{k+1}$ and $\mathbf{x}^k$ are far apart the secant approximation of the Hessian might not be very good. If a quasi-Newton method succeeds in solving a nonlinear program, however, successive iterates get closer and closer together as they converge to $\mathbf{x}^\star$, and then the approximation $\mathbf{G} \approx \mathbf{H}^{-1}$ gets better. (a) Under what circumstances does $\mathbf{G}$ approach $\mathbf{H}^{-1}$, in the sense that $\|\mathbf{G} - \mathbf{H}^{-1}\| \to 0$ as $k \to \infty$? (b) Does this happens for the `gns` problem? (c) Does it happen for the `rb` problem?

13.5.31 [E] In the DFP and BFGS algorithms, why would it be unsafe to always use a step length of $\alpha = 1$ rather than doing a line search? Why is it necessary to use a *Wolfe* line search? Can a full step *ever* be used? Explain.

13.5.32 [P] Write a MATLAB routine `dfpfs.m` that uses `chkwlf.m` to find out whether a full step satisfies the Wolfe conditions, and if so takes it rather than using the line search to find a suitable step.

13.5.33 [P] In the BFGS error curve of §13.4.5 the relative solution error can be seen to sometimes increase from one iteration to the next. (a) Modify `bfgs.m` to keep a record point and to return that instead of the current iterate `xk`. (b) Modify `dfp.m` to keep a record point and return that instead of the current iterate `xk`. Do these changes affect the appearance of the error curve?

14

Conjugate-Gradient Methods

When we used steepest descent to solve the `gns` problem in §10, we observed in the contour diagram that each step taken by the algorithm was at right angles to the previous one. Algebraically, two vectors are **orthogonal** if and only if their dot product is zero [147, §2.5]. In solving `gns` the first two steepest-descent steps are

$$\alpha_0\mathbf{d}^0 \approx \begin{bmatrix} \texttt{-2.0217} \\ \texttt{-1.5403} \end{bmatrix} \qquad \text{and} \qquad \alpha_1\mathbf{d}^1 \approx \begin{bmatrix} \texttt{0.82772} \\ \texttt{-1.0864} \end{bmatrix},$$

and their precise dot product is

$$[\alpha_0\mathbf{d}^0]^\top[\alpha_1\mathbf{d}^1] = 0$$

or

$$\mathbf{d}^{0\top}\mathbf{d}^1 = 0.$$

The fact that these vectors are related at all suggests a new way of thinking about how to choose descent directions. Rather than relying on the sort of analysis we used in §10 and §13, which was based on the Taylor's series approximation to $f(\mathbf{x})$, perhaps it would be a good idea to somehow make $\mathbf{d}^k$ depend explicitly on $\mathbf{d}^{k-1}$, $\mathbf{d}^{k-2}\ldots\mathbf{d}^0$. Zigzagging contributes to the slow convergence of steepest descent, which took 12 iterations to solve `gns` to within $\epsilon = 10^{-6}$, but making each descent direction depend on the previous ones in a more subtle way leads to an algorithm that can solve problems like `gns` exactly in no more than n iterations.

14.1 Unconstrained Quadratic Programs

A nonlinear program in which the objective is quadratic and the constraints, if any, are linear is called a **quadratic program** [5, §16.0] [1, §11.2]. The `gns` problem has the quadratic objective $4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1$ so it is a quadratic program and can be written in the form

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}}\ f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x} \quad \text{starting from} \quad \mathbf{x}^0 = [2,2]^\top$$

with

$$\mathbf{Q} = \begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}.$$

A quadratic function's symmetric $\mathbf{Q}$ matrix is also its Hessian, and this one is positive definite so the `gns` objective is strictly convex and $f(\mathbf{x})$ has a unique global minimizing point (we encountered some other strictly convex quadratic programs in §8.6 and §8.7). In principle we can minimize a strictly convex quadratic objective analytically, as shown at the top of the next page.

$$\begin{aligned} \nabla f(\mathbf{x}) = \mathbf{Q}\mathbf{x} - \mathbf{b} &= \mathbf{0} \\ \mathbf{x} &= \mathbf{Q}^{-1}\mathbf{b} \end{aligned}$$

However, as explained in §8.6.5 and §8.7.5, it is often preferable for both accuracy and speed to solve the nonlinear program numerically instead.

From any point $\mathbf{x}^k$ we can do an exact line search of a strictly convex quadratic function in any direction $\mathbf{d}^k$ by analytically solving the following one-dimensional minimization problem.

$$\underset{\alpha}{\text{minimize}} \quad f(\alpha) \equiv f(\mathbf{x}^k + \alpha\mathbf{d}^k) = \tfrac{1}{2}(\mathbf{x}^k + \alpha\mathbf{d}^k)^\top\mathbf{Q}(\mathbf{x}^k + \alpha\mathbf{d}^k) - \mathbf{b}^\top(\mathbf{x}^k + \alpha\mathbf{d}^k)$$

Setting the derivative with respect to α equal to zero,

$$\begin{aligned} \frac{df}{d\alpha} = [\mathbf{Q}(\mathbf{x}^k + \alpha\mathbf{d}^k)]^\top\mathbf{d}^k - \mathbf{b}^\top\mathbf{d}^k &= 0 \\ (\mathbf{x}^k + \alpha\mathbf{d}^k)^\top\mathbf{Q}^\top\mathbf{d}^k &= \mathbf{b}^\top\mathbf{d}^k. \end{aligned}$$

The matrix $\mathbf{Q}$ is symmetric, so

$$\alpha\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k = \mathbf{b}^\top\mathbf{d}^k - \mathbf{x}^{k\top}\mathbf{Q}\mathbf{d}^k$$

and we find that $f(\alpha)$ is minimized at

$$\alpha^\star = -\frac{[\mathbf{Q}\mathbf{x}^k - \mathbf{b}]^\top\mathbf{d}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}.$$

The contours of a strictly convex quadratic function are **ellipsoids** [149, §12.6] in $\mathbb{R}^n$ (see §24.3.1). If $\mathbf{Q}$ happens also to be diagonal then each contour is a **right ellipsoid** because its axes make right angles to the coordinate hyperplanes. In that case we can find the optimal value of each x_j by minimizing the function along the jth coordinate direction, and thereby reach $\mathbf{x}^\star$ in at most n steps [5, §5.1].

Unfortunately, even in the elite guild of functions that are quadratic and strictly convex it is rare to find one with a diagonal Hessian. The $\mathbf{Q}$ matrix of the `gns` problem is not diagonal, and the graph we drew in §10.4 shows its elliptical objective contours *tilted* with respect to the coordinate hyperplanes. Minimizing that function along the coordinate directions leaves us far from $\mathbf{x}^\star$ after $n = 2$ steps (see Exercise 14.8.11).

14.2 Conjugate Directions

Fortunately, many nondiagonal $\mathbf{Q}$ matrices can be **diagonalized**. Suppose we could find a square matrix $\mathbf{S}$, with columns $\mathbf{s}^1 \ldots \mathbf{s}^n$, such that $\mathbf{S}^\top\mathbf{Q}\mathbf{S} = \mathbf{\Delta}$ where $\mathbf{\Delta}$ is a diagonal matrix. What properties would the vectors $\mathbf{s}^1 \ldots \mathbf{s}^n$ need to have? By the rules of matrix multiplication,

$$\mathbf{\Delta}_{ij} = \mathbf{s}^{i\top}\mathbf{Q}\mathbf{s}^j.$$

The diagonal elements of $\mathbf{\Delta}$ are sure to come out positive, because if $i = j$ and $\mathbf{Q}$ is positive definite then $\mathbf{s}^{j\top}\mathbf{Q}\mathbf{s}^j > 0$ by the §10.7 definition of a positive-definite matrix.

For the off-diagonal elements of $\mathbf{\Delta}$ to be zero we need

$$\mathbf{s}^{i\top}\mathbf{Q}\mathbf{s}^j = 0 \quad \text{for all } i \neq j.$$

Nonzero vectors $\mathbf{s}^i$ and $\mathbf{s}^j$ that have this property are said to be **conjugate** with respect to $\mathbf{Q}$, or $\mathbf{Q}$–conjugate [1, §8.8.1]. The orthogonal $\mathbf{d}^k$ generated by steepest descent are thus conjugate with respect to $\mathbf{I}$. Because $\mathbf{Q}$ is positive definite and symmetric, if the vectors $\mathbf{s}^1 \ldots \mathbf{s}^n$ are $\mathbf{Q}$–conjugate then [4, Exercise 13.2.9] they are linearly independent (see §28.2) so $\mathbf{S}$ is nonsingular and we can write

$$\mathbf{Q} = \mathbf{S}^{-\top}\mathbf{\Delta}\mathbf{S}^{-1}.$$

Then our quadratic objective function becomes

$$f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top[\mathbf{S}^{-\top}\mathbf{\Delta}\mathbf{S}^{-1}]\mathbf{x} - \mathbf{b}^\top\mathbf{x} = \tfrac{1}{2}[\mathbf{x}^\top\mathbf{S}^{-\top}]\mathbf{\Delta}[\mathbf{S}^{-1}\mathbf{x}] - \mathbf{b}^\top\mathbf{x} = \tfrac{1}{2}[\mathbf{S}^{-1}\mathbf{x}]^\top\mathbf{\Delta}[\mathbf{S}^{-1}\mathbf{x}] - \mathbf{b}^\top\mathbf{x}.$$

If we let $\mathbf{w} = \mathbf{S}^{-1}\mathbf{x}$ then $\mathbf{x} = \mathbf{S}\mathbf{w}$ and $\mathbf{b}^\top\mathbf{x} = \mathbf{b}^\top\mathbf{S}\mathbf{w} = [\mathbf{S}^\top\mathbf{b}]^\top\mathbf{w}$. If we let $\mathbf{a} = \mathbf{S}^\top\mathbf{b}$ then $\mathbf{b}^\top\mathbf{x} = \mathbf{a}^\top\mathbf{w}$. Then in $\mathbf{w}$-space the objective is

$$f(\mathbf{w}) = \tfrac{1}{2}\mathbf{w}^\top\mathbf{\Delta}\mathbf{w} - \mathbf{a}^\top\mathbf{w}$$

and its Hessian matrix $\mathbf{\Delta}$ is diagonal. Now we can find $\mathbf{w}^\star$ by doing at most n exact line searches on $f(\mathbf{w})$ in the coordinate directions, as described above, and then $\mathbf{x}^\star = \mathbf{S}\mathbf{w}^\star$.

If $\mathbf{Q}$ is small it is easy to find vectors that are $\mathbf{Q}$–conjugate by using the definition. For the `gns` problem, if we arbitrarily pick $\mathbf{s}^1 = [1, 0]^\top$, then for $\mathbf{s}^2$ to be $\mathbf{Q}$–conjugate to $\mathbf{s}^1$ we need

$$\mathbf{s}^{1\top}\mathbf{Q}\mathbf{s}^2 = \begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} 8 & 4 \\ 4 & 4 \end{bmatrix}\begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = 8s_1 + 4s_2 = 0$$

so, for example, $\mathbf{s}^2 = [\frac{1}{2}, -1]^\top$ would work; conjugate directions are not unique. Then

$$\mathbf{S} = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & -1 \end{bmatrix} \qquad \mathbf{\Delta} = \mathbf{S}^\top\mathbf{Q}\mathbf{S} = \begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix} \qquad \mathbf{a} = \mathbf{S}^\top\mathbf{b} = \begin{bmatrix} 3 \\ \frac{3}{2} \end{bmatrix}$$

and the $\mathbf{w}$–space objective $f(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top\mathbf{\Delta}\mathbf{w} - \mathbf{a}^\top\mathbf{w} = 4w_1^2 + w_2^2 - 3w_1 - \frac{3}{2}w_2$ can be minimized one variable at a time like this.

$$\frac{\partial f}{\partial w_1} = 8w_1 - 3 = 0 \Rightarrow w_1^\star = \tfrac{3}{8} \qquad \frac{\partial f}{\partial w_2} = 2w_2 - \tfrac{3}{2} = 0 \Rightarrow w_2^\star = \tfrac{3}{4}$$

Then

$$\mathbf{x}^\star = \mathbf{S}\mathbf{w}^\star = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & -1 \end{bmatrix}\begin{bmatrix} \frac{3}{8} \\ \frac{3}{4} \end{bmatrix} = \begin{bmatrix} \frac{3}{4} \\ -\frac{3}{4} \end{bmatrix}.$$

Each coordinate direction $\mathbf{e}^j$ in $\mathbf{w}$–space maps to the direction $\mathbf{S}\mathbf{e}^j = \mathbf{s}^j$ in $\mathbf{x}$–space, so we could alternatively do at most n exact line searches on $f(\mathbf{x})$ in the **conjugate directions** $\mathbf{d}^k = \mathbf{s}^k$ to reach $\mathbf{x}^\star$.

To illustrate the use of conjugate directions in solving a quadratic program numerically, I wrote the MATLAB program `easy.m` listed below.

```
 1 % easy.m: solve gns exactly in only n=2 steps
 2
 3 % data for the gns problem
 4 Q=[8,4;4,4];  % matrix of the quadratic form in x-space
 5 b=[3;0];      % linear-term coefficients in x-space
 6 x=[2;2];      % starting point xzero
 7
 8 % conjugate directions
 9 s1=[1;0];     % arbitrary first conjugate direction
10 s2=[1/2;-1];  % second direction chosen Q-conjugate to s1
11
12 % minimize f(w) in the coordinate directions w1 and w2
13 S=[s1,s2];    % diagonalizing matrix
14 Delta=S'*Q*S; % matrix of objective quadratic form in w-space
15 a=S'*b;       % linear-term objective coefficients in w-space
16 w=inv(S)*x;                                % map xzero to w-space
17 d1=[1;0];                                  % w1 coordinate direction
18 alphaw1=-(Delta*w-a)'*d1/(d1'*Delta*d1); % exact line search step
19 w=w+alphaw1*d1;                            % take step in w1-direction
20 d2=[0;1];                                  % w2 coordinate direction
21 alphaw2=-(Delta*w-a)'*d2/(d2'*Delta*d2); % exact line search step
22 w=w+alphaw2*d2;                            % take step in w2-direction
23 xwstar=S*w                                 % map result back to x-space
24
25 % minimize f(x) in the conjugate directions s1 and s2
26 x=[2;2];                                   % starting point
27 alphax1=-(Q*x-b)'*s1/(s1'*Q*s1);           % exact line search step
28 x=x+alphax1*s1;                            % step in s1-direction
29 alphax2=-(Q*x-b)'*s2/(s2'*Q*s2);           % exact line search step
30 xsstar=x+alphax2*s2                        % step in s2-direction
```

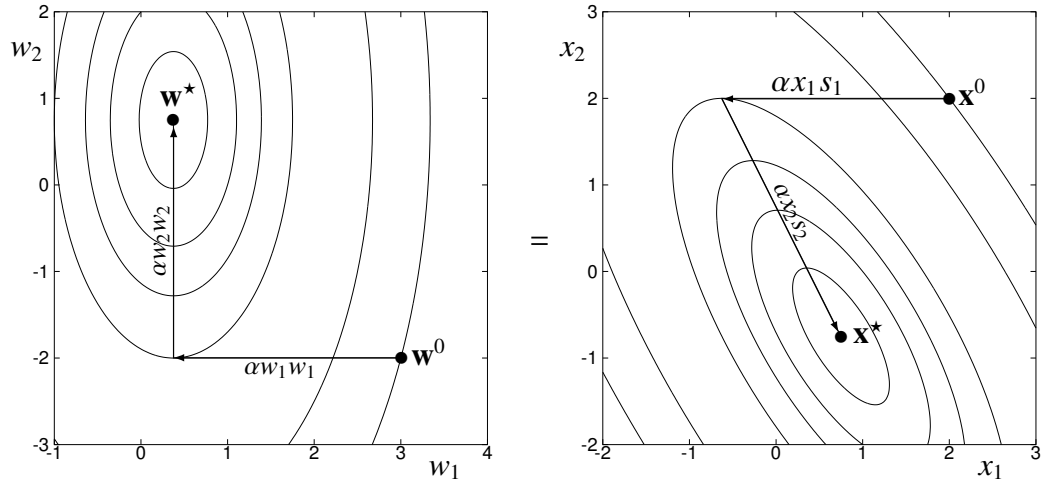
The program begins by $\boxed{\texttt{3-5}}$ fixing the values of $\mathbf{Q}$ and $\mathbf{b}$ and by $\boxed{\texttt{6}}$ initializing `x` to the starting point $\mathbf{x}^0$. This data suffices to precisely describe the `gns` problem as a quadratic program. Then $\boxed{\texttt{8-10}}$ it fixes the values of the $\mathbf{Q}$–conjugate vectors $\mathbf{s}^1$ and $\mathbf{s}^2$.

To minimize $f(\mathbf{w})$ it $\boxed{\texttt{13-15}}$ finds $\mathbf{S}$ and from it $\mathbf{\Delta}$ and $\mathbf{a}$, to define the problem in $\mathbf{w}$–space, and $\boxed{\texttt{16}}$ maps the starting `x` to $\mathbf{w}$–space. Next $\boxed{\texttt{17}}$ it sets $\mathbf{d}^1$ to the first coordinate direction, $\boxed{\texttt{18}}$ uses the formula we derived above for the optimal step $\alpha^\star$ in that direction, and $\boxed{\texttt{19}}$ updates the first component of `w` to $w_1^\star$. Then $\boxed{\texttt{20-22}}$ it repeats the process in the w_2-direction to update the second component of `w` to $w_2^\star$. Finally $\boxed{\texttt{23}}$ it transforms $\mathbf{w}^\star$ back to $\mathbf{x}$-space as `xwstar`.

A simpler way of solving the problem is to $\boxed{\texttt{25-30}}$ minimize $f(\mathbf{x})$ over the conjugate directions $\mathbf{s}^1$ and $\mathbf{s}^2$. To do that the program $\boxed{\texttt{26}}$ sets `x` to $\mathbf{x}^0$, $\boxed{\texttt{27}}$ finds the optimal step in the $\mathbf{s}^1$ direction, $\boxed{\texttt{28}}$ updates `x` in that direction, and $\boxed{\texttt{29-30}}$ repeats the process to update `x` in the $\mathbf{s}^2$ direction.

Running the program produces the output shown on the next page. Either approach finds the answer in two steps. The convergence trajectories in $\mathbf{w}$–space and $\mathbf{x}$–space are plotted (using another program) to the right of the output from `easy.m`. The steps in the $\mathbf{x}$-space picture are obviously not orthogonal; instead they are $\mathbf{Q}$–conjugate.


```
octave:1> easy
xwstar =

   0.75000
  -0.75000

xsstar =

   0.75000
  -0.75000

octave:2> quit
```



In **w**-space every step after the first is a steepest-descent step in addition to being a coordinate-direction step, but in **x**-space no step is necessarily in the steepest-descent direction (or the Newton direction).

14.3 Generating Conjugate Directions

For larger problems or to automate the process illustrated in §14.2, we need a more systematic method of finding conjugate directions. Here are some possible approaches.

- If $\mathbf{Q}$ is diagonalizable (if, for instance, the symmetric matrix has distinct eigenvalues as in [150, Theorem 24.7]) then its eigenvectors are $\mathbf{Q}$–conjugate. I will have more to say in §14.7.2 about diagonalizing $\mathbf{Q}$ by using its eigenvectors.
- The Gram-Schmidt orthogonalization procedure [87, §4.18] can be modified to generate vectors that are $\mathbf{Q}$–conjugate.
- If $\mathbf{Q}$ is positive definite and an exact line search is used, the DFP algorithm of §13.4.5 generates $\mathbf{d}^k$ that are $\mathbf{Q}$–conjugate [1, Theorem 8.8.6]. By the time we have generated them all, we have solved the nonlinear program.

All of these methods require a lot of computation, so conjugate gradient algorithms do something simpler. The idea is to generate the conjugate directions iteratively as the minimization algorithm proceeds, in the manner of DFP, but by using these easier updates [5, §5.1]

$$\begin{aligned}\mathbf{r}^k &= \mathbf{Q}\mathbf{x}^k - \mathbf{b}\\ \mathbf{d}^k &= -\mathbf{r}^k + \beta_k \mathbf{d}^{k-1}\end{aligned}$$

where β_k is chosen to make $\mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{d}^k = 0$. That this is actually possible is the first of several surprising things about conjugate gradient algorithms! We can find a formula for β_k by reasoning as shown at the top of the next page.

$$
\begin{aligned}
\mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{d}^k &= 0 \\
\mathbf{d}^{(k-1)\top}\mathbf{Q}(-\mathbf{r}^k + \beta_k\mathbf{d}^{k-1}) &= 0 \\
\mathbf{d}^{(k-1)\top}\mathbf{Q}\beta_k\mathbf{d}^{k-1} &= \mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{r}^k \\
\beta_k\mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{d}^{k-1} &= \mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{r}^k \\
\beta_k &= \frac{\mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{r}^k}{\mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{d}^{k-1}} \\
\beta_k &= \frac{\mathbf{r}^{k\top}\mathbf{Q}\mathbf{d}^{k-1}}{\mathbf{d}^{(k-1)\top}\mathbf{Q}\mathbf{d}^{k-1}}
\end{aligned}
$$

The quantities in the numerator and denominator are both scalars, so β_k is just a number. If $\mathbf{x}^k = \mathbf{x}^\star$ so that $\mathbf{Q}\mathbf{x}^k = \mathbf{b}$, the **residual** $\mathbf{r}^k$ is zero and $\beta_k = 0$.

14.4 The Conjugate Gradient Algorithm

Using the formulas for β_k and $\mathbf{r}^k$ along with results that we obtained earlier, we can construct the following algorithm for solving the quadratic program

$$\underset{\mathbf{x}}{\text{minimize}}\ f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x}$$

where $\mathbf{Q}$ is positive definite and symmetric.

$\mathbf{r}^0 = \mathbf{Q}\mathbf{x}^0 - \mathbf{b}$	residual at starting point
$\mathbf{d}^0 = -\mathbf{r}^0$	first direction is steepest descent
for $k = 0 \ldots n-1$	exactly n steps are needed
$\quad \alpha_k = -\dfrac{\mathbf{r}^{k\top}\mathbf{d}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}$	this is the optimal step length
$\quad \mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k\mathbf{d}^k$	move to the next point
$\quad \mathbf{r}^{k+1} = \mathbf{Q}\mathbf{x}^{k+1} - \mathbf{b}$	update the residual
$\quad \beta_{k+1} = \dfrac{\mathbf{r}^{(k+1)\top}\mathbf{Q}\mathbf{d}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}$	use the simple formula
$\quad \mathbf{d}^{k+1} = -\mathbf{r}^{k+1} + \beta_{k+1}\mathbf{d}^k$	to generate the next conjugate direction
end	

In deriving the formula for β_k we insisted only that $\mathbf{d}^k$ be $\mathbf{Q}$–conjugate with $\mathbf{d}^{k-1}$, but all of the $\mathbf{d}^k$ generated by this algorithm are in fact mutually $\mathbf{Q}$–conjugate [67, §10.2]. Further, $\mathbf{r}^{k\top}\mathbf{d}^p = 0$ for $p = 0 \ldots k-1$, so each residual is orthogonal to all of the previous descent directions, and $\mathbf{r}^{k\top}\mathbf{r}^p = 0$ for $p = 0 \ldots k-1$ so each residual is also orthogonal to all of the previous residuals. (Because $\mathbf{r}^k = \mathbf{Q}\mathbf{x}^k - \mathbf{b} = \nabla f(\mathbf{x}^k)$, successive gradients of the objective are

orthogonal rather than $\mathbf{Q}$–conjugate, so "conjugate gradients" is a misnomer.) Using these remarkable properties of the algorithm, we can simplify the formulas for α_k and β_k.

In the algorithm we used

$$\alpha_k = -\frac{[\mathbf{Q}\mathbf{x}^k - \mathbf{b}]^\top \mathbf{d}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k} = \frac{-\mathbf{r}^{k\top}\mathbf{d}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}.$$

The algorithm sets $\mathbf{d}^{k+1} = -\mathbf{r}^{k+1} + \beta_{k+1}\mathbf{d}^k$ so $\mathbf{d}^k = -\mathbf{r}^k + \beta_k\mathbf{d}^{k-1}$ and the numerator in the expression for α_k is

$$-\mathbf{r}^{k\top}\mathbf{d}^k = -\mathbf{r}^{k\top}(-\mathbf{r}^k + \beta_k\mathbf{d}^{k-1}) = \mathbf{r}^{k\top}\mathbf{r}^k - \beta_k\mathbf{r}^{k\top}\mathbf{d}^{k-1}.$$

Each residual is orthogonal to the previous direction, so the last term is zero. Thus,

$$\boxed{\alpha_k = \frac{\mathbf{r}^{k\top}\mathbf{r}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}.}$$

In the algorithm we used

$$\beta_{k+1} = \frac{\mathbf{r}^{(k+1)\top}\mathbf{Q}\mathbf{d}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}.$$

In this expression the term $\mathbf{Q}\mathbf{d}^k$ can be written in a different way. Notice that

$$\mathbf{r}^{k+1} - \mathbf{r}^k = (\mathbf{Q}\mathbf{x}^{k+1} - \mathbf{b}) - (\mathbf{Q}\mathbf{x}^k - \mathbf{b}) = \mathbf{Q}(\mathbf{x}^{k+1} - \mathbf{x}^k).$$

The algorithm sets $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k\mathbf{d}^k$ so $(\mathbf{x}^{k+1} - \mathbf{x}^k) = \alpha_k\mathbf{d}^k$. Thus $\mathbf{Q}(\alpha_k\mathbf{d}^k) = \mathbf{r}^{k+1} - \mathbf{r}^k$ or $\mathbf{Q}\mathbf{d}^k = (\mathbf{r}^{k+1} - \mathbf{r}^k)/\alpha_k$. Substituting in the formula for β_{k+1} we find

$$\beta_{k+1} = \frac{\mathbf{r}^{(k+1)\top}(\mathbf{r}^{k+1} - \mathbf{r}^k)/\alpha_k}{\mathbf{d}^{k\top}(\mathbf{r}^{k+1} - \mathbf{r}^k)/\alpha_k} = \frac{\mathbf{r}^{(k+1)\top}\mathbf{r}^{k+1} - \mathbf{r}^{(k+1)\top}\mathbf{r}^k}{\mathbf{d}^{k\top}\mathbf{r}^{k+1} - \mathbf{d}^{k\top}\mathbf{r}^k}.$$

Each residual is orthogonal to the previous direction, so the first term in the denominator is zero. Each residual is orthogonal to the previous residual, so the second term in the numerator is zero. Finally, for the second term in the denominator we found above that $-\mathbf{r}^{k\top}\mathbf{d}^k = \mathbf{r}^{k\top}\mathbf{r}^k$. Thus,

$$\boxed{\beta_{k+1} = \frac{\mathbf{r}^{(k+1)\top}\mathbf{r}^{k+1}}{\mathbf{r}^{k\top}\mathbf{r}^k}.}$$

In the algorithm we used $\mathbf{r}^{k+1} = \mathbf{Q}\mathbf{x}^{k+1} - \mathbf{b}$, but recently we found $\mathbf{Q}\mathbf{d}^k = (\mathbf{r}^{k+1} - \mathbf{r}^k)/\alpha_k$ so instead we could write

$$\boxed{\mathbf{r}^{k+1} = \mathbf{r}^k + \alpha_k\mathbf{Q}\mathbf{d}^k.}$$

Using the boxed expressions we can restate the algorithm given above in the following slightly more efficient way.

$$
\begin{aligned}
&\mathbf{r}^0 = \mathbf{Q}\mathbf{x}^0 - \mathbf{b}\\
&\mathbf{d}^0 = -\mathbf{r}^0\\
&\text{for}\quad k = 0 \ldots n-1\\
&\qquad \alpha_k = \frac{\mathbf{r}^{k\top}\mathbf{r}^k}{\mathbf{d}^{k\top}\mathbf{Q}\mathbf{d}^k}\\
&\qquad \mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k \mathbf{d}^k\\
&\qquad \mathbf{r}^{k+1} = \mathbf{r}^k + \alpha_k \mathbf{Q}\mathbf{d}^k\\
&\qquad \beta_{k+1} = \frac{\mathbf{r}^{(k+1)\top}\mathbf{r}^{k+1}}{\mathbf{r}^{k\top}\mathbf{r}^k}\\
&\qquad \mathbf{d}^{k+1} = -\mathbf{r}^{k+1} + \beta_{k+1}\mathbf{d}^k\\
&\text{end}
\end{aligned}
$$

In this form it is called the **conjugate gradient algorithm**. Although it can solve unconstrained strictly convex quadratic programs by finding the unique $\mathbf{x}^\star$ where

$$\nabla f(\mathbf{x}^\star) = \mathbf{Q}\mathbf{x}^\star - \mathbf{b} = \mathbf{0},$$

its most frequent use is for solving symmetric positive definite systems of linear algebraic equations $\mathbf{Q}\mathbf{x} = \mathbf{b}$ when n is large [87, §6.13]. In that case $\mathbf{Q}$ is typically also sparse [100, §11.6] and the products $\mathbf{Q}\mathbf{d}^k$ are typically found without storing the zero elements of $\mathbf{Q}$ [4, §13.2].

In perfect arithmetic, convergence is achieved by doing exactly as many iterations as $\mathbf{Q}$ has distinct eigenvalues. In practice [67, §10.2.7] rounding errors lead to a loss of orthogonality among the residuals and $\mathbf{x}^\star$ might not be found in a finite number of steps; the observed convergence of the algorithm is linear with constant

$$c \le \left(\frac{\sqrt{\kappa(\mathbf{Q})} - 1}{\sqrt{\kappa(\mathbf{Q})} + 1}\right)$$

so its actual speed depends on the condition number of $\mathbf{Q}$.

To experiment with the conjugate gradient algorithm I wrote the `cg.m` routine listed at the top of the next page. The Octave session below the listing shows that $n = 2$ iterations are enough to find an accurate solution to the `gns` problem and $n = 4$ are enough to find an accurate solution to the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where [20, Exercise 6.6.3d]

$$\mathbf{A} = \begin{bmatrix} 6 & 2 & 1 & -1 \\ 2 & 4 & 1 & 0 \\ 1 & 1 & 4 & -1 \\ -1 & 0 & -1 & 3 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$


```
 1 function [xstar,kp,beta]=cg(xzero,kmax,epz,Q,b)
 2 % minimize 1/2 x'Qx - b'x by conjugate gradients
 3   xk=xzero;
 4   rk=Q*xk-b;
 5   d=-rk;
 6   for kp=1:kmax
 7       if(norm(rk) <= epz)
 8          xstar=xk
 9          return
10       end
11       alpha=(rk'*rk)/(d'*Q*d);
12       xk=xk+alpha*d;
13       rkp=rk+alpha*Q*d;
14       beta=(rkp'*rkp)/(rk'*rk);
15       d=-rkp+beta*d;
16       rk=rkp;
17   end
18   xstar=xk;
19 end
```

Some work could be saved by computing `Q*d` once and using the result in both [11] and [13]. As $\mathbf{x} \to \mathbf{x}^\star$ the residual $\mathbf{r}^k \to \mathbf{0}$, so if the specified `kmax` is higher than needed the convergence test [7] might be necessary to avoid a 0/0 NaN (see §28.3.3) in the calculation [14] of `beta`.

```
octave:1> format long
octave:2> Q=[8,4;4,4];
octave:3> b=[3;0];
octave:4> xzero=[2;2];
octave:5> [xstar,kp,beta]=cg(xzero,2,1e-6,Q,b)
xstar =

   0.750000000000000
  -0.750000000000000

kp =  2
beta =  5.28511293092642e-31
octave:6> A=[6,2,1,-1;2,4,1,0;1,1,4,-1;-1,0,-1,3];
octave:7> b=[1;1;1;1];
octave:8> xzero=[0;0;0;0];
octave:9> [xstar,kp,beta]=cg(xzero,4,1e-6,A,b)
xstar =

   0.1675392670157068
   0.0890052356020942
   0.3089005235602095
   0.4921465968586388

kp =  4
beta =  1.05706753467554e-29
octave:10> A\b
ans =

   0.1675392670157068
   0.0890052356020942
   0.3089005235602094
   0.4921465968586387

octave:11> quit
```


14.5 The Fletcher-Reeves Algorithm

The conjugate gradient algorithm is very effective for unconstrained minimization when the objective happens to be quadratic and strictly convex, but its inner workings are intimately dependent on those luxurious and rather unusual problem characteristics. Can we somehow make use of the conjugate-directions idea in solving nonlinear programs that might be neither quadratic nor convex?

One answer to this question is the **Fletcher-Reeves algorithm**, which results from modifying the conjugate gradient algorithm to use $\nabla f(\mathbf{x}^k)$ in place of $\mathbf{r}^k$ and a Wolfe line search having $\mu > 0$ and $\eta < \frac{1}{2}$ (see §12.3.1) instead of the analytic formula for α_k. The resulting `flrv.m` routine is listed below.

```
 1 function [xstar,kp,rc]=flrv(xzero,xl,xh,kmax,epz,fcn,grd)
 2 % Fletcher-Reeves algorithm
 3   n=size(xzero,1);
 4   xk=xzero;
 5   gk=grd(xk);
 6   d=-gk;
 7   mu=0.0001;
 8   eta=0.4;
 9   smax=52;
10   for kp=1:kmax
11       if(norm(gk) <= epz)
12          xstar=xk;
13          rc=0;
14          return
15       end
16       tol=1000*epz*norm(gk);
17       [astar,rcw,kw]=wolfe(xk,d,xl,xh,n,fcn,grd,mu,eta,tol,smax);
18       if(rcw > 2) break; end
19       xk=xk+astar*d;
20       gkp=grd(xk);
21       beta=(gkp'*gkp)/(gk'*gk);
22       d=-gkp+beta*d;
23       gk=gkp;
24   end
25   xstar=xk;
26   rc=1;
27 end
```

Now instead of using simple formulas to find and update $\mathbf{r}^k$ we need to [5,20] invoke `grd`, and instead of using a simple formula to find α_k we need to [17] invoke `wolfe`, so some of the magic of conjugate gradients clearly does not survive the trip from nice special case to general nonlinear program. I used the same Wolfe parameters [7-9,16] as in `ntw.m` but interrupted the calculations [18] if a Wolfe point cannot be found. The Octave session on the next page shows that `flrv.m` solves `gns` in `kp-1=2` iterations just as `cg.m` did. It also solves `rb` from some starting points but, alas, *not* from its catalog starting point.

The Fletcher-Reeves algorithm has linear convergence and does not require storing a matrix, so it is an alternative to steepest descent. As these results show, it can be faster.


```
octave:1> format long
octave:2> xzero=[2;2];
octave:3> xl=[-2;-2];
octave:4> xh=[3;3];
octave:5> [xstar,kp,rc]=flrv(xzero,xl,xh,100,1e-16,@gns,@gnsg)
xstar =

   0.749999998604734
  -0.749999998586113

kp =  3
rc =  1
octave:6> xl=[-2;-1];
octave:7> xh=[2;2];
octave:8> xzero=[0;-0.5];
octave:9> [xstar,kp,rc]=flrv(xzero,xl,xh,100,1e-16,@rb,@rbg)
xstar =

   0.999999999999989
   0.999999999999979

kp =  62
rc =  1
octave:10> xzero=[0;0.5];
octave:11> [xstar,kp,rc]=flrv(xzero,xl,xh,100,1e-16,@rb,@rbg)
xstar =

   1.00000000000001
   1.00000000000002

kp =  54
rc =  1
octave:12> xzero=[-1.2;1];
octave:13> [xstar,kp,rc]=flrv(xzero,xl,xh,100,1e-16,@rb,@rbg)
xstar =

   1.42796766633753
   2.00000000000000

kp =  47
rc =  1
octave:14> quit
```

14.6 The Polak-Ribière Algorithm

In §14.4 we used certain remarkable properties of the conjugate gradient algorithm to simplify the formula for β. Those same properties permit other choices for β, which reduce to the conjugate gradient formula whenever an exact line search is used and $f(\mathbf{x})$ happens to be a strictly convex quadratic. The alternative that seems to perform best is the method of Polak and Ribière [130, §2.3] which uses

$$\beta_{k+1} = \frac{\mathbf{r}^{(k+1)\top}(\mathbf{r}^{k+1} - \mathbf{r}^k)}{\mathbf{r}^{k\top}\mathbf{r}^k}.$$

In the conjugate gradient algorithm $\mathbf{r}^{(k+1)\top}\mathbf{r}^k = 0$, so in the ideal case this formula for β reduces to the one we used in the Fletcher-Reeves algorithm. The scalar β_{k+1} is the amount by which $\mathbf{d}^{k+1}$ is deflected from the direction of steepest descent $-\mathbf{r}^{k+1} = -\nabla f(\mathbf{x}^{k+1})$ towards the direction $\mathbf{d}^k$ (recall that $\mathbf{d}^{k+1} = -\mathbf{r}^{k+1} + \beta_{k+1}\mathbf{d}^k$). Here the amount of deflection is reduced if successive gradients are almost the same, because that suggests the Hessian matrix might already be close to diagonal. In general this formula can result in a $\mathbf{d}^k$ that is not a descent direction [5, §5.2] so line 21 in the code below ensures that β is nonnegative.

```
 1 function [xstar,kp,rc]=plrb(xzero,xl,xh,kmax,epz,fcn,grd)
 2 % Polak-Ribiere algorithm
 3   n=size(xzero,1);
 4   xk=xzero;
 5   gk=grd(xk);
 6   d=-gk;
 7   mu=0.0001;
 8   eta=0.4;
 9   smax=52;
10   for kp=1:kmax
11       if(norm(gk) <= epz)
12          xstar=xk;
13          rc=0;
14          return
15       end
16       tol=1000*epz*norm(gk);
17       [astar,rcw,kw]=wolfe(xk,d,xl,xh,n,fcn,grd,mu,eta,tol,smax);
18       if(rcw > 2) break; end
19       xk=xk+astar*d;
20       gkp=grd(xk);
21       beta=max(0,(gkp'*(gkp-gk))/(gk'*gk));
22       d=-gkp+beta*d;
23       gk=gkp;
24   end
25   xstar=xk;
26   rc=1;
27 end
```

This routine finds $\mathbf{x}^\star$ in fewer iterations than `flrv.m` for the `gns` problem and also for the `rb` problem starting from $\mathbf{x}^0 = [0, -\frac{1}{2}]^\top$ and $\mathbf{x}^0 = [0, +\frac{1}{2}]^\top$, but unlike `flrv.m` it also solves `rb` from the catalog starting point.

```
octave:1> format long
octave:2> xl=[-2;-2];
octave:3> xh=[3;3];
octave:4> xzero=[-1.2;1];
octave:5> [xstar,kp,rc]=plrb(xzero,xl,xh,100,1e-16,@rb,@rbg)
xstar =

   0.999999999999978
   0.999999999999957

kp =  18
rc =  1
octave:6> quit
```


Polak-Ribière uses far fewer iterations than its competitor `sdfs.m` in solving this problem (see §10.6.2) though at the cost of much more complicated updates. Several other formulas for β have been proposed [5, §5.2] but the best of them are said to be only competitive with Polak-Ribière.

14.7 Quadratic Functions

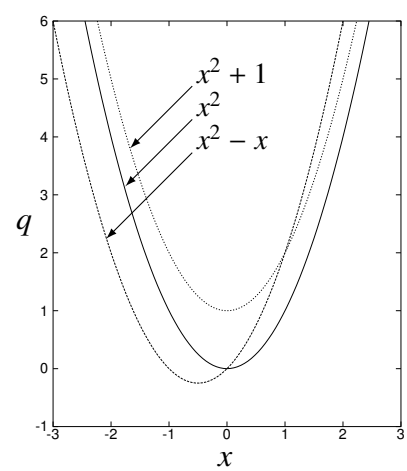
The quadratic objective of the `gns` problem is a strictly convex function because its $\mathbf{Q}$ matrix is positive definite, and as we have seen that makes its contours ellipses. In future Chapters we will encounter other quadratics whose contours are ellipses (or higher-dimensional ellipsoids) as well as quadratics that are *not* convex. To help you draw and interpret contour diagrams in two dimensions, and to help you imagine how these functions behave in higher dimensions, this Section presents a survey of quadratic forms in general and a more detailed analysis of ellipses in particular.

14.7.1 Quadratic Forms in $\mathbb{R}^2$

Any quadratic function of $\mathbf{x} \in \mathbb{R}^n$ can be represented as

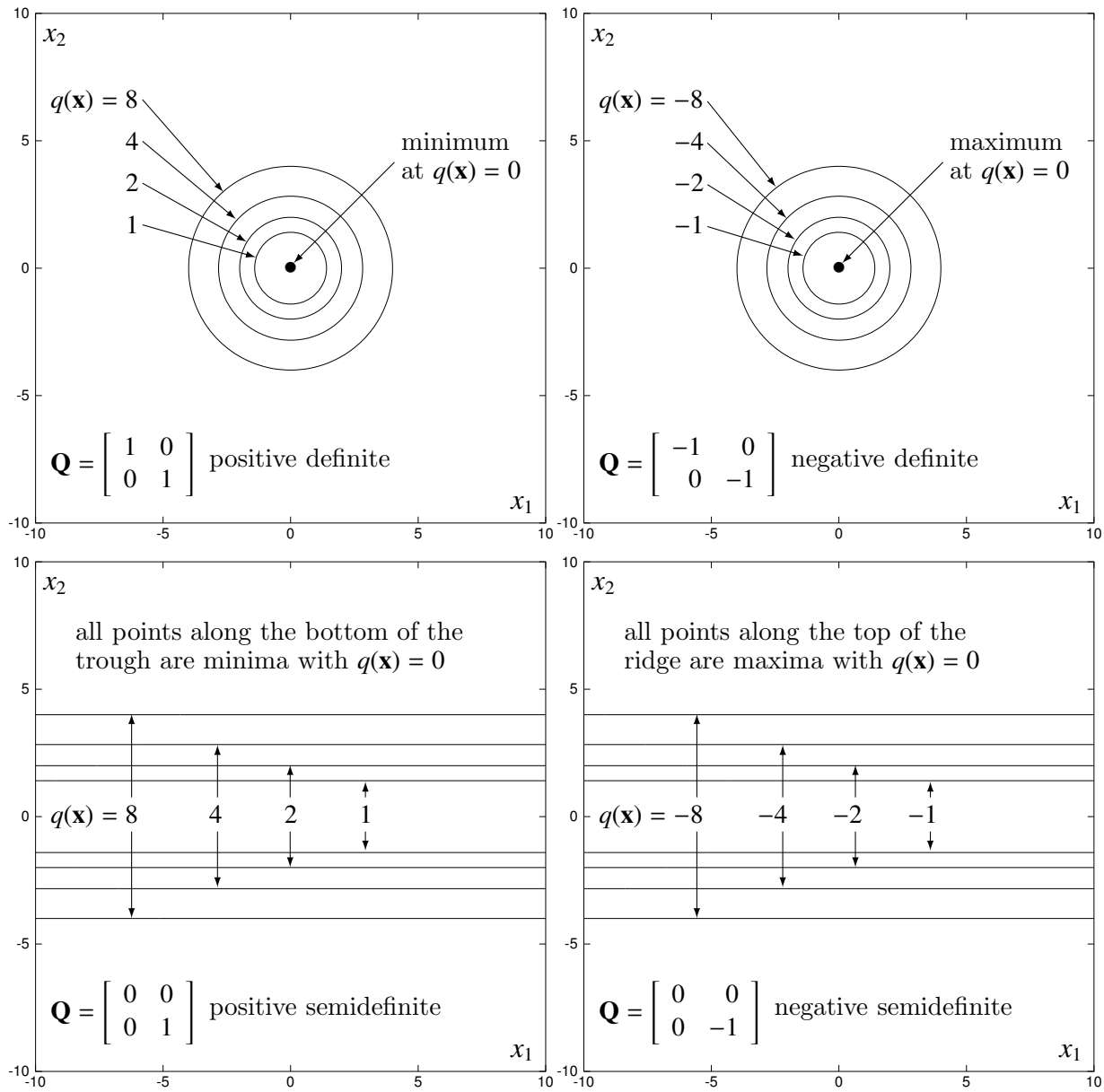
$$q(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x} + d$$

where the $n \times n$ matrix $\mathbf{Q}$ is symmetric but otherwise arbitrary, $\mathbf{c}$ is an $n \times 1$ vector, and d is a scalar (this is the notation I will use in §22).



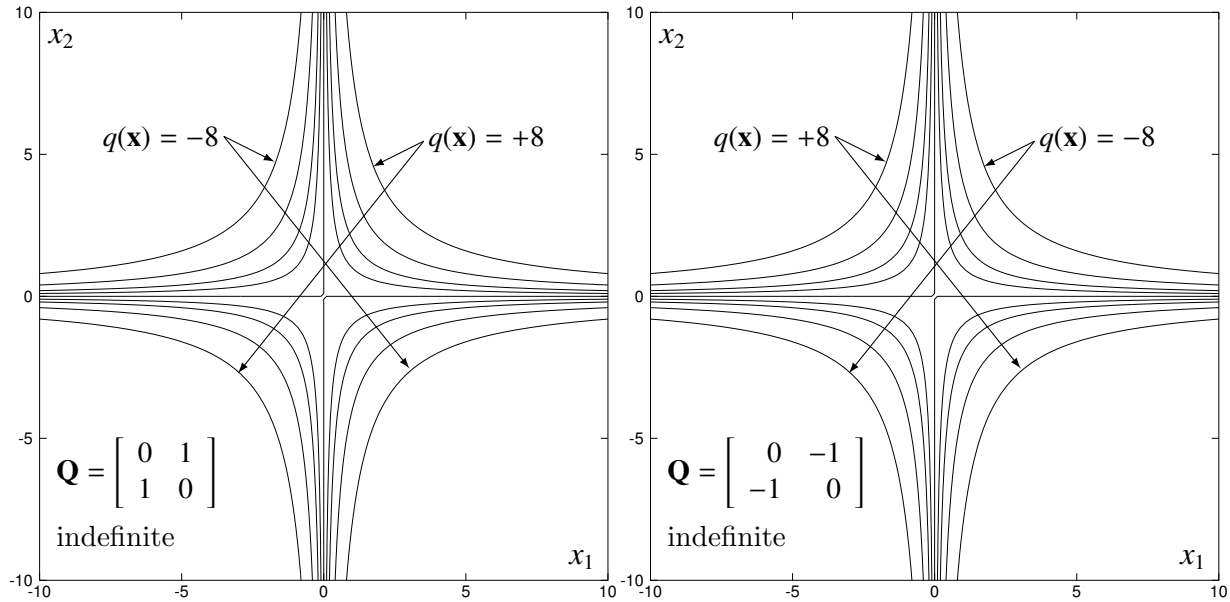
The constant d simply raises or lowers the graph of the function, while the linear term displaces the graph in $\mathbf{x}$–space as well as raising or lowering it. It is easy to see from the $n = 1$ example plotted to the right that these effects change the position of the graph but not its shape or orientation. In contrast, changing $\mathbf{Q}$ can change the shape or orientation of the graph of $q(\mathbf{x})$ and of its contours, and we can study these effects by varying $\mathbf{Q}$ while holding $\mathbf{c}$ and d fixed.

The graphs on the next two pages show the contours of $q(\mathbf{x})$ with $\mathbf{c} = \mathbf{0}$ and $d = 0$ for various matrices $\mathbf{Q}$. In the top left panel on the next page $\mathbf{Q}$ is positive definite; $q(\mathbf{x}) = x_1^2 + x_2^2$ and the contours of the strictly convex bowl are circles that get bigger as the function value becomes more positive. In the top right panel $\mathbf{Q}$ is **negative definite** because $-\mathbf{Q}$ is positive definite [110, p139]; $q(\mathbf{x}) = -x_1^2 - x_2^2$ and the contours of the strictly concave inverted bowl get bigger as the function value becomes more negative. The bottom left graph shows the contours of a straight trough having parabolic cross section when $\mathbf{Q}$ is positive semidefinite and $q(\mathbf{x}) = 0x_1^2 + 1x_2^2$. The bottom right graph shows the contours of a parabolic ridge when $\mathbf{Q}$ is



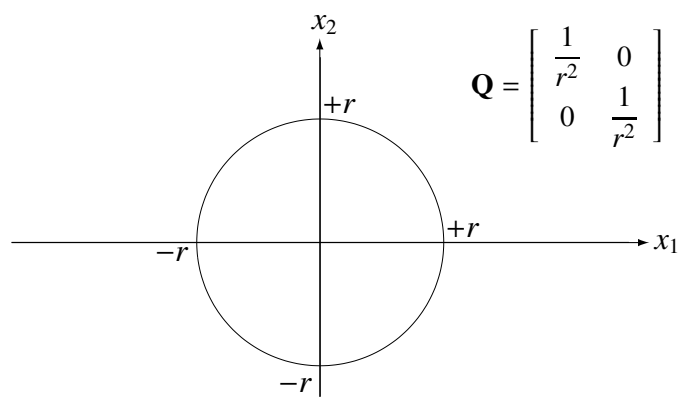
negative semidefinite because $-\mathbf{Q}$ is positive semidefinite [110, p139] and $q(\mathbf{x}) = 0x_1^2 - 1x_2^2$. The final two graphs, at the top of the following page, show the saddle-point contours of $q(\mathbf{x})$ when $\mathbf{Q}$ is indefinite.

For other matrices $\mathbf{Q}$ the circles can become ellipses and they can be tilted, the lines can be vertical or tilted, and the saddle can be oriented differently, but these pairs of pictures represent the only three kinds of contour diagram that a quadratic in $\mathbb{R}^2$ can produce. In higher dimensions the graph of $q(\mathbf{x})$ can be a more complicated object whose projection onto different two-dimensional flats can have any of these three characters, so if $\mathbf{Q}$ is indefinite and $n > 2$ then $q(\mathbf{x})$ might have multiple extrema and saddle points.

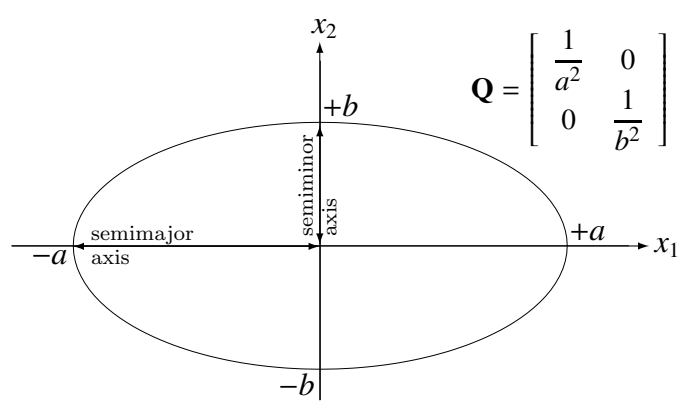


14.7.2 Ellipses

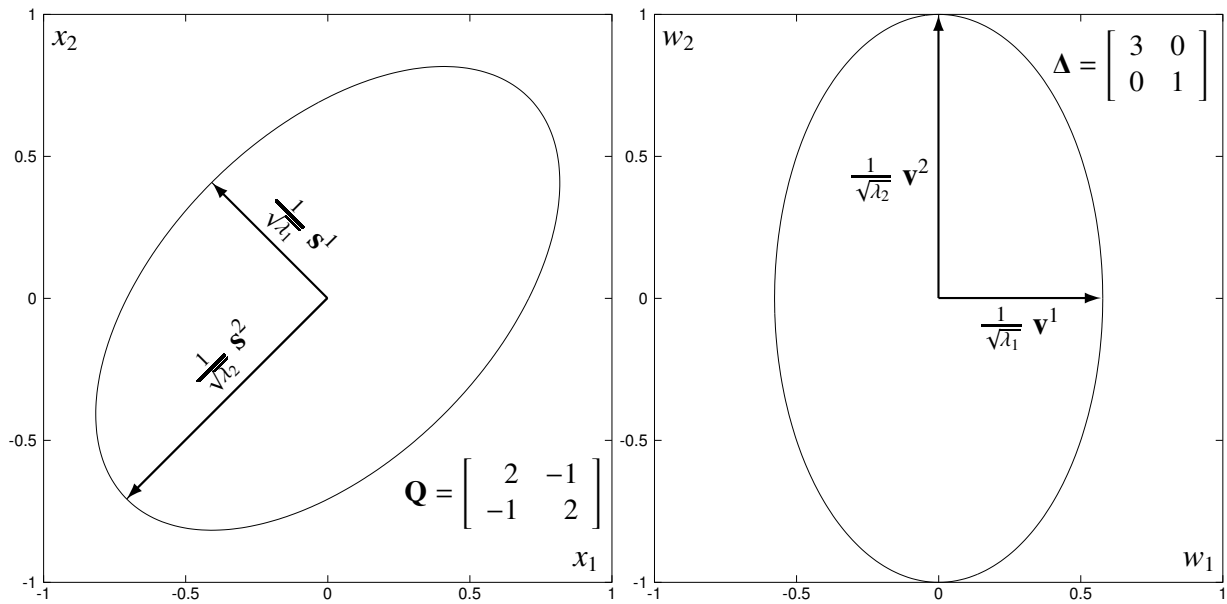
The simplest ellipse is a circle. The circles we plotted in §14.7.1 are some contours of $q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x}$ where $\mathbf{Q} = \mathbf{I}$. If $\mathbf{Q}$ is $1/r^2$ times the identity matrix, then the $q(\mathbf{x}) = \frac{1}{2}$ contour or $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ describes the circle $x_1^2/r^2 + x_2^2/r^2 = 1$ having radius r, pictured above to the right.



If $\mathbf{Q}$ is again diagonal but its diagonal elements are different, then [149, §11.6] $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ describes an ellipse $x_1^2/a^2 + x_2^2/b^2 = 1$ as pictured below to the right. Its axes are parallel to the coordinate axes, so adopting the terminology of §14.1 it is a **right ellipse**. The longer axis is called the **major axis** and the shorter axis is called the **minor axis**. Their halflengths, the **semimajor** and **semiminor** axes, are the numbers a and b that are squared in the denominators of x_1^2 and x_2^2.



Making the off-diagonal elements of $\mathbf{Q}$ nonzero (but equal to each other because we assumed the matrix is symmetric) tilts the ellipse with respect to the coordinate axes, as in the example shown on the left below [147, p242-243]. Notice that this is an ellipse rather than a circle even though the diagonal elements of $\mathbf{Q}$ happen to be equal.



The semimajor and semiminor axes of this ellipse, which are marked in the figure, depend on the matrix elements in a more complicated way than for a right ellipse. To find out how, we can diagonalize $\mathbf{Q}$ as we did in §14.2 to rotate the tilted ellipse into alignment with the coordinate axes. Once again we use a square matrix $\mathbf{S}$ whose columns are $\mathbf{Q}$–conjugate, but now we will make those columns unit eigenvectors of $\mathbf{Q}$. First, proceeding as in §11.5, we find the eigenvalues of $\mathbf{Q}$ like this.

$$|\mathbf{Q} - \lambda\mathbf{I}| = \begin{vmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{vmatrix} = (2-\lambda)^2 - 1 = \lambda^2 - 4\lambda + 3 = 0 \quad \Rightarrow \quad \lambda_1 = 3, \;\; \lambda_2 = 1.$$

Then the eigenvectors $\mathbf{s}^1$ and $\mathbf{s}^2$ satisfy $\mathbf{Q}\mathbf{s}^j = \lambda_j\mathbf{s}^j$.

$$\mathbf{Q}\mathbf{s}^1 = \lambda_1\mathbf{s}^1 \;\; \Rightarrow \;\; \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} s_1^1 \\ s_2^1 \end{bmatrix} = 3\begin{bmatrix} s_1^1 \\ s_2^2 \end{bmatrix} \;\; \Rightarrow \;\; -\mathbf{s}_1^1 - \mathbf{s}_2^1 = 0$$

$$\mathbf{Q}\mathbf{s}^2 = \lambda_2\mathbf{s}^2 \;\; \Rightarrow \;\; \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} s_1^2 \\ s_2^2 \end{bmatrix} = 1\begin{bmatrix} s_1^2 \\ s_2^2 \end{bmatrix} \;\; \Rightarrow \;\; -\mathbf{s}_1^2 + \mathbf{s}_2^2 = 0$$

Two eigenvectors of unit length that satisfy these equations are $\mathbf{s}^1 = [-1/\sqrt{2},\ +1/\sqrt{2}]^\top$ and $\mathbf{s}^2 = [-1/\sqrt{2},\ -1/\sqrt{2}]^\top$. The eigenvalues are distinct, so these vectors are sure to be $\mathbf{Q}$–conjugate and $\mathbf{Q}$ is sure to be diagonalizable. We can calculate $\mathbf{S}^\top\mathbf{Q}\mathbf{S} = \mathbf{\Delta}$ as follows.

$$\mathbf{S} = \begin{bmatrix} \mathbf{s}^1 & \mathbf{s}^2 \end{bmatrix} = \begin{bmatrix} \frac{-1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{+1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \quad \text{so} \quad \mathbf{S}^\top\mathbf{Q} = \frac{1}{\sqrt{2}}\begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} -3 & 3 \\ -1 & -1 \end{bmatrix}$$

and

$$\mathbf{S}^\top\mathbf{Q}\mathbf{S} = \frac{1}{\sqrt{2}}\begin{bmatrix} -3 & 3 \\ -1 & -1 \end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix} -1 & -1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{\Delta}$$

is the diagonal matrix of the eigenvalues. Using $\mathbf{Q} = \mathbf{S}^{-\top}\mathbf{\Delta}\mathbf{S}^{-1}$ we can rewrite the $\mathbf{x}$–space equation of the ellipse in terms of $\mathbf{\Delta}$:

$$\mathbf{x}^\top\mathbf{Q}\mathbf{x} = \mathbf{x}^\top[\mathbf{S}^{-\top}\mathbf{\Delta}\mathbf{S}^{-1}]\mathbf{x} = [\mathbf{x}^\top\mathbf{S}^{-\top}]\mathbf{\Delta}[\mathbf{S}^{-1}\mathbf{x}] = [\mathbf{S}^{-1}\mathbf{x}]^\top\mathbf{\Delta}[\mathbf{S}^{-1}\mathbf{x}] = 1.$$

Now if we let $\mathbf{w} = \mathbf{S}^{-1}\mathbf{x}$ the equation of the ellipse in $\mathbf{w}$–space is

$$\mathbf{w}^\top\mathbf{\Delta}\mathbf{w} = 1,$$

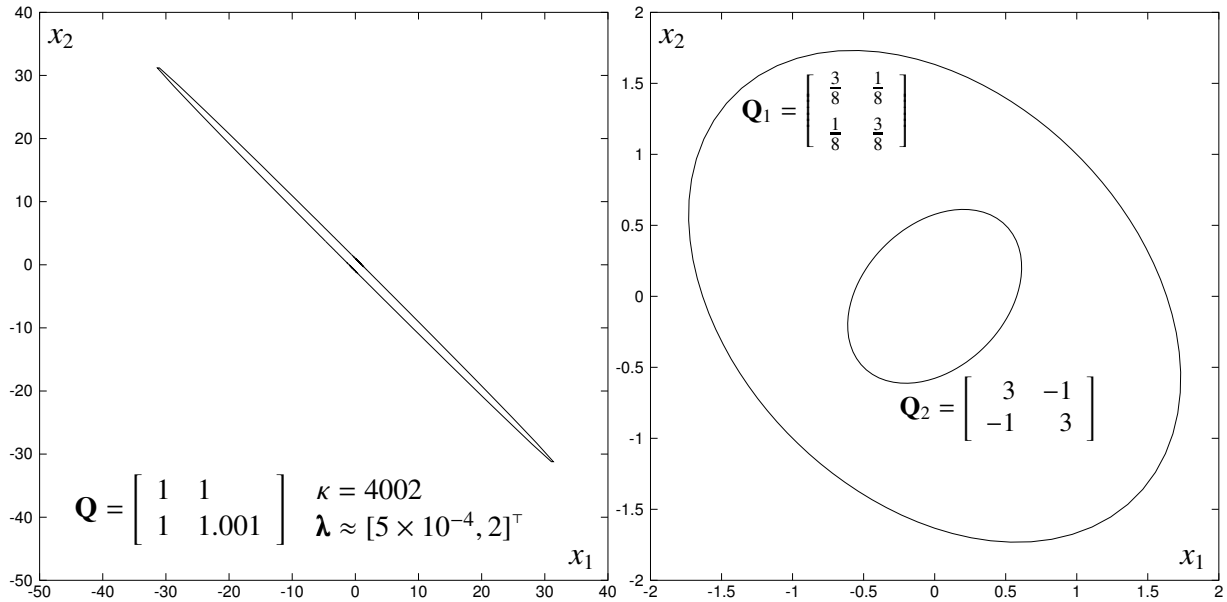
which is plotted in the right graph on the previous page. Because $\mathbf{\Delta}$ is a diagonal matrix its eigenvalues are just its diagonal elements, so it is still true that $\lambda_1 = 3$ and $\lambda_2 = 1$; the eigenvalues are preserved in the rotation. In $\mathbf{w}$–space the eigenvectors $\mathbf{v}^1$ and $\mathbf{v}^2$ satisfy $\mathbf{\Delta}\mathbf{v}^j = \lambda_j\mathbf{v}^j$.

$$\mathbf{\Delta}\mathbf{v}^1 = \lambda_1\mathbf{v}^1 \quad \Rightarrow \quad \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} v_1^1 \\ v_2^1 \end{bmatrix} = 3\begin{bmatrix} v_1^1 \\ v_2^2 \end{bmatrix} \quad \Rightarrow \quad \mathbf{v}^2 = 0$$

$$\mathbf{\Delta}\mathbf{v}^2 = \lambda_2\mathbf{v}^2 \quad \Rightarrow \quad \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} v_1^2 \\ v_2^2 \end{bmatrix} = 1\begin{bmatrix} v_1^2 \\ v_2^2 \end{bmatrix} \quad \Rightarrow \quad \mathbf{v}^1 = 0$$

Two eigenvectors of unit length that satisfy these equations are $\mathbf{v}^1 = [1,0]^\top = \mathbf{e}^1$ and $\mathbf{v}^2 = [0,1]^\top = \mathbf{e}^2$, the unit vectors in the coordinate directions. These eigenvectors point in the directions of the axes of the $\mathbf{w}$–space ellipse. Because it is a right ellipse, its semimajor and semiminor axes are respectively $1/\sqrt{1} = 1$ and $1/\sqrt{3} \approx 0.58$, or $1/\sqrt{\lambda_2}$ and $1/\sqrt{\lambda_1}$. The vectors shown in the right graph pointing from the center of the ellipse to the ends of its major and minor axes are thus $\mathbf{v}^2/\sqrt{\lambda_2}$ and $\mathbf{v}^1/\sqrt{\lambda_1}$ as shown. If the right ellipse is rotated to produce the picture on the left, these vectors rotate along with it, so in $\mathbf{x}$–space they are $\mathbf{s}^2/\sqrt{\lambda_2}$ and $\mathbf{s}^1/\sqrt{\lambda_1}$ as shown. Thus the half-axes of any ellipse, whether or not it is a right ellipse, are $1/\sqrt{\lambda_1}$ and $1/\sqrt{\lambda_2}$.

The eigenvalues $\boldsymbol{\lambda}$ of a matrix depend on its condition number (see §18.4.2) so the condition number $\kappa(\mathbf{Q})$ affects the shape of the ellipse $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$. The left picture on the next page shows that the ellipse corresponding to a matrix having even a moderate condition number is very thin (here 4 units) compared to its length ($\approx$89 units). In higher dimensions the ellipsoid corresponding to a badly-conditioned matrix can be thin in several dimensions,



compared to its longest axis. This is the manifestation in geometry of a numerical phenomenon which, as we first observed in §10.6.2, limits the performance of many optimization algorithms.

The ellipse corresponding to a matrix differs in size and shape from the ellipse corresponding to its inverse, as shown on the right above where the matrices $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are inverses of each other. If $\mathbf{s}$ is a unit eigenvector of $\mathbf{Q}_1$ with associated eigenvalue λ then

$$\begin{aligned} \mathbf{Q}_1\mathbf{s} &= \lambda\mathbf{s} \\ \mathbf{Q}_1^{-1}\mathbf{Q}_1\mathbf{s} &= \lambda\mathbf{Q}_1^{-1}\mathbf{s} \\ \left(\frac{1}{\lambda}\right)\mathbf{s} &= \mathbf{Q}_2\mathbf{s} \end{aligned}$$

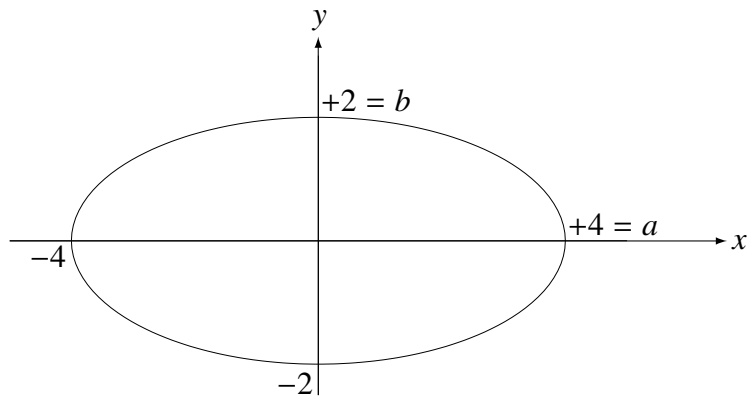
so $\mathbf{Q}_2$ also has $\mathbf{s}$ as a unit eigenvector, with the associated eigenvalue $1/\lambda$. Thus the ellipse axes point in the same directions, but their lengths are different if $\lambda \neq 1$.

In §24 we will be interested in the volume $\mathcal{V}$ of an ellipsoid, which can be computed in several different ways. In $\mathbb{R}^2$ this volume is just the area of an ellipse and is easily found by integration [146, p421-422]. The ellipse pictured on the next page has the matrix

$$\mathbf{Q} = \begin{bmatrix} \frac{1}{16} & 0 \\ 0 & \frac{1}{4} \end{bmatrix} \qquad \text{so its equation is} \qquad \frac{x^2}{4^2} + \frac{y^2}{2^2} = 1$$

and its semimajor and semiminor axes are $a = 4$ and $b = 2$ as shown. In the first quadrant the height of the curve is given by

$$y = b\sqrt{1 - \frac{x^2}{a^2}} = \frac{b}{a}\sqrt{a^2 - x^2}$$



so the area of the whole ellipse is

$$\mathcal{V} = 4\int_0^a \frac{b}{a}\sqrt{a^2 - x^2}\ dx = \pi ab = 8\pi \approx 25.1$$

We saw earlier that the semimajor and semiminor axes can be found from the eigenvalues of $\mathbf{Q}$, which for this example are $\lambda_1 = \frac{1}{16}$ and $\lambda_2 = \frac{1}{4}$. Thus

$$\mathcal{V} = \pi ab = \pi\left(\frac{1}{\sqrt{\lambda_1}}\right)\left(\frac{1}{\sqrt{\lambda_2}}\right) = \pi\left(\frac{1}{1/4} \times \frac{1}{1/2}\right) = 8\pi.$$

Another way of writing this formula for the volume uses the product of the reciprocals of the eigenvalues of $\mathbf{Q}$.

$$\mathcal{V} = \pi\sqrt{\frac{1}{\lambda_1}\frac{1}{\lambda_2}} = \pi\sqrt{\frac{1}{1/16} \times \frac{1}{1/4}} = \pi\sqrt{16 \times 4} = \pi\sqrt{64} = 8\pi$$

The reciprocals of the eigenvalues of $\mathbf{Q}$ are just the eigenvalues of $\mathbf{Q}^{-1}$. By using the eigenvectors of $\mathbf{Q}^{-1}$ we can diagonalize it, and this change of coordinates has the effect of rotating the corresponding ellipse without changing its size or shape. The matrix of that rotated ellipse is diagonal with the eigenvalues on the diagonal, so the product of its diagonals its just its determinant. In our example $\mathbf{Q}^{-1}$ is already diagonal, and its determinant is the product of its eigenvalues.

$$\left|\mathbf{Q}^{-1}\right| = \begin{vmatrix} 16 & 0 \\ 0 & 4 \end{vmatrix} = 64$$

Then we can find the volume as

$$\mathcal{V} = \pi\sqrt{|\mathbf{Q}^{-1}|} = \pi\sqrt{64} = 8\pi.$$

Diagonalizing a matrix does not change its eigenvalues, so even if $\mathbf{Q}^{-1}$ is not diagonal we can use this formula for the volume of the ellipse defined by $\mathbf{Q}$. The factor that appears before the square root is the volume $\mathcal{V}_1$ of a **unit ball**, which is just an epsilon-neighborhood of

radius 1 (see §9.3). In $\mathbb{R}^2$ the unit ball is a unit circle, so its volume is its area π. In $\mathbb{R}^n$ the volume of a unit ball is given [69, p620] by this formula.

$$\mathcal{V}_1 = \frac{\pi^{n/2}}{\Gamma\left(1+\frac{n}{2}\right)} = \begin{cases} \dfrac{\pi^{\lfloor n/2 \rfloor}}{\prod_{j=0}^{\lfloor n/2 \rfloor - 1}\left(\frac{n}{2}-j\right)} & n \text{ even} \\[2ex] \dfrac{\pi^{\lfloor n/2 \rfloor}}{\prod_{j=0}^{\lfloor n/2 \rfloor}\left(\frac{n}{2}-j\right)} & n \text{ odd} \end{cases}$$

The gamma function $\Gamma(t)$ is defined by an integral (see §25.6) but when its argument is a multiple of $\frac{1}{2}$ as in this case it can be evaluated as a continued product [116, p534]. The expressions on the right use the **floor function** $\lfloor n/2 \rfloor$ to obtain [94, §1.2.4] the highest integer less than or equal to $n/2$ (this is different from $n/2$ only when n is odd). To use the formula

$$\mathcal{V} = \mathcal{V}_1 \sqrt{|\mathbf{Q}^{-1}|}.$$

it is not actually necessary to invert $\mathbf{Q}$, because $|\mathbf{Q}^{-1}| = 1/|\mathbf{Q}|$.

Now we can generalize from the first formula we found for the area of an ellipse in terms of its semimajor and semiminor axes: if an ellipsoid in $\mathbb{R}^n$ has half-axes h_j then its volume is

$$\mathcal{V} = \mathcal{V}_1 \prod_{j=1}^{n} h_j.$$

14.7.3 Plotting Ellipses

We can plot the elliptical contours of a strictly convex quadratic by using the `gridcntr.m` routine of §9.1 to compute function values and the MATLAB `contour()` command to interpolate between them and draw the curves. Often, however we will have occasion to plot a single ellipse (as I did several times in §14.7.2) and then it is more convenient to exactly find points on that particular curve and use the MATLAB `plot()` command to connect them. In this Section I will assume for notational simplicity that the ellipse is described as the locus of points where

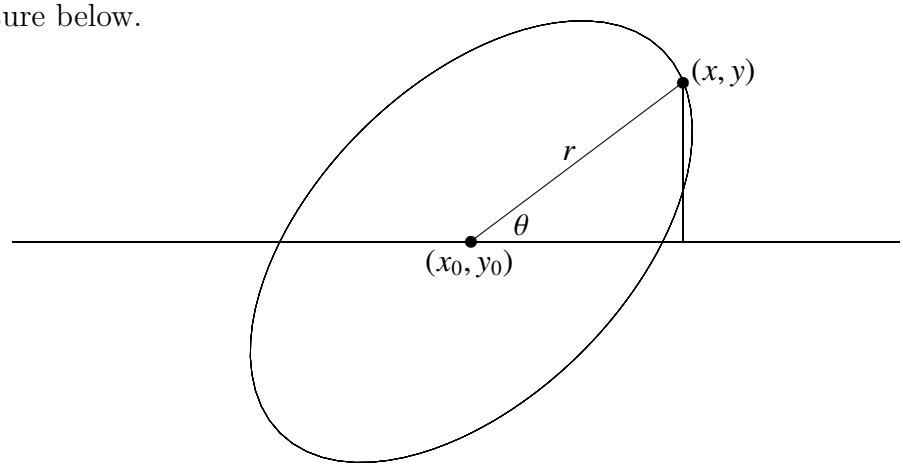
$$\begin{bmatrix} x-x_0 & y-y_0 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x-x_0 \\ y-y_0 \end{bmatrix} = 1.$$

We have assumed that the matrix is symmetric so in practice it will turn out that b and c get the same value, but calling these elements by different names will make what follows easier to understand.

Points on the curve can be computed by finding the lowest and highest x coordinates where the ellipse is defined, dividing that interval into equally-spaced x values, and using a formula to calculate the height of the curve at each. There are of course two y values for each

x, so some logic is required to distinguish the upper and lower branches of the ellipse and to ensure that their ends connect, but despite this complication the approach has a simple implementation (see Exercise 14.8.52). Unfortunately, the figure it generates often includes the artifact of a vertical segment at each end of the ellipse, even when the increment in x is made very small.

To produce a curve that is more likely to look smooth when a reasonable number of points are used, we will instead take the approach of incrementing the central angle θ shown in the picture below.



Here an arbitrary point (x, y) on the ellipse is a distance r from the center (x_0, y_0) at an angle θ from the horizontal. From this geometry we find

$$\begin{aligned} \frac{y - y_0}{x - x_0} &= \tan(\theta) \\ y - y_0 &= (x - x_0)\tan(\theta) \end{aligned}$$

$$\boxed{y = y_0 + (x - x_0)\tan(\theta).}$$

From the equation of the ellipse,

$$a(x - x_0)^2 + (b + c)(x - x_0)(y - y_0) + d(y - y_0)^2 = 1.$$

Substituting for $(y - y_0)$ in this equation,

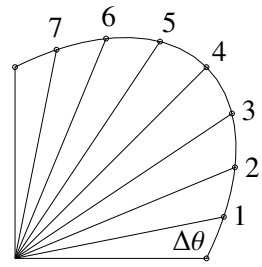
$$\begin{aligned} a(x - x_0)^2 + (b + c)(x - x_0)\Big[(x - x_0)\tan(\theta)\Big] + d\Big[(x - x_0)^2\tan^2(\theta)\Big] &= 1 \\ (x - x_0)^2\Big[a + (b + c)\tan(\theta) + d\tan^2(\theta)\Big] &= 1. \\ (x - x_0)^2 &= \frac{1}{a + (b + c)\tan(\theta) + d\tan^2(\theta)} \end{aligned}$$

$$\boxed{x = x_0 + \frac{1}{\sqrt{a + (b + c)\tan(\theta) + d\tan^2(\theta)}}.}$$

Using the boxed equations I wrote the MATLAB function `ellipse.m`, which is listed below and on the next page.

The input parameters [1] are `xz` = x_0, `yz` = y_0, the matrix `Q` = $\mathbf{Q}$, and `smax`, which is the number of interior points to use in constructing each quadrant of the figure. The return parameters `xt` and `yt` are vectors of length `tmax` containing the coordinates to be plotted, and the return code signals success (`rc=0`) or failure (`rc=1`).

The routine begins [4] by computing `tmax`, the total number of points that will be used. At $\theta = \pi/2$ and $\theta = 3\pi/2$ the analysis breaks down, so for each quadrant I found the coordinates of its first endpoint separately from those of its `smax` interior points. The picture shows the first quadrant divided into `smax+1=8` wedges, with endpoints at $\theta = 0$ and $\theta = \pi/2$ and interior points numbered $1\ldots7$, spaced equally at increments of $\Delta\theta = (\pi/2)/(\texttt{smax}+1)$. The last boundary point, at $\theta = \pi/2$, is the first boundary point of the next quadrant, so to cover the four quadrants takes `4*(1+smax)` points. To close the curve the first point of the first quadrant must repeated, yielding the formula in the code.



Next the routine [5-6] zeros `xt` and `yt` and [7] computes the determinant of `Q`. If [8] either leading principal minor is nonpositive, the routine resigns [10] with [9] `rc=1`. Otherwise [12-13] it copies the elements `Q(1,1)`... into less verbose variable names and [14] initializes `t`, which counts the points that have been found so far.

```
 1 function [xt,yt,rc,tmax]=ellipse(xz,yz,Q,smax)
 2 % plot the ellipse (x-xz)'Q(x-xz)=1
 3
 4   tmax=4*(1+smax)+1;                  % points to be returned
 5   xt=zeros(tmax,1);                   % fix sizes
 6   yt=zeros(tmax,1);                   % of coordinate vectors
 7   detQ=Q(1,1)*Q(2,2)-Q(2,1)*Q(1,2);   % determinant of Q
 8   if(Q(1,1) <= 0 || detQ <= 0)        % test leading principal minors
 9      rc=1;                            % not pd => not an ellipse
10      return                           % give up
11   end
12   a=Q(1,1); b=Q(1,2);                 % extract
13   c=Q(2,1); d=Q(2,2);                 % its elements
14   t=0;                                % initialize point count
15
16 % first quadrant
17   t=t+1;                              % count the point
18   xt(t)=xz+1/sqrt(a);                 % x at theta=0
19   yt(t)=yz;                           % y at theta=0
20   for s=1:smax                        % find smax interior points
21       theta=(pi/2)*(s/(smax+1));                  % angle
22       denom=a+(c+b)*tan(theta)+d*(tan(theta))^2;  % denominator
23       t=t+1;                                      % count the point
24       xt(t)=xz+1/sqrt(denom);                     % x at theta
25       yt(t)=yz+(xt(t)-xz)*tan(theta);             % y at theta
26   end                                 % end of quadrant
27
```



```
28 % second quadrant
29   t=t+1;
30   xt(t)=xz;
31   yt(t)=yz+1/sqrt(d);
32   for s=1:smax
33       theta=(pi/2)+(pi/2)*(s/(smax+1));
34       denom=a+(c+b)*tan(theta)+d*(tan(theta))^2;
35       t=t+1;
36       xt(t)=xz-1/sqrt(denom);
37       yt(t)=yz+(xt(t)-xz)*tan(theta);
38   end
39
40 % third quadrant
41   t=t+1;
42   xt(t)=xz-1/sqrt(a);
43   yt(t)=yz;
44   for s=1:smax
45       theta=pi+(pi/2)*(s/(smax+1));
46       denom=a+(c+b)*tan(theta)+d*(tan(theta))^2;
47       t=t+1;
48       xt(t)=xz-1/sqrt(denom);
49       yt(t)=yz+(xt(t)-xz)*tan(theta);
50   end
51
52 % fourth quadrant
53   t=t+1;
54   xt(t)=xz;
55   yt(t)=yz-1/sqrt(d);
56   for s=1:smax
57       theta=(3*pi/2)+(pi/2)*(s/(smax+1));
58       denom=a+(c+b)*tan(theta)+d*(tan(theta))^2;
59       t=t+1;
60       xt(t)=xz+1/sqrt(denom);
61       yt(t)=yz+(xt(t)-xz)*tan(theta);
62   end
63
64 % close the ellipse
65   t=t+1;
66   xt(t)=xz+1/sqrt(a);
67   yt(t)=yz;
68   rc=0;
69
70 end
```

The calculations for the first quadrant of the graph begin 17-19 with the first boundary point. Then 20-26 the interior points are found. As can be seen from the picture above, point `s` is at the angle 21

$$\theta_{\mathtt{s}} = \frac{\mathtt{s}\pi/2}{\mathtt{smax}+1}.$$

The quantity that appears under the radical in the formula for x is here called `denom` 22. The point counter `t` is incremented 23 and the formulas are used 24-25 to find the coordinates of the point. The code for the other quadrants is similar but varies to account for the changing geometry of the picture (see Exercise 14.8.53).

The final stanza in the code 64-68 repeats the starting point of the curve 66-67 = 18-19 and 68 sets `rc=0` to signal success. This routine was used to draw the pictures in §14.7.2, and I will use it in future Chapters whenever it is necessary to plot an ellipse.

14.8 Exercises

14.8.1 [E] Use the definition of orthogonality to show that the coordinate directions $\mathbf{e}^j$ are mutually orthogonal.

14.8.2 [E] Steepest descent generates successive search directions that are orthogonal. Why does that happen?

14.8.3 [H] Two vectors $\mathbf{u}$ and $\mathbf{v}$ have the dot product $\mathbf{u}^\top\mathbf{v} = \|\mathbf{u}\| \times \|\mathbf{v}\| \times \cos(\theta)$, where θ is the angle between the vectors measured in the plane that contains them both [146, §11.3]. (a) Prove this equality. (b) Show that the algebraic and geometric definitions of orthogonality imply each other.

14.8.4 [H] What is necessary for a constrained nonlinear program to be a *quadratic* program? Find $\mathbf{Q}$, $\mathbf{b}$, and c such that $f(\mathbf{x}) = 2x_1^2 + 2x_1x_2 + 2x_2^2 - 3(x_1 + x_2 + 1) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x} + c$. Why can the constant c be ignored in minimizing $f(\mathbf{x})$?

14.8.5 [E] If $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x}$, what is its Hessian matrix $\mathbf{H}$? Is $\mathbf{H}$ a function of $\mathbf{x}$?

14.8.6 [H] If $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x}$ and $\mathbf{Q}$ is positive definite, then the system of linear algebraic equations $\mathbf{Q}\mathbf{x} = \mathbf{b}$ has a unique solution. (a) Why is it sometimes preferable to minimize $f(\mathbf{x})$ rather than simply solving the linear system? (b) For

$$\mathbf{Q} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

solve $\mathbf{Q}\mathbf{x} = \mathbf{b}$ both ways.

14.8.7 [E] What line search step length minimizes the function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x}$ if we start at the point $\bar{\mathbf{x}}$ and search in the direction $\mathbf{d}$?

14.8.8 [H] In §14.1, I claimed that if $f(\alpha) = \frac{1}{2}(\mathbf{x}^k + \alpha\mathbf{d}^k)^\top\mathbf{Q}(\mathbf{x}^k + \alpha\mathbf{d}^k)$ then

$$\frac{df}{d\alpha} = [\mathbf{Q}(\mathbf{x}^k + \alpha\mathbf{d}^k)]^\top\mathbf{d}^k - \mathbf{b}^\top\mathbf{d}^k.$$

Show that this claim is true.

14.8.9 [E] What is an *ellipsoid*? What is a *right ellipsoid*? What must be true of a quadratic function's $\mathbf{Q}$ matrix for the contours of the function to be right ellipsoids? Explain why it is easy to minimize a function whose contours are right ellipsoids.

14.8.10 [E] In solving the `gns` problem in §14.2 we found the conjugate directions $s^1 = [1, 0]^\top$ and $s^2 = [\frac{1}{2}, -1]^\top$. Show that each $x^j = \mathbf{S}\mathbf{e}^j$ where $\mathbf{e}^j$ is a coordinate direction.

14.8.11 [P] The *cyclic coordinate descent* algorithm (see §25.7.2) is like steepest descent except that it uses the coordinate directions $\mathbf{e}^1, \mathbf{e}^2 \ldots \mathbf{e}^n, \mathbf{e}^1, \mathbf{e}^2 \ldots \mathbf{e}^n, \ldots$ as the search directions. (a) When does this algorithm produce the same sequence of iterates as the conjugate gradient algorithm? (b) Find analytically an expression for the optimal step in direction $\mathbf{e}^j$ if this algorithm is used to solve the `gns` problem. (c) Write a MATLAB program that solves the

`gns` problem using cyclic coordinate descent. (`d`) Plot an error curve and use it to estimate the algorithm's rate and constant of convergence.

14.8.12 `[E]` What does it mean to *diagonalize* a matrix?

14.8.13 `[H]` Show that if $\mathbf{S}^\top\mathbf{Q}\mathbf{S} = \mathbf{\Delta}$ then, because of the rules of matrix multiplication, $\mathbf{\Delta}_{ij} = \mathbf{s}^{i\top}\mathbf{Q}\mathbf{s}^j$. If $\mathbf{Q}$ is positive definite, how do we know that $\mathbf{s}^{i\top}\mathbf{Q}\mathbf{s}^i > 0$ for $i = 1 \ldots n$?

14.8.14 `[H]` Is the matrix

$$\mathbf{Q} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

diagonalizable? If yes, find linearly independent columns $\mathbf{s}^1$ and $\mathbf{s}^2$ of $\mathbf{S}$ such that $\mathbf{S}^\top\mathbf{Q}\mathbf{S} = \mathbf{\Delta}$ is a diagonal matrix; if no, explain why that is impossible.

14.8.15 `[E]` If $\mathbf{Q} = \mathbf{I}$, find two $\mathbf{Q}$–conjugate vectors $\mathbf{u}$ and $\mathbf{v}$ *other* than $\mathbf{e}^1$ and $\mathbf{e}^2$.

14.8.16 `[E]` Is there a matrix $\mathbf{A}$ such that the vectors $\mathbf{u} = [1, -2]^\top$ and $\mathbf{v} = [-3, 6]^\top$ are $\mathbf{A}$–conjugate? If yes, find $\mathbf{A}$; if no, explain why $\mathbf{u}$ and $\mathbf{v}$ cannot be $\mathbf{A}$–conjugate.

14.8.17 `[H]` The function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x}$ where

$$\mathbf{Q} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

has its strict global minimum at $\mathbf{x}^\star = [-1, -1]^\top$. (`a`) Find linearly independent vectors $\mathbf{u}$ and $\mathbf{v}$ that are $\mathbf{Q}$–conjugate. (`b`) Use $\mathbf{u}$ and $\mathbf{v}$ to diagonalize $\mathbf{Q}$, and rewrite the function as $f(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top\mathbf{\Delta}\mathbf{w} - \mathbf{a}^\top\mathbf{w}$ where $\mathbf{\Delta}$ is a diagonal matrix. (`c`) Find the minimizing point $\mathbf{w}^\star$ of $f(\mathbf{w})$ by any means you like. (`d`) From $\mathbf{w}^\star$, find $\mathbf{x}^\star$. (`e`) Minimize $f(\mathbf{x})$ by any means you like, and confirm that you find $\mathbf{x}^\star$.

14.8.18 `[E]` If $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x}$ and we diagonalize $\mathbf{Q}$ by finding a matrix $\mathbf{S}$ such that $\mathbf{S}^\top\mathbf{Q}\mathbf{S} = \mathbf{\Delta}$, then we can write $f(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top\mathbf{\Delta}\mathbf{w} - \mathbf{a}^\top\mathbf{w}$. (`a`) To minimize $f(\mathbf{w})$ by searching in conjugate directions, what directions should we use? (`b`) To minimize $f(\mathbf{x})$ by searching in conjugate directions, what directions should we use?

14.8.19 `[E]` What is the maximum number of steps required to minimize a strictly convex quadratic function of n variables by doing exact line searches along conjugate directions? What is the minimum number of steps that might be sufficient?

14.8.20 `[P]` In §14.3 several ways are suggested for generating conjugate directions. If $\mathbf{Q}$ is symmetric and has distinct eigenvalues then its eigenvectors are $\mathbf{Q}$–conjugate. Write a program to solve the `gns` problem by using that approach. Hint: use the MATLAB statement `[S,Lambda,Sinv]=svd(Q)` to find a matrix `S` whose columns are orthonormal eigenvectors of $\mathbf{Q}$, and then do an exact analytic line search in each of those directions.

14.8.21 `[P]` In §14.3 several ways are suggested for generating conjugate directions. If $\mathbf{Q}$ is positive definite and an exact line search is used, the DFP algorithm generates $\mathbf{d}^k$ that

are $\mathbf{Q}$–conjugate. Write a program to solve the `gns` problem by using that approach, and confirm numerically that the directions it generates are conjugate.

14.8.22 [E] Conjugate gradient algorithms use a simple method of generating conjugate directions. What is it?

14.8.23 [E] How is the residual $\mathbf{r} = \mathbf{Q}\mathbf{x} - \mathbf{b}$ that is used in the conjugate gradient algorithm related to the gradient of $f(\mathbf{x})$?

14.8.24 [E] The conjugate gradient algorithm computes residual vectors $\mathbf{r}^k$ and direction vectors $\mathbf{d}^k$. Which of these vectors are $\mathbf{Q}$–conjugate? Which of them are orthogonal?

14.8.25 [P] When we solved the `gns` problem in §14.2 by searching conjugate directions, we arbitrarily chose $\mathbf{s}^1 = [1, 0]^\top$. When the conjugate gradient algorithm is used to solve the problem, the first direction it chooses is that of steepest descent (see lines 4-5 in `cg.m`). (a) Write a MATLAB program that uses `cg.m` to solve the `gns` problem and plots its convergence trajectory over contours of the objective. How does this picture compare to the $\mathbf{x}$-space plot in §14.2? (b) Modify `cg.m` to use the arbitrary direction $\mathbf{s}^1 = [1, 0]^\top$ as its first `d`. Does it still solve `gns` in two steps? Does the algorithm still have the properties discussed in §14.4? (c) In the conjugate gradient algorithm, *why* must $\mathbf{r}^0 = \mathbf{Q}\mathbf{x}^0 - \mathbf{b}$ in order for $\mathbf{d}^1$ and $\mathbf{d}^0$ to be $\mathbf{Q}$–conjugate?

14.8.26 [E] In §14.4, pseudocode is listed for two versions of the conjugate gradient algorithm. How much arithmetic is saved by using the second version rather than the first? Show how `cg.m` can be rewritten to require only one matrix-vector multiplication per iteration.

14.8.27 [P] Suppose all the elements of $\mathbf{Q}$ are zero except for the diagonal, whose elements are all 10, and the superdiagonal and subdiagonal, whose elements are all 1. Write a MATLAB function `Qd(d)` that receives a vector $\mathbf{d}$ and returns the product $\mathbf{Q}\mathbf{d}$ without storing any of the elements of $\mathbf{Q}$. Test your routine using randomly-generated vectors $\mathbf{d} \in \mathbb{R}^{1000}$. How can you tell whether the results are correct?

14.8.28 [E] What is the order of convergence of the conjugate gradient algorithm? How does its convergence constant depend on $\mathbf{Q}$? Why in practice might it not find $\mathbf{x}^\star$ precisely in n or fewer iterations?

14.8.29 [P] Consider the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ where [20, Exercise 8.1.26]

$$\mathbf{A} = \begin{bmatrix} 1.59 & 1.69 & 2.13 \\ 1.69 & 1.31 & 1.72 \\ 2.13 & 1.72 & 1.85 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

(a) Solve the linear system using the MATLAB backslash operator. (b) Solve the linear system using the function `cg.m` of §14.4. (c) The conjugate gradient algorithm is guaranteed to work only if $\mathbf{A}$ is positive definite and symmetric. Is the $\mathbf{A}$ given above positive definite and symmetric?

14.8.30 [P] The MATLAB command `A=hilb(n)` returns the $n \times n$ **Hilbert matrix A**, which has $a_{ij} = 1/(i + j - 1)$. The condition number of the Hilbert matrix grows very fast as n increases, so if it is the coefficient matrix in $\mathbf{Ax} = \mathbf{b}$ the linear system becomes numerically troublesome as n increases. Write a program that uses `cg.m` to solve $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is the $n \times n$ Hilbert matrix and $\mathbf{b} = \mathbf{1}$, starting from $\mathbf{x}^0 = \mathbf{0}$, for several values of n. Plot $k^\star(n)$, the number of iterations required to achieve an error level of $\epsilon = 10^{-6}$, as a function of n.

14.8.31 [P] The Fletcher-Reeves and Polak-Ribière algorithms are both generalizations of the conjugate gradient algorithm. (a) How do they differ from it, and from each other? (b) Use `flrv.m` and `plrb.m` to solve the `gpr` problem pictured in §9.3, starting from $\mathbf{x}^0 = [2, 3]^\top$. How do the two algorithms compare? (c) Use `flrv.m` and `plrb.m` to solve the Himmelblau 28 problem [80, p428],

$$\text{minimize } f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2,$$

starting from $\mathbf{x}^0 = [1, 1]^\top$. How do the two algorithms compare?

14.8.32 [P] The Fletcher-Reeves and Polak-Ribière algorithms are competitive with steepest descent because all three have linear convergence and don't use the Hessian. Write a MATLAB program that invokes `sdw.m`, `flrv.m`, and `plrb.m` to solve a problem one step at a time and plot the convergence trajectories and error curves of the three algorithms. Use your program to compare the algorithms when they are used to solve the problems (a) `gns` and (b) `rb`.

14.8.33 [H] Show that the Polak-Ribière formula for β_{k+1} can result in a $\mathbf{d}^{k+1}$ that is not a descent direction. In the code for `plrb.m`, what direction is used if the formula yields a negative number?

14.8.34 [H] The Polak-Ribière formula for β_{k+1} can be viewed as implementing the heuristic that if $\nabla f(\mathbf{x}^k)$ has the same direction at successive points then steepest descent will lead to $\mathbf{x}^\star$. Construct an $\mathbb{R}^2$ example problem in which that happens. Can you construct an example in which the heuristic fails?

14.8.35 [E] If $q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Qx} + \mathbf{c}^\top\mathbf{x} + d$ and $\mathbf{x} \in \mathbb{R}^2$, explain how the graph of the function is affected by changing (a) d; (b) $\mathbf{c}$. Illustrate your answers by drawing contour diagrams, assuming $\mathbf{Q}$ is a positive definite matrix.

14.8.36 [E] Suppose that $q(\mathbf{x})$ is a quadratic function of $\mathbf{x} \in \mathbb{R}^n$. (a) Write down a formula for $q(\mathbf{x})$. Carefully describe the contours of $q(\mathbf{x})$ if $n = 2$ and the function is (b) strictly convex; (c) concave but not strictly concave; (d) neither convex nor concave.

14.8.37 [E] How can we tell of a matrix is *negative definite*? How can we tell if it is negative semidefinite?

14.8.38 [H] If $\mathbf{x} \in \mathbb{R}^2$, write down a function $q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Qx}$ whose contours are (a) vertical lines; (b) slanting lines.

14.8.39 [E] Describe the three kinds of contour diagram that a quadratic in $\mathbb{R}^2$ can produce.

14.8.40 [E] What makes an ellipse a circle? What makes an ellipse a right ellipse? If an ellipse has the equation $x_1^2/16 + x_2^2/36 = 1$, what are its semiminor and semimajor axes?

14.8.41 [E] A certain ellipse defined by $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ has axes that are not parallel to the coordinate axes. (a) What must be true of $\mathbf{Q}$? Write down all the properties you can think of. (b) How do the semiminor and semimajor axes of the ellipse depend on $\mathbf{Q}$? (c) How do the directions of its axes depend on the matrix?

14.8.42 [H] In §14.7.2 we found for the matrix on the left below the eigenvectors on the right.

$$\mathbf{Q} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \qquad \begin{aligned} \mathbf{s}^1 &= \left[-1/\sqrt{2}, +1/\sqrt{2}\right]^\top \\ \mathbf{s}^2 &= \left[-1/\sqrt{2}, -1/\sqrt{2}\right]^\top \end{aligned}$$

Show that $\mathbf{s}^1$ and $\mathbf{s}^2$ are $\mathbf{Q}$–conjugate vectors.

14.8.43 [H] Suppose an ellipse $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ has the matrix on the left. (a) Show that the formula on the right gives the angle θ by which its graph is tilted.

$$\mathbf{Q} = \begin{bmatrix} q_1 & q_0 \\ q_0 & q_2 \end{bmatrix} \qquad \theta = \tfrac{1}{2}\arctan\left(\frac{q_0}{q_1 - q_2}\right)$$

(b) How can the eigenvectors $\mathbf{s}^1$ and $\mathbf{s}^2$ of $\mathbf{Q}$ be used to find θ?

14.8.44 [E] If a matrix $\mathbf{Q}$ is diagonalized by writing it as $\mathbf{Q} = \mathbf{S}^{-\top}\mathbf{\Delta}\mathbf{S}^{-1}$, what are the diagonal elements of $\mathbf{\Delta}$? What are the off-diagonal elements of $\mathbf{\Delta}$?

14.8.45 [E] How is the shape of an ellipse $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ affected by the condition number of $\mathbf{Q}$?

14.8.46 [E] Suppose that $\mathbf{s}_1$ and $\mathbf{s}_2$ are unit eigenvectors of the 2×2 positive definite matrix $\mathbf{Q}$. (a) How can you find unit eigenvectors of $\mathbf{Q}^{-1}$? (b) How are the eigenvalues of the two matrices related? (c) How does the ellipsoid defined by $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ differ in appearance from the ellipsoid defined by $\mathbf{x}^\top\mathbf{Q}^{-1}\mathbf{x} = 1$?

14.8.47 [E] Give formulas for finding the area of an ellipse whose equation is $\mathbf{x}^\top\mathbf{Q}\mathbf{x} = 1$ if you know (a) its semimajor and semiminor axes; (b) the eigenvalues of $\mathbf{Q}$; (c) the determinant of $\mathbf{Q}^{-1}$; (d) the determinant of $\mathbf{Q}$.

14.8.48 [E] Give a precise definition of the term *unit ball.* Evaluate the expressions $\lfloor -5.3 \rfloor$ and $\lfloor 5.3 \rfloor$.

14.8.49 [H] Use the definition of $\mathcal{V}_1$, the volume of a unit ball in $\mathbb{R}^n$, to show that the volume of a unit ball is (a) π in $\mathbb{R}^2$; (b) $\frac{4}{3}\pi$ in $\mathbb{R}^3$. (c) What is the volume of a unit ball in $\mathbb{R}^1$?

14.8.50 [H] An ellipse in $\mathbb{R}^3$ has all of its half-axes equal to 2. What is its volume?

14.8.51 [E] Describe two ways of plotting an ellipse in MATLAB.

14.8.52 [P] Suppose an ellipse is defined as the locus of points where

$$\begin{bmatrix} x - x_0 & y - y_0 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x - x_0 \\ y - y_0 \end{bmatrix} = 1$$

and consider the problem of plotting its curve. (a) Derive a formula that gives y as a function of the other variables. (b) Find the range of x over which the ellipse is defined. (c) Write a MATLAB routine `[xt,yt]=ellipsx(xzero,yzero,Q,tmax)` that finds `tmax` points on the curve at equally-spaced values of x in the range over which the ellipse is defined, and returns their coordinates in the vectors `xt` and `yt` for plotting with the MATLAB command `plot(xt,yt)`. (d) Test your routine by using it to plot each ellipse in §14.7.2 for which **Q** is given. How many points `tmax` do you need to get curves that look smooth? (e) Use `ellipse.m` to plot the same ellipses. How many points does it require?

14.8.53 [H] In the `ellipse.m` routine of §14.7.3, the coordinates of the first point in the second quadrant are given by `xt(t)=xz` and `yt(t)=yz+1/sqrt(d)`. (a) Where in the graph of the ellipse does this point appear? (b) Why is it necessary to use a formula different from the one we derived for $y(\theta)$ at this value of θ? (c) Explain why this formula is correct at that angle.

15

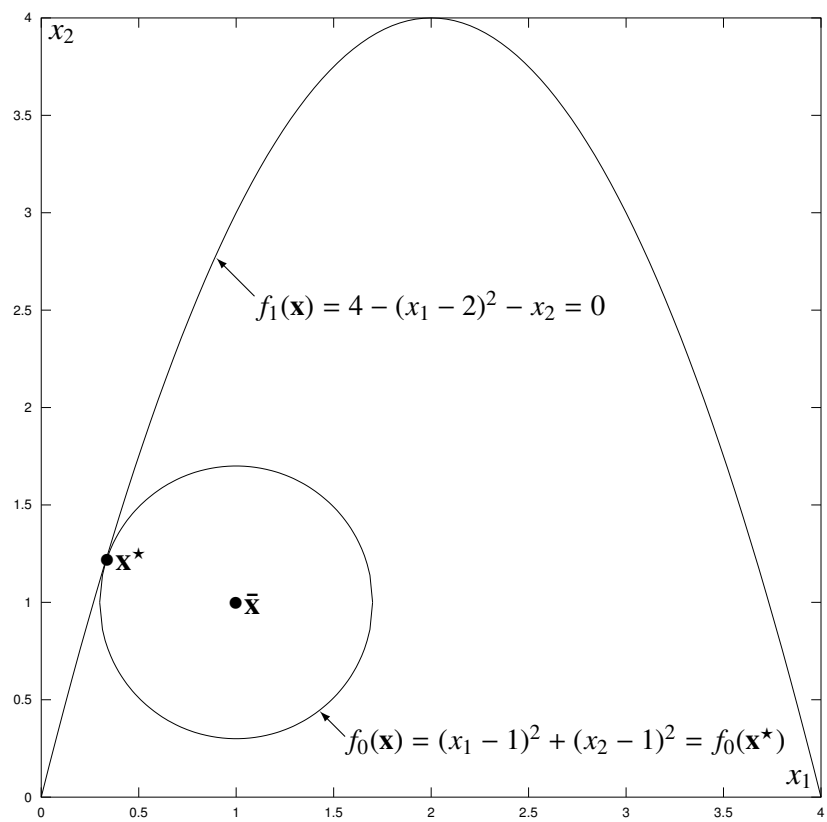
Equality Constraints

Since leaving Chapter 8 we have indulged the simple and carefree vocation of minimizing $f(\mathbf{x})$ over all of $\mathbb{R}^n$, but most practical applications of nonlinear programming give rise to models in which $\mathbf{x}^\star$ must also satisfy constraints. Our first application, the `garden` problem of §8.1, had inequality constraints, and §8.2 illustrated several different methods of enforcing them. With this Chapter we begin a more careful study of those same methods, starting with the easier case of constraints that are equations [3, §9.3].

The nonlinear program below, which I will call `arch1` (see §28.7.5), has $m = 1$ nonlinear equality constraint.

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) &= (x_1 - 1)^2 + (x_2 - 1)^2 \\ \text{subject to} \quad f_1(\mathbf{x}) &= 4 - (x_1 - 2)^2 - x_2 \;=\; 0 \end{aligned}$$

There are only two variables so, as we did in §8.2.1 for the `garden` problem, we can get to know this example by drawing its graph.



The *unconstrained* minimizing point of $f_0(\mathbf{x})$ is obviously, from either the picture or the objective formula, $\bar{\mathbf{x}} = [1, 1]^\top$, where $f_0(\bar{\mathbf{x}}) = 0$. That point does not satisfy the constraint, because $f_1(\bar{\mathbf{x}}) = 4 - (1-2)^2 - 1 = 2 \neq 0$. A higher contour of the objective does touch the zero contour of the constraint, at $\mathbf{x}^\star \approx [0.33, 1.20]^\top$ where $f_0(\mathbf{x}^\star) \approx 0.49$.

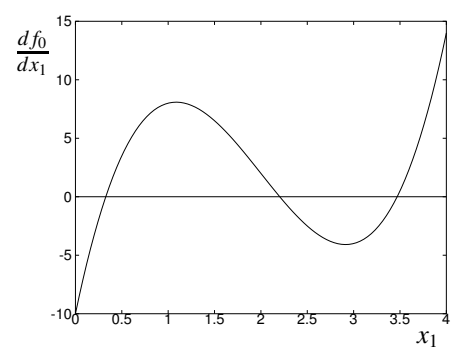
To find $\mathbf{x}^\star$ analytically we can use calculus as in §8.2.2. From the constraint equation we find that $x_2 = 4 - (x_1 - 2)^2$, and substituting that expression into the formula for $f_0(\mathbf{x})$ yields a **reduced objective** in which the number of variables has been reduced from $n = 2$ to $n - m = 1$.

$$f_0(x_1) = (x_1 - 1)^2 + (4 - (x_1 - 2)^2 - 1)^2$$

At the optimal point its derivative is zero, so we can find $x_1^\star$ by solving

$$\begin{aligned} \frac{df_0}{dx_1} = 2(x_1 - 1) + 2(3 - (x_1 - 2)^2)(-2(x_1 - 2)) &= 0 \\ \text{or} \qquad 4x_1^3 - 24x_1^2 + 38x_1 - 10 &= 0. \end{aligned}$$

To do that I wrote the MATLAB program `cubslv.m` listed below. It begins by 3-10 producing the graph to the right. From the graph I was able to bracket the roots 11 and then 13-23 find them precisely using the built-in zero-finder `fzero` 17. The notation `@(x1)` makes the formula for the cubic an "anonymous function" of `x1` [50, §11.10.2] so that it can be passed directly to `fzero`. (We will use `fzero` again in §17.3.1, for finding the roots of a nonlinear algebraic equation that is not a cubic.)




```
 1 % cubslv.m: find the stationary points of arch1
 2 % a graph shows where the roots are approximately
 3 set(gca,'FontSize',35)
 4 for p=1:100
 5     x1(p)=4*(p-1)/99;
 6     y(1,p)=4*x1(p)^3-24*x1(p)^2+38*x1(p)-10;
 7     y(2,p)=0;
 8 end
 9 plot(x1,y)
10 print -deps -solid cubslv.eps
11 xzeros=[0,1;2,3;3,4];
12
13 % then we can find them precisely
14 printf('     x1       x2       f0          g        h\n')
15 for r=1:3
16     xzero=xzeros(r,:);
17     x1=fzero(@(x1)4*x1^3-24*x1^2+38*x1-10,xzero);
18     x2=4*x1-x1^2;
19     f=(x1-1)^2+(x2-1)^2;
20     g=4*x1^3-24*x1^2+38*x1-10;
21     h=12*x1^2-48*x1+38;
22     printf('%7.5f  %7.5f  %7.4f  %7.7f  %7.3f\n',x1,x2,f,abs(g),h)
23 end
```

This program prints the output shown at the top of the next page, where `g` means df_0/dx_1 and `h` means $d^2 f_0/dx_1^2$. The zero values of `g` confirm that the three points are stationary, and from the value of `f0` and the sign of `h` we can classify them as the global minimum $\mathbf{x}^\star$, a global maximum, and a local minimum (see Exercise 15.6.4).

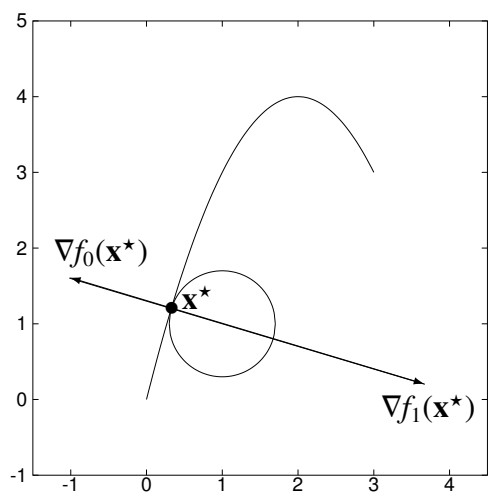

```
octave:1> cubslv
     x1       x2       f0          g        h
0.32702  1.20113   0.4934  0.0000000   23.586    ← global minimum
2.20336  3.95864  10.2017  0.0000000   -9.504    ← global maximum
3.46962  1.84022   6.8050  0.0000000   15.917    ← local minimum
octave:2> quit
```

If a nonlinear program has m equality constraints we should in principle be able to use them, as we did in this example and in §8.2.2, to eliminate m of the variables. Then we can minimize the reduced objective to find the optimal values of the remaining variables, and back-substitute into the equalities to get the values of the variables we eliminated. Unfortunately it is seldom possible to do that analytically if there are $m > 1$ nonlinear equalities, and it might not be possible even if there is only one [3, p274-278].

15.1 Parameterization of Constraints

The optimal point has another property that we could use to find it. This graph of our example shows $\nabla f_0(\mathbf{x}^\star)$ and $\nabla f_1(\mathbf{x}^\star)$ drawn to scale. Because the optimal contour of f_0 is tangent to the zero contour of f_1 at $\mathbf{x}^\star$, the gradients point in exactly opposite directions and are related by $\nabla f_0(\mathbf{x}^\star) = -\lambda \nabla f_1(\mathbf{x}^\star)$, where the scalar λ is the ratio of their lengths. Computing the gradients we find

$$\begin{bmatrix} 2(x_1 - 1) \\ 2(x_2 - 1) \end{bmatrix} = -\lambda \begin{bmatrix} -2(x_1 - 2) \\ -1 \end{bmatrix}.$$



The optimal point is also on the curve $f_1(\mathbf{x}) = 0$, so $\mathbf{x}^\star$ and λ satisfy the following equations.

$$\begin{aligned} 2(x_1 - 1) &= 2\lambda(x_1 - 2) \\ 2(x_2 - 1) &= \lambda \\ 4 - (x_1 - 2)^2 - x_2 &= 0 \end{aligned}$$

Solving this system by eliminating λ and x_2 we get a single equation in x_1,

$$4x_1^3 - 24x_1^2 + 38x_1 - 10 = 0,$$

which is the same cubic we found earlier. So this approach yields $\mathbf{x}^\star \approx [0.32702, 1.20113]^\top$ as before, with $\lambda^\star = 2(x_2^\star - 1) \approx 0.40226$.

There is an important connection between the substitution approach and the gradient approach, which we can see by considering a different way of using equality constraints to eliminate variables.

In our example the feasible set is all of the points on the curve described by

$$f_1(\mathbf{x}) = 4 - (x_1 - 2)^2 - x_2 = 0.$$

Suppose we let $t = x_1 - 2$. This choice of t means that $x_1 = 2 + t$ and we can rewrite the constraint equation as $4 - t^2 - x_2 = 0$. Thus the curve that is the feasible set has the following **parametric representation**.

$$\begin{aligned} x_1(t) &= 2 + t \\ x_2(t) &= 4 - t^2 \end{aligned}$$

As t varies from -2 to 2, the point $[x_1(t), x_2(t)]^\top$ sweeps out the contour $f_1(\mathbf{x}) = 0$ shown in the first picture. Substituting the above expressions into the formula for the objective,

$$\begin{aligned} f_0(x_1(t), x_2(t)) &= ((2 + t) - 1)^2 + ((4 - t^2) - 1)^2 \\ f_0(t) &= (1 + t)^2 + (3 - t^2)^2. \end{aligned}$$

This is just the reduced objective expressed in terms of t, and setting its derivative to zero like this

$$\begin{aligned} \frac{df_0}{dt} = 2(1 + t) + 2(3 - t^2)(-2t) &= 0 \\ 2 + 2t - 12t + 4t^3 &= 0 \\ 4t^3 - 10t + 2 &= 0 \end{aligned}$$

yields another cubic whose roots correspond to the stationary points we found before. But the parameterization also has an interesting geometric interpretation. If we let

$$\mathbf{g}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} 2 + t \\ 4 - t^2 \end{bmatrix} \quad \text{then} \quad \frac{d\mathbf{g}}{dt} = \begin{bmatrix} 1 \\ -2t \end{bmatrix}.$$

We can also write the constraint gradient as a function of t.

$$\nabla f_1(\mathbf{x}) = \begin{bmatrix} -2(x_1 - 2) \\ -1 \end{bmatrix} \quad \text{so} \quad \nabla f_1(t) = \begin{bmatrix} -2(x_1(t) - 2) \\ -1 \end{bmatrix} = \begin{bmatrix} -2((2 + t) - 2) \\ -1 \end{bmatrix} = \begin{bmatrix} -2t \\ -1 \end{bmatrix}$$

Now notice that

$$[\nabla f_1(\mathbf{x})]^\top \left[\frac{d\mathbf{g}}{dt}\right] = \begin{bmatrix} -2t & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -2t \end{bmatrix} = -2t + 2t = 0.$$

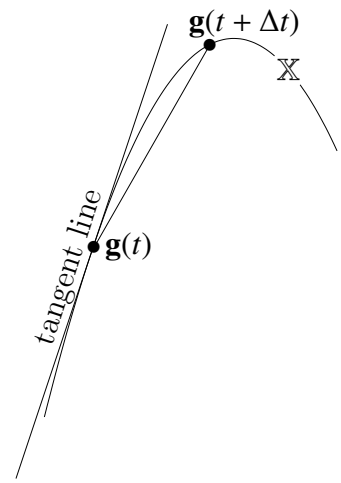
These vectors are orthogonal, which means that $d\mathbf{g}/dt$ is *tangent* to the curve $f_1(\mathbf{x}) = 0$. In other words, $d\mathbf{g}/dt$ is tangent to the feasible set $\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^2 \mid f_1(\mathbf{x}) = 0\}$.

On $\mathbb{X}$, $x_2 = 4 - (x_1 - 2)^2$ so the slope of the curve is

$$\frac{dx_2}{dx_1} = -2(x_1 - 2) = 4 - 2x_1.$$

Thus, for example, at $\mathbf{x} = [0, 0]^\top$ we have $dx_2/dx_1 = 4$ so $\mathbb{X}$, a curve in $\mathbb{R}^2$, has slope 4. At $\mathbf{x} = [0, 0]^\top$ we have $t = -2$ so $d\mathbf{g}/dt = [1, 4]^\top$ and this vector in $\mathbb{R}^2$ also has slope $4/1 = 4$. Because of the definition of the derivative,

$$\frac{d\mathbf{g}}{dt} = \lim_{\Delta t \to 0} \frac{\mathbf{g}(t + \Delta t) - \mathbf{g}(t)}{\Delta t}$$



is tangent to $\mathbb{X}$ for *every* value of t. As Δt approaches zero the chord in the picture to the right approaches the tangent line, so that is the direction of $d\mathbf{g}/dt$.

Earlier we noticed that the gradient of the objective is orthogonal to $\mathbb{X}$ at $\mathbf{x}^\star$. But $d\mathbf{g}/dt$ is tangent to $\mathbb{X}$, so $\nabla f_0(\mathbf{x})$ must be orthogonal to $d\mathbf{g}/dt$ at $\mathbf{x}^\star$. The objective gradient is

$$\nabla f_0(\mathbf{x}) = \begin{bmatrix} 2(x_1 - 1) \\ 2(x_2 - 1) \end{bmatrix} = \begin{bmatrix} 2([2 + t] - 1) \\ 2([4 - t^2] - 1) \end{bmatrix} = \begin{bmatrix} 2 + 2t \\ 6 - 2t^2 \end{bmatrix}$$

so at $\mathbf{x}^\star$ we must have

$$\begin{aligned} [\nabla f_0(t)]^\top \left[\frac{d\mathbf{g}}{dt}\right] = \begin{bmatrix} 2 + 2t & 6 - 2t^2 \end{bmatrix} \begin{bmatrix} 1 \\ -2t \end{bmatrix} &= 0 \\ (2 + 2t) + (6 - 2t^2)(-2t) &= 0 \\ 4t^3 - 10t + 2 &= 0. \end{aligned}$$

This is the same cubic we found before by minimizing the parameterized objective.

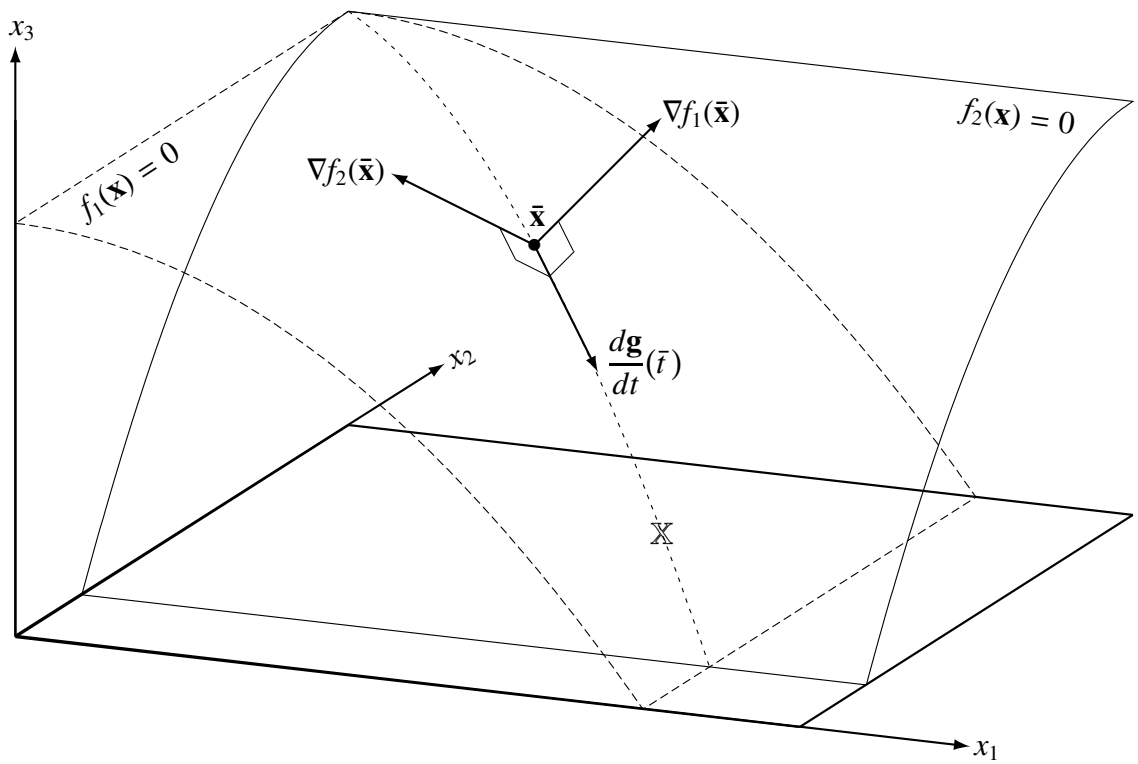
We have shown for this problem that if we can write $\mathbf{x} = \mathbf{g}(t)$, then $d\mathbf{g}/dt$ is a vector that is tangent to $\mathbb{X}$ and therefore orthogonal to $\nabla f_1(t)$ everywhere and to $\nabla f_0(t)$ at $t^\star$. Then we can use the collinearity of $\nabla f_1(t^\star)$ and $\nabla f_0(t^\star)$ to find $t^\star$, and the parameterization to find $\mathbf{x}^\star$.

15.2 The Lagrange Multiplier Theorem

The parameterization approach can be generalized to solve problems having $m > 1$ equality constraints, without using the constraints to explicitly eliminate m of the variables. An equality-constrained nonlinear program

$$\begin{aligned} \underset{\mathbf{x} \in \mathbb{R}^n}{\text{minimize}} \quad & f_0(\mathbf{x}) \\ \text{subject to} \quad & f_i(\mathbf{x}) = 0 \quad \text{for} \quad i = 1 \ldots m \end{aligned}$$

has the feasible set $\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n \mid f_i(\mathbf{x}) = 0,\ i = 1 \ldots m\}$, which is the intersection of the m hypersurfaces $f_i(\mathbf{x}) = 0$ in $\mathbb{R}^n$. For example, if $n = 3$ and $m = 2$ then $\mathbb{X}$ is the curve that is the intersection of two constraint hypersurfaces, as pictured below.



In general $\mathbb{X}$ is of dimension $n - m$, so we need $n - m$ parameters t_p to describe it. Suppose we parameterize $\mathbb{X}$ by letting $x_j = g_j(\mathbf{t})$ where $j = 1 \ldots n$ and $\mathbf{t} \in \mathbb{R}^{n-m}$. Then

$$\mathbf{x} = \mathbf{g}(\mathbf{t}) = \begin{bmatrix} g_1(\mathbf{t}) \\ \vdots \\ g_n(\mathbf{t}) \end{bmatrix} \quad \text{and} \quad f_0(\mathbf{x}) = f_0(g_1(\mathbf{t}) \ldots g_n(\mathbf{t}))$$

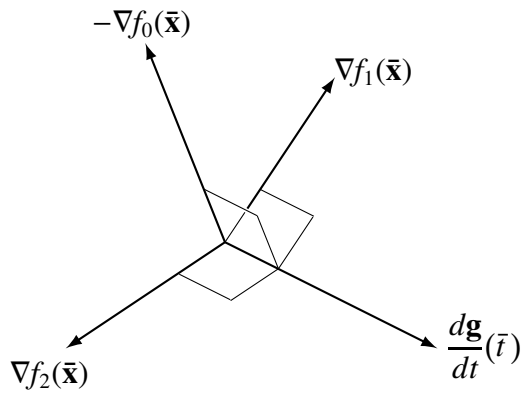
so, by the chain rule,

$$\begin{aligned} \frac{\partial f_0}{\partial t_p} &= \frac{\partial f_0}{\partial x_1}\frac{\partial g_1}{\partial t_p} + \cdots + \frac{\partial f_0}{\partial x_n}\frac{\partial g_n}{\partial t_p} \\ &= \nabla f_0(\mathbf{x})^\top \begin{bmatrix} \frac{\partial g_1}{\partial t_p} \\ \vdots \\ \frac{\partial g_n}{\partial t_p} \end{bmatrix} = \nabla f_0(\mathbf{x})^\top \left[\frac{\partial \mathbf{g}}{\partial t_p}\right] \quad \text{for } p = 1 \ldots n - m. \end{aligned}$$

Each vector $[\partial\mathbf{g}/\partial t_p]$ is tangent to $\mathbb{X}$. In the picture above $n-m=1$ so there is one parameter t, the feasible set $\mathbb{X}$ is the curve where the surfaces intersect, and $[d\mathbf{g}/dt]$ is tangent to it. Because each vector $[\partial\mathbf{g}/\partial t_p]$ is tangent to $\mathbb{X}$, each must be orthogonal to all of the constraint gradients. In the picture, $[d\mathbf{g}/dt\,]$ is orthogonal at $\bar{t}$ to both $\nabla f_1(\bar{\mathbf{x}})$ and $\nabla f_2(\bar{\mathbf{x}})$.

If $\bar{\mathbf{x}} = \mathbf{g}(\bar{\mathbf{t}})$ is a local minimizing point then it is a stationary point of $f_0(\mathbf{t})$, so $\partial f_0/\partial t_p = 0$ for $p = 1\ldots n-m$. Then $0 = \nabla f_0(\bar{\mathbf{x}})^\top[\partial\mathbf{g}/\partial t_p]$, and each $[\partial\mathbf{g}/\partial t_p]$ is orthogonal to $\nabla f_0(\bar{\mathbf{x}})$ also.

In the picture I omitted objective contours for clarity but they are also hypersurfaces, and if $\bar{\mathbf{x}}$ is a minimizing point the objective contour passing through $\bar{\mathbf{x}}$ is tangent to $\mathbb{X}$ so its gradient is orthogonal to $[d\mathbf{g}/dt]$. For this example the orthogonality of all three gradients to $[d\mathbf{g}/dt]$ looks (from a more convenient angle) like this, so the three gradients all lie in the same 2-dimensional hyperplane (see Exercise 15.6.13).



In general $\nabla f_0(\bar{\mathbf{x}}), \nabla f_1(\bar{\mathbf{x}})\ldots\nabla f_m(\bar{\mathbf{x}})$ all lie in the same m-dimensional hyperplane, so if the constraint gradients are linearly independent (see §28.2.4) then the objective gradient can be written as a linear combination of them, like this.

$$-\nabla f_0(\bar{\mathbf{x}}) = \lambda_1\nabla f_1(\bar{\mathbf{x}}) + \cdots + \lambda_m\nabla f_m(\bar{\mathbf{x}})$$

For a given set of constraint equalities it might be hard to find a parameterization $\mathbf{x} = \mathbf{g}(\mathbf{t})$ for which the system of equations $\nabla f_0(\mathbf{t})^\top[\partial\mathbf{g}/\partial t_p] = 0,\ p = 1\ldots n-m$, can be solved analytically, so it might seem that we are back almost where we began when we found it impossible to use the equalities to eliminate m of the variables analytically. Fortunately, *it is never actually necessary to find or use a parameterization.* If the constraint gradients are linearly independent then all that is needed to be able to write the objective gradient as a linear combination of the constraint gradients is that *some parameterization exists.* Whether that is true for a given problem is answered by the **implicit function theorem** [148, p571-579]. In the context of equality-constrained nonlinear programming, the hypotheses and conclusions of the implicit function theorem are incorporated into the **Lagrange multiplier theorem** [110, §7.2] stated at the top of the next page. What we noticed about the gradients in the examples discussed above is true in general if the hypotheses of the Lagrange multiplier theorem are satisfied.

Theorem: existence of Lagrange multipliers

given the NLP $\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x})$
subject to $f_i(\mathbf{x}) = 0, \quad i = 1\ldots m,$

if $\bar{\mathbf{x}}$ is a local minimizing point for NLP
$n > m$ (there are more variables than constraints)
the $f_i(\mathbf{x})$ have continuous first partials with respect to the x_j
the $\nabla f_i(\bar{\mathbf{x}}),\ i = 1\ldots m$, are linearly independent

then there exists a vector $\boldsymbol{\lambda} \in \mathbb{R}^m$ such that

$$\nabla f_0(\bar{\mathbf{x}}) + \sum_{i=1}^{m} \lambda_i \nabla f_i(\bar{\mathbf{x}}) = \mathbf{0}.$$

The requirement that the constraint gradients be linearly independent is called a **constraint qualification**, and the scalars λ_i are called **Lagrange multipliers**.

15.3 The Method of Lagrange

The Lagrange multiplier theorem suggests the following systematic procedure for finding analytically the local minimizing points of an equality-constrained nonlinear program [78, §3.2].

1. Verify that
   $n > m$ and
   for $i = 1\ldots m$ and $j = 1\ldots n$ the derivative $\partial f_i/\partial x_j$ is a continuous function of $\mathbf{x}$.

2. Form the **Lagrangian** function $\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x})$.

3. Find *all* solutions $(\bar{\mathbf{x}},\bar{\boldsymbol{\lambda}})$ to these **Lagrange conditions**.

$$\begin{aligned} \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) &= \nabla f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla f_i(\mathbf{x}) = \mathbf{0} \\ \frac{\partial \mathcal{L}}{\partial \lambda_i} &= f_i(\mathbf{x}) = 0, \quad i = 1\ldots m \end{aligned}$$

   The first or **stationarity condition** provides n equations and the second or **feasibility condition**, which can also be written $\nabla_{\boldsymbol{\lambda}}\mathcal{L} = \mathbf{0}$, provides m equations, and together these are enough to determine the n components of $\bar{\mathbf{x}}$ and the m components of $\bar{\boldsymbol{\lambda}}$.

4. Verify that the constraint gradients are linearly independent at the points $\bar{\mathbf{x}}$.

5. Classify the solutions $(\bar{\mathbf{x}},\bar{\boldsymbol{\lambda}})$ to identify the local minimizing points.

We can solve the `arch1` problem of §15.0 by using the method of Lagrange, as follows.

1. Verify that $n > m$: $2 > 1$ ✓

2. Verify that the partial derivatives are continuous:

$$\frac{\partial f_0}{\partial x_1} = 2(x_1 - 1) \qquad \frac{\partial f_0}{\partial x_2} = 2(x_2 - 1) \qquad \frac{\partial f_1}{\partial x_1} = -2(x_1 - 2) \qquad \frac{\partial f_1}{\partial x_2} = -1$$

These functions are all continuous. ✓

3. Form the Lagrangian.

$$\mathcal{L}(\mathbf{x}, \lambda) = (x_1 - 1)^2 + (x_2 - 1)^2 + \lambda(4 - (x_1 - 2)^2 - x_2)$$

4. Solve the Lagrange conditions.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_1} &= 2(x_1 - 1) - 2\lambda(x_1 - 2) = 0 \\ \frac{\partial \mathcal{L}}{\partial x_2} &= 2(x_2 - 1) - \lambda = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= 4 - (x_1 - 2)^2 - x_2 = 0 \end{aligned}$$

Substituting $\lambda = 2(x_2 - 1)$ and $x_2 = 4 - (x_1 - 2)^2$ into the first equation and simplifying yields

$$4x_1^3 - 24x_1^2 + 38x_1 - 10 = 0$$

which is the same cubic we found in §15.0. The **Lagrange points** $(\bar{\mathbf{x}}, \bar{\lambda})$ are thus the same points we found before.

$\bar{x}_1$	$\bar{x}_2$	$\bar{\lambda}$
0.32702	1.20113	0.40226
2.20336	3.95864	5.91728
3.46962	1.84022	1.68044

For this problem $\bar{\lambda}$ turns out to be positive at each Lagrange point, but in general a Lagrange multiplier for an equality-constrained problem can have either sign (see Exercise 15.6.24).

5. Verify that the constraint gradients are linearly independent at $\bar{\mathbf{x}}$: since there is only one constraint and $\nabla f_1(\bar{\mathbf{x}}) \neq \mathbf{0}$, that gradient is linearly independent. ✓

6. Classify the Lagrange points to identify the local minimizing points: in §15.0 we argued based on the second derivative of the reduced objective that the first and last points on the list above are minima, and based on the function value at those two points that the first one is the global minimum.

Lagrange multipliers play the same role in equality-constrained nonlinear programming that dual variables play in linear programming, and here also they can be interpreted as shadow prices [78, §3.3] (also see §16.9). Recall from §5.1.4 that the shadow price associated with a constraint is the change in the optimal objective value that results from changing the right-hand side of the constraint by one unit.

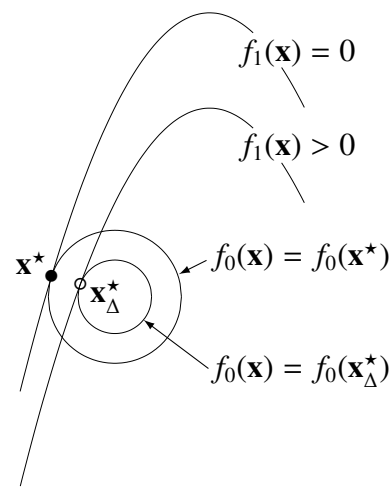
Suppose that in `arch1` we relax the constraint enough to move the optimal point to

$$\mathbf{x}^\star_\Delta = \mathbf{x}^\star + \mathbf{\Delta}$$

where the vector

$$\mathbf{\Delta} = \delta\,\nabla f_1(\mathbf{x}^\star) = \delta\begin{bmatrix} -2(x_1^\star - 2) \\ -1 \end{bmatrix} = \begin{bmatrix} \delta(4 - 2x_1^\star) \\ -\delta \end{bmatrix}$$

is orthogonal to the constraint contour. This changes the graphical solution as shown on the right. To compute the shadow price associated with the constraint we need $f_0(\mathbf{x}^\star_\Delta)$ and $f_1(\mathbf{x}^\star_\Delta)$ as functions of δ.



$$\begin{aligned}
f_0(\mathbf{x}^\star_\Delta) &= ([x_1 + \delta(4 - 2x_1)] - 1)^2 + ([x_2 - \delta] - 1)^2 \\
&= (x_1 - 1)^2 + (x_2 - 1)^2 + \delta^2(4 - 2x_1)^2 + \delta^2 + 2\delta(4 - 2x_1)(x_1 - 1) - 2\delta(x_2 - 1) \\
&= f_0(\mathbf{x}^\star) + \delta^2(4 - 2x_1)^2 + \delta^2 + 2\delta(4 - 2x_1)(x_1 - 1) - 2\delta(x_2 - 1) \\
f_1(\mathbf{x}^\star_\Delta) &= 4 - ([x_1 + \delta(4 - 2x_1)] - 2)^2 - [x_2 - \delta] \\
&= 4 - (x_1 - 2)^2 - x_2 - \delta^2(4 - 2x_1)^2 - 2\delta(4 - 2x_1)(x_1 - 2) + \delta \\
&= f_1(\mathbf{x}^\star) - \delta^2(4 - 2x_1)^2 - 2\delta(4 - 2x_1)(x_1 - 2) + \delta
\end{aligned}$$

The change in the objective value per unit change in the constraint value is then

$$\frac{f_0(\mathbf{x}^\star_\Delta) - f_0(\mathbf{x}^\star)}{f_1(\mathbf{x}^\star_\Delta) - f_1(\mathbf{x}^\star)} = \frac{\delta^2(4 - 2x_1)^2 + \delta^2 + 2\delta(4 - 2x_1)(x_1 - 1) - 2\delta(x_2 - 1)}{-\delta^2(4 - 2x_1)^2 - 2\delta(4 - 2x_1)(x_1 - 2) + \delta}.$$

Dividing numerator and denominator by δ and taking the limit as $\delta \to 0$, we find the shadow price

$$\frac{\partial f_0}{\partial f_1} = \frac{2(4 - 2x_1^\star)(x_1^\star - 1) - 2(x_2^\star - 1)}{-2(4 - 2x_1^\star)(x_1^\star - 2) + 1} \approx -0.40226$$

which is the negative of the $\lambda^\star$ we reported earlier. (increasing f_1 lets us decrease f_0). We can [161, §3.2] use the definition of the Lagrangian to show that in general

$$\boxed{\frac{\partial f_0}{\partial f_i} = -\lambda_i.}$$

$$\begin{aligned}\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) &= f_0(\mathbf{x}) + \sum_{p=1}^{m} \lambda_p f_p(\mathbf{x}) \\ f_0(\mathbf{x}) &= -\sum_{p=1}^{m} \lambda_p f_p(\mathbf{x}) + \mathcal{L}(\mathbf{x},\boldsymbol{\lambda}).\end{aligned}$$

Differentiating with respect to f_i,

$$\frac{\partial f_0}{\partial f_i} = -\lambda_i + \frac{\partial \mathcal{L}}{\partial f_i}$$

which is the result we want if the second term is zero. We can think of computing $\partial\mathcal{L}/\partial f_i$ by relaxing the ith constraint, finding $\mathcal{L}(\mathbf{x}^\star_\delta, \boldsymbol{\lambda}^\star_\delta)$ and $f_i(\mathbf{x}^\star_\delta)$, forming the ratio of the changes to $\mathcal{L}$ and f_i, and taking the limit as $\delta \to 0$, as in the example above. That makes $\mathcal{L}$ and f_i both functions of δ, so that

$$\frac{\partial \mathcal{L}}{\partial f_i} = \frac{\partial \mathcal{L}/\partial \delta}{\partial f_i/\partial \delta}.$$

Each of the derivatives with respect to δ is really a directional derivative in the direction $\nabla f_i(\mathbf{x}^\star)$, so using the result from §12.2.1 we can find them like this.

$$\begin{aligned}\partial\mathcal{L}/\partial\delta &= \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}^\star)^\top \nabla f_i(\mathbf{x}^\star) \\ \partial f_i/\partial\delta &= \nabla f_i(\mathbf{x})^\top \nabla f_i(\mathbf{x}^\star)\end{aligned}$$

At $\mathbf{x}^\star$ the gradient of the Lagrangian is zero so the first of these derivatives is zero, and at $\mathbf{x}^\star$ the derivative $\partial f_i/\partial\delta$ is the square of the norm of the constraint gradient. If the hypotheses of the Lagrange multiplier theorem are satisfied then the constraint gradients are linearly independent at $\mathbf{x}^\star$ so $\nabla f_i(\mathbf{x}^\star) \neq \mathbf{0}$ (see Exercise 15.6.20); thus $\partial\mathcal{L}/\partial f_i = 0$ and the result is established. If $\nabla f_i(\mathbf{x}^\star) = \mathbf{0}$ and also $\nabla f_0(\mathbf{x}^\star) = \mathbf{0}$ then the constraint is inactive so $\lambda_i = 0$.

In using the method of Lagrange it is often difficult to be sure that you have found *all* solutions to the Lagrange conditions. In `arch1` the three algebraic equations were equivalent to a single cubic, which we know from the fundamental theorem of algebra [8, Exercise 16.15] has exactly three roots. Some of the roots might have turned out to be complex (and therefore not meaningful for the optimization problem) or repeated, but at least we could be sure that we had found them all. Usually the Lagrange conditions involve functions other than polynomials, and then it might not be obvious even how many solutions there are. Numerical methods are typically required in solving the Lagrange conditions for real problems, and sometimes they are helpful even for toy problems like `arch1`, so as discussed in §8.3 it is often more convenient to apply a numerical minimization algorithm from the outset. Using many ideas from this Chapter, we will begin our study of algorithms for equality-constrained nonlinear programs in §18.

15.4 Classifying Lagrange Points Analytically

Another practical difficulty in using the method of Lagrange is classifying the solutions to the Lagrange conditions once they have all been found. If the other hypotheses of the Lagrange multiplier theorem are satisfied then every local minimum is a Lagrange point, but not every Lagrange point is necessarily a local minimum (as illustrated by the `arch1` example) nor even a stationary point [74, p62].

15.4.1 Problem-Specific Arguments

Sometimes it is possible to prove that a Lagrange point $\bar{\mathbf{x}}$ is a local minimum by using particular characteristics of the problem or of the point.

- If $n = 2$, a contour plot like the one in §15.0 can be used to approximate, and thereby identify as a minimum, a point that has been found analytically by using the method of Lagrange.
- If the problem is known to have a minimizing point and the Lagrange conditions can be shown to have a *unique* solution, then because every local minimum is a Lagrange point the unique Lagrange point must be the minimizing point.
- If the Lagrange points are known to all be stationary points, the one yielding the lowest value of the objective must be the constrained minimizing point.
- If the objective function is convex and the constraints are linear, the problem is a convex program; at a Lagrange point the constraint gradients must be linearly independent, so the Lagrange points are global minima (see §16.6).

Usually no such *ad hoc* argument is possible, and resort must be made to one of the more general approaches described next.

15.4.2 Testing the Reduced Objective

In studying our example we derived two equivalent formulas for the reduced objective, one in terms of $\mathbf{x}$ and the other in terms of t.

$$\begin{aligned} f_0(x_1) &= (x_1-1)^2 + (3-(x_1-2)^2)^2 & f_0''(x_1) &= 12x_1^2 - 48x_1 + 38 \\ f_0(t) &= (1+t)^2 + (3-t^2)^2 & f_0''(t) &= 12t^2 - 10 \end{aligned}$$

Because we knew $f_0(x_1)$ we were able using the MATLAB program `cubslv.m` in §15.0 to calculate $f_0''(\bar{x}_1)$ and, based on §10.7, to classify the three stationary points by the sign of this second derivative. Because we know a parameterization of the constraints we can do the same thing using $f_0''(\bar{t})$. All of these results are summarized on the next page.

$\bar{x}_1$	$\bar{x}_2$	$f''(\bar{\mathbf{x}})$	$\bar{t}$	$f''(\bar{t})$
0.32702	1.20113	23.586	-1.67298	23.586
2.20336	3.95864	-9.504	0.20336	-9.504
3.46962	1.84022	15.917	1.46962	15.917

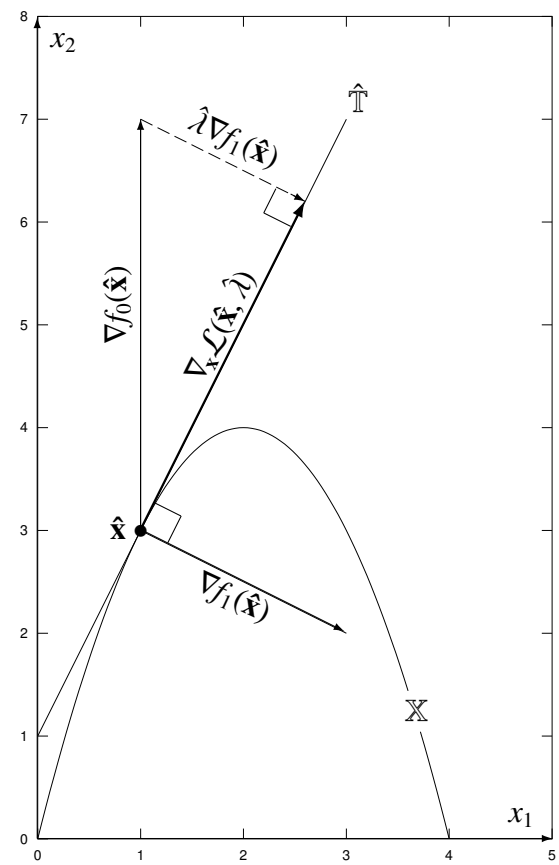
Either way we see that the second point is a maximum and the others are minima. If $n - m$ had been greater than 1 it would have been necessary to check the definiteness of the $(n - m) \times (n - m)$ Hessian matrix of the reduced objective.

This approach is seldom useful in practice, because usually we can't solve the constraints to find a reduced objective in terms of $\mathbf{x}$ or parameterize them to find a reduced objective in terms of $\mathbf{t}$. However, the idea that we might check the Hessian of a reduced objective motivates the easier (though still complicated) approach of the next Section.

15.4.3 Second Order Conditions

Suppose we construct a hyperplane $\hat{\mathbb{T}}$ that is tangent to the feasible set $\mathbb{X}$ at a point $\hat{\mathbf{x}} \in \mathbb{X}$. For $\hat{\mathbb{T}}$ to be tangent to $\mathbb{X}$ at $\hat{\mathbf{x}}$ it must be orthogonal to each of the constraint gradients there and pass through $\hat{\mathbf{x}}$, so

$$\hat{\mathbb{T}} = \{\mathbf{x} \in \mathbb{R}^n \mid \nabla f_i(\hat{\mathbf{x}})^\top(\mathbf{x} - \hat{\mathbf{x}}) = 0 \text{ for } i = 1 \ldots m\}.$$



For a given feasible point $\hat{\mathbf{x}}$, points $\mathbf{x}$ that are on $\hat{\mathbb{T}}$ must satisfy these m linear equations in the n variables x_j. We will assume the constraint gradients are linearly independent, so that we could solve this system to express m of the variables in terms of the others. The graph on the right pictures a hyperplane $\hat{\mathbb{T}}$ that is tangent at $\hat{\mathbf{x}} = [1, 3]^\top$ to the contour $f_1(\mathbf{x}) = 0$ in the `arch1` problem.

Now consider the gradient of the Lagrangian at $\hat{\mathbf{x}}$.

$$\nabla_{\mathbf{x}} \mathcal{L}(\hat{\mathbf{x}}, \hat{\boldsymbol{\lambda}}) = \nabla f_0(\hat{\mathbf{x}}) + \sum_{i=1}^{m} \hat{\lambda}_i \nabla f_i(\hat{\mathbf{x}})$$

By the construction of $\hat{\mathbb{T}}$, the gradients $\nabla f_i(\hat{\mathbf{x}})$ are each orthogonal to $\hat{\mathbb{T}}$; in `arch1`, $\nabla f_1(\hat{\mathbf{x}})$ is orthogonal to $\hat{\mathbb{T}}$ as shown. In the gradient of the Lagrangian, the term

$$\sum_{i=1}^{m} \hat{\lambda}_i \nabla f_i(\hat{\mathbf{x}})$$

is a linear combination of vectors orthogonal to $\hat{\mathbb{T}}$, so it is also orthogonal to $\hat{\mathbb{T}}$; in other words,

its orthogonal projection on $\hat{\mathbb{T}}$ is the zero vector. Thus, at any point on $\mathbb{X}$, assuming as we did that the constraint gradients are linearly independent, *the gradient of the Lagrangian is the orthogonal projection on* $\hat{\mathbb{T}}$ *of the gradient of the objective* [78, §3.6].

The graph on the previous page provides a geometric demonstration of the vector identity

$$\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\lambda}) = \nabla f_0(\hat{\mathbf{x}}) + \hat{\lambda}\nabla f_1(\hat{\mathbf{x}})$$

and shows that $\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\lambda})$ is the orthogonal projection of $\nabla f_0(\hat{\mathbf{x}})$ onto $\hat{\mathbb{T}}$. In general each $\nabla f_i(\hat{\mathbf{x}})$ is orthogonal to $\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\boldsymbol{\lambda}})$ so

$$\nabla f_i(\hat{\mathbf{x}})^{\mathsf{T}}\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\boldsymbol{\lambda}}) = 0 \quad \text{for} \quad i = 1\ldots m,$$

and these equations determine the $\hat{\lambda}_i$. For `arch1`, we have

$$\nabla f_1(\hat{\mathbf{x}}) = \begin{bmatrix} -2(\hat{x}_1 - 2) \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix} \qquad \nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\lambda}) = \begin{bmatrix} 2(\hat{x}_1 - 1) \\ 2(\hat{x}_2 - 1) \end{bmatrix} + \hat{\lambda}\begin{bmatrix} -2(\hat{x}_1 - 2) \\ -1 \end{bmatrix} = \begin{bmatrix} 0 + 2\hat{\lambda} \\ 4 - \hat{\lambda} \end{bmatrix}.$$

These vectors are orthogonal so it must be that at this $\hat{\mathbf{x}}$ we have

$$\begin{bmatrix} 2 & -1 \end{bmatrix}\begin{bmatrix} 0 + 2\hat{\lambda} \\ 4 - \hat{\lambda} \end{bmatrix} = 2(0 + 2\hat{\lambda}) - 1\,(4 - \hat{\lambda}) = 5\hat{\lambda} - 4 = 0 \quad \text{or} \quad \hat{\lambda} = \tfrac{4}{5}.$$

Thus the vectors pictured on the previous page are these.

$$\nabla f_0(\hat{\mathbf{x}}) = \begin{bmatrix} 2(\hat{x}_1 - 1) \\ 2(\hat{x}_2 - 1) \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix} \qquad \lambda\nabla f_1(\hat{\mathbf{x}}) = \begin{bmatrix} \frac{8}{5} \\ -\frac{4}{5} \end{bmatrix} \qquad \nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\lambda}) = \begin{bmatrix} \frac{8}{5} \\ \frac{16}{5} \end{bmatrix}$$

When we solve the Lagrange conditions we are finding points $(\bar{\mathbf{x}},\bar{\boldsymbol{\lambda}})$ where the orthogonal projection of $\nabla f_0(\bar{\mathbf{x}})$ onto $\overline{\mathbb{T}}$ is *zero* (you can convince yourself that this happens by imagining what the construction on the previous page would look like at $\mathbf{x}^\star$ in the first picture of §15.1).

There is nothing special about $\hat{\mathbf{x}}$ except that it is on $\mathbb{X}$, so imagine now that we construct the tangent hyperplane $\hat{\mathbb{T}}$ at some arbitrary point $(x_1, x_2) \in \mathbb{X}$. There $x_2 = 4 - (x_1 - 2)^2$, so $\hat{\mathbb{T}}$ is a line with slope $dx_2/dx_1 = -2(x_1 - 2) = 4 - 2x_1$. The Lagrangian and its gradient are as we found earlier.

$$\begin{aligned} f_0(\mathbf{x}) &= (x_1 - 1)^2 + (x_2 - 1)^2 \\ f_1(\mathbf{x}) &= 4 - (x_1 - 2)^2 - x_2 \\ \text{so} \quad \mathcal{L} &= (x_1 - 1)^2 + (x_2 - 1)^2 + \lambda\left[4 - (x_1 - 2)^2 - x_2\right] \\ \text{and} \quad \nabla_{\mathbf{x}}\mathcal{L} &= \begin{bmatrix} 2(x_1 - 1) \\ 2(x_2 - 1) \end{bmatrix} + \lambda\begin{bmatrix} -2(x_1 - 2) \\ -1 \end{bmatrix} = \begin{bmatrix} \dfrac{\partial\mathcal{L}}{\partial x_1} \\ \dfrac{\partial\mathcal{L}}{\partial x_2} \end{bmatrix} \end{aligned}$$

The vectors $\nabla f_1(\mathbf{x})$ and $\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x},\boldsymbol{\lambda})$ are still orthogonal, so

$$\begin{bmatrix} -2(x_1-2) & -1 \end{bmatrix}\begin{bmatrix} 2(x_1-1)-2\lambda(x_1-2) \\ 2(x_2-1)-\lambda \end{bmatrix} = 0.$$

Computing the dot product and solving for λ we find that

$$\begin{aligned} -2(x_1-2)\,[2(x_1-1)-2\lambda(x_1-2)] - 1\,[2(x_2-1)-\lambda] &= 0 \\ -4(x_1-2)(x_1-1)+4\lambda(x_1-2)^2-2(x_2-1)+\lambda &= 0 \\ \lambda\left[4(x_1-2)^2+1\right] &= 4(x_1-2)(x_1-1)+2(x_2-1) \\ \lambda &= \frac{4(x_1-2)(x_1-1)+2(x_2-1)}{4(x_1-2)^2+1}. \end{aligned}$$

How does the value of the Lagrangian vary along the tangent line $\hat{\mathbb{T}}$ as we change x_1? Thinking of $\mathcal{L}$ on $\hat{\mathbb{T}}$ as a function of x_1 and $x_2(x_1)$, we find by the chain rule that

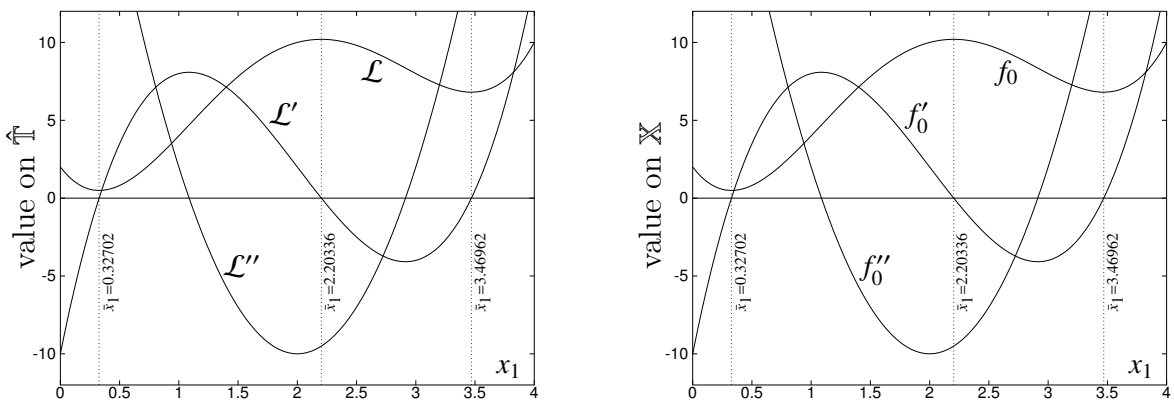
$$\begin{aligned} \mathcal{L}' &= \frac{\partial\mathcal{L}}{\partial x_1} + \frac{\partial\mathcal{L}}{\partial x_2}\frac{dx_2}{dx_1} \\ &= [2(x_1-1)-2\lambda(x_1-2)] + [2(x_2-1)-\lambda]\,[4-2x_1] \\ &= 2(x_1-1)-2\lambda(x_1-2)-4x_1x_2+4x_1+2\lambda x_1+8x_2-8-4\lambda \\ \mathcal{L}'' &= 2-2\lambda-4\left(x_1\frac{dx_2}{dx_1}+x_2\right)+4+2\lambda+8\frac{dx_2}{dx_1} \\ &= 2-4\,[x_1(4-2x_1)+x_2]+4+8(4-2x_1). \end{aligned}$$

where λ is given by the expression above. The graph on the left at the top of the next page shows how $\mathcal{L}$, $\mathcal{L}'$, and $\mathcal{L}''$ vary with x_1 on $\hat{\mathbb{T}}$.

Next recall the reduced objective and its derivatives, which we also found earlier.

$$\begin{aligned} f_0(\mathbf{x}) &= (x_1-1)^2+(x_2-1)^2 \\ \text{but}\quad x_2 &= 4-(x_1-2)^2 \\ \text{so}\quad f_0(x_1) &= (x_1-1)^2+(3-(x_1-2)^2)^2 \\ f_0'(x_1) &= 4x_1^3-24x_1^2+38x_1-10 \\ f_0''(x_1) &= 12x_1^2-48x_1+38 \end{aligned}$$

The graph on the right at the top of the next page shows how f_0, $f_0{}'$, and $f_0{}''$ vary with x_1 on $\mathbb{X}$. These pictures confirm that $\mathcal{L}(x_1) = f_0(x_1)$ (which is not surprising, since $f_1(\mathbf{x}) = 0$ on $\mathbb{X}$) and also show that $\mathcal{L}'(x_1) = f_0{}'(x_1)$ and $\mathcal{L}''(x_1) = f_0{}''(x_1)$ (see Exercise 15.6.31). The Lagrange points are the local minima and maximum of $\mathcal{L}(x_1) = f_0(x_1)$, located where $\mathcal{L}'(x_1) = f_0'(x_1) = 0$, and the sign of $\mathcal{L}''(x_1) = f_0{}''(x_1)$ at each Lagrange point indicates whether it is a minimum or a maximum.



It is true not just for this example but in general that *the Hessian of the Lagrangian on* $\hat{\mathbb{T}}$, which I will call $\mathbf{H}_{\mathcal{L}}$, *is precisely the Hessian of the reduced objective function on* $\mathbb{X}$. Thus, to classify Lagrange points based on the definiteness of the Hessian of the reduced objective, we can instead test the definiteness of the Hessian of the Lagrangian on $\hat{\mathbb{T}}$. Usually, that is much easier to do.

We defined the tangent hyperplane $\hat{\mathbb{T}}$ in such a way that it passes through $\hat{\mathbf{x}}$, but the orthogonal projection of $\nabla f_0(\hat{\mathbf{x}})$ onto *any* hyperplane parallel to $\hat{\mathbb{T}}$ would also be $\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\boldsymbol{\lambda}})$. In particular, we would reach the same conclusions if we projected the objective gradient onto the hyperplane

$$\mathbb{T} = \{\mathbf{x} \in \mathbb{R}^n \mid \nabla f_i(\hat{\mathbf{x}})^\top \mathbf{x} = 0 \quad \text{for} \quad i = 1 \ldots m\},$$

which passes through the origin instead of through $\hat{\mathbf{x}}$. We can therefore classify a Lagrange point $\bar{\mathbf{x}}$, based on the reduced objective at $\bar{\mathbf{x}}$, by determining the definiteness of the Hessian of the Lagrangian on $\mathbb{T}$, as described next [3, p284-286][110, §7.2].

Theorem: classification of Lagrange points

given the NLP $\quad \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x})$
$\quad$ subject to $f_i(\mathbf{x}) = 0, \quad i = 1 \ldots m,$

if $\quad \mathbb{T} = \{\mathbf{x} \in \mathbb{R}^n \mid \nabla f_i(\bar{\mathbf{x}})^\top \mathbf{x} = 0 \text{ for } i = 1 \ldots m\}$
$\quad (\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ is a Lagrange point
$\quad \mathbf{x}^\top \mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})\mathbf{x} > 0$ for all nonzero vectors $\mathbf{x} \in \mathbb{T}$

then $\quad \bar{\mathbf{x}}$ is a strict local minimizing point.

The hypotheses of this theorem are called the **second-order sufficient conditions** [5, Theorem 12.6] [4, Theorem 14.16] [107, §10.5], because they test the Hessian or second derivative of the Lagrangian on $\mathbb{T}$ and they are sufficient to ensure that a Lagrange point $\bar{\mathbf{x}}$ is a strict local minimum.

We can classify the Lagrange points of the `arch1` problem using this theorem as follows.

$$\begin{aligned}
\nabla f_1(\bar{\mathbf{x}})^\top \mathbf{x} &= \begin{bmatrix} -2(\bar{x}_1 - 2), & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = -2(\bar{x}_1 - 2)x_1 - x_2 = 0 \\
\text{so} \quad \mathbb{T} &= \left\{ \mathbf{x} \in \mathbb{R}^2 \mid x_2 = -2(\bar{x}_1 - 2)x_1 \right\} \\
\mathcal{L}(\mathbf{x}, \bar{\lambda}) &= (x_1 - 1)^2 + (x_2 - 1)^2 + \bar{\lambda}(4 - (x_1 - 2)^2 - x_2).
\end{aligned}$$

On $\mathbb{T}$ the Lagrangian is thus

$$\mathcal{L}_{\mathbb{T}}(x_1, \bar{\lambda}) = (x_1 - 1)^2 + ([-2(\bar{x}_1 - 2)x_1] - 1)^2 + \bar{\lambda}(4 - (x_1 - 2)^2 - [-2(\bar{x}_1 - 2)x_1]).$$

Because $n - m = 1$ the Hessian of this Lagrangian is just its second derivative.

$$\begin{aligned}
\frac{d\mathcal{L}_{\mathbb{T}}}{dx_1} &= 2(x_1 - 1) + 2(-2(\bar{x}_1 - 2)x_1 - 1)[-2(\bar{x}_1 - 2)] + \bar{\lambda}[-2(x_1 - 2) + 2(\bar{x}_1 - 2)] \\
&= 2x_1 - 2 + 4(\bar{x}_1 - 2)(2(\bar{x}_1 - 2)x_1 + 1) + \bar{\lambda}(-2x_1 + 4 + 2\bar{x}_1 - 4) \\
&= 2x_1 - 2 + 8(\bar{x}_1 - 2)^2 x_1 + 4(\bar{x}_1 - 2) + 2\bar{\lambda}(\bar{x}_1 - x_1) \\
\frac{d^2\mathcal{L}_{\mathbb{T}}}{dx_1^2} &= 2 + 8(\bar{x}_1 - 2)^2 - 2\bar{\lambda} = h
\end{aligned}$$

Evaluating this quantity at the three stationary points of `arch1`, we find that the values it takes on match those we found earlier for $f''(\bar{\mathbf{x}})$ by using substitution and for $f''(\bar{t})$ by using parameterization.

$\bar{x}_1$	$\bar{\lambda}$	h	classification
0.32702	0.40226	23.586	$> 0 \Rightarrow$ minimum
2.20336	5.91728	-9.504	$< 0 \Rightarrow$ maximum
3.46962	1.68044	15.917	$> 0 \Rightarrow$ minimum

Remember that $\mathbf{H}_{\mathcal{L}}$ must be positive *definite* on $\mathbb{T}$ to ensure that $\bar{\mathbf{x}}$ is a strict *local* minimum. Just as the method of Lagrange is more likely to be analytically tractable than either substitution or parameterization for finding stationary points, using the second-order conditions is more likely to be analytically tractable for classifying them.

15.5 Classifying Lagrange Points Numerically

In §15.4.3 we defined the hyperplane tangent to the feasible set at a Lagrange point $\bar{\mathbf{x}}$ (translated to pass through the origin) by specifying the conditions that $\mathbf{x}$ must satisfy in order to be on it:

$$\mathbb{T} = \{\mathbf{x} \in \mathbb{R}^n \mid \nabla f_i(\bar{\mathbf{x}})^\top \mathbf{x} = 0 \text{ for } i = 1 \ldots m\}.$$

A different characterization of the points on $\mathbb{T}$, while less geometrically intuitive, is more convenient to use in numerical calculations (this approach is discussed in more detail in §22.1.1).

If we make the gradients of the constraints at $\bar{\mathbf{x}}$ the rows of an $m \times n$ matrix $\mathbf{A}$, then for $\mathbf{x}$ to be on $\mathbb{T}$ it must be in the **nullspace** [147, §2.4] of $\mathbf{A}$.

$$\mathbf{A} = \begin{bmatrix} \nabla f_1(\bar{\mathbf{x}})^\top \\ \vdots \\ \nabla f_m(\bar{\mathbf{x}})^\top \end{bmatrix} \quad \Rightarrow \quad \mathbb{T} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} = \mathbf{0}\}$$

The nullspace $\mathbb{T}$ of the matrix $\mathbf{A}$ is the $n-m$ dimensional subspace of $\mathbb{R}^n$ on which $\mathbf{A}\mathbf{x} = \mathbf{0}$. If linearly-independent vectors $\mathbf{z}^p \in \mathbb{R}^n$ span that subspace, so that they form a basis for $\mathbb{T}$, then we can write any $\mathbf{x} \in \mathbb{T}$ as some combination $y_1\mathbf{z}^1 + \ldots + y_{n-m}\mathbf{z}^{n-m}$ of those basis vectors. In other words, if we make the basis vectors $\mathbf{z}^p$ the columns of an $n \times (n-m)$ matrix $\mathbf{Z}$, then every $\mathbf{x}$ that is on $\mathbb{T}$ can be written as $\mathbf{x} = \mathbf{Z}\mathbf{y}$ for some $\mathbf{y} \in \mathbb{R}^{n-m}$. Then to show that $\mathbf{x}^\top \mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})\mathbf{x} > 0$ for all nonzero vectors $\mathbf{x} \in \mathbb{T}$ it suffices to show that $(\mathbf{Z}\mathbf{y})^\top \mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})\mathbf{Z}\mathbf{y} = \mathbf{y}^\top \mathbf{Z}^\top \mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})\mathbf{Z}\mathbf{y} > 0$ for all nonzero vectors $\mathbf{y} \in \mathbb{R}^{n-m}$.

The quantity $\overline{\mathbf{H}} = \mathbf{Z}^\top \mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})\mathbf{Z}$ is called the **projected Hessian** [5, p337] of the Lagrangian, and the second-order condition for $\bar{\mathbf{x}}$ to be a local minimum is satisfied if this matrix is positive definite. To find $\overline{\mathbf{H}}$ we need to compute $\mathbf{Z}$, whose columns form a basis for the nullspace of $\mathbf{A}$. This basis is not unique so various algorithms have been contrived to find one (e.g., [150, Theorem 5.2], [147, §2.4.2N], [91, §2]) but for our purposes the MATLAB `null()` function [50, p381], which is based on the singular-value decomposition of $\mathbf{A}$, will do nicely. Once we know $\overline{\mathbf{H}}$ we can determine its definiteness by examining its eigenvalues as described in §11.5.

To implement this scheme I wrote the `socheck.m` routine listed at the top of the next page. The program begins 6-9 by computing the Hessian of the Lagrangian `HL` at the given Lagrange point (`x, lambda`). Then 11-14 it evaluates the constraint gradients to construct the `A` matrix, 15 finds a basis for the nullspace of `A`, and 16 uses it to compute `Hbar`. The final stanza of code 18-29 finds the eigenvalues of `Hbar` and decides based upon them whether to signal that `Hbar` is positive definite (`flag=1`) or positive semidefinite (`flag=0`) or neither (`flag=-1`).

When `socheck.m` is used to classify the Lagrange points we found for the `arch1` problem, it produces the output shown below. These results confirm our earlier determination (several times) that these points are a local minimum, a local maximum, and a local minimum.

```
octave:1> x=[0.32702;1.20113];
octave:2> lambda=0.40226;
octave:3> flag=socheck(1,x,lambda,@arch1g,@arch1h)
flag =  1
octave:4> x=[2.20336;3.95864];
octave:5> lambda=5.91728;
octave:6> flag=socheck(1,x,lambda,@arch1g,@arch1h)
flag = -1
octave:7> x=[3.46962;1.84022];
octave:8> lambda=1.68044;
octave:9> flag=socheck(1,x,lambda,@arch1g,@arch1h)
flag =  1
```



```
 1 function flag=socheck(m,x,lambda,grd,hsn)
 2 % classify a Lagrange point (x,lambda)
 3 % by examining the eigenvalues
 4 % of the projected Hessian of the Lagrangian
 5
 6   HL=hsn(x,0);                    % Hessian of objective
 7   for i=1:m                       % add in the sum of multiplier
 8       HL=HL+lambda(i)*hsn(x,i);   % times Hessian of constraint
 9   end                             % to get Hessian of Lagrangian
10
11   for i=1:m                       % construct the matrix
12       g=grd(x,i);                 % whose rows are
13       A(i,:)=g';                  % the constraint gradients
14   end                             % so that Ax=0 on T
15   Z=null(A);                      % get a basis for the nullspace
16   Hbar=Z'*HL*Z;                   % use it to project HL onto T
17
18   flag=+1;                        % assume Hbar will be pd
19   ev=eig(Hbar);                   % find the eigenvalues of Hbar
20   n=size(x,1);                    % number of variables
21   for p=1:n-m                     % check all eigenvalues of Hbar
22       if(abs(ev(p)) < 1e-8)       % if small assume zero
23         flag=0;                   % which makes Hbar psd
24         continue                  % and check the next eigenvalue
25       end                         % done checking for Hbar psd
26       if(ev(p) < 1e-8)            % if negative
27         flag=-1;                  % that makes Hbar not psd
28         break                     % no further checking is needed
29       end                         % done checking for Hbar not psd
30   end                             % done checking eigenvalues
31
32 end
```

The routines that `socheck` uses to compute gradients and Hessians for the `arch1` problem are listed below. The parameters passed into `arch1g.m` and `arch1h.m` are `x`, the point at which a gradient or Hessian is to be computed; and `i`, the index of the function whose gradient or Hessian is needed. The `switch` statement 4-9 computes the appropriate quantity depending on the `case` specified by the value of `i`. Thus, for example, for `case 0` 5-6 `arch1g.m` returns the gradient of f_0 and `arch1h.m` returns the Hessian of f_0.

```
 1 function g=arch1g(x,i)
 2 % return the gradient of function i
 3
 4   switch(i)
 5     case 0
 6       g=[2*(x(1)-1);2*(x(2)-1)];
 7     case 1
 8       g=[-2*(x(1)-2);-1];
 9   end
10
11 end
```

```
 1 function H=arch1h(x,i)
 2 % return the Hessian of function i
 3
 4   switch(i)
 5     case 0
 6       H=[2,0;0,2];
 7     case 1
 8       H=[-2,0;0,0];
 9   end
10
11 end
```

In future Chapters we will have many occasions to compute function, gradient, or Hessian values for nonlinear programs that have constraints, and I will always code those routines in this standard way.

The problem given below, which I will call `hill` (see §28.7.6), has constraint surfaces that resemble those pictured in §15.2.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 + x_3^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & 4 - \frac{1}{9}x_1^2 - x_3 = 0 \\ & f_2(\mathbf{x}) & = & 4 - \frac{4}{9}(4 - x_2)^2 - x_3 = 0 \end{array}$$

Both of its Lagrange points are minima.

```
octave:1> format long
octave:2> x=[3.23137107379720;2.38431446310140;2.83980455371408];
octave:3> lambda=[9;-3.32039089257184];
octave:4> flag=socheck(2,x,lambda,@hillg,@hillh)
flag =  1
octave:5> x=[-3.23137107379720;2.38431446310140;2.83980455371408];
octave:6> flag=socheck(2,x,lambda,@hillg,@hillh)
flag =  1
```

The `arch1` problem has $n = 2$ and $m = 1$, while `hill` has $n = 3$ and $m = 2$. Finally, consider the `one23` problem (see §28.7.7), which has $n = 3$ and $m = 1$.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 + x_2^2 + x_3^3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1 + x_2 + x_3 - 1 = 0 \end{array}$$

The Octave session below tests two Lagrange points, one a min and the other a max.

```
octave:7> xa=[-0.0773502691896257;0.5;0.5773502691896257];
octave:8> xb=[1.077350269189626;0.5;-0.577350269189626];
octave:9> lambda=-1;
octave:10> flag=socheck(1,xa,-1,@one23g,@one23h)
flag = 1
octave:11> f0a=xa(1)+xa(2)^2+xa(3)^3
f0a =  0.365099820540249
octave:12> flag=socheck(1,xb,-1,@one23g,@one23h)
flag = -1
octave:13> f0b=xb(1)+xb(2)^2+xb(3)^3
f0b =  1.13490017945975
octave:14> quit
```

15.6 Exercises

15.6.1 [E] What is a *reduced objective* of a nonlinear program? How is a reduced objective formed? What gets reduced in forming a reduced objective?

15.6.2 [E] Explain what the MATLAB function `fzero` does, and how to use it. When it is used in the `cubslv.m` program of §15.0 its final parameter is `xzero`. What is the meaning of that parameter, and what values does it take on when the program is run?

15.6.3 [H] In §15.0 we used the MATLAB program `cubslv.m` to solve a cubic equation for the values that x_1 takes on at the stationary points of the `arch1` problem. But as Cardano reports in his *Ars Magna*, first published in 1545 CE [163] it is possible to find the roots of a cubic equation as closed-form algebraic expressions. (a) Find on the internet the prescription for solving a cubic equation analytically. (b) Use it to derive formulas for the roots of our cubic, $4x^3 - 24x^2 + 38x - 10 = 0$. (c) Evaluate the formulas to confirm that the numerical solutions we found are correct. (d) Which approach do you prefer, the numerical or the analytic? Make an argument to support your view.

15.6.4 [P] In §15.0 we found three stationary points for the `arch1` problem, one of which corresponds to the optimal point we found graphically. (a) Explain the reasoning used there to classify the stationary points as the global minimum, a global maximum, and a local minimum. (b) Write a MATLAB program to graph the zero constraint contour and the objective contours passing through the other two stationary points. What is the graphical significance of the two stationary points that are not $\mathbf{x}^\star$?

15.6.5 [H] If we use the equality constraints of a nonlinear program to find formulas for m of the variables in terms of the other $n - m$ variables, then we can substitute those formulas into the objective and solve the resulting unconstrained optimization. Give examples of nonlinear equalities that *cannot* be used in this way (a) when $m = 2$; (b) when $m = 1$.

15.6.6 [E] In the example of §15.0, why do $\nabla f_0(\mathbf{x}^\star)$ and $\nabla f_1(\mathbf{x}^\star)$ point in opposite directions?

15.6.7 [H] Suppose the problem of §15.0 is modified to make $f_0(\mathbf{x}) = (x_1 + 1)^2 + (x_2 - 1)^2$. (a) Find the new $\mathbf{x}^\star$ numerically, and confirm your solution graphically. (b) Do $\nabla f_0(\mathbf{x}^\star)$ and $\nabla f_1(\mathbf{x}^\star)$ still point in opposite directions? Find $\lambda^\star$.

15.6.8 [P] In §15.1 we used a parametric representation of the feasible set. Write a MATLAB program that plots the feasible set using the command `plot(x1,x2)`, where `x1` and `x2` are vectors containing the x_1 and x_2 coordinates of points on the curve. To compute the vector elements `x1(p)` and `x2(p)` use a loop that finds the value of t corresponding to the `p`th point to be plotted and then the formulas for $x_1(t)$ and $x_2(t)$ to find the coordinates.

15.6.9 [P] In §15.1 we parameterized the constraint of the `arch1` problem by finding $\mathbf{g}(t) = [x_1(t), x_2(t)]^\top$ and we derived a cubic $4t^3 - 10t + 2 = 0$ whose roots are the stationary points $\bar{t}$. (a) Show that the points $\bar{t}$ correspond to the stationary points $\bar{\mathbf{x}}$ that we found for the problem. (b) At each stationary point compute $d\mathbf{g}/dt$ and $\nabla f_0(\mathbf{x})$, and show that the vectors are orthogonal. (c) Write a MATLAB program to graph the feasible set, and to draw at each stationary point the vector $d\mathbf{g}/dt$.

15.6.10 [E] How do we know in general that if $\mathbf{g}(t)$ is a parameterization of a constraint then the vector $d\mathbf{g}/dt$ is tangent to the zero contour of the constraint?

15.6.11 [H] In §15.1 we parameterized the constraint $f_1(\mathbf{x}) = 0$ of the `arch1` problem as $\mathbf{g}(t) = [2 + t, 4 - t^2]^\top$ and we found $f_0(t) = (1 + t)^2 + (3 - t^2)^2$. Show that $df_0/dt = \nabla f_0(t)^\top [d\mathbf{g}/dt]$ at every feasible point, and that $df_0/dt = 0$ at $t^\star$.

15.6.12 [E] In §15.2 we argued that at a minimizing point of an equality-constrained nonlinear program, the gradients of the constraints and the gradient of the objective all lie in the same m-dimensional hyperplane. (a) Outline the argument that we used to establish this fact. (b) What is required in order for it to be possible to write the gradient of the objective as a linear combination of the gradients of the constraints?

15.6.13 [P] The `hill` problem of §15.5 has constraint surfaces similar to those depicted in §15.2. (a) Find a parameterization of the feasible set $\mathbb{X}$. (b) Write $\mathbf{x} = \mathbf{g}(t)$ and $f_0(\mathbf{x}) = f_0(g_1(t), g_2(t))$. (c) Use the chain rule to find df_0/dt and show that it is equal to $\nabla f_0(\mathbf{x})^\top[d\mathbf{g}/dt]$. (d) Show that $\nabla f_0(\bar{\mathbf{x}})$, $\nabla f_1(\bar{\mathbf{x}})$, and $\nabla f_2(\bar{\mathbf{x}})$ are all orthogonal to $[d\mathbf{g}/dt]$ at $\bar{t}$ and thus lie in the same plane. (e) Write $\nabla f_0(\bar{\mathbf{x}})$ as a linear combination of the constraint gradients, and find λ_1 and λ_2. (f) Use the equation you found in part `e` and the constraints to solve the problem. (g) Use the method of Lagrange to solve the problem. Hint: Describe $\mathbb{X}$ by an equation relating x_1 and x_2. Use that result and a constraint to find x_3 as a function of x_2. Then write the objective in terms of x_2 only, and use the MATLAB function `fzero` to solve the resulting cubic. (h) Both Lagrange points of this problem are minima; explain how this is possible.

15.6.14 [H] In solving an equality-constrained nonlinear program, we can write the gradient of the objective as a linear combination of the gradients of the constraints if the constraint gradients are independent and some parameterization of the constraints exists. (a) Given a set of constraint gradients, how can you determine whether they are linearly independent? Describe a computational procedure. (b) How can you determine whether a parameterization of the constraints exists? (c) Is it ever necessary to find a parameterization of the constraints?

15.6.15 [E] State the Lagrange multiplier theorem. What is a constraint qualification? What is a Lagrange multiplier? What is a Lagrange point? What can you deduce about a feasible point $\hat{\mathbf{x}}$ if $n > m$, the $f_i(\mathbf{x})$ have continuous first partials with respect to the x_j at $\hat{\mathbf{x}}$, and the $\nabla f_i(\hat{\mathbf{x}})$ are linearly independent, but *no* set of numbers λ_i solves this system of linear equations?

$$\nabla f_0(\hat{\mathbf{x}}) + \sum_{i=1}^{m} \lambda_i \nabla f_i(\hat{\mathbf{x}}) = \mathbf{0}$$

15.6.16 [H] It is required to find the point on the curve described by $7x_1 - 3x_2^2 = 0$ that is closest to the point $[3, 1]^\top$. (a) Formulate this problem as a nonlinear program. (b) Use the method of Lagrange to find $\mathbf{x}^\star$. (c) Solve the problem graphically to check your answer.

15.6.17 [H] Use the method of Lagrange to solve this nonlinear program. The optimal value is zero.

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} \quad f_0(\mathbf{x}) &= 2x_1^2 + 5x_2^2 + 11x_3^2 + 20x_1x_2 - 4x_1x_3 + 16x_2x_3 + 9 \\ \text{subject to} \quad f_1(\mathbf{x}) &= x_1^2 + x_2^2 + x_3^2 = 1 \end{aligned}$$

15.6.18 [H] Use the method of Lagrange to solve this nonlinear program. There are four Lagrange points.

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} \quad f_0(\mathbf{x}) &= x_1^2 + x_2^2 + x_3^2 \\ \text{subject to} \quad f_1(\mathbf{x}) &= x_1x_2x_3 = 1 \end{aligned}$$

15.6.19 [H] Use the method of Lagrange to solve this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 - 2x_2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 - 1 = 0 \end{array}$$

15.6.20 [H] A collection of (one or more) vectors $\mathbf{y}^1 \ldots \mathbf{y}^m$ in $\mathbb{R}^m$ is **linearly independent** [1, p751] if and only if

$$\sum_{i=1}^{m} \lambda_i \mathbf{y}_i = \mathbf{0} \quad \Rightarrow \quad \lambda_i = 0 \quad \text{for} \quad i = 1 \ldots m.$$

(a) Explain why a single zero vector is not linearly independent but a single nonzero vector is. (b) Can a set of m vectors be linearly independent if any one of them is the zero vector? Explain. (c) Modify the constraint of the `arch1` problem to be $f_1(\mathbf{x}) = (x_1 - x_1^\star)^2 + (x_2 - x_2^\star)^2 = 0$ so that the feasible set consists of the single point $\mathbf{x}^\star$ and $\nabla f_1(\mathbf{x}^\star) = \mathbf{0}$. Write the Lagrange conditions for this problem, and show that they are not satisfied by $\mathbf{x}^\star$.

15.6.21 [H] The following nonlinear program has $n = 3 > 2 = m$, and all of its $\partial f_i / \partial x_j$ are continuous functions.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & x_3 - x_1^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_3 - x_2 - 3 = 0 \\ & f_2(\mathbf{x}) & = & x_3 + x_2 - 3 = 0 \end{array}$$

(a) Sketch the constraint contours and the $f_0(\mathbf{x}) = 0$ objective contour in $\mathbb{R}^3$. Label the feasible set $\mathbb{X}$ in your picture. (b) Write down the Lagrange conditions for this problem, calling the multiplier for the first constraint λ_1 and the multiplier for the second constraint λ_2. (c) Solve the Lagrange conditions to find $\bar{\mathbf{x}}$ and $\bar{\boldsymbol{\lambda}}$, and mark $\bar{\mathbf{x}}$ in your picture. (d) Confirm that the constraint gradients are linearly independent at $\bar{\mathbf{x}}$. (e) Is $\bar{\mathbf{x}}$ optimal for the nonlinear program? Explain.

15.6.22 [H] Apply the method of Lagrange to this nonlinear program [78, Example 2.2].

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^3 + x_1 x_2 - x_2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2 = 0 \end{array}$$

(a) Is the Lagrange point you found a stationary point? (b) Is it the constrained minimum? (c) Are the hypotheses of the Lagrange multiplier theorem satisfied? Explain.

15.6.23 [E] Suppose we use the method of Lagrange to solve a nonlinear program having equality constraints and $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ is the optimal Lagrange point. If $\nabla f_i(\mathbf{x}^\star) \neq \mathbf{0}$, what is the shadow price associated with the constraint $f_i(\mathbf{x}) = 0$?

15.6.24 [H] When the method of Lagrange is used to solve the `arch1` problem, $\lambda^\star$ turns out to be positive. Use the method of Lagrange to solve the following problem, and show that $\lambda^\star$ turns out to be negative.

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \;\; f_0(\mathbf{x}) &= -(x_1-1)^2-(x_2-1)^2 \\ \text{subject to} \;\; f_1(\mathbf{x}) &= 4-(x_1-2)^2-x_2 \;=\; 0 \end{aligned}$$

How does the *graphical* solution of this problem differ from that of `arch1`? Interpret the negative $\lambda^\star$ as a ratio of gradient lengths, and as a shadow price.

15.6.25 [E] Is every Lagrange point a local minimum? Is every Lagrange point a stationary point? Explain.

15.6.26 [E] What difficulties can arise in testing a reduced objective to classify a Lagrange point? Describe two other general approaches to the analytical classification of Lagrange points, comparing their difficulty and applicability.

15.6.27 [H] In our study of the `arch1` problem in §15.4.3, we defined

$$\hat{\mathbb{T}} = \{\mathbf{x}\in\mathbb{R}^n \mid \nabla f_i(\hat{\mathbf{x}})^\top(\mathbf{x}-\hat{\mathbf{x}}) = 0 \text{ for } i = 1\ldots m\}\,.$$

Use this definition to find the equation of the straight line that is $\hat{\mathbb{T}}$ at $\hat{\mathbf{x}} = [1,3]^\top$ and show that it is tangent to the $f_1(\mathbf{x}) = 0$ contour there.

15.6.28 [H] Show that if $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are vectors in $\mathbb{R}^n$, $\mathbf{u}\perp\mathbf{w}$, $\mathbf{v}\perp\mathbf{w}$, and a and b are scalars, then $(a\mathbf{u}+b\mathbf{v})\perp\mathbf{w}$. The symbol $\perp$ means that the vectors are orthogonal.

15.6.29 [E] Verify the accuracy of the first picture in §15.4.3 by confirming that the vectors plotted there are drawn to scale and have the relationships described. Confirm analytically that the vectors drawn at right angles to one another are indeed orthogonal. What determines the value of λ?

15.6.30 [E] Suppose that at some point $\hat{\mathbf{x}}$ which is feasible for a nonlinear program the constraint gradients are linearly independent and $\mathbb{T} = \{\mathbf{x}\in\mathbb{R}^n \mid \nabla f_i(\hat{\mathbf{x}})^\top\mathbf{x} = 0 \text{ for } i = 1\ldots m\}$ is a hyperplane tangent to the constraints. (a) Explain why the orthogonal projection of the objective gradient onto $\mathbb{T}$ is the gradient of the Lagrangian. (b) Explain why, on $\mathbb{T}$, the Hessian of the reduced objective is equal to the Hessian of the Lagrangian.

15.6.31 [P] Use a symbolic algebra program such as Maple or Mathematica, or carry out the calculations by hand, to confirm the algebraic equality of the expressions found in §15.4.3 for (a) $\mathcal{L}(x_1)$ and $f_0(x_1)$; (b) $\mathcal{L}'(x_1)$ and $f_0'(x_1)$; (c) $\mathcal{L}''(x_1)$ and $f_0''(x_1)$.

15.6.32 [P] The first picture in §15.4.3 shows that $\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\lambda})$ is the projection of $\nabla f_0(\hat{\mathbf{x}})$ onto $\hat{\mathbb{T}}$. (a) Draw in the hyperplane $\mathbb{T}$, and confirm that the projection of $\nabla f_0(\hat{\mathbf{x}})$ onto $\mathbb{T}$ is also $\nabla_{\mathbf{x}}\mathcal{L}(\hat{\mathbf{x}},\hat{\lambda})$. (b) Find $\mathcal{L}(x_1)$, $\mathcal{L}'(x_1)$, and $\mathcal{L}''(x_1)$ on $\mathbb{T}$ as functions of x_1, and write a MATLAB program to plot them. How does your graph differ from the §15.4.3 graph of those functions on $\hat{\mathbb{T}}$?

15.6.33 [E] To classify Lagrange points based on the definiteness of the Hessian of the reduced objective, we can test the definiteness of the Hessian of the Lagrangian on $\mathbb{T}$ instead. What makes that possible?

15.6.34 [E] Explain how to use the second-order sufficient conditions to test whether a Lagrange point $\bar{\mathbf{x}}$ is a local minimizing point.

15.6.35 [H] Use the second-order sufficient conditions to classify the Lagrange point that we found in §8.2.3 for the `garden` problem.

15.6.36 [H] Consider the following nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & -3x_1x_3 - 4x_2x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2^2 + x_3^2 - 4 = 0 \\ & f_2(\mathbf{x}) & = & x_1x_3 - 3 = 0 \end{array}$$

(a) Use the method of Lagrange to find all of the Lagrange points. The optimal value is -17.
(b) Use the second-order sufficient conditions to classify each Lagrange point, and report $\mathbf{x}^\star$.

15.6.37 [H] In §15.4.3 we encountered the second-order sufficient conditions, which state that if $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ is a Lagrange point and $\mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ is positive definite on $\mathbb{T}$, then $\bar{\mathbf{x}}$ is a strict local minimum. The **second-order necessary conditions** [5, Theorem 12.5] [4, Theorem 14.15] [107, §10.5] state that if the Lagrange point $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ is a local minimum and the gradients of the constraints are linearly independent there, then $\mathbf{H}_{\mathcal{L}}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ is positive semidefinite on $\mathbb{T}$. Does this result add to our suite of techniques for classifying Lagrange points? Explain.

15.6.38 [H] (a) The Lagrange conditions stated in the theorem of §15.2 are first-order necessary conditions for problems having equality constraints. Show that when $m = 0$ they reduce to the first order necessary conditions stated in the theorem of §10.7 for unconstrained problems. (b) How are the second-order necessary conditions given in Ex 15.6.37 for equality-constrained problems related to the second-order necessary conditions given in §10.7 for unconstrained problems? (c) How are the second-order sufficient conditions given in §15.4.3 for equality-constrained problems related to the strong second-order sufficient conditions given in §10.7 for unconstrained problems?

15.6.39 [E] What is the *nullspace* of a matrix? What is the *projected Hessian* of a Lagrangian? What does the MATLAB `null()` function take as an argument and return as a result? Outline the calculation performed by the MATLAB program `socheck.m`, and explain how it is used.

15.6.40 [E] In §15.5 I described the standard approach that I will use for coding MATLAB routines to compute function values, gradient vectors, and Hessian matrices for nonlinear programs that have constraints. Explain what this approach is, and how it works.

15.6.41 [H] Use the method of Lagrange to solve the `one23` problem described in §15.5.

15.6.42 [P] The following problem is based on Himmelblau 5 [80, p397].

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 + x_3^2 - 25 = 0 \\ & f_2(\mathbf{x}) & = & 8x_1 + 14x_2 + 7x_3 - 56 = 0 \end{array}$$

The optimal point is alleged to be $\mathbf{x}^\star = [3.512, 0.217, 3.552]^\top$ and I found (by using the `mults` routine of §16.10) the corresponding Lagrange multipliers to be $\boldsymbol{\lambda} = [1.22346, 0.27493]^\top$. Is this solution really a minimizing point?

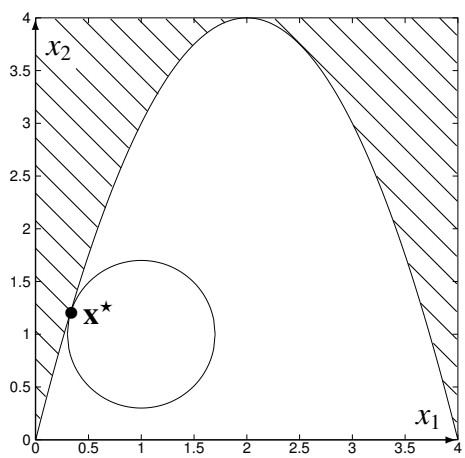
16

Inequality Constraints

When we solved the `garden` problem by using calculus in §8.2.2 and by using the Lagrange method in §8.2.3, we pretended that it was necessary to *guess* which constraints would be tight at $\mathbf{x}^\star$ and which would be slack. That guess was easy to make, because we had already studied the problem graphically in §8.2.1. In the same easy way, we can decide based on the pictures below that the inequality is active on the left but inactive on the right.

`arch2`

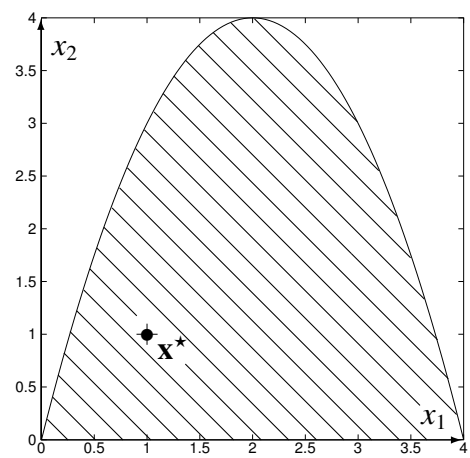
$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) &= (x_1-1)^2+(x_2-1)^2 \\ \text{subject to} \quad f_1(\mathbf{x}) &= 4-(x_1-2)^2-x_2 \;\le\; 0 \end{aligned}$$



Here we can see that the constraint is tight at $\mathbf{x}^\star$, so we can treat it as an equality and solve the problem using the Lagrange method. When we did that in §15 we found $\mathbf{x}^\star \approx [0.33, 1.20]^\mathsf{T}$. At that optimal point, $f_1(\mathbf{x}^\star) = 0$ and $\lambda^\star \approx 0.402 \neq 0$.

`arch3`

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) &= (x_1-1)^2+(x_2-1)^2 \\ \text{subject to} \quad f_1(\mathbf{x}) &= 4-(x_1-2)^2-x_2 \;\ge\; 0 \end{aligned}$$



Here we see that the constraint is slack at $\mathbf{x}^\star$, so we can ignore it. In the Lagrange-method formulation this can be accomplished by setting $\lambda = 0$, which makes $\mathcal{L}(\mathbf{x}, \lambda) = f_0(\mathbf{x})$. Now $\mathbf{x}^\star = [1, 1]^\mathsf{T}$ and at that optimal point $f_1(\mathbf{x}^\star) = 2 \neq 0$ and $\lambda^\star = 0$.

Either the constraint is tight, so that $f_1(\mathbf{x}^\star) = 0$, or $\lambda^\star = 0$ so that the constraint is out of the problem. This relationship between the value of an inequality constraint and the value of its associated Lagrange multiplier holds in general [78, Example 2.4] and in the next Section it will provide us with an automatic way of figuring out, in the process of finding $\mathbf{x}^\star$, whether an inequality is tight or slack. This will lead [3, §9.4] to an analytic method that we can use to solve inequality-constrained nonlinear programs even when we can't draw a graph.

16.1 Orthogonality

At the optimal point of an inequality-constrained nonlinear program, either $f_i(\mathbf{x}) = 0$ because constraint i is active or $\lambda_i = 0$ because it is not. We can express this relationship algebraically by requiring that

$$\boxed{\lambda_i f_i(\mathbf{x}) = 0} \qquad \text{for each } \; i = 1 \ldots m.$$

We don't know, when we begin solving a problem, which of the $f_i(\mathbf{x}^\star)$ or $\lambda_i^\star$ (or possibly both) will turn out to be zero, but if we append the boxed condition to the Lagrange conditions then any point $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ that satisfies them all will tell us, by the values of the $\bar{\lambda}_i$, which constraints are tight and which are slack at $\bar{\mathbf{x}}$. This is analogous to complementary slackness in linear programming (see §5.1.5) so this condition is sometimes [1, §4.2.8] called the **complementary slackness condition**. It can be stated in another way if we think of the multipliers as a vector $\boldsymbol{\lambda}$ and the constraint function values as a vector $\mathbf{f}(\mathbf{x})$, like this.

$$\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix} \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix}$$

If for each $i = 1 \ldots m$ either $f_i = 0$ or $\lambda_i = 0$ or both, then $\boldsymbol{\lambda}^\top \mathbf{f} = 0$, so the vectors are orthogonal. I will therefore refer to the boxed condition as the **orthogonality condition**.

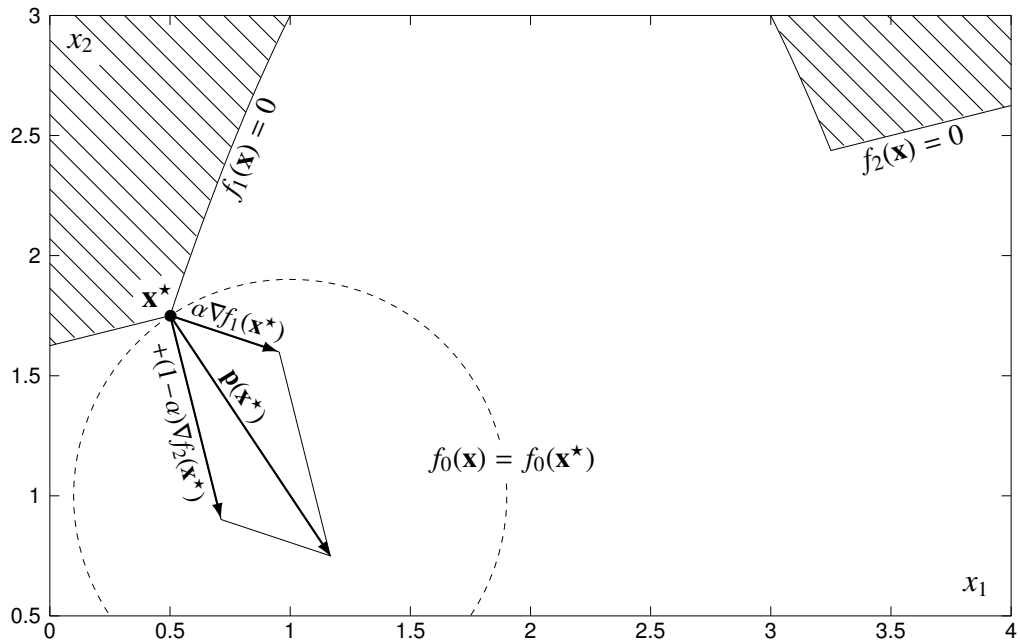
16.2 Nonnegativity

If there is only one constraint $f_1(\mathbf{x}) \le 0$ and it is tight at a local minimum $\bar{\mathbf{x}}$ (as at $\mathbf{x}^\star$ in `arch2`) then the objective and constraint gradients point in opposite directions so $-\nabla f_0(\bar{\mathbf{x}}) = \lambda \nabla f_1(\bar{\mathbf{x}})$ with $\lambda > 0$. If the constraint is slack at $\bar{\mathbf{x}}$ (as at $\mathbf{x}^\star$ in `arch3`) then $\lambda = 0$. Thus $\lambda \ge 0$.

In §15.2 we saw that if two constraints are active at $\bar{\mathbf{x}}$ then their gradients and $-\nabla f_0(\bar{\mathbf{x}})$ all lie in the same 2-dimensional hyperplane. In fact, in the diagram shown there $-\nabla f_0(\bar{\mathbf{x}})$ is *between* the constraint gradients so it can be written as a *nonnegative* linear combination of them and again $\boldsymbol{\lambda} \ge \mathbf{0}$. A simpler example illustrating this phenomenon is the problem below, which I will call `arch4`.

$$\begin{array}{rrcll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1-1)^2 + (x_2-1)^2 & \\ \text{subject to} & f_1(\mathbf{x}) & = & 4 - (x_1-2)^2 - x_2 & \le \; 0 \\ & f_2(\mathbf{x}) & = & \frac{13}{8} + \frac{1}{4}x_1 - x_2 & \le \; 0 \end{array}$$

The graph on the next page shows that the feasible set of `arch4` is like that of `arch2` but truncated on each side by the new constraint. Both constraints are active at the optimal point, which is where the optimal objective contour touches their left intersection.



That turns out to be at $\mathbf{x}^\star = [\frac{1}{2}, \frac{7}{4}]^\top$, where we have

$$\nabla f_0(\mathbf{x}^\star) = \begin{bmatrix} -1 \\ \frac{3}{2} \end{bmatrix}, \quad \nabla f_1(\mathbf{x}^\star) = \begin{bmatrix} 3 \\ -1 \end{bmatrix}, \quad \text{and} \quad \nabla f_2(\mathbf{x}^\star) = \begin{bmatrix} \frac{1}{4} \\ -1 \end{bmatrix}.$$

To write $-\nabla f_0(\mathbf{x}^\star) = \lambda_1 \nabla f_1(\mathbf{x}^\star) + \lambda_2 \nabla f_2(\mathbf{x}^\star)$ we need

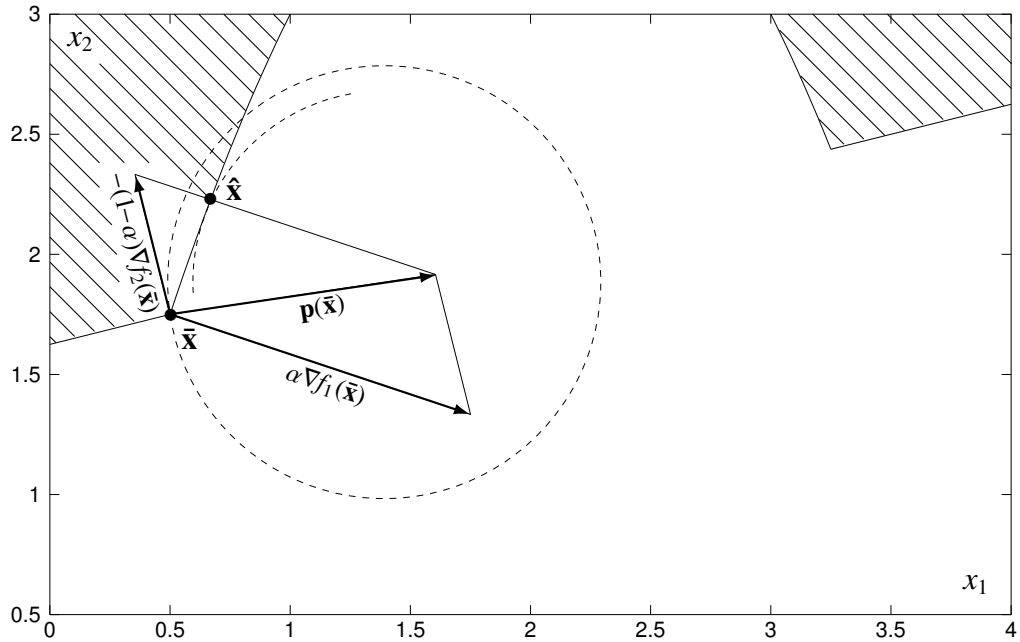
$$\begin{bmatrix} 1 \\ -\frac{3}{2} \end{bmatrix} = \lambda_1 \begin{bmatrix} 3 \\ -1 \end{bmatrix} + \lambda_2 \begin{bmatrix} \frac{1}{4} \\ -1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 3 & \frac{1}{4} \\ -1 & -1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -\frac{3}{2} \end{bmatrix}$$

which has the solution $\boldsymbol{\lambda} = [\frac{5}{22}, \frac{14}{11}]^\top$. The relationship between the gradients is easy to visualize graphically if we rewrite the nonnegative linear combination above as a convex combination (see §3.5). Letting $\alpha = \lambda_1/(\lambda_1 + \lambda_2) = \frac{5}{33}$, which makes $(1 - \alpha) = \lambda_2/(\lambda_1 + \lambda_2) = \frac{28}{33}$,

$$\mathbf{p}(\mathbf{x}^\star) = \frac{-\nabla f_0(\mathbf{x}^\star)}{\lambda_1 + \lambda_2} = \frac{\lambda_1 \nabla f_1(\mathbf{x}^\star)}{\lambda_1 + \lambda_2} + \frac{\lambda_2 \nabla f_2(\mathbf{x}^\star)}{\lambda_1 + \lambda_2} = \alpha \nabla f_1(\mathbf{x}^\star) + (1 - \alpha) \nabla f_2(\mathbf{x}^\star).$$

The picture above shows $\mathbf{p}(\mathbf{x}^\star)$, the scaled negative gradient of the objective, as this convex combination of the constraint gradients.

It is true in general that if the gradients of the active constraints are linearly independent (see §28.2.4) at a local minimizing point $\bar{\mathbf{x}}$, then the scaled negative gradient of the objective at $\bar{\mathbf{x}}$ can be written as a convex combination of the constraint gradients at $\bar{\mathbf{x}}$. Above, this convex combination is the long diagonal of the parallelogram; in higher dimensions it is the diameter of a polyhedron in $\mathbb{R}^n$ (see the first drawing in §3.6.1)



If the objective in `arch4` were different, might its negative gradient at a local minimizing point fall *outside* the arc between the constraint gradients? Suppose we modify the `arch4` problem by rotating its optimal objective contour about the `arch4` optimal point, which I will here call $\bar{\mathbf{x}}$, until $\mathbf{p}(\bar{\mathbf{x}})$ is no longer between $\nabla f_2(\bar{\mathbf{x}})$ and $\nabla f_1(\bar{\mathbf{x}})$. That is the situation in the picture above (I arbitrarily chose a rotation of 27°). It is still possible to write

$$-\nabla f_0(\bar{\mathbf{x}}) = \lambda_1 \nabla f_1(\bar{\mathbf{x}}) + \lambda_2 \nabla f_2(\bar{\mathbf{x}}),$$

but only if $\boldsymbol{\lambda} \approx [0.67265, -0.94114]^\top$, so the linear combination is no longer nonnegative. Now $\mathbf{p}(\bar{\mathbf{x}}) = -\nabla f_0(\bar{\mathbf{x}})/(\lambda_1 + |\lambda_2|)$ and to write it as a convex combination we must use the *negative* of $\nabla f_2(\bar{\mathbf{x}})$ like this.

$$\mathbf{p}(\bar{\mathbf{x}}) = \alpha \nabla f_1(\bar{\mathbf{x}}) + (1-\alpha)\left[-\nabla f_2(\bar{\mathbf{x}})\right]$$

Here $\alpha = \lambda_1/(\lambda_1 + |\lambda_2|) = 0.41681$, which makes $(1-\alpha) = 0.58319$, and it is this convex combination that is pictured in the graph above. Unfortunately, the formerly-optimal objective contour now *intersects* the feasible set, so $\bar{\mathbf{x}}$ is no longer optimal (the new optimal point is $\hat{\mathbf{x}}$). In order for the optimal objective contour not to cross over the zero contour of one constraint or the other, $-\nabla f_0(\mathbf{x})$ must remain between the two constraint gradients, and that means it can be represented as a *nonnegative* linear combination of them.

It is true in general [1, §4.2.13] that if $\bar{\mathbf{x}}$ is a local minimizing point and the gradients of the active constraints $f_i(\bar{\mathbf{x}}) \le 0$ are linearly independent there, then if we write

$$-\nabla f_0(\bar{\mathbf{x}}) = \sum_{i=1}^{m} \lambda_i f_i(\bar{\mathbf{x}})$$

it will turn out that $\lambda_i \ge 0$ for $i = 1 \ldots m$.

We can make use of this fact in solving inequality-constrained nonlinear programs by requiring that

$$\boxed{\lambda_i \geq 0} \quad \text{for each } i = 1 \ldots m.$$

and I will refer to this as the **nonnegativity condition**.

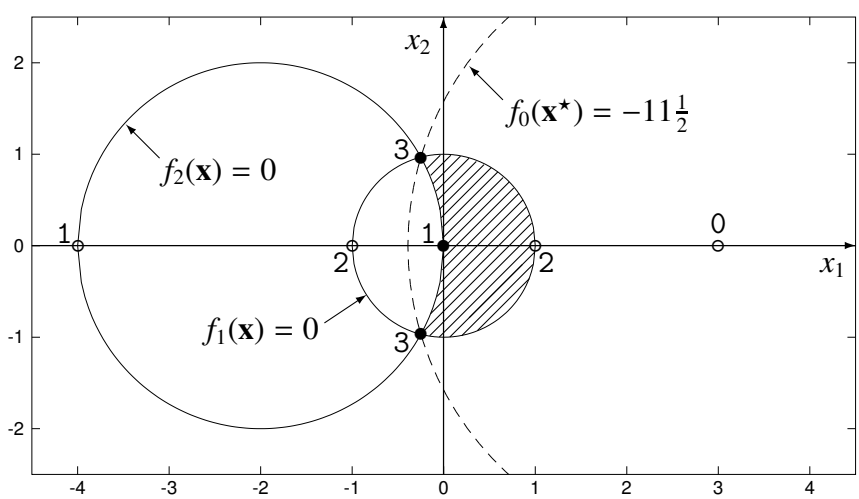
16.3 The Karush-Kuhn-Tucker Conditions

Combining the results of §16.1 and §16.2 with those of §15.3 we get a set of conditions that play the same role for inequality-constrained nonlinear programs that the Lagrange conditions play for problems having equality constraints.

$$\boxed{\begin{array}{rcl} \nabla f_0(\mathbf{x}) + \displaystyle\sum_{i=1}^{m} \lambda_i \nabla f_i(\mathbf{x}) & = & 0 \\ \left.\begin{array}{rcl} f_i(\mathbf{x}) & \leq & 0 \\ \lambda_i f_i(\mathbf{x}) & = & 0 \\ \lambda_i & \geq & 0 \end{array}\right\} i = 1 \ldots m \end{array}} \quad \begin{array}{l} \text{stationarity} \\ \\ \text{feasibility} \\ \text{orthogonality} \\ \text{nonnegativity} \end{array}$$

Together these are called the Karush-Kuhn-Tucker conditions, because [164] they were discovered first (in 1939) by William Karush [90] and then (in 1951) independently by Harold W. Kuhn and Albert W. Tucker [97]. We will refer to the boxed conditions as the **KKT conditions** and to a point that satisfies them as a **KKT point**, and we will call the multipliers λ_i that satisfy them **KKT multipliers**.

By using the KKT conditions we can find local minimizing points for some inequality-constrained nonlinear programs. To see how, consider the `moon` problem (see §28.7.11) pictured below.



Here we want to maximize the radius of a circle centered at $(3, 0)$ while remaining in the feasible set that is shown crosshatched. An algebraic statement of the problem is given on the left below and rewritten in the standard form of §8.1 on the right.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & (x_1-3)^2+x_2^2 & \\ \text{subject to} & x_1^2+x_2^2 & \le\ 1 \\ & (x_1+2)^2+x_2^2 & \ge\ 2^2 \end{array} \qquad \begin{array}{llll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & -(x_1-3)^2-x_2^2 & \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2+x_2^2-1 & \le\ 0 \\ & f_2(\mathbf{x}) & = & -(x_1+2)^2-x_2^2+4 & \le\ 0 \end{array}$$

From the Lagrangian of the standard-form problem we can write the KKT conditions, as follows.

$$\begin{aligned} \mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) &= f_0(\mathbf{x})+\lambda_1 f_1(\mathbf{x})+\lambda_2 f_2(\mathbf{x}) \\ &= (-x_1^2-x_2^2+6x_1-9)+\lambda_1(x_1^2+x_2^2-1)+\lambda_2(-x_1^2-x_2^2-4x_1) \end{aligned}$$

$$\left.\begin{aligned} \frac{\partial L}{\partial x_1} &= -2x_1+6+2\lambda_1 x_1-2\lambda_2 x_1-4\lambda_2=0 \quad \text{(A)} \\ \frac{\partial L}{\partial x_2} &= -2x_2+2\lambda_1 x_2-2\lambda_2 x_2=0 \quad \text{(B)} \end{aligned}\right\} \nabla_{\mathbf{x}}\mathcal{L}=\mathbf{0} \quad \text{stationarity}$$

$$\left.\begin{aligned} \frac{\partial L}{\partial \lambda_1} &= x_1^2+x_2^2-1\le 0 \quad \text{(C)} \\ \frac{\partial L}{\partial \lambda_2} &= -x_1^2-x_2^2-4x_1\le 0 \quad \text{(D)} \end{aligned}\right\} \nabla_{\boldsymbol{\lambda}}\mathcal{L}\le\mathbf{0} \quad \text{feasibility}$$

$$\left.\begin{aligned} &\lambda_1(x_1^2+x_2^2-1)=0 \quad \text{(E)} \\ &\lambda_2(-x_1^2-x_2^2-4x_1)=0 \quad \text{(F)} \end{aligned}\right\} \boldsymbol{\lambda}^{\mathsf{T}}\mathbf{f}=\mathbf{0} \quad \text{orthogonality}$$

$$\left.\begin{aligned} &\lambda_1\ge 0 \quad \text{(G)} \\ &\lambda_2\ge 0 \quad \text{(H)} \end{aligned}\right\} \boldsymbol{\lambda}\ge\mathbf{0} \quad \text{nonnegativity}$$

In solving KKT conditions it is often helpful to consider cases corresponding to the possible combinations of slack and tight constraints. For this problem the possibilities are described in the table below, where the logical value 0 means the constraint is assumed to be slack (it is *false* that $f_i(\mathbf{x})=0$ so $f_i(\mathbf{x})<0$ and $\lambda_i=0$) and 1 means the constraint is assumed to be tight (it is *true* that $f_i(\mathbf{x})=0$ so λ_i can be nonzero). The case number, used later to refer to each combination, is the value of the resulting binary number.

$f_1(\mathbf{x})=0$	$f_2(\mathbf{x})=0$	case number
0	0	0
0	1	1
1	0	2
1	1	3

Below, each case is analyzed to illustrate the sort of reasoning that is necessary to find points satisfying the KKT conditions. Some different (and possibly more elegant) sequence of steps might work in each chain of implications to arrive at the same conclusions.

CASE 0 ($\lambda_1 = 0$, $\lambda_2 = 0$): substituting these values into the conditions leads to a contradiction ⨳, because the point $[3, 0]^\top$, marked ∘ and labeled 0 in the picture, is infeasible.

$$
\begin{aligned}
\text{(A)} &\Rightarrow -2x_1 + 6 = 0 \Rightarrow x_1 = 3 \\
\text{(B)} &\Rightarrow -2x_2 = 0 \Rightarrow x_2 = 0 \\
\text{(C)} &\Rightarrow x_1^2 + x_2^2 - 1 = 3^2 + 0^2 - 1 = 8 \not\leq 0 \text{ ⨳}
\end{aligned}
$$

CASE 1 ($\lambda_1 = 0$, $\lambda_2 \neq 0$): the point $[0, 0]^\top$, marked • and labeled 1, satisfies all of the KKT conditions with $\lambda_2 = \frac{3}{2}$; the point $[-4, 0]^\top$, marked ∘ and also labeled 1, is infeasible.

$$
\begin{aligned}
\text{(B)} &\Rightarrow -2x_2 - 2\lambda_2 x_2 = 0 \\
&\Rightarrow \lambda_2 = -1 \quad \text{or} \quad x_2 = 0 \\
\text{(H)} &\Rightarrow \lambda_2 = -1 \not\geq 0 \text{ ⨳} \quad \text{so} \quad x_2 = 0 \\
\text{(F)} &\Rightarrow -x_1^2 - x_2^2 - 4x_1 = -x_1^2 - (0)^2 - 4x_1 = 0 \\
&\Rightarrow -x_1^2 - 4x_1 = 0 \\
&\Rightarrow x_1 = 0 \quad \text{or} \quad x_1 = -4 \\
\text{(C)} &\Rightarrow x_1^2 + x_2^2 - 1 = (-4)^2 + 0^2 - 1 = 15 \not\leq 0 \text{ ⨳} \quad \text{so} \quad x_1 = 0 \\
\text{(A)} &\Rightarrow -2x_1 + 6 - 2\lambda_2 x_1 - 4\lambda_2 = -2(0) + 6 - 2\lambda_2(0) - 4\lambda_2 = 0 \\
&\Rightarrow 6 - 4\lambda_2 = 0 \\
&\Rightarrow \lambda_2 = \tfrac{3}{2}
\end{aligned}
$$

CASE 2 ($\lambda_1 \neq 0$, $\lambda_2 = 0$): the point $[-1, 0]^\top$ is infeasible; the point $[1, 0]^\top$ is feasible but requires $\lambda_1 < 0$. Both points are marked ∘ and labeled 2.

$$
\begin{aligned}
\text{(B)} &\Rightarrow -2x_2 + 2\lambda_1 x_2 = 0 \\
&\Rightarrow x_2 = 0 \quad \text{or} \quad \lambda_1 = 1 \\
\text{(A)} &\Rightarrow -2x_1 + 6 + 2\lambda_1 x_1 = -2x_1 + 6 + 2(1)x_1 = 6 \neq 0 \text{ ⨳} \quad \text{so} \quad x_2 = 0 \\
\text{(C)} &\Rightarrow x_1^2 + x_2^2 - 1 = x_1^2 + 0^2 - 1 = 0 \\
&\Rightarrow x_1 = \pm 1 \\
\text{(D)} &\Rightarrow -x_1^2 - x_2^2 - 4x_1 = -(-1)^2 - (0)^2 - 4(-1) = 3 \not\leq 0 \text{ ⨳} \quad \text{so} \quad x_1 \neq -1 \\
\text{(A)} &\Rightarrow -2x_1 + 6 + 2\lambda_1 x_1 = -2(+1) + 6 + 2\lambda_1(+1) = 0 \\
&\Rightarrow \lambda_1 = -2 \\
\text{(G)} &\Rightarrow \lambda_1 = -2 \neq 0 \text{ ⨳} \quad \text{so} \quad x_1 \neq +1
\end{aligned}
$$

CASE 3 ($\lambda_1 \neq 0$, $\lambda_2 \neq 0$): the points $[-\frac{1}{4}, +\sqrt{\frac{15}{16}}]^\top$ and $[-\frac{1}{4}, -\sqrt{\frac{15}{16}}]^\top$, which are marked • and labeled 3, both satisfy all of the KKT conditions, with $\lambda_1 = \frac{5}{2}$ and $\lambda_2 = \frac{3}{2}$.

$$
\begin{aligned}
\text{E} \quad &\Rightarrow \quad x_1^2 + x_2^2 - 1 = 0 \\
&\Rightarrow \quad x_2^2 = 1 - x_1^2 \\
\text{F} \quad &\Rightarrow \quad -x_1^2 - x_2^2 - 4x_1 = -x_1^2 - (1 - x_1^2) - 4x_1 = 0 \\
&\Rightarrow \quad x_1 = -\tfrac{1}{4} \\
\text{E} \quad &\Rightarrow \quad x_2^2 = 1 - \left(-\tfrac{1}{4}\right)^2 = 1 - \tfrac{1}{16} \\
&\Rightarrow \quad x_2 = \pm\sqrt{\tfrac{15}{16}} \\
\text{A} \quad &\Rightarrow \quad -2x_1 + 6 + 2\lambda_1 x_1 - 2\lambda_2 x_1 - 4\lambda_2 = -2\left(-\tfrac{1}{4}\right) + 6 + 2\lambda_1\left(-\tfrac{1}{4}\right) - 2\lambda_2\left(-\tfrac{1}{4}\right) - 4\lambda_2 = 0 \\
&\Rightarrow \quad 6\tfrac{1}{2} - \tfrac{1}{2}\lambda_1 - 3\tfrac{1}{2}\lambda_2 = 0 \\
\text{B} \quad &\Rightarrow \quad -2x_2 + 2\lambda_1 x_2 - 2\lambda_2 x_2 = -2\left(\pm\sqrt{\tfrac{15}{16}}\right) + 2\lambda_1\left(\pm\sqrt{\tfrac{15}{16}}\right) - 2\lambda_2\left(\pm\sqrt{\tfrac{15}{16}}\right) = 0 \\
&\Rightarrow \quad -2 + 2\lambda_1 - 2\lambda_2 = 0 \\
&\Rightarrow \quad \lambda_1 = \lambda_2 + 1 \\
\text{A} \quad &\Rightarrow \quad 6\tfrac{1}{2} - \tfrac{1}{2}\lambda_1 - 3\tfrac{1}{2}\lambda_2 = 6\tfrac{1}{2} - \tfrac{1}{2}(\lambda_2 + 1) - 3\tfrac{1}{2}\lambda_2 = 0 \\
&\Rightarrow \quad \lambda_2 = \tfrac{3}{2} \\
\text{B} \quad &\Rightarrow \quad \lambda_1 = \lambda_2 + 1 = \left(\tfrac{3}{2}\right) + 1 = \tfrac{5}{2}
\end{aligned}
$$

Among the four cases, we found these three points that satisfy the KKT conditions.

x_1	x_2	λ_1	λ_2	$f_0(\mathbf{x})$
0	0	0	$\frac{3}{2}$	-9
$-\frac{1}{4}$	$+\sqrt{\frac{15}{16}}$	$\frac{5}{2}$	$\frac{3}{2}$	$-11\frac{1}{2}$
$-\frac{1}{4}$	$-\sqrt{\frac{15}{16}}$	$\frac{5}{2}$	$\frac{3}{2}$	$-11\frac{1}{2}$

The `moon` problem thus has the two alternate optima listed at the bottom of the table. In the picture they are the horns of the moon, passed through by the optimal objective contour. To solve the KKT conditions by hand is often an arduous task even for simple nonlinear programs like this one, and it can be an impossible task for problems of realistic size and complexity. The KKT conditions are more difficult to analyze than the Lagrange conditions because of the extra orthogonality and nonnegativity requirements. Where human diligence fails, Maple or Mathematica might succeed as illustrated in §8.2.4 for the `garden` problem, but usually the most effective tool for solving real problems is a numerical minimization algorithm. Using many ideas from this Chapter, we will begin our study of algorithms for inequality-constrained nonlinear programs in §19.

16.4 The KKT Theorems

Two of the KKT points that we found for the `moon` problem were global minima, but what about the third point? Can it ever happen that a local minimum is not a KKT point, or that some KKT points are not local minima? These questions are answered by the **KKT theorems** [1, §4.2] [5, §12.4] [4, §14.5] stated below.

Theorem: existence of KKT multipliers

given the NLP $\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x})$
subject to $f_i(\mathbf{x}) \le 0, \quad i = 1 \ldots m,$

if the $f_i(\mathbf{x}),\ i = 0 \ldots m$, are differentiable
$\bar{\mathbf{x}}$ is a local minimizing point for NLP
the $\nabla f_i(\bar{\mathbf{x}}),\ i \in \mathbb{I} = \{i \mid f_i(\bar{\mathbf{x}}) = 0,\ i = 1 \ldots m\}$, are linearly independent
or some other constraint qualification holds

then there exists a vector $\bar{\boldsymbol{\lambda}} \in \mathbb{R}^m$ such that
$(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ satisfies the KKT conditions.

The Lagrange multiplier theorem of §15.2 demands that the gradients of the equality constraints be linearly independent, but when the active constraints are inequalities it is sometimes possible to prove the existence of KKT multipliers even if that is not true; we will take up constraint qualifications in §16.7. Because the hypotheses of this theorem are necessary to ensure that a local minimum $\bar{\mathbf{x}}$ is a KKT point, they are often referred to as the **KKT necessary conditions**; if the functions are differentiable and a constraint qualification holds but there is *no* $\bar{\boldsymbol{\lambda}}$ that satisfies these conditions, then $\bar{\mathbf{x}}$ *cannot* be a local minimum.

Theorem: the KKT points of a convex program are global minima

given the NLP $\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x})$
subject to $f_i(\mathbf{x}) \le 0, \quad i = 1 \ldots m,$

if $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ satisfies the KKT conditions
the $f_i(\mathbf{x}),\ i = 0 \ldots m$, are convex functions

then $\bar{\mathbf{x}}$ is a global minimizing point.

Proof (based on [1, Theorem 4.2.16]):

To show that such a KKT point $\bar{\mathbf{x}}$ is a global minimizer we will show that no other feasible point $\hat{\mathbf{x}}$ has a lower objective value. Again let $\mathbb{I} = \{i \mid f_i(\bar{\mathbf{x}}) = 0\}$ be the indices of the constraints that are active at $\bar{\mathbf{x}}$. By the definition of convexity (see §11.1) we have for each constraint $i \in \mathbb{I}$ that

$$f_i(\alpha\hat{\mathbf{x}} + [1-\alpha]\bar{\mathbf{x}}) \le \alpha f_i(\hat{\mathbf{x}}) + (1-\alpha) f_i(\bar{\mathbf{x}}) \quad \text{for all } \alpha \in [0,1].$$

But $f_i(\bar{\mathbf{x}}) = 0$ because $i \in \mathbb{I}$, and $f_i(\hat{\mathbf{x}}) \leq 0$ because we assumed that $\hat{\mathbf{x}}$ is feasible, so

$$f_i(\alpha\hat{\mathbf{x}} + [1-\alpha]\bar{\mathbf{x}}) \leq 0 \quad \text{for} \quad \alpha \in [0,1].$$

Each active constraint already has a value $f_i(\bar{\mathbf{x}}) = 0$ at the KKT point, so moving towards $\hat{\mathbf{x}}$ does not increase the constraint value. The direction $\mathbf{d} = \hat{\mathbf{x}} - \bar{\mathbf{x}}$ is therefore a non-ascent direction of $f_i(\mathbf{x})$, which means (see §10.8) that $\nabla f_i(\bar{\mathbf{x}})^\top \mathbf{d} \leq 0$. At the KKT point $\bar{\mathbf{x}}$ we have $\lambda_i \geq 0$, so the sum

$$\sum_{i=1}^{m} \lambda_i \nabla f_i(\bar{\mathbf{x}})^\top \mathbf{d} \leq 0$$

is likewise nonpositive. Because $\bar{\mathbf{x}}$ is a KKT point it satisfies the stationarity condition,

$$\nabla f_0(\bar{\mathbf{x}}) + \sum_{i-1}^{m} \lambda_i \nabla f_i(\bar{\mathbf{x}}) = \mathbf{0}.$$

Dotting each term in this equation with the direction vector $\mathbf{d}$ and rearranging, we find

$$\nabla f_0(\bar{\mathbf{x}})^\top \mathbf{d} = -\sum_{i-1}^{m} \lambda_i \nabla f_i(\bar{\mathbf{x}})^\top \mathbf{d}.$$

We established just above that the sum on the right-hand side is nonpositive, so $\nabla f_0(\bar{\mathbf{x}})^\top \mathbf{d} \geq 0$. By the support inequality for convex functions (see §11.2),

$$f_0(\hat{\mathbf{x}}) \geq f_0(\bar{\mathbf{x}}) + \nabla f_0(\bar{\mathbf{x}})^\top \mathbf{d} \geq f_0(\bar{\mathbf{x}})$$

for every feasible $\hat{\mathbf{x}}$. Thus $\bar{\mathbf{x}}$ must be a global minimizing point. □

Because the hypotheses of this theorem are sufficient to ensure that $\bar{\mathbf{x}}$ is a global minimum, they are often referred to as the **KKT sufficient conditions**.

16.5 The KKT Method

Now we can formalize the method that we used in §16.3 to solve the `moon` problem.

1. Put the nonlinear program into standard form:

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ & f_0(\mathbf{x}) \\ \text{subject to } & f_i(\mathbf{x}) \ \leq\ 0, \quad i = 1\ldots m. \end{aligned}$$

2. Verify that the objective and constraint functions are differentiable (this is required by the KKT necessary conditions).

3. Form the Lagrangian $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x})$.

4. Write down the KKT conditions for the problem.

$$\begin{aligned} \nabla f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla f_i(\mathbf{x}) &= 0 \\ \left.\begin{aligned} f_i(\mathbf{x}) &\leq 0 \\ \lambda_i f_i(\mathbf{x}) &= 0 \\ \lambda_i &\geq 0 \end{aligned}\right\} & \; i = 1 \ldots m \end{aligned}$$

5. Find *all* solutions to the KKT conditions. Consider as a separate case each of the 2^m possible combinations of active and inactive constraints. For each case, simplify the KKT conditions by setting the appropriate λ_i to zero. Then solve the equalities, deciding between alternative solutions by looking for contradictions with the inequalities. Only when each possible alternative has been shown to lead to either a contradiction or a point that satisfies all of the conditions, move on to the next case.

6. Summarize the KKT points $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ that you found, and verify that the gradients $\nabla f_i(\bar{\mathbf{x}})$ of the active constraints are linearly independent (or that some other constraint qualification holds) at each of them.

7. Classify the solutions to identify the local minimizing points. If the problem is convex then by the KKT sufficient conditions every KKT point is a global minimum; otherwise each point must be classified by the techniques discussed in §15.4 and §15.5, assuming tight constraints to be equalities and omitting slack constraints from the analysis.

In applying the KKT method it is helpful to remember the implications of the KKT theorems, which are pictured in the diagram below (assuming the $f_i(\mathbf{x})$ are differentiable).

$$\begin{tikzcd} \bar{\mathbf{x}} \text{ is a local minimum} \arrow[d, Rightarrow, "\text{constraint qualification}"] & \bar{\mathbf{x}} \text{ is a global minimum} \arrow[l, Rightarrow] \\ (\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}}) \text{ satisfies the KKT conditions} \arrow[ur, Rightarrow, "\text{convexity}"'] & \end{tikzcd}$$

If a constraint qualification (such as linear independence of the gradients of the active constraints) holds then every local minimum satisfies the KKT conditions, but other points that are not local minima might also satisfy them. If the problem is convex then every point that satisfies the KKT conditions is a global minimum. Every global minimum is also a local minimum, so if a constraint qualification is satisfied a global minimum also satisfies the KKT conditions. However, none of the implications in this diagram works in the opposite direction! This is further evidence that the analytic theory of nonlinear programming, despite its elegance and beauty, has only limited power *unless the problem is convex.*

16.6 Convex Programs

In proving the second KKT theorem, we needed $\bar{\mathbf{x}} + \alpha(\hat{\mathbf{x}} - \bar{\mathbf{x}})$ to be feasible for all $\alpha \in [0, 1]$, and the convexity of the constraint functions ensured it would be. That is just a special case of the following more general result.

Theorem: convex constraints $f_i(\mathbf{x}) \le 0$ have a convex intersection

given the NLP $\quad \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x})$
$\quad$ subject to $f_i(\mathbf{x}) \le 0, \quad i = 1 \ldots m,$

if $\quad$ the $f_i(\mathbf{x}),\ i = 1 \ldots m,$ are convex functions

then $\quad \mathbb{X} = \{x \in \mathbb{R}^n \mid f_i(\mathbf{x}) \le 0,\ i = 1 \ldots m\}$ is a convex set.

Proof:

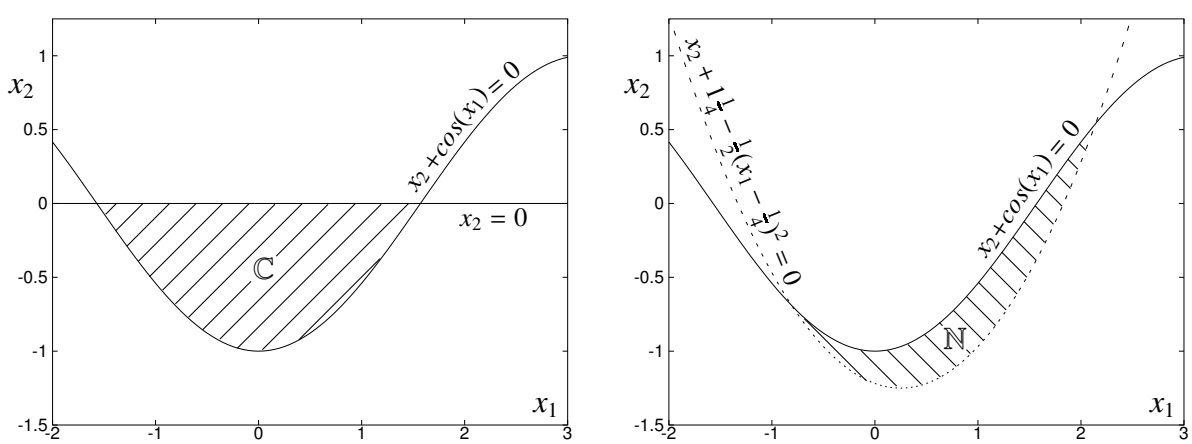
Let $\mathbb{S}_i(z) = \{\mathbf{x} \in \mathbb{R}^n \mid f_i(\mathbf{x}) \le z\}$ be the z level set of $f_i(\mathbf{x})$ (see Exercise 11.7.3) and pick two points $\hat{\mathbf{x}} \in \mathbb{S}_i(z)$ and $\bar{\mathbf{x}} \in \mathbb{S}_i(z)$. Then $f_i(\hat{\mathbf{x}}) \le z$ and $f_i(\bar{\mathbf{x}}) \le z$. Now let $\mathbf{x} = \alpha\hat{\mathbf{x}} + (1-\alpha)\bar{\mathbf{x}}$. Because $f_i(\mathbf{x})$ is a convex function,

$$\begin{aligned} f_i(\mathbf{x}) &\le \alpha f_i(\hat{\mathbf{x}}) + (1-\alpha) f_i(\bar{\mathbf{x}}) \\ &\le \alpha z + (1-\alpha) z = z \end{aligned}$$

so $\mathbf{x} \in \mathbb{S}_i(z)$, and $\mathbb{S}_i(z)$ must be a convex set. The feasible set $\mathbb{X}$ of a standard-form nonlinear program is the intersection of the zero level sets $\mathbb{S}_i(0)$ of its constraints, and the intersection of convex sets is convex (see Exercise 3.7.26) so $\mathbb{X}$ is convex. □

According to this theorem, a convex program has a convex feasible set. However, not every NLP with a convex feasible set is a convex program; a standard-form NLP is a convex program only if its objective and all of its constraints are convex functions (see §11.2). A nonconvex constraint can yield a feasible set that is convex like

$$\begin{aligned} \mathbb{C} &= \{\mathbf{x} \in \mathbb{R}^2 \mid x_2 \ge -\cos(x_1) \cap x_2 \le 0 \cap x_1 \in [-2, 2]\} && \text{on the left or nonconvex like} \\ \mathbb{N} &= \{\mathbf{x} \in \mathbb{R}^2 \mid x_2 \le -\cos(x_1) \cap x_2 \ge -1\tfrac{1}{4} + \tfrac{1}{2}(x_1 - \tfrac{1}{4})^2\} && \text{on the right.} \end{aligned}$$



A problem with equality constraints can be written in standard form, as explained in §8.1, by replacing each equality with **opposing inequalities**, like this.

$$\begin{array}{rl} \text{minimize} & f_0(\mathbf{x}) \\ \text{subject to} & f_1(\mathbf{x}) = 0 \end{array} \quad \longrightarrow \quad \begin{array}{rrcl} \text{minimize} & f_0(\mathbf{x}) & & \\ \text{subject to} & f_1(\mathbf{x}) & \leq & 0 \\ & -f_1(\mathbf{x}) & \leq & 0 \end{array}$$

The inequality-constrained problem has the KKT conditions derived below.

$$\begin{aligned} \mathcal{L} &= f_0(\mathbf{x}) + \lambda_1 f_1(\mathbf{x}) + \lambda_2 \left[-f_1(\mathbf{x})\right] \\ \nabla_{\mathbf{x}}\mathcal{L} &= \nabla_{\mathbf{x}} f_0(\mathbf{x}) + \lambda_1 \nabla_{\mathbf{x}} f_1(\mathbf{x}) - \lambda_2 \nabla_{\mathbf{x}} f_1(\mathbf{x}) = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda_1} &= f_1(\mathbf{x}) \leq 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda_2} &= -f_1(\mathbf{x}) \leq 0 \\ \lambda_1 f_1(\mathbf{x}) &= 0 \\ \lambda_2 \left[-f_1(\mathbf{x})\right] &= 0 \\ \lambda_1 &\geq 0 \\ \lambda_2 &\geq 0 \end{aligned}$$

Recall from §2.9.3 that a variable unconstrained in sign can be written as the difference between nonnegative variables. If we let $\lambda = \lambda_1 - \lambda_2$, we can rewrite the KKT conditions above as follows.

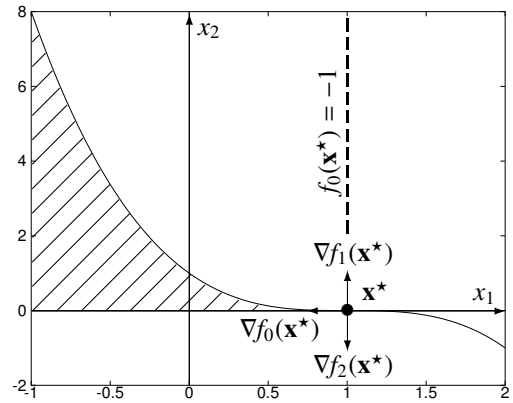
$$\begin{aligned} \mathcal{L} &= f_0(\mathbf{x}) + (\lambda_1 - \lambda_2) f_1(\mathbf{x}) \\ &= f_0(\mathbf{x}) + \lambda f_1(\mathbf{x}) \\ \nabla_{\mathbf{x}}\mathcal{L} &= \nabla_{\mathbf{x}} f_0(\mathbf{x}) + (\lambda_1 - \lambda_2) \nabla_{\mathbf{x}} f_1(\mathbf{x}) \\ &= \nabla_{\mathbf{x}} f_0(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} f_1(\mathbf{x}) = 0 \\ \left.\begin{array}{rcl} f_1(\mathbf{x}) & \leq & 0 \\ f_1(\mathbf{x}) & \geq & 0 \end{array}\right\} \Rightarrow f_1(\mathbf{x}) = 0 \quad \text{or} \quad \frac{\partial \mathcal{L}}{\partial \lambda} &= f_1(\mathbf{x}) = 0 \\ \lambda_1 f_1(\mathbf{x}) - \lambda_2 f_1(\mathbf{x}) = (\lambda_1 - \lambda_2) f_1(\mathbf{x}) \quad = \quad \lambda f_1(\mathbf{x}) &= 0 \\ \lambda &\quad \text{free} \end{aligned}$$

These are precisely the Lagrange conditions for the equality-constrained problem. It is true in general that *the Lagrange conditions are a special case of the KKT conditions when the constraints are equalities.* In order for this problem to be a convex program, $f_0(\mathbf{x})$ and both of the inequality constraint functions $f_1(\mathbf{x})$ and $-f_1(\mathbf{x})$ must be convex, but if $f_1(\mathbf{x})$ is convex then $-f_1(\mathbf{x})$ is concave. The only way for both constraint inequalities to be convex is if they are linear, because then each is simultaneously convex *and* concave. If $f_1(\mathbf{x})$ is linear then the feasible set is a hyperplane in $\mathbb{R}^n$, which is a convex set. If $f_1(\mathbf{x})$ is *non*linear, then the feasible set is a *curved* hypersurface, which is *not* convex. An equality-constrained NLP is a convex program if and only if $f_0(\mathbf{x})$ is convex and the constraints are *linear.*

16.7 Constraint Qualifications

This nonlinear program [97] [1, §4.2.10], which I will call `cq1` (see §28.7.12) has $\mathbf{x}^\star = [1, 0]^\mathsf{T}$.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & -x_1 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2 - (1-x_1)^3 \le 0 \\ & f_2(\mathbf{x}) & = & -x_2 \le 0 \end{array}$$



From the Lagrangian

$$\mathcal{L} = -x_1 + \lambda_1\left[x_2 - (1-x_1)^3\right] + \lambda_2(-x_2)$$

we derive the following KKT conditions.

$$\begin{array}{rcl} \dfrac{\partial\mathcal{L}}{\partial x_1} & = & -1 - 3\lambda_1(1-x_1)^2(-1) = 0 \quad \text{A} \\ \dfrac{\partial\mathcal{L}}{\partial x_2} & = & \lambda_1 - \lambda_2 = 0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda_1} & = & x_2 - (1-x_1)^3 \le 0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda_2} & = & -x_2 \le 0 \\ \lambda_1 f_1(\mathbf{x}) & = & \lambda_1\left[x_2 - (1-x_1)^3\right] = 0 \\ \lambda_2 f_2(\mathbf{x}) & = & \lambda_2(-x_2) = 0 \\ \lambda_1 & \ge & 0 \\ \lambda_2 & \ge & 0 \end{array}$$

At the optimal point condition A reduces to

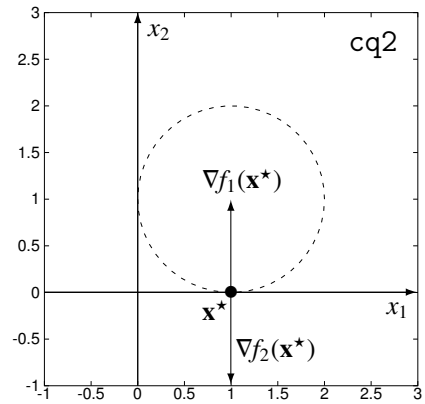
$$\begin{array}{rcl} -1 - 3\lambda_1(1-1)^2(-1) & = & 0 \\ \text{or} \quad -1 & = & 0 \ \text{※}. \end{array}$$

Oops! The other conditions are met, but $\mathbf{x}^\star$ is *not* a KKT point because it does not satisfy any constraint qualification. The one we have been using is linear independence of the gradients of the active constraints, but for this problem we find

$$\nabla f_0(\mathbf{x}^\star) = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \qquad \nabla f_1(\mathbf{x}^\star) = \begin{bmatrix} -3(1-x_1^\star)(-1) \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \nabla f_2(\mathbf{x}^\star) = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

so $\nabla f_1(\mathbf{x}^\star)$ and $\nabla f_2(\mathbf{x}^\star)$ are linearly dependent vectors and $\nabla f_0(\mathbf{x}^\star)$ cannot be written as a linear combination of them. This deplorable situation is also clear from the graph.

It is, however, possible for the optimal point of a nonlinear program to satisfy the KKT conditions even though the constraint gradients are *not* linearly independent there. Consider the following problem, which I will call `cq2` (see §28.7.13).



$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1-1)^2+(x_2-1)^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2 \le 0 \\ & f_2(\mathbf{x}) & = & -x_2 \le 0 \end{array}$$

From the Lagrangian

$$\mathcal{L} = (x_1-1)^2+(x_2-1)^2+\lambda_1(x_2)+\lambda_2(-x_2)$$

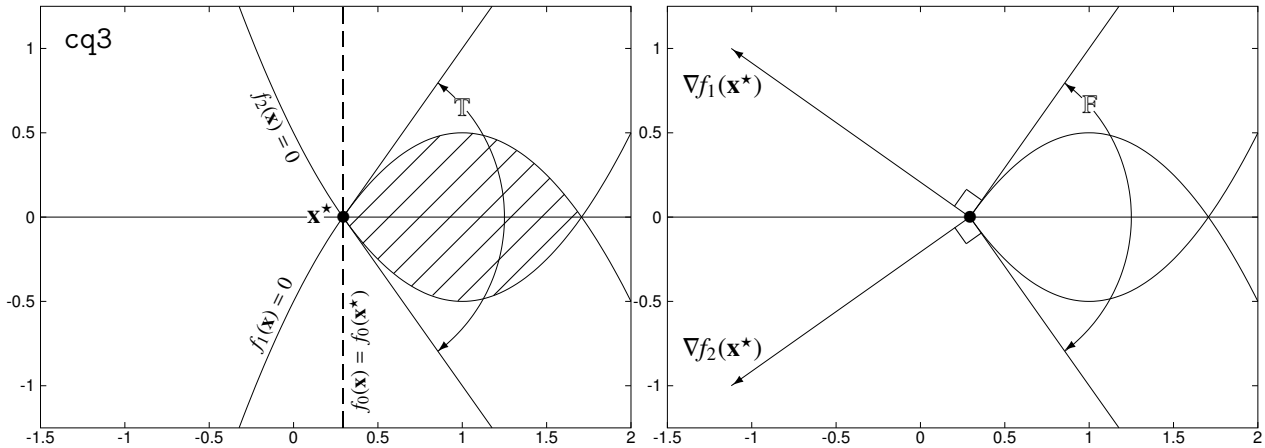
we derive the following KKT conditions.

$$\begin{array}{rcll} \dfrac{\partial\mathcal{L}}{\partial x_1} & = & 2(x_1-1)=0 & \text{(A)} \\ \dfrac{\partial\mathcal{L}}{\partial x_2} & = & 2(x_2-1)+\lambda_1-\lambda_2=0 & \text{(B)} \\ \dfrac{\partial\mathcal{L}}{\partial \lambda_1} & = & x_2 \le 0 & \text{(C)} \\ \dfrac{\partial\mathcal{L}}{\partial \lambda_2} & = & -x_2 \le 0 & \text{(D)} \\ \lambda_1 f_1(\mathbf{x}) & = & \lambda_1 x_2 = 0 & \text{(E)} \\ \lambda_2 f_2(\mathbf{x}) & = & \lambda_2(-x_2)=0 & \text{(F)} \\ \lambda_1 & \ge & 0 & \text{(G)} \\ \lambda_2 & \ge & 0 & \text{(H)} \end{array}$$

The opposing inequalities make the x_1 axis the feasible set. From (A) we get $x_1^\star = 1$, and from (C) and (D) together we get $x_2^\star = 0$. Then (B) requires that $\lambda_2 = \lambda_1 - 2$, and any value of $\lambda_1 \ge 2$ will do, so that λ_2 is (H) nonnegative. When the gradients of the active constraints are linearly dependent the λ_i are not uniquely determined, but in this case we could still use the KKT method to find $\mathbf{x}^\star$.

Why does the optimal point of `cq2` satisfy the KKT conditions while the optimal point of `cq1` does not? The answer lies in the geometry of their feasible sets. The example below, which I will call `cq3` (see §28.7.14), has the optimal point $\mathbf{x}^\star = [1-\frac{1}{\sqrt{2}}, 0]^\top$, which satisfies the KKT conditions.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2 - \frac{1}{2} + (x_1-1)^2 \le 0 \\ & f_2(\mathbf{x}) & = & -x_2 - \frac{1}{2} + (x_1-1)^2 \le 0 \end{array}$$



The picture on the left above shows the constraint contours and feasible set for cq3, along with lines drawn from $\mathbf{x}^\star$ tangent to the feasible set at that point. These lines delimit a **cone of tangents**, which is marked $\mathbb{T}$. To define the cone of tangents formally [1, §5.1.1] [5, Example 12.4], consider a sequence of feasible points $\mathbf{x}^1$, $\mathbf{x}^2 \ldots$ approaching $\mathbf{x}^\star$. Then

$$\mathbf{d} = \lim_{k\to\infty} \frac{\mathbf{x}^k - \mathbf{x}^\star}{\|\mathbf{x}^k - \mathbf{x}^\star\|}$$

is the limiting direction of the chord between $\mathbf{x}^k$ and $\mathbf{x}^\star$ as $\mathbf{x}^k$ approaches $\mathbf{x}^\star$. The cone of tangents $\mathbb{T}(\mathbf{x}^\star)$ is the set of all possible such limiting directions $\mathbf{d}$.

The picture on the right above shows the gradients of the active constraints at $\mathbf{x}^\star$ along with the **cone of feasible directions** that they determine,

$$\mathbb{F} = \{\mathbf{d} \in \mathbb{R}^n \mid \nabla f_i(\mathbf{x}^\star)^\top \mathbf{d} \le 0,\ i \in \mathbb{I}\}$$

where $\mathbb{I} = \{i \mid f_i(\mathbf{x}^\star) = 0\}$ are the indices of the active inequalities (here $\mathbb{I} = \{1, 2\}$).

In proving the first KKT theorem of §16.4 (see Exercise 16.11.37) it is necessary [1, §5.2] to establish in one way or another that $\mathbb{T} = \mathbb{F}$, which is called the **Abadie constraint qualification**. The sets $\mathbb{T}$ and $\mathbb{F}$ are equal if the gradients of the active constraints are linearly independent, as in cq3, but they can also be equal in other circumstances. In the cq2 problem, for example, the entire x_1 axis is feasible and we have $\mathbb{T} = \{\mathbf{d} \in \mathbb{R}^2 \mid d_2 = 0\}$. Using the gradients of the constraints, which are both active, we find

$$\nabla f_1(\mathbf{x}^\star)^\top \mathbf{d} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = d_2 \qquad \nabla f_2(\mathbf{x}^\star)^\top \mathbf{d} = \begin{bmatrix} 0 & -1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = -d_2$$

so $\mathbb{F} = \{\mathbf{d} \mid d_2 \le 0 \cap -d_2 \le 0\} = \{\mathbf{d} \mid d_2 = 0\}$ and $\mathbb{T} = \mathbb{F}$. At the optimal point of the cq1 problem the constraint gradients are the same as for cq2, so once again $\mathbb{F} = \{\mathbf{d} \mid d_2 = 0\}$. However, in cq1 the x_1 axis is feasible only to the left of $\mathbf{x}^\star$, so $\mathbb{T} = \{\mathbf{d} \mid d_2 = 0 \cap d_1 \le 0\}$, $\mathbb{T} \ne \mathbb{F}$, and the hypotheses of the theorem are not satisfied.

It is not always easy to find $\mathbb{T}$ or even $\mathbb{F}$ for a given constraint set, especially when $n > 2$. Fortunately, a hierarchy of stronger conditions have been discovered (linear independence being the strongest) which are easier to check and which imply the Abadie constraint qualification if they happen to be satisfied [1, §5.2] [108, Figure 7.3.2]. All of these conditions are called constraint qualifications, and any of them can be used to fulfill that hypothesis of the first KKT theorem. Various proofs have been provided based on these different conditions, but the conclusions of the theorem are true whenever $\mathbb{T} = \mathbb{F}$ (and the other hypotheses are satisfied) even if some stronger constraint qualification assumed in a proof, such as linear independence, is not satisfied.

There are special cases in which a constraint qualification is *always* satisfied.

- If the constraint functions are convex (as in a convex program) and the feasible set has an interior relative to $\mathbb{R}^n$ (it is not **flat**) then **Slater's condition** is satisfied. Recall from §3 that a feasible point $\hat{\mathbf{x}} \in \mathbb{R}^n$ is an interior point if $f_i(\hat{\mathbf{x}}) < 0$ for $i = 1 \ldots m$. The example `cq3` satisfies Slater's condition.

- If the active constraints are all linear functions (as in a linear program) then $\mathbb{T} = \mathbb{F}$ [5, Lemma 12.7]. The example `cq2` fits this description.

- If there is a single active constraint and its gradient is not zero then the linear independence condition is satisfied.

If an NLP has differentiable functions and a constraint qualification is satisfied at a local minimum $\bar{\mathbf{x}}$, then by the first KKT theorem $\bar{\mathbf{x}}$ is sure to be a KKT point. This does not rule out the possibility that a local minimum $\bar{\mathbf{x}}$ will satisfy the KKT conditions even if the hypotheses of the theorem are *not* met. In particular, it is possible (though no longer guaranteed) for a local minimum $\bar{\mathbf{x}}$ to satisfy the KKT conditions even if a constraint qualification is *not* satisfied there (see Exercise 16.11.35).

Some authors [4, §14.5.1] [78, §4.10] [107, §10.2] refer to a feasible point that satisfies a constraint qualification (or a particular constraint qualification) as a **regular point**.

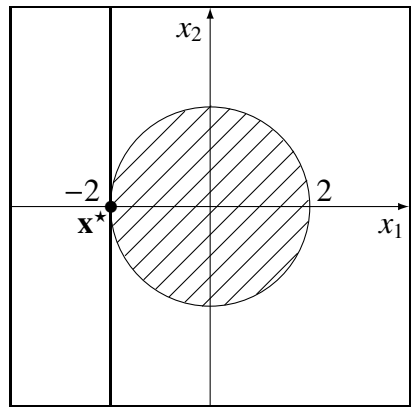
16.8 NLP Solution Phenomena

In our study of linear programming you might have been puzzled by some topics at first, but after you understood them you probably did not find them too surprising. In a world where everything obeys the laws of superposition and scaling, life is predictable, safe, and not overly stimulating. We have already noticed several ways in which nonlinear programs, especially when they are nonconvex, can be more interesting, perilous, and exciting. The most striking difference is that they can have local minima, which makes them a lot harder to solve, but there are also less obvious ways in which they can astonish and delight the intrepid student. This Section describes a few of them.

16.8.1 Redundant and Necessary Constraints

The problem below has the graphical solution shown on the right.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 - 4 \le 0 \\ & f_2(\mathbf{x}) & = & -x_1 - 2 \le 0 \end{array}$$



The gradients of the constraints are not independent at $\mathbf{x}^\star$, but this is a convex program and its feasible set has an interior so Slater's condition provides a constraint qualification. From the Lagrangian

$$\mathcal{L} = x_1 + \lambda_1(x_1^2 + x_2^2 - 4) + \lambda_2(-x_1 - 2)$$

we derive the following KKT conditions.

$$\begin{array}{rcl} \dfrac{\partial\mathcal{L}}{\partial x_1} & = & 1 + 2\lambda_1 x_1 - \lambda_2 = 0 \\ \dfrac{\partial\mathcal{L}}{\partial x_2} & = & 2\lambda_1 x_2 = 0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda_1} & = & x_1^2 + x_2^2 - 4 \le 0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda_2} & = & -x_1 - 2 \le 0 \\ \lambda_1 f_1(\mathbf{x}) & = & \lambda_1(x_1^2 + x_2^2 - 4) = 0 \\ \lambda_2 f_2(\mathbf{x}) & = & \lambda_2(-x_1 - 2) = 0 \\ \lambda_1 & \ge & 0 \\ \lambda_2 & \ge & 0 \end{array}$$

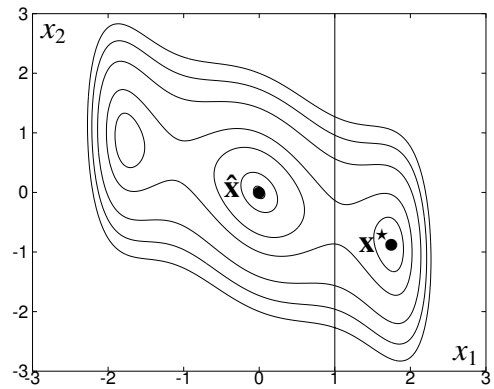
Solving these conditions we find $\mathbf{x}^\star = [-2, 0]^\top$ as shown in the picture, with $\boldsymbol{\lambda}^\star = [\frac{1}{4}, 0]^\top$. Both constraints are satisfied with equality, but because $\lambda_2^\star = 0$ we can deduce that the second one is redundant. Sure enough, removing it from the problem (such as by erasing its contour from the graphical solution) does not change the optimal point.

Now consider this problem (see §28.7.15) which I will call `branin` after the person who contrived the objective; that function is famous in unconstrained optimization as the **three-hump camel-back**. But I have introduced a constraint to bound x_1.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & 2x_1^2 - \frac{21}{20}x_1^4 + \frac{1}{6}x_1^6 + x_1x_2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & -x_1 + 1 \le 0 \end{array}$$

We can write the KKT conditions for `branin` in the usual way.

$$
\begin{aligned}
\mathcal{L} &= 2x_1^2 - \tfrac{21}{20}x_1^4 + \tfrac{1}{6}x_1^6 + x_1x_2 + x_2^2 + \lambda(-x_1+1) \\
\frac{\partial \mathcal{L}}{\partial x_1} &= 4x_1 - \tfrac{21}{5}x_1^3 + x_1^5 + x_2 - \lambda = 0 \\
\frac{\partial \mathcal{L}}{\partial x_2} &= x_1 + 2x_2 = 0 \\
\frac{\partial \mathcal{L}}{\partial \lambda} &= -x_1 + 1 \le 0 \\
\lambda f_1(\mathbf{x}) &= \lambda(-x_1+1) = 0 \\
\lambda &\ge 0
\end{aligned}
$$

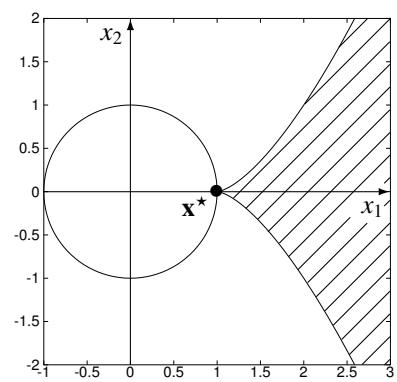


These conditions are satisfied at $\mathbf{x}^\star \approx [1.74755, -0.87372]^\top$ with $f_0(\mathbf{x}^\star) \approx 0.2986$ and $\lambda^\star = 0$. Thus the constraint is slack, as shown in the contour diagram on the right, and its shadow price is zero. If we remove it from the problem, however, the optimal point becomes the unconstrained minimum at $\hat{\mathbf{x}} = [0, 0]^\top$ with $f_0(\hat{\mathbf{x}}) = 0$. A constraint that is inactive at optimality can be omitted from a linear programming model, or from a convex nonlinear programming model, without changing the optimal point. In a nonconvex program, a constraint might be necessary, rather than redundant, even though its optimal KKT multiplier is zero.

16.8.2 Implicit Variable Bounds

The problem below (which is similar to [5, §15.3]) has the graphical solution shown on the right.

$$
\begin{aligned}
\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) &= x_1^2 + x_2^2 \\
\text{subject to} \quad f_1(\mathbf{x}) &= -(x_1-1)^3 + x_2^2 \le 0
\end{aligned}
$$



This problem has no constraint qualification (see Exercise 16.11.43) so we cannot solve it using the KKT method. However, because the constraint is active we might be able to use it to eliminate a variable.

$$
\begin{aligned}
x_2^2 &= (x_1-1)^3 \\
f_0(x_1) &= x_1^2 + (x_1-1)^3 \\
\frac{df_0}{dx_1} &= 2x_1 + 3(x_1-1)^2 = 2x_1 + 3(x_1^2 - 2x_1 + 1) = 0 \\
3x_1^2 - 4x_1 + 3 &= 0 \\
x_1 &= \frac{4 \pm \sqrt{(-4)^2 - 4(3)(3)}}{6} = \frac{4 \pm \sqrt{16-36}}{6} = \frac{4 \pm \sqrt{-20}}{6} \quad ⨳
\end{aligned}
$$

A complex value for x_1 has no meaning for the optimization problem, so something has gone wrong. What is the actual minimum value of the reduced objective?

$$\begin{aligned} f_0(x_1) &= x_1^3 - 2x_1^2 + 3x_1 - 1 \\ \lim_{x_1 \to -\infty} f_0(x_1) &= x_1^3 - [\text{lower order terms}] = -\infty \end{aligned}$$

The reduced objective is unbounded! To see how this happened, consider that

$$\begin{aligned} (x_1 - 1)^3 = x_2^2 &\Rightarrow (x_1 - 1)^3 \geq 0 \quad \text{in order for } x_2 \text{ to be real} \\ &\Rightarrow (x_1 - 1) \geq 0 \\ &\Rightarrow x_1 \geq 1. \end{aligned}$$

By eliminating the equality constraint we inadvertently removed from the problem the implicit constraint $x_1 \geq 1$, which could have been (and should have been) included *explicitly* in the model. The problem of minimizing the reduced objective subject to that requirement *does* have a constraint qualification, so we can solve it using the KKT method.

$$\begin{aligned} \underset{x_1 \in \mathbb{R}^1}{\text{minimize}} \quad f_0(x_1) &= x_1^2 + (x_1 - 1)^3 \\ \text{subject to} \quad f_1(x_1) &= -x_1 + 1 \leq 0 \end{aligned}$$

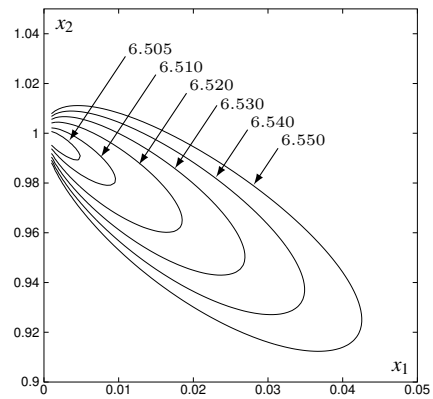
$$\begin{aligned} \mathcal{L} &= x_1^2 + (x_1 - 1)^3 + \lambda(-x_1 + 1) \\ \frac{\partial \mathcal{L}}{\partial x_1} &= 2x_1 + 3(x_1 - 1)^2 - \lambda = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= -x_1 + 1 \leq 0 \\ \lambda f_1(\mathbf{x}) &= \lambda(-x_1 + 1) = 0 \\ \lambda &\geq 0 \end{aligned}$$

These conditions have the unique solution $x_1^\star = 1$ with $\lambda^\star = 2$, and we deduce from the original constraint that $x_2^\star = (x_1^\star - 1)^3 = 0$ as we found graphically.

16.8.3 Ill-Posed Problems

A nonlinear program can, as I pointed out in §8.2.1, have a finite optimal value that is not a minimum and is therefore never attained. It is also possible for the optimal value to be attained at a finite point that cannot be found using the KKT theory because no constraint qualification is satisfied, as in the `cq1` problem of §16.7 or the first version of the example in §16.8.2. A more subtle variation on this theme is exemplified by the problem on the next page (see §28.7.16), which I will call `hearn` after its inventor [76].

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f_0(\mathbf{x}) = \frac{(1-x_2)^2}{2x_1} + \frac{(2-x_1)^2}{2x_2} + 5x_1 + 4x_2 + \frac{1}{2} \\ \text{subject to} \quad & \mathbf{x} \in \left\{\mathbf{x} \in \mathbb{R}^2 \,\middle|\, x_1 > 0, x_2 > 0\right\} \cup [0,1]^\top \cup [2,0]^\top \end{aligned}$$



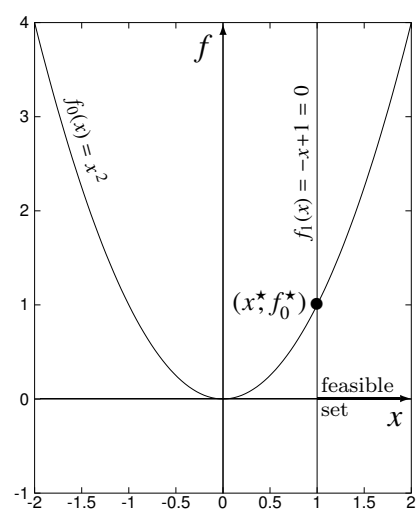
From the contour plot shown to the right we can guess that $\mathbf{x}^\star = [0,1]^\top$ and $f_0(\mathbf{x}^\star) = 6\frac{1}{2}$. Unfortunately, f_0 cannot be evaluated at that point (this accounts for the missing parts of the contours near $x_1 = 0$). There is only one active constraint so the linear independence constraint qualification is satisfied, but it is hard to use the KKT theory to find $\mathbf{x}^\star$ because $\nabla_{\mathbf{x}}\mathcal{L}$ is not defined there. Problems like `hearn` are said to be **ill-posed** [105, p123] because the nonlinear programming model breaks down at the optimal point. We will also consider a problem to be ill-posed if (like this one) the feasible set does not include all of its boundary points, or if it has infima instead of minima, or if it lacks a constraint qualification, or [2, p79-80] if it is badly-scaled.

Nonconvexity is a property of nonlinear programs that often cannot be avoided in practical applications, but an ill-posed model must always be suspected of being unrealistic (bilevel programs such as the one we studied in §1.6, which always lack a constraint qualification, are a rare exception). Some ill-posed problems (e.g., `cq1` and `hearn`) yield to numerical methods, but others so far do not. From now on we will assume that the nonlinear programs we are trying to solve are well-posed.

16.9 Duality in Nonlinear Programming

This one-dimensional optimization has the graphical solution to the right.

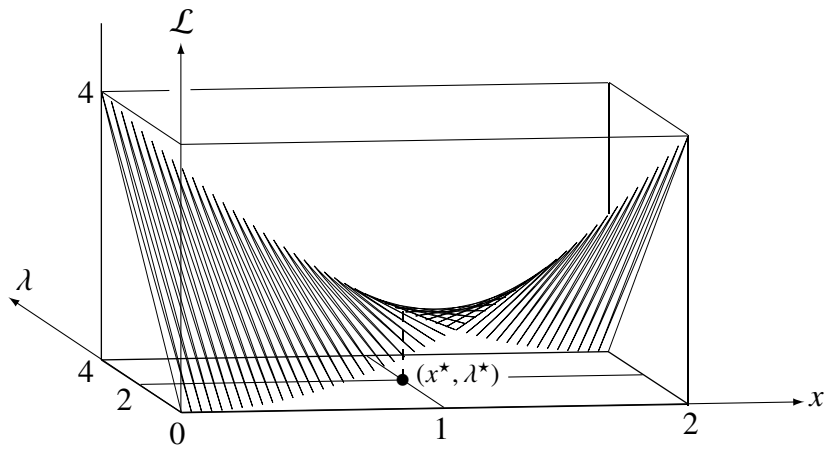
$$\begin{aligned} \underset{x\in\mathbb{R}^1}{\text{minimize}} \quad f_0(x) &= x^2 \\ \text{subject to} \quad f_1(x) &= -x+1 \le 0 \end{aligned}$$



Its Lagrangian yields the KKT conditions below, which are satisfied at $x^\star = 1$ with $\lambda^\star = 2$.

$$\begin{aligned} \mathcal{L}(x,\lambda) &= x^2 + \lambda(-x+1) \\ \frac{\partial \mathcal{L}}{\partial x} &= 2x - \lambda = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= -x+1 \le 0 \\ \lambda f_1(x) &= \lambda(-x+1) = 0 \\ \lambda &\ge 0 \end{aligned}$$

Because the problem has only one x variable and one KKT multiplier, we can draw the surface plot of $\mathcal{L}(x, \lambda)$ shown below (I generated data with a FORTRAN program and then used `gnuplot`).



The Lagrangian goes up if we move from $(x^\star, \lambda^\star)$ either way along the x direction and it stays flat if we move from $(x^\star, \lambda^\star)$ either way along the λ direction. In other words, the minimizing point $(x^\star, \lambda^\star)$ of the Lagrangian satisfies this definition [161, §2.6] of a **saddle point**:

$$\mathcal{L}(x^\star, \lambda) \le \mathcal{L}(x^\star, \lambda^\star) \le \mathcal{L}(x, \lambda^\star) \quad \text{for all } (x, \lambda).$$

In the picture, at each possible value of x there is some value of λ where the Lagrangian takes on its highest value. For which value of x is that maximum Lagrangian value the *lowest*? When $x = 0$ we have (from the formula for $\mathcal{L}$ on the previous page) $\mathcal{L}(\lambda) = \lambda$, so in the picture the surface has height 4 at $\lambda = 4$, and it gets higher as λ increases outside the frame of the picture. When $x = 2$ we have $\mathcal{L}(\lambda) = 4 - \lambda$, so the surface has height 4 at $\lambda = 0$, and it gets higher if λ becomes negative. But at $x = 1$, $\mathcal{L}(\lambda) = 1$ for every value of λ, and that is the lowest value over x of the highest Lagrangian over λ. Thus, $x^\star$ solves this problem.

$$\underset{x}{\text{minimize}} \left\{ \sup_{\lambda} \mathcal{L}(x, \lambda) \right\}$$

Because the highest value of $\mathcal{L}(\lambda)$ at a given $x \ne x^\star$ is not attained at a finite value of λ, here I have used the supremum operator to describe this value, rather than the maximum.

In the picture, at each possible value of λ there is some value of x where the Lagrangian takes on its lowest value. For which value of λ is that minimum Lagrangian the *highest*? For this problem, it happens when $\lambda = 2$, and by reasoning similar to that above $\lambda^\star$ solves this problem.

$$\underset{\lambda}{\text{maximize}} \left\{ \inf_{x} \mathcal{L}(x, \lambda) \right\}$$

In case the lowest value of $\mathcal{L}(x)$ at a given $\lambda \ne \lambda^\star$ is not attained at a finite value of x, here I have used the infimum operator over x rather than the minimum.

For our example we have $\mathcal{L}(x,\lambda) = x^2 + \lambda(-x+1)$, and we find

$$\sup_\lambda \mathcal{L}(x,\lambda) = \begin{cases} 1 & \text{for } x = 1 \\ \infty & \text{for } x \neq 1 \end{cases}$$

If $x = 1$ then $\mathcal{L} = 1$ for all values of λ. If $x > 1$ then $(-x+1) < 0$ and we can make $\mathcal{L}$ as high as we like by letting $\lambda \to -\infty$. If $x < 1$ then $(-x+1) > 0$ and we can make $\mathcal{L}$ as high as we like by letting $\lambda \to +\infty$.

Looking in the other direction, we find

$$\inf_x \mathcal{L}(x,\lambda) = \begin{cases} 0 & \text{for } \lambda = 0 \\ \lambda - \frac{1}{4}\lambda^2 & \text{for } \lambda \neq 0 \end{cases}$$

If $\lambda = 0$ then $\mathcal{L} = x^2$, which is lowest at $x = 0$, where $\mathcal{L} = 0$. If $\lambda \neq 0$ then $\mathcal{L} = x^2 + \lambda(-x+1)$ is lowest where

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x} &= 2x - \lambda = 0 \\ x &= \tfrac{1}{2}\lambda \end{aligned}$$

and at that value of x we have

$$\begin{aligned} \mathcal{L}(\lambda) &= (\tfrac{1}{2}\lambda)^2 + \lambda(-\tfrac{1}{2}\lambda + 1) \\ &= \tfrac{1}{4}\lambda^2 - \tfrac{1}{2}\lambda^2 + \lambda \\ &= \lambda - \tfrac{1}{4}\lambda^2. \end{aligned}$$

Thus, we find that

$$\min_x \sup_\lambda \mathcal{L} = \min_x \{1, \infty\} = 1 \quad \text{at } x^\star = 1$$

and

$$\max_\lambda \inf_x \mathcal{L} = \max_\lambda \left\{0, \lambda - \tfrac{1}{4}\lambda^2\right\} = \max_\lambda \left\{\lambda - \tfrac{1}{4}\lambda^2\right\}$$

Letting $w = \lambda - \frac{1}{4}\lambda^2$ we can perform the indicated maximization like this.

$$\begin{aligned} \frac{dw}{d\lambda} &= 1 - \tfrac{1}{2}\lambda = 0 \\ \lambda^\star &= 2. \end{aligned}$$

The graph of $\mathcal{L}(\mathbf{x},\boldsymbol{\lambda})$ is a hypersurface in $\mathbb{R}^{n+m}$ and is therefore usually hard to visualize, but as in this example it is true in general [1, Theorems 6.2.5-6] that if a nonlinear program is convex and has a constraint qualification then its Lagrangian has a saddle point, every saddle point of the Lagrangian satisfies the KKT conditions for the nonlinear program, and every KKT point is a saddle point.

16.9.1 The Lagrangian Dual

In the analysis above we assumed nothing about the sign of λ, but we would reach the same conclusions if we assumed it to be nonnegative (as we know from §16.2 that it must be at the optimal point). Assuming now that $\boldsymbol{\lambda} \geq \mathbf{0}$ and using the same sort of reasoning we applied to the example, we can find the "min sup" and "max inf" problems corresponding to the standard form nonlinear program,

$$\begin{array}{rll} \text{NLP:} & \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) \\ & \text{subject to} & f_i(\mathbf{x}) \leq 0, \ i = 1 \ldots m. \end{array}$$

This problem has

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f_0 + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x})$$

so we can deduce that

$$\sup_{\boldsymbol{\lambda}} \mathcal{L} = \begin{cases} f_0(\mathbf{x}) & \text{if } f_i(\mathbf{x}) \leq 0 \ \text{ for } i = 1 \ldots m \\ \infty & \text{otherwise.} \end{cases}$$

If even one constraint function is positive then we can make $\mathcal{L}$ as big as we like by letting the corresponding λ_i approach infinity. However, if $f_i(\mathbf{x}) \leq 0$ for $i = 1 \ldots m$ then including any of them will reduce $\mathcal{L}$, so its supremum is when $\boldsymbol{\lambda} = \mathbf{0}$ and $\mathcal{L} = f_0(\mathbf{x})$. Then

$$\min_{\mathbf{x}} \sup_{\boldsymbol{\lambda}} \mathcal{L} = \min_{\mathbf{x}} \{\infty, \ (f_0(\mathbf{x}) \text{ provided that } f_i(\mathbf{x}) \leq 0, \ i = 1 \ldots m)\}$$

so the minimum over $\mathbf{x}$ of the supremum of $\mathcal{L}$ over $\boldsymbol{\lambda}$ is the solution to the **primal problem**

$$\begin{array}{rll} \mathscr{P}: & \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) \\ & \text{subject to} & f_i(\mathbf{x}) \leq 0, \ i = 1 \ldots m, \end{array}$$

which is just NLP again. The maximum over $\boldsymbol{\lambda}$ of the infimum over $\mathbf{x}$ is the solution to the **Lagrangian dual problem**,

$$\begin{array}{rll} \mathscr{D}: & \underset{\boldsymbol{\lambda}\in\mathbb{R}^m}{\text{maximize}} & \theta(\boldsymbol{\lambda}) = \inf\limits_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) \\ & \text{subject to} & \lambda_i \geq 0, \ i = 1 \ldots m, \end{array}$$

which is the "max inf" problem with the added harmless assumption we used above that the KKT multipliers are nonnegative.

The primal and dual of a nonlinear program are related just as the primal and dual of a linear program are related, but in ways that are in some cases more subtle [1, Theorems 6.2.1,4] [5, Theorem 12.13] [109]. The main results are summarized on the next page.

NLP Duality Relations

1. If $\bar{\mathbf{x}}$ is feasible for $\mathscr{P}$ and $\bar{\boldsymbol{\lambda}}$ is feasible for $\mathscr{D}$, then $f_0(\bar{\mathbf{x}}) \geq \theta(\bar{\boldsymbol{\lambda}})$. If $f_0(\bar{\mathbf{x}}) > \theta(\bar{\boldsymbol{\lambda}})$, the difference between them is called the **duality gap**.

2. If $\bar{\mathbf{x}}$ is feasible for $\mathscr{P}$ and $\bar{\boldsymbol{\lambda}}$ is feasible for $\mathscr{D}$, and if also $f_0(\bar{\mathbf{x}}) = \theta(\bar{\boldsymbol{\lambda}})$, then $\bar{\mathbf{x}}$ solves $\mathscr{P}$ and $\bar{\boldsymbol{\lambda}}$ solves $\mathscr{D}$.

3. If $\mathscr{D}$ is unbounded, then $\mathscr{P}$ is infeasible.

4. If $\mathscr{P}$ is unbounded, then $\mathscr{D}$ is also unbounded.

5. If NLP is a convex program and Slater's constraint qualification is satisfied, then $f_0(\mathbf{x}^\star) = \theta(\boldsymbol{\lambda}^\star)$.

6. If NLP has each $f_i(\mathbf{x})$ differentiable and convex, and $\mathbf{x}^\star$ solves $\mathscr{P}$, and a constraint qualification is satisfied at $\mathbf{x}^\star$, and $\boldsymbol{\lambda}^\star$ solves $\mathscr{D}$ with $\inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ occurring at $\bar{\mathbf{x}}$, and if $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ is a *strictly* convex function of $\mathbf{x}$ at $\bar{\mathbf{x}}$, then $\bar{\mathbf{x}} = \mathbf{x}^\star$.

As discussed in §15.3, the dual variables λ_i can be viewed as shadow prices, so slack primal constraints $f_i(\mathbf{x}^\star) < 0$ correspond to zero KKT multipliers $\lambda_i^\star = 0$ and positive KKT multipliers $\lambda_i > 0$ correspond to tight primal constraints $f_i(\mathbf{x}^\star) = 0$.

As in linear programming it sometimes turns out that the dual of a nonlinear program is easier to solve than the primal. If the rather demanding provisions of NLP Duality Relation 6 are met (or, if the duality gap is zero, maybe even if they are not) the primal solution can be recovered from the dual. To exploit this fact special numerical methods have been developed for solving the Lagrangian dual problem [1, §6.4-6.5].

The Lagrangian dual can be constructed, and NLP Duality Relations 1-4 can be used, even if $\mathscr{P}$ is not a convex program [1, Example 6.2.2] and even if its objective and constraint functions are not differentiable. Lagrangian duality has therefore also been used in the development of alternatives to the branch-and-bound algorithm for integer programming.

16.9.2 The Wolfe Dual

The Lagrangian dual can be hard to use in practice because of the need to find the global infimum of $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda})$, but in some settings its great virtue of being indifferent to nonconvexity and nondifferentiability might not actually be needed. If NLP has each $f_i(\mathbf{x})$ convex and continuously differentiable (each derivative $\partial f_i / \partial x_j$ exists and is itself continuous [148, p151]) then for a fixed $\bar{\boldsymbol{\lambda}}$, $\inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \bar{\boldsymbol{\lambda}})$ occurs at the point $\bar{\mathbf{x}}$ if and only if $\nabla_{\mathbf{x}} \mathcal{L}(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}}) = \mathbf{0}$ [4, §14.8.3] [161, §2.6.1]. This is just an application of the first-order necessary conditions from §10.7.

Then if we maximize $\mathcal{L}(\mathbf{x},\boldsymbol{\lambda})$ over $\boldsymbol{\lambda}$ while insisting that $\nabla_{\mathbf{x}}\mathcal{L}(\bar{\mathbf{x}},\bar{\boldsymbol{\lambda}}) = \mathbf{0}$, we are really just maximizing $\theta(\boldsymbol{\lambda})$, so we can rewrite $\mathscr{D}$ in the form of the **Wolfe dual problem**

$$\begin{array}{rll} \mathscr{D}: & \underset{\boldsymbol{\lambda}\in\mathbb{R}^m}{\text{maximize}} & \mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) \\ & \text{subject to} & \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) = \mathbf{0} \\ & & \lambda_i \geq 0,\ i = 1\ldots m. \end{array}$$

To use the Wolfe dual (which is also referred to as the **classical dual** because it was discovered first) NLP must be a convex program. If it also satisfies Slater's condition then NLP Duality Relation 5 ensures there is no duality gap. If in addition one or more of the $f_i(\mathbf{x})$ happen to be strictly convex, so that $\mathcal{L}$ is strictly convex, then Relation 6 ensures that solving the Wolfe dual will produce $\mathbf{x}^\star$ along with $\boldsymbol{\lambda}^\star$.

16.9.3 Some Handy Duals

LINEAR PROGRAMS. The LP below is the minimization problem of the standard dual pair first introduced in §5.

$$\begin{array}{rlrcl} \mathscr{P}: & \text{minimize} & \mathbf{c}^\top\mathbf{x} & & \\ & \text{subject to} & \mathbf{A}\mathbf{x} & \geq & \mathbf{b} \\ & & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

This is an instance of NLP in which the functions happen all to be linear, so it meets the requirements to have a Wolfe dual. According to the prescription in §16.9.2, that is

$$\begin{array}{rrcl} \text{maximize} & \mathcal{L}(\mathbf{x},\mathbf{y},\boldsymbol{\lambda}) & = & \mathbf{c}^\top\mathbf{x} + \mathbf{y}^\top(\mathbf{b}-\mathbf{A}\mathbf{x}) + \boldsymbol{\lambda}^\top(-\mathbf{x}) \\ \text{subject to} & \nabla_{\mathbf{x}}\mathcal{L} & = & \mathbf{c} - \mathbf{A}^\top\mathbf{y} - \boldsymbol{\lambda} = \mathbf{0} \\ & \mathbf{y} & \geq & \mathbf{0} \\ & \boldsymbol{\lambda} & \geq & \mathbf{0} \end{array}$$

where $\mathbf{y}$ is a vector of KKT multipliers corresponding to the rows of $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and $\boldsymbol{\lambda}$ is a vector of KKT multipliers corresponding to the rows of $\mathbf{x} \geq \mathbf{0}$. Using the equality constraint, we can rewrite the objective like this.

$$\begin{aligned} \mathbf{c}^\top\mathbf{x} + \mathbf{y}^\top(\mathbf{b}-\mathbf{A}\mathbf{x}) + \boldsymbol{\lambda}^\top(-\mathbf{x}) &= \mathbf{c}^\top\mathbf{x} + \mathbf{y}^\top\mathbf{b} - \mathbf{y}^\top\mathbf{A}\mathbf{x} - \boldsymbol{\lambda}^\top\mathbf{x} \\ &= (\mathbf{c}^\top - \mathbf{y}^\top\mathbf{A} - \boldsymbol{\lambda}^\top)\mathbf{x} + \mathbf{y}^\top\mathbf{b} \\ &= (\mathbf{c} - \mathbf{A}^\top\mathbf{y} - \boldsymbol{\lambda})^\top\mathbf{x} + \mathbf{y}^\top\mathbf{b} \\ &= \mathbf{y}^\top\mathbf{b} \end{aligned}$$

The constraints can also be simplified, because

$$\left.\begin{array}{r} \mathbf{c} - \mathbf{A}^\top\mathbf{y} = \boldsymbol{\lambda} \\ \boldsymbol{\lambda} \geq \mathbf{0} \end{array}\right\} \Rightarrow \mathbf{c} - \mathbf{A}^\top\mathbf{y} \geq \mathbf{0}.$$

Thus the dual of the primal LP is

$$\begin{array}{rrcl} \mathscr{D}: \text{ maximize} & \mathbf{b}^\top\mathbf{y} & & \\ \text{subject to} & \mathbf{A}^\top\mathbf{y} & \leq & \mathbf{c} \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

which is the max problem of our standard dual pair.

QUADRATIC PROGRAMS. Recall from §14.1 that a quadratic program has the form

$$\begin{array}{rrcl} \mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & = & \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} & \leq & \mathbf{c} \end{array}$$

where $\mathbf{Q}$ is a symmetric matrix. The functions are continuously differentiable, so if $\mathbf{Q}$ is positive definite we can write its Wolfe dual as

$$\begin{array}{rrcl} \underset{\boldsymbol{\lambda}\in\mathbb{R}^m}{\text{maximize}} & \mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) & = & \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} - \mathbf{b}^\top\mathbf{x} + \boldsymbol{\lambda}^\top(\mathbf{A}\mathbf{x} - \mathbf{c}) \\ \text{subject to} & \nabla_{\mathbf{x}}\mathcal{L} & = & \mathbf{Q}\mathbf{x} - \mathbf{b} + \mathbf{A}^\top\boldsymbol{\lambda} = \mathbf{0} \\ & \boldsymbol{\lambda} & \geq & \mathbf{0}. \end{array}$$

Solving the equality constraint for $\mathbf{x}$ we find

$$\begin{aligned} \mathbf{Q}\mathbf{x} - \mathbf{b} + \mathbf{A}^\top\boldsymbol{\lambda} &= \mathbf{0} \\ \mathbf{Q}\mathbf{x} &= \mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda} \\ \mathbf{x} &= \mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda}), \end{aligned}$$

which we can substitute into the dual objective to obtain an optimization in terms of only $\boldsymbol{\lambda}$. I did the calculation one term at a time, as follows.

$$\begin{aligned} \mathbf{x}^\top\mathbf{Q}\mathbf{x} &= [\mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda})]^\top\mathbf{Q}[\mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda})] \\ &= (\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda})^\top\mathbf{Q}^{-\top}\mathbf{Q}\mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda}) \\ &= (\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda})^\top\mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda}) \\ &= \mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{b} - 2\mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} + \boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} \\ \mathbf{b}^\top\mathbf{x} &= \mathbf{b}^\top[\mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda})] \\ &= \mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{b} - \mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} \\ \boldsymbol{\lambda}^\top(\mathbf{A}\mathbf{x} - \mathbf{c}) &= \boldsymbol{\lambda}^\top\mathbf{A}[\mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda})] - \boldsymbol{\lambda}^\top\mathbf{c} \\ &= \boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{b} - \boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} - \boldsymbol{\lambda}^\top\mathbf{c} \end{aligned}$$

Substituting the final expression for each quantity into the dual objective yields the result on the next page.

$$\begin{aligned}\mathcal{L}(\boldsymbol{\lambda}) &= \tfrac{1}{2}\mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{b} - \mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} + \tfrac{1}{2}\boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} - \mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{b} + \mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} \\ &\qquad + \boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{b} - \boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} - \boldsymbol{\lambda}^\top\mathbf{c} \\ &= -\tfrac{1}{2}\mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{b} - \tfrac{1}{2}\boldsymbol{\lambda}^\top\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^\top\boldsymbol{\lambda} + \boldsymbol{\lambda}^\top(\mathbf{A}\mathbf{Q}^{-1}\mathbf{b} - \mathbf{c})\end{aligned}$$

Thus the dual of the quadratic program is

$$\begin{array}{lrcl}\mathscr{D}: \underset{\boldsymbol{\lambda}\in\mathbb{R}^m}{\text{maximize}} & \mathcal{L}(\boldsymbol{\lambda}) & = & -\tfrac{1}{2}\mathbf{b}^\top\mathbf{Q}^{-1}\mathbf{b} - \tfrac{1}{2}\boldsymbol{\lambda}^\top(\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^\top)\boldsymbol{\lambda} + \boldsymbol{\lambda}^\top(\mathbf{A}\mathbf{Q}^{-1}\mathbf{b} - \mathbf{c}) \\ \text{subject to} & \boldsymbol{\lambda} & \geq & \mathbf{0}.\end{array}$$

Although this problem is a quadratic program like the primal, its constraints are simply nonnegativities. That makes it easy to solve $\mathscr{D}$ numerically, either as an unconstrained problem by enforcing lower bounds of zero in the line search (see §12.2.2) or by using a special-purpose algorithm such as gradient projection [5, §16.7]. The constraints of $\mathscr{P}$ are linear so a constraint qualification is satisfied (see §16.7) and according to NLP Duality Relation 6 we can recover the primal solution as $\mathbf{x}^\star = \mathbf{Q}^{-1}(\mathbf{b} - \mathbf{A}^\top\boldsymbol{\lambda}^\star)$. Once again we see how pleasant life can be in that tiny neighborhood of the nonlinear programming universe where everything is perfectly smooth and strictly convex!

SUPPORT VECTOR MACHINES. In §8.7.4 we studied the formulation of one particular strictly convex quadratic program, the soft-margin SVM.

$$\begin{array}{lrcll}\underset{\mathbf{p}\,q\,\boldsymbol{\xi}}{\text{minimize}} & \mathbf{p}^\top\mathbf{p} + c\displaystyle\sum_{i=1}^{n}\xi_i & & & \\ \text{subject to} & y_i(\mathbf{p}^\top\mathbf{x}_i + q) & \geq & 1 - \xi_i & i = 1\ldots n \\ & \xi_i & \geq & 0 & i = 1\ldots n\end{array}$$

Recall that in this model n is the number of data points and m is the number of dimensions. The vectors $\mathbf{x}_i \in \mathbb{R}^m$, $i = 1\ldots n$ and $\mathbf{y} \in \mathbb{R}^n$ are the scaled constant data of the problem, and the compromise parameter $c > 0$ is a fixed scalar. The unknowns to be determined by the optimization are the predictor variables $\mathbf{p} \in \mathbb{R}^m$ and intercept $q \in \mathbb{R}^1$, and the resulting classification errors $\boldsymbol{\xi} \in \mathbb{R}^n$.

To derive the Wolfe dual of this problem it is prudent for sanity to first restate it in a more compact form. First consider the dot products that appear in the first n constraints,

$$\mathbf{p}^\top\mathbf{x}_i = \mathbf{x}_i^\top\mathbf{p} = [x_{i1}\ldots x_{im}]\begin{bmatrix} p_1 \\ \vdots \\ p_m\end{bmatrix} \quad i = 1\ldots n.$$

If we make the vectors $\mathbf{x}_i$ the columns of an $m \times n$ matrix $\mathbf{X}$, then we can represent all of these dot products by the single matrix-vector product $\mathbf{X}^\top\mathbf{p}$ shown on the next page.

$$\mathbf{X}^\top\mathbf{p} = \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix} \begin{bmatrix} p_1 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} \mathbf{x_1}^\top\mathbf{p} \\ \vdots \\ \mathbf{x_n}^\top\mathbf{p} \end{bmatrix}$$

To add q to each row, we can add the vector $q\mathbf{1}$ to this matrix-vector product, where $\mathbf{1} \in \mathbb{R}^n$ is a vector of all 1s. To multiply each row by its y_i, we can make the y_i values the diagonal entries of an $n \times n$ diagonal matrix $\mathbf{Y}$ and premultiply by $\mathbf{Y}$. A vector that represents $1 - \xi_i$ for $i = 1 \ldots n$ is $\mathbf{1} - \boldsymbol{\xi}$. Using these ideas the first n scalar constraints can be replaced by the vector constraint

$$\begin{aligned} \mathbf{Y}(\mathbf{X}^\top\mathbf{p} + q\mathbf{1}) &\geq \mathbf{1} - \boldsymbol{\xi} \\ \text{or} \qquad \mathbf{Y}\mathbf{X}^\top\mathbf{p} + q\mathbf{y} &\geq \mathbf{1} - \boldsymbol{\xi} \end{aligned}$$

where the last step uses the fact that $\mathbf{Y1} = \mathbf{y}$. Finally, we can restate the SVM primal problem like this.

$$\begin{aligned} \mathscr{P}: \ \underset{\mathbf{p}\,q\,\boldsymbol{\xi}}{\text{minimize}} \quad & \mathbf{p}^\top\mathbf{p} + c\mathbf{1}^\top\boldsymbol{\xi} \\ \text{subject to} \quad & \mathbf{Y}\mathbf{X}^\top\mathbf{p} + q\mathbf{y} \geq \mathbf{1} - \boldsymbol{\xi} \\ & \boldsymbol{\xi} \geq \mathbf{0} \end{aligned}$$

The Lagrangian of this problem is

$$\begin{aligned} \mathcal{L}(\mathbf{p}, q, \boldsymbol{\xi}) &= \mathbf{p}^\top\mathbf{p} + c\mathbf{1}^\top\boldsymbol{\xi} + \boldsymbol{\lambda}^\top(\mathbf{1} - \boldsymbol{\xi} - \mathbf{Y}\mathbf{X}^\top\mathbf{p} - q\mathbf{y}) + \boldsymbol{\gamma}^\top(-\boldsymbol{\xi}) \\ &= \mathbf{p}^\top\mathbf{p} + c\mathbf{1}^\top\boldsymbol{\xi} + \boldsymbol{\lambda}^\top\mathbf{1} - \boldsymbol{\lambda}^\top\boldsymbol{\xi} - \boldsymbol{\lambda}^\top\mathbf{Y}\mathbf{X}^\top\mathbf{p} - q\boldsymbol{\lambda}^\top\mathbf{y} - \boldsymbol{\gamma}^\top\boldsymbol{\xi} \end{aligned}$$

The first constraint in the Wolfe dual is that the gradient of the Lagrangian with respect to the variables of optimization is zero. Starting with the $\mathbf{p}$ variables, we have

$$\begin{aligned} \nabla_{\mathbf{p}}\mathcal{L} &= 2\mathbf{p} - (\boldsymbol{\lambda}^\top\mathbf{Y}\mathbf{X}^\top)^\top = \mathbf{0} \\ \mathbf{p} &= \tfrac{1}{2}\mathbf{X}\mathbf{Y}^\top\boldsymbol{\lambda} = \tfrac{1}{2}\mathbf{X}\mathbf{Y}\boldsymbol{\lambda} \end{aligned}$$

where the last step makes use of the fact that the diagonal matrix $\mathbf{Y}$ is its own transpose. Continuing with the other variables, we also have

$$\begin{aligned} \nabla_q\mathcal{L} &= \frac{\partial\mathcal{L}}{\partial q} = -\boldsymbol{\lambda}^\top\mathbf{y} = 0 \\ \nabla_{\boldsymbol{\xi}}\mathcal{L} &= c\mathbf{1} - \boldsymbol{\lambda} - \boldsymbol{\gamma} = 0. \end{aligned}$$

Using these relations we can simplify the Lagrangian and rewrite it in terms of only the KKT multipliers $\boldsymbol{\lambda}$ and $\boldsymbol{\gamma}$ and the problem data, as follows.

$$\begin{aligned} \mathcal{L} &= (\tfrac{1}{2}\mathbf{X}\mathbf{Y}\boldsymbol{\lambda})^\top(\tfrac{1}{2}\mathbf{X}\mathbf{Y}\boldsymbol{\lambda}) - (\boldsymbol{\lambda}^\top\mathbf{Y}\mathbf{X}^\top)(\tfrac{1}{2}\mathbf{X}\mathbf{Y}\boldsymbol{\lambda}) + (c\mathbf{1}^\top - \boldsymbol{\lambda}^\top - \boldsymbol{\gamma}^\top)\boldsymbol{\xi} + \boldsymbol{\lambda}^\top\mathbf{1} - q\boldsymbol{\lambda}^\top\mathbf{y} \\ &= -\tfrac{1}{4}(\mathbf{X}\mathbf{Y}\boldsymbol{\lambda})^\top(\mathbf{X}\mathbf{Y}\boldsymbol{\lambda}) + \boldsymbol{\lambda}^\top\mathbf{1} \\ &= -\tfrac{1}{4}\boldsymbol{\lambda}^\top(\mathbf{Y}\mathbf{X}^\top\mathbf{X}\mathbf{Y})\boldsymbol{\lambda} + \boldsymbol{\lambda}^\top\mathbf{1} \end{aligned}$$

Then the Wolfe dual is

$$\begin{array}{lrcl} \underset{\boldsymbol{\lambda},\boldsymbol{\gamma}}{\text{maximize}} & \mathcal{L}(\boldsymbol{\lambda}) & = & \boldsymbol{\lambda}^\top\mathbf{1} - \frac{1}{4}\boldsymbol{\lambda}^\top(\mathbf{Y}\mathbf{X}^\top\mathbf{X}\mathbf{Y})\boldsymbol{\lambda} \\ \text{subject to} & \boldsymbol{\lambda}^\top\mathbf{y} & = & 0 \\ & c\mathbf{1} - \boldsymbol{\lambda} - \boldsymbol{\gamma} & = & 0 \\ & \boldsymbol{\lambda} & \geq & \mathbf{0} \\ & \boldsymbol{\gamma} & \geq & \mathbf{0}. \end{array}$$

This problem can be further simplified, because

$$\left.\begin{array}{r} c\mathbf{1} - \boldsymbol{\lambda} = \boldsymbol{\gamma} \\ \boldsymbol{\gamma} \geq \mathbf{0} \end{array}\right\} \Rightarrow c\mathbf{1} - \boldsymbol{\lambda} \geq \mathbf{0} \Rightarrow \boldsymbol{\lambda} \leq c\mathbf{1}.$$

Thus we can write the SVM dual as

$$\begin{array}{lrcl} \mathscr{D}: \underset{\boldsymbol{\lambda}}{\text{maximize}} & \mathcal{L}(\boldsymbol{\lambda}) & = & \boldsymbol{\lambda}^\top\mathbf{1} - \frac{1}{4}\boldsymbol{\lambda}^\top\mathcal{K}\boldsymbol{\lambda} \\ \text{subject to} & \boldsymbol{\lambda}^\top\mathbf{y} & = & 0 \\ & \boldsymbol{\lambda} & \geq & \mathbf{0} \\ & \boldsymbol{\lambda} & \leq & c\mathbf{1} \end{array}$$

where the **kernel** $\mathcal{K} = \mathbf{Y}\mathbf{X}^\top\mathbf{X}\mathbf{Y}$ is a constant matrix that depends only on the data. This dual is easier than the primal, because it has n variables rather than $m + n + 1$ and its only non-bound constraint $\boldsymbol{\lambda}^\top\mathbf{y} = 0$ is a linear equality.

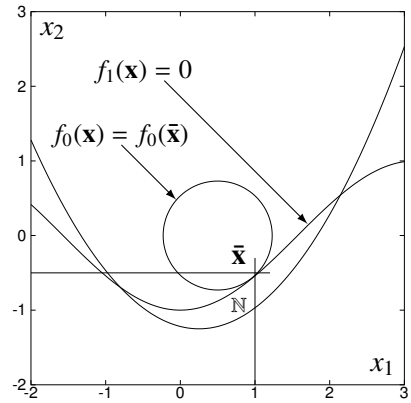
However, the main virtue of the SVM dual is that it permits the use of nonlinear classifiers. By replacing the kernel $\mathcal{K}$ by a different function of the data it is possible to separate the observations into categories based not a hyperplane but on a curved hypersurface [4, §14.8.5]. This extension to the original SVM model shows that duality can play an important role not only in the solution of nonlinear programming problems but also in their formulation.

16.10 Finding KKT Multipliers Numerically

The reason for solving a nonlinear program is usually to find $\mathbf{x}^\star$, because the optimal decision variables tell us what to do in the application setting that gave rise to the optimization problem. However, $\boldsymbol{\lambda}^\star$ is often also of interest, because the dual variables are shadow prices that tell which constraints are active and how strongly they affect the optimal objective value. As we shall see in §26.3.1, $\boldsymbol{\lambda}^\star$ is also used in the measurement of solution error when evaluating the performance of an algorithm by computational experiments.

When we solve a primal nonlinear program analytically by the KKT method, we find $\boldsymbol{\lambda}^\star$ along with $\mathbf{x}^\star$. When we solve a dual nonlinear program analytically we obviously get $\boldsymbol{\lambda}^\star$, and according to NLP Duality Relation 6 we might be able to recover $\mathbf{x}^\star$. But most of this book is about *numerical* methods, and most of them deliver an approximation to $\mathbf{x}^\star$ only. Given such a near-optimal (or maybe not-so-near-optimal) point, is there some way that we can find the corresponding KKT multipliers $\boldsymbol{\lambda}^\star$?

To explore this question consider the problem below, which I will call it `nset` (see §28.7.17).



$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1 - \frac{1}{2})^2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & \cos(x_1) + x_2 \leq 0 \\ & f_2(\mathbf{x}) & = & \frac{1}{2}(x_1 - \frac{1}{4})^2 - x_2 - 1\frac{1}{4} \leq 0 \end{array}$$

The feasible region of this nonlinear program is the set $\mathbb{N}$ described in §16.6 and pictured again to the right. The optimal contour of the objective is drawn tangent to the feasible set at $\mathbf{x}^\star$, which might be approximated by a numerical algorithm (or by us looking at the graph) as $\bar{\mathbf{x}} = [1, -\frac{1}{2}]^\top$. A point that is in the boundary of the feasible set satisfies the feasibility and orthogonality conditions for this problem, with $\lambda_2 = 0$ and with $\lambda_1 > 0$ to be determined by the remaining KKT conditions. From the Lagrangian

$$\mathcal{L} = \left(x_1 - \tfrac{1}{2}\right)^2 + x_2^2 + \lambda_1\Big(\cos(x_1) + x_2\Big) + \lambda_2\left(\tfrac{1}{2}(x_1 - \tfrac{1}{4})^2 - x_2 - 1\tfrac{1}{4}\right)$$

we can write the stationarity condition $\nabla_{\mathbf{x}}\mathcal{L} = \mathbf{0}$ as follows.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_1} &= 2\left(x_1 - \tfrac{1}{2}\right) - \lambda_1 \sin(x_1) + \lambda_2\left(x_1 - \tfrac{1}{4}\right) = 0 \\ \frac{\partial \mathcal{L}}{\partial x_2} &= 2x_2 + \lambda_1 - \lambda_2 = 0 \end{aligned}$$

Substituting $\bar{x}_1 = 1$, $\bar{x}_2 = -\frac{1}{2}$, and $\bar{\lambda}_2 = 0$, these equations reduce to

$$\begin{aligned} 1 - 0.84147\lambda_1 &= 0 \\ -1 + \lambda_1 &= 0. \end{aligned}$$

There are fewer active constraints than there are variables (as is typical) so λ_1 is overdetermined by a system of equations that is slightly inconsistent. How shall we pick a value that comes as close as possible to satisfying the stationarity conditions?

In §1.5.2 we minimized a sum of absolute values; here we can use the same approach to minimize the sum of the absolute row deviations in the equations above. In this optimization problem I have included the KKT nonnegativity constraint on λ_1.

$$\begin{array}{rrcl} \underset{\lambda_1}{\text{minimize}} & z & = & \left|1 - 0.84147\lambda_1\right| + \left|-1 + \lambda_1\right| \\ \text{subject to} & \lambda_1 & \geq & 0 \end{array}$$

Recall that we can recast this as a linear program. Any number y can be written as $y = p - q$ where p and q are nonnegative numbers, one or both of which are zero; then $|y| = p + q$. Using this idea we can rewrite our optimization as the linear program at the top of the next page.

$$\begin{array}{lrcl}
\underset{\lambda_1,\mathbf{d}}{\text{minimize}} & z & = & (d_1^+ + d_1^-) + (d_2^+ + d_2^-) \\
\text{subject to} & d_1^+ - d_1^- & = & 1 - 0.84147\lambda_1 \\
 & d_2^+ - d_2^- & = & -1 + \lambda_1 \\
 & d_1^+, d_1^-, d_2^+, d_2^-, \lambda_1 & \geq & 0
\end{array}$$

Putting this linear program in standard form, we get this initial tableau.

	d_1^+	d_1^-	d_2^+	d_2^-	λ_1
0	1	1	1	1	0
1	1	−1	0	0	0.84147
1	0	0	1	−1	1

I used the `pivot` program to find $\bar{\lambda}_1 = 1$. This yields the **residual** $z = -0.15853$, which is a measure of the amount by which the stationarity equations are inconsistent.

We can generalize this approach to work for any standard-form nonlinear program. The KKT stationarity condition requires that

$$\frac{\partial f_0}{\partial x_j} + \sum_{i \in \mathbb{I}} \lambda_i \frac{\partial f_i}{\partial x_j} = 0, \quad j = 1 \ldots n$$

where as usual $\mathbb{I}$ is the indices of the active constraints. The deviation for row j in this set of equations is the quantity on the left hand side, which we can represent as $d_j^+ - d_j^-$. Then the absolute row deviation is $d_j^+ + d_j^-$ and our linear program becomes

$$\begin{array}{lrcl}
\underset{\mathbf{d}^+,\mathbf{d}^-,\boldsymbol{\lambda}}{\text{minimize}} & z & = & \displaystyle\sum_{j=1}^{n} (d_j^+ + d_j^-) \\
\text{subject to} & d_j^+ - d_j^- - \displaystyle\sum_{i \in \mathbb{I}} \lambda_i \frac{\partial f_i}{\partial x_j} & = & \displaystyle\frac{\partial f_0}{\partial x_j} \quad j = 1 \ldots n \\
 & \mathbf{d}^+, \mathbf{d}^-, \boldsymbol{\lambda} & \geq & \mathbf{0}
\end{array}$$

with the tableau

$$\mathtt{T} = \begin{array}{|c|ccccccccc|}
\hline
 & d_1^+ & \cdots & d_n^+ & d_1^- & \cdots & d_n^- & \lambda_1 & \cdots & \lambda_{|\mathbb{I}|} \\
\hline
0 & 1 & \cdots & 1 & 1 & \cdots & 1 & 0 & \cdots & 0 \\
\hline
\partial f_0/\partial x_1 & 1 & \cdots & 0 & -1 & \cdots & 0 & -\partial f_1/\partial x_1 & \cdots & -\partial f_{|\mathbb{I}|}/\partial x_1 \\
\vdots & 0 & \ddots & 0 & 0 & \ddots & 0 & \vdots & \cdots & \vdots \\
\partial f_0/\partial x_n & 0 & \cdots & 1 & 0 & \cdots & -1 & -\partial f_1/\partial x_n & \cdots & -\partial f_{|\mathbb{I}|}/\partial x_n \\
\hline
\end{array}$$

where each derivative is evaluated at the approximate minimizing point $\bar{\mathbf{x}}$. To construct this tableau for an arbitrary NLP and $\bar{\mathbf{x}}$ and then solve the linear program, I wrote the `mults.m` routine listed at the top of the next page.


```
 1 function [lambda,z]=mults(iact,x,grd)
 2 % estimate KKT multipliers
 3 % by minimizing the sum of absolute row deviations
 4 % in the stationarity condition of the NLP
 5
 6   n=size(x,1);                    % number of variables in NLP
 7   mact=size(iact,2);              % number of active constraints
 8   T=zeros(1+n,1+2*n+mact);        % the LP tableau is this big
 9
10   g0=grd(x,0);                    % the NLP objective gradient
11   T(:,1)=[0;g0];                  % is the LP constant column
12
13   for j=1:n                       % each LP d+ variable column
14       T(:,1+j)=[1;zeros(n,1)];    % has cost coefficient 1
15       T(1+j,1+j)=1;               % and +1 in constraint row j
16   end
17
18   for j=1:n                       % each LP d- variable column
19       T(:,1+n+j)=[1;zeros(n,1)]; % has cost coefficient 1
20       T(1+j,1+n+j)=-1;            % and -1 in constraint row j
21   end
22
23   for i=1:mact                    % each LP lambda variable column
24       gi=grd(x,iact(i));          % has zero cost and
25       T(:,1+2*n+i)=[0;-gi];       % negative constraint gradient
26   end
27
28   [dpdmla,rc,Tnew]=simplex(T,n,2*n+mact); % solve the LP
29   lambda=dpdmla(2*n+1:2*n+mact);          % return the multipliers
30   z=Tnew(1,1);                            % and the residual
31
32 end
```

The inputs 1 to `mults.m` are `iact`, a list of the indices of the active constraints; `x`, the point to be tested; and `grd`, a pointer to a routine that returns the gradient of a given function. The tableau `T` 8 is constructed one column at a time working left to right, and then 28 the `simplex` routine of §4.1 is used to solve the LP. The optimal KKT multipliers 29 and objective value 30 are extracted from the solution for return.

In the `nset` example, `iact=[1]` because only $f_1(\mathbf{x})$ is tight at $\bar{\mathbf{x}}$. Here `nsetg(x,i)` returns $\nabla f_i(\mathbf{x})$ in the standard way that was described in §15.5. When I used `mults.m` to compute λ_1, it produced this output.

```
octave:1> xbar=[1;-0.5];
octave:2> [lambda1,z]=mults([1],xbar,@nsetg)
lambda1 =  1
z = -0.15853
octave:3> format long
octave:4> xhat=[0.967281605376012;-0.567539804600159];
octave:5> [lambda1,z]=mults([1],xhat,@nsetg)
lambda1 =  1.13507960920032
z = -1.09691717070893e-15
```

For our approximate minimizing point $\bar{\mathbf{x}} = [1, -\frac{1}{2}]^\top$ we get the same results as before, but using a more precise estimate $\hat{\mathbf{x}}$ of the optimal point yields a different multiplier value and a much smaller residual. A sensitive way to assess the accuracy of a numerical solution to

a nonlinear program is by using `mults.m` or a program like it to compute the corresponding $\boldsymbol{\lambda}$, and observing the size of the residual. If a proposed solution really is a KKT point, the equations of the stationarity condition should be very nearly consistent.

The routine has no trouble finding correct multipliers for the `cq2` problem (even though its active constraint gradients are linearly dependent) but it is no more successful at finding multipliers for the `cq1` problem than we were.

```
octave:6> xstar=[1;0];
octave:7> iact=[1,2];
octave:8> [lambda,z]=mults(iact,xstar,@cq2g)
lambda =

   2   0

z = 0
octave:9> [lambda,z]=mults(iact,xstar,@cq1g)
lambda =

   0   0

z = -1
octave:10> quit
```

The linear program in `mults.m` makes sense only if each equation of the stationarity conditions can be satisfied for *some* vector `lambda`, but in §16.7 we observed for `cq1` that

$$\nabla f_0(\mathbf{x}^\star) = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \qquad \nabla f_1(\mathbf{x}^\star) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \nabla f_2(\mathbf{x}^\star) = \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

The first component of $\nabla f_0(\mathbf{x}^\star)$ is nonzero while the first component of both $\nabla f_1(\mathbf{x}^\star)$ and $\nabla f_2(\mathbf{x}^\star)$ is zero, so there is *no* value of $\boldsymbol{\lambda}$ for which $\partial f_0/\partial x_1 = \lambda_1 \partial f_1/\partial x_1 + \lambda_2 \partial f_2/\partial x_1$. These constraint gradients are said not to **cover** the objective gradient, and that is necessary for `mults.m` to work. More sophisticated implementations of the algorithm used in `mults.m` begin by checking whether this coverage condition is satisfied. It is less severe than requiring the constraint gradients to be linearly independent, but it is sure to be met only if $\mathbb{T} = \mathbb{F}$.

16.11 Exercises

16.11.1 [E] How do we deal with slack constraints in solving a nonlinear program by the Lagrange method? How do we discover which constraints are slack when we solve a nonlinear program by the KKT method?

16.11.2 [E] What does the KKT *orthogonality condition* require? Why does it have that name? What purpose does it serve in the KKT method for solving inequality-constrained nonlinear programs?

16.11.3 [E] The discussion in §16.2 makes use of the idea that a vector can be *between* two other vectors. What do I mean by that? If three vectors lie in a plane, isn't each between the other two? Explain.

16.11.4 [E] Show that in the `arch4` problem of §16.2, (a) $\mathbf{x}^\star = [\frac{1}{2}, \frac{7}{4}]^\top$; (b) $\boldsymbol{\lambda}^\star = [\frac{5}{22}, \frac{14}{11}]^\top$.

16.11.5 [E] What is the difference between a *nonnegative linear combination* of vectors and a *convex combination*? Illustrate your answer with an example.

16.11.6 [E] Give a geometrical argument to explain why, for the `arch4` problem, $-\nabla f_0(\mathbf{x}^\star)$ must fall between $\nabla f_1(\mathbf{x}^\star)$ and $\nabla f_2(\mathbf{x}^\star)$. What is sufficient to ensure that for a nonlinear program in standard form $-\nabla f_0(\mathbf{x}^\star)$ can be written as a nonnegative linear combination of the constraint gradients at $\mathbf{x}^\star$?

16.11.7 [E] What does the KKT *nonnegativity condition* require? What purpose does it serve [3, p293] in the KKT method for solving inequality-constrained nonlinear programs?

16.11.8 [H] If either $\lambda_i = 0$ or $f_i(\mathbf{x}) = 0$ or both for $i = 1 \ldots m$, then $\boldsymbol{\lambda}^\top \mathbf{f} = 0$. What must be true in order for $\boldsymbol{\lambda}^\top \mathbf{f} = 0$ to ensure that either $\lambda_i = 0$ or $f_i(\mathbf{x}) = 0$ or both for $i = 1 \ldots m$?

16.11.9 [E] How do the KKT conditions differ from the Lagrange conditions? How do KKT multipliers differ from Lagrange multipliers?

16.11.10 [P] Use a computer algebra system such as Maple or Mathematica to solve the KKT conditions for the `moon` problem of §16.3, and confirm that it reports the same KKT points we found by hand.

16.11.11 [E] What conditions are necessary to ensure that for a nonlinear program in standard form, if $\bar{\mathbf{x}}$ is a local minimizing point then there is a vector $\bar{\boldsymbol{\lambda}}$ such that $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ satisfies the KKT conditions?

16.11.12 [E] What conditions are sufficient to ensure that for a nonlinear program in standard form, if $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ satisfies the KKT conditions then $\bar{\mathbf{x}}$ is a global minimizing point?

16.11.13 [E] The hypotheses of the two KKT theorems are referred to respectively as the KKT *necessary conditions* and the KKT *sufficient conditions*. (a) From memory, write down the necessary conditions. (b) From memory, write down the sufficient conditions.

16.11.14 [E] If a nonlinear program in our standard form has m inequality constraints, how many possible combinations of active and inactive constraints are there?

16.11.15 [E] How can you classify KKT points to identify the local minima among them?

16.11.16 [E] Under what circumstances does a global minimizing point for a standard-form nonlinear program satisfy the KKT conditions?

16.11.17 [H] The inequalities $x_1 + x_2 \geq 4$ and $2x_1 + x_2 \geq 5$ define a convex set $\mathbb{S} \subset \mathbb{R}^2$ [74, Exercise 6-7]. (a) Formulate a nonlinear program whose solution can be used to find the minimum distance from the origin to $\mathbb{S}$. (b) Solve the nonlinear program graphically.

(c) Use the KKT method to solve the nonlinear program analytically, and confirm that you get the solution you found graphically.

16.11.18 [H] Consider the following nonlinear program, in which a and b are constant parameters.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1-a)^2+(x_2-b)^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & (x_1-12)^2+(x_2-12)^2-7^2 \le 0 \\ & f_2(\mathbf{x}) & = & x_1+x_2-20 \le 0 \end{array}$$

Use the KKT method to find $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ when (a) $a = 11$ and $b = 14$; (b) $a = 20$ and $b = 8$. (c) Check your answers by solving the problems graphically.

16.11.19 [H] Use the KKT method to solve this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & -x_1-2x_2+x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1+x_2-1 \le 0 \\ & f_2(\mathbf{x}) & = & x_1-1 \le 0 \\ & f_3(\mathbf{x}) & = & -x_2 \le 0 \end{array}$$

16.11.20 [H] Use the KKT method to solve this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1+2x_2-x_2^3 \\ \text{subject to} & f_1(\mathbf{x}) & = & 2x_1+x_2-1 \le 0 \\ & f_2(\mathbf{x}) & = & -x_1 \le 0 \\ & f_3(\mathbf{x}) & = & -x_2 \le 0 \end{array}$$

Check your answer by solving the problem graphically.

16.11.21 [H] Consider the following nonlinear program.

$$\begin{array}{rl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}} & -x_1^2-4x_2^2+4x_1x_2+x_1-12x_2 \\ \text{subject to} & x_1+x_2 \ge 4 \end{array}$$

Show that $\mathbf{x}^\star = [\frac{61}{18}, \frac{11}{18}]^\top$ is the only solution to the KKT conditions, and find $\lambda^\star$.

16.11.22 [H] Find all of the KKT points for this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2-x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & -(x_1-2)^2-x_2^2+4 \le 0 \end{array}$$

What is the optimal value?

16.11.23 [P] This nonlinear program [1, Exercise 4.10] has a KKT point at $\bar{\mathbf{x}} = [1, 2, 5]^\top$.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & 2x_1^2 + x_2^2 + 2x_3^2 + x_1x_3 - x_1x_2 + x_1 + 2x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 - x_3 \le 0 \\ & f_2(\mathbf{x}) & = & x_1 + x_2 + 2x_3 \le 16 \\ & f_3(\mathbf{x}) & = & -x_1 - x_2 \le -3 \end{array}$$

(a) Write the KKT conditions for this problem. (b) Confirm that $\bar{\mathbf{x}}$ satisfies the KKT conditions. (c) Determine whether or not $\bar{\mathbf{x}}$ is a minimizing point. (d) Use a symbolic algebra program such as Maple or Mathematica, or tedious hand calculations, to find *all* of the KKT points. Is $\bar{\mathbf{x}}$ optimal?

16.11.24 [E] A convex program has a convex feasible set, but there are two different ways in which a nonlinear program that has a convex feasible set might *not* be a convex program. What are they?

16.11.25 [H] An NLP that has a nonconvex constraint can have a feasible set that is either convex or nonconvex, as illustrated by the sets named $\mathbb{C}$ and $\mathbb{N}$ in §16.6. (a) Prove analytically that $\mathbb{C}$ is a convex set. (b) Prove analytically that $\mathbb{N}$ is *not* a convex set.

16.11.26 [H] For an inequality-constrained nonlinear program in standard form, the KKT conditions require that $\boldsymbol{\lambda}^\star \ge \mathbf{0}$. In §16.6 I claimed that the Lagrange conditions are a special case of the KKT conditions when the constraints are equalities. Yet when the Lagrange method is used to solve an equality-constrained nonlinear program, the Lagrange multipliers can turn out to have either sign. How is this possible?

16.11.27 [E] Can a convex program have an equality constraint? Explain.

16.11.28 [H] The problem `cq1` of §16.7 does not satisfy any constraint qualification. (a) Modify the problem by adding the constraint $x_1 \le 1$. Write down the KKT conditions for the modified problem, and show that they are satisfied at $\mathbf{x}^\star = [1, 0]^\top$. (b) Find $\mathbb{T}$ and $\mathbb{F}$ for the new problem. (c) Explain why $\mathbf{x}^\star$ satisfies the KKT conditions for this problem but not for `cq1`.

16.11.29 [E] Give precise definitions for (a) the *cone of tangents*; (b) the *cone of feasible directions*. Why are they important in the KKT theory of nonlinear programming? What does it mean if they are different from each other?

16.11.30 [H] The feasible set of a certain nonlinear program is defined by opposing inequalities as $\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^2 \mid f_1(\mathbf{x}) \le 0 \cap -f_1(\mathbf{x}) \le 0\}$. (a) If $f_1(\mathbf{x}) = x_1^2 + x_2^2 - 2$, show that $\mathbb{F} = \mathbb{T}$. (b) If [5, p318] $f_1(\mathbf{x}) = (x_1^2 + x_2^2 - 2)^2$, is it still true that $\mathbb{F} = \mathbb{T}$? Explain.

16.11.31 [E] If a nonlinear program has linearly dependent constraint gradients at its optimal point $\mathbf{x}^\star$, is it possible for $\mathbf{x}^\star$ to satisfy the KKT conditions? If so, what is necessary to *ensure* that $\mathbf{x}^\star$ satisfies the KKT conditions, even though the constraint gradients there are linearly dependent?

16.11.32 [H] Use the KKT method to solve this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 - \frac{1}{3}x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1 - x_2 \le 0 \\ & f_2(\mathbf{x}) & = & x_1 + x_2 \le 0 \\ & f_3(\mathbf{x}) & = & x_1 \le 0 \end{array}$$

Show that a constraint qualification is satisfied at the optimal point.

16.11.33 [H] Consider this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & -(x_1 - 1)^3 + x_2^2 \le 0 \end{array}$$

(a) Find $\mathbf{x}^\star$. (b) Show that no constraint qualification is satisfied at $\mathbf{x}^\star$.

16.11.34 [H] Show that if an NLP has equality constraints (whether they are stated as equalities or as pairs of opposing inequalities) the only way to get $\mathbb{T} = \mathbb{F}$ at a Lagrange point $\bar{\mathbf{x}}$ is for their gradients to be linearly independent there.

16.11.35 [H] Suppose the objective of cq1 is replaced by $f_0(\mathbf{x}) = (x_1-1)^2+(x_2-1)^2$. (a) Show that the optimal point $\mathbf{x}^\star = [1, 0]^\top$ now satisfies the KKT conditions. (b) Does changing the objective affect $\mathbb{T}$ or $\mathbb{F}$? If yes, show that $\mathbb{T} = \mathbb{F}$ now. If no, explain how $\mathbf{x}^\star$ can satisfy the KKT conditions even though $\mathbb{T} \neq \mathbb{F}$.

16.11.36 [E] List three special cases in which a constraint qualification is always satisfied.

16.11.37 [H] Prove the first KKT theorem of §16.4, assuming for the constraint qualification that $\mathbb{T} = \mathbb{F}$. First show that if $\bar{\mathbf{x}}$ is a local minimizing point then $\mathbb{T} \cap \{\mathbf{d} \mid \nabla f_0(\bar{\mathbf{x}})^\top \mathbf{d} < 0\} = \emptyset$. Then show that the system of inequalities

$$\begin{array}{rcl} \nabla f_0 \bar{\mathbf{x}}^\top \mathbf{d} & < & 0 \\ \nabla f_i \bar{\mathbf{x}}^\top \mathbf{d} & \le & 0, \ i \in \mathbb{I} \end{array}$$

has no solution $\mathbf{d}$. Finally, use Farkas' theorem (see Exercise 5.5.30) to establish the conclusion of the first KKT theorem.

16.11.38 [H] Use the KKT method to solve this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & -3x_1 + \frac{1}{2}x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 - 1 \le 0 \\ & f_2(\mathbf{x}) & = & -x_1 \le 0 \\ & f_3(\mathbf{x}) & = & -x_2 \le 0 \end{array}$$

Confirm that $\lambda_3 = 0$ even though the third constraint is active at the optimal point. Using a contour diagram, explain the significance of this zero KKT multiplier.

16.11.39 [E] Can a constraint whose optimal KKT multiplier is zero be removed from a nonlinear program without changing the optimal point? Explain.

16.11.40 [H] This nonlinear program [3, Exercise 9.33] has $\mathbf{x}^\star = [2, 2, 2]^\top$.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1 - 10)^2 + (x_2 - 10)^2 + (x_2 - 10)^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 + x_3^2 - 12 \le 0 \\ & f_2(\mathbf{x}) & = & -x_1 - x_2 - 2x_3 \le 0 \end{array}$$

Use the KKT conditions to show that one of the constraints is redundant. Why does removing it not change the optimal point?

16.11.41 [H] Consider this nonlinear program [3, Exercise 9.34].

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 + x_2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 - 1 \le 0 \\ & f_2(\mathbf{x}) & = & -x_1^2 - x_2^2 + 1 \le 0 \end{array}$$

Use the KKT conditions to show that the optimal KKT multipliers are not uniquely determined, and provide an interpretation of what that means.

16.11.42 [H] The problem of §16.8.1 has KKT conditions that are satisfied by $\mathbf{x}^\star = [-2, 0]^\top$ and $\boldsymbol{\lambda}^\star = [\frac{1}{4}, 0]^\top$. (a) Confirm that the KKT conditions are also satisfied by $\mathbf{x}^\star = [-2, 0]^\top$ and $\boldsymbol{\lambda}^\star = [0, 1]^\top$, in which it is the *first* constraint that appears to be redundant. (b) Can the first constraint be removed from the problem without changing its solution? Explain. (c) How is this problem related to the one discussed in §16.8.2?

16.11.43 [H] Find $\mathbb{F}$ and $\mathbb{T}$ for the example of §16.8.2, and show that they are unequal. Find $\mathbb{F}$ and $\mathbb{T}$ when the constraint $x_1 \ge 1$ is included, and show that they are equal.

16.11.44 [E] In §16.8.3, I describe several properties of a nonlinear program any of which will lead us to classify the problem as ill-posed. What are those properties? Are ill-posed problems always nonsense?

16.11.45 [H] In the `hearn` problem of §16.8.3, $f_0(\mathbf{x}^\star)$ is undefined because the first fraction is 0/0. However, the major axes of the contours plotted there appear to fall on the line $x_2 = 1 - 20x_1$, which terminates at $\mathbf{x}^\star = [0, 1]^\top$. (a) Substitute this expression for x_2 into the formula for $f_0(\mathbf{x})$ and solve the resulting 1-dimensional optimization problem. (b) Plot contours of the original objective for $f_0(\mathbf{x}) \in [6.6, 10.75]$. Where is the approximation accurate?

16.11.46 [H] In §16.9.0 we used the graph of the Lagrangian to derive the primal and dual of a nonlinear program. (a) Explain why $\mathbf{x}^\star$ is the solution to the "min sup" problem and $\boldsymbol{\lambda}^\star$ is the solution to the "max inf" problem. (b) How are saddle points of a Lagrangian related to the KKT points of the nonlinear program?

16.11.47 [P] The example of §16.9.0 is a convex program with a constraint qualification, so the graph of its Lagrangian is sure to be shaped like a saddle. Modify the example to make it nonconvex, and plot $\mathcal{L}(x, \lambda)$ for the modified problem. Is the surface still shaped like a saddle?

16.11.48 [E] Assuming NLP is a nonlinear program in standard form, write down, from memory if you can, (a) the primal problem $\mathscr{P}$; (b) the Lagrangian dual problem $\mathscr{D}$.

16.11.49 [H] Write down all of the ways in which the NLP Duality Relations of §16.9.1 differ from the LP Duality Relations of §5.1. When is it possible to recover the optimal vector for a primal NLP from the solution of the NLP's dual?

16.11.50 [H] Does every nonlinear program have a Lagrangian dual? If not, what is required to ensure that it does? Does every nonlinear program have a Wolfe dual? If not, what is required to ensure that it does?

16.11.51 [E] Explain why the Wolfe dual is usually easier to find than the Lagrangian dual.

16.11.52 [H] If $\mathscr{P}$ is a convex program with continuously differentiable $f_i(\mathbf{x})$ then we can form its Wolfe dual $\mathscr{D}$. Is $\mathscr{D}$ necessarily a convex program? Justify your answer.

16.11.53 [E] How is the Wolfe dual of a linear program related to the LP dual we studied in §5?

16.11.54 [H] The `hearn` problem is discussed in §16.8.3. (a) Show that the following nonlinear program can be regarded as a dual of that problem.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & \frac{1}{2}y_1^2 + y_1 - y_2 - 2y_3 & \\ \text{subject to} & \frac{1}{2}y_2^2 + y_3 - 5 & \leq \ 0 \\ & \frac{1}{2}y_3^2 + y_2 - 4 & \leq \ 0 \end{array}$$

(b) Is this problem also ill-posed?

16.11.55 [E] Give two reasons why it might be advantageous to work with a nonlinear program's dual rather than its primal.

16.11.56 [H] To solve a nonlinear program's dual analytically by using the KKT method, it is necessary to introduce KKT multipliers. It would be natural to call these multipliers $\mathbf{x}$, but under what circumstances are their optimal values the same as the optimal values of the primal variables $\mathbf{x}$?

16.11.57 [E] When we solve a nonlinear program for $\mathbf{x}^\star$, why might we also care about $\boldsymbol{\lambda}^\star$?

16.11.58 [P] In the example of §16.10, I passed `@nsetg` as a parameter to `mults.m` so that it could compute gradients of the functions in the `nset` problem. Code the MATLAB routine `nsetg.m` in the standard way described in §15.5, and repeat the calculation using `mults.m` to prove that it works.

16.11.59 [P] Use the `mults.m` program to find $\boldsymbol{\lambda}^\star$ for the `cq3` problem of §16.7.

16.11.60 [E] A research paper describes a new nonlinear program and states its optimal point. Given $\mathbf{x}^\star$ you might be able to solve the KKT conditions analytically for $\boldsymbol{\lambda}^\star$, thus confirming that $\mathbf{x}^\star$ is at least a KKT point. How else might you check whether $\mathbf{x}^\star$ is a KKT point?

16.11.61 [P] The set named $\mathbb{N}$ in §16.6 is the feasible set of the following nonlinear program.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & -(x_1 - \frac{1}{2})^2 - x_2^2 \\ \text{subject to} & \mathbf{x} & \in & \mathbb{N} \end{array}$$

(a) Solve the problem graphically to estimate $\mathbf{x}^\star$. (b) Write down the KKT conditions and explain how they can be used to approximate $\boldsymbol{\lambda}^\star$. (c) Use the `mults.m` program of §16.10 to find the KKT multipliers corresponding to your estimate of $\mathbf{x}^\star$. How big is the residual? (d) Use the KKT conditions to compute $\mathbf{x}^\star$ precisely. If at some step you need to solve an equation numerically, remember the MATLAB `fzero` function discussed in §15.0. (e) Use the `mults.m` program to find the KKT multipliers corresponding to your more accurate estimate of $\mathbf{x}^\star$. How big is the residual now?

16.11.62 [E] What condition must be satisfied by a nonlinear program in order for it to be possible to find $\boldsymbol{\lambda}^\star$ from $\mathbf{x}^\star$ by using the algorithm implemented in `mults.m`?

17

Trust-Region Methods

The numerical algorithms for nonlinear optimization that we have studied so far all solve unconstrained problems. Such methods are important not only because some applications give rise to problems without constraints, but also because many algorithms for problems having constraints work by solving a sequence of unconstrained problems. Steepest descent, Newton and quasi-Newton methods, and conjugate-gradient methods all generate $\mathbf{x}^{k+1}$ from $\mathbf{x}^k$ by taking either a full step determined by a formula or an optimal step determined by a line search. **Trust-region methods** [5, §4] [4, §11.6] also solve unconstrained nonlinear programs, but in a fundamentally different way. The conceptual basis of the trust-region approach is more sophisticated than the simple ideas behind the descent methods, and its development requires the KKT theory that was introduced in §16. Trust-region methods do sometimes work better than descent methods, but they are also worth studying because their construction illustrates the artful orchestration of many important ideas you have learned about nonlinear programming.

17.1 Restricted-Steplength Algorithms

In §13 we developed `ntfs.m` to implement modified Newton descent, and found that it achieves superlinear convergence in solving even the nonconvex `rb` problem. It can also solve this problem, which I will call `h35` (see §28.7.18), provided we start near $\mathbf{x}^\star = [3, \frac{1}{2}]^\top$.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & v_1^2 + v_2^2 + v_3^2 \\ \text{where} & v_t & = & c_t - x_1(1 - x_2^t), \quad t = 1, 2, 3 \\ & c_1 & = & 1.5 \\ & c_2 & = & 2.25 \\ & c_3 & = & 2.625 \end{array}$$

```
octave:1> xzero=[2.5;0.3];
octave:2> epz=1e-6;
octave:3> gama=1;
octave:4> [xnewt,kp,nm,rc]=ntfs(xzero,20,epz,@h35g,@h35h,gama)
xnewt =

   3.00000
   0.50000

kp =  6
nm = 0
rc = 0
```

Only 5 iterations were used (`kp=6`) and each found the Hessian of f_0 positive definite (`nm=0`).

The routines `h35g.m` and `h35h.m` that are passed to `ntfs` are listed below; `h35.m` is used later. In all three routines, `t` is an index on the terms in the objective.

```
% compute a gradient of h35
function g=h35g(x)
  c=[1.5;2.25;2.625];
  g=[0;0];
  for t=1:3
      v=c(t)-x(1)*(1-x(2)^t);
      dvdx1=-(1-x(2)^t);
      dvdx2=t*x(1)*x(2)^(t-1);
      g=g+[2*v*dvdx1;2*v*dvdx2];
  end
end

% compute an objective value of h35
function f=h35(x)
  c=[1.5;2.25;2.625];
  f=0;
  for t=1:3
      v=c(t)-x(1)*(1-x(2)^t);
      f=f+v^2;
  end
end
```

```
% compute a Hessian of h35
function h=h35h(x)
  c=[1.5;2.25;2.625];
  h=[0,0;
     0,0];
  for t=1:3
      v=c(t)-x(1)*(1-x(2)^t);
      dvdx1=-(1-x(2)^t);
      dvdx2=t*x(1)*x(2)^(t-1);
      dvdx1dx1=0;
      dvdx1dx2=t*x(2)^(t-1);
      dvdx2dx2=(t-1)*t*x(1)*x(2)^(t-2);
      dvdx2dx1=t*x(2)^(t-1);
      h=h+[2*v*dvdx1dx1+2*dvdx1^2, 2*v*dvdx1dx2+2*dvdx1*dvdx2;
           2*v*dvdx2dx1+2*dvdx2*dvdx1, 2*v*dvdx2dx2+2*dvdx2^2];
  end
end
```

Unfortunately, if we move the starting point just a little farther from $\mathbf{x}^\star$, `ntfs.m` diverges.

```
octave:5> xzero=[1;0.6];
octave:6> gama=0.5;
octave:7> [xnewt,kp,nm,rc]=ntfs(xzero,1,epz,@h35g,@h35h,gama)
xnewt =

   8.8686
  -1.5310

kp =  1
nm = 0
rc =  1
octave:8> [xnewt,kp,nm,rc]=ntfs(xzero,2,epz,@h35g,@h35h,gama)
xnewt =

    4.7604
   21.4216

kp =  2
nm =  6
rc =  1
octave:9> [xnewt,kp,nm,rc]=ntfs(xzero,3,epz,@h35g,@h35h,gama)
xnewt =

  -5.3543e+08
   2.0081e+08

kp =  3
nm =  33
rc =  1
```

The algorithm takes ever-longer steps, soon finding itself in territory where the Hessian of the objective is far from positive definite (as shown by the growth of `nm`). Trying five iterations

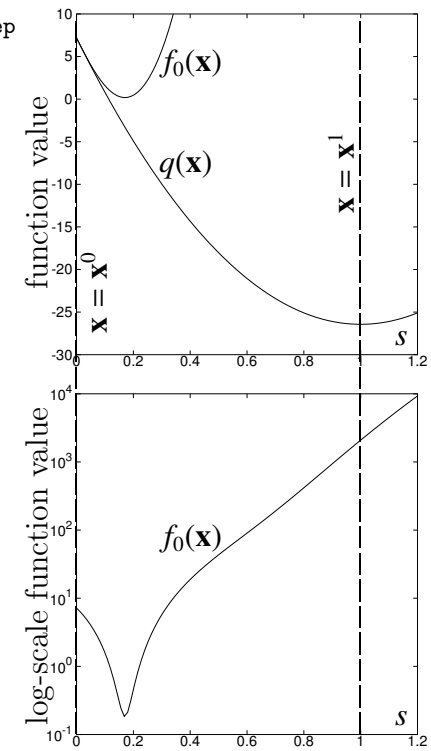
puts `ntfs.m` into an endless loop, as it fails (because of overflow in `h35h.m`) to find a factorable Hessian. Why does our faithful `ntfs.m` code now betray us with this lunatic behavior?

Recall from §13.1 that at each iteration Newton descent minimizes the quadratic model function

$$f_0(\mathbf{x}) \approx q(\mathbf{x}) = f_0(\mathbf{x}^k) + \nabla f_0(\mathbf{x}^k)^\top(\mathbf{x}-\mathbf{x}^k) + \tfrac{1}{2}(\mathbf{x}-\mathbf{x}^k)^\top\mathbf{H}(\mathbf{x}^k)(\mathbf{x}-\mathbf{x}^k).$$

This $q(\mathbf{x})$ matches $f_0(\mathbf{x})$ at $\mathbf{x}^k$ in value, gradient, and Hessian, but unless $f_0(\mathbf{x})$ is itself quadratic $q(\mathbf{x})$ departs from $f_0(\mathbf{x})$ as we move toward $\mathbf{x}^{k+1}$. To study this phenomenon in our example, I wrote the program below to plot both functions as $\mathbf{x}$ moves from $\mathbf{x}^0$ to $\mathbf{x}^1$.

```
% mismatch.m: study how q(x) departs from f(x) in taking first step
clear; clf; set(gca,'FontSize',30)

x0=[1;0.6];             % starting point
x1=[8.8686;-1.5310];    % first iterate produced by ntfs.m
d=x1-x0;                % full Newton step (nm was zero)
for t=1:101             % at each of 101 points
    s(t)=0.012*(t-1);   % along that direction
    x=x0+s(t)*d;        % find x
                        % evaluate the objective and model
    f(t)=h35(x);
    q(t)=h35(x0)+h35g(x0)'*(x-x0)+0.5*(x-x0)'*h35h(x0)*(x-x0);
end

figure(1); set(gca,'FontSize',30)
hold on
axis([0,1.2,-30,10],'square')
plot(s,f)               % plot objective on a linear scale
plot(s,q)               % plot quadratic on a linear scale
hold off
print -deps -solid mislin.eps

figure(2); set(gca,'FontSize',30)
hold on
axis([0,1.2],'square')
semilogy(s,f)           % plot objective on a log scale
hold off
print -deps -solid mislog.eps
```



The top graph shows that the $\mathbf{x}^1$ returned by `ntfs.m` is indeed the minimizing point of q in the Newton descent direction, but it is far beyond the minimizing point of f_0 in that direction. The model function and the objective match near $\mathbf{x}^0$, but at $\mathbf{x}^1$ they look completely different. The bottom picture uses a log scale to plot $f_0(\mathbf{x})$ and shows that at $\mathbf{x}^1$, contrary to the quadratic model, the objective is actually rising steeply.

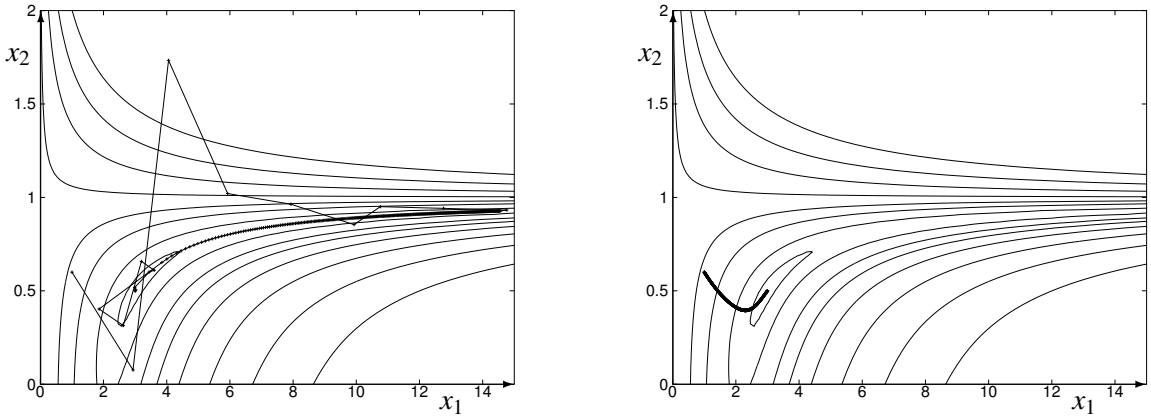
A simple way of avoiding this kind of blunder is to ensure $q(\mathbf{x})$ is a good approximation to $f_0(\mathbf{x})$ by prohibiting steps that are too big [59, §5]. To study this idea I wrote the program on the next page, which keeps each step taken by our modified Newton algorithm from being longer than r. We happen to know the optimal point of `h35` so it is convenient to use $\|\mathbf{x}^\star - \mathbf{x}^0\|$ as a natural unit of distance, and based on it I chose two values of r `16,18` to compare. For each r the program `14-49` solves the problem `21-34` one iteration at a time by `33` moving in the directions `30` suggested by `ntfs.m` but in steps `32` no longer than r.


```
% newth35.m: restrict Newton step length to solve h35
clear;clf

xzero=[1;0.6];                          % starting point
xstar=[3;0.5];                          % catalog optimal point

xl=[ 0.0;0.0];                          % lower bounds for picture
xh=[15.0;2.0];                          % upper bounds for picture
ng=100; vc=[0.1,1,4,8,16,32,64,128];    % set contouring parameters
[xc,yc,zc]=gridcntr(@h35,xl,xh,ng);     % get function values on grid

epz=1e-6;                               % convergence tolerance
gama=0.5;                               % weight for modified Newton
for tr=1:2                              % try two step-restrictions
    if(tr == 1)                         % the first experiment
       r=norm(xstar-xzero);             % allows big steps
    else                                % the second
       r=0.001*norm(xstar-xzero);       % requires tiny steps
    end                                 % finished setting r

    xk=zeros(1500); yk=zeros(1500);     % fix array sizes
    x=xzero;                            % starting point
    for k=1:1500;                       % do iterations
        xk(k)=x(1);                     % remember the point
        yk(k)=x(2);                     % for plotting later

        [xnewt,kp,nm,rc]=ntfs(x,1,epz,@h35g,@h35h,gama); % new point

        if(rc==0) break; end            % stop on zero gradient
        d=xnewt-x;                      % direction to move
        if(norm(d) < epz) break; end    % stop on short enough step
        s=min(r,norm(d));               % limit the steplength
        x=x+s*(d/norm(d));              % and move to the next xk
    end                                 % done with iterations
    k                                   % report iterations used

    figure(tr); set(gca,'FontSize',30) % separate the plots
    hold on                             % begin plot
    axis([xl(1),xh(1),xl(2),xh(2)]);    % set axes
    contour(xc,yc,zc,vc)                % draw contour lines
    plot(xk(1:k),yk(1:k),'+');          % plot convergence trajectory
    plot(xk(1:k),yk(1:k));              % plot connecting lines
    hold off                            % done with plot
    if(tr == 1)                         % if big steps
       print -deps -solid nth35a.eps       % call the picture this
    else                                % if tiny steps
       print -deps -solid nth35b.eps       % call the picture this
    end                                 % done printing the graph
end                                     % done with step-restrictions
```

The iterations of this **restricted steplength algorithm** are plotted over contours of the `h35` objective in the graphs on the next page.

The picture on the left shows the convergence trajectory, plotted as + signs connected by line segments, when each modified Newton step is restricted in length to $r = \|\mathbf{x}^\star - \mathbf{x}^0\|$. Taking a single step of that length in the direction $\mathbf{x}^\star - \mathbf{x}^0$ would solve the problem. Our algorithm takes a more roundabout path, but it does eventually find its way from $\mathbf{x}^0 = [1, 0.6]^\top$ to $\mathbf{x}^\star = [3, \frac{1}{2}]^\top$, which is a big improvement over the abject failure of `ntfs.m` when we let it

decide for itself how far to go. Making r bigger than this results in an even more chaotic path to $\mathbf{x}^\star$, until a value of r is reached above which the algorithm again fails to converge.



The picture on the right shows that restricting the steps to length $r = 0.001\|\mathbf{x}^\star - \mathbf{x}^0\|$ yields a more direct path to the optimal point. Making r even less than this does not result in a further decrease in the length of the convergence trajectory.

These experimental findings suggest we should use an r at or below the value that yields the shortest path (if we had some way of knowing ahead of time what that critical value is). However, the `newth35.m` program delivers another output $\boxed{35}$ and it reveals a big difference in the number of iterations required to reach $\mathbf{x}^\star$. The larger value of r lets us solve the problem in `k=335` iterations, while the smaller value of r requires `k=1015`. Each iteration takes CPU time, so if performance matters we should use the *biggest* r that still lets us solve the problem at all (it is also hard to imagine being able to figure out this critical value ahead of time).

17.2 An Adaptive Modified Newton Algorithm

Instead of permanently setting r at either extreme, it is better to continuously adjust it as the algorithm proceeds. That way it is possible to strike a balance between taking a few big steps, some of which are likely to increase the distance to $\mathbf{x}^\star$, and taking many tiny steps each more likely to decrease that distance.

Suppose that in using modified Newton descent the step we take from $\mathbf{x}^k$ is $\mathbf{d}^k$. To restrict it we can instead let $s_k = \min(r, \|\mathbf{d}^k\|)$ and take a step $\mathbf{p}^k = s_k[\,\mathbf{d}^k/\|\mathbf{d}^k\|\,]$ in the recommended descent direction but of length s_k. If the full step happens to be no longer than r, then $\mathbf{p}^k = \mathbf{d}^k$; otherwise $\mathbf{p}^k$ is a step of length r in the direction $\mathbf{d}^k$.

Modified Newton descent fails when, as in our example, the actual objective function is too different at $\mathbf{x}^{k+1}$ from the model function that matched it exactly at $\mathbf{x}^k$. The quadratic model predicts that the objective will go down by a certain amount as a result of taking the step $\mathbf{p}^k$, so one way to assess its fidelity is to compare that prediction with the actual

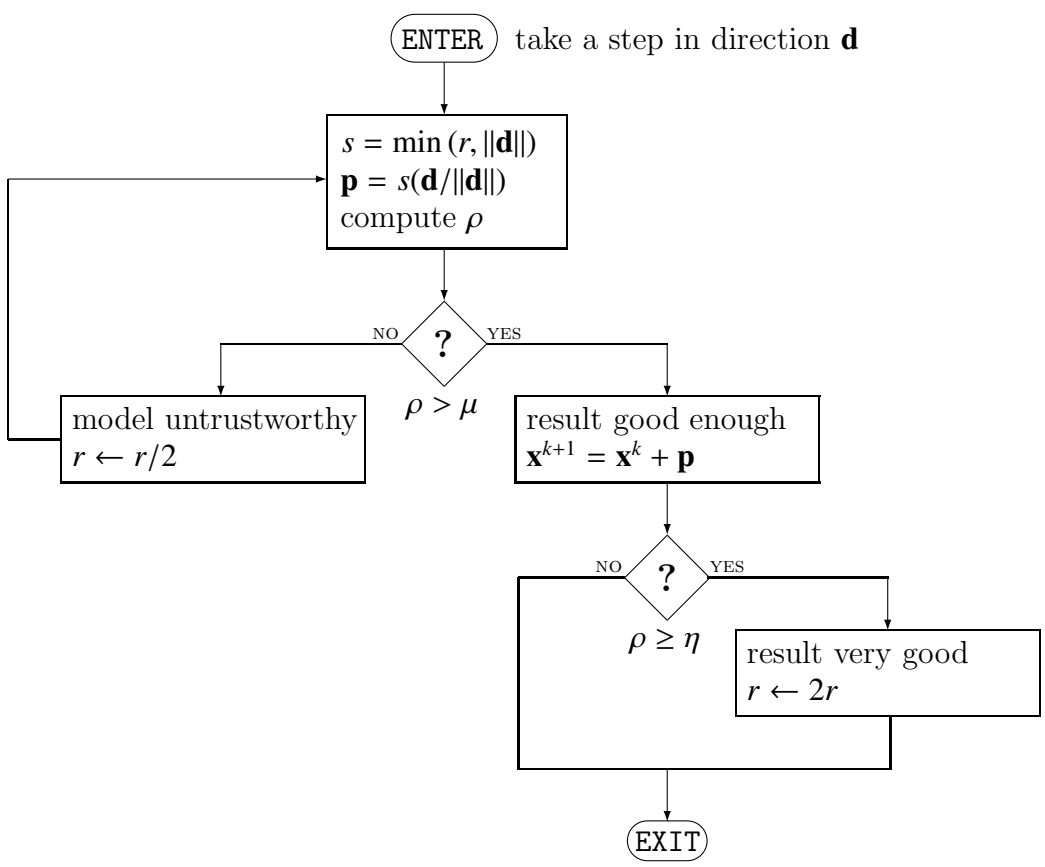
objective reduction we observe. Then we can allow a step $\mathbf{p}^k$ only if the actual reduction in the objective, $f_0(\mathbf{x}^k) - f_0(\mathbf{x}^k + \mathbf{p}^k)$, is not too different from the objective reduction predicted by the model, which is $f_0(\mathbf{x}^k) - q(\mathbf{x}^k + \mathbf{p}^k)$ where

$$q(\mathbf{x}^k + \mathbf{p}^k) = f_0(\mathbf{x}^k) + \nabla f_0(\mathbf{x}^k)^\top \mathbf{p}^k + \tfrac{1}{2}\mathbf{p}^{k\top}\mathbf{H}(\mathbf{x}^k)\mathbf{p}^k.$$

We can decide whether the quadratic model is trustworthy based on the value of the **objective reduction ratio**

$$\rho = \frac{\text{actual reduction}}{\text{predicted reduction}} = \frac{f_0(\mathbf{x}^k) - f_0(\mathbf{x}^k + \mathbf{p}^k)}{f_0(\mathbf{x}^k) - q(\mathbf{x}^k + \mathbf{p}^k)}.$$

If ρ is much different from 1, then the model is suspect. If the actual reduction is much *less* than predicted, so that ρ is less than or equal to μ (typically chosen to be $\frac{1}{4}$), then we must have stepped too far, to a place where $q(\mathbf{x})$ is no longer a good approximation of $f_0(\mathbf{x})$, and we should reduce r. If, on the other hand, the actual reduction is *greater* than predicted, then even though the model is wrong we should take the step! We are after all trying to minimize the function, and if fate provides us with a better point than expected we can tolerate the indignity of being shown that our model is wrong. If the actual reduction is still bigger, so that ρ is greater than or equal to η (typically chosen to be $\frac{3}{4}$) then it even makes sense to increase r. This policy is summarized in the flowchart below.



Accepting a trial steplength in a restricted-steplength algorithm only if it yields at least the expected objective decrease is somewhat analogous to enforcing the sufficient decrease Wolfe condition in a descent method that uses a line search.

The MATLAB routine `ntrs.m`, whose listing begins below, implements modified Newton descent but with steps limited in length to the r produced by the algorithm in the flowchart.

```
function [xstar,kp,nm,rc,r]=ntrs(xzero,rzero,kmax,epz,fcn,grd,hsn,gama)
% adaptive modified Newton algorithm

  n=size(xzero,1);                 % get number of variables
  xk=xzero;                        % set starting point
  r=rzero;                         % set starting steplength
  mu=0.25; eta=0.75;               % set r adjustment parameters
  nm=0;                            % no Hessian modifications yet
  for kp=1:kmax                    % allow kmax descent iterations
      g=grd(xk);                   % find uphill direction
      if(norm(g) <= epz)           % is xk stationary?
         xstar=xk;                   % yes; declare xk optimal
         rc=0;                       % flag convergence
         return                      % and return
      end                          % no; continue iterations
      H=hsn(xk);                   % get current Hessian matrix
      [U,pz]=chol(H);              % try to factor it

      if(pz~=0)                    % is H positive definite?
        if(gama >= 1 || gama < 0)  % no; is modification possible?
           xstar=xk;                 % no; gama value prevents that
           rc=2;                     % flag nonconvergence
           return                    % and return
        end                        % yes; modification possible
        tmax=1022;                 % limit modifications
        for t=1:tmax               % repeat until limit or success
          H=gama*H+(1-gama)*eye(n);  % average with identity
          nm=nm+1;                   % count the modification
          [U,pt]=chol(H);            % try again to factor
          if(pt==0) break; end       % positive definite now?
        end                        % no; continue modifications
        if(pt~=0)                  % was modification successful?
          xstar=xk;                  % no; factorization still fails
          rc=3;                      % flag nonconvergence
          return                     % and return
        end                        % yes; modification succeeded
      end                          % now Hd=U'Ud=-g

      y=U'\(-g);                   % solve U'y=-g for y
      dk=U\y;                      % solve Ud=y for d
      if(xk+dk==xk)                % is the Newton step too small?
         xstar=xk;                 % yes; further descent impossible
         rc=4;                     % flag nonconvergence
         return                    % and return
      end                         % no; continue iterations
```

This routine differs from `ntfs.m` in several ways. First, it requires [1] a pointer `fcn` to a routine that finds the value of the objective function at a given point, and it returns [1] the final step length `r`.

Second, it modifies the Hessian, if that is necessary, by using a process that cannot loop endlessly (compare lines 19-37 of this code with lines 15-24 of `ntfs.m` in §13.2). If the initial factorization 17 fails and 20 `gama` has a value that permits `H` to be modified, this routine allows only `tmax` 25-26 modifications. Here `gama` is interpreted as in `ntfs.m`: `gama=0` means that if `H` is not positive definite steepest descent should be used for this iteration, and `gama=1` means that if `H` is not positive definite the routine should resign with `rc=2`. I will have more to say in §17.5 about the choice of `tmax=1022`. As soon as successive averagings of `H` with the identity 27 have made `H` positive definite 29 the modification process is interrupted 30 and the factors of `H` are used as in `ntfs.m`. If `tmax` adjustments do not yield a Hessian that is positive definite 32 then 33-35 the routine sets `rc=3` and resigns. If `H` is successfully factored `U` is used 39-40 to find the descent direction `dk`, but if taking that Newton step would not change `xk` 41 the routine returns 42 the current point as `xstar` and 43 `rc=4`.

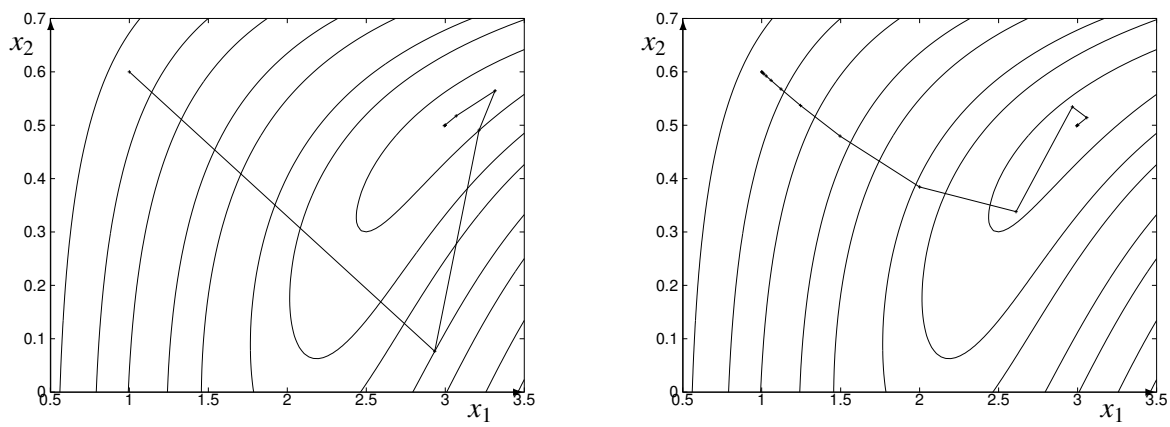
```
47      if(kp==1)                    % start with rzero only if positive
48         if(rzero <= 0) r=norm(dk); end  % else use full Newton step
49      end                          % done initializing r
50      tmax=52;                     % limit steplength adjustments
51      for t=1:tmax                 % repeat until limit or success
52          s=min(r,norm(dk));         % restrict steplength to r
53          p=s*(dk/norm(dk));         % find trial step
54          fxk=fcn(xk);                   % function value at xk
55          gxk=grd(xk);                   % gradient at xk
56          hxk=hsn(xk);                   % Hessian at xk
57          qxtry=fxk+gxk'*p+0.5*p'*hxk*p; % quadratic model prediction
58          xtry=xk+p;                     % find trial point
59          fxtry=fcn(xtry);               % actual function value
60          if(fxk==qxtry)                 % does the model go downhill?
61             rho=(mu+eta)/2;               % no; any decrease is enough
62          else                           % yes; continue adjustment
63             rho=(fxk-fxtry)/(fxk-qxtry);  % reduction ratio
64          end                            % done finding rho
65          if(rho > mu)                   % enough reduction?
66             xk=xtry;                    % yes; accept trial step
67             if(rho >= eta) r=2*r; end   % increase r if possible
68             break                       % and continue descent
69          else                         % no, stepped too far
70             r=r/2;                    % reduce steplength
71             if(r == 0) break; end     % if process fails give up
72          end                          % finished testing rho
73      end                              % finished adjusting r
74      if(rho <= mu)                % was r adjustment successful?
75         xstar=xtry;                 % no; return trial point
76         rc=5;                       % flag nonconvergence
77         return                      % and resign
78      end                          % yes; r adjustment succeed
79   end                             % continue descent
80
81   xstar=xk;                       % out of iterations
82   rc=1;                           % so no convergence yet
83 end
```

The third difference between this routine and `ntfs.m` is that here, instead of using the full modified-Newton step `dk`, the step `p` that we take is determined using the steplength

adjustment algorithm described above. If no steplength limit is provided on input, `r` is initialized [47-49] on the first descent iteration to the length of the full modified-Newton step. This yields [52] `s=norm(dk)` and [53] `p=dk` on the first iteration of the [51-73] loop. The function value [54], gradient [55], and Hessian [56] are found at `xk` to construct the model function $q(\mathbf{x}^k)$, which is used [57] to predict the objective value at the trial point `xk+p`. Then [58-59] `fcn` is used to find the actual function value at the trial point, and the ratio of reductions `rho` is calculated [60-64]. If the quadratic model function does not descend at all [60] (which would result in a division by zero at [63]) then any decrease in the objective is sufficient so `rho` is set [61] to a value bigger than `mu` but less than `eta`; otherwise we use [63] the formula given above.

If `rho` is high enough [65], the trial point is accepted [66], `r` might be increased [67], and the steplength adjustment process is interrupted [68]. The descent iterations then continue [79] using the new `xk`. If `rho` is too low [69] then [70] `r` is reduced and the steplength adjustment iterations continue. If `tmax` iterations of steplength adjustment are exhausted without achieving a suitable `rho` [74] then the routine [76] sets `rc=5` and resigns [77]. If `kmax` descent iterations are completed without convergence having been achieved [11], that loop terminates [79] and the routine returns with `rc=1` [82].

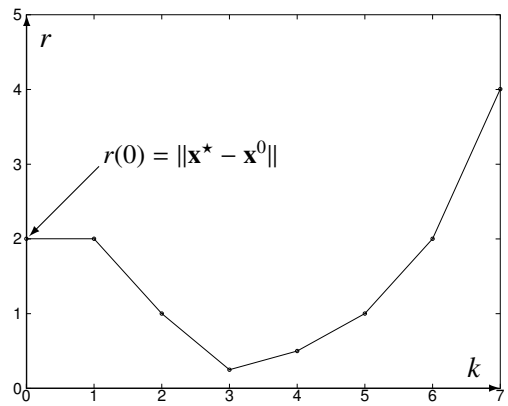
To test `ntrs.m` I wrote the program `ntrsh35.m` listed on the next page, which produces the graphs below.

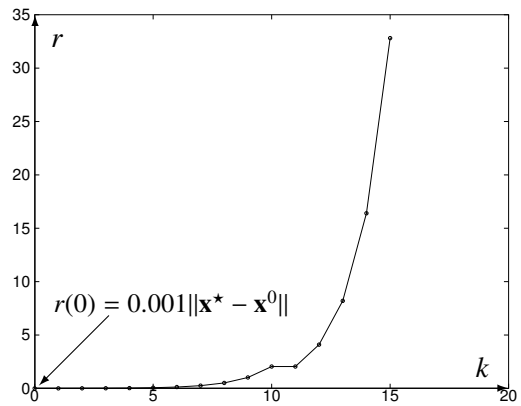


On the left the steplength adjustment process begins with $r = \|\mathbf{x}^\star - \mathbf{x}^0\|$, and `ntrs` converges in `k=7` iterations; on the right it begins with $r = 0.001\|\mathbf{x}^\star - \mathbf{x}^0\|$ and convergence takes `k=15` iterations. The performance of this algorithm is dramatically better than the one using fixed values of r that we studied in §17.1, so these pictures are scaled differently from those.

The graphs on the page after the listing show the steplength limit `r` being adjusted in each iteration `k` of the algorithm. On the left the large starting value $r = \|\mathbf{x}^\star - \mathbf{x}^0\|$ results in second and third steps that would be too long, so `r` is reduced. When the very small starting value $r = 0.001\|\mathbf{x}^\star - \mathbf{x}^0\|$ is used on the right, the quadratic model is initially accurate so `r` is left unchanged for several iterations. In both cases the model underestimates the objective reduction near $\mathbf{x}^\star$ so `r` is repeatedly doubled.


```
% ntrsh35.m: solve h35 using ntrs.m
clear;clf

xzero=[1;0.6];                          % starting point
xstar=[3;0.5];                          % catalog optimal point

xl=[ 0.5;0.0];                          % lower bounds for picture
xh=[ 3.5;0.7];                          % upper bounds for picture
ng=100; vc=[0.1,0.5,1,2,3,4.5,6,8];     % set contouring parameters
[xc,yc,zc]=gridcntr(@h35,xl,xh,ng);     % get function values on grid

epz=1e-6;                               % convergence tolerance
gama=0.5;                               % weight for modified Newton
for tr=1:2
    if(tr == 1)                         % the first experiment
       r=norm(xstar-xzero);             % allows big steps
    else                                % the second
       r=0.001*norm(xstar-xzero);       % requires tiny steps
    end                                 % finished setting r

    xk=zeros(1500); yk=zeros(1500);     % fix array sizes
    x=xzero;                            % starting point
    rzero=r;                            % starting steplength
    for k=1:20                          % do iterations
        xk(k)=x(1);                     % remember the point
        yk(k)=x(2);                     % for plotting later
        rk(k)=r;                        % remember the steplength
        kk(k)=k-1;                      % and iteration in which used

        [xstar,kp,nm,rc,r]=ntrs(x,rzero,1,epz,@h35,@h35g,@h35h,gama);

        if(rc==0) break; end            % stop on zero gradient
        x=xstar;                        % start next iteration
        rzero=r;                        % where this one left off
    end                                 % done with iterations
    k                                   % report iterations used

    figure(tr); set(gca,'FontSize',30)  % separate convergence plots
    hold on                             % begin plot
    axis([xl(1),xh(1),xl(2),xh(2)]);    % set axes
    contour(xc,yc,zc,vc)                % draw contour lines
    plot(xk(1:k),yk(1:k),'+');          % plot convergence trajectory
    plot(xk(1:k),yk(1:k));              % plot connecting lines
    hold off                            % done with plot
    if(tr == 1)                         % if big steps
       print -deps -solid rsh35a.eps       % call the picture this
    else                                % if tiny steps
       print -deps -solid rsh35b.eps       % call the picture this
    end                                 % done printing the graph
    figure(2+tr); set(gca,'FontSize',30) % separate steplength plots
    hold on                             % begin plot
    plot(kk,rk,'o')                     % plot r vs k
    plot(kk,rk)                         % plot connecting lines
    hold off                            % done with plot
    if(tr == 1)                         % if starting r
       print -deps -solid ntrsra.eps       % call the picture this
    else                                % if tiny starting r
       print -deps -solid ntrsrb.eps       % call the picture this
    end                                 % done printing the graph
end                                     % done with experiments
```



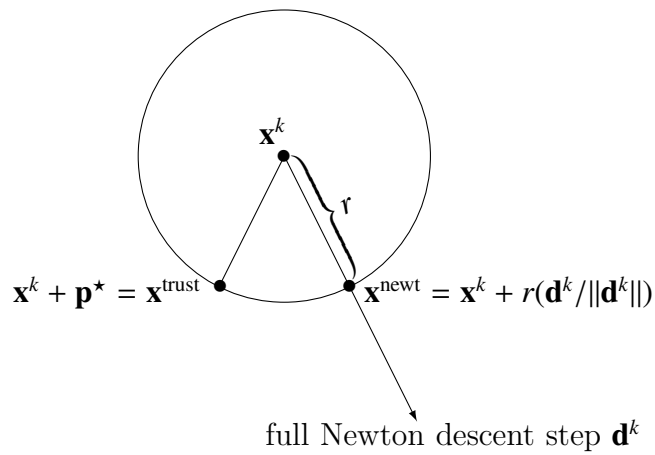
I also tried invoking `ntrs.m` with `r=0` to default its initial steplength to the length of the first full modified-Newton step. For `h35` that is $\|\mathbf{d}^0\| \approx 8$, compared to the initial steplength of $\|\mathbf{x}^\star - \mathbf{x}^0\| \approx 2$ that we used in `ntrsh35.m`. Now the routine solves the problem in 8 iterations.

```
octave:1> xzero=[1;0.6];
octave:2> rzero=0;
octave:3> epz=1e-6;
octave:4> gama=0.5;
octave:5> [xstar,kp,nm,rc,r]=ntrs(xzero,rzero,20,epz,@h35,@h35g,@h35h,gama)
xstar =

   3.00000
   0.50000

kp =  9
nm =  3
rc = 0
r =  16.304
octave:6> quit
```

17.3 Trust-Region Algorithms

Steepest descent, Newton descent, and conjugate gradient methods are each based on a model function. Adaptively adjusting the steplength helps to ensure that the model matches the objective throughout each step, so that the successive descent directions recommended by the model actually go downhill. In §17.2 we developed an adaptive-steplength modified-Newton algorithm that outperforms the full-step version of modified Newton on `h35`. Sometimes it is possible to further improve the performance of an adaptive-steplength algorithm.

If restricting the length of the step from $\mathbf{x}^k$ ensures that the model function is a good description of the objective along the descent direction, then the model function might be a good match to the objective over a whole **trust region** around $\mathbf{x}^k$. We will take this to be a ball of radius r in $\mathbb{R}^n$ (but see [5, p97] [4, p391]) so in two dimensions it is the disk pictured on the next page.



Ideally we would like $\mathbf{x}^{k+1}$ to minimize $f_0(\mathbf{x})$ over the trust region, but finding that point is as hard as the original optimization. If the model is accurate over the trust region, however, we can approximate that point by minimizing $q(\mathbf{x})$ over the trust region. This will almost certainly yield an $\mathbf{x}^{\text{trust}} = \mathbf{x}^{k+1}$ different from $\mathbf{x}^{\text{newt}} = \mathbf{x}^k + r(\mathbf{d}^k/\|\mathbf{d}^k\|)$. It is after all the *full* step $\mathbf{d}^k$ that minimizes $q(\mathbf{x})$ in the Newton descent direction, so if $r < \|\mathbf{d}^k\|$ then $\mathbf{x}^{\text{newt}}$ does *not* minimize $q(\mathbf{x})$. If the steplength is limited to $r < \|\mathbf{d}^k\|$ the minimizing point of $q(\mathbf{x})$ over the trust region is some other point in its boundary, and if $\|\mathbf{d}^k\| < r$ then the minimizing point is interior to the trust region. To find $\mathbf{x}^{k+1} = \mathbf{x}^k + \mathbf{p}^\star$ as the (boundary or interior) point in the trust region having the lowest value of $q(\mathbf{x})$, we must solve the **trust-region subproblem**:

$$\begin{array}{rrcl} \underset{\mathbf{p}}{\text{minimize}} & q(\mathbf{x}^k + \mathbf{p}) & = & f_0(\mathbf{x}^k) + \nabla f_0(\mathbf{x}^k)^\top \mathbf{p} + \frac{1}{2}\mathbf{p}^\top \mathbf{H}(\mathbf{x}^k)\mathbf{p} \\ \text{subject to} & \|\mathbf{p}\| & \leq & r. \end{array}$$

This inequality-constrained nonlinear program has differentiable functions and the linear independence constraint qualification, so $\mathbf{p}^\star$ will be among the points that satisfy its KKT conditions.

$$\begin{array}{rrcl} & \mathcal{L} & = & f_0(\mathbf{x}^k) + \nabla f_0(\mathbf{x}^k)^\top \mathbf{p} + \frac{1}{2}\mathbf{p}^\top \mathbf{H}(\mathbf{x}^k)\mathbf{p} + \lambda(\|\mathbf{p}\| - r) \\ \text{A} & \nabla_{\mathbf{p}}\mathcal{L} & = & \mathbf{0} + \nabla f_0(\mathbf{x}^k) + \mathbf{H}(\mathbf{x}^k)\mathbf{p} + \lambda \nabla_{\mathbf{p}}(\|\mathbf{p}\|) = 0 \\ \text{B} & \|\mathbf{p}\| & \leq & r \\ \text{C} & \lambda(\|\mathbf{p}\| - r) & = & 0 \\ \text{D} & \lambda & \geq & 0 \end{array}$$

The lettered lines are respectively the stationarity, feasibility, orthogonality, and nonnegativity conditions. Recalling from §10.6.3 or §28.1.3 that $\nabla_{\mathbf{p}}\|\mathbf{p}\| = \mathbf{p}/\|\mathbf{p}\|$,

$$\text{A} \Rightarrow \nabla f_0(\mathbf{x}^k) + \mathbf{H}(\mathbf{x}^k)\mathbf{p} + \frac{\lambda}{\|\mathbf{p}\|}\mathbf{p} = \mathbf{0}.$$

Letting $u = \lambda/\|\mathbf{p}\|$ we can rewrite this equation as

$$\mathbf{H}(\mathbf{x}^k)\mathbf{p} + u\mathbf{p} = -\nabla f_0(\mathbf{x}^k)$$

or

$$\boxed{(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I})\mathbf{p} = -\nabla f_0(\mathbf{x}^k).}$$

If $\mathbf{x}^k$ is not already stationary we take a step, so $\|\mathbf{p}\| \neq 0$ and

$$\text{C} \Rightarrow \frac{\lambda}{\|\mathbf{p}\|}(\|\mathbf{p}\| - r) = 0$$

or

$$\boxed{u(\|\mathbf{p}\| - r) = 0.}$$

If r is big enough so that $\mathbf{p}$ is inside the trust region then the constraint is slack, $\lambda = 0$, $u = 0$, and the first boxed equation says

$$\mathbf{H}(\mathbf{x}^k)\mathbf{p} = -\nabla f_0(\mathbf{x}^k)$$

so $\mathbf{p} = -\mathbf{H}^{-1}(\mathbf{x}^k)\nabla f_0(\mathbf{x}^k)$ is the full Newton step. If the constraint is tight the equation says that $(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I})\mathbf{p} = -\nabla f_0(\mathbf{x}^k)$, so

$$\begin{aligned} \mathbf{p} &= -\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1} \nabla f_0(\mathbf{x}^k) \\ \|\mathbf{p}\| &= r. \end{aligned}$$

Substituting the first of these formulas into the second we find that $u^\star$ is a root of

$$\varphi(u) = \left\| \left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1} \nabla f_0(\mathbf{x}^k) \right\| - r = 0.$$

In general this nonlinear algebraic equation has $2n$ roots, which we probably cannot find analytically. The one we want makes $u \geq 0$ as required by the KKT conditions, and makes the matrix $(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I})$ positive definite so that $\mathbf{p}$ is a descent direction.

17.3.1 Solving the Subproblem Exactly

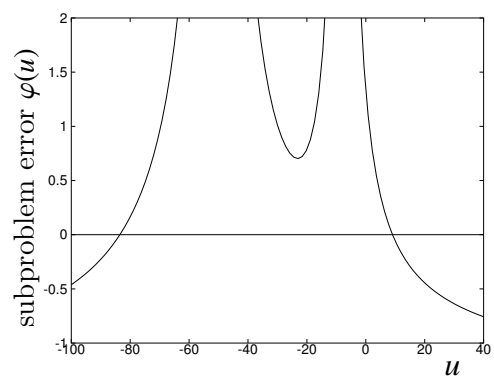
To find the best point $\mathbf{x}^{\text{trust}} = \mathbf{x}^k + \mathbf{p}^\star$ in the trust-region boundary when the full Newton step is longer than r, we can solve $\varphi(u) = 0$ for $u^\star$ numerically and then compute $\mathbf{p}^\star = -(\mathbf{H}(\mathbf{x}^k) + u^\star\mathbf{I})^{-1}\nabla f_0(\mathbf{x}^k)$. To see what is involved in doing that consider this problem, which I will call `bss1` (see §28.7.19).

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) \quad = \quad (x_1 - 2)^4 + (x_1 - 2x_2)^2$$

The program that begins below declares 6-7 the optimal and starting points for this problem. Then 10-16 it finds the first Newton descent step $\mathbf{d}^0$ and 19-21 moves in that direction a distance r chosen 20 to be less than $\|\mathbf{d}^0\|$. That ensures the optimal point of the trust-region subproblem will be in the boundary of the trust region. The routines `bss1.m`, `bss1g.m`, `bss1h.m`, and `truste.m` that are used in the program are listed to the right.

```
 1 % bss1trust.m: study the first step in solving bss1
 2 clear;clf
 3 global r=0 g=zeros(2,1) H=zeros(2,2);
 4 xl=[-2;0];
 5 xh=[ 2;4];
 6 xstar=[2;1];
 7 xzero=[0;3];
 8 diary
 9
10 % find the first Newton descent step
11 H=bss1h(xzero);
12 [U,tp]=chol(H);
13 tp
14 g=bss1g(xzero);
15 y=U'\(-g);
16 d0=U\y;
17 nd0=norm(d0)
18
19 % step in that direction a distance r
20 r=0.5*norm(xstar-xzero)
21 xnewt=xzero+r*(d0/norm(d0));
22 fnewt=bss1(xnewt)
23
24 % plot the trust-region error function
25 for t=1:101
26     u=-100+0.01*(t-1)*140;
27     xt(t)=u;
28     yt(t)=truste(u);
29 end
30 figure(1); set(gca,'FontSize',30)
31 hold on
32 axis([-100,40,-1,2])
33 plot(xt,yt)
34 plot([-100;40],[0;0])
35 hold off
36 print -deps -solid bss1phi.eps
37
38 % solve the trust region subproblem exactly
39 uzero=[0,20];
40 ustar=fzero(@truste,uzero);
41 Hstar=H+ustar*eye(2);
42 eigs=eig(Hstar)
43 [err,p]=truste(ustar);
44 xtrust=xzero+p;
45 ftrust=bss1(xtrust)
```

```
function f=bss1(x)
  f=(x(1)-2)^4+(x(1)-2*x(2))^2;
end

function g=bss1g(x)
  g=[4*(x(1)-2)^3+2*(x(1)-2*x(2));
     2*(x(1)-2*x(2))*(-2)];
end

function h=bss1h(x)
  h=[12*(x(1)-2)^2+2, -4;
     -4,              8];
end

function [err,p]=truste(u)
% find trust region subproblem error
  global r g H;
  p=-inv(H+u*eye(2))*g;
  err=norm(p)-r;
end
```




```
tp = 0
nd0 =  2.7487
r =  1.4142
fnewt =  16.024
eigs =

   16.700
   59.456

ftrust =  11.280
```

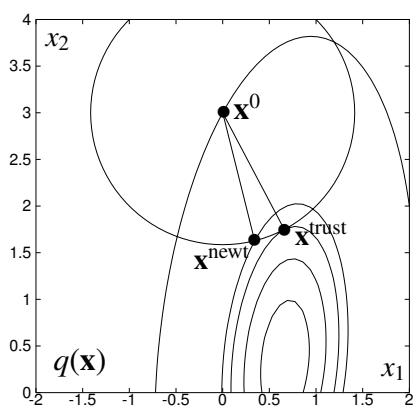
Then 24-36 the program plots φ as a function of u. Based on the graph, shown to the right, we can see that the solution we want is between $u = 0$ and $u = 20$, so using that search interval 39 the MATLAB function `fzero` is invoked 40 to find $u^\star > 0$ (see §15.0 for

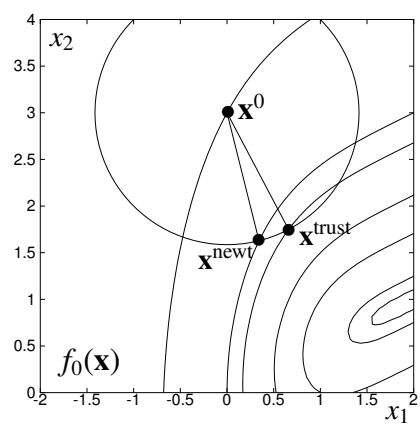
a description of `fzero`). The resulting $\mathbf{H} + u^\star \mathbf{I}$ is found [41] and the eigenvalues [42] of this matrix confirm that it is positive definite. The output below the graph also shows that `H` is positive definite (`tp=0`) and that `r` is indeed less than `nd0` $= \|\mathbf{d}^0\|$ [17].

```
47 % plot the trust region over contours of q(x)
48 figure(2); set(gca,'FontSize',30)
49 hold on
50 axis([xl(1),xh(1),xl(2),xh(2)],"equal")
51 [xt,yt]=circle(xzero(1),xzero(2),r,101);
52 plot(xt,yt)
53 [xc,yc,zc,zmin,zmax]=gridcntr(@qbss1,xl,xh,50);
54 vc=[7,10,qbss1(xtrust),qbss1(xnewt),qbss1(xzero)];
55 contour(xc,yc,zc,vc)
56 plot([xzero(1);xnewt(1)],[xzero(2);xnewt(2)])
57 plot([xzero(1);xtrust(1)],[xzero(2);xtrust(2)])
58 hold off
59 print -deps -solid bss1q.eps
60
61 % plot the trust region over contours of f(x)
62 figure(3); set(gca,'FontSize',30)
63 hold on
64 axis([xl(1),xh(1),xl(2),xh(2)],'equal')
65 [xt,yt]=circle(xzero(1),xzero(2),r,101);
66 plot(xt,yt)
67 [xc,yc,zc,zmin,zmax]=gridcntr(@bss1,xl,xh,50);
68 vc=[0.05,0.25,2,5,bss1(xtrust),bss1(xnewt),bss1(xzero)];
69 contour(xc,yc,zc,vc)
70 plot([xzero(1);xnewt(1)],[xzero(2);xnewt(2)])
71 plot([xzero(1);xtrust(1)],[xzero(2);xtrust(2)])
72 hold off
73 print -deps -solid bss1f.eps
```

```
% compute a q(x) value for bss1
function f=qbss1(x)
  xz=[0;3];
  fx=bss1(xz);
  gx=bss1g(xz);
  hx=bss1h(xz);
  p=x-xz;
  f=fx+gx'*p+0.5*p'*hx*p;
end
```

The Newton descent step bounded by steplength r is called `xnewt` in the program [21], while the point having lowest objective value in the boundary of the trust region, $\mathbf{x}^0 + \mathbf{p}^\star$ [44], is called `xtrust`. The final two stanzas of the program, listed on the left above, plot $\mathbf{x}^{\text{newt}}$ and $\mathbf{x}^{\text{trust}}$ over contours of [47-59] the quadratic model function listed on the right and [61-73] the objective function; in both pictures the circle is the trust region. On the left it is clear that `xtrust` is on a lower contour of the model function than is `xnewt`. In fact it is on the *lowest* contour of $q(\mathbf{x})$ that is in the boundary of the trust region, confirming that the $u^\star$ we found really does solve the trust-region subproblem.





The model function is a close approximation to $f_0(\mathbf{x})$ over this step, as can be seen by comparing the two contour diagrams, so it is not surprising that on the right `xtrust` is also on a lower contour of $f_0(\mathbf{x})$ than is `xnewt` (though not quite on the *lowest* contour, which would be tangent to the trust region). To be precise, the program's output shows that `xtrust` yields an objective value of `ftrust` ≈ 11.280 while `xnewt` yields a noticeably higher objective value of `fnewt` ≈ 16.024.

17.3.2 Solving the Subproblem Quickly

Letting $\mathbf{x}^{k+1}$ be the solution of the trust-region subproblem can speed convergence, but finding that point precisely is hard. The error function $\varphi(u)$ depends on $\mathbf{H}(\mathbf{x}^k)$ and $\nabla f_0(\mathbf{x}^k)$ in such a way that an algorithm to find the root we want ends up being complicated if it is going to work in every case. The CPU time required for these calculations might be more than we save by using $\mathbf{x}^{\text{trust}}$ instead of $\mathbf{x}^{\text{newt}}$. Thus, although the approach illustrated in §17.3.1 can be generalized [5, §4.3 and p170-171] I will not describe the myriad details here.

Instead, we will study a much simpler way of *approximating* the solution to the trust-region subproblem. To see where this idea comes from we need to consider a still simpler example, so suppose now that in minimizing some objective we start at $\mathbf{x}^0 = [0, 0]^\top$ and

$$q(\mathbf{x}) = (x_1 - 2)^2 + 10(x_2 + 1)^2$$

is the quadratic model function that matches $f_0(\mathbf{x})$ at that point. Because $q(\mathbf{x})$ agrees with $f_0(\mathbf{x})$ at $\mathbf{x}^0$ in gradient and Hessian as well as in value,

$$\begin{aligned} \nabla f_0(\mathbf{x}^0) = \nabla q(\mathbf{x}^0) = \begin{bmatrix} 2(x_1^0 - 2) \\ 20(x_2^0 + 1) \end{bmatrix} = \begin{bmatrix} 2(0-2) \\ 20(0+1) \end{bmatrix} &= \begin{bmatrix} -4 \\ 20 \end{bmatrix} \\ \mathbf{H}(\mathbf{x}^0) = \mathbf{H}_q(\mathbf{x}^0) &= \begin{bmatrix} 2 & 0 \\ 0 & 20 \end{bmatrix} \end{aligned}$$

and the full Newton descent step from $\mathbf{x}^0$ is

$$\mathbf{d}^{\mathrm{N}} = -\mathbf{H}^{-1}(\mathbf{x}^0)\nabla f_0(\mathbf{x}^0) = -\begin{bmatrix} 2 & 0 \\ 0 & 20 \end{bmatrix}^{-1} \begin{bmatrix} -4 \\ 20 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{20} \end{bmatrix} \begin{bmatrix} 4 \\ -20 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}.$$

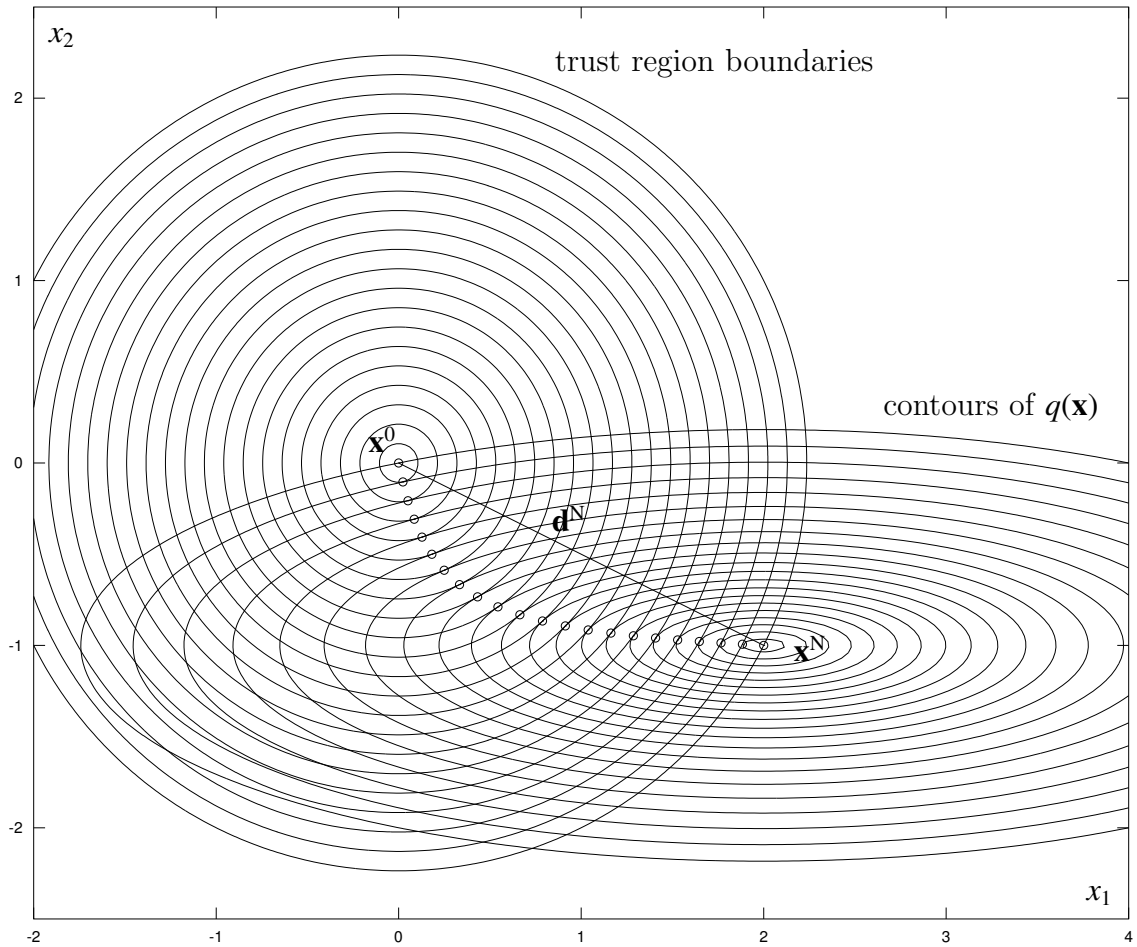
The picture on the next page shows contours of $q(\mathbf{x})$, its minimizing point $\mathbf{x}^{\mathrm{N}} = [2, -1]^\top$, and the full Newton step $\mathbf{d}^{\mathrm{N}}$ leading from $\mathbf{x}^0$ to $\mathbf{x}^{\mathrm{N}}$.

If we draw a trust region about $\mathbf{x}^0$ having radius

$$r \le \|\mathbf{d}^{\mathrm{N}}\| = \sqrt{2^2 + 1^2} = \sqrt{5} \approx 2.23$$

then the solution of the trust-region subproblem will be in the boundary of the trust region rather than its interior. For any such r we can find $\mathbf{p}^\star$ graphically as the point where the trust region boundary is tangent to a contour of $q(\mathbf{x})$; there is no way to make $q(\mathbf{x})$ lower than that contour value without leaving that trust region.

The picture below shows the graphical solution of the subproblem for our example at several values of r between 0 and $\|\mathbf{d}^{\mathrm{N}}\|$.



We can find these points ∘ exactly by reasoning as follows.

$$\begin{aligned}
\mathbf{H}(\mathbf{x}^k) + u\mathbf{I} &= \begin{bmatrix} 2+u & 0 \\ 0 & 20+u \end{bmatrix} \\
\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1} &= \begin{bmatrix} \frac{1}{2+u} & 0 \\ 0 & \frac{1}{20+u} \end{bmatrix} \\
\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1} \nabla f_0(\mathbf{x}^k) &= \begin{bmatrix} \frac{1}{2+u} & 0 \\ 0 & \frac{1}{20+u} \end{bmatrix} \begin{bmatrix} -4 \\ 20 \end{bmatrix} = \begin{bmatrix} \frac{-4}{2+u} \\ \frac{20}{20+u} \end{bmatrix} = -\mathbf{p} \\
\|\mathbf{p}\| &= \sqrt{\left(\frac{-4}{2+u}\right)^2 + \left(\frac{20}{20+u}\right)^2}
\end{aligned}$$

Thus for each value of r we can solve

$$\varphi(u) = \sqrt{\left(\frac{-4}{2+u}\right)^2 + \left(\frac{20}{20+u}\right)^2} - r = 0 \quad \text{or} \quad \left(\frac{-4}{2+u}\right)^2 + \left(\frac{20}{20+u}\right)^2 = r^2$$

for $u^\star(r)$ and then find

$$\mathbf{p}^\star(r) = \begin{bmatrix} \dfrac{4}{2+u^\star(r)} \\ \dfrac{-20}{20+u^\star(r)} \end{bmatrix}.$$

To carry out the calculation exactly for an arbitrary value of r we must use a numerical root finder just as we did for the example of §17.3.1, but for the extreme values of r we can find $\mathbf{p}^\star$ analytically.

When $r = \|\mathbf{p}\| = 0$ any trust-region subproblem is solved by the $u^\star$ that makes (see §17.3.1 and Exercise 17.6.27)

$$\left\|\left(\mathbf{H}(\mathbf{x}^k) + u^\star\mathbf{I}\right)^{-1}\right\| = 0.$$

Notice that

$$\lim_{u\to\infty}\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1} = \lim_{u\to\infty}(u\mathbf{I})^{-1} = \lim_{u\to\infty}\left(\frac{1}{u}\right)\mathbf{I} = [\mathbf{0}]\,.$$

The norm of the zero matrix is zero, so $u(r) = \infty$ solves the trust-region subproblem at $r = 0$, and $u^\star(r = 0) = \infty$. We can find the direction of the corresponding $\mathbf{p}^\star(0)$ by reasoning in a similar way.

$$\lim_{u\to\infty}\frac{\mathbf{p}}{\|\mathbf{p}\|} = \lim_{u\to\infty}\left[\frac{-\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1}\nabla f_0(\mathbf{x}^k)}{\left\|-\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1}\nabla f_0(\mathbf{x}^k)\right\|}\right] = \lim_{u\to\infty}\left[\frac{-\frac{1}{u}\nabla f_0(\mathbf{x}^k)}{\left\|-\frac{1}{u}\nabla f_0(\mathbf{x}^k)\right\|}\right] = \frac{-\nabla f_0(\mathbf{x}^k)}{\|\nabla f_0(\mathbf{x}^k)\|}$$

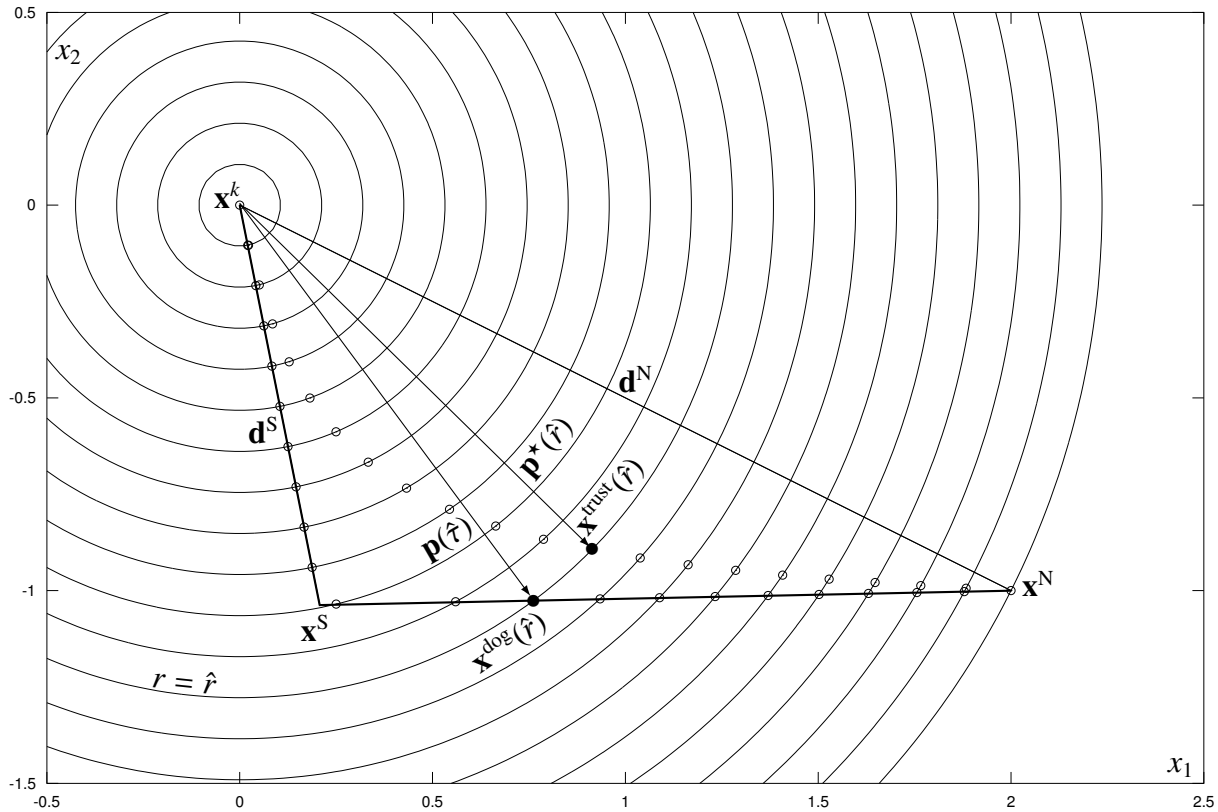
This shows that the limiting direction of $\mathbf{p}$ as $r \to 0$ is the direction of steepest descent.

The largest value of r for which the Newton descent step is in the boundary of the trust region is the length of the full Newton step, $\mathbf{d}^{\mathrm{N}} = -[\mathbf{H}(\mathbf{x}^k)]^{-1}\nabla f_0(\mathbf{x}^k)$, and this corresponds to $u = 0$ so $u^\star(r = \|\mathbf{d}^{\mathrm{N}}\|) = 0$. There we find

$$\lim_{u\to 0}\frac{\mathbf{p}}{\|\mathbf{p}\|} = \lim_{u\to 0}\left[\frac{-\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1}\nabla f_0(\mathbf{x}^k)}{\left\|-\left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1}\nabla f_0(\mathbf{x}^k)\right\|}\right] = \frac{-\left(\mathbf{H}(\mathbf{x}^k)\right)^{-1}\nabla f_0(\mathbf{x}^k)}{\left\|-\left(\mathbf{H}(\mathbf{x}^k)\right)^{-1}\nabla f_0(\mathbf{x}^k)\right\|} = \frac{\mathbf{d}^{\mathrm{N}}}{\|\mathbf{d}^{\mathrm{N}}\|}$$

so the limiting direction of $\mathbf{p}$ as $r \to \|\mathbf{d}^{\mathrm{N}}\|$ is the direction of Newton descent.

We have shown that when r is close to zero $\mathbf{p}^\star(r)$ is close to the direction of steepest descent, and when r is close to the length of the full Newton step $\mathbf{p}^\star(r)$ is close to the direction of Newton descent. This suggests approximating $\mathbf{p}^\star(r)$ by the piecewise linear **dogleg** drawn with thick lines in the picture on the following page.



The first edge of the dogleg is the full steepest-descent step $\mathbf{d}^S$ from $\mathbf{x}^k$, minimizing $q(\mathbf{x})$ in that direction at $\mathbf{x}^S$ (in our example we assumed we are taking the *first* step, so $\mathbf{x}^k = \mathbf{x}^0$). The second edge of the dogleg connects $\mathbf{x}^S$ to $\mathbf{x}^N$. The point $\mathbf{x}^N$ is the full Newton-descent step $\mathbf{d}^N$ from $\mathbf{x}^k$, and minimizes $q(\mathbf{x})$ in that direction. The point where the dogleg intersects each trust-region boundary is the approximation that we will use in place of the exact solution for that radius; here both are plotted as points. The points representing the exact solution and dogleg solution at $r = \hat{r}$ are solid, and they are labeled $\mathbf{x}^{\text{trust}}(\hat{r})$ and $\mathbf{x}^{\text{dog}}(\hat{r})$ respectively. There is nothing special about the triangle whose vertices are $\mathbf{x}^k$, $\mathbf{x}^S$, and $\mathbf{x}^N$; in general it is scalene and can be oriented at any angle to the coordinate axes.

The dogleg approximation is exact at both ends and not too bad in the middle. If we did not solve the trust-region subproblem but merely restricted the steplength taken by Newton descent to the trust-region radius, then for a given radius our next iterate would be the intersection of that trust region with the line from $\mathbf{x}^k$ to $\mathbf{x}^N$. For any trust-region radius less than the full Newton step, the dogleg approximation comes closer than that to the exact subproblem solution. The dogleg solution is always between $\mathbf{d}^S$ and $\mathbf{d}^N$.

The point where the dogleg intersects each trust-region boundary can be found algebraically, using a formula that depends on which part of the dogleg crosses the circle. Any point $\mathbf{x}^k + \mathbf{p}(\tau)$ on the dogleg can be described using the parameterization [5, p74] at the top of the next page.

$$\mathbf{p}(\tau) = \begin{cases} \tau\mathbf{d}^{\mathrm{S}} & 0 \le \tau \le 1 \quad \text{steepest-descent edge} \\ \mathbf{d}^{\mathrm{S}} + (\tau - 1)(\mathbf{d}^{\mathrm{N}} - \mathbf{d}^{\mathrm{S}}) & 1 \le \tau \le 2 \quad \text{connecting edge} \end{cases}$$

If $r \le \|\mathbf{d}^{\mathrm{S}}\|$ then the steepest-descent edge of the dogleg crosses the trust-region boundary at a point

$$\mathbf{p}(\tau) = \tau\mathbf{d}^{\mathrm{S}} \quad \text{where} \quad \tau = \frac{r}{\|\mathbf{d}^{\mathrm{S}}\|}.$$

This is just a restricted step in the steepest-descent direction.

If $r \ge \|\mathbf{d}^{\mathrm{S}}\|$ then it is the connecting edge of the dogleg that crosses the trust-region boundary, at a point $\mathbf{x}^k + \mathbf{p}(\tau)$ where the vector $\mathbf{p}(\tau)$ has length r. We can find the τ where that happens as follows.

$$\begin{aligned}
\left\| \mathbf{p}(\tau) \right\| = \left\| \mathbf{d}^{\mathrm{S}} + (\tau - 1)(\mathbf{d}^{\mathrm{N}} - \mathbf{d}^{\mathrm{S}}) \right\| &= r \\
\sum_{j=1}^{n} \left[d_j^{\mathrm{S}} + (\tau - 1)(d_j^{\mathrm{N}} - d_j^{\mathrm{S}}) \right]^2 &= r^2 \\
\sum \left[(d_j^{\mathrm{S}})^2 + 2(\tau - 1)(d_j^{\mathrm{S}})(d_j^{\mathrm{N}} - d_j^{\mathrm{S}}) + (\tau - 1)^2 (d_j^{\mathrm{N}} - d_j^{\mathrm{S}})^2 \right] &= r^2 \\
\sum (d^{\mathrm{S}})^2 + 2(\tau - 1) \sum (d_j^{\mathrm{S}})(d_j^{\mathrm{N}} - d_j^{\mathrm{S}}) + (\tau - 1)^2 \sum (d_j^{\mathrm{N}} - d_j^{\mathrm{S}})^2 - r^2 &= 0 \\
(\tau - 1)^2 \left[\sum (d_j^{\mathrm{N}} - d_j^{\mathrm{S}})^2 \right] + (\tau - 1) \left[\sum 2(d_j^{\mathrm{S}})(d_j^{N} - d_j^{S}) \right] + \left[\sum (d_j^{\mathrm{S}})^2 - r^2 \right] &= 0
\end{aligned}$$

This is a quadratic $a(\tau - 1)^2 + b(\tau - 1) + c = 0$ with coefficients

$$\begin{aligned}
a &= \sum (d_j^{\mathrm{N}} - d_j^{\mathrm{S}})^2 &&= (\mathbf{d}^{\mathrm{N}} - \mathbf{d}^{\mathrm{S}})^{\top} (\mathbf{d}^{\mathrm{N}} - \mathbf{d}^{\mathrm{S}}) \\
b &= \sum 2(d_j^{\mathrm{S}})(d_j^{N} - d_j^{S}) &&= 2(\mathbf{d}^{\mathrm{S}})^{\top} (\mathbf{d}^{\mathrm{N}} - \mathbf{d}^{\mathrm{S}}) \\
c &= \sum (d_j^{\mathrm{S}})^2 - r^2 &&= (\mathbf{d}^{\mathrm{S}})^{\top} \mathbf{d}^{\mathrm{S}} - r^2
\end{aligned}$$

so we can solve it analytically to find

$$\tau = 1 + \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

provided the discriminant is nonnegative. That is certainly true if $c \le 0$, which holds if $(\mathbf{d}^{\mathrm{S}})^{\top}\mathbf{d}^{\mathrm{S}} - r^2 \le 0$ or $r \ge \|\mathbf{d}^{\mathrm{S}}\|$ as we assumed. To ensure that $\tau \ge 1$ so we are on the connecting part of the dogleg, we should take the positive square root. Then we can find $\mathbf{p}(\tau) = \mathbf{d}^{\mathrm{S}} + (\tau - 1)(\mathbf{d}^{\mathrm{N}} - \mathbf{d}^{\mathrm{S}})$.

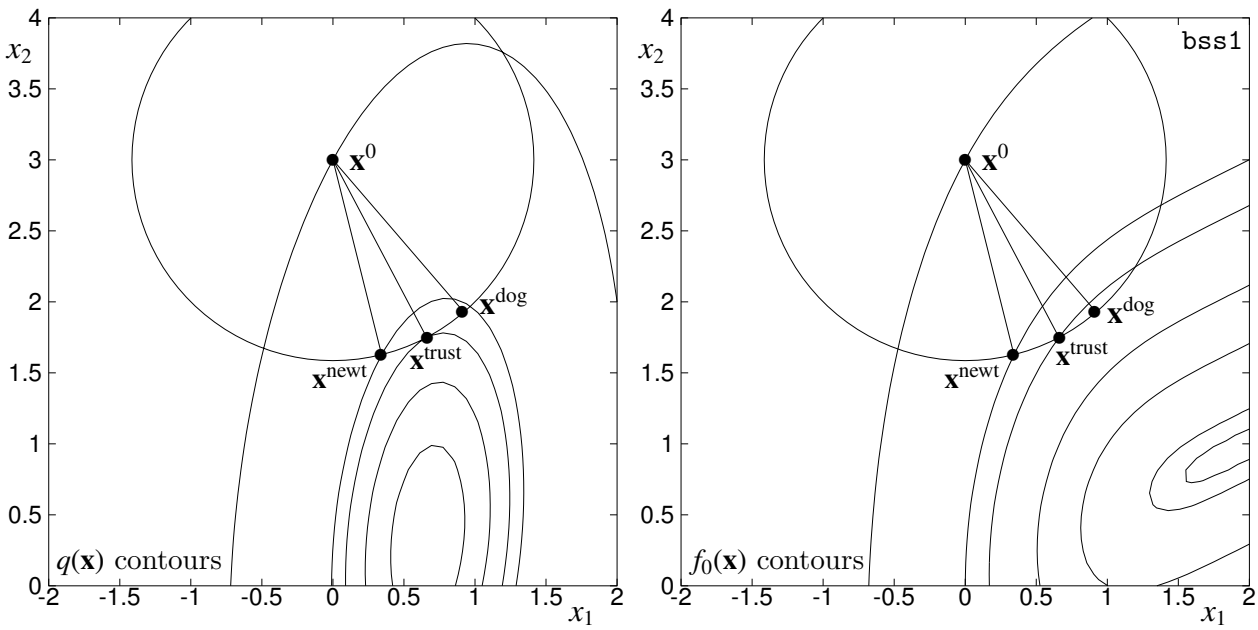
I implemented these calculations in the MATLAB routine `dogsub.m`, which is listed on the next page. Its inputs are the Hessian `H` and gradient `g` at $\mathbf{x}^k$ and the radius `r` of the trust region; it returns the dogleg step `p` and a return code `rcs` to inform the caller if the factorization of `H` fails.


```
 1 function [p,rcs]=dogsub(H,g,r)
 2 % solve the trust-region subproblem approximately
 3
 4   [U,pz]=chol(H);                % find dn, the full Newton step
 5   if(pz~=0)                      % H positive definite?
 6      rcs=1;                        % report failure
 7      return                        % and give up
 8   end                            % now Hd=U'Ud=-g
 9   y=U'\(-g);                     % solve U'y=-g for y
10   dn=U\y;                        % solve Ud=y for dn
11
12   if(norm(dn) <= r)              % inside trust region?
13      p=dn;                       % yes; take full Newton step
14   else                           % otherwise
15      ds=-((g'*g)/(g'*H*g))*g;    % find steepest descent full step
16      if(r <= norm(ds));          % on steepest-descent dogleg part
17         tau=r/norm(ds);          % find where on the dogleg
18         p=tau*ds;                % and the step to there
19      else;                       % on connecting dogleg part
20         a=(dn-ds)'*(dn-ds);      % find
21         b=2*ds'*(dn-ds);         % coefficients
22         c=ds'*ds-r^2;            % of quadratic
23
24         tau=1+(-b+sqrt(b^2-4*a*c))/(2*a); % find where on dogleg
25
26         p=ds+(tau-1)*(dn-ds);    % and the step to there
27      end                         % finished finding dogleg part
28   end                            % finished solving subproblem
29
30   rcs=0;                         % report success
31 end
```

The routine begins [4-10] by finding the full Newton step `dn`. If that falls within the trust region [12] it is used [13] as the dogleg step `p`. Otherwise [15] the formula we derived in §10.5 is used to find the full steepest-descent step `ds`. If `r` is no more than the length of that step [16] then the trust-region boundary intersects the steepest-descent part of the dogleg, so the first formula on the previous page is used [17-18] to find `p`. Otherwise [20-22] the Newton and steepest-descent steps $\mathbf{d}^{\mathrm{N}}$ and $\mathbf{d}^{\mathrm{S}}$ are used to compute the coefficients `a`, `b`, and `c`, the quadratic formula is used [24] to find τ, and the second formula on the previous page is used [26] to find `p`. I tested `dogsub.m` by finding $\mathbf{p}(\hat{\tau})$ and hence $\mathbf{x}^{\mathrm{dog}}(\hat{r})$ for our example, as shown below ($\hat{r}$ can be found by counting contour lines in the picture).

```
octave:1> H=[2,0;0,20];
octave:2> g=[-4;20];
octave:3> rhat=(12/21)*sqrt(5)
rhat =  1.2778
octave:4> [p,rcs]=dogsub(H,g,rhat)
p =

   0.76326
  -1.02473

rcs = 0
octave:5> norm(p)
ans =  1.2778
octave:6> quit
```


I did not specify an $f_0(\mathbf{x})$ for this example so we can't compute the objective reduction achieved by moving to $\mathbf{x}^{\text{dog}}$. But I did modify the `bss1trust.m` program of §17.3.1 to use `dogsub.m` and to plot the step to $\mathbf{x}^{\text{dog}}$ for that example along with the steps to $\mathbf{x}^{\text{newt}}$ and $\mathbf{x}^{\text{trust}}$ (see Exercise 17.6.32). In the contour diagrams below, $q(\mathbf{x}^{\text{dog}})$ is not as low as $q(\mathbf{x}^{\text{trust}})$ but $f_0(\mathbf{x}^{\text{dog}}) = 9.9546$ happens to be lower than $f_0(\mathbf{x}^{\text{trust}}) = 11.280$.



17.4 An Adaptive Dogleg Newton Algorithm

To implement the trust-region idea I wrote the MATLAB function `trust.m` listed on the next page. It begins [4-7] by finding the function value, gradient, and Hessian at the starting point and [9-16] initializing the trust-region radius `r` to the length of a full Newton step from there. Then [19-53] it performs up to `kmax` optimization iterations. The first stanza in the optimization loop [20-24] tests for convergence. The second stanza [26-47] is our familiar radius-adjustment scheme, but now [27] it calculates a new step `p` for each trial radius. This new $\mathbf{p}$ in turn affects the value of ρ and hence the determination of whether the trial radius provides sufficient objective decrease, so at the conclusion of the process `r` does provide sufficient decrease and `p` is the dogleg solution of the trust-region subproblem for that radius. This simultaneous determination of `r` and `p` is essential for achieving the advantage of using a solution to the trust-region subproblem, and is the defining characteristic of the trust-region approach. The third stanza [49-52] performs the move to the new point and updates the function value, gradient, and Hessian so that the quadratic model [33] will be evaluated correctly in the next iteration.

```
 1 function [xstar,kp,rc]=trust(xzero,kmax,epz,fcn,grd,hsn)
 2 % adaptive dogleg Newton algorithm
 3
 4   x=xzero;                          % set starting point
 5   f=fcn(x);                         % construct
 6   g=grd(x);                         % quadratic
 7   H=hsn(x);                         % model
 8
 9   [U,pz]=chol(H);                   % find dn, full Newton step
10   if(pz~=0)                         % is H positive definite?
11      rc=3;                          % no; report error
12      return                         % and give up
13   end                               % done checking factorization
14   y=U'\(-g);                        % solve U'y=-g for y
15   dn=U\y;                           % solve Ud=y for dn
16   r=norm(dn);                       % its length is initial r
17   mu=0.25; eta=0.75; tmax=52;       % set r adjustment parameters
18
19   for kp=1:kmax                     % up to kmax optimization steps
20     if(norm(g) <= epz)              % is x close to stationary?
21        xstar=x;                       % yes; declare x optimal
22        rc=0;                          % report convergence
23        return                         % and return
24     end                             % not done yet
25
26     for t=1:tmax                    % find best p for a suitable r
27       [p,rcs]=dogsub(H,g,r);        % p from trust region subproblem
28       if(rcs~=0)                    % is H positive definite?
29          r=r/2;                       % no; reduce r
30          continue                     % and try again
31       end                           % done checking subproblem
32       xtry=x+p;                     % trial point
33       qtry=f+g'*p+0.5*p'*H*p;       % quadratic model value there
34       ftry=fcn(xtry);               % actual objective value there
35       rho=(f-ftry)/(f-qtry);        % reduction ratio
36       if(rho > mu)                  % accept trial step?
37          if(rho >= eta) r=2*r; end    % yes; increase r if possible
38          break                        % found suitable r and best p
39       else                          % model is untrustworthy
40          r=r/2;                       % reduce trust region radius
41       end                           % finished testing trial step
42     end                             % finished adjusting radius
43     if(rho <= mu)                   % did radius adjustment succeed?
44       rc=2;                          % no; report failure
45       xstar=xtry;                    % return the trial point
46       return                         % and give up
47     end                             % finished checking success
48
49     x=xtry;                         % move to the accepted point
50     f=fcn(x);                       % update
51     g=grd(x);                       % quadratic
52     H=hsn(x);                       % model
53   end                               % continue optimization steps
54   rc=1;                             % report out of iterations
55   xstar=x;                          % return the current solution
56
57 end
```

If the Hessian is not positive definite at the starting point 10-13 the routine reports that fact and resigns, but if it becomes non-positive-definite later then `dogsub` returns `rcs=1`.

In that case, in the hope that we have merely stepped too far, `r` is reduced 28-31 and the radius-adjustment process continues. If at the end of `tmax` radius-adjustment iterations no satisfactory `r` has been found 43-47 the routine reports that and resigns, but the reason could be either that the ρ test failed or that `H` could not be made non-positive-definite.

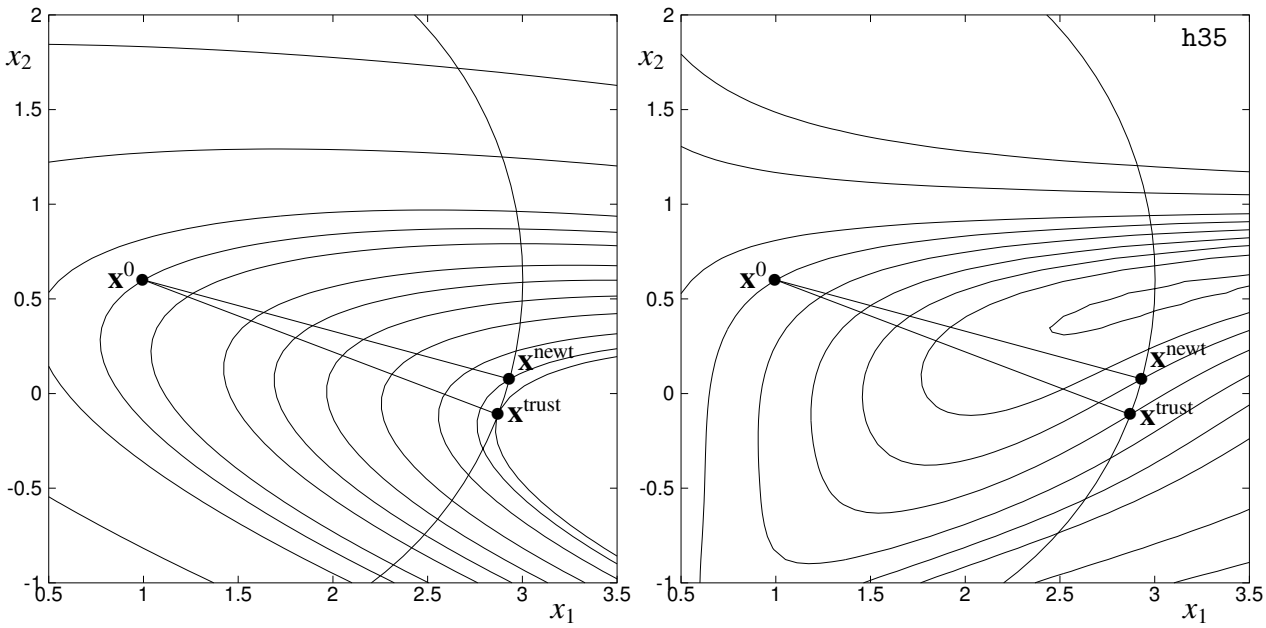
To test `trust.m` I used it to solve `bss1` and `h35`. Because the new algorithm is based on plain Newton descent I compared its behavior to that of `ntplain.m`.

```
octave:1> kmax=100;
octave:2> epz=1e-6;
octave:3> xzero=[0;3];
octave:4> [xstar,kp,rc]=trust(xzero,kmax,epz,@bss1,@bss1g,@bss1h)
xstar =

   1.99543
   0.99772

kp =  16
rc = 0
octave:5> [xstar,kp]=ntplain(xzero,kmax,epz,@bss1g,@bss1h)
xstar =

   1.99543
   0.99772

kp =  16
octave:6> xzero=[1;0.6];
octave:7> [xstar,kp,rc]=trust(xzero,kmax,epz,@h35,@h35g,@h35h)
xstar =

   3.00000
   0.50000

kp =  8
rc = 0
octave:8> [xstar,kp]=ntplain(xzero,kmax,epz,@h35g,@h35h)
xstar =

   2.9753e-14
   1.0000e+00

kp =  79
octave:9> quit
```

The iterates generated by `trust.m` and `ntplain.m` are identical for `bss1` because every full Newton step falls within the trust region; in that case the algorithm reduces to Newton descent. On `h35` `trust.m` finds $\mathbf{x}^\star$ in 7 iterations, one fewer than `ntrs.m` took from the same starting r, while `ntplain.m` converges to the stationary but non-optimal point $[0, 1]^\mathsf{T}$.

When an objective is convex like that of `bss1` it is not uncommon for its quadratic model function to remain a good approximation even far from where it was constructed, and then all the splendid machinery of the trust-region algorithm gains us nothing. When the objective is nonconvex like that of `h35` it is more likely that the quadratic model is a good approximation only near where it is constructed, and then the radius-adjustment and dogleg schemes can come into play.

It is a tragic irony of nonlinear programming (and not the last we will encounter!) that the trust-region algorithm can be frustrated by the same nonconvexity that affords it the opportunity to speed convergence. There are two reasons for this. First, nonconvexity makes it more likely that a subproblem solution $\mathbf{x}^{\text{trust}}$ will yield a *higher* objective value than $\mathbf{x}^{\text{newt}}$. If we take the first step in solving `h35` with $r = \|\mathbf{x}^\star - \mathbf{x}^0\| \approx 2$ and find the restricted Newton and exact trust-region steps, we get the points plotted in the graphs below.



The model function on the left looks like the objective on the right at $\mathbf{x}^0$, and $\rho = 0.26$ at $\mathbf{x}^{\text{trust}}$ so no radius adjustment is called for. As expected, the subproblem solution $\mathbf{x}^{\text{trust}}$ does fall on a lower contour of the model function than does $\mathbf{x}^{\text{newt}}$. But on the right we see that $\mathbf{x}^{\text{trust}}$ falls on a *higher* contour of the objective than does $\mathbf{x}^{\text{newt}}$. It is the nonconvexity of f_0 that makes the value of $\mu = \frac{1}{4}$ not quite big enough in this case. We could increase μ, but that would lead to shorter steps being taken in situations where longer ones could be used, also slowing convergence. Fortunately, one misstep does not mean that the trust-region approach will fail in subsequent iterations or be ineffective overall.

The second pernicious effect of nonconvexity is that encountering a Hessian which is not positive definite forces `trust.m` to reduce the trust-region radius, resulting in slower convergence. A small value of r condemns the algorithm to short steps, which are (adding insult to injury) probably along the steepest-descent part of the dogleg. It might seem that we could simply modify $\mathbf{H}$ when it is non-positive-definite, but when we derived the dogleg approximation we assumed that $\mathbf{d}^{\text{N}}$ is a full Newton step. If we use a modified Newton step instead then $\mathbf{x}^{\text{dog}}$ no longer approximates $\mathbf{x}^{\text{trust}}$. When we solve the subproblem

$$\left\| \left(\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}\right)^{-1} \nabla f_0(\mathbf{x}^k) \right\| = r > 0$$

we are in effect modifying the Hessian to require that $\mathbf{H}(\mathbf{x}^k) + u\mathbf{I}$ be positive definite, but to do that by using the dogleg scheme we need to first find $\mathbf{d}^{\mathrm{N}}$ and that requires $\mathbf{H}(\mathbf{x}^k)$ to be positive definite. It is because of this snag in using the dogleg approximation that some authors [5, p76] advocate more sophisticated approaches for solving the subproblem when $f_0(\mathbf{x})$ is nonconvex (see Exercise 17.6.40). As mentioned at the beginning of this Section, those techniques significantly increase the complexity of the algorithm and might not decrease the CPU time it consumes even if they do save iterations. Experiments have shown [4, p394] that trust-region methods are comparable in performance to descent methods using a line search, though one approach or the other might work better on a particular problem.

The **Levenberg-Marquardt algorithm** was the first trust-region method proposed [104] [111] and solves problems of the form

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \;\; f_0(\mathbf{x}) = \sum_{t=1}^{T} [w_t(\mathbf{x})]^2.$$

In that special case it is possible to use an approximate Hessian that is positive definite (except at $\mathbf{x}^\star$) even if the w_t are nonconvex functions, and to solve the subproblem by techniques that exploit the special structure of the approximate Hessian. Introduced at the dawn of nonlinear programming, this method was once almost universally used for parameter-estimation problems [132, p678-679] like the first one described in §8.5. Its technical details [5, p259-261] are also beyond the scope of this text.

17.5 Bounding Loops

Algorithms that are infinitely convergent, including many used in numerical optimization, are typically implemented in procedural programming languages by using a loop. Ideally some sequence of numerical calculations is repeated until the result changes by less than a convergence tolerance. Unfortunately, even if an algorithm can be proved to converge in exact arithmetic it is possible for roundoff errors to prevent the stopping test from ever being satisfied. A loop that terminates based on any condition other than a count of its iterations is a **free loop** [100, §13.3.5] and is at risk of never terminating at all, but it is often impossible to determine based simply on the rules of the algorithm how many repetitions might be needed to reach a given tolerance. In the case of nonlinear (and especially nonconvex) programming, the actual behavior of an algorithm also depends on the problem being solved. Fortunately, the same properties of floating-point numbers that prevent the exact analysis of an algorithm sometimes permit us to deduce an ultimate limit on the number of iterations that can usefully be performed.

When I described in §13.2 how `ntfs.m` modifies the Hessian, I blithely remarked that "the process continues until `H` is close enough to the identity that it is positive definite," but when we used the routine to solve `h35` in §17.1 it entered an endless loop of unsuccessful modifications. In `ntrs.m` I bounded the loop so it will end instead, but are 1022 Hessian modifications enough? How did I choose that rather peculiar limit?

In either `ntrs.m` or `ntfs.m`, if `chol()` finds that `H` is not positive definite we update the Hessian to

$$\mathbf{H} \leftarrow \gamma\mathbf{H} + (1-\gamma)\mathbf{I}$$

where $\gamma \in [0, 1)$. If a_0 is a diagonal element of $\mathbf{H}$ and b_0 is an off-diagonal element, repeating this process produces new values of those elements as follows.

$$\begin{aligned}
a_1 &= \gamma \cdot a_0 + (1-\gamma)\cdot 1 = \gamma(a_0 - 1) + 1 & b_1 &= \gamma \cdot b_0 + (1-\gamma)\cdot 0 = \gamma b_0 \\
a_2 &= \gamma(\gamma(a_0 - 1) + 1) + (1-\gamma) = \gamma^2(a_0 - 1) + 1 & b_2 &= \gamma(\gamma b_0) = \gamma^2 b_0 \\
&\vdots & &\vdots \\
a_t &= \gamma^t(a_0 - 1) + 1 & b_t &= \gamma^t b_0 \\
\lim_{t\to\infty} a_t &= 0 \cdot (a_0 - 1) + 1 = 1 & \lim_{t\to\infty} b_t &= 0 \cdot b_0 = 0
\end{aligned}$$

In practice $\mathbf{H}$ typically becomes positive definite after only one or a few modifications, but we can establish an upper bound on t by assuming that we really want to replace $\mathbf{H}$ by $\mathbf{I}$. In that case it is pointless to continue past the first modification that yields a b_t smaller than the smallest floating-point number and an a_t that is indistinguishable from 1. In other words, we are sure to have done enough modifications if

$$a_t = \gamma^t |a_0 - 1| + 1 \le \texttt{eps} + 1 \quad \text{and} \quad b_t = \gamma^t |b_0| \le \texttt{realmin}$$

Here `eps` is MATLAB's name for **machine epsilon** (about 2×10^{-16}) and `realmin` is the smallest normalized number (about 2×10^{-308}) [50, §3.1.1]. These machine constants are special binary numbers [100, §4.7] so I used base-2 logarithms to solve for t.

$$t \lg(\gamma) + \lg|a_0 - 1| \le \lg(\texttt{eps}) \quad \text{and} \quad t \lg(\gamma) + \lg|b_0| \le \lg(\texttt{realmin})$$

$$t = \max\left\{0, \frac{\lg(\texttt{eps}) - \lg|a_0 - 1|}{\lg(\gamma)}, \frac{\lg(\texttt{realmin}) - \lg|b_0|}{\lg(\gamma)}\right\}.$$

To find this limit on Hessian modifications for some typical situations we can compute $\lg(\texttt{eps}) = -52$ and $\lg(\texttt{realmin}) = -1022$, and let $\gamma = \frac{1}{2}$ so that $\lg(\gamma) = -1$ (these values are all exact). If $\mathbf{H} = \mathbf{I}$ then $a_0 = 1$ and $b_0 = 0$ so we have

$$t = \max\left\{0, \frac{-52 - (-\infty)}{-1}, \frac{-1022 - (-\infty)}{-1}\right\} = \max\{0, -\infty, -\infty\} = 0$$

because the identity requires no modification. If we have $a_0 = 0$ and $b_0 = 1$, then

$$t = \max\left\{0, \frac{-52 - (0)}{-1}, \frac{-1022 - (0)}{-1}\right\} = 1022.$$

Turning a Hessian with zeros on the diagonal and ones everywhere else into the identity seemed to me the most extreme situation that `ntrs.m` might encounter, so I chose `tmax=1022`. It would be nice to have a sharper bound on the number of modifications required, but this extravagant bound is far better than none at all! By changing the assumptions in the analysis above you can find a value for `tmax` that reflects your own most pessimistic expectations. To be sure of not understating t it is necessary to select the diagonal element a_0 and the off-diagonal element b_0 from $\mathbf{H}$ so that the numbers $\lg|a_0 - 1|$ and $\lg|b_0|$ have their highest values. The logarithm is an increasing function, so to maximize these quantities you can use the highest values you expect for $|\mathbf{H}_{ii} - 1|$ and for $|\mathbf{H}_{ij}|$ when $i \neq j$. Of course no entry can be bigger than `realmax`, the highest floating-point value (about $2 \times 10^{+308}$).

I used a slightly different argument in §12.2 to set a limit on the number of bisections in `bls.m`, our first line search routine. If the starting interval of uncertainty has length 1, how many times t can we divide it in half before the result is so small that compared to 1 it is invisible? That would be the smallest value of t such that $1 \times (\frac{1}{2})^t \leq \texttt{eps}$ or

$$t = \frac{\lg(\texttt{eps})}{\lg(\frac{1}{2})} = \frac{-52}{-1} = 52.$$

Because of the way floating-point numbers are represented and machine epsilon is defined, this is the number of fraction bits in an 8-byte floating-point number [100, p58]. I have used the same limit wherever repeated bisections are performed, most recently in implementing the steplength adjustment algorithm of §17.2 in `ntrs.m` and `trust.m`. Here too you might think I have misjudged the perversity of numerical calculations and decide to argue for a limit that is higher or lower. As in all aspects of algorithm design, you should have a rational basis for your decision rather than picking a number arbitrarily.

17.6 Exercises

17.6.1 [E] Most applications of nonlinear programming give rise to problems that have constraints, but algorithms for solving unconstrained problems are still important. Give two reasons why.

17.6.2 [E] How do trust-region methods differ from descent methods that use a line search?

17.6.3 [E] At each iteration, Newton descent minimizes a quadratic model function $q(\mathbf{x})$. (a) Give a formula for $q(\mathbf{x})$. (b) In what attributes does $q(\mathbf{x})$ match the objective $f_0(\mathbf{x})$?

17.6.4 [E] If a quadratic model function $q(\mathbf{x})$ is constructed at $\mathbf{x}^k$, how far from $\mathbf{x}^k$ does it remain a faithful representation of $f_0(\mathbf{x})$? Explain.

17.6.5 [E] Why is it that the performance of a descent method can sometimes be improved by restricting the length of the steps that it takes? Why is it undesirable to take many short steps?

17.6.6 [H] In a restricted-steplength algorithm, why is it desirable to continuously adjust the steplength as the problem is solved? Explain how to calculate $\mathbf{p}^k$, a step of length no greater than r in the direction $\mathbf{d}^k$.

17.6.7 [E] State two reasons why `ntfs.m` might fail.

17.6.8 [E] The *objective reduction ratio*

$$\rho = \frac{f_0(\mathbf{x}^k) - f_0(\mathbf{x}^k + \mathbf{p}^k)}{f_0(\mathbf{x}^k) - q(\mathbf{x}^k + \mathbf{p}^k)}$$

measures the trustworthiness of the quadratic model $q(\mathbf{x})$. (a) For what values of ρ is the quadratic model a trustworthy representation of $f_0(\mathbf{x})$? (b) For what values of ρ does the steplength adjustment algorithm of §17.2 accept the trial steplength? (c) When does it make sense to increase the steplength? (d) When should the steplength be decreased?

17.6.9 [E] The `ntrs.m` routine of §17.2 returns a parameter `rc`. (a) What does the value of this parameter indicate? (b) Make a table showing the various values that it can take on and what they mean.

17.6.10 [P] Write a MATLAB program that invokes `ntrs.m` to solve a problem one iteration at a time. Use this code to solve the `rb` and `gpr` problems, which are described in §28.7. For each problem, plot the steplength `r` as a function of iteration `k` and explain why it changes when it does.

17.6.11 [P] Plot the error curve of `ntrs.m` when it is used to solve the `h35` problem, and estimate the algorithm's order of convergence.

17.6.12 [H] Steepest descent, Newton descent, and conjugate gradient methods are each based on a model function. On what model is each of these algorithms based?

17.6.13 [P] Using the steplength-adjustment idea of §17.2, revise `sdfs.m` to produce `sdrs.m`, an adaptive-steplength steepest-descent algorithm. Compare the behavior of your routine to that of `sdfs.m` when they are both used to solve `h35`. Does using an adaptive steplength appear, based on this one experiment, to make steepest descent more robust?

17.6.14 [E] Would it make sense to use the steplength-adjustment idea of §17.2 in the conjugate-gradient routine `plrb.m`? Explain your answer.

17.6.15 [E] What is a *trust region*?

17.6.16 [H] Show that $q(\mathbf{x}^k + \mathbf{p}) = f_0(\mathbf{x}^k) + \nabla f_0(\mathbf{x}^k)^\top \mathbf{p} + \frac{1}{2}\mathbf{p}^\top \mathbf{H}(\mathbf{x}^k)\mathbf{p}$.

17.6.17 [P] In §17.3.1, the first iteration in solving the `bss1` problem gives rise to a particular trust region. Write a MATLAB program that computes the objective reduction ratio ρ at points distributed throughout the trust region and draws a contour diagram showing how ρ varies. Does this example conform to the assumption that if $q(\mathbf{x})$ is a good approximation to $f_0(\mathbf{x})$ over the restricted Newton step then it is also a good approximation throughout the trust region?

17.6.18 [E] Why does minimizing the quadratic model function $q(\mathbf{x})$ over a trust region usually yield a point different from the restricted Newton step? Write an optimization problem whose solution is the minimizing point of $q(\mathbf{x})$ over a trust region of radius r.

17.6.19 [E] In a trust-region algorithm, what is the optimal step if the radius of the trust region is greater than the length of the full Newton step? What equations must be solved to find the optimal step if the radius of the trust region is less than the length of the full Newton step?

17.6.20 [H] Show that in general the nonlinear algebraic equation $\varphi(u) = 0$ derived in §17.3 has $2n$ roots. (Here n is the number of variables x_j in the optimization problem.)

17.6.21 [E] If in a trust-region algorithm we solve the equation $\varphi(u) = 0$, why is it necessary to choose the root $u^\star$ that makes the matrix $[\mathbf{H}(\mathbf{x}^k) + u^\star\mathbf{I}]$ positive definite? Why is it necessary that $u^\star$ be nonnegative?

17.6.22 [E] The exact solution of a trust-region subproblem minimizes the quadratic model function $q(\mathbf{x})$ over the trust region. (a) Why doesn't that necessarily minimize $f_0(\mathbf{x})$ over the trust region? (b) Under what circumstances are the two minima exactly the same? (c) Can $f_0(\mathbf{x}^{\text{trust}})$ ever be less than $q(\mathbf{x}^{\text{trust}})$? Explain.

17.6.23 [H] Write down *two* functions $f_a(\mathbf{x})$ and $f_b(\mathbf{x})$, different from one another by more than just an additive constant, for which the quadratic model constructed at $\mathbf{x}^0 = [0, 0]^\top$ is $q(\mathbf{x}) = (x_1 - 2)^2 + 10(x_2 + 1)^2$.

17.6.24 [E] If we know the quadratic model function $q(\mathbf{x})$ that matches a certain function $f_0(\mathbf{x})$ at $\hat{\mathbf{x}}$, but we don't know an equation for $f_0(\mathbf{x})$, how can we find $\nabla f_0(\hat{\mathbf{x}})$ and $\mathbf{H}(\hat{\mathbf{x}})$?

17.6.25 [E] Under what circumstances is the solution of a trust-region subproblem (a) in the boundary of the trust region; (b) in the interior of the trust region?

17.6.26 [E] In §17.3.2 the second picture shows the graphical solution of a trust-region subproblem for several values of r between 0 and $\|\mathbf{d}^{\text{N}}\|$. Plot the solution of the subproblem for values of r *bigger* than $\|\mathbf{d}^{\text{N}}\|$.

17.6.27 [H] Show that when $r = 0$ the trust-region subproblem is solved by $u^\star$ if

$$\left\|\left(\mathbf{H}(\mathbf{x}^k) + u^\star\mathbf{I}\right)^{-1}\right\| = 0.$$

17.6.28 [E] If the radius r of a trust region is very small, what is the direction of the step $\mathbf{p}^\star$ that solves the trust-region subproblem? If $r = \|\mathbf{d}^{\text{N}}\|$, what is the direction of $\mathbf{p}^\star$?

17.6.29 [E] If $\mathbf{p}^\star$ is the optimal solution of a trust-region subproblem when the trust region has radius r, describe the dogleg that approximates $\mathbf{p}^\star(r)$. Once a dogleg has been constructed, how is it used?

17.6.30 [H] If in constructing the dogleg approximation to $\mathbf{p}^\star(r)$ the Hessian is the identity matrix, what does the dogleg look like?

17.6.31 [E] Explain what `dogsub.m` does, and how it works.

17.6.32 [P] Modify the `bss1trust.m` program of §17.3.1 to use `dogsub.m` and to plot the step to $\mathbf{x}^{\text{dog}}$ for that example along with the steps to $\mathbf{x}^{\text{newt}}$ and $\mathbf{x}^{\text{trust}}$, and confirm that you obtain the pictures given in §17.3.2.

17.6.33 [P] The trust region approach is an *alternative* to descent methods that use a line search, so the algorithms we have developed in this Chapter do not enforce bounds on the variables even though it is sometimes desirable to do so. Modify the steplength-adjustment scheme to ensure that each step remains within bounds on the variables, and revise (a) `ntrs.m` and (b) `trust.m` to incorporate this feature. Test your code by imposing bounds on the variables in `h35`.

17.6.34 [E] Suppose that in solving a trust-region subproblem, the restricted Newton step goes to $\mathbf{x}^{\text{newt}}$, the exact subproblem solution is $\mathbf{x}^{\text{trust}}$, and the dogleg solution is $\mathbf{x}^{\text{dog}}$. (a) Arrange $q(\mathbf{x}^{\text{trust}})$, $q(\mathbf{x}^{\text{newt}})$, and $q(\mathbf{x}^{\text{dog}})$ in ascending order. (b) Say everything you know about the relative values of $f_0(\mathbf{x}^{\text{trust}})$, $f_0(\mathbf{x}^{\text{newt}})$, and $f_0(\mathbf{x}^{\text{dog}})$.

17.6.35 [E] How does the radius-adjustment part of a trust-region algorithm such as `trust.m` work differently from the radius-adjustment part of a restricted-steplength algorithm such as `ntrs.m`?

17.6.36 [P] Plot the convergence trajectory of `trust.m` over contours of the objective when the algorithm is used to solve the `h35` problem.

17.6.37 [P] Plot the error curve of `trust.m` when it is used to solve the `h35` problem, and estimate the algorithm's order of convergence.

17.6.38 [P] Solve `bss1` from $\mathbf{x}^0 = [2, 5]^\top$ using `ntrs.m` and `trust.m`, and explain your results.

17.6.39 [H] Show that $\mathbf{x} = [0, 1]^\top$ is a stationary point, but not a minimizing point, of `h35`.

17.6.40 [P] Study the advice given in [5, §4.3] about solving the trust-region subproblem exactly, and write a MATLAB routine `[p,rcs]=trustsub(H,g,r,tol)` that returns the subproblem solution correct to within `tol`. Revise `trust.m` to invoke this routine in place of `dogsub.m`, and compare the performance of the new version to that of the old when both are used to solve `h35`. The MATLAB `tic` and `toc` commands can be used to measure the time that a calculation uses.

17.6.41 [P] In `ntrs.m` `60-61` I was careful to guard against dividing by zero if $f(\mathbf{x}^k) \equiv q(\mathbf{x}^k + \mathbf{p})$, but to keep `trust.m` simple I took no such precaution there. (a) Explain how it might happen that the quadratic model function does not decrease in stepping from $\mathbf{x}^k$ to $\mathbf{x}^k + \mathbf{p}$. (b) Modify `trust.m` to incorporate the safeguard. (c) Is it always desirable to test the denominator before attempting a division?

17.6.42 [E] Explain why the trust-region approach might not be faster than Newton descent for minimizing a nearly-quadratic convex function. Describe two ways in which the trust-region algorithm can be frustrated if the function being minimized is *not* convex.

17.6.43 [H] The Levenberg-Marquardt algorithm minimizes an objective of the form

$$f_0(\mathbf{x}) = \sum_{t=1}^{T} [w_t(\mathbf{x})]^2$$

and uses the approximation $\mathbf{H}(\mathbf{x}) \approx \mathbf{J}(\mathbf{x})^\top \mathbf{J}(\mathbf{x})$, where $\mathbf{J}(\mathbf{x})$ is a Jacobian matrix whose rows are the gradients of the functions w_t [59, p92]. For (a) `bss1` and (b) `h35` write the objective as a sum of squares, find $\mathbf{J}$ as a function of $\mathbf{x}$, write a MATLAB function to return the approximate Hessian for a given $\mathbf{x}$, and compare the approximation to the true value of the Hessian at some points of interest for the problem. (c) Write down a function that *cannot* be expressed as a sum of squares.

17.6.44 [E] What role does the Levenberg-Marquardt algorithm play in the glorious history of numerical optimization?

17.6.45 [E] What is a *free loop*? Code in MATLAB an example of a free loop and an example of a bounded loop. Why might an algorithm that has an analytic proof of convergence continue forever anyway if it is implemented using a free loop?

17.6.46 [E] Define the following MATLAB quantities: (a) `eps`; (b) `realmin`; (c) `realmax`. What are their approximate values?

17.6.47 [H] In `ntrs.m` the Hessian modification loop is bounded, but in three §13 routines it is not! Revise each of the following codes to bound that loop: (a) `ntfs.m`; (b) `nt.m`; (c) `ntw.m`.

17.6.48 [H] The code in this book places an upper limit on the iterations of every algorithm that repeatedly divides a number by two. What is that limit, and why? Propose an alternative, and explain its rational basis.

17.6.49 [P] The steplength adjustment scheme of §17.2 doubles r whenever a step reduces the objective by enough. How many such doublings can be performed before `r` exceeds `realmax`? When that happens `r` acquires the special byte code for `Inf` [100, §4.7], and any subsequent attempts to divide it by two just yield `Inf` again. Revise `ntrs.m` to guard against this by increasing `r` more slowly (when an increase is permitted) rather than by doubling it. Ideally r should get big enough to permit the use of full Newton steps when the model is good, but remain small enough that it can be reduced quickly if the model becomes bad. However your scheme works, it should ensure that no matter how many times `r` is increased it always remains less than `realmax`.

17.6.50 [H] In §17.5, I assumed that our calculations are performed using 8-byte numbers conforming to the IEEE floating-point standard [84] because that is the precision used by MATLAB. How do the iteration limits we found change if instead the calculations are performed using 4-byte reals [100, §4.2]?

17.6.51 [P] Write a MATLAB program that averages **H** with **I** repeatedly using $\gamma = \frac{1}{2}$, and perform 1022 iterations to transform

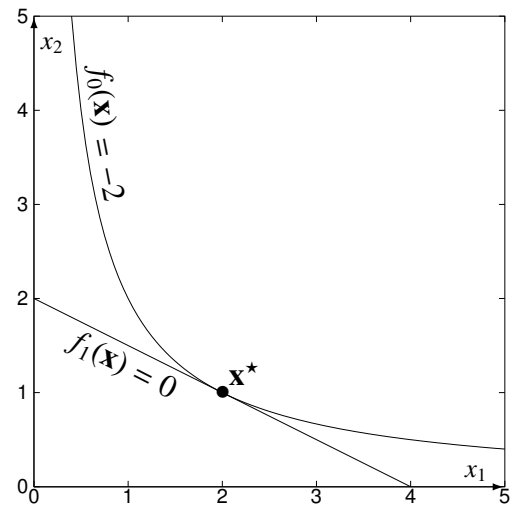
$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \text{into} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

(a) Confirm that the diagonal elements of the result are within `eps` of 1 and that the off-diagonal elements of the result are less than `realmin`. (b) Explain why the diagonal elements remain not precisely 1 and the off-diagonal elements remain not precisely 0 (Hint: IEEE floating-point arithmetic supports **subnormal** numbers [125, p20-21]). (c) How many iterations are needed to obtain diagonal elements that are precisely 1 and off-diagonal elements that are precisely 0? Can you explain why based on the kind of analysis we did in §17.5?

18

The Quadratic Penalty Method

Consider this equality-constrained nonlinear program, which I will call `p1` (it is Example 16.5 of [5]; see §28.7.20).



$$
\begin{array}{lrcl}
\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & -x_1x_2 = z \\
\text{subject to} & f_1(\mathbf{x}) & = & x_1 + 2x_2 - 4 = 0 \\
& \mathbf{x}^0 & = & \texttt{[4,4]}^\top \\
& \mathbf{x}^\star & = & \texttt{[2,1]}^\top \\
& z^\star & = & \texttt{-2}
\end{array}
$$

We can solve this problem analytically by using the Lagrange method of §15.3 as follows.

$$\mathcal{L}(\mathbf{x}, \lambda) = -x_1x_2 + \lambda(x_1 + 2x_2 - 4)$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_1} &= -x_2 + \lambda = 0 \\
\frac{\partial \mathcal{L}}{\partial x_2} &= -x_1 + 2\lambda = 0 \\
x_1 + 2x_2 - 4 &= 0
\end{aligned}
$$

These conditions are satisfied at $\mathbf{x}^\star$ with $\lambda^\star = 1$. Problem `p1` is related to the unconstrained nonlinear program below.

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \;\; \pi(\mathbf{x};\mu) = f_0(\mathbf{x}) + \mu[f_1(\mathbf{x})]^2 = -x_1x_2 + \mu(x_1 + 2x_2 - 4)^2$$

Because $f_1(\mathbf{x}^\star) = 0$ the optimal values of the two problems are equal, so $\pi(\mathbf{x}^\star) = f_0(\mathbf{x}^\star)$. The quantity $\mu[f_1(\mathbf{x})]^2$ is called a **penalty term**, and the parameter $\mu \geq 0$ is the **penalty multiplier**. If $\mu = 0$ this **penalty problem** of `p1` is unbounded; if $\mu > 0$ then minimizing $\pi(\mathbf{x})$ yields a compromise between minimizing $f_0(\mathbf{x})$ and satisfying the constraint. We can solve this problem analytically by finding the stationary points of $\pi(\mathbf{x})$.

$$
\begin{aligned}
\frac{\partial \pi}{\partial x_1} &= -x_2 + 2\mu(x_1 + 2x_2 - 4) = 0 \\
\frac{\partial \pi}{\partial x_2} &= -x_1 + 4\mu(x_1 + 2x_2 - 4) = 0
\end{aligned}
$$

These conditions are satisfied by

$$x_1 = \frac{16\mu}{8\mu - 1} \qquad x_2 = \frac{8\mu}{8\mu - 1}$$

and in the limit as $\mu \to \infty$ we find for the original problem that $x_1^\star = 2$ and $x_2^\star = 1$. We can also deduce $\lambda^\star$, by comparing the stationarity conditions for the two problems.

$$\begin{aligned}
\pi(\mathbf{x};\mu) &= f_0(\mathbf{x}) + \mu[f_1(\mathbf{x})]^2 \\
\text{so at optimality} \quad \nabla\pi(\mathbf{x};\mu) &= \nabla f_0(\mathbf{x}) + 2\mu f_1(\mathbf{x})\nabla f_1(\mathbf{x}) = \mathbf{0} \\
\mathcal{L}(\mathbf{x},\lambda) &= f_0(\mathbf{x}) + \lambda f_1(\mathbf{x}) \\
\text{so at optimality} \quad \nabla\mathcal{L}(\mathbf{x},\lambda) &= \nabla f_0(\mathbf{x}) + \lambda\nabla f_1(\mathbf{x}) = \mathbf{0}
\end{aligned}$$

Thus $\lambda(\mu) = 2\mu f_1[\mathbf{x}(\mu)]$. For our example, using the expressions we found above for $x_1(\mu)$ and $x_2(\mu)$,

$$\begin{aligned}
\lambda(\mu) &= 2\mu(x_1 + 2x_2 - 4) \\
&= 2\mu\left(\frac{16\mu}{8\mu - 1} + 2\frac{8\mu}{8\mu - 1} - 4\right) = \frac{8\mu}{8\mu - 1}.
\end{aligned}$$

Taking the limit as $\mu \to \infty$ we find for the original problem that $\lambda^\star = 1$.

It was Richard Courant who first suggested [32] (in a quite different context) studying the stationarity conditions of $\pi(\mathbf{x};\mu)$ as $\mu \to \infty$. That idea led subsequently to the development of the penalty and barrier methods [57] that are our topic in this Chapter and the next.

18.1 The Quadratic Penalty Function

The analytic approach we used above suggests a numerical method for solving equality-constrained nonlinear programs.

1. Form the **quadratic penalty function** $\pi(\mathbf{x};\mu) = f_0(\mathbf{x}) + \mu \sum_{i=1}^{m} [f_i(\mathbf{x})]^2$.
2. Set μ to a large value.
3. Solve the unconstrained penalty problem.

We have already developed a suite of routines for solving unconstrained problems, and it would be convenient to use them for minimizing the quadratic penalty function. To do that it will be necessary to provide MATLAB routines that compute the value, gradient, and Hessian of $\pi(\mathbf{x};\mu)$. In specifying an equality-constrained nonlinear program such as `p1`, on the other hand, it would be easiest to code MATLAB routines that compute the value, gradient, and Hessian of $f_i(\mathbf{x})$, where $i = 1 \ldots m$, in the standard way that I described in §15.5.

Above we found the gradient of $\pi(\mathbf{x};\mu)$ in terms of the $f_i(\mathbf{x})$ for `p1` by an application of the chain rule; the gradient of $[f_1(\mathbf{x})]^2$ is twice the quantity in brackets times the gradient of what's inside.

$$\begin{aligned}\pi(\mathbf{x};\mu) &= f_0(\mathbf{x}) + \mu[f_1(\mathbf{x})]^2 \\ \nabla\pi(\mathbf{x};\mu) &= \nabla f_0(\mathbf{x}) + 2\mu[f_1(\mathbf{x})]^1\nabla f_1(\mathbf{x})\end{aligned}$$

These are the scalar components of $\nabla\pi(\mathbf{x};\mu)$.

$$\begin{aligned}\frac{\partial\pi}{\partial x_1} &= \frac{\partial f_0}{\partial x_1} + 2\mu f_1\frac{\partial f_1}{\partial x_1} \\ \frac{\partial\pi}{\partial x_2} &= \frac{\partial f_0}{\partial x_2} + 2\mu f_1\frac{\partial f_1}{\partial x_2}\end{aligned}$$

To find the Hessian we differentiate again using the chain and product rules.

$$\begin{aligned}\mathbf{H}_\pi(\mathbf{x};\mu) &= \begin{bmatrix} \frac{\partial^2 f_0}{\partial {x_1}^2} + 2\mu\left(f_1\frac{\partial^2 f_1}{\partial {x_1}^2} + \frac{\partial f_1}{\partial x_1}\frac{\partial f_1}{\partial x_1}\right) & \frac{\partial^2 f_0}{\partial x_1\partial x_2} + 2\mu\left(f_1\frac{\partial^2 f_1}{\partial x_1\partial x_2} + \frac{\partial f_1}{\partial x_2}\frac{\partial f_1}{\partial x_1}\right) \\ \frac{\partial^2 f_0}{\partial x_2\partial x_1} + 2\mu\left(f_1\frac{\partial^2 f_1}{\partial x_2\partial x_1} + \frac{\partial f_1}{\partial x_1}\frac{\partial f_1}{\partial x_2}\right) & \frac{\partial^2 f_0}{\partial {x_2}^2} + 2\mu\left(f_1\frac{\partial^2 f_1}{\partial {x_2}^2} + \frac{\partial f_1}{\partial x_2}\frac{\partial f_1}{\partial x_2}\right)\end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial^2 f_0}{\partial {x_1}^2} & \frac{\partial^2 f_0}{\partial x_1\partial x_2} \\ \frac{\partial^2 f_0}{\partial x_2\partial x_1} & \frac{\partial^2 f_0}{\partial {x_2}^2}\end{bmatrix} + 2\mu f_1\begin{bmatrix} \frac{\partial^2 f_1}{\partial {x_1}^2} & \frac{\partial^2 f_1}{\partial x_1\partial x_2} \\ \frac{\partial^2 f_1}{\partial x_2\partial x_1} & \frac{\partial^2 f_1}{\partial {x_2}^2}\end{bmatrix} + 2\mu\begin{bmatrix}\frac{\partial f_1}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2}\end{bmatrix}\begin{bmatrix}\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2}\end{bmatrix} \\ &= \mathbf{H}_{f_0}(\mathbf{x}) + 2\mu f_1(\mathbf{x})\mathbf{H}_{f_1}(\mathbf{x}) + 2\mu\nabla f_1(\mathbf{x})\nabla f_1(\mathbf{x})^\top\end{aligned}$$

To compute these quantities I wrote the `p1pi.m`, `p1pig.m`, and `p1pih.m` routines listed below.

```
function f=p1pi(x)
  global mu
  f=p1(x,0)+mu*(p1(x,1))^2;
end
```

```
function g=p1pig(x)
  global mu
  g=p1g(x,0);
  g=g+2*mu*p1(x,1)*p1g(x,1);
end
```

```
function H=p1pih(x)
  global mu
  H=p1h(x,0);
  H=H+2*mu*p1(x,1)*p1h(x,1);
  H=H+2*mu*p1g(x,1)*p1g(x,1)';
end
```

Each of these routines can have only the single formal parameter `x`, because our unconstrained minimization codes will invoke them as `fcn(x)`, `grd(x)`, and `hsn(x)`. To compute the value and derivatives of π it is necessary also to know μ, so that number must be passed as a global parameter.

The values, gradients, and Hessians of the functions defining problem `p1` are computed by the routines `p1.m`, `p1g.m`, and `p1h.m` listed on the next page. Recall that `i=0` refers to the objective function $f_0(\mathbf{x})$ and `i=1` refers to the constraint function $f_1(\mathbf{x})$.


```
function f=p1(x,i)
  switch(i)
    case 0
      f=-x(1)*x(2);
    case 1
      f=(x(1)+2*x(2)-4);
    end
end
```

```
function g=p1g(x,i)
  switch(i)
  case 0
    g=[-x(2);
       -x(1)];
  case 1
    g=[1;
       2];
  end
end
```

```
function H=p1h(x,i)
  switch(i)
  case 0
    H=[ 0,-1;
       -1, 0];
  case 1
    H=[0,0;
       0,0];
  end
end
```

Using these six routines to define the quadratic penalty function for the `p1` problem, I tried `ntchol.m` for several values of μ. Recall from §13.1 that `ntchol.m` implements the plain full-step Newton algorithm, finding the descent direction by the factor-and-solve approach.

```
octave:1> format long
octave:2> xzero=[4;4];
octave:3> kmax=100;
octave:4> epz=1e-6;
octave:5> global mu=1
octave:6> [xstar,kp]=ntchol(xzero,kmax,epz,@p1pig,@p1pih)
ans =

   2.28571428571429
   1.14285714285714

kp =  2
octave:7> mu=100;
octave:8> [xstar,kp]=ntchol(xzero,kmax,epz,@p1pig,@p1pih)
ans =

   2.00250312891095
   1.00125156445565

kp =  2
octave:9> mu=1e11
mu =  100000000000
octave:10> [xstar,kp]=ntchol(xzero,kmax,epz,@p1pig,@p1pih)
ans =

   2.00000000000250
   1.00000000000125

kp =  100
octave:11> quit
```

According to the analytic results we derived above we should find for $\mu = 10^{11}$

$$x_1 = \frac{16\mu}{8\mu - 1} = \frac{16 \times 10^{11}}{8 \times 10^{11} - 1} = 2.00000000000250$$

$$x_2 = \frac{8\mu}{8\mu - 1} = \frac{8 \times 10^{11}}{8 \times 10^{11} - 1} = 1.00000000000125$$

and that is what we found. Further increasing μ pushes the trailing nonzero digits off to the right until, within machine precision, we get $\mathbf{x}^\star$ exactly.

In p1 the objective f_0 is not convex but the equality constraint f_1 is linear so it *is* convex. At some value of μ (see Exercise 18.5.11) the penalty problem becomes a convex program and thus easy for ntchol.m to solve. What happens if we try a problem in which f_0 is convex but f_1 is a nonlinear equality, which makes the problem not convex? To find out I experimented with this problem, which I will call p2 (it is Example 9.2.4 of [1]; see §28.7.21).

$$\begin{array}{rrcl} \text{minimize} & f_0(\mathbf{x}) & = & (x_1-2)^4+(x_1-2x_2)^2 = z \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 - x_2 = 0 \\ & \mathbf{x}^0 & = & \texttt{[1,2]}^\top \\ & \mathbf{x}^\star & = & \texttt{[0.945582993415968,0.894127197437503]}^\top \\ & z^\star & = & \texttt{1.94618371044280} \end{array}$$

This problem is just bss1 with an added constraint; I used the constraint to eliminate x_2 and solved the resulting cubic numerically to find $\mathbf{x}^\star$. The problem has the function, gradient, and Hessian routines listed below.

```
function f=p2(x,i)
  switch(i)
    case 0
      f=(x(1)-2)^4+(x(1)-2*x(2))^2;
    case 1
      f=x(1)^2-x(2);
    end
end
```

```
function g=p2g(x,i)
  switch(i)
  case 0
    g=[4*(x(1)-2)^3+2*(x(1)-2*x(2));
       2*(x(1)-2*x(2))*(-2)];
  case 1
    g=[2*x(1);
       -1];
  end
end
```

```
function H=p2h(x,i)
  switch(i)
  case 0
    H=[12*(x(1)-2)^2+2,-4;
       -4,8];
  case 1
    H=[2,0;
       0,0];
  end
end
```

We could code the calculation of $\pi(\mathbf{x};\mu)$ and its derivatives for this problem by writing routines like p1pi.m, p1pig.m, and p1pih.m, but with only slightly more work I wrote these routines instead (both pi and pie are reserved words in MATLAB so I used pye).

```
1 function f=pye(x)
2   global prob m mu
3   fcn=str2func(prob);
4   f=fcn(x,0);
5   for i=1:m
6     f=f+mu*(fcn(x,i))^2;
7   end
8 end
9
10
11
```

```
function g=pyeg(x)
  global prob m mu
  fcn=str2func(prob);
  grd=str2func([prob,'g']);
  g=grd(x,0);
  for i=1:m
    g=g+2*mu*fcn(x,i)*grd(x,i);
  end
end
```

```
function H=pyeh(x)
  global prob m mu
  fcn=str2func(prob);
  grd=str2func([prob,'g']);
  hsn=str2func([prob,'h']);
  H=hsn(x,0);
  for i=1:m
    H=H+2*mu*fcn(x,i)*hsn(x,i);
    H=H+2*mu*grd(x,i)*grd(x,i)';
  end
end
```

These three routines work for *any* problem. To see how, first consider pye.m. It begins [2] by receiving μ and two other global parameters. The variable prob contains a character string naming the problem we want to solve (e.g., p2), and m is the number of constraints in the problem. To invoke the routine that returns function values for the problem prob we need a **function handle** or pointer to the appropriate file (e.g., p2.m) so I used [3] the MATLAB built-in function str2func [50, §11.10] to obtain it as fcn. Then $\pi(\mathbf{x};\mu)$ is accumulated in f one term at a time. The first term [4] is the objective, to which we add

5-7 μ times each constraint. The `pyeg.m` and `pyeh.m` routines are similar to `pye.m`, but they calculate respectively the gradient and the Hessian of $\pi(\mathbf{x};\mu)$ by generalizing on the formulas we derived above. In `pyeg.m` string concatenation is used 4 to manufacture the name of a gradient routine (e.g., `p2g`) so that it can be used in `str2func` to find the function handle `grd`, and in `pyeh.m` the same technique is used 5 to find the function handle `hsn`.

Using these six routines I tried to solve `p2`, as shown in the Octave session below. With $\mu = 0$ the constraint is out of the problem, so `ntchol.m` returns the same unconstrained minimum that it finds for `bss1`. Increasing μ as we did in solving `p1` does move the optimal point of the `p2` penalty problem closer to $\mathbf{x}^\star$, but soon `chol()` reports that $\mathbf{H}_\pi$ is no longer positive definite. Only a small amount of penalty for violating the nonlinear equality $f_1(\mathbf{x}) = 0$ can be added into π before the penalty problem becomes too nonconvex to solve using full-step Newton descent.

```
octave:1> format long
octave:2> xzero=[1;2];
octave:3> kmax=100;
octave:4> epz=1e-6;
octave:5> [xstar,kp]=ntchol(xzero,kmax,epz,@bss1g,@bss1h)
xstar =

   1.994861768913827
   0.997430884456914

kp =  14
octave:6> global prob='p2' m=1 mu=0
octave:7> [xpi,kp]=ntchol(xzero,kmax,epz,@pyeg,@pyeh)
xpi =

   1.994861768913827
   0.997430884456914

kp =  14
octave:8> mu=4;
octave:9> [xpi,kp]=ntchol(xzero,kmax,epz,@pyeg,@pyeh)
xpi =

   1.039593971730643
   0.800276298664026

kp =  5
octave:10> mu=16;
octave:11> [xpi,kp]=ntchol(xzero,kmax,epz,@pyeg,@pyeh)
error: chol: matrix not positive definite
error: called from:
error:   /home/mike/Texts/IMP/ntchol.m at line 10, column 8
octave:12> quit
```

To investigate the causes of this failure I wrote the `p2nonpd.m` program on the next page. It plots for `p2` the same contours of $\pi(\mathbf{x};\mu)$ at four values of μ, and uses the `plotpd.m` routine of §13.2 to draw plus signs where $\mathbf{H}_\pi$ is positive definite. The output of the program consists of the four graphs on the page after the listing.

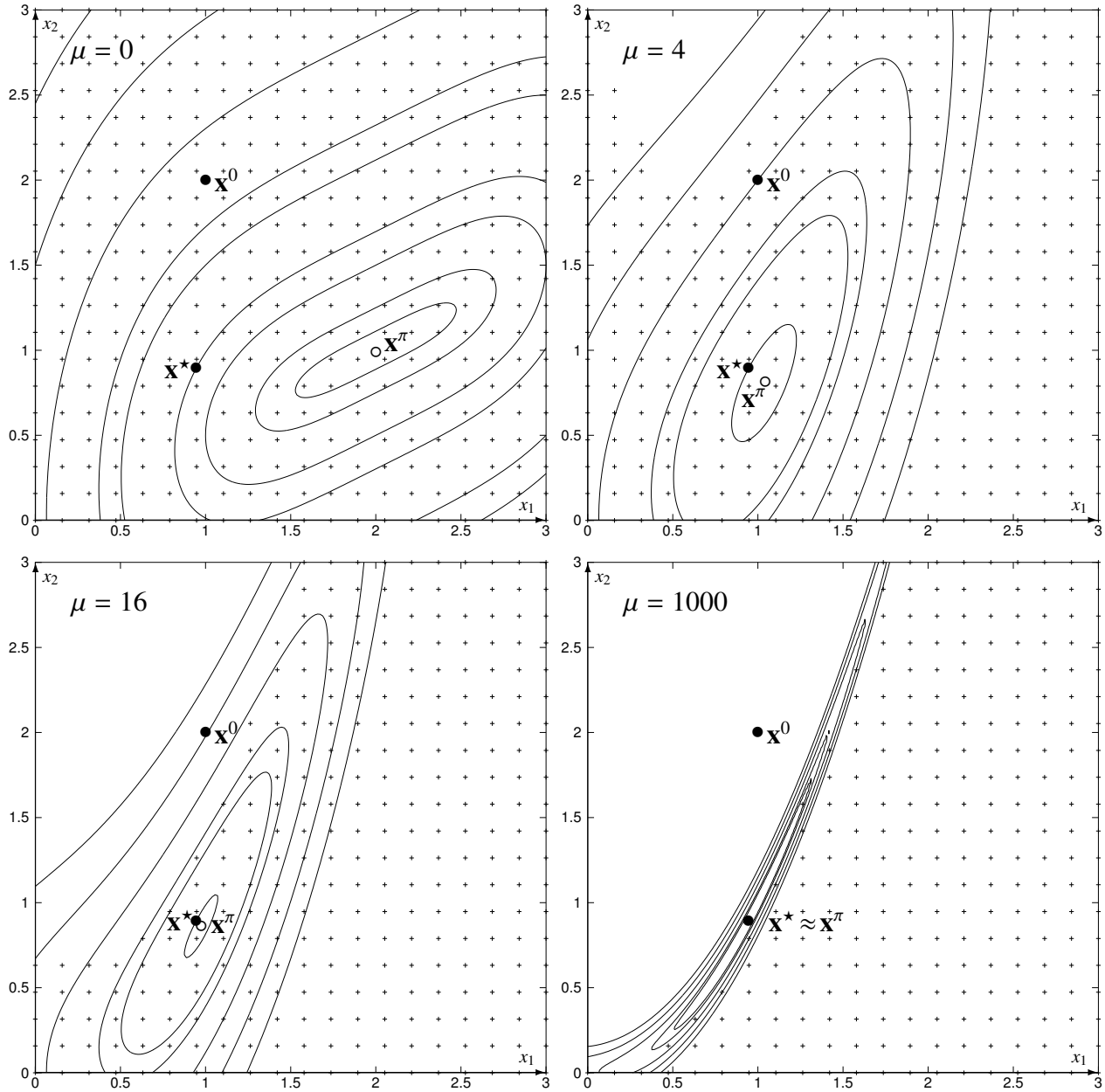

```
 1 % p2nonpd.m: study the nonconvexity of the p2 problem
 2 clear;clf
 3
 4 global prob='p2' m=1 mu=0                      % specify the problem
 5 xl=[0;0]; xh=[3;3];                            % bounds for plots
 6 xstar=[0.945582993415968;0.894127197437503]; % optimal point of p2
 7 vc=[40,25,14,7,5,pye(xstar),1,.25,.05];        % fix contour levels
 8 mus=[0,4,16,1000];                             % multiplier values
 9
10 for t=1:4                                      % consider 4 cases
11     mu=mus(t);                                 % set multiplier value
12     figure(t); set(gca,'FontSize',20)          % separate pictures
13     axis([xl(1),xh(1),xl(2),xh(2)],'square') % scale graph axes
14     hold on                                    % start plot
15     [xc,yc,zc]=gridcntr(@pye,xl,xh,200);       % grid penalty function
16     contour(xc,yc,zc,vc)                       % plot penalty contours
17     plotpd(xl,xh,20,@pyeh)                     % plot pd points
18     plot(1,2,'o')                              % plot starting point
19     plot(xstar(1),xstar(2),'o')                % plot optimal point
20     hold off                                   % done with plot
21     switch(t)                                  % print the picture
22       case 1; print -deps p2nonpd1.eps         % mu=1
23       case 2; print -deps p2nonpd2.eps         % mu=8
24       case 3; print -deps p2nonpd3.eps         % mu=16
25       case 4; print -deps p2nonpd4.eps         % mu=1000
26     end                                        % done printing
27 end                                            % done with cases
```

The program begins by [4] giving values to the global parameters that will be needed by `pye.m` and `pyeh.m`. Pointers to those routines are passed to `gridcntr` [15] and `plotpd` [17]. Next it sets [5] bounds and [6-7] contour levels for the graphs and [8] the four values of μ that will be used. Then, for each value of μ [11] it [15-16] plots the contours of $\pi(\mathbf{x};\mu)$ and [17] marks points where $\mathbf{H}_\pi$ is positive definite. The program also plots [18] $\mathbf{x}^0 = [1,2]^\top$ and [19] $\mathbf{x}^\star$ for the `p2` problem.

When $\mu = 0$ the constraint is out of the problem, so the top left panel on the next page shows the contours of the `p2` objective. That is the same as the `bss1` objective, so this picture looks like the one we drew for `bss1` at the end of §17.3.2. The starting and optimal points for `p2` are marked with closed circles • and are labeled $\mathbf{x}^0$ and $\mathbf{x}^\star$ respectively. The minimizing point $\mathbf{x}^\pi$ of $\pi(\mathbf{x};0)$, which is at $[2,1]^\top$, is marked with an open circle ∘.

Increasing μ squeezes the contour lines together, moving $\mathbf{x}^\pi$ closer to $\mathbf{x}^\star$. At $\mu = 1000$, in the bottom right panel, $\mathbf{x}^\pi$ is indistinguishable from $\mathbf{x}^\star$, and the banana shape of the contours clearly reveals the nonconvexity of the penalty function. As $\mu \to \infty$, $\mathbf{x}^\pi$ approaches $\mathbf{x}^\star$ and the contours of $\pi(\mathbf{x};\mu)$ approach the zero contour of $f_1(\mathbf{x})$, which is just the curve $x_2 = x_1^2$.

In the upper left panel the field of plus signs covers the whole graph, showing that $\mathbf{H}_\pi(\mathbf{x};0) = \mathbf{H}_{f_0}(\mathbf{x})$ is positive definite everywhere. Letting $\mu = 4$ in the upper right panel produces a region of $\mathbb{R}^2$ over which $\mathbf{H}_\pi$ is not positive definite, and in the bottom panels we see that increasing μ makes the clear region grow. If the path taken by Newton descent from $\mathbf{x}^0$ includes an iterate where the Hessian is not positive definite, then `ntchol.m` will fail as we observed in the Octave session above. As $\mu \to \infty$, the boundary of this toxic region approaches the constraint contour, so that Newton descent is possible only from its right.



This experiment reveals two reasons why p2 is hard to solve using Newton descent. First, in this problem the penalty function is nonconvex, and it becomes more nonconvex as μ is increased. Second, the region in which $\mathbf{H}_\pi$ is not positive definite grows as μ is increased, eventually engulfing $\mathbf{x}^0$ and in the limit touching $\mathbf{x}^\star$.

Now that we understand this problem it is obvious that we could make our method work by choosing a starting point in the region of $\mathbb{R}^2$ where $\mathbf{H}_\pi$ is positive definite, but for an arbitrary problem in $\mathbb{R}^n$ we don't know where that region is. We could also make our method work by using modified Newton descent, but only by accepting slower convergence.

18.2 Minimizing the Quadratic Penalty Function

Suppose that in solving the p2 problem we had begun by minimizing $\pi(\mathbf{x};\mu)$ with $\mu = 0$. Then, starting from the $\mathbf{x}^\pi$ in the top left panel on the previous page, we could have used a larger μ without making $\mathbf{H}_\pi$ non-positive-definite (in fact, proceeding from that starting point in that problem, we could have made μ as big as we liked).

If in solving an equality-constrained nonlinear program that has the penalty function $\pi(\mathbf{x};\mu)$ there is some path of iterates

$$\mathbf{x}^k = \operatorname*{argmin}_{\mathbf{x}} \pi(\mathbf{x};\mu_k)$$

leading from $\mathbf{x}^0$ to $\mathbf{x}^\star$ along which each $\mathbf{H}_\pi(\mathbf{x}^k;\mu_k)$ is positive definite for some μ_k, then we can solve the original problem by doing a sequence of full-step Newton descent minimizations of π using a suitably chosen multiplier μ_k at each step. In general there is no way of knowing beforehand what sequence of multipliers will ensure that $\mathbf{H}_\pi(\mathbf{x}^k;\mu_k)$ remains positive definite, but if $\mathbf{H}_\pi(\mathbf{x}^0;0)$ is positive definite then a reasonable heuristic [1, p484] is to start with a small value of μ^0 and increase it at every step. This leads to the following refinement of our earlier method.

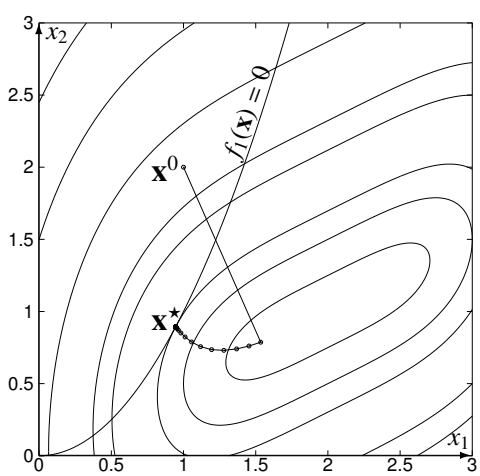
1. Form the quadratic penalty function as usual.
2. Set μ to a small value.
3. Starting from $\mathbf{x}^0$ solve the unconstrained penalty problem to get $\mathbf{x}^\pi$.
4. Replace $\mathbf{x}^0$ by $\mathbf{x}^\pi$ and increase μ.
5. If more accuracy is desired `GO TO` step 3.

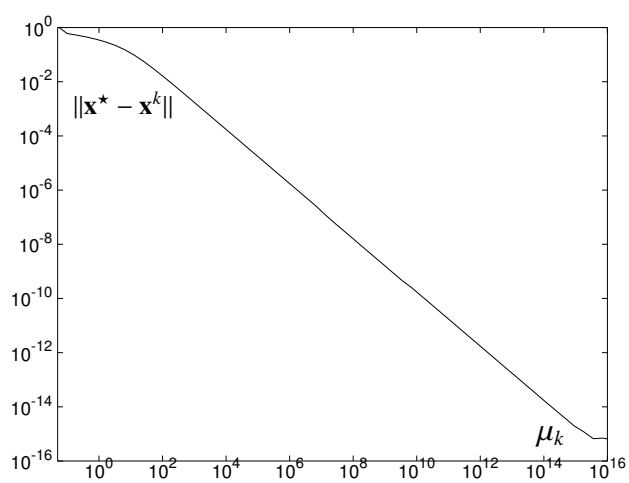
To try this idea I wrote the program `p2pen.m` listed on the next page. The program begins [5-7] by describing the problem and [9-18] plotting contours of the objective and constraint functions; because μ is initialized to zero [5], `pye.m` returns values of $f_0(\mathbf{x})$ to `gridcntr.m`. Then, starting with a small positive value of μ [21] `p2pen.m` does 59 iterations (this is just enough to get the exact answer) of [28] solving the penalty problem, [29] using the result as the next starting point, and [30] increasing μ. When it is run it produces the output shown below, which is $\mathbf{x}^\star$ for the p2 problem.

```
octave:1> p2pen
xpi =

   0.945582993415968
   0.894127197437503
```

The `p2pen.m` program also [23-26] captures the iterates of the algorithm so that it can plot the [34-37] convergence trajectory and [38-43] error curve shown below the listing.


```
% p2pen.m: solve p2 by a sequence of penalty problems
clear;clf
format long

global prob='p2' m=1 mu=0                    % specify the problem
xl=[0;0]; xh=[3;3];                          % bounds for graph
xstar=[0.945582993415968;0.894127197437503]; % optimal point of p2
vc=[40,25,14,7,5,pye(xstar),1,.25];          % fix contour levels
figure(1); set(gca,'FontSize',30)            % first picture
axis([xl(1),xh(1),xl(2),xh(2)],'square')     % scale graph axes
hold on                                      % start plot
[xc,yc,zc]=gridcntr(@pye,xl,xh,200);         % grid p2 objective
contour(xc,yc,zc,vc)                         % plot the contours
for p=1:200                                  % compute
    xp(p)=2*(p-1)/(200-1);                   % points on
    yp(p)=xp(p)^2;                           % the equality
end                                          % constraint
plot(xp,yp)                                  % and plot them

xzero=[1;2];                                 % starting point
mu=0.05;                                     % starting multiplier
for k=1:59                                   % do the sequence
    xk(k)=xzero(1);                          % for plotting later
    yk(k)=xzero(2);                          % save current point
    muk(k)=mu;                               % and current multiplier
    err(k)=norm(xstar-xzero);                % and solution error

    xpi=ntchol(xzero,10,1e-6,@pyeg,@pyeh);   % solve penalty problem
    xzero=xpi;                               % start from there
    mu=2*mu;                                 % with higher multiplier
end                                          % end of sequence
xpi                                          % report final point

plot(xk,yk,'o')                              % penalty solutions
plot(xk,yk)                                  % connected by lines
hold off                                     % done with plot
print -deps -solid p2pen.eps                 % print the plot
figure(2); set(gca,'FontSize',30)            % second picture
axis([0.05,1e16,1e-16,1])                    % scale graph axes
hold on                                      % start error plot
loglog(muk,err)                              % log(err) vs log(mu)
hold off                                     % is like log(err) vs k
print -deps -solid p2err.eps                 % print the plot
```





The first step of the algorithm, with $\mu = 0.05$, is in a direction close to that of Newton descent for minimizing the objective. As μ increases the trajectory turns toward satisfying the constraint, and as $\mathbf{x}^\star$ is approached the steps get shorter.

Although Newton descent has second-order convergence in solving each penalty problem, the error curve shows that the convergence of the overall quadratic penalty algorithm is only linear (see Exercise 18.5.20).

18.3 A Quadratic Penalty Algorithm

Unfortunately, depending on the original problem it might be that no matter how we choose the μ_k there is no sequence of penalty problems leading from $\mathbf{x}^0$ to $\mathbf{x}^\star$ in which each $\mathbf{H}_\pi(\mathbf{x}^k;\mu_k)$ is positive definite. If such a sequence does exist, our heuristic for generating the μ_k might not produce it, because we just double μ at each step without paying any attention to $\mathbf{H}_\pi(\mathbf{x};\mu)$. In §18.1 we solved the penalty problem for `p1` with $\mu = 1$, $\mu = 100$, and $\mu = 10^{11}$, but the approach we used in `p2pen.m` would fail for that problem on the first iteration because $\mathbf{H}_\pi(\mathbf{x}^0;0.05)$ is not positive definite.

```
octave:1> format long
octave:2> xzero=[4;4];
octave:3> kmax=100;
octave:4> epz=1e-6;
octave:5> global prob='p1' m=1 mu=0.05
octave:6> [xstar,kp]=ntchol(xzero,kmax,epz,@pyeg,@pyeh)
error: chol: matrix not positive definite
error: called from:
error:   /home/mike/Texts/IMP/ntchol.m at line 10, column 8
octave:7> quit
```

To be practical, an implementation of the quadratic penalty method must be robust against nonconvexity. That means using modified Newton to solve the penalty problems, even as we earnestly hope that solving them in sequence as we gradually increase μ_k will avoid or reduce the need for Hessian modifications and the resulting dilution of second-order convergence. I therefore used the `ntrs.m` routine of §17.2 in place of `ntchol.m` in the `penalty.m` routine on the next page.

This routine begins by copying [3] the input parameter for the `name` of the problem into the global variable `prob`, [4] the input number `meq` of equality constraints into the global variable `m`, and [5] the input value of μ_0 into the global variable `mu`. Then [6] it starts the solution process at the given starting point $\mathbf{x}^0$ and [9-19] solves a sequence of no more than `kmax` penalty problems using the same approach as in `p2pen.m`: the optimal solution is found [10] at the current μ, that point is used [17] as the starting point for the next iteration, and [18] the multiplier is increased. Testing showed 10 iterations of `ntrs.m` to be sufficient.

The performance of the algorithm depends on the proportion of penalty problem solutions that require $\mathbf{H}_\pi$ to be modified, so this routine [11-13] counts those iterations for [1] return to the caller as `nm`. The return code `rc` from `ntrs.m` and the multiplier μ are also passed back.


```
 1 function [xstar,kp,rc,mu,nm]=penalty(name,meq,xzero,muzero,epz)
 2   global prob m mu                % for pye, pyeg, pyeh
 3     prob=name;                      % specify the problem
 4     m=meq;                          % and the constraint count
 5     mu=muzero;                      % and the starting multiplier
 6   xpi=xzero;                      % starting point
 7   nm=0;                           % no Hessian adjustments yet
 8   kmax=1029;                      % keep mu < realmax
 9   for kp=1:kmax
10       [xstar,kpp,nmp,rc]=ntrs(xpi,0,10,epz,@pye,@pyeg,@pyeh,0.5);
11       if(nmp > 0)
12          nm=nm+1;                 % count iterations modifying H
13       end                         % in the hope there will be few
14       if(norm(xstar-xpi) <= epz) % close enough?
15          return                   % yes; return
16       end                         % no; continue
17       xpi=xstar;                  % optimal point is new start
18       mu=2*mu;                    % increase the multiplier
19   end                             % end of penalty problem sequence
20 end
```

Unlike `p2pen.m` this routine includes a convergence test [14], so [8] I set `kmax` to its largest possible value rather than requiring the user to specify it as an input parameter. There is no point in making `mu` higher than the highest floating-point number, so `kmax` should be chosen so that

$$\begin{aligned}
\mu_0 \times 2^{\texttt{kmax-1}} &< \texttt{realmax} \\
\lg(\mu_0) + (\texttt{kmax-1})\lg(2) &< \lg(\texttt{realmax}) \\
\texttt{kmax-1} &< \lg(\texttt{realmax}) - \lg(0.05) \\
\texttt{kmax} &= \lfloor 1024 - (-4.319^{+}) + 1 \rfloor = 1029.
\end{aligned}$$

Here I have used base-2 logarithms as in §17.5, and the floor function (see §14.7.2).

To test `penalty.m` I used it to solve both of our test problems.

```
octave:1> format long
octave:2> [xstar,kp,rc,mu,nm]=penalty('p1',1,[4;4],0.05,1e-16)
xstar =

   2.00000000000000
   1.00000000000000

kp =  56
rc =  1
mu =  1.80143985094820e+15
nm =  2
octave:3> [xstar,kp,rc,mu,nm]=penalty('p2',1,[1;2],0.05,1e-16)
xstar =

   0.945582993415968
   0.894127197437503

kp =  59
rc =  4
mu =  14411518807585588
nm = 0
octave:4> quit
```


In solving the `p1` problem, `ntrs.m` finds $\mathbf{H}_\pi$ non-positive-definite in each of the first two penalty function minimizations (when I looked into this I found that 70 averagings with the identity were required in each case) so `penalty.m` returns `nm=2`. Thus, of the 56 iterations it used to find $\mathbf{x}^\star$, 54 used plain Newton descent and the others essentially steepest descent.

Exact solutions were found for both `p1` and `p2` in far fewer penalty-algorithm iterations than the 1029 allowed, but for neither problem did `penalty.m` return `rc=0`. In solving the final penalty problem `ntrs.m` failed to achieve the specified convergence criterion of $\|\nabla\pi\| \le 10^{-16}$, in the case of `p1` using all 1029 of the iterations it was allowed and in the case of `p2` resigning with a Newton step too small to change $\mathbf{x}^k$. Because of the way in which $\mathbf{H}_\pi(\mathbf{x};\mu)$ depends on μ and the relentless growth of μ as the optimal point is approached, numerical difficulties inevitably arise in the use of this algorithm even when it succeeds. We will examine these in detail for problem `p2` in the next Section.

In `penalty.m` I used the same `epz` value to control both the quadratic penalty algorithm and the solution by `ntrs.m` of each penalty problem, but a more sophisticated implementation might pass a different tolerance to `ntrs.m` (see Exercise 18.5.21) or make its iteration limit depend on the number of variables n. The algorithm might also be improved by making the increase of μ depend upon the Hessian that we are trying to keep positive definite, or [5, p501] on the difficulty of minimizing the penalty function.

18.4 The Awkward Endgame

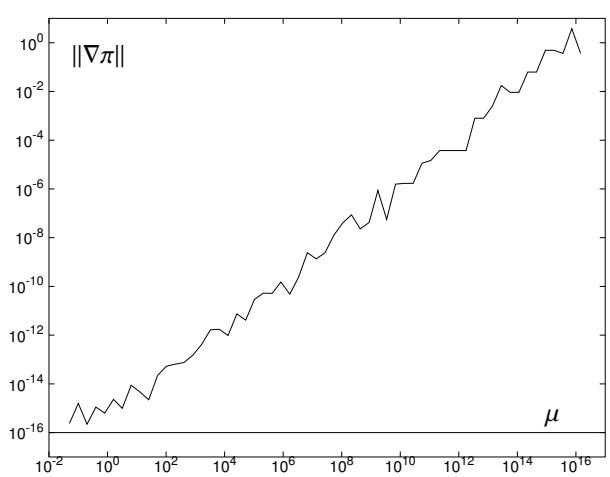
It is a cliché of nonlinear programming [1, p481-482] [4, §16.3-4] [5, p505-506] that the quadratic penalty method runs into trouble just as it is about to solve the problem. We saw evidence of this in §18.3, where `ntrs.m` failed to achieve the specified convergence criterion in solving the final penalty problem of `p2` even though the constrained minimizing point of `p2` been found by then. Alas, difficulties in minimizing $\pi(\mathbf{x};\mu)$ for large values of μ can easily result in getting the *wrong* answer to the original nonlinear program.

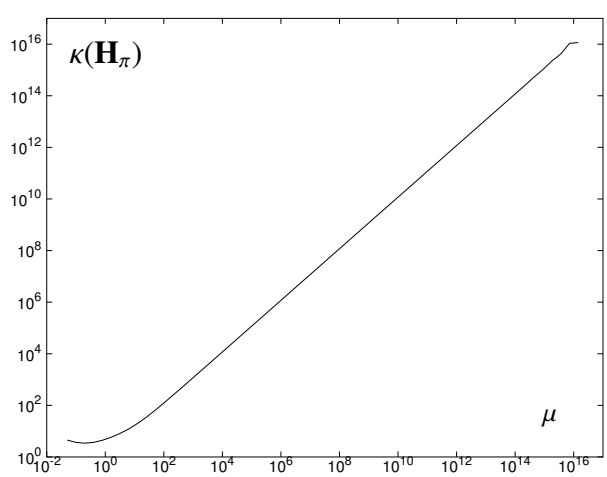
18.4.1 A Numerical Autopsy

To study this phenomenon I wrote the `ill.m` program listed on the next page. Like `p2pen.m` this program solves the `p2` problem by the quadratic penalty algorithm, but here we save [11] the norm of $\nabla\pi(\mathbf{x}^k;\mu_k)$, and [13] the condition number of $\mathbf{H}_\pi(\mathbf{x}^k;\mu_k)$ at each iteration, and [22-34] plot them versus [10] the penalty multiplier.

The left-hand graph below the listing shows that `ntrs.m` was able to find a very precise answer to the penalty problem when μ was small, but returned progressively less-stationary approximations to $\mathbf{x}^\pi$ as μ increased. The `ntrs.m` stopping condition of $\|\nabla\pi\| \le 10^{-16}$ was actually violated for every iteration performed by `penalty.m`. Fortunately the answers produced by `ntrs.m` were good enough for long enough that the quadratic penalty algorithm found a very precise solution to the original problem anyway.


```
% ill.m: monitor the penalty algorithm solution of p2
clear;clf
format long

global prob='p2' m=1 mu=0                    % specify the problem
xzero=[1;2];                                 % starting point
mu=0.05;                                     % starting multiplier
for k=1:59                                   % do the sequence
    xpi=ntrs(xzero,0,1029,1e-16,@pye,@pyeg,@pyeh,0.5); % solve
    mus(k)=mu;                               % remember the multiplier
    g(k)=norm(pyeg(xpi));                    % remember the gradient
    H=pyeh(xpi);                             % get the Hessian of pi
    kpa(k)=cond(H);                          % and remember its condition
    if(k==58)                                % at this iteration
       x58=xpi;                                % save the current point
       mu58=mu;                                % and the current multiplier
    end                                      % for study later
    xzero=xpi;                               % restart from the solution
    mu=2*mu;                                 % with higher multiplier
end                                          % end of sequence

figure(1); set(gca,'FontSize',25)            % separate the picture
hold on                                      % start the picture
axis([1e-2,1e17,1e-17,1e1])                  % set axes
loglog(mus,g)                                % plot norm(g) vs mu
plot([1e-2,1e17],[1e-16,1e-16])              % draw a line at 1e-16
hold off                                     % end the picture
print -deps -solid illg.eps                  % and print it
figure(2); set(gca,'FontSize',25)            % separate the picture
hold on                                      % start the picture
axis([1e-2,1e17,1e0,1e17])                   % set axes
loglog(mus,kpa)                              % plot kappa(H) vs mu
hold off                                     % end the picture
print -deps -solid illk.eps                  % and print it

mu=mu58;          % this was the multiplier just before the end
x58               % this was the iterate
f58=pye(x58)      % get the penalty function value there
g58=pyeg(x58)     % and the gradient
H58=pyeh(x58)     % and the Hessian
d=-inv(H58)*g58   % find the Newton descent direction
x59=x58+d         % take a full step in that direction
f59=pye(x59)      % and find the penalty function value there
```





To illuminate why Newton descent yields such rough answers when μ is large, the program also 14-17 saves $\mathbf{x}^{58}$ and μ_{58}, and 36-43 performs the final step of Newton descent one calculation at a time. The Octave session below shows those results.

```
octave:1> format long
octave:2> ill

x58 =

   0.945582993415968
   0.894127197437503

f58 =  1.94618371044279
g58 =

  -3.34866039113023
   1.77068560583615

H58 =

   51542923688977504  -27254574187494620
  -27254574187494620   14411518807585596

d =

   8.59026933787421e-17
  -5.17706537361828e-18

x59 =

   0.945582993415969
   0.894127197437503

f59 =  1.94618371044279
octave:3> inv(H58)
warning: inverse: matrix singular to machine precision, rcond = 6.78774e-17
ans =

   0.0341947606913586   0.0646679683473550
   0.0646679683473550   0.1222978621760420

octave:4> quit
```

The gradient g58 = $\nabla\pi(\mathbf{x}^{58};\mu_{58})$ is far from zero, but that turns out not to matter very much because the Hessian H58 = $\mathbf{H}_\pi(\mathbf{x}^{58};\mu_{58})$ is so huge that when it is inverted to find the full Newton step, d comes out tiny. In fact, taking the full Newton step from $\mathbf{x}^{58}$ to $\mathbf{x}^{59}$ changes only the last digit in x_1, and f59 = $\pi(\mathbf{x}^{59};\mu_{59})$ is the same as f58 = $\pi(\mathbf{x}^{58};\mu_{58})$ to machine precision so this tiny move made no difference at all in the value of π. It is hard for Newton descent to make much progress at getting $\nabla\pi$ to be zero when it has to take steps like this!

The bad news is that d is wrong even for many of the iterations when it is *not* tiny. The reason for this is that when μ is high, $\mathbf{H}_\pi(\mathbf{x};\mu)$ is close enough to singular that its inverse (or factors) cannot be found precisely using floating-point arithmetic. It is easy to see how $\mathbf{H}_\pi$ can approach singularity if we examine the p1 problem, because that penalty Hessian is a function only of μ.

Recall that for `p1`

$$\begin{aligned}\frac{\partial\pi}{\partial x_1} &= -x_2 + 2\mu(x_1 + 2x_2 - 4) = 0\\ \frac{\partial\pi}{\partial x_2} &= -x_1 + 4\mu(x_1 + 2x_2 - 4) = 0.\end{aligned}$$

Computing second derivatives we find that

$$\mathbf{H}_\pi = \begin{bmatrix} 2\mu & -1+4\mu \\ -1+4\mu & 8\mu \end{bmatrix}$$

is the matrix we must invert or factor. How accurately that can be done depends on its condition number, which was defined in §10.6.2 as

$$\kappa(\mathbf{H}_\pi) = \left\|\mathbf{H}_\pi\right\| \left\|\mathbf{H}_\pi^{-1}\right\|.$$

Ideally (see §18.4.2) we want the condition number of the Hessian to be 1, but as μ increases we find that

$$\begin{aligned}\lim_{\mu\to\infty} \mathbf{H}_\pi &= \mu\overline{\mathbf{H}} \quad \text{where} \quad \overline{\mathbf{H}} = \begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix}\\ \lim_{\mu\to\infty} \kappa(\mathbf{H}_\pi) &= \lim_{\mu\to\infty} \kappa(\mu\overline{\mathbf{H}})\\ &= \lim_{\mu\to\infty} \|\mu\overline{\mathbf{H}}\|\ \|(\mu\overline{\mathbf{H}})^{-1}\|\\ &= \lim_{\mu\to\infty} \|\overline{\mathbf{H}}\|\ \|\overline{\mathbf{H}}^{-1}\|\\ &= \kappa(\overline{\mathbf{H}}).\end{aligned}$$

Unfortunately the matrix $\overline{\mathbf{H}}$ above has determinant zero so it is singular, and [67, §2.7.2] the condition number of a singular matrix is $+\infty$. For `p2` the penalty Hessian is a function of $\mathbf{x}$ as well as of μ so it is harder to study analytically, but `ill.m` computes its condition number numerically and the right-hand graph below that listing shows its growth with μ.

In §10 and §14 we encountered the condition number of the Hessian in the context of its influence on the convergence constant for the steepest descent and conjugate gradient algorithms, which are always first-order. In contrast, Newton descent is always second-order, and as I mentioned in §13 its convergence constant does not depend on the condition number of the Hessian. These attributes make it the method of choice for minimizing the penalty function at each iteration of the quadratic penalty algorithm [5, p501]. However, as we have seen in the `p2` example, ill-conditioning of the Hessian does have a pronounced effect on the accuracy with which the Newton descent direction can be found. It is a tragic irony of nonlinear programming that as μ goes to infinity, so that $\mathbf{x}^\pi \to \mathbf{x}^\star$, inevitably also $\kappa(\mathbf{H}_\pi(\mathbf{x};\mu)) \to \infty$ so that the penalty problem solutions become more and more imprecise.

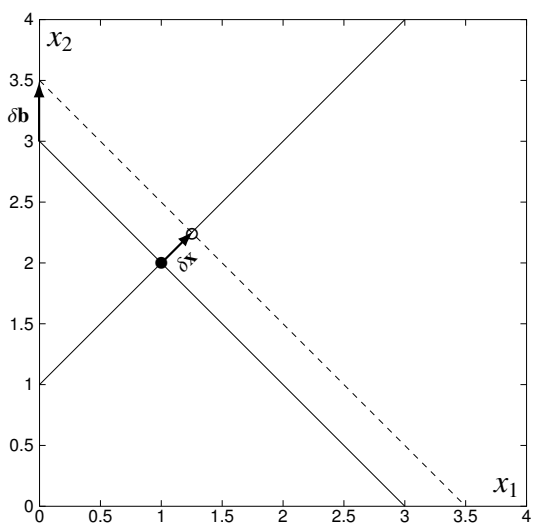
The relative speeds of these limiting processes determine how close the algorithm can get to $\mathbf{x}^\star$, and often that turns out to be not very.

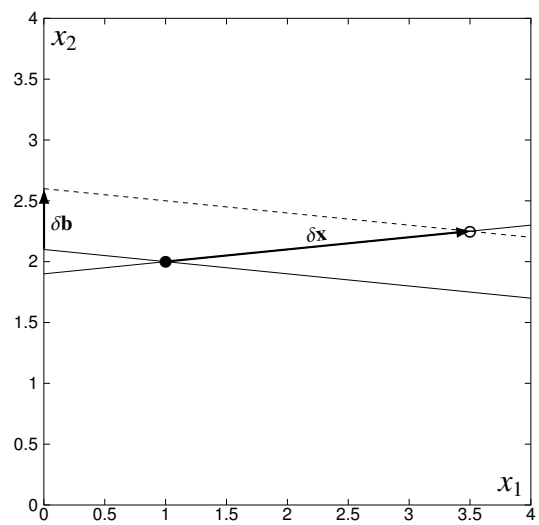
The fact that the quadratic penalty method requires μ to approach infinity, driving $\mathbf{H}_\pi$ towards singularity, is a big drawback of the algorithm and provides strong motivation for the more sophisticated penalty methods that we will take up in §20.

18.4.2 The Condition Number of a Matrix

I have claimed several times that it is hard to solve $\mathbf{Ax} = \mathbf{b}$ precisely when $\mathbf{A}$ has a high condition number, but why is that? To study this question we will consider these systems of linear equations, which both have the solution $\mathbf{x} = [1, 2]^\top$ at the intersections of the solid lines in the graphs below.

$$\begin{aligned} x_2 &= -x_1 + 3 \\ x_2 &= x_1 + 1 \end{aligned} \qquad\qquad \begin{aligned} x_2 &= -0.1x_1 + 2.1 \\ x_2 &= 0.1x_1 + 1.9 \end{aligned}$$





Adding 0.5 to the y–intercept of the first equation in each system produces the dashed lines and changes the solutions to $[1.25, 2.25]^\top$ on the left and $[3.5, 2.25]^\top$ on the right.

On the left the intercept change $\delta\mathbf{b} = [0.5, 0]^\top$ results in a change in the solution of $\delta\mathbf{x} = [0.25, 0.25]^\top$. Comparing the lengths of these vectors we find

$$\begin{aligned} \|\delta\mathbf{b}\| &= \sqrt{0.5^2 + 0^2} &&= 0.5 \\ \|\delta\mathbf{x}\| &= \sqrt{0.25^2 + 0.25^2} &&\approx 0.354 \end{aligned}$$

so the change in $\mathbf{x}$ is a little less than the change in $\mathbf{b}$. On the right the same $\delta\mathbf{b}$ produces a much bigger change in $\mathbf{x}$, $\delta\mathbf{x} = [2.5, 0.25]^\top$, with length

$$\|\delta\mathbf{x}\| = \sqrt{2.5^2 + 0.25^2} \approx 2.512$$

The linear systems above can be written as

$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} 0.1 & 1 \\ -0.1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2.1 \\ 1.9 \end{bmatrix}$$

or as

$$\mathbf{Ax} = \begin{bmatrix} p & 1 \\ -p & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2+p \\ 2-p \end{bmatrix} = \mathbf{b}$$

where $p = 1$ on the left and $p = 0.1$ on the right. The matrix $\mathbf{A}(p)$ is singular in the limit as $p \to 0$, but it has leading principal minors p and $2p$ so it is positive definite for all $p > 0$. As $p \to 0$ the angle between the solid lines in the graphical solution approaches zero for a fixed $\delta\mathbf{b}$, and $\delta\mathbf{x}$ consequently grows without bound.

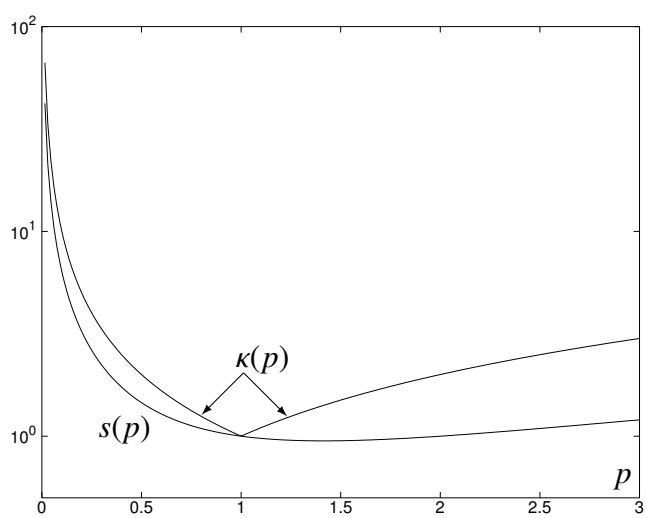
For the linear system $\mathbf{Ax} = \mathbf{b}$, the **sensitivity** s of the solution $\mathbf{x}$ to a small change in $\mathbf{b}$ is the relative change in $\mathbf{x}$ divided by the relative change in $\mathbf{b}$ [147, §7.2].

$$s = \left(\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|}\right)\Big/\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}\right)$$

The left system above, with $p = 1$, has sensitivity $s = 1$; the right system, with $p = 0.1$, has sensitivity $s \approx 6.364$. The sensitivity of a linear system depends on $\mathbf{b}$ and $\delta\mathbf{b}$ as well as on $\mathbf{A}$, but it is bounded by the **condition number** κ [150, §III.12] of the coefficient matrix.

$$s \le \kappa(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2$$

The 2-norm of a matrix is $\|\mathbf{A}\|_2 = +\sqrt{\lambda_{\max}}$ where $\lambda_{\max}$ is the maximum eigenvalue (always real) of $\mathbf{A}^\top\mathbf{A}$ (see §10.6.3). The condition number of a matrix is never less than 1, and a matrix $\mathbf{A}$ having $\kappa(\mathbf{A}) = 1$ is said to be perfectly conditioned. The graph below shows s and κ as functions of p for our matrix $\mathbf{A}(p)$. The vertical axis has a logarithmic scale.



From this picture it is clear that $\kappa(p)$ is an upper bound on $s(p)$, that they are equal only when the matrix is perfectly conditioned, and that

$$\lim_{p\to 0} s(p) = \lim_{p\to 0} \kappa(p) = +\infty.$$

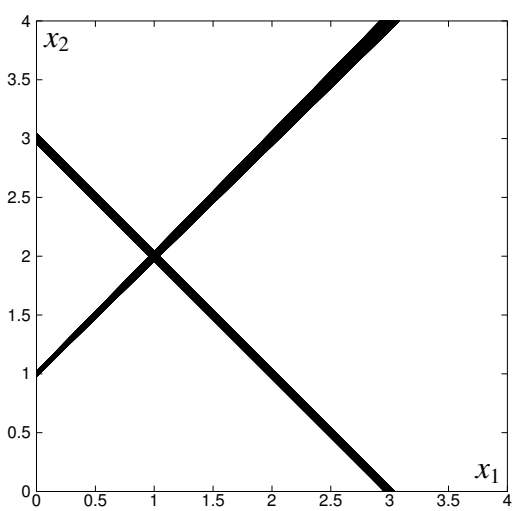
This analysis shows that if the coefficient matrix of a linear system is badly conditioned then small changes in the right-hand side can produce large changes in the solution. It can also be shown that the solution has the same sensitivity to small changes in the elements of the coefficient matrix.

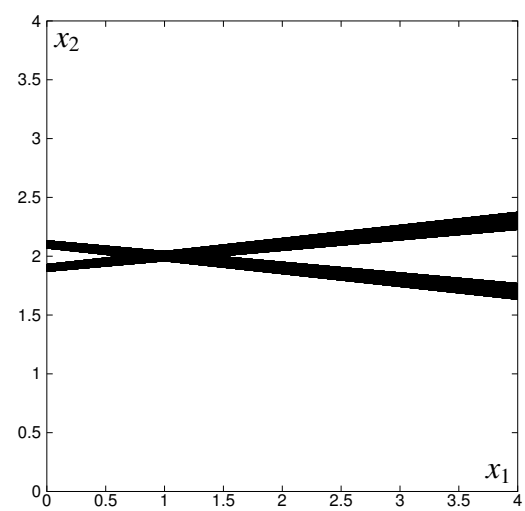
Why does it matter how sensitive the solution of $\mathbf{Ax} = \mathbf{b}$ is to the data of the problem? If $\mathbf{A}$ is positive definite and we know exactly what $\mathbf{A}$ and $\mathbf{b}$ are, can't we solve the system for $\mathbf{x}$? Well, not exactly, at least not if we are using a computer to do the arithmetic [154, §3]. Because of the way floating-point numbers are represented and stored, computed results are always contaminated by roundoff error, and if the system is badly conditioned even tiny errors can be magnified enough to make the answer too imprecise to be useful.

Suppose we want to solve the systems considered above, but the computer stores the numbers in such a way that all we know is an interval $(\min, \max)$ in which each coefficient must fall [134]. For example, if the value of each coefficient is known to within ± 0.01 the systems could be described by these equations.

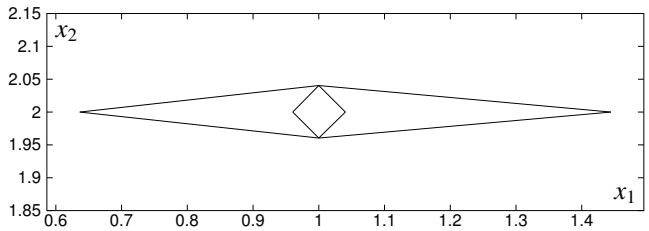
$$\begin{bmatrix} (0.99, 1.01) & (0.99, 1.01) \\ (-1.01, -0.99) & (0.99, 1.01) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} (2.99, 3.01) \\ (0.99, 1.01) \end{bmatrix} \qquad \begin{bmatrix} (0.09, 0.11) & (0.99, 1.01) \\ (-0.11, -0.09) & (0.99, 1.01) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} (2.09, 2.11) \\ (1.89, 1.91) \end{bmatrix}$$

For what values of $\mathbf{x}$ are these equations satisfied? In each equation each of the six coefficients has a minimum and maximum value, so if we want to examine all of the possible solutions we need to consider $2^6 = 64$ combinations of the extreme parameter values. I wrote a program to do that and plotted all of the solutions to each system, obtaining the graphs below.





The result of our uncertainty about the true values of the coefficients is that the line representing each equation, rather than being of zero width, is a thick wedge. Instead of being a single point, each intersection of two wedges is a diamond-shaped region. These regions of uncertainty for the two systems are enlarged below for comparison.



The true solution to the well-conditioned system could be anywhere inside the small diamond, and the true solution to the ill-conditioned one could be anywhere inside the much larger diamond. Of course the computer will return a single answer for each calculation, but if the data are represented with the limited precision we have assumed then we have no basis for preferring that result to any of the others in the region of uncertainty.

In actual floating-point calculations roundoff is even more pernicious than this picture suggests, because it is not just the problem data that are stored imprecisely; each intermediate result that is generated in the process of solving the equations is also computed and stored imprecisely. A more realistic simulation would thus produce an even larger region of uncertainty around the true solution of these systems.

In real problems the data are typically known (and stored by a computer) much more precisely than we have assumed. Floating-point calculations are usually carried out at a precision of 52 fraction bits, equivalent to 15–17 decimal digits. On the other hand, roundoff accumulates with the number of calculations performed and often we must solve linear systems having $n \gg 2$ variables, so the difficulty illustrated by our simple example is often encountered in practice. A widely-used rule of thumb is that in finding $\mathbf{x}$ one must expect to lose $\log_{10}(\kappa)$ of the digits that are correct in $\mathbf{b}$. The $p = 0.1$ example above has $\kappa(\mathbf{A}) = 10$ so the last digit in each component of $\mathbf{x}$ might be wrong; in solving the `p2` problem `ill.m` found $\kappa(\mathbf{H}_\pi) \approx 10^{16}$ at the end of the solution process, so by then all 16 of the digits in $\mathbf{d}$ had probably entered the realm of fiction.

18.5 Exercises

18.5.1 [E] Can the quadratic penalty method be used to solve nonlinear programs having inequality constraints? Explain.

18.5.2 [E] If $\pi(\mathbf{x};\mu)$ is the penalty function corresponding to a nonlinear program whose objective is $f_0(\mathbf{x})$, why is $\pi(\mathbf{x}^\star;\mu) = f_0(\mathbf{x}^\star)$ for all values of μ?

18.5.3 [E] Suppose that a nonlinear program has the form

$$\begin{aligned} &\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x}) \\ &\text{subject to } f_i(\mathbf{x}) \;=\; 0, \quad i = 1\ldots m. \end{aligned}$$

(a) Write a formula for the associated quadratic penalty function $\pi(\mathbf{x};\mu)$. (b) Write a formula for $\nabla\pi(\mathbf{x};\mu)$. (c) Write a formula for $\mathbf{H}_\pi(\mathbf{x};\mu)$.

18.5.4 [E] For the `p1` problem, we found that $\lambda(\mu) = 2\mu f_1(\mathbf{x})$. Give a detailed explanation of that derivation. Why is it based on a correspondence between $\nabla\mathcal{L}$ and $\nabla\pi$?

18.5.5 [E] How is the `p2` problem related to the `bss1` problem?

18.5.6 [H] Solve the `p2` problem of §18.1 by analytically finding the stationary points of $\pi(\mathbf{x};\mu)$ and taking limits as $\mu \to \infty$. How practical do you think this approach is?

18.5.7 [P] In §18.1 we solved the `p1` problem for three values of μ by using `ntchol.m`, which takes full Newton steps. (a) Repeat the experiment using `ntw.m`, which uses a Wolfe line search. (b) Repeat the experiment using `plrb.m`, which implements the Polak-Ribière algorithm. In both parts use variable bounds of $\mathbf{x}^{\mathrm{H}} = [5, 5]^\top$ and $\mathbf{x}^{\mathrm{L}} = [0, 0]^\top$. To get accurate results you might need to reduce the value of `epz`.

18.5.8 [H] In §15.5, I described the standard way in which this text writes function, gradient, and Hessian routines to specify a nonlinear program with constraints. (a) Explain how the MATLAB routines `pye.m`, `pyeg.m`, and `pyeh.m` work with those problem-specifying routines to compute the quadratic penalty function of the nonlinear program. (b) Why is it necessary to pass the parameters `prob`, `m`, and `mu` as global variables? What do these variables represent?

18.5.9 [E] What does the MATLAB function `str2func()` do? What is the result of the string concatenation operation `['p1','g']`?

18.5.10 [P] In §18.1 we tried to solve the `p2` problem for three values of μ by using `ntchol.m`, which takes full Newton steps. (a) Repeat the experiment using `ntw.m`, which uses a Wolfe line search. (b) Repeat the experiment using `plrb.m`, which implements the Polak-Ribière algorithm. In both parts use variable bounds of $\mathbf{x}^{\mathrm{H}} = [3, 3]^\top$ and $\mathbf{x}^{\mathrm{L}} = [0, 0]^\top$. Do these unconstrained minimization routines work better than `ntchol.m` for solving this problem?

18.5.11 [H] In §18.1, I claimed that for the `p1` problem $\pi(\mathbf{x};\mu)$ is convex above a certain value of μ and therefore easy for `ntchol.m` to solve. (a) Derive a formula for $\mathbf{H}_\pi(\mathbf{x};\mu)$ for the `p1` problem. (b) Find the values of μ for which the matrix is positive definite. (c) A nondecreasing convex function of a convex function is convex [1, Exercise 3.10], but the square is *not* a nondecreasing function. What must be true of $f_1(\mathbf{x})$ in order for the penalty term $[f_1(\mathbf{x})]^2$ to be a convex function of $\mathbf{x}$? Show that $[x_1 + 2x_2 - 4]^2$ is a convex function.

18.5.12 [E] Give two reasons why plain Newton descent might fail to solve a quadratic penalty problem.

18.5.13 [E] Describe in words the quadratic penalty algorithm. Why does it increase the penalty multiplier gradually? What order of convergence does it have?

18.5.14 [E] Why does the `penalty.m` routine of §18.3 use modified Newton descent rather than plain Newton descent? If we are going to use modified Newton descent to solve the quadratic penalty problem, why bother to increase μ gradually?

18.5.15 [E] Why does `penalty.m` use an iteration limit of `kmax=1029`? Evaluate the expressions $\lfloor -5.3 \rfloor$ and $\lfloor 5.3 \rfloor$.

18.5.16 [P] In `penalty.m`, I chose `kmax=1029` based on the assumption that $\mu_0 = 0.05$, but then I made `muzero` an input parameter so that it can be given a *higher* value. If the routine is invoked with `muzero` set to a *lower* value than 0.05, `kp` should be allowed to get higher than 1029. Modify the code to calculate `kmax` from `muzero`, but don't let `kmax` exceed the highest value allowed for a MATLAB loop limit (see §4.1).

18.5.17 [P] In the Chapter introduction we derived for problem `p1` expressions for x_1 and x_2 that satisfy the Lagrange conditions for a stationary point of $\pi(\mathbf{x})$. (a) Write a MATLAB program that plots, over contours of the `p1` problem, the trajectory of $\mathbf{x}^\pi(\mu)$ as μ increases from 0 to a large value. (b) Write a MATLAB program that uses `penalty.m` to solve `p1` one iteration at a time, starting from $\mathbf{x}^0 = [0, 0]^\top$ and plots the convergence trajectory over contours of `p1`. (c) How should these two trajectories be related? Explain any differences between them.

18.5.18 [P] Use `penalty.m` to solve the following problem, which was first presented in Exercise 15.6.36.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & -3x_1x_3 - 4x_2x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2^2 + x_3^2 - 4 = 0 \\ & f_2(\mathbf{x}) & = & x_1x_3 - 3 = 0 \end{array}$$

18.5.19 [P] Use `penalty.m` to solve the following problem, which was first presented in Exercise 15.6.42.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 + x_3^2 - 25 = 0 \\ & f_2(\mathbf{x}) & = & 8x_1 + 14x_2 + 7x_3 - 56 = 0 \end{array}$$

18.5.20 [P] The quadratic penalty algorithm has linear convergence, but the convergence constant (affecting the slope of the error curve) depends on the speed of the method used to minimize $\pi(\mathbf{x};\mu)$ at each step of the algorithm. (a) Revise `p2pen.m` to use `sdfs.m` rather than `ntrs.m` and compare its error curve to that presented in §18.2. (b) What happens to the performance of `penalty.m` if `ntrs.m` finds it necessary to modify $\mathbf{H}_\pi$ at every step?

18.5.21 [P] In implementing the quadratic penalty algorithm it is wasteful of effort to solve the penalty problem precisely while its solution is still far from $\mathbf{x}^\star$ for the original equality-constrained nonlinear program. Modify `penalty.m` to make the tolerance used by `ntrs.m` depend on $\|\mathbf{x}^{k+1} - \mathbf{x}^k\|$. How does this change affect the performance of the algorithm in solving problems `p1` and `p2`?

18.5.22 [P] Consider the following nonlinear program [5, p500].

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -5x_1^2 + x_2^2 & \\ \text{subject to} & x_1 & = \; 1. \end{array}$$

(a) Solve the problem by inspection. (b) Write the corresponding quadratic penalty function. (c) Use `ntchol.m` to minimize the quadratic penalty function, starting from $\mathbf{x}^0 = [2, 2]^\mathsf{T}$. (d) Use `penalty.m` to solve the problem. (e) Explain why the penalty problem cannot be solved for certain values of μ.

18.5.23 [H] Consider the following nonlinear program [1, Exercise 9.7].

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1^3 + x_2^3 & \\ \text{subject to} & x_1 + x_2 & = \; 1. \end{array}$$

(a) Solve the problem analytically. (b) Explain why the corresponding penalty problem cannot be solved for *any* value of μ. (c) Is this problem ill-posed in the sense of §16.8.3?

18.5.24 [E] State two significant drawbacks of the quadratic penalty algorithm.

18.5.25 [E] Why is Newton descent the method of choice for minimizing the quadratic penalty function? When is it possible to find the Newton descent direction $\mathbf{d}$ accurately?

18.5.26 [H] Explain why, in solving the `p2` problem with `penalty.m`, the final quadratic penalty problem could not be solved precisely by `ntrs.m`.

18.5.27 [H] When using Newton descent to minimize a quadratic penalty function, it is necessary to solve the equation $[\mathbf{H}_\pi(\mathbf{x};\mu)]\mathbf{d} = -\mathbf{g}$ for the descent direction $\mathbf{d}$. Why is it hard to find $\mathbf{d}$ precisely when μ has a high value? What determines how close the quadratic penalty algorithm can get to $\mathbf{x}^\star$?

18.5.28 [E] What is the condition number of an identity matrix, $\kappa(\mathbf{I})$? What is $\kappa(2\mathbf{I})$? What is the condition number of a singular matrix?

18.5.29 [H] Compute by hand the condition number of the matrix

$$\mathbf{A} = \begin{bmatrix} 7 & 5 \\ 5 & 3 \end{bmatrix}.$$

18.5.30 [E] What MATLAB function returns the condition number of a matrix?

18.5.31 [E] In solving the linear system $\mathbf{Ax} = \mathbf{b}$, how is the sensitivity s of the solution $\mathbf{x}$ to a small change in $\mathbf{b}$ related to the condition number κ of the matrix $\mathbf{A}$? When are s and κ equal?

18.5.32 [P] In §18.4.2 the sensitivity s of the solution $\mathbf{x}$ to a small change in $\mathbf{b}$ is graphed as a function of p for the linear system in the example, using the solution $\mathbf{x} = [1, 2]^\top$ and the fixed intercept change $\delta\mathbf{b} = [0.5, 0]^\top$. For a given value of p, $\delta\mathbf{x} = \bar{\mathbf{x}} - \mathbf{x}$ where $\bar{\mathbf{x}}$ solves $\mathbf{A}(p)\bar{\mathbf{x}} = \mathbf{b}(p) + \delta\mathbf{b}$. (a) Write a MATLAB program to calculate $s(p)$ for $p = 0.015, 0.030, \ldots, 3$ and reproduce the graph. (b) On the same axes plot the condition number $\kappa(p)$ of $\mathbf{A}(p)$.

18.5.33 [E] What role does roundoff error play in frustrating the accurate solution of a linear system $\mathbf{Ax} = \mathbf{b}$ whose coefficient matrix $\mathbf{A}$ is badly conditioned? How much of the precision present in $\mathbf{b}$ is typically lost if $\mathbf{A}$ has condition number κ?

18.5.34 [H] Consider the following dual pair [161, §12.2.1], in which $\pi(\mathbf{x}; \mu)$ is the quadratic penalty function corresponding to an equality-constrained nonlinear program.

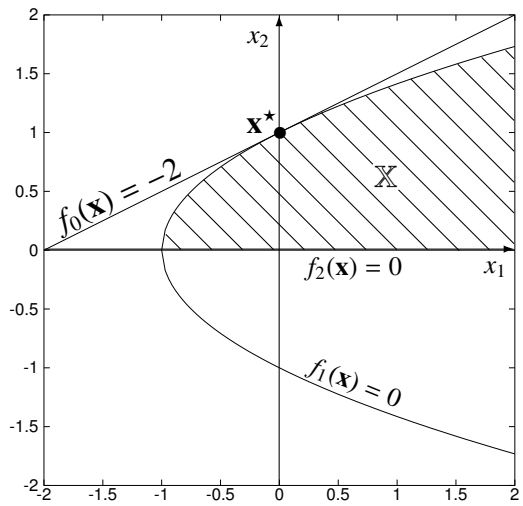
$$\mathscr{P} : \underset{\mathbf{x}}{\text{minimize}} \left\{ \sup_{\mu} \pi(\mathbf{x}; \mu) \right\} \qquad \mathscr{D} : \underset{\mu}{\text{maximize}} \left\{ \inf_{\mathbf{x}} \pi(\mathbf{x}; \mu) \right\}$$

(a) Show that the solution to $\mathscr{D}$ is $\mu = +\infty$ at the point $\mathbf{x}^\pi(\mu)$ obtained by solving the penalty problem. (b) Show that the solution to $\mathscr{P}$ is the optimal solution $\mathbf{x}^\star$ of the original equality-constrained nonlinear program. (c) Under what conditions does the solution to the penalty problem equal $\mathbf{x}^\star$?

19

The Logarithmic Barrier Method

Consider this inequality-constrained nonlinear program, which I will call `b1` (it is Example 16.1 of [4]; see §28.7.22).



$$
\begin{array}{lrcl}
\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 - 2x_2 = z \\
\text{subject to} & f_1(\mathbf{x}) & = & -x_1 + x_2^2 - 1 \le 0 \\
 & f_2(\mathbf{x}) & = & -x_2 \le 0 \\
 & \mathbf{x}^0 & = & \texttt{[0.5,0.5]}^\top \\
 & \mathbf{x}^\star & = & \texttt{[0,1]}^\top \\
 & z^\star & = & \texttt{-2}
\end{array}
$$

We can solve this problem analytically by using the KKT method of §16.5 as follows.

$$\mathcal{L}(\mathbf{x},\lambda) = x_1 - 2x_2 + \lambda_1(-x_1 + x_2^2 - 1) + \lambda_2(-x_2)$$

$$
\begin{array}{rcl}
\dfrac{\partial\mathcal{L}}{\partial x_1} & = & 1 - \lambda_1 = 0 \\
\dfrac{\partial\mathcal{L}}{\partial x_2} & = & -2 + 2x_2\lambda_1 - \lambda_2 = 0 \\
\dfrac{\partial\mathcal{L}}{\partial \lambda_1} & = & -x_1 + x_2^2 - 1 \le 0 \\
\dfrac{\partial\mathcal{L}}{\partial \lambda_2} & = & -x_2 \le 0 \\
\lambda_1 f_1(\mathbf{x}) & = & \lambda_1(-x_1 + x_2^2 - 1) = 0 \\
\lambda_2 f_2(\mathbf{x}) & = & \lambda_2(-x_2) = 0 \\
\lambda_1 & \ge & 0 \\
\lambda_2 & \ge & 0
\end{array}
$$

These conditions are satisfied at $\mathbf{x}^\star$ with $\boldsymbol{\lambda}^\star = [1,0]^\top$. Problem `b1` is related to the unconstrained **barrier problem** below,

$$
\begin{array}{rcl}
\underset{\mathbf{x}\in\mathbb{R}^2_+}{\text{minimize}}\ \beta(\mathbf{x};\mu) & = & f_0(\mathbf{x}) - \mu\ln[-f_1(\mathbf{x})] - \mu\ln[-f_2(\mathbf{x})] \\
 & = & (x_1 - 2x_2) - \mu\ln(1 + x_1 - x_2^2) - \mu\ln(x_2)
\end{array}
$$

in which the **logarithmic barrier terms** involving the natural logarithm function $\ln(\bullet)$ and the nonnegative **barrier multiplier** μ are defined only for points $\mathbf{x}$ that are strictly

interior to the feasible set $\mathbb{X}$. If $\mu = 0$ this barrier problem of `b1` is unbounded; if $\mu > 0$ then minimizing $\beta(\mathbf{x};\mu)$ yields a compromise between minimizing $f_0(\mathbf{x})$ and staying away from the boundary of $\mathbb{X}$. We can solve the barrier problem analytically by finding the stationary points of $\beta(\mathbf{x};\mu)$.

$$\frac{\partial\beta}{\partial x_1} = 1 - \frac{\mu}{1 + x_1 - x_2^2} = 0 \quad \text{A}$$

$$\frac{\partial\beta}{\partial x_2} = -2 - \frac{\mu(-2x_2)}{1 + x_1 - x_2^2} - \frac{\mu}{x_2} = 0 \quad \text{B}$$

$$\text{A} \Rightarrow x_1 = x_2^2 + \mu - 1 \quad \text{C}$$

$$\text{B} \Rightarrow -2 - \frac{\mu(-2x_2)}{1 + (x_2^2 + \mu - 1) - x_2^2} - \frac{\mu}{x_2} = -2 + \frac{2x_2\mu}{\mu} - \frac{\mu}{x_2} = 0$$

$$-2x_2 + 2x_2^2 - \mu = 0$$

$$x_2^2 - x_2 - \tfrac{1}{2}\mu = 0$$

$$\boxed{x_2(\mu) = \frac{1 + \sqrt{1 + 2\mu}}{2}}$$

Because $x_2 \geq 0$ we must use the positive square root. Then we can find

$$\text{C} \Rightarrow x_1 = x_2^2 - 1 + \mu$$

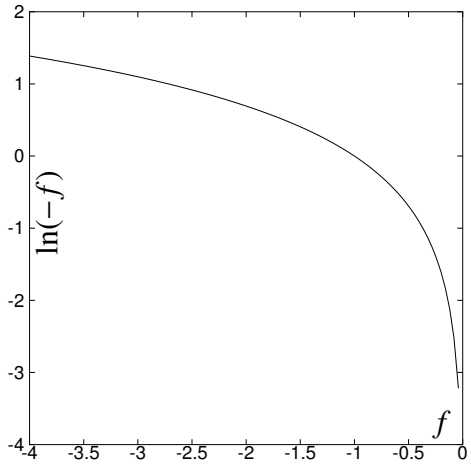
$$x_1 = \left(\frac{1 + \sqrt{1 + 2\mu}}{2}\right)^2 - 1 + \mu$$

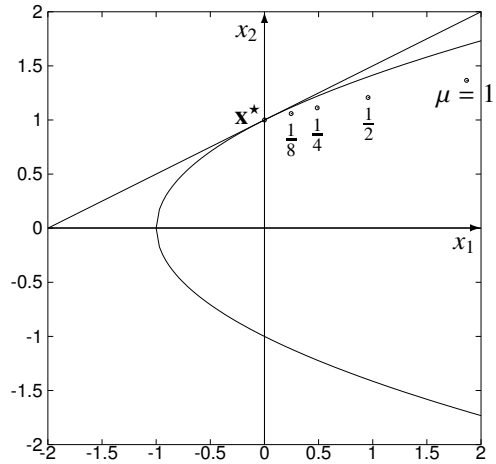
$$x_1 = \tfrac{1}{4}[1 + 2\sqrt{1 + 2\mu} + (1 + 2\mu)] - 1 + \mu$$

$$\boxed{x_1(\mu) = \frac{\sqrt{1 + 2\mu} + 3\mu - 1}{2}}$$

The boxed equations specify the point $\mathbf{x}(\mu)$ that minimizes $\beta(\mathbf{x};\mu)$ for a given value of the barrier multiplier. At high values of μ that point turns out to be deep in the interior of the feasible set, because the logarithmic barrier terms in β impose a high cost for being close to the boundary. Imagine what happens if we hold μ constant at some large value and move $\mathbf{x}$ toward the upper boundary of $\mathbb{X}$. The value of $f_1(\mathbf{x})$ approaches 0 from below, so $\ln[-f_1(\mathbf{x})]$ approaches $-\infty$ (see the top left graph on the next page). That would *increase* β, so for this value of μ the minimizing point of β must be far from the boundary.

Decreasing μ makes the logarithmic barrier terms count for less in $\beta(\mathbf{x};\mu)$ and thus allows $\mathbf{x}(\mu)$ (points in the top right graph on the next page) to get closer to the boundary and hence to the optimal point. Taking the limits of the boxed expressions as $\mu \to 0$ we find $\mathbf{x}^\star = [0, 1]^\top$.





By comparing the analytic solutions of `b1` and its barrier problem we can also deduce $\boldsymbol{\lambda}^\star$ as a function of μ.

$$\begin{aligned}
\beta(\mathbf{x};\mu) &= f_0(\mathbf{x}) - \mu\ln[-f_1(\mathbf{x})] - \mu\ln[-f_2(\mathbf{x})] \\
\text{so at optimality}\quad \nabla\beta(\mathbf{x};\mu) &= \nabla f_0(\mathbf{x}) - \frac{\mu(-1)}{-f_1(\mathbf{x})}\nabla f_1(\mathbf{x}) - \frac{\mu(-1)}{-f_2(\mathbf{x})}\nabla f_2(\mathbf{x}) = \mathbf{0} \\
\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) &= f_0(\mathbf{x}) + \lambda_1 f_1(\mathbf{x}) + \lambda_2 f_2(\mathbf{x}) \\
\text{so at optimality}\quad \nabla\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) &= \nabla f_0(\mathbf{x}) + \lambda_1\nabla f_1(\mathbf{x}) + \lambda_2\nabla f_2(\mathbf{x}) = \mathbf{0}
\end{aligned}$$

Using the formulas we found above for $x_1(\mu)$ and $x_2(\mu)$,

$$\begin{aligned}
\lambda_1 = \frac{-\mu}{x_2^2 - x_1 - 1} &= \frac{-\mu}{\left(\dfrac{1+\sqrt{1+2\mu}}{2}\right)^2 - \left(\dfrac{\sqrt{1+2\mu}+3\mu-1}{2}\right) - 1} \\
&= \frac{-\mu}{\frac{1}{4}\left[1 + 2\sqrt{1+2\mu} + (1+2\mu)\right] - \frac{1}{2}\left[\sqrt{1+2\mu} + 3\mu - 1\right] - 1} \\
&= \frac{-\mu}{-\mu} = 1 \\
\lambda_2 = \frac{\mu}{x_2} &= \frac{\mu}{\dfrac{1+\sqrt{1+2\mu}}{2}} = \frac{2\mu}{1+\sqrt{1+2\mu}}.
\end{aligned}$$

Taking limits of the final expressions for λ_1 and λ_2 as $\mu \to 0$ we find $\boldsymbol{\lambda}^\star = [1, 0]^\top$. In general [5, §19.6] a nonlinear program in standard form has the barrier problem

$$\underset{\mathbf{x}\in\mathbb{R}^n_+}{\text{minimize}}\ f_0(\mathbf{x}) - \mu\sum_{i=1}^m \ln[-f_i(\mathbf{x})] \qquad \text{which yields} \qquad \lambda_i(\mu) = \frac{\mu}{-f_i[\mathbf{x}(\mu)]} \geq 0.$$

Writing $\mathbf{x}$ and $\boldsymbol{\lambda}$ as functions of μ in the stationarity condition for the original nonlinear program, and rearranging the above formula for $\lambda_i(\mu)$, we find

$$\begin{aligned} \nabla f_0[\mathbf{x}(\mu)] + \sum_{i=1}^{m} \lambda_i(\mu)\nabla f_i[\mathbf{x}(\mu)] &= \mathbf{0} \\ \lambda_i(\mu) f_i[\mathbf{x}(\mu)] &= -\mu \\ \lambda_i(\mu) &\geq 0. \end{aligned}$$

The last two lines and $\mu > 0$ imply feasibility, so the three together are equivalent to the KKT conditions for the original nonlinear program except that in place of orthogonality we have $\lambda_i f_i(\mathbf{x}) = -\mu$. If $\bar{\mathbf{x}}$ is a local minimum for the original problem, and if $\bar{\lambda}_i > 0$ for each constraint that is active at $\bar{\mathbf{x}}$, and if every neighborhood about $\bar{\mathbf{x}}$ contains some points at which the constraints are strictly satisfied, then it can be shown [57, §3.1] [4, §16.2] that in the limit as $\mu \to 0$ the barrier problem has a solution that approaches $\bar{\mathbf{x}}$.

Notice also that if the original problem is a convex program then $\beta(\mathbf{x};\mu)$, at points strictly interior to $\mathbb{X}$, is a convex function of $\mathbf{x}$. If the constraint function $f_i(\mathbf{x})$ is convex then $-f_i(\mathbf{x})$ is concave. The logarithm is a nondecreasing concave function, and a nondecreasing concave function of a concave function is concave (see Exercise 19.6.8). Thus $\ln[-f_i(\mathbf{x})]$ is concave and $-\ln[-f_i(\mathbf{x})]$ is convex. The barrier multiplier μ is nonnegative and we assumed $f_0(\mathbf{x})$ is convex, so

$$\beta(\mathbf{x};\mu) = f_0(\mathbf{x}) + \sum_{i=1}^{m} -\mu \ln[-f_i(\mathbf{x})]$$

is the sum of convex functions and therefore must be convex. Problem `b1` is a convex program, so its barrier function is convex and should thus be easy to minimize (also see [57, p65-66]).

19.1 The Logarithmic Barrier Function

The analytic approach we used above suggests a numerical method for solving inequality-constrained nonlinear programs.

1. Form the **logarithmic barrier function** $\beta(\mathbf{x};\mu) = f_0(\mathbf{x}) - \mu \sum_{i=1}^{m} \ln[-f_i(\mathbf{x})]$.
2. Set μ to a small positive value.
3. Solve the unconstrained barrier problem, starting from a strictly feasible point $\mathbf{x}^0$ and generating only iterates $\mathbf{x}^k$ that are strictly feasible.

We will specify inequality-constrained nonlinear programs in the standard way that I described in §15.5, by writing MATLAB routines to compute the values, gradients, and Hessians of the $f_i(\mathbf{x})$. For `b1` these routines are listed at the top of the next page.


```
function f=b1(x,i)
  switch(i)
    case 0
      f=x(1)-2*x(2);
    case 1
      f=-x(1)+x(2)^2-1;
    case 2
      f=-x(2);
  end
end
```

```
function g=b1g(x,i)
  switch(i)
    case 0
      g=[1;-2];
    case 1
      g=[-1;2*x(2)];
    case 2
      g=[0;-1];
  end
end
```

```
function H=b1h(x,i)
  switch(i)
  case 0
    H=[0,0;
       0,0];
  case 1
    H=[0,0;
       0,2];
  case 2
    H=[0,0;
       0,0];
  end
end
```

The value, gradient, and Hessian of the barrier function are given by these formulas,

$$\begin{aligned}
\beta(\mathbf{x};\mu) &= f_0(\mathbf{x}) - \mu \sum_{i=1}^{m} \ln[-f_i(\mathbf{x})] \\
\nabla\beta(\mathbf{x};\mu) &= \nabla f_0(\mathbf{x}) - \mu \sum_{i=1}^{m} \frac{1}{f_i(\mathbf{x})} \nabla f_i(\mathbf{x}) \\
\mathbf{H}_\beta(\mathbf{x};\mu) &= \mathbf{H}_{f_0}(\mathbf{x}) - \mu \sum_{i=1}^{m} \left(\frac{1}{f_i(\mathbf{x})} \mathbf{H}_{f_i}(\mathbf{x}) + \frac{-1}{f_i(\mathbf{x})^2} \nabla f_i(\mathbf{x}) \nabla f_i(\mathbf{x})^\top \right)
\end{aligned}$$

which we can evaluate using routines similar to the `pye.m`, `pyeg.m`, and `pyeh.m` routines that we wrote in §18.1 to find the value, gradient, and Hessian of the quadratic penalty function. Here I used `bta.m` for the name of the routine that computes the value of $\beta(\mathbf{x};\mu)$, because `beta` is a reserved word in MATLAB.

```
function f=bta(x)
  global prob m mu
  fcn=str2func(prob);
  f=fcn(x,0);
  for i=1:m
    f=f-mu*log(-fcn(x,i));
  end
end
```

```
function g=btag(x)
  global prob m mu
  fcn=str2func(prob);
  grd=str2func([prob,'g']);
  g=grd(x,0);
  for i=1:m
    g=g-mu*grd(x,i)/fcn(x,i);
  end
end
```

```
function H=btah(x)
  global prob m mu
  fcn=str2func(prob);
  grd=str2func([prob,'g']);
  hsn=str2func([prob,'h']);
  H=hsn(x,0);
  for i=1:m
    f=fcn(x,i);
    g=grd(x,i);
    H=H-mu*hsn(x,i)/f+mu*g*g'/(f^2);
  end
end
```

In §18 we were able to minimize $\pi(\mathbf{x};\mu)$ by using unconstrained minimization routines we had already written, but it would be sheer luck if any of them succeeded in minimizing $\beta(\mathbf{x};\mu)$. Those routines, knowing nothing about inequality-constrained nonlinear programs like `b1`, are almost certain to generate some iterates $\mathbf{x}^k$ that are not strictly feasible. For the logarithm of a negative number MATLAB returns a complex value, so an infeasible $\mathbf{x}^k$ yields a complex $\beta(\mathbf{x}^k;\mu)$ for any $\mu > 0$. In the example on the next page, `xoops` is infeasible for `b1` and yields a complex value of `bta(x)`.


```
octave:1> global prob='b1' m=2 mu=1e-16
octave:2> xzero=[0.5;0.5];
octave:3> kmax=100;
octave:4> epz=1e-16;
octave:5> [xoops,kp,nm,rc]=ntfs(xzero,kmax,epz,@btag,@btah,0.5)
xoops =

  -1.3839e+16
   1.7857e+15

kp =  100
nm =  99
rc =  1
octave:6> f=bta(xoops)
f = -1.7411e+16 - 3.1416e-16i
```

Only function values that are real numbers are meaningful in an optimization problem. Of course some minimizers use just gradients and Hessians, and the formulas given earlier for those quantities do not involve logarithms, but where $\beta(\mathbf{x};\mu)$ is undefined its derivatives are also undefined [148, p144]. As shown above, stepping to an infeasible point does *not* interrupt `ntfs.m`, but does render its output useless.

To make use of the barrier method we clearly need a different unconstrained optimization routine that can minimize $\beta(\mathbf{x};\mu)$ along a trajectory of strictly feasible points [1, §9.4]. To meet this need I wrote the `ntfeas.m` function listed on the next page. The routine begins each descent iteration by 7-12 testing convergence, 13-19 factoring $\mathbf{H}_\beta$, and 20-21 finding the full Newton step `d`. Next it checks 24-29 whether the resulting trial point `xtry` 23 is strictly feasible for the original inequality constraints. If it is not, `xtry` would step too far, so `d` is halved 33 and the feasibility test is repeated. This **backtracking line search** [4, p378] is reminiscent of the steplength adaptation we used in §17.2, but now instead of adjusting the step based on the fidelity of a quadratic model we shorten it until `xtry` is strictly feasible. The calculations below show that this strategy is effective for solving problem `b1`, producing a point $\mathbf{x}^\beta$ that is close to $\mathbf{x}^\star$ and to our analytic solution of the barrier problem. There were `nr=11` iterations in which `ntfeas.m` found it necessary to restrict the length of the step it took. Allowing the routine to use more iterations changes the trailing 4 digits in the first component of $\mathbf{x}^\beta$, but roundoff prevents them from ever being found exactly.

```
octave:7> format long
octave:8> [xbeta,kp,rc,nr]=ntfeas(xzero,kmax,epz,@b1,2)
xbeta =

   1.99999975004497e-06
   1.00000049999975e+00

kp =  100
rc =  1
nr =  11
octave:9> x1=(sqrt(1+2*mu)+3*mu-1)/2
x1 =  1.99999975003529e-06
octave:10> x2=(1+sqrt(1+2*mu))/2
x2 =  1.00000049999975
octave:11> quit
```



```
function [xbeta,kp,rc,nr]=ntfeas(xzero,kmax,epz,fcn,m)
% interior-point plain Newton to minimize beta(x;mu)

  xk=xzero;                        % start from given point
  nr=0;                            % no steplength restrictions yet
  for kp=1:kmax                    % do up to kmax descent steps
      g=btag(xk);                  % gradient of beta
      if(norm(g) <= epz)           % close enough to stationary?
         xbeta=xk;                 % yes; take the current iterate
         rc=0;                     % flag convergence
         return                    % and return
      end                          % done checking convergence
      H=btah(xk);                  % Hessian of beta
      [U,p]=chol(H);               % factor it
      if(p ~= 0)                   % is it non-pd?
         xbeta=xk;                 % yes; take the current iterate
         rc=2;                     % flag nonconvergence
         return                    % and return
      end                          % done checking H pd
      y=U'\(-g);                   % solve for
      d=U\y;                       % full Newton step
      for t=1:52                   % make sure step stays in S
          xtry=xk+d;               % compute trial step
          ok=true;                 % assume xtry feasible
          for i=1:m                % check each inequality
              if(fcn(xtry,i) >= 0) % is constraint i violated?
                 ok=false;         % yes
              end                  % stepped outside of S
          end                      % done checking feasibility
          if(ok)                   % if xtry is feasible
             break                 % accept it
          else                     % otherwise
             d=d/2;                % decrease steplength
          end                      % and try again
      end                          % finished restricting step
      if(ok)                       % did we find one that works?
         xk=xtry;                  % yes; accept it
      else                         % otherwise
         xbeta=xk;                 % no Newton step stays in S
         rc=3;                     % flag nonconvergence
         return                    % and return
      end                          % the step is inside S
      if(t > 1) nr=nr+1; end       % count steplength restrictions
  end                              % continue Newton descent
  xbeta=xk;                        % take the current iterate
  rc=1;                            % and flag out of iterations

end
```

Here is another problem, which I will call `b2` (it is Example 9.4.4 of [1]; see §28.7.23). It is identical to problem `p2` of §18.1 except that the constraint is now an inequality.

$$\begin{array}{rrcl} \text{minimize} & f_0(\mathbf{x}) & = & (x_1-2)^4+(x_1-2x_2)^2=z \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2-x_2\le 0 \\ & \mathbf{x}^0 & = & [1,2]^\top \\ & \mathbf{x}^\star & = & [0.945582993415968,0.894127197437503]^\top \\ & z^\star & = & 1.94618371044280 \end{array}$$

Because the inequality constraint of this problem is active at optimality, `b2` has the same solution as `p2`. The functions $f_0(\mathbf{x})$ and $f_1(\mathbf{x})$ are the same in `b2` and `p2`, so the function, gradient, and Hessian calculations for the two problems are also identical, and to compute those quantities for `b2` we can just use the `p2.m`, `p2g.m`, and `p2h.m` routines of §18.1. Of course `bta.m`, `btag.m`, and `btah.m` will use the function values, gradients, and Hessians differently from the way that `pye.m`, `pyeg.m`, and `pyeh.m` did. In `b2` the constraint is an inequality, so like `b1` this problem is a convex program. As we noticed in §19.0 a convex program has a barrier function that is convex, so we might expect to minimize it easily. Here is what happened when I tried.

```
octave:1> global prob='p2' m=1 mu=20
octave:2> format long
octave:3> xzero=[1;2];
octave:4> epz=1e-16;
octave:5> kmax=100;
octave:6> [xbeta,kp,rc,nr]=ntfeas(xzero,kmax,epz,@p2,1)
xbeta =

   0.638265583994080
   1.945012286792191

kp =  7
rc = 0
nr = 0
octave:7> mu=1;
octave:8> [xbeta,kp,rc,nr]=ntfeas(xzero,kmax,epz,@p2,1)
xbeta =

   0.879760693576738
   0.997960886231180

kp =  100
rc =  1
nr =  1
octave:9> mu=0.5;
octave:10> [xbeta,kp,rc,nr]=ntfeas(xzero,kmax,epz,@p2,1)
xbeta =

   0.907484329825742
   0.949577675539676

kp =  100
rc =  1
nr =  1
octave:11> mu=1e-16
mu =  1.00000000000000e-16
octave:12> [xbar,kp,rc,nr]=ntfeas(xzero,kmax,epz,@p2,1)
xbar =

   1.17606481226886
   1.38312844265700

kp =  100
rc =  1
nr =  100
octave:13> quit
```


As I decreased μ from 20 to 1 to $\frac{1}{2}$, `ntfeas.m` successfully minimized the barrier function so that $\mathbf{x}^\beta$ moved closer to $\mathbf{x}^\star$, but setting $\mu = 10^{-16}$ (as we did above to solve `b1`) yielded $\mathbf{x}^\beta \approx [1.18, 1.38]^\top$ which is far from optimal. In the cases where `ntfeas.m` returned `rc=1`, I tried increasing `kmax`, but stationarity to within the very tight tolerance of `epz` $= 10^{-16}$ was never achieved and the optimal points changed very little from those printed above.

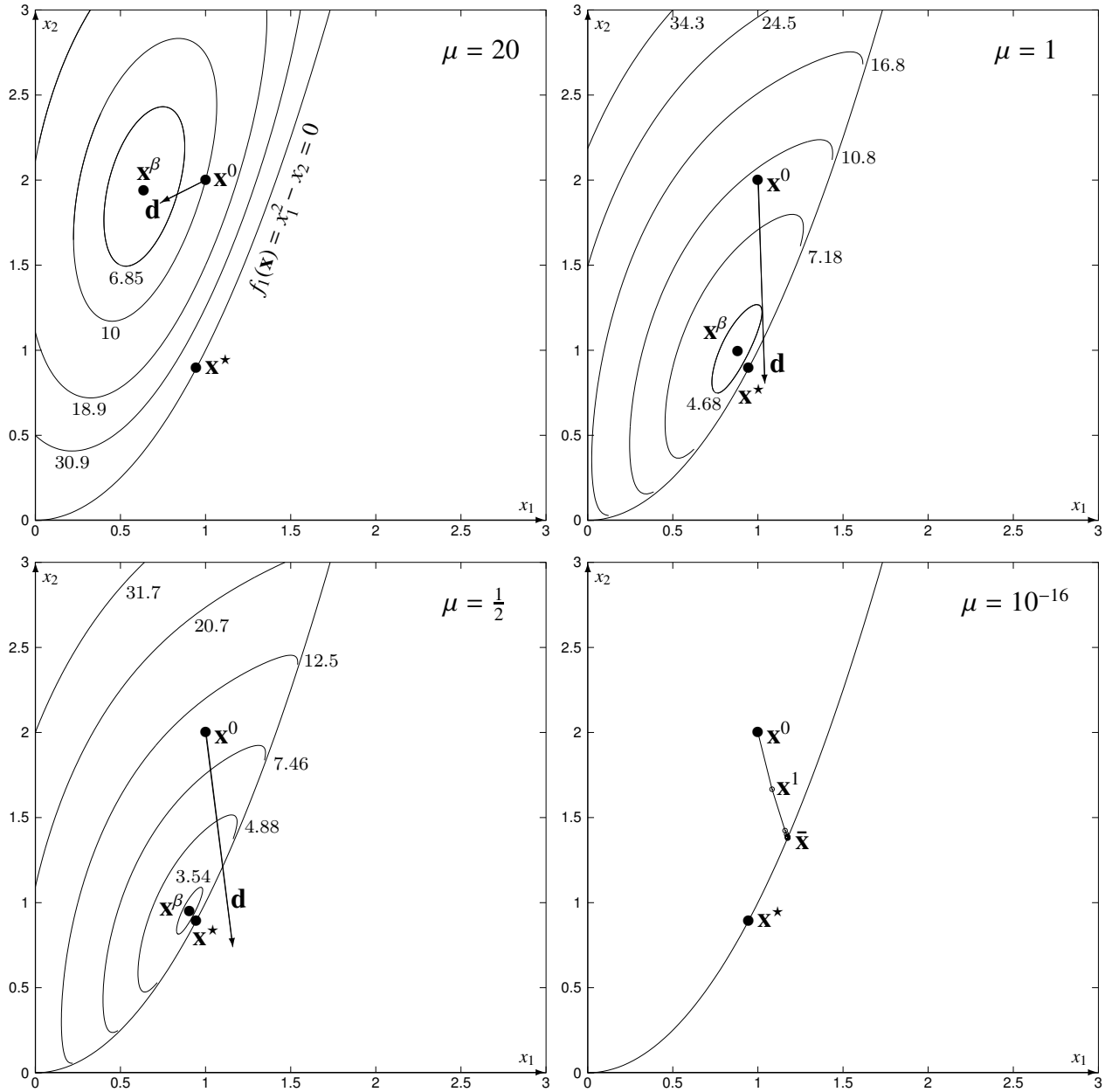
To investigate the behavior of $\beta(\mathbf{x};\mu)$ for `b2` I plotted its contours (§19.5 explains how) for $\mu = 20$, $\mu = 1$, and $\mu = \frac{1}{2}$, as shown in the first three pictures on the next page. Each of the minimizing points $\mathbf{x}^\beta$ shown in these graphs was correctly located by `ntfeas.m`, whose first step in each case was in the Newton descent direction labeled $\mathbf{d}$. The feasible set $\mathbb{X}$ of this problem is the region above the zero contour of the constraint, and β is defined only at points strictly interior to $\mathbb{X}$. In the top left picture, when $\mu = 20$, the contours of β are closed curves entirely within $\mathbb{X}$, so `ntfeas.m` can take full Newton steps (in the Octave session it reports `nr=0`). When $\mu = 1$, five of the six contours shown end at the boundary of $\mathbb{X}$, and in order to stay within $\mathbb{X}$ `ntfeas.m` is obliged to shorten its first step (it reports `nr=1`). However, the contour shown about $\mathbf{x}^\beta$ is still a closed curve inside $\mathbb{X}$ so there are still Newton directions pointing inward. Further reducing μ decreases the size of this level set that is entirely contained in $\mathbb{X}$, at the same time it deflects $\mathbf{d}$ away from $\mathbf{x}^\beta$.

The bottom right picture shows the convergence trajectory that `ntfeas.m` follows in computing the final result `xbar` printed above. Each of the 100 iterations is plotted as a separate point, but they accumulate at $\bar{\mathbf{x}}$ so only the first few are distinct. The first full Newton step again goes outside $\mathbb{X}$, so the algorithm $\boxed{\text{22–35}}$ repeatedly bisects it until $\mathbf{x}^1$ is feasible. Now, however, μ is so small that there are *no* Newton directions pointing inward. The contours of β are essentially straight lines parallel to the constraint contour at $\mathbf{x}^\star$, so for clarity I have not shown them. The only direction that `ntfeas.m` can move from $\mathbf{x}^1$ or any of the subsequent iterates is toward the boundary of $\mathbb{X}$, but it can't pass the boundary so $\mathbf{d}$ approaches zero. This phenomenon is called **jamming** [1, p560], and we will encounter it again in §23.

To avoid the risk of jamming at a suboptimal boundary point of $\mathbb{X}$, a barrier algorithm must stay far enough inside the feasible set for long enough to get close enough to $\mathbf{x}^\star$ before μ gets so small that the only direction left to go is out. Such an algorithm is called an **interior-point method**.

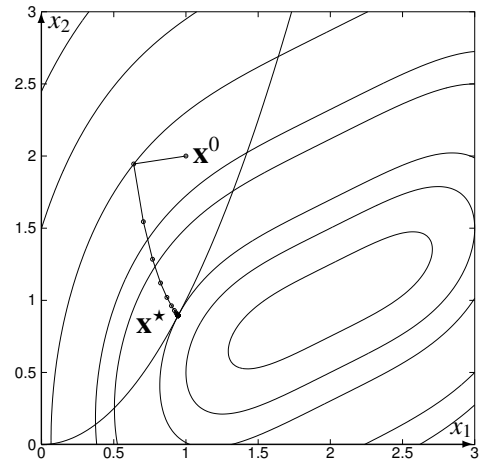
19.2 Minimizing the Barrier Function

If in minimizing β with $\mu = 10^{-16}$ we had started not from $\mathbf{x}^0$ but from the $\mathbf{x}^\beta$ we found for $\mu = \frac{1}{2}$, it seems plausible from the contour diagrams on the next page that we would have reached $\mathbf{x}^\star$ instead of stalling at $\bar{\mathbf{x}}$. This suggests that instead of solving a single barrier problem with μ set very small we should instead solve a sequence of barrier problems, each starting from the solution of the previous one, for values of μ that decrease gradually toward zero. This idea is described beneath the pictures on the next page.



1. Form the logarithmic barrier function as usual.
2. Set μ to a high value.
3. Starting from a strictly feasible $\mathbf{x}^0$ solve the unconstrained barrier problem with a method that generates only strictly feasible iterates $\mathbf{x}^k$, to get $\mathbf{x}^\beta$.
4. Replace $\mathbf{x}^0$ by $\mathbf{x}^\beta$ and decrease μ.
5. If more accuracy is desired `GO TO` step 3.

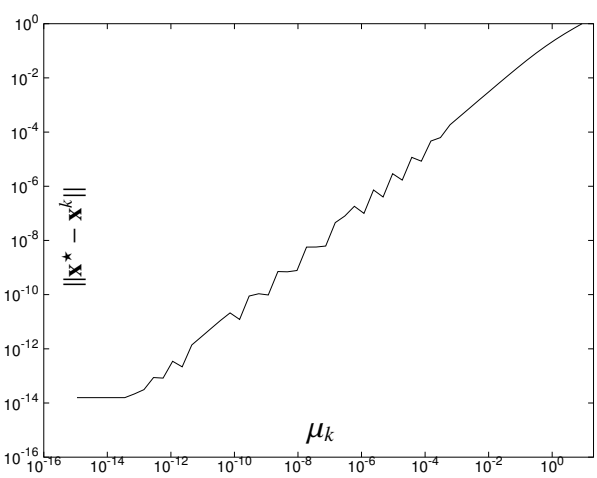
To try this idea I wrote the program `b2bar.m` listed on the next page. This code is like the `p2pen.m` program of §18.2, but it [28] uses `ntfeas.m` rather than `ntchol.m` to stay strictly feasible, [21] initializes `mu` to 20 rather than to 0.05, and [30] halves the value of `mu` on each iteration rather than doubling it.

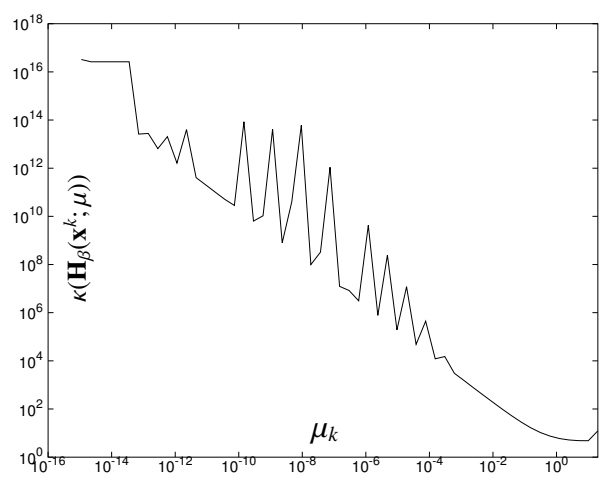


The convergence trajectory of the algorithm, shown to the right, resembles that of the quadratic penalty algorithm, but this program uses [22] only 55 iterations because that happens to be enough to produce the exact answer. Its first step is from $\mathbf{x}^0$ to the $\mathbf{x}^\beta$ that we found earlier for $\mu = 20$, pictured in the top left graph on the previous page.

The first few iterations are strongly deflected away from the boundary of the feasible set, so $\mathbf{x}^\star$ is approached from the inside. By the time μ gets to be small, so that steps away from or parallel to the boundary are no longer possible, the optimal point has been found. In addition to the zero contour of the constraint, this convergence graph includes contours of the original objective to show graphically that $\mathbf{x}^\star$ is indeed optimal.

The algorithm's error curve, shown on the left below, descends as μ *decreases* (from right to left) and because of steplength restrictions to avoid going infeasible it has more bumps than the one we plotted for the quadratic penalty method, but it reveals that this method also has linear convergence even though each step uses Newton descent.





The accuracy of the barrier method is limited by ill-conditioning of $\mathbf{H}_\beta$ just as the accuracy of the penalty method was limited by ill-conditioning of $\mathbf{H}_\pi$. The graph on the right above shows how the condition number of the barrier Hessian grows as μ decreases for `b2`. Because of the huge condition number reached at the end of the solution process the final Newton directions `d` are probably very inaccurate, but by then the steps are too tiny for that to matter.


```
% b2bar.m: solve b2 by a sequence of barrier problems
clear;clf
format long

global prob='p2' m=1 mu=0                    % specify the problem
xl=[0;0]; xh=[3;3];                          % bounds for graph
xstar=[0.945582993415968;0.894127197437503]; % optimal point of p2
vc=[40,25,14,7,5,bta(xstar),1,.25];          % fix contour levels
figure(1); set(gca,'FontSize',25)            % first picture
hold on                                      % start plot
axis([xl(1),xh(1),xl(2),xh(2)],'square')     % scale graph axes
[xc,yc,zc]=gridcntr(@bta,xl,xh,200);         % grid b1 objective
contour(xc,yc,zc,vc)                         % plot the contours
for p=1:200                                  % compute
    xp(p)=2*(p-1)/(200-1);                   % points on
    yp(p)=xp(p)^2;                           % the equality
end                                          % constraint
plot(xp,yp)                                  % and plot them

xzero=[1;2];                                 % starting point
mu=20;                                       % starting multiplier
for k=1:55                                   % do the sequence
    xk(k)=xzero(1);                          % for plotting later
    yk(k)=xzero(2);                          % save current point
    muk(k)=mu;                               % and current multiplier
    err(k)=norm(xstar-xzero);                % and solution error
    kappa(k)=cond(btah(xzero));              % and Hessian condition
    xbeta=ntfeas(xzero,10,1e-6,@p2,1);       % solve barrier problem
    xzero=xbeta;                             % start from there
    mu=mu/2;                                 % with lower multiplier
end                                          % end of sequence
xbeta                                        % report final point

plot(xk,yk,'o')                              % barrier solutions
plot(xk,yk)                                  % connected by lines
hold off                                     % done with plot
print -deps -solid b2bar.eps                 % print the plot
figure(2); set(gca,'FontSize',25)            % second picture
hold on                                      % start error plot
axis([1e-16,20,1e-16,1])                     % scale graph axes
loglog(muk,err)                              % log(err) vs log(mu)
hold off                                     % like log(err) vs k
print -deps -solid b2err.eps                 % print the plot
figure(3); set(gca,'FontSize',25)            % third picture
hold on                                      % start condition plot
axis([1e-16,20,1,1e18])                      % scale graph axes
loglog(muk,kappa)                            % log(kappa) vs log(mu)
hold off                                     % like log(kappa) vs k
print -deps -solid b2kappa.eps               % print the plot
```

19.3 A Barrier Algorithm

Problems `b1` and `b2` are both convex programs, but many applications give rise to inequality-constrained nonlinear programs that are *not* convex. A practical implementation of the barrier method must allow for the possibility that $\mathbf{H}_\beta(\mathbf{x}^k;\mu)$ will be non-positive-definite at

some points, by using modified rather than plain Newton descent. Of course we hope that starting each iteration from the optimal point of the previous one as we gradually decrease μ will allow full Newton steps to be used most of the time.

In `nt.m`, `ntw.m`, `ntfs.m`, and `ntrs.m` we wrote code to factor a Hessian that might not be positive definite, so that process should now be familiar enough that we can encapsulate it in a separate MATLAB function. The `hfact.m` routine listed below performs the `ntrs.m` version of Hessian factorization.

```
 1 function [U,rc,nm]=hfact(H,gama)
 2 % factor H, modifying it if necessary
 3
 4   nm=0;                          % prepare to count modifications
 5   [U,pz]=chol(H);                % try to factor H
 6   if(pz~=0)                      % is it positive definite?
 7     if(gama >= 1 || gama < 0)    % no; is modification possible?
 8        rc=1;                     % no; gama value prevents that
 9        return                    % resign
10     end                          % yes; modification possible
11     n=size(H,1);                 % find number of variables
12     tmax=1022;                   % limit modifications
13     for t=1:tmax                 % repeat until limit or success
14       H=gama*H+(1-gama)*eye(n);   % average with identity
15       nm=nm+1;                     % count the modification
16       [U,pt]=chol(H);              % try again to factor
17       if(pt==0) break; end         % positive definite now?
18     end                          % no; continue modifications
19     if(pt~=0)                    % was modification successful?
20       rc=2;                      % no; factorization still fails
21       return                     % resign
22     end                          % yes; modification succeeded
23   end                            % factorization complete
24   rc=0;                          % signal success
25
26 end
```

This function delivers [1] the Cholesky factor `U`, a return code `rc` to indicate what happened, and a count `nm` of the Hessian modifications performed. If [24] `rc=0`, the matrix was factored after `nm` modifications; if [8] `rc=1`, modification was required but was not allowed; and if [20] `rc=2`, `tmax` modifications did not succeed in making the matrix positive definite. This routine interprets the parameter `gama` in the standard way first described in §13.2. Below, the positive semidefinite matrix of §11.4.2 is averaged with the identity once and the positive definite result is successfully factored.

```
octave:1> H=[10,5,0;5,15,5;0,5,2];
octave:2> [U,rc,nm]=hfact(H,0.5)
U =

   2.34521   1.06600   0.00000
   0.00000   2.61985   0.95425
   0.00000   0.00000   0.76773

rc = 0
nm =  1
octave:3> quit
```


I revised `ntfeas.m` to factor $\mathbf{H}_\beta$ using `hfact.m` instead of the `chol()` function, producing the routine `ntin.m` listed below. It returns [11] `rc=0` if the convergence criterion is satisfied, or [48] `rc=1` if convergence is not achieved in `kmax` iterations, or [18] `rc=2` if `hfact.m` fails. This routine [21] counts and [1] returns as `nm` the descent iterations in which the Hessian required modification.

```
 1 function [xbeta,kp,rc,nr,nm]=ntin(xzero,kmax,epz,fcn,m)
 2 % interior-point modified Newton to minimize beta(x;mu)
 3
 4   xk=xzero;                        % start from given point
 5   nr=0;                            % no steplength restrictions yet
 6   nm=0;                            % no Hessian modifications yet
 7   for kp=1:kmax                    % do up to kmax descent steps
 8       g=btag(xk);                  % gradient of beta
 9       if(norm(g) <= epz)           % close enough to stationary?
10          xbeta=xk;                 % yes; take the current iterate
11          rc=0;                     % flag convergence
12          return                    % and return
13       end                          % done checking convergence
14       H=btah(xk);                  % Hessian of beta
15       [U,rcf,nmf]=hfact(H,0.5);    % factor it
16       if(rcf ~= 0)                 % did the factoring fail?
17          xbeta=xk;                 % yes; take the current iterate
18          rc=2;                     % flag nonconvergence
19          return                    % and return
20       end                          % done factoring H
21       if(nmf > 0) nm=nm+1; end     % count iterations modifying H
22       y=U'\(-g);                   % solve for
23       d=U\y;                       % full Newton step
24       for t=1:52                   % make sure step stays in S
25           xtry=xk+d;               % compute trial step
26           ok=true;                 % assume xtry feasible
27           for i=1:m                % check each inequality
28               if(fcn(xtry,i) >= 0) % is constraint i violated?
29                  ok=false;         % yes
30               end                  % stepped outside of S
31           end                      % done checking feasibility
32           if(ok)                   % if xtry is feasible
33              break                 % accept it
34           else                     % otherwise
35              d=d/2;                % decrease steplength
36           end                      % and try again
37       end                          % finished restricting step
38       if(ok)                       % did we find one that works?
39          xk=xtry;                  % yes; accept it
40       else                         % otherwise
41          xbeta=xk;                 % no Newton step stays in S
42          rc=3;                     % flag nonconvergence
43          return                    % and return
44       end                          % the step is inside S
45       if(t > 1) nr=nr+1; end       % count steplength restrictions
46   end                              % continue Newton descent
47   xbeta=xk;                        % take the current iterate
48   rc=1;                            % and flag out of iterations
49
50 end
```


Then I wrote the `barrier.m` code below, which is similar to the `penalty.m` routine of §18.3. Instead of `ntrs.m`, this routine uses `ntin.m` to minimize the barrier function, so it is necessary to [11] pass it a function handle `fcn` [6] of the routine that computes function values for problem `prob`.

```
 1 function [xstar,kp,rc,mu,nm]=barrier(name,mineq,xzero,muzero,epz)
 2   global prob m mu               % for bta, btag, btah
 3     prob=name;                     % specify the problem
 4     m=mineq;                       % and the constraint count
 5     mu=muzero;                     % and the starting multiplier
 6   fcn=str2func(prob);            % get function routine handle
 7   xbeta=xzero;                   % starting point
 8   nm=0;                          % no Hessian adjustments yet
 9   kmax=1023;                     % keep mu > realmin
10   for kp=1:kmax
11       [xstar,kpb,rc,nr,nmb]=ntin(xbeta,10,epz,fcn,m);
12       if(nmb > 0)
13          nm=nm+1;                % count iterations modifying H
14       end                        % in the hope there will be few
15       if(norm(xstar-xbeta) <= epz) % close enough?
16          return                  % yes; return
17       end                        % no; continue
18       xbeta=xstar;               % optimal point is new start
19       mu=mu/2;                   % decrease the multiplier
20   end                            % end of barrier problem sequence
21 end
```

Now μ is [19] decreased at each iteration, and there is no point in making it smaller than the smallest floating-point value so I chose `kmax` like this.

$$\begin{aligned}
\mu_0 \times (\tfrac{1}{2})^{\texttt{kmax-1}} &\geq \texttt{realmin} \\
\lg(\mu_0) + (\texttt{kmax-1})\lg(\tfrac{1}{2}) &\geq \lg(\texttt{realmin}) \\
(\texttt{kmax-1})(-1) &\geq \lg(\texttt{realmin}) - \lg(1) \\
(\texttt{kmax-1}) &\leq -\lg(\texttt{realmin}) = 1022 \\
\texttt{kmax} &= 1023
\end{aligned}$$

To test `barrier.m` I used it to solve `b1` and `b2`, obtaining the results shown at the top of the next page. Exact solutions were found for both problems, but for neither did `barrier.m` return `rc=0`; this algorithm exhibits the same sort of endgame behavior we observed for `penalty.m`, and for the same reasons (see §18.4). Both problems have convex barrier functions, so the mystery presented by these results is why it was necessary to modify $\mathbf{H}_\beta$ (resulting in $\texttt{nm} > 0$). To investigate this I had `ntin.m` report the first `H` that `hfact.m` found to be numerically non-positive-definite in solving `b1`, and discovered that its second leading principal minor comes out exactly zero (see Exercise 19.6.22). Earlier we observed that as μ decreases, $\mathbf{H}_\beta$ becomes more and more ill-conditioned, and in this case that process culminates in a Hessian that is precisely singular. Using an `epz` value of 10^{-9} rather than 10^{-16} makes the non-positive-definite Hessians go away, which suggests that they are yet another phantom of floating point arithmetic *in extremis.*


```
octave:1> format long
octave:2> [xstar,kp,rc,mu,nm]=barrier('b1',2,[0.5;0.5],1,1e-16)
xstar =

   1.33253925708181e-16
   1.00000000000000e+00

kp =  56
rc =  1
mu =  2.77555756156289e-17
nm =  10
octave:3> [xstar,kp,rc,mu,nm]=barrier('p2',1,[1;2],20,1e-16)
xstar =

   0.945582993415968
   0.894127197437503

kp =  56
rc =  1
mu =  5.55111512312578e-16
nm =  4
octave:4> quit
```

19.4 Comparison of Penalty and Barrier Methods

Although the quadratic penalty method of §18 and the logarithmic barrier method of this Chapter differ significantly in the details of their implementation, they are closely related in underlying philosophy and share many general attributes. Both treat constraints by incorporating them into an objective function and both solve a sequence of unconstrained optimizations, each starting at the optimal point of the previous one, as μ approaches an extreme value. In both algorithms the Hessian of the penalty or barrier objective becomes badly conditioned as that happens, making Newton descent the preferred algorithm for solving the unconstrained problems. Both algorithms exhibit only linear convergence, and the ill-conditioning of the Hessian as the optimal point is approached results in roundoff errors that limit the accuracy that can be attained by either.

The attributes in which the methods differer show a charming symmetry, making it useful to think of the relationship between them as a sort of duality. Here is a comparison of the two particular algorithms we have studied.

quadratic penalty method	logarithmic barrier method
for = constraints	for $\leq$ constraints
$\pi(\mathbf{x};\mu) = f_0(\mathbf{x}) + \mu \sum_{i=1}^m [f_i(\mathbf{x})]^2$	$\beta(\mathbf{x};\mu) = f_0(\mathbf{x}) - \mu \sum_{i=1}^m \ln[-f_i(\mathbf{x})]$
$\mu \to \infty$	$\mu \to 0$
$\mathbf{x}^k$ approach $\mathbf{x}^\star$ from outside of $\mathbb{X}$	$\mathbf{x}^k$ approach $\mathbf{x}^\star$ from inside of $\mathbb{X}$
$\mathbf{x}^0$ and all $\mathbf{x}^k$ infeasible	$\mathbf{x}^0$ and all $\mathbf{x}^k$ feasible
$\lambda_i(\mu) = 2\mu f_i[\mathbf{x}(\mu)]$	$\lambda_i(\mu) = -\mu / f_i[\mathbf{x}(\mu)]$
basis of exact penalty methods §20	basis of interior point methods §21

There are [1, §9.4] variants of the barrier method that use $\beta(\mathbf{x};\mu) = f_0(\mathbf{x}) - \mu \sum_{i=1}^{m}[1/f_i(\mathbf{x})]$ instead of logarithms, variants of the barrier method that can handle equality constraints along with inequalities [1, §9.2], and variants of the penalty method [1, §9.1] [124, p509-510] that can handle inequality constraints along with equalities (see §25.2). The classical penalty and barrier methods that we have glimpsed in §18 and this Chapter are actually part of a single larger subject with a long and complicated history [57]. Rather than exploring that subject in greater breadth, we will take up in §20 and §21 faster and more robust algorithms that are based on the classical methods but avoid their numerical pitfalls.

19.5 Plotting Contours of the Barrier Function

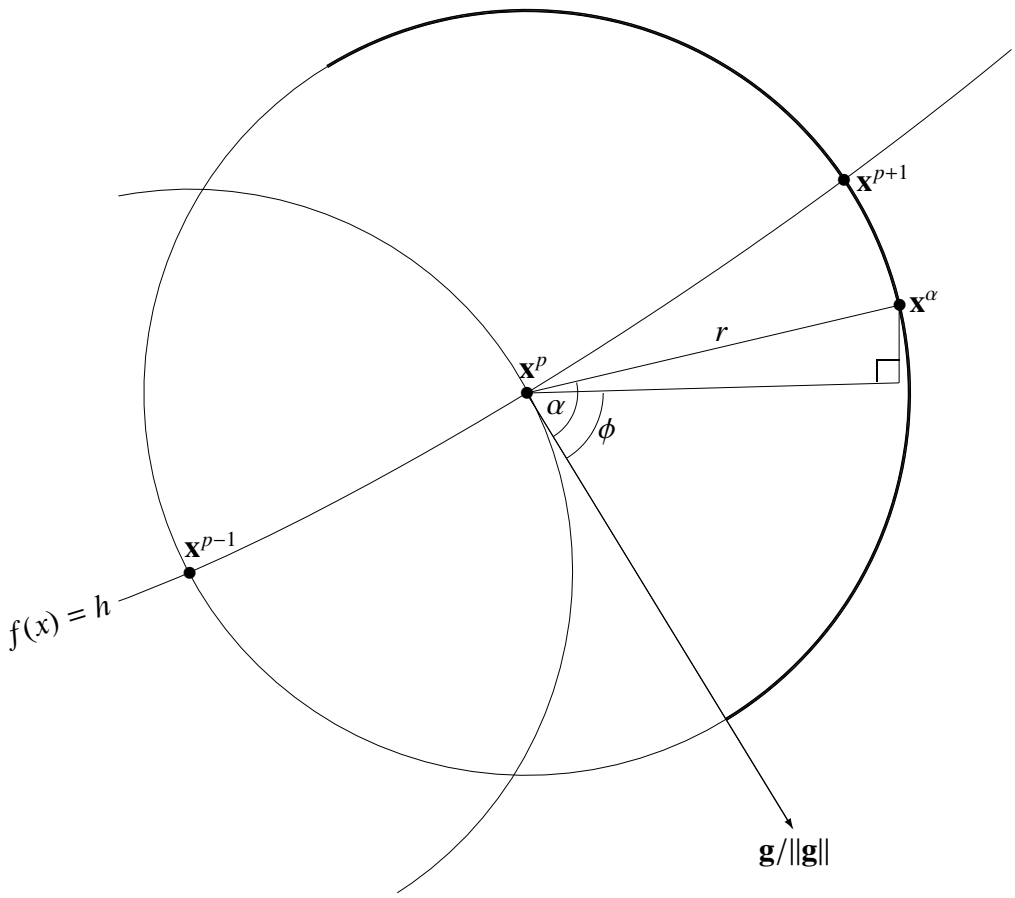
Since §9.1 we have drawn contour diagrams by using `gridcntr.m` to compute values of the function on a rectangular grid of points and then the MATLAB `contour` command to make the picture. The line segments that `contour` plots to approximate each level curve are actually found by the `contourc` command [50, p248] using **grid interpolation**, an algorithm that needs all of the function values on the grid. In the contour diagrams of §19.2 the grid unavoidably includes some points where $\beta(\mathbf{x};\mu)$ is undefined because $\mathbf{x}$ is infeasible, so I had to use a different approach.

Suppose that we have found points $\mathbf{x}^0, \mathbf{x}^1 \ldots \mathbf{x}^p$, each a distance r from the previous one, along the curve where $\beta(\mathbf{x}) = h$. If we draw a circle of radius r centered at $\mathbf{x}^p$ then the curve will cross it at $\mathbf{x}^{p-1}$ and $\mathbf{x}^{p+1}$ as shown in the picture on the next page. To find $\mathbf{x}^{p+1}$ from $\mathbf{x}^p$ we can search the thick semicircle, facing away from $\mathbf{x}^{p-1}$, between the direction of the gradient vector $\mathbf{g} = \nabla f(\mathbf{x})$ (where $\alpha = 0$) and the opposite direction (where $\alpha = \pi$). If the contour were a straight line then $\mathbf{x}^{p+1}$ would be at the center of this arc, but in general we must examine trial points

$$\mathbf{x}^\alpha = \mathbf{x}^p + \begin{bmatrix} r\cos(\alpha - \phi) \\ r\sin(\alpha - \phi) \end{bmatrix} \qquad \text{where} \qquad \phi = -\arctan\left(\frac{g_2}{g_1}\right)$$

in a zero-finding algorithm to determine the α where $f = \beta(\mathbf{x}^\alpha) - h = 0$. Then we can construct a new circle about $\mathbf{x}^{p+1}$ and continue the process. This approach to plotting a contour is called **curve following**. Using the `chkfea.m` routine below to avoid infeasible points, I wrote the `curve.m` routine listed on the next two pages.

```
function [nofea]=chkfea(xp,fcn,m)
% return true if xp is infeasible, false if feasible
  nofea=false;
  for i=1:m
      f=fcn(xp,i);
      if(f >= 0)
        nofea=true;
        return
      end
  end
end
```


```
function [h,rc,npt]=curve(name,mineq,muin,xstart,r,mxpt,dir)
% draw a single beta contour containing xstart

  global prob m mu    % prepare to use bta() and btag()
  prob=name;          % by filling in
  m=mineq;            % the global
  mu=muin;            % variables

  xp=xstart;          % start drawing a contour at xstart
  fcn=str2func(prob);       % pointer to function routine
  nofea=chkfea(xp,fcn,m);  % starting point feasible?
  if(nofea)                 % if not then
     h=realmax;       % beta=infinity
     rc=8;            % signal failure
     npt=0;           % without drawing any points
     return           % and resign
  end                 % starting point is feasible
  h=bta(xp);          % it is on this contour
  xc=zeros(1,mxpt);   % initialize x coordinates of contour
  yc=zeros(1,mxpt);   % initialize y coordinates of contour
  left=0;             % the first search spans from 0
  right=dir*pi;       % to +180 degrees or -180 degrees
  tol=1e-6;           % set tolerance for finding the curve
  rc=0;               % assume we will succeed in drawing the contour
  closed=false;       % assume the contour will not be closed
```



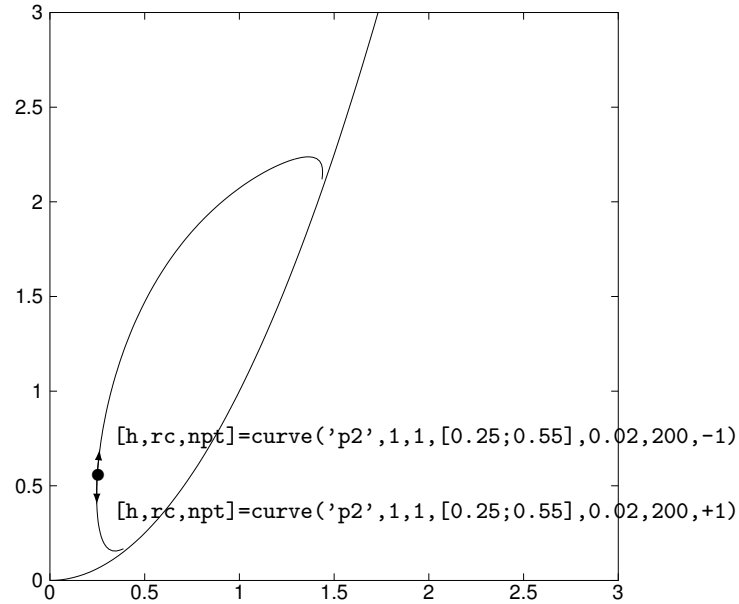
```
  for p=1:mxpt        % find points on contour
      nozro=false;    % assume we will find this point
      nofea=false;    % assume the point will be feasible
      xc(p)=xp(1);    % x-coordinate to plot
      yc(p)=xp(2);    % y-coordinate to plot
      npt=p;          % number of points successfully found
      if(p > 2)                   % if far enough from start
         if(norm(xp-xstart) < r)  % check whether we have returned there
           closed=true;           % if so we have plotted a closed curve
           break                  % so this contour is done
         end          % otherwise we can continue
      end             % done checking for a closed curve
      g=btag(xp);                          % gradient at current point
      phi=-sign(g(2))*atan2(g(2),g(1));    % angle it is above x(1) axis
      al=left;                             % search from this angle
      xl=xp+[r*cos(al-phi);r*sin(al-phi)]; % which yields this point
      nofea=chkfea(xl,fcn,m);              % is it feasible?
      if(nofea) break; end                 % if not give up
      fl=bta(xl)-h;                        % else it has this bta error
      ar=right;                            % search to this angle
      xr=xp+[r*cos(ar-phi);r*sin(ar-phi)]; % which yields this point
      nofea=chkfea(xr,fcn,m);              % is it feasible?
      if(nofea) break; end                 % if not give up
      fr=bta(xr)-h;                        % else it has this bta error
      for t=1:52                           % do up to 52 bisections
          alpha=(al+ar)/2;                         % try the midpoint angle
          xa=xp+[r*cos(alpha-phi);r*sin(alpha-phi)]; % point at new angle
          nofea=chkfea(xa,fcn,m);                  % is it feasible?
          if(nofea) break; end             % if not give up
          if(norm(xr-xl) < tol)            % close enough?
             xp=xa;                        % yes; this is the root
             break                         % save it to plot
          end                              % done testing convergence
          f=bta(xa)-h;      % not done; find bta error at new root guess
          if(f*fl < 0)      % sign change from left to center?
             ar=alpha;      % yes
             xr=xa;         % move right end of interval to center
             fr=f;          % update that function value
             continue       % and keep bisecting
          end               % done testing
          if(f*fr < 0)      % sign change from center to right end?
             al=alpha;      % yes
             xl=xa;         % move left end of interval to center
             fl=f;          % update that function value
             continue       % and keep bisecting
          end               % done testing
          nozro=true        % no sign change; declare failure
          break             % and give up
      end                           % done accumulating points on contour
      if(nofea || nozro) break; end  % if no root was found contour is done
      left=alpha-pi/2;               % otherwise next search interval
      right=alpha+pi/2;     % is semicircle centered on this angle
  end                       % this contour is finished

  plot(xc(1:npt),yc(1:npt)) % plot the curve and report what happened
  if(nozro)  rc=rc+1; end   % bisection failed
  if(nofea)  rc=rc+2; end   % contour encountered boundary of S
  if(closed) rc=rc+4; end   % contour is a closed curve
end
```


The routine begins 4-7 by giving values to the global variables `prob`, `m`, and `mu` so that we can compute $\beta(\mathbf{x};\mu)$ and its gradient. Next 9-17 it checks the starting point of the contour for feasibility, 18 finds the contour level `h` at that point, and 19-25 does some initializations. The variable `dir`, which is 1 an input parameter, is $+1$ or -1 to indicate the direction in which the contour is to be traced.

Next 27-79 up to `mxpt` points $\mathbf{x}^p$ are found on the contour. The coordinates of the current point (for `p=1` the starting point) are saved 30-31 for plotting later. If the point we just found is not the first or second but it is back where we began 33-38 then the curve must be closed so 36 the contour is finished. Otherwise 39-40 we find the gradient of the function and, using the formula given above, its angle ϕ below the horizontal. Then bisection (see §28.3.1) is used 41-75 to find the angle α where 60 $f = \beta(\mathbf{x}^\alpha) - h = 0$. The range of angles bracketing the curve, initially 21-22 `[left,right]` $= [0, \pm 180°]$, is used to set the starting limits `al` 41 and `ar` 46 of the bisection search. The point on the circle at each of these angles is 42,47 found and 43-44,48-49 checked for feasibility. If the endpoints are feasible the function error is found 45,50 at each. Then 51-75 the interval is bisected up to 52 times. Each iteration begins by 52 finding the midpoint of the angle interval, 53 finding the corresponding point on the circle, and 54-55 checking it for feasibility. If convergence is achieved 56-59 the point is 57 accepted. Otherwise one half 61-66 or the other 67-72 of the angle interval is discarded if the other half contains the root, and the `t` loop continues. If the sign of the function error does not change over either interval there is no root, so 73 we declare failure and 74,76 end the contour. If the bisection process succeeds in finding this point on the contour, the angle interval to search for the next point is 77-78 set to the semicircle straddling the angle α of the current point. Thus the search-interval determination described and pictured earlier is actually used only for the first point.

When all of the points that are going to be found have been found, the curve is 81 plotted as a sequence of `npt` line segments. Finally 82-84 `rc` is set to tell the caller what happened. If `rc=0` on return, `npt = mxpt` points were found and plotted; if `rc=4` the contour was a closed curve so probably `npt < mxpt`. The other return codes indicate that a boundary of the feasible set was encountered or that the algorithm failed. The value of r determines how close to the boundary a contour can be drawn, and how sharp a turn in the contour the algorithm can follow, so to get an accurate picture it might be necessary to use a small radius and to allow a correspondingly large number of points. Using more points increases the work performed by the routine and thus the CPU time required to draw the contour.

The graph on the next page shows one contour in the $\mu = 1$ picture of §19.2, which was drawn using two `curve.m` invocations. Each uses `xstart` $= [0.25, 0.55]^\top$, which is marked by a dot • in the picture. The top invocation, using `dir=-1`, follows the curve in the clockwise direction from that point to the boundary of the feasible set, while the bottom invocation using `dir=+1` follows the curve in the counterclockwise direction from `xstart` to the boundary (I added the arrows). Each invocation of `curve.m` returned the contour level `h=10.8198712385442`, `rc=2` because the curve stopped at a boundary of the feasible set, and `npt=25` showing that fewer points were necessary than the `mxpt=200` that were allowed.



19.6 Exercises

19.6.1 [E] If the barrier problem corresponding to a certain nonlinear program is

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x}) - \mu \sum_{i=1}^{m} \ln[-f_i(\mathbf{x})]$$

write down the nonlinear program.

19.6.2 [E] For what values of $\mathbf{x}$ is the logarithmic barrier function defined if $\mu > 0$? What is the logarithmic barrier function if $\mu = 0$? If μ has a high value, what is likely to be true of a point $\mathbf{x}^\beta$ that minimizes the barrier function?

19.6.3 [E] In using the logarithmic barrier method, what must happen to μ in order for $\mathbf{x}^\beta$ to approach $\mathbf{x}^\star$?

19.6.4 [P] Consider the following nonlinear program, which is an inequality-constrained version of problem p1.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -x_1 x_2 & \\ \text{subject to} & x_1 + 2x_2 - 4 & \leq\ 0 \end{array}$$

(a) Write the corresponding barrier function $\beta(\mathbf{x};\mu)$ and minimize it analytically to obtain formulas for x_1^β and x_2^β as functions of μ. (b) Show that β has a local minimum that approaches $\mathbf{x}^\star = [2,1]^\top$ as $\mu \to 0$. (c) Show that β has another stationary point that approaches $[0,0]^\top$ as $\mu \to 0$, and classify it. (d) Starting from $[1,1]^\top$, use ntin.m to minimize β numerically as you reduce μ. (e) Starting from $[0,0]^\top$, use ntin.m to minimize β numerically as you reduce μ. (f) Can every inequality-constrained nonlinear program be solved by the barrier algorithm? The discussion in [4, p610] sheds some light on this question.

19.6.5 [P] If a nonlinear program in standard form is solved using the logarithmic barrier method, the KKT multiplier $\lambda_i^\star$ associated with constraint i can be approximated at each value of μ by a simple formula. (a) Write down the formula. (b) Use the `mults.m` program of §16.10 to find the KKT multiplier corresponding to the catalog $\mathbf{x}^\star$ for problem `b2`. (c) Confirm that, in the limit as $\mu \to 0$, the formula for $\lambda(\mu)$ produces that value. (d) Use `mults.m` to show that the point $\bar{\mathbf{x}}$ where our naïve solution of `b2` in §19.2 jammed, is *not* a KKT point.

19.6.6 [E] Show that under suitable conditions the solution to a barrier problem approaches the solution of the KKT conditions for the corresponding nonlinear program.

19.6.7 [E] When is a logarithmic barrier function convex?

19.6.8 [H] The logarithmic barrier function involves a sum of logarithms. (a) Prove that a nondecreasing concave function of a concave function is concave. (b) Prove that the logarithm is a nondecreasing concave function.

19.6.9 [E] Show that if $y = \ln[-f(x)]$ then $dy/dx = +[1/f(x)]\, df/dx$. What happened to the minus sign?

19.6.10 [H] Derive formulas for the gradient and Hessian of the barrier function corresponding to a standard-form nonlinear program. For what values of $\mathbf{x}$ are these quantities defined?

19.6.11 [E] Why are general-purpose unconstrained minimization routines likely to fail when solving a barrier problem? What must be true of an unconstrained minimization routine in order for it to succeed in solving a barrier problem? Explain how `ntfeas.m` works.

19.6.12 [H] For $\ln(-1)$ MATLAB returns `log(-1)=0.00000+3.14159i`. (a) Explain where this result comes from. How can a logarithm be complex? (b) Are complex numbers meaningful in the optimization models we study in this book? (c) If complex numbers are produced in the course of a calculation but the end result is real, does MATLAB give any indication? Is such an end result useful in solving an optimization problem?

19.6.13 [E] Our example problem `p2`, which is an equality-constrained nonlinear program, is defined by the MATLAB routines `p2.m`, `p2g.m`, and `p2h.m`. (a) How can these same routines be used to define the example problem `b2`, in which the constraints are inequalities? (b) If the functions that define these two problems are the same, why is it that `b2` is a convex program while `p2` is not?

19.6.14 [E] What is *jamming*? How can it be prevented in minimizing $\beta(\mathbf{x};\mu)$?

19.6.15 [E] Explain in detail how `ntfeas.m` fails to solve problem `b2` when `mu=1e-16`.

19.6.16 [E] Explain the basic idea of the barrier algorithm. What is its order of convergence? What happens to $\mathbf{H}_\beta(\mathbf{x};\mu)$ as μ decreases?

19.6.17 [P] In §19.3 the MATLAB routine `hfact.m` is introduced. (a) What does it do? (b) Its return code `rc` can be `0`, `1`, or `2`. What do these return codes mean? (c) If `hfact.m` is invoked with `gama=1`, what happens if `H` is positive definite? (d) Use MATLAB to confirm that if

$$\mathbf{H} = \begin{bmatrix} 10 & 5 & 0 \\ 5 & 15 & 5 \\ 0 & 5 & 2 \end{bmatrix} \quad \text{and} \quad \gamma = \tfrac{1}{2}$$

the `U` returned by `hfact.m` is indeed a Cholesky factor of the matrix as modified.

19.6.18 [P] Revise the following routines to use `hfact.m` rather than the `chol()` command to factor the Hessian: (a) `nt.m` (§13.3.1); (b) `ntw.m` (§13.3.2); (c) `ntfs.m` (§13.2); (d) `ntrs.m` (§17.2).

19.6.19 [E] The MATLAB routine `ntin.m` is described in its title as implementing an interior-point modified Newton algorithm. (a) What makes it an interior-point algorithm? (b) What makes it a modified Newton algorithm?

19.6.20 [E] If $\mu_0 = 1$ and $\mu_k = \mu_{k-1}/2$, what is the maximum value of `k` that we need to consider if we are computing with 8-byte floating-point numbers (which MATLAB uses by default)? Why?

19.6.21 [P] In `barrier.m`, I chose `kmax=1023` based on the assumption that $\mu_0 = 1$, but then I made `muzero` an input parameter so that it can be given a *lower* value. If the routine is invoked with `muzero` set to a *higher* value than 1, `kp` should be allowed to get higher than `1023`. Modify the code to calculate `kmax` from `muzero`, but don't let `kmax` exceed the highest value allowed for a MATLAB loop limit (see §4.1).

19.6.22 [P] When `barrier.m` is used to solve a problem with `epz` set too small, $\mathbf{H}_\beta$ typically becomes numerically non-positive-definite near the end of the solution process, so that `nm` > 0 is reported, even if the original problem is a convex program. (a) Modify `ntin.m` to report the first `H` that `hfact.m` finds to be non-positive definite. (b) Repeat the solution of `b1` by `barrier.m` reported in §19.3, and show that the first non-positive-definite `H` has its second leading principal minor equal to zero as claimed. (c) Hessians that are numerically non-positive-definite are also encountered by `barrier.m` in solving `b2`. Repeat the experiment reported in §19.3 and show that the first non-positive-definite `H` has its second leading principal minor *negative.* How can that happen?

19.6.23 [P] By construction, the logarithmic barrier function $\beta(\mathbf{x};\mu)$ has its minimizing point (or points) strictly inside the feasible set. If $\beta(\mathbf{x};\mu)$ can be accurately approximated by a quadratic, then full Newton steps should remain inside the feasible set and it might not be necessary to guard against generating infeasible points. (a) Construct the quadratic function $q(\mathbf{x})$ that Newton descent uses to model $\beta(\mathbf{x}^0;\frac{1}{2})$ for the `b2` problem, and show that the first Newton step based on it is to an infeasible point. (b) Plot contours of $q(\mathbf{x})$ and $\beta(\mathbf{x};\frac{1}{2})$ to illustrate the mismatch between the model and the function.

19.6.24 [P] Consider the following problem [5, Example 19.1]

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & (x_1 + \frac{1}{2})^2 + (x_2 - \frac{1}{2})^2 \\ \text{subject to} & x_1 \in [0, 1] \\ & x_2 \in [0, 1] \end{array}$$

(a) Solve the problem graphically. (b) Use `barrier.m` to solve the problem numerically.

19.6.25 [P] Consider the following problem [1, Exercise 9.18].

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & (x_1 - 5)^2 + (x_2 - 3)^2 & & \\ \text{subject to} & 2x_1 + 2x_2 & \leq & 6 \\ & -4x_1 + 2x_2 & \leq & 4 \end{array}$$

(a) Solve the problem graphically. (b) Use `barrier.m` to solve the problem numerically.

19.6.26 [P] Use `barrier.m` to solve the following inequality-constrained nonlinear programs: (a) the `arch2` problem of §16.0; (b) the `arch4` problem of §16.2; (c) the `moon` problem of §16.3; (d) the `cq1` problem of §16.7; (e) the `cq3` problem of §16.7; (f) the problem of Exercise 16.11.21.

19.6.27 [P] We solved the `b1` problem numerically in §19.3, and in §19.0 we plotted points representing its analytic solution $\mathbf{x}^\beta(\mu)$ for a few values of μ. (a) Modify `barrier.m` so that it can be used to solve a problem one iteration at a time. (b) Write a program that uses your modified `barrier.m` to solve the `b1` problem one iteration at a time starting from $\mathbf{x}^0 = [\frac{1}{2}, \frac{1}{2}]^\mathsf{T}$, and plot its convergence trajectory along with the zero contours of its constraints. (c) How is this convergence trajectory related to the points we plotted from the analytic solution? (d) Use the `curve.m` contour plotter of §19.5 to add contours of $\beta(\mathbf{x}; \mu)$ to your convergence trajectory plot.

19.6.28 [E] Write down all the ways you can think of in which barrier and penalty methods differ. Write down all the ways you can think of in which they are similar.

19.6.29 [E] Explain how MATLAB can be used to plot the contours of a function by using the grid interpolation algorithm. Is it ever impossible to use this approach? Explain.

19.6.30 [E] Describe the basic idea of the curve-following algorithm for plotting a contour. What are the advantages and drawbacks of this approach?

19.6.31 [E] What does `chkfea.m` return? What role does it play in the `curve.m` routine?

19.6.32 [E] Answer the following questions about the `curve.m` routine. (a) How does the user select the function value of the contour to be plotted? (b) What does the input parameter `dir` control? (c) How does the routine know if the contour is a closed curve? (d) What determines how close a contour can be drawn to a boundary of the feasible set? (e) What happens if `mxpt` is set too low? Too high?

19.6.33 [P] Modify the `b2bar.m` program of §19.2 to plot the `b2` objective contours by using `curve.m` rather than `gridcntr.m` and the MATLAB `contour` command. Which approach is easier?

19.6.34 [P] The `curve.m` routine draws a single contour of the function $\beta(\mathbf{x};\mu)$ for a given value of μ. Generalize it to plot a single contour of an arbitrary function $f(\mathbf{x})$. How can you use the new routine to plot a contour of $\beta(\mathbf{x};\mu)$?

20

Exact Penalty Methods

When the classical penalty method of §18 works at all it converges only linearly, and it has limited accuracy because $\mathbf{H}_\pi$ becomes badly conditioned as $\mu \to \infty$ and that degrades the precision with which Newton descent directions can be computed near the optimal point. Although we were able to find $\mathbf{x}^\star$ exactly for the simple demonstration problems we considered, the algorithm is of limited use for the larger and more difficult optimizations that typically arise in practical applications.

In the classical algorithm the exact solution $\mathbf{x}^\pi(\mu)$ to the penalty problem approaches $\mathbf{x}^\star$ only in the limit as $\mu \to \infty$. This drawback has inspired the development of algorithms that can find $\mathbf{x}^\star$ exactly *without* passing to a limit. Instead of minimizing the classical penalty function these methods minimize an **exact penalty function** having $\mathbf{x}^\pi(\mu) = \mathbf{x}^\star$ at a *finite* value of μ.

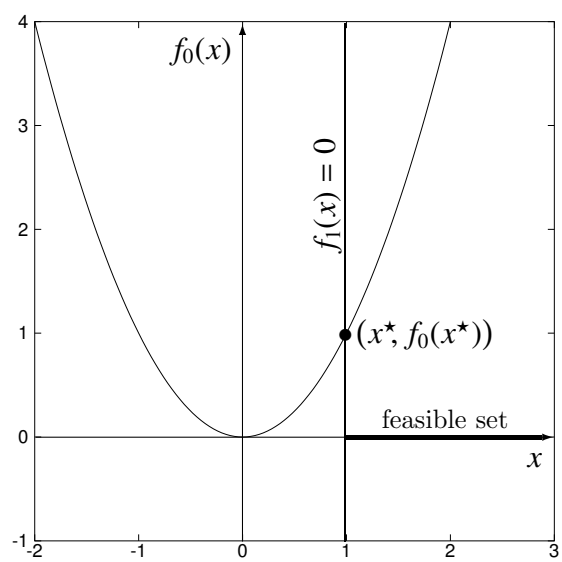
20.1 The Max Penalty Method

To see how it is possible for a penalty function to have the miraculous property of being exact, consider the following inequality-constrained nonlinear program in one dimension, which I will call `ep1` (see §28.7.24).

$$\begin{array}{rrcl} \underset{x\in\mathbb{R}^1}{\text{minimize}} & f_0(x) & = & x^2 \\ \text{subject to} & f_1(x) & = & 1-x \;\le\; 0 \end{array}$$

We can solve this problem using the KKT method, as follows.

$$\begin{array}{rcll} \mathcal{L}(x,\lambda) & = & x^2+\lambda(1-x) & \\ \dfrac{\partial\mathcal{L}}{\partial x} & = & 2x-\lambda & = \;0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda} & = & 1-x & \le\; 0 \\ & & \lambda(1-x) & = \;0 \\ & & \lambda & \ge\; 0 \end{array}$$



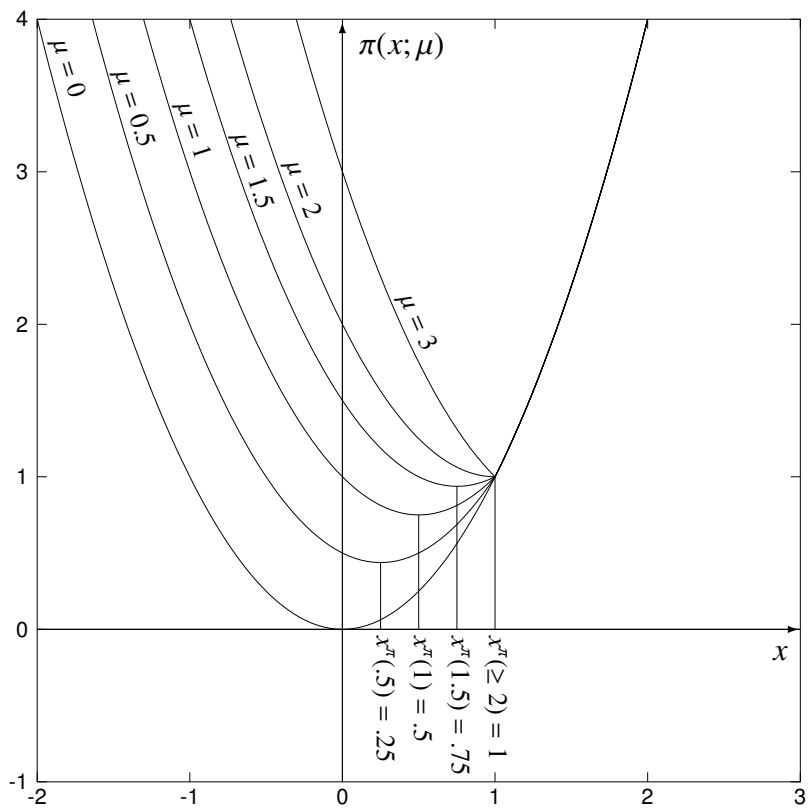
The optimality conditions are satisfied at $x^\star = 1$ with $\lambda^\star = 2$. This problem is related to the following unconstrained minimization [1, §9.3] [5, §17.2] [2, §5.3.1] [4, §16.5] [57, §4.1]:

$$\begin{array}{lrcccl} \underset{x\in\mathbb{R}^1}{\text{minimize}} & \pi(x;\mu) & = & f_0(x) & + & \mu \max[0, f_1(x)] \\ & & = & x^2 & + & \mu \max[0, (1-x)]\,. \end{array}$$

The penalty term

$$\mu \max[0, (1-x)] = \begin{cases} \mu(1-x) & \text{if } x \le 1 \\ 0 & \text{if } x \ge 1 \end{cases}$$

is always nonnegative, but it adds nothing to π unless x is infeasible. We can solve the **max penalty problem** above graphically for given values of μ, as shown below.



If $x \le 1$ then $1 - x \ge 0$ and $f_1(x) \ge 0$, so x^π minimizes $\pi(x;\mu) = x^2 + \mu(1-x)$ where

$$\frac{d\pi}{dx} = 2x - \mu = 0 \quad \Rightarrow \quad x^\pi = \mu/2 \le 1.$$

If $x \ge 1$ then $1 - x \le 0$ and $f_1(x) \le 0$, so $\pi(x;\mu) = x^2$ and x^π solves

$$\left.\begin{array}{ll} \underset{x\in\mathbb{R}^1}{\text{minimize}} & x^2 \\ \text{subject to} & x \ge 1 \end{array}\right\} \quad \Rightarrow \quad x^\pi = 1.$$

Approaching $x^\star = 1$ from below, $x \le 1$ so $x^\pi(\mu) = \mu/2$. When x^π reaches $x^\star$ we have $\mu/2 = 1$ or $\mu = 2$; we will call this **inflection value** $\bar{\mu}$. Notice in the picture that when $\mu = 2$ the curve of $\pi(x; 2)$ has a horizontal tangent among its subgradients, and thus its minimum, at $x^\star$. At $x^\star$ each curve has a left-handed [146, Exercise 2.1.49] slope of $2x^\star - \mu = 2 - \mu$, but the right-handed slope is 2 so only the curve for $\mu = 0$ has a continuous derivative there.

Approaching $x^\star = 1$ from above, $x \ge 1$ so $x^\pi(\mu) = 1$. In other words, for all $\mu \ge \bar{\mu}$

$$\left.\begin{array}{l}\underset{x\in\mathbb{R}^1}{\text{minimize}} \quad \pi(x;\mu) = x^2 + \mu \max\,[0, (1-x)]\end{array}\right\} \quad \text{solves} \quad \left\{\begin{array}{ll}\underset{x\in\mathbb{R}^1}{\text{minimize}} & x^2 \\ \text{subject to} & x \ge 1\end{array}\right.$$

which is `ep1`. If μ is given a finite value that is high enough (in this case at least $\bar{\mu} = 2$) then the solution to the penalty problem is *exactly* the solution of the original nonlinear program. The $\bar{\mu}$ we found for this example is equal to $\lambda^\star = 2$, and it can be shown [5, Theorem 17.3] that in general

$$\bar{\mu} = \max_i |\lambda_i^\star|.$$

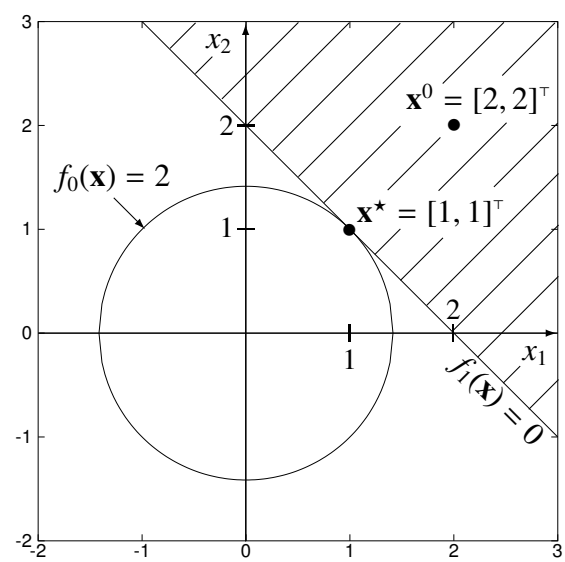
The nonsmoothness of the max penalty function becomes more obvious if we generalize `ep1` to two dimensions, yielding the following problem which I will call `ep2` (see §28.7.25).

$$\begin{array}{lrcl} \underset{x\in\mathbb{R}^2}{\text{minimize}} & f_0(x) & = & x_1^2 + x_2^2 \\ \text{subject to} & f_1(x) & = & 2 - x_1 - x_2 \quad \le \quad 0 \end{array}$$

$$\mathcal{L}(x, \lambda) = x_1^2 + x_2^2 + \lambda(2 - x_1 - x_2)$$

The KKT conditions for this problem are

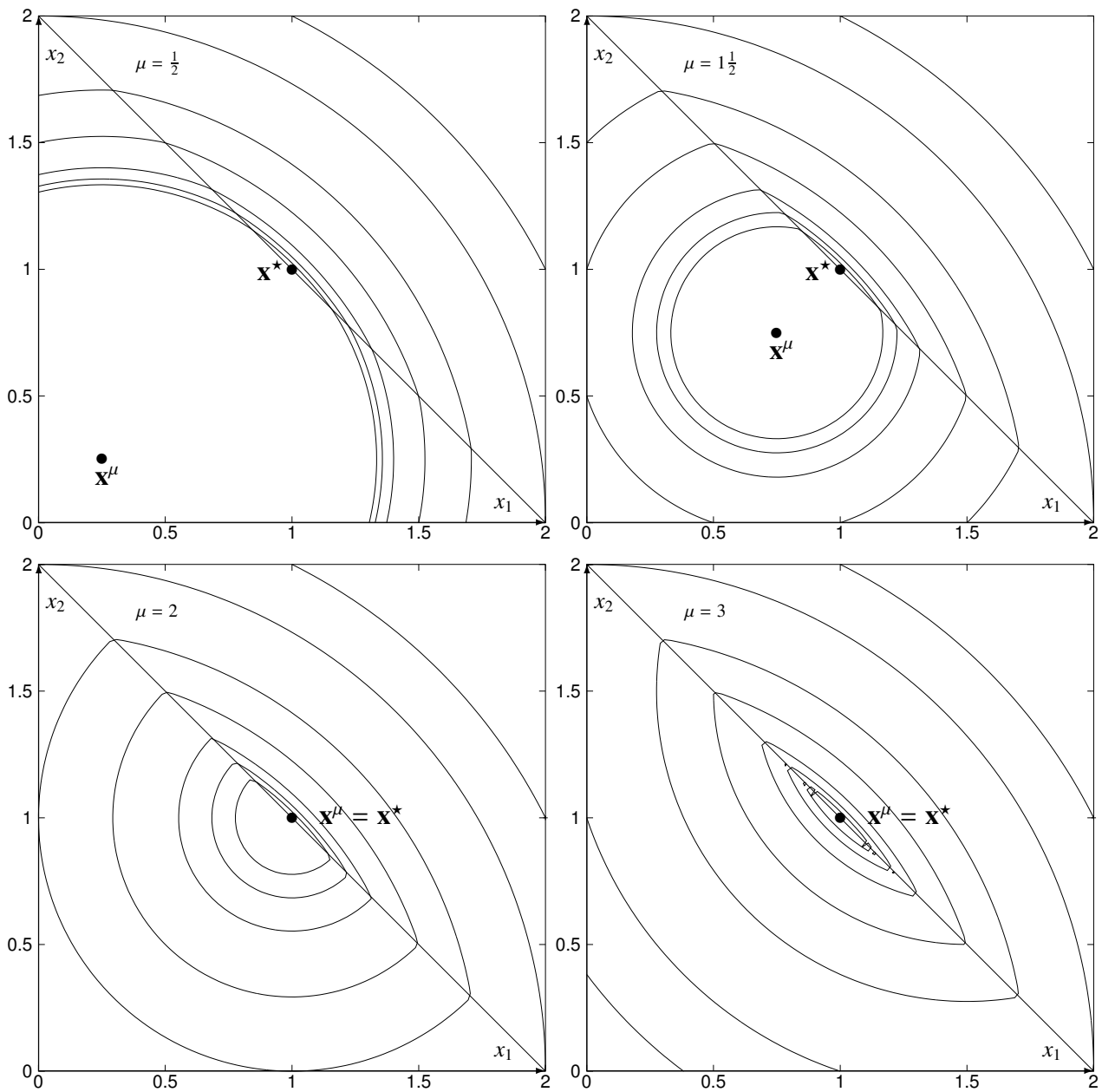
$$\begin{array}{rcll} \dfrac{\partial \mathcal{L}}{\partial x_1} & = & 2x_1 - \lambda & = \; 0 \\ \dfrac{\partial \mathcal{L}}{\partial x_2} & = & 2x_2 - \lambda & = \; 0 \\ \dfrac{\partial \mathcal{L}}{\partial \lambda} & = & 2 - x_1 - x_2 & \le \; 0 \\ & & \lambda(2 - x_1 - x_2) & = \; 0 \\ & & \lambda & \ge \; 0 \end{array}$$



and they are satisfied at $\mathbf{x}^\star = [1, 1]^\top$ with $\lambda^\star = 2$. The corresponding max penalty function is

$$\pi(\mathbf{x}; \mu) = x_1^2 + x_2^2 + \mu \max\,[0, (2 - x_1 - x_2)]$$

whose contours are plotted for several values of μ on the next page. Each graph shows the same set of contours for $\pi(\mathbf{x}; \mu)$, which have cusps where they meet the constraint contour $f_1(\mathbf{x}) = 0$. At these cusps (i.e., at every point on the constraint contour) $\nabla\pi(\mathbf{x})$ is discontinuous.



When $\mathbf{x}$ is infeasible $\pi(\mathbf{x};\mu) = x_1^2 + x_2^2 + \mu(2 - x_1 - x_2)$ and this looks like $\mathcal{L}(\mathbf{x};\lambda)$ so $\bar{\mu} = \lambda^\star = 2$. We find analytically, by reasoning as we did for `ep1`, that

$$\mathbf{x}^\pi(\mu) = \begin{cases} [\mu/2, \mu/2]^\top & \mu \le \bar{\mu} \\ [1, 1]^\top & \mu \ge \bar{\mu} \end{cases}$$

and this is confirmed by the graphs. In the bottom two panels, where $\mu \ge \bar{\mu}$, the contours change shape as μ increases but the minimizing point of $\pi(\mathbf{x};\mu)$ is always $\mathbf{x}^\star = [1, 1]^\top$.

To compute the value, gradient, and Hessian of the max penalty function I wrote the MATLAB routines listed here.

```
function f=epy(x)
  global prob m mu
  fcn=str2func(prob);
  f=fcn(x,0);
  for i=1:m
    if(fcn(x,i) > 0)
       f=f+mu*fcn(x,i);
    end
  end
end
```

```
function g=epyg(x)
  global prob m mu
  fcn=str2func(prob);
  grd=str2func([prob,'g']);
  g=grd(x,0);
  for i=1:m
    if(fcn(x,i) > 0)
      g=g+mu*grd(x,i);
    end
  end
end
```

```
function H=epyh(x)
  global prob m mu
  fcn=str2func(prob);
  hsn=str2func([prob,'h']);
  H=hsn(x,0);
  for i=1:m
    if(fcn(x,i) > 0)
      H=H+mu*hsn(x,i);
    end
  end
end
```

These resemble the `pye.m`, `pyeg.m`, and `pyeh.m` routines of §18.1, and assume as they do that the MATLAB functions specifying the original nonlinear program are coded in the standard way described in §15.5. These routines compute the function, gradient, and Hessian for `ep2` in that way.

```
function f=ep2(x,i)
  switch(i)
    case 0
      f=x(1)^2+x(2)^2;
    case 1
      f=2-x(1)-x(2);
    end
end
```

```
function g=ep2g(x,i)
  switch(i)
    case 0
      g=[2*x(1);2*x(2)];
    case 1
      g=[-1;-1];
    end
end
```

```
function H=ep2h(x,i)
  switch(i)
    case 0
      H=[2,0;0,2];
    case 1
      H=[0,0;0,0];
    end
end
```

Using the six routines listed above I tried to solve `ep2` with `ntfs.m`, producing the results shown at the top of the next page. In each experiment the routine returned `nm=0`, so it used full-step Newton descent.

For $\mu \le 2$ `1>-4>` the algorithm finds $\mathbf{x}^\pi(\mu) = [\mu/2, \mu/2]^\top$ as expected. Because the penalty function has its minimum at points $\mathbf{x}^\mu$ that are infeasible, all of the $\mathbf{x}^k$ except $\mathbf{x}^0$ fall on that side of the constraint and $\pi(\mathbf{x};\mu) = x_1^2 + x_2^2 + \mu(2 - x_1 - x_2)$ for every step in the solution process except the first. (The first step minimizes $\pi(\mathbf{x};\mu) = x_1^2 + x_2^2$, essentially resetting the starting point to the origin.)

However, for $\mu = 3$ `5>-8>` the algorithm bounces back and forth between $\bar{\mathbf{x}}(\mu) = [\mu/2, \mu/2]^\top$ and $\hat{\mathbf{x}}(\mu) = [0, 0]^\top$ and never converges. At $\mathbf{x}^k = [1.5, 1.5]^\top$ the constraint is satisfied, so $\pi(\mathbf{x};\mu) = x_1^2 + x_2^2$ has its minimum at $[0, 0]^\top$ and the algorithm moves there; at $\mathbf{x}^{k+1} = [0, 0]^\top$ the constraint is violated, so $\pi(\mathbf{x};\mu) = x_1^2 + x_2^2 + 3(2 - x_1 - x_2)$ has its minimum at $[1.5, 1.5]^\top$ and the algorithm moves there; this process repeats until the iteration limit is met. For $\mu > 2$ the penalty function is minimized on precisely the zero contour of the constraint, so Newton descent generates iterates on both sides, the formula for $\pi(\mathbf{x};\mu)$ *changes* during the solution process, and the quadratic model

$$q(\mathbf{x}) = \pi(\mathbf{x}^k) + \nabla\pi(\mathbf{x}^k)^\top(\mathbf{x} - \mathbf{x}^k) + \tfrac{1}{2}(\mathbf{x} - \mathbf{x}^k)^\top \mathbf{H}_\pi(\mathbf{x}^k)(\mathbf{x} - \mathbf{x}^k)$$


```
octave:1> global prob='ep2' m=1 mu=1
octave:2> [xpi,kp,nm]=ntfs([2;2],10,1e-6,@epyg,@epyh,0.5)
xpi =

   0.50000
   0.50000

kp =  3
nm = 0
octave:3> mu=2;
octave:4> [xpi,kp,nm]=ntfs([2;2],10,1e-6,@epyg,@epyh,0.5)
xpi =

   1.00000
   1.00000

kp =  3
nm = 0
octave:5> mu=3;
octave:6> [xbar,kp,nm]=ntfs([2;2],10,1e-6,@epyg,@epyh,0.5)
xbar =

   1.5000
   1.5000

kp =  10
nm = 0
octave:7> [xhat,kp,nm]=ntfs([2;2],11,1e-6,@epyg,@epyh,0.5)
xhat =

   4.4409e-16
   4.4409e-16

kp =  11
nm = 0
octave:8> [xbar,kp,nm]=ntfs([2;2],12,1e-6,@epyg,@epyh,0.5)
xbar =

   1.5000
   1.5000

kp =  12
nm = 0
octave:9> quit
```

that is assumed by Newton descent is a *different function* from one iteration to the next.

$$\begin{array}{rll} \text{At } \mathbf{x}^k = [\tfrac{3}{2}, \tfrac{3}{2}]^\top & \bar{q}(\mathbf{x}) \equiv x_1^2 + x_2^2 & \nabla\bar{q}(\mathbf{x}) = [2x_1, 2x_2]^\top \\ \text{but at } \mathbf{x}^{k+1} = [0,0]^\top & \hat{q}(\mathbf{x}) \equiv x_1^2 + x_2^2 + \mu(2 - x_1 - x_2) & \nabla\hat{q}(\mathbf{x}) = [2x_1 - \mu, 2x_2 - \mu]^\top. \end{array}$$

Because the gradient $\nabla\pi(\mathbf{x};\mu)$ of the max penalty function for `ep2` is discontinuous, the gradients $\nabla\bar{q}(\mathbf{x})$ and $\nabla\hat{q}(\mathbf{x})$, which are actually used by Newton descent, differ unless $\mu = 0$.

Newton descent assumes [5, Theorem 3.5] [4, Theorem 2.6] that the function being minimized will have continuous first and second derivatives at every $\mathbf{x}^k$, including $\mathbf{x}^\star$. At $\mathbf{x}^\star$ the max penalty function for `ep2` does have continuous second derivatives, with

$$\mathbf{H}_\pi(\mathbf{x}) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

but its first derivatives are discontinuous so it is not surprising that `ntfs.m` is unable to minimize it for $\mu > \bar{\mu}$ [4, p625]. Using a bisection line search rather than full steps results in an implementation of Newton descent that is somewhat more robust against discontinuities in the gradient. Here `nt.m` solves `ep2`, producing the expected results for μ lower than $\bar{\mu}$, equal to $\bar{\mu}$, slightly higher than $\bar{\mu}$, and much higher than $\bar{\mu}$.

```
octave:1> global prob='ep2' m=1 mu=1
octave:2> [xstar,kp,nm,rc]=nt([2;2],[-2;-2],[3;3],100,1e-16,@epyg,@epyh,0.5)
xstar =

   0.50000
   0.50000

kp =  2
nm = 0
rc = 0
octave:3> mu=2
mu =  2
octave:4> [xstar,kp,nm,rc]=nt([2;2],[-2;-2],[3;3],100,1e-16,@epyg,@epyh,0.5)
xstar =

   0.98828
   0.98828

kp =  100
nm = 0
rc =  1
octave:5> mu=2.01
mu =  2.0100
octave:6> [xstar,kp,nm,rc]=nt([2;2],[-2;-2],[3;3],100,1e-16,@epyg,@epyh,0.5)
xstar =

   1.00000
   1.00000

kp =  100
nm = 0
rc =  1
octave:7> mu=3
mu =  3
octave:8> [xstar,kp,nm,rc]=nt([2;2],[-2;-2],[3;3],100,1e-16,@epyg,@epyh,0.5)
xstar =

   1.00000
   1.00000

kp =  100
nm = 0
rc =  1
octave:9> quit
```

Alas, `nt.m` fails to minimize the max penalty function for other problems (see Exercise 20.4.10), and the other unconstrained minimizers we have studied enjoy only mixed success

in solving `ep2` (see Exercise 20.4.11) and other problems. **Subgradient optimization methods** [1, §8.9] are designed to minimize a nonsmooth function that is convex (like the max penalty function for `ep2`) but applying one successfully to a particular problem requires fine-tuning of algorithm parameters and careful attention to numerous other implementation details, so the approach is difficult to use in practice and beyond the scope of this text. Of course we could always resort to an algorithm that uses only function values, such as pattern search, but those methods are typically very slow.

Using the trick of §1.5.1 we can instead reformulate the max penalty problem on the left as the smooth optimization on the right.

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ \ f_0(\mathbf{x}) + \mu \sum_{i=1}^{m} \max[0, f_i(\mathbf{x})] \quad \longleftrightarrow \quad \begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n\ \mathbf{t}\in\mathbb{R}^m}{\text{minimize}} & f_0(\mathbf{x}) + \mu \sum_{i=1}^{m} t_i \\ \text{subject to} & t_i \geq 0, \\ & t_i \geq f_i(\mathbf{x}),\ i = 1 \ldots m \end{array}$$

We initially introduced a penalty function to move the constraints into the objective and in this reformulation inequalities reappear, so it might seem that we are back where we began; instead of finding a way to solve an inequality-constrained problem we have just rewritten it as another inequality-constrained problem. However, the new problem is not quite the standard-form nonlinear program we started with. If at each step $\mathbf{x}^k$ of a penalty algorithm that increases μ we [5, p511-513] [2, §5.31] replace the objective in this reformulation by a quadratic approximation to the Lagrangian at $\mathbf{x}^k$ and each constraint by its linear approximation there, we get a subproblem that might be much easier to solve than the original optimization. We will return to this rather complicated idea in §23.2.4, after we have studied algorithms for solving linearly-constrained quadratic programs.

The max penalty method discussed above can be modified to handle equality constraints instead of or in addition to inequalities, by using one of the following (also nonsmooth) penalty terms

$$\mu \sum_{\text{equalities}} |f_i(\mathbf{x})| \qquad [1, §9.3]$$
$$\mu \max_{\text{equalities}} |f_i(\mathbf{x})| \qquad [2, §5.3.1]$$

but instead of investigating that variation we will now turn our attention to a different penalty function which, in addition to being exact, is also smooth.

20.2 The Augmented Lagrangian Method

Consider the equality-constrained nonlinear program on the next page, which I will call `al2` (it resembles [5, Example 17.1]; see §28.7.26). The equality constraint is nonlinear so it is nonconvex, but for $\lambda > 0$ the Lagrangian is a strictly convex function of $\mathbf{x}$.

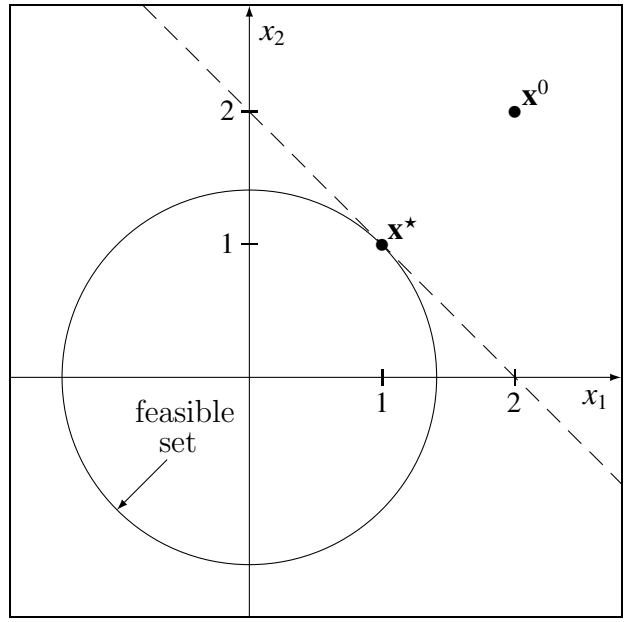
$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(x) & = & -x_1 - x_2 \\ \text{subject to} & f_1(x) & = & x_1^2 + x_2^2 - 2 \;=\; 0 \\ & \mathbf{x}^0 & = & [2,2]^\top \end{array}$$

$$\mathcal{L}(\mathbf{x},\lambda) = -x_1 - x_2 + \lambda(x_1^2 + x_2^2 - 2)$$

The Lagrange conditions for this problem

$$\begin{array}{rcl} \dfrac{\partial\mathcal{L}}{\partial x_1} & = & -1 + 2\lambda x_1 \;=\; 0 \\ \dfrac{\partial\mathcal{L}}{\partial x_2} & = & -1 + 2\lambda x_2 \;=\; 0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda} & = & x_1^2 + x_2^2 - 2 \;=\; 0 \end{array}$$

are satisfied by $x_1^\star = 1$, $x_2^\star = 1$ with $\lambda^\star = \frac{1}{2}$.



20.2.1 Minimizing a Convex Lagrangian

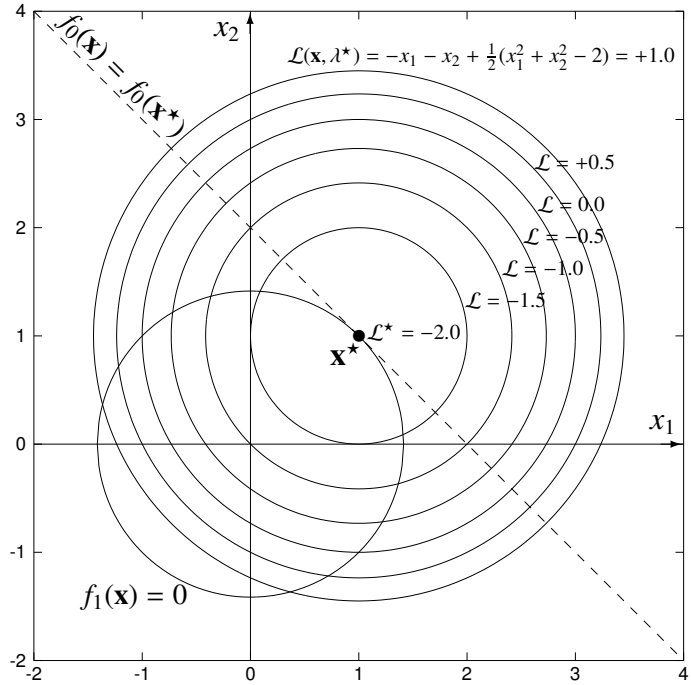
The optimal point $(\mathbf{x}^\star,\lambda^\star) = \left([1,1]^\top, \frac{1}{2}\right)$ of `al2` satisfies $\nabla_x\mathcal{L} = \mathbf{0}$ and $\nabla_\lambda\mathcal{L} = 0$, so it is a stationary point of $\mathcal{L}(\mathbf{x},\lambda)$. Also, $f_1(\mathbf{x}^\star) = 0$ so $\mathcal{L}(\mathbf{x}^\star,\lambda) = f_0(\mathbf{x}^\star)$. Thus we could find $\mathbf{x}^\star$ by solving the nonlinear program on the right below in place of the one on the left [4, §16.6].

$$\begin{array}{rl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) \\ \text{subject to} & f_1(\mathbf{x}) = 0 \end{array} \quad\longleftrightarrow\quad \begin{array}{rl} \underset{\mathbf{x}\in\mathbb{R}^2\ \lambda\in\mathbb{R}^1}{\text{minimize}} & \mathcal{L}(\mathbf{x},\lambda) \\ \text{subject to} & f_1(\mathbf{x}) = 0 \end{array}$$

If somehow we knew ahead of time that $\lambda^\star = \frac{1}{2}$ then we could find $\mathbf{x}^\star$ for `al2` by minimizing $\mathcal{L}(\mathbf{x},\lambda^\star)$ without enforcing the constraint. Because $(\mathbf{x}^\star,\lambda^\star)$ is a stationary point of $\mathcal{L}(\mathbf{x},\lambda)$, it must be that $\nabla_\lambda\mathcal{L}(\mathbf{x}^\star,\lambda^\star) = f_1(\mathbf{x}^\star) = 0$. Thus the $\mathbf{x}^\star$ we find by minimizing $\mathcal{L}(\mathbf{x},\lambda^\star)$ is sure to satisfy the constraint.

$$\begin{array}{rcl} \dfrac{\partial\mathcal{L}(\mathbf{x},\lambda^\star)}{\partial x_1} & = & -1 + 2(\tfrac{1}{2})x_1 \;=\; 0 \;\Rightarrow\; x_1^\star \;=\; 1 \\ \dfrac{\partial\mathcal{L}(\mathbf{x},\lambda^\star)}{\partial x_2} & = & -1 + 2(\tfrac{1}{2})x_2 \;=\; 0 \;\Rightarrow\; x_2^\star \;=\; 1 \\ f_1(\mathbf{x}^\star) & = & 1^2 + 1^2 - 2 \;=\; 0 \;\checkmark \end{array}$$

The picture on the next page shows the graphical solution of the right-hand problem above for $\lambda = \lambda^\star$, from which it is clear that $\mathcal{L}(\mathbf{x},\lambda^\star)$ has a unique minimizing point at $\mathbf{x}^\star$. Because the constraint is satisfied, $\mathcal{L}(\mathbf{x}^\star,\lambda^\star) = f_0(\mathbf{x}^\star)$ and the *unconstrained* minimizing point of $\mathcal{L}(\mathbf{x},\lambda^\star)$ is the same as the *constrained* minimizing point of $f_0(\mathbf{x})$ subject to $f_1(\mathbf{x}) = 0$.



When is it true that given $\boldsymbol{\lambda}^\star$ we can find $\mathbf{x}^\star$ for an equality-constrained NLP by minimizing $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ over $\mathbf{x}$ while ignoring the constraints? It is certainly true if $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ has a unique minimizing point $\mathbf{x}^\star$, because such a point must satisfy $\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\mathbf{x}^\star, \boldsymbol{\lambda}^\star) = \mathbf{0}$ and that means the constraints are satisfied. The Lagrangian certainly has a unique minimizing point if its Hessian matrix is positive definite. That is true for `al2`, which has

$$\begin{matrix} \dfrac{\partial^2 \mathcal{L}}{\partial {x_1}^2} = 2\lambda & \dfrac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} = 0 \\ \dfrac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} = 0 & \dfrac{\partial^2 \mathcal{L}}{\partial {x_2}^2} = 2\lambda \end{matrix} \quad \text{so that at } \lambda^\star = \tfrac{1}{2} \qquad \mathbf{H}_{\mathcal{L}}(\mathbf{x}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

It is true in general that if $\mathbf{H}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ is positive definite and we know $\boldsymbol{\lambda}^\star$, then we can find $\mathbf{x}^\star$ by ignoring the equalities and simply minimizing $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$.

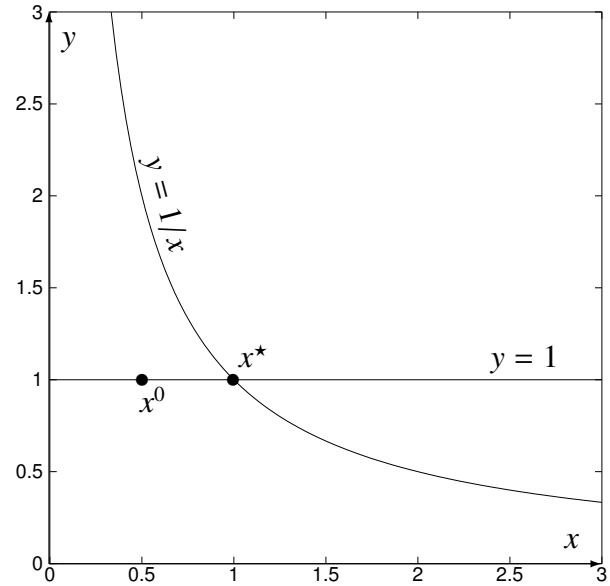
20.2.2 Minimizing a Nonconvex Lagrangian

Now consider the equality-constrained nonlinear program on the next page, which I will call `al1` (see §28.7.27). Notice that it has only one variable and that $x^\star = 1$ is the only feasible point. For $\lambda = \lambda^\star = -1$ its Lagrangian is *not* a convex function of $\mathbf{x}$.

$$\frac{d^2 \mathcal{L}}{dx^2} = \frac{2\lambda}{x^3} \quad \text{so at} \quad \lambda^\star = -1, \quad H_{\mathcal{L}}(x, \lambda^\star) = \left[\frac{-2}{x^3} \right].$$

$$\begin{array}{rrcl} \underset{x\in\mathbb{R}^1}{\text{minimize}} & f_0(x) & = & -x \\ \text{subject to} & f_1(x) & = & \dfrac{1}{x} - 1 \;=\; 0 \\ & x^0 & = & \frac{1}{2} \end{array}$$

$$\mathcal{L}(x,\lambda) = -x + \lambda\left(\frac{1}{x} - 1\right)$$



The Lagrange conditions for this problem

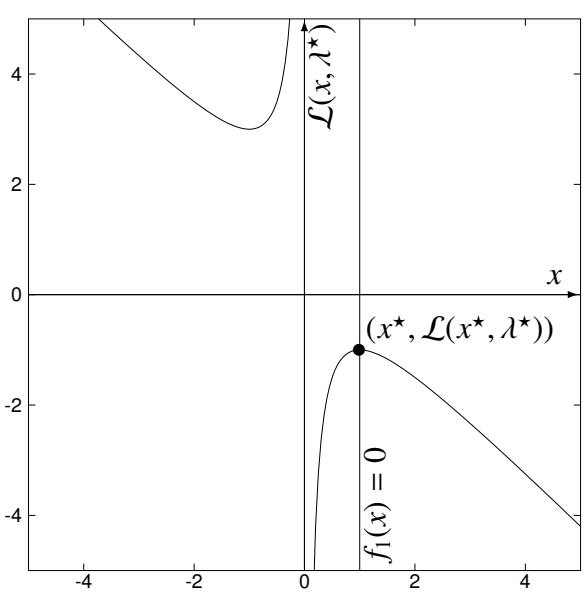
$$\begin{array}{rcl} \dfrac{\partial\mathcal{L}}{\partial x} & = & -1 + \lambda\left(\dfrac{-1}{x^2}\right) \;=\; 0 \\ \dfrac{\partial\mathcal{L}}{\partial\lambda} & = & \dfrac{1}{x} - 1 \;=\; 0 \end{array}$$

are satisfied at $x^\star = 1$ with $\lambda^\star = -1$.

It is still true that we can solve the nonlinear program on the right below in place of the one on the left.

$$\begin{array}{rl} \underset{x\in\mathbb{R}^1}{\text{minimize}} & f_0(x) \\ \text{subject to} & f_1(x) = 0 \end{array} \quad\longleftrightarrow\quad \begin{array}{rl} \underset{x\in\mathbb{R}^1\ \lambda\in\mathbb{R}^1}{\text{minimize}} & \mathcal{L}(x,\lambda) \\ \text{subject to} & f_1(x) = 0 \end{array}$$

Now, however, knowing $\lambda^\star$ ahead of time does not let us ignore the constraint. In the graphical solution of the right-hand problem, shown to the left below, $\mathcal{L}(x,\lambda^\star) = -x - [(1/x) - 1]$ has stationary points at $x = \pm 1$. The constraint requires $x = +1$ so the local minimum at $x = -1$ is infeasible, and $\mathcal{L}$ has no global minimum value because.



$$\lim_{x\to 0^+} \mathcal{L}(x,\lambda^\star) = \lim_{x\to+\infty} \mathcal{L}(x,\lambda^\star) = -\infty.$$

When we enforce the constraint it is the other stationary point of $\mathcal{L}(x,\lambda^\star)$, the local maximum, that turns out to be optimal for the right-hand problem.

This example illustrates that when $\mathcal{L}(\mathbf{x},\lambda^\star)$ is not strictly convex, minimizing it is equivalent to minimizing $f_0(\mathbf{x},\lambda^\star)$ subject to the constraints only if we actually enforce the constraints. Of course it is also still necessary to know $\lambda^\star$.

20.2.3 The Augmented Lagrangian Function

When minimizing $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ one way of enforcing the constraints is to move them into the objective by using a classical penalty term, to form the **augmented Lagrangian penalty function**

$$\pi(\mathbf{x}, \boldsymbol{\lambda}; \mu) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x}) + \mu \sum_{i=1}^{m} [f_i(\mathbf{x})]^2.$$

For `al1`, we find

$$\begin{aligned}
\pi(x, \lambda; \mu) &= -x + \lambda\left(\frac{1}{x} - 1\right) + \mu\left(\frac{1}{x} - 1\right)^2 \\
\frac{d\pi}{dx} &= -1 + \lambda\left(\frac{-1}{x^2}\right) + 2\mu\left(\frac{1}{x} - 1\right)^1\left(\frac{-1}{x^2}\right) \\
\frac{d^2\pi}{dx^2} &= \frac{2\lambda}{x^3} + 2\mu\left[\left(\frac{1}{x} - 1\right)\left(\frac{2}{x^3}\right) + \left(\frac{-1}{x^2}\right)\left(\frac{-1}{x^2}\right)\right] \\
&= \frac{2\lambda}{x^3} + 2\mu\left[\frac{2(1-x)}{x^4} + \frac{1}{x^4}\right] = \frac{2\lambda}{x^3} + 2\mu\left[\frac{3-2x}{x^4}\right].
\end{aligned}$$

If $\lambda = \lambda^\star = -1$ and $x = x^\star = 1$ then

$$\frac{d\pi}{dx} = -1 + (-1)\left(\frac{-1}{1^2}\right) + 2\mu\left(\frac{1}{1} - 1\right)\left(\frac{-1}{1^2}\right) = 0$$

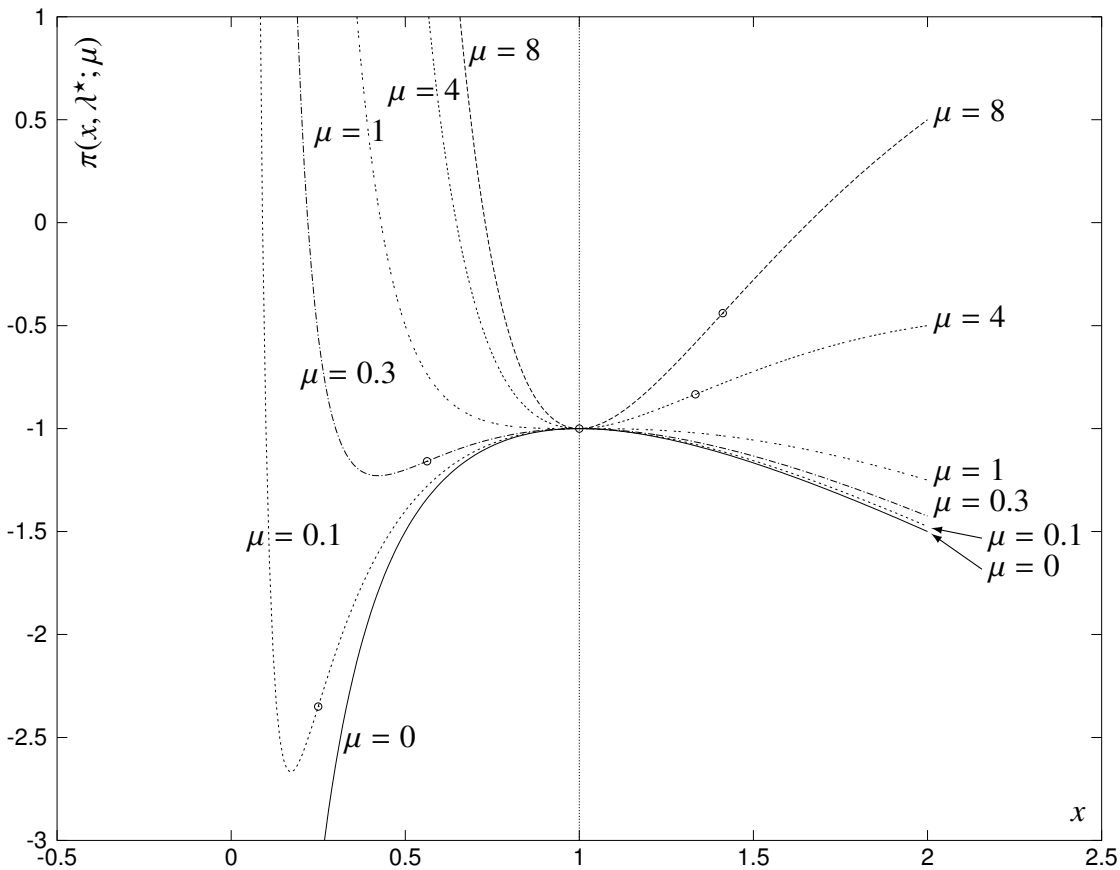
so $(x^\star, \lambda^\star)$ is a stationary point of $\pi(x, \lambda; \mu)$ no matter what value μ has. Whether that point is a minimum, a maximum, or an inflection point of $\pi(x, \lambda; \mu)$ depends on the sign of

$$\frac{d^2\pi}{dx^2} = \frac{2(-1)}{1^3} + 2\mu\left[\frac{3 - 2(1)}{1^4}\right] = -2 + 2\mu.$$

If $\bar{\mu} = 1$ then

$$\begin{array}{lllll}
\mu > \bar{\mu} & \Rightarrow & -2 + 2\mu > 0 & \Rightarrow & (x^\star, \lambda^\star) \text{ is a minimizing point of } \pi; \\
\mu = \bar{\mu} & \Rightarrow & -2 + 2\mu = 0 & \Rightarrow & (x^\star, \lambda^\star) \text{ is an inflection point of } \pi; \\
\mu < \bar{\mu} & \Rightarrow & -2 + 2\mu < 0 & \Rightarrow & (x^\star, \lambda^\star) \text{ is a maximizing point of } \pi.
\end{array}$$

To study the behavior of the augmented Lagrangian for `al1`, I plotted, at the top of the next page, $\pi(x, \lambda^\star; \mu)$ as a function of x for several values of μ. This picture confirms that when $\lambda = \lambda^\star = -1$, $x^\star = 1$ is a minimizing point for $\mu > \bar{\mu} = 1$, an inflection point for $\mu = \bar{\mu}$, and a maximizing point for $\mu < \bar{\mu}$. It is true in general that the augmented Lagrangian is an exact penalty function [1, Theorem 9.3.3] [5, Theorem 17.5] and that it works, as shown in this example, by changing its shape.



Unfortunately, if $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ is not a strictly convex function of $\mathbf{x}$ then $\pi(\mathbf{x}, \boldsymbol{\lambda}^\star; \mu)$ is not necessarily a convex function of $\mathbf{x}$ for *all* $\mathbf{x}$ even if $\mu > \bar{\mu}$. In this example, if $\lambda = \lambda^\star = -1$ then π is convex between $x = 0$ (to which all of the curves shown above are asymptotic) and the value $x = \hat{x}$ at which its Hessian is zero. Using the formula we derived above,

$$\begin{aligned}
\frac{d^2\pi}{dx^2} &= \frac{-2}{x^3} + 2\mu\left[\frac{3-2x}{x^4}\right] &= 0 \\
-2x + 2\mu(3-2x) &= 0 \\
x(-2-4\mu) &= -6\mu \\
\hat{x} &= \frac{6\mu}{2+4\mu} \qquad \lim_{\mu\to\infty} \hat{x} = \frac{6}{4} = \frac{3}{2}.
\end{aligned}$$

In the picture above the inflection point $\hat{x}$ is plotted as an open circle $\circ$ for each value of $\mu > 0$. When $\mu = \bar{\mu} = 1$ the penalty function is convex only between $x = 0$ and $x = \hat{x} = x^\star = 1$; for higher values of μ it is convex between $x = 0$ and $x = \hat{x} < \frac{3}{2}$. This limited region of local convexity makes $\pi(\mathbf{x}, \boldsymbol{\lambda}; \mu)$ hard to minimize even though it is smooth on the interior of its domain.

Rearranging the terms in the formula given above for the augmented Lagrangian reveals that it is just the quadratic penalty function of §18 plus the constraint part of the Lagrangian.

$$\pi(\mathbf{x},\boldsymbol{\lambda};\mu) = \underbrace{f_0(\mathbf{x}) + \mu\sum_{i=1}^{m}[f_i(\mathbf{x})]^2}_{\text{quadratic penalty function}} + \sum_{i=1}^{m}\lambda_i f_i(\mathbf{x})$$

The value, gradient, and Hessian of the augmented Lagrangian are therefore respectively the value, gradient, and Hessian of the quadratic penalty function plus the Lagrange-multiplier-weighted sum of the values, gradients, and Hessians of the constraints, which I coded in the following MATLAB routines.

```
function f=aug(x)
  global prob m mu lambda
  fcn=str2func(prob);
  f=pye(x);
  for i=1:m
      f=f+lambda(i)*fcn(x,i);
  end
end
```

```
function g=augg(x)
  global prob m mu lambda
  grd=str2func([prob,'g']);
  g=pyeg(x);
  for i=1:m
      g=g+lambda(i)*grd(x,i);
  end
end
```

```
function H=augh(x)
  global prob m mu lambda
  hsn=str2func([prob,'h']);
  H=pyeh(x);
  for i=1:m
      H=H+lambda(i)*hsn(x,i);
  end
end
```

The Lagrangian for problem `al2` is strictly convex, so the augmented Lagrangian is also strictly convex even without a penalty term and its minimizing point $\mathbf{x}^\star = [1,1]^\top$ can be found exactly with $\mu = 0$.

```
octave:1> global prob='al2' m=1 mu=0 lambda=0.5
octave:2> [xstar,kp]=ntplain([2;2],20,1e-6,@augg,@augh)
xstar =

   1
   1

kp =  2
octave:3> quit
```

The Lagrangian for problem `al1` is not convex, but the augmented Lagrangian is locally convex over an interval that depends on μ. That interval includes $x^0 = \frac{1}{2}$, and for $\mu > \bar{\mu} = 1$ it also includes $x^\star = 1$.

```
octave:1> global prob='al1' m=1 mu=1 lambda=-1
octave:2> [xstar,kp]=ntplain(0.5,20,1e-6,@augg,@augh)
xstar =  0.99966
kp =  14
octave:3> mu=1.01;
octave:4> [xstar,kp]=ntplain(0.5,20,1e-6,@augg,@augh)
xstar =  1.00000
kp =  13
octave:5> mu=8;
octave:6> [xstar,kp]=ntplain(0.5,20,1e-6,@augg,@augh)
xstar =  1.00000
kp =  8
octave:7> quit
```


20.2.4 An Augmented Lagrangian Algorithm

When we solved `al1` and `al2` in §20.2.3 we used the following approach.

1. Form the penalty function $\pi(\mathbf{x}, \boldsymbol{\lambda}; \mu) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x}) + \mu \sum_{i=1}^{m} [f_i(\mathbf{x})]^2$.
2. Set $\boldsymbol{\lambda} = \boldsymbol{\lambda}^\star$.
3. Set $\mu > \bar{\mu}$ so that π is locally convex at $\mathbf{x}^\star$.
4. Solve the resulting unconstrained penalty problem for $\mathbf{x}^\star$.

Of course this is not a practical strategy for solving arbitrary equality-constrained nonlinear programs, because for most problems we initially know nothing about $\boldsymbol{\lambda}^\star$. Also, unless $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^\star)$ is strictly convex so that $\bar{\mu} = 0$, all we know about $\bar{\mu}$ is that it must be positive. Fortunately it is possible to estimate $\boldsymbol{\lambda}^\star$ by minimizing π and to find a value of μ that is greater than $\bar{\mu}$ without knowing what $\bar{\mu}$ is.

Given any vector $\boldsymbol{\lambda}$ and scalar μ, if $\bar{\mathbf{x}}$ is a stationary point of the augmented Lagrangian then

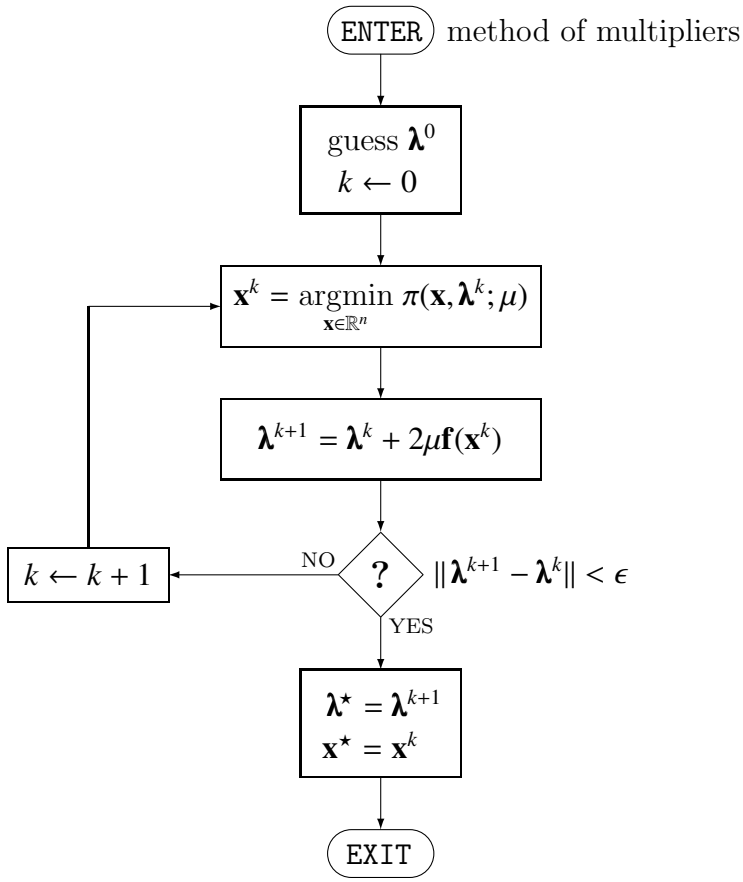
$$\begin{aligned} \nabla_x \pi(\bar{\mathbf{x}}, \boldsymbol{\lambda}; \mu) &= \nabla_x f_0(\bar{\mathbf{x}}) + \sum_{i=1}^{m} \lambda_i \nabla_x f_i(\bar{\mathbf{x}}) + 2\mu \sum_{i=1}^{m} f_i(\bar{\mathbf{x}}) \nabla_x f_i(\bar{\mathbf{x}}) \\ &= \nabla_x f_0(\bar{\mathbf{x}}) + \sum_{i=1}^{m} \left[\lambda_i + 2\mu f_i(\bar{\mathbf{x}})\right] \nabla_x f_i(\bar{\mathbf{x}}) = \mathbf{0}. \end{aligned}$$

If $(\bar{\mathbf{x}}, \boldsymbol{\lambda})$ were optimal it would satisfy the equality constraints, so it would also be a stationary point of the Lagrangian and satisfy

$$\nabla_x \mathcal{L}(\bar{\mathbf{x}}, \boldsymbol{\lambda}^\star) = \nabla_x f_0(\bar{\mathbf{x}}) + \sum_{i=1}^{m} \lambda_i^\star \nabla_x f_i(\bar{\mathbf{x}}) = \mathbf{0}$$

with $\lambda_i^\star = \lambda_i + 2\mu f_i(\bar{\mathbf{x}})$.

If $(\bar{\mathbf{x}}, \boldsymbol{\lambda})$ is *not* optimal it turns out [5, §17.3] [4, §16.6] that our estimate of $\boldsymbol{\lambda}$ can be improved by using this formula in the **method of multipliers** algorithm flowcharted on the next page. In the flowchart, $\mathbf{f}(\mathbf{x}^k)$ is the vector whose elements are the function values $f_i(\mathbf{x}^k)$, $i = 1 \ldots m$. At each iteration k the method of multipliers finds $\mathbf{x}^k$ and $\boldsymbol{\lambda}^{k+1}$ to minimize π and thus make $\nabla_x \mathcal{L}(\mathbf{x}^k, \boldsymbol{\lambda}^{k+1}) = \mathbf{0}$. Thus the stationarity conditions for the original problem are satisfied at every iteration. As $\lambda_i^{k+1} - \lambda_i^k \to 0$, also $2\mu f_i(\mathbf{x}^k) \to 0$ so $f_i(\mathbf{x}^k) \to 0$ and feasibility is gradually attained. If the algorithm converges to produce $\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k$ then $\mathbf{x}^{k+1} = \mathbf{x}^\star$ and $f_i(\mathbf{x}^k) = f_i(\mathbf{x}^\star) = 0$ for $i = 1 \ldots m$ so that $\nabla_\lambda \mathcal{L} = \mathbf{0}$. In that case the method of multipliers yields a point $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ that minimizes the augmented Lagrangian for the given value of μ.



The method of multipliers can be thought of [17, §2] [4, §16.6.1] as a gradient *ascent* algorithm for solving the *dual* of the following equality-constrained nonlinear program.

$$\begin{aligned} \mathscr{P}: \quad & \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f_0(\mathbf{x}) + \mu \sum_{i=1}^{m} [f_i(\mathbf{x}]^2 \\ & \text{subject to} \quad f_i(\mathbf{x}) = 0, \quad i = 1 \ldots m \end{aligned}$$

This problem's Lagrangian is just $\pi(\mathbf{x}, \boldsymbol{\lambda}; \mu)$, so its Lagrangian dual is (see Exercise §20.4.38)

$$\mathscr{D}: \quad \underset{\boldsymbol{\lambda}\in\mathbb{R}^m}{\text{maximize}} \quad g(\boldsymbol{\lambda}) \quad \text{where} \quad g(\boldsymbol{\lambda}) = \underset{\mathbf{x}\in\mathbb{R}^n}{\operatorname{argmin}}\ \pi(\mathbf{x}, \boldsymbol{\lambda}; \mu).$$

To maximize $g(\boldsymbol{\lambda})$ we can take steps in the direction of its gradient

$$\nabla_{\boldsymbol{\lambda}} g(\boldsymbol{\lambda}^k) = \underset{\mathbf{x}\in\mathbb{R}^n}{\operatorname{argmin}}\ \nabla_{\boldsymbol{\lambda}} \pi(\mathbf{x}, \boldsymbol{\lambda}^k; \mu) = \mathbf{f}\left(\underset{\mathbf{x}\in\mathbb{R}^n}{\operatorname{argmin}}\ \pi(\mathbf{x}, \boldsymbol{\lambda}^k; \mu)\right) = \mathbf{f}(\mathbf{x}^k)$$

and that is just what the flowchart above does, with a steplength of 2μ. Because of this interpretation, the method of multipliers is sometimes referred to as a **dual ascent** algorithm.

In order for the `argmin` box in the flowchart to succeed, $\pi(\mathbf{x},\boldsymbol{\lambda};\mu)$ must actually have a minimizing point $\mathbf{x}^k$ for *each* $\boldsymbol{\lambda}^k$ generated by the algorithm, not just for $\boldsymbol{\lambda}^\star$. That will be assured if $\mathbf{H}_\pi(\mathbf{x}^k,\boldsymbol{\lambda}^k;\mu)$ is positive definite at every iterate. As we discovered in §20.2.3, the region of $\mathbb{R}^n$ in which that is true depends on μ. Increasing μ enlarges the region of local convexity, at least up to some maximum size, so if $\mathbf{H}_\pi$ becomes non-positive-definite at some iteration it makes sense to increase μ, and that is what I have done in the MATLAB implementation `auglag.m` listed below.

```
 1 function [xstar,lambda,kl,rc,mu]=auglag(name,meq,xzero,epz,kmax)
 2 % solve an equality-constrained nonlinear program by augmented Lagrangian
 3
 4   global prob m mu lambda  % to aug.m, augg.m, and augh.m
 5   prob=name;               % pass the problem name
 6   m=meq;                   % the number of equality constraints
 7   mu=1;                    % the initial value of mu
 8   lambda=zeros(1,m);       % and the initial value of lambda
 9   fcn=str2func(prob);      % get a pointer to the function routine
10   xstar=xzero;             % start at the starting point
11   rc=1;                    % default rc to indicate nonconvergence
12
13   for kl=1:kmax
14 %     minimize the penalty function
15       [xnew,kn,nm]=ntrs(xstar,0,kmax,epz,@aug,@augg,@augh,0.5);
16       for kx=1:10
17           if(nm > 0)
18              mu=2*mu;
19              [xnew,kn,nm]=ntrs(xstar,0,kmax,epz,@aug,@augg,@augh,0.5);
20           else
21              break
22           end
23       end
24       xstar=xnew;
25
26 %     update the multipliers
27       esq=0;
28       for i=1:m
29           delta=2*mu*fcn(xstar,i);
30           lambda(i)=lambda(i)+delta;
31           esq=esq+delta^2;
32       end
33
34 %     test convergence
35       if(sqrt(esq) <= epz)
36          rc=0;
37          return
38       end
39   end
40 end
```

This routine begins by [4-8] sharing the problem data with `aug.m`, `augg.m`, and `augh.m`, which will be used to compute the value and derivatives of π. The multiplier μ will later be increased if necessary by [18] successive doublings, so it is [7] arbitrarily given the positive starting value of 1. The unknown vector of Lagrange multipliers is initialized [8] to zero (this is sure to be wrong, because equality constraints must be tight at optimality). The routine

that computes function values for the original problem will be needed later [29] so [9] the pointer `fcn` to it is found here. Then the solution is initialized [10] to the starting point and [11] the return code is set to `1` in case convergence is not achieved.

The method of multipliers is implemented in the loop [13-39] over `kl`. Its first stanza [14-24] solves the penalty problem for the current estimate of $\boldsymbol{\lambda}$; this is the argmin box of the flowchart. If in the first attempt [15] at minimizing π `ntrs.m` generates one or more iterates at which $\mathbf{H}_\pi(\mathbf{x}^{\mathtt{kn}})$ is not positive definite then `nm`, the number of Hessian modifications it performed, is [17] greater than zero. In that case μ is [18] doubled and [19] the minimization is attempted again. If the Hessian is still not positive definite μ is doubled again, and so on up to 10 times in the `kx` loop.

When the minimization of π is successful with `nm=0` [20-21] or the `kx` loop completes because $\mathbf{H}_\pi(\mathbf{x}^{\mathtt{kn}})$ is non-positive-definite at the final value of μ, the last point returned by `ntrs.m` is [24] taken as optimal for this value of $\boldsymbol{\lambda}$ and the method of multipliers continues to the next box of the flowchart. Here [26-32] each λ_i is incremented [30] by [29] $\delta = 2\mu f_i(\mathbf{x}^{\mathtt{kl}})$. This loop [27-31] also computes the square $e^2 = \sum_{i=1}^m \delta^2$ of the error in the estimate of $\boldsymbol{\lambda}$.

The decision box of the flowchart is implemented next [34-38]. If it finds [35] that $e = \|\boldsymbol{\lambda}^{k+1} - \boldsymbol{\lambda}^k\| < \epsilon$ it sets `rc=0` to indicate success and returns [37] the current values [1] of `xstar` and `lambda`, which are now presumably optimal.

I tested `auglag.m` on five of the equality-constrained examples we have considered, and the Octave session on the next page shows that it found $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ for each of them. The `al2`, `p1`, and `p2` problems have strictly convex Lagrangians and hence $\bar{\mu} = 0$, so it is not surprising that `auglag.m` leaves μ at its initial value of 1 in solving them. The `al1` problem has a nonconvex Lagrangian and we found analytically that $\mu > 1$ is required to make $\pi(\mathbf{x}, \boldsymbol{\lambda}; \mu)$ strictly convex at $x^\star$, so in solving that problem `auglag.m` increases μ from its initial value. In its travels from x^0 to $x^\star$ `ntrs.m` must have visited two points at which $\mathbf{H}_\pi(\mathbf{x}^{\mathtt{kn}})$ was not positive definite, because the starting value $\mu = 1$ was doubled twice to reach $\mu = 4$. The `one23` problem has a nonconvex objective, and `auglag.m` finds an optimal point different from those reported in §15.5 (see Exercise 20.4.40). Each of these problems has only one constraint, but some examples having more are suggested in the Exercises so you can confirm that the algorithm works for $m > 1$.

20.2.5 Conclusion

The augmented Lagrangian algorithm discussed in §20.2.4 has several important virtues.

- *It is exact;* modulo roundoff it can find, at a finite value of μ, solutions $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ that are precise and that precisely satisfy the equality constraints.
- *It is numerically stable;* because μ need not get very big, the condition number of $\mathbf{H}_\pi$ need not get very bad.
- *It might be faster than the classical penalty method;* see Exercise 20.4.34.


```
octave:1> format long
octave:2> [xstar,lambda,kp,rc,mu]=auglag('al1',1,0.5,1e-14,40)
xstar =  1.00000000000000
lambda = -1.00000000000000
kp =  31
rc = 0
mu =  4
octave:3> [xstar,lambda,kp,rc,mu]=auglag('al2',1,[2;2],1e-14,20)
xstar =

   1
   1

lambda =  0.500000000000000
kp =  13
rc = 0
mu =  1
octave:4> [xstar,lambda,kp,rc,mu]=auglag('p1',1,[4;4],1e-15,20)
xstar =

   2.00000000000000
   1.00000000000000

lambda =  1
kp =  20
rc = 0
mu =  1
octave:5> [xstar,lambda,kp,rc,mu]=auglag('p2',1,[1;2],1e-15,100)
xstar =

   0.945582993415969
   0.894127197437503

lambda =  3.37068560583615
kp =  96
rc = 0
mu =  1
octave:6> [xstar,lambda,kp,rc,mu]=auglag('one23',1,[0;0;0],1e-15,10)
xstar =

  -0.0773502691896258
   0.5000000000000000
   0.5773502691896257

lambda = -1
kp =  2
rc = 0
mu =  1024
octave:7> quit
```

Many refinements are possible [5, §17.4] to our simple implementation. The multiplier μ can be increased if an iteration of the method of multipliers produces too small a decrease in $\|\mathbf{f}(\mathbf{x})\|$, or even at every iteration, rather than only when $\mathbf{H}_\pi$ is non-positive-definite. I used the same tolerance `epz` everywhere, but the performance of the algorithm can be improved by making the tolerance for minimizing π different from the tolerance for the method of multipliers; then the tolerance for minimizing π can be made to depend on $\|\mathbf{f}(\mathbf{x}^{\mathtt{kl}})\|$ so that $\mathbf{x}^{\mathtt{kl}}$ is found more precisely as $\boldsymbol{\lambda}^\star$ is approached. I have used the same iteration limit `kmax`

everywhere but it might also be better to use different limits for `ntrs.m` and the `kl` loop. Production codes typically use more sophisticated methods to minimize π. By using the dual it is possible [1, p497-499] to derive a different formula for $\boldsymbol{\lambda}^{k+1}$, which leads to a more complicated version of the algorithm having faster convergence.

The augmented Lagrangian method can be modified to handle inequality constraints by introducing nonnegative slack variables [1, p499-501] as in this example.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_1(\mathbf{x}) & \leq \; 0 \end{array} \quad \longleftrightarrow \quad \begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_1(\mathbf{x}) + s & = \; 0 \\ & s & \geq \; 0 \end{array}$$

The penalty problem of the reformulation,

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n\ \lambda\in\mathbb{R}^1\ s\in\mathbb{R}^1}{\text{minimize}} & \pi(\mathbf{x}, s, \boldsymbol{\lambda}; \mu) = f_0(\mathbf{x}) + \lambda_1[f_1(\mathbf{x}) + s] + \mu[f_1(\mathbf{x}) + s]^2 \\ \text{subject to} & s \geq 0 \end{array}$$

can be solved using an algorithm (such as a descent method with a restricted line search) that enforces the bound on s.

20.3 Alternating Direction Methods of Multipliers

An equality-constrained nonlinear program that has certain special properties can be solved by the **alternating direction method of multipliers** or **ADMM** [17, §3] [2, §7.4], a modification of the method of multipliers that facilitates the use of **parallel processing** [100, §16.2]. Performing several parts of the calculation concurrently on different processors can reduce the wall-clock time required to complete an optimization. It can sometimes also permit the solution of large problems (see §25.7) by distributing among several computers a matrix that is too big to store on any one of them.

A **separable function** is a sum of terms each involving a different subset of the variables. If a partitioning of variables that makes the functions of a nonlinear program separable is the *same* for each function, then the variables are said to be **separable variables**. This problem, which I will call `admm` (see §28.7.28), has a separable objective function.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) = & x_1^2 + x_2^2 + x_3^2 + x_4^2 \\ \text{subject to} & \mathbf{A}\mathbf{x} = & \begin{bmatrix} 3x_1 - x_2 - 2x_3 - x_4 \\ -4x_1 + x_2 + 5x_3 + 2x_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \end{array}$$

It also has two other properties that are necessary for ADMM: the objective is convex and the constraints are linear.

If we solve the `admm` problem using the augmented Lagrangian algorithm, the method of multipliers iteration consists of these two updates (see the flowchart in §20.2.4).

$$\begin{aligned}\mathbf{x}^k &= \underset{\mathbf{x}\in\mathbb{R}^4}{\operatorname{argmin}}\ \pi(\mathbf{x},\boldsymbol{\lambda}^k;\mu)\\ \boldsymbol{\lambda}^{k+1} &= \boldsymbol{\lambda}^k + 2\mu\left(\mathbf{A}\mathbf{x}^k - \mathbf{b}\right)\end{aligned}$$

Here $\pi(\mathbf{x},\boldsymbol{\lambda}^k;\mu)$ is minimized with respect to x_1, x_2, x_3, and x_4 jointly. Now suppose we partition the variables by letting

$$\mathbf{y}_1 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{y}_2 = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} \qquad \mathbf{A}_1 = \begin{bmatrix} 3 & -1 \\ -4 & 1 \end{bmatrix} \qquad \mathbf{A}_2 = \begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}$$

In terms of the new variables the problem becomes

$$\begin{aligned}\underset{\mathbf{y}_1\ \mathbf{y}_2}{\text{minimize}} \quad & f_0(\mathbf{y}) = \mathbf{y}_1{}^\top\mathbf{y}_1 + \mathbf{y}_2{}^\top\mathbf{y}_2\\ \text{subject to} \quad & \mathbf{A}_1\mathbf{y}_1 + \mathbf{A}_2\mathbf{y}_2 = \mathbf{b}\end{aligned}$$

20.3.1 Serial ADMM

ADMM solves the partitioned version of the `admm` problem by enlarging the method of multipliers iteration to consist of these three updates, in which μ is now a fixed stepsize for dual ascent.

$$\begin{aligned}\mathbf{y}_1^{k+1} &= \underset{\mathbf{y}_1}{\operatorname{argmin}}\ \pi(\mathbf{y}_1,\mathbf{y}_2^k,\boldsymbol{\lambda}^k;\mu)\\ \mathbf{y}_2^{k+1} &= \underset{\mathbf{y}_2}{\operatorname{argmin}}\ \pi(\mathbf{y}_1^{k+1},\mathbf{y}_2,\boldsymbol{\lambda}^k;\mu)\\ \boldsymbol{\lambda}^{k+1} &= \boldsymbol{\lambda}^k + 2\mu\left(\mathbf{A}_1\mathbf{y}_1^{k+1} + \mathbf{A}_2\mathbf{y}_2^{k+1} - \mathbf{b}\right)\end{aligned}$$

Now the augmented Lagrangian penalty function is

$$\pi(\mathbf{y}_1,\mathbf{y}_2,\boldsymbol{\lambda};\mu) = \mathbf{y}_1{}^\top\mathbf{y}_1 + \mathbf{y}_2{}^\top\mathbf{y}_2 + \boldsymbol{\lambda}^\top(\mathbf{A}_1\mathbf{y}_1 + \mathbf{A}_2\mathbf{y}_2 - \mathbf{b}) + \mu(\mathbf{A}_1\mathbf{y}_1 + \mathbf{A}_2\mathbf{y}_2 - \mathbf{b})^\top(\mathbf{A}_1\mathbf{y}_1 + \mathbf{A}_2\mathbf{y}_2 - \mathbf{b}).$$

Letting $\mathbf{v}_2 = \mathbf{A}_2\mathbf{y}_2^k - \mathbf{b}$ the objective of the first subproblem reduces to

$$\begin{aligned}\pi(\mathbf{y}_1,\mathbf{y}_2^k,\boldsymbol{\lambda};\mu) &= \mathbf{y}_1{}^\top\mathbf{y}_1 + \mathbf{y}_2^{k\top}\mathbf{y}_2^k + \boldsymbol{\lambda}^\top(\mathbf{A}_1\mathbf{y}_1 + \mathbf{v}_2) + \mu(\mathbf{A}_1\mathbf{y}_1 + \mathbf{v}_2)^\top(\mathbf{A}_1\mathbf{y}_1 + \mathbf{v}_2)\\ &= \mathbf{y}_1{}^\top\mathbf{y}_1 + \boldsymbol{\lambda}^\top\mathbf{A}_1\mathbf{y}_1 + \mu(\mathbf{A}_1\mathbf{y}_1 + \mathbf{v}_2)^\top(\mathbf{A}_1\mathbf{y}_1 + \mathbf{v}_2) + [\mathbf{y}_2^{k\top}\mathbf{y}_2^k + \boldsymbol{\lambda}^\top\mathbf{v}_2].\end{aligned}$$

Letting $\mathbf{v}_1 = \mathbf{A}_1\mathbf{y}_1^k - \mathbf{b}$ the objective of the second subproblem reduces to

$$\begin{aligned}\pi(\mathbf{y}_1^{k+1},\mathbf{y}_2,\boldsymbol{\lambda};\mu) &= \mathbf{y}_1^{(k+1)\top}\mathbf{y}_1^{k+1} + \mathbf{y}_2{}^\top\mathbf{y}_2 + \boldsymbol{\lambda}^\top(\mathbf{A}_2\mathbf{y}_2 + \mathbf{v}_1) + \mu(\mathbf{A}_2\mathbf{y}_2 + \mathbf{v}_1)^\top(\mathbf{A}_2\mathbf{y}_2 + \mathbf{v}_1)\\ &= \mathbf{y}_2{}^\top\mathbf{y}_2 + \boldsymbol{\lambda}^\top\mathbf{A}_2\mathbf{y}_2 + \mu(\mathbf{A}_2\mathbf{y}_2 + \mathbf{v}_1)^\top(\mathbf{A}_2\mathbf{y}_2 + \mathbf{v}_1) + [\mathbf{y}_1^{(k+1)\top}\mathbf{y}_1^{k+1} + \boldsymbol{\lambda}^\top\mathbf{v}_1].\end{aligned}$$

In each subproblem objective the term in square brackets is held constant during that minimization and can therefore be ignored. To solve `admm` using this algorithm I wrote the MATLAB program and subroutines listed on the next page.

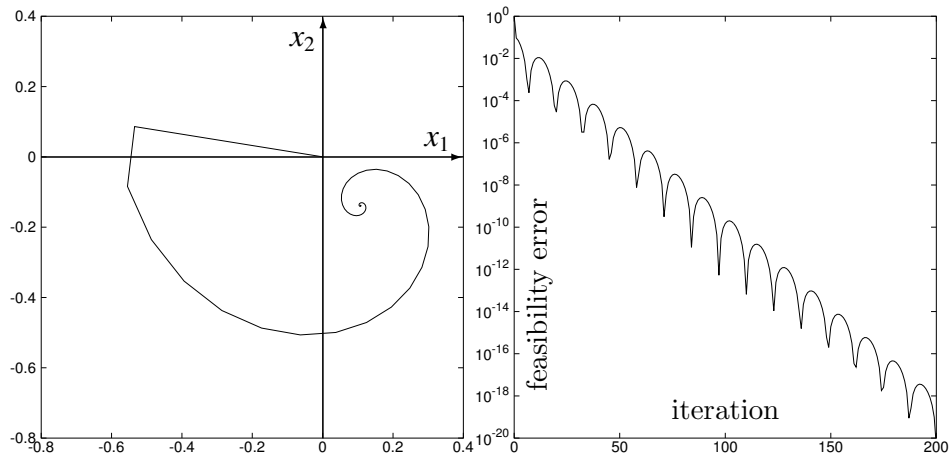

```
 1 % admm.m: serial ADMM with immediate updates
 2 clear; format long; clf
 3 global mu=1 A=zeros(2,2) lambda=ones(2,1) v=zeros(2,1)
 4
 5 xzero=[0;0;0;0];                            % unconstrained optimum
 6 y1=xzero(1:2); y2=xzero(3:4);               % partition variables
 7 A1=[3,-1;-4,1]; A2=[-2,-1;5,2]; b=[-1;3];   % partition constraints
 8
 9 x1k(1)=y1(1); x2k(1)=y1(2);                 % save y1 coordinates
10 delta=2*mu*(A1*y1+A2*y2-b);                 % feasiblity correction
11 ezero=delta'*delta;                         % starting error
12 err(1)=1; its(1)=0;                         % prepare to plot error
13
14 for k=1:200     % do method-of-multiplier iterations
15     v=A2*y2-b;  % constraint terms fixed while optimizing over y1
16     A=A1;       % y1 partition of constraints
17
18     y1new=ntrs(y1,0,10,1e-12,@admmf,@admmg,@admmh);
19
20     y1=y1new;   % update y1 as soon as possible
21     v=A1*y1-b;  % constraint terms fixed while optimizing over y2
22     A=A2;       % y2 partition of constraints
23
24     y2new=ntrs(y2,0,10,1e-12,@admmf,@admmg,@admmh);
25
26     y2=y2new;                   % update y2 as soon as possible
27     delta=2*mu*(A1*y1+A2*y2-b); % feasibility correction
28     lambda=lambda+delta;        % update lambda
29
30     x1k(k+1)=y1(1); x2k(k+1)=y1(2);          % save y1 coordinates
31     err(k+1)=delta'*delta/ezero; its(k+1)=k; % save error
32 end
33
34 xstar=[y1;y2] % report optimal point
35 lambda        % and optimal multipliers
36
37 figure(1)     % plot convergence
38 set(gca,'FontSize',25); hold on
39 axis([-0.8,0.4,-0.8,0.4],'square')
40 plot(x1k,x2k)
41 plot([0,0],[-0.8,0.4])
42 plot([-0.8,0.4],[0,0])
43 hold off
44 print -deps -solid admmcnv.eps
45
46 figure(2)     % plot error curve
47 set(gca,'FontSize',25); hold on
48 axis([0,200,1e-20,1e0],'square')
49 semilogy(its,err)
50 hold off
51 print -deps -solid admmerr.eps
```

```
function f=admmf(y)
  global mu A lambda v
  f=y'*y+lambda'*A*y+mu*(A*y+v)'*(A*y+v);
end

function g=admmg(y)
  global mu A lambda v
  g=2*y+A'*lambda+2*mu*A'*(A*y+v);
end

function H=admmh(y)
  global mu A
  n=size(y,1);
  H=2*eye(n)+2*mu*A'*A;
end
```

The program includes many lines for saving and plotting results, but the implementation of the algorithm itself is very simple. It begins by [6] initializing `y1` and `y2` and [7] defining the problem data. Then [14-32] the loop over `k` cycles through the three ADMM updates. Each iteration performs [15-20] the argmin over `y1`, then [21-26] the argmin over `y2`, and finally [27-28] the update to the Lagrange multipliers. The minimizations of π are carried out [18,24] by `ntrs.m`, which invokes the routines `admmf.m`, `admmg.m`, and `admmh.m` listed on the right.

The Octave session below shows 1> the output from the program and 2> that the point is feasible for the equality constraints. It is not hard to show (see Exercise 20.4.43) that this $\mathbf{x}^\star$ and $\boldsymbol{\lambda}^\star$ satisfy the Lagrange conditions for the original problem.

```
octave:1> admm
xstar =

   0.107692308873322
  -0.138461536641094
   0.646153846017645
   0.169230771097437

lambda =

  -0.892307673096215
  -0.615384607645544

octave:2> [A1,A2]*xstar-b
ans =

   1.28333454973983e-10
   1.48717482773009e-10

octave:3> xstar'*xstar
ans =  0.476923077129108
```



The program also plots the convergence trajectory of $\mathbf{y}_1$ (the convergence trajectory of $\mathbf{y}_2$ is very similar) and the error curve. ADMM clearly has [54] linear convergence; the bumps result from the alternation between optimizing in the $\mathbf{y}_1$ direction and optimizing in the $\mathbf{y}_2$ direction.

20.3.2 Parallel ADMM

As I mentioned in §20.3.0 an important motivation for ADMM is that it can facilitate the use of parallel processing. If we partition the variables of a problem into 2 subsets as we did above, then we can perform each argmin operation on a different processor. In the simplest parallel computing configuration processor 1 is assigned to finding $\mathbf{y}_1^{k+1}$, processor 2 is assigned to finding $\mathbf{y}_2^{k+1}$, and processor 0 is assigned to finding $\boldsymbol{\lambda}^{k+1}$ and carrying out the other steps of the algorithm. To solve a problem with many variables we could partition them into p subsets, enlarge the method of multipliers iteration to include p argmin updates yielding $\mathbf{y}_1^{k+1} \ldots \mathbf{y}_p^{k+1}$, and use a different processor to do each minimization.

Unfortunately, if the updates of the $\mathbf{y}_i$ are like those we used in solving `admm` above,

$$\begin{aligned}
\mathbf{y}_1^{k+1} &= \operatorname*{argmin}_{\mathbf{y}_1} \pi(\mathbf{y}_1, \mathbf{y}_2^k, \boldsymbol{\lambda}^k; \mu) \\
\mathbf{y}_2^{k+1} &= \operatorname*{argmin}_{\mathbf{y}_2} \pi(\mathbf{y}_1^{k+1}, \mathbf{y}_2, \boldsymbol{\lambda}^k; \mu),
\end{aligned}$$

they cannot be done at the same time; before we can start finding $\mathbf{y}_2^{k+1}$ we need to know $\mathbf{y}_1^{k+1}$.

ADMM is sometimes described [17, p14] as a version of the method of multipliers in which each cycle of updates is similar to an iteration of the **Gauss-Seidel algorithm** for solving a system of linear algebraic equations [20, p386-387]. In an iterative method for solving $\mathbf{Ax} = \mathbf{b}$, each new approximation x_j^{k+1} to a solution component is calculated from a formula involving the other components x_i, $i \neq j$. In the Gauss-Seidel method the values assumed for $i = 1 \ldots j-1$ are the most recently calculated ones, x_i^{k+1}, while the values assumed for $i = j+1 \ldots n$ are the values obtained in the previous iteration, x_i^k. This is an improvement over the **Jacobi algorithm**, in which the formula for x_j^{k+1} involves only the x_i^k, because always using the latest information speeds convergence.

One way to parallelize ADMM is to use the $\mathbf{y}_i^k$ from the previous iteration to find each $\mathbf{y}_i^{k+1}$, as in the Jacobi algorithm. Then the argmin updates for the `admm` problem look like this.

$$\begin{aligned}
\mathbf{y}_1^{k+1} &= \operatorname*{argmin}_{\mathbf{y}_1} \pi(\mathbf{y}_1, \mathbf{y}_2^k, \boldsymbol{\lambda}^k; \mu) \\
\mathbf{y}_2^{k+1} &= \operatorname*{argmin}_{\mathbf{y}_2} \pi(\mathbf{y}_1^k, \mathbf{y}_2, \boldsymbol{\lambda}^k; \mu)
\end{aligned}$$

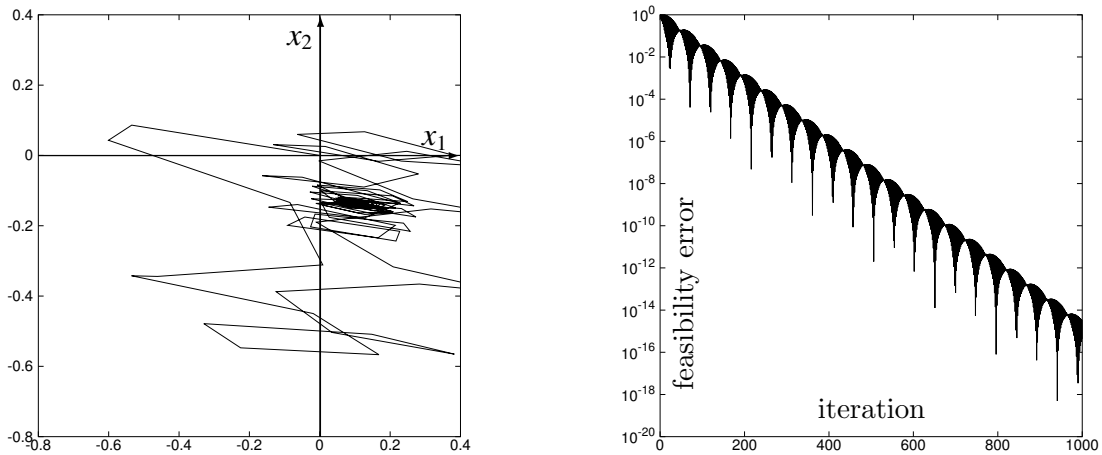
Because each uses quantities that are already known at the beginning of iteration $k+1$, these minimizations can be performed concurrently on different processors. At the beginning of each iteration the **master program** running on processor 0 computes the quantity we have called $\mathbf{v}_2$ and sends it along with $\boldsymbol{\lambda}^k$ and the submatrix $\mathbf{A}_1$ to processor 1. It also computes $\mathbf{v}_1$ and sends it along with $\boldsymbol{\lambda}^k$ and the submatrix $\mathbf{A}_2$ to processor 2. Then processor 0 waits as the **worker programs** on processors 1 and 2 solve their respective optimization problems and transmit the results $\mathbf{y}_1^{k+1}$ and $\mathbf{y}_2^{k+1}$ back. When both subproblem solutions have arrived, processor 0 uses them to compute $\boldsymbol{\lambda}^{k+1}$ and the iteration is complete. The data transmissions that I have just described are typically accomplished [100, §16.2.2] by calling the subroutines of a **message passing library** such as **MPI** [118].

To simulate this process in MATLAB, I wrote the program `padmm.m` listed on the next page. It is identical to `admm.m` except that the updates of `y1` and `y2` are now delayed until the end of each iteration. That way the `y1` used in finding `y1new` [18] and the `y2` used in finding `y2new` [23] were both found in the previous iteration, and the updates are parallelizable as described above. When the program is run it produces the output below, which is in good agreement with what we found using `admm.m` above. ADMM still works if we use Jacobi-style updates so that they can be computed in parallel.

```
octave:2> padmm
xstar =

   0.107692324821060
  -0.138461540054718
   0.646153833238240
   0.169230760918215

lambda =

  -0.892307817274887
  -0.615384398445065
```

```
% padmm.m: parallel ADMM with delayed updates
clear; format long; clf
global mu=1 A=zeros(2,2) lambda=ones(2,1) v=zeros(2,1)

xzero=[0;0;0;0];                            % unconstrained optimum
y1=xzero(1:2); y2=xzero(3:4);               % partition variables
A1=[3,-1;-4,1]; A2=[-2,-1;5,2]; b=[-1;3];   % partition constraints

x1k(1)=y1(1); x2k(1)=y1(2);                 % save y1 coordinates
delta=2*mu*(A1*y1+A2*y2-b);                 % feasiblity correction
ezero=delta'*delta;                         % starting error
err(1)=1; its(1)=0;                         % prepare to plot error

for k=1:1000    % do method-of-multiplier iterations
    v=A2*y2-b;  % constraint terms fixed while optimizing over y1
    A=A1;       % y1 partition of constraints

    y1new=ntrs(y1,0,10,1e-12,@admmf,@admmg,@admmh);

    v=A1*y1-b;  % constraint terms fixed while optimizing over y2
    A=A2;       % y2 partition of constraints

    y2new=ntrs(y2,0,10,1e-12,@admmf,@admmg,@admmh);

    delta=2*mu*(A1*y1+A2*y2-b); % feasibility correction
    lambda=lambda+delta;        % update lambda

    y1=y1new;   % wait to update y1 and y2
    y2=y2new;   % until the end of the iteration

    x1k(k+1)=y1(1); x2k(k+1)=y1(2);          % save y1 coordinates
    err(k+1)=delta'*delta/ezero; its(k+1)=k; % save error
end

xstar=[y1;y2] % report optimal point
lambda        % and optimal multipliers

figure(1)     % plot convergence
set(gca,'FontSize',25); hold on
axis([-0.8,0.4,-0.8,0.4],'square')
plot(x1k,x2k)
plot([0,0],[-0.8,0.4])
plot([-0.8,0.4],[0,0])
hold off
print -deps -solid padmmcnv.eps

figure(2)     % plot error curve
set(gca,'FontSize',25); hold on
axis([0,1000,1e-20,1e0],'square')
semilogy(its,err)
hold off
print -deps -solid padmmerr.eps
```

Alas, as shown by the error curve on the next page it takes `padmm.m` more than 1000 iterations to reach a feasibility error comparable to that achieved by `admm.m` in 200. The parallel algorithm converges much more slowly than the serial one, perhaps because the trajectory of its iterates is chaotic. Using ADMM in this way is worthwhile only if there are enough subproblems that solving them in parallel more than makes up for this slow convergence.



ADMM plays an important role in big data applications and is routinely used to solve very large problems (see §25.7) but [17, §3.2.2] its slow rate of convergence makes it most useful in settings where only modest accuracy is required. Finding ways to parallelize its Gauss-Seidel-style updates is an active area of research [129, §2.6].

20.4 Exercises

20.4.1 [E] What makes a penalty function *exact*? Give formulas for two penalty functions that are exact.

20.4.2 [E] Write down the max penalty problem corresponding to the standard-form nonlinear program. At what points is the max penalty function nondifferentiable?

20.4.3 [E] The solution of a max penalty problem is characterized by an *inflection value* of the multiplier, which we called $\bar{\mu}$. (a) What is its significance? (b) How is it related to the values of the Lagrange multipliers in the optimal solution of the original nonlinear program?

20.4.4 [H] Explain why, for problem ep2,

$$\mathbf{x}^{\pi}(\mu) = \begin{cases} [\mu/2, \mu/2]^{\top} & \mu \le \bar{\mu} \\ [1, 1]^{\top} & \mu \ge \bar{\mu} \end{cases}$$

20.4.5 [E] What do these MATLAB routines compute, and how do they work? (a) `epy.m`, (b) `epyg.m`, and (c) `epyh.m`.

20.4.6 [H] Newton descent might get lucky and find the minimum of a max penalty function, but it is not sure to work and often it fails. (a) Describe one way in which the full-step algorithm can fail. (b) Explain why the version of the algorithm that uses a bisection line search is more robust against discontinuities in the gradient. (c) Present an example to show how the line search version can also fail.

20.4.7 [H] When we try to minimize the max penalty function for `ep2` with `ntfs.m`, for $\mu > \bar{\mu}$ the algorithm generates iterates that alternate between $\bar{\mathbf{x}}(\mu) = [\mu/2, \mu/2]^\top$ and $\hat{\mathbf{x}}(\mu) = [0, 0]^\top$. Explain why this behavior could not happen if $\nabla\pi(\mathbf{x}; \mu)$ were continuous.

20.4.8 [P] When we used Newton descent to minimize the max penalty function for `ep2` in §20.1, its first step was based on the quadratic model function $q(\mathbf{x}) = x_1^2 + x_2^2$ that describes $\pi(\mathbf{x}; \mu)$ at the feasible starting point $\mathbf{x}^0 = [2, 2]^\top$. That step yields $\mathbf{x}^1 = [0, 0]^\top$, where the penalty function and its quadratic model abruptly change to $q(\mathbf{x}) = \pi(\mathbf{x}; \mu) = x_1^2 + x_2^2 + \mu(2 - x_1 - x_2)$, leading to $\mathbf{x}^2 = [\frac{3}{2}, \frac{3}{2}]^\top$ and the cycling nonconvergence we observed. It is possible by using the `ntrs.m` routine of §17.2 to instead approach $\mathbf{x}^\star$ by generating only iterates at which our initial $q(\mathbf{x})$ remains a good approximation to the function. (a) Show by using MATLAB that `ntrs.m` can solve `ep2`. (b) Does the max penalty function play any role when this approach is used? Explain.

20.4.9 [H] Give an example of a scalar function $y = f(x)$ of $x \in \mathbb{R}^1$ having a discontinuous first derivative but a continuous second derivative.

20.4.10 [P] In §20.1 we found that `nt.m` successfully minimizes the max penalty function for the `ep2` problem. Try `nt.m` on each of the following inequality-constrained problems: (a) `ep1`; (b) `b1`; (c) `b2`.

20.4.11 [P] In §20.1 we found that `ntfs.m` fails to minimize the max penalty function for the `ep2` problem. Try each of the following unconstrained minimizers on that problem: (a) `ntw.m`; (b) `sdfs.m`; (c) `sdw.m`; (d) `plrb.m`. Do any of them return the correct $\mathbf{x}^\mu$ for values of $\mu \in [0, 10]$?

20.4.12 [E] Describe three different ways of reliably minimizing the max penalty function even though it has a discontinuous gradient.

20.4.13 [H] In §20.1, I claimed that the nonlinear program on the right is equivalent to the one on the left.

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f_0(\mathbf{x}) + \mu \sum_{i=1}^m \max[0, f_i(\mathbf{x})] \qquad \longleftrightarrow \qquad \begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) + \mu \displaystyle\sum_{i=1}^m t_i \\ \text{subject to} & t_i \geq 0 \\ & t_i \geq f_i(\mathbf{x}), \end{array}$$

(a) Explain why the two are equivalent. (b) Reformulate the max penalty problem for `ep2` and write the KKT conditions for the resulting constrained problem. Are they satisfied by the optimal solution to `ep2`?

20.4.14 [E] How can the max penalty method described in §20.1 be modified to handle equality constraints?

20.4.15 [H] Show that for $\lambda > 0$ the Lagrangian for `al2`, $\mathcal{L}(\mathbf{x}, \lambda) = -x_1 - x_2 + \lambda(x_1^2 + x_2^2 - 2)$, is a strictly convex function of $\mathbf{x}$.

20.4.16 [H] Explain why the nonlinear program on the right below has the same optimal point as the nonlinear program on the left.

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) \\ \text{subject to} & f_i(\mathbf{x}) = 0,\ i = 1\ldots m \end{array} \qquad \begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n\ \boldsymbol{\lambda}\in\mathbb{R}^m}{\text{minimize}} & \mathcal{L}(\mathbf{x},\lambda) \\ \text{subject to} & f_i(\mathbf{x}) = 0,\ i = 1\ldots m \end{array}$$

Do the two problems have the same optimal value?

20.4.17 [E] Suppose we know $\boldsymbol{\lambda}^\star$ for an equality-constrained nonlinear program, and that we solve $\nabla\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}^\star) = \mathbf{0}$ for $\bar{\mathbf{x}}$. Is it ever true that $\bar{\mathbf{x}} = \mathbf{x}^\star$? If so, explain how that can happen.

20.4.18 [E] When is the unconstrained minimizing point of $\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}^\star)$ the same as the constrained minimizing point of $f_0(\mathbf{x})$ subject to $f_i(\mathbf{x}) = 0,\ i = 1\ldots m$? Explain why this is true of `al2`.

20.4.19 [H] Problem `al1` can be reformulated in a way that makes $\mathcal{L}$ a convex function of x. (a) Rewrite the constraint to make it linear. (b) Use the Lagrange method to solve the resulting problem for $(x^\star,\lambda^\star)$. (c) Graphically minimize $\mathcal{L}(x,\lambda^\star)$ subject to the constraint. (d) Is it possible to find $\mathbf{x}^\star$ by minimizing $\mathcal{L}(x,\lambda^\star)$ *without* enforcing the constraint? Explain why or why not.

20.4.20 [E] Write down the augmented Lagrangian penalty function corresponding to this nonlinear program.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_i(\mathbf{x}) & = \ 0,\ i = 1\ldots m \end{array}$$

20.4.21 [E] The augmented Lagrangian for `al1` has a stationary point at $(x^\star,\lambda^\star)$. (a) Does this point depend on the value of μ? (b) What determines whether this point is a minimizing point, and inflection point, or a maximizing point? In the graph of $\pi(x,\lambda^\star;\mu)$ given in §20.2.1, on which of the curves is $x^\star$ a minimizing point? On which is it an inflection point? On which is it a maximizing point? (c) Explain why this augmented Lagrangian is an exact penalty function, and describe how it works.

20.4.22 [E] Suppose that μ is chosen so that an augmented Lagrangian has a minimizing point at $\mathbf{x}^\star$. What determines the region of $\mathbb{R}^n$ over which π is a convex function of $\mathbf{x}$?

20.4.23 [P] We saw in §18.4 that the classical penalty method suffers from the drawback that $\mathbf{H}_\pi$ becomes badly conditioned as $\mu \to \infty$. An exact penalty function is minimized at $\mathbf{x}^\pi(\mu) = \mathbf{x}^\star$ for a finite (and usually small) positive value of μ, so this difficulty does not arise. But what would happen if μ did (e.g., in the augmented Lagrangian algorithm) reach a high positive value? Determine experimentally how the condition number of $\mathbf{H}_\pi(\mathbf{x}^0,\boldsymbol{\lambda}^\star;\mu)$ varies with μ for (a) `al1`; (b) `al2`. (c) Analytically compute $\mathbf{H}_\pi$ for `al2`, and show that your formula explains what you observed in part `b`.

20.4.24 [E] The MATLAB routines `aug.m`, `augg.m` and `augh.m` compute respectively the value, gradient, and Hessian of the augmented Lagrangian for a nonlinear program. Explain how these routines work. Why do they invoke `pye.m`, `pyeg.m`, and `pyeh.m`?

20.4.25 [E] In §20.2.3 we solved the `al2` and `al1` problems numerically by minimizing the augmented Lagrangian function. (a) Why is $\bar{\mu} = 0$ for `al2`? (b) Why is $\bar{\mu} = 1$ for `al1`? (c) In finding $\mathbf{x}^\star$ in this way, why is it necessary for μ to be strictly greater than $\bar{\mu}$?

20.4.26 [E] One strategy for solving an equality-constrained nonlinear program is to form the augmented Lagrangian function $\pi(\mathbf{x}, \boldsymbol{\lambda}^\star; \mu)$, set $\mu > \bar{\mu}$, and minimize π. Why is this approach seldom practical?

20.4.27 [E] In the augmented Lagrangian algorithm of §20.2.4 we use the update formula $\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + 2\mu\mathbf{f}(\mathbf{x}^k)$ to refine our estimate of the Lagrange multiplier vector. Explain where this formula comes from.

20.4.28 [E] Describe the method of multipliers algorithm. Do its iterates satisfy the stationarity conditions of the original nonlinear program? Do they satisfy the feasibility conditions? Explain.

20.4.29 [E] In our implementation `auglag.m` of the augmented Lagrangian algorithm, what strategy is used for setting the value of the multiplier μ? What is the highest possible value that μ can attain?

20.4.30 [P] Use `auglag.m` to solve the following problem [4, Example 16.12]

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & e^{3x_1} + e^{-4x_2} \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 - 1 = 0 \end{array}$$

starting from $\mathbf{x}^0 = [-1, 1]^\top$.

20.4.31 [P] Use `auglag.m` to solve the following problem, which was first presented in Exercise 15.6.36,

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & -3x_1x_3 - 4x_2x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_2^2 + x_3^2 - 4 = 0 \\ & f_2(\mathbf{x}) & = & x_1x_3 - 3 = 0 \end{array}$$

starting from $\mathbf{x}^0 = [1, 1, 2]$. Are there other starting points from which the algorithm finds $\mathbf{x}^\star$? Are there starting points from which the algorithm fails?

20.4.32 [P] Use `auglag.m` to solve the following problem, which was first presented in Exercise 15.6.42.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) & = & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2^2 + x_3^2 - 25 = 0 \\ & f_2(\mathbf{x}) & = & 8x_1 + 14x_2 + 7x_3 - 56 = 0 \end{array}$$

20.4.33 [E] List three virtues of the augmented Lagrangian algorithm. Does it have any drawbacks?

20.4.34 [P] Because each iteration of the augmented Lagrangian algorithm involves the solution of the subproblem to minimize $\pi(\mathbf{x}, \boldsymbol{\lambda}^k; \mu)$, the convergence behavior of the overall

algorithm is difficult to characterize analytically. However, an error curve can be measured experimentally. (a) Revise `auglag.m` to make it serially reusable (see §10.6.1). (b) Write a MATLAB program that invokes your revised version of the routine one iteration at a time, remembering at the end of each iteration the error in the current $\mathbf{x}^k$ and the total number of `ntrs.m` iterations k_{tot} consumed so far in the solution process. This requires adding up the number of iterations that `ntrs.m` uses each time it is invoked. (c) In your program plot the common logarithm of the solution error versus k_{tot}. (d) Run your program on the `al1` test problem. (e) By inspection of the resulting error curve estimate the order of convergence of the algorithm.

20.4.35 [P] Using the serially reusable version of `auglag.m` that you wrote for Exercise 20.4.34, plot the convergence trajectory of the algorithm as it solves the `ep2` problem of §20.1.

20.4.36 [P] Some implementations of the augmented Lagrangian algorithm increase μ if an iteration of the method of multipliers produces too small a decrease in $\|\mathbf{f}(\mathbf{x})\|$. Revise `auglag.m` to incorporate this refinement. How much decrease in infeasibility should be required? Should this amount of decrease change as the optimal point is approached?

20.4.37 [P] Some implementations of the augmented Lagrangian algorithm make the tolerance for minimizing π depend on $\|\mathbf{f}(\mathbf{x})\|$ so that $\mathbf{x}^{\mathtt{kl}}$ is found more precisely as $\boldsymbol{\lambda}^\star$ is approached. Revise `auglag.m` to incorporate this refinement, and conduct experiments to investigate its effect on the performance of the algorithm.

20.4.38 [H] In §20.2.4 we saw that the method of multipliers can be thought of as using gradient ascent to solve the dual of a certain equality-constrained nonlinear program. Show that the dual of the problem $\mathscr{P}$ is $\mathscr{D}$ as claimed.

20.4.39 [H] In §20.2.4 we saw that the method of multipliers can be thought of as using gradient ascent to solve the dual of a certain equality-constrained nonlinear program. The argument presented there implicitly contains several assumptions concerning the functions involved. For example, it assumes that the inf of π is actually attained. (a) List all of the unspoken assumptions that must be true in order for this derivation to work, and state conditions on the $f_i(\mathbf{x})$ that must be satisfied to ensure that it does work. (b) Identify one of the assumptions which, if it is not true, leads to failure of the method of multipliers.

20.4.40 [P] In §20.2.4, `auglag.m` finds a solution to the `one23` problem different from those reported in §15.5. Do numerical calculations to show that it is an alternate optimal point.

20.4.41 [E] If a nonlinear program has certain properties it can be solved by the alternating direction method of multipliers. What are those properties? If a problem has them, why might ADMM be preferable to some other algorithm for solving it?

20.4.42 [E] What is a *separable function*? When does a nonlinear program have *separable variables*?

20.4.43 [P] Use the Lagrange method to solve `admm` analytically, and confirm that the result given in §20.3.1 is correct. The Lagrange conditions for this problem are a system of linear equations that you can solve easily using MATLAB.

20.4.44 [E] Describe the ADMM algorithm in words. What role does the penalty multiplier μ play in the algorithm? What is the algorithm's order of convergence? Is this algorithm ideally suited to producing extremely accurate results?

20.4.45 [P] Simplify the `admm.m` program of §20.3.1 by removing all of the code that is devoted to saving and plotting intermediate results. Show that the original and simplified programs produce the same printed output.

20.4.46 [P] Use the ADMM algorithm to solve the `admm` problem by partitioning the variables into the subsets $\{x_1\}$ and $\{x_2, x_3, x_4\}$.

20.4.47 [H] Partitioning the variables of a nonlinear program into p subsets for ADMM enlarges the method of multipliers iteration to include p argmin updates yielding $\mathbf{y}_1^{k+1} \ldots \mathbf{y}_p^{k+1}$. (a) If Gauss-Seidel-style updates are used, write down the first, second, and last of them to show the pattern of variable subsets that are held constant and allowed to vary during each subproblem minimization. (b) Can these updates be performed in parallel? Explain. (c) How do your answers change if Jacobi-style updates are used?

20.4.48 [E] Suppose that the updates in an ADMM implementation are to be performed in parallel. Describe a possible configuration of independent processors that could be used to perform this calculation, and describe the flow of data between them as the iterations of the algorithm progress. How are these data transmissions typically accomplished?

20.4.49 [E] Describe the advantages and drawbacks of the serial and parallel ADMM algorithms. When is it worthwhile to use the parallel approach?

20.4.50 [P] Write a program in FORTRAN, C, or C++ that implements the ADMM algorithm and uses MPI subroutine calls for message passing to solve the `admm` problem on 3 processors of a parallel computer. Does performing two updates concurrently result in a net speedup of the calculation?

20.4.51 [H] Several of the programs available on the NEOS web server (see §8.3.1) are based on the algorithms discussed in this Chapter [5, §17.5]. By searching the web, find out which of the programs are based on which of the algorithms.

21

Interior-Point Methods

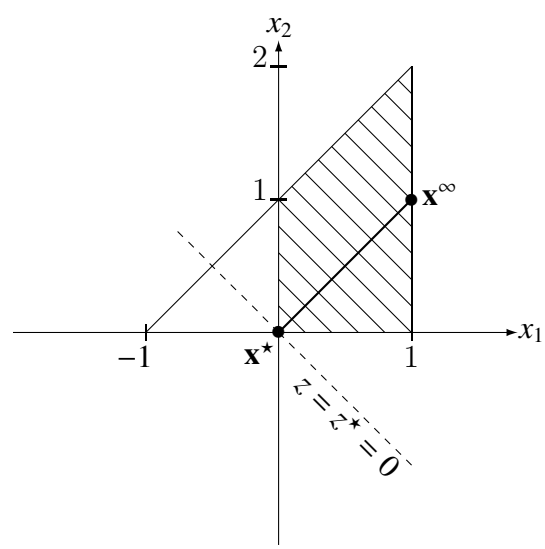
When the classical barrier method of §19 works at all it converges only linearly, and it has limited accuracy because $\mathbf{H}_\beta$ becomes badly conditioned as $\mu \to 0$ and that degrades the precision with which Newton descent directions can be computed near the optimal point. Although we were able to find $\mathbf{x}^\star$ exactly for the simple demonstration problems we considered, the algorithm is of limited use for the larger and more difficult optimizations that typically arise in practical applications.

It has, however, inspired the development of more effective algorithms for solving inequality-constrained mathematical programs from inside the feasible set. These interior-point methods move the nonnegativity constraints of the original problem into a barrier function and numerically solve the Lagrange conditions for the resulting constrained barrier problem.

21.1 Interior-Point Methods for LP

We will begin our exploration of interior-point methods by examining one that is an alternative to the simplex method for solving certain linear programs [4, §10] [5, §14] [2, §5.1.1-5.1.2]. Consider this problem, which I will call `in1` (see §28.5.15).

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1 + x_2 & = & z \\ \text{subject to} & -x_1 + x_2 & \le & 1 \\ & x_1 & \le & 1 \\ & \mathbf{x} & \ge & \mathbf{0} \end{array}$$



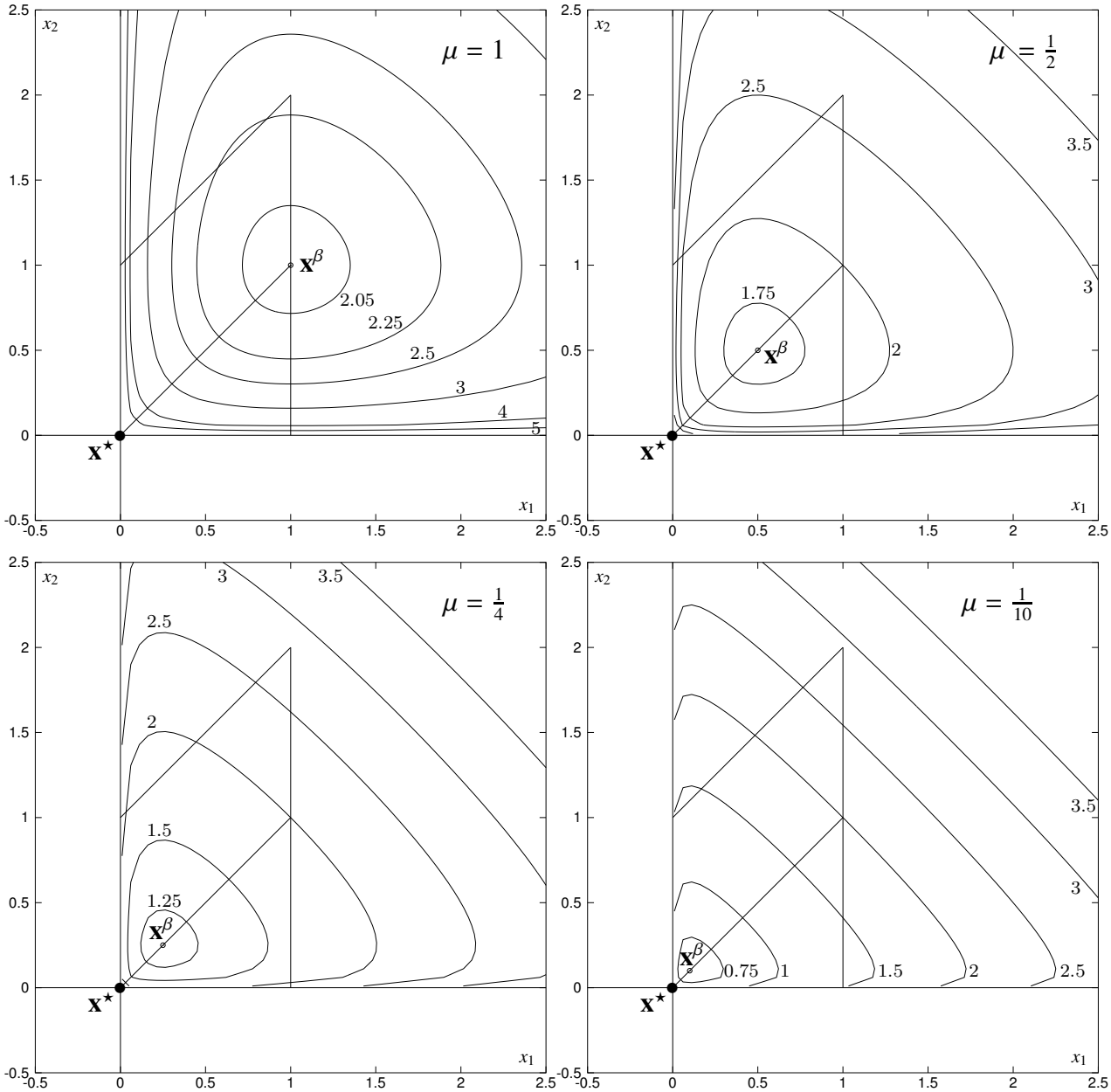
If we incorporate the nonnegativity constraints into this barrier function

$$\beta(\mathbf{x};\mu) = x_1 + x_2 - \mu\ln(x_1) - \mu\ln(x_2)$$

then problem `in1` is related to the following inequality-constrained barrier problem.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2_+}{\text{minimize}} & \beta(\mathbf{x};\mu) & & \\ \text{subject to} & -x_1 + x_2 & \le & 1 \\ & x_1 & \le & 1 \end{array}$$

In each contour diagram on the next page I have plotted the $\mathbf{x}^\beta$ that solves this barrier problem for the given value of μ. Starting from the **analytic center** $\mathbf{x}^\infty = [1,1]^\top$ as $\mu \to 0$ these solutions approach $\mathbf{x}^\star = [0,0]^\top$ along the **central path** drawn as a thick line above.



Interior to the feasible set the inequality constraints are slack, so we can find the central path analytically by minimizing $\beta(\mathbf{x};\mu)$ over $\mathbf{x}$.

$$\frac{\partial\beta}{\partial x_1} = 1 - \frac{\mu}{x_1} = 0 \quad\Rightarrow\quad x_1(\mu) = \mu$$
$$\frac{\partial\beta}{\partial x_2} = 1 - \frac{\mu}{x_2} = 0 \quad\Rightarrow\quad x_2(\mu) = \mu$$

Then as $\mu \to 0$ we get $\mathbf{x}(\mu) \to [0,0]^\top = \mathbf{x}^\star$. Of course this is not a very practical way of

solving linear programs. A barrier algorithm must gradually approach the zero hyperplanes of the slack and coordinate variables that define the optimal vertex even if that vertex is not the origin. Also, unless we can draw a graph we need some way of explicitly enforcing the inequalities that are not nonnegativities, because we left them out of the barrier function.

21.1.1 A Primal-Dual Formulation

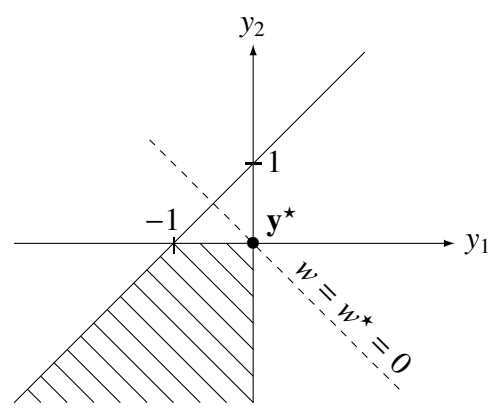
For the slack variables to come into play we need to start with a problem in which they actually appear, so on the left below I reformulated `in1` into standard form. According to §5.2.1 the dual of this problem is the inequality-constrained linear program on the right, which I put into standard form below the primal.

$$\begin{array}{lrrrrcl}
\mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & x_1 & +x_2 & & & & \\
\text{subject to} & -x_1 & +x_2 & +x_3 & & = & 1 \\
& x_1 & & & +x_4 & = & 1 \\
& x_1 & x_2 & x_3 & x_4 & \geq & 0
\end{array}
\qquad
\mathbf{x}^\star = [0,0,1,1]^\top
$$

$$\longrightarrow
\begin{array}{lrrcl}
\underset{\mathbf{y}\in\mathbb{R}^2}{\text{maximize}} & y_1 & +y_2 & & \\
\text{subject to} & -y_1 & +y_2 & \leq & 1 \\
& y_1 & & \leq & 1 \\
& y_1 & & \leq & 0 \\
& & y_2 & \leq & 0 \\
& y_1 & y_2 & & \text{free}
\end{array}
$$

$$\begin{array}{lrrrrrrcl}
\mathscr{D}: \underset{\mathbf{y}\in\mathbb{R}^2}{\text{minimize}} & -y_1 & -y_2 & & & & & = & w \\
\text{subject to} & -y_1 & +y_2 & +s_1 & & & & = & 1 \\
& y_1 & & & +s_2 & & & = & 1 \\
& y_1 & & & & +s_3 & & = & 0 \\
& & y_2 & & & & +s_4 & = & 0 \\
& y_1 & y_2 & & & & & & \text{free} \\
& & & s_1 & s_2 & s_3 & s_4 & \geq & 0
\end{array}
$$

$$\mathbf{y}^\star = [0,0]^\top$$
$$\mathbf{s}^\star = [1,1,0,0]^\top$$



In general the standard-form linear program and its **standard-form dual** are

$$\begin{array}{lrcl}
\mathscr{P}: \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\
\text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\
& \mathbf{x} & \geq & \mathbf{0}
\end{array}
\qquad
\begin{array}{lrcl}
\mathscr{D}: \underset{\mathbf{y}\in\mathbb{R}^m,\,\mathbf{s}\in\mathbb{R}^n}{\text{minimize}} & -\mathbf{b}^\top\mathbf{y} & & \\
\text{subject to} & \mathbf{A}^\top\mathbf{y}+\mathbf{s} & = & \mathbf{c} \\
& \mathbf{y}\text{ free},\ \mathbf{s} & \geq & 0
\end{array}
$$

where for `in1` we have

$$\mathbf{A}_{m\times n} = \begin{bmatrix} -1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{b}_{m\times 1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \mathbf{c}_{n\times 1} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

We can form a barrier problem for $\mathscr{P}$ by including the nonnegativity constraints in β as we did for the inequality-constrained `in1`. Here I will use the approach and notation of [4, §10.6 and §10.2]; for a quite different derivation see [2, §5.1].

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^n_+}{\text{minimize}} & \beta(\mathbf{x};\mu) & = & \mathbf{c}^\top\mathbf{x} - \mu\displaystyle\sum_{j=1}^{n}\ln(x_j) \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \end{array}$$

This is an equality-constrained nonlinear program, which we can solve analytically by the Lagrange method. If we let $\mathbf{y}$ be the vector of m Lagrange multipliers associated with the rows of the equality constraint, then $\mathcal{L}(\mathbf{x},\mathbf{y}) = \beta(\mathbf{x};\mu)+\mathbf{y}^\top(\mathbf{b}-\mathbf{A}\mathbf{x})$ and the Lagrange conditions are

$$\begin{aligned} \nabla_x\mathcal{L} &= \nabla\beta - \mathbf{A}^\top\mathbf{y} = \mathbf{0} \\ \nabla_y\mathcal{L} &= \mathbf{b} - \mathbf{A}\mathbf{x} = \mathbf{0}. \end{aligned}$$

The gradient of this barrier function is

$$\nabla\beta = \mathbf{c} - \mu\begin{bmatrix} \dfrac{1}{x_1} \\ \vdots \\ \dfrac{1}{x_n} \end{bmatrix} = \mathbf{c} - \mu[\mathbf{X}^{-1}\mathbf{1}]$$

where $\mathbf{X}$ is a diagonal matrix whose diagonals are the x_j and $\mathbf{1}$ is a vector of n 1's. Then we can write the first Lagrange condition as

$$\mathbf{c} - \mu\mathbf{X}^{-1}\mathbf{1} - \mathbf{A}^\top\mathbf{y} = \mathbf{0}.$$

But if we let $\mathbf{s} = \mathbf{c} - \mathbf{A}^\top\mathbf{y}$ as in the constraint of $\mathscr{D}$, this is just

$$\mathbf{s} - \mu\mathbf{X}^{-1}\mathbf{1} = \mathbf{0} \quad \text{so} \quad \mathbf{s}\mathbf{X} = \mu\mathbf{1} \quad \text{or} \quad s_jx_j = \mu,\ j = 1\ldots n.$$

Thus the barrier problem above is solved by the vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{s}$, all functions of μ, that satisfy this **Lagrange system** of equations and inequalities.

$$\begin{array}{rl} \mathbf{A}\mathbf{x} = \mathbf{b} & \text{primal feasibility} \\ \mathbf{A}^\top\mathbf{y} + \mathbf{s} = \mathbf{c} & \text{dual feasibility} \\ s_jx_j = \mu,\ j = 1\ldots n & \text{interiority} \\ \mathbf{y}\text{ free}, \mathbf{x} \geq \mathbf{0}, \mathbf{s} \geq \mathbf{0} & \text{nonnegativity} \end{array}$$

The **interiority condition** ensures, because $\mu > 0$, that $x_j > 0$ and $s_j > 0$ and therefore that both $\mathbf{x}$ and $\mathbf{y}$ are *strictly* feasible. In the limit as $\mu \to 0$ this condition approaches the

complementary slackness condition of §5.1.5. There we saw that if $\mathbf{x}$ is feasible for $\mathscr{P}$ and $\mathbf{y}$ is feasible for $\mathscr{D}$ and complementary slackness holds, then the vectors are optimal for their respective problems. Thus as $\mu \to 0$, $\mathbf{x}$ and $\mathbf{y}$ approach optimality for $\mathscr{P}$ and $\mathscr{D}$.

Sometimes it is easy to find vectors $\mathbf{x}^0$, $\mathbf{y}^0$, and $\mathbf{s}^0$ that satisfy the feasibility conditions exactly. For `in1`, the graphical solution of the primal shows that $x_1 = \frac{1}{2}$, $x_2 = \frac{1}{2}$ is interior to the feasible set for the original problem, and we can satisfy $\mathbf{Ax} = \mathbf{b}$ by adjusting the slacks in its $\mathscr{P}$ to be $x_3 = 1$ and $x_4 = \frac{1}{2}$. From the graphical solution of the dual we see that $\mathbf{y} = [-\frac{1}{2}, -\frac{1}{2}]^\top$ is interior to the feasible set of that problem, and we can satisfy $\mathbf{A}^\top\mathbf{y} + \mathbf{s} = \mathbf{c}$ by adjusting the slacks in its $\mathscr{D}$ to be $\mathbf{s} = [1, \frac{3}{2}, \frac{1}{2}, \frac{1}{2}]^\top$. Unfortunately, vectors $\mathbf{x}^0$ and $\mathbf{s}^0$ constructed in this way usually do *not* have the property that $s_j x_j = \mu,\ j = 1 \ldots n$ for a given μ (or for any μ).

21.1.2 Solving the Lagrange System

To find vectors $\mathbf{x}^\beta$, $\mathbf{y}^\beta$, and $\mathbf{s}^\beta$ that satisfy *all* of the conditions we must use an algorithm for solving simultaneous nonlinear algebraic equations. One approach is to think of moving from a trial point $(\mathbf{x}, \mathbf{y}, \mathbf{s})$ to a new point $(\mathbf{x} + \Delta\mathbf{x}, \mathbf{y} + \Delta\mathbf{y}, \mathbf{s} + \Delta\mathbf{s})$ where, for $j = 1 \ldots n$, the corrections Δx_j and Δs_j are chosen so that the interiority condition is satisfied exactly at the new point.

$$\begin{aligned} (x_j + \Delta x_j)(s_j + \Delta s_j) &= \mu \\ s_j x_j + s_j \Delta x_j + x_j \Delta s_j + \Delta x_j \Delta s_j &= \mu \end{aligned}$$

Near the solution Δx_j and Δs_j will both be small, so we will assume that their product is exactly zero (see [4, §10.2.2] for a way to avoid making this simplification). Then

$$s_j \Delta x_j + x_j \Delta s_j = \mu - x_j s_j$$

and the interiority requirement for $j = 1 \ldots n$ can be expressed like this

$$\begin{bmatrix} s_1 & & \\ & \ddots & \\ & & s_n \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_n \end{bmatrix} + \begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix} \begin{bmatrix} \Delta s_1 \\ \vdots \\ \Delta s_n \end{bmatrix} = \begin{bmatrix} \mu \\ \vdots \\ \mu \end{bmatrix} - \begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix} \begin{bmatrix} s_1 & & \\ & \ddots & \\ & & s_n \end{bmatrix} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

or in more compact form as $\mathbf{S}\Delta\mathbf{x} + \mathbf{X}\Delta\mathbf{s} = \mu\mathbf{1} - \mathbf{XS1}$ where $\mathbf{S}$ is a diagonal matrix whose diagonal elements are the s_j.

To preserve primal feasibility we must choose $\Delta\mathbf{x}$ so that $\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b}$, but $\mathbf{A}(\mathbf{x}) = \mathbf{b}$ so it must be that $\mathbf{A}\Delta\mathbf{x} = \mathbf{0}$. To preserve dual feasibility we must choose $\Delta\mathbf{y}$ and $\Delta\mathbf{s}$ so that $\mathbf{A}^\top(\mathbf{y} + \Delta\mathbf{y}) + (\mathbf{s} + \Delta\mathbf{s}) = \mathbf{c}$, but $\mathbf{A}^\top\mathbf{y} + \mathbf{s} = \mathbf{c}$ so it must be that $\mathbf{A}^\top\Delta\mathbf{y} + \Delta\mathbf{s} = \mathbf{0}$.

We now have three conditions which, if they are satisfied by $\Delta\mathbf{x}$, $\Delta\mathbf{y}$, and $\Delta\mathbf{s}$, ensure that the new point will preserve primal and dual feasibility and come closer to satisfying the

interiority condition:

$$\begin{array}{lrcl} \text{(A)} & \mathbf{S}\Delta\mathbf{x} + \mathbf{X}\Delta\mathbf{s} & = & \mu\mathbf{1} - \mathbf{XS1} \\ \text{(B)} & \mathbf{A}\Delta\mathbf{x} & = & \mathbf{0} \\ \text{(C)} & \mathbf{A}^\top\Delta\mathbf{y} + \Delta\mathbf{s} & = & \mathbf{0}. \end{array}$$

These equations can be solved analytically by reasoning as follows.

$$\begin{array}{lcl} \text{(C)} & \Rightarrow & \boxed{\Delta\mathbf{s} = -\mathbf{A}^\top\Delta\mathbf{y}} \\ \text{(A)} & \Rightarrow & \mathbf{S}\Delta\mathbf{x} + \mathbf{X}(-\mathbf{A}^\top\Delta\mathbf{y}) = \mu\mathbf{1} - \mathbf{XS1} \end{array}$$

Premultiplying the second of these equations by the conformable product $\mathbf{AS}^{-1}$ we get

$$\mathbf{AS}^{-1}\mathbf{S}\Delta\mathbf{x} + \mathbf{AS}^{-1}\mathbf{X}(-\mathbf{A}^\top\Delta\mathbf{y}) = \mathbf{AS}^{-1}(\mu\mathbf{1} - \mathbf{XS1})$$

In this equation the first term $\mathbf{AS}^{-1}\mathbf{S}\Delta\mathbf{x} = \mathbf{A}\Delta\mathbf{x} = \mathbf{0}$. In the second term let $\mathbf{D}_{n\times n} = \mathbf{S}^{-1}\mathbf{X}$, and in the last term let $\mathbf{v}_{n\times 1} = (\mu\mathbf{1} - \mathbf{XS1})$. Then

$$\begin{array}{rcl} \mathbf{AD}(-\mathbf{A}^\top\Delta\mathbf{y}) & = & \mathbf{AS}^{-1}\mathbf{v} \\ -(\mathbf{ADA}^\top)\Delta\mathbf{y} & = & \mathbf{AS}^{-1}\mathbf{v} \\ \boxed{\Delta\mathbf{y} \quad = \quad -(\mathbf{ADA}^\top)^{-1}\mathbf{AS}^{-1}\mathbf{v}} \end{array}$$

Finally,

$$\begin{array}{lrcl} \text{(A)} \Rightarrow & \mathbf{S}\Delta\mathbf{x} & = & \mathbf{v} - \mathbf{X}\Delta\mathbf{s} \\ & \Delta\mathbf{x} & = & \mathbf{S}^{-1}\mathbf{v} - \mathbf{S}^{-1}\mathbf{X}\Delta\mathbf{s} \\ & \boxed{\Delta\mathbf{x} \quad = \quad \mathbf{S}^{-1}\mathbf{v} - \mathbf{D}\Delta\mathbf{s}} \end{array}$$

Using the boxed formulas we can calculate the corrections $\Delta\mathbf{y}$, $\Delta\mathbf{s}$, and $\Delta\mathbf{x}$ in that order. Then moving from $(\mathbf{x}, \mathbf{y}, \mathbf{s})$ to $(\mathbf{x} + \Delta\mathbf{x}, \mathbf{y} + \Delta\mathbf{y}, \mathbf{s} + \Delta\mathbf{s})$ should solve the barrier problem for a given μ. However, in our analysis we assumed that $\Delta x_j \Delta s_j = 0$ and far from optimality that is not quite true, so interiority might not hold exactly at the first point we generate. The `deltas.m` routine listed on the next page therefore repeats the correction process to ensure that the Lagrange system is solved precisely. The code begins by [2-5] defining the data for our example problem. Then, starting [6-8] from the $\mathbf{x}^0$, $\mathbf{y}^0$, and $\mathbf{s}^0$ provided by the user it [9-20] performs ten iterations (that turns out to be more than enough). In each iteration it is necessary to [10-12] construct the diagonal matrices $\mathbf{X}$ and $\mathbf{S}$ and compute $\mathbf{D} = \mathbf{S}^{-1}\mathbf{X}$. Because $\mathbf{X}$ changes at each iteration so does [13] $\mathbf{v}$. The MATLAB locution `ones(n,1)` [13] produces $\mathbf{1}$, the vector of n 1's. Then [14-16] we can evaluate $\Delta\mathbf{y}$, $\Delta\mathbf{s}$, and $\Delta\mathbf{x}$ and [17-19] move to the next point. When the solution of the nonlinear equations has been found, the routine checks [21] primal feasibility, [22] dual feasibility, and [23] the interiority condition. The MATLAB locution `s.*x` [23] computes the n-vector whose j'th element is $s_j x_j$.


```
function [x,y,s]=deltas(xzero,yzero,szero,mu)
  n=4;
  A=[-1,1,1,0;1,0,0,1];
  b=[1;1];
  c=[1;1;0;0];
  x=xzero;
  y=yzero;
  s=szero;
  for k=1:10
      X=diag(x);
      S=diag(s);
      D=inv(S)*X;
      v=mu*ones(n,1)-X*S*ones(n,1);
      dy=-inv(A*D*A')*A*inv(S)*v;
      ds=-A'*dy;
      dx=inv(S)*v-D*ds;
      x=x+dx;
      s=s+ds;
      y=y+dy;
  end
  primal=A*x-b
  dual=A'*y+s-c
  interior=s.*x
end
```

```
octave:1> format long
octave:2> yzero=[-0.5;-0.5];
octave:3> szero=[1;1.5;0.5;0.5];
octave:4> xzero=[0.5;0.5;1;0.5];
octave:5> mu=0.5;
octave:6> interior=xzero.*szero
interior =

   0.500000000000000
   0.750000000000000
   0.500000000000000
   0.250000000000000

octave:7> [x,y,s]=deltas(xzero,yzero,szero,mu)
primal =

   2.22044604925031e-16
  -4.44089209850063e-16

dual =

   2.22044604925031e-16
   0.00000000000000e+00
   0.00000000000000e+00
   0.00000000000000e+00

interior =

   0.500000000000000
   0.500000000000000
   0.500000000000000
   0.500000000000000

x =

   0.377908041731624
   0.337685765339289
   1.040222276392335
   0.622091958268375

y =

  -0.480666499215998
  -0.803739693713089

s =

   1.323073194497091
   1.480666499215998
   0.480666499215998
   0.803739693713089
```

The Octave session on the right shows that the feasible $(\mathbf{x}^0, \mathbf{s}^0)$ we chose earlier has $x_2^0 s_2^0 = 0.75$ and $x_4^0 s_4^0 = 0.25$, both far from $\mu = 0.5$, but the new point produced by ten correction iterations satisfies primal feasibility, dual feasibility, and interiority to within machine precision. The first two components of this $\mathbf{x}^\beta$ differ from the $x_1(\frac{1}{2}) = x_2(\frac{1}{2}) = \frac{1}{2}$ that we found for our initial formulation using the inequality-constrained `in1`, because the central path is now in $\mathbb{R}^{10}$. Thus, although the inequality-constrained and standard-form linear programs are intimately related, the corresponding barrier problems behave somewhat differently.

The output on the next page uses another $\mathbf{x}^0$ strictly feasible for $\mathscr{P}$ and another μ, both just as plausible as the values we chose above. Once again we find a solution to the Lagrange system that precisely satisfies primal and dual feasibility and the interiority requirement. However, this point violates the nonnegativity condition because x_4 and s_4 are both less than zero! Our derivation of the formulas for $\Delta\mathbf{x}$, $\Delta\mathbf{y}$, and $\Delta\mathbf{s}$ assumed that $\mathbf{x}$ and $\mathbf{s}$ would remain strictly positive, so in solving the Lagrange system we must explicitly guard against any component becoming negative.


```
octave:1> format long
octave:2> yzero=[-0.5;-0.5];
octave:3> szero=[1;1.5;0.5;0.5];
octave:4> xzero=[0.9;0.9;1;0.1];
octave:5> mu=0.1;
octave:6> interior=xzero.*szero
interior =

   0.9000000000000000
   1.3500000000000001
   0.5000000000000000
   0.0500000000000000

octave:7> [x,y,s]=deltas(xzero,yzero,szero,mu)
primal =

   3.55271367880050e-15
   6.66133814775094e-16

dual =

   0
   0
   0
   0

interior =

   0.100000000000000
   0.100000000000000
   0.100000000000000
   0.100000000000000

x =

   1.1161544051135346
   0.0952849525989713
   2.0208694525145670
  -0.1161544051135338 ☜ violates nonnegativity

y =

  -0.0494836516409163
   0.8609230093534216

s =

   0.0895933390056621
   1.0494836516409163
   0.0494836516409163
  -0.8609230093534216 ☜ violates nonnegativity
```

21.1.3 Solving the Linear Program

To keep from violating nonnegativity when solving the Lagrange system of our barrier problem we can restrict the corrections to $\alpha\Delta\mathbf{x}$, $\alpha\Delta\mathbf{y}$, and $\alpha\Delta\mathbf{s}$, where $\alpha > 0$ is chosen to keep $\mathbf{x}$ and $\mathbf{s}$ strictly positive. A new coordinate value such as $x_4 + \alpha\Delta x_4$ runs the risk of being

negative only if $\Delta x_4 < 0$. In that case to avoid stepping too far we need

$$\begin{aligned} x_4 + \alpha\Delta x_4 &> 0 \\ \alpha\Delta x_4 &> -x_4 \\ \alpha &< -x_4/\Delta x_4. \end{aligned}$$

In the last step dividing by $\Delta x_4 < 0$ changes the sense of the inequality. To keep *every* x_j and s_j strictly positive we can use

$$\alpha < \min\left\{\min_{\Delta x_j<0} \frac{-x_j}{\Delta x_j}\ ,\ \min_{\Delta s_j<0} \frac{-s_j}{\Delta s_j}\right\}.$$

In solving our barrier problem with an algorithm that gradually reduces μ, it is fortunately *not* necessary at each step to solve the Lagrange system precisely as we did with `deltas.m` in §21.1.2; one correction is enough. Each barrier problem solution $(\mathbf{x}^\beta,\ \mathbf{y}^\beta,\ \mathbf{s}^\beta)$ is only an approximation to $(\mathbf{x}^\star, \mathbf{y}^\star, \mathbf{s}^\star)$ anyway, and as $\mu \to 0$ each $s_j x_j \to 0$ so $\Delta x_j \Delta s_j \to 0$, our formula (A) becomes exact, and the barrier problem can be solved precisely in one step.

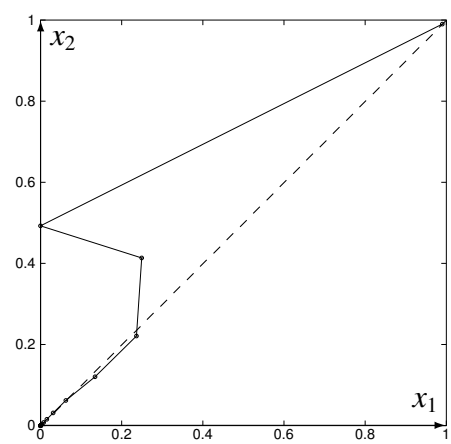
To implement the algorithm that we have developed I wrote the MATLAB routine `lpin.m` listed below.

```
function [xstar,ystar]=lpin(A,b,c,xzero,yzero)
% minimize c'x subject to Ax=b and x nonnegative
% by a primal-dual interior point algorithm

  x=xzero;
  y=yzero;
  s=c-A'*yzero;
  epz=1e-9;
  mu=1;
  n=size(xzero,1);
  for k=1:52
      X=diag(x);
      S=diag(s);
      D=inv(S)*X;
      v=mu*ones(n,1)-X*S*ones(n,1);
      dy=-inv(A*D*A')*A*inv(S)*v;
      ds=-A'*dy;
      dx=inv(S)*v-D*ds;
      if(norm(dy)<epz && norm(ds)<epz && norm(dx)<epz) break; end
      alpha=1;
      for j=1:n
          if(dx(j) < 0) alpha=min(alpha,0.99999*(-x(j)/dx(j))); end
          if(ds(j) < 0) alpha=min(alpha,0.99999*(-s(j)/ds(j))); end
      end
      y=y+alpha*dy;
      s=s+alpha*ds;
      x=x+alpha*dx;
      mu=mu/2;
  end
  xstar=x;
  ystar=y;

end
```


The routine begins [5-6] by initializing `x` and `y` to the starting vectors [1] provided, assuming that $\mathbf{x}^0$ is strictly feasible for $\mathscr{P}$ and $\mathbf{y}^0$ is an interior point of $\mathscr{D}$. Next [7] it sets $\mathbf{s}^0 = \mathbf{c} - \mathbf{A}^\top \mathbf{y}^0$ to establish dual feasibility, and sets [8] a convergence tolerance and [9] a starting value for μ. The number `n` of x_j variables [10] is needed to construct $\mathbf{1}$ [15]. Then [11-29] a sequence of up to 52 barrier problems are solved. The part of this code [12-18] that computes $\Delta\mathbf{y}$, $\Delta\mathbf{s}$, and $\Delta\mathbf{x}$ is familiar from `deltas.m`, but here [19] these quantities are also used to test for convergence. At any iteration of the algorithm it is possible for one or two of these vectors to be very small, so it is necessary to test all three. The steplength α is determined [20-24] using the formula above. The strictness of the inequality is enforced by using only 0.99999 of the smallest permissible step; this is called a **fraction to the boundary rule** [5, p567]. The current $\mathbf{y}^\beta$, $\mathbf{s}^\beta$, and $\mathbf{x}^\beta$ are used [25-27] as the starting point for the next iteration, and [28] μ is reduced. When convergence is achieved or the iteration limit is met [30-31] the current primal and dual solutions are returned [1] in `xstar` and `ystar`.

The Octave session on the next page illustrates the use of this code to solve the standard-form versions of `in1` and the `brewery` problem of §1.3.1. For both problems the answers that `lpin.m` finds differ from the true vertex solutions by only on the order of `epz`.

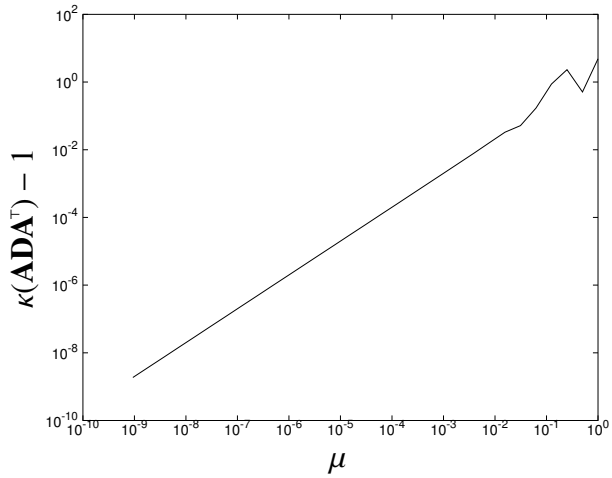
I used a modified version of `lpin.m` to plot the coordinates $x_1(\mu)$ and $x_2(\mu)$ generated by the algorithm in solving the standard-form version of the `in1` problem, obtaining the picture below. The dashed line is the central path that we found in §21.1.1 for the inequality-constrained problem.



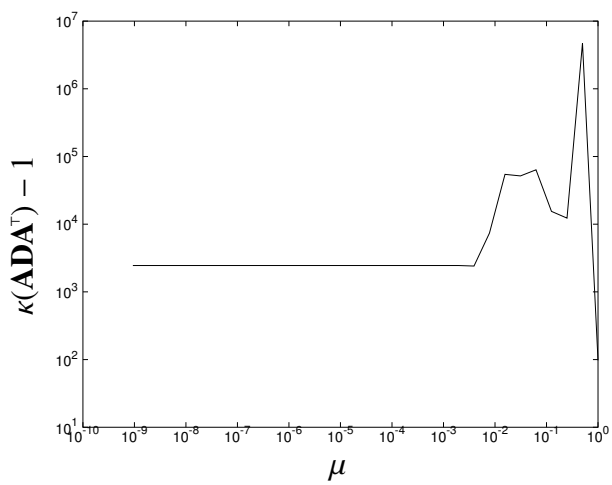
On the first iteration our solution of the Lagrange system generates a step $\Delta\mathbf{x}$ that would cause the new point to have a negative x_1 coordinate, so the code finds $\alpha \approx 0.513$ and steps to just short of the x_2 axis. The other iterates were all able to use $\alpha = 1$, and demonstrate that when that is possible they tend to follow the central path. The primal-dual algorithm is thus a **path-following method** [4, p346] [5, p399].

The numerical stability of the calculations performed by `lpin.m` depends on the condition number $\kappa(\mathbf{ADA}^\top)$, so I plotted $\kappa - 1$ as a function of μ for both `in1` and `brewery` to the right of the output on the next page. As μ decreases, for `in1` the condition number approaches 1 and for `brewery` it approaches only 2453, so this algorithm does not suffer from the terminal ill-conditioning we observed for the barrier algorithm of §19.3.


```
octave:1> format long
octave:2> A=[-1,1,1,0;1,0,0,1];
octave:3> b=[1;1];
octave:4> c=[1;1;0;0];
octave:5> xzero=[0.5;0.5;1;0.5];
octave:6> yzero=[-0.5;-0.5];
octave:7> [xstar,ystar]=lpin(A,b,c,xzero,yzero)
xstar =

   9.31322574615479e-10
   9.31322574615479e-10
   1.00000000000000e+00
   9.99999999068677e-01

ystar =

  -9.31322574615479e-10
  -9.31322574615479e-10
```




```
octave:8> A=[7,10,8,12,1,0,0;1,3,1,1,0,1,0;2,4,1,3,0,0,1];
octave:9> b=[160;50;60];
octave:10> c=[-90;-150;-60;-70;0;0;0];
octave:11> xzero=[1;1;1;1;123;44;50];
octave:12> yzero=[-1;-1;-52];
octave:13> [xstar,ystar]=lpin(A,b,c,xzero,yzero)
xstar =

   5.00000006881941e+00
   1.24999999717530e+01
   9.93410746256510e-11
   2.44281331046683e-11
   2.48352686564128e-10
   7.50000004087680e+00
   9.93410746256510e-11

ystar =

  -7.50000000021110e+00
  -2.48352685207770e-10
  -1.87499999993232e+01

octave:14> quit
```



The simplex method uses pivots to move along the edges of the feasible set while interior-point methods use more expensive iterations, but hopefully fewer of them, to cross its interior. If the optimal point is unique and nondegenerate the primal-dual algorithm approaches a vertex solution in the limit, but if not it can converge to an interior point of the optimal set (see Exercise 21.4.15) so production codes use a **basis recovery procedure** to find the nearest vertex exactly. Unlike the simplex algorithm, interior-point methods do not stall doing degenerate pivots and they are insensitive to the number of vertices between $\mathbf{x}^0$ and $\mathbf{x}^\star$. Interior-point methods can (and some even do) have polynomial worst-case complexity [5, §14.1] in stark contrast to the exponential worst-case complexity of the simplex method (see §7.9). In practice interior-point methods are said to perform better than the simplex method on linear programs that are [5, p392] large and in which [4, p329] the matrix $\mathbf{ADA}^{\mathsf{T}}$ is sparse with a pattern of nonzeros that makes it easy to factor.

Many refinements and extensions have been made to the primal-dual interior-point method for linear programming, including [4, §10.3] ways to deal with infeasible starting points and diagnose infeasible problems and [5, p415] ways to solve convex quadratic programs. In addition to the primal-dual approach that we have studied, **potential-reduction** [5, §14.3] and **affine-scaling** [4, §10.5] formulations have also been used to derive interior-point methods for linear programming. All of these topics are beyond the scope of this text.

21.2 Newton's Method for Systems of Equations

The iterative scheme that we used in §21.1.2 to solve the Lagrange system of the LP barrier problem is a customization for that purpose of a more general algorithm called **Newton's method for systems** [20, §9.2] [30, §5.2] [4, §2.7.1] [5, §11.1]. The Newton descent algorithm we studied in §13 is a different customization, to solve the particular system of nonlinear equations represented by $\nabla f_0(\mathbf{x}) = \mathbf{0}$. In §21.3 and §23.2 we will need Newton's method for systems again, in the more general form developed next.

21.2.1 From One Dimension to Several

Recall that if $x \in \mathbb{R}^1$ and $f(x)$ is a scalar function that has the two-term Taylor's series

$$f(x^k + \Delta) \approx f(x^k) + f'(x^k) \cdot \Delta,$$

where $f'(x^k)$ means df/dx evaluated at x^k, then provided $f'(x^k) \neq 0$ we can make $f(x^k + \Delta) \approx 0$ by picking $\Delta = -f(x^k)/f'(x^k)$. The algorithm below (see §28.3.2) repeats this process of correcting the point, and if all goes well it converges to an x where $f(x) = 0$.

	$k = 0$	start from x^0
1	$\Delta = -f(x^k)/f'(x^k)$	find the correction
	$x^{k+1} = x^k + \Delta$	update the estimate of the root
	$k = k + 1$	count the iteration
	`GO TO 1`	and repeat

We can use a similar approach to solve a system of r equations $f_i(\mathbf{x}) = 0$ for a point $\mathbf{x} \in \mathbb{R}^r$ where they are all satisfied. Now each correction $\mathbf{\Delta} \in \mathbb{R}^r$ is the vector that satisfies the system of linear equations

$$\begin{aligned} f_1(\mathbf{x}^k + \mathbf{\Delta}) \approx f_1(\mathbf{x}^k) + \nabla f_1(\mathbf{x}^k)^\top \mathbf{\Delta} &= 0 \\ &\vdots \\ f_r(\mathbf{x}^k + \mathbf{\Delta}) \approx f_r(\mathbf{x}^k) + \nabla f_r(\mathbf{x}^k)^\top \mathbf{\Delta} &= 0 \end{aligned} \quad \text{or} \quad \begin{bmatrix} f_1(\mathbf{x}^k) \\ \vdots \\ f_r(\mathbf{x}^k) \end{bmatrix} + \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \\ \vdots \\ \nabla f_r(\mathbf{x}^k)^\top \end{bmatrix} \mathbf{\Delta} = \mathbf{0}.$$

We can call the vector whose elements are the function values $\mathbf{f}(\mathbf{x}^k)$ and the matrix whose rows are the gradients of the functions $\mathbf{J}(\mathbf{x})$, and write these equations in the form

$$\mathbf{f}(\mathbf{x}^k) + \mathbf{J}(\mathbf{x}^k)\boldsymbol{\Delta} = \mathbf{0}.$$

Then if $\mathbf{J}$ has an inverse we can solve for

$$\boldsymbol{\Delta} = [\mathbf{J}(\mathbf{x}^k)]^{-1}[-\mathbf{f}(\mathbf{x}^k)]$$

and generalize the scalar algorithm like this.

	$k = 0$	start from $\mathbf{x}^0$
1	$\boldsymbol{\Delta} = [\mathbf{J}(\mathbf{x}^k)]^{-1}[-\mathbf{f}(\mathbf{x}^k)]$	find the correction vector
	$\mathbf{x}^{k+1} = \mathbf{x}^k + \boldsymbol{\Delta}$	update the estimate of the root
	$k = k + 1$	count the iteration
	`GO TO 1`	and repeat

To try this algorithm on a set of nonlinear equations we need to know, for each element of the **Jacobian matrix**

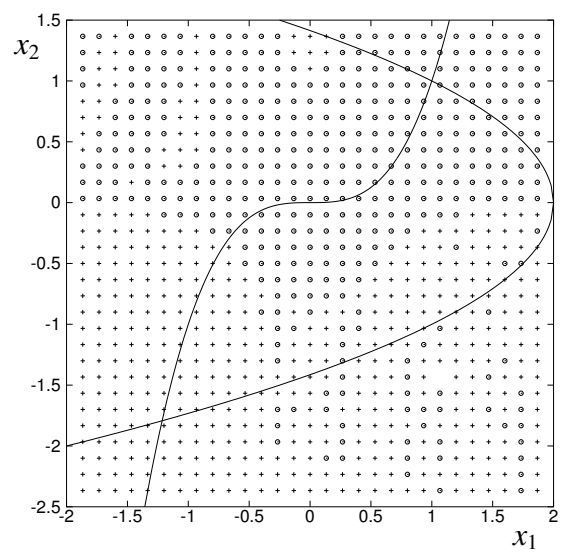
$$\mathbf{J}_{ij}(\mathbf{x}) = [\partial f_i(\mathbf{x})/\partial x_j],$$

a formula from which we can compute its value at a given point $\mathbf{x}^k$. For example, the nonlinear system on the left below has the Jacobian on the right.

$$\begin{array}{rclcl} f_1(\mathbf{x}) & = & x_1^3 - x_2 & = & 0 \\ f_2(\mathbf{x}) & = & x_1 + x_2^2 - 2 & = & 0 \end{array} \qquad \mathbf{J}(\mathbf{x}) = \begin{bmatrix} 3x_1^2 & -1 \\ 1 & 2x_2 \end{bmatrix}$$

To use Newton's method for systems I wrote the MATLAB routine `nteg.m` listed below.

```
function [x,k]=nteg(xzero)
% Newton's method for systems example
  x=xzero;
  f=zeros(2,1);
  epz=1e-12;
  for k=1:20
      f(1)=x(1)^3-x(2);
      f(2)=x(1)+x(2)^2-2;
      J(1,1)=3*x(1)^2;
      J(1,2)=-1;
      J(2,1)=1;
      J(2,2)=2*x(2);
      delta=inv(J)*(-f);
      x=x+delta;
      if(norm(delta) < epz) break; end
  end
end
```



The Octave session on the next page shows that from $\mathbf{x}^0 = [\frac{1}{2}, \frac{1}{2}]^\top$ the algorithm finds the root at $[1, 1]^\top$ and from $\mathbf{x}^0 = [-1, -1]^\top$ it finds the root near $[-1.21, -1.79]^\top$. I wrote another program to solve the problem starting from each point in the grid shown above, and marked

the point with a + or a o depending on which zero was returned. Nonlinear systems can have multiple roots, and to find a particular one we must start close enough to it. In some problems the algorithm diverges if the starting point is not close enough to a root.

```
octave:1> format long
octave:2> [x,k]=nteg([0.5;0.5])
x =

   1
   1

k =  7
octave:3> f1=x(1)^3-x(2)
f1 = 0
octave:4> f2=x(1)+x(2)^2-2
f2 = 0
octave:5> [x,k]=nteg([-1;-1])
x =

  -1.21486232248842
  -1.79300371513514

k =  6
octave:6> f1=x(1)^3-x(2)
f1 = 0
octave:7> f2=x(1)+x(2)^2-2
f2 =  4.44089209850063e-16
octave:8> quit
```

Steps 3>,4> and 6>,7> confirm that at each root both function values are zero.

21.2.2 Solving the LP Lagrange System Again

In §21.1 we could have used the general form of Newton's method for systems to solve these Lagrange conditions for the in1 problem [5, §14.1-14.2].

$$\begin{array}{lcrl}
\mathbf{f}_p(\mathbf{x}) & = & \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} & \text{primal feasibility, } m \text{ rows} \\
\mathbf{f}_d(\mathbf{y},\mathbf{s}) & = & \mathbf{A}^\top\mathbf{y} + \mathbf{s} - \mathbf{c} = \mathbf{0} & \text{dual feasibility, } n \text{ rows} \\
\mathbf{f}_c(\mathbf{x},\mathbf{s}) & = & s_j x_j - \mu = 0,\ j = 1 \ldots n & \text{complementary slackness, } n \text{ rows}
\end{array}$$

The variables in this system are $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, and $\mathbf{s} \in \mathbb{R}^n$. Its Jacobian therefore has $2n+m$ rows, each corresponding to a row in these equations, and $2n+m$ columns corresponding to the variables. We can describe the contents of this Jacobian succinctly by introducing the notation

$$\nabla^\top \mathbf{f}(\mathbf{x}) = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_r(\mathbf{x})^\top \end{bmatrix}$$

to represent the matrix whose r rows are the gradients of the functions making up the vector $\mathbf{f}$.

Using the definition given in §21.2.1, we can write the Jacobian for the Lagrange system as

$$\mathbf{J}(\mathbf{x}) = \begin{array}{c} \begin{array}{ccc} \mathbf{x} & \mathbf{y} & \mathbf{s} \end{array} \\ \begin{bmatrix} \nabla_{\mathbf{x}}^{\top}\mathbf{f}_p(\mathbf{x}) & \mathbf{0}_{m\times m} & \mathbf{0}_{m\times n} \\ \mathbf{0}_{n\times n} & \nabla_{\mathbf{y}}^{\top}\mathbf{f}_d(\mathbf{y}) & \nabla_{\mathbf{s}}^{\top}\mathbf{f}_d(\mathbf{s}) \\ \nabla_{\mathbf{x}}^{\top}\mathbf{f}_c(\mathbf{x}) & \mathbf{0}_{n\times m} & \nabla_{\mathbf{s}}^{\top}\mathbf{f}_c(\mathbf{s}) \end{bmatrix} \end{array} \begin{array}{l} \\ \text{primal feasibility} \\ \text{dual feasibility} \\ \text{complementary slackness} \end{array}$$

Each submatrix is easy to find if we examine a typical row in the corresponding matrix equation. The primal feasibility condition $\mathbf{Ax}-\mathbf{b} = \mathbf{0}$ has m rows and its row i is the equation on the left below, in which $A_i = [a_{i1} \dots a_{in}]$ is row i of $\mathbf{A}$.

$$f_i(\mathbf{x}) = A_i\mathbf{x} - b_i = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i = 0 \qquad \nabla_{\mathbf{x}} f_i(\mathbf{x}) = \begin{bmatrix} \partial f_i/\partial x_1 \\ \vdots \\ \partial f_i/\partial x_n \end{bmatrix} = \begin{bmatrix} a_{i1} \\ \vdots \\ a_{in} \end{bmatrix}$$

The gradient of this function is the column vector on the right, so $\nabla_{\mathbf{x}} f_i(\mathbf{x})^{\top} = A_i$ and

$$\nabla_{\mathbf{x}}^{\top}\mathbf{f}_p(\mathbf{x}) = \begin{bmatrix} \nabla_{\mathbf{x}} f_1(\mathbf{x})^{\top} \\ \vdots \\ \nabla_{\mathbf{x}} f_n(\mathbf{x})^{\top} \end{bmatrix} = \begin{bmatrix} A_1 \\ \vdots \\ A_n \end{bmatrix} = \mathbf{A}.$$

The dual feasibility condition $\mathbf{A}^{\top}\mathbf{y}+\mathbf{s}-\mathbf{c} = \mathbf{0}$ has n rows, and its row i is the equation below.

$$f_i(\mathbf{y}, \mathbf{s}) = a_{1i}y_1 + a_{2i}y_2 + \cdots + a_{mi}y_m + s_i - c_i = 0$$

The gradients of this function are

$$\nabla_{\mathbf{y}} f_i(\mathbf{y}) = \begin{bmatrix} \partial f_i/\partial y_1 \\ \vdots \\ \partial f_i/\partial y_m \end{bmatrix} = \begin{bmatrix} a_{1i} \\ \vdots \\ a_{mi} \end{bmatrix} \qquad \nabla_{\mathbf{s}} f_i(\mathbf{s}) = \begin{bmatrix} \partial f_i/\partial s_1 \\ \vdots \\ \partial f_i/\partial s_n \end{bmatrix} \text{ where } \frac{\partial f_i}{\partial s_j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

so

$$\nabla_{\mathbf{y}}^{\top}\mathbf{f}_d(\mathbf{y}) = \begin{bmatrix} \nabla_{\mathbf{y}} f_1(\mathbf{y})^{\top} \\ \vdots \\ \nabla_{\mathbf{y}} f_n(\mathbf{y})^{\top} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix} = \mathbf{A}^{\top}$$

and

$$\nabla_{\mathbf{s}}^{\top}\mathbf{f}_d(\mathbf{s}) = \begin{bmatrix} \nabla_{\mathbf{s}} f_1(\mathbf{s})^{\top} \\ \vdots \\ \nabla_{\mathbf{s}} f_n(\mathbf{s})^{\top} \end{bmatrix} = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix} = \mathbf{I}_{n\times n}.$$

The complementary slackness condition has n rows and its row i is the equation below.

$$f_i(\mathbf{x}, \mathbf{s}) = s_i x_i - \mu = 0$$

The gradients of this function are

$$\nabla_{\mathbf{x}} f_i(\mathbf{x}) = \begin{bmatrix} \partial f_i/\partial x_1 \\ \vdots \\ \partial f_i/\partial x_n \end{bmatrix} \text{ where } \frac{\partial f_i}{\partial x_j} = \begin{cases} s_i & i = j \\ 0 & i \neq j \end{cases}$$

and

$$\nabla_{\mathbf{s}} f_i(\mathbf{s}) = \begin{bmatrix} \partial f_i/\partial s_1 \\ \vdots \\ \partial f_i/\partial s_n \end{bmatrix} \text{ where } \frac{\partial f_i}{\partial s_j} = \begin{cases} x_i & i = j \\ 0 & i \neq j \end{cases}$$

so

$$\nabla_{\mathbf{x}}^{\top} \mathbf{f}_c(\mathbf{x}) = \begin{bmatrix} \nabla_{\mathbf{x}} f_1(\mathbf{x})^{\top} \\ \vdots \\ \nabla_{\mathbf{x}} f_n(\mathbf{x})^{\top} \end{bmatrix} = \begin{bmatrix} s_1 & & \\ & \ddots & \\ & & s_n \end{bmatrix} = \mathbf{S} \quad \text{and} \quad \nabla_{\mathbf{s}}^{\top} \mathbf{f}_c(\mathbf{s}) = \begin{bmatrix} \nabla_{\mathbf{s}} f_1(\mathbf{s})^{\top} \\ \vdots \\ \nabla_{\mathbf{s}} f_n(\mathbf{s})^{\top} \end{bmatrix} = \begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix} = \mathbf{X}.$$

Assembling the pieces we get the complete Jacobian

$$\mathbf{J}(\mathbf{x}, \mathbf{y}, \mathbf{s}) = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{\top} & \mathbf{I} \\ \mathbf{S} & \mathbf{0} & \mathbf{X} \end{bmatrix}$$

from which we can compute

$$\begin{bmatrix} \Delta\mathbf{x} \\ \Delta\mathbf{y} \\ \Delta\mathbf{s} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{\top} & \mathbf{I} \\ \mathbf{S} & \mathbf{0} & \mathbf{X} \end{bmatrix}^{-1} \begin{bmatrix} -\mathbf{f}_p(\mathbf{x}) \\ -\mathbf{f}_d(\mathbf{y}, \mathbf{s}) \\ -\mathbf{f}_c(\mathbf{x}, \mathbf{s}) \end{bmatrix}.$$

To compare this formula to the boxed equations of §21.1.2, I wrote the MATLAB program `ntdeltas.m` listed on the next page. After `3-13` specifying the `in1` problem data it finds the corrections `15-19` in the sequential fashion and then by constructing `23-28` the Jacobian and `29-31` the vector of function values and `33` using the formula derived above for the general form of the algorithm.

The Octave session below the listing shows that the two approaches do yield the same first set of corrections in solving the `in1` problem. If our `lpin.m` routine were revised to use the general form of Newton's method for systems (see Exercise 21.4.23) it would produce the same results we found before.


```
% ntdeltas.m: compare deltas found two ways
  mu=0.5;
  n=4;
  m=2;
  A=[-1,1,1,0;1,0,0,1];
  b=[1;1];
  c=[1;1;0;0];
  x=[0.5;0.5;1;0.5];
  y=[-0.5;-0.5];
  s=[1;1.5;0.5;0.5];
  X=diag(x);
  S=diag(s);
  D=inv(S)*X;

% use the boxed formulas of Section 21.1.2
  v=mu*ones(n,1)-X*S*ones(n,1);
  dy=-inv(A*D*A')*A*inv(S)*v;
  ds=-A'*dy;
  dx=inv(S)*v-D*ds;
  printf('%8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f\n',dx,dy,ds)

% use the general form of Newton's method for systems
  J=zeros(2*n+m,2*n+m);
  J(1:m,1:n)=A;
  J(m+1:m+n,n+1:n+m)=A';
  J(m+1:m+n,n+m+1:2*n+m)=eye(n,n);
  J(m+n+1:2*n+m,1:n)=S;
  J(m+n+1:2*n+m,n+m+1:2*n+m)=X;
  fp=A*x-b;
  fd=A'*y+s-c;
  fc=S*X*ones(n,1)-mu*ones(n,1);
  F=[fp;fd;fc];
  du=inv(J)*(-F);
  printf('%8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f %8.5f\n',du)
```

```
octave:1> ntdeltas
-0.16667 -0.16667 -0.00000  0.16667 -0.00000 -0.33333  0.33333  0.00000  0.00000  0.33333
-0.16667 -0.16667  0.00000  0.16667  0.00000 -0.33333  0.33333  0.00000  0.00000  0.33333
octave:2> quit
```

21.3 Interior-Point Methods for NLP

Several of the ideas we used in deriving an interior-point method for linear programming generalize naturally to the case where the functions are nonlinear [4, §16.7] [5, §19]. Our problem `b1`, which is restated on the left below, can be reformulated as shown on the right by adding nonnegative slack variables s_1 and s_2.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1 - 2x_2 & & \\ \text{subject to} & -x_1 + x_2^2 - 1 & \leq & 0 \\ & -x_2 & \leq & 0 \end{array} \qquad \begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2\ \mathbf{s}\in\mathbb{R}^2}{\text{minimize}} & x_1 - 2x_2 & & \\ \text{subject to} & -x_1 + x_2^2 - 1 + s_1 & = & 0 \\ & -x_2 + s_2 & = & 0 \\ & \mathbf{s} & \geq & \mathbf{0} \end{array}$$

Now we can form a barrier problem by moving the nonnegativities $\mathbf{s} \geq \mathbf{0}$ into the barrier function.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2\ \mathbf{s}\in\mathbb{R}^2_+}{\text{minimize}} & \beta(\mathbf{x},\mathbf{s};\mu) & = & x_1 - 2x_2 - \mu[\ln(s_1) + \ln(s_2)] \\ \text{subject to} & -x_1 + x_2^2 - 1 + s_1 & = & 0 \\ & -x_2 + s_2 & = & 0 \end{array}$$

This is an equality-constrained nonlinear program that we can solve using the Lagrange method. If we let λ_1 and λ_2 be the Lagrange multipliers associated with the equalities, then

$$\mathcal{L}(\mathbf{x},\mathbf{s},\boldsymbol{\lambda}) = x_1 - 2x_2 - \mu[\ln(s_1) + \ln(s_2)] + \lambda_1(-x_1 + x_2^2 - 1 + s_1) + \lambda_2(-x_2 + s_2)$$

and the Lagrange conditions are

$$\begin{array}{rcl} \nabla_{\mathbf{x}}\mathcal{L} & = & \begin{bmatrix} 1 - \lambda_1 \\ -2 + 2\lambda_1 x_2 - \lambda_2 \end{bmatrix} = \mathbf{0} \\ \nabla_{\mathbf{s}}\mathcal{L} & = & \begin{bmatrix} -\mu/s_1 + \lambda_1 \\ -\mu/s_2 + \lambda_2 \end{bmatrix} = \mathbf{0} \\ \nabla_{\boldsymbol{\lambda}}\mathcal{L} & = & \begin{bmatrix} -x_1 + x_2^2 - 1 + s_1 \\ -x_2 + s_2 \end{bmatrix} = \mathbf{0} \end{array}$$

For a given μ, we can solve these nonlinear algebraic equations numerically by using Newton's method for systems. If we multiply through by the denominators in the second set of equations and let

$$\mathbf{v} = \begin{bmatrix} \mathbf{x} \\ \mathbf{s} \\ \boldsymbol{\lambda} \end{bmatrix} \quad \text{or} \quad \begin{array}{rl} & [v_1,\ v_2,\ v_3,\ v_4,\ v_5,\ v_6]^\top \\ = & [x_1,\ x_2,\ s_1,\ s_2,\ \lambda_1,\ \lambda_2]^\top \end{array}$$

we can rewrite the Lagrange system like this.

$$\begin{array}{rcll} f_1(\mathbf{v}) & = & 1 - v_5 & = 0 \\ f_2(\mathbf{v}) & = & -2 + 2v_5v_2 - v_6 & = 0 \\ f_3(\mathbf{v}) & = & -\mu + v_3v_5 & = 0 \\ f_4(\mathbf{v}) & = & -\mu + v_4v_6 & = 0 \\ f_5(\mathbf{v}) & = & -v_1 + v_2^2 - 1 + v_3 & = 0 \\ f_6(\mathbf{v}) & = & -v_2 + v_4 & = 0 \end{array}$$

Then using $\mathbf{J}_{ij} = \partial f_i/\partial v_j$ the Jacobian is

$$\mathbf{J}(\mathbf{v}) = \begin{bmatrix} 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 2v_5 & 0 & 0 & 2v_2 & -1 \\ 0 & 0 & v_5 & 0 & v_3 & 0 \\ 0 & 0 & 0 & v_6 & 0 & v_4 \\ -1 & 2v_2 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

To solve the `b1` problem using this formulation I wrote the program `b1in.m` listed below and on the next page. The starting point must be chosen so that the Jacobian there is not singular, so for our problem we need

$$|\mathbf{J}(\mathbf{v})| = 2v_4v_5^2 + v_5v_6 \neq 0.$$

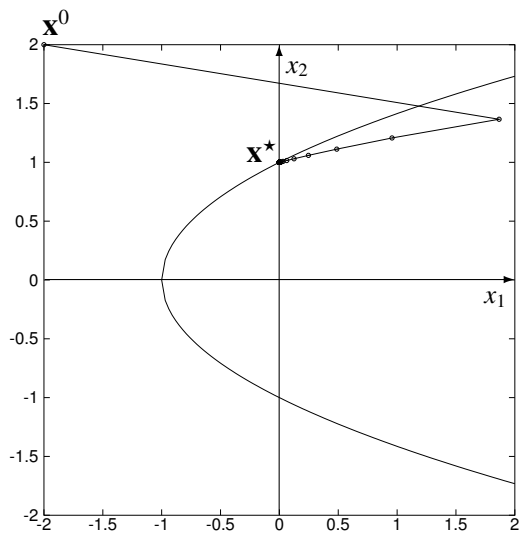
To satisfy this condition I set $v_4 = 0$ and $v_5 = v_6 = 1$. To show that $\mathbf{x}^0$ need not be feasible for the inequalities of the original problem I set $v_1 = x_1 = -2$ and $v_2 = x_2 = 2$, which violates the first constraint. Finally, to show that the equality constraints in the barrier problem need not be satisfied at the starting point I set $v_3 = s_1 = 0$ [5].

```
% b1in.m: interior-point solution of b1

clear; clf
xstar=[0;1];                % optimal x for finding error to plot
v=[-2;2;0;0;1;1];           % starting J must be nonsingular
mu=1;                       % starting barrier multiplier
for k=1:52                  % solve a sequence of barrier problems
    x(k)=v(1);              % save each iterate
    y(k)=v(2);              % for plotting later
    for t=1:10              % use Newton's method for systems
        J=zeros(6,6);       % start with a zero Jacobian
        J(1,5)=-1;          % and fill in the nonzero elements
        J(2,2)=2*v(5);
        J(2,5)=2*v(2);
        J(2,6)=-1;
        J(3,3)=v(5);
        J(3,5)=v(3);
        J(4,4)=v(6);
        J(4,6)=v(4);
        J(5,1)=-1;
        J(5,2)=2*v(2);
        J(5,3)=1;
        J(6,2)=-1;
        J(6,4)=1;

        F=zeros(6,1);                 % make F a column vector
        F(1)=1-v(5);                  % and fill in the function values
        F(2)=-2+2*v(5)*v(2)-v(6);
        F(3)=-mu+v(3)*v(5);
        F(4)=-mu+v(4)*v(6);
        F(5)=-v(1)+v(2)^2-1+v(3);
        F(6)=-v(2)+v(4);

        d=inv(J)*(-F);      % find the correction vector
        alpha=1;            % make sure s and lambda stay positive
        for j=3:6
            if(d(j) < 0) alpha=min(alpha,0.99999*(-v(j)/d(j))); end
        end
        v=v+alpha*d;        % take the restricted step
    end                     % Lagrange conditions solved for this mu
    mus(k)=mu;                        % remember mu
    xerr(k)=norm([v(1);v(2)]-xstar); % and the error in x
    kappa(k)=cond(J);                 % and the condition of J
    mu=mu/2;                % decrease mu
end                         % and continue
v                           % write the answer
```


For each value of μ in the sequence $1, \frac{1}{2}, \ldots, \frac{1}{2^{51}}$ 7-45 the program does 10 iterations of Newton's method for systems 10-40 to solve the Lagrange conditions of the barrier problem. Each iteration consists of 11-24 updating the Jacobian, 26-32 updating the function vector, 34 solving $\mathbf{Jd} = -\mathbf{F}$ for the correction d, 35-38 restricting the steplength, and 39 updating the estimate of the root. The $\mathbf{x}$ iterates 8-9, the μ values 41, the solution error 42, and the condition number of $\mathbf{J}$ 43 are all saved for plotting later.

Our formulation assumed that $\mathbf{s} > \mathbf{0}$ so that $\mathbf{x}$ is strictly feasible, and the second Lagrange condition requires that $\lambda_i = \mu/s_i > 0$, so the root that we want will have both $\mathbf{s}$ and $\boldsymbol{\lambda}$ positive. In the convergence trajectory on the right below 48-62 the first step from the infeasible start $[-2, 2]^\top$ is to such a point, and the steplength restriction ensures that both $\mathbf{s}$ and $\boldsymbol{\lambda}$ remain positive after that.

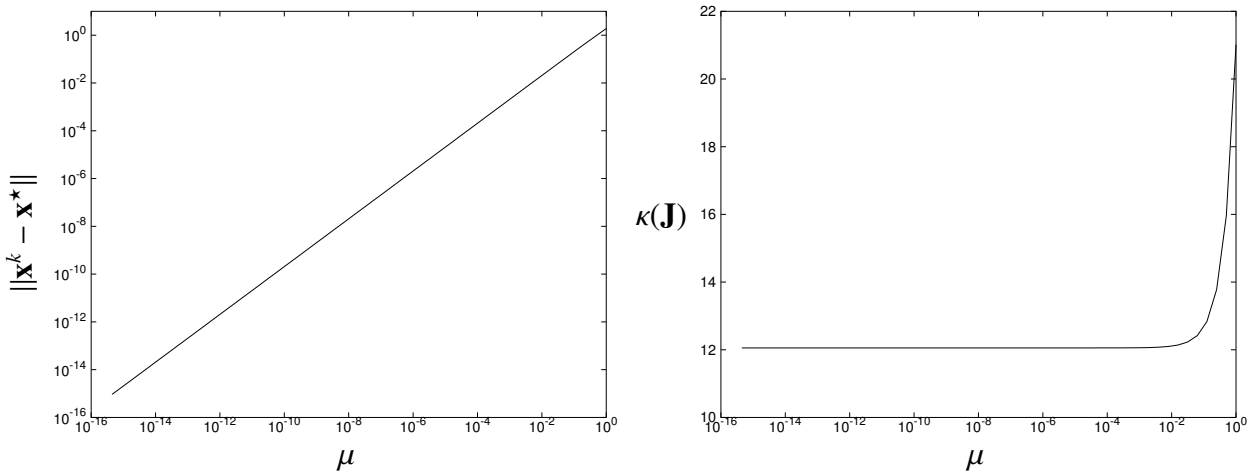
```
48 % plot the convergence trajectory
49 figure(1); set(gca,'FontSize',25)
50 hold on
51 axis([-2,2,-2,2],'square')
52 for p=1:101
53     xp(p)=-1+3*0.01*(p-1);
54     ypp(p)=sqrt(1+xp(p));
55     ypm(p)=-sqrt(1+xp(p));
56 end
57 plot(xp,ypp)  % second constraint upper branch
58 plot(xp,ypm)  % second constraint lower branch
59 plot(x,y,'o') % iterates
60 plot(x,y)     % connected by lines
61 hold off
62 print -deps -solid b1intrj.eps
63
64 % plot solution error
65 figure(2); set(gca,'FontSize',25)
66 hold on
67 axis([1e-16,1,1e-16,10])
68 loglog(mus,xerr)
69 hold off
70 print -deps -solid b1inerr.eps
71
72 % plot condition number of J
73 figure(3); set(gca,'FontSize',25)
74 hold on
75 axis([1e-16,1,10,22])
76 semilogx(mus,kappa)
77 hold off
78 print -deps -solid b1incnd.eps
```




```
octave:2> b1in
v =

   8.9638e-16
   1.0000e+00
   4.4409e-16
   1.0000e+00
   1.0000e+00
   4.4409e-16

octave:3> quit
```

The Octave session below the graph reports 46 a minimizing point $\mathbf{x} = [v_1, v_2]^\top$ close to $\mathbf{x}^\star = [0, 1]^\top$, positive slack variables $\mathbf{s} = [v_3, v_4]^\top \approx [0, 1]^\top$ satisfying the equality constraints of the barrier problem, and $\boldsymbol{\lambda} = [v_5, v_6]^\top \approx [1, 0]^\top = \boldsymbol{\lambda}^\star$.

The error curve on the left at the top of the next page shows that this algorithm has linear convergence like the classical barrier method of §19, but the graph on the right shows that $\mathbf{J}$, unlike the classical barrier Hessian, remains well-conditioned throughout the solution process.



21.3.1 A Primal-Dual Formulation

The same approach can be used to derive an interior-point algorithm for solving the standard-form nonlinear program on the left below. We begin by adding slack variables to obtain the equality-constrained problem on the right.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_i(\mathbf{x}) & \leq \; 0, \quad i = 1\ldots m \end{array} \qquad \begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n \;\; \mathbf{s}\in\mathbb{R}^m_+}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_i(\mathbf{x}) + s_i & = \; 0, \quad i = 1\ldots m \\ & \mathbf{s} & \geq \; \mathbf{0} \end{array}$$

The corresponding barrier problem is

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n \;\; \mathbf{s}\in\mathbb{R}^m_+}{\text{minimize}} & \beta(\mathbf{x},\mathbf{s};\mu) & = \; f_0(\mathbf{x}) - \mu \displaystyle\sum_{i=1}^m \ln(s_i) \\ \text{subject to} & f_i(\mathbf{x}) + s_i & = \; 0, \quad i = 1\ldots m \end{array}$$

which has the Lagrangian

$$\mathcal{L}(\mathbf{x},\mathbf{s},\boldsymbol{\lambda}) = f_0(\mathbf{x}) - \mu \sum_{i=1}^m \ln(s_i) + \sum_{i=1}^m \lambda_i [f_i(\mathbf{x}) + s_i]$$

and thus the following Lagrange conditions.

$$\mathbf{f}_p(\mathbf{x},\boldsymbol{\lambda}) = \nabla_{\mathbf{x}}\mathcal{L} = \nabla f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i \nabla f_i(\mathbf{x}) = \mathbf{0}$$

$$\nabla_{\mathbf{s}}\mathcal{L} = -\mu \begin{bmatrix} 1/s_1 \\ \vdots \\ 1/s_m \end{bmatrix} + \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{or} \quad \mathbf{f}_c(\mathbf{s},\boldsymbol{\lambda}) = -\mu \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} + \begin{bmatrix} s_1\lambda_1 \\ \vdots \\ s_m\lambda_m \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\mathbf{f}_d(\mathbf{x},\mathbf{s}) = \nabla_{\boldsymbol{\lambda}}\mathcal{L} = \begin{bmatrix} f_1(\mathbf{x}) + s_1 \\ \vdots \\ f_m(\mathbf{x}) + s_m \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

The Jacobian of this **primal-dual system** [5, p567] is

$$\mathbf{J}(\mathbf{x},\mathbf{s},\boldsymbol{\lambda}) = \begin{bmatrix} \nabla_{\mathbf{x}}^{\top}\mathbf{f}_p(\mathbf{x}) & \mathbf{0}_{n\times m} & \nabla_{\boldsymbol{\lambda}}^{\top}\mathbf{f}_p(\boldsymbol{\lambda}) \\ \mathbf{0}_{m\times n} & \nabla_{\mathbf{s}}^{\top}\mathbf{f}_c(\mathbf{s}) & \nabla_{\boldsymbol{\lambda}}^{\top}\mathbf{f}_c(\boldsymbol{\lambda}) \\ \nabla_{\mathbf{x}}^{\top}\mathbf{f}_d(\mathbf{x}) & \nabla_{\mathbf{s}}^{\top}\mathbf{f}_d(\mathbf{s}) & \mathbf{0}_{m\times m} \end{bmatrix}$$

in which (see Exercise 21.4.28) if we let $\boldsymbol{\Lambda}$ be the diagonal matrix whose diagonal elements are the λ_i,

$$\begin{aligned}
\nabla_{\mathbf{x}}^{\top}\mathbf{f}_p(\mathbf{x}) &= \mathbf{H}_{f_0}(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \mathbf{H}_{f_i}(\mathbf{x}) && = \texttt{Jpx} \\
\nabla_{\boldsymbol{\lambda}}^{\top}\mathbf{f}_p(\boldsymbol{\lambda}) &= \begin{bmatrix} \nabla f_1(\mathbf{x}) & \dots & \nabla f_m(\mathbf{x}) \end{bmatrix} && = \texttt{Jplambda} \\
\nabla_{\mathbf{s}}^{\top}\mathbf{f}_c(\mathbf{s}) &= \boldsymbol{\Lambda} && = \texttt{Jcs} \\
\nabla_{\boldsymbol{\lambda}}^{\top}\mathbf{f}_c(\boldsymbol{\lambda}) &= \mathbf{S} && = \texttt{Jclambda} \\
\nabla_{\mathbf{x}}^{\top}\mathbf{f}_d(\mathbf{x}) &= \begin{bmatrix} \nabla f_1(\mathbf{x})^{\top} \\ \vdots \\ \nabla f_m(\mathbf{x})^{\top} \end{bmatrix} && = \texttt{Jdx} \\
\nabla_{\mathbf{s}}^{\top}\mathbf{f}_d(\mathbf{s}) &= \mathbf{I} && = \texttt{Jds}.
\end{aligned}$$

To solve the standard-form nonlinear program using this primal-dual formulation I wrote the MATLAB function `nlpin.m` listed on the next page. This routine uses $\mathbf{v}^{\top} = [\mathbf{x}^{\top}, \mathbf{s}^{\top}, \boldsymbol{\lambda}^{\top}]$ so $\mathbf{x}$ = `v(1:n)`, $\mathbf{s}$ = `v(n+1:n+m)`, and $\boldsymbol{\lambda}$ = `v(n+m+1:n+2*m)`. For a starting point it $\boxed{6}$ sets $\mathbf{x} = \mathbf{x}^0$ but $\boxed{5}$ $\mathbf{s} = \mathbf{1}$ and $\boldsymbol{\lambda} = \mathbf{1}$ so that the $\mathbf{S}$ and $\boldsymbol{\Lambda}$ submatrices of $\mathbf{J}$ are identities. Then it does up to 52 iterations $\boxed{\texttt{8-53}}$ of the barrier algorithm, each of which uses just one iteration of Newton's method for systems to solve the Lagrange conditions. Each barrier iteration uses the formulas we derived above to construct $\boxed{\texttt{9-29}}$ the Jacobian $\mathbf{J}(\mathbf{v})$ and $\boxed{\texttt{31-42}}$ the vector $\mathbf{F}(\mathbf{v})$ of function values at the current point. Then $\boxed{44}$ it solves $\mathbf{Jd} = -\mathbf{F}$ for the direction $\mathbf{d}$, $\boxed{\texttt{45-48}}$ restricts the step if that is necessary to keep $\mathbf{s} > \mathbf{0}$ and $\boldsymbol{\lambda} > \mathbf{0}$, and $\boxed{49}$ updates the estimate of the solution. If the restricted step is small enough $\boxed{51}$ the current point is returned $\boxed{54}$ for `xstar`; otherwise $\boxed{52}$ μ is decreased and the iterations continue.

To test the algorithm I solved our `b1` and `b2` example problems, obtaining the results shown in the Octave session at the top of the page after the listing. Recall that `b2` has the same function, gradient, and Hessian routines as `p2`. These optimal points are exact, except for `xstar(1)` in the solution of `b1`, which should be zero but is always on the order of `epz` (I tried increasing the iteration limit beyond 52 but that made no difference).


```
function [xstar,k]=nlpin(xzero,m,epz,fcn,grd,hsn)
% solve a standard-form nonlinear program by a primal-dual interior point algorithm

  n=size(xzero,1);
  v=ones(n+2*m,1);
  v(1:n)=xzero;
  mu=1;
  for k=1:52
      Jpx=hsn(v(1:n),0);
        for i=1:m
            Jpx=Jpx+v(n+m+i)*hsn(v(1:n),i);
        end
      Jps=zeros(n,m);
      for i=1:m
          Jplambda(:,i)=grd(v(1:n),i);
      end
      Jcx=zeros(m,n);
      Jcs=zeros(m,m);
        for i=1:m
            Jcs(i,i)=v(n+m+i);
        end
      Jclambda=zeros(m,m);
        for i=1:m
            Jclambda(i,i)=v(n+i);
        end
      Jdx=Jplambda';
      Jds=eye(m,m);
      Jdlambda=zeros(m,m);
      J=[Jpx,Jps,Jplambda;Jcx,Jcs,Jclambda;Jdx,Jds,Jdlambda];

      F=zeros(2*m+n,1);
      F(1:n)=grd(v(1:n),0);
        for i=1:m
            F(1:n)=F(1:n)+v(n+m+i)*grd(v(1:n),i);
        end
      F(n+1:n+m)=-mu*ones(m,1);
        for i=1:m
            F(n+i)=F(n+i)+v(n+i)*v(n+m+i);
        end
      for i=1:m
        F(n+m+i)=fcn(v(1:n),i)+v(n+i);
      end

      d=inv(J)*(-F);
      alpha=1;
      for j=n+1:n+2*m
          if(d(j) < 0) alpha=min(alpha,0.99999*(-v(j)/d(j))); end
      end
      v=v+alpha*d;

      if(norm(d) <= epz) break; end
      mu=mu/2;
  end
  xstar=v(1:n);

end
```



```
octave:1> format long
octave:2> [xstar,k]=nlpin([-2;2],2,1e-15,@b1,@b1g,@b1h)
xstar =

   9.32124747758204e-16
   1.00000000000000e+00

k =  52
octave:3> [xstar,k]=nlpin([1;2],1,1e-15,@p2,@p2g,@p2h)
xstar =

   0.945582993415968
   0.894127197437503

k =  51
octave:4> quit
```

21.3.2 A Primal Formulation

In the Lagrange conditions that we derived above, the equation $\mathbf{f}_c(\mathbf{s},\boldsymbol{\lambda}) = \mathbf{0}$ says that $\lambda_i s_i = \mu$ for $i = 1 \ldots m$, but the equation $\mathbf{f}_d(\mathbf{x},\mathbf{s}) = \mathbf{0}$ says that $s_i = -f_i(\mathbf{x})$, so both of these conditions can be replaced by $\lambda_i f_i(\mathbf{x}) = -\mu$ or $\lambda_i = -\mu/f_i(\mathbf{x})$. Substituting this expression into the equation $\mathbf{f}_p(\mathbf{x},\boldsymbol{\lambda}) = \mathbf{0}$, the Lagrange conditions simplify to the **primal system**

$$\nabla f_0(\mathbf{x}) + \sum_{i=1}^{m} \left(\frac{-\mu}{f_i(\mathbf{x})} \right) \nabla f_i(\mathbf{x}) = \mathbf{0}.$$

These nonlinear algebraic equations have (see Exercise 21.4.33) the Jacobian

$$\mathbf{J}(\mathbf{x}) = \mathbf{H}_{f_0}(\mathbf{x}) - \mu \sum_{i=1}^{m} \left(\frac{f_i(\mathbf{x})\mathbf{H}_{f_i}(\mathbf{x}) - \nabla f_i(\mathbf{x}) \nabla f_i(\mathbf{x})^\top}{f_i(\mathbf{x})^2} \right)$$

which at $n \times n$ elements is smaller than the $(n+2m) \times (n+2m)$ one we found for the primal-dual formulation. This Jacobian is also symmetric, so it can be stored in $\frac{1}{2}n(n+1)$ memory locations rather than requiring even n^2. Further, if the original problem is a convex program then $\mathbf{J}$ is positive semidefinite, and if one or more of its functions happens to be strictly convex then $\mathbf{J}$ is positive definite. If $\mathbf{J}$ is positive definite and symmetric then efficient methods can be used to solve the linear system $\mathbf{Jd} = -\mathbf{F}$, such as Cholesky factorization if n is small or the conjugate gradient algorithm (see §14.4 and [5, p571]) if n is large. Thus, eliminating $\mathbf{s}$ and $\boldsymbol{\lambda}$ from the Lagrange system yields a formulation with some appealing properties.

Unfortunately it also introduces the complication that we can no longer keep $\mathbf{x}$ strictly feasible by keeping $\mathbf{s}$ strictly positive with a simple ratio test. The `nlpinp.m` routine listed on the next page instead uses a backtracking line search (see §19.1) to restrict the steplength. A more serious drawback of this approach is that now the starting point $\mathbf{x}^0$ must be strictly feasible, for only then can we be sure that the line search will keep each subsequent $\mathbf{x}^k$ interior to the feasible set.


```
function [xstar,k]=nlpinp(xzero,m,epz,fcn,grd,hsn)
% minimize f0(x) subject to fi(x)<=0 for i=1..m by a primal interior point algorithm

  x=xzero;
  mu=1;
  for k=1:52
      F=grd(x,0);
      for i=1:m
          F=F-mu*(grd(x,i)/fcn(x,i));
      end
      J=hsn(x,0);
      for i=1:m
          J=J-mu*(fcn(x,i)*hsn(x,i)-grd(x,i)*grd(x,i)')/(fcn(x,i)^2);
      end
      d=inv(J)*(-F);
      if(norm(d) <= epz) break; end
      alpha=1;
      for t=1:52
          ok=true;
          for i=1:m
              if(fcn(x+alpha*d,i) < 0) continue; end
              ok=false;
              break
          end
          if(ok) break; end
          alpha=alpha/2;
      end
      x=x+alpha*d;
      mu=mu/2;
  end
  xstar=x;

end
```

To test `nlpinp.m` I used it to solve `b1` and `b2` with the results shown below.

```
octave:1> format long
octave:2> [xstar,k]=nlpinp([0.5;0.5],2,1e-14,@b1,@b1g,@b1h)
xstar =

   2.81250581634848e-14
   1.00000000000001e+00

k =  47
octave:3> [xstar,k]=nlpinp([1;2],1,1e-14,@p2,@p2g,@p2h)
xstar =

   0.945582993415948
   0.894127197437538

k =  43
octave:4> [xstar,k]=nlpinp([1;2],1,1e-15,@p2,@p2g,@p2h)
warning: inverse: matrix singular to machine precision, rcond = 5.09993e-17
xstar =

   0.945582993415970
   0.894127197437508

k =  45
octave:5> quit
```


With `epz` = 10^{-14} the algorithm finds points that are very close to optimal for these problems, but tightening the tolerance further provokes a complaint about $\mathbf{J}$ being numerically singular! This formulation, because it eliminates the dual variables λ, results in a Jacobian that does *not* remain well-conditioned as $\mu \to 0$ [5, p571].

21.3.3 Accelerating Convergence

The barrier algorithms we have considered so far all have linear convergence. They set $\mu_{k+1} = \frac{1}{2}\mu_k$, and those that use a fraction-to-the-boundary rule restrict each step to go no closer than $\sigma = 0.99999$ of the way. If the interior-point method of §21.3.1 is modified to instead decrease μ towards 0 and increase σ towards 1 in a way that depends on the progress of the iterations, it is possible to get quadratic convergence, at least near the optimal point. Various complicated heuristics have been proposed [5, p572-573], but we will investigate a simple one [4, §16.7.2] that depends on a merit function.

A **merit function** [4, p513] [5, p575] is a scalar function $\phi(\mathbf{v})$ that measures how far a trial point $\mathbf{v}$ is from satisfying the optimality conditions. For example, in our primal-dual formulation the Lagrange conditions for the barrier problem are

$$\mathbf{F}(\mathbf{v}) = \begin{bmatrix} \mathbf{f}_p(\mathbf{v}) \\ \mathbf{f}_c(\mathbf{v}) \\ \mathbf{f}_d(\mathbf{v}) \end{bmatrix} = \mathbf{0}.$$

One measure of how far a given $\mathbf{v}$ is from satisfying them is $\|\mathbf{F}(\mathbf{v})\|$, because that norm is zero at a Lagrange point and increases if we move away. It is convenient for a merit function to be 1 at the starting point, so we will use $\phi(\mathbf{v}^k) = \|\mathbf{F}(\mathbf{v}^k)\|/\|\mathbf{F}(\mathbf{v}^0)\|$.

If certain other conditions are satisfied [4, Theorem 16.17] we can get second-order convergence by setting

$$\begin{aligned} \sigma_k &= \max\left\{\tfrac{1}{2}, 1 - \phi(\mathbf{v}^{k-1})\right\} \\ \mu_{k+1} &= \min\left\{\tfrac{1}{2}\phi(\mathbf{v}^k), \phi(\mathbf{v}^k)^2\right\}. \end{aligned}$$

The prescription for σ_k ensures that $\mathbf{x}^\star$ is approached from the interior of the feasible set, but permits $\mathbf{x}^k$ to get very close to the boundary when it is near $\mathbf{x}^\star$. In the formula for μ the first term is smaller near the starting point, when $\phi \approx 1$, but when $\phi < \frac{1}{2}$ the result of the min operation becomes the second term, so that μ decreases quadratically as $\mathbf{x}^\star$ is approached.

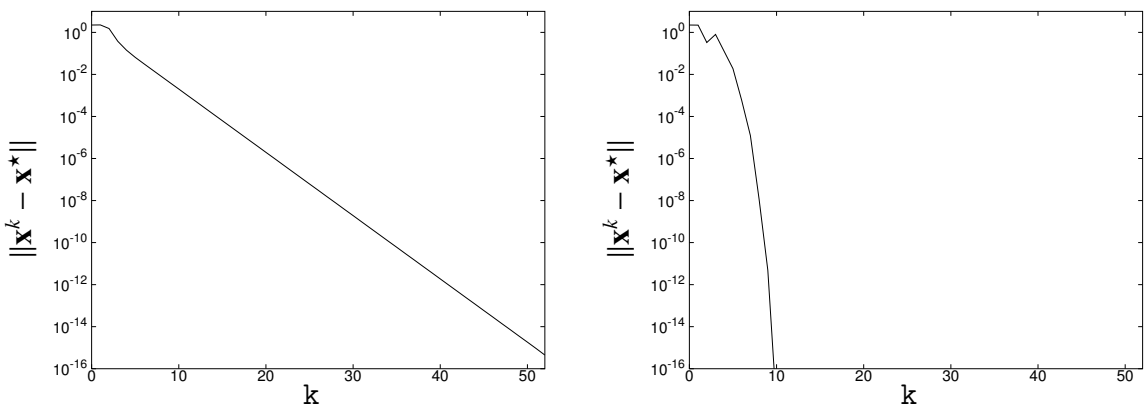
To try this idea I revised the `b1in.m` program of §21.3.0 to produce `b1inq.m`, which is listed on the next two pages. This program solves `b1` twice, first (`method=1`) using $\mu_{k+1} = \frac{1}{2}\mu_k$ and $\sigma = 0.99999$ `10,36,54` and again (`method=2`) using the scheme described above `10,38-41,56`. The same starting value of $\mu_1 = \frac{1}{2}$ is used `10` in both cases so that they can be compared. In each case the program does only one iteration of Newton's method for systems in each barrier iteration, which we found in `nlpin.m` is sufficient.


```
% b1inq.m: accelerated interior-point solution of b1

clear; clf
xstar=[0;1];

for method=1:2                        % try both strategies
    v=[-2;2;0;0;1;1];
    ks(1)=0;                          % starting point
    xerr(1)=norm([v(1);v(2)]-xstar);  % has this error
    mu=0.5;
    for k=1:52
        J=zeros(6,6);
        J(1,5)=-1;
        J(2,2)=2*v(5);
        J(2,5)=2*v(2);
        J(2,6)=-1;
        J(3,3)=v(5);
        J(3,5)=v(3);
        J(4,4)=v(6);
        J(4,6)=v(4);
        J(5,1)=-1;
        J(5,2)=2*v(2);
        J(5,3)=1;
        J(6,2)=-1;
        J(6,4)=1;

        F=zeros(6,1);
        F(1)=1-v(5);
        F(2)=-2+2*v(5)*v(2)-v(6);
        F(3)=-mu+v(3)*v(5);
        F(4)=-mu+v(4)*v(6);
        F(5)=-v(1)+v(2)^2-1+v(3);
        F(6)=-v(2)+v(4);

        if(method==1)
           sigma=0.99999;             % fixed fraction-to-boundary
        else
           phi=sqrt(F'*F);            % merit function
           if(k==1) mzero=phi; end    % remember first value
           phi=phi/mzero;             % and use it to normalize each value
           sigma=max(0.5,1-phi);      % contingent fraction-to-boundary
        end

        d=inv(J)*(-F);
        alpha=1;
        for j=3:6
            if(d(j) < 0) alpha=min(alpha,sigma*(-v(j)/d(j))); end
        end
        v=v+alpha*d;
        ks(k+1)=k;
        xerr(k+1)=norm([v(1);v(2)]-xstar);

        if(method==1)
           mu=mu/2;                   % fixed reduction in mu
        else
           mu=min(0.5*phi,phi^2);     % contingent reduction in mu
        end
    end
```



```
59      figure(method)                    % plot each error curve
60      axis([0,52,1e-16,1e1])
61      set(gca,'FontSize',30)
62      hold on
63      semilogy(ks,xerr)
64      hold off
65      switch(method)
66        case 1; print -deps -solid b1inl.eps
67        case 2; print -deps -solid b1inq.eps
68      end
```

This final stanza of `b1inq.m` plots the error curves below. Here `8,50,63` I have used the increasing iteration count `k`, rather than the decreasing multiplier value μ, as the independent variable. On the left, when $\mu_{k+1} = \frac{1}{2}\mu_k$, those quantities are related, but on the right, because μ_k depends on the progress of the algorithm rather than simply on `k`, they are not. The starting error is about 2.3; the lowest error achieved using the first method is on the order of 10^{-15}, that of the second on the order of 10^{-16} (perhaps because it allows a closer approach to the boundary).



The graph on the right clearly shows the superior performance of the `method=2` scheme, which could also be used to improve `nlpin.m` (see Exercise 21.4.35).

21.3.4 Other Variants

If $\mathbf{J}$ is a positive-definite matrix then $\mathbf{d} = \mathbf{J}^{-1}(-\mathbf{F})$ is a descent direction. For $\mathbf{J}$ to be positive definite it is necessary that $\mathbf{H}_{\mathcal{L}}(\mathbf{x})$ be positive definite, but depending on the problem that might not be true for some $\mathbf{x}$. In that case it is possible [4, p644] to add a multiple of the identity to that Hessian, as in the modified Newton algorithm, to make that submatrix of $\mathbf{H}$ positive definite. Another approach [5, p575-576] is to use a quasi-Newton approximation for $\mathbf{H}_{\mathcal{L}}$, which is sure to be positive definite and might be easier to calculate.

A different way to ensure progress toward optimality is by enforcing an Armijo condition (see §12.3.1) in the selection of α, so that each step achieves a sufficient decrease in the merit function [5, §19.4].

It is possible to include equality constraints in the primal-dual formulation [5, §19.2], at the price of losing its intuitive connection to the classical barrier algorithm (see §25.2).

21.4 Exercises

21.4.1 [E] A linear program in standard form is (see §2.1)

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

What is its standard-form dual?

21.4.2 [H] In §21.1.1 we defined the interiority condition. (a) What does it ensure? (b) In §19.0 we derived from the classical barrier problem a set of conditions that look like the KKT conditions for a nonlinear program, except that the condition corresponding to orthogonality sets the product to $-\mu$ rather than to zero. Can this condition be regarded as an interiority requirement? Explain.

21.4.3 [H] How does Newton's method for solving systems of nonlinear algebraic equations differ from the Newton descent algorithm?

21.4.4 [E] What MATLAB expression returns $\mathbf{1}_{n\times 1}$, a vector of all 1's? What does the MATLAB expression `s.*x` compute, and how does this differ from $\mathbf{s}^\top\mathbf{x}$?

21.4.5 [E] Tell the story of the interior-point method for linear programming, according to §21.1. What barrier problem did we use? How did we solve that barrier problem? What role is played by Newton's method for systems? How do we ensure that the algorithm will never generate an infeasible point? In your account try to convey the drama and suspense of the adventure as well as the awe and delight you experienced at its triumphant conclusion.

21.4.6 [E] In our interior-point algorithm for linear programming, what determines the order in which $\Delta\mathbf{y}$, $\Delta\mathbf{s}$, and $\Delta\mathbf{x}$ must be calculated?

21.4.7 [H] As $\mu \to 0$ in our interior-point algorithm for linear programming, $s_j x_j \to 0$ for $j = 1 \ldots n$. Show that this implies $\Delta x_j \Delta s_j \to 0$ for $j = 1 \ldots n$.

21.4.8 [E] The `lpin.m` routine of §21.1.3 is much simpler than the simplex method implementation of §4.1 (which consists of `simplex.m`, `phase0.m`, `phase1.m`, `phase2.m`, and `minr.m`), and it solves the dual at the same time it solves the primal. Why have interior-point methods not completely displaced the simplex method for solving linear programs?

21.4.9 [H] The linear algebra coded in `lpin.m` involves 4 explicit inverse calculations, which for reasons explained in §8.6.1 we would always prefer to avoid. Recast these calculations to use matrix factorizations and forward- and back-substitutions instead. Is it possible that any of the matrix factorizations might fail? Explain.

21.4.10 [H] If $\mathbf{x}$ solves a linear program $\mathscr{P}$ and $\mathbf{y}$ solves its dual $\mathscr{D}$ then $\mathbf{c}^\top\mathbf{x} = \mathbf{b}^\top\mathbf{y}$. Show that if the interior-point method of §21.1 is used to solve the linear program, the duality gap is given by $\mathbf{c}^\top\mathbf{x} - \mathbf{b}^\top\mathbf{y} = n\mu$.

21.4.11 [P] In §21.1.3 I used a modified version of `lpin.m` to plot the coordinates $x_1(\mu)$ and $x_2(\mu)$ generated by the algorithm in solving the standard-form version of the `in1` problem, obtaining a graph of the convergence trajectory. (a) Modify `lpin.m` to draw the graph. What makes an interior-point method a path-following method? (b) Modify `lpin.m` to plot $\kappa(\mathbf{ADA}^\top)$ as a function of μ, and obtain the condition-number graphs given for the `in1` and `brewery` problems.

21.4.12 [P] Determine experimentally the order of convergence of the interior-point method for linear programming.

21.4.13 [H] When `lpin.m` is used to solve the `brewery` problem in §21.1.3, it reports $\mathbf{y}^\star \approx [-7.5, 0, -18.75]^\top$. In §5.1.4 we learned that the dual variables are the shadow prices of the primal constraints, and for the `brewery` problem those are positive. What is the relationship between the $\mathbf{y}$ variables of the interior-point formulation and the shadow prices for malt, hops, and yeast? Explain.

21.4.14 [P] Use `lpin.m` to solve the following linear program [4, Example 10.1].

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -x_1 - 2x_2 & & \\ \text{subject to} & -2x_1 + x_2 & \leq & 2 \\ & -x_1 + 2x_2 & \leq & 7 \\ & x_1 + 2x_2 & \leq & 3 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

21.4.15 [P] If a linear program has multiple optimal solutions or is degenerate because its dual has multiple optimal solutions, the primal-dual interior-point method can converge to an interior point of the optimal set rather than to a vertex of the feasible set. (a) Show that when `lpin.m` is used to solve the `dp4` primal of §5.1.6, which has multiple optimal solutions, it converges to $\bar{\mathbf{x}} = [\frac{2}{3}, \frac{2}{3}, \frac{2}{3}]^\top$. In solving this problem the matrix $\mathbf{ADA}^\top$ becomes numerically singular $\boxed{16}$ before `k` reaches its limit of `52`, so to verify that the iterates approach $\bar{\mathbf{x}}$ you will need to modify the code to print out the iterates or to return the current iterate when the matrix inversion fails. (b) Confirm that $\bar{\mathbf{x}}$ is interior to the optimal set of this problem. (c) Why does $\mathbf{ADA}^\top$ become badly conditioned? (d) As mentioned in §4.5.3 a degenerate vertex can be made nondegenerate by perturbing the right-hand sides of the constraints that intersect there. Does doing this to the `dp4` problems prevent $\mathbf{ADA}^\top$ from becoming badly-conditioned when `lpin.m` is used to solve the primal? (e) Suggest an algorithm for recovering the basic feasible solution (of the unperturbed constraints) that is closest to $\bar{\mathbf{x}}$.

21.4.16 [P] The nonlinear algebraic equation $\sin(x) = \frac{1}{2}x$ has one root at zero and another near $x = 2$. (a) Use Newton's method to approximate the root near $x = 2$ by hand calculations. (b) Write a MATLAB program that uses Newton's method to find that root precisely. (c) Modify your program to find the root at $x = 0$.

21.4.17 [E] What problem does Newton's method for systems of equations solve? Describe the algorithm. Explain why the correction vector is the solution of a system of *linear* algebraic equations.

21.4.18 [E] What is a Jacobian matrix? Give a formula for the (i, j)'th element of a Jacobian. Is a Jacobian necessarily symmetric? Is it necessarily positive definite? Is it even necessarily nonsingular?

21.4.19 [P] This system of nonlinear algebraic equations [77, Example 5.11] has two real solutions.

$$\begin{aligned} x_1^2 - x_2 &= \tfrac{1}{2} \\ -x_1 + x_2^2 &= \tfrac{1}{2} \end{aligned}$$

(a) Write a MATLAB program that uses Newton's method for systems to find both roots. (b) Write a MATLAB program that produces for this problem a graph like the one in §21.2.1 showing for each point on a grid which zero Newton's method converges to.

21.4.20 [H] Show that Newton descent is a way of using Newton's method for systems to solve the system of equations represented by $\nabla f_0(\mathbf{x}) = \mathbf{0}$.

21.4.21 [E] In §21.2.2 I introduced the notation $\nabla^\top \mathbf{f}(\mathbf{x})$. What does it mean?

21.4.22 [E] In §21.2.2 two formulas are given for $\mathbf{J}(\mathbf{x})$. (a) Explain why the zero submatrices are zero, and why they have the dimensions given for them in the first formula. (b) Explain the general approach I used there to find the nonzero submatrices.

21.4.23 [P] Revise the `lpin.m` routine to use the general form of Newton's method for systems as described in §21.2.2, and verify that your new version produces the same solutions to the `in1` and `brewery` problems that we found in §21.1.3.

21.4.24 [E] In §21.3 we used an interior-point method to solve the `b1` problem. (a) Why was it necessary to add slacks? (b) Is the Jacobian that we found symmetric? (c) In using Newton's method for systems, what properties must the starting point have? (d) Why is it necessary to restrict the steplength? (e) What order of convergence does the algorithm have? (f) Does this algorithm have any advantages over the barrier method of §19?

21.4.25 [P] In §21.3 we derived an interior-point formulation of the `b1` problem. To obtain the Lagrange system I multiplied the equations $-\mu/s_i + \lambda_i = 0$ through by s_i and used $\mu + s_i\lambda_i = 0$ instead. Why did I do that? Hint: modify `b1in.m` to solve the problem using the equations in their original form, and study its behavior.

21.4.26 [H] In §21.3 we derived an interior-point formulation of the `b1` problem, and chose a starting point $\mathbf{v}^0$ such that $|\mathbf{J}(\mathbf{v})| = 2v_4v_5^2 + v_5v_6 \neq 0$. Compute the determinant of the Jacobian using expansion by minors (see §11.4.1) to confirm that it is given by this formula.

21.4.27 [E] Tell the story of the interior-point method for nonlinear programming according to §21.3.1, outlining all of the steps in the derivation of the algorithm.

21.4.28 [H] In §21.3.1 we added slacks to the standard-form NLP, constructed a corresponding barrier problem, and wrote its Lagrange system. Derive the formulas given there for the elements of the Jacobian of that Lagrange system.

21.4.29 [H] The `nlpin.m` routine of §21.3.1 uses the starting point $\mathbf{v}^0 = [\mathbf{x}^{0\top}, \mathbf{1}^\top, \mathbf{1}^\top]^\top$, so that the $\mathbf{S}$ and $\mathbf{\Lambda}$ submatrices of $\mathbf{J}$ are identities. Does this ensure that $\mathbf{J}$ is nonsingular? If not, what would ensure that?

21.4.30 [P] Use `nlpin.m` to solve the following problem, which I will call `ek1` (see §28.7.29).

$$\begin{array}{rrcll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1-20)^4+(x_2-12)^4 & \\ \text{subject to} & f_1(\mathbf{x}) & = & 8e^{(x_1-12)/9}-x_2+4 & \le\ 0 \\ & f_2(\mathbf{x}) & = & 6(x_1-12)^2+25x_2-600 & \le\ 0 \\ & f_3(\mathbf{x}) & = & -x_1+12 & \le\ 0 \end{array}$$

We will encounter this example again in §24.

21.4.31 [P] Use `nlpin.m` to solve the nonlinear programs of (a) Exercise 19.6.4; (b) Exercise 19.6.24; (c) Exercise 19.6.25.

21.4.32 [P] Use `nlpin.m` to solve the following inequality-constrained nonlinear programs: (a) the `arch2` problem of §16.0; (b) the `arch4` problem of §16.2; (c) the `moon` problem of §16.3; (d) the `cq1` problem of §16.7; (e) the `cq3` problem of §16.7; (f) the problem of Exercise 16.11.21.

21.4.33 [H] In §21.3.2, the Lagrange conditions derived in §21.3.1 are simplified to obtain the smaller primal system. (a) Explain this simplification. (b) Derive the formula for the Jacobian of the primal system. (c) Show that this Jacobian is symmetric. (d) List some advantages and drawbacks of this formulation. (e) Explain how `nlpinp.m` keeps each $\mathbf{x}^k$ feasible.

21.4.34 [E] What is a merit function? How does a merit function differ from a measure of solution error such as $\|\mathbf{x}^k-\mathbf{x}^\star\|/\|\mathbf{x}^0-\mathbf{x}^\star\|$? Suggest two possible merit functions that could be used to monitor the progress of an algorithm for nonlinear programming.

21.4.35 [P] The interior-point method of §21.3.1 has linear convergence, but if it is modified slightly the resulting algorithm can achieve quadratic convergence. (a) Describe the modifications that `b1inq.m` uses. Are these the only possible modifications that can lead to quadratic convergence? (b) Write `nlpinq.m` by modifying `nlpin.m` in the same way, and also to make it serially reusable (see §10.6.1). (c) Write a program that uses your `nlpinq.m` routine to solve `b1` one iteration at a time, and plot an error curve that agrees with the one that `b1inq.m` produced. (d) Try your `nlpinq.m` routine on the `ek1` problem described in Exercise 21.4.30. (e) Can the ideas of §21.3.3 be used to get quadratic convergence in the classical barrier algorithm of §19?

21.4.36 [P] Write a MATLAB routine `nlpinb.m` by modifying `nlpin.m` to use a BFGS approximation (see §13.4.3) in place of $\mathbf{H}_{\mathcal{L}}(\mathbf{x})$. Does your routine solve `b1` and `b2`?

21.4.37 [P] Write a MATLAB routine `nlpina.m` by modifying `nlpin.m` to impose an Armijo condition on α. Take the same approach that we used in imposing the sufficient decrease condition in `wolfe.m` (see §12.3.2). Try your routine on the problem of Exercise 19.6.4.

21.4.38 [H] Several of the programs available on the NEOS web server (see §8.3.1) are based on the algorithms discussed in this Chapter [5, §19.9]. By searching the web, find out which of the programs are based on which of the algorithms.

22

Quadratic Programming

In §14 we developed the conjugate gradient method for minimizing a quadratic objective, and studied its generalization to the Fletcher-Reeves and Polak-Ribière algorithms for the *unconstrained* minimization of arbitrary functions. The parameter estimation model of §8.5 and least-squares regression models of §8.6 are examples of unconstrained quadratic programs.

In general a quadratic program has a quadratic objective and linear constraints [5, §16.0] [1, §11.2]. Constrained quadratic programs arise in many practical applications, such as the SVM models of §8.7, and as subproblems in some methods for the *constrained* minimization of arbitrary functions, such as the reduced-Newton algorithm of §22.3 and the sequential quadratic programming and quadratic max penalty algorithms we will take up in §23.

Constrained quadratic programs are just nonlinear programs, so they can be solved by using the methods of §18 and §20.2 when the constraints are equations or by the methods of §19, §20.1, and §21 when they are inequalities. However, special-purpose algorithms have been devised to exploit the structure of the problem [5, §16] and they are simpler, faster, and more reliable than the general-purpose methods. Several of them, including **Lemke's method** [3, §9.8] [1, §11.2], the **symmetric indefinite factorization method**, and the **Shur-complement method**, are based on directly solving the KKT conditions, but we will consider a more general approach called the **nullspace method**.

22.1 Equality Constraints

The easiest constrained quadratic programs to solve are those in which the constraints are equalities. Consider the following example, which I will call `qp1` (see §28.7.30).

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & q(\mathbf{x}) & = & x_1^2 + x_2^2 + 2x_3^2 + 2x_4^2 + x_1x_4 + x_2x_3 \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \begin{bmatrix} 3x_1 - x_2 - 2x_3 - x_4 \\ -4x_1 + x_2 + 5x_3 + 2x_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \end{array}$$

The matrix $\mathbf{A}$ is the Jacobian of the constraints, and this one happens to have rows that are linearly independent so we can get a feasible starting point by finding a basic solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$. To do that I used the `pivot` program, as shown at the top of the next page. The final tableau corresponds to the basic solution $\bar{\mathbf{x}}$, shown below, so $\mathbf{A}\bar{\mathbf{x}} = \mathbf{b}$.

$$\bar{\mathbf{x}} = \begin{bmatrix} -2 \\ -5 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{A}\bar{\mathbf{x}} = \begin{bmatrix} 3 & -1 & -2 & -1 \\ -4 & 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} -2 \\ -5 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \checkmark$$


```
> This is PIVOT, Unix version 4.0
> For a list of commands, enter HELP.
>
< tableau 2 5
< names x1 x2 x3 x4

     x1  x2  x3  x4
 0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.

< insert
T( 1, 1)... = -1 3 -1 -2 -1
T( 2, 1)... = 3 -4 1 5 2

     x1  x2  x3  x4
-1.  3. -1. -2. -1.
 3. -4.  1.  5.  2.

< every
> Pivots will be allowed everywhere.
< pivot 1 2

            x1  x2         x3         x4
-0.3333333  1. -.33333333 -0.6666667 -.33333333
 1.6666667  0. -.33333333  2.3333333 0.66666667

< pivot 2 3

     x1  x2  x3  x4
-2.  1.  0. -3. -1.
-5.  0.  1. -7. -2.

< quit
> STOP
```

If we let $\mathbf{y} = \mathbf{x} - \bar{\mathbf{x}}$ then $\mathbf{Ay} = \mathbf{Ax} - \mathbf{A}\bar{\mathbf{x}} = \mathbf{b} - \mathbf{b} = \mathbf{0}$. The system $\mathbf{Ay} = \mathbf{0}$ is said to be **homogeneous** [87, p28] because it has a zero right-hand side. For reasons that will be apparent shortly it is convenient if the constraint equations are homogeneous, so I used

$$\mathbf{x} = \mathbf{y} + \bar{\mathbf{x}} \qquad \text{or} \qquad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 - 2 \\ y_2 - 5 \\ y_3 \\ y_4 \end{bmatrix}$$

to rewrite the $\mathbf{x}$ version of `qp1` in terms of $\mathbf{y}$, obtaining this version.

$$\begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^4}{\text{minimize}} & q(\mathbf{y}) & = & y_1^2 + y_2^2 + 2y_3^2 + 2y_4^2 + y_1y_4 + y_2y_3 - 4y_1 - 10y_2 - 5y_3 - 2y_4 + 29 \\ \text{subject to} & \mathbf{Ay} & = & \begin{bmatrix} 3y_1 - y_2 - 2y_3 - y_4 \\ -4y_1 + y_2 + 5y_3 + 2y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{array}$$

A quadratic program whose constraints are equalities can always be rewritten in this way if it is feasible, because then some $\bar{\mathbf{x}}$ satisfies $\mathbf{A}\bar{\mathbf{x}} = \mathbf{b}$.

22.1.1 Eliminating Variables

We can reformulate the **y** version of `qp1` as an unconstrained problem by using the constraint equations to write any two of the variables in terms of the others. Here `pivot` finds a solution to $\mathbf{Ay} = \mathbf{0}$ in which y_1 and y_2 are basic (compare this session to the earlier one).

```
> This is PIVOT, Unix version 4.0
> For a list of commands, enter HELP.
>
< tableau 2 5
< names y1 y2 y3 y4

     y1  y2  y3  y4
 0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.

< insert
T( 1, 1)... = 0 3 -1 -2 -1
T( 2, 1)... = 0 -4 1 5 2

     y1  y2  y3  y4
 0.  3. -1. -2. -1.
 0. -4.  1.  5.  2.

< every
> Pivots will be allowed everywhere.
< pivot 1 2

     y1  y2         y3         y4
 0.  1. -.33333333 -0.6666667 -.33333333
 0.  0. -.33333333  2.3333333 0.66666667

< pivot 2 3

     y1  y2  y3  y4
 0.  1.  0. -3. -1.
 0.  0.  1. -7. -2.

< quit
> STOP
```

The final tableau says that $y_1 = 3y_3 + y_4$ and $y_2 = 7y_3 + 2y_4$. Substituting these expressions into the objective yields this unconstrained problem.

$$\underset{y_3\ y_4}{\text{minimize}} \quad q(y_3, y_4) \quad = \quad 67y_3^2 + 8y_4^2 + 39y_3y_4 - 87y_3 - 26y_4 + 29$$

Then we can find a stationary point, which happens to be a minimum.

$$\left.\begin{array}{rcrcl} \dfrac{\partial q(y_3, y_4)}{\partial y_3} & = & 134y_3 + 39y_4 - 87 & = & 0 \\ \dfrac{\partial q(y_3, y_4)}{\partial y_4} & = & 39y_3 + 16y_4 - 26 & = & 0 \end{array}\right\} \Rightarrow \begin{array}{rcrcl} y_3^\star & = & \frac{54}{89} & \approx & 0.60674 \\ y_4^\star & = & \frac{13}{89} & \approx & 0.14607 \\ y_1^\star = 3y_3^\star + y_4^\star & = & \frac{175}{89} & \approx & 1.96629 \\ y_2^\star = 7y_3^\star + 2y_4^\star & = & \frac{404}{89} & \approx & 4.53933 \end{array}$$

That makes the optimal point for the **x** version of the problem

$$\mathbf{x}^\star = \mathbf{y}^\star + \bar{\mathbf{x}} = \begin{bmatrix} \frac{175}{89} \\ \frac{404}{89} \\ \frac{54}{89} \\ \frac{13}{89} \end{bmatrix} + \begin{bmatrix} -2 \\ -5 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -\frac{3}{89} \\ -\frac{41}{89} \\ \frac{54}{89} \\ \frac{13}{89} \end{bmatrix} \approx \begin{bmatrix} -0.03371 \\ -0.46067 \\ 0.60674 \\ 0.14607 \end{bmatrix}.$$

Substituting $y_1 = 3y_3 + y_4$ and $y_2 = 7y_3 + 2y_4$ confines the minimizing point of $q(\mathbf{y})$ to the flat defined by $\mathbf{Ay} = \mathbf{0}$, because then

$$\mathbf{Ay} = \begin{bmatrix} 3 & -1 & -2 & -1 \\ -4 & 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 3y_3 + y_4 \\ 7y_3 + 2y_4 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \mathbf{0}$$

no matter what values we pick for y_3 and y_4. That is why there is no need to explicitly enforce the constraints. Every vector that satisfies $\mathbf{Ay} = \mathbf{0}$ can be generated by assigning suitable values to y_3 and y_4, either in the formula above or in the linear combination

$$\mathbf{y} = y_3 \begin{bmatrix} 3 \\ 7 \\ 1 \\ 0 \end{bmatrix} + y_4 \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix} = y_3\mathbf{v} + y_4\mathbf{w} \qquad \text{where} \qquad \mathbf{v} = \begin{bmatrix} 3 \\ 7 \\ 1 \\ 0 \end{bmatrix} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix}.$$

The linearly independent vectors **v** and **w** form a basis for the nullspace of **A**, by which I mean that every vector **y** such that $\mathbf{Ay} = \mathbf{0}$ can be written as some linear combination of **v** and **w** (this idea was introduced in §15.5).

Basis vectors for the nullspace of **A** naturally emerge from the process of using the equality constraints to eliminate variables, as we just discovered, but they can also be calculated directly from **A**. This procedure [147, §2.4.2N] yields the same **v** and **w** we found above.

> Pivot in $\mathbf{Ay} = \mathbf{0}$ to produce $\mathbf{Uy} = \mathbf{0}$ where **U** has m identity columns (we did this above to figure out the formulas for eliminating y_1 and y_2). Then, in turn, give each nonbasic variable the value 1 while keeping the other nonbasic variables zero, and solve $\mathbf{Uy} = \mathbf{0}$ for the basic variables. The $n - m$ vectors produced in this way are a basis for the nullspace of **A**.

When we pivoted in **A** to find a basic solution we produced

$$\mathbf{U} = \begin{bmatrix} 1 & 0 & -3 & -1 \\ 0 & 1 & -7 & -2 \end{bmatrix}.$$

To follow the procedure, we let $y_3 = 1$ and $y_4 = 0$ and solve $\mathbf{Uy} = \mathbf{0}$ for y_1 and y_2.

$$\mathbf{Uy} = \begin{bmatrix} 1 & 0 & -3 & -1 \\ 0 & 1 & -7 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \mathbf{0} \qquad \begin{array}{rcl} y_1 - 3 & = & 0 \;\Rightarrow\; y_1 = 3 \\ y_2 - 7 & = & 0 \;\Rightarrow\; y_2 = 7 \end{array} \qquad \mathbf{v} = \begin{bmatrix} 3 \\ 7 \\ 1 \\ 0 \end{bmatrix}$$

Then we let $y_3 = 0$ and $y_4 = 1$ and solve $\mathbf{Uy} = \mathbf{0}$ for y_1 and y_2.

$$\mathbf{Uy} = \begin{bmatrix} 1 & 0 & -3 & -1 \\ 0 & 1 & -7 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \mathbf{0} \qquad \begin{array}{rcl} y_1 - 1 & = & 0 \;\Rightarrow\; y_1 = 1 \\ y_2 - 2 & = & 0 \;\Rightarrow\; y_2 = 2 \end{array} \qquad \mathbf{w} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$

These are the same basis vectors we found above by eliminating variables.

In §15.5 we used the MATLAB function `null()` to find a basis for the nullspace of a matrix. The Octave session on the next page does that for this example, obtaining a result `Z` whose $n - m$ columns are the basis vectors. These basis vectors, which I called `z1` and `z2`, have different values from the $\mathbf{v}$ and $\mathbf{w}$ we found above and they are **orthonormal**; their dot product is zero and they both have unit length so $\boxed{\texttt{5>}}$ $\mathbf{Z}^\top\mathbf{Z} = \mathbf{I}_{(n-m)\times(n-m)}$. Just as we can write any vector that satisfies $\mathbf{Ay} = \mathbf{0}$ as a linear combination of $\mathbf{v}$ and $\mathbf{w}$, we can also write any vector that satisfies $\mathbf{Ay} = \mathbf{0}$ as a linear combination of `z1` and `z2`. For example, we know that $\mathbf{y}^\star$ $\boxed{\texttt{6>}}$ is feasible so it must satisfy $\mathbf{Ay} = \mathbf{0}$, and it can be written $\boxed{\texttt{7>}}$ as a linear combination of `z1` and `z2` (see Exercise 22.4.22).

Because of the special structure of $\mathbf{v}$ and $\mathbf{w}$ we can deduce from them the formulas for y_1 and y_2 that we used above to eliminate those variables from $q(\mathbf{y})$. To use `z1` and `z2` to transform the $\mathbf{y}$ version of `qp1` into an unconstrained problem it is easier to use the fact, as we did in §15.5, that if linearly-independent vectors $\mathbf{z}^p \in \mathbb{R}^n$ form a basis for the nullspace of $\mathbf{A}$ then we can write any $\mathbf{y}$ that satisfies $\mathbf{Ay} = \mathbf{0}$ as some combination $t_1\mathbf{z}^1 + \ldots + t_{n-m}\mathbf{z}^{n-m}$ of those basis vectors. Because the $\mathbf{z}^p$ are the columns of the $n \times (n-m)$ matrix $\mathbf{Z}$, every $\mathbf{y}$ that is in the nullspace can be written as $\mathbf{y} = \mathbf{Zt}$ for some $\mathbf{t} \in \mathbb{R}^{n-m}$.

If we use a more compact notation for the $\mathbf{y}$ version of `qp1`,

$$\begin{array}{lrcl} \underset{\mathbf{y}\in\mathbb{R}^4}{\text{minimize}} & q(\mathbf{y}) & = & \frac{1}{2}\mathbf{y}^\top\mathbf{Qy} + \mathbf{c}^\top\mathbf{y} + d \\ \text{subject to} & \mathbf{Ay} & = & \mathbf{0} \end{array} \qquad \text{where} \quad \mathbf{Q} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 1 & 0 \\ 0 & 1 & 4 & 0 \\ 1 & 0 & 0 & 4 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} -4 \\ -10 \\ -5 \\ -2 \end{bmatrix} \quad d = 29$$

then we can substitute $\mathbf{y} = \mathbf{Zt}$ to obtain the unconstrained problem

$$\underset{\mathbf{t}\in\mathbb{R}^2}{\text{minimize}} \;\; q(\mathbf{t}) = \tfrac{1}{2}[\mathbf{Zt}]^\top\mathbf{Q}[\mathbf{Zt}] + \mathbf{c}^\top[\mathbf{Zt}] + d = \tfrac{1}{2}\mathbf{t}^\top[\mathbf{Z}^\top\mathbf{QZ}]\mathbf{t} + \mathbf{c}^\top\mathbf{Zt} + d.$$

Here $\mathbf{Q}$ is symmetric, $\mathbf{t}$ has dimension $n - m = 4 - 2 = 2$, the quantity $\mathbf{Z}^\top\mathbf{QZ}$ is called the **reduced Hessian** of $q(\mathbf{y})$ [5, p452], and $\mathbf{Ay} = \mathbf{AZt} = \mathbf{0}$ is satisfied for all $\mathbf{t}$ so $\mathbf{AZ} = \mathbf{0}$.


```
octave:1> A=[3,-1,-2,-1;-4,1,5,2];
octave:2> Z=null(A);
octave:3> z1=Z(:,1)
z1 =

   0.34929
   0.88961
   0.19104
  -0.22383

octave:4> z2=Z(:,2)
z2 =

   0.21732
   0.19840
  -0.23625
   0.92606

octave:5> Z'*Z
ans =

   1.0000e+00  -4.4615e-17
  -4.4615e-17   1.0000e+00

octave:6> ystar=[175/89;404/89;54/89;13/89]
ystar =

   1.96629
   4.53933
   0.60674
   0.14607

octave:7> 4.808244*z1+1.319865*z2
ans =

   1.96629
   4.53933
   0.60674
   0.14607

octave:8> quit
```

To solve this reduced problem numerically I wrote these routines to calculate the value and derivatives of $q(\mathbf{t})$

```
function f=qp1t(t)
  A=[3,-1,-2,-1;-4,1,5,2];
  Z=null(A);
  Q=[2,0,0,1;
     0,2,1,0;
     0,1,4,0;
     1,0,0,4];
  c=[-4;-10;-5;-2];
  d=29;
  f=0.5*t'*(Z'*Q*Z)*t+c'*Z*t+d;
end
```

```
function g=qp1tg(t)
  A=[3,-1,-2,-1;-4,1,5,2];
  Z=null(A);
  Q=[2,0,0,1;
     0,2,1,0;
     0,1,4,0;
     1,0,0,4];
  c=[-4;-10;-5;-2];
  g=zeros(2,1);
  g=(Z'*Q*Z)*t+(c'*Z)';
end
```

```
function H=qp1th(t)
  A=[3,-1,-2,-1;-4,1,5,2];
  Z=null(A);
  Q=[2,0,0,1;
     0,2,1,0;
     0,1,4,0;
     1,0,0,4];
  H=Z'*Q*Z;
end
```

and used plain Newton descent starting (arbitrarily) from $\mathbf{t}^0 = [0, 0]^\top$.


```
octave:1> [tstar,kp]=ntplain([0;0],10,1e-6,@qp1tg,@qp1th)
tstar =

   4.8082
   1.3199

kp =  2
octave:2> A=[3,-1,-2,-1;-4,1,5,2];
octave:3> Z=null(A);
octave:4> ystar=Z*tstar
ystar =

   1.96629
   4.53933
   0.60674
   0.14607

octave:5> xstar=ystar+[-2;-5;0;0]
xstar =

  -0.03371
  -0.46067
   0.60674
   0.14607

octave:6> quit
```

The reduced problem has a strictly convex objective and Newton descent minimizes a strictly convex quadratic in a single step, so `ntplain.m` returns `kp=2`.

22.1.2 Solving the Reduced Problem

We performed a complicated sequence of calculations in §22.1.1 to solve `qp1`, but it is easy to summarize what we did. First we found, by pivoting to a basic solution of $\mathbf{Ax} = \mathbf{b}$, a point $\bar{\mathbf{x}}$ that is feasible for the equality constraints. Then we let $\mathbf{y} = \mathbf{x} - \bar{\mathbf{x}}$ and rewrote the original quadratic program on the left as the one on the right.

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x} + d \\ \text{subject to} & \mathbf{Ax} = \mathbf{b} \end{array} \longrightarrow \begin{array}{ll} \underset{\mathbf{y}\in\mathbb{R}^n}{\text{minimize}} & q(\mathbf{y}) = \frac{1}{2}(\bar{\mathbf{x}} + \mathbf{y})^\top\mathbf{Q}(\bar{\mathbf{x}} + \mathbf{y}) + \mathbf{c}^\top(\bar{\mathbf{x}} + \mathbf{y}) + d \\ \text{subject to} & \mathbf{Ay} = \mathbf{0} \end{array}$$

Then we found $\mathbf{Z}$, whose columns are a basis for the nullspace of $\mathbf{A}$, and made the substitution $\mathbf{y} = \mathbf{Zt}$ to obtain this unconstrained minimization.

$$\underset{\mathbf{t}\in\mathbb{R}^{n-m}}{\text{minimize}} \quad q(\mathbf{t}) = \frac{1}{2}(\bar{\mathbf{x}} + \mathbf{Zt})^\top\mathbf{Q}(\bar{\mathbf{x}} + \mathbf{Zt}) + \mathbf{c}^\top(\bar{\mathbf{x}} + \mathbf{Zt}) + d$$

Finally, we used `ntplain.m` to minimize $q(\mathbf{t})$.

This process can be simplified by solving the unconstrained problem with a customized version of Newton descent.

At each iteration k we could find the gradient and Hessian of $q(\mathbf{t})$,

$$\begin{aligned}\nabla_{\mathbf{t}} q(\mathbf{t}^k) &= \mathbf{Z}^\top \mathbf{Q}(\bar{\mathbf{x}} + \mathbf{Z}\mathbf{t}^k) + \mathbf{Z}^\top \mathbf{c} \\ \mathbf{H}_q(\mathbf{t}^k) &= \mathbf{Z}^\top \mathbf{Q}\mathbf{Z},\end{aligned}$$

and solve $\mathbf{H}_q(\mathbf{t}^k)\,\mathbf{p}^k = -\nabla_{\mathbf{t}} q(\mathbf{t}^k)$ or

$$\mathbf{Z}^\top \mathbf{Q}\mathbf{Z}\mathbf{p}^k = -\mathbf{Z}^\top \mathbf{Q}(\bar{\mathbf{x}} + \mathbf{Z}\mathbf{t}^k) - \mathbf{Z}^\top \mathbf{c}$$

for the direction $\mathbf{p}^k$ of Newton descent in $\mathbf{t}$-space,

$$\mathbf{p}^k = -[\mathbf{Z}^\top \mathbf{Q}\mathbf{Z}]^{-1}[\mathbf{Z}^\top \mathbf{Q}(\bar{\mathbf{x}} + \mathbf{Z}\mathbf{t}^k) + \mathbf{Z}^\top \mathbf{c}].$$

The direction in $\mathbf{y}$-space, or in $\mathbf{x}$-space, corresponding to $\mathbf{p}^k$ is $\mathbf{d}^k = \mathbf{Z}\mathbf{p}^k$, and $\bar{\mathbf{x}} + \mathbf{Z}\mathbf{t}^k = \mathbf{x}^k$, so in terms of $\mathbf{x}^k$ this **reduced-Newton direction** [4, p550] is

$$\mathbf{d}^k = -\mathbf{Z}[\mathbf{Z}^\top \mathbf{Q}\mathbf{Z}]^{-1}\mathbf{Z}^\top[\mathbf{Q}\mathbf{x}^k + \mathbf{c}]$$

and by using it for the descent steps we can solve `qp1` without introducing either $\mathbf{y}$ or $\mathbf{t}$. To implement this idea I wrote the `qeplain.m` routine listed below.

```
function [xstar,kp]=qeplain(Q,c,A,xzero,kmax,epz)
% solve an equality-constrained quadratic program
  Z=null(A);
  Hinv=Z*(inv(Z'*Q*Z))*Z';
  xk=xzero;
  for kp=1:kmax
%     find the full Newton step on the flat
      d=-Hinv*(Q*xk+c);

%     take the step
      xk=xk+d;

%     test for convergence
      if(norm(d) <= epz) break; end
  end
  xstar=xk;
end
```

Here `qeplain.m` finds the same solution to `qp1` that we got using `ntplain.m`, and once again Newton descent requires only one iteration.

```
octave:1> xzero=[-2;-5;0;0];
octave:2> Q=[2,0,0,1;0,2,1,0;0,1,4,0;1,0,0,4];
octave:3> c=[0;0;0;0];
octave:4> A=[3,-1,-2,-1;-4,1,5,2];
octave:5> [xstar,kp]=qeplain(Q,c,A,xzero,10,1e-6)
xstar =

  -0.033708
  -0.460674
   0.606742
   0.146067

kp =  2
```


Unfortunately `qeplain.m` has several conspicuous shortcomings. The first is that it [4] computes the explicit inverse of a matrix, which is on principle always undesirable. As I first mentioned in §8.6.1, inverting a large matrix is expensive and likely imprecise. That is why, ever since §13.1, we have preferred Gauss elimination for solving square linear systems, such as $\mathbf{Hd} = -\mathbf{g}$ in Newton descent. The reduced Hessian $\mathbf{Z}^\top\mathbf{QZ}$ is $(n-m)\times(n-m)$, so in `qp1` it is only 2×2, but in a real application it might be much bigger and then it would be faster and more accurate to carry out the calculation of $\mathbf{Z}[\mathbf{Z}^\top\mathbf{QZ}]^{-1}\mathbf{Z}^\top$ by using the factor-and-solve approach. If $\mathbf{Z}^\top\mathbf{QZ}$ is positive definite we can find its Cholesky factors $\mathbf{U}^\top\mathbf{U}$ and write

$$\mathbf{Z}[\mathbf{Z}^\top\mathbf{QZ}]^{-1}\mathbf{Z}^\top = \mathbf{Z}[\mathbf{U}^\top\mathbf{U}]^{-1}\mathbf{Z}^\top = \mathbf{ZU}^{-1}\mathbf{U}^{-\top}\mathbf{Z}^\top = [\mathbf{ZU}^{-1}][\mathbf{ZU}^{-1}]^\top = \mathbf{VV}^\top$$

where $\mathbf{V} = \mathbf{ZU}^{-1}$. Then to find $\mathbf{V}$ we can solve the matrix equation $\mathbf{VU} = \mathbf{Z}$, which is easy because $\mathbf{U}$ is triangular. To see how, consider this example in which $n = 4$ and $m = 1$.

$$\mathbf{V}_{n\times(n-m)}\mathbf{U}_{(n-m)\times(n-m)} = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \\ v_{31} & v_{32} & v_{33} \\ v_{41} & v_{42} & v_{43} \end{bmatrix}\begin{bmatrix} 3 & 2 & 5 \\ 0 & 1 & 4 \\ 0 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 8 \\ 5 & 3 & 2 \\ 1 & 7 & 4 \\ 3 & 2 & 1 \end{bmatrix} = \mathbf{Z}_{n\times(n-m)}$$

$$\begin{aligned}
3v_{11} &= 2 &&\Rightarrow\quad v_{11} = 2/3 \\
3v_{21} &= 5 &&\Rightarrow\quad v_{21} = 5/3 \\
3v_{31} &= 1 &&\Rightarrow\quad v_{31} = 1/3 \\
3v_{41} &= 3 &&\Rightarrow\quad v_{41} = 1 \\
2v_{11} + 1v_{12} &= 4 &&\Rightarrow\quad v_{12} = (4 - 2v_{11})/1 = 8/3 \\
2v_{21} + 1v_{22} &= 3 &&\Rightarrow\quad v_{22} = (3 - 2v_{21})/1 = -1/3 \\
2v_{31} + 1v_{32} &= 7 &&\Rightarrow\quad v_{32} = (7 - 2v_{31})/1 = 19/3 \\
2v_{41} + 1v_{42} &= 2 &&\Rightarrow\quad v_{42} = (2 - 2v_{41})/1 = 0 \\
5v_{11} + 4v_{12} + 6v_{13} &= 8 &&\Rightarrow\quad v_{13} = (8 - 5v_{11} - 4v_{12})/6 = -1 \\
5v_{21} + 4v_{22} + 6v_{23} &= 2 &&\Rightarrow\quad v_{23} = (2 - 5v_{21} - 4v_{22})/6 = -5/6 \\
5v_{31} + 4v_{32} + 6v_{33} &= 4 &&\Rightarrow\quad v_{33} = (4 - 5v_{31} - 4v_{32})/6 = -23/6 \\
5v_{41} + 4v_{42} + 6v_{43} &= 1 &&\Rightarrow\quad v_{43} = (1 - 5v_{41} - 4v_{42})/6 = -2/3
\end{aligned}$$

If we perform the calculations in this order then, in turn, each

$$v_{ij} = \frac{z_{ij} - \sum_{k=1}^{j-1} u_{kj}v_{ik}}{u_{jj}}$$

where the summation is empty if $j = 1$. I wrote the `trislv.m` routine listed on the next page to carry out the steps for matrices of arbitrary size (the `k` loop is not executed if `j` is `1`).


```
function V=trislv(U,Z)
% solve VU=Z, where U is upper triangular, for V
  n=size(Z,1);
  m=n-size(Z,2)
  V=zeros(n,n-m);
  for j=1:n-m
      for i=1:n
          V(i,j)=Z(i,j);
          for k=1:j-1
              V(i,j)=V(i,j)-V(i,k)*U(k,j);
          end
          V(i,j)=V(i,j)/U(j,j);
      end
  end
end
```

This Octave session uses `trislv.m` and then the MATLAB `/` operator to produce the result we found by hand.

```
octave:1> U=[3,2,5;0,1,4;0,0,6];
octave:2> Z=[2,4,8;5,3,2;1,7,4;3,2,1];
octave:3> V=trislv(U,Z)
V =

   0.66667   2.66667  -1.00000
   1.66667  -0.33333  -0.83333
   0.33333   6.33333  -3.83333
   1.00000   0.00000  -0.66667

octave:4> V=Z/U
V =

   0.66667   2.66667  -1.00000
   1.66667  -0.33333  -0.83333
   0.33333   6.33333  -3.83333
   1.00000   0.00000  -0.66667

octave:5> quit
```

It would not make sense to write $\mathbf{V} = \mathbf{Z}/\mathbf{U}$ as a mathematical equation because these are matrices, but MATLAB carries out the command `V=Z/U` $\boxed{>4}$ by doing calculations like the ones performed by `trislv.m`. Thus we can replace the calculation of `Hinv` in `qeplain.m` by factoring `Z'*Q*Z` to get `U`, solving for `V=Z/U`, and finding `Hinv=V*V'`. If there are $m = 0$ rows in $\mathbf{A}$ and $\mathbf{b}$, so that we are seeking an unconstrained minimizing point of $q(\mathbf{x})$, we can still use this scheme by setting $\mathbf{Z} = \mathbf{I}_{n\times n}$. Then we will be factoring $\mathbf{Q}$ and `d=-Hinv*(Q*xk+c)` will be the unconstrained Newton descent step.

The second shortcoming of `qeplain.m` is that $\mathbf{Z}^\top\mathbf{Q}\mathbf{Z}$ (or $\mathbf{Q}$, if $m = 0$) might not be positive definite; then it either has no inverse or the resulting $\mathbf{d}$ is not a descent direction. Here is an example of a positive semidefinite quadratic program, which I will call `qp2` (see 28.7.31).

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1^2 \\ \text{subject to} & x_1 = 1 \end{array} \qquad \mathbf{Q} = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \mathbf{Z} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

This problem has $\mathbf{Z}^\top\mathbf{Q}\mathbf{Z} = 0$, but all points $[1, x_2]^\top$ are optimal and we need not give up on trying to find one of them. If the matrix we must factor is negative definite or indefinite (see Exercise 22.4.19) then the optimal value of the quadratic program is $-\infty$, but if it is positive semidefinite as in this case we might be able, by modifying it, to find a nonstrict local minimum [5, p454].

It also might happen that an equality-constrained quadratic program is not really an optimization problem at all. If the rows of $\mathbf{A}$ are linearly independent there can't be more than n of them, but there can be exactly n. Then the constraint equations are a square system as in this example, which I will call `qp3` (see 28.7.32). Now $n - m = 0$ so `null()` returns an empty matrix for `Z`, and the scheme we used in `qeplain.m` *cannot* be made to work.

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1^2 + 3x_2^2 \\ \text{subject to} & x_1 + x_2 = 4 \\ & 2x_1 - x_2 = 2 \end{array}$$

$$\begin{array}{c|cc} & x_1 & x_2 \\ \hline 0 & 0 & 0 \\ \hline 4 & 1 & 1 \\ 2 & 2 & -1 \end{array} \longrightarrow \begin{array}{c|cc} & x_1 & x_2 \\ \hline 0 & 0 & 0 \\ \hline 4 & 1 & 1 \\ -6 & 0 & -3 \end{array} \longrightarrow \begin{array}{c|cc} & x_1 & x_2 \\ \hline 0 & 0 & 0 \\ \hline 2 & 1 & 0 \\ 2 & 0 & 1 \end{array}$$

However, we can find $\mathbf{x}^\star$ as the unique solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$, which is also the only feasible point, by pivoting as shown above. If we use the `newseq.m` routine of §4.1 to do that, it will delete redundant rows and report if the equality constraints happen *not* to be consistent, and if $n > m$ it will yield a feasible starting point (in the same way that we found one by using `pivot` in §22.1.1) to spare the user the trouble of finding one.

Using these ideas I wrote the routine `qpeq.m` listed on the next page. Its long second stanza [10-40] finds a feasible starting point `xzero` and the inverse `Hinv` of the reduced Hessian. If there are constraints, [13] $\mathbf{A}$ and $\mathbf{b}$ are inserted into a tableau `T` along with a zero objective row and [14] `newseq.m` is used to find a feasible starting point. If `newseq.m` reports the problem infeasible [15-18] `qpeq.m` sets `rc=3` and resigns. If `newseq.m` succeeds, then `S(j)` is zero if x_j is nonbasic or the row index in `T` of the identity `1` for that column if x_j is basic. Using `S` the basic solution is extracted by [19-23] filling in its nonzero elements from the $\mathbf{b}$ part of `Tnew`. If $m = n$ [24-28] this starting point is returned as $\mathbf{x}^\star$. Otherwise [30] `Z` is found to span the nullspace of $\mathbf{A}$. If there are no constraints, `xzero` is the zero vector [11] and `Z` is [32] set to the identity as discussed above.

Next `qpeq.m` [34] invokes the `hfact.m` routine of §19.3 to factor the reduced Hessian, after modifying it if necessary. If `hfact.m` fails [35-38] `qpeq.m` [36] sets `rc=2` and [37] resigns. Otherwise [39] it uses the MATLAB `/` operator discussed above to find `V` and [40] calculates $\mathbf{Z}[\mathbf{Z}^\top\mathbf{Q}\mathbf{Z}]^{-1}\mathbf{Z}^\top$ as `Hinv=V'*V`.

Then, starting from $\mathbf{x}^0$ [43] the routine performs up to `kmax` iterations [44-52] of modified Newton descent on the flat defined by $\mathbf{A}\mathbf{x} = \mathbf{b}$. If the step `d` becomes shorter than the convergence tolerance [46-50] the current point `xk` is declared optimal [47] and the routine returns with `rc=0` [48] to indicate success. If `kmax` iterations are consumed without satisfying the convergence test, it [53] takes the current point as `xstar` and [54] sets `rc=1` to indicate that the iteration limit was met.


```
function [xstar,kp,rc,nm]=qpeq(Q,c,A,b,kmax,epz)
% minimize (1/2)x'Qx+c'x subject to Ax=b

% size up the problem
  n=size(Q,1);                    % number of variables
  m=size(A,1);                    % number of equality constraints
  kp=0;                           % no iterations yet
  nm=0;                           % no modifications yet

% find a starting point and the inverse of the reduced Hessian
  xzero=zeros(n,1);               % use the origin if unconstrained
  if(m > 0)                       % if there are constraints
     T=[0,zeros(1,n);b,A];                                  % tableau
     [Tnew,S,tr,mr,rc0]=newseq(T,m+1,n+1,[1:m+1],m+1);  % seek basis
     if(rc0 ~= 0)                                           % success?
        rc=3;                     % report constraints inconsistent
        return                    % and give up
     end
     for j=1:n                    % extract
         if(S(j) ~= 0)            % the basic solution
            xzero(j)=Tnew(S(j),1); % to use
         end                      % as the starting point
     end
     if(mr-1 == n)                % is the system square?
        xstar=xzero;              % if so this is the optimal point
        rc=0;                     % report success
        return                    % and return it
     end
     A=Tnew(2:mr,2:n+1);          % A without redundant constraints
     Z=null(A);                   % get a basis for the nullspace
  else                            % no constraints
     Z=eye(n);                    % Z=I makes Z'*Q*Z=Q
  end
  [U,rch,nm]=hfact(Z'*Q*Z,0.5);   % factor the reduced Hessian
  if(rch ~= 0)                    % success?
     rc=2;                        % report modification failed
     return                       % and give up
  end
  V=Z/U;                          % solve VU=Z
  Hinv=V*V';                      % find Z*inv(Z'QZ)*Z'

% do modified Newton descent in the flat defined by the constraints
  xk=xzero;                       % start here
  for kp=1:kmax                   % do up to kmax iterations
      d=-Hinv*(Q*xk+c);           % full reduced Newton step
      if(norm(d) <= epz)          % converged?
         xstar=xk;                % yes; save optimal point
         rc=0;                    % report success
         return                   % and return
      end
      xk=xk+d;                    % take the step
  end                             % of reduced Newton steps
  xstar=xk;                       % save the current point
  rc=1;                           % report out of iterations

end
```

In the Octave session on the next page, `qpeq.m` finds optimal points for the $\mathbf{x}$ version of `qp1`, the unconstrained objective of `qp1` in terms of y_3 and y_4, the positive-semidefinite `qp2` problem, and the `qp3` problem in which $\mathbf{A}$ is square.


```
octave;1> % qp1 x version
octave:1> Q=[2,0,0,1;0,2,1,0;0,1,4,0;1,0,0,4];
octave:2> c=[0;0;0;0];
octave:3> A=[3,-1,-2,-1;-4,1,5,2];
octave:4> b=[-1;3];
octave:5> [xstar,kp,rc,nm]=qpeq(Q,c,A,b,10,1e-6)
xstar =

  -0.033708
  -0.460674
   0.606742
   0.146067

kp =  2
rc = 0
nm = 0
octave:6> % qp1 unconstrained (y3,y4) version
octave:6> Q=[134,39;39,16];
octave:7> c=[-87;-26];
octave:8> A=[]; b=[];
octave:9> [ystar,kp,rc,nm]=qpeq(Q,c,A,b,10,1e-6)
ystar =

   0.60674
   0.14607

kp =  2
rc = 0
nm = 0
octave:10> % qp2
octave:10> Q=[2,0;0,0];
octave:11> c=[0;0];
octave:12> A=[1,0];
octave:13> b=[1];
octave:14> [xstar,kp,rc,nm]=qpeq(Q,c,A,b,10,1e-6)
xstar =

   1
   0

kp =  1
rc = 0
nm =  1
octave:15> % qp3
octave:15> Q=[2,0;0,6];
octave:16> A=[1,1;2,-1];
octave:17> b=[4;2];
octave:18> [xstar,kp,rc,nm]=qpeq(Q,c,A,b,10,1e-6)
xstar =

   2
   2

kp = 0
rc = 0
nm = 0
```

In the second solution `>6->9` `ystar` = $[y_3^\star, y_4^\star]^\top$. In the solution of `qp2` `10>-14>` one modification is made to the reduced Hessian so `nm=1` and the `xzero` found by `newseq.m` is optimal so `kp=1`. In the solution of `qp3` no minimization steps are needed so `kp=0`.

22.2 Inequality Constraints

In qp1 the constraints are equalities so both are active at optimality. For that problem we found in §22.1.2 the optimal point $\mathbf{x}^{=} = [-0.033708, -0.460674, 0.606742, 0.146067]^{\top}$, for which $q(\mathbf{x}^{=}) = 0.70787$. If we make the constraints inequalities instead we get the following problem, which I will call qp4 (see §28.7.33).

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & q(\mathbf{x}) & = & x_1^2 + x_2^2 + 2x_3^2 + 2x_4^2 + x_1x_4 + x_2x_3 \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \begin{bmatrix} 3x_1 - x_2 - 2x_3 - x_4 \\ -4x_1 + x_2 + 5x_3 + 2x_4 \end{bmatrix} \leq \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \end{array}$$

This problem has a different optimal point $\mathbf{x}^{\leq}$, at which the first constraint is tight while the second is slack. Knowing that the **active set** consists of only the first constraint, we can find that point by using qpeq.m as shown below. Allowing the optimal point to come unstuck from the boundary of the feasible set, which is now a polyhedron, and move interior to the second constraint reduces the optimal objective value $\boxed{\texttt{6>}}$ to $q(\mathbf{x}^{\leq}) = 0.067308$.

```
octave:1> Q=[2,0,0,1;0,2,1,0;0,1,4,0;1,0,0,4];
octave:2> c=zeros(4,1);
octave:3> Abar=[3,-1,-2,-1];
octave:4> bbar=[-1];
octave:5> [xineq,kp,rc,nm]=qpeq(Q,c,Abar,bbar,10,1e-6)
xineq =

  -0.250000
   0.038462
   0.057692
   0.096154

kp =  2
rc = 0
nm = 0
octave:6> q=0.5*xineq'*Q*xineq
q =  0.067308
```

In solving qp4 with qpeq.m I just left out the row of $\mathbf{A}$ and the row of $\mathbf{b}$ corresponding to the constraint that is slack at optimality. We could solve any quadratic program with linear inequality constraints in this way, if only we knew ahead of time what its active set was going to be. There are m rows in $\mathbf{A}$ and $\mathbf{b}$, so the number of possible active sets is [116, A.2.4(18)]

$$\sum_{k=0}^{m} \binom{m}{k} = 2^m.$$

Recall from §16.1 that the KKT orthogonality condition provides us with an automatic way of figuring out, in the process of finding $\mathbf{x}^{\star}$ analytically, whether an inequality constraint is active or inactive at the optimal point. Assuming that none of the constraints

are redundant, if $\lambda_i^\star > 0$ then constraint i is tight and if $\lambda_i^\star = 0$ then constraint i is slack. In the KKT method we try all 2^m possible ways of making some KKT multipliers zero and the others nonzero. Here we will describe each such combination by its **working set** $\mathcal{W} = [w_1, w_2, \ldots, w_m]$, a vector of flags in which $w_i = 1$ if constraint i is tight and $w_i = 0$ if it is slack. For `qp4` (or any problem having $m = 2$ inequality constraints) these are the possible working sets.

$$\mathcal{W}_0 = [0,0] \qquad \mathcal{W}_1 = [0,1] \qquad \mathcal{W}_2 = [1,0] \qquad \mathcal{W}_3 = [1,1]$$

The subscripts on $\mathcal{W}$ identifying these working sets are the decimal values of their bit strings and are the case numbers that we would use in solving the problem by the KKT method.

If inequality i will be slack at $\mathbf{x}^\star$ but, not knowing that ahead of time, we assume it is an equality by insisting that $\lambda_i \neq 0$, then *if we find a feasible stationary point* the corresponding λ_i comes out *negative* [5, p470] [4, p565]. In terms of the resource-allocation model of optimization, the shadow price of such a constraint is negative because if we allow some of the corresponding resource to not be used that permits a different feasible solution, which uses more of some other resource and thereby yields higher revenue. We should remove this **sticking constraint** from the working set so that the optimal point is allowed to move interior to the feasible region rather than being stuck to its boundary.

If inequality i will be tight at $\mathbf{x}^\star$ but we assume it is slack and take it out of the problem by insisting that $\lambda_i = 0$, then the stationary point we find violates the ignored constraint. This happened for CASE 2 of the `moon` problem solution in §16.3, where $\mathbf{x} = [-1, 0]^\top$ violates the second constraint. If this happens we should add that **blocking constraint** [5, p469] to the working set so the optimal point is not allowed to move outside of the feasible region.

These observations suggest a strategy, outlined on the next page, for finding the active set and in the process $\mathbf{x}^\star$ and $\boldsymbol{\lambda}^\star$.

The feasible starting point required by stanza 1 of the algorithm could be an interior point, but it is easier to start at a boundary point (perhaps a vertex) as described in §22.2.1. If upon entering stanza 2 there are n tight constraints then the equalities in the working set are a square system whose solution is $\mathbf{x}^k$, and no Newton step can be taken. This cannot happen at $\mathbf{x}^0$ because the working set is initialized to empty. In §22.2.2 we will derive a steplength rule that keeps $\mathbf{x}^{k+1}$ feasible for the inactive inequalities. If $q(\mathbf{x})$ is not strictly convex on the flat defined by the working set, then more than one Newton descent step might be needed to find a minimizing point precisely [4, p569-570]. However, it is often sufficient to take a single step between updates of $\mathcal{W}$ [5, p477-478] so for simplicity that is what we will do (see Exercise 22.4.42). In stanza 3, if $w_i = 0$ then $\lambda_i = 0$ but if $w_i = 1$ then λ_i satisfies the Lagrange conditions for the equality-constrained subproblem; in §22.2.3 we will derive a formula for finding those nonzero λ_i. If there is a redundant constraint then it might be that more than n inequalities are tight at a vertex. Including them all in the working set would make the equality constraints of the subproblem an overdetermined system, greatly complicating implementation, so when blocking constraints are activated in stanza 5 we will take care not to end up with more than n of them.

1. Find a point $\mathbf{x}^0$ that satisfies $\mathbf{Ax} \le \mathbf{b}$.
   Initialize $\mathcal{W}$ by setting $w_i = 0$ for $i = 1 \ldots m$.
   set $k = 0$.
2. Find $\mathbf{x}^{k+1}$ by taking one Newton step toward minimizing $q(\mathbf{x})$ subject to
   $\mathbf{x}$ being in the flat defined by the tight constraints and
   $\mathbf{x}$ remaining feasible for the slack constraints;
   let $k \leftarrow k + 1$.
3. Compute the Lagrange multipliers $\boldsymbol{\lambda}^k$ at $\mathbf{x}^k$.
4. Release sticking constraints by updating $\mathcal{W}$
   for each i with $w_i = 1$ and $\lambda_i \le 0$, let $w_i = 0$.
5. Activate blocking constraints by updating $\mathcal{W}$
   for each i with $w_i = 0$, if the constraint is tight
   and moving farther in the Newton direction would violate it,
   let $w_i = 1$.
6. Test for convergence: if $\mathcal{W}$ changed GO TO 2;
   otherwise $\mathcal{W}$ is the active set,
   $\mathbf{x}^\star = \mathbf{x}^k$ is the optimal point, and
   $\boldsymbol{\lambda}^\star = \boldsymbol{\lambda}^k$ is the vector of optimal KKT multipliers.

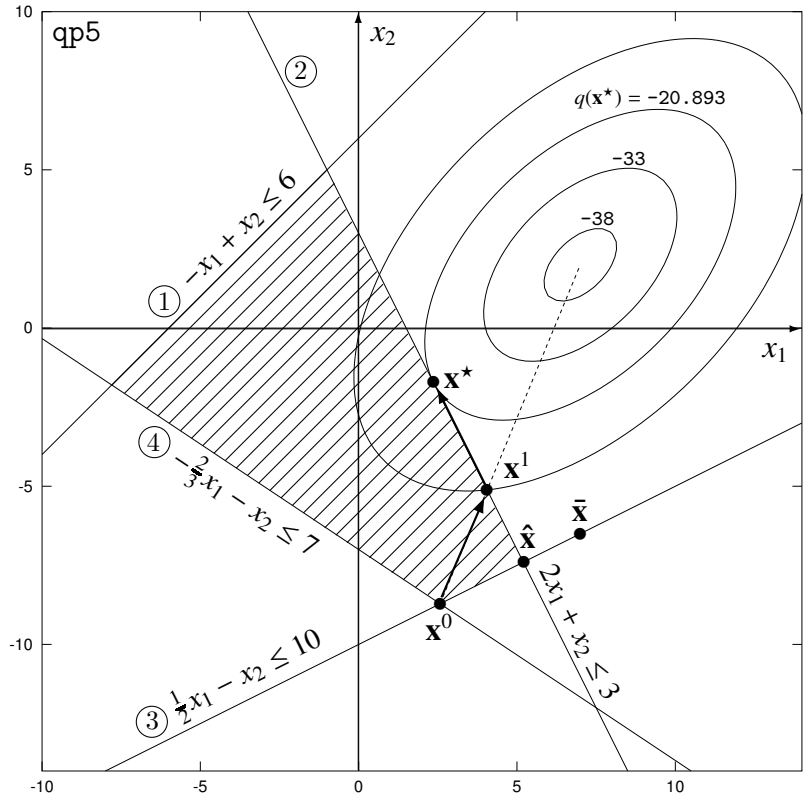
22.2.1 Finding a Feasible Starting Point

For the algorithm outlined above to work it is essential that its starting point be feasible. As in `qpeq.m` we can use the machinery of linear programming to find such a point, but because the constraints are now inequalities the process is quite a bit more complicated. Consider the following example, which I will call `qp5` (see §28.7.34) and whose graphical solution is shown on the next page.

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x} \\ \text{subject to} & \mathbf{Ax} \le \mathbf{b} \end{array}$$

$$\mathbf{Q} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} -12 \\ 3 \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} -1 & 1 \\ 2 & 1 \\ \frac{1}{2} & -1 \\ -\frac{2}{3} & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 6 \\ 3 \\ 10 \\ 7 \end{bmatrix}.$$

Now the x_j, which are unrestricted in sign, must each be written as the difference between nonnegative variables. Recall from §2.9.3 that this can be accomplished by introducing a single new variable $t \ge 0$ and using the substitution $\mathbf{x} = \mathbf{u} - t\mathbf{1}$. Adding slack variables s_i to make the constraints equalities, they become $\mathbf{Au} - t\mathbf{A1} + \mathbf{s} = \mathbf{b}$.



These equalities are the constraint rows in this tableau for qp5.

$$\mathtt{T} = \begin{array}{c|ccc|c|ccc|}
 & u_1 & \cdots & u_n & t & s_1 & \cdots & s_m \\
\hline
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\hline
\mathbf{b} & & \mathbf{A} & & \mathbf{-A1} & & \mathbf{I} & \\
\hline
\end{array} = \begin{array}{c|ccccccc|c}
 & u_1 & u_2 & t & s_1 & s_2 & s_3 & s_4 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
\hline
6 & -1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
3 & 2 & 1 & -3 & 0 & 1 & 0 & 0 & 2 \\
10 & \frac{1}{2} & -1 & \frac{1}{2} & 0 & 0 & 1 & 0 & 3 \\
7 & -\frac{2}{3} & -1 & 1\frac{2}{3} & 0 & 0 & 0 & 1 & 4 \\
\hline
\end{array}$$

The tableau has an objective row because one is expected by our linear programming routines, but we are concerned only with the constraints. Pivoting in T to a basic feasible solution in which $n = 2$ slack variables are nonbasic yields a vertex of the feasible set.

$$\mathtt{T1} = \begin{array}{r|rrrrrrr}
 & u_1 & u_2 & t & s_1 & s_2 & s_3 & s_4 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
11.28571 & 1 & -1 & 0 & 0 & 0 & 1.42857 & -0.42857 \\
8.71429 & 0 & -1 & 1 & 0 & 0 & 0.57143 & 0.42857 \\
17.28571 & 0 & 0 & 0 & 1 & 0 & 1.42857 & -0.42857 \\
6.57143 & 0 & 0 & 0 & 0 & 1 & -1.14286 & 2.14286 \\
\hline
\end{array}$$

$u_1 = 11.28571$
$u_2 = 0$
$t = 8.71429$
$x_1^0 = u_1 - t = 2.57143$
$x_2^0 = u_2 - t = -8.71429$

Tableau `T1` delivers the starting point $\mathbf{x}^0 = [2.57143, -8.71429]^\top$, which is in exact arithmetic $\mathbf{x}^0 = [\frac{18}{7}, -\frac{61}{7}]^\top$.

To automate these calculations I wrote the MATLAB routine `feas.m` listed below. It begins 4-7 by constructing the tableau `T` according to the prescription given above. Next it uses the 8-10 `newseq.m` and 15 `phase1.m` routines of §4.1 to produce `T1`, in which the basis columns are as far left as possible. Then 20-21 it extracts the value of `t` from the result. The basic sequence of `T1` is returned in `S1`; its entry `S1(j)` is zero if the jth variable is nonbasic or the row number in `T1` of the 1 in that identity column if the variable is basic. The added `t` is always the $n+1$st variable, so `S1(n+1)` tells whether it is basic. In the final `qp5` tableau, `t` is basic with its identity column 1 in the second constraint row (`S1(3)=3`), so its value is b_2 or `T1(3,1)`. This `t` is used to 22 initialize every x_j^0. Then 23-27 the values of the basic u_j are extracted from `T1` and added to the x_j^0 to produce $\mathbf{x}^0 = -t\mathbf{1} + \mathbf{u}$ for return.

```
 1 function [xzero,rc]=feas(A,b)
 2 % find a point that satisfies Ax <= b
 3
 4   m=size(A,1);                                    % constraints
 5   n=size(A,2);                                    % variables
 6   nc=1+n+1+m;                                     % columns
 7   T=[zeros(1,nc);b,A,-A*ones(n,1),eye(m)];        % form tableau
 8   mr=1+m;                                         % rows
 9   tr=[1:mr];                                      % row indices
10  [T0,S0,trnew,mrnew,rc0]=newseq(T,mr,nc,tr,mr);   % move basis left
11   if(rc0 ~= 0)                                    % infeasible 1?
12      rc=1;                                        % signal failure
13      return                                       % and give up
14   end
15  [T1,S1,rc1]=phase1(T0,S0,mrnew,nc,trnew,mrnew);  % find feasible
16   if(rc1 ~= 0)                                    % infeasible 2?
17      rc=2;                                        % signal failure
18      return                                       % and give up
19   end
20   t=0;                                            % zero if nonbasic
21   if(S1(n+1) ~= 0) t=T1(S1(n+1),1); end           % this if basic
22   xzero=-t*ones(n,1);                             % x=-te
23   for j=1:n                                       % decision vars
24       if(S1(j) ~= 0)                              % basic?
25          xzero(j)=xzero(j)+T1(S1(j),1);           % x=-te+u
26       end
27   end
28   rc=0;                                           % signal success
29
30 end
```

In the Octave session below I used `feas.m` to find `xzero` for `qp5`.

```
octave:1> A=[-1,1;2,1;1/2,-1;-2/3,-1];
octave:2> b=[6;3;10;7];
octave:3> xzero=feas(A,b)
xzero =

   2.5714
  -8.7143

octave:4> quit
```


It is possible for the system of inequalities $\mathbf{Ax} \le \mathbf{b}$ to be infeasible even though $\mathbf{x}$ is free, so `feas.m` traps both `11-14` infeasible form 1 (see Exercise 22.4.32) and `16-19` infeasible form 2. For example, these **inconsistent inequalities**

$$\begin{aligned} 2x_1 + 3x_2 &\le -5 \\ -2x_1 - 3x_2 &\le -5 \end{aligned}$$

cannot both be satisfied, and `feas.m` reports that fact by returning a nonzero `rc` value.

```
octave:1> A=[2,3;-2,-3];
octave:2> b=[-5;-5];
octave:3> [xzero,rc]=feas(A,b)
warning: feas: some elements in list of return values are undefined
xzero = [](0x0)
rc =  2
octave:4> quit
```

For `qp5` we found $\mathbf{x}^0 = [\frac{18}{7}, -\frac{61}{7}]^\top$. That point is a vertex of the feasible set in $\mathbb{R}^2$, as shown in the graphical solution of the problem. In tableau `T1` the slacks s_3 and s_4 are zero because they are nonbasic, so $\mathbf{x}^0$ is the intersection of the zero hyperplanes for constraints ③ and ④ in the picture. That point would be infeasible if we were solving a linear program, but in `qp5` the variables are not assumed to be nonnegative.

Other quadratic programs have constraint sets for which the process implemented in `feas.m` yields a `T1` in which fewer than n slack variables are nonbasic, and then the resulting $\mathbf{x}^0$ is *not* a vertex in $\mathbf{x}$–space. For example, we could delete constraints ①, ③, and ④ from `qp5` without changing $\mathbf{x}^\star$. Then `feas.m` finds a feasible starting point that is in the boundary of constraint ②.

```
octave:1> A=[2,1];
octave:2> b=[3];
octave:3> [xzero,rc]=feas(A,b)
xzero =

   1.50000
  -0.00000

rc = 0
octave:4> A*xzero-b
ans = 0
octave:5> quit
```

22.2.2 Respecting Inactive Inequalities

In §22.1 you learned how to minimize $q(\mathbf{x})$ subject to $\mathbf{x}$ being in a flat that is defined by equality constraints, but doing so here in the way that we did in `qpeq.m` might yield a point that violates the inequalities we have ignored. Suppose that in solving `qp5` from the vertex $\mathbf{x}^0 = [\frac{18}{7}, -\frac{61}{7}]^\top$ the active set algorithm releases constraint ④ so that $\mathcal{W} = [0, 0, 1, 0]$. Then the only active constraint is ③ which we can write as $\bar{\mathbf{A}}\mathbf{x} = \bar{\mathbf{b}}$ where

$$\bar{\mathbf{A}} = \begin{bmatrix} \frac{1}{2} & -1 \end{bmatrix} \qquad \text{and} \qquad \bar{\mathbf{b}} = \begin{bmatrix} 10 \end{bmatrix}.$$

To take a full Newton descent step in the flat defined by this constraint (i.e., along its zero hyperplane) we would perform the calculations shown below.

```
octave:1> Q=[2,-1;-1,2];
octave:2> c=[-12;3];
octave:3> xzero=[18/7;-61/7];
octave:4> Abar=[1/2,-1];
octave:5> Z=null(Abar);
octave:6> U=hfact(Z'*Q*Z,0.5);
octave:7> V=Z/U;
octave:8> Hinv=V*V';
octave:9> d=-Hinv*(Q*xzero+c)
d =

   4.4286
   2.2143

octave:10> xbar=xzero+d
xbar =

   7.0000
  -6.5000

octave:11> A=[-1,1;2,1;1/2,-1;-2/3,-1];
octave:12> b=[6;3;10;7];
octave:13> A*xbar-b
ans =

  -19.50000
    4.50000
    0.00000
   -5.16667

octave:14> quit
```

The reduced-Newton direction vector $\boxed{\texttt{9>}}$ `d` $= [4.4286, 2.2143]^\intercal$, or in exact arithmetic $\mathbf{d} = [\frac{31}{7}, \frac{31}{14}]^\intercal$, has slope $\frac{1}{2}$ so it points along the edge corresponding to constraint 3 and thus lies on the flat defined by $\mathcal{W} = [0, 0, 1, 0]$. However, taking the full step in that direction yields a point $\bar{\mathbf{x}}$ that violates the second inequality $\boxed{\texttt{13>}}$ and is thus outside of the feasible set (see the picture). This is a disaster for the active set strategy, because if some $\mathbf{x}^k$ violates *any* constraint then the signs of the Lagrange multipliers tell us nothing and algorithm stanza 3 is likely not to identify the correct working set.

Taking the full step minimizes the objective in the direction $\mathbf{d}^k$, so it would never make sense to take a step *longer* than that. But if the full step would violate an inequality we must take a *shorter* step, to $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha\mathbf{d}^k$ where $\alpha < 1$. For $\mathbf{x}^{k+1}$ to remain feasible α must be chosen so that

$$\begin{aligned} \mathbf{A}\mathbf{x}^{k+1} &\leq \mathbf{b} \\ \mathbf{A}(\mathbf{x}^k + \alpha\mathbf{d}^k) &\leq \mathbf{b} \\ \mathbf{A}\mathbf{x}^k + \alpha\mathbf{A}\mathbf{d}^k &\leq \mathbf{b}. \end{aligned}$$

At the first step in solving qp5 we have $\mathbf{x}^k = \mathbf{x}^0 = [\frac{18}{7}, -\frac{61}{7}]^\top$ and $\mathbf{d}^k = \mathbf{d}^0 = [\frac{31}{7}, \frac{31}{14}]^\top$ so this system of inequalities is

$$\mathbf{A}\mathbf{x}^k + \alpha\mathbf{A}\mathbf{d}^k = \begin{bmatrix} -1 & 1 \\ 2 & 1 \\ \frac{1}{2} & -1 \\ -\frac{2}{3} & -1 \end{bmatrix} \begin{bmatrix} \frac{18}{7} \\ -\frac{61}{7} \end{bmatrix} + \alpha \begin{bmatrix} -1 & 1 \\ 2 & 1 \\ \frac{1}{2} & -1 \\ -\frac{2}{3} & -1 \end{bmatrix} \begin{bmatrix} \frac{31}{7} \\ \frac{31}{14} \end{bmatrix} \leq \begin{bmatrix} 6 \\ 3 \\ 10 \\ 7 \end{bmatrix} = \mathbf{b}.$$

Computing the matrix-vector products we find

$$\begin{array}{rrcccl} -\frac{79}{7} & -\frac{31}{14} & \alpha & \leq & 6 & \Rightarrow \quad \alpha \geq -\frac{242}{31} \approx -7.8 \qquad (1) \\ -\frac{25}{7} & +\frac{155}{14} & \alpha & \leq & 3 & \Rightarrow \quad \alpha \leq \frac{92}{155} \approx 0.59 \qquad (2) \\ 10 & +0 & \alpha & \leq & 10 & \Rightarrow \quad \alpha \text{ can be anything} \qquad (3) \\ 7 & -\frac{1519}{294} & \alpha & \leq & 7 & \Rightarrow \quad \alpha \geq 0. \qquad (4) \end{array}$$

Constraint (1) is slack at $\mathbf{x}^0$, and to violate it by sliding along the constraint (3) hyperplane we would have to go down and to the left 7.8 lengths of $\mathbf{d}^k$, to the vertex where the constraint (3) hyperplane and the constraint (1) hyperplane cross. To move in that direction, opposite of $\mathbf{d}^k$, it would be necessary to make α negative, and as long as $\alpha \geq -7.8$ the point $\mathbf{x}^k + \alpha\mathbf{d}^k$ satisfies constraint (1). Of course in solving qp5 we do not intend to go that way; to move in the descent direction $\mathbf{d}^k$ we are interested only in values of $\alpha \geq 0$.

Constraint (2) is also slack at $\mathbf{x}^0$, but we could violate it by sliding along the constraint (3) hyperplane up and to the right past the vertex $\hat{\mathbf{x}}$ where the constraint (3) hyperplane and the constraint (2) hyperplane cross. To remain feasible for constraint (2) we should stop at $\hat{\mathbf{x}}$, where $\alpha \approx +0.59$.

Constraint (3) is tight at $\mathbf{x}^0$ and at all points $\mathbf{x}^0 + \alpha\mathbf{d}^k$ along its contour, so the third inequality above does not limit α.

Constraint (4) is also tight at $\mathbf{x}^0$. Sliding down and to the left along the constraint (3) hyperplane ($\alpha < 0$) would violate constraint (4), but sliding up and to the right leaves it satisfied; $\bar{\mathbf{x}}$, for example, is feasible for constraint (4). Thus the bottom inequality permits any $\alpha \geq 0$.

Now consider a different hypothetical situation in which we start the solution of qp5 with $\mathcal{W} = [0, 0, 1, 0]$ as before, but from the point $\hat{\mathbf{x}} = [\frac{26}{5}, -\frac{37}{5}]^\top$. The reduced Newton direction vector still lies on the zero hyperplane of constraint (3) but now it turns out to be $\hat{\mathbf{d}} = [\frac{9}{5}, \frac{9}{10}]^\top$, so for a step in that direction to remain feasible α must satisfy

$$\mathbf{A}\hat{\mathbf{x}} + \alpha\mathbf{A}\hat{\mathbf{d}} = \begin{bmatrix} -1 & 1 \\ 2 & 1 \\ \frac{1}{2} & -1 \\ -\frac{2}{3} & -1 \end{bmatrix} \begin{bmatrix} \frac{26}{5} \\ -\frac{37}{5} \end{bmatrix} + \alpha \begin{bmatrix} -1 & 1 \\ 2 & 1 \\ \frac{1}{2} & -1 \\ -\frac{2}{3} & -1 \end{bmatrix} \begin{bmatrix} \frac{9}{5} \\ \frac{9}{10} \end{bmatrix} \leq \begin{bmatrix} 6 \\ 3 \\ 10 \\ 7 \end{bmatrix} = \mathbf{b}.$$

or

$$\begin{array}{rrcccl} -\frac{63}{5} & -\frac{9}{10} & \alpha & \leq & 6 & \Rightarrow \quad \alpha \geq -\frac{62}{3} \approx -20.7 \quad (1) \\ 3 & +\frac{9}{2} & \alpha & \leq & 3 & \Rightarrow \quad \alpha \leq 0 \quad (2) \\ 10 & +0 & \alpha & \leq & 10 & \Rightarrow \quad \alpha \text{ can be anything} \quad (3) \\ \frac{59}{15} & -\frac{21}{10} & \alpha & \leq & 7 & \Rightarrow \quad \alpha \geq -\frac{92}{63} \approx -1.46 \quad (4) \end{array}$$

The first inequality once again shows that to violate constraint (1) by sliding along the constraint (3) zero contour it is necessary to go down and to the left, this time past $\alpha = -\frac{62}{3}$. The last inequality shows that to violate constraint (4) it is also necessary to go down and to the left, past $\mathbf{x}^0$ which corresponds to $\alpha = -\frac{92}{63}$. The third inequality again says that we cannot violate constraint (3) by sliding along its zero contour. Now, however, the second inequality requires that $\alpha \leq 0$; from $\hat{\mathbf{x}}$ it is not possible to move in the $+\mathbf{d}$ direction without leaving the feasible set.

This example shows (from inequality (3) for each starting point we considered) that α is not limited by a constraint that is assumed to be active, because a constraint cannot be violated by moving along its zero contour. In higher dimensions none of the constraints that are assumed to be active can be violated by moving in the flat on which they are all satisfied. It is the constraints that are assumed to be inactive (those having $w_i = 0$) that determine bounds on the steplength α [5, p469] [1, Exercise 11.19]. These inequalities fall into four categories.

First, if a constraint i with $w_i = 0$ has $A_i\mathbf{x}^k < b_i$ so that it is strictly satisfied, and if $A_i\mathbf{d}^k \leq 0$ so that moving in the $+\mathbf{d}^k$ direction *does not* decrease the amount by which it is satisfied, then this constraint does not prevent us from making α as high as we like. This is what happened in the first and last inequalities we deduced for starting from $\mathbf{x}^0$ or $\hat{\mathbf{x}}$.

Second, if a constraint i with $w_i = 0$ has $A_i\mathbf{x}^k < b_i$ so that it is strictly satisfied, but $A_i\mathbf{d}^k > 0$ so that moving in the $+\mathbf{d}^k$ direction *does* decrease the amount by which it is satisfied, then to stay feasible we must have

$$\begin{aligned} A_i\mathbf{x}^k + \alpha A_i\mathbf{d}^k &\leq b_i \\ \alpha A_i\mathbf{d}^k &\leq b_i - A_i\mathbf{x}^k \\ \alpha &\leq \frac{b_i - A_i\mathbf{x}^k}{A_i\mathbf{d}^k} = r. \end{aligned}$$

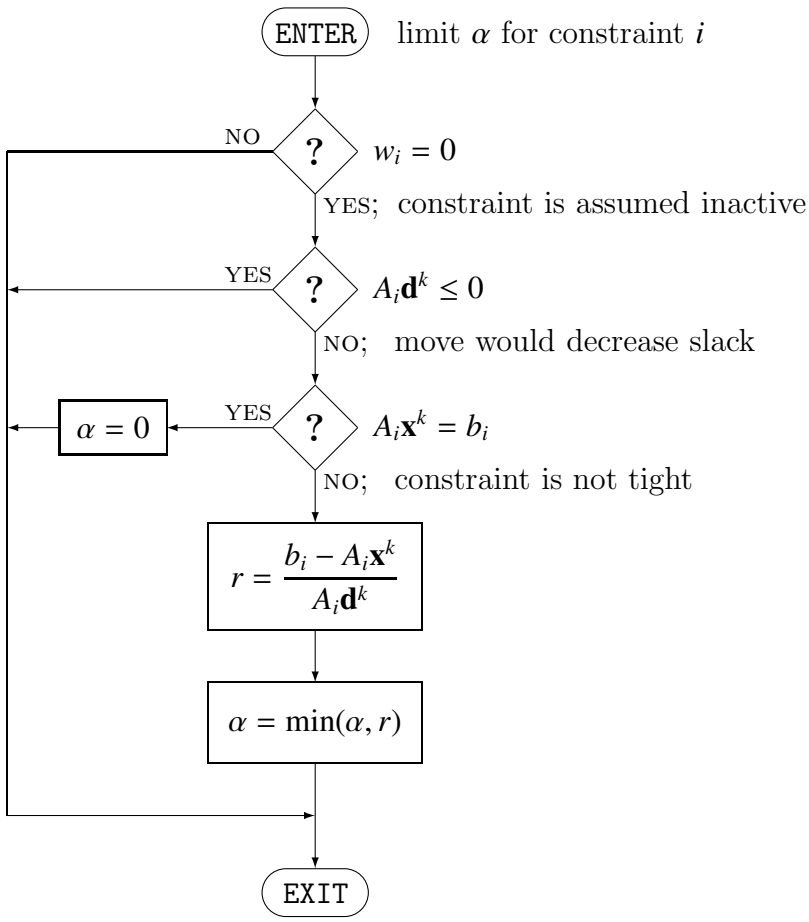
This is what happened in the second inequality we deduced for starting from $\mathbf{x}^0$ or $\hat{\mathbf{x}}$.

Third, if a constraint i with $w_i = 0$ has $A_i\mathbf{x}^k = b_i$ so that it is satisfied with equality, and if $A_i\mathbf{d}^k < 0$ so that moving in the $+\mathbf{d}^k$ direction *increases* the amount by which it is satisfied (i.e., makes it slack) then this constraint only requires $\alpha \geq 0$. This is what happened in the last inequality we deduced starting from $\mathbf{x}^0$.

Fourth, if a constraint i with $w_i = 0$ has $A_i\mathbf{x}^k = b_i$ so that it is satisfied with equality, but $A_i\mathbf{d}^k > 0$ so that moving in the $+\mathbf{d}^k$ direction *decreases* the amount by which it is satisfied (i.e., violates it) then this constraint demands that $\alpha \leq 0$. Since we are interested only in

nonnegative steplengths, this means that $\alpha = 0$ and no step can be taken with this active set. This is what happened in the second inequality we deduced starting from $\hat{\mathbf{x}}$.

We hope to take the full reduced-Newton step at each iteration of the active set algorithm, so we will initialize α to 1, but to avoid violating inactive inequalities we will examine each constraint and use it to limit α as discussed above. The logic of this process is summarized in the flowchart below.



If $w_i \neq 0$ the constraint is assumed to be active so it does not limit α. If $w_i = 0$ the constraint is assumed to be inactive and might limit α.

If $A_i\mathbf{d}^k \le 0$ then moving in the direction $\mathbf{d}^k$ would not decrease the slack in the constraint, so it does not limit α. If $A_i\mathbf{d}^k > 0$ then moving in the direction $\mathbf{d}^k$ would decrease the slack in the constraint and might limit α.

If $A_i\mathbf{x}^k < b_i$ then we can move a distance r in the direction $\mathbf{d}^k$ without violating this constraint; if r is less than the current value of α we must decrease α to this value of r. If $A_i\mathbf{x}^k = b_i$ then the constraint is tight and we cannot decrease its slack from zero, so $\alpha = 0$. (This is just a special case of limiting α to r, but in the code it will be necessary to handle it separately so I indicated that here.)

When the process described by the flowchart has been applied to each constraint, the resulting value of α is a steplength that will preserve the feasibility of $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha\mathbf{d}^k$ for all of the inequalities that are assumed to be inactive.

22.2.3 Computing the Lagrange Multipliers

Stanza 3 of the active set algorithm calls for computing the $\boldsymbol{\lambda}^k$ corresponding to each $\mathbf{x}^k$, and we can do that by using the Lagrange conditions for the subproblem of stanza 2. In general each equality-constrained subproblem has this form

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & q(\mathbf{x}) & = & \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x} \\ \text{subject to} & \bar{\mathbf{A}}\mathbf{x} & = & \bar{\mathbf{b}} \end{array}$$

where $\bar{\mathbf{A}}$ and $\bar{\mathbf{b}}$ are the rows of $\mathbf{A}\mathbf{x} = \mathbf{b}$ that are in the current working set. There are

$$\overline{\mathrm{m}} = \sum_{i=1}^{m} w_i$$

such rows, so, $\overline{\mathrm{m}} \le m$. If the minimizing point of $q(\mathbf{x})$ happens to be at a vertex of the polyhedron defined by $\mathbf{A}\mathbf{x} \le \mathbf{b}$ then $\overline{\mathrm{m}} = n$, but the minimizing point could be at some non-vertex boundary point in which case $\overline{\mathrm{m}} < n$, or interior to the feasible set in which case $\overline{\mathrm{m}} = 0$. Provided none of the constraints are redundant, no vertex is degenerate and $\overline{\mathrm{m}}$ can never exceed n. Thus $0 \le \overline{\mathrm{m}} \le \min(m, n)$. A quadratic program whose constraints are inequalities can have $m < n$ (as in `qp4`) or $m = n$ or $m > n$ (as in `qp5`).

The Lagrangian for the equality-constrained subproblem is

$$\mathcal{L}(\mathbf{x}, \bar{\boldsymbol{\lambda}}) = \tfrac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x} + \bar{\boldsymbol{\lambda}}^\top[\bar{\mathbf{A}}\mathbf{x} - \bar{\mathbf{b}}],$$

where $\bar{\boldsymbol{\lambda}}$ is the rows of $\boldsymbol{\lambda}$ corresponding to the active constraints. From $\mathcal{L}$ we can write down these Lagrange conditions for the subproblem.

$$\begin{array}{ll} \mathbf{Q}\mathbf{x} + \mathbf{c} + \bar{\mathbf{A}}^\top\bar{\boldsymbol{\lambda}} = \mathbf{0} & \text{stationarity} \\ \bar{\mathbf{A}}\mathbf{x} - \bar{\mathbf{b}} = \mathbf{0} & \text{feasibility} \end{array}$$

At each iteration k in the active set algorithm the KKT multipliers corresponding to the inactive constraints of the original problem are zero, and we can find those corresponding to the active constraints, which I will denote $[\bar{\boldsymbol{\lambda}}]^k$, by solving the Lagrange stationarity condition

$$\bar{\mathbf{A}}^\top[\bar{\boldsymbol{\lambda}}]^k = -[\mathbf{Q}\mathbf{x}^k + \mathbf{c}].$$

This linear system has n equations but only $\overline{\mathrm{m}}$ variables, so it is probably overdetermined, and if $\mathbf{x}^k$ is not exactly equal to $\mathbf{x}^\star$ it is probably also inconsistent. One way to find the $\boldsymbol{\lambda}$ that comes closest to satisfying these equations is to minimize the sum of the squares of the row deviations. This is the same calculation we performed in §8.6.1 to find the coefficients

in a least-squares regression model. To recapitulate that analysis in this setting it will be convenient to temporarily simplify our notation by letting

$$\mathbf{B} = \bar{\mathbf{A}}^\top \qquad \mathbf{u} = [\bar{\boldsymbol{\lambda}}]^k \qquad \mathbf{g} = \mathbf{Q}\mathbf{x}^k + \mathbf{c}$$

so that the linear system above is

$$\mathbf{B}\mathbf{u} = -\mathbf{g}.$$

Then the row deviations e_j are elements of the vector

$$\mathbf{e} = \mathbf{g} + \mathbf{B}\mathbf{u}$$

and the sum of their squares is

$$E = (\mathbf{g} + \mathbf{B}\mathbf{u})^\top(\mathbf{g} + \mathbf{B}\mathbf{u}) = \mathbf{g}^\top\mathbf{g} + 2\mathbf{u}^\top(\mathbf{B}^\top\mathbf{g}) + (\mathbf{B}\mathbf{u})^\top(\mathbf{B}\mathbf{u})$$

Setting the derivative with respect to $\mathbf{u}$ equal to zero,

$$\begin{aligned} \nabla_{\mathbf{u}} E = 2\mathbf{B}^\top\mathbf{g} + 2\mathbf{B}^\top(\mathbf{B}\mathbf{u}) &= \mathbf{0} \\ \mathbf{B}^\top\mathbf{g} + (\mathbf{B}^\top\mathbf{B})\mathbf{u} &= \mathbf{0} \end{aligned}$$

If $\mathbf{B}^\top\mathbf{B}$ is nonsingular, we can find the Lagrange multipliers like this.

$$\begin{aligned} (\mathbf{B}^\top\mathbf{B})^{-1}(\mathbf{B}^\top\mathbf{g}) + (\mathbf{B}^\top\mathbf{B})^{-1}(\mathbf{B}^\top\mathbf{B})\mathbf{u} &= \mathbf{0} \\ (\mathbf{B}^\top\mathbf{B})^{-1}(\mathbf{B}^\top\mathbf{g}) &= -\mathbf{u} \end{aligned}$$

In terms of the fussier notation we began with, we have shown that

$$[\bar{\boldsymbol{\lambda}}]^k = -(\bar{\mathbf{A}}\bar{\mathbf{A}}^\top)^{-1}(\bar{\mathbf{A}}\mathbf{g}) = -\mathbf{A}^+[\mathbf{Q}\mathbf{x}^k + \mathbf{c}]$$

where $\mathbf{A}^+ = (\bar{\mathbf{A}}\bar{\mathbf{A}}^\top)^{-1}\bar{\mathbf{A}}$ is the $(\overline{\mathrm{m}} \times n)$ pseudoinverse of $\bar{\mathbf{A}}$ [4, §15.3]. To calculate $\mathbf{A}^+$ in a numerically stable way we can use the factor-and-solve approach. If we let $\bar{\mathbf{A}}\bar{\mathbf{A}}^\top = \mathbf{U}^\top\mathbf{U}$, then $\mathbf{U}^\top\mathbf{U}\mathbf{A}^+ = \bar{\mathbf{A}}$. If we let $\mathbf{U}\mathbf{A}^+ = \mathbf{V}$ then $\mathbf{U}^\top\mathbf{V} = \bar{\mathbf{A}}$. Then we can solve the matrix equation $\mathbf{U}^\top\mathbf{V} = \bar{\mathbf{A}}$ for $\mathbf{V}$ and the matrix equation $\mathbf{U}\mathbf{A}^+ = \mathbf{V}$ for $\mathbf{A}^+$.

To solve these matrix equations using MATLAB as in §22.1.2, we need the unknown matrix in each case to appear on the left; thus we will actually solve $\mathbf{V}^\top\mathbf{U} = \bar{\mathbf{A}}^\top$ for $\mathbf{V}^\top$ and then $[\mathbf{A}^+]^\top\mathbf{U}^\top = \mathbf{V}^\top$ for $[\mathbf{A}^+]^\top$, which we can transpose to get $\mathbf{A}^+$. To see how this works suppose that in solving the first system we represent $\mathbf{V}^\top$ by `Vt`, $\bar{\mathbf{A}}$ by `Abar`, and $\mathbf{U}$ by `U`. Then the MATLAB operation `Vt=Abar'/U` is [50, §8.3] conceptually equivalent to finding `Abar'*inv(U)`, but it is computed without forming the inverse of `U`.

To implement this plan I wrote the MATLAB routine `getlgm.m` listed below. Its input parameters are `m`, the total number of constraints; `Abar`, the matrix whose rows are the

transposes of the gradients of the active inequalities; `W`, the current working set; and `g`, the gradient of the objective (which is $\mathbf{Qx}^k + \mathbf{c}$ in the discussion above).

```
 1 function [lambda,rc]=getlgm(m,Abar,W,g)
 2 % compute Lagrange multipliers
 3
 4   lambda=zeros(m,1);           % zero out multipliers
 5   [U,rc]=hfact(Abar*Abar',1);  % factor and set return code
 6   if(rc ~= 0) return; end      % give up if factoring failed
 7   Vt=Abar'/U;                  % solve
 8   Aplus=(Vt/U')';              % for pseudoinverse
 9   ibar=0;                      % need to index rows of Abar
10   for i=1:m                    % fill in nonzero multipliers
11       if(W(i) == 1)            % is this constraint tight?
12          ibar=ibar+1;                 % next row
13          lambda(i)=-Aplus(ibar,:)*g;  % use formula
14       end                             % done with constraint
15   end                          % done with multipliers
16
17 end
```

The routine begins [4] by initializing $\boldsymbol{\lambda}$ to the zero vector in anticipation of filling in the nonzero elements later. Then [5] it uses the `hfact.m` routine of §19.3 to factor $\bar{\mathbf{A}}\bar{\mathbf{A}}^\top$ and get $\mathbf{U}$. Here I set the second parameter of `hfact.m` to `1` rather than the value of `0.5` that we typically use in factoring a Hessian matrix. Recall from §13.2 that this is the weighting factor γ used in modifying the matrix if it is not positive definite. If $\bar{\mathbf{A}}\bar{\mathbf{A}}^\top$ is not positive definite there is something wrong so it would not make sense to modify it, and using $\gamma = 1$ causes `hfact.m` to resign with `rc=1` instead. If that happens the routine takes the [6] error return.

Next [7-8] the calculations described above are used to find $\mathbf{A}^+$, which is used [10-15] to calculate the λ_i. The elements of `lambda` are indexed by `i`, but the rows of `Aplus` are indexed by `ibar`.

To test `getlgm.m` I used it to confirm numerically a calculation that we performed analytically for the `moon` problem, whose function, gradient, and Hessian routines are listed below.

```
function f=moon(x,i)
  switch(i)
    case 0
      f=-(x(1)-3)^2-x(2)^2;
    case 1
      f=x(1)^2+x(2)^2-1;
    case 2
      f=-(x(1)+2)^2-x(2)^2+4;
  end
end
```

```
function g=moong(x,i)
  switch(i)
    case 0
      g=[-2*(x(1)-3);-2*x(2)];
    case 1
      g=[2*x(1);2*x(2)];
    case 2
      g=[-2*(x(1)+2);-2*x(2)];
  end
end
```

```
function H=moonh(x,i)
  switch(i)
    case 0
      H=[-2,0;0,-2];
    case 1
      H=[2,0;0,2];
    case 2
      H=[-2,0;0,-2];
  end
end
```

In CASE 2 of the KKT solution in §16.3 we assumed the working set `W=[1,0]` at the point $\mathbf{x} = [1, 0]^\top$ and deduced analytically that $\lambda_1 = -2$.


```
octave:1> x=[1;0];
octave:2> Abar=[moong(x,1)'];
octave:3> g=moong(x,0);
octave:4> W=[1;0];
octave:5> [lambda,rc]=getlgm(2,Abar,W,g)
lambda =

  -2
   0

rc = 0
octave:6> quit
```

22.2.4 An Active Set Implementation

Using the ideas discussed above I wrote the `qpin.m` routine listed on the next two pages. Because of the logic of this routine, `k` counts iterations completed rather than that number plus one.

In the first stanza [4-7] `tol` determines [76] how negative a Lagrange multiplier must be before we consider its constraint to be sticking and [49,93] how close to zero a constraint must be for us to consider it tight. This **zero tolerance** should be a small positive number so that slight imprecisions in the floating point calculations do not lead to constraint misclassifications.

If [11] there are any constraints, the second stanza [12] uses `feas.m` as suggested in §22.2.1 to detect infeasibility [13-16] or set a feasible starting point. The active set starts empty [19-21] as explained in §22.2.0, without regard to which constraints are actually active at $\mathbf{x}^0$.

Then control enters a long loop [25-109] of up to `kmax` optimization iterations. Each iteration begins [26-42] by finding the Newton descent direction in the flat defined by the active constraints. If [27] there are exactly `n` active constraints then their intersection is optimal so [28] `rc=0` is set to signal convergence and [29] the iterations are interrupted. If [30] there are no active constraints then, as suggested in §22.1.2, [31] $\mathbf{Z} = \mathbf{I}$ to do unconstrained Newton descent. Otherwise the code proceeds [32-42] as in `qpeq.m` to find `Hinv` and `d`.

The next stanza [44-58] implements the process described by the flowchart of §22.2.2 to determine a step length `alpha` that does not violate any of the inactive inequalities. Mathematically $\mathtt{r} \geq 0$, but roundoff errors in the floating-point calculations can give it a tiny negative value so [54] in that case it is reset to zero.

Then, having determined a descent direction and step length, the routine [61] takes the reduced-Newton step to complete stanza 2 of the algorithm we outlined.

Now, if any constraints are active [64] `getlgm.m` is used [67] to find the Lagrange multipliers corresponding to the active constraints. The next stanza [73-82] checks the Lagrange multiplier of each active constraint and [76-79] if λ_i is negative releases the constraint by setting $w_i = 0$. In that case a change has been made to `W`, so the logical variable `OK` is [78] set to `false` indicating that convergence has not yet been achieved.

Next [84-102] the routine rebuilds `Abar` from scratch and counts its rows to update `mbar`. If a constraint is already in the working set [88-90] it is retained; otherwise [91-101] it might


```
function [xstar,k,rc,W,lambda]=qpin(Q,c,A,b,kmax,epz)
% minimize (1/2)x'Qx+c'x subject to Ax<=b

% initialize
  n=size(Q,1);                      % number of variables
  m=size(A,1);                      % number of inequalities
  tol=1e-6;                         % zero tolerance

% find a feasible starting point
  xzero=zeros(n,1);                 % use origin if unconstrained
  if(m > 0)                         % if there are constraints
     [xzero,rcf]=feas(A,b);         % get a feasible starting point
     if(rcf ~= 0)                   % success?
        rc=4;                       % no; signal failure
        return                      % and give up
     end                            % feasible point has been found
     OK=false;                      % active set has not been found
  end
  W=zeros(1,m); OK=true;            % working set starts empty
  Abar=zeros(0,n); mbar=0;          % active A starts empty
  lambda=zeros(m,1);                % multipliers start zero
  xk=xzero;                         % now have initial xk, W, Abar

  rc=1;                             % in case of nonconvergence
  for k=1:kmax
%     find reduced Newton direction
      if(mbar == n)                 % if active constraints square
         rc=0;                      % signal success
         break                      % and return unique solution
      elseif(mbar == 0)             % subproblem is unconstrained
         Z=eye(n);                  % Z=I makes Z'*Q*Z=Q
      else                          % 0 < mbar < n
         Z=null(Abar);              % get a basis for the nullspace
      end                           % now have Z
      [U,rch]=hfact(Z'*Q*Z,0.5);    % factor the reduced Hessian
      if(rch ~= 0)                  % success?
         rc=3;                      % report modification failed
         break                      % and give up
      end                           % now Z'QZ=U'U
      V=Z/U;                        % solve VU=Z for V
      Hinv=V*V';                    % find Hinv=Z*inv(Z'QZ)*Z'
      d=-Hinv*(Q*xk+c);             % full reduced Newton step

%     find step length
      alpha=1;                      % full step if no constraints
      for i=1:m                     % examine each constraint
          if(W(i) == 0)                       % assumed inactive?
             if(A(i,:)*d <= 0) continue; end  % increasing slack OK
             if(abs(A(i,:)*xk-b(i)) < tol)    % if already tight
                alpha=0;                        % cannot tighten
                break                           % no move possible
             else                             % move decreases slack
                r=(b(i)-A(i,:)*xk)/(A(i,:)*d);  % maximum step
                r=max(r,0);                     % can't be negative
                alpha=min(alpha,r);             % make alpha no more
             end                              % this constraint done
          end                           % ignored constraints checked
      end                               % now have alpha

```



```
 60 %     take reduced Newton step
 61       xk=xk+alpha*d;                % take the step
 62
 63       OK=true;                      % assume W will not change
 64       if(mbar > 0)                  % any constraints active?
 65 %        find Lagrange multipliers
 66          g=Q*xk+c;                        % objective gradient
 67          [lambda,rcg]=getlgm(m,Abar,W,g); % use formula
 68          if(rcg ~= 0)                     % was there trouble?
 69             rc=2;                         % report Abar*Abar' not pd
 70             break                         % and give up
 71          end                        % now have multipliers
 72
 73 %        release sticking constraints
 74          for i=1:m                  % examine each constraint
 75              if(W(i) == 1)          % is it active?
 76                 if(lambda(i)<-tol)  % yes; is it sticking?
 77                    W(i)=0;          % sticking; make it inactive
 78                    OK=false;        % not converged yet
 79                 end
 80              end
 81          end                        % sticking constraints released
 82       end
 83
 84 %     find constraint values and activate blocking constraints
 85       Abar=zeros(0,n);              % active A empty
 86       mbar=0;                       % assume no active constraints
 87       for i=1:m                     % examine each constraint
 88           if(W(i) == 1)             % is it in the working set?
 89              Abar=[Abar;A(i,:)];    % already active; copy its A row
 90              mbar=mbar+1;           % and count it
 91           else                             % not in working set
 92              if(mbar == n) break; end      % no more than n active
 93              if(abs(A(i,:)*xk-b(i)) < tol) % if it is tight
 94                 if(A(i,:)*d > 0)           % and move would violate
 95                    W(i)=1;             % it is blocking; activate it
 96                    OK=false;           % not converged yet
 97                    mbar=mbar+1;        % increase the count
 98                    Abar=[Abar;A(i,:)]; % add the row to Abar
 99                 end
100              end
101           end
102       end                           % blocking constraints active
103
104 %     test for convergence
105       if(OK && norm(d)<epz)         % W unchanged and step small?
106          rc=0;                      % signal success
107          break                      % and return
108       end                           % convergence tested
109    end
110    xstar=xk;                        % return the current point
111
112 end
```

be blocking. As mentioned earlier it is inconvenient to have more than n constraints active, so [92] if `mbar` reaches `n` no more are added to the working set. Otherwise, if the constraint is tight [93] and moving in the direction $\mathbf{d}^k$ would violate it [94], then it is a blocking constraint. It is [95] activated, [97] counted, and [98] appended to `Abar`. A change has been made to `W`, so `OK` is [96] set to `false` indicating that convergence has not yet been achieved.

In the final stanza of the optimization loop 104-108 convergence is judged to have occurred 105 if `W` has stopped changing *and* `xk` is a stationary point in the flat of the active constraints. Only then, or if `mbar=n` 29, is the current iterate returned 110 as `xstar` with 106,28 `rc=0` to signal success. If `kmax` iterations are consumed without satisfying any convergence criterion then 110 the current iterate is returned for `xstar` but with 24 `rc=1`.

To test `qpin.m`, I used it to solve `qp5` in the Octave session below. With `kmax=0` the routine returns 2> $\mathbf{x}^0 = [2.5714, -8.7143]^\top$, which is the same starting point we found in §22.2.1. With `kmax=1` one iteration of reduced-Newton descent is allowed. The active set starts out

```
octave:1> % qp5
octave:1> Q=[2,-1;-1,2];c=[-12;3];A=[-1,1;2,1;1/2,-1;-2/3,-1];b=[6;3;10;7];
octave:2> [xzero]=qpin(Q,c,A,b,0,1e-6)
xzero =

   2.5714
  -8.7143

octave:3> [x1,k,rc]=qpin(Q,c,A,b,1,1e-6)
x1 =

   4.0584
  -5.1168

k =  1
rc =  1
octave:4> [xstar,k,rc,W]=qpin(Q,c,A,b,3,1e-6)
xstar =

   2.3571
  -1.7143

k =  3
rc = 0
W =

   0   1   0   0

octave:5> % qp4
octave:5> Q=[2,0,0,1;0,2,1,0;0,1,4,0;1,0,0,4];c=zeros(4,1);A=[3,-1,-2,-1;-4,1,5,2];b=[-1;3];
octave:6> [xstar,k,rc,W]=qpin(Q,c,A,b,3,1e-6)
xstar =

  -0.250000
   0.038462
   0.057692
   0.096154

k =  3
rc = 0
W =

   1   0

octave:7> quit
```


empty 19-21 so this descent step is unconstrained 30-31 except by the steplength limitation 47-57 that prevents any inequality from being violated. This results 3> in the step to $\mathbf{x}^1 = [4.0584, -5.1168]^\top$ shown in the picture. That is as far as we can go in the unconstrained Newton direction without violating constraint 2. At $\mathbf{x}^1$ constraint 2 is 93-100 identified as blocking.

With `kmax=2` two iterations of reduced-Newton descent are allowed. In iteration `1` constraint 2 is found to be active, and iteration `2` does not release it, so the reduced Newton direction is along its zero hyperplane. We can minimize $q(\mathbf{x})$ on that flat without violating any inequality, which results in the second and final (`rc=0`) step 4> to $\mathbf{x}^\star = [2.3571, -1.7143]^\top$.

Finally 5>-6> I used `qpin.m` to solve `qp4` as an inequality-constrained problem, obtaining the same result that we found, using `qpeq.m` with only the first constraint in the problem, at the beginning of §22.2.

22.3 A Reduced-Newton Algorithm

In §14.5 we generalized the conjugate gradient algorithm for minimizing a quadratic objective, to derive the Fletcher-Reeves algorithm for minimizing an objective that need not be quadratic. That involved replacing formulas by function calls to compute the value and gradient of the objective.

We can generalize the nullspace and active set algorithms of §22.1 and §22.2 in a similar way, to solve problems in which the constraints are still linear equalities or inequalities but the objective is not necessarily quadratic, by using the gradient $\nabla f_0(\mathbf{x}^k)$ in place of $\mathbf{Q}\mathbf{x}^k + \mathbf{c}$ and the Hessian $\mathbf{H}(\mathbf{x}^k)$ in place of $\mathbf{Q}$. The methods that result are both called **reduced-Newton algorithms** [4, p550-552]. Here we will study the one for equality constraints.

The `rneq.m` routine listed on the next page is `qpeq.m` modified as described above. In place of `Q` and `c` the calling sequence 1 now includes the function pointers `grd` and `hsn`. The number of variables is taken 5 to be the number of columns in `A`, so if there are no constraints we must pass `A=zeros(0,n)` rather than `A=[]`. Now the Hessian depends on $\mathbf{x}$, so `Hinv` changes from one iteration to the next and must be recomputed 38-45 for each descent step.

To test `rneq.m`, I used the following problem which I will call `rnt` (see §28.7.35).

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1+x_4)^4 + (x_2+x_3)^2 \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \begin{bmatrix} 3x_1 - x_2 - 2x_3 - x_4 \\ -4x_1 + x_2 + 5x_3 + 2x_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \end{array}$$

This problem has the same linear equality constraints as `qp1` but its objective, while strictly convex, is no longer quadratic.


```
function [xstar,k,rc,nm]=rneq(grd,hsn,A,b,kmax,epz)
% minimize f(x) subject to Ax=b

% size up the problem
  n=size(A,2);                      % number of variables
  m=size(A,1);                      % number of equality constraints
  k=0;                              % no iterations yet
  nm=0;                             % no modifications yet

% find a starting point and a basis for nullspace of A
  xzero=zeros(n,1);                 % use the origin if unconstrained
  if(m > 0)                         % if there are constraints
     T=[0,zeros(1,n);b,A];                                % tableau
     [Tnew,S,tr,mr,rc0]=newseq(T,m+1,n+1,[1:m+1],m+1);  % seek basis
     if(rc0 ~= 0)                                         % success?
        rc=3;                       % report constraints inconsistent
        return                      % and give up
     end
     for j=1:n                      % extract
         if(S(j) ~= 0)              % the basic solution
            xzero(j)=Tnew(S(j),1); % to use
         end                        % as the starting point
     end
     if(mr-1 == n)                  % is the system square?
        xstar=xzero;                % if so this is the optimal point
        rc=0;                       % report success
        return                      % and return it
     end
     A=Tnew(2:mr,2:n+1);            % A without redundant constraints
     Z=null(A);                     % get a basis for the nullspace
  else                              % no constraints
     Z=eye(n);                      % Z=I makes Z'*H*Z=H
  end

% do modified Newton descent in the flat defined by the constraints
  xk=xzero;                         % start here
  for k=1:kmax                      % do up to kmax iterations
      H=hsn(xk);                        % find the Hessian here
      [U,rch,nm]=hfact(Z'*H*Z,0.5);     % factor the reduced Hessian
      if(rch ~= 0)                      % success?
         rc=2;                          % report modification failed
         return                         % and give up
      end
      V=Z/U;                        % solve VU=Z
      Hinv=V*V';                    % find Z*inv(Z'HZ)*Z'
      d=-Hinv*grd(xk);              % full reduced Newton step
      xk=xk+d;                      % take the step
      if(norm(d) <= epz)            % converged?
         xstar=xk;                  % yes; save optimal point
         rc=0;                      % report success
         return                     % and return
      end
  end                               % of reduced Newton steps
  xstar=xk;                         % save the current point
  rc=1;                             % report out of iterations

end
```

The routines `rnt.m`, `rntg.m`, and `rnth.m` listed at the top of the next page calculate the value, gradient, and Hessian of $f_0(\mathbf{x})$.


```
function f=rnt(x)
  f=  (x(1)+x(4))^4;
  f=f+(x(2)+x(3))^2;
end
```

```
function g=rntg(x)
  g=[4*(x(1)+x(4))^3;
     2*(x(2)+x(3));
     2*(x(2)+x(3));
     4*(x(1)+x(4))^3];
end
```

```
function H=rnth(x)
  H=[12*(x(1)+x(4))^2, 0, 0, 12*(x(1)+x(4))^2;
     0, 2, 2, 0;
     0, 2, 2, 0;
     12*(x(1)+x(4))^2, 0, 0, 12*(x(1)+x(4))^2];
end
```

In the Octave session below, `rneq.m` solves the `rnt` problem. In exact arithmetic the point returned is $\mathbf{x}^\star = [-\frac{1}{10}, -\frac{6}{10}, \frac{6}{10}, \frac{1}{10}]^\top$ which yields an objective value of $f_0(\mathbf{x}^\star) = 0$. Because $f_0(\mathbf{x})$ must be nonnegative this is its minimum value, and because $\mathbf{x}^\star$ is also feasible it is optimal. The objective is not quadratic, so Newton descent does not minimize it in one step.

```
octave:1> A=[3,-1,-2,-1;-4,1,5,2];
octave:2> b=[-1;3];
octave:3> [xstar,k,rc,nm]=rneq(@rntg,@rnth,A,b,50,1e-6)
xstar =

  -0.100000
  -0.600000
   0.600000
   0.099998

k =  34
rc = 0
nm = 0
octave:4> A*xstar
ans =

  -1.00000
   3.00000

octave:5> rnt(xstar)
f =  1.8018e-23
octave:6> quit
```

To further investigate the behavior of `rneq.m` I wrote `rneqplot.m`, listed on the next page, to plot its convergence trajectory in t_1–t_2 space.

In §22.1.1 we found that if the columns of $\mathbf{Z}$ span the nullspace of $\mathbf{A}$ then every vector $\mathbf{y}$ in that nullspace can be written as $\mathbf{Zt}$ for some $\mathbf{t} \in \mathbb{R}^{n-m}$, and in §22.1.2 we used that fact twice to find vectors in $\mathbf{y}$–space corresponding to vectors in $\mathbf{t}$–space. In `rneqplot.m` it is necessary to find the vector in $\mathbf{t}$–space that corresponds to a vector $\mathbf{y}$ that is in the nullspace of $\mathbf{A}$, and this can also be done using $\mathbf{Z}$. Of course $\mathbf{Z}$ is never square so it has no inverse, but if the basis it contains is orthonormal then $\mathbf{t} = \mathbf{Z}^\top\mathbf{y}$, because

$$\mathbf{t} = \mathbf{Z}^\top\mathbf{y} = \mathbf{Z}^\top(\mathbf{Zt}) = \mathbf{It} = \mathbf{t}.$$

Then

$$\mathbf{y} = \mathbf{Zt} = \mathbf{Z}(\mathbf{Z}^\top\mathbf{y}) \qquad \text{so} \qquad \mathbf{ZZ}^\top\mathbf{y} = \mathbf{y}$$

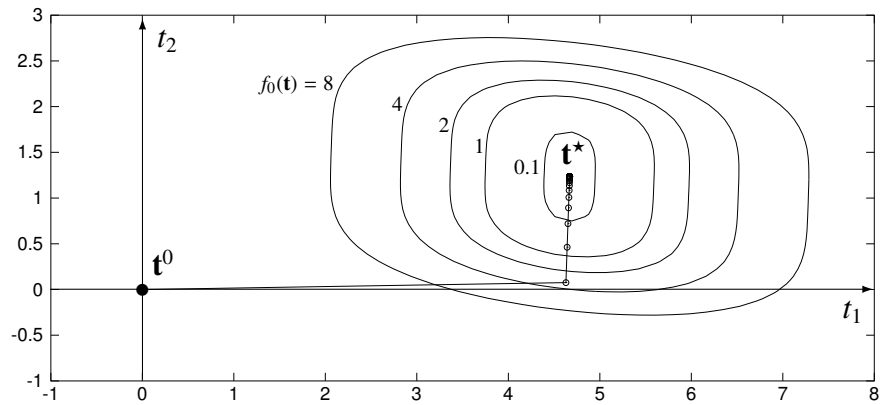
for every $\mathbf{y}$ that is in the nullspace of $\mathbf{A}$, even though in general $\mathbf{ZZ}^\top \neq \mathbf{I}$ (this is another way


```
 1 % rneqplot.m: plot convergence trajectory of rneq.m solving rnt
 2 clear; clf; set(gca,'FontSize',20)
 3
 4 A=[3,-1,-2,-1;-4,1,5,2];
 5 b=[-1;3];
 6 xbar=[-2;-5;0;0];
 7 Z=null(A);
 8 for kmax=0:34
 9     [xstar,k,rc,nm]=rneq(@rntg,@rnth,A,b,kmax,1e-6);
10     t=Z'*(xstar-xbar);
11     t1(kmax+1)=t(1);
12     t2(kmax+1)=t(2);
13 end
14
15 tl=[-1;-1];
16 th=[ 8; 3];
17 ng=50;
18 for i=1:ng;
19     t1i(i)=tl(1)+(th(1)-tl(1))*((i-1)/(ng-1));
20     for j=1:ng;
21         t2i(j)=tl(2)+(th(2)-tl(2))*((j-1)/(ng-1));
22         t=[t1i(i);t2i(j)];
23         x=Z*t+xbar;
24         zi(j,i)=rnt(x);
25     end
26 end
27 v=[0.1,1,2,4,8];
28
29 hold on
30 axis([tl(1),th(1),tl(2),th(2)],'equal')
31 contour(t1i,t2i,zi,v)
32 plot(t1,t2)
33 plot(t1,t2,'o')
34 hold off
35 print -deps -solid rneq.eps
```

of defining the nullspace of $\mathbf{A}$). Recalling from §22.1.0 that $\mathbf{x} = \mathbf{y} + \bar{\mathbf{x}}$, where $\mathbf{A}\bar{\mathbf{x}} = \mathbf{b}$, we can move back and forth between $\mathbf{t}$–space and $\mathbf{x}$–space by using the formulas

$$\begin{aligned} \mathbf{x} &= \mathbf{Z}\mathbf{t} + \bar{\mathbf{x}} \\ \mathbf{t} &= \mathbf{Z}^{\top}(\mathbf{x} - \bar{\mathbf{x}}). \end{aligned}$$

This program begins [4-6] by stating the data for `rnt` and [7] finding an orthonormal basis `Z` for the nullspace of `A`. Then [8-13] it solves the problem repeatedly, each time allowing `rneq.m` to use only `kmax` iterations. To keep `rneq.m` simple I did *not* make it serially reusable, so the only way we can capture the convergence trajectory is by using this approach even though it is very inefficient (in §26.3 we shall see that it has other drawbacks as well). Each point `xstar` returned by `rneq.m` is [10] transformed to $\mathbf{t}$–space and its coordinates are [11-12] saved for plotting later. Next [15-27] the program generates coordinates on a grid of points in $\mathbf{t}$–space, [23] transforms each point to $\mathbf{x}$–space, and [24] saves the function value there for contouring later. The last stanza plots [31] contours of the objective and [32-33] the convergence trajectory of the algorithm, producing the picture on the next page.



The starting point $\mathbf{x}^0 = \bar{\mathbf{x}} = [-2, -5, 0, 0]^\top$ is the origin in the t_1–t_2 hyperplane. Notice that the contours of $f_0(\mathbf{t})$ are not ellipses, and that reduced-Newton descent stutters its way from $\mathbf{t}^1$ to $\mathbf{t}^\star$ along a perfectly straight line.

22.4 Exercises

22.4.1 [E] What properties make a nonlinear program a *quadratic program*? Why are quadratic programs of special interest? Give an algebraic statement that can describe any quadratic program.

22.4.2 [P] In §22.1 and §22.2 we studied special-purpose algorithms for solving constrained quadratic programs. These problems can also be solved by general-purpose nonlinear programming methods introduced in earlier Chapters, and those methods can be specialized to take advantage of the structure of quadratic programs. (a) Use the `auglag.m` routine of §20.2.4 to solve the `qp1` problem. (b) Use the `nlpin.m` routine of §21.3.1 to solve the `qp4` problem. (c) How might these algorithms be specialized to exploit the special structure of a quadratic program? Hint: see [5, §16.6]. (d) Why might the special algorithms work better than the general ones for problems having linear constraints?

22.4.3 [E] Name one method for quadratic programming that is *not* discussed in this Chapter. The methods that *are* discussed in this Chapter are all based on the same general approach; what is it called?

22.4.4 [E] Is a quadratic program easier to solve when it has equality constraints, or when it has inequality constraints? Why?

22.4.5 [E] Suppose that $\bar{\mathbf{x}}$ is a feasible point for $\mathbf{Ax} = \mathbf{b}$. What substitution of variables can be used to write these equations as a homogeneous system?

22.4.6 [H] Suppose we have an $m \times n$ linear system $\mathbf{Ay} = \mathbf{0}$, with $m \leq n$. (a) How can we deduce formulas giving m of the variables in terms of the others? (b) How can we use such formulas to find m vectors, each of length n, that span the nullspace of $\mathbf{A}$?

22.4.7 [E] If `A` is a matrix with fewer rows than columns, describe the result `Z` of the MATLAB statement `Z=null(A)`.

22.4.8 [E] What makes a set of vectors *orthonormal*?

22.4.9 [H] If the columns of $\mathbf{Z}$ are basis vectors for the nullspace of a matrix $\mathbf{A}$, explain why any vector $\mathbf{y}$ that satisfies $\mathbf{Ay} = \mathbf{0}$ can be written as $\mathbf{y} = \mathbf{Zt}$. How long is the vector $\mathbf{t}$?

22.4.10 [H] In §22.1.2 we derived a formula for the reduced-Newton direction $\mathbf{d}^k$. (a) What is a *reduced Hessian matrix*? (b) Explain the derivation of $\mathbf{p}^k$, the direction of Newton descent in $\mathbf{t}$-space. (c) Why is $\mathbf{Zp}^k$ the corresponding direction in $\mathbf{y}$-space? (d) Why is this also the corresponding direction in $\mathbf{x}$-space? (e) State the formula for $\mathbf{d}^k$ in terms of $\mathbf{x}^k$.

22.4.11 [H] Suppose that $\mathbf{VU} = \mathbf{Z}$ where $\mathbf{V}$, $\mathbf{U}$ and $\mathbf{Z}$ are matrices and $\mathbf{U}$ is upper-triangular. How can the matrix equation be solved for $\mathbf{V}$ (a) using elementary arithmetic operations; (b) using the MATLAB division operator? (c) Explain the factor-and-solve approach that we used to compute $\mathbf{Z}[\mathbf{Z}^\top\mathbf{QZ}]^{-1}\mathbf{Z}^\top$.

22.4.12 [H] Show that solving $\mathbf{Z}^\top\mathbf{QZp}^k = -\mathbf{Z}^\top\mathbf{Q}(\bar{\mathbf{x}} + \mathbf{Zt}^k) - \mathbf{Z}^\top\mathbf{c}$ for $\mathbf{p}^k$ in the reduced-Newton algorithm is equivalent to applying Newton's method for systems to the Lagrange conditions for the original quadratic program.

22.4.13 [H] An alternative to using the factor-and-solve approach to find $\mathbf{Z}[\mathbf{Z}^\top\mathbf{QZ}]^{-1}\mathbf{Z}^\top$ is to use the conjugate gradient algorithm of §14.4 to solve $\mathbf{Z}^\top\mathbf{QZp}^k = -\mathbf{Z}^\top\mathbf{Q}(\bar{\mathbf{x}} + \mathbf{Zt}^k) - \mathbf{Z}^\top\mathbf{c}$ for $\mathbf{p}^k$. (a) Explain how to do this. (b) Are there any advantages to this approach?

22.4.14 [E] In the active set algorithm, if $\overline{\mathrm{m}} = n$ describe the feasible set of the equality-constrained subproblem.

22.4.15 [E] What are the return variables from the MATLAB routine `qpeq.m`, what do they represent, and what values can they take on? Why does the routine use `hfact.m`?

22.4.16 [P] Consider this equality-constrained quadratic program [4, Example 15.1].

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & q(\mathbf{x}) & = & \frac{1}{2}x_1^2 - \frac{1}{2}x_3^2 + 4x_1x_2 + 3x_1x_3 - 2x_2x_3 \\ \text{subject to} & \mathbf{Ax} & = & x_1 - x_2 - x_3 = -1 \end{array}$$

(a) Use `qpeq.m` to find $\mathbf{x}^\star = [-\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]^\top$. (b) Show that $\mathbf{Q}$ is indefinite. (c) Is $\mathbf{x}^\star$ a minimizing point? Explain. (d) If in a quadratic program $\mathbf{Q}$ is positive definite, can the reduced Hessian ever be non-positive-definite? If no, prove it; if yes, provide an example.

22.4.17 [P] Consider the following problem [4, Exercise 2.1].

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & q(\mathbf{x}) & = & \frac{1}{2}\mathbf{x}^\top\mathbf{Qx} \\ \text{subject to} & \mathbf{Ax} & = & \mathbf{b} \end{array} \quad \mathbf{Q} = \begin{bmatrix} 0 & -13 & -6 & -3 \\ -13 & 23 & -9 & 3 \\ -6 & -9 & -12 & 1 \\ -3 & 3 & 1 & -1 \end{bmatrix} \; \mathbf{A} = \begin{bmatrix} 2 & 1 & 2 & 1 \\ 1 & 1 & 3 & -1 \end{bmatrix} \; \mathbf{b} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

(a) Apply `qpeq.m` to this problem. Is the point you found optimal? How do you know? (b) Write a MATLAB program to plot an error curve showing the convergence of `qpeq.m` when it used to solve this problem. What is the algorithm's order of convergence?

22.4.18 [P] Use `qpeq.m` to solve the following problem [5, Example 16.2].

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & 3x_1^2 + 2x_1x_2 + x_1x_3 + \frac{5}{2}x_2^2 + 2x_2x_3 + 2x_3^2 - 8x_1 - 3x_2 - 3x_3 \\ \text{subject to} & x_1 + x_3 = 3 \\ & x_2 + x_3 = 0 \end{array}$$

Show that the reduced Hessian is positive definite. Is $\mathbf{x}^\star = [2, -1, 1]^\top$ optimal?

22.4.19 [H] Our study of equality-constrained quadratic programs in §22.1 was based on an analysis of the example problem `qp1`. Suppose that instead of that problem we had begun with this one, which has the same constraints but a different objective.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & q(\mathbf{x}) & = & x_1^2 + 2x_1x_2 + 3x_2x_3 + 4x_3x_4 + x_4^2 \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \begin{bmatrix} 3x_1 - x_2 - 2x_3 - x_4 \\ -4x_1 + x_2 + 5x_3 + 2x_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \end{array}$$

(a) What parts of the development in §22.1 are affected by this change? (b) Show that the problem can be recast as the following unconstrained optimization.

$$\underset{y_3\ y_4}{\text{minimize}} \quad q(y_3, y_4) = 72y_3^2 + 6y_4^2 + 42y_3y_4 - 85y_3 - 22y_4 + 24$$

(c) Find a stationary point $\bar{\mathbf{y}}$ of this function, and the corresponding $\bar{\mathbf{x}}$. (d) Characterize $\bar{\mathbf{y}}$ by describing the behavior of the reduced objective. (e) Apply `qpeq.m` to the $\mathbf{x}$ version of this problem. Does it find a minimizing point? Explain.

22.4.20 [H] In §22.1.1 we found $\mathbf{v}$ and $\mathbf{w}$, basis vectors spanning the nullspace of $\mathbf{A}$, in two ways. In the first approach we used substitution to eliminate two of the variables and saw that basis vectors emerge naturally from that process. (a) Describe the second way in which we found $\mathbf{v}$ and $\mathbf{w}$. (b) Explain how it is possible to deduce the formulas $y_1 = 3y_3 + y_4$ and $y_2 = 7y_3 + 2y_4$ from the basis vectors $\mathbf{v}$ and $\mathbf{w}$.

22.4.21 [P] In §22.1.1 we found $\mathbf{v}$ and $\mathbf{w}$, basis vectors spanning the nullspace of $\mathbf{A}$, in two ways. In the second approach we calculated them directly from $\mathbf{A}$ by using a procedure that involves solving $\mathbf{U}\mathbf{y} = \mathbf{0}$ for different values of the basic variables. Write a MATLAB function `Z=strang(A)` that implements this algorithm, and show that your code produces the basis vectors $\mathbf{v}$ and $\mathbf{w}$ that we found by hand.

22.4.22 [P] In §22.1.1 we used the MATLAB function `null()` to find basis vectors `z1` and `z2` spanning the nullspace of $\mathbf{A}$, and wrote $\mathbf{y}^\star = [\frac{175}{89}, \frac{404}{89}, \frac{54}{89}, \frac{13}{89}]^\top \approx$ `4.8082*z1+1.3199*z2`.

Show how multiple regression (see §8.6.2) can be used to find the coefficients of `z1` and `z2` in this formula.

22.4.23 [E] In solving an equality-constrained quadratic program by the method of §22.1.2 each iterate $\mathbf{x}^k$ satisfies $\mathbf{Ax} = \mathbf{b}$, yet $\mathbf{b}$ does not appear in the formula for the reduced-Newton direction $\mathbf{d}^k$. How does the right-hand side vector of the equality constraints enter the solution process, so that $\mathbf{Ax}^\star = \mathbf{b}$ at the end?

22.4.24 [H] If $\mathbf{A}_{m\times n}$ has full row rank then what happens in solving the equality-constrained quadratic program if (a) $n > m$; (b) $n = m$? (c) Is it possible to have $n < m$?

22.4.25 [H] Show that in the reduced-Newton algorithm of §22.1.2 any nullspace basis $\mathbf{Z}$ yields the same descent direction $\mathbf{p}$.

22.4.26 [H] (a) Find the dual of the equality-constrained quadratic program. (b) Find the dual of the inequality-constrained quadratic program.

22.4.27 [E] In §16.3 we developed a systematic method for finding all the solutions to a set of KKT conditions. How is the working set $\mathcal{W}$ of §22.2 related to that method? What values can the w_i take on, and what do they mean?

22.4.28 [E] What precisely is a *sticking constraint*? A *blocking constraint*? If a sticking constraint is removed, could the objective function go up? Could it go down? If a blocking constraint is activated, could the objective function go up? Could it go down?

22.4.29 [H] Outline the steps of the active set algorithm presented in §22.2.0. Why is it necessary to compute at each $\mathbf{x}^k$ the corresponding Lagrange multipliers $\boldsymbol{\lambda}^k$? What formula can be used to do that? Why is it necessary to compute at each $\mathbf{x}^k$ the values of the constraints that are assumed to be inactive?

22.4.30 [E] Explain the convergence trajectory of `qpin.m` when it is used to solve `qp5`.

22.4.31 [E] In the `qp5` problem of §22.2.1, $\mathbf{x} = \mathbf{0}$ is feasible for $\mathbf{Ax} \le \mathbf{b}$ and could be used as a starting point. Why is the origin not necessarily feasible for an arbitrary quadratic program?

22.4.32 [H] The purpose of the `feas.m` routine in §22.2.1 is to find some point $\mathbf{x}^0$ that is in $\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} \le \mathbf{b}\}$. (a) Describe in words the heuristic that the routine employs to do this. (b) Explain the construction of the initial linear programming tableau `T`. How many rows and columns are in each partition? (c) How many slack variables are basic in `T`? (d) How many slack variables must be nonbasic in `T1` for its basic feasible solution to correspond to a vertex of $\mathbb{X}$? (e) The `feas.m` routine transforms `T` into `T1` by using `newseq.m` followed by `phase1.m`. What is necessary to ensure that this approach produces a `T1` having n slack columns nonbasic? (f) What are the properties of the $\mathbf{x}^0$ returned by `feas.m` if `T1` has *fewer* than n slack columns nonbasic? (g) Can it ever happen that the $\mathbf{x}^0$ returned by `feas.m` is not in $\mathbb{X}$? (h) If the $\mathbf{x}^0$ returned by `feas.m` is in $\mathbb{X}$, can it ever happen that it is not a vertex of the polyhedron defined by $\mathbf{Ax} \le \mathbf{b}$?

22.4.33 [H] In the active set algorithm of §22.2.2 the longest step we can take without violating inequality constraint i is sometimes limited to

$$\alpha \le \frac{b_i - A_i\mathbf{x}^k}{A_i\mathbf{d}^k}.$$

(a) When must this limit be imposed on α? (b) How is this minimum-ratio rule related to the one we used in §2.4.4 for selecting a pivot row in the simplex method for linear programming?

22.4.34 [E] In §16.10 we solved the overdetermined stationarity conditions of a nonlinear program for $\boldsymbol{\lambda}$ by using linear programming to minimize the sum of the absolute values of the row deviations. Why can't we take that approach in the active set algorithm of §22.2, rather than using least squares to find $\boldsymbol{\lambda}^k$?

22.4.35 [E] Suppose that in solving an inequality-constrained quadratic program, we find a point $\bar{\mathbf{x}}$ that minimizes $q(\mathbf{x})$ on the flat defined by the current working set. If the Lagrange multiplier corresponding to an active constraint turns out to be negative, can we drop that constraint from the working set? Explain.

22.4.36 [H] In §22.2.0, I claimed that "If inequality i will be slack at $\mathbf{x}^\star$ but, not knowing that ahead of time, we assume it is an equality by insisting that $\lambda_i \neq 0$, then if we find a feasible stationary point the corresponding λ_i comes out negative." It is also unfortunately true that at a non-optimal KKT point λ_i might be negative for a constraint that we have *correctly* assumed is active at optimality. (a) Show that this is true by solving the KKT conditions for the following problem [4, Example 15.7] assuming $\mathcal{W} = [0, 1, 1]$.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & q(\mathbf{x}) & = & \frac{1}{2}(x_1-3)^2 + (x_2-2)^2 \\ \text{subject to} & -2x_1 + x_2 & \le & 0 \\ & x_1 + x_2 & \le & 4 \\ & -x_2 & \le & 0 \end{array}$$

(b) What happens inside the active set algorithm if at some iteration we mistakenly release a constraint that will actually turn out to be tight at optimality?

22.4.37 [H] In §22.2.0 I claimed that "If inequality i will be tight at $\mathbf{x}^\star$ but we assume it is slack and take it out of the problem by insisting that $\lambda_i = 0$, then the stationary point we find violates the ignored constraint... If this happens we should add that blocking constraint to the working set..." Yet when activating blocking constraints in `qpin.m` we 92 ignore some blocking constraints if there are more than n of them. What happens inside `qpin.m` if at some iteration we neglect to activate a constraint that will actually turn out to be tight at optimality?

22.4.38 [E] The `getlgm.m` routine of §22.2.3 uses the factor-and-solve approach to find `Aplus`. This involves finding the Cholesky factors of the matrix `Abar*Abar'`, for which I used the `hfact.m` routine. Why did I invoke `hfact.m` with $\gamma = 1$? What happens if $\bar{\mathbf{A}}\bar{\mathbf{A}}^\top$ is not positive definite?

22.4.39 [H] In §22.2.2 we derived rules for restricting the reduced-Newton steplength α in the active set algorithm so as to avoid violating the inequality constraints that are not in the current working set. The table below summarizes the four categories of constraint status that affect how α must be restricted to prevent the contemplated step from violating constraint i. For each cell of the table, specify the corresponding restriction on α.

	$A_i\mathbf{d}^k \le 0$	$A_i\mathbf{d}^k > 0$
$A_i\mathbf{x}^k = b_i$	(a)	(b)
$A_i\mathbf{x}^k < b_i$	(c)	(d)

22.4.40 [H] The final paragraph in §22.2.3 discusses the use of the MATLAB "right division" operator `/` to solve the matrix equations $\mathbf{U}^\top\mathbf{V} = \bar{\mathbf{A}}$ and $\mathbf{U}\mathbf{A}^+ = \mathbf{V}$. The matrices involved in these calculations differ in size, and some of them are not square. (a) Explain why using MATLAB to solve these equations required first transposing both sides. (b) Find the dimensions of $\bar{\mathbf{A}}$, $\bar{\mathbf{A}}\bar{\mathbf{A}}^\top$, $(\bar{\mathbf{A}}\bar{\mathbf{A}}^\top)^{-1}$, $\mathbf{U}$, $\mathbf{U}^\top$, $\mathbf{V}$, $\mathbf{V}^\top$, and $\mathbf{A}^+$. (c) Show that all of the operations described in the text are conformable. (d) What is required of the dimensions of matrices $\mathbf{E}$ and $\mathbf{F}$ in order for the MATLAB operation `G=E/F` to be conformable, and what are the resulting dimensions of $\mathbf{G}$? (e) In [50, §8.3] the Octave manual says that "If the system is not square... a minimum norm solution is computed." In the context of `qpin.m`, does this lead to a difference between the results that we get using the factor-and-solve approach and those we would get by computing the inverse explicitly? Explain.

22.4.41 [E] What is a *zero tolerance*, and why is it used?

22.4.42 [P] Modify `qpin.m` to perform up to 10 iterations of Newton descent in minimizing $q(\mathbf{x})$ for each working set $\mathcal{W}$. Why might this be necessary?

22.4.43 [E] Explain how the simplex method for linear programming is an active set method. How does it differ from our active set method for quadratic programming?

22.4.44 [P] The active set implementation of §22.2.4 uses `hfact.m` to factor `Z'*Q*Z` so it will modify the matrix if necessary and factor the positive definite result, but this might not lead to the successful solution of a nonconvex problem [5, p467]. Devise an inequality-constrained quadratic program having a nonconvex objective and report what happens when you attempt to solve it using `qpin.m`.

22.4.45 [E] How can the active set strategy be modified to solve quadratic programs that have both inequality and equality constraints? Can it be used to solve problems that have *only* equality constraints?

22.4.46 [E] Explain how the reduced-Newton algorithm described in §22.3 differs from (a) the nullspace quadratic programming algorithm implemented in `qpeq.m`; (b) the restricted-steplength Newton algorithm described in §17.2.

22.4.47 [P] Our active set algorithm for solving inequality-constrained quadratic programs can be generalized to solve problems in which the constraints are still linear inequalities but the objective need not be quadratic. (a) Taking the same approach that we used to generalize `qpeq.m` to produce `rneq.m`, modify `qpin.m` to produce `rnin.m`. (b) Show that your routine solves the problem that results from changing the constraints of `rnt` from $\mathbf{Ax} = \mathbf{b}$ to $\mathbf{Ax} \leq \mathbf{b}$. How do you know that your solution is correct?

22.4.48 [E] What is the MATLAB locution for making `A` a matrix with zero rows but a nonzero number of columns?

22.4.49 [H] Show that the objective function of problem `rnt` is strictly convex. Why does solving it with reduced-Newton descent require several iterations?

22.4.50 [P] Try solving `rnt` with `auglag.m`, from $[-2, -5, 0, 0]^\top$, $[-0.1, -0.6, 0.6, 0.1]^\top$, and other starting points, and explain your results.

22.4.51 [P] The equality constraints of problem `rnt` can be used to eliminate the variables x_1 and x_2 from the problem, yielding a reduced problem in x_3 and x_4 that is unconstrained. (a) Use this substitution of variables to derive the reduced problem. (b) Can you solve this unconstrained minimization analytically? Explain. (b) Confirm numerically that $\mathbf{x}^\star = [-0.1, -0.6, 0.6, 0.1]^\top$ is a stationary point for the reduced problem. (c) Confirm numerically that $\mathbf{x}^\star$ is a minimizing point of the reduced problem.

22.4.52 [H] Show that if the orthonormal columns of $\mathbf{Z}$ span the nullspace of $\mathbf{A}$ then $\mathbf{ZZ}^\top\mathbf{y} = \mathbf{y}$ if and only if $\mathbf{y}$ is a vector in the nullspace of $\mathbf{A}$.

22.4.53 [E] If a minimization routine is *not* serially reusable, how can the iterates $\mathbf{x}^k$ that it generates in the course of solving a problem be captured? What are the advantages and drawbacks of the approach you propose, compared to making the routine serially reusable?

22.4.54 [H] Several of the programs available on the NEOS web server (see §8.3.1) are based on the algorithms discussed in this Chapter [5, §16.8]. By searching the web, find out which of the programs are based on which of the algorithms.

23

Feasible-Point Methods

The classical barrier method of §19 and the interior-point algorithm of §21.3 solve general inequality-constrained nonlinear programs by approaching $\mathbf{x}^\star$ from the inside of a feasible region that has positive volume in $\mathbb{R}^n$. The classical penalty method of §18 and the augmented Lagrangian algorithm of §20.2 solve general equality-constrained nonlinear programs by approaching $\mathbf{x}^\star$ from points that are infeasible, satisfying the constraint equations only at optimality.

In §22 we studied several algorithms in which each iterate is confined to the hyper*plane,* of dimension less than n, that is defined by a set of linear constraints. Because those algorithms try to satisfy the constraints at each iteration they belong to a category called **feasible-point methods** [4, §15] [1, §10]. The algorithms developed in this Chapter are also feasible-point methods, but some of them can solve arbitrary nonlinear programs. In these algorithms each iteration is confined at least approximately to a hyper*surface* of dimension less than n, but the constraints of the original problem need not be linear.

23.1 Reduced-Gradient Methods

The reduced-Newton algorithm of §22.3, implemented in `rneq.m`, takes Newton descent steps in the flat defined by linear equality constraints. Taking steepest descent steps instead [4, p552-553] results in a reduced-gradient method, which can be generalized to solve problems having nonlinear equality constraints.

23.1.1 Linear Constraints

The original **reduced-gradient method** was proposed [158] [107, §11.6] [1, §10.6] as an extension of the simplex algorithm, so the variables were assumed to be nonnegative and the calculations were organized in a tableau. The approach suggested above, doing steepest descent in the nullspace of the equalities, is equivalent but less restrictive and much simpler. The `rsdeq.m` routine listed on the next page is the `rneq.m` routine of §22.3 modified to take full steepest descent steps in the flat defined by the constraints. This code differs from `rneq.m` only in its final stanza, so you might find it helpful to review §22.3 now.

In each iteration of steepest descent `rsdeq` finds 37 the reduced Hessian `rH` and 38 reduced gradient `rg` at the current iterate `xk` and 39 uses the formula from §10.5 to find the length of the reduced full steepest-descent step. Next it finds 40 the projection $\mathbf{t}^k$ of $\mathbf{x}^k$, and 41 $\mathbf{t}^{k+1}$ as $\mathbf{t}^k$ plus the full step in the negative reduced-gradient direction. Then 42 it

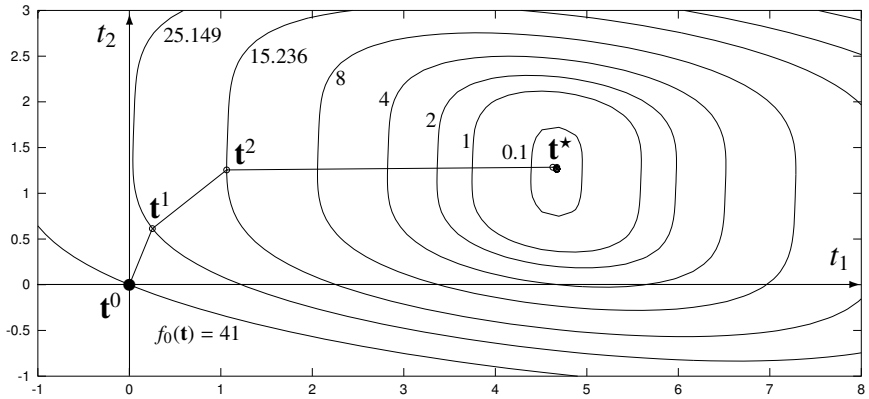

```
 1 function [xstar,k,rc]=rsdeq(grd,hsn,A,b,kmax,epz)
 2 % minimize f(x) subject to Ax=b
 3
 4 % size up the problem
 5   n=size(A,2);                      % number of variables
 6   m=size(A,1);                      % number of equality constraints
 7   k=0;                              % no iterations yet
 8
 9 % find a starting point and a basis for nullspace of A
10   xzero=zeros(n,1);                 % use the origin if unconstrained
11   if(m > 0)                         % if there are constraints
12      T=[0,zeros(1,n);b,A];                                % tableau
13      [Tnew,S,tr,mr,rc0]=newseq(T,m+1,n+1,[1:m+1],m+1);  % seek basis
14      if(rc0 ~= 0)                                         % success?
15         rc=3;                      % report constraints inconsistent
16         return                     % and give up
17      end
18      for j=1:n                     % extract
19          if(S(j) ~= 0)             % the basic solution
20             xzero(j)=Tnew(S(j),1); % to use
21          end                       % as the starting point
22      end
23      if(mr-1 == n)                 % is the system square?
24         xstar=xzero;               % if so this is the optimal point
25         rc=0;                      % report success
26         return                     % and return it
27      end
28      A=Tnew(2:mr,2:n+1);           % A without redundant constraints
29      Z=null(A);                    % get a basis for the nullspace
30   else                             % no constraints
31      Z=eye(n);                     % Z=I does sd unconstrained
32   end
33
34 % full-step steepest descent in the flat defined by the constraints
35   xk=xzero;                        % start here
36   for k=1:kmax                     % do up to kmax iterations
37       rH=Z'*hsn(xk)*Z;             % Hessian in the flat
38       rg=Z'*grd(xk);               % gradient in the flat
39       astar=(rg'*rg)/(rg'*rH*rg);  % full step
40       tk=Z'*(xk-xzero);            % current point in the flat
41       tkp=tk+astar*(-rg);          % new point in the flat
42       xk=Z*tkp+xzero;              % new point in R^n
43       if(norm(rg) <= epz)          % converged?
44          xstar=xk;                 % yes; save optimal point
45          rc=0;                     % report success
46          return                    % and return
47       end
48   end                              % of reduced Newton steps
49   xstar=xk;                        % save the current point
50   rc=1;                            % report out of iterations
51 end
```

transforms $\mathbf{t}^{k+1}$ back to $\mathbf{x}$–space as the updated `xk`. If [43-47] the reduced gradient is shorter than `epz` the current iterate is [44] accepted as `xstar` and the routine returns with `rc=0` to signal success. If convergence is not achieved in `kmax` iterations [49] the current iterate is also taken as `xstar` but the routine returns `rc=1` to signal that the iteration limit was met.

To test `rsdeq.m` I used it to solve the `rnt` problem of §22.3 as shown in the Octave session on the next page. Then, using a program similar to `rneqplot.m`, I plotted the algorithm's


```
octave:1> A=[3,-1,-2,-1;-4,1,5,2];
octave:2> b=[-1;3];
octave:3> [x0]=rsdeq(@rntg,@rnth,A,b,0,1e-6)
x0 =

  -2.00000
  -5.00000
   0.00000
   0.00000

octave:4> f0=rnt(x0)
f0 = 41.000
octave:5> [x1]=rsdeq(@rntg,@rnth,A,b,1,1e-6)
x1 =

  -1.778744
  -4.654548
  -0.097060
   0.512438

octave:6> Z=null(A);
octave:7> t1=Z'*(x1-x0)
t1 =

   0.25136
   0.61410

octave:8> f1=rnt(x1)
f1 =  25.149
octave:9> [x2]=rsdeq(@rntg,@rnth,A,b,2,1e-6)
x2 =

  -1.355805
  -3.805254
  -0.093644
   0.925126

octave:10> t2=Z'*(x2-x0)
t2 =

   1.0629
   1.2559

octave:11> f2=rnt(x2)
f2 =  15.236
octave:12> [xstar,k,rc]=rsdeq(@rntg,@rnth,A,b,10000,1e-6)
xstar =

  -0.098889
  -0.598890
   0.598889
   0.104443

k =  3321
rc = 0
octave:13> quit
```

convergence trajectory as shown on the page after. Although `rsdeq.m` requires 3321 iterations to meet the convergence criterion of `norm(rg)` $\leq 10^{-6}$ in solving `rnt`, it is clear from



the picture that $\mathbf{t}^3$ is already a good approximation to $\mathbf{t}^\star$ (see Exercise 23.3.5). Because each $\mathbf{x}^k$ is feasible $\mathbf{x}^3$ might, depending on the application that gave rise to the `rnt` problem, be close enough to use in place of $\mathbf{x}^\star$.

23.1.2 Nonlinear Constraints

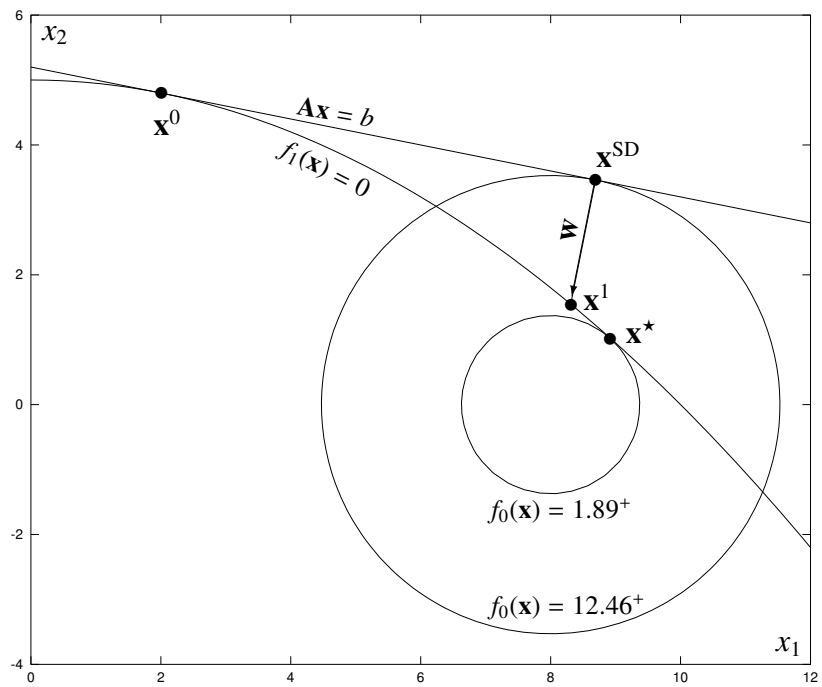
A differentiable function $f_i(\mathbf{x})$ can be approximated in the vicinity of $\mathbf{x}^k$ by its first-order Taylor's series expansion about that point,

$$f_i(\mathbf{x}) \approx f_i(\mathbf{x}^k) + \nabla f_i(\mathbf{x}^k)^\top(\mathbf{x} - \mathbf{x}^k),$$

so a set of differentiable nonlinear constraints $f_i(\mathbf{x}) = 0,\ i = 1 \ldots m$ can be approximated near $\mathbf{x}^k$ by the linear constraints $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \\ \vdots \\ \nabla f_m(\mathbf{x}^k)^\top \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top\mathbf{x}^k - f_1(\mathbf{x}^k) \\ \vdots \\ \nabla f_m(\mathbf{x}^k)^\top\mathbf{x}^k - f_m(\mathbf{x}^k) \end{bmatrix}.$$

If in the reduced-gradient algorithm of §23.1.1 we used these formulas to recompute $\mathbf{A}$ and $\mathbf{b}$ at each iteration, then each steepest-descent step would be confined to the flat that approximates the nonlinear constraints at $\mathbf{x}^k$. Of course the resulting next point would probably not fall precisely on the curved constraint surface, which it must do if the linear approximation there is to represent the surface accurately. To restore feasibility we could move from the new point on the flat, in a direction orthogonal to the flat, just far enough to satisfy the original constraints. Then we could use that feasible point for $\mathbf{x}^{k+1}$. Updating the linearization of the constraints at each iteration, taking one steepest descent step in the resulting flat, and restoring feasibility by moving outside the flat, is the essence of the **generalized reduced-gradient algorithm** or **GRG** [1, 612-613]. The idea is illustrated in the graph on the next page, which shows the first GRG iteration in solving the problem given below the picture.



I will call this problem `grg2` because it has $n = 2$ variables (see §28.7.36).

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1 - 8)^2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & \frac{1}{20}x_1^2 + x_2 - 5 = 0 \end{array}$$

At the feasible starting point $\mathbf{x}^0 = [2, \frac{24}{5}]^\top$, the nonlinear constraint $f_1(\mathbf{x}) = 0$ has the linear approximation $\mathbf{A}\mathbf{x} = b$ where

$$\begin{aligned} \mathbf{A} &= \nabla f_1(\mathbf{x}^0)^\top = [\tfrac{1}{10}x_1^0, 1] = [\tfrac{1}{5}, 1] \\ b &= \nabla f_1(\mathbf{x}^0)^\top \mathbf{x}^0 - f_1(\mathbf{x}^0) = [\tfrac{1}{5}, 1]\begin{bmatrix} 2 \\ \frac{24}{5} \end{bmatrix} - \left(\tfrac{1}{20}2^2 + \tfrac{24}{5} - 5\right) = \tfrac{26}{5} \end{aligned}$$

or $\frac{1}{5}x_1 + x_2 = \frac{26}{5}$. This flat has dimension $n - m = 2 - 1 = 1$ so it is just the tangent line drawn above. A single steepest-descent step minimizes $f_0(\mathbf{x})$ along that line to yield the point $\mathbf{x}^{\mathrm{SD}} = [\frac{113}{13}, \frac{45}{13}]^\top$, at which the nonlinear equality is far from satisfied. Moving orthogonal to the flat until touching the constraint produces the next iterate $\mathbf{x}^1 \approx [8.3095, 1.5476]^\top$.

You should be aware that other authors use the name GRG to refer to algorithms that are slightly different from the one pictured above. For example, the algorithm described in [3, p311-315] omits the feasibility-restoration step and in fact generates infeasible iterates when used to solve the example given there. The algorithm described in [4, §15.6] restores feasibility, but it uses Newton descent rather than steepest descent and so might be described more precisely as a generalized reduced Newton method (see Exercise 23.3.13).

In the graph above it is easy to see the orthogonal direction in which we must move to restore feasibility, but how can this correction step be accomplished algebraically?

Points $\mathbf{y} = \mathbf{x} - \mathbf{x}^0$ that are on the tangent line are in the nullspace of $\mathbf{A}$,

$$\mathfrak{Z} = \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{y} = \mathbf{0}\}.$$

That means *each row of* $\mathbf{A}$ *is orthogonal to* $\mathbf{y}$. In our example, $\mathbf{A} = [\frac{1}{5}, 1]$ has only one row and that row is orthogonal to every vector $\mathbf{y}$ in the tangent line. For example,

$$\mathbf{y} = \mathbf{x}^{\mathrm{SD}} - \mathbf{x}^0 = [\tfrac{113}{13}, \tfrac{45}{13}]^\top - [2, \tfrac{24}{5}]^\top = [\tfrac{87}{13}, -\tfrac{87}{65}]^\top$$

is orthogonal to the row of $\mathbf{A}$ because

$$\mathbf{A}\mathbf{y} = [\tfrac{1}{5}, 1]\begin{bmatrix} \frac{87}{13} \\ -\frac{87}{65} \end{bmatrix} = 0.$$

In the picture the vector $[\frac{1}{5}, 1]^\top$ would point up and to the right; to get from $\mathbf{x}^{\mathrm{SD}}$ to $\mathbf{x}^1$ we moved in the opposite direction by the vector $\mathbf{w}$ as shown.

In general $\mathbf{A}$ has m rows, and each of them is orthogonal to every vector $\mathbf{y}$ that is in the nullspace of $\mathbf{A}$. In fact, every vector $\mathbf{w}$ in the space that is spanned by the rows of $\mathbf{A}$ is orthogonal to the flat. In other words, every vector in the space that is spanned by the columns of $\mathbf{A}^\top$ is orthogonal to the flat. This set of vectors is called the column space or **range space** of $\mathbf{A}^\top$ [147, §2.4] [4, §3.2],

$$\mathfrak{R} = \{\mathbf{w} \in \mathbb{R}^n \mid \mathbf{w} = \mathbf{A}^\top\mathbf{p} \text{ for some } \mathbf{p} \in \mathbb{R}^m\}.$$

Just as we found an orthonormal basis for the nullspace of $\mathbf{A}$ by using the MATLAB command `Z=null(A)`, we can find an orthonormal basis for the range space of $\mathbf{A}^\top$ by using the MATLAB command `R=orth(A')`. The Octave session on the next page performs these calculations for our example, and shows that the vector `y` we found above is a linear combination of the one basis column in `Z` and our vector `w` is a linear combination of the one basis column in `R`. In finding an orthonormal basis for the range space, just as in finding an orthonormal basis for the nullspace, MATLAB uses the singular-value decomposition [150, §5].

Now we can confirm the claim that each vector in the nullspace of $\mathbf{A}$ is orthogonal to every vector in the range space of $\mathbf{A}^\top$ by computing the dot product

$$\mathbf{w}^\top\mathbf{y} = (\mathbf{A}^\top\mathbf{p})^\top\mathbf{y} = \mathbf{p}^\top(\mathbf{A}\mathbf{y}) = \mathbf{p}^\top\mathbf{0} = 0.$$

This property makes $\mathfrak{Z}$ and $\mathfrak{R}$ **orthogonal subspaces**. Because $\mathfrak{Z}$ contains *all* vectors $\mathbf{y}$ that are in the nullspace and $\mathfrak{R}$ contains *all* vectors $\mathbf{w}$ that are orthogonal to the nullspace, these two subspaces account for all of $\mathbb{R}^n$ and each is said to be the **orthogonal complement** of the other [147, §2.5]. That means that any vector $\mathbf{x} \in \mathbb{R}^n$ can be written uniquely as the


```
octave:1> A=[1/5,1];
octave:2> Z=null(A)
Z =

  -0.98058
   0.19612

octave:3> y=[87/13;-87/65]
y =

   6.6923
  -1.3385

octave:4> -6.8248*Z
ans =

   6.6923
  -1.3385

octave:5> R=orth(A')
R =

   0.19612
   0.98058

octave:6> w=[1/5;1]
w =

   0.20000
   1.00000

octave:7> 1.0198*R
ans =

   0.20000
   1.00000
```

sum of a nullspace component $\mathbf{y} \in \mathfrak{Z}$ and a range space component $\mathbf{w} \in \mathfrak{R}$, or

$$\mathbf{x} = \mathbf{y} + \mathbf{w} = \mathbf{Z}_{n\times(n-m)}\mathbf{t}_{(n-m)\times 1} + \mathbf{R}_{n\times m}\mathbf{p}_{m\times 1}.$$

The elements of $\mathbf{t}$ are as usual the coefficients in a linear combination of the columns of $\mathbf{Z}$, and the elements of $\mathbf{p}$ are the coefficients in a linear combination of the columns of the range space basis matrix $\mathbf{R}$. To decompose a vector $\mathbf{x}$ into its nullspace and range space components, we can find these coefficients by solving the linear system,

$$\begin{bmatrix} \mathbf{Z} & \vdots & \mathbf{R} \end{bmatrix} \begin{bmatrix} \mathbf{t} \\ \cdots \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{x} \end{bmatrix}$$

which has a total of n variables and in which the **basis matrix** $\mathbf{B} = [\mathbf{Z} : \mathbf{R}]$ is $n \times n$. This matrix has columns that are orthonormal vectors so it is an **orthogonal matrix** [147, p119-122] and has the inverse $\mathbf{B}^{-1} = \mathbf{B}^{\top}$. The Octave session below solves the linear system above to find the nullspace and range space components of the vector $\mathbf{d}^1 = \mathbf{x}^1 - \mathbf{x}^0$ in our grg2 example, and shows that they are equal to $(\mathbf{x}^{\mathrm{SD}} - \mathbf{x}^0) \in \mathfrak{Z}$ and $(\mathbf{x}^1 - \mathbf{x}^{\mathrm{SD}}) \in \mathfrak{R}$.


```
octave:1> A=[1/5,1];
octave:2> x0=[2;24/5];
octave:3> xsd=[113/13;45/13];
octave:4> x1=[8.30951894845300;1.54759474226502];
octave:5> Z=null(A);
octave:6> R=orth(A');
octave:7> B=[Z,R]
B =

  -0.98058   0.19612
   0.19612   0.98058

octave:8> d=x1-x0
d =

   6.3095
  -3.2524

octave:9> tp=B\d
tp =

  -6.8248
  -1.9518

octave:10> Z*tp(1)
ans =

   6.6923
  -1.3385

octave:11> xsd-x0
ans =

   6.6923
  -1.3385

octave:12> R*tp(2)
ans =

  -0.38279
  -1.91394

octave:13> x1-xsd
ans =

  -0.38279
  -1.91394
```

To complete the feasibility-restoration step in iteration k of the GRG algorithm [4, p583] we need to find a point $\mathbf{x}^{k+1} = \mathbf{x}^{\mathrm{SD}} + \mathbf{w}$ in $\mathfrak{R}$ where the nonlinear constraints are satisfied. For $\mathbf{w}$ to be in the range space of $\mathbf{A}^{\mathrm{T}}$ we must be able to write it as $\mathbf{w} = \mathbf{Rp}$, and for the constraints to be satisfied we need

$$\begin{aligned} f_1(\mathbf{x}^{\mathrm{SD}} + \mathbf{Rp}) &= 0 \\ &\vdots \\ f_m(\mathbf{x}^{\mathrm{SD}} + \mathbf{Rp}) &= 0. \end{aligned}$$

These m nonlinear algebraic equations in the m unknowns $p_1 \dots p_m$ can be solved by using Newton's method for systems. Recall from §21.2 that given an estimate $\mathbf{p}^s$ of the solution we solve $\mathbf{f}(\mathbf{p}^s) + \mathbf{J}(\mathbf{p}^s)\boldsymbol{\Delta} = \mathbf{0}$ for the correction $\boldsymbol{\Delta} = [\mathbf{J}(\mathbf{p}^s)]^{-1}[-\mathbf{f}(\mathbf{p}^s)]$, improve the estimate to $\mathbf{p}^{s+1} = \mathbf{p}^s + \boldsymbol{\Delta}$, let $s \leftarrow s+1$, and repeat the process until the estimate stops changing. The vector $\mathbf{f}(\mathbf{p}^s)$ contains the values of the nonlinear constraint functions at the current estimate of the solution and the Jacobian matrix $\mathbf{J}(\mathbf{p}^s)$ has rows that are the transposes of the constraint gradients there, as shown below.

$$\mathbf{f}(\mathbf{p}^s) = \begin{bmatrix} f_1(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}\mathbf{p}^s) \\ \vdots \\ f_m(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}\mathbf{p}^s) \end{bmatrix} \qquad \mathbf{J}(\mathbf{p}^s) = \begin{bmatrix} \nabla_{\mathbf{p}} f_1(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}\mathbf{p}^s)^\top \\ \vdots \\ \nabla_{\mathbf{p}} f_m(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}\mathbf{p}^s)^\top \end{bmatrix}$$

To determine $\mathbf{x}^1$ in our example, we need to find p to make $f_1(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}p) = 0$, so for the first iteration of the GRG algorithm

$$\begin{aligned} \mathbf{f}(\mathbf{p}^s) &= f_1(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}p^s) \\ &= f_1\left(\begin{bmatrix} \frac{113}{13} \\ \frac{45}{13} \end{bmatrix} + \mathbf{R}p^s\right) \\ &= \tfrac{1}{20}\left(\tfrac{113}{13} + R_1 p^s\right)^2 + \left(\tfrac{45}{13} + R_2 p^s\right) - 5 \end{aligned}$$

$$\begin{aligned} \mathbf{J}(\mathbf{p}^s) &= \nabla_{\mathbf{p}} f_1(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}p^s)^\top \\ &= \frac{d}{dp^s}\left(\tfrac{1}{20}\left[\tfrac{113}{13} + R_1 p^s\right]^2 + \left[\tfrac{45}{13} + R_2 p^s\right] - 5\right) \\ &= \tfrac{1}{10}\left[\tfrac{113}{13} + R_1 p^s\right] R_1 + R_2. \end{aligned}$$

Then at step s of Newton's method for systems

$$\boldsymbol{\Delta} = \frac{-\frac{1}{20}\left(\frac{113}{13} + R_1 p^s\right)^2 - \left(\frac{45}{13} + R_2 p^s\right) + 5}{\frac{1}{10}\left[\frac{113}{13} + R_1 p^s\right] R_1 + R_2}.$$

In the Octave session below, I used this formula to find $\mathbf{x}^1$ for our example.

```
octave:2> A=[1/5,1];
octave:3> R=orth(A');
octave:4> p=0;
octave:5> for s=1:4
> delta=(-(1/20)*(113/13+R(1)*p)^2-(45/13+R(2)*p)+5)/((1/10)*(113/13+R(1)*p)*R(1)+R(2))
> p=p+delta;
> end
delta = -1.9455
delta = -0.0063649
delta = -6.8127e-08
delta = 0
octave:6> x1=[113/13;45/13]+R*p
x1 =

   8.3095
   1.5476

octave:7> f1=(1/20)*x1(1)^2+x1(2)-5
f1 = 0
```


In computing $\mathbf{J}(\mathbf{p})$ above we found $\nabla_{\mathbf{p}} f_1(\mathbf{x})$ by hand, but the gradient routine that we write in defining a nonlinear program computes only $\nabla_{\mathbf{x}} f_i(\mathbf{x})$. Here are MATLAB routines that compute the values and derivatives of the functions for problem `grg2`.

```
function f=grg2(x,i)
  switch(i)
    case 0
      f=(x(1)-8)^2+x(2)^2;
    case 1
      f=(1/20)*x(1)^2+x(2)-5;
    end
end
```

```
function g=grg2g(x,i)
  switch(i)
    case 0
      g=[2*(x(1)-8);2*x(2)];
    case 1
      g=[(1/10)*x(1);1];
    end
end
```

```
function H=grg2h(x,i)
  switch(i)
    case 0
      H=[2,0;0,2];
    case 1
      H=[(1/10),0;0,0];
    end
end
```

Any vector in $\mathbb{R}^n$ can be decomposed into a component in the nullspace of $\mathbf{A}_{m\times n}$ and a component in the range space of $\mathbf{A}^{\top}_{n\times m}$. We can find those components of $\nabla_{\mathbf{x}} f_i(\mathbf{x})$ like this.

$$\begin{bmatrix} \nabla_{\mathbf{t}} f_i(\mathbf{x}) \\ \nabla_{\mathbf{p}} f_i(\mathbf{x}) \end{bmatrix} = \mathbf{B}^{-1} \begin{bmatrix} \nabla_{\mathbf{x}} f_i(\mathbf{x}) \end{bmatrix} = \mathbf{B}^{\top} \begin{bmatrix} \nabla_{\mathbf{x}} f_i(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \mathbf{Z}^{\top} \\ \cdots\cdots \\ \mathbf{R}^{\top} \end{bmatrix} \begin{bmatrix} \nabla_{\mathbf{x}} f_i(\mathbf{x}) \end{bmatrix}$$

$$\begin{aligned} \nabla_{\mathbf{t}} f_i(\mathbf{x}) &= \mathbf{Z}^{\top} \nabla_{\mathbf{x}} f_i(\mathbf{x}) \\ \nabla_{\mathbf{p}} f_i(\mathbf{x}) &= \mathbf{R}^{\top} \nabla_{\mathbf{x}} f_i(\mathbf{x}) \end{aligned}$$

With the bottom formula we can calculate from $\nabla_{\mathbf{x}} f_1(\mathbf{x})$ the gradient with respect to $\mathbf{p}$ that we found earlier in computing the Jacobian by hand.

$$\begin{aligned} f_1(\mathbf{x}) &= \tfrac{1}{20} x_1^2 + x_2 - 5 \\ \nabla_{\mathbf{x}} f_1(\mathbf{x}) &= \begin{bmatrix} \frac{1}{10} x_1 \\ 1 \end{bmatrix} \\ \nabla_{\mathbf{x}} f_1\left(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}\mathbf{p}^s\right) &= \begin{bmatrix} \frac{1}{10}\left(x_1^{\mathrm{SD}} + R_1 \mathbf{p}^s\right) \\ 1 \end{bmatrix} \\ \nabla_{\mathbf{p}} f_1\left(\mathbf{x}^{\mathrm{SD}} + \mathbf{R}\mathbf{p}^s\right) &= \mathbf{R}^{\top} \begin{bmatrix} \frac{1}{10}\left(x_1^{\mathrm{SD}} + R_1 \mathbf{p}^s\right) \\ 1 \end{bmatrix} = R_1 \tfrac{1}{10}\left(\tfrac{113}{13} + R_1 \mathbf{p}^s\right) + R_2(1) \ \checkmark \end{aligned}$$

Using the ideas discussed above, I implemented the GRG algorithm in the MATLAB routine `grg.m` that is listed on the next page. The routine performs up to `kmax` descent iterations $\boxed{\texttt{7-38}}$, each of which begins by $\boxed{\texttt{8-11}}$ linearizing the constraints and $\boxed{\texttt{12-13}}$ finding bases for the corresponding nullspace and range space. The second stanza of the descent loop finds $\boxed{\texttt{15}}$ the gradient of the objective at the current point and $\boxed{\texttt{16}}$ its nullspace component. If the reduced gradient is small enough $\boxed{\texttt{17-21}}$ the current point is returned as `xstar` along with `rc=0` to signal convergence. Otherwise the reduced Hessian $\boxed{\texttt{22}}$ is used $\boxed{\texttt{23}}$ to compute the length of a full reduced steepest-descent step, and the resulting point `xsd` is $\boxed{\texttt{24}}$ found. Then $\boxed{\texttt{27-36}}$ Newton's method for systems of equations is used to restore feasibility. At each of up to $\boxed{\texttt{27}}$ 20 trial points `xtry` $\boxed{\texttt{28}}$ the function value vector $\boxed{\texttt{30}}$ and Jacobian $\boxed{\texttt{31}}$ are calculated, the correction vector `delta` is found by solving $\mathbf{J}\boldsymbol{\Delta} = -\mathbf{F}$ $\boxed{\texttt{33}}$, and the current estimate of the


```
 1 function [xstar,k,rc]=grg(fcn,grd,hsn,n,m,xzero,kmax,epz)
 2 % minimize f(x) subject to F(x)=0.
 3
 4   F=zeros(m,1);                    % declare sizes
 5   A=zeros(m,n); J=zeros(m,n-m);    % of built-up arrays
 6   xk=xzero;                        % feasible starting point
 7   for k=1:kmax                     % do up to kmax iterations
 8       for i=1:m                    % for each constraint
 9           g=grd(xk,i);             % find its gradient
10           A(i,:)=g';               % construct its linear approximation
11       end                          % constraint linearization ready
12       Z=null(A);                   % get a basis for the nullspace
13       R=orth(A');                  % get a basis for the range space
14
15       g=grd(xk,0);                 % objective gradient
16       rg=Z'*g;                     % reduced gradient
17       if(norm(rg) <= epz)          % converged?
18          xstar=xk;                 % yes; save optimal point
19          rc=0;                     % report success
20          return                    % and return
21       end                          % done with convergence test
22       rH=Z'*hsn(xk,0)*Z;           % reduced Hessian
23       astar=(rg'*rg)/(rg'*rH*rg); % length of full steepest descent
24       xsd=xk-Z*(astar*rg);         % take the step in R^n
25
26       p=zeros(m,1);                % initialize correction step
27       for s=1:20                   % Newton's method for systems
28           xtry=xsd+R*p;                 % trial point
29           for i=1:m                     % for each constraint
30               F(i)=fcn(xtry,i);         % get function value
31               J(i,:)=(R'*grd(xtry,i))'; % get del p value
32           end                           % F and J updated for p
33           delta=J\(-F);                 % correction
34           p=p+delta;                    % update guess at p
35           if(norm(delta) <= epz) break; end  % close enough?
36       end                          % Newton's method done
37       xk=xsd+R*p;                  % restore feasibility
38   end                              % reduced gradient step done
39   xstar=xk;                        % save the current point
40   rc=1;                            % report out of iterations
41
42 end
```

range-space coefficient vector **p** is [34] updated. If the correction vector is short enough [35] the Newton's method loop is interrupted, and [37] the current iterate is updated. If `kmax` iterations are consumed without satisfying the convergence criterion [17] the routine returns [39] the current point as `xstar` along with `rc=1` to signal nonconvergence.

To test `grg.m` I used it to solve the `grg2` problem and the following problem from [3, p311-315] (see §28.7.37), which I will call `grg4`.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2 + x_3^2 + x_4 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2 + x_2 + 4x_3 + 4x_4 - 4 = 0 \\ & f_2(\mathbf{x}) & = & -x_1 + x_2 + 2x_3 - 2x_4^2 + 2 = 0 \end{array}$$

The Octave session on the next page shows the results, in which each coordinate is correct through its last digit.


```
octave:1> format long
octave:2> [xstar,k,rc]=grg(@grg2,@grg2g,@grg2h,2,1,[2;24/5],100,1e-14)
xstar =

   8.91488339968883
   1.02624269849762

k =  14
rc = 0
octave:3> [xstar,k,rc]=grg(@grg4,@grg4g,@grg4h,4,2,[0;-8;3;0],100,1e-16)
xstar =

  -0.500000000000000
  -4.824791814486018
   1.534057450405037
   0.609640503216468

k =  71
rc = 0
```

23.2 Sequential Quadratic Programming

Consider the following equality-constrained nonlinear program, which I will call `sqp1` (see §28.7.38).

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & e^{x_1-1}+e^{x_2+1} \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1^2+x_2^2-1=0 \end{array}$$

The problem is strictly convex, so we can solve it by finding the unique point satisfying its Lagrange conditions.

$$\mathcal{L} = e^{x_1-1}+e^{x_2+1}+\lambda(x_1^2+x_2^2-1)$$

$$\begin{array}{rcccl} \dfrac{\partial\mathcal{L}}{\partial x_1} & = & e^{x_1-1}+2\lambda x_1 & = & 0 \\ \dfrac{\partial\mathcal{L}}{\partial x_2} & = & e^{x_2+1}+2\lambda x_2 & = & 0 \\ \dfrac{\partial\mathcal{L}}{\partial \lambda} & = & x_1^2+x_2^2-1 & = & 0 \end{array}$$

This system of nonlinear algebraic equations is analytically intractable but we can approximate its solution numerically by using Newton's method for systems, in which

$$\mathbf{f}(\mathbf{x},\lambda)=\begin{bmatrix} e^{x_1-1}+2\lambda x_1 \\ e^{x_2+1}+2\lambda x_2 \\ x_1^2+x_2^2-1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{J}(\mathbf{x},\lambda)=\begin{bmatrix} e^{x_1-1}+2\lambda & 0 & 2x_1 \\ 0 & e^{x_2+1}+2\lambda & 2x_2 \\ 2x_1 & 2x_2 & 0 \end{bmatrix}.$$

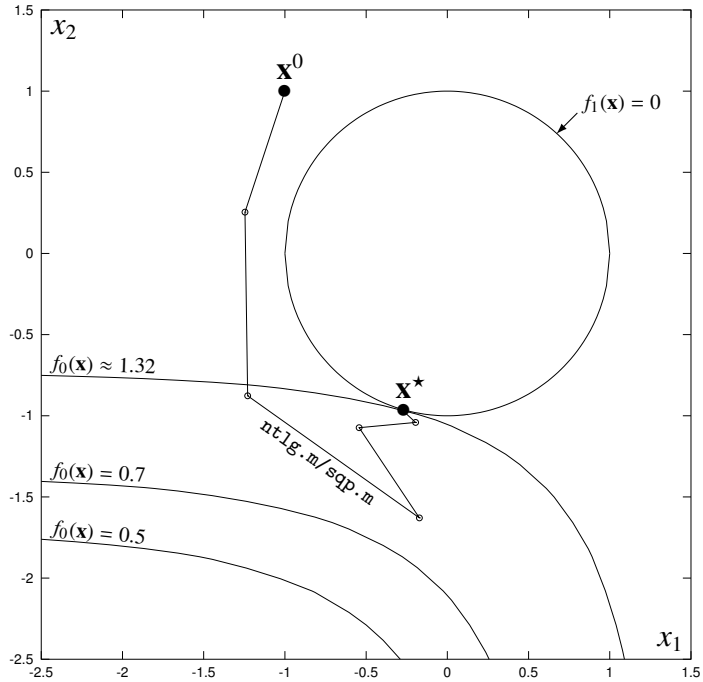
The MATLAB program on the next page implements Newton's method for systems using these formulas, and plots the resulting iterates over a contour diagram to show the convergence trajectory of the algorithm.


```
 1 % sqp1plot.m: graphical solution of sqp1
 2 clear; clf; set(gca,'FontSize',15)
 3 format long
 4
 5 xzero=[-1;1];                        % starting point
 6 xk(1)=xzero(1);                      % save coordinates
 7 yk(1)=xzero(2);                      % for plotting
 8 x=xzero;                             % start solution there
 9 lambda=1;                            % guess starting lambda
10 for k=1:10                           % Newton's method for systems
11     f=[exp(x(1)-1)+2*lambda*x(1);    % update function vector
12        exp(x(2)+1)+2*lambda*x(2);
13        x(1)^2+x(2)^2-1];
14     J=[exp(x(1)-1)+2*lambda,0,2*x(1); % update Jacobian
15        0,exp(x(2)+1)+2*lambda,2*x(2);
16        2*x(1),2*x(2),0];
17     delta=J\(-f);                    % find correction
18     x=x+delta(1:2);                  % update x part of solution
19     xk(k+1)=x(1);                    % save coordinates
20     yk(k+1)=x(2);                    % for ploting
21     lambda=lambda+delta(3);          % update lambda of solution
22 end                                  % of Newton's method
23 xstar=x                              % report optimal point
24 lambda                               % report optimal lambda
25
26 xl=[-2.5;-2.5];                      % lower limits for plot
27 xh=[1.5;1.5];                        % upper limits for plot
28 ng=20;                               % grid points for contouring
29 [xc,yc,zc]=gridcntr(@sqp1c,xl,xh,ng); % function values on grid
30
31 hold on                                % start graph
32 axis([xl(1),xh(1),xl(2),xh(2)],'equal') % set axes
33 v=[0.5,0.7,sqp1c(xstar)];              % contour levels
34 contour(xc,yc,zc,v)                  % contours of objective
35 for p=1:101                          % find points
36     x(p)=-1+2*0.01*(p-1);            % on zero contour
37     yp(p)=+sqrt(1-x(p)^2);           % of the
38     ym(p)=-sqrt(1-x(p)^2);           % constraint
39 end
40 plot(x,yp)                           % plot zero contour
41 plot(x,ym)                           % of the constraint
42 plot(xk,yk)                          % plot convergence trajectory
43 plot(xk,yk,'o')                      % mark the iterates
44 hold off                             % done with plot
45 print -deps -solid sqp1.eps          % print it
```

The loop 10-22 over `k` performs the iterations of Newton's method for systems and 19-20 saves the coordinates of each iterate `x` for 42-43 plotting. The remaining calculations are typical of those we have used in the past to study the behavior of other algorithms. The `sqp1c.m` routine, which `gridcntr.m` uses to compute objective values, is listed here.

```
function f=sqp1c(x)
  f=exp(x(1)-1)+exp(x(2)+1);
end
```

When the program is run it produces the picture and printed output shown on the next page, which suggest that this **Newton-Lagrange method** [2, §5.4.2] [4, §15.5] might be a good way to solve problems like `sqp1`.




```
octave:1> sqp1plot
xstar =

  -0.263290964724888
  -0.964716470209894

lambda =  0.536900432125476
```

23.2.1 A Newton-Lagrange Algorithm

The general equality-constrained nonlinear program

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) \\ \text{subject to} & f_i(\mathbf{x}) = 0 \;\; i = 1\ldots m \end{array}$$

has the Lagrangian $\mathcal{L} = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x})$ and these optimality conditions.

$$\begin{aligned} \nabla_{\mathbf{x}}\mathcal{L} &= \begin{bmatrix} \dfrac{\partial f_0}{\partial x_1} & + & \lambda_1 \dfrac{\partial f_1}{\partial x_1} & + \cdots + & \lambda_m \dfrac{\partial f_m}{\partial x_1} \\ \vdots & & \vdots & & \vdots \\ \dfrac{\partial f_0}{\partial x_n} & + & \lambda_1 \dfrac{\partial f_1}{\partial x_n} & + \cdots + & \lambda_m \dfrac{\partial f_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_n \\ \mathbf{f}_{n+1} \\ \vdots \\ \mathbf{f}_{n+m} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\ \nabla_{\boldsymbol{\lambda}}\mathcal{L} &= \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix} \end{aligned}$$

Each boldface function represents the equation to its left; e.g., $\mathbf{f}_1 = \dfrac{\partial f_0}{\partial x_1} + \sum_{i=1}^{m} \lambda_i \dfrac{\partial f_i}{\partial x_1}$.

To solve $\mathbf{f} = \mathbf{0}$ using Newton's method for systems, we must find the function vector

$$\mathbf{f}(\mathbf{x}, \boldsymbol{\lambda}) = \begin{bmatrix} \nabla_{\mathbf{x}} f_0 + \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} f_i \\ f_1 \\ \vdots \\ f_m \end{bmatrix}$$

and the Jacobian matrix $\mathbf{J}$,

$$\mathbf{H}_{f_0} + \sum_{i=1}^{m} \lambda_i \mathbf{H}_{f_i} = \mathbf{H}_{\mathcal{L}} \qquad\qquad \nabla_{\mathbf{x}} f_m$$

$$\mathbf{J}(\mathbf{x}, \boldsymbol{\lambda}) = \begin{bmatrix} [\nabla_{\mathbf{x}} \mathbf{f}_1]^\top & [\nabla_{\boldsymbol{\lambda}} \mathbf{f}_1]^\top \\ \vdots & \vdots \\ [\nabla_{\mathbf{x}} \mathbf{f}_n]^\top & [\nabla_{\boldsymbol{\lambda}} \mathbf{f}_n]^\top \\ [\nabla_{\mathbf{x}} \mathbf{f}_{n+1}]^\top & [\nabla_{\boldsymbol{\lambda}} \mathbf{f}_{n+1}]^\top \\ \vdots & \vdots \\ [\nabla_{\mathbf{x}} \mathbf{f}_{n+m}]^\top & [\nabla_{\boldsymbol{\lambda}} \mathbf{f}_{n+m}]^\top \end{bmatrix} = \begin{bmatrix} \boxed{\begin{matrix} \frac{\partial \mathbf{f}_1}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial \mathbf{f}_n}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}_n}{\partial x_n} \end{matrix}} & \begin{matrix} \frac{\partial \mathbf{f}_1}{\partial \lambda_1} & \cdots & \boxed{\frac{\partial \mathbf{f}_1}{\partial \lambda_m}} \\ \vdots & & \vdots \\ \frac{\partial \mathbf{f}_n}{\partial \lambda_1} & \cdots & \boxed{\frac{\partial \mathbf{f}_n}{\partial \lambda_m}} \end{matrix} \\ \begin{matrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \boxed{\frac{\partial f_m}{\partial x_1} \quad \cdots \quad \frac{\partial f_m}{\partial x_n}} \end{matrix} & \boxed{\begin{matrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{matrix}} \end{bmatrix}.$$

$$[\nabla_{\mathbf{x}} f_m]^\top \qquad\qquad \mathbf{0}_{m \times m}$$

Computing the gradients indicated on the left yields the matrix on the right. It can be viewed as composed of submatrices, some of which I have boxed. Each submatrix can be calculated from gradients and Hessians of the f_i. The submatrix on the upper left has elements such as

$$\frac{\partial \mathbf{f}_1}{\partial x_1} = \frac{\partial}{\partial x_1}\left(\frac{\partial f_0}{\partial x_1} + \lambda_1 \frac{\partial f_1}{\partial x_1} + \cdots + \lambda_m \frac{\partial f_m}{\partial x_1}\right) = \frac{\partial^2 f_0}{\partial {x_1}^2} + \sum_{i=1}^{m} \lambda_i \frac{\partial^2 f_i}{\partial {x_1}^2}$$

which is the $(1, 1)$ element of $\mathbf{H}_{\mathcal{L}}$. The submatrix in the upper right has elements such as

$$\frac{\partial \mathbf{f}_1}{\partial \lambda_m} = \frac{\partial}{\partial \lambda_m}\left(\frac{\partial f_0}{\partial x_1} + \lambda_1 \frac{\partial f_1}{\partial x_1} + \cdots + \lambda_m \frac{\partial f_m}{\partial x_1}\right) = \frac{\partial f_m}{\partial x_1}$$

so it is actually the gradient of f_m with respect to $\mathbf{x}$. Using these formulas for $\mathbf{f}$ and $\mathbf{J}$, I wrote the `ntlg.m` routine listed on the next page.


```
 1 function [xstar,k,rc,lstar]=ntlg(fcn,grd,hsn,n,m,xzero,lzero,kmax,epz)
 2 % Newton-Lagrange algorithm for equality-constrained problems
 3
 4   x=xzero;                          % starting point
 5   lambda=lzero;                     % starting multipliers
 6   rc=1;                             % in case of no convergence
 7   for k=1:kmax                      % do Newton's method for systems
 8       f=zeros(n+m,1);                 % fill in function vector
 9       f(1:n)=grd(x,0);                % gradient of objective
10       for i=1:m                       % for each constraint
11           lamg=lambda(i)*grd(x,i);    % weighted constraint gradient
12           f(1:n)=f(1:n)+lamg;         % accumulate gradient of L
13           f(n+i)=fcn(x,i);            % fill in function value
14       end                             % done with f
15       J=zeros(n+m,n+m);               % fill in Jacobian matrix
16       J(1:n,1:n)=hsn(x,0);            % Hessian of objective
17       for i=1:m                       % for each constraint
18           lamH=lambda(i)*hsn(x,i);    % weighted constraint Hessian
19           J(1:n,1:n)=J(1:n,1:n)+lamH; % accumulate Hessian of L
20           J(1:n,n+i)=grd(x,i) ;       % fill in constraint gradient
21           J(n+i,1:n)=grd(x,i)';       % and its transpose
22       end                             % done with J
23       delta=J\(-f);                   % find correction
24       x=x+delta(1:n);                 % adjust x
25       lambda=lambda+delta(n+1:n+m);   % adjust lambda
26       if(norm(delta) <= epz)          % close enough?
27          rc=0;                        % signal success
28          break                        % and return
29       end                             % done testing convergence
30   end;                              % Lagrange conditions solved
31   xstar=x;                          % return current iterate
32   lstar=lambda;                     % and current multipliers
33
34 end
```

This routine does 7-30 up to `kmax` iterations of Newton's method for systems. Each iteration begins by constructing 8-14 $\mathbf{f}(\mathbf{x}^k,\boldsymbol{\lambda}^k)$ and 15-22 $\mathbf{J}(\mathbf{x}^k,\boldsymbol{\lambda}^k)$. The gradient of the Lagrangian 11-12 and the Hessian of the Lagrangian 18-19 are built up by adding in one constraint gradient or constraint Hessian at a time. Then the correction $\boldsymbol{\Delta}$ is found by 23 solving the equation $\mathbf{J}\boldsymbol{\Delta}+\mathbf{f}=\mathbf{0}$, and the current estimates of the solution point and Lagrange multipliers are 24-25 updated to

$$\begin{bmatrix} \mathbf{x}^{k+1} \\ \boldsymbol{\lambda}^{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^{k} \\ \boldsymbol{\lambda}^{k} \end{bmatrix} + \boldsymbol{\Delta}.$$

If the correction is small enough 26 the routine 27 sets `rc=0` and 28,31-32 returns the current point and multipliers as the answer. If `kmax` iterations are consumed without satisfying the convergence criterion the routine also returns 31-32 the current point and multipliers, along with `rc=1` 6 to show that convergence was not achieved.

Routines `sqp1.m`, `sqp1g.m`, and `sqp1h.m`, which compute the values, gradients, and Hessians for `sqp1`, are listed at the top of the next page. The Octave session below them shows that `ntlg.m` delivers the same answer we found earlier for that problem.


```
function f=sqp1(x,i)
  switch(i)
    case 0
      f=exp(x(1)-1)+exp(x(2)+1);
    case 1
      f=x(1)^2+x(2)^2-1;
    end
end
```

```
function g=sqp1g(x,i)
  switch(i)
    case 0
      g=[exp(x(1)-1);
         exp(x(2)+1)];
    case 1
      g=[2*x(1);2*x(2)];
    end
end
```

```
function H=sqp1h(x,i)
  switch(i)
    case 0
      H=[exp(x(1)-1),0;
         0,exp(x(2)+1)];
    case 1
      H=[2,0;0,2];
    end
end
```

```
octave:1> format long
octave:2> xzero=[-1;1];
octave:3> lzero=1;
octave:4> [xstar,k,rc,lstar]=ntlg(@sqp1,@sqp1g,@sqp1h,2,1,xzero,lzero,10,1e-14)
xstar =

  -0.263290964724888
  -0.964716470209894

k =  10
rc = 0
lstar =  0.536900432125476
```

23.2.2 Equality Constraints

In §23.2.1 we developed a Newton-Lagrange algorithm for solving the nonlinear program

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) \\ \text{subject to} & f_i(\mathbf{x}) = 0, \quad i = 1\ldots m. \end{array}$$

At each step k that algorithm solves the linear system $\mathbf{J}\boldsymbol{\Delta} + \mathbf{f} = \mathbf{0}$ or

$$\begin{bmatrix} \mathbf{H}_{\mathcal{L}} & \nabla f_1 & \cdots & \nabla f_m \\ \nabla f_1{}^\top & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \nabla f_m{}^\top & 0 & \cdots & 0 \end{bmatrix} \boldsymbol{\Delta} + \begin{bmatrix} \nabla f_0 + \sum_{i=1}^m \lambda_i \nabla f_i \\ f_1 \\ \vdots \\ f_m \end{bmatrix} = \mathbf{0}$$

for the correction vector $\boldsymbol{\Delta}$. It is an interesting coincidence that this system of algebraic equations is precisely the Lagrange conditions for the following quadratic program.

$$\begin{array}{ll} \underset{\mathbf{p}\in\mathbb{R}^n}{\text{minimize}} & q(\mathbf{p}) = \frac{1}{2}\mathbf{p}^\top[\mathbf{H}_{\mathcal{L}}(\mathbf{x}^k)]\mathbf{p} + \mathbf{p}^\top[\nabla\mathcal{L}(\mathbf{x}^k)] = \frac{1}{2}\mathbf{p}^\top\mathbf{Q}\mathbf{p} + \mathbf{p}^\top\mathbf{c} \\ \text{subject to} & \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \\ \vdots \\ \nabla f_m(\mathbf{x}^k)^\top \end{bmatrix} \mathbf{p} + \begin{bmatrix} f_1(\mathbf{x}^k) \\ \vdots \\ f_m(\mathbf{x}^k) \end{bmatrix} = \mathbf{A}\mathbf{p} - \mathbf{b} = \mathbf{0} \end{array}$$

To prove the claim we can write down the Lagrange conditions for this problem, bearing in mind that $\mathbf{Q}$, $\mathbf{c}$, $\mathbf{A}$, and $\mathbf{b}$ are constants evaluated at the $\mathbf{x}^k$ for which we are finding $\mathbf{p}$. The quadratic program above has this Lagrangian, in which the multipliers are called $\boldsymbol{\mu}$.

$$\mathcal{L}_{\text{qp}}(\mathbf{p}, \boldsymbol{\mu}) = \frac{1}{2}\mathbf{p}^\top\mathbf{Q}\mathbf{p} + \mathbf{c}^\top\mathbf{p} + \boldsymbol{\mu}^\top[\mathbf{A}\mathbf{p} - \mathbf{b}]$$

From it we find these optimality conditions.

$$
\begin{array}{rrcrcl}
\nabla_{\mathbf{p}}\mathcal{L}_{\text{qp}}= & \mathbf{Qp} & + & \mathbf{c}+\mathbf{A}^{\top}\boldsymbol{\mu} & = & \mathbf{0} \\
 & \mathbf{Qp} & + & \mathbf{A}^{\top}\boldsymbol{\mu} & = & -\mathbf{c} \\
 & [\mathbf{H}_{\mathcal{L}}(\mathbf{x}^k)]\mathbf{p} & + & [\nabla f_1(\mathbf{x}^k)\ \cdots\ \nabla f_m(\mathbf{x}^k)]\,\boldsymbol{\mu} & = & -[\nabla\mathcal{L}(\mathbf{x}^k)] \\
 & & 1 & [\mathbf{H}_{\mathcal{L}}(\mathbf{x}^k),\ \ \nabla f_1(\mathbf{x}^k)\ \cdots\ \nabla f_m(\mathbf{x}^k)]\begin{bmatrix}\mathbf{p}\\ \boldsymbol{\mu}\end{bmatrix} & = & -\left[\nabla f_0(\mathbf{x}^k)+\sum_{i=1}^{m}\lambda_i\nabla f_i(\mathbf{x}^k)\right] \\
\nabla_{\boldsymbol{\mu}}\mathcal{L}_{\text{qp}}= & & & \mathbf{Ap}-\ \mathbf{b} & = & \mathbf{0} \\
 & & & \mathbf{Ap}+0\boldsymbol{\mu} & = & \mathbf{b} \\
 & & 2 & \begin{bmatrix}\nabla f_1(\mathbf{x}^k)^{\top} & 0 & \cdots & 0\\ \vdots & \vdots & & \vdots\\ \nabla f_m(\mathbf{x}^k)^{\top} & 0 & \cdots & 0\end{bmatrix}\begin{bmatrix}\mathbf{p}\\ \boldsymbol{\mu}\end{bmatrix} & = & -\begin{bmatrix}f_1(\mathbf{x}^k)\\ \vdots\\ f_m(\mathbf{x}^k)\end{bmatrix}
\end{array}
$$

Combining the final version 1 of the first condition with the final version 2 of the second and letting

$$\boldsymbol{\Delta}=\begin{bmatrix}\mathbf{p}\\ \boldsymbol{\mu}\end{bmatrix}$$

yields $\mathbf{J}\boldsymbol{\Delta}=-\mathbf{f}$. This means that at each iteration of the Newton-Lagrange algorithm we could find the $\mathbf{p}$ part of $\boldsymbol{\Delta}$ by solving the quadratic program instead of using Newton's method for systems. If we solve the quadratic program by using its optimality conditions above we also get the $\boldsymbol{\mu}$ part of $\boldsymbol{\Delta}$, but that is just the same as solving the Lagrange conditions for the original problem so we are back to using Newton's method for systems. If instead we solve the quadratic program numerically, then it is necessary to compute $\boldsymbol{\mu}$ separately using the formula we derived in §22.2.3,

$$\boldsymbol{\mu}^k=-\mathbf{A}^{+}[\mathbf{Qx}^k+\mathbf{c}] \qquad \text{where} \qquad \mathbf{A}^{+}=[\mathbf{AA}^{\top}]^{-1}\mathbf{A}.$$

If the original nonlinear program is convex like `sqp1`, then in finding $\boldsymbol{\Delta}^k$ it does not matter whether we use Newton's method for systems or solve the quadratic subproblem numerically for $\mathbf{p}^k$ and then find $\boldsymbol{\mu}^k$. However, if the problem is *nonconvex* then blindly solving the Lagrange conditions might yield a stationary point that is not even a local minimum (see §15.3). It is also possible that $\mathbf{J}(\mathbf{x}^k)$ will be singular at some iterate, in which case the Newton-Lagrange algorithm fails entirely. Both these humiliations might be avoided by using a quadratic program solver, which will actually try to minimize the Lagrangian of the original problem and which can modify the Hessian of the Lagrangian if necessary to keep it positive definite. This strategy leads to the simplest form of the **sequential quadratic programming** or **SQP** algorithm [5, §18], which I implemented in the `sqp.m` routine on the next page (not to be confused with Octave's built-in function of the same name, which we used in §8.3.1 and §8.7).


```
 1 function [xstar,k,rc,lstar]=sqp(fcn,grd,hsn,n,m,xzero,lzero,kmax,epz)
 2 % SQP algorithm for equality-constrained problems
 3
 4   x=xzero;                        % starting point
 5   lambda=lzero;                   % starting multipliers
 6   A=zeros(m,n); b=zeros(m,1);     % prepare A and b to be built up
 7   rc=1;                           % in case of no convergence
 8   for k=1:kmax                    % minimize the Lagrangian
 9       Q=hsn(x,0);                 % objective Hessian
10       c=grd(x,0);                 % objective gradient
11       for i=1:m                   % for each constraint
12           Q=Q+lambda(i)*hsn(x,i);   % find Lagrangian Hessian
13           g=grd(x,i);               % constraint gradient
14           c=c+lambda(i)*g;          % find Lagrangian gradient
15           A(i,:)=g';                % linearize constraint
16           b(i)=-fcn(x,i);           % linearize constraint
17       end                         % done preparing qp subproblem
18
19      [p,kq,rcq,nm]=qpeq(Q,c,A,b,50,1e-16); % solve the qp subproblem
20       if(rcq > 1)
21          rc=2;
22          break
23       end
24
25       x=x+p;                      % update x
26       [U,rch,nm]=hfact(A*A',1);   % factor
27       Vt=A'/U;                    % and solve
28       Aplus=(Vt/U')';             % to find the pseudoinverse
29       mu=-Aplus*(Q*p+c);          % find the change in lambda
30       lambda=lambda+mu;           % update lambda
31       if(norm(p) <= epz)          % close enough?
32          rc=0;                      % signal success
33          break                      % and return
34       end                           % done testing convergence
35   end;                            % Lagrange conditions solved
36   xstar=x;                        % return current iterate
37   lstar=lambda;                   % and current multipliers
38 end
```

Like `ntlg.m` this routine finds a point $(\mathbf{x}^\star,\boldsymbol{\lambda}^\star)$ that satisfies the Lagrange conditions of the original nonlinear program, but instead of using Newton's method for systems it solves a sequence of up to `kmax` quadratic subproblems for the corrections $\mathbf{p}$ to $\mathbf{x}$ and separately calculates the corresponding corrections $\boldsymbol{\mu}$ to $\boldsymbol{\lambda}$. Each iteration begins by finding the current values of $\mathbf{Q}$ [9,12], $\mathbf{c}$ [10,14], $\mathbf{A}$ [6,15], and $\mathbf{b}$ [6,16] defining the quadratic program. Then this routine [19] invokes the `qpeq.m` routine of §22.1.2 to solve the subproblem and [25] uses the result to find $\mathbf{x}^{k+1} = \mathbf{x}^k + \mathbf{p}$. To update the Lagrange multiplier estimates it [26-28] computes $\mathbf{A}^+$, [29] uses the formula we derived in §22.2.3, and [30] adjusts `lambda`. If the $\mathbf{x}$ adjustment $\mathbf{p}$ is short enough [31-34] it sets `rc=0` and returns early. If `kmax` iterations are consumed without achieving convergence, it [36-37] returns the current estimates `xstar` and `lstar` anyway, but with `rc=1` still set [7]. In the Octave session on the next page `sqp.m` finds exactly the same answer to `sqp1` that we found using `ntlg.m` in §23.2.1.

The Newton-Lagrange algorithm is not a feasible point method, as is clear from its convergence trajectory graph in §23.2.0, and because our SQP algorithm generates the same


```
octave:1> format long
octave:2> [xstar,k,rc,lstar]=sqp(@sqp1,@sqp1g,@sqp1h,2,1,[-1;1],1,10,1e-14)
xstar =

  -0.263290964724888
  -0.964716470209894

k =  10
rc =  0
lstar =  0.536900432125476
```

iterates for `sqp1` it is not a feasible point method either. However, in our implementation SQP does *make use* of a feasible point method, for solving the quadratic subproblems.

23.2.3 Inequality Constraints

In §23.2.2 we showed that solving each equality-constrained quadratic subproblem in the SQP algorithm is equivalent to doing one iteration of Newton's method for systems on the Lagrange conditions for the original nonlinear program, but it can also be interpreted in another way. *The subproblem minimizes a quadratic approximation to the Lagrangian of the original problem, subject to a linear approximation of the original problem's constraints.* This suggests that if the original problem has inequality constraints we might use exactly the same strategy, solving the resulting inequality-constrained quadratic subproblems with an active-set algorithm such as the one we implemented in the `qpin.m` routine of §22.2.4. This is referred to as the **IQP approach** [5, p530] to sequential quadratic programming. I implemented this idea in the `iqp.m` routine listed on the next page.

The caller supplies [1] a starting point `xzero`, which is used [6-11] to guess starting Lagrange multipliers; μ_i^0 is set to 0 if constraint i is satisfied or to 1 if the inequality is violated. The routine does up to `kmax` optimization iterations [15-37]. Each iteration begins [16-24] with the construction of the subproblem, whose objective [16-21] is a quadratic approximation to the Lagrangian and whose constraints [22-23] are a linear approximation to the original constraints. Then [26] the `qpin.m` routine of §22.2.4 is invoked to solve the quadratic program and the step `p` that it returns is used [32] to update the current estimate `xk` of the optimal point. The Lagrange multipliers `mu` are updated to those returned by `qpin.m` (as in [5, Algorithm 18.1]). If [33] the step was short enough an early exit [35] is taken with [34] `rc=0`, but if `kmax` iterations are consumed without satisfying the convergence criterion [38] the current point is returned in `xstar` with [14] `rc=1`.

The final stanza [41-54] is needed because the multipliers $\boldsymbol{\mu}$ returned by `qpin.m`, while correct for the quadratic program, are *not* the same as the multipliers $\boldsymbol{\lambda}$ for the original problem. According to the Lagrange conditions a solution $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ to the original problem satisfies

$$\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x},\boldsymbol{\lambda}) = \nabla_{\mathbf{x}} f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} f_i(\mathbf{x}) = \nabla_{\mathbf{x}} f_0(\mathbf{x}) + \bar{\mathbf{A}}^{\mathsf{T}}\bar{\boldsymbol{\lambda}} = \mathbf{0}.$$

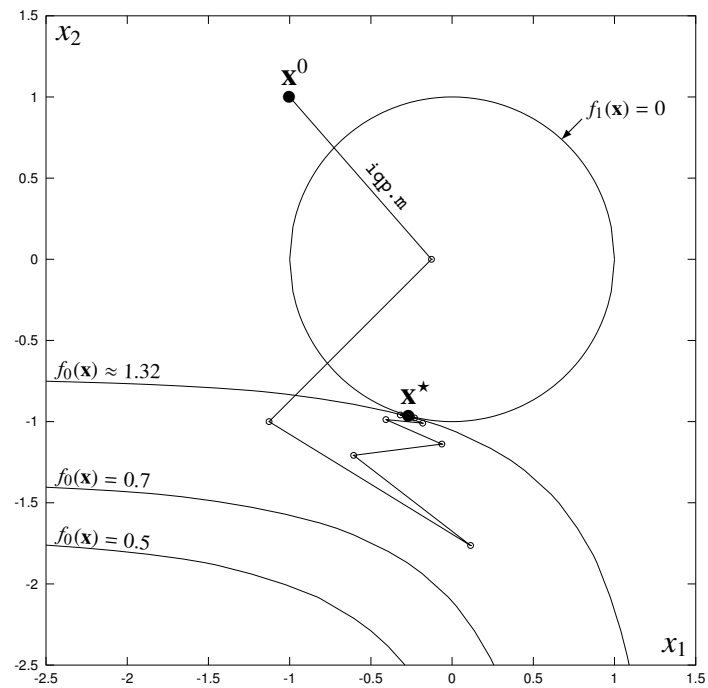
where $\bar{\mathbf{A}}$ is the matrix whose rows are the transposes of the gradients of the active constraints.


```
function [xstar,k,rc,lambda,mustar]=iqp(fcn,grd,hsn,m,xzero,kmax,epz)
% SQP algorithm for inequality-constrained problems

  n=size(xzero,1);                % variables
  xk=xzero;                       % starting point
  for i=1:m                       % consider each constraint
      mu(i)=0;                    % assume its multiplier is 0
      if(fcn(xk,i) > 0)           % but if xzero violates it
         mu(i)=1;                 % make its multiplier 1
      end
  end
  A=zeros(m,n);                   % prepare A to be built up
  b=zeros(m,1);                   % prepare b to be built up
  rc=1;                           % anticipate nonconvergence
  for k=1:kmax                    % minimize the Lagrangian
      Q=hsn(xk,0);                % objective Hessian
      c=grd(xk,0);                % objective gradient
      for i=1:m                   % consider each constraint
          Q=Q+mu(i)*hsn(xk,i);    % find Lagrangian Hessian
          g=grd(xk,i);            % constraint gradient
          c=c+mu(i)*g;            % find Lagrangian gradient
          A(i,:)=g';              % linearize constraint
          b(i)=-fcn(xk,i);        % linearize constraint
      end                         % done preparing qp subproblem

    [p,kq,rcq,W,mu]=qpin(Q,c,A,b,50,1e-14); % solve subproblem
     if(rcq > 1)
        rc=rcq;
        return
     end

     xk=xk+p;                     % update xk
     if(norm(p) <= epz)           % close enough?
        rc=0;                     % signal success
        break                     % and return
     end                          % done testing convergence
  end;                            % Lagrange conditions solved
  xstar=xk;                       % return current iterate
  mustar=mu;                      % return current QP multipliers

% find multipliers corresponding to the original problem
  Abar=zeros(0,n);
  mbar=0;
  for i=1:m
      if(W(i) == 1)
         mbar=mbar+1;
         Abar(mbar,:)=grd(xk,i)';
      end
  end
  lambda=zeros(m,1);
  if(mbar > 0)
     g=grd(xk,0);
     [lambda,rc]=getlgm(m,Abar,W,g);
  end

end
```

This requires $\bar{\boldsymbol{\lambda}}^\star = -(\bar{\mathbf{A}}\bar{\mathbf{A}}^\top)^{-1}\bar{\mathbf{A}}\,[\nabla_{\mathbf{x}} f_0(\mathbf{x}^\star)]$ but the multipliers returned by `qpin.m` for the problem of minimizing the Lagrangian are $\boldsymbol{\mu}^\star = -(\bar{\mathbf{A}}\bar{\mathbf{A}}^\top)^{-1}\bar{\mathbf{A}}\,(\mathbf{Q}\mathbf{x}^\star + \mathbf{c})$. Because $q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x}$

is an approximation to $\mathcal{L}(\mathbf{x},\boldsymbol{\lambda})$, its gradient $(\mathbf{Qx}+\mathbf{c})$ is usually different from $\nabla_{\mathbf{x}} f_0(\mathbf{x})$ even at $\mathbf{x}^\star$. The Octave session below shows the `sqp1` problem being solved by `iqp.m`, which treats the constraint as an inequality. The optimal point `xstar` and Lagrange multiplier `lstar` that it reports are the same as those we found before, but `mustar` ≠ `lstar` because 5>-6> $\nabla_{\mathbf{x}} q(\mathbf{x}^\star) \neq \nabla_{\mathbf{x}} f_0(\mathbf{x}^\star)$.

```
octave:1> format long
octave:2> [xstar,k,rc,lstar,mustar]=iqp(@sqp1,@sqp1g,@sqp1h,1,[-1;1],100,1e-15)
xstar =

  -0.263290964724888
  -0.964716470209894

k =  76
rc = 0
lstar =  0.536900432125476
mustar =  0.274477270192722
octave:3> Q=sqp1h(xstar,0)+mustar*sqp1h(xstar,1);
octave:4> c=sqp1g(xstar,0)+mustar*sqp1g(xstar,1);
octave:5> Q*xstar+c
ans =

  -0.0807856409522170
  -1.0226202924282342

octave:6> sqp1g(xstar,0)
ans =

   0.282722065471052
   1.035913379468513
```



Using a program like `sqp1plot.m` I plotted the algorithm's convergence trajectory on the problem, shown to the right. This is reminiscent of the jagged curve we observed for `ntlg.m` (and hence `sqp.m`).

I confirmed that `iqp.m` solves all of the inequality-constrained example problems we have considered so far. The function value and derivative routines for `arch4` are listed here.

```
function f=arch4(x,i)
  switch(i)
    case 0
      f=(x(1)-1)^2+(x(2)-1)^2;
    case 1
      f=4-(x(1)-2)^2-x(2);
    case 2
      f=13/8+(1/4)*x(1)-x(2);
  end
end
```

```
function g=arch4g(x,i)
  switch(i)
    case 0
      g=[2*(x(1)-1);2*(x(2)-1)];
    case 1
      g=[-2*(x(1)-2);-1];
    case 2
      g=[1/4;-1];
  end
end
```

```
function H=arch4h(x,i)
  switch(i)
    case 0
      H=[2,0;0,2];
    case 1
      H=[-2,0;0,0];
    case 2
      H=[0,0;0,0];
  end
end
```



```
octave:1> format long
octave:2> [xstar,k,rc,lstar]=iqp(@p2,@p2g,@p2h,1,[1;2],30,1e-16)
xstar =

   0.945582993415968
   0.894127197437503

k =  25
rc = 0
lstar =  3.37068560583615
octave:3> [xstar,k,rc,lstar]=iqp(@b1,@b1g,@b1h,2,[-2;2],10,1e-6)
xstar =

   4.44089209850062e-16
   1.00000000000000e+00

k =  3
rc = 0
lstar =

   1.000000000000000
   0.000000000000000

octave:4> [xstar,k,rc,lstar]=iqp(@moon,@moong,@moonh,2,[-2;2],10,1e-6)
xstar =

  -0.250000000000000
   0.968245836551858

k =  6
rc = 0
lstar =

   2.50000000000000
   1.50000000000000

octave:5> x2=sqrt(15/16)
x2 =  0.968245836551854
octave:6> [xstar,k,rc,lstar]=iqp(@arch4,@arch4g,@arch4h,2,[1;1],20,1e-6)
xstar =

   0.500000000000000
   1.750000000000000

k =  13
rc = 0
lstar =

   0.227272727272727
   1.272727272727273

octave:7> lambda1=5/22
lambda1 =  0.227272727272727
octave:8> lambda2=14/11
lambda2 =  1.27272727272727
```

This Octave session shows some representative results. In a few cases (e.g., `b1`) it was necessary to use a starting point other than the one given as part of the problem definition. The `moon` problem and the `arch4` problem (of §16.2) are both nonconvex.

23.2.4 A Quadratic Max Penalty Algorithm

The generalized reduced-gradient algorithm of §23.1.2 and the sequential quadratic programming algorithms of §23.2.2 and §23.2.3 all blithely linearize nonlinear constraints. If we do this at a point $\mathbf{x}^k$ that is feasible for the nonlinear constraints then, at least at that point, the resulting linear equations or inequalities will also be satisfied. If $\mathbf{x}^k$ is infeasible, however, the linearized constraints might not be satisfied anywhere. Consider the following problem, which I will call `incon` (see §28.7.39).

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1 - 1 \le 0 \\ & f_2(\mathbf{x}) & = & -x_1^2 + 4 \le 0 \end{array}$$

If $x_1 \le -2$ both inequalities are satisfied, so these constraints are not inconsistent. Now suppose that we linearize them about the infeasible point $\mathbf{x}^k = [1, 0]^\top$. Following the prescription in §23.1.2 we find

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \\ \nabla f_2(\mathbf{x}^k)^\top \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2x_1^k & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2 & 0 \end{bmatrix} \\ \mathbf{b} &= \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \mathbf{x}^k - f_1(\mathbf{x}^k) \\ \nabla f_2(\mathbf{x}^k)^\top \mathbf{x}^k - f_2(\mathbf{x}^k) \end{bmatrix} = \begin{bmatrix} 1 \times 1 - (1 - 1) \\ -2 \times 1 - (-[1^2] + 4) \end{bmatrix} = \begin{bmatrix} 1 \\ -5 \end{bmatrix} \end{aligned}$$

so the linearized constraints $\mathbf{Ax} \le \mathbf{b}$ require

$$\begin{array}{rcl} x_1 & \le & 1 \\ -2x_1 & \le & -5 \end{array} \qquad \text{or} \qquad \begin{array}{rcl} x_1 & \le & 1 \\ x_1 & \ge & 2\tfrac{1}{2} \end{array} \quad ⨳.$$

This happens only rarely, but it is lethal to the algorithms of this Chapter. If linearized equality constraints are inconsistent then $\mathbf{Ax} = \mathbf{b}$ has no nullspace and in `grg.m` the gradient calculation `rg=Z'*g` at line $\boxed{16}$ fails because `Z` is empty. In `sqp.m` and `iqp.m` inconsistent linearized constraints make the subproblem an infeasible quadratic program.

To experiment with `incon` I used the routines below to compute the values and derivatives of its functions.

```
function f=incon(x,i)
  switch(i)
    case 0
      f=x(1)^2+x(2)^2;
    case 1
      f= x(1)-1;
    case 2
      f=-x(1)^2+4;
  end
end
```

```
function g=incong(x,i)
  switch(i)
    case 0
      g=[2*x(1);2*x(2)];
    case 1
      g=[1;0];
    case 2
      g=[-2*x(1);0];
  end
end
```

```
function H=inconh(x,i)
  switch(i)
    case 0
      H=[2,0;0,2];
    case 1
      H=[0,0;0,0];
    case 2
      H=[-2,0;0,0];
  end
end
```

Here is what happens when `iqp.m` tries to solve the problem.


```
octave:1> [xstar,k,rc]=iqp(@incon,@incong,@inconh,2,[1;0],1,1e-6)
warning: feas: some elements in list of return values are undefined
warning: qpin: some elements in list of return values are undefined
warning: iqp: some elements in list of return values are undefined
xstar = [](0x0)
k =  1
rc =  4
octave:2> quit
```

The return code `rc=4` means the subproblem was infeasible; `feas.m` failed to find a starting point, so `qpin.m` had to resign before taking its first step and that 27-30 stopped `iqp.m`.

The threat of inconsistent constraints can be removed [5, p536] by reformulating the original nonlinear program as a penalty problem. In the case of inequality constraints this yields the optimization on the right.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_i(\mathbf{x}) \le 0, & i = 1\ldots m \end{array} \quad \longrightarrow \quad \begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n\ \mathbf{t}\in\mathbb{R}^m}{\text{minimize}} & \pi(\mathbf{x},\mathbf{t};\mu) & = & f_0(\mathbf{x}) + \mu\sum_{i=1}^m t_i \\ \text{subject to} & -t_i & \le & 0, \\ & f_i(\mathbf{x}) - t_i & \le & 0, \quad i = 1\ldots m \end{array}$$

If the original constraints are consistent, then solving a sequence of penalty problems with increasing values of μ drives $\mathbf{t}$ to zero and yields $\mathbf{x}^\star$ for the original problem. But the penalty problem is feasible even if the original constraints are *not* consistent, so it is also feasible if their linearizations are not consistent [5, p536]. This problem is sometimes referred to as the **elastic mode** formulation of the standard-form nonlinear program on the left above. We have encountered it twice before, in §8.7.4 as the soft-margin SVM model and in §20.1 as a reformulation of the nonsmooth max penalty problem on the left below.

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_0(\mathbf{x}) + \mu\sum_{i=1}^m \max[0, f_i(\mathbf{x})] \quad \longleftrightarrow \quad \begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n\ \mathbf{t}\in\mathbb{R}^m}{\text{minimize}} & \pi(\mathbf{x},\mathbf{t};\mu) & = & f_0(\mathbf{x}) + \mu\sum_{i=1}^m t_i \\ \text{subject to} & -t_i & \le & 0, \\ & f_i(\mathbf{x}) - t_i & \le & 0, \quad i = 1\ldots m \end{array}$$

We found that the max penalty problem is, because of its nondifferentiability, very hard for algorithms such as `ntfs.m`. To solve the smooth reformulation I proposed replacing its objective by a quadratic approximation to its Lagrangian and each constraint by its linear approximation, but of course this is just what `iqp.m` does. The **quadratic max penalty algorithm** uses `iqp.m` to solve a sequence of penalty problems in which μ gradually increases. To compute the values and derivatives of the objective and constraints in the smooth penalty problem from the values and derivatives of the functions f_i in the original problem, we can use interface routines similar to the `pye.m`, `pyeg.m`, and `pyeh.m` routines of §18.1.

To implement this idea I wrote the `emiqp.m` routine listed on the next page.


```
 1 function [xstar,k,rc,lstar,pn,tstar]=emiqp(name,mi,xzero,kmax,epz)
 2 % solve elastic mode penalty problem using iqp
 3
 4   global prob m pn              % share these with em.m, emg.m, emh.m
 5     prob=name;                  % character name of original problem
 6     m=mi;                       % constraints in original problem
 7     pn=1;                       % starting penalty multiplier
 8   n=size(xzero,1);              % variables in original problem
 9   yk=[xzero;zeros(mi,1)];       % starting [x;0]
10   fcn=str2func(prob);           % get function handle
11   for i=1:mi
12       yk(n+i)=max(0,fcn(xzero,i)); % initialize t(i) for feasibility
13   end
14
15   rc=1;
16   for k=1:kmax
17       [ystar,ki,rci,lambda]=iqp(@em,@emg,@emh,2*mi,yk,100,epz);
18       if(rci > 2)
19          rc=rci;
20          break
21       end
22
23       if(norm(ystar-yk) < epz)  % close enough?
24          rc=0;                  % signal success
25          if(rci == 2) rc=2; end % or that multipliers not found
26          break                  % and interrupt iterations
27       else
28          yk=ystar;              % start at current point
29          pn=2*pn;               % double the penalty multiplier
30       end
31   end
32   xstar=ystar(1:n);             % best x so far
33   tstar=ystar(n+1:n+m);         % best t so far
34   lstar=lambda(mi+1:2*mi);      % multipliers of original constraints
35
36 end
```

This routine receives [1] in `name` the character string name of the problem to be solved, and [4-7] passes it, the number of constraints `m`, and the penalty multiplier `pn`, as global parameters to the `em.m`, `emg.m`, and `emh.m` routines listed on the next page. Then, collecting the variables in one vector

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \mathbf{t} \end{bmatrix}, \qquad \text{it } \boxed{9\text{-}13} \text{ initializes} \qquad y_j^0 = \begin{cases} x_j^0 & \text{for} \quad j = 1 \ldots n \\ \max[0, f_{j-n}(\mathbf{x}^0)] & \text{for} \quad j = n+1 \ldots n+m. \end{cases}$$

This makes $t_i = 0$ if constraint i is satisfied at $\mathbf{x}^0$ or $t_i = f_i(\mathbf{x}^0)$ if it is violated, so that the constraints of the penalty problem are all satisfied at $\mathbf{y}^0$. Then the routine does up to `kmax` optimization iterations [16-31], each of which begins by invoking `iqp.m` [17] to solve the penalty problem at the current value of `pn` (initially [7] `pn=1`). If the step is short enough [23] the iterations are interrupted [24-26] and [32-34] the current solution is returned. Otherwise [27-28] the current point is taken as the starting point for the next iteration, the penalty multiplier is [29] doubled, and the iterations continue. If `kmax` iterations are consumed without satisfying the convergence criterion the routine returns [32-34] the current solution


```
 1 function f=em(y,i)
 2   global prob m pn
 3   fcn=str2func(prob);
 4   n=size(y,1)-m;
 5   x=y(1:n);
 6   t=y(n+1:n+m);
 7
 8
 9   if(i == 0)
10      f=fcn(x,0)+pn*t'*ones(m,1);
11   elseif(i <= m)
12      f=-t(i);
13   else
14      f=fcn(x,(i-m))-t(i-m);
15   end
16 end
17
18
```

```
function g=emg(y,i)
  global prob m pn
  grd=str2func([prob,'g']);
  n=size(y,1)-m;
  x=y(1:n);
  t=y(n+1:n+m);

  g=zeros(n+m,1);
  if(i == 0)
     g(1:n)=grd(x,0);
     g(n+1:n+m)=pn*ones(m,1);
  elseif(i <= m)
     g(n+i)=-1;
  else
     g(1:n)=grd(x,(i-m));
     g(n+(i-m))=-1;
  end
end
```

```
function H=emh(y,i)
  global prob m
  hsn=str2func([prob,'h']);
  n=size(y,1)-m;
  x=y(1:n);


  H=zeros(n+m,n+m);
  if(i == 0)
     H(1:n,1:n)=hsn(x,0);
  elseif(i > m)
     H(1:n,1:n)=hsn(x,(i-m));
  end
end
```

with [15] `rc=1`. Otherwise the return code is `0` if both $\mathbf{x}^\star$ and $\boldsymbol{\lambda}^\star$ were found [24] or `2` if only $\mathbf{x}^\star$ was found [25] or [18-19] the return code from `iqp.m` if `rci > 2`.

Each of the interface routines, listed above, begins by [3] getting a pointer to the function, gradient, or Hessian routine of the original problem, [4] deducing the number of variables `n` in the original problem, and [5-6] extracting from `y` the vectors `x` and if needed `t`. Then, based on the index `i` of the function in the penalty problem, it computes the value, gradient, or Hessian of the `i`'th penalty problem function for return.

To test `emiqp.m` I used it to solve problems `ep2`, `sqp1`, and `arch4`. The output on the next page shows the algorithm finding exact solutions to these problems at modest values of the penalty multiplier `pn`. The max penalty problem `ep2` that gave us so much trouble in §20.1 is easy for this algorithm. In `ep2` and `sqp1` the single constraint can't be inconsistent, so in each case $t^\star = 0$; in `arch4` there are 2 original constraints and they are also consistent, so $\mathbf{t}^\star = \mathbf{0}$.

What about the `incon` problem, for which the constraints linearized at $\mathbf{x}^0 = [1, 0]^\top$ are inconsistent? To find out I used `emiqp.m` to attempt a solution of that problem.

```
octave:1> [xstar,k,rc]=emiqp('incon',2,[1;0],10,1e-6)
xstar =

   1.0000e+00
  -4.9304e-32

k =  1
rc = 0
```

Unlike `iqp.m` this routine makes no complaint about an infeasible quadratic subproblem, so the elastic mode reformulation was successful. Unfortunately, `emiqp.m` makes no progress from the starting point, reporting immediately (`k=1`) and with bravado (`rc=0`) an answer that is not even feasible! Alas, in this problem the constraint $f_2(\mathbf{x}) = -x_1^2 + 4$ is nonconvex, and this leads to a nonconvex Lagrangian which `qpin.m` fails to correctly minimize on the flat of the linearized constraints. Trying `emiqp.m` on the other inequality constrained examples


```
octave:1> format long
octave:2> [xstar,k,rc,lstar,pn,tstar]=emiqp('ep2',1,[2;2],10,1e-6)
xstar =

   1.00000000000000
   1.00000000000000

k =  3
rc = 0
lstar =  2.00000000000000
pn =  4
tstar = 0
octave:3> [xstar,k,rc,lstar,pn,tstar]=emiqp('sqp1',1,[-1;1],10,1e-15)
xstar =

  -0.263290964724888
  -0.964716470209894

k =  2
rc = 0
lstar =  0.536900432125476
pn =  2
tstar = 0
octave:4> [xstar,k,rc,lstar,pn,tstar]=emiqp('arch4',2,[1;1],20,1e-6)
xstar =

   0.500000000000000
   1.750000000000000

k =  2
rc = 0
lstar =

   0.227272727272727
   1.272727272727273

pn =  2
tstar =

   0
   0
```

we have considered so far reveals it can solve only half of them. Some failures of `emiqp.m` result from the penalty objective getting harder to minimize as the penalty multiplier is increased (see §18.4) while others result from its use of `iqp.m` to solve the subproblems.

Our routines `sqp.m` and `iqp.m` work on the test problems that I tried, but they are less likely than naïve realizations of other algorithms to work for problems that are badly behaved. In sequential quadratic programming everything hinges on solving the subproblems. Because the quadratic programs are manufactured by the SQP or IQP algorithm they are likely to have various pathologies, so reliable performance demands that the subproblem solver be extremely robust. The `qpeq.m` and `qpin.m` routines of §22 meet the pedagogical needs of this introduction, but they are not sufficiently bulletproof to serve in production code. In addition to solving subproblems that are nonconvex, a practical implementation of the sequential quadratic programming idea must somehow deal with subproblems that are unbounded.

When `iqp.m` tries to solve `b1` from its catalog starting point $\mathbf{x}^0 = [\frac{1}{2}, \frac{1}{2}]^\top$, for example, it fails because a subproblem is unbounded.

Nonconvexity can be somewhat mitigated by using a line search rather than taking full steps [5, p534-535]. In deciding whether to accept a trial step or instead try a shorter one it is common practice to insist that $(\mathbf{x}^{k+1}, \boldsymbol{\lambda}^{k+1})$ be better than $(\mathbf{x}^k, \boldsymbol{\lambda}^k)$ in the sense that the move decreases a merit function [4, p576-580]; recall from §21.3.3 that this ensures each step reduces either the objective or the infeasibility or both. Merit functions have a theory of their own and introduce numerous further complications [5, §15.4].

Sequential quadratic programming uses Hessians of the constraints as well as of the objective, so unless n is small evaluating them requires a lot of arithmetic. Practical implementations therefore often use quasi-Newton approximations for either the Hessians of the individual functions or the Hessian of the Lagrangian [4, p576] [5, p536-540], and this can also make the algorithm more robust against nonconvexity.

Each quadratic program is supposed to approximate the Lagrangian and constraints of the original problem in the neighborhood of $\mathbf{x}^k$, so in solving the subproblem we might use a restricted-steplength algorithm (see §17.2) or trust-region approach (see §17.3) to ensure that $q(\mathbf{x})$ remains a good model of the original Lagrangian. If the subproblem is unbounded it will fail this test, and in that case the step taken in `sqp.m` or `iqp.m` might be shortened to produce a subproblem that is more useful.

23.3 Exercises

23.3.1 [E] How are the classical barrier method and the interior-point algorithm for nonlinear programming similar to each other? How are the classical penalty method and the augmented Lagrangian algorithm similar to each other? How do these two algorithm types differ from each other, and from the quadratic programming methods discussed in §22? What characterizes a *feasible-point method*? Are all of the algorithms described in this Chapter feasible point methods? Do they all *use* some feasible point method?

23.3.2 [E] How does a *reduced-gradient method* differ from a *reduced-Newton method*? How does `rsdeq.m` differ from `rneq.m`?

23.3.3 [P] Use `rsdeq.m` to solve the `qp1` problem. How many steepest-descent iterations are required to satisfy the convergence criterion `norm(rg)` $\leq 10^{-6}$?

23.3.4 [E] In `rsdeq.m`, the vector `tk` is the projection of the iterate `xk` onto the flat defined by the equality constraints. (a) Why is it necessary to project `xk` onto the flat? (b) Is `tkp` also in the flat? If so, what causes it to be in the flat?

23.3.5 [P] Continue the calculations illustrated in §23.1.1 to find the iterate $\mathbf{t}^3$ generated by `rsdeq.m` in solving problem `rnt`. What is `norm(rg)` at the corresponding $\mathbf{x}^3$?

23.3.6 [H] Show that a set of differentiable nonlinear constraints $f_i(\mathbf{x}) = 0,\ i = 1 \ldots m$ can be approximated near $\mathbf{x}^k$ by the linear constraints $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \\ \vdots \\ \nabla f_m(\mathbf{x}^k)^\top \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \mathbf{x}^k - f_1(\mathbf{x}^k) \\ \vdots \\ \nabla f_m(\mathbf{x}^k)^\top \mathbf{x}^k - f_m(\mathbf{x}^k) \end{bmatrix}.$$

23.3.7 [E] Describe in words the *generalized reduced-gradient algorithm.* What is the dimension of the flat in which the steepest-descent step is taken? Why is it necessary to restore feasibility after taking the steepest-descent step? In what direction does the algorithm move to make this correction?

23.3.8 [E] How are the *nullspace* of the $m \times n$ matrix $\mathbf{A}$ and the *range space* of $\mathbf{A}^\top$ related? What are their dimensions? What MATLAB command can be used to find a basis for each? Show that each vector in the nullspace of $\mathbf{A}$ is orthogonal to every vector in the range space of $\mathbf{A}^\top$.

23.3.9 [H] What makes two vector spaces *orthogonal complements* of each other? How can a vector $\mathbf{x}$ be decomposed into components lying in the nullspace of a matrix $\mathbf{A}$ and the range space of $\mathbf{A}^\top$? What is a *basis matrix,* how can it be constructed, and what makes it an orthogonal matrix? How can we find the inverse of a basis matrix?

23.3.10 [H] In its feasibility-restoration step, how does the GRG algorithm determine how far to move into the range space of $\mathbf{A}^\top$? Explain the formulas used in §23.1.2 for the function vector $\mathbf{f}(\mathbf{p}^s)$ and Jacobian matrix $\mathbf{J}(\mathbf{p}^s)$.

23.3.11 [E] Suppose $\mathbf{R}$ contains a basis for the range space of an $n \times m$ matrix $\mathbf{A}^\top$. (a) What are the dimensions of $\mathbf{R}$? (b) How can we compute the projection of a vector $\mathbf{v} \in \mathbb{R}^n$ onto the range space of $\mathbf{A}^\top$?

23.3.12 [H] Explain each multiplication by `Z`, `Z'`, `R`, or `R'` in `grg.m`. What do they do?

23.3.13 [P] In §23.1.2 we generalized the reduced-gradient algorithm for nonlinear constraints. In a similar way, generalize the reduced-Newton algorithm, as implemented in the `rneq.m` routine of §22.3, for nonlinear constraints. Compare the performance of your routine to that of `grg.m` when both are used to solve the `grg4` problem.

23.3.14 [H] The generalized reduced-gradient algorithm of §23.1.2 works for problems having equality constraints $f_i(\mathbf{x}) = 0$. Suppose we add slack variables $\mathbf{s}$ to rewrite the constraints of an inequality-constrained problem as equalities. If `grg.m` finds a solution $(\mathbf{x}^\star, \mathbf{s}^\star)$ to the reformulated problem in which coincidentally $\mathbf{s} \geq \mathbf{0}$, is $\mathbf{x}^\star$ optimal for the inequality-constrained problem?

23.3.15 [P] Can the generalized reduced-gradient idea be used for solving inequality-constrained problems by embedding it in an active-set algorithm such as the one we implemented in `qpin.m`? Consider the following approach. Starting from a feasible $\mathbf{x}^0$, examine the values of the constraint functions at $\mathbf{x}^k$ to determine which are tight and which are slack. Linearize

the tight constraints about that point and do one step of steepest descent in the flat defined by the active constraints. Then use Newton's method for systems to restore feasibility for the original constraints and produce $\mathbf{x}^{k+1}$. Write a MATLAB routine to implement this idea, and test it on the `sqp1` and `arch4` problems.

23.3.16 [H] Show that problem `sqp1` of §23.2.0 is strictly convex. Verify that its Lagrange conditions are satisfied at $\mathbf{x}^\star = [-0.263290964724888; -0.964716470209894]^\top$ with $\lambda^\star = 0.536900432125477$.

23.3.17 [H] In §23.2.1 I derived the Jacobian matrix that must be used in Newton's method for systems to solve the Lagrange conditions of a general equality-constrained nonlinear program. Explain in detail where the submatrices of this Jacobian come from.

23.3.18 [P] When the Newton-Lagrange algorithm implemented in `ntlg.m` is used to solve the `sqp1` problem from $(\mathbf{x}^0, \lambda^0) = ([0, 1]^\top, 0)$, it does not find $\mathbf{x}^\star$. Explain why.

23.3.19 [E] Explain how solving a certain quadratic program is equivalent to taking one step of Newton's method for systems in the Newton-Lagrange algorithm. Why, if the quadratic program is solved numerically, is it not *completely* equivalent?

23.3.20 [E] The sequential quadratic programming algorithm implemented in `sqp.m` is like the Newton-Lagrange algorithm implemented in `ntlg.m` except that it finds the $\mathbf{p}$ part of each correction step $\mathbf{\Delta}$ in Newton's method for systems by solving a quadratic program. Then it has to find the $\boldsymbol{\mu}$ part of $\mathbf{\Delta}$ separately. How is this an improvement over `ntlg.m`?

23.3.21 [P] Use `sqp.m` to solve (a) the `grg2` problem; (b) the `grg4` problem.

23.3.22 [P] Revise `sqp.m` to find `mu` by invoking the `getlgm.m` routine of §22.2.3 rather than using the open code of 26-29. Test for a nonzero return code from `getlgm.m` and if it fails return `rc=3` from `sqp.m`.

23.3.23 [E] In §23.2.0 we used Newton's method for systems of equations to solve the optimality conditions for an equality-constrained nonlinear program. In the interior point algorithm of §21.3 we used Newton's method for systems to solve the optimality conditions for an inequality-constrained nonlinear program. In what other ways are the resulting algorithms similar but (quite) different?

23.3.24 [E] Describe the IQP approach to sequential quadratic programming. How does `iqp.m` differ from `sqp.m`, and why? In `iqp.m`, why is `mustar` ≠ `lambda`?

23.3.25 [H] Suppose that some nonlinear program has m inequality constraints $f_i(\mathbf{x}) \le 0$ and that $\bar{\mathbf{A}}$ is a matrix whose rows are the transposes of the gradients of the $\bar{m} \le m$ constraints that are active at $\mathbf{x}^k$. (a) What are the dimensions of $\bar{\mathbf{A}}^\top$? (b) If $\bar{\boldsymbol{\lambda}}$ is a vector of the Lagrange multipliers corresponding to the active constraints, what is its length? (c) Show that

$$\sum_{k=1}^{m} \lambda_i \nabla f_i(\mathbf{x}^k) = \bar{\mathbf{A}}^\top \bar{\boldsymbol{\lambda}}.$$

23.3.26 [P] In §23.2.3 I explained why the Lagrange multipliers $\boldsymbol{\mu}$ returned by `qpin.m` to `iqp.m` are different from the Lagrange multipliers $\boldsymbol{\lambda}$ for the original nonlinear program, but the reason I gave is not the *only* reason. Study the calculation of the Lagrange multipliers in `qpin.m` and propose a reason why the multipliers it returns might be slightly wrong. Hint: when is `Abar` updated?

23.3.27 [P] In `iqp.m`, why is it necessary to initialize the Lagrange multipliers? How are they initialized? Conduct experiments to determine how the performance of the algorithm is affected by using $\boldsymbol{\mu}^0 = \mathbf{0}$ or $\boldsymbol{\mu}^0 = \mathbf{1}$ instead.

23.3.28 [P] In §24 we will study the following convex inequality-constrained nonlinear program, which was introduced in Exercise 21.4.30 as problem `ek1`.

$$\begin{array}{lrcll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1-20)^4+(x_2-12)^4 & \\ \text{subject to} & f_1(\mathbf{x}) & = & 8e^{(x_1-12)/9}-x_2+4 & \le\ 0 \\ & f_2(\mathbf{x}) & = & 6(x_1-12)^2+25x_2-600 & \le\ 0 \\ & f_3(\mathbf{x}) & = & -x_1+12 & \le\ 0 \end{array}$$

(a) Use `iqp.m` to solve `ek1` from its catalog starting point $\mathbf{x}^0 = [18, 21]^\top$. (b) At each iteration the algorithm computes $\mathbf{x}^{k+1} = \mathbf{x}^k + \mathbf{p}$, so $\mathbf{p} = \mathbf{x}^{k+1} - \mathbf{x}^k$. Show that its linear approximation of the constraints can be written as $\mathbf{Ap} \le \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} \nabla f_1(\mathbf{x}^k)^\top \\ \nabla f_2(\mathbf{x}^k)^\top \\ \nabla f_3(\mathbf{x}^k)^\top \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} -f_1(\mathbf{x}^k) \\ -f_2(\mathbf{x}^k) \\ -f_3(\mathbf{x}^k) \end{bmatrix}.$$

(c) This problem has $n = 2$, so each constraint is approximated by a straight line, and the equations of these straight lines are given by $\mathbf{Ax}^{k+1} = \mathbf{b} + \mathbf{Ax}^k$. Write a MATLAB program that draws the zero contours of the three constraints, finds $\mathbf{A}$ and $\mathbf{b}$ at $\mathbf{x}^0$, and plots the line approximating each constraint. These lines should form a polyhedral approximation to the feasible set. (d) Run `iqp.m` for one iteration and plot the point $\mathbf{x}^1$. With reference to the figure formed by the linear approximation of the constraints, explain why this point is produced by the first iteration of `iqp.m`.

23.3.29 [P] Use `iqp.m` solve the `p2` problem of §18.1.

23.3.30 [P] When `iqp.m` solves the `moon` problem of §16.3 from $\mathbf{x}^0 = [-2, 2]^\top$ it finds the optimal point $[-\frac{1}{4}, +\sqrt{15/16}]^\top$. Find a starting point from which `iqp.m` converges to the other optimal point $[-\frac{1}{4}, -\sqrt{15/16}]^\top$ instead.

23.3.31 [P] The `iqp.m` routine of §23.2.3 is capable of solving the `b1` problem of §19.0 if $\mathbf{x}^0 = [-2, 2]^\top$, but not if $\mathbf{x}^0 = [\frac{1}{2}, \frac{1}{2}]^\top$. Show that for this catalog starting point `iqp.m` fails because its first quadratic subproblem is unbounded.

23.3.32 [E] Show that if two nonlinear constraints $f_1(\mathbf{x}) \le 0$ and $f_2(\mathbf{x}) \le 0$ are linearized at a point that satisfies them both, then the linear approximations are consistent at that point.

23.3.33 [H] In the `incon` problem of §23.2.4, $f_2(\mathbf{x})$ is a nonconvex function and linearizing the constraints at a point where $x_1 = 1$ results in linear constraints that are inconsistent. If a set of constraints $f_i(\mathbf{x}) \le 0$ are all *convex* functions, is it possible for their linearizations to be inconsistent?

23.3.34 [E] Why precisely do `grg.m`, `sqp.m`, and `iqp.m` fail if at some iterate $\mathbf{x}^k$ the linearized constraints are inconsistent? Describe a reformulation of the standard-form nonlinear program that can be used to remove the threat of inconsistent constraint linearizations.

23.3.35 [H] In the elastic mode formulation, what is $\mathbf{t}^\star$ if the original constraints are (a) consistent; (b) inconsistent?

23.3.36 [H] The elastic mode formulation of a standard-form nonlinear program is feasible even if the original constraints are not. The following problem has inconsistent constraints and is therefore infeasible.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1 + x_2 \\ \text{subject to} & f_1(\mathbf{x}) & = & x_1 + x_2 \le 1 \\ & f_2(\mathbf{x}) & = & x_1 + x_2 \ge 2 \end{array}$$

(a) Write down the elastic mode formulation of this problem and show that it is feasible. (b) Use the KKT method and take a limit to solve the penalty problem analytically.

23.3.37 [E] How are the soft-margin SVM model and the max penalty problem related to the elastic mode formulation of a standard-form nonlinear program?

23.3.38 [E] In §23.2.4 we implemented the quadratic max penalty algorithm in `emiqp.m`. (a) Briefly describe the algorithm in words. (b) In `emiqp.m`, what does the vector `yk` represent? How is it initialized? (c) When `emiqp.m` invokes `iqp.m` it passes the function pointers `@em`, `@emg.m`, and `@emh.m`. What do these routines compute? (d) How do `em.m`, `emg.m`, and `emh.m` know the current value of the penalty multiplier? (e) List the possible return code values `rc` from `emiqp.m` and explain what each signifies.

23.3.39 [H] The interface routines `em.m`, `emg.m`, and `emh.m` compute the values and derivatives of the functions in the elastic mode penalty problem. (a) Derive formulas for these quantities in terms of the values and derivatives of the functions in the original nonlinear program. (b) Explain how the code in these routines evaluates your formulas to compute `f`, `g`, and `H`.

23.3.40 [P] Use `emiqp.m` to solve the `nset` problem of §16.10.

23.3.41 [P] The `iqp.m` and `emiqp.m` routines give different results for the `incon` problem. (a) Why does `iqp.m` stop with `rc=4` while `emiqp.m` returns `rc=0`? (b) Why does `emiqp.m` return $\mathbf{x}^\star = \mathbf{x}^0$? (c) Investigate in detail the failure of `iqp.m` and `qpin.m` to solve this problem.

23.3.42 [P] The convergence criterion I used in `iqp.m` is that [33] `norm(p) <= epz`, but the Lagrange multipliers `mu` returned by `qpin.m` are used [19,21] in constructing the quadratic

approximation so an argument can be made that convergence has not been achieved unless `mu` also stops changing. (`a`) Modify `iqp.m` to also enforce this requirement for convergence. (`b`) Using this version of `iqp.m`, try solving the `incon` problem with `emiqp.m`. Does it solve the problem now? Explain.

23.3.43 `[E]` State three possible reasons why `emiqp.m` might fail.

23.3.44 `[E]` How might `qpeq.m` and `qpin.m` (and hence `sqp.m` and `iqp.m`) be made more robust? Describe strategies to deal with (`a`) nonconvexity of the Lagrangian; (`b`) unbounded quadratic program subproblems. (`c`) How might the computational workload of the Hessian evaluations in a sequential quadratic programming implementation be reduced?

23.3.45 `[P]` If the quadratic subproblem that is constructed at iteration $\mathbf{x}^k$ of a sequential quadratic programming algorithm is unbounded, that suggests we have stepped too far. (`a`) Outline modifications to `iqp.m` and `qpin.m` that will detect this condition and shorten the step to try again. (`b`) Revise the code to implement your plan. (`c`) Test the new version of `iqp.m` by using it to solve problem `b1` from $\mathbf{x}^0 = [\frac{1}{2}, \frac{1}{2}]^\top$. Do your modifications effectively reject unbounded quadratic subproblems and thereby permit this problem to be solved?

23.3.46 `[H]` The quadratic max penalty algorithm proposed in §23.2.4 constructs each subproblem (inside `iqp.m`) by making a quadratic approximation to the *Lagrangian* of the penalty problem and a linear approximation to each of its constraints. A simpler algorithm constructs the quadratic subproblems by making a quadratic approximation to the *objective* of the penalty problem and a linear approximation to each of its constraints, and uses `qpin.m` directly to solve each subproblem. Unfortunately this approach often converges to a point that is not optimal. Why? Hint: if $m > 1$ that non-optimal point is typically an intersection of zero contours of the constraints.

23.3.47 `[H]` Several of the programs available on the NEOS web server (see §8.3.1) are based on the algorithms discussed in this Chapter [5, §18.8]. By searching the web, find out which of the programs are based on which of the algorithms.

24

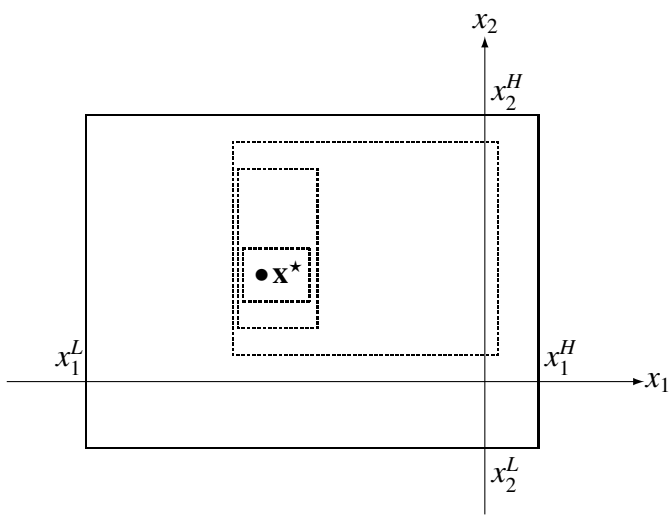
Ellipsoid Algorithms

The story of nonlinear programming has led us from pure random search, the most primitive and mindless numerical technique, to sequential quadratic programming, the most sophisticated and complex. To conclude our study of methods we now return almost to the beginning, with a simple approach whose haphazard meanderings, like those of pure random search, appear almost aimless. Ellipsoid algorithms are effective only for problems having no more than a few dozen variables, but they are robust and easy to use and have an elegant theoretical basis that makes them quite different from the other methods we have studied.

24.1 Space Confinement

In implementing the algorithms of §10–§23 I have often taken full descent steps for simplicity, so the role that variable bounds play in governing our search for $\mathbf{x}^\star$ has not always been obvious. But even if the bounds are not used explicitly in line searching they are implicitly present whenever we select a plausible starting point, and in practical applications they are essential for the other reasons outlined in §9.5.

If the bounds for a problem have been properly chosen, we can be sure that $\mathbf{x}^\star \in [\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ as illustrated below.



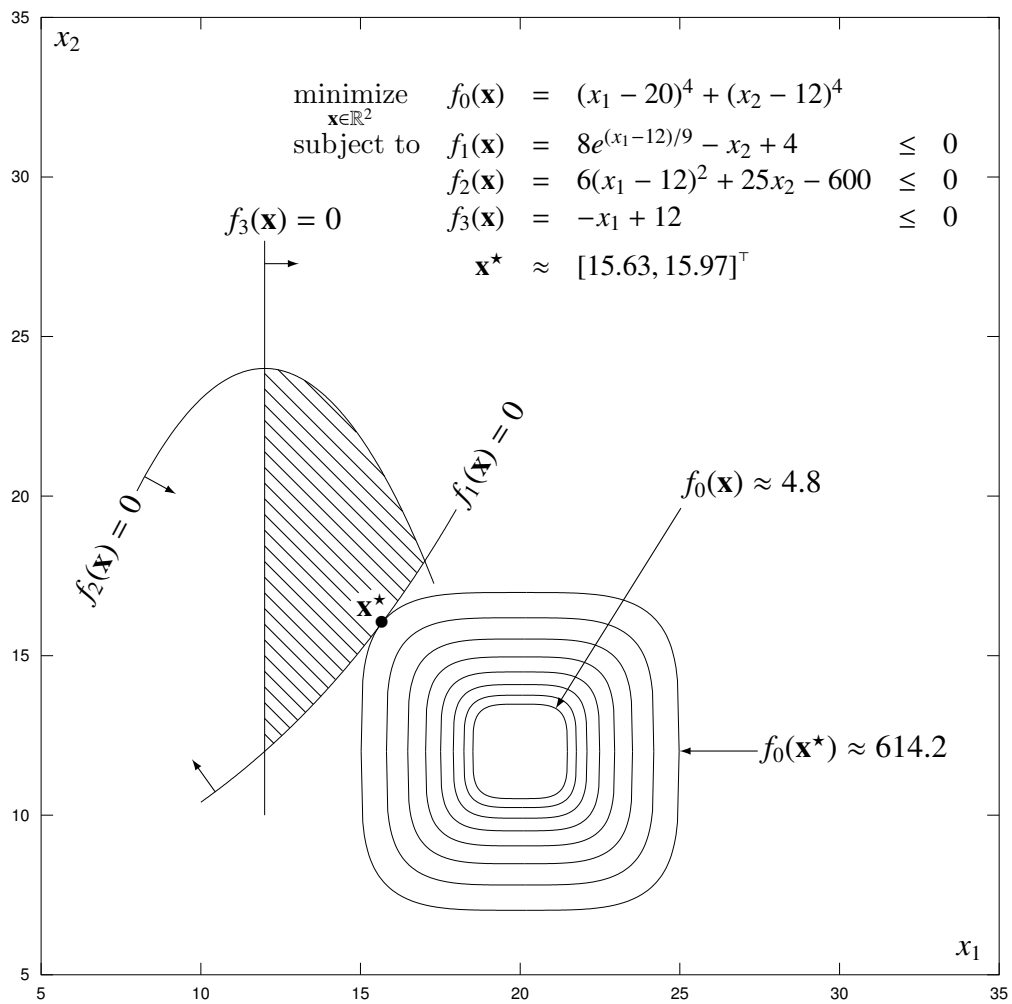
Suppose that it were possible, by performing some calculations involving the bounds and the functions $f_i(\mathbf{x})$ that define the problem, to construct a smaller box that also encloses $\mathbf{x}^\star$. If by repeating the process we could produce a sequence of progressively smaller boxes each containing $\mathbf{x}^\star$, such as those drawn dashed in the figure, then in the limit we would know the point exactly.

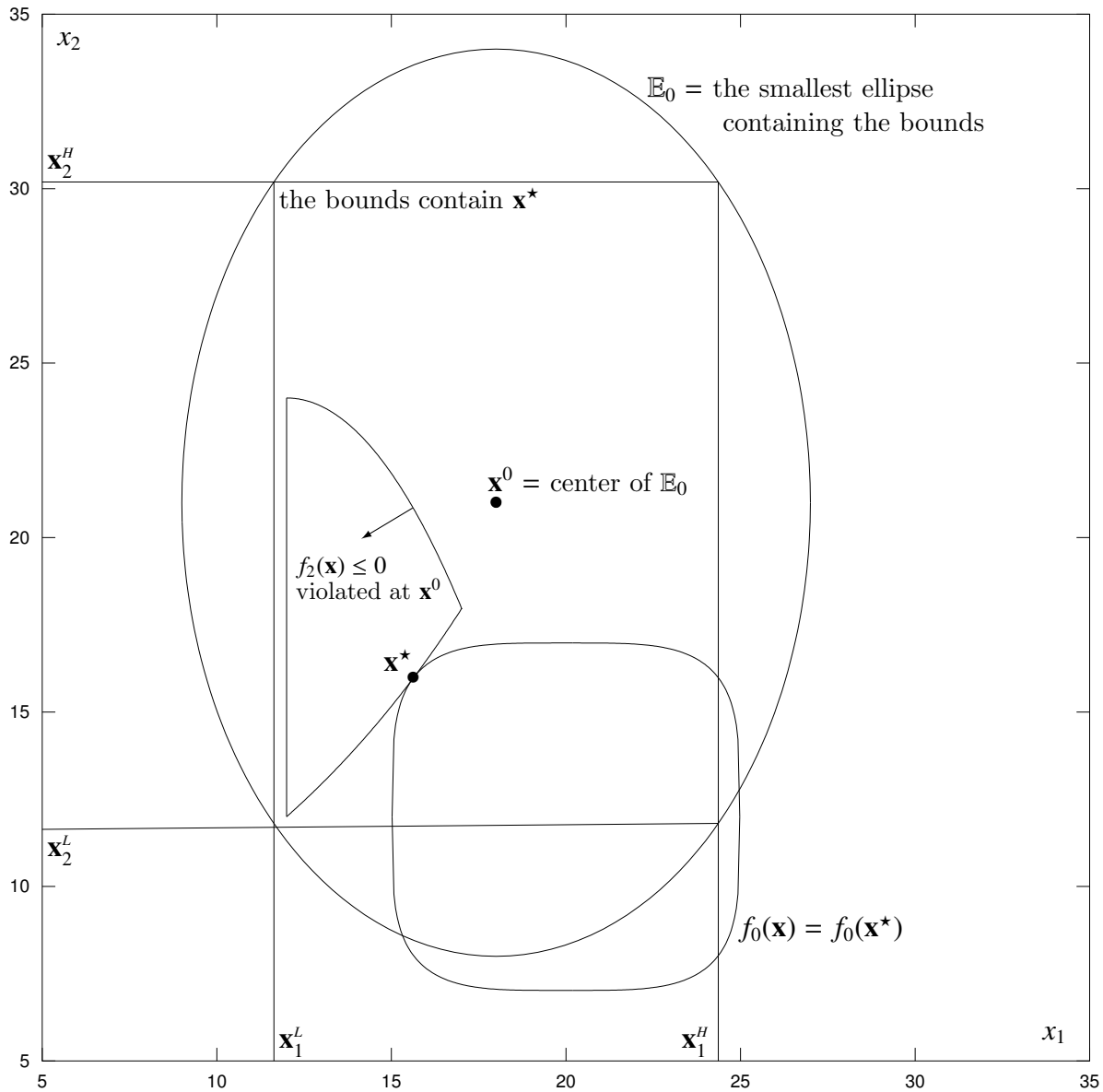
Although it is possible to realize this **space-confinement** idea by dicing the region enclosed by the bounds into successively smaller hyperrectangles [1, p675-683], it is algebraically more convenient to use simpler geometric figures. The **Nelder-Mead algorithm** [121] [120, §14], a venerable technique for unconstrained nonlinear programming, attempts to envelop $\mathbf{x}^\star$ in successively smaller simplices; **ellipsoid algorithms** are so called because they attempt to envelop $\mathbf{x}^\star$ in successively smaller ellipsoids.

One of the ellipsoid algorithm variants we will study also provides an easy way to progressively tighten the bounds, allowing us to carry out the process suggested by the picture.

24.2 Shor's Algorithm for Inequality Constraints

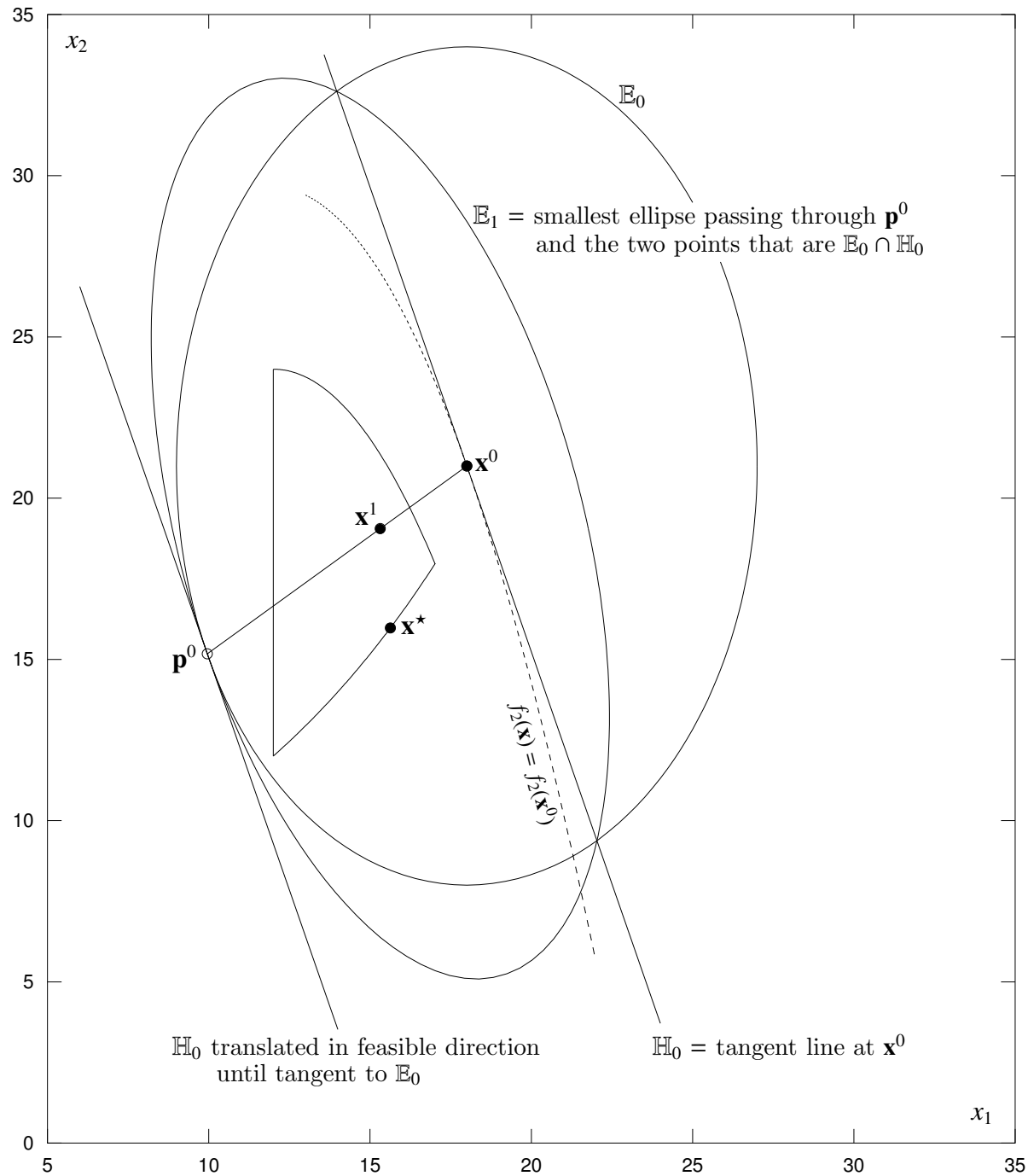
The simplest ellipsoid method is due to Shor [143]. To illustrate the basic idea of **Shor's algorithm** I will graphically perform its first few steps in solving the `ek1` problem below (this problem [3, p315] was first introduced in Exercise 21.4.30; see §28.7.29).



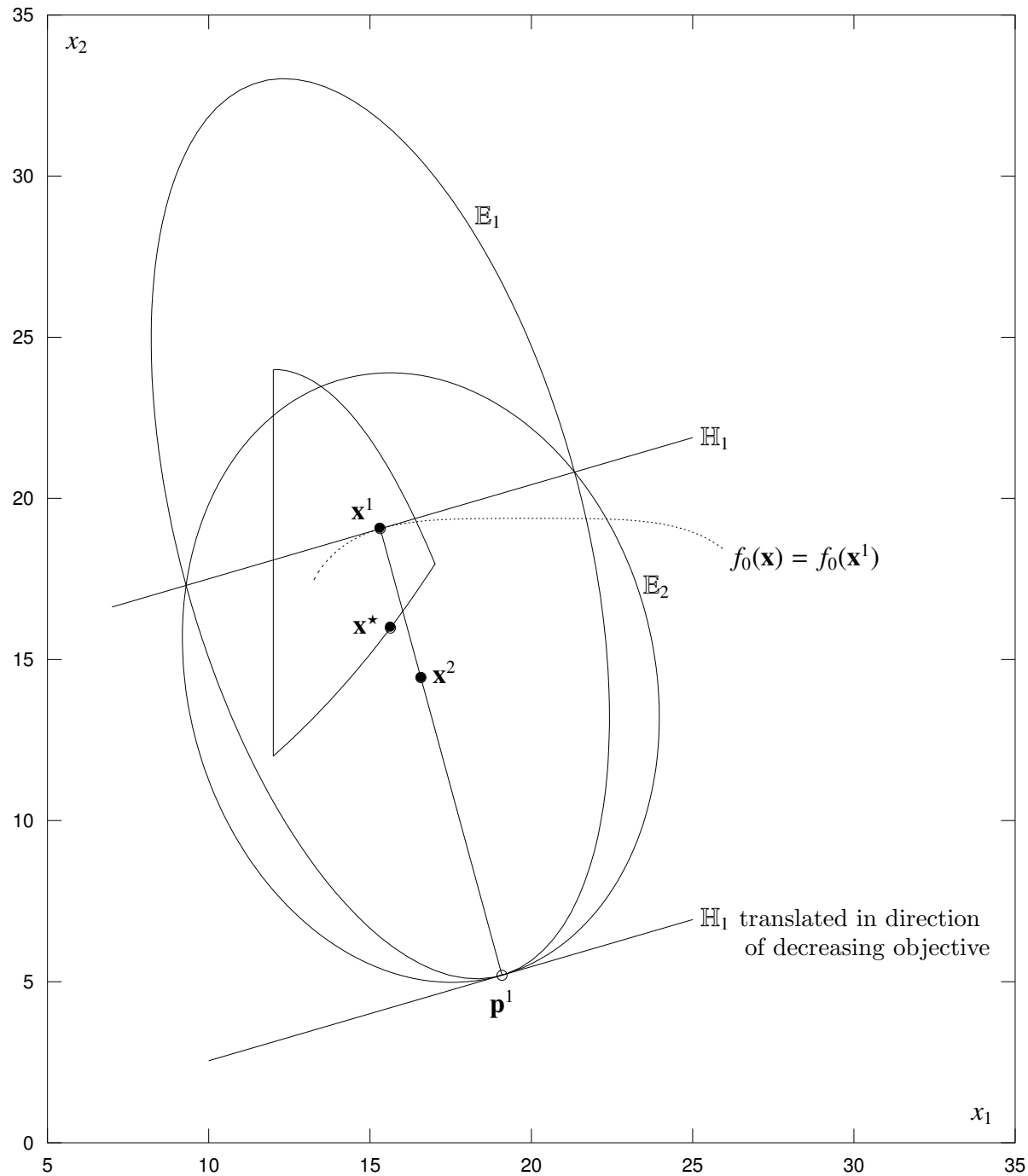


We begin with bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ on the variables. These bounds contain the feasible set so they must include $\mathbf{x}^\star$, and the ellipse enclosing the bounds also contains $\mathbf{x}^\star$. Many ellipses can be found that pass through the corners of the box, and we pick the smallest of them to be $\mathbb{E}_0$. The center $\mathbf{x}^0$ of $\mathbb{E}_0$ is the midpoint of the bounds. From the picture we can see that $\mathbf{x}^0$ violates the constraint $f_2(\mathbf{x}) \le 0$; the other two constraints happen to be satisfied there.

On the next page I have drawn the contour $f_2(\mathbf{x}) = f_2(\mathbf{x}^0)$ through $\mathbf{x}^0$ and a line $\mathbb{H}_0$ tangent to the contour at that point. This line divides $\mathbb{E}_0$ in half. All of the points in the upper-right half of $\mathbb{E}_0$ are even more infeasible for $f_2(\mathbf{x}) \le 0$ than $\mathbf{x}^0$ is, so we can throw that half of $\mathbb{E}_0$ away. To do that we translate $\mathbb{H}_0$ parallel to itself, in the direction of satisfying the constraint, until it is tangent to $\mathbb{E}_0$ at the point $\mathbf{p}^0$. Then we construct a new ellipse



$\mathbb{E}_1$ as the smallest one passing through $\mathbf{p}^0$ and the two points that are the intersection of $\mathbb{E}_0$ with $\mathbb{H}_0$. This is called a **phase 1** iteration of the algorithm. As we shall see later, the center $\mathbf{x}^1$ of $\mathbb{E}_1$ is on the line between $\mathbf{x}^0$ and $\mathbf{p}^0$ (in $\mathbb{R}^2$ it is one-third of the way). The new point $\mathbf{x}^1$ happens to be feasible, so a violated constraint can't be used to bisect $\mathbb{E}_1$.



However, we can see from the contour of $f_0(\mathbf{x})$ passing through $\mathbf{x}^1$ that the top half of $\mathbb{E}_1$ contains only points having a higher objective value than $f_0(\mathbf{x}^1)$ and can therefore be thrown away. As before we translate $\mathbb{H}_1$ parallel to itself until it is tangent to $\mathbb{E}_1$ at $\mathbf{p}^1$ and then construct $\mathbb{E}_2$ as the smallest ellipse passing through $\mathbf{p}^1$ and $\mathbb{E}_1 \cap \mathbb{H}_1$. This is called a **phase 2** iteration of the algorithm.

Each bisection of an ellipse by a line through its center is called a **center cut**. When the cutting line is tangent to the contour of a violated constraint (as is $\mathbb{H}_0$) the iteration is called a **feasibility cut**; when it is tangent to a contour of the objective (as is $\mathbb{H}_1$) the iteration is called an **optimality cut**. The new point $\mathbf{x}^2$ happens to violate the constraint $f_1(\mathbf{x}) \le 0$ so the next step in the algorithm will be another feasibility cut, but phase 1 and phase 2 iterations typically occur in an irregular sequence as the algorithm progresses. Each ellipse $\mathbb{E}_k$ is smaller than the previous one $\mathbb{E}_{k-1}$, and each contains $\mathbf{x}^\star$, so for this problem the $\mathbf{x}^k$ converge to $\mathbf{x}^\star$ as $k \to \infty$.

24.3 The Algebra of Shor's Algorithm

To complete the solution of a nonlinear program by carrying out Shor's method graphically would be impractical in $\mathbb{R}^2$ and hopeless in higher dimensions. Fortunately it is possible to find $\mathbb{E}_k$, $\mathbb{H}_k$, $\mathbf{p}^k$ and $\mathbf{x}^k$ by doing algebra rather than geometry, and then we will be able (in §24.4) to implement the algorithm by doing numerical calculations.

24.3.1 Ellipsoids in $\mathbb{R}^n$

In §14.7.2 I described an ellipsoid centered at the origin as the locus of points satisfying $\mathbf{x}^\top \square \mathbf{x} = 1$, where $\square$ is a positive-definite symmetric matrix. There it was convenient to call the matrix $\mathbf{Q}$, but in discussing the ellipsoid algorithm it is more convenient to call the matrix $\mathbf{Q}^{-1}$ and describe the ellipsoid centered at the iterate $\mathbf{x}^k$ as

$$\mathbb{E}_k = \left\{ \mathbf{x} \in \mathbb{R}^n \;\middle|\; (\mathbf{x} - \mathbf{x}^k)^\top \mathbf{Q}_k^{-1} (\mathbf{x} - \mathbf{x}^k) = 1 \right\}.$$

Then it will turn out that $\mathbf{Q}_{k+1}$ can be obtained from $\mathbf{Q}_k$ by a simple rank-one update, while $\mathbf{Q}_{k+1}^{-1}$ depends on $\mathbf{Q}_k^{-1}$ in a much more complicated way (the two updates are related by the Sherman-Morrison-Woodbury formula of §13.4.4; see Exercise 24.10.22). The resulting algorithm will manipulate only $\mathbf{Q}$, so that $\mathbf{Q}^{-1}$ is never actually needed.

With the definition above we can use linear algebra to find the ellipsoid $\mathbb{E}_0$ passing through the corners of the box $\mathbb{B} = \{ \mathbf{x} \in \mathbb{R}^n \mid \mathbf{x}^{\mathrm{L}} \le \mathbf{x} \le \mathbf{x}^{\mathrm{H}} \}$ that is formed by the bounds. To touch all of the corners $\mathbb{E}_0$ must be a right ellipsoid, so from symmetry $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}})$. To find $\mathbf{Q}_0^{-1}$ it is helpful to make a transformation of coordinates that centers the box $\mathbb{B}$ at the origin and scales its sides to unit length. To do this we can let $z_j = (x_j - x_j^0)/(x_j^H - x_j^L)$, or

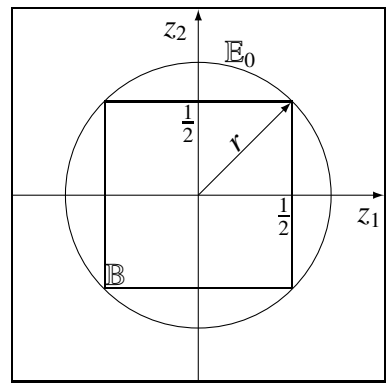
$$\mathbf{z} = \begin{bmatrix} 1/(x_1^H - x_1^L) & 0 & \cdots & 0 \\ 0 & 1/(x_2^H - x_2^L) & \cdots & 0 \\ 0 & 0 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & 0 \\ 0 & 0 & \cdots & 1/(x_n^H - x_n^L) \end{bmatrix} \begin{bmatrix} x_1 - x_1^0 \\ \vdots \\ x_n - x_n^0 \end{bmatrix} = \mathbf{W}(\mathbf{x} - \mathbf{x}^0).$$

Then $(\mathbf{x} - \mathbf{x}^0) = \mathbf{W}^{-1}\mathbf{z}$. To find the $\mathbf{z}$–space representation of the box $\mathbb{B}$, we can reason as follows. If $\mathbf{x} \in \mathbb{B}$ then

$$\begin{array}{rcccl}
\mathbf{x}^{\mathrm{L}} & \leq & \mathbf{x} & \leq & \mathbf{x}^{\mathrm{H}} \\
\mathbf{x}^{\mathrm{L}} - \mathbf{x}^0 & \leq & \mathbf{x} - \mathbf{x}^0 & \leq & \mathbf{x}^{\mathrm{H}} - \mathbf{x}^0 \\
\mathbf{x}^{\mathrm{L}} - \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}}) & \leq & \mathbf{W}^{-1}\mathbf{z} & \leq & \mathbf{x}^{\mathrm{H}} - \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}}) \\
-\frac{1}{2}(\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}}) & \leq & \mathbf{W}^{-1}\mathbf{z} & \leq & +\frac{1}{2}(\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}}) \\
-\frac{1}{2}\mathbf{W}(\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}}) & \leq & \mathbf{z} & \leq & +\frac{1}{2}\mathbf{W}(\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}}).
\end{array}$$

But $\mathbf{W}(\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}}) = \mathbf{1}$, so

$$\mathbb{B} = \left\{ \mathbf{z} \;\middle|\; -\tfrac{1}{2}\mathbf{1} \leq \mathbf{z} \leq +\tfrac{1}{2}\mathbf{1} \right\}.$$



The transformation to $\mathbf{z}$–space has made $\mathbb{B}$ a hypercube of side length 1 centered at the origin, so the smallest ellipsoid passing through its corners is an n–dimensional hypersphere. The picture to the right shows the box and its circumscribing hypersphere for $\mathbf{z} \in \mathbb{R}^2$, where the hypersphere is a circle of radius

$$r = \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2} = \sqrt{2\left(\frac{1}{2}\right)^2} = \frac{\sqrt{2}}{2}.$$

For $\mathbf{z} \in \mathbb{R}^n$, the hypersphere has radius

$$r = \sqrt{\underbrace{\left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^2}_{n \text{ terms}}} = \sqrt{n\left(\frac{1}{2}\right)^2} = \frac{\sqrt{n}}{2}$$

so its equation is $\mathbf{z}^\top\mathbf{z} = r^2 = n/4$ or $\mathbf{z}^\top(4/n)\mathbf{z} = 1$. Above we found that $\mathbf{z} = \mathbf{W}(\mathbf{x} - \mathbf{x}^0)$ so in $\mathbf{x}$–space the hypersphere is the ellipsoid whose equation is

$$\left[\mathbf{W}(\mathbf{x} - \mathbf{x}^0)\right]^\top (4/n) \left[\mathbf{W}(\mathbf{x} - \mathbf{x}^0)\right] = 1 \qquad \text{or} \qquad (\mathbf{x} - \mathbf{x}^0)^\top \underbrace{\mathbf{W}^\top(4/n)\mathbf{W}}_{\mathbf{Q}_0^{-1}} (\mathbf{x} - \mathbf{x}^0) = 1.$$

Thus

$$\mathbf{Q}_0^{-1} = \frac{4}{n}\mathbf{W}^\top\mathbf{W} = \frac{4}{n}\begin{bmatrix} 1/(x_1^H - x_1^L)^2 & 0 & \cdots & 0 \\ 0 & 1/(x_2^H - x_2^L)^2 & \cdots & 0 \\ 0 & 0 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & 0 \\ 0 & 0 & \cdots & 1/(x_n^H - x_n^L)^2 \end{bmatrix}$$

and

$$\mathbf{Q}_0 = \frac{n}{4}\begin{bmatrix} (x_1^H - x_1^L)^2 & 0 & \cdots & 0 \\ 0 & (x_2^H - x_2^L)^2 & \cdots & 0 \\ 0 & 0 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & 0 \\ 0 & 0 & \cdots & (x_n^H - x_n^L)^2 \end{bmatrix}.$$

So from the bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ we can easily find the $\mathbf{x}^0$ and $\mathbf{Q}_0$ defining the starting ellipsoid $\mathbb{E}_0$. Here is a MATLAB function that performs the calculation.

```
function [xzero,Qzero]=eainit(xl,xh)
% find the smallest ellipsoid enclosing given bounds

  xzero=(xl+xh)/2;                        % midpoint of bounds
  n=size(xzero,1);                        % number of variables
  Qzero=zeros(n,n);                       % zero matrix
  for j=1:n                               % fill in
      Qzero(j,j)=(n/4)*(xh(j)-xl(j))^2;   % the diagonal
  end                                     % elements

end
```

To find the initial ellipsoid in solving `ek1`, illustrated above, I chose the bounds [3, p316]

$$\begin{aligned} \mathbf{x}^{\mathrm{H}} &= \left[18 + 9/\sqrt{2},\ 21 + 13/\sqrt{2}\right]^{\top} \\ \mathbf{x}^{\mathrm{L}} &= \left[18 - 9/\sqrt{2},\ 21 - 13/\sqrt{2}\right]^{\top} \end{aligned} \quad \text{so that} \quad \begin{aligned} \mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}} &= \left[18/\sqrt{2},\ 26/\sqrt{2}\right]^{\top} \\ \mathbf{x}^{\mathrm{H}} + \mathbf{x}^{\mathrm{L}} &= [36,\ 42]^{\top}. \end{aligned}$$

They yield $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{H}} + \mathbf{x}^{\mathrm{L}}) = [18, 21]^{\top}$ and

$$\mathbf{Q}_0 = \frac{2}{4}\begin{bmatrix} (18/\sqrt{2})^2 & 0 \\ 0 & (26/\sqrt{2})^2 \end{bmatrix} = \begin{bmatrix} 81 & 0 \\ 0 & 169 \end{bmatrix}.$$

The Octave session on the next page shows `eainit.m` finding these results `1>-3>`.

Although the algorithm implementation will use and update $\mathbf{Q}_k$ rather than its inverse, to plot an ellipsoid $\mathbb{E}_k$ we need to use the matrix $\mathbf{Q}_k^{-1}$ that appears in its definition. For the starting ellipse we found above,

$$\mathbf{Q}_0^{-1} = \begin{bmatrix} \frac{1}{81} & 0 \\ 0 & \frac{1}{169} \end{bmatrix}.$$

The Octave session shows `5>-7>` how, using $\mathbf{x}^0$ and $\mathbf{Q}_0^{-1}$, the `ellipse.m` routine of §14.7.3 can be used to draw the ellipse $\mathbb{E}_0$ in the first figure of the `ek1` graphical solution.


```
octave:1> xl=[18-9/sqrt(2);21-13/sqrt(2)]
xl =

   11.636
   11.808

octave:2> xh=[18+9/sqrt(2);21+13/sqrt(2)]
xh =

   24.364
   30.192

octave:3> [xzero,Qzero]=eainit(xl,xh)
xzero =

   18
   21

Qzero =

    81.00000     0.00000
     0.00000   169.00000

octave:4> Qinv=inv(Qzero)
Qinv =

   0.012346  -0.000000
   0.000000   0.005917

octave:5> [xt,yt,rc,tmax]=ellipse(xzero(1),xzero(2),Qinv,25);
octave:6> plot(xt,yt)
octave:7> axis('equal')
octave:8> quit
```

24.3.2 Hyperplanes in $\mathbb{R}^n$

Each hyperplane generated by Shor's algorithm is tangent to a contour of one of the functions in the optimization problem. If $\mathbb{H}_k$ is tangent at $\mathbf{x}^k$ to the contour $f_i(\mathbf{x}) = f_i(\mathbf{x}^k)$, it is said to **support** the contour at $\mathbf{x}^k$ (see §11.2) and it can be described as

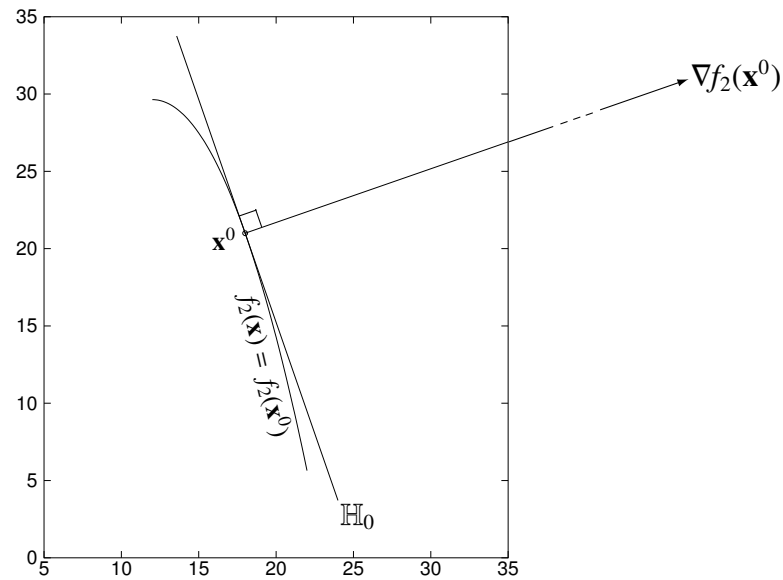
$$\mathbb{H}_k = \left\{ \mathbf{x} \in \mathbb{R}^n \;\middle|\; \nabla f_i(\mathbf{x}^k)^\top(\mathbf{x} - \mathbf{x}^k) = 0 \right\}.$$

In our graphical solution of `ek1` the hyperplane $\mathbb{H}_0$ supports the contour $f_2(\mathbf{x}) = f_2(\mathbf{x}^0)$ at $\mathbf{x}^0 = [18, 21]^\top$, and using the definition above we can find its equation.

$$\begin{aligned} f_2(\mathbf{x}) &= 6(x_1 - 12)^2 + 25x_2 - 600 \\ \nabla f_2(\mathbf{x}^0) &= \begin{bmatrix} 12(x_1^0 - 12) \\ 25 \end{bmatrix} = \begin{bmatrix} 72 \\ 25 \end{bmatrix} \\ \text{On } \mathbb{H}_0, \quad \nabla f_2(\mathbf{x}^0)^\top \mathbf{x} &= \nabla f_2(\mathbf{x}^0)^\top \mathbf{x}^0 = \begin{bmatrix} 72 & 25 \end{bmatrix} \begin{bmatrix} 18 \\ 21 \end{bmatrix} = 1821. \end{aligned}$$

Thus the hyperplane is

$$72x_1 + 25x_2 = 1821.$$



The graph above is an excerpt of the second picture in the §24.2 graphical solution of `ek1`, showing part of the contour $f_2(\mathbf{x}) = f_2(\mathbf{x}^0)$, its gradient $\nabla f_2(\mathbf{x}^0)$, and the supporting hyperplane $\mathbb{H}_0$. For every point $\mathbf{x} \in \mathbb{H}_0$ the vector $(\mathbf{x} - \mathbf{x}^0)$ is orthogonal to $\nabla f_2(\mathbf{x}^0)$, so $\nabla f_2(\mathbf{x}^0)^\top(\mathbf{x} - \mathbf{x}^0) = 0$.

The gradient vector is about 76 units long so it can't be drawn to scale in the frame of the picture, but $\mathbb{H}_0$ is determined by the *direction* of the gradient rather than its length. In the definition of $\mathbb{H}_k$ we could replace $\nabla f_i(\mathbf{x}^k)$ by $\mathbf{g} = \nabla f_i(\mathbf{x}^k)/\|\nabla f_i(\mathbf{x}^k)\|$, the normalized gradient or **unit normal** to the hyperplane, and we will also find other places where it is possible to use $\mathbf{g}$ in place of $\nabla f_i(\mathbf{x})$.

In finding $\mathbb{H}_0$ above I rearranged the equation in the definition as $\nabla f_i(\mathbf{x}^k)^\top\mathbf{x} = \nabla f_i(\mathbf{x}^k)^\top\mathbf{x}^k$, but $\nabla f_i(\mathbf{x}^k)^\top\mathbf{x}^k$ is just a scalar constant (for this cut it came out 1821). Changing the constant translates the hyperplane but does not affect its slope, so every hyperplane parallel to $\mathbb{H}_k$ has the equation $\mathbf{g}^\top\mathbf{x} = \kappa$ for some constant κ.

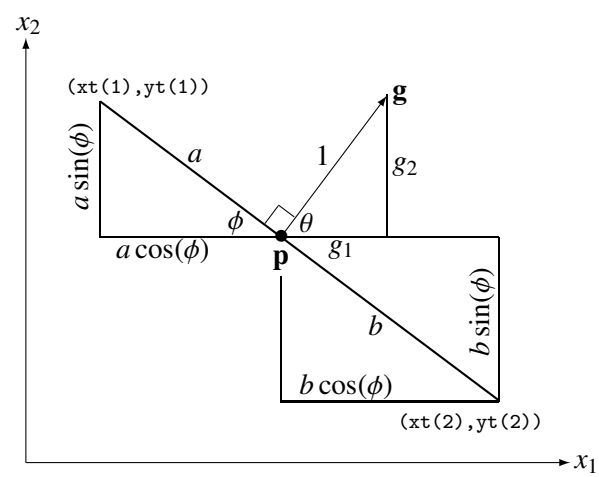
As we study ellipsoid algorithms it will often be helpful to plot some hyperplane in $\mathbb{R}^2$. Given $\mathbf{x}^k$ and $\nabla f_i(\mathbf{x}^k)$, it is not difficult to find the equation as we did above and then to work out the coordinates of the endpoints of the line to be drawn. Despite the fact that this process is trivial (or perhaps *because* it is trivial) it is also tedious and easy to get wrong, so I wrote the `hplane.m` routine listed on the next page to automate the calculations. Its input parameters [1] are the gradient vector `del`, a point `p` on the hyperplane, and scalars `a` and `b` specifying how far the line should extend on each side of that point. The code begins by [7] normalizing the gradient and [8-23] handling special cases. If the gradient is slanted [25-28] it sets the endpoints of the line segment by using the formulas derived below the listing. These Octave commands plot the $\mathbb{H}_0$ that is shown above and in §24.2.

```
octave:1> [xhp,yhp]=hplane([72;25],13.5,[18;21],18.3)
octave:2> plot(xhp,yhp)
octave:3> axis('equal')
```



```
 1 function [xt,yt]=hplane(del,a,p,b)
 2 % return in xt and yt the endpoints of a line segment
 3 % that is part of the hyperplane del'x=del'p
 4 % and goes from a units on one side of p to b units on the other
 5
 6   xt=zeros(2,1); yt=zeros(2,1);           % xt and yt are columns
 7   g=del/norm(del);                        % unit normal to H
 8   if(g(1) == 0 && g(2) == 0) return; end  % if zero give up
 9
10   if(g(1) == 0)          % if gradient is vertical
11      yt(1)=p(2);         % draw
12      xt(1)=p(1)-a;       % a
13      yt(2)=p(2);         % horizontal
14      xt(2)=p(1)+b;       % line
15      return
16   end
17   if(g(2) == 0)          % if gradient is horizontal
18      xt(1)=p(1);         % draw
19      yt(1)=p(2)-a;       % a
20      xt(2)=p(1);         % vertical
21      yt(2)=p(2)+b;       % line
22      return
23   end
24
25   xt(1)=p(1)-a*g(2);     % gradient is slanted
26   yt(1)=p(2)+a*g(1);     % draw a line
27   xt(2)=p(1)+b*g(2);     % orthogonal to
28   yt(2)=p(2)-b*g(1);     % the gradient
29
30 end
```

In the construction to the right the thick line is part of the hyperplane orthogonal to $\mathbf{g}$ at the point $\mathbf{p}$. The gradient vector makes an angle θ with the horizontal so the hyperplane makes the angle $\phi = \pi/2 - \theta$ with the horizontal. The projections of the a and b parts of the line onto the coordinate directions are



$$
\begin{aligned}
a\cos(\phi) &= a\sin(\theta) = ag_2 \\
a\sin(\phi) &= a\cos(\theta) = ag_1 \\
b\cos(\phi) &= b\sin(\theta) = bg_2 \\
b\sin(\phi) &= b\cos(\theta) = bg_1
\end{aligned}
$$

so the endpoints of the line are given by the formulas 25-28 in the code.

24.3.3 Finding the Next Ellipsoid

Given $\mathbf{x}^k$ and $\mathbf{Q}_k$ defining the ellipsoid

$$\mathbb{E}_k = \left\{ \mathbf{x} \;\middle|\; (\mathbf{x} - \mathbf{x}^k)^\top \mathbf{Q}_k^{-1} (\mathbf{x} - \mathbf{x}^k) = 1 \right\}$$

and the hyperplane

$$\mathbb{H}_k = \left\{ \mathbf{x} \;\middle|\; \mathbf{g}^\top(\mathbf{x} - \mathbf{x}^k) = 0 \right\}$$

cutting $\mathbb{E}_k$ through its center, Shor's algorithm finds $\mathbf{x}^{k+1}$ and $\mathbf{Q}_{k+1}$ defining an ellipsoid

$$\mathbb{E}_{k+1} = \left\{ \mathbf{x} \;\middle|\; (\mathbf{x} - \mathbf{x}^{k+1})^\top \mathbf{Q}_{k+1}^{-1} (\mathbf{x} - \mathbf{x}^{k+1}) = 1 \right\}$$

that is the smallest one passing through $\mathbf{p}^k$ and $\mathbb{E}_k \cap \mathbb{H}_k$. In this Section we will derive update formulas [3, p318] that give $\mathbf{x}^{k+1}$ and $\mathbf{Q}_{k+1}$ in terms of $\mathbf{x}^k$, $\mathbf{Q}_k$, and $\mathbf{g}$ [98, §2.2].

To study the properties of $\mathbb{E}_{k+1}$ it is again helpful to make a transformation of coordinates, this time to a space in which $\mathbb{E}_k$ is shifted and scaled to be a hypersphere of radius 1 centered at the origin. We can do this by writing $\mathbf{Q}_k$ as the product of its Cholesky factors, $\mathbf{Q}_k = \mathbf{U}^\top\mathbf{U}$, and letting $\mathbf{w} = \mathbf{U}^{-\top}(\mathbf{x} - \mathbf{x}^k)$. Then $(\mathbf{x} - \mathbf{x}^k) = \mathbf{U}^\top\mathbf{w}$ so

$$(\mathbf{x} - \mathbf{x}^k)^\top \mathbf{Q}_k^{-1} (\mathbf{x} - \mathbf{x}^k) = [\mathbf{U}^\top\mathbf{w}]^\top [\mathbf{U}^\top\mathbf{U}]^{-1} [\mathbf{U}^\top\mathbf{w}] = \mathbf{w}^\top\mathbf{U}[\mathbf{U}^{-1}\mathbf{U}^{-\top}]\mathbf{U}^\top\mathbf{w} = \mathbf{w}^\top\mathbf{w}$$

and

$$\mathbb{E}_k = \left\{ \mathbf{w} \;\middle|\; \mathbf{w}^\top\mathbf{w} = 1 \right\}.$$

Making the same change of variables in the definition of $\mathbb{H}_k$, $\mathbf{g}^\top(\mathbf{x} - \mathbf{x}^k) = \mathbf{g}^\top\mathbf{U}^\top\mathbf{w} = (\mathbf{U}\mathbf{g})^\top\mathbf{w}$ so in $\mathbf{w}$–space the gradient vector becomes $\mathbf{v} = \mathbf{U}\mathbf{g}$ and

$$\mathbb{H}_k = \left\{ \mathbf{w} \;\middle|\; \mathbf{v}^\top\mathbf{w} = 0 \right\}.$$

The pictures on the next page show a typical iteration when $\mathbf{w} \in \mathbb{R}^2$, in which $\mathbb{E}_{k+1}$ is the next ellipsoid that we are trying to find. (These ellipses and hyperplanes are actually those of the first step in the §24.2 graphical solution of `ek1`, transformed to $\mathbf{w}$–space).

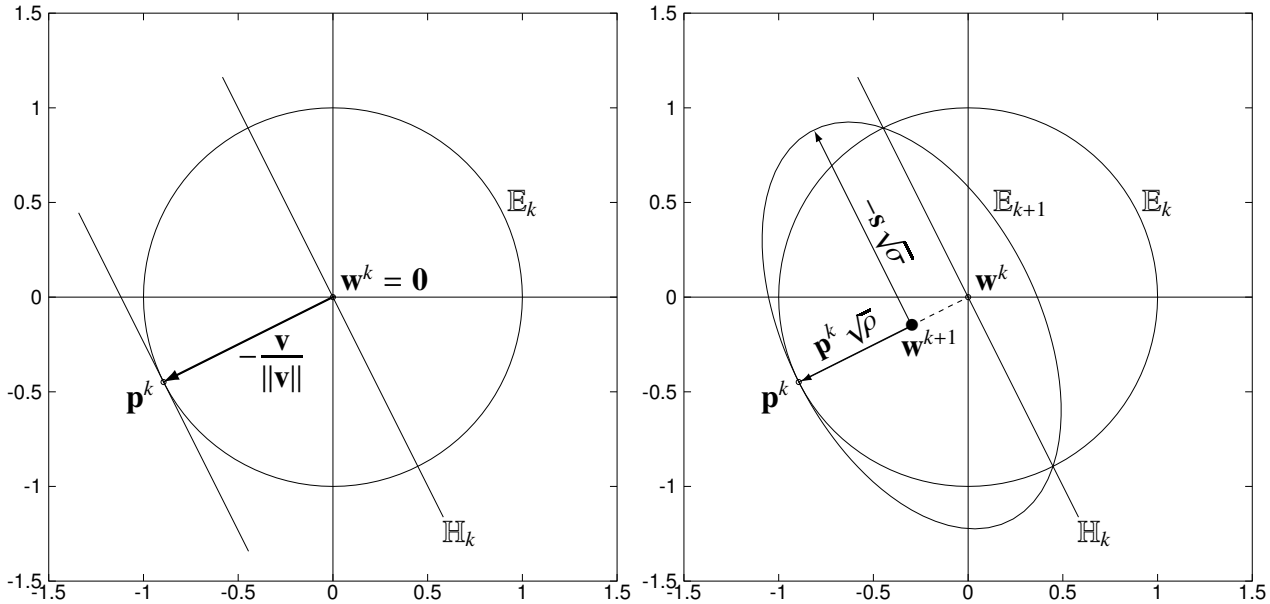
The algorithm moves $\mathbb{H}_k$ parallel to itself in the $-\mathbf{v}$ direction until it is tangent to $\mathbb{E}_k$ at $\mathbf{p}^k$. Because $\mathbb{E}_k$ is a hypersphere the point $\mathbf{p}^k$ is in the direction $-\mathbf{v}$ from the center of $\mathbb{E}_k$ (which we made the origin) and because $\mathbb{E}_k$ has unit radius $\mathbf{p}^k$ is a distance of 1 from the origin. Thus $\mathbf{p}^k = -\mathbf{v}/\|\mathbf{v}\|$; in $\mathbf{w}$–space, $\mathbf{p}^k$ is just a unit normal to $\mathbb{H}_k$.

We can also transform $\mathbb{E}_{k+1}$ to $\mathbf{w}$–space. First notice that

$$\begin{aligned}
(\mathbf{x} - \mathbf{x}^{k+1}) &= (\mathbf{x} - \mathbf{x}^k) + \mathbf{x}^k - \mathbf{x}^{k+1} \\
&= \mathbf{U}^\top\mathbf{w} - (\mathbf{x}^{k+1} - \mathbf{x}^k) \\
&= \mathbf{U}^\top\left[\mathbf{w} - \mathbf{U}^{-\top}\left(\mathbf{x}^{k+1} - \mathbf{x}^k\right)\right] \\
&= \mathbf{U}^\top\left(\mathbf{w} - \mathbf{w}^{k+1}\right).
\end{aligned}$$

Then we can write

$$\begin{aligned}
(\mathbf{x} - \mathbf{x}^{k+1})^\top \mathbf{Q}_{k+1}^{-1} (\mathbf{x} - \mathbf{x}^{k+1}) &= \left[\mathbf{U}^\top(\mathbf{w} - \mathbf{w}^{k+1})\right]^\top \mathbf{Q}_{k+1}^{-1} \left[\mathbf{U}^\top(\mathbf{w} - \mathbf{w}^{k+1})\right] \\
&= (\mathbf{w} - \mathbf{w}^{k+1})^\top \left[\mathbf{U}\mathbf{Q}_{k+1}^{-1}\mathbf{U}^\top\right] (\mathbf{w} - \mathbf{w}^{k+1})
\end{aligned}$$



so the ellipsoid matrix $\mathbf{Q}_{k+1}^{-1}$ of $\mathbb{E}_{k+1}$ is transformed to $\mathbf{G}^{-1} = \mathbf{U}\mathbf{Q}_{k+1}^{-1}\mathbf{U}^\top$ and in $\mathbf{w}$–space

$$\mathbb{E}_{k+1} = \left\{ \mathbf{w} \;\middle|\; \left(\mathbf{w} - \mathbf{w}^{k+1}\right)^\top \mathbf{G}^{-1} \left(\mathbf{w} - \mathbf{w}^{k+1}\right) = 1 \right\}.$$

The geometry of the iteration in $\mathbf{w}$–space makes the vector from $\mathbf{w}^{k+1}$ to $\mathbf{p}^k$ collinear with the vector from $\mathbf{w}^k$ to $\mathbf{p}^k$, and this has three important consequences. First, $\mathbf{w}^{k+1}$ falls on that line, so $\mathbf{w}^{k+1} = \mathbf{w}^k + \alpha\mathbf{p}^\mathbf{k} = \alpha\mathbf{p}^\mathbf{k}$ for some step $\alpha \in [0, 1]$. Second, because the vector from $\mathbf{w}^{k+1}$ to $\mathbf{p}^k$ points in the direction of the minor axis of $\mathbb{E}_{k+1}$, the $\mathbf{w}$–space matrix $\mathbf{G}^{-1}$ of $\mathbb{E}_{k+1}$, and hence also its inverse $\mathbf{G}$, must by construction have $-\mathbf{v}/\|\mathbf{v}\| = \mathbf{p}^k$ as a unit eigenvector (see §14.7.2). I will use ρ to denote the associated eigenvalue of $\mathbf{G}$, so $\mathbf{G}\mathbf{p}^k = \rho\mathbf{p}^k$. Third, from symmetry all the other axes of $\mathbb{E}_{k+1}$ have the same length, so the unit eigenvectors of $\mathbf{G}$ in those directions all have the same associated eigenvalue, which I will call σ. The eigenvalues of $\mathbf{G}^{-1}$ are thus $1/\rho$ and $1/\sigma$, so the half-axes of $\mathbb{E}_{k+1}$ have lengths $1/\sqrt{1/\sigma}$ and $1/\sqrt{1/\rho}$ as shown in the picture on the right. There I call the major-axis unit eigenvector $\mathbf{s}$.

Many ellipsoids $\mathbb{E}_{k+1}$ can be constructed passing through $\mathbf{p}^k$ and $\mathbb{E}_k \cap \mathbb{H}_k$. Each can be characterized by the eigenvalues ρ and σ, which in turn depend on α. To investigate this dependence it is helpful to do yet another transformation of coordinates that rotates the picture to make $\mathbb{E}_{k+1}$ a right ellipse centered at the origin of $\mathbf{z}$–space, as shown on the next page. Let $\mathbf{S}$ be a matrix whose columns are the unit eigenvectors of $\mathbf{G}$, arranged so that $\mathbf{p}^k$ is its rightmost column, and let $\mathbf{\Lambda}$ be a diagonal matrix of the corresponding eigenvalues.

$$\mathbf{S} = \left[\mathbf{s}^1\ \mathbf{s}^2 \cdots \mathbf{s}^{n-1}\ \mathbf{p}^k\right] \qquad \mathbf{\Lambda} = \begin{bmatrix} \sigma & & & \\ & \ddots & & \\ & & \sigma & \\ & & & \rho \end{bmatrix}$$

Then $\mathbf{GS} = \mathbf{S\Lambda}$ so $\mathbf{G} = \mathbf{S\Lambda S}^{-1}$ and $\mathbf{G}^{-1} = \mathbf{S\Lambda}^{-1}\mathbf{S}^{-1}$. But $\mathbf{S}$ is an orthogonal matrix because its columns $\mathbf{s}^j$ are mutually orthogonal, so $\mathbf{S}^{-1} = \mathbf{S}^\top$ and $\mathbf{G}^{-1} = \mathbf{S\Lambda}^{-1}\mathbf{S}^\top$. Substituting this expression for $\mathbf{G}^{-1}$ into the definition of $\mathbb{E}_{k+1}$ and letting $\mathbf{S}^\top(\mathbf{w} - \mathbf{w}^{k+1}) = \mathbf{z}$ we find

$$(\mathbf{w} - \mathbf{w}^{k+1})^\top\mathbf{S\Lambda}^{-1}\mathbf{S}^\top(\mathbf{w} - \mathbf{w}^{k+1}) = \left[\mathbf{S}^\top(\mathbf{w} - \mathbf{w}^{k+1})\right]^\top\mathbf{\Lambda}^{-1}\left[\mathbf{S}^\top(\mathbf{w} - \mathbf{w}^{k+1})\right]$$

$$\mathbb{E}_{k+1} = \left\{\, \mathbf{z} \;\middle|\; \mathbf{z}^\top\mathbf{\Lambda}^{-1}\mathbf{z} = 1 \,\right\}.$$

Also, $\mathbf{w} - \mathbf{w}^{k+1} = \mathbf{S}^{-\top}\mathbf{z} = \mathbf{Sz}$ so $\mathbf{w} = \mathbf{w}^{k+1} + \mathbf{Sz}$. Then

$$\mathbf{w}^\top\mathbf{w} = (\mathbf{w}^{k+1} + \mathbf{Sz})^\top(\mathbf{w}^{k+1} + \mathbf{Sz}) = (\mathbf{w}^{k+1})^\top\mathbf{w}^{k+1} + 2\mathbf{z}^\top\mathbf{S}^\top\mathbf{w}^{k+1} + \mathbf{z}^\top\mathbf{z} = (\mathbf{z} + \mathbf{S}^\top\mathbf{w}^{k+1})^\top(\mathbf{z} + \mathbf{S}^\top\mathbf{w}^{k+1})$$

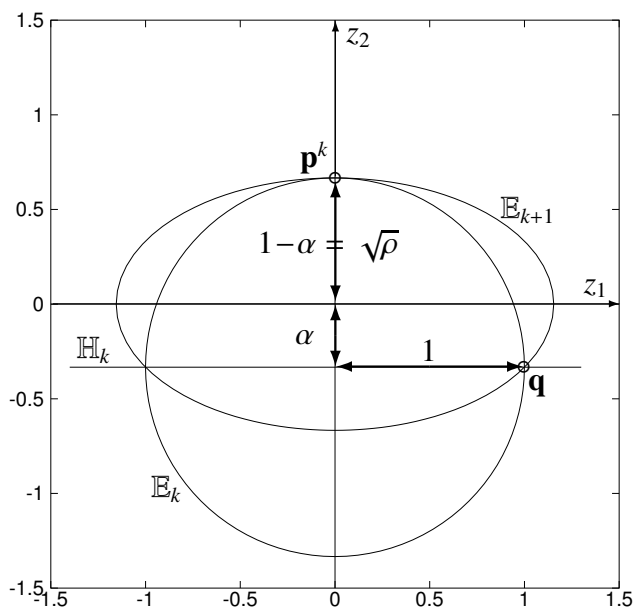
so

$$\mathbb{E}_k = \left\{\, \mathbf{z} \;\middle|\; \left[\mathbf{z} + \mathbf{S}^\top\mathbf{w}^{k+1}\right]^\top\left[\mathbf{z} + \mathbf{S}^\top\mathbf{w}^{k+1}\right] = 1 \,\right\}$$

and $\mathbf{v}^\top\mathbf{w} = \mathbf{v}^\top(\mathbf{w}^{k+1} + \mathbf{Sz})$ so

$$\mathbb{H}_k = \left\{\, \mathbf{z} \;\middle|\; \mathbf{v}^\top\mathbf{Sz} = -\mathbf{v}^\top\mathbf{w}^{k+1} \,\right\}.$$

I used the above definitions of $\mathbb{E}_k$, $\mathbb{H}_k$ and $\mathbb{E}_{k+1}$ in $\mathbf{z}$–space to plot the graph below.



This picture makes it obvious that $1 - \alpha = \sqrt{\rho}$, so $\rho = (1 - \alpha)^2$. In $\mathbb{R}^2$ the ellipsoids intersect at $\mathbf{q} = [1, -\alpha]^\top$ but because we arranged above for z_n to be the direction of the minor axis of $\mathbb{E}_{k+1}$, in $\mathbb{R}^n$ that point is in the z_1–z_n plane and has these coordinates.

$$\mathbf{q} = [1,\ \underbrace{0, \cdots, 0,}_{n-2 \text{ terms}}\ -\alpha]^\top$$

Points on $\mathbb{E}_{k+1}$ satisfy

$$\mathbf{z}^\top \mathbf{\Lambda}^{-1}\mathbf{z} = \frac{z_1^2}{\sigma} + \frac{z_2^2}{\sigma} + \cdots + \frac{z_{n-1}^2}{\sigma} + \frac{z_n^2}{\rho} = 1$$

and the point $\mathbf{q}$ is on $\mathbb{E}_{k+1}$ so

$$\frac{1}{\sigma} + \frac{0}{\sigma} + \cdots + \frac{0}{\sigma} + \frac{\alpha^2}{\rho} \quad = \quad \frac{1}{\sigma} + \frac{\alpha^2}{\rho} = 1.$$

Using $\rho = (1-\alpha)^2$ and solving for σ,

$$\frac{1}{\sigma} = 1 - \frac{\alpha^2}{\rho} = 1 - \frac{\alpha^2}{(1-\alpha)^2} = \frac{(1-\alpha)^2 - \alpha^2}{(1-\alpha)^2} = \frac{(1-2\alpha+\alpha^2) - \alpha^2}{(1-\alpha)^2} = \frac{1-2\alpha}{(1-\alpha)^2}$$

$$\sigma = \frac{(1-\alpha)^2}{1-2\alpha}$$

The formulas we have derived for $\rho(\alpha)$ and $\sigma(\alpha)$ define a family of ellipsoids $\mathbb{E}_{k+1}$ passing through $\mathbf{p}^k$ and $\mathbb{E}_k \cap \mathbb{H}_k$, parameterized by the step length α, from which we are to select the one having the smallest volume. Ratios of volumes are preserved by the transformations we have made, so the smallest ellipsoid in $\mathbf{z}$–space will also be the smallest ellipsoid in $\mathbf{w}$–space and in $\mathbf{x}$–space. Using a formula we derived in §14.7.2, in $\mathbf{z}$–space the volume of $\mathbb{E}_{k+1}$ is

$$\mathcal{V} = \mathcal{V}_1 \sqrt{|\mathbf{\Lambda}|}$$

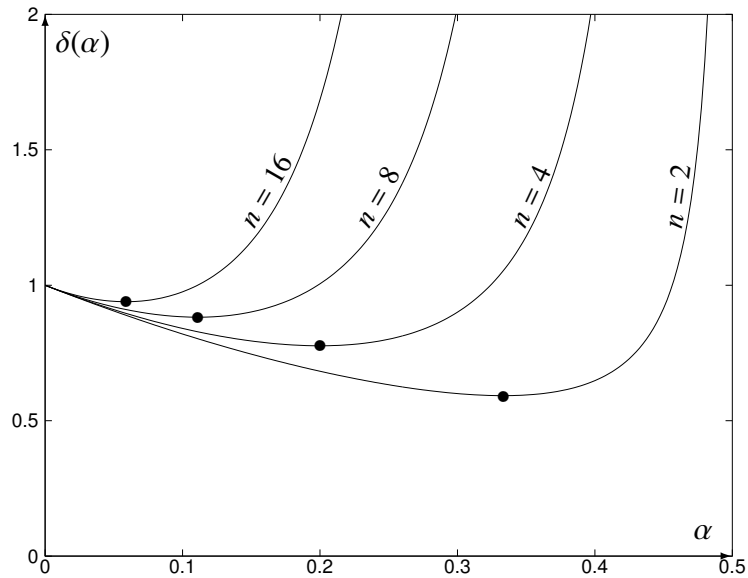
where $\mathcal{V}_1$ is the volume of a unit ball in $\mathbb{R}^n$ and the determinant is the product of the diagonals of $\mathbf{\Lambda}$. If we let

$$\delta(\alpha) = |\mathbf{\Lambda}| = \rho\,\sigma^{n-1} = (1-\alpha)^2 \left[\frac{(1-\alpha)^2}{1-2\alpha}\right]^{n-1} = (1-\alpha)^{2n}(1-2\alpha)^{1-n}$$

then to find the α that yields the ellipsoid of smallest volume we need only minimize $\delta(\alpha)$. Our analysis breaks down for $n = 1$ because $\delta(\alpha) = (1-\alpha)^2$ has its minimum at $\alpha = 1$ and that does not make sense when the ellipsoids are collinear line segments (the algorithm reduces to bisection in that case). For $n > 1$, the formula yields $\delta(\alpha) < 0$ for $\alpha > \frac{1}{2}$ and a division by zero for $\alpha = \frac{1}{2}$. Because $\mathbb{E}_{k+1}$ is symmetric about its major axes the requirement that it pass through $\mathbf{p}^k$ and also $\mathbb{E}_k \cap \mathbb{H}_k$ can be met only if $\alpha < \frac{1}{2}$. To study the behavior of $\delta(\alpha)$ in more detail I plotted the function for $\alpha \in [0, \frac{1}{2})$ on the next page. It appears to be convex (see Exercise 24.10.20) at least for the values of n that I tried, so I set the derivative to zero and solved for α.

$$\frac{d\delta}{d\alpha} = (1-\alpha)^{2n}\left[(1-n)(1-2\alpha)^{-n}(-2)\right] + (1-2\alpha)^{1-n}\left[(2n)(1-\alpha)^{2n-1}(-1)\right] = 0$$

$$(1-\alpha)^{2n}(1-n)(1-2\alpha)^{-n} = -n(1-\alpha)^{2n-1}(1-2\alpha)^{1-n}$$



Because $\alpha < \frac{1}{2}$ the terms $(1-2\alpha)^{-n}$ and $(1-\alpha)^{2n}$ are positive, so we can divide both sides by those factors.

$$\begin{aligned}
(1-n) &= -n(1-\alpha)^{-1}(1-2\alpha) \\
(1-\alpha)(1-n) &= -n+2n\alpha \\
n\alpha-\alpha+1-n &= 2n\alpha-n \\
1 &= n\alpha+\alpha=\alpha(n+1) \\
\alpha &= 1/(n+1)
\end{aligned}$$

This minimizing value of α is shown as a point on each of the curves plotted above. Substituting in the formulas we found earlier,

$$\begin{aligned}
\rho=(1-\alpha)^2 &= \left(1-\frac{1}{n+1}\right)^2=\frac{n^2}{(n+1)^2} \qquad \text{and} \qquad 1-2\alpha=1-\frac{2}{n+1}=\frac{n-1}{n+1} \\
\sigma=\frac{(1-\alpha)^2}{1-2\alpha} &= \frac{n^2}{(n+1)^2}\times\frac{n+1}{n-1}=\frac{n^2}{(n+1)(n-1)}=\frac{n^2}{n^2-1}.
\end{aligned}$$

To see how these eigenvalues characterize $\mathbb{E}_{k+1}$ we can write

$$\mathbf{\Lambda}=\begin{bmatrix}\sigma & & & \\ & \ddots & & \\ & & \sigma & \\ & & & \rho\end{bmatrix}=\sigma\mathbf{I}-(\sigma-\rho)\mathbf{D} \qquad \text{where} \qquad \mathbf{D}=\begin{bmatrix}0 & & & \\ & \ddots & & \\ & & 0 & \\ & & & 1\end{bmatrix}.$$

Then $\mathbf{G} = \mathbf{S}\boldsymbol{\Lambda}\mathbf{S}^\top = \mathbf{S}[\sigma\mathbf{I}-(\sigma-\rho)\mathbf{D}]\mathbf{S}^\top = \sigma\mathbf{S}\mathbf{I}\mathbf{S}^\top-(\sigma-\rho)\mathbf{S}\mathbf{D}\mathbf{S}^\top$. Because $\mathbf{S}^\top = \mathbf{S}^{-1}$, the first matrix product in the final expression is $\sigma\mathbf{S}\mathbf{I}\mathbf{S}^\top = \sigma\mathbf{I}$. Because $\mathbf{D}$ is zero except for its bottom right element which is 1, in the second matrix product $\mathbf{S}\mathbf{D}\mathbf{S}^\top = \mathbf{p}^k\mathbf{p}^{k\top}$. Thus $\mathbf{G} = \sigma\mathbf{I} - (\sigma-\rho)\mathbf{p}^k\mathbf{p}^{k\top}$.

Using our earlier definitions of $\mathbf{G}^{-1} = \mathbf{U}\mathbf{Q}_{k+1}^{-1}\mathbf{U}^\top$ and $\mathbf{U}^\top\mathbf{U} = \mathbf{Q}_k$ along with $\mathbf{p}^k = -\mathbf{U}\mathbf{g}/\|\mathbf{U}\mathbf{g}\|$, we can find $\mathbf{Q}_{k+1}$ in terms of ρ and σ.

$$\begin{aligned}
\mathbf{G}^{-1} &= \mathbf{U}\mathbf{Q}_{k+1}^{-1}\mathbf{U}^\top \\
\mathbf{U}^{-1}\mathbf{G}^{-1}\mathbf{U}^{-\top} &= \mathbf{Q}_{k+1}^{-1} \\
\mathbf{Q}_{k+1} &= \mathbf{U}^\top\mathbf{G}\mathbf{U} \\
&= \mathbf{U}^\top[\sigma\mathbf{I} - (\sigma-\rho)\,\mathbf{p}^k\mathbf{p}^{k\top}]\mathbf{U} \\
&= \sigma\mathbf{U}^\top\mathbf{U} - (\sigma-\rho)\,\mathbf{U}^\top\mathbf{p}^k\mathbf{p}^{k\top}\mathbf{U} \\
&= \sigma\mathbf{U}^\top\mathbf{U} - (\sigma-\rho)\,\frac{\mathbf{U}^\top\mathbf{U}\mathbf{g}\mathbf{g}^\top\mathbf{U}^\top\mathbf{U}}{[\mathbf{U}\mathbf{g}]^\top[\mathbf{U}\mathbf{g}]} \\
&= \sigma\mathbf{U}^\top\mathbf{U} - (\sigma-\rho)\,\frac{\mathbf{U}^\top\mathbf{U}\mathbf{g}\mathbf{g}^\top\mathbf{U}^\top\mathbf{U}}{\mathbf{g}^\top\mathbf{U}^\top\mathbf{U}\mathbf{g}} \\
&= \sigma\left(\mathbf{Q}_k - \frac{\sigma-\rho}{\sigma}\,\frac{\mathbf{Q}_k\mathbf{g}\mathbf{g}^\top\mathbf{Q}_k}{\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}}\right)
\end{aligned}$$

Then using the expressions we derived above for ρ and σ we find that

$$\frac{\sigma-\rho}{\sigma} = 1 - \frac{\rho}{\sigma} = 1 - \frac{n^2}{(n+1)^2}\times\frac{n^2-1}{n^2} = 1 - \frac{n-1}{n+1} = \frac{2}{n+1}$$

Finally, letting

$$\boxed{\mathbf{d} = \frac{-\mathbf{Q}_k\mathbf{g}}{\sqrt{\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}}}} \qquad \text{so that} \qquad \mathbf{d}\mathbf{d}^\top = \frac{\mathbf{Q}_k\mathbf{g}\mathbf{g}^\top\mathbf{Q}_k}{\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}}$$

we get this $\mathbf{Q}$ update.

$$\boxed{\mathbf{Q}_{k+1} = \frac{n^2}{n^2-1}\left(\mathbf{Q}_k - \frac{2}{n+1}\,\mathbf{d}\mathbf{d}^\top\right)}$$

Above we found that $\mathbf{w}^{k+1} = \alpha\mathbf{p}^k = \mathbf{U}^{-\top}(\mathbf{x}^{k+1}-\mathbf{x}^k)$, so using $\alpha = 1/(n+1)$ we get this $\mathbf{x}$ update.

$$\begin{aligned}
\alpha\mathbf{U}^\top\mathbf{p}^k &= \mathbf{x}^{k+1} - \mathbf{x}^k \\
\mathbf{x}^{k+1} &= \mathbf{x}^k + \alpha\mathbf{U}^\top\mathbf{p}^k = \mathbf{x}^k + \alpha\mathbf{U}^\top\left(\frac{-\mathbf{U}\mathbf{g}}{\|\mathbf{U}\mathbf{g}\|}\right) = \mathbf{x}^k + \alpha\,\frac{-\mathbf{Q}_k\mathbf{g}}{\sqrt{\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}}}
\end{aligned}$$

$$\boxed{\mathbf{x}^{k+1} = \mathbf{x}^k + \frac{1}{n+1}\,\mathbf{d}}$$

24.4 Implementing Shor's Algorithm

The boxed updates on the previous page lead to the algorithm below for solving the standard-form nonlinear program

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & f_0(\mathbf{x}) & \\ \text{subject to} & f_i(\mathbf{x}) \;\le\; 0 & i = 1\ldots m. \end{array}$$

$\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{H}} + \mathbf{x}^{\mathrm{L}})$	pick a starting point
$\mathbf{Q}_0 = \operatorname{diag}(\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}})$	pick a starting inverse matrix
for $k = 0\ldots k_{\max}$	do up to $k_{\max}$ iterations
select i	index of a violated constraint, or 0 if $\mathbf{x}^k$ is feasible
$\mathbf{g} = \nabla f_i(\mathbf{x}^k)/\lVert\nabla f_i(\mathbf{x}^k)\rVert$	find normalized gradient of constraint or objective
$\mathbf{d} = -\mathbf{Q}_k\mathbf{g}\Big/\sqrt{\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}}$	find direction vector
$\mathbf{x}^{k+1} = \mathbf{x}^k + \dfrac{1}{n+1}\,\mathbf{d}$	take the step
if$\Big(\left\lVert\mathbf{x}^{k+1} - \mathbf{x}^k\right\rVert < \mathrm{tol}\Big)$ return	test for convergence
$\mathbf{Q}_{k+1} = \dfrac{n^2}{n^2-1}\left(\mathbf{Q}_k - \dfrac{2}{n+1}\mathbf{d}\mathbf{d}^\top\right)$	update the ellipsoid inverse matrix
end	

I implemented this idea in the `ea.m` routine that is listed on the following pages. The routine requires as input $\boxed{1}$ the center point $\mathbf{x}^0$ of $\mathbb{E}_0$ and the starting inverse $\mathbf{Q}_0$ of its ellipsoid matrix. Recall that although the matrix in the definition of $\mathbb{E}_k$ is $\mathbf{Q}_k^{-1}$ we use and update its inverse $\mathbf{Q}_k$, which is initially $\mathbf{Q}_0$. Bounds on the variables could be used to produce `xzero` and `Qzero`, as suggested in the first two algorithm steps above and as implemented in the `eainit.m` routine of §24.3.1. The other input parameters are the number of constraints `m`, an iteration limit `kmax`, a convergence tolerance `tol`, and pointers `fcn` and `grd` to routines that compute the values and gradients of the f_i. In the `ek1.m` and `ek1g.m` routines listed below the parameter `i` is the index of the function whose value or gradient is to be found.

```
function f=ek1(x,i)
  switch(i)
    case 0 % objective
      f=(x(1)-20)^4+(x(2)-12)^4;
    case 1
      f=8*exp((x(1)-12)/9)-x(2)+4;
    case 2
      f=6*(x(1)-12)^2+25*x(2)-600;
    case 3
      f=-x(1)+12;
  end
end
```

```
function g=ek1g(x,i)
  switch(i)
    case 0
      g=[4*(x(1)-20)^3;4*(x(2)-12)^3];
    case 1
      g=[8*exp((x(1)-12)/9)*(1/9);-1];
    case 2
      g=[6*2*(x(1)-12);25];
    case 3
      g=[-1;0];
  end
end
```



```
 1 function [xstar,rc,k,Qstar]=ea(xzero,Qzero,m,kmax,tol,fcn,grd)
 2 % do up to kmax iterations of the ellipsoid algorithm to solve
 3 % minimize fcn(x,0) subject to fcn(x,i) <= 0, i=1..m
 4
 5 % compute constants used in the updates
 6   n=size(xzero,1);
 7   a=1/(n+1);
 8   b=2*a;
 9   c=n^2/(n^2-1);
10
11   x=xzero;
12   Q=Qzero;
13   rc=1;
14   for k=1:kmax
15 %     find a function to use in making the cut
16       icut=0;
17       for i=1:m
18           if(fcn(x,i) > 0)
19              icut=i;
20              break
21           end
22       end
23
24 %     find the gradient and normalize it
25       g=grd(x,icut);
26       ng=0;
27       for j=1:n
28           ng=max(ng,abs(g(j)));
29       end
30       if(ng == 0)             % gradient zero
31          rc=3;
32          break
33       else
34          g=g/ng;
35       end
```

The return parameter `xstar` is the best point found so far, which might be far from optimal if convergence has not yet been achieved. The return code `rc` reports what happened, and `Qstar` is the inverse matrix of the ellipsoid whose center is `xstar`. This routine is serially-reusable so it can be called again, passing `xstar` and `Qstar` for the starting ellipsoid, to continue a solution process that was interrupted because the iteration limit was met.

The first stanza $\boxed{\texttt{5-9}}$ finds the constants used in the update formulas. The second stanza $\boxed{\texttt{11-12}}$ initializes the ellipsoid center and inverse matrix and $\boxed{\texttt{13}}$ sets `rc=1` in anticipation that the iteration limit will be met. Then $\boxed{\texttt{14}}$ begins a loop of up to `kmax` iterations. The first step in each iteration $\boxed{\texttt{15-22}}$ is to find the index `icut` of a violated constraint $\boxed{\texttt{18-21}}$ or, if `x` is feasible, of the objective $\boxed{\texttt{16}}$. If `m` is zero MATLAB does not perform the loop so `icut=0` on every iteration and objective cuts are used to solve the unconstrained problem.

The third stanza $\boxed{\texttt{24-35}}$ finds the gradient $\boxed{\texttt{25}}$ of the function used for the cut and $\boxed{\texttt{26-29,34}}$ normalizes it by its L^∞ norm (this makes the gradient component that is largest in absolute value equal to plus or minus 1). This scaling reduces roundoff error in the calculation of $\mathbf{d}$, but it does not affect the theoretical behavior of the algorithm so the more expensive L^2 norm could be used instead. If $\boxed{\texttt{30}}$ the gradient element largest in absolute value is zero then the gradient is zero and the iterations cannot continue. This can happen even when $\mathbf{x}^k \neq \mathbf{x}^\star$


```
37 %     find the direction in which to move the ellipsoid center
38       gqg=g'*Q*g;
39       if(gqg <= 0)            % ellipsoid matrix not PD
40          rc=2;
41          break
42       else
43          d=-Q*g/sqrt(gqg);
44       end
45
46 %     check for convergence
47       xnew=x+a*d;
48       if(norm(xnew-x) < tol)  % close enough
49          rc=0;
50          break
51       else
52          Qnew=c*(Q-b*d*d');
53       end
54
55 %     update the ellipsoid for the next iteration
56       x=xnew;
57       Q=0.5*(Qnew+Qnew');
58   end
59   xstar=x;                    % done or out of iterations
60   Qstar=Q;
61
62 end
```

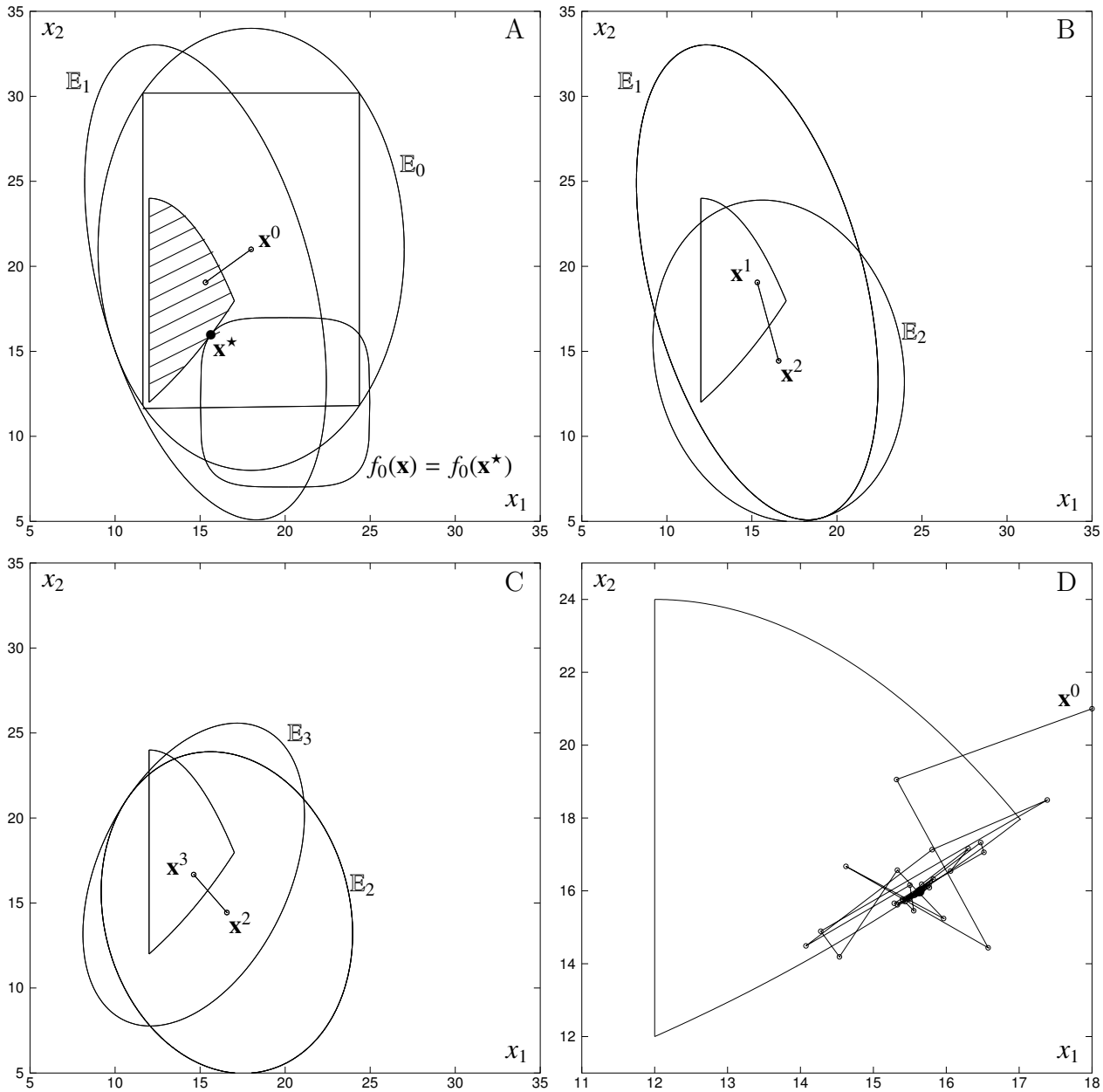
if the function being used for the cut is a constraint that happens to be stationary at `x`. In that case the routine 31-32 resigns with `rc=3`.

Next 38 the normalized gradient `g` is used to find `gqg` = $\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}$. We have assumed that $\mathbf{Q}_0$ is a positive-definite matrix, and in perfect arithmetic the update formula ensures that every $\mathbf{Q}_k$ remains positive definite. However, as the algorithm proceeds the ellipsoids get smaller so the elements of `Q` get smaller, and depending on the problem the ellipsoids can also get long and thin or **aspheric** so that `Q` is badly conditioned. Eventually the resulting roundoff errors make `Q` **numerically non-positive-definite**. so that `gqg` comes out nonpositive and the calculations cannot continue. In that case the routine 40-41 resigns with `rc=2`. Until that happens, `gqg` can be used 43 to compute the direction vector $\mathbf{d}$.

The next iterate $\mathbf{x}^{k+1}$ is found 47 from the $\mathbf{x}$ update formula and 48 the length of the step from $\mathbf{x}^k$ to $\mathbf{x}^{k+1}$ is used to test for convergence. If the step is short enough, the routine 49-50 returns with `rc=0` to signal success. If convergence has not been achieved 52 the $\mathbf{Q}$ update is used to find `Qnew` = $\mathbf{Q}_{k+1}$.

Finally 56-57 the ellipsoid center and inverse matrix are updated and 58 the iterations continue. As the iterations progress and the entries of `Q` become small, roundoff errors can cause it to become slightly unsymmetric, so symmetry is restored 57 by making the new matrix the average of $\mathbf{Q}_{k+1}$ with its transpose [53].

To test `ea.m`, I used it to solve the `ek1` problem one iteration at a time with the results shown on the next page. Panel A shows the feasible set for problem `ek1`, the optimal contour of its objective function, and its optimal point $\mathbf{x}^\star$. The given variable bounds define a box, and ellipsoid $\mathbb{E}_0$ with center $\mathbf{x}^0$ is constructed as the smallest ellipsoid containing the box.



A phase 1 cut is used to construct $\mathbb{E}_1$ as the smallest ellipsoid containing the feasible half of $\mathbb{E}_0$. In Panel B a phase 2 cut has generated ellipsoid $\mathbb{E}_2$ with center $\mathbf{x}^2$, and in panel C another phase 1 cut has generated ellipsoid $\mathbb{E}_3$ with center $\mathbf{x}^3$. Panel D shows, at enlarged scale, the first 40 iterates in the convergence trajectory. The numerical coordinates of the $\mathbf{x}^k$ agree with those tabulated in [3, p320], ending with $\mathbf{x}^{40} = [15.661895, 16.015822]^\top$. This point is not very close to $\mathbf{x}^\star$, but if the algorithm is allowed to use more iterations it gets closer.


```
octave:1> format long
octave:2> xzero=[18;21];
octave:3> Qzero=[81,0;0,169];
octave:4> [xstar,rc,k]=ea(xzero,Qzero,3,200,1e-6,@ek1,@ek1g)
xstar =

   15.6294920320109
   15.9737701208319

rc =  0
k =  159
octave:5> [xstar,rc,k,Qstar]=ea(xzero,Qzero,3,300,1e-16,@ek1,@ek1g)
xstar =

   15.6294908453665
   15.9737685420984

rc =  2
k =  222
Qstar =

  -3.32732478525693e-15  -4.42673506606062e-15
  -4.42673506606062e-15  -5.88941104635132e-15

octave:6> quit
```

With tol = 10^{-6} the convergence criterion is satisfied after `k=159` iterations. With tol = 10^{-16} `gqg` becomes nonpositive at iteration 222, so from the given `xzero` `2>` and `Qzero` `3>` the final `xstar` `5>` is the most accurate solution this algorithm can find.

24.5 Ellipsoid Algorithm Convergence

When we solve `ek1` with `ea.m` each ellipsoid is smaller than its predecessor, $\mathbf{x}^\star$ is inside all of them, and $\|\mathbf{x}^{k+1} - \mathbf{x}^k\| \to 0$ as $k \to \infty$. If we assume that we can do perfect arithmetic (so that, for example, $\mathbf{Q}_k$ never becomes non-positive-definite) then conditions can be established [98, §2.3] [56] that guarantee this desirable behavior. To explain them it will be helpful to restate the standard-form nonlinear program like this.

$$\begin{aligned} &\underset{\mathbf{x}\in\mathbb{X}}{\text{minimize}} \quad f_0(\mathbf{x}) \\ &\text{where} \quad \mathbb{X} = \{\, \mathbf{x} \in \mathbb{R}^n \;\big|\; f_i(\mathbf{x}) \le 0, \quad i = 1 \ldots m \,\} \end{aligned}$$

Then Shor's ellipsoid algorithm is sure to converge if all of the following are true:

- $\mathbf{x}^\star \in E_0$, the optimal point is inside the starting ellipsoid;
- $f_i(\mathbf{x})$ is a convex function for $i = 0 \ldots m$, so that the problem is a convex program;
- $\mathbb{E}_0 \cap \mathbb{X}$ has positive volume relative to $\mathbb{R}^n$, which requires that $\mathbb{X}$ be of full dimension rather than being flat.

The algorithm often works even if the first two conditions are not satisfied (especially of $\mathbb{X}$ is a convex set) but it always fails if $\mathbb{X}$ is not of full dimension. Shor's algorithm finds interior points, so it is not surprising that it depends on $\mathbb{X}$ having an interior relative to $\mathbb{R}^n$. This rules out problems having equality constraints written as opposing inequalities.

When the algorithm converges its speed depends on how fast the ellipsoids shrink. We found in 14.7.2 that the volume of an ellipsoid is proportional to the square root of the determinant of its inverse matrix, and others [73] have found a formula for the ratio $\daleth(n)$ of the volumes of successive ellipsoids in terms of n.

$$\frac{\mathcal{V}(\mathbb{E}_{k+1})}{\mathcal{V}(\mathbb{E}_k)} = \frac{\sqrt{|\mathbf{Q}_{k+1}|}}{\sqrt{|\mathbf{Q}_k|}} \quad = \quad \daleth(n) = \sqrt{\frac{n-1}{n+1}}\left(\frac{n}{\sqrt{n^2-1}}\right)^n$$

The volumes thus decrease in geometric progression with ratio $\daleth(n) < 1$. If each $\mathbb{E}_k$ were a hypersphere of radius r_k then their volumes would be in the ratio r_{k+1}^n / r_k^n and we would have

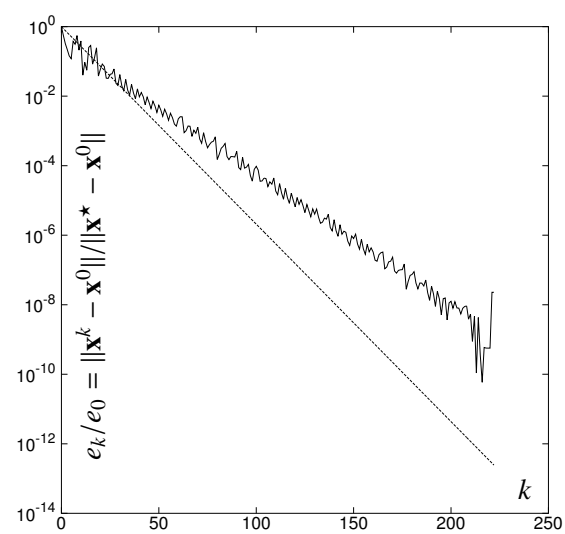
$$r_{k+1}^n = \daleth(n) r_k^n \qquad \text{or} \qquad r_{k+1} = r_k \sqrt[n]{\daleth(n)}.$$

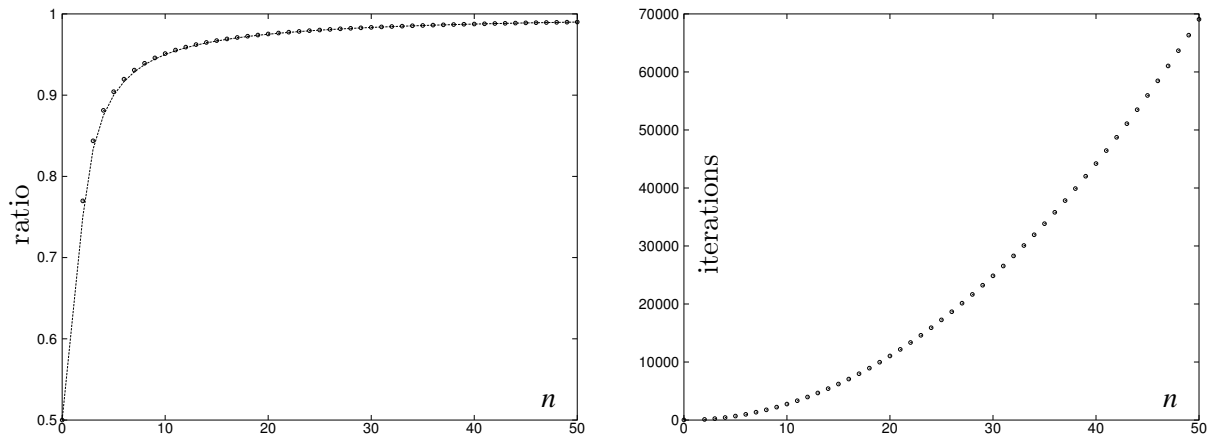
Because $\mathbf{x}^k \in \mathbb{E}_k$ and $\mathbf{x}^\star \in \mathbb{E}_k$, the errors $e_k = \|\mathbf{x}^k - \mathbf{x}^\star\|$ would decrease in geometric progression along with the radii r_k, so that

$$e_k = e_0 \left[\sqrt[n]{\daleth(n)}\right]^k \qquad \text{or} \qquad \frac{e_k}{e_0} = c^k \quad \text{with} \quad c = \sqrt[n]{\daleth(n)} < 1$$

This formula describes linear convergence (see §9.2) and that is the order that is typically observed for Shor's algorithm, but because the $\mathbb{E}_k$ are really not all hyperspheres but tend to become aspheric the convergence constant c is almost always closer to 1 than this analysis predicts.

To study the convergence of `ea.m` I plotted the relative error in $\mathbf{x}^k$ as a function of k for problem `ek1` in the graph to the right. Both curves stop at $k = 222$, when the ellipsoid matrix becomes numerically non-positive-definite. The straight line plots the formula we derived above and the wiggly line shows the observed performance of the algorithm. The actual convergence trajectory is roughly linear, as predicted, but it is not as steep as predicted; the theoretical slope is achieved only initially, because in solving this problem the ellipsoids become progressively more needle-shaped.





The best-case convergence constant c that we found above depends on the **ellipsoid volume reduction ratio**

$$\daleth(n) = \sqrt{\frac{n-1}{n+1}}\left(\frac{n}{\sqrt{n^2-1}}\right)^n \approx 1 - \frac{1}{2n}$$

and hence strongly on the number n of variables in the problem. The graph on the left above shows $\daleth(n)$ as points and the approximation as a solid curve. If n is big then $\daleth$ is close to 1 so the ellipsoid volumes decrease only slowly and the algorithm takes a long time to find a precise answer. The graph on the right above shows as a function of n how many iterations are needed in the best case to reduce the solution error to 0.000001 of its original value.

24.6 Recentering

When Shor's algorithm fails to find an answer as precise as we would like to a problem it should be able to solve, the reason is almost always that $\mathbf{Q}_k$ has become numerically non-positive-definite because repeated cuts have made $\mathbb{E}_k$ highly aspheric. When this happens a more precise solution can often be obtained by restarting the algorithm using new bounds centered on the best point found so far. This **recentering strategy** also has the virtue of gradually tightening bounds on the coordinates of $\mathbf{x}^\star$; that provides a measure of the precision to which $\mathbf{x}^\star$ is known, which is useful in many practical applications. To implement the idea it is necessary to keep the record point and record value (see §9.1). This is itself a worthwhile improvement to the basic algorithm in view of the wild excursions of its iterates.

The scheme is outlined in the flowchart on the next page. This is the algorithm we implemented in `ea.m` except that it keeps $\mathbf{x}^r$ and includes the blocks enclosed by the dashed box. Now, instead of giving up when $\mathbf{g}^\top\mathbf{Q}_k\mathbf{g} \le 0$ we recenter. Since this shrinks the bounds we can use their separation as the convergence criterion, so if $\|\mathbf{x}^{\mathrm{H}} - \mathbf{x}^{\mathrm{L}}\| <$ `tol` this algorithm

reports success and stops. Recentering is not possible until a feasible point has been found, so if the starting ellipsoid becomes non-positive-definite before that happens the problem is reported to be infeasible. If recentering is possible, the distance w_j between the current bounds x_j^{L} and x_j^{H} in each coordinate direction is reduced by the factor $\mathrm{ɔ}$ and this new width is used to center the bounds on x_j^r. Then we find the smallest ellipsoid containing the reduced bounds and replace $\mathbf{x}^k$ and the defective $\mathbf{Q}_k$ by the center and inverse matrix of the new ellipsoid.

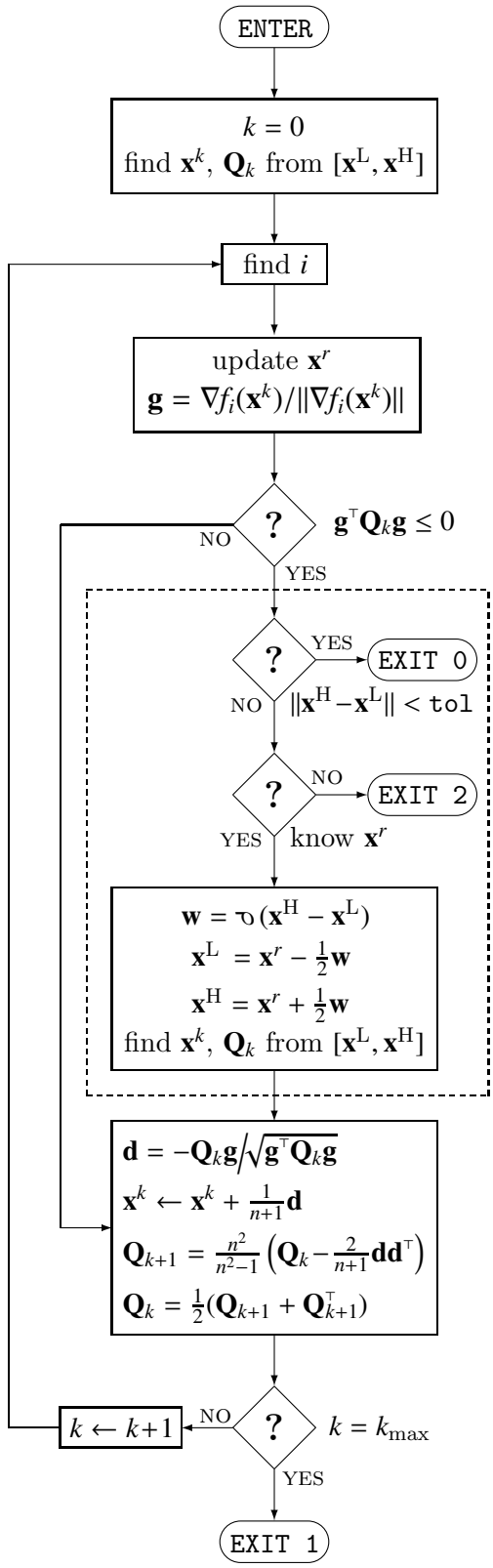
I implemented the algorithm in the `wander.m` routine that is listed on the following pages. Now in place of $\mathbf{x}^0$ and $\mathbf{Q}_0$ the starting bounds `xl` and `xh` are input parameters, and instead of $\mathbf{x}^\star$ and $\mathbf{Q}^\star$ the return parameters include the final bounds `xlr` and `xhr` bracketing the record point `xr`. Unlike `ea.m` this routine is *not* serially reusable.

Like `ea.m` this code begins [5-9] by computing the constants used in the ellipsoid update formulas. Then [10] it initializes the factor `shr` that will be used in shrinking the variable bounds. I set this parameter to

$$\mathrm{ɔ} = \tfrac{1}{10}\left(1 - \frac{1}{2n}\right)$$

but some other fraction of $\mathrm{ʃ}(n)$ or its approximation might work better in a particular case. Next [12-13] the `eainit.m` routine of §24.3.1 is used to find the starting values of `x` and `Q`. The ellipsoid algorithm iterations begin [15-25] as in `ea.m`, but then a stanza [27-34] is interposed to remember the objective value `fr` and iterate `xr` at the feasible point having the lowest objective value found so far. The next stanza [36-47] finds, just as in `ea.m`, the normalized gradient to use in making the cut.

The flowchart blocks in the dashed box are implemented by the next stanza [49-72]. We compute and test `gqg` [50-51] as in `ea.m` and if it is still positive [68-72] update the ellipsoid as usual. Otherwise [52] the difference of the bounds is tested and if it is small enough [53-54,78-79] the routine returns the



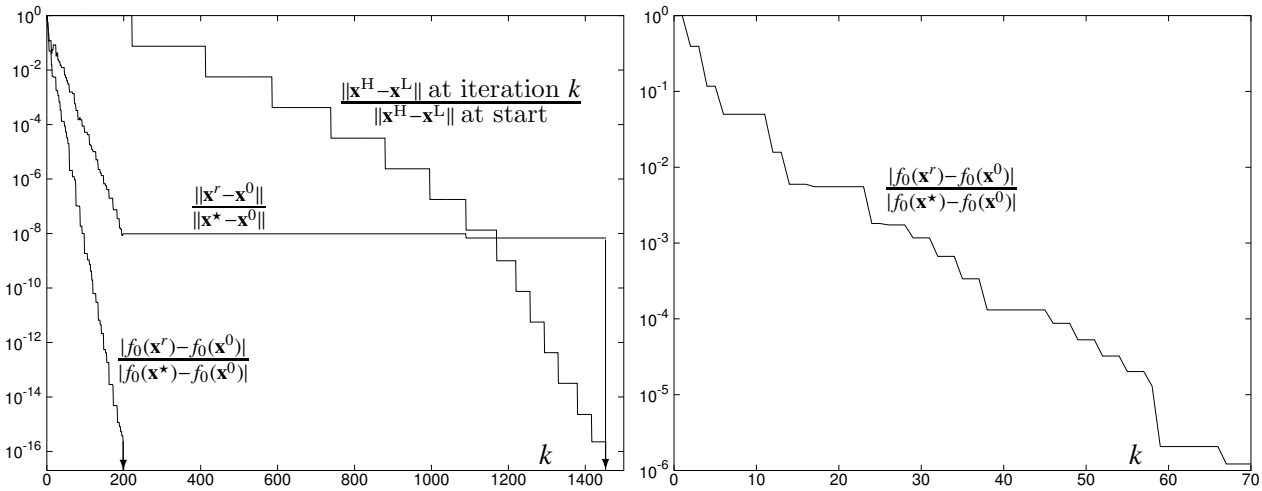

```
 1 function [xlr,xr,xhr,rc,k]=wander(xl,xh,m,kmax,tol,fcn,grd)
 2 % do up to kmax iterations of the recentering ellipsoid algorithm
 3 % to minimize fcn(x,0) subject to fcn(x,i) <= 0, i=1..m
 4
 5 % compute constants used in the updates
 6   n=size(xl,1);
 7   a=1/(n+1);
 8   b=2*a;
 9   c=n^2/(n^2-1);
10   shr=0.1*(1-1/(2*n));
11
12 % initialize the ellipsoid center and matrix
13   [x,Q]=eainit(xl,xh);
14
15   rc=1;
16   fr=realmax;
17   for k=1:kmax
18 %     find a function to use in making the cut
19       icut=0;
20       for i=1:m
21           if(fcn(x,i) > 0)
22              icut=i;
23              break
24           end
25       end
26
27 %     update the record point
28       if(icut == 0)
29          fobj=fcn(x,0);
30          if(fobj < fr)
31             fr=fobj;
32             xr=x;
33          end
34       end
35
36 %     find the gradient and normalize it
37       g=grd(x,icut);
38       ng=0;
39       for j=1:n
40           ng=max(ng,abs(g(j)));
41       end
42       if(ng == 0)
43          rc=3;
44          break
45       else
46          g=g/ng;
47       end
```

current record point and bounds along with `rc=0` to signal success. If 56 no feasible point has yet been found, the routine returns 64-65 with the starting bounds 78-79 and `rc=2` to signal infeasibility. Otherwise 57-62 recentering is done before 74-77 the iterations continue.

I used `wander.m` to solve the `ek1` problem in the Octave session on the next page. Setting `tol` $= 10^{-13}$ produced bounds equal to the record point, which yields the catalog optimal objective value. In solving many problems `wander.m` can find `xl` = `xr` = `xh` = $\mathbf{x}^\star$ to machine precision, though at the cost of many iterations. This solution 5> took about half a second on a 1 GHz processor, but problems having many variables run for much longer. When implemented in FORTRAN the algorithm is useful for problems having n up to about 50 [52].


```
%     recenter or take the next step
      gqg=g'*Q*g;
      if(gqg <= 0)                        % is Q non-pd?
         if(norm(xh-xl) < tol)            % yes; xr close enough?
            rc=0;                         % yes; flag convergence
            break                         % and return
         else                             % not close enough
            if(fr < realmax)              % know a record point?
               for j=1:n                  % yes
                   w=shr*(xh(j)-xl(j));   % new bound width
                   xl(j)=xr(j)-0.5*w;     % new lower bound
                   xh(j)=xr(j)+0.5*w;     % new upper bound
               end                        % bounds now recentered
               [xnew,Qnew]=eainit(xl,xh); % find a new ellipsoid
            else                          % no record point
               rc=2;                      % can't recenter
               break                      % so give up
            end
         end
      else                                % Q is still pd
        d=-Q*g/sqrt(gqg);                 % find direction vector
        xnew=x+a*d;                       % find next center
        Qnew=c*(Q-b*d*d');                % and ellipsoid matrix
      end

%     update the ellipsoid for the next iteration
      x=xnew;
      Q=0.5*(Qnew+Qnew');
  end
  xlr=xl;
  xhr=xh;

end
```

```
octave;1> format long
octave:2> xl=[18-9/sqrt(2);21-13/sqrt(2)];
octave:3> xh=[18+9/sqrt(2);21+13/sqrt(2)];
octave:4> [xlr,xr,xhr,rc,k]=wander(xl,xh,3,2000,1e-13,@ek1,@ek1g)
xlr =

   15.6294909238917
   15.9737686465698

xr =

   15.6294909238917
   15.9737686465698

xhr =

   15.6294909238917
   15.9737686465699

rc = 0
k =  1417
octave:5> tic;[xlr,xr,xhr,rc,k]=wander(xl,xh,3,2000,1e-13,@ek1,@ek1g);toc
Elapsed time is 0.47477 seconds.
octave:6> fr=ek1(xr,0)
fr =  614.212097203404
```

To study the convergence of `wander.m` I plotted in the left graph above the relative error in `fr`, the relative error in `xr`, and the relative width of the bounds, as functions of `k`. The relative errors in `fr` and `xr` both decrease linearly until about `k=200`, when $f_0(\mathbf{x}^k) = f_0(\mathbf{x}^\star)$ and the relative error in `fr` plunges to $-\infty$. Shortly after that $\mathbf{Q}_k$ becomes non-positive-definite for the first time and a recentering occurs, narrowing the bounds. No better point is found until the 7th recentering, about `k=1100`, when the error in `xr` decreases slightly. It is only after the 15th recentering that iteration 1417 produces $\mathbf{x}^k = \mathbf{x}^\star$ and the relative error in `xr` plunges to $-\infty$. Notice that as the recentered ellipsoids get smaller the interval between resets decreases. In other problems the error curve for `fr` also plateaus so that the optimal objective value is attained only after some recenterings. It can also happen that $\mathbf{x}^k$ moves outside of the original variable bounds; this is what makes it possible for the algorithm to sometimes find $\mathbf{x}^\star$ even if the original bounds do not contain it. In that case recentering can produce new bounds that are not contained within the starting bounds.

The sudden decrease of relative errors in both `fr` and `xr` that is evident at the very beginning of the curves in the left graph is typical of the algorithm. To make this phenomenon easier to see I have enlarged that part of the `fr` convergence trajectory in the graph on the right. Thanks to this behavior the ellipsoid algorithm might find a record point that is a good approximate solution to a nonlinear program more quickly than a higher-order method (see the example in §26.3).

24.7 Shah's Algorithm for Equality Constraints

As I mentioned in §24.5, Shor's algorithm always fails if the feasible set is not of full dimension, so it can't be used to solve problems that have equality constraints. If the equality constraints are linear, however, a different ellipsoid algorithm can be devised that keeps every direction vector $\mathbf{d}^k$, and hence every iterate $\mathbf{x}^k$, in the flat of the equalities.

Suppose that the equality constraints of the nonlinear program are $\mathbf{Ax} = \mathbf{b}$ and that at iteration k of the algorithm $\mathbb{E}_k$ has center $\mathbf{x}^k \in \mathbb{F} = \{ \mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} = \mathbf{b} \}$ and ellipsoid inverse matrix $\mathbf{Q}_k$. If the normalized gradient $\mathbf{g}$ is used to make a center cut and the cutting hyperplane is translated parallel to itself until it is tangent to $\mathbb{E}_k$ at $\mathbf{x}^k + \mathbf{d}^k = \mathbf{p}^k \in \mathbb{F}$, then the vector $\mathbf{d}^k$ is optimal for

$$\begin{array}{lll} \underset{\mathbf{d}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{g}^\top(\mathbf{x}^k + \mathbf{d}) & \\ \text{subject to} & \left((\mathbf{x}^k + \mathbf{d}) - \mathbf{x}^k\right)^\top \mathbf{Q}_k^{-1} \left((\mathbf{x}^k + \mathbf{d}) - \mathbf{x}^k\right) = 1 \quad \text{or} & \mathbf{d}^\top \mathbf{Q}_k^{-1} \mathbf{d} = 1 \\ & \mathbf{A}\left(\mathbf{x}^k + \mathbf{d}\right) = \mathbf{b} \quad \text{or} & \mathbf{Ad} = \mathbf{0}. \end{array}$$

Solving this problem by the Lagrange method yields [141, §2.2]

$$\mathbf{d} = -\frac{\left(\mathbf{Q} - \mathbf{QA}^\top(\mathbf{AQA}^\top)^{-1}\mathbf{AQ}\right)\mathbf{g}}{\sqrt{\mathbf{g}^\top\left(\mathbf{Q} - \mathbf{QA}^\top(\mathbf{AQA}^\top)^{-1}\mathbf{AQ}\right)\mathbf{g}}}.$$

If this formula is used for the direction vector in the ellipsoid algorithm, then we have **Shah's algorithm**. Shah also solved some problems having nonlinear equality constraints [142] by linearizing them at each $\mathbf{x}^k$. If that approach is accompanied by a feasibility-restoration step it resembles the generalized reduced-gradient algorithm of §23.1.2, but using the ellipsoid algorithm rather than steepest descent to minimize $f_0(\mathbf{x})$ on the flat allows the algorithm to solve problems that have both equality and inequality constraints.

24.8 Other Variants

The most obvious refinement of Shor's algorithm is to use **deep cuts** [56]. In the graphical solution of §24.2 we constructed $\mathbb{H}_0$ to support the contour $f_2(\mathbf{x}) = f_2(\mathbf{x}^0)$ of the violated constraint at the center $\mathbf{x}^0$ of $\mathbb{E}_0$. If we had instead searched the line between $\mathbf{x}^0$ and $\mathbf{p}^0$ for its intersection with the contour $f_2(\mathbf{x}) = 0$, we could have constructed $\mathbb{H}_k$ tangent to the feasible set at that point. It is also possible to make deep optimality cuts [98, p43-45]. Using a deep cut throws away more of the old ellipsoid and thereby speeds the reduction of ellipsoid volume. In practice, although some ways of generating deep cuts slightly improve on the efficiency of the center-cut version [47] they make the algorithm more complicated and do nothing to address its fundamentally linear convergence. Using deep cuts also makes the algorithm less likely to solve problems in which some or all of the f_i are nonconvex.

An even faster reduction in the ellipsoid volumes can result from using **wedge cuts** [51]. If two constraints are violated we can construct a hyperplane supporting each and find the smallest ellipsoid $\mathbb{E}_{k+1}$ enclosing the wedge that they cut out of $\mathbb{E}_k$. This strategy also reduces the robustness of the algorithm, and its rank–2 updates are significantly more complex than Shor's rank–1 updates. Because of the extra calculations that wedge cuts require, they, like deep cuts, turn out not to provide much improvement in efficiency.

If as Shor's algorithm approaches $\mathbf{x}^\star$ it could guess that some inequalities will be slack at optimality, it could save work by no longer evaluating those functions in the search for a violated constraint. If it could guess that some inequalities will be tight at optimality, it could treat them as equalities in the manner of Shah's algorithm, which effectively reduces the dimensionality of the problem and thus accelerates convergence [141, §2.7]. An active set strategy can be contrived that does both of these things [137] based on statistics about which of the constraints were found to be violated during the previous iterations of the algorithm. In solving a problem with many constraints, the resulting convergence can be superlinear as constraints are ignored or made equalities.

The ability of the ellipsoid algorithm to identify the feasible set and find an approximate solution early in its iterations suggests that it might be used to provide a good starting point and active set estimate (a hot start) for algorithms that are less robust but have quadratic convergence near the optimal point. When a second-order method cannot continue, as for example when a sequential quadratic programming algorithm generates an infeasible subproblem, the ellipsoid algorithm can be invoked to refine the solution or move to an $\mathbf{x}^k$ from which the more sophisticated algorithm can resume. These ideas have been used to construct effective **hybrid algorithms** that combine SQP with the ellipsoid algorithm [128].

24.9 Summary

As we have seen, ellipsoid algorithms have only first order convergence, with a constant that quickly approaches 1 as n increases, so they are too slow for problems having more than a few dozen variables. For this reason they are certainly *not* practical, as people once hoped they might be, for solving linear programming problems [37]. However, they do have some endearing properties when they are used to solve *non*linear programs.

Although ellipsoid algorithms are sure to converge only if the $f_i(\mathbf{x})$ are all convex functions, in practice they are much more likely to solve nonconvex programs than are other methods of constrained optimization [52]. They are also relatively insensitive to imprecisions in the function and gradient values [99]. This is an important advantage when those values must be approximated by simulation and in **on-line applications** such as feedback control, when they are the result of physical measurements. The robustness of ellipsoid methods makes them ideal for small, highly-nonconvex type–2 problems such as parameter estimation (see §8.5) and semi-infinite formulations of robot path planning [115].

Ellipsoid algorithms often find a good approximate solution very quickly, and they are capable of finding very precise solutions. The record points they return are, modulo round-off, strictly feasible, in contrast to the approximately feasible solutions produced by other methods. When recentering is used, the optimal point is accompanied by a useful interval of uncertainty in each coordinate direction.

Thus, despite their quirks and because of them, ellipsoid algorithms deserve a place in our catalog of methods for nonlinear optimization.

24.10 Exercises

24.10.1 [E] What is a *space confinement* algorithm, and how does it work? Name two space confinement algorithms.

24.10.2 [E] Describe in words the basic idea of Shor's ellipsoid algorithm.

24.10.3 [E] In Shor's algorithm, (a) what is a *center cut*? A *feasibility cut*? An *optimality cut*? (b) How is a *phase 1* iteration different from a *phase 2* iteration? What must be true about $\mathbf{x}^k$ for the next step in the algorithm to be a phase 1 iteration? For it to be a phase 2 iteration? (c) In what pattern do phase 1 and phase 2 iterations typically occur?

24.10.4 [H] The nonlinear program [3, Exercise 9.50]

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & 2x_1^2 - x_1 + x_2^2 & \\ \text{subject to} & 8x_1 + 8x_2 & \leq \quad 1 \end{array}$$

has $\mathbf{x}^\star \in \mathbb{E}_0 = \{\, \mathbf{x} \in \mathbb{R}^2 \mid x_1^2 + x_2^2 \leq 1 \,\}$. (a) Perform the first step of Shor's algorithm graphically, showing $\mathbf{x}^0$, $\mathbb{E}_0$, $\mathbb{H}_0$, $\mathbf{p}^0$, $\mathbf{x}^1$, and an approximate sketch of $\mathbb{E}_1$. (b) Perform the second step graphically.

24.10.5 [H] The following equation describes an ellipse.

$$\frac{(x_1 - 1)^2}{9} + \frac{(x_2 - 2)^2}{16} = 1$$

(a) Graph the ellipse. (b) Find a vector $\mathbf{x}^0$ and positive-definite symmetric matrix $\mathbf{Q}_0$ so that the ellipse is described by

$$(\mathbf{x} - \mathbf{x}^0)^\top \mathbf{Q}_0^{-1} (\mathbf{x} - \mathbf{x}^0) = 1.$$

24.10.6 [E] Why in discussing the ellipsoid algorithm do we call the matrix that defines an ellipsoid $\mathbf{Q}^{-1}$ rather than $\mathbf{Q}$? Does Shor's ellipsoid algorithm manipulate $\mathbf{Q}$, or $\mathbf{Q}^{-1}$?

24.10.7 [H] In §24.3.1, I claim that $\mathbb{E}_0$ must be a right ellipsoid if it is to touch all the corners of the box that is formed by the variable bounds. (a) Explain why that is true. (b) How did we find the *smallest* ellipsoid touching all the corners? (b) If the ellipsoid $\mathbb{E}_0 = \left\{ \mathbf{x} \;\middle|\; (\mathbf{x} - \mathbf{x}^0)^\top \mathbf{Q}_0^{-1} (\mathbf{x} - \mathbf{x}^0) = 1 \right\}$, what formula can be used to find $\mathbf{Q}_0$ from the bounds on the variables? (c) What routine can be used to compute `xzero` and `Qzero`? (d) If `xzero` and `Qzero` define an ellipse, how can the `ellipse.m` routine of §14.7.3 be used to draw the ellipse?

24.10.8 [H] If `eainit.m` is used to find the center and inverse matrix defining an ellipsoid and returns the values below, what must have been the bounds `xh` and `xl` on the variables?

$$\mathbf{x}^0 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad \mathbf{Q}_0 = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 30 \end{bmatrix}$$

24.10.9 [E] What does it mean to say that a hyperplane *supports* the contour of a function? How can such a supporting hyperplane be described algebraically?

24.10.10 [E] What is the *unit normal* to a hyperplane? Why is it possible to describe the hyperplane algebraically using its *unit* normal rather than its normal vector? What happens to the hyperplane defined by $\mathbf{g}^\top\mathbf{x} = \kappa$ if κ is changed?

24.10.11 [E] Explain how to use the `hplane.m` routine of §24.3.2 to plot a hyperplane.

24.10.12 [P] Constraint hyperplanes are important in the geometry of linear programming so §3 discusses them in some detail, but it does not use the §24.3.2 definition of a hyperplane or even mention the gradient of a linear function. (a) Show that the hyperplane corresponding to the constraint $x_1+2x_2 \le 4$ can also be described as $\mathbb{H} = \{\, \mathbf{x} \in \mathbb{R}^2 \mid [1,2]^\top(\mathbf{x}-[0,2]^\top) = 0 \,\}$. (b) Write a MATLAB program that uses `hplane.m` to draw the hyperplane.

24.10.13 [H] In §24.3.3 we transformed $\mathbb{E}_k$ to $\mathbf{w}$–space, where it becomes a hypersphere of radius 1 centered at the origin. (a) Explain in detail how this transformation was accomplished. (b) Explain what happens to the hyperplane $\mathbb{H}_k$ under this transformation. (c) Explain what happens to the next ellipsoid $\mathbb{E}_{k+1}$ under this transformation.

24.10.14 [H] In §24.3.3 the geometry of the update in $\mathbf{w}$–space allowed us to write down a formula for the point $\mathbf{p}^k$. Explain how.

24.10.15 [H] In §24.3.3, we transformed $\mathbb{E}_k$, $\mathbb{H}_k$, and $\mathbb{E}_{k+1}$ to $\mathbf{z}$–space, where $\mathbb{E}_{k+1}$ is a right ellipsoid. (a) Explain in detail how this transformation was accomplished. (b) If the eigenvalues of $\mathbf{G}$ are ρ and σ, why are the axis half-lengths of $\mathbb{E}_{k+1}$ given by $\sqrt{\rho}$ and $\sqrt{\sigma}$? How are the (unnormalized) eigenvectors of $\mathbf{G}$ and $\mathbf{G}^{-1}$ related? (c) Explain how we found ρ and σ as functions of α. (d) Explain how we found the value of α that minimizes the volume of $\mathbb{E}_{k+1}$. Why must α be less than $\frac{1}{2}$ if $n > 1$?

24.10.16 [H] In §24.3.3 I claimed that "Many ellipsoids $\mathbb{E}_{k+1}$ can be constructed passing through $\mathbf{p}^k$ and $\mathbb{E}_k \cap \mathbb{H}_k$. Each can be characterized by the eigenvalues ρ and σ..." Explain precisely how $\mathbb{E}_{k+1}$ is characterized by ρ and σ.

24.10.17 [E] Write down the updates for finding $\mathbf{Q}_{k+1}$ and $\mathbf{x}^{k+1}$ from $\mathbf{Q}_k$, $\mathbf{x}^k$, and $\mathbf{g}$ in Shor's ellipsoid algorithm.

24.10.18 [P] In §24.3.3 three graphs are used to explain the steps in the derivation. Write a MATLAB program that reproduces these graphs.

24.10.19 [H] Shor's algorithm moves the hyperplane $\mathbf{g}^\top\mathbf{x} = \mathbf{g}^\top\mathbf{x}^k$ parallel to itself until it is tangent to $\mathbb{E}_k$ at $\mathbf{p}^k$, so the equation of the tangent hyperplane is $\mathbf{g}^\top\mathbf{x} = \mathbf{g}^\top\mathbf{p}^k$. The point $\mathbf{p}^k$ can therefore be found as the optimal point of this nonlinear program.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{g}^\top\mathbf{x} & \\ \text{subject to} & (\mathbf{x}-\mathbf{x}^k)^\top\mathbf{Q}_k^{-1}(\mathbf{x}-\mathbf{x}^k) & = \quad 1 \end{array}$$

Use the Lagrange method to show that $\mathbf{p}^k = \mathbf{x}^k - \mathbf{Q}_k\mathbf{g}/\sqrt{\mathbf{g}^\top\mathbf{Q}_k\mathbf{g}}$.

24.10.20 [H] Show that the function $\delta(\alpha)$ of §24.3.3 is convex on the interval $\alpha \in [0, \frac{1}{2})$.

24.10.21 [H] This optimization problem [3, Exercise 9.51] is a convex program.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & (x_1-2)^2 + x_2^2 & \\ \text{subject to} & x_1^2 + x_2^2 & \leq \ 1 \end{array}$$

(a) Find $\mathbf{x}^\star$. (b) Suppose that Shor's algorithm is used to solve the problem, with the circle defined by the constraint as $\mathbb{E}_0$. Find a formula giving the first component of iterate $\mathbf{x}^k$ as a function of k.

24.10.22 [H] Shor's algorithm is easy to describe as a rank-one update to $\mathbf{Q}$, but it can also be implemented by updating $\mathbf{Q}^{-1}$. (a) Show that if $\mathbf{A}_{k+1} = \mathbf{A}_k + \mathbf{v}\mathbf{v}^\top$, then $\mathbf{A}_{k+1}^{-1}$ is not necessarily a rank-one update of $\mathbf{A}_k^{-1}$. (b) Use the Sherman-Morrison-Woodbury formula of §13.4.4 to derive an update to $\mathbf{Q}_k^{-1}$ that yields $\mathbf{Q}_{k+1}^{-1}$.

24.10.23 [E] Outline the steps in Shor's algorithm. What sort of nonlinear program can it solve? How does the ellipsoid matrix $\mathbf{Q}_k$ that the algorithm manipulates enter into the definition of the ellipsoid $\mathbb{E}_k$? What routine can be used to find `xzero` and `Qzero` from bounds on the variables?

24.10.24 [E] The `ek1.m` and `ek1g.m` routines are listed in §24.4. What does `g=ek1g(x,2)` return?

24.10.25 [E] The `ea.m` routine is listed in §24.4. (a) How are the variables `a`, `b`, and `c` calculated by that code [7–9] related to the variables α, σ, and ρ that we used to derive the update formulas in §24.3.3? (b) In the code, what is the meaning of the variable `icut`? What is its value if `m=0`? (c) How does the code normalize each gradient vector? (c) How is convergence judged to have occurred? (d) Why does the code [57] update `Q` to the average of `Qnew` and its transpose? Is the result always symmetric even if `Q` is not? (e) Why, after computing `xnew` [47] and finding [48] that it is close enough to `x`, does the routine return `x` [59] as the optimal point rather than `xnew`? (f) What are the return parameters from the routine if the iteration limit is met before convergence is achieved?

24.10.26 [E] Describe the input and output parameters of `ea.m`. List the possible return codes and explain what they mean. How can you tell whether the `xstar` that is returned satisfied the convergence criterion?

24.10.27 [P] Show how `ea.m` can be called repeatedly to continue a solution process that was interrupted because the iteration limit was met.

24.10.28 [P] Can `ea.m` be used to solve an unconstrained nonlinear program? If not, explain why not. If so, use it to solve the `rb` problem of §9.1.

24.10.29 [P] In each iteration of Shor's algorithm, `ea.m` begins the search for a violated constraint from `i=1`. This can result in the phase 1 cuts favoring one or a few constraints having low indices. The ellipsoids are less likely to become long and thin if the phase 1 cuts

are more evenly distributed over all of the constraints. Revise the code so that the search for a violated constraint in each iteration begins with the next constraint after the one that was most recently used for a phase 1 cut. In this **constraint rotation scheme** the constraint after $i = m$ is $i = 1$. How does the solution to `ek1` found by your revised code compare to that found by the original version?

24.10.30 [P] The `ea.m` implementation of Shor's algorithm fails if the violated constraint chosen for a cut happens to have a zero gradient at $\mathbf{x}^k$. (a) Explain why the code must resign in that case. (b) Does this indicate that there is something wrong with the nonlinear program? Construct an example to illustrate the phenomenon. (c) Revise the code so that if a violated constraint has a zero gradient the search continues in hopes of finding a violated constraint that does *not* have a zero gradient at $\mathbf{x}^k$. Your code should resign only if *every* constraint that is violated at $\mathbf{x}^k$ has a zero gradient. (d) Test your code on the example you devised and show that it works while the original version of `ea.m` fails with `rc=3`.

24.10.31 [P] The `ea.m` implementation of Shor's algorithm normalizes `g` by dividing each element by the absolute value of its absolutely largest element. (a) Why is it necessary to perform *any* normalization of the gradient vector? (b) Revise the code to divide `g` by its Euclidean length instead. (c) Compare the behavior of your code to that of the original `ea.m`. Does using the L^2 norm to normalize g result in better performance? Does it use more CPU time?

24.10.32 [P] In the `ea.m` implementation of Shor's algorithm, why does the quantity `gqg` approach zero as $\mathbf{x}^k \to \mathbf{x}^\star$? Why might `Q` become ill-conditioned? To illustrate your explanation, print the numerical values of relevant quantities in the code as the solution to a problem is approached.

24.10.33 [E] Give a qualitative description of the convergence trajectory of `ea.m` when it is used to solve the `ek1` problem.

24.10.34 [P] Use `ea.m` to solve the following inequality-constrained nonlinear programs: (a) the `arch2` problem of §16.0; (b) the `arch4` problem of §16.2; (c) the `moon` problem of §16.3; (d) the `cq1` problem of §16.7; (e) the `cq3` problem of §16.7. In each case explain how you chose $\mathbb{E}_0$ and, if the algorithm is unsuccessful, why it fails.

24.10.35 [E] Is the ellipsoid algorithm a descent method? Explain.

24.10.36 [E] State the conditions that must be satisfied to ensure that Shor's algorithm will converge. Might the algorithm work even if these conditions are not satisfied?

24.10.37 [P] The convex set $\mathbb{C}$ of §16.6 is the intersection of two nonconvex inequality constraints. (a) Is Shor's algorithm sure to be able to solve a nonlinear program having these constraints? (b) Apply Shor's algorithm to the `nset` problem of §16.10. Is it successful in finding the optimal point?

24.10.38 [P] If Shor's algorithm is applied to a nonconvex problem it can converge to a point that is not a minimizing point, as shown by the following example [3, Exercise 9.55].

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & (x_1-15)^2+x_2^2 & & \\ \text{subject to} & x_1^2+x_2^2 & \geq & 25 \\ & (x_1-3)^2+x_2^2 & \leq & 25 \end{array} \qquad \text{with} \qquad \mathbb{E}_0 = \left\{ \mathbf{x}\in\mathbb{R}^2 \;\middle|\; \frac{(x_1+1)^2}{100}+\frac{x_2^2}{25} \leq 1 \right\}$$

(a) Solve the problem graphically. (b) Verify graphically that $\mathbf{x}^\star \in \mathbb{E}_0$. (c) Perform the first iteration of the algorithm graphically. Is $\mathbf{x}^\star$ in the $\mathbb{E}_1$ you have sketched? (d) Use the update formulas to find $\mathbf{Q}_1$ and show analytically that $\mathbf{x}^\star \notin \mathbb{E}_1$. (e) To what point does `ea.m` converge when it is applied to this problem? (f) Can you find an $\mathbb{E}_0$ from which `ea.m` converges to $\mathbf{x}^\star$?

24.10.39 [H] What does it mean to say that a set has positive volume *relative to* $\mathbb{R}^n$? Give an example of a set that has positive volume in $\mathbb{R}^2$, and show that it has zero volume in $\mathbb{R}^3$.

24.10.40 [E] Can Shor's algorithm solve a problem in which an equality constraint is written as opposing inequalities?

24.10.41 [E] What is $\daleth(n)$, the ratio of the volumes of successive ellipsoids in Shor's algorithm? This formula has a simple approximation that is quite accurate. What is it?

24.10.42 [H] Show analytically that $\lim_{n\to\infty} \daleth(n) = 1 - \frac{1}{2n}$. Hint: $\lim_{y\to\infty}(1+1/y)^y = e$.

24.10.43 [E] Shor's algorithm has linear convergence. Explain how the best-case relative error e_k/e_0 after iteration k depends on k and on the number of variables n. Why does this theoretical result typically underestimate the observed convergence constant?

24.10.44 [P] The **asphericity** of an ellipse is the ratio of its longest axis to its shortest axis. Write a MATLAB program based on `ea.m` that computes the asphericity of each $\mathbb{E}_k$ generated in solving the `ek1` problem with Shor's algorithm, and plots that number as a function of `k`.

24.10.45 [H] The `ea.m` implementation of Shor's algorithm uses $\|\mathbf{x}^{k+1}-\mathbf{x}^k\|_2$ as the criterion for deciding whether convergence has been achieved. Suggest two different measures of solution error that might be used instead.

24.10.46 [E] Sometimes Shor's algorithm stops before finding an answer as precise as we would like, even though the conditions for convergence given in §24.5 are satisfied. When this happens, what is the usual reason? What can be done to find a more precise answer?

24.10.47 [E] Explain in words the *recentering strategy* described in §24.6. What are its advantages? Why does it require the keeping of a record point?

24.10.48 [E] If the recentering algorithm of §24.6 takes the error exit `2`, what must have happened during the calculations? What does it mean about the problem?

24.10.49 [H] Is `wander.m` serially reusable? If yes, present computational evidence to prove your claim; if not, explain why it is not.

24.10.50 [E] What role is played in `wander.m` by the variable `shr`? What value does `shr` have if $n = 2$? What convergence criterion does the routine use? List its possible return codes and their meanings.

24.10.51 [E] Explain how `wander.m` keeps the record value and record point. What are the meanings of its input and return variables?

24.10.52 [E] In MATLAB, how can you find out the elapsed time used by a calculation?

24.10.53 [H] Is it possible in the ellipsoid algorithm for $\mathbf{x}^k$ to move outside of the starting bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$? How far can it go?

24.10.54 [P] Use `wander.m` to solve the following inequality-constrained nonlinear programs: (a) the `arch2` problem of §16.0; (b) the `arch4` problem of §16.2; (c) the `moon` problem of §16.3; (d) the `cq1` problem of §16.7; (e) the `cq3` problem of §16.7. In each case explain how you chose the starting bounds and, if the algorithm is unsuccessful, why it fails.

24.10.55 [E] Describe in words how Shah's algorithm works. What is its purpose?

24.10.56 [H] Derive the formula for the direction $\mathbf{d}$ in Shah's algorithm.

24.10.57 [P] Write an implementation of Shah's algorithm that solves problems having (a) both inequality constraints and linear equality constraints; (b) both inequality constraints and nonlinear equality constraints. To restore feasibility use Newton's method for systems as in §23.1.2.

24.10.58 [E] Explain the following refinements of Shor's algorithm, and describe their benefits and drawbacks: (a) using deep cuts; (b) using wedge cuts; (c) using an active set strategy. (d) Could these refinements also be applied to the recentering algorithm of §24.6?

24.10.59 [E] What does it mean to provide a *hot start* for an algorithm? What is a *hybrid algorithm*?

24.10.60 [H] The center-cut ellipsoid algorithm is sometimes described as "bisection in n dimensions." (a) Show that the bisection line search can be regarded as an application of Shor's algorithm when $n = 1$. (b) What does `ea.m` do if $n = 1$? Revise the code to perform bisection if $n = 1$.

24.10.61 [E] Summarize the advantages and drawbacks of ellipsoid algorithms. For what kinds of problems are they most suitable?

25

Solving Nonlinear Programs

Throughout our study of nonlinear programming I have tried to teach you practical algorithms, but to keep the exposition simple and the MATLAB code short I have avoided discussing certain issues that arise in solving real problems. The time has come to address those issues, if only in the limited way permitted by the introductory character of this text.

25.1 Summary of Methods

The table on the next page catalogs the nonlinear program solvers we have developed. It omits `ntplain.m`, `ntchol.m`, `qeplain.m`, and `ntfeas.m` because each of those routines was used only to illustrate some difficulty that was then overcome by the routines that *are* listed. It also omits `bls.m` and `wolfe.m`, which are of course solvers for unconstrained nonlinear programs having $n = 1$. Some of the listed routines use these line search codes, and some of the listed routines use other listed routines; for example, `penalty.m` uses `ntrs.m`.

When you have decided to attempt the numerical solution of a nonlinear program you can begin by consulting this table. Trying one (or all) of the solvers that fit your problem might turn up an optimal point without further ado. Alas, it is more likely that each solver will fail for one reason or another. These simple routines were all written not as industrial-strength code but merely to help you understand the algorithms they implement. Production implementations, such as those discussed in §8.3.1, might work better for solving your problem, and now that you understand the algorithms you can make effective use of those black-box codes. But often they fail too. Then, instead of using software that someone else wrote, the best approach is to use the ideas you have learned (and those discussed below) to devise a custom algorithm or algorithm variant that is a perfect fit to your problem.

Some problems have both equality and inequality constraints, but no solver on our list can handle both. Robustness against nonconvexity can be improved by using a line search or restricted-step approach, but most of our codes take full steps instead. In a real problem the components of $\mathbf{x}^\star$ might differ by many orders of magnitude, but so far I have said nothing about the effects of bad scaling or how to mitigate them. Depending on problem scaling, the absolute tests for convergence that we have used might stop an algorithm too soon or not at all. Many real problems involve functions that lack analytic derivatives, so their gradient and Hessian components can't be computed from formulas. Finally, some problems involve so many variables or constraints that the classical algorithms we have studied are mostly useless, and then we must resort to methods that are useless for solving classical nonlinear programs. The rest of this Chapter is devoted to these important practical matters.

algorithm family	implementations presented in this text	≤	=	note
steepest descent	[xstar,k]=sd(xzero,xl,xh,n,kmax,epz,grd)	□	□	
	[xstar,kp]=sdfs(xzero,kmax,epz,grd,hsn)	□	□	
	[xstar,k]=sdw(xzero,xl,xh,n,kmax,epz,fcn,grd)	□	□	
Newton descent	[xstar,kp,nm,rc]=nt(xzero,xl,xh,kmax,epz,grd,hsn,gama)	□	□	
	[xstar,kp,nm,rc]=ntfs(xzero,kmax,epz,grd,hsn,gama)	□	□	
	[xstar,kp,nm,rc]=ntw(xzero,xl,xh,kmax,epz,fcn,grd,hsn,gama)	□	□	
quasi-Newton	[xstar,Gstar,kp,rc]=dfp(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)	□	□	
	[xstar,Gstar,kp,rc]=dfpfs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)	□	□	
	[xstar,Gstar,kp,rc]=bfgs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)	□	□	
	[xstar,Gstar,kp,rc]=bfgsfs(xzero,Gzero,xl,xh,kmax,epz,fcn,grd)	□	□	
conjugate	[xstar,kp,beta]=cg(xzero,kmax,epz,Q,b)	□	□	1
grd	[xstar,kp,rc]=flrv(xzero,xl,xh,kmax,epz,fcn,grd)	□	□	
	[xstar,kp,rc]=plrb(xzero,xl,xh,kmax,epz,fcn,grd)	□	□	
trust region	[xstar,kp,nm,rc,r]=ntrs(xzero,rzero,kmax,epz,fcn,grd,hsn,gama)	□	□	
	[xstar,kp,rc]=trust(xzero,kmax,epz,fcn,grd,hsn)	□	□	
nullspace	[xstar,kp,rc,nm]=qpeq(Q,c,A,b,kmax,epz)	□	■	2
	[xstar,k,rc,W,lambda]=qpin(Q,c,A,b,kmax,epz)	■	□	2
	[xstar,k,rc]=rsdeq(grd,hsn,A,b,kmax,epz)	□	■	3
	[xstar,k,rc,nm]=rneq(grd,hsn,A,b,kmax,epz)	□	■	3
penalty	[xstar,kp,rc,mu,nm]=penalty(name,meq,xzero,muzero,epz)	□	■	
barrier	[xbeta,kp,rc,nr,nm]=ntin(xzero,kmax,epz,fcn,m)	■	□	
	[xstar,kp,rc,mu,nm]=barrier(name,mineq,xzero,muzero,epz)	■	□	
exact penalty	[xstar,k,rc,lstar,pn,tstar]=emiqp(name,mi,xzero,kmax,epz)	■	□	
	[xstar,lambda,kl,rc,mu]=auglag(name,meq,xzero,epz,kmax)	□	■	
interior point	[xstar,k]=nlpin(xzero,m,epz,fcn,grd,hsn)	■	□	
	[xstar,k]=nlpinp(xzero,m,epz,fcn,grd,hsn)	■	□	
feasible point	[xstar,k,rc]=grg(fcn,grd,hsn,n,m,xzero,kmax,epz)	□	■	
	[xstar,k,rc,lstar]=ntlg(fcn,grd,hsn,n,m,xzero,lzero,kmax,epz)	□	■	
	[xstar,k,rc,lstar]=sqp(fcn,grd,hsn,n,m,xzero,lzero,kmax,epz)	□	■	
	[xstar,k,rc,lambda,mustar]=iqp(fcn,grd,hsn,m,xzero,kmax,epz)	■	□	
ellipsoid	[xstar,rc,k,Qstar]=ea(xzero,Qzero,m,kmax,tol,fcn,grd)	■	□	
	[xlr,xr,xhr,rc,k]=wander(xl,xh,m,kmax,tol,fcn,grd)	■	□	

1. This routine minimizes a quadratic objective.
2. This routine minimizes a quadratic objective subject to linear constraints.
3. This routine minimizes a general objective subject to linear constraints.

25.2 Mixed Constraints

Many applications yield nonlinear programs that have a mixture of equality and inequality constraints. The **algorithm extensions** required to handle **mixed constraints** are trivial for some methods but intricate for others.

25.2.1 Natural Algorithm Extensions

In §19.4, I mentioned that the quadratic penalty and logarithmic barrier ideas have been combined to produce hybrid algorithms capable of solving problems that include both equality and inequality constraints. Minimizing

$$\Omega(\mathbf{x};\mu) = f_0(\mathbf{x}) + \mu \sum_{i=m_i+1}^{m_i+m_e} [f_i(\mathbf{x})]^2 - \frac{1}{\mu} \sum_{i=1}^{m_i} \ln[-f_i(\mathbf{x})]$$

in a sequence of unconstrained optimizations, each starting at the optimal point of the previous one and using a value of μ greater than the previous value, yields an algorithm that behaves like its parents. It requires a starting point that is strictly feasible for the inequalities, converges linearly under the right conditions, and is prone to the numerical woes discussed in §18.4.

In §20.1, I mentioned that the max penalty method can be used to solve problems that include both inequality and equality constraints, if we minimize

$$\Omega(\mathbf{x};\mu) = f_0(\mathbf{x}) + \mu \sum_{i=1}^{m_i} \max[0, f_i(\mathbf{x}] + \mu \sum_{i=m_i+1}^{m_i+m_e} |f_i(\mathbf{x})|$$

in a sequence of unconstrained optimizations each starting at the optimal point of the previous one and using a value of μ greater than the previous value. This objective, because it is not smooth, is troublesome for the unconstrained minimization algorithms we have studied.

In §21.3.4, I mentioned that equality constraints can be included along with inequalities in formulating the interior point method for nonlinear programming. This adds terms for the equalities to the Lagrangian

$$\mathcal{L}(\mathbf{x},\mathbf{s},\boldsymbol{\lambda}) = f_0(\mathbf{x}) - \mu \sum_{i=1}^{m_i} \ln(s_i) + \sum_{i=1}^{m_i} \lambda_i[f_i(\mathbf{x}) + s_i] + \sum_{i=m_i+1}^{m_i+m_e} \lambda_i f_i(\mathbf{x})$$

of §21.3.1, enlarging $\nabla_{\boldsymbol{\lambda}}\mathcal{L}$ and the Jacobian of the primal-dual system.

25.2.2 Extensions Beyond Constraint Affinity

Other algorithms for constrained nonlinear programming have a pronounced **constraint affinity** for either equalities or inequalities. For example, the ellipsoid method has a simpler realization for inequality constraints than for equalities, while sequential quadratic programming is simpler if the constraints are equalities than if they are inequalities.

Some algorithms with an affinity for equality constraints can be made to work for problems that also have inequality constraints by adding slack variables to make the inequalities into equalities and then using a bounded line search to keep the slack variables nonnegative. This idea is discussed in §20.2.5.

Some algorithms with an affinity for equality constraints can be made to work for problems that also have inequality constraints by using an active-set strategy to ignore the slack inequalities and treat the tight ones as equations. In §22.2.4 we used this idea to get from `qpeq.m` to `qpin.m`, which could in turn be generalized to handle equality constraints too. The resulting quadratic program solver could then be used to generalize `iqp.m` so that it would handle equality and inequality constraints in the same problem. Active set strategies have also been devised [137] for algorithms that solve problems in which the inequality constraints are not linear, but they are much more complicated than the one we developed for linear inequalities.

Some algorithms with an affinity for inequality constraints can be made to work for problems that also have equality constraints, by constructing a flat that supports the hypersurface of the equalities at $\mathbf{x}^k$, minimizing the objective within that flat subject only to the inequalities, projecting the resulting point back onto the hypersurface, and repeating the process. This is a generalization of the GRG algorithm we derived in §23.1.2.

25.2.3 Implementing Algorithm Extensions

Extending an algorithm to handle mixed constraints introduces complications to both the theory of the method and its implementation. Of these the most obvious is the need to distinguish between the m_e equality and m_i inequality constraints. Both numbers must be input parameters to the solver, so that it can invoke the value, gradient, and Hessian routines that define the problem with the correct function index, $i \in \{1 \ldots m_i\}$ for the inequalities or $i \in \{m_i + 1 \ldots m_i + m_e\}$ for the equalities. Those routines must then be coded in a way that puts the objective first, the inequalities next, and the equalities last.

Complex algorithm extensions, such as those described in §25.2.2, tend to be far less robust than the algorithm they are extending. It must be an irresistible temptation for an implementer, or for the architect of a scientific subprogram library, to provide a code that can in principle solve problems having any mixture of constraints, but the result can be less than completely satisfactory. When these methods for mixed constraints fail, practitioners often resort to problem-specific *ad hoc* approaches. If the m_e equality constraints can be used to analytically eliminate m_e of the variables, the remaining problem will have only inequality constraints. If it is possible to make a good guess at which inequalities will be active at optimality, or if the number of possible active sets is small enough that you can try them all, then it is necessary to solve only problems having equality constraints. Some problems are separable (see §25.7.1) in a way that permits their solution by alternately solving subproblems that involve only the equalities or only the inequalities, and then a separate solver can be used for each set of constraints.

25.3 Global Optimization

Recall from §16.6 that a convex program is a standard-form NLP in which all of the functions are convex. Every minimizing point of a convex program is a global minimizer, and if the objective is *strictly* convex there is only one such point. These properties make convex programs easy to solve using the algorithms we have studied. Unfortunately (or fortunately, depending on your interests) most applications of nonlinear optimization give rise to problems that are *not* convex programs.

25.3.1 Finding A Minimizing Point

A nonlinear program that is not a convex program can be *hard* to solve even if it has a unique minimizing point, as we discovered in §17.1 when we studied `h35`. For that problem we found that, compared to full-step modified Newton descent, a restricted-step method is more likely to reach $\mathbf{x}^\star$ from a distant starting point and takes fewer iterations when both work. Our restricted-step method adjusts the steplength dynamically, accepting a trial step only if it yields at least the objective decrease predicted by the quadratic model of the function. This is somewhat analogous to enforcing the sufficient-decrease (Armijo) Wolfe condition in a descent method that uses a line search, so it is not surprising that `ntw.m` also solves `h35` quickly.

```
octave:1> xzero=[1;0.6];
octave:2> xl=[0;0];
octave:3> xh=[15;2];
octave:4> [xstar,kp,nm,rc]=ntw(xzero,xl,xh,100,1e-16,@h35,@h35g,@h35h,0.5)
xstar =

   3.00000
   0.50000

kp =  10
nm = 0
rc = 0
octave:5> quit
```

Using restricted-step methods and enforcing the Wolfe conditions are **globalization strategies** [4, §11.5] [5, §3.2] that improve the robustness and performance of a nonlinear programming algorithm. The simplest way to gain their benefit is by using a line search to solve the unconstrained subproblems of an algorithm that has subproblems, rather than taking full steps. We did that in `penalty.m` and `auglag.m` by using `ntrs.m` rather than `ntfs.m` to minimize the penalty function at each value of μ. It is also possible in some algorithms that do not explicitly solve unconstrained subproblems to insist that the step from $\mathbf{x}^k$ to $\mathbf{x}^{k+1}$ actually go downhill. The table on the next page summarizes the steps that are taken by the constrained optimization routines listed in §25.1, and reveals many opportunities to replace a full step by a restricted step or a Wolfe line search (some cases are identified as "tricky" because taking less than the full step would affect other aspects of

code	step from $\mathbf{x}^k$ to $\mathbf{x}^{k+1}$	globalizable?
`qpeq.m`	full modified Newton on flat of =	yes
`qpin.m`	longest modified Newton in slack ≤ on flat of tight ≤	yes
`rsdeq.m`	full steepest descent on flat of =	yes
`rneq.m`	full modified Newton on flat of =	yes
`penalty.m`	uses `ntrs.m`	done
`ntin.m`	longest reduced Newton interior to feasible set	yes
`barrier.m`	full to next point from `ntin.m`	yes
`emiqp.m`	full to next point from `iqp.m`	yes
`auglag.m`	uses `ntrs.m`	done
`nlpin.m`	longest primal-dual interior to feasible set	yes
`nlpinp.m`	full primal interior to feasible set	yes
`grg.m`	full steepest-descent on tangent hyperplane	yes
`ntlg.m`	full Newton-Lagrange	tricky
`sqp.m`	full to next point from `qpeq.m`	tricky
`iqp.m`	full to next point from `qpin.m`	yes
`ea.m`	full to next ellipsoid center	tricky
`wander.m`	full to next ellipsoid center, or recenter	tricky

the algorithm). The use of line searches in interior point methods was mentioned in §21.3.4, and their use in sequential quadratic programming algorithms was discussed at the end of §23.2.4.

Globalizing a full-step algorithm by restricting the length of its steps or searching the line between each $\mathbf{x}^k$ and the proposed next point increases the complexity of the implementation and might increase its running time on problems that it would have solved by taking full steps. As I first mentioned in §9.4, there is usually a tradeoff between robustness and speed.

The trust-region idea can also be used to devise globalization strategies [4, §11.6] [5, §4.2]. Some authors refer to restricted-step methods as trust-region methods, but the algorithm we developed in §17.3 does more than just limit the step length. In our trust-region method, if the full modified Newton step is too long we instead move to a point that minimizes the quadratic model of the function on the trust-region boundary, and this step will usually be in a direction different from that of the Newton step. If the problem is unconstrained that does not matter, so we can expect `trust.m` to be a robust method for unconstrained minimization. But many algorithms for constrained nonlinear programming pick the direction of each step in a way that preserves or leads to satisfaction of the constraints, and stepping in a different direction might prevent the algorithm from achieving that goal. In the parlance of the table above, this puts globalization by trust regions in the "tricky" category for several of our methods (see Exercise 25.8.17). Using the trust-region idea for constrained minimization is a research area involving the design of new algorithms that are based upon it from the beginning. In this context the trust-region idea might be realized using a proposed direction other than the Newton direction or a model function other than the quadratic approximation to f_0 [1, §10.3].

25.3.2 Finding The Best Minimizing Point

A nonlinear program that is not a convex program can have several local minima (see §9.3) and finding one that is a global minimum is in general hard (see §7.9). Algorithms have been proposed [126] [4, references listed in §2.8] for solving nonconvex programs in certain classes, such as linearly-constrained indefinite quadratic programs [1, §11.2], but except for those special cases all we can do is make the most artful possible use of algorithms for general nonlinear programming and hope for the best.

We kept a record point in implementing only two of the methods we have studied, pure random search and the ellipsoid algorithm, because in both it is likely that $f_0(\mathbf{x}^{k+1}) > f_0(\mathbf{x}^k)$ in some iterations even when the problem is convex. But if the problem is nonconvex that can also happen when the other methods are used, so keeping a record point is an important globalization strategy for all of them. This is especially true when there are multiple local minima, because that introduces the possibility that an algorithm will visit the global minimum but subsequently become trapped at a higher local minimum. Keeping a record point makes any algorithm implementation more complicated, and if the feasibility of the current point is not already known checking that also makes the code run slower, but if you intend to solve problems that are not convex it is always worth the trouble.

The ellipsoid algorithm is more likely than other methods to find a global minimum of a nonconvex problem, probably because its lunatic excursions sample widely-spaced points early in the solution process. This behavior is especially desirable when there are multiple local minima, so if n is small enough and the problem has only inequality constraints it makes sense to try `wander.m` or a hybrid algorithm of the sort described in §24.8.

The idea of sampling widely-spaced points is often implemented in a more deliberate way by using the **multistart strategy**, in which one or more algorithms are run from randomly-selected starting points and the best solution is taken to be the global optimum.

25.4 Scaling

This harmless-looking unconstrained minimization [5, p26] has $\mathbf{x}^\star = [0, 0]^\top$ for any $s \geq 0$.

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) = sx_1^2 + x_2^2$$

To solve it numerically I wrote these routines to compute the value and derivatives of f_0.

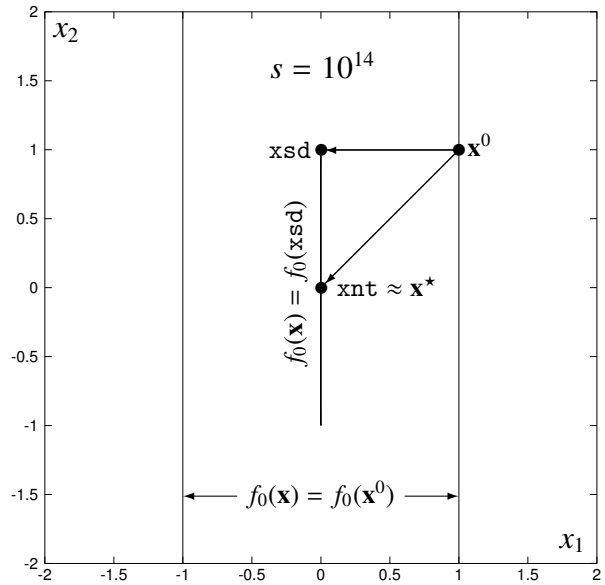
```
function f=scl(x)
  global s
  f=s*x(1)^2+x(2)^2;
end
```

```
function g=sclg(x)
  global s
  g=[2*s*x(1);2*x(2)];
end
```

```
function H=sclh(x)
  global s
  H=[2*s,0;0,2];
end
```

The Octave session on the next page shows our steepest-descent code `sd.m`, which uses the bisection line search `bls.m`, solving the problem easily for $s = 1$ `5>-6>` but failing to solve it at all for $s = 10^{14}$ `7>-8>`. Increasing the iteration limit `kmax` does not help.


```
octave:1> xz=[1;1];
octave:2> xl=[-10;-10];
octave:3> xh=[10;10];
octave:4> kmax=1000;
octave:5> tol=1e-16;
octave:5> global s=1
octave:6> xsd=sd(xz,xl,xh,2,kmax,tol,@sclg)
xsd =

   0
   0

octave:7> s=1e14
s =  1.0000e+14
octave:8> xsd=sd(xz,xl,xh,2,kmax,tol,@sclg)
xsd =

  -2.1803e-14
   1.0000e+00

octave:8> kmax=1;
octave:9> xnt=nt(xz,xl,xh,1,tol,@sclg,@sclh,0.5)
xnt =

   4.2188e-15
   4.6629e-15
```



When $s = 1$ the contours of $f_0(\mathbf{x})$ are circles, so from $\mathbf{x}^0 = [1, 1]^\top$ the direction of steepest descent points at $\mathbf{x}^\star = [0, 0]^\top$ and only one line search is needed to get there.

When $s = 10^{14}$ the contours of $f_0(\mathbf{x})$ are right ellipses so tall compared to their width that their sides appear to be vertical lines. The picture above shows two such contours, passing through $\mathbf{x}^0 = [1, 1]^\top$ and $\mathtt{xsd} \approx [-2\times10^{-14}, 1]^\top$. At the starting point the normalized direction of steepest descent $\mathbf{g}^0 = -\nabla f_0(\mathbf{x}^0)/\|\nabla f_0(\mathbf{x}^0)\| \approx [-1, -10^{-14}]^\top$, and the first step that `sd.m` takes is to $\mathbf{x}^1 \approx [4 \times 10^{-15}, 1]^\top$. There the direction of steepest descent is $\mathbf{g}^1 \approx [-0.4, -0.9]^\top$ but the elliptical contours of f_0 are so compressed that the minimum in that direction is found only a tiny distance away, at $\mathbf{x}^2 \approx [-3 \times 10^{-14}, 1]^\top$. Subsequent iterations alternate between approximately these two points, so no progress is ever made in reducing x_2 toward $x_2^\star = 0$.

An unconstrained optimization is said [5, p26] to be **poorly scaled** if there are indices i and j and points $\mathbf{x}$ for which $\partial f_0(\mathbf{x})/\partial x_i \gg \partial f_0(\mathbf{x})/\partial x_j$. In our example with $s = 10^{14}$ this condition is satisfied where $sx_1 \gg x_2$ or $x_1 \gg 10^{-14}x_2$, which is almost everywhere that $x_1 \neq 0 = x_1^\star$.

The Octave session above shows `8>-9>` that `nt.m`, which also uses the `bls.m` line search, gets very close to $\mathbf{x}^\star$ in a single step (in 4 iterations it gets within `tol`). Some algorithms are more affected than others by poor scaling; steepest descent is sensitive [107, p222-225] because scaling the variables changes the direction of search, while Newton descent is **scale-invariant** [59, §3.3] (but see [5, Example 19.1]). Conjugate-gradient methods are sensitive [5, p585], as are quasi-Newton methods [1, p420] except for those that are **self-scaling** [107, §9.6] [59, p59]. Poor scaling can be mitigated in the trust-region method by using trust regions that are ellipsoids rather than hyperspheres [5, p95-97].

Poor scaling can make a sensitive algorithm fail altogether, but even if it does not it can cause problems by accelerating the growth of roundoff errors [2, p230] and by increasing the condition number of the Hessian (see §18.4.2), which degrades the convergence constant for steepest descent and conjugate gradient methods [2, p70-77].

25.4.1 Scaling Variables

Suppose that before attempting the solution of

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \;\; f_0(\mathbf{x}) = sx_1^2 + x_2^2 = (\sqrt{s}x_1)^2 + x_2^2$$

we had made the substitutions $y_1 = \sqrt{s}x_1$ and $y_2 = x_2$ or

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \sqrt{s} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \mathbf{D}\mathbf{x}.$$

Then we could have used `sd.m` to solve

$$\underset{\mathbf{y}\in\mathbb{R}^2}{\text{minimize}} \;\; f_0(\mathbf{y}) = y_1^2 + y_2^2,$$

obtaining $\mathbf{y}^\star = [0, 0]^\top$ easily, from which

$$\mathbf{x}^\star = \mathbf{D}^{-1}\mathbf{y}^\star = \begin{bmatrix} \frac{1}{\sqrt{s}} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1^\star \\ y_2^\star \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

This is called **diagonal scaling** [1, p29] because to scale $\mathbf{x}$ we find $\mathbf{y} = \mathbf{D}\mathbf{x}$ where $\mathbf{D}$ is a diagonal matrix.

Applications involving physical measurements sometimes give rise to optimizations that are poorly scaled because of the units in which the data of the problem are expressed. In that case the bounds $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ can be used to find a diagonal scaling of the variables according to [2, p230]

$$y_j = \frac{x_j - \frac{1}{2}(x_j^{\mathrm{H}} + x_j^{\mathrm{L}})}{\frac{1}{2}(x_j^{\mathrm{H}} - x_j^{L})}, \quad j = 1 \ldots n.$$

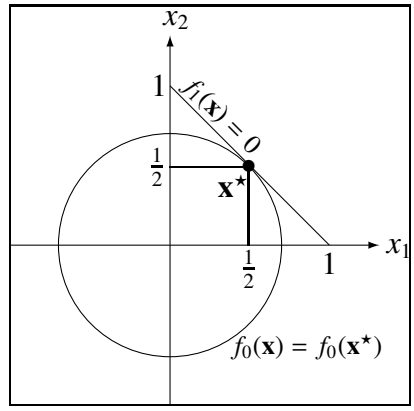
If $\mathbf{x}^{\mathrm{L}} \le \mathbf{x}^\star \le \mathbf{x}^{\mathrm{H}}$ and the solution process can find the optimal point without exceeding those bounds, then each y_j that it generates will lie in the range $[-1, 1]$. Depending on the problem this might help to ensure that the partials $\partial f_0/\partial y_j$ are not wildly different in magnitude.

25.4.2 Scaling Constraints

Our example of poorly scaled variables is difficult for `sd.m` when $s = 10^{14}$ because then the $\partial f_0(\mathbf{x})/\partial x_j$ are almost everywhere vastly different *from each other.* In a constrained optimization, the Lagrange multipliers depend on the scaling of the constraints [107, p402-403] and trouble can arise whenever a $\lambda_i = -\partial f_0/\partial f_i$ is vastly different *from 1.*

This problem has $\mathbf{x}^\star = [\frac{1}{2}, \frac{1}{2}]^\top$ with $\lambda^\star = 1/s$.

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & s(1 - x_1 - x_2) = 0 \end{array}$$



To solve it I wrote the MATLAB routines `sclc.m`, `sclcg.m`, and `sclch.m`, and used `auglag.m` as shown in the Octave session below the function listings. When $s = 1$ `1>` the algorithm succeeds but when s is very big `3>` or very small `5>` it fails.

```
function f=sclc(x,i)
  global s
  switch(i)
    case 0
      f=x(1)^2+x(2)^2;
    case 1
      f=s*(1-x(1)-x(2));
    end
end
```

```
function g=sclcg(x,i)
  global s
  switch(i)
    case 0
      g=[2*x(1);2*x(2)];
    case 1
      g=[-s;-s];
    end
end
```

```
function H=sclch(x,i)
  switch(i)
    case 0
      H=[2,0;0,2];
    case 1
      H=[0,0;0,0];
    end
end
```

```
octave:1> global s=1
octave:2> [xstar,lambda]=auglag('sclc',1,[2;2],1e-16,40)
ans =

   0.50000
   0.50000

lambda =  1.0000
octave:3> s=1e14
s =  1.0000e+14
octave:4> [xstar,lambda]=auglag('sclc',1,[2;2],1e-16,40)
ans =

  -1305.0
   1306.0

lambda = 0
octave:5> s=1e-14
s =  1.0000e-14
octave:6> [xstar,lambda]=auglag('sclc',1,[2;2],1e-16,40)
ans =

   1.0009e-28
   1.0009e-28

lambda =  8.0000e-13
```

The precise mechanism by which failure can occur because of poorly scaled constraints differs from one algorithm to another; in `auglag.m` the method of multipliers does not converge to the optimal point. In this example $\mathbf{H}_{f_1}$ does not depend on s, but in a problem where it does poor scaling of the constraint could lead to that matrix being badly conditioned [4, 7.6.4].

If there are m equality or inequality constraints we can use diagonal scaling to multiply each by a constant, like this.

$$\mathbf{F} = \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix} \quad \longrightarrow \quad \mathbf{DF} = \begin{bmatrix} d_{11} f_1(\mathbf{x}) \\ \vdots \\ d_{mm} f_m(\mathbf{x}) \end{bmatrix}$$

25.5 Convergence Testing

Algorithms for nonlinear optimization are infinitely convergent (see §9.2) so when they work at all $\mathbf{x}^k$ keeps getting closer to $\mathbf{x}^\star$ as k increases, and in perfect arithmetic that process might go on indefinitely. But floating-point numbers have finite precision, practical applications do not require perfect results, and we can't wait forever. How do we decide when an answer is close enough? Various tests of the form

```
if(ε_k < ϵ) STOP
```

have been proposed [1, p323] [98, §2.4] in which ε_k is some measure of the error or uncertainty in $\mathbf{x}^k$. In previous Chapters we have used several different **absolute error** measures for ε_k, including the norm of a step length, the norm of a gradient, the absolute value of a directional derivative, and the distance between shrinking variable bounds.

The trouble with using absolute measures of error for ε_k is that they are sensitive to scaling. If every $\mathbf{x}^k$ has components close to 1 then requiring $\|\mathbf{x}^{k+1} - \mathbf{x}^k\| < 0.01$ stops the algorithm when $\mathbf{x}^k$ is known to within about 1%, but if some $\mathbf{x}^k$ has components that are 10^{-6} or 10^{+6} the algorithm might stop long before finding a useful answer, or never.

We could instead use a **relative error** measure such as $\varepsilon_k = \|\mathbf{x}^{k+1} - \mathbf{x}^k\| / \|\mathbf{x}^k\|$, but this fails if the $\mathbf{x}^k$ approach $\mathbf{0}$ as $k \to \infty$ or if $\mathbf{x}^k = \mathbf{0}$ for some finite k.

The more complicated measure of step length

$$\varepsilon_k = \frac{\|\mathbf{x}^{k+1} - \mathbf{x}^k\|}{1 + \|\mathbf{x}^k\|}$$

tries to avoid the problems of the absolute and relative measures by behaving like relative error when $\|\mathbf{x}^k\|$ is large and like absolute error when $\|\mathbf{x}^k\|$ is small.

A quite different approach to measuring the difference between two floating-point numbers is based on comparing their bit strings [100, p68-69]. According to the IEEE standard [84] an 8-byte value (which MATLAB uses) is stored in a doubleword of 64 bits. The first bit denotes the sign of the number, the next 11 bits the biased exponent, and the final 52 bits the binary fraction. If the components x_j^{k+1} and x_j^k start to disagree at bit b then they are different in $e_j = 64 - b + 1$ bits and we could measure the difference between $\mathbf{x}^{k+1}$ and $\mathbf{x}^k$ by

$$\varepsilon_k = \max_{j \in 1 \ldots n} e_j.$$

25.6 Calculating Derivatives

Suppose we want to solve the following unconstrained convex minimization, which I will call the egg problem (see §28.7.40).

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) = e^{(x_1-2)^2}\Gamma(x_2) \qquad \text{where} \qquad \Gamma(t) = \int_0^\infty y^{t-1}e^{-y}dy$$

Here $\Gamma(t)$ is the **gamma function** [116, §3.3]. To find the stationary points of $f_0(\mathbf{x})$ we need only set its derivatives to zero and solve the resulting algebraic equations, in which $\Psi(t)$ is the **digamma function** [6, §6.3].

$$\begin{aligned}\frac{\partial f_0}{\partial x_1} &= \Gamma(x_2)e^{(x_1-2)^2}(2(x_1-2)) = 0 \\ \frac{\partial f_0}{\partial x_2} &= e^{(x_1-2)^2}\,\frac{d\Gamma(x_2)}{dx_2} = e^{(x_1-2)^2}\,\Psi(x_2)\Gamma(x_2) = 0\end{aligned} \qquad \text{where} \quad \Psi(t) = \int_0^\infty \left(\frac{e^{-y}}{y} - \frac{e^{-ty}}{1-e^{-y}}\right)dy$$

The first stationarity condition is satisfied by $\bar{x}_1 = 2$, but it is far from obvious what $\bar{x}_2$ should be to satisfy the second so an analytic solution to this problem appears unlikely. To minimize $f_0(\mathbf{x})$ using a gradient-based algorithm we need numerical values of its partial derivatives, but Octave has no built-in function for $\Psi(t)$.

Nonlinear programs often involve functions whose derivatives are inconvenient, expensive, or impossible to calculate from a formula; I have referred to such problems as type–2. If a function value is the numerical solution of a differential equation as in §8.5, or the output of a simulation, or the result of a physical measurement, then there is no closed-form expression for its derivative and to approximate its gradient or Hessian we must resort to finite differencing [20, §4.1] [30, §7.1].

25.6.1 Forward-Difference Approximations

Finite-difference derivatives are based on the Taylor's series approximation of the function and on the definition of a derivative. Recall (see §28.1.2) that if $x \in \mathbb{R}^1$ and $f(x)$ is sufficiently smooth we can write

$$f(x+\Delta) = f(x) + \Delta f'(x) + \frac{\Delta^2}{2}f''(\xi)$$

where ξ is some point in the interval $[x, x+\Delta]$. Solving for the derivative and assuming the f'' term is relatively small,

$$f'(x) \;=\; \frac{f(x+\Delta)-f(x)}{\Delta} - \frac{\Delta}{2}f''(\xi) \;\approx\; \frac{f(x+\Delta)-f(x)}{\Delta}$$

and for $\mathbf{x} \in \mathbb{R}^n$ we can approximate the partial derivatives of $f(\mathbf{x})$ as

$$\boxed{\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x}+\Delta\mathbf{e}^i)-f(\mathbf{x})}{\Delta}}$$

where $\mathbf{e}^i$ is as usual the unit vector having a 1 for its ith component and zeros elsewhere. The error in this **forward difference approximation** is no greater than $(\Delta/2)f''(\xi)$, which is proportional to Δ, so it is said to be **of order** Δ or $O(\Delta)$ [5, p631]. To approximate a single partial derivative in this way requires 2 function evaluations; to find a gradient vector requires $n+1$.

To approximate the second derivatives of f we can forward-difference our approximation to $\partial f/\partial x_i$ in the j direction, like this.

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}(\mathbf{x})\right) \approx \frac{\frac{\partial f}{\partial x_i}(\mathbf{x}+\Delta\mathbf{e}^j) - \frac{\partial f}{\partial x_i}(\mathbf{x})}{\Delta}$$

We will use the approximation given at the top of the page for the right-hand term in the numerator of this fraction, and the one below for the left-hand term.

$$\frac{\partial f}{\partial x_i}(\mathbf{x}+\Delta\mathbf{e}^j) \approx \frac{f([\mathbf{x}+\Delta\mathbf{e}^j]+\Delta\mathbf{e}^i)-f(\mathbf{x}+\Delta\mathbf{e}^j)}{\Delta}$$

Then

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) \approx \frac{1}{\Delta}\left(\frac{f(\mathbf{x}+\Delta\mathbf{e}^j+\Delta\mathbf{e}^i)-f(\mathbf{x}+\Delta\mathbf{e}^j)}{\Delta} - \frac{f(\mathbf{x}+\Delta\mathbf{e}^i)-f(\mathbf{x})}{\Delta}\right)$$

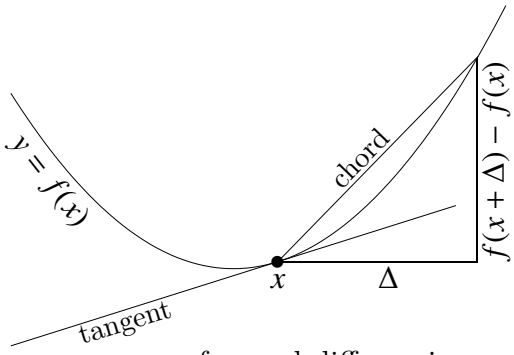
or [5, p202]

$$\boxed{\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) \approx \frac{f(\mathbf{x}+\Delta\mathbf{e}^i+\Delta\mathbf{e}^j)-f(\mathbf{x}+\Delta\mathbf{e}^i)-f(\mathbf{x}+\Delta\mathbf{e}^j)+f(\mathbf{x})}{\Delta^2}}$$

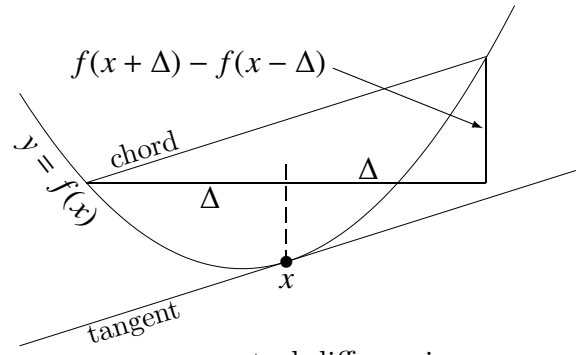
The error in this approximation is also $O(\Delta)$. To approximate a single second partial derivative in this way requires 4 function evaluations; to find a symmetric Hessian matrix requires $\frac{1}{2}n(n+1)+n+1 = (\frac{1}{2}n+1)(n+1)$ of them.

25.6.2 Central-Difference Approximations

Forward-differencing approximates the slope of the tangent line at x by the slope of a chord between x and $x+\Delta$, as shown in the left-hand picture at the top of the next page. It is more accurate to use the chord between $x-\Delta$ and $x+\Delta$, as shown on the right, so that x is the midpoint of the interval. This approximation is exact for a quadratic, and in these pictures $f(x)$ is a quadratic so on the right the chord is exactly parallel to the tangent line.



forward differencing



central differencing

To find a formula for the centered approximation to the derivative we again use the Taylor's series approximation of the function. Subtracting the approximation of the function at $x-\Delta$ from that at $x+\Delta$, we get

$$\begin{aligned} f(x+\Delta) &= f(x) + \Delta f'(x) + \frac{(+\Delta)^2}{2} f''(x) + O(\Delta^3) \\ \ominus \quad f(x-\Delta) &= f(x) - \Delta f'(x) + \frac{(-\Delta)^2}{2} f''(x) + O(\Delta^3) \\ \hline f(x+\Delta) - f(x-\Delta) &= \qquad 2\Delta f'(x) \qquad\qquad + O(\Delta^3). \end{aligned}$$

Here the error terms are different but of the same order, so I have denoted them all by $O(\Delta^3)$. Solving for f', assuming that the error is small compared to the derivative, and generalizing as we did before to the case of $\mathbf{x} \in \mathbb{R}^n$, we get this **central difference approximation** for the first partial derivatives of $f(\mathbf{x})$.

$$\boxed{\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \Delta \mathbf{e}^i) - f(\mathbf{x} - \Delta \mathbf{e}^i)}{2\Delta}}$$

The error in this approximation is $O(\Delta^3/\Delta) = O(\Delta^2)$, and to approximate $\nabla f(\mathbf{x})$ using this formula requires $2n$ function values.

To approximate the second derivatives of f we can central-difference the above approximation to $\partial f/\partial x_i$ as follows.

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}(\mathbf{x})\right) \approx \frac{\dfrac{\partial f}{\partial x_i}(\mathbf{x} + \Delta \mathbf{e}^j) - \dfrac{\partial f}{\partial x_i}(\mathbf{x} - \Delta \mathbf{e}^j)}{2\Delta}$$

Using the formula that is boxed above, we can approximate the terms in the numerator of this fraction as shown on the next page.

$$\begin{aligned}\frac{\partial f}{\partial x_i}(\mathbf{x}+\Delta\mathbf{e}^j) &\approx \frac{f([\mathbf{x}+\Delta\mathbf{e}^j]+\Delta\mathbf{e}^i)-f([\mathbf{x}+\Delta\mathbf{e}^j]-\Delta\mathbf{e}^i)}{2\Delta}\\ \frac{\partial f}{\partial x_i}(\mathbf{x}-\Delta\mathbf{e}^j) &\approx \frac{f([\mathbf{x}-\Delta\mathbf{e}^j]+\Delta\mathbf{e}^i)-f([\mathbf{x}-\Delta\mathbf{e}^j]-\Delta\mathbf{e}^i)}{2\Delta}\end{aligned}$$

Then

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) \approx \frac{1}{2\Delta}\left(\frac{f(\mathbf{x}+\Delta\mathbf{e}^j+\Delta\mathbf{e}^i)-f(\mathbf{x}+\Delta\mathbf{e}^j-\Delta\mathbf{e}^i)}{2\Delta} - \frac{f(\mathbf{x}-\Delta\mathbf{e}^j+\Delta\mathbf{e}^i)-f(\mathbf{x}-\Delta\mathbf{e}^j-\Delta\mathbf{e}^i)}{2\Delta}\right)$$

or

$$\boxed{\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) \approx \frac{f(\mathbf{x}+\Delta\mathbf{e}^i+\Delta\mathbf{e}^j)-f(\mathbf{x}-\Delta\mathbf{e}^i+\Delta\mathbf{e}^j)-f(\mathbf{x}+\Delta\mathbf{e}^i-\Delta\mathbf{e}^j)+f(\mathbf{x}-\Delta\mathbf{e}^i-\Delta\mathbf{e}^j)}{4\Delta^2}}$$

The error in this approximation is also $O(\Delta^2)$. To fill in a symmetric Hessian matrix using this formula requires $4(\frac{1}{2}n(n+1)) = 2n(n+1)$ function values.

25.6.3 Computational Costs

Central-difference derivative approximations are much more accurate than forward-difference approximations, but they also take more work. The table below compares the number of function values required to the number of gradient or Hessian elements being approximated.

		to approximate a gradient			to approximate a symmetric Hessian		
	variables	f values	elements	ratio	f values	elements	ratio
forward	2	3	2	1.50	6	3	2.00
	10	11	10	1.10	66	55	1.20
	100	101	100	1.01	5151	5050	1.02
	n	$n+1$	n	$(n+1)/n$	$(\frac{1}{2}n+1)(n+1)$	$\frac{1}{2}n(n+1)$	$(n+2)/n$
central	2	4	2	2	12	3	4
	10	20	10	2	220	55	4
	100	200	100	2	20200	5050	4
	n	$2n$	n	2	$2n(n+1)$	$\frac{1}{2}n(n+1)$	4

Many optimization algorithms can tolerate derivatives that are slightly imprecise, so if a gradient component is more than twice as expensive to calculate as a function value, or if a Hessian component is more than four times as expensive, then using a central difference approximation might save CPU time; for forward differencing the ratios are even smaller. Otherwise it is faster to evaluate gradients and Hessians using formulas, if they are available.

25.6.4 Finding the Best Δ

In §25.6.1 and §25.6.2 we approximated derivatives by ignoring higher-order terms in the Taylor's series expansion of $f(x)$, which introduces a **truncation error** t. In forward differencing this error is $O(\Delta)$, so in the worst case $t \propto \Delta$; in central differencing the error is $O(\Delta^2)$, so we will assume that $t \propto \Delta^2$. To minimize truncation error we should make Δ small.

But the formulas we found all involve small differences between relatively large numbers, so evaluating our approximations with floating-point arithmetic also introduces **cancellation error** [100, §4.3]. In both forward and central differencing this error is $r \propto 1/\Delta$ [5, p196]. To minimize this roundoff error we should make Δ big.

To find the best compromise between truncation error and roundoff error, we must minimize the *total* error $E = t + r$ in each approximation. Assuming constants of proportionality a, b, c, and d we can use calculus to find the stationary points of $E(\Delta)$ like this.

forward differencing

$$\begin{aligned} E &= t + r = a\Delta + b/\Delta \\ \frac{dE}{d\Delta} &= a - \frac{b}{\Delta^2} = 0 \\ \Delta^2 &= b/a \\ \Delta^\star &= \sqrt[2]{b/a} \end{aligned}$$

central differencing

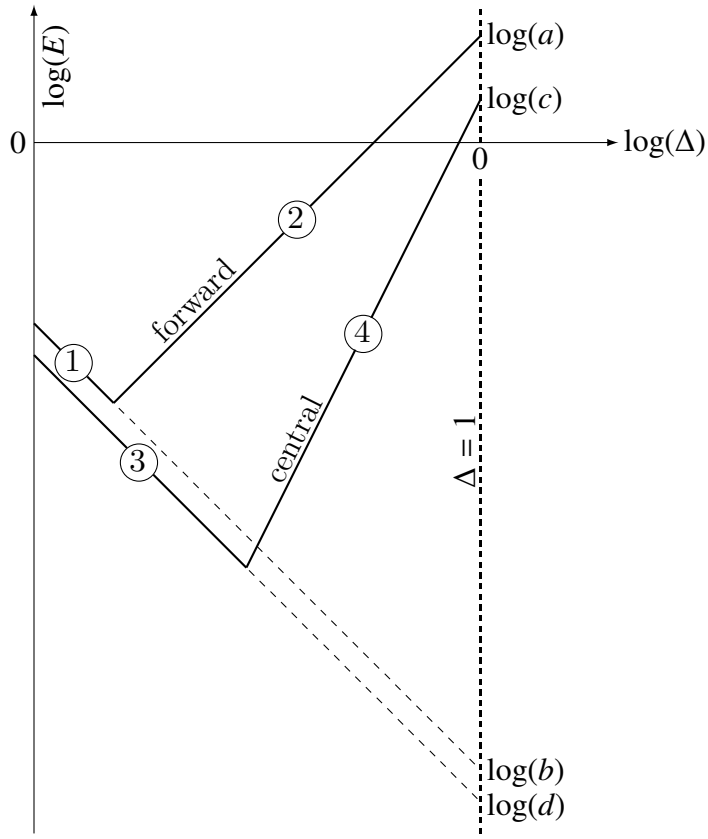
$$\begin{aligned} E &= t + r = c\Delta^2 + d/\Delta \\ \frac{dE}{d\Delta} &= 2\Delta c - \frac{d}{\Delta^2} = 0 \\ \Delta^3 &= d/(2c) \\ \Delta^\star &= \sqrt[3]{d/(2c)} \end{aligned}$$

Each $\Delta^\star$ is the unique minimizing point of the corresponding total error. The numbers a, b, c, and d depend on which derivative we approximate and on the function $f(x)$. These values are hard to calculate from first principles, but they can sometimes be deduced from experimental measurements as follows.

When Δ is very small, t is negligible compared to r and $E(\Delta) \approx r$; when Δ is very big, r is negligible compared to t and $E(\Delta) \approx t$. Using these simplifications we can predict what a graph of $\log(E)$ versus $\log(\Delta)$ might look like at the extreme values of Δ.

	forward differencing	central differencing
Δ small	$E \approx b/\Delta$	$E \approx d/\Delta$
	(1) $\log(E) \approx \log(b) - \log(\Delta)$	(3) $\log(E) \approx \log(d) - \log(\Delta)$
Δ big	$E \approx a\Delta$	$E \approx c\Delta^2$
	(2) $\log(E) \approx \log(a) + \log(\Delta)$	(4) $\log(E) \approx \log(c) + 2\log(\Delta)$

The picture on the next page plots the straight lines that make up the graph in this highly simplified error model, and from it we can see that a, b, c, and d are just the values of E at the points where those lines intersect $\Delta = 1$. In drawing this illustration I assumed that central differencing produces more accurate estimates than forward differencing at every Δ, and that it achieves its highest accuracy at a larger value of Δ than central differencing.

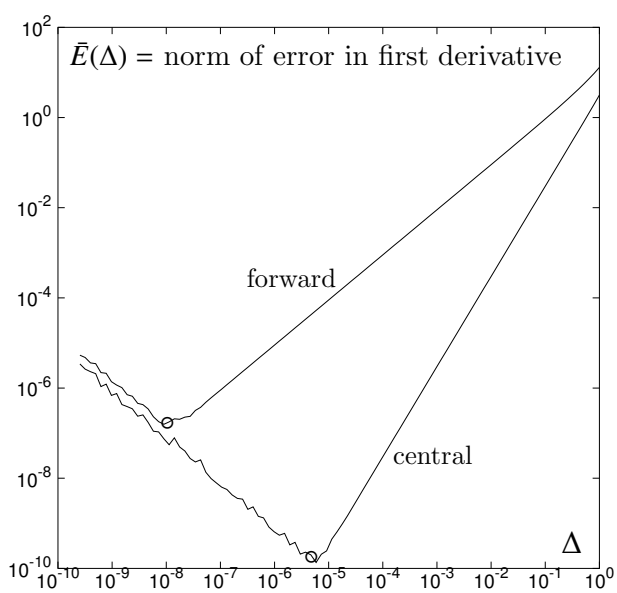


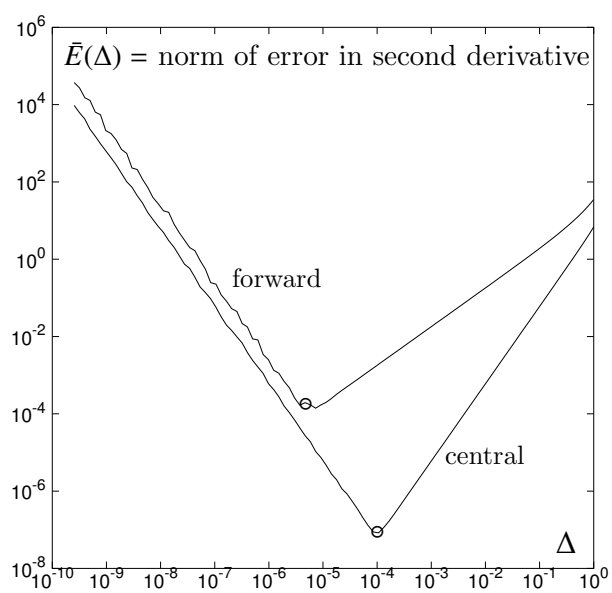
Each line segment corresponds to the equation having the same label. The line segments labeled 1 and 3 have slope −1, the line segment labeled 2 has slope +1, and the line segment labeled 4 has slope +2.

To study $E(\Delta)$ experimentally, I wrote the MATLAB programs listed on the next page. They find the first and second derivatives of $f(x) = e^x$ exactly and by using the approximations we found earlier, and produce the plots shown below. These graphs have the general appearance predicted by the error model we derived above, and the curves have their minima at these approximate values of $\Delta^\star$:

derivative	forward	central
$f'(x)$	9.0×10^{-9}	5.8×10^{-6}
$f''(x)$	7.3×10^{-6}	1.1×10^{-4}

We could also use graphs like these to estimate values for a, b, c, and d and then find the values of $\Delta^\star$ as the points where the line segments in the error model intersect.






```
 1 % first.m: approximate f' for f(x)=exp(x)
 2 clear;clf;set(gca,'FontSize',25)
 3 delta=1.25;
 4 for i=1:100
 5     delta=0.8*delta;
 6     deltai(i)=delta;
 7     dyfe=0;
 8     dyce=0;
 9
10 %   first derivative of e^x
11     for j=1:101
12         x=.01*(j-1);
13         y=exp(x);
14         xpd=x+delta;
15         ypd=exp(xpd);
16         xmd=x-delta;
17         ymd=exp(xmd);
18
19
20
21
22
23 %       forward differencing
24         dyf=(ypd-y)/delta;
25         dyfe=dyfe+(dyf-exp(x))^2;
26
27 %       central differencing
28         dyc=(ypd-ymd)/(2*delta);
29         dyce=dyce+(dyc-exp(x))^2;
30     end
31
32 %   find the norm of each set of errors
33     ndyfe(i)=sqrt(dyfe);
34     ndyce(i)=sqrt(dyce);
35 end
36
37 % plot the errors
38 hold on
39 axis('square')
40 loglog(deltai,ndyfe)
41 loglog(deltai,ndyce)
42 hold off
43 print -deps -solid first.eps
```

```
 1 % second.m: approximate f'' for f(x)=exp(x)
 2 clear;clf;set(gca,'FontSize',25)
 3 delta=1.25;
 4 for i=1:100
 5     delta=0.8*delta;
 6     deltai(i)=delta;
 7     d2yfe=0;
 8     d2yce=0;
 9
10 %   second derivative of e^x
11     for j=1:101
12         x=.01*(j-1);
13         y=exp(x);
14         xpd=x+delta;
15         ypd=exp(xpd);
16         xp2d=x+2*delta;
17         yp2d=exp(xp2d);
18         xmd=x-delta;
19         ymd=exp(xmd);
20         xm2d=x-2*delta;
21         ym2d=exp(xm2d);
22
23 %       forward differencing
24         d2yf=(yp2d-2*ypd+y)/delta^2;
25         d2yfe=d2yfe+(d2yf-exp(x))^2;
26
27 %       central differencing
28         d2yc=(yp2d-2*y+ym2d)/(2*delta)^2;
29         d2yce=d2yce+(d2yc-exp(x))^2;
30     end
31
32 %   find the norm of each set of errors
33     nd2yfe(i)=sqrt(d2yfe);
34     nd2yce(i)=sqrt(d2yce);
35 end
36
37 % plot the errors
38 hold on
39 axis('square')
40 loglog(deltai,nd2yfe)
41 loglog(deltai,nd2yce)
42 hold off
43 print -deps -solid second.eps
```

In each program listed above, the loop over `i` [4-35] considers values of Δ from $1.25 \times 0.8 = 1$ down to $1.25 \times 0.8^{101} \approx 1.6 \times 10^{-10}$. For each value of `delta` the loop over `j` [11-30] considers 101 values of `x` equally spaced [12] on $[0, 1]$. At each value of `x` it computes the [24] forward and [28] central difference approximations at that point, accumulates [25,29] the squares of the errors in the approximations, and [33-34] saves the square root of each sum. Thus each error curve plotted on the previous page actually shows the 2-norm of the error in the approximation over the 101 values of $x \in [0, 1]$, or

$$\bar{E}(\Delta_i) = \sqrt{\sum_{j=1}^{101} (\text{error}_j)^2}.$$

Theoretical arguments [4, §12.4.1] [5, §8.1] yield the following recommendations for $\Delta^\star$, which are marked on the graphs by small circles ∘ to show that they are close to the approximate values we found experimentally.

derivative	forward	central
$f'(x)$	$\sqrt[2]{\mathtt{u}} \approx 1.1 \times 10^{-8}$	$\sqrt[3]{\mathtt{u}} \approx 4.8 \times 10^{-6}$
$f''(x)$	$\sqrt[3]{\mathtt{u}} \approx 4.8 \times 10^{-6}$	$\sqrt[4]{\mathtt{u}} \approx 1.0 \times 10^{-4}$

Here $\mathtt{u} = 1.110223024625157 \times 10^{-16}$ is the **unit roundoff** (see §28.3.3). Of course not all functions are e^x, and not every x is in $[0, 1]$ (see Exercise 25.8.52) but most codes use fixed values for Δ anyway.

25.6.5 Computing Finite-Difference Approximations

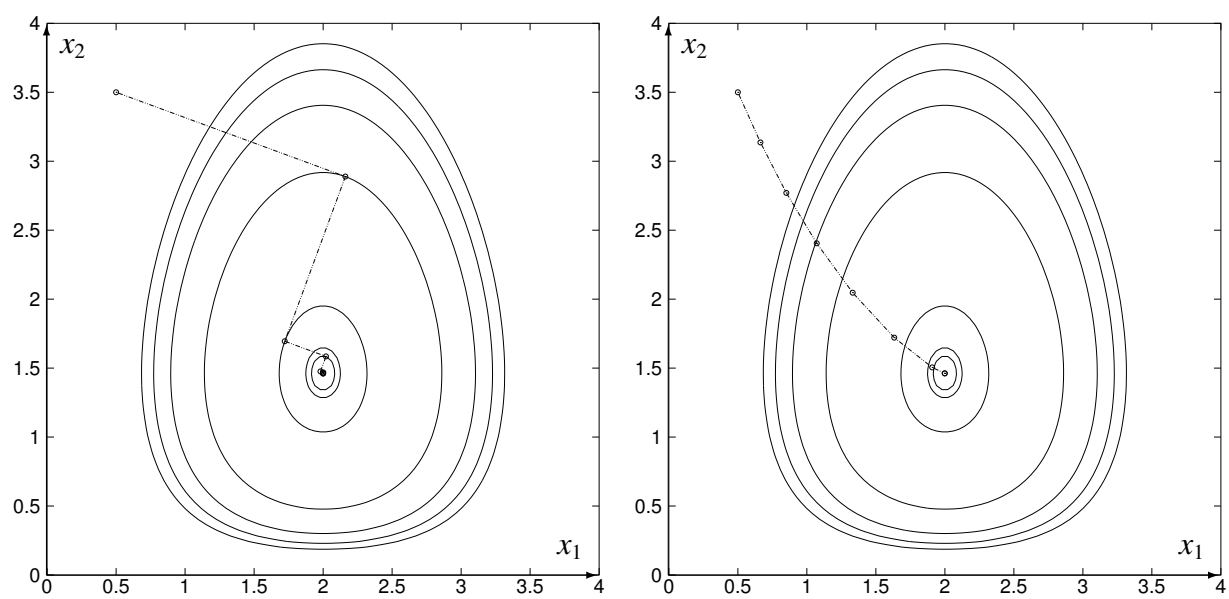
Using the formulas we derived and the recommended values of Δ, I wrote the MATLAB routines `gradcd.m` and `hesscd.m` listed on the next page; forward differencing can be implemented in a similar way. To test these routines I used them in the `eggg.m` and `eggh.m` routines listed below.

```
function f=egg(x)
  f=exp((x(1)-2)^2)*gamma(x(2));
end
```

```
function g=eggg(x)
  g=gradcd(@egg,x,2);
end
```

```
function h=eggh(x)
  h=hesscd(@egg,x,2);
end
```

Then I used `egg.m`, `eggg.m`, and `eggh.m` to solve the problem of §25.6.0 with `sd.m` and `ntfs.m`, whose convergence trajectories are plotted below over contours of the objective. Here finite difference derivatives work well for both steepest descent and Newton descent.



```
 1 function g=gradcd(fcn,x,ii)
 2 % approximate the gradient of function ii by central differencing
 3
 4   delta=4.80699951035563e-06;   % u^(1/3)
 5   n=size(x,1);                  % number of variables
 6   e=zeros(n,1);                 % e is a column of zeros
 7   g=zeros(n,1);                 % g is a column
 8
 9   for j=1:n                     % for each coordinate direction
10       e(j)=1;                   % make e the j'th unit vector
11       xpd=x+delta*e;            % step forward by delta
12       ypd=fcn(xpd,ii);          % find the function value there
13       xmd=x-delta*e;            % step back by delta
14       ymd=fcn(xmd,ii);          % find the function value there
15       g(j)=(ypd-ymd)/(2*delta); % find approximation
16       e(j)=0;                   % put e back to a zero vector
17   end                           % done with the directions
18
19 end
```

This routine estimates the partial derivatives $\partial f_{ii}/\partial x_j$ one at a time in the loop 9-17 over `j`. First 10 the `j`'th 1 in the unit vector `e` is filled in. Then `fcn` is used to find the function value at 12 $\mathbf{x}+\Delta\mathbf{e}$ and 14 $\mathbf{x}-\Delta\mathbf{e}$, and 15 the formula of §25.6.2 is used to approximate the gradient element. Finally 16 the 1 is removed from `e`, returning it to the zero vector.

```
 1 function h=hesscd(fcn,x,ii)
 2 % approximate the Hessian of function ii by central differencing
 3
 4   delta=1.02661016097495e-04;      % u^(1/4)
 5   n=size(x,1);                     % number of variables
 6   ei=zeros(n,1);                   % ei is a column of zeros
 7   ej=zeros(n,1);                   % ej is a column of zeros
 8
 9   for j=1:n                        % for each column
10       ej(j)=1;                     % make ej the j'th unit vector
11       for i=j:n                    % for each row in lower triangle
12           ei(i)=1;                 % make ei the i'th unit vector
13           xpp=x+delta*ei+delta*ej; % ++ step
14           fpp=fcn(xpp,ii);         % function value
15           xmp=x-delta*ei+delta*ej; % -+ step
16           fmp=fcn(xmp,ii);         % function value
17           xpm=x+delta*ei-delta*ej; % +- step
18           fpm=fcn(xpm,ii);         % function value
19           xmm=x-delta*ei-delta*ej; % -- step
20           fmm=fcn(xmm,ii);         % function value
21
22           h(i,j)=(fpp-fmp-fpm+fmm)/(4*delta^2); % find approximation
23
24           h(j,i)=h(i,j);           % fill in the symmetric element
25           ei(i)=0;                 % put ei back to a zero vector
26       end                          % done with rows for this column
27       ej(j)=0;                     % put ej back to a zero vector
28   end                              % done with columns
29
30 end
```

This routine uses two unit vectors, `ej` 7,10,27 and `ei` 6,12,25 corresponding to the columns and rows of the Hessian, and saves work 11 by exploiting Hessian symmetry 24.

25.6.6 Checking Gradients and Hessians

If the functions in the nonlinear program you want to solve have gradients and Hessians that can be computed from formulas, you will almost certainly want to use those in preference to finite-difference approximations. All you need to do is work out the formulas and code MATLAB functions to evaluate them, as we have done for numerous examples in earlier Chapters. Unfortunately, even for a problem in which the functions are very simple, it turns out to be surprisingly difficult to get the analytic derivatives and the MATLAB code exactly right. It is fortunate for me that, by comparing the output of my code to finite-difference approximations, I can find most of my mistakes.

The `gradtest.m` routine listed below checks a gradient routine that is coded in the way we have used for problems having constraints (the second argument of `fcn` or `grd` is the index of the function whose value or gradient is to be computed).

```
 1 function [reldif,mxdiff,mxdifx]=gradtest(fcn,grd,xl,xh,ii)
 2 % compare analytic to finite-difference gradient for function ii
 3
 4   n=size(xh,1);                    % number of variables
 5   x=zeros(n,1);                    % x is a column
 6   mxdiff=0;                        % no maximum difference yet
 7   for k=1:100                      % try 100 points in [xl,xh]
 8       for j=1:n                              % with
 9           x(j)=xl(j)+rand()*(xh(j)-xl(j));   % random
10       end                                    % components
11
12       ga=grd(x,ii);                % analytic gradient
13       gf=gradcd(fcn,x,ii);         % finite difference gradient
14       for j=1:n                    % compare each component
15           diff=abs(ga(j)-gf(j));   % difference between components
16           if(diff > mxdiff)        % keep track of the
17              mxdiff=diff;          % biggest difference
18              mxdifx=x;             % and where it occurred
19           end                      % done with comparison
20       end                          % done with components
21   end                              % done with trial points
22
23   nrm=norm(gradcd(fcn,mxdifx,ii)); % size of approximate gradient
24   if(nrm < 1e-6)                   % if it is tiny
25      reldif=-1;                    % relative error is meaningless
26   else                             % norm is not tiny
27      reldif=mxdiff/nrm;            % relative error is usefull
28   end
29
30 end
```

The routine works by repeatedly [7] generating a point at random within the variable bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ [8-10], finding [12] the supposed gradient of function `ii` and [13] its central-difference approximation at that point, and [14-20] remembering the absolutely largest difference between them. Then [23] it finds the norm of the approximate gradient at the point where the difference is greatest. If this number is too small [24] to use in computing a relative difference the routine [25] returns the meaningless value -1 for that quantity; otherwise [26-27] it re-

turns the relative difference between the analytic and finite-difference gradients, along with the maximum absolute error and the point where it happened. If the differences are small then `grd` is probably computing the gradient of function `ii` correctly (of course `fcn` and `grd` can also be consistent if both are wrong). The Octave session below shows that the gradients returned by `ek1g.m` for constraint `1` are close to those obtained by finite differencing function values from `ek1.m`, but that the gradients returned by `arch4g.m` are not.

```
octave:1> xl=[18-9/sqrt(2);21-13/sqrt(2)];
octave:2> xh=[18+9/sqrt(2);21+13/sqrt(2)];
octave:3> reldif=gradtest(@ek1,@ek1g,xl,xh,1)
reldif =  2.4235e-10
octave:4> reldif=gradtest(@ek1,@arch4g,xl,xh,1)
reldif =  13.262
```

Because `gradtest.m` uses central differencing, a relative error larger than 10^{-6} suggests a coding mistake in either the function routine or the gradient routine or both.

The `hesstest.m` routine listed below checks a Hessian routine in the same way that `gradtest.m` checks a gradient routine.

```
function [reldif,mxdiff,mxdifx]=hesstest(fcn,hsn,xl,xh,ii)
% compare analytic to finite-difference Hessian for function ii

  n=size(xh,1);                    % number of variables
  x=zeros(n,1);                    % x is a column
  mxdiff=0;                        % no maximum difference yet
  mxdifx=x;                        % if none return origin
  for k=1:100                      % try 100 points in [xl,xh]
      for j=1:n                            % with
          x(j)=xl(j)+rand()*(xh(j)-xl(j)); % random
      end                                  % components

    ha=hsn(x,ii);                  % analytic Hessian
    hf=hesscd(fcn,x,ii);           % finite difference Hessian
    for i=1:n                      % compare
    for j=1:n                      % each element
        diff=abs(ha(i,j)-hf(i,j)); % difference between elements
        if(diff > mxdiff)          % keep track of the
           mxdiff=diff;            % biggest difference
           mxdifx=x;               % and where it occurred
        end                        % done with comparison
    end                            % done
    end                            % with elements
  end                              % done with trial points

  nrm=norm(hesscd(fcn,mxdifx,ii)); % size of approximate Hessian
  if(nrm < 1e-6)                   % if it is tiny
     reldif=-1;                    % relative error is meaningless
  else                             % norm is not tiny
     reldif=mxdiff/nrm;            % relative error is usefull
  end

end
```

The Octave session on the next page illustrates its use.

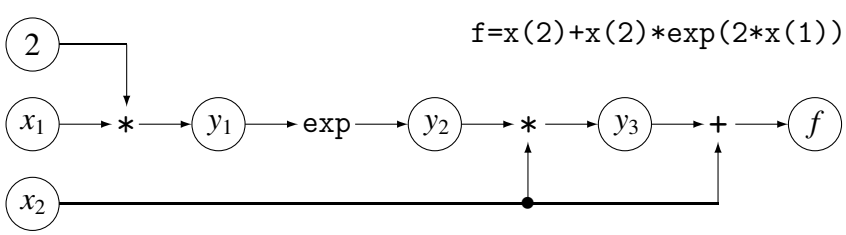

```
octave:1> xl=[0;0];
octave:2> xh=[3;3];
octave:3> reldif=hesstest(@p2,@p2h,xl,xh,1)
reldif =  3.1313e-08
octave:4> [reldif,mxdiff]=hesstest(@p1,@p2h,xl,xh,1)
reldif = -1
mxdiff =  2.0000
octave:5> quit
```

Here `3>` I found that Hessian matrices returned for constraint `1` of problem `p2` agree with their central-difference approximations, but that they do not agree with central difference approximations to the Hessian of the first constraint in `p1`. In that case `4>` `hesscd.m` returns -1 for `reldif` because the Hessian of `p1` happens to be near zero, but the large value of `mxdiff` reveals that `p1.m` and `p2h.m` do not describe the same problem. Because `hesstest.m` uses central differencing, a relative difference greater than 10^{-4} suggests a coding mistake in either the function routine or the Hessian routine or both.

Gradient and Hessian routines for which `gradtest.m` and `hesstest.m` report good agreement with central difference approximations can still be wrong, but if the agreement is *not* good then they are almost *certainly* wrong. However skilled you might be at finding derivatives and implementing their calculation in MATLAB, it is a good policy to test every gradient and Hessian routine you write. If your favorite minimization algorithm fails on a problem you think it should be able to solve, the trouble is probably in the function, gradient, or Hessian routine so your first step should be to test them for consistency.

25.6.7 Automatic Differentiation

When a computer program evaluates an arithmetic expression, it performs a sequence of operations each having one output and either one or two inputs. If the program is running on a single processor, these operations must be performed in order one at a time. For example, $f(x_1, x_2) = x_2 + x_2 e^{2x_1}$ could be evaluated by the sequence of operations pictured below.



This diagram is called a **parse tree** [21, §6.2.1]. A language compiler or a processor such as MATLAB generates internally a tabular representation of the parse tree to determine the sequence of machine instructions it will use in evaluating an expression. The operations shown in this parse tree are `*` and `+`, each of which takes two inputs, and `exp` which takes only one. The result of each operation except the last is an **intermediate variable**. In this parse tree the intermediate variables are y_1, y_2, and y_3.

Each intermediate or final variable is the result of a single arithmetic operation or elementary function invocation. This makes it easy to write down analytic expressions for the partial derivatives of an intermediate or final variable with respect to the one or two inputs of the operation that produced it. For the parse tree above we get the following derivatives.

$$
\begin{array}{rclrclrcl}
y_1 & = & 2x_1 & \dfrac{\partial y_1}{\partial x_1} & = & 2 & & & \\
y_2 & = & e^{y_1} & \dfrac{\partial y_2}{\partial y_1} & = & e^{y_1} & & & \\
y_3 & = & x_2 y_2 & \dfrac{\partial y_3}{\partial x_2} & = & y_2 & \dfrac{\partial y_3}{\partial y_2} & = & x_2 \\
f & = & x_2 + y_3 & \dfrac{\partial f}{\partial x_2} & = & 1 + \dfrac{\partial y_3}{\partial x_2} & \dfrac{\partial f}{\partial y_3} & = & 1
\end{array}
$$

Then we can use the chain rule to find $\nabla f(\mathbf{x})$, like this.

$$
\begin{aligned}
\frac{\partial f}{\partial x_1} &= \frac{\partial y_3}{\partial x_1} = \frac{\partial y_3}{\partial y_2} \times \frac{\partial y_2}{\partial y_1} \times \frac{\partial y_1}{\partial x_1} = x_2 \times e^{y_1} \times 2 = 2x_2 e^{2x_1} \\
\frac{\partial f}{\partial x_2} &= 1 + \frac{\partial y_3}{\partial x_2} = 1 + y_2 = 1 + e^{y_1} = 1 + e^{2x_1} \\
\nabla f(\mathbf{x}) &= \begin{bmatrix} 2x_2 e^{2x_1} \\ 1 + e^{2x_1} \end{bmatrix}
\end{aligned}
$$

The same techniques that a compiler uses to generate a parse tree can be used in a program that does **automatic differentiation** [5, §8.2] [4, §12.4.2] by performing calculations like the ones we did by hand above. The rules of differentiation that you learned in calculus are used to find the partial derivatives of the intermediate variables in the parse tree, and the chain rule is used to combine them and find the partial derivatives that make up the gradient or Hessian of the function. Some implementations carry out this process symbolically, so that the result is a formula for each partial derivative which we can then code into a routine to calculate the gradient numerically. Other implementations carry out the process numerically as part of a nonlinear program solver, producing each gradient or Hessian value as it is needed by the minimization routine without ever explicitly displaying formulas for the derivatives.

When the process is carried out symbolically it is conceptually equivalent to using a computer algebra package such as Maple to find formulas for the partial derivatives. However, some programs that have been developed for symbolic differentiation can read the computer source code of a routine for calculating $f(\mathbf{x})$ and generate computer source code for a routine to calculate $\nabla f(\mathbf{x})$, so that no human intervention is required. This eliminates coding errors as well as errors in calculus.

Automatic differentiation is most useful for problems in which the functions are too complicated to easily differentiate by hand or the derivatives are too complicated to easily code by hand. Unfortunately these are precisely the circumstances that yield a huge parse tree, cumbersome to store and expensive to process, and this has led to the development of an extensive body of theory and technique for managing the parse tree and constraining its growth. Practical software tools have been developed for both symbolic and numerical automatic differentiation [5, p217] of function routines coded in FORTRAN, C, C++, and MATLAB, and this technology remains an active area of research in computer science so future improvements are likely.

25.7 Large Problems

The table in §25.1 lists several routines for nonlinear optimization. Which would you use to solve this problem? For reasons that will become clear its name is `big` (see §28.7.41).

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & f_0(\mathbf{x}) = \sum_{j=1}^{n} a_j(x_j - 1)^2 \\ \text{subject to} \quad & \min\left(\frac{1}{a_j}, a_j\right) \le x_j \le \max\left(\frac{1}{a_j}, a_j\right), \quad j = 1 \ldots n. \end{aligned}$$

For a given vector of nonzero constants $[a_1, \ldots, a_n]^\top$ the objective is quadratic and the constraints are simple bounds, so any of our routines that can handle inequalities would seem suitable. But is that still true if the number of variables is, say, 1 million? In that case an $n \times n$ matrix has 10^{12} elements, and to store them as floating-point numbers would require some 8 terabytes of memory. This effectively rules out `qpin.m`, which uses an $n \times n$ matrix `Q` to describe the quadratic objective and a matrix `A`, here $2n \times n$, to describe the linear constraints. It also rules out `ntin.m`, `barrier.m`, `emiqp.m`, and `iqp.m`, all of which use Hessian matrices, as well as `nlpin.m` and `nlpinp.m`, which use Jacobians. The ellipsoid algorithm routines `ea.m` and `wander.m` are out of the question too, because they store an ellipsoid matrix and because their convergence constant would differ from 1 by only 5×10^{-13}.

To solve problems that are large we need methods whose storage requirements and running times grow no faster than linearly with the number of variables and constraints. Methods like that are effective only for problems that also have special properties.

25.7.1 Problem Characteristics

A few of the applications of nonlinear programming listed in the table of §8.4 routinely have very large instances, among them **machine learning** [7] [177] formulations such as these three which we have studied: compressed sensing (§1.8), regression (§8.6), and classification (§8.7). I contrived the `big` example to exhibit in a simplified way several characteristics that are typical of such problems.

- SIMPLICITY. An instance of the `big` problem is completely characterized by the single constant vector $\boldsymbol{a}$, the functions are easy to compute, and finding a numerical solution would be straightforward if n were small.

- STRUCTURE. This problem is **component separable** [17, §4.4.2] because each term in the objective and each pair of constraints involves only a single variable. The constraints all look alike, and the terms in the objective function all look alike.

- CONVEXITY. If the a_j are positive this is a convex program with a strictly convex objective, and if the a_j are neither very big nor very small it is well-scaled.

- SMOOTHNESS. The objective and constraints of `big` are continuous functions of $\mathbf{x}$ that can be computed from formulas, as are all of their derivatives; in other problems from this class the objective might include nonsmooth terms that can be handled by the techniques described in §1.5.3.

The technical term-of-art for nonlinear programs having these attributes is that they are **nice** [14]. The craft of solving a large application problem consists of formulating a model that is as nice as possible without being completely unrealistic [2, §2.7] and then devising a method that takes advantage of that niceness in such a way that it can work for large n.

25.7.2 Coordinate Descent

One way to exploit the nice attributes of our `big` problem is to start from a feasible point, do a line search in the x_1 direction between the given bounds on x_1, then search from that point in the x_2 direction between the given bounds on x_2, and so on (see Exercise 14.8.11). This **cyclic coordinate descent** algorithm [5, §9.3] [1, §8.5] might not find $\mathbf{x}^\star$ even if $f_0(\mathbf{x})$ is strictly convex, and if it does that might be only after cycling through the coordinates multiple times, but it does have the virtue of not needing to store an $n \times n$ matrix. Because the problem is separable the directional derivative in iteration k is simply

$$\frac{\partial f_0}{\partial x_j} = 2a_j(x_j^k - 1)$$

so we can use a bisection line search without ever having to compute or store a gradient vector. To solve the problem using this idea I wrote the MATLAB program `big.m` listed on the next page. It assumes that x_j^{L} corresponds to $\alpha = 0$ in the line search and that x_j^{H} corresponds to $\alpha = 1$.

This routine allows for the possibility of doing `cmax` cycles `5-22` through the coordinate directions; in each cycle it `6-21` searches in each of the `n` coordinate directions. It begins each search by `7` setting $\alpha^{\mathrm{L}} = 0$ and $\alpha^{\mathrm{H}} = 1$. Next it uses the formulas in the problem statement to compute the bounds `8` x_j^{L} and `9` x_j^{H}, and finds `10` the $\alpha \in [0, 1]$ corresponding to the given $\mathbf{x}^0$.


```
 1 function x=big(a,x,cmax,smax)
 2 % solve the big problem using cyclic coordinate descent
 3
 4 n=size(x,1);                        % get number of variables
 5 for c=1:cmax                        % do cmax cycles
 6     for j=1:n                       % in each coordinate direction
 7         al=0; ah=1;                 % search for alpha in [0,1]
 8         xl=min(a(j),1/a(j));        % which keeps x between xl
 9         xh=max(a(j),1/a(j));        % and xh
10         alpha=(x(j)-xl)/(xh-xl);    % alpha at start for cycle c
11         for s=1:smax                % do bisections
12             fp=2*a(j)*(x(j)-1);     % directional derivative
13             if(fp < 0)              % is min to the right?
14                al=alpha;            % if so increase lower bound
15             else                    % no; min is to the left
16                ah=alpha;            % decrease upper bound
17             end
18             alpha=(al+ah)/2;        % bisect interval in alpha
19             x(j)=xl+alpha*(xh-xl);  % find corresponding x(j)
20         end                         % bisections done
21     end                             % coordinates done
22 end                                 % cycles done
```

Then it does exactly `smax` iterations of the bisection line search algorithm 11-20 using 12 the formula given above to find the directional derivative. Convergence tests could be used in the loop over `s`, at the price of making the code more complicated. This routine does not store any matrices, and the only vectors it uses are `a` and `x` (`xl` and `xh` are scalars).

To study the behavior of `big.m`, I solved two $n = 2$ instances of the problem as shown in the Octave session to the right, and with a different program I plotted the convergence trajectories shown on the next page. Setting `a=[2,3]` 1> makes this the first problem instance.

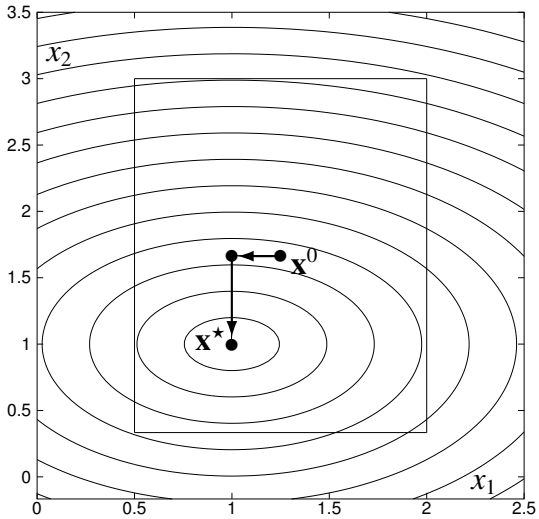
$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & f_0(\mathbf{x}) = 2(x_1-1)^2 + 3(x_2-1)^2 \\ \text{subject to} \quad & \tfrac{1}{2} \le x_1 \le 2 \\ & \tfrac{1}{3} \le x_2 \le 3 \end{aligned}$$

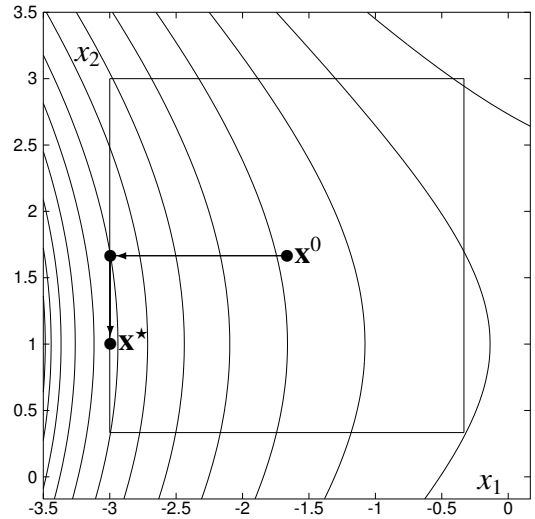
This convex program has $\mathbf{x}^\star = [1, 1]^\top$ interior to the bounds. Setting `a=[-3,3]` 4> makes this the second problem instance.

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & f_0(\mathbf{x}) = -3(x_1-1)^2 + 3(x_2-1)^2 \\ \text{subject to} \quad & -3 \le x_1 \le -\tfrac{1}{3} \\ & \tfrac{1}{3} \le x_2 \le \ \ 3 \end{aligned}$$

This objective is nonconvex but cyclic coordinate descent works anyway, finding $\mathbf{x}^\star = [-3, 1]^\top$ in the boundary of the feasible set. It is also possible to solve this problem with a rougher line search 8>,10> but only if several cycles are used.

```
octave:1> a=[2,3];
octave:2> x=[5/4;5/3];
octave:3> x=big(a,x,1,20)
x =

   1.00000
   1.00000

octave:4> a=[-3,3];
octave:5> x=[-5/3;5/3];
octave:6> x=big(a,x,1,20)
x =

  -3.00000
   1.00000

octave:7> x=[-5/3;5/3];
octave:8> x=big(a,x,1,10)
x =

  -2.99870
   0.99870

octave:9> x=[-5/3;5/3];
octave:10> x=big(a,x,3,10)
x =

  -3.0000
   1.0000
```



Next I tried solving progressively larger problem instances, as shown in the Octave sessions below. On the left I chose each a_j at random from the interval $[2, 3]$ and used a starting point having each element equal to $\frac{5}{4}$, the midpoint of the interval $[\frac{1}{2}, 2]$. On the right I chose each a_j at random from the interval $[-3, 3]$ and initialized each x_j to the midpoint of the resulting bounds on that variable.

```
octave:1> n=1e2;
octave:2> a=2+rand(n,1);
octave:3> x=(5/4)*ones(n,1);
octave:4> tic;x=big(a,x,1,30);toc
Elapsed time is 0.08875 seconds.
octave:5> n=1e3;
octave:6> a=2+rand(n,1);
octave:7> x=(5/4)*ones(n,1);
octave:8> tic;x=big(a,x,1,30);toc
Elapsed time is 0.87576 seconds.
octave:9> n=1e4;
octave:10> a=2+rand(n,1);
octave:11> x=(5/4)*ones(n,1);
octave:12> tic;x=big(a,x,1,30);toc
Elapsed time is 8.73533 seconds.
octave:13> x
x =

   1.00000
   1.00000
   1.00000
   1.00000
   1.00000
   :
```

```
octave:1> n=1e2;
octave:2> a=-3+6*rand(1,n);
octave:3> a=-3+6*rand(n,1);
octave:4> xl=min(1./a,a);
octave:5> xh=max(1./a,a);
octave:6> x=(xl+xh)/2;
octave:7> tic;x=big(a,x,1,30);toc
Elapsed time is 0.087055 seconds.
octave:8> n=1e3;
octave:9> a=-3+6*rand(n,1);
octave:10> xl=min(1./a,a);
octave:11> xh=max(1./a,a);
octave:12> x=(xl+xh)/2;
octave:13> tic;x=big(a,x,1,30);toc
Elapsed time is 0.865 seconds.
octave:14> n=1e4;
octave:15> a=-3+6*rand(n,1);
octave:16> xl=min(1./a,a);
octave:17> xh=max(1./a,a);
octave:18> x=(xl+xh)/2;
octave:19> tic;x=big(a,x,1,30);toc
Elapsed time is 8.645 seconds.
octave:20> [x,a,xl,xh]
ans =

   1.0000e+00  2.6742e+00  3.7395e-01  2.6742e+00
  -1.6120e+00 -1.6120e+00 -1.6120e+00 -6.2036e-01
   1.0000e+00  6.3547e-01  6.3547e-01  1.5736e+00
   1.0000e+00  1.0572e+00  9.4592e-01  1.0572e+00
  -2.5717e+00 -2.5717e+00 -2.5717e+00 -3.8885e-01
   :
```


In both experiments the execution time of `big.m` is proportional to n, so if we continue to use MATLAB we can expect to solve the $n = 10^6$ case conjectured at the beginning of this Section in about 15 minutes of CPU time. To store $\boldsymbol{a}$ and $\mathbf{x}$ for a problem of that size will require only about 16 megabytes of memory, well within the capacity of modern computers.

Coordinate descent has several variants differing in the rule that is used to determine the order in which the directions are searched [1, §8.5].

name	order of search directions
cyclic	$1, 2, \ldots, n$ and repeat
Aitkin double sweep	$1, 2, \ldots, n, n-1, n-2, \ldots, 1$ and repeat
Gauss-Southwell	search in the direction of the largest $\left\lvert \partial f_0(\mathbf{x}^k)/\partial x_j \right\rvert$
random	use a random permutation of the indices

25.7.3 Method Characteristics

To be tractable large problems must be nice, so they typically have the characteristics described in §25.7.1. Methods that are practical for such problems must exploit those characteristics, so they also tend to have stereotypical attributes. Our toy implementation of cyclic coordinate descent is far from sophisticated, but even it exhibits the other properties described below.

- Algorithms for big problems are usually based on simple ideas, and employ data structures that grow only linearly with n.
- They exploit the special structure of the model they are targeted to solve. This includes the convexity or strict convexity of the functions, the nature of the constraints (simple bounds, inequalities, equalities), the algebraic form of the objective function (e.g., quadratic) and of the constraint functions (e.g., linear), any variable bounds that can be deduced in the formulation process, and any regularity or pattern in the coefficients of the objective or constraints. Even if a problem is not component-separable like `big` it might be **block separable** [17, §4.4.1] so that it has **partially separable functions** [5, §7.4], permitting various economies such as replacing a large Hessian by several much smaller sparse matrices.
- They are sophisticated in the details of their implementation, employing highly-efficient algorithms for numerical linear algebra [17, §4.2] and, if matrices are involved at all, sparse matrix techniques [87] [100, §11.6] to conserve memory and processor cycles. They carefully coordinate the iteration limits, tolerances, and other parameters used in their sub-algorithms, and [17, §3.4.4 & §4.3.2] adjust some tolerances as the iterations of the main algorithm progress. They are invariably coded in a compiled language such as FORTRAN, C, or C++ rather than in an interpreted one such as MATLAB or Python.

- They use parallel processing if that is possible. If the problem is separable and the computing environment supports the concurrent use of multiple processors [100, §16.2] (e.g., in a distributed-computing cloud) a method might execute several parts of the algorithm in parallel.

- Their goal is improvement, not perfection. Nice models often end up being only approximate anyway, so imprecise solutions are good enough and rough tolerances can often be used in obtaining them [17, §3.2.2]. In most settings that give rise to large problems, an optimization result that permits even a small improvement over current practice might be considered a success.

Cyclic coordinate descent happened to work for our `big` problem, but it cannot be used with equality constraints. The table below lists some other approaches whose memory requirements scale in an approximately linear way with n. Some of these methods use Hessian matrices that are sparse, or involve matrix-vector products that can be calculated without storing the matrix (this idea was first mentioned in §14.4).

method	$\leq$	$=$	references
steepest descent	□	□	§10.4
Fletcher-Reeves	□	□	§14.5
Polak-Ribière	□	□	§14.6
Hessian-free Newton	□	□	[5, p170]
limited-memory quasi-Newton	□	□	[5, §7.2] [4, §13.5]
sparse quasi-Newton	□	□	[5, §7.3]
ADMM	□	■	§20.3
gradient projection	■	□	[5, §16.7]
block coordinate descent	■	□	[2, §3.7]

The tail that is wagging the dog of mathematical programming at this moment in history is machine learning, and it is constantly fueling the development of new algorithms for large problems.

25.7.4 Semi-Analytic Results

Some nonlinear programs can be solved analytically, yielding $\mathbf{x}^\star$ as a vector of numbers or as a vector of algebraic expressions involving the problem data. Even when this is not possible, if the problem is highly-structured (as many nice problems are) it might be possible to construct its solution by applying some rules rather than by performing an explicit numerical minimization. I mentioned in §1.8 that the compressed sensing problem has such a **semi-analytic** solution, and the output from our §25.7.2 experiments with `big.m` suggests that a set of rules might yield $\mathbf{x}^\star$ for that problem too.

You probably noticed that when I generated $a_j \in [2,3]$ the answer `big.m` found with $n = 10^4$ was $\mathbf{x}^\star = [1,1,\ldots,1]^\top$, the unconstrained minimizing point for $f_0(\mathbf{x})$. Of course if $a_j > 0$ then the interval defined by the bounds *always* contains 1; this is illustrated for $n = 2$ by the left contour diagram of §25.7.2. If $a_j > 0$, then $x_j^\star = 1$.

When $a_j < 0$ it appears that $x_j^\star = x_j^L$, and of course this makes sense too. If, for example, $a_j = -2$ then $x_j^\star$ must be negative, because $x_j \in [-2, -\frac{1}{2}]$. The objective term we are trying to minimize is $-2(x_j - 1)^2$, so we should make x_j as negative as possible, which puts it at its lower bound. If $a_j < 0$, then $x_j^\star = \min(a_j, 1/a_j)$.

Just by thinking about the problem we could (as perhaps you did from the beginning) deduce, without using the theory of nonlinear programming or doing any numerical calculations at all, that

$$x_j^\star = \begin{cases} 1 & \text{if } a_j > 0 \\ \min\left(a_j, 1/a_j\right) & \text{if } a_j < 0. \end{cases}$$

Often a little insight can make a daunting but highly-structured problem trivial. No one has yet succeeded in teaching me how to be clever, so I will not presume to teach that to you. However, some authors who *are* clever have made the attempt; for example, the great mathematician George Polya called the sort of argument we have just used **plausible reasoning**. He claims [173, p vi] that one can learn how to use plausible reasoning only by imitation and practice, but then he goes on to elaborate general theories of mathematical insight and [174] discovery. If you are engaged in the search for clever reformulations of highly-structured large problems you might enjoy reading what he has to say.

25.7.5 Nasty Problems

Earlier I claimed that for a large problem to be tractable it must be nice, but what if a large problem whose solution would be valuable happens to be downright nasty? In practice people try every algorithm that seems plausible, ignoring the warnings printed on the package, and hope for the best [167]. This is what we did when we tried cyclic coordinate descent on the `big` problem with some of the $a_j < 0$, and found $\mathbf{x}^\star$ anyway. Of course it is always less risky to use a special-purpose method that is designed for the specific nastiness in question.

Nondifferentiability is a nastiness endemic to many important models. We have reformulated our way around it on several occasions, but sometimes those tricks do not work. The general-purpose classical subgradient methods for convex nonsmooth programming are hard to use, as I mentioned in §20.1, so extravagant efforts have been (and are being) devoted to the construction of special-purpose algorithms for particular nonsmooth problems that are otherwise nice. These include [17, §6] clever incarnations of the ADMM approach discussed in §20.3, [2, §3.6] **proximal algorithms** such as [102] **mirror descent**, and [122] **smoothing methods**. All of these ideas, and the interesting applications that motivate their development, are, regrettably, beyond the scope of this introduction.

25.8 Exercises

25.8.1 [E] This Chapter concerns various issues that arise in solving real nonlinear programs. What are some of these issues? Why did I put off discussing them until now?

25.8.2 [E] Are the codes listed in §25.1 likely to solve any and all nonlinear programs you might encounter? Are the black-box codes described in §8.3.1 likely to do so? Explain.

25.8.3 [E] If you encounter a nonlinear program that cannot be solved by any code that you know of or can find by diligently searching the internet, what should you do? (a) start checking fortune cookies for the optimal point; (b) change your major to Art History; (c) use everything you have learned to construct an algorithm that fits the problem.

25.8.4 [E] Of the nonlinear programming codes that we have developed, which are made to solve problems having equality constraints? Which are made to solve problems having inequality constraints?

25.8.5 [E] Some algorithms have a natural extension that permits them, at least in principle, to handle both equality and inequality constraints. Give one example.

25.8.6 [P] Write a MATLAB routine `penbar.m` to solve problems having both equality and inequality constraints by minimizing

$$\Omega(\mathbf{x};\mu) = f_0(\mathbf{x}) + \mu \sum_{i=m_i+1}^{m_i+m_e} [f_i(\mathbf{x})]^2 - \frac{1}{\mu} \sum_{i=1}^{m_i} [\ln[-f_i(\mathbf{x})]$$

in a sequence of unconstrained optimizations, each starting at the optimal point of the previous one and using a value of μ twice the previous value. Test your code by using it to solve this nonlinear program.

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 \\ \text{subject to} & f_1(\mathbf{x}) & = & 1 - x_1 - x_2 \le 0 \\ & f_2(\mathbf{x}) & = & 1 + x_1 - x_2 = 0 \end{array}$$

25.8.7 [E] What is *constraint affinity*?

25.8.8 [E] Describe one way in which an algorithm with an affinity for equality constraints might be extended to also handle inequality constraints. Describe one way in which an algorithm with an affinity for inequality constraints might be extended to also handle equality constraints. Are the resulting extended algorithms likely to be as robust as their unextended progenitors? Explain.

25.8.9 [P] Write a MATLAB routine `sqpie.m` that combines the ideas from `sqp.m` and `iqp.m` to solve problems having both inequality and equality constraints. Test your code by using it to solve the nonlinear program of Exercise 25.8.6.

25.8.10 [E] If a nonlinear program has several equality constraints but only one inequality constraint, suggest a way of solving the problem with a code that can handle only equality constraints.

25.8.11 [P] The diameter of a polygon is the greatest distance between two of its vertices. Unit-diameter polygons with an odd number of sides have maximum area when they are regular, but when the number of sides is even the largest polygon need not be the regular one. The area of the largest unit-diameter octagon, approximately 0.7268684827517009, is the optimal value of the following nonlinear program [9, §3], and the coordinates of the irregular octagon's vertices can be deduced from the elements of $\mathbf{x}^\star$ and $\mathbf{y}^\star$.

$$\begin{aligned}
\underset{\mathbf{x}\in\mathbb{R}^5,\,\mathbf{y}\in\mathbb{R}^5}{\text{maximize}} \quad & \tfrac{1}{2}\big[(x_2+x_3-4x_1)y_1+(3x_1-2x_3+x_5)y_2+(3x_1-2x_2+x_4)y_3 \\
& \qquad +(x_3-2x_1)y_4+(x_2-2x_1)y_5\big]+x_1 \\
\text{subject to} \quad & (x_1-x_2)^2+(y_1-y_2)^2 \le 1 \\
& (-x_1+x_3-x_5)^2+(y_1-y_3+y_5)^2 \le 1 \\
& (x_1-x_2+x_4)^2+(y_1-y_2+y_4)^2 \le 1 \\
& (x_1-x_3)^2+(-y_1+y_3)^2 \le 1 \\
& (2x_1-x_2-x_3+x_5)^2+(-y_2+y_3-y_5)^2 \le 1 \\
& (2x_1-x_2)^2+y_2^2 \le 1 \\
& (x_1-x_2)^2+(y_1-y_2-1)^2 \le 1 \\
& (2x_1-x_2-x_3)^2+(-y_2+y_3)^2 \le 1 \\
& (x_3-x_5)^2+(-y_3+y_5)^2 \le 1 \\
& (-x_1+x_3-x_5)^2+(y_1-y_3+y_5-1)^2 \le 1 \\
& (2x_1+x_3-x_5)^2+(-y_3+y_5)^2 \le 1 \\
& (2x_1-x_2-x_3+x_4+x_5)^2+(-y_2+y_3+y_4-y_5)^2 = 1 \\
& (-2x_1+x_2-x_4)^2+(y_2-y_4)^2 \le 1 \\
& (x_1-x_2+x_4)^2+(y_1-y_2+y_4-1)^2 \le 1 \\
& (x_1-x_3)^2+(1-y_1+y_3)^2 \le 1 \\
& (x_2-x_4)^2+(y_2-y_4)^2 \le 1 \\
& (2x_1-x_3)^2+y_3^2 \le 1 \\
& (2x_2-x_2-x_3+x_4)^2+(-y_2+y_3+y_4)^2 \le 1 \\
& x_2-x_3 \ge 0 \\
& x_j^2+y_j^2 = 1 \qquad j=1\ldots5 \\
& 0 \le x_1 \le \tfrac{1}{2} \\
& 0 \le x_j \le 1 \qquad j=2,3,4,5
\end{aligned}$$

Notice that this problem has two equality constraints, one of which is difficult to remove algebraically. (a) Using an algorithm of your choice, compute a numerical solution to this problem. (b) What is the area of a regular unit octagon?

25.8.12 [E] Does a convex program necessarily have a unique optimal point? Does a nonlinear program that is not a convex program necessarily have multiple optimal points? Explain.

25.8.13 [E] Why does `ntrs.m` work better than `ntfs.m` for solving the `h35` problem? Why does `ntw.m` work better than `ntfs.m` for solving that problem?

25.8.14 [E] What is a *globalization strategy* and why might an algorithm designer wish to use one? Name four globalization strategies.

25.8.15 [P] One way to globalize an NLP solver is by searching the line between $\mathbf{x}^k$ and $\mathbf{x}^k + \mathbf{d}^k$, where $\mathbf{d}^k$ is a full step, for an optimal step of length $\alpha^\star < 1$. Then the algorithm can use $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^\star \mathbf{d}^k$ rather than taking the full step. (a) Modify `grg.m` to use a Wolfe line search in this manner on the tangent hyperplane, rather than taking a full steepest-descent step. (b) Use the resulting code to solve the `grg2` and `grg4` problems. How does this version perform, compared to the original `grg.m`?

25.8.16 [E] How does the trust-region algorithm described in §17.3 differ from the restricted-step algorithm described in §17.1?

25.8.17 [P] One way to globalize an NLP solver is by using the trust region idea. (a) Of the NLP routines listed in the table of §25.3.1, which could be modified in a simple way to use a trust region approach? (b) Modify `penalty.m` to use `trust.m` instead of `ntrs.m` for solving the subproblems. (c) Use the resulting code to solve the `p1` and `p2` problems. How does this version perform, compared to the original `penalty.m`?

25.8.18 [H] In the trust-region algorithm of §17.3, if the full modified Newton step exceeds the radius of the trust region we move to the point that minimizes the quadratic model of the function over the trust-region boundary. (a) Could the trust-region idea be used in a setting where the desired descent direction is instead the direction of steepest descent? (b) Could the trust-region idea be used in a setting where the model used to approximate $f_0(\mathbf{x})$ near $\mathbf{x}^k$ is linear instead of quadratic? (c) If $\mathbf{d}^k$ is the direction of steepest descent and the model is $q(\mathbf{x}^k + \mathbf{p}) = f_0(\mathbf{x}^k) + \nabla f_0(\mathbf{x}^k)^\top \mathbf{p}$, explain how the method would find $\mathbf{p}^\star$. Would it be possible to find $\mathbf{p}^\star$ by using a dogleg approximation?

25.8.19 [E] Explain in detail why `sd.m` fails to solve the unconstrained optimization of §25.4 when $s = 10^{14}$, making reference to the graph that is presented there to illustrate the phenomenon.

25.8.20 [P] In §25.4 we found that `sd.m` fails to solve this unconstrained optimization when $s = 10^{14}$.

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) = s x_1^2 + x_2^2$$

(a) By using `sd.m` to solve the problem for values of $s \in [10^0 \ldots 10^{14}]$, find the smallest value of s at which the algorithm fails. (b) Use `sdfs.m` to attempt the problem with $s = 10^{14}$, and explain the result.

25.8.21 [E] What does it mean to say that an unconstrained optimization is *poorly scaled*?

25.8.22 [E] Describe the sensitivity to variable scaling of the methods we have studied for unconstrained optimization. What are some effects of poor scaling?

25.8.23 [E] What is *diagonal scaling*? If we find the optimal point $\mathbf{y}^\star$ for a problem that has been diagonally scaled using the matrix $\mathbf{D}$, how can we recover $\mathbf{x}^\star$?

25.8.24 [H] If it is known that the optimal point of an unconstrained optimization will have components $x_1^\star \in [1000, 2000]$ and $x_2^\star \in [0.01, 0.1]$, find a diagonal matrix $\mathbf{D}$ that can be used to compute scaled variables y_1 and y_2 each ranging from -1 to 1.

25.8.25 [E] How can you tell whether the constraints of a problem are poorly scaled?

25.8.26 [P] In §25.4.2 we studied a constrained optimization on which `auglag.m` fails if the constraint is poorly scaled. Try solving the problem for $s = 1$, $s = 10^{14}$, and $s = 10^{-14}$ with (a) `grg.m`; (b) `ntlg.m`; (c) `sqp.m`. (d) `rsdeq.m`; (e) `rneq.m`.

25.8.27 [E] Find, among the routines that are listed in §25.1, one that returns when a step length is small enough and one that returns when a gradient is small enough.

25.8.28 [E] Explain the difference between an *absolute* and a *relative* measure of solution error. What advantages and drawbacks does each have?

25.8.29 [H] There is a measure of step length that avoids the problems associated with using $\|\mathbf{x}^{k+1} - \mathbf{x}^k\|$ and $\|\mathbf{x}^{k+1} - \mathbf{x}^k\|/\|\mathbf{x}^k\|$. (a) What is it? (b) How can you use the same idea to construct a measure of gradient norm that is neither absolute nor relative?

25.8.30 [P] In §25.5, I described a way of measuring the difference between two floating-point numbers by comparing their bit strings. How many bits must match if the two numbers are to have (a) the same algebraic sign; (b) the same sign and biased exponent; (c) the same sign and exponent and the same p leading fraction bits; (d) exactly the same value. (e) Using MATLAB or another programming language of your choice, write a routine that returns e, the number of least-significant bits in which two 8-byte values differ. (f) How can this routine be used to find an error $\mathcal{E}$ that measures the difference between two vectors whose components are floating-point numbers?

25.8.31 [E] Why in solving a nonlinear program might it be desirable to approximate derivatives by finite differencing? Write down all the reasons you can think of.

25.8.32 [H] Suppose finite differencing is used to approximate the gradient of a function that is not smooth. (a) How might the approximate gradient differ from the true one? Give an example to illustrate your answer. (b) Do you think a gradient-based optimization method is more likely to solve a problem that is not smooth if analytic derivatives are used, or if finite difference approximations are used? Give an argument or example to support your answer.

25.8.33 [P] Write down the Taylor's series expansion of $f(x) = e^x$ about the point $x = 0$, to obtain

$$f(\Delta) = f(0) + \Delta f'(0) + \frac{\Delta^2}{2} f''(\xi).$$

At what point $\xi \in [0, \Delta]$ is this equation satisfied? Find ξ numerically if $\Delta = 1$.

25.8.34 [E] What assumptions did we make in deriving forward-difference formulas to approximate $f'(x)$ and $f''(x)$?

25.8.35 [E] Write down the formula we derived for the forward-difference approximation of (a) $\partial f/\partial x_i$; (b) $\partial^2 f/\partial x_i \partial x_j$. (c) In the formulas of §25.6.1 what does the notation $\mathbf{e}^i$ mean? (d) Why are $n + 1$ function evaluations required to approximate a gradient using forward differencing? (e) Why are $(\frac{1}{2}n + 1)(n + 1)$ function evaluations required to approximate a Hessian using forward differencing?

25.8.36 [E] When forward differencing is used to approximate a derivative, the truncation error depends on the increment Δ. (a) If Δ is doubled, what happens to the truncation error in the approximation? (b) What does it mean to say that some quantity is "$\mathcal{O}(\Delta)$?"

25.8.37 [H] By using the Taylor's series expansion for $f(x)$, show that the worst-case truncation error in a forward-difference approximation of $f'(x)$ is proportional to Δ. Can the error ever be zero?

25.8.38 [E] Why does central-differencing have a smaller truncation error than forward-differencing, for the same increment Δ? Give a plausibility argument based on a picture, rather than an abstract proof based on equations.

25.8.39 [H] Show that a central-difference derivative approximation is exact if $f(x)$ is a quadratic function. Is a central-difference Hessian approximation also exact?

25.8.40 [E] Write down the formula we derived for the central-difference approximation of (a) $\partial f/\partial x_i$; (b) $\partial^2 f/\partial x_i \partial y_j$. (c) Why are $2n$ function evaluations required to approximate a gradient using central differencing? (d) Why are $2n(n + 1)$ function evaluations required to approximate a Hessian using central differencing?

25.8.41 [E] When central differencing is used to approximate a derivative, the truncation error depends on the increment Δ. (a) If Δ is doubled, what happens to the truncation error in the approximation? (b) Of what order is the truncation error in this approximation?

25.8.42 [H] By using the Taylor's series expansion for $f(x)$, show that the truncation error in a central-difference approximation of $f'(x)$ is proportional to Δ^2.

25.8.43 [E] Is it ever faster to approximate a gradient or Hessian by finite differencing of function values than it is to evaluate a formula for the elements of the gradient or Hessian? If yes, when? If no, why not?

25.8.44 [E] There are algorithms that can approximate the derivative of a function much more accurately than central differencing does, by using more function evaluations. Why are these methods seldom used in numerical optimization?

25.8.45 [E] Finite-difference derivative approximations are inaccurate due to both truncation error and roundoff error. (a) Explain the difference between these errors. (b) How does each depend on the finite-difference interval Δ? (c) How can we find the value of Δ that minimizes the total error in a derivative approximation?

25.8.46 [H] In §25.6.4 we derived expressions for the stationary points of $E(\Delta)$ in forward- and central-difference derivative approximations. (a) Show that each $\Delta^\star$ is a unique minimizing point of the corresponding total error. (b) The expressions for total error involve constants a, b, c, and d. How can these numbers be found?

25.8.47 [E] In §25.6.4 we derived a simple error model that accurately predicts the behavior of forward-difference and central-difference derivative approximations. (a) Explain the reasoning that we used and the piecewise-linear error curves that result. (b) The error in a central-difference approximation grows faster as Δ is increased beyond its optimal value than does the error in a forward-difference approximation. Why? (c) According to this model, can a forward-difference approximation *ever* be more accurate than a central-difference approximation, for the same Δ? Explain.

25.8.48 [P] In §25.6.4 we used the MATLAB programs `first.m` and `second.m` to plot curves of $E(\Delta)$ versus Δ. (a) Modify each program to enlarge the vertical axis of the graph it generates. (b) Use the enlarged graphs to estimate numerical values for the constants a, b, c, and d in the error model we derived. (c) Use those numbers to estimate $\Delta^\star$ for each of the four cases shown, from the formulas we obtained by minimizing $E(\Delta)$ analytically. (d) Use those numbers to estimate $\Delta^\star$ for each of the four cases shown, by calculating the intersection points of the straight lines in the ideal graph of the error model. Are your estimates close to the values of $\Delta^\star$ we found experimentally?

25.8.49 [E] Explain how the error $\bar{E}(\Delta)$ is determined in the `first.m` and `second.m` programs of §25.6.4.

25.8.50 [E] In terms of the unit roundoff u, what values of $\Delta^\star$ are recommended for approximating $f'(x)$ and $f''(x)$ by the forward and central difference formulas we derived?

25.8.51 [P] Modify the `first.m` and `second.m` programs of §25.6.4 to approximate $f'(x)$ and $f''(x)$ for $f(x) = \sqrt{x}$. Do the curves of error versus Δ look similar to those for $f(x) = e^x$? Do they have minima near the recommended values of $\Delta^\star$?

25.8.52 [P] If we knew the exact value of $f'(x)$ at a given point $\bar{x}$, we could approximate $f'(\bar{x})$ by forward differencing using various trial values of Δ and thereby find $\Delta^\star$ to minimize the total error in the forward-difference approximation. The central-difference approximation of $f'(x)$ is much more accurate than the forward-difference approximation, so for the purposes of implementing this idea we can consider it exact. This scheme finds a $\Delta^\star$ that is appropriate to the shape of the function $f(x)$ at the point $\bar{x}$. We can then use that value of $\Delta^\star$ to approximate $f'(x)$ by forward differencing at other points sufficiently near $\bar{x}$. (a) Write a MATLAB routine `fdints.m` that uses this approach to find, for a given function $f(\mathbf{x})$ and point $\bar{\mathbf{x}}$, the optimal step $\Delta_j^\star$ to use in each direction j for making forward-difference approximations of $\nabla f(\mathbf{x})$ near $\bar{\mathbf{x}}$. Explain how you chose the interval to use in the central-difference approximation, and how you search for the optimal Δ_j. (b) Is the $\Delta^\star$ returned by your routine for $f(x) = e^x$ and $\bar{x} = \frac{1}{2}$ close to the value we found in §25.6.4? Find an $f(x)$ and an $\bar{x}$ for which the $\Delta^\star$ returned

by your routine differs significantly from the value recommended in §25.6.4. (c) What would be necessary to extend this idea to find optimal intervals for use in central differencing?

25.8.53 [P] In §25.6.5, I used `gradcd.m` and `hesscd.m` to solve the `egg` problem. (a) Explain how. (b) Use `sd.m` and `ntfs.m` to solve the problem as accurately as possible. What is $\mathbf{x}^\star$? (c) Write a MATLAB program to reproduce the convergence-trajectory graphs for `sd.m` and `ntfs.m` when they are used to solve the problem.

25.8.54 [P] Write MATLAB routines (a) `gradfd.m` and (b) `hessfd.m` to compute gradient and Hessian approximations by forward differencing. Test them by using them to solve the `egg` problem with `sd.m` and `ntfs.m`. Why might these routines sometimes be preferable to `gradcd.m` and `hesscd.m`?

25.8.55 [E] Suppose that you want to solve a nonlinear program by one of the algorithms discussed in this book, and that you write three routines to compute respectively the values of the objective and constraint functions, their gradients, and their Hessians. Is it possible, even after you have carefully hand-checked your formulas and MATLAB coding, that these routines are wrong? What more can you do to discover inconsistencies between them?

25.8.56 [E] The MATLAB routines `gradtest.m` and `hesstest.m` are described in §25.6.6. (a) Explain how they work. (b) What significance does a return value of `reldif=-1` have? (c) What values of `reldif` suggest that there is a mistake in coding a gradient routine? (d) What values of `reldif` suggest that there is a mistake in coding a Hessian routine? (e) How might a function, gradient, or Hessian routine be wrong even though `gradtest.m` and `hesstest.m` report that all gradients and Hessians tested are very close to their central-difference approximations?

25.8.57 [E] What is the basic idea of *automatic differentiation*? Does it produce formulas, or numbers? What is a *parse tree*? What is true of the operations that appear in a parse tree? What is an *intermediate variable* of a parse tree?

25.8.58 [H] Consider the function $f(\mathbf{x}) = x_2(1 + e^{2x_1})$. (a) Draw a parse tree for evaluating the function. (b) Write down expressions for the partial derivative of each intermediate variable with respect to the inputs of the operation that produced it. (c) Use the chain rule to combine those partial derivatives and find $\nabla f(\mathbf{x})$.

25.8.59 [E] When automatic differentiation is carried out symbolically, it is conceptually equivalent to using a computer algebra package such as Maple to find formulas for the partial derivatives. What additional capabilities do some programs for automatic differentiation have? Why are they desirable?

25.8.60 [P] Use `qpin.m` to solve the `big` problem with (a) `a=[2,3]`; (b) `a=[-3,3]`. (c) Find by experiment the largest value of n for which you can solve the problem by using `qpin.m`, generating coefficient vectors `a` and starting points `x` at random after the fashion of §25.7.2.

25.8.61 [E] If an algorithm is to be effective for solving large problems, how should its storage requirements and running time grow as functions of n? Which of the routines listed in the table of §25.1 satisfy that requirement?

25.8.62 [E] Name three machine learning applications that we have studied.

25.8.63 [E] Large optimization problems that are tractable typically have certain attributes. What are they? What is the technical term for a problem that has them?

25.8.64 [E] Explain the terms (a) *component-separable*; (b) *block-separable*; (c) *partially-separable.* Why are these important attributes for a large nonlinear program to have?

25.8.65 [E] What are the two steps involved in solving a large nonlinear program, according to the glib description of that art given in §25.7.1?

25.8.66 [E] Describe in words the cyclic coordinate descent algorithm. What are its advantages and drawbacks?

25.8.67 [H] In `big.m`, would it save time to use a convergence test in the line search? Explain.

25.8.68 [P] It was easy to use cyclic coordinate descent on `big.m` because the inequality constraints of that problem are simple bounds on the variables. (a) Describe how the method might be applied to an inequality-constrained nonlinear program whose constraints are *not* simple bounds. (b) Write a MATLAB function to implement your idea, and use it to solve the `ek1` problem.

25.8.69 [E] Explain what the MATLAB expression `min(1./a,a)` produces, when `a` is a vector.

25.8.70 [P] Modify `big.m` to use the random coordinate descent algorithm. How does this affect the speed of the program? Does it affect the storage required?

25.8.71 [H] In `big.m` we used a line search to find α. Modify the derivation in §10.5 to find a formula for the full coordinate descent step. Would it be a useful alternative to searching the line when n is large? Explain.

25.8.72 [E] Describe the characteristics that are typical of effective methods for attempting the solution of large nonlinear programs. What are some of the problem characteristics that these methods exploit? Why is an approximate solution to a large problem often good enough?

25.8.73 [E] What must be true if parallel computing is to be used in solving a large nonlinear program?

25.8.74 [E] Explain why `qpeq.m`, `rsdeq.m`, `rneq.m`, `penalty.m`, `auglag.m`, `grg.m`, `ntlg.m`, and `sqp.m` are not listed in the §25.7.3 table of methods suitable for large problems.

25.8.75 [H] Find out about limited-memory quasi-Newton methods and explain how they work.

25.8.76 [E] What is a *semi-analytic* result, and how does it differ from an analytic solution of a nonlinear program?

25.8.77 [E] If in the `big` problem we admit the case where some $a_j = 0$, how does this change the rule for constructing $\mathbf{x}^\star$?

25.8.78 [H] List the places in this text where we have encountered nonlinear programs having nondifferentiable functions, and describe the tricks we have used to solve them. Are there nonsmooth nonlinear programs for which these tricks do not work?

26

Algorithm Performance Evaluation

In §9.4, I charted the space of nonlinear optimization methods on orthogonal axes of robustness versus speed and described the history of the discipline as a search for some Northeast Passage leading to an algorithm that solves every problem quickly. Since then we have seen that there is no such method, and that two dimensions are not enough for a picture explaining why. Each algorithm has its own personality, with a spectrum of important attributes. What is its constraint affinity? How do its memory footprint and execution time scale with problem size? Can it be implemented in a way that permits the use of parallel processing? Does something limit the accuracy of the solutions it can find? How close to feasible are they? It is silly to ask for a rank ordering of methods that differ in so many ways.

Yet performance does matter. Nonconvex optimization is hard, in the technical sense of §7.9. In that context all of our methods are really just **heuristics**, reasonable strategies that might or might not work on any given problem, and some are observed to work better than others. Convex optimization is easy, because then the methods we have studied can be proved to converge, but in this context also different methods do not work equally well. Which algorithm will work best in practice for solving a particular class of problems? Which problems are most likely to be solved by a particular algorithm? These questions are not silly at all. Unfortunately, their answers are largely beyond the reach of theory.

In Chapters 10, 13, 14, and 17-24 we often dissected the progress of an algorithm in minute detail to study the workings of its logic and numerics as it solved one particular problem. Such an investigation can illustrate and explain how a method should ideally work on a problem that perfectly fits its design, but cannot predict what the algorithm will do with the more varied and realistic problems encountered in practice or how it will perform compared to some other method. A more general analysis might allow predictions like those to be made, but analyzing even a simple algorithm in general is usually mathematically intractable. In the rare instance when the mathematical analysis of an algorithm succeeds it often yields only asymptotic results [72, §4] or predicts worst-case performance, while it is average or typical performance that is of interest for the evaluation and comparison of nonlinear programming methods. In the analytic study of computational complexity, an algorithm is considered "good" if the time and space it uses grow no faster than polynomial functions of problem size [55], but this is not much help in distinguishing between heuristics when all of them (or none of them) fit that description. A useful algorithm must be numerically stable and yield accurate results, but only rarely (as in §25.6.4) is a floating-point calculation simple enough that a realistic analytic model can be found for roundoff error.

To answer important practical questions that do not yield to analysis, algorithm developers and users frequently resort to numerical experiments, with goals including these:

- to find the best existing method for solving a certain problem or class of problems;
- to reveal possible improvements that might be made to an algorithm, or to determine whether some change actually is an improvement;
- to discover what class of problems can be solved by a new algorithm;
- to demonstrate to others that a new algorithm actually works.

To study the performance of an algorithm experimentally we "just" need to try it on some problems and see how quickly it solves them. People have been doing this since the dawn of mathematical programming, so in addition to the many research papers that incidentally include experimental results there is an extensive literature about how to conduct experiments and report findings (e.g., [34] [42] [44] [48] [85] [139]).

Of course it is not the algorithm itself that we try in a computational experiment, but a computer program that implements the algorithm, so to learn about the algorithm we must make deductions from the behavior of the code. For example, if an evaluation that is based on speed is to be unbiased, it must somehow control for any factors affecting the running time of the program other than the algorithm itself, such as how the code is written and compiled and the environment in which it is run. The logical basis of **computational testing** is the assumption that there is some way to do that, or in other words that the following proposition is true.

A computer program can be used as a laboratory instrument for the experimental study of the algorithm it implements.

We can test using only a limited number of problems, so if our experiments are to accurately predict how the algorithm will perform on average the problems must be carefully chosen to represent the class of interest.

Some algorithms yield crude results very quickly while others produce more exact solutions but only if we are willing to wait. To interpret the results of our experiments it will be necessary to decide precisely what it means for an algorithm to have solved an optimization problem.

Thus, computational testing turns out to be fraught with thorny philosophical issues and subtle practical difficulties much like those that beset other experimental sciences. Just as it is possible to conduct meaningful experiments in physics and biology despite imperfections in apparatus, limitations of measuring equipment, and the foibles of human experimenters, it is also possible to avoid many of the pitfalls of computational testing. The goal of this Chapter is to address some of the issues that most commonly arise in the experimental study of optimization methods.

26.1 Algorithm vs Implementation

An algorithm (see §9.0) is an abstract recipe for performing a computation, so it can be stated using mathematical formulas or in pseudocode, or in a flowchart, or perhaps in other ways similarly unrelated to any actual implementation. An algorithm is thus a special sort of disembodied *idea.* In contrast, a **program** is a particular string of symbols in a particular source language, precisely specifying a particular sequence of arithmetic and logical operations to be performed by a real computer. Even after an algorithm is implemented in a program, so that the two are now typographically inseparable, we can retain a clear conceptual distinction between the idea and its realization. Properties that belong to the algorithm should remain **invariant** across all possible implementations, while properties belonging to the program might vary from one implementation to another. One ideal (though tedious) way of specifying an algorithm would be to provide a collection of *all* its possible implementations.

26.1.1 Specifying the Algorithm

Just where does the algorithm leave off and the program begin? That depends on the tradeoff we make between the generality and the strength of the conclusions we hope to draw about the algorithm from observations of the program. This is because the only observations of the program that are helpful in understanding the algorithm are those that would be true about *any* implementation of the algorithm *as it is specified.*

We might specify the algorithm in only a very general way, by describing the high-level processes to be used and the effects to be achieved, omitting most details. A sorting algorithm might be "exchange the elements of a list to put them in order." An algorithm for solving $\mathbf{Ax} = \mathbf{b}$ might read "perform elementary row operations on $\mathbf{A}$ so that the components of $\mathbf{x}$ can be found by successive divisions and back-substitutions." An algorithm for nonlinear optimization could require that we "generate a sequence of points in $\mathbb{R}^n$ such that the objective is lower at each point than at the preceding one." The vagueness of these algorithm specifications prohibits us from reporting minute details we might notice about the behavior of programs that implement them, because almost all such details are merely the result of arbitrary choices in the particular way each program was written. We could of course formulate general statements such as "sorting this way takes longer when the list gets bigger," or "solving $\mathbf{Ax} = \mathbf{b}$ like this doesn't work very well if $\mathbf{A}$ is large and sparse," or "this method of optimization sometimes gets stuck if the problem is nonconvex." These are true statements about the algorithms, but they are not very interesting; in fact, they are platitudes that we could state without performing any experiments at all. A vague algorithm specification leads to conclusions that have wide scope but are not very precise or specific.

At the opposite extreme we might take a particular computer program as the statement of the algorithm it implements, so that every tiny coding detail is included in the specification. The classic performance studies of Colville [28], Himmelblau [80] and Schittkowski [140],

among many others less famous, are fundamentally comparisons of computer programs rather than of algorithms. The object of study (the algorithm) is the same as the instrument of experimentation (the program) so the algorithm evaluation problem is reduced to describing what the program does. If the algorithm is the program, we are still talking about the algorithm if we report implementation-dependent specifics such as "the insertion sort ran in 0.1 seconds on my computer," or "Gauss elimination failed when it encountered a zero pivot," or "the steepest-descent program reported $\mathbf{x}^\star = [1.01, 0.99]^\top$ for Rosenbrock's problem." These statements are very definite and precise, but they are only very narrowly applicable. We can report many details about exactly how a program works, but they probably won't describe the behavior of other programs implementing the same algorithm.

In physics, the motions of particular objects are of less interest than the laws governing the motions of objects generally. In a similar way particular codes, being ephemeral things that seldom outlast even their authors, are of only limited or transient interest in mathematical programming. The central problem of computational testing is the design of experiments that reveal something about the intrinsic properties of algorithms rather than merely the idiosyncrasies of computer programs. For experimentation to yield conclusions that are both interesting and widely applicable, it is necessary to begin with an algorithm description that is neither so vague that nothing useful can be deduced from observations of any implementation, nor so precise that the conclusions we draw pertain to just one. The algorithm should be specified just precisely enough so that measurements will be able to reveal the intrinsic properties that are to be studied, and the experiments should ask only questions relating to properties of the algorithm as it is specified.

26.1.2 Designing Experiments

The behavior of algorithms, like other scientific questions, can be studied by formulating hypotheses that are testable by experiment. Once the algorithm has been specified in such a way that useful conclusions about it can in principle be deduced from measurements of a program, we need to design an experiment that permits such measurements to be made. For example, the running time or **efficiency** of a numerical method depends on both the algorithm and its implementation. A single absolute measurement of running time contains both algorithm and implementation effects, so it doesn't tell much about the intrinsic efficiency of the algorithm. But if we compare two *different* algorithms (perhaps choosing one of them as a standard) then implementation effects might be largely removed in the comparison, allowing us to conclude that one algorithm is inherently more efficient than the other. In order for the effects of coding details to cancel out, the programs must be written in the most naïve and straightforward way permitted by the algorithm specifications, so as to avoid inadvertently introducing refinements at the level of the coding. Any special data structures, memory reference patterns, or coding techniques should be explicit in the algorithm, not just hidden in the code. If several obvious implementations are possible they can all be tested to reveal the implementation effects; in this case it is the algorithm effects

that cancel out in the comparison. Programs being compared must be compiled in the same way, without allowing compiler optimization to rearrange the calculations.

Every program contains **convenience code** that has nothing to do with carrying out the steps of the algorithm it implements, but which must be present if we are to conduct experiments. Reading problem data, validating parameter values input by the experimenter, and writing out intermediate results so that we can watch the progress of the calculation are all things that we do *not* want to consider parts of the algorithm itself. In many testing environments the computational effort used by convenience code is *greater* than that used by **algorithm code**, so it cannot be neglected. It is essential to exclude from measurements of computational effort any that is expended in executing convenience code.

Different strategies are called for in the design of experiments for measuring other algorithm properties, such as accuracy, numerical stability, **reliability** (the proportion solved of problems within the theoretical limits of the algorithm), **robustness** (the proportion solved of problems *outside* the algorithm's theoretical limits), and sensitivity to imprecise function and derivative values [99]. Whatever is being measured, comparisons should be designed so that algorithm and implementation effects can be separated.

Many optimization codes have adjustable parameters that control their behavior (thus reducing the problem of solving a nonlinear program with n variables to the problem of tuning a program that has p adjustable parameters). Unless tuning these parameters is an explicit step in the algorithm specification, they should be fixed during the process of computational experimentation, and the same values should be used for all of the test problems.

26.2 Test Problems

In a comparison of several methods for nonlinear programming, any desired outcome can usually be achieved by judiciously selecting the test problems and their starting points. This can lead to the subconscious (or intentional) introduction of bias in an experimental study of algorithms, just as data censoring or lack of controls can bias experimental work in other fields. The same principles of laboratory discipline and professional ethics that prevail elsewhere in science must therefore be followed in computational testing. The most fundamental of these principles is that others should be able to repeat the work and confirm or deny the findings. This demands that the test problems you used and the programs you tested be easily available to others. If you have inadvertently cooked the books maybe someone will discover it by trying a *different* set of problems.

At least some of the test problems used in a computational study should be chosen from standard collections (e.g., [28] [31] [80, §a] [81]; also see the references listed in §8.4) rather than manufactured by the experimenter. If an algorithm has some particular special property, at least some test problems should be chosen or constructed to reveal that property.

All of the algorithms in a computational comparison should be given the *same* information about each test problem, unless the object of the experiment is simply to show the

effect of the difference in information. For example, if an algorithm requiring only function values is compared to one that also uses gradient information, the second algorithm ought to approximate its gradients from function values rather than calculating them from formulas. To see why this precaution is necessary, consider this algorithm: Get $\mathbf{x}^\star$ from the problem definition and print it out. It would not make sense to "provide each algorithm with the information it needs" in comparing this method to one that finds $\mathbf{x}^\star$ by actually solving the nonlinear program. A similar objection could be raised to providing bounds on the variables to an algorithm that can make use of them in a comparison to some method that cannot, though in that case it is less obvious how the bias might be eliminated.

The starting point for a problem should be determined by the problem definition, so that it isn't subject to manipulation by the experimenter. If several different starting points are of interest, they should be the fixed starting points of several different (though otherwise identical) problems.

The literature on computational testing (e.g., [34] [42] [85]) discusses other more technical considerations that can enter into the selection and description of test problems.

26.2.1 Defining the Problems

In §8.3.1 we used the file `garden.mod` to define the `garden` problem for submission to a NEOS solver via AMPL. That file included $\mathbf{x}^0$ and formulas for the objective and constraint functions. Elsewhere we have used MATLAB routines in the standard way that I first described in §15.5. For a problem named `prob` they are as follows.

`f=prob(x,i)`	returning the value `f` of function `i` at the point `x`
`g=probg(x,i)`	returning the gradient `g` of function `i` at the point `x`
`H=probh(x,i)`	returning the Hessian `H` of function `i` at the point `x`

We have used the convention that `i=0` designates the objective, `i=1...mi` the inequality constraints (if any), and `i=mi+1...mi+me` the equality constraints (if any).

In a typical testing environment (see §26.4) the algorithms of interest are implemented in a compiled language, and then the function, gradient, and Hessian subprograms defining each problem are coded that way too. If a large number of test problems are used it is helpful for the files defining them to be named in a standard way and managed systematically, to ensure that each experiment uses the intended function and derivative routines.

To facilitate the automation of a computational testing plan it is also helpful to **catalog**, in some machine-readable way, complete information to identify and characterize each test problem, including the items listed at the top of the next page.

For a problem to be useful in testing, its solution $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ must be precisely known. Some algorithms return $\boldsymbol{\lambda}^\star$ as well as $\mathbf{x}^\star$ but others do not. When $\mathbf{x}^\star$ is known it is often possible to determine $\boldsymbol{\lambda}^\star$ from the KKT conditions, either analytically or by using the `mults.m` program described in §16.10. The starting point is the midpoint of the bounds, $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}})$, so it need not be separately cataloged.

`prob`	prefix in the names of files defining functions and derivatives
n	number of variables
m_i	number of inequality constraints
m_e	number of equality constraints
$\mathbf{x}^{\mathrm{L}}$	lower bounds on the variables
$\mathbf{x}^{\mathrm{H}}$	upper bounds on the variables
$\mathbf{x}^\star$	exact optimal point
$\boldsymbol{\lambda}^\star$	exact KKT multipliers at the optimal point
provenance	where the problem came from (e.g., literature citations)
aliases	other names by which the problem is known

It is not uncommon for a published problem, whether it appears in a research article or in a curated collection, to be **defective**. Some problems are infeasible, unbounded, or ill-posed (see §16.8.3). Many problem statements contain typographical errors, ambiguities, imprecise data, or wrong answers [33, §1.1.3]; many do not include KKT multipliers or variable bounds. A handful of problems have been used repeatedly by the mathematical programming research community over many years and appear in several collections with different names or **aliases**. Occasionally a problem appearing in one collection is alleged to be the same as a problem appearing in another while they are actually different because of a transcription error or misidentification. Citations to original sources are also frequently garbled by misspellings, incorrect page numbers, and other mistakes. Because of these potential pitfalls it is necessary to validate each test problem you contemplate using. Whenever you publish a test problem you should, as a courtesy to other experimenters, diligently ensure that it is correct.

26.2.2 Constructing Bounds

If bounds on the variables will be used by an algorithm for any of the purposes mentioned in §9.5 they can be chosen in a way that biases the results of computational experiments. The most obvious influence of the bounds is through the starting point, but many algorithms are also affected by changing the width of the bounds even if their midpoint remains the same. The **catalog bounds** for each test problem should therefore be determined in some consistent mechanical way that gives them the properties listed below while preserving as much of the original problem statement as possible. To have these desirable properties the bounds we catalog might need to be wider than the limits on the variables that we obtain from the problem statement.

- The catalog bounds `[xl,xh]` should contain as tightly as possible any bounds $[\mathbf{x}^{\mathrm{L}}, \mathbf{x}^{\mathrm{H}}]$ that are specified in the problem statement or implied by the constraints.
- The midpoint of the catalog bounds will be the starting point; this should be the given starting point $\mathbf{x}^0$ if a starting point is given.

- The catalog bounds should contain the optimal point;
- The midpoint of the catalog bounds should differ from the optimal point in all of its components, unless the problem statement requires otherwise.
- The width of the catalog bounds `xh(j)-xl(j)` in any direction j should not be too small compared to $x_j^\star$.

How these complicated and interdependent requirements are met for a given problem will depend on the information provided in its original statement. We must assume that $\mathbf{x}^\star$ is known. For each $j \in \{1 \ldots n\}$ the problem statement might or might not specify x_j^0, x_j^L, or x_j^H, but for those quantities that are given we will insist that $x_j^0 \neq x_j^\star \neq x_j^L \neq x_j^H$, and that $x_j^L < x_j^H$. If any of these inequalities are violated the problem is either defective or cannot be used in testing unless the results are interpreted in a way that is unique to the problem.

The original problem statement might include a functional constraint that is a variable bound; in the problem below $x_1 \geq 3$ so $x_1^L = 3$. In solving this problem some algorithms might be able to make use of the lower bound on x_1, but all must enforce the explicit constraint.

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1^2 + x_2^2 & \text{from } \mathbf{x}^0 = [5,5]^\top \\ \text{subject to} & -x_1 + 3 \;\leq\; 0 & \end{array}$$

The original problem statement might include a bound that is not a functional constraint; in the problem below we are meant to avoid evaluating the square root where it is not defined, so $x^L = 0$ but there is no explicit nonnegativity constraint.

$$\underset{x}{\text{minimize}} \quad \cos(\sqrt{x})$$

Often it is possible to deduce bounds on the variables from constraints that are more complicated than simple variable bounds.

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -(x_1-1)^2 - (x_2+1)^2 \\ \text{subject to} & x_1^2 + x_2^2 \leq 4 \\ & x_2 \geq 0 \end{array}$$

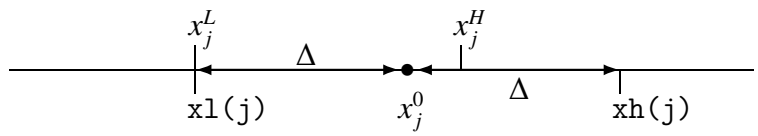
Here the first constraint limits the extreme values that each variable can take on. Notice that $x_1^2 + x_2^2 \leq 4 \Rightarrow x_1^2 \leq 4$, so $x_1 \in [-2, 2]$. Also, $x_1^2 + x_2^2 \leq 4 \Rightarrow x_2^2 \leq 4$, so $|x_2| \leq 2$, but the second constraint rules out negative values so $x_2 \in [0, 2]$. Together these constraints imply the variable limits $x_1^L = -2$, $x_1^H = 2$, $x_2^L = 0$, and $x_2^H = 2$.

The catalog bounds `xl` and `xh` that we adopt for each of these examples (see Exercise 26.6.17) must contain the variable limits x_j^L and x_j^H that we have deduced from the problem statements, but to ensure that they also have the other properties listed above we must pay attention to the optimal point for each problem and to the starting point when one is specified. The formulas given on the next page show one way in which that can be done.

case	x_j^L	x_j^H	x_j^0	bounds calculation
0	□	□	□	$\lvert x_j^\star\rvert \geq 10^{-6}\lVert\mathbf{x}^\star\rVert \begin{cases} \texttt{xl(j)} = \min\left(0.1x_j^\star,\ 10x_j^\star\right) \\ \texttt{xh(j)} = \max\left(10x_j^\star,\ 0.1x_j^\star\right) \end{cases}$ else $\begin{cases} \texttt{xl(j)} = -0.1 \\ \texttt{xh(j)} = 10 \end{cases}$
1	□	□	■	$\Delta = \lvert x_j^0 - x_j^\star\rvert$ $\texttt{xl(j)} = x_j^0 - 10\Delta$ $\texttt{xh(j)} = x_j^0 + 10\Delta$
2	□	■	□	$\Delta = \max\left([x_j^H - x_j^\star],\ 0.01\times\frac{1}{2}[x_j^H + x_j^\star]\right)$ $\texttt{xl(j)} = x_j^\star - 0.1\Delta$ $\texttt{xh(j)} = x_j^H$
3	□	■	■	$\Delta = x_j^H - x_j^0$ $\texttt{xl(j)} = x_j^0 - \Delta$ $\texttt{xh(j)} = x_j^H$
4	■	□	□	$\Delta = \max\left([x_j^\star - x_j^L],\ 0.01\times\frac{1}{2}[x_j^\star + x_j^L]\right)$ $\texttt{xl(j)} = x_j^L$ $\texttt{xh(j)} = x_j^\star + 10\Delta$
5	■	□	■	$\Delta = x_j^0 - x_j^L$ $\texttt{xl(j)} = x_j^L$ $\texttt{xh(j)} = x_j^0 + \Delta$
6	■	■	□	$\Delta = \frac{1}{2}\left(x_j^L + x_j^H\right)$ $\texttt{xl(j)} = x_j^L$ $\texttt{xh(j)} = x_j^H$
7	■	■	■	$\Delta = \max\left(x_j^0 - x_j^L, x_j^H - x_j^0\right)$ $\texttt{xl(j)} = x_j^0 - \Delta$ $\texttt{xh(j)} = x_j^0 + \Delta$

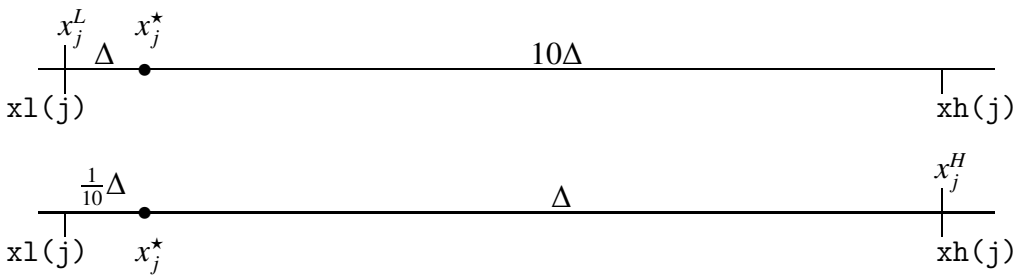
These rules are regrettably arcane, but they do have the virtue of having been used in successful computational studies [33, Appendix 2] [88, Appendix A]. They are of course essentially arbitrary (that is the whole point) and different ones might make more sense to you, but *some* rules must be used if the constructed bounds are to be unbiased.

In case 7 a starting point and both limits are determined by the original problem statement, so the catalog bounds are constructed as shown below; `xh(j)` $> x_j^H$ to make the given $\mathbf{x}^0$ the midpoint of the catalog bounds. The rationale for the formulas in cases 5 and 3 is similar to that used here.



In case 6 no starting point is specified, so the catalog bounds are the given limits and $\mathbf{x}^0$ is their midpoint.

In cases 4 and 2 only one limit is determined by the problem statement, so the catalog bounds are based on its distance from the optimal point. However, if the distance between the given limit and the optimal point is less than 1% of the average of their coordinate values, Δ is taken to be that average instead.



In case 1 only a starting point is given, so its distance to the optimal point is used to construct catalog bounds symmetric about the starting point.

In case 0 only the optimal point is known. If its jth coordinate is different enough from zero, it is used to construct bounds asymmetric about $x_j^\star$; if the solution coordinate is too small to use in that way, the bounds are set to $[-0.1, 10]$.

It is possible for the bounds produced by some of these rules to exclude the optimal point; in each case they should be widened if that happens by repeatedly decreasing `xl` and increasing `xh` by the Δ for that case until $\mathbf{x}^\star \in [\texttt{xl}, \texttt{xh}]$ (this is the only reason Δ is computed in case 6).

26.3 Error vs Effort

The algorithm implementations discussed in earlier Chapters typically test for convergence by comparing a tolerance `epz` to some quantity that should approach zero as $k \to \infty$. For example, in unconstrained minimization the objective gradient `g` approaches zero so the test usually looks like this.

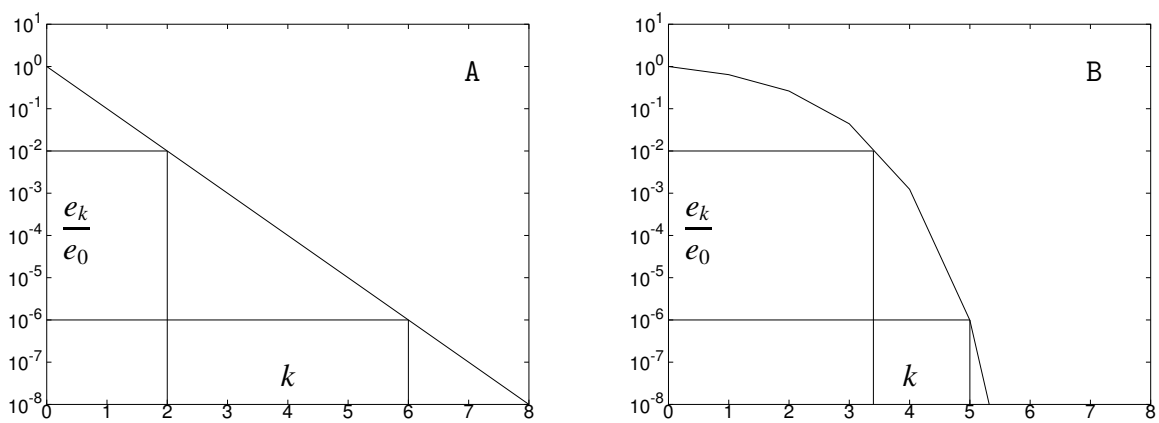
```
if(norm(g) <= epz) break; end
```


Suppose that programs implementing algorithms A and B are used to solve the same problem, and that each passes this test of **gradient norm error** upon completing the number of iterations k shown in the table below. Which algorithm is the faster of the two?

epz	A	B
10^{-2}	4	8
10^{-6}	7	6

Method A satisfies the criterion $\|\nabla f_0(\mathbf{x}^k)\| \leq \epsilon$ in fewer iterations than B when $\epsilon = 10^{-2}$ but needs more when $\epsilon = 10^{-6}$, so the answer depends on how close to stationary our approximation of $\mathbf{x}^\star$ must be in order for the problem to be considered "solved."

To gain a more complete understanding of how these algorithms behave we might replace the table by the following error curves (see §9.1), which show how each method decreases the **relative distance error** $e_k/e_0 = \|\mathbf{x}^k - \mathbf{x}^\star\|/\|\mathbf{x}^0 - \mathbf{x}^\star\|$ as k increases.



By the criterion $e_k/e_0 \leq \epsilon$, method A again converges in fewer iterations than B when $\epsilon = 10^{-2}$ and needs more when $\epsilon = 10^{-6}$. Now, however, we can see the relative error e_k/e_0 for *every* value of k, and this lets us recognize algorithm A's convergence as linear and algorithm B's as quadratic.

Unfortunately, graphs of solution error versus iteration count are not very useful for comparing algorithms unless the *only* thing we care about is their order of convergence. The amount of computation required to perform an iteration of B probably differs from the amount needed for an iteration of A, and in either algorithm the work done in one iteration might differ from the work done in another. It would therefore be misleading to plot the curves above on the same set of axes, and they do *not* permit us to say which algorithm takes less work to reach some level of error. To do that we must use a more meaningful measure of computational effort; we will consider some possibilities below.

Using e_k/e_0 to measure solution error can also be misleading when comparing algorithms. The distance in $\mathbb{R}^n$ between an iterate and an optimal point tells us nothing directly about the objective value or feasibility of the iterate, and if there are multiple optima we need a

rule for deciding which one to use in computing $e_k = \|\mathbf{x}^k - \mathbf{x}^\star\|$. In a constrained problem a strictly feasible point $\hat{\mathbf{x}}$ and a grossly infeasible point $\bar{\mathbf{x}}$ can be the *same* distance from $\mathbf{x}^\star$, but they are *not* equally suitable as a solution to the problem! Even if none of these difficulties arise in a particular algorithm comparison it does not make sense to ignore the value of the objective, whose minimization is after all the immediate goal of the optimization. Thus we would also prefer a more meaningful measure of solution error.

26.3.1 Measuring Solution Error

As we solve a problem the objective approaches its optimal value, so the **function error** $f_0(\mathbf{x}^k) - f_0(\mathbf{x}^\star)$ is as natural a measure of solution quality as the distance error e_k. At infeasible points the function error might be negative, so we had better use its absolute value. Now the violation of a constraint can also contribute to the error of an iterate, if before combining it with the function error we scale it to reflect its effect on the objective value. Recall from §15.3 that perturbing a constraint that is tight at $\mathbf{x}^\star$ changes the optimal objective value by the shadow price

$$\frac{\partial f_0}{\partial f_i} = -\lambda_i^\star,$$

where $\lambda_i^\star$ is the constraint's KKT multiplier at $\mathbf{x}^\star$. Using this scale factor leads to the **combined solution error**

$$\varepsilon_k = \left|f_0(\mathbf{x}^k) - f_0(\mathbf{x}^\star)\right| + \sum_{i=1}^{m_i+m_e} \left|\lambda_i^\star f_i(\mathbf{x}^k)\right|.$$

A problem with equality constraints can have KKT multipliers of either sign and $f_i(\mathbf{x}^k)$ that are nonzero for $\mathbf{x}^k \neq \mathbf{x}^\star$ even if $\lambda_i^\star \neq 0$, so it is necessary to take the absolute value of each constraint-violation term. This measure has the highly desirable properties that

$$\begin{array}{lcl}
\varepsilon_k & = & 0 \quad \text{if} \quad \mathbf{x}^k = \mathbf{x}^\star \\
\varepsilon_k & > & 0 \quad \text{if} \quad \mathbf{x}^k \neq \mathbf{x}^\star \\
\varepsilon_k & & \text{increases with objective error} \\
\varepsilon_k & & \text{increases with violations of constraints that are active at } \mathbf{x}^\star \\
\varepsilon_k & = & \text{objective error if there are no constraints}
\end{array}$$

Notice that it ignores violations of inequalities that are slack at optimality (for which $\lambda_i^\star = 0$). The MATLAB routine `cse.m` listed below returns ε_k at a given point $\mathbf{x}^k$.

```
function ek=cse(xk,fstar,lambda,fcn)
  ek=abs(fcn(xk,0)-fstar);
  m=size(lambda,1);
  for i=1:m
      ek=ek+abs(lambda(i)*fcn(xk,i));
  end
end
```


If our algorithm evaluations based on one test problem are to be comparable to those based on another, we must use an error measure that is insensitive to their starting points. Therefore, as we did for e_k in §9.1, we will normalize ε_k by its value at $\mathbf{x}^0$ and describe the performance of an algorithm by plotting the **log relative combined solution error**

$$\mathcal{E}_k = \log_{10}\left(\frac{\varepsilon_k}{\varepsilon_0}\right)$$

of its iterates, or `LRCSE`, as a function of computational effort. Each such curve begins at $\mathcal{E}_0 = \log_{10}(1) = 0$. Because `LRCSE` uses $\boldsymbol{\lambda}^\star$ it can't be used in studying a problem that lacks a constraint qualification.

26.3.2 Counting Function Evaluations

Above I argued that k is a bad measure of computational effort because an iteration of one algorithm might take much more work than an iteration of another. For example, an iteration of the ellipsoid algorithm requires on average $\frac{1}{2}m$ function evaluations and a single gradient calculation, while each iteration of the primal-dual interior point algorithm requires $m+1$ Hessians, $m+1$ gradients, and m function values. An accurate comparison of the effort used by these algorithms should somehow take into account this difference between them.

If a nonlinear program is big and complicated, most of the work required to solve it might be in the `NFE` function evaluations, `NGE` gradient evaluations, and `NHE` Hessian evaluations that are used by an algorithm. If finding each element of a gradient vector or symmetric Hessian matrix takes about as much work as finding a single function value, then it seems reasonable to use the **equivalent function evaluations**

$$\texttt{EFE} = \texttt{NFE} + n \times \texttt{NGE} + \tfrac{1}{2}n(n+1) \times \texttt{NHE}$$

performed by an algorithm as a measure of the computational effort it expends.

The program listed on the next page uses the `ea.m` routine of §24.4 to solve the `ek1` problem and plots, in the pictures below the listing, the `LRCSE` of each iterate versus the `EFE`s consumed. The `ek1efe.m` and `ek1gefe.m` routines shown to the right of the program are **stub routines** whose only purpose is to count a function or gradient evaluation [2-3] before invoking `ek1.m` or `ek1g.m` to perform it [4].

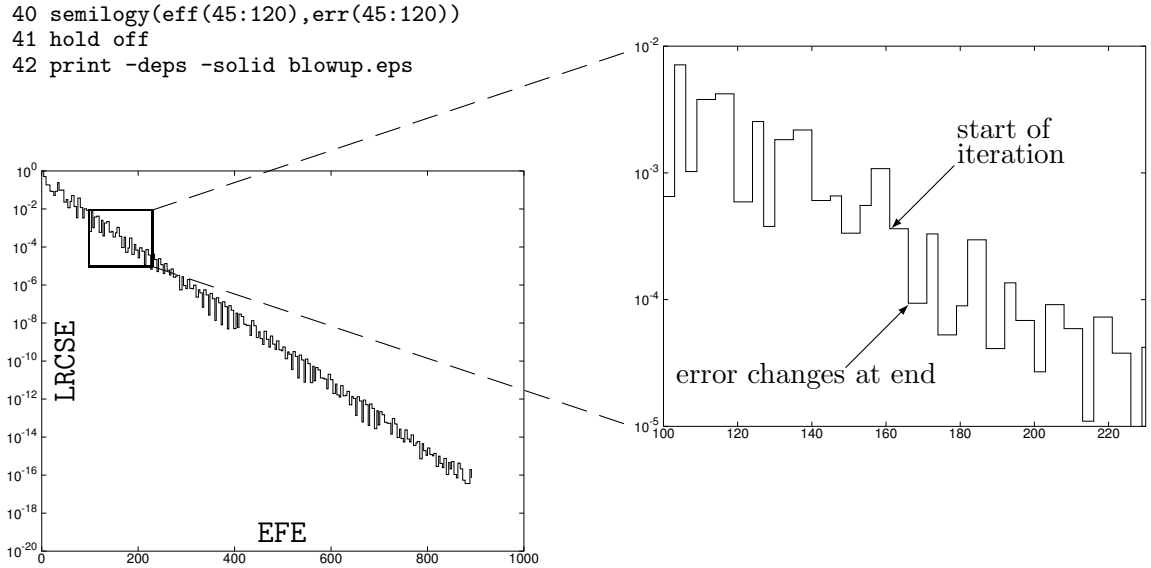
The program begins [3] by initializing the global variables `NFE` and `NGE` to zero. Then it sets [5] $\mathbf{x}^0$, [6] $\mathbf{Q}_0$, [7] n, and [8] m for the `ek1` problem. Next [10-12] it finds the combined solution error `erz` $= \varepsilon_k$ when $k = 0$, [14] sets the starting relative error `err(1)` $= \varepsilon_k/\varepsilon_0 = 1$, and [15] sets the starting effort `eff(1)` $= 0$ `EFE`s.

The loop over `k` [17-28] invokes `ea.m` repeatedly [18] to solve the problem one iteration at a time with a zero convergence tolerance. In each invocation the input value of `xk` is the starting point $\mathbf{x}^{k-1}$ for iteration k and the output value of `xk` is the iterate $\mathbf{x}^k$ generated by the iteration; `Qk` is similarly updated. After each iteration the return code from `ea.m` is tested [19] and the loop is exited prematurely if `ea.m` cannot continue.


```
% eaefe.m: plot LRCSE versus EFE for the ellipsoid algorithm when it is used to solve ek1
clear; clf
global NFE=0 NGE=0

xk=[18;21];
Qk=[80,0;0,169];
n=2;
m=3;

fstar=614.21209720340380;
lambda=[250.99653438461144;0;0];
erz=cse(xk,fstar,lambda,@ek1);
ke=1;
err(ke)=1;
eff(ke)=0;

for k=1:300
    [xk,rc,kused,Qk]=ea(xk,Qk,m,1,0,@ek1efe,@ek1gefe);
    if(rc > 1) break; end

    EFE=NFE+n*NGE;
    ke=ke+1;
    eff(ke)=EFE;
    err(ke)=err(ke-1);
    ke=ke+1;
    eff(ke)=EFE;
    err(ke)=cse(xk,fstar,lambda,@ek1)/erz;
end
rc
k

figure(1)
set(gca,'FontSize',25)
semilogy(eff,err)
print -deps -solid eaefe.eps
figure(2)
hold on
set(gca,'FontSize',25)
axis([100,230,1e-5,1e-2])
semilogy(eff(45:120),err(45:120))
hold off
print -deps -solid blowup.eps
```

```
function f=ek1efe(x,i)
  global NFE
  NFE=NFE+1;
  f=ek1(x,i);
end
```

```
function g=ek1gefe(x,i)
  global NGE
  NGE=NGE+1;
  g=ek1g(x,i);
end
```



When the program is run it outputs 29-30 `rc=2` indicating that `Qk` became computationally non-positive-definite at `k=209`. In finding each new iterate, `ea.m` invokes `ek1efe.m` repeatedly and `ek1gefe.m` once, and they increment `NFE` and `NGE`. At the end of each iteration the program 21 updates `EFE` by using those numbers in the formula we derived above, and remembers that measure of effort 26 for plotting along with 27 the relative combined solution error of the current point. Statements 22-24 generate the square wave curve discussed next. Finally 32-42 the program plots the graphs.

The **error-vs-effort curve** [48] [139] that results is a square wave, because each $\mathbf{x}^k$ is produced only at the *end* of iteration k; while the calculations for that iteration are being performed ε remains what it was at the beginning of the iteration. The amount of work required to perform iteration k is thus the width of the horizontal segment at error level ε_{k-1}. Usually, as in this example, the iterations do not all take the same amount of work. The linear order of the ellipsoid algorithm's convergence is still evident in the left picture, despite the fact that its horizontal axis is now `EFE` rather than k, but its convergence constant can no longer be deduced from the slope. On these axes, however, we could plot `LRCSE` vs `EFE` for another algorithm and make a valid comparison of the two (see Exercise 26.6.31).

In justifying the use of equivalent function evaluations to measure computational effort, I argued that they account for most of the work required to solve a big and complicated nonlinear program. If the functions and derivatives are very expensive to compute, as they are in many type-2 problems, it is often true that those calculations dwarf the updates that constitute the algorithm itself. But solving a hard problem takes a long time, so most of the computational testing that is done to guide algorithm development (and choice) uses problems that are more like `ek1` and the other little examples we have considered in this book. In solving them even a simple algorithm might do more work in the updates than it does in evaluating functions, gradients, and Hessians. Often this other work is proportional to `EFE` and then using `EFE` as an error measure might be reasonable anyway [98, p280-284]. But that will not be true if the ratio of update work to `EFE` differs from one algorithm to another or if any of the algorithms involve a significant amount of fixed overhead [112, p337-359]. There are also situations in which it is not obvious what should count as a function evaluation; in measuring the effort used by a sequential quadratic programming algorithm, for example, how should we attribute the work that is done in solving the QP subproblems? Thus, although `EFE` is widely used (e.g., [137]) and often useful, it is far from the ideal measure of effort.

26.3.3 Measuring Processor Time

The work that an algorithm does in iteration k includes not only evaluating the functions, gradients, and Hessians that it needs but also performing arithmetic and logical operations on those quantities to find $\mathbf{x}^{k+1}$. For example, the `ea.m` routine, which we invoked in the `eaefe.m` program of §26.3.2, normalizes the gradient that it will use to make the cut, finds the direction in which to move the ellipsoid center, computes $\mathbf{x}^{k+1}$, and updates the ellipsoid

matrix. The simplest way to include these operations in our accounting of computational effort is to measure CPU time instead of counting only EFEs.

The MATLAB `tic` and `toc` commands, which we used in §24.6 and §25.7.2, provide low-resolution measurements of wallclock time. That includes keyboard interactions, system background activities such as periodically checking for email, and time spent by other foreground tasks that are sharing the processor and sometimes get their turn to run. An estimate of the CPU time used by one program based on `tic` and `toc` is therefore not accurate enough to be useful in most performance studies. Instead we will use the MATLAB function

```
[total,user,system]=cputime()
```

which returns only the processor time that has been consumed by the MATLAB session in which it is invoked. The return value `total` is the sum of `user` and `system`, where `system` tells the CPU seconds spent doing things like displaying the MATLAB command window. It is `user` we want, because that tells the CPU seconds spent executing our commands.

Using `EFE` to measure computational effort ignores the work of an algorithm's updates and thereby underestimates the effort expended, but using all of the CPU time consumed by the program produces a gross *over*estimate. The effort we want to measure is only that which is used in performing the steps of the algorithm under test. As I mentioned in §26.1.2, a test program that carries out our experiments always includes convenience code that is not part of the algorithm and should therefore not be timed. To avoid timing convenience code it is necessary to **instrument** the program by inserting statements to measure the time spent performing different segments of the code. I instrumented the program `eacpu.m`, listed on the next page, to segregate the time `talg` that it spends performing the steps of the algorithm (boxed) from the time that it spends executing convenience code.

Most of the program has nothing to do with the algorithm. The second stanza 6–9 consists of necessary initializations, so it is bracketed by invocations of `cputime()`. The first invocation 5 gets the user time `u1` before the initializations are performed and the second 10 gets the user time `u2` after; then `talg` can be incremented, from its initial value of zero 3, by the difference `u2-u1`. The invocation of `ea.m` within the loop 22 is also necessary for performing the algorithm, so it too is bracketed by `cputime()` invocations. The first 21 gets `u1` before `ea` is entered, and the second 23 gets `u2` after `ea` returns, so that 25 `talg` can be incremented by their difference (including the time `ea.m` spent in `ek1.m` and `ek1g.m`). The rest of the program resembles `eaefe.m` except that the stub routines are no longer needed and I have (for reasons that will be clear) simplified the plotting of error versus effort 36.

I also 26 printed the value of `talg` after each iteration of the algorithm, as shown to the right of the listing, and from this output it is obvious that this program is *unsuccessful* in timing this algorithm. Often consecutive values of `talg` were identical, so in the output I replaced them by a single vertical ellipsis. When `talg` did not change it was because the `cputime()` invocations bracketing a code segment returned `u1` and `u2` values that were the same. When `talg` did change it always increased by exactly one step of `0.004` seconds, and


```
 1 % eacpu.m: plot LRCSE versus CPU for the ellipsoid algorithm when it is used to solve ek1
 2 clear;
 3 talg=0;
 4
 5 [t1,u1,s1]=cputime();
 6 xk=[18;21];
 7 Qk=[80,0;0,169];
 8 n=2;
 9 m=3;
10 [t2,u2,s2]=cputime();
11
12 talg=talg+(u2-u1);
13 fstar=614.21209720340380;
14 lambda=[250.99653438461144;0;0];
15 erz=cse(xk,fstar,lambda,@ek1);
16 ke=1;
17 err(ke)=1;
18 eff(ke)=talg;
19
20 for k=1:300
21     [t1,u1,s1]=cputime();
22     [xk,rc,kused,Qk]=ea(xk,Qk,m,1,0,@ek1,@ek1g);
23     [t2,u2,s2]=cputime();
24
25     talg=talg+(u2-u1);
26     printf('%3i %f\n',k,talg)
27     if(rc > 1) break; end
28     ke=ke+1;
29     eff(ke)=talg;
30     err(ke)=err(ke-1);
31     ke=ke+1;
32     eff(ke)=talg;
33     err(ke)=cse(xk,fstar,lambda,@ek1)/erz;
34 end
35
36 semilogy(eff,err)
```

```
octave:1> eacpu
  1 0.000000
  2 0.000000
  3 0.004001
  :
 11 0.004001
 12 0.008001
  :
 16 0.008001
 17 0.012001
  :
 30 0.012001
 31 0.016001
  :
 44 0.016001
 45 0.020001
  :
 49 0.020001
 50 0.024001
  :
 58 0.024001
 59 0.028002
  :
 63 0.028002
 64 0.032002
  :
 72 0.032002
 73 0.036002
  :
 77 0.036002
 78 0.040003
  :
 86 0.040003
 87 0.044003
  :
 91 0.044003
 92 0.048003
  :
105 0.048003
106 0.052003
  :
119 0.052003
120 0.056003
  :
133 0.056003
134 0.060004
  :
138 0.060004
139 0.064004
  :
147 0.064004
148 0.068004
  :
152 0.068004
153 0.072005
  :
161 0.072005
162 0.076005
  :
164 0.076005
167 0.080005
  :
180 0.080005
181 0.084005
  :
194 0.084005
195 0.088005
  :
197 0.088005
  :
199 0.088005
200 0.092005
  :
209 0.092005
octave:5> quit
```

running the program several times produced entirely different patterns of repeated `talg` values, so they are all just useless instrumental noise.

On my computer the `cputime()` function has the standard Unix CPU timing resolution of 0.01 seconds, which is longer than the time it takes to execute either the initialization stanza 6-9 or a single one-iteration invocation of `ea.m` 22 in solving `ek1`. Only much longer (or slower) code segments can be accurately timed by using `cputime()` in MATLAB.

To use processor time as a measure of effort it is essential to exclude convenience code; that often requires the timing of short code segments, which is difficult to do accurately. By dint of certain low cunning it is possible in Unix [100, §18.5.1] [88, §2.2.3.1] to indirectly make CPU time measurements with a precision of 1 μs, and in the next Section we shall see how to measure wallclock time with a precision even finer than that, but these techniques can be used only if the algorithm under test is implemented in a compiled language such as FORTRAN (see §26.4).

CPU time measurements are intuitively appealing and often reported, but different processors run at different speeds so times measured on one machine are (unlike EFEs) hard to compare with times measured on another. Thus, even when they are accurate, CPU time measurements are not always ideal for describing the results of computational experiments.

26.3.4 Counting Processor Cycles

Some processors admirably permit their cycle clock to be inspected by a running program, and this information can be used to count the cycles that were used in carrying out a given sequence of source code statements. To obtain the current cycle count it is necessary to execute a machine-language instruction that reads the processor clock, and this is practical only from a compiled programming language. To show how experiments can be conducted using programs in a compiled language I will pick the simplest one, classical FORTRAN [100]. Even if you have never seen this language before you will probably be able to understand the code discussed below. Classical FORTRAN does only scalar arithmetic and it requires arrays and some scalar variables to be explicitly dimensioned and typed, but otherwise it is quite similar to MATLAB. The suffix `D0` (that's a zero) indicates that a constant is `REAL*8`.

The program `eacyc.f` listed on the next page uses the ellipsoid algorithm to solve a nonlinear program one iteration at a time, so in its broad outline it resembles the MATLAB program `eacpu.m` of §26.3.3. It begins `3-5` by using `COMMON` (similar to the `global` statement in MATLAB) to find out about the problem that is being solved. When this program is compiled it will be linked with the function and gradient routines, always named `FCN` and `GRD`, that define the problem, and the descriptors in `COMMON` will be given values there. The second `7-9` and third `11-13` stanzas type and dimension variables that are used later.

The first stanza of executable code `15-22` uses the formulas in §24.3.1 to compute $\mathbf{x}^0$ and $\mathbf{Q}_0$ from the bounds $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$. The next stanza `24-30` initializes the performance measurement process, so it is part of the code's instrumentation. The combined solution error depends on $f_0(\mathbf{x}^\star)$ so `25` `FCN` is invoked to find `FSTAR` at the optimal point `XSTAR`. Then `CSE`, a FORTRAN equivalent of the MATLAB routine `cse.m`, is invoked `26` to find the combined solution error ε_0 = `ERZ` at the starting point. The `LRCSE` at that point is $\mathcal{E}_0 = \log_{10}(\varepsilon_0/\varepsilon_0) = 0$ so `28` `ERR(1)` is set to 0. The starting effort `CYALG` is zero cycles `29` (an integer) so `EFF(1)` is also set `30` to zero (the corresponding real number).

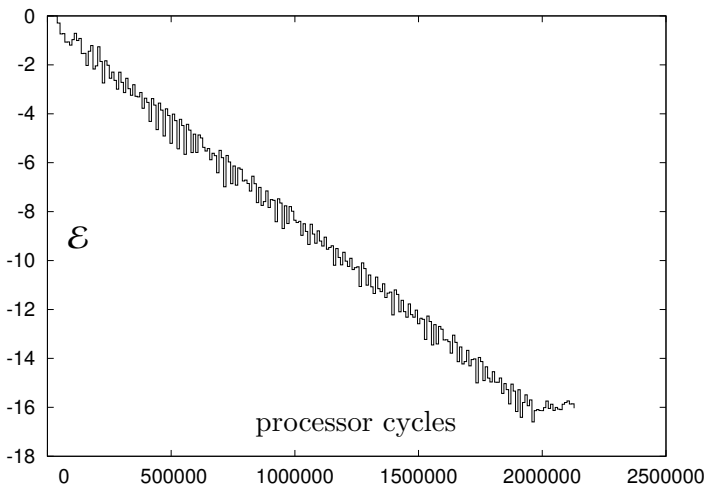
Then `32-55` comes a loop of iterations over `K`. Each begins `34` by invoking the `GETCYC` subroutine of [100, §18.5.3] to read the cycle clock, saving its value in `CY1`. Then subroutine `EA` is invoked `35` to perform one iteration of the ellipsoid algorithm. The next stanza sets `39` $\mathbf{x}^k = \mathbf{x}^{k+1}$ and `41` $\mathbf{Q}_k = \mathbf{Q}_{k+1}$. Then `46` the cycle clock is read again and its value saved in `CY2`. The `EA` routine sets the same return code values as `ea.m`, so if `RC=1` more iterations are possible. The cumulative cycles used by the algorithm, `CYALG`, is incremented `48` by the difference `(CY2-CY1)` between the count after performing the iteration and the count before. This effort value is `50,53` remembered along with `51` $\mathcal{E}_{k-1}$ and `54` $\mathcal{E}_k$ to form the next step in the square wave of error-vs-effort, and `55` the iterations continue.


```
C     eacyc.f: clock ea.f as it solves a problem
C
C     access desciptors from the problem definition
      COMMON /PROB/ NGC,N,MI,ME,XL,XH,XSTAR,LAMBDA
      REAL*8 XL(50),XH(50),XSTAR(50),LAMBDA(50)
C
C     type and dimension algorithm variables
      REAL*8 XK(50),XKP(50),QK(50,50),QKP(50,50)
      INTEGER*4 RC
C
C     prepare to count processor cycles
      INTEGER*8 CY1,CY2,CYALG
      REAL*8 ERR(601),EFF(601),FCN,FSTAR,CSE,ERZ
C
C     find starting point and ellipsoid matrix from bounds
      DO 1 J=1,N
           XK(J)=0.5D0*(XL(J)+XH(J))
           DO 2 I=1,N
                QK(I,J)=0.D0
    2      CONTINUE
           QK(J,J)=(DFLOAT(N)/4.D0)*(XH(J)-XL(J))**2
    1 CONTINUE
C
C     save starting error and effort
      FSTAR=FCN(XSTAR,N,0)
      ERZ=CSE(XK,N,FSTAR,LAMBDA,MI)
      KE=1
      ERR(KE)=0.D0
      CYALG=0
      EFF(KE)=DFLOAT(CYALG)
C
C     do more than enough iterations, one at a time
      DO 3 K=1,300
           CALL GETCYC(CY1)
           CALL EA(XK,N,QK,50,MI,1,0.D0, XKP,QKP,RC)
C
C          result of this iteration is starting point for the next
           DO 4 J=1,N
                XK(J)=XKP(J)
                DO 5 I=1,N
                     QK(I,J)=QKP(I,J)
    5           CONTINUE
    4      CONTINUE
C
C          save error and effort at this point
           CALL GETCYC(CY2)
           IF(RC .GT. 1) GO TO 6
           CYALG=CYALG+(CY2-CY1)
           KE=KE+1
           EFF(KE)=DFLOAT(CYALG)
           ERR(KE)=ERR(KE-1)
           KE=KE+1
           EFF(KE)=DFLOAT(CYALG)
           ERR(KE)=DLOG10(CSE(XKP,N,FSTAR,LAMBDA,MI)/ERZ)
    3 CONTINUE
C
C     write the (effort,error) coordinates to standard out
    6 WRITE(6,901) (EFF(K),ERR(K),K=1,KE)
  901 FORMAT(2(1X,1PE13.6))
      STOP
      END
```


The invocations of EA [35] all use a convergence tolerance of zero, so the ellipsoid algorithm iterations continue until $\mathbf{Q}_k$ becomes non-positive-definite or the function to be used for a cut has a zero gradient at $\mathbf{x}^k$. When one of those things happens EA returns RC > 1 and [47] control transfers out of the iteration loop to statement 6 [58] where the accumulated (effort,error) coordinates are written out. The terminal session excerpt below shows how I compiled the program to solve the ek1 problem and ran the resulting executable, redirecting its output to the file ek1.e.

```
unix[1] ftn eacyc.f ea.f matmpy.f cse.f ek1.f getcyc.c
unix[2] a.out > ek1.e
```

FORTRAN does not have built-in graphics so I used gnuplot to graph the ek1.e data, generating the error-vs-effort curve below.



The final data point in the file, for iteration 222, shows a cycle count of 2128548. Thus, on average one EA iteration takes about 9600 clock cycles, or 9.6 μs on a 1 GHz processor. It is not surprising that cputime(), with a resolution of 10000 μs, was unable to time single iterations of ea.m (the compiled code of EA runs much faster than ea.m, but probably not by a factor of 1000).

On the next page the listing of EA is too long for a single column so lines [75-103] are printed to the right of lines [1-74]. EA is closely modeled on ea.m (as you should convince yourself by comparing them) and it works the same way. In some places the two routines perform arithmetic operations in a different order, so there are tiny differences in the accumulation of roundoff error and the numbers they generate are not identical. However, throughout the solution process the $\mathbf{x}^k$ agree in at least the first 6 significant digits so for our purposes the MATLAB and FORTRAN implementations are numerically equivalent.

MATMPY is a matrix multiplication routine that is invoked [57,58,79] by EA. The final listing on the page is of CSE, a FORTRAN clone of the MATLAB cse.m routine. The GETCYC subprogram that we used above to read the cycle clock is written in the C programming language, and it is listed in [100, p501].


```
C     ea.f
C
      SUBROUTINE EA(XZERO,N,QZERO,LDQ,M,KMAX,TOL, XSTAR,QSTAR,RC)
C     do up to kmax iterations of the ellipsoid algorithm to solve
C     minimize fcn(x,0) subject to fcn(x,ii) <= 0, ii=1..m
C
C     declare formal parameters
      REAL*8 XZERO(N),QZERO(LDQ,*),TOL,XSTAR(N),QSTAR(LDQ,*)
      INTEGER*4 RC
C
C     declare local variables
      REAL*8 X(N),G(N),QG(N),D(N),XNEW(N)
      REAL*8 DDT(LDQ,LDQ),Q(LDQ,LDQ),QNEW(LDQ,LDQ)
      REAL*8 FN,A,B,C,FCN,NG,NSQ,GQG,DX
C
C     compute constants used in the updates
      FN=DFLOAT(N)
      A=1.D0/(FN+1.D0)
      B=2.D0*A
      C=FN**2/(FN**2-1.D0)
C
C     initialize current ellipsoid center and matrix
      DO 1 J=1,N
           X(J)=XZERO(J)
           DO 2 I=1,N
                Q(I,J)=QZERO(I,J)
    2      CONTINUE
    1 CONTINUE
      RC=1
C
      DO 3 K=1,KMAX
C          find a function to use in making the cut
           ICUT=0;
           DO 4 II=1,M
                IF(FCN(X,N,II) .GT. 0.D0) THEN
                   ICUT=II
                   GO TO 5
                ENDIF
    4      CONTINUE
C
C          find the gradient and normalize it
    5      CALL GRD(X,N,ICUT, G)
           NG=0.D0
           DO 6 J=1,N
                NG=DMAX1(NG,DABS(G(J)))
    6      CONTINUE
           IF(NG .EQ. 0.D0) THEN
              RC=3
              GO TO 7
           ELSE
              DO 8 J=1,N
                   G(J)=G(J)/NG
    8         CONTINUE
           ENDIF
C
C          find the direction in which to move the ellipsoid center
           CALL MATMPY(Q,LDQ,G,N,N,N,1, QG,N)
           CALL MATMPY(G,1,QG,N,1,N,1, GQG,1)
           IF(GQG .LE. 0.D0) THEN
              RC=2
              GO TO 7
           ELSE
              DO 9 J=1,N
                   D(J)=-QG(J)/DSQRT(GQG)
    9         CONTINUE
           ENDIF
C
C          check for convergence
           NSQ=0.D0
           DO 10 J=1,N
                DX=A*D(J)
                XNEW(J)=X(J)+DX
                NSQ=NSQ+DX**2
   10      CONTINUE
           IF(DSQRT(NSQ) .LT. TOL) THEN
              RC=0
              GO TO 7
           ELSE
              CALL MATMPY(D,N,D,1,N,1,N, DDT,LDQ)
              DO 11 J=1,N
              DO 11 I=1,N
                   QNEW(I,J)=C*(Q(I,J)-B*DDT(I,J))
   11         CONTINUE
           ENDIF
C
C          update the current ellipsoid center and matrix
           DO 12 J=1,N
                X(J)=XNEW(J)
                DO 13 I=1,N
                     Q(I,J)=0.5D0*(QNEW(I,J)+QNEW(J,I))
   13           CONTINUE
   12      CONTINUE
    3 CONTINUE
C
C     return the current point as optimal
    7 DO 14 J=1,N
           XSTAR(J)=X(J)
           DO 15 I=1,N
                QSTAR(I,J)=Q(I,J)
   15       CONTINUE
   14 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MATMPY(A,LDA,B,LDB,M,N,P, C,LDC)
C     compute the matrix product C(MxP)=A(MxN)*B(NxP)
C
      REAL*8 A(LDA,*),B(LDB,*),C(LDC,*)
      INTEGER*4 P
C
      DO 1 I=1,M
      DO 1 J=1,P
C          dot the Ith row of A with the Jth column of B
           C(I,J)=0.D0
           DO 2 K=1,N
                C(I,J)=C(I,J)+A(I,K)*B(K,J)
    2      CONTINUE
    1 CONTINUE
      RETURN
      END
```

```
      FUNCTION CSE(XK,N,FSTAR,LAMBDA,M)
      REAL*8 CSE,XK(N),FSTAR,LAMBDA(M),FCN
      CSE=DABS(FCN(XK,N,0)-FSTAR)
      DO 1 II=1,M
           CSE=CSE+DABS(LAMBDA(II)*FCN(XK,N,II))
    1 CONTINUE
      RETURN
      END
```


The `ek1` example shows that a resolution of 1 clock cycle is fine enough to permit accurate measurements of effort to be made even for the short statement sequences that result from excluding convenience code. Unfortunately, clock cycles elapse with wallclock time, so like `tic` and `toc` (though much more accurately) they count everything the processor does. For clock cycles to be a useful measure of the effort expended by an algorithm, it is necessary to keep the operating system from interrupting the instrumented program while we are conducting an experiment. In a Unix environment it is possible to do that (at least mostly) by taking certain draconian precautions [100, §18.5.4]. Random leakage of non-algorithm effort into cycle count measurements always makes the intervals look longer than they really are, so the noise can also be removed by repeating an experiment several times, saving each interval measurement, and combining the data to use the lowest cycle count observed for each interval.

Some computers adjust the processor clock speed dynamically to conserve battery charge or prevent chip overheating, but in a Unix environment it is possible to discover the current speed from within a running program [100, §18.5.5]. This number can be used to convert cycle counts into nanoseconds, and if only algorithm work is included the result is a very precise measurement of CPU time.

26.3.5 Problem Definition Files

The only piece of our `eacyc` program that remains to be discussed is the **problem definition file** `ek1.f`, which is listed on the next page. The `FCN` and `GRD` routines are straightforward transliterations into FORTRAN of `ek1.m` and `ek1g.m`, which we wrote in §24.4. The `HSN` routine computes Hessians for `ek1` in case we want to solve the problem using an algorithm that requires them.

The rest of the `ek1` problem definition consists of the descriptors I suggested in §26.2.1: n, m_i, m_e, $\mathbf{x}^{\mathrm{L}}$, $\mathbf{x}^{\mathrm{H}}$, $\mathbf{x}^\star$, $\boldsymbol{\lambda}^\star$, the provenance of the problem, aliases by which it is known, and the prefix string used to identify it in filenames. The prefix string `ek1` can be deduced from the filename `ek1.f`. The `BLOCK DATA` subprogram $\boxed{\texttt{3-14}}$ sets the values of the problem descriptors that are numbers, and provides in the variable `NGC` a problem number that can be used to access the appropriate record in a separate catalog file for the problem's provenance and aliases (and possibly other information). The problem number `29` refers to Subsection 29 in §28.7, which is our test problem catalog. Setting these quantities in code by initializing variables in the `COMMON` block `/PROB/` makes it possible to summarize in this single file all of the problem information that we need in order to use it in testing. Our program `eacyc.f` gets all of the `ek1` problem descriptors it requires from `/PROB/`.

The vectors `XL`, `XH`, `XSTAR`, and `LAMBDA` are $\boxed{\texttt{6,8,10,12}}$ each given `50` elements, more than the `2` that are needed for `ek1`, and the unused elements are $\boxed{\texttt{7,9,11,13}}$ initialized to zeros. This is so that the same standard layout can be used for the `COMMON` block `/PROB/` no matter what problem we want to describe, provided $n \le 50$ and $m_i + m_e \le 50$. Each of the nonlinear programs we have considered in this book could be defined in this compact way.


```
C     ek1.f
C
      BLOCK DATA
      COMMON /PROB/ NGC,N,MI,ME,XL,XH,XSTAR,LAMBDA
      INTEGER*4 NGC/29/,N/2/,MI/3/,ME/0/
      REAL*8 XL(50)/11.63603896932107D0,11.80761184457488D0,
     ;                  48*0.D0/
      REAL*8 XH(50)/24.36396103067893D0,30.19238815542512D0,
     ;                  48*0.D0/
      REAL*8 XSTAR(50)/15.62949090230634D0,15.97376861785225D0,
     ;                  48*0.D0/
      REAL*8 LAMBDA(50)/250.9965343846114D0,0.D0,0.D0,
     ;                  47*0.D0/
      END
C
      FUNCTION FCN(X,N,II)
      REAL*8 FCN,X(N)
      IF(II.EQ.0) FCN=(X(1)-20.D0)**4+(X(2)-12.D0)**4
      IF(II.EQ.1) FCN=8.D0*DEXP((X(1)-12.D0)/9.D0)-X(2)+4.D0
      IF(II.EQ.2) FCN=6.D0*(X(1)-12.D0)**2+25.D0*X(2)-600.D0
      IF(II.EQ.3) FCN=-X(1)+12.D0
      RETURN
      END
C
      SUBROUTINE GRD(X,N,II, G)
      REAL*8 X(N),G(N)
      IF(II.EQ.0) THEN
         G(1)=4.D0*(X(1)-20.D0)**3
         G(2)=4.D0*(X(2)-12.D0)**3
      ELSEIF(II.EQ.1) THEN
         G(1)=8.D0*DEXP((X(1)-12.D0)/9.D0)*(1.D0/9.D0)
         G(2)=-1.D0
      ELSEIF(II.EQ.2) THEN
         G(1)=6.D0*2.D0*(X(1)-12.D0)
         G(2)=25.D0
      ELSEIF(II.EQ.3) THEN
         G(1)=-1.D0
         G(2)= 0.D0
      ENDIF
      RETURN
      END
C
      SUBROUTINE HSN(X,N,II, H,LDH)
      REAL*8 X(N),H(LDH,*)
      H(1,1)=0.D0
      H(2,1)=0.D0
      H(1,2)=0.D0
      H(2,2)=0.D0
      IF(II.EQ.0) THEN
         H(1,1)=12.D0*(X(1)-20.D0)**2
         H(2,2)=12.D0*(X(2)-12.D0)**2
      ELSEIF(II.EQ.1) THEN
         H(1,1)=(8.D0/81.D0)*DEXP((X(1)-12.D0)/9.D0)
      ELSEIF(II.EQ.2) THEN
         H(1,1)=12.D0
      ENDIF
      RETURN
      END
```


26.3.6 Practical Considerations

The programs `eaefe.m`, `eacpu.m`, and `eacyc.f` were easy to write, because both `ea.m` and its FORTRAN equivalent `ea.f` can be invoked repeatedly to solve a problem one iteration at a time. Often it is of interest to evaluate an algorithm whose implementation is *not* serially reusable. Then the progress of the method from one iteration to the next can be monitored only within the user-supplied routines that it invokes during each iteration to compute function, gradient, and Hessian values. If CPU time or cycle count is being used as the measure of effort, the timing or counting must be suspended in those routines while the error and effort measures are updated and stored or written to a file; in that case stubs must be used between the algorithm code and the routines that define the problem.

Both `ea.m` and `ea.f` also have the property that *all* of the computational effort they expend can rightly be accounted to the algorithm they implement. That made it possible for us to exclude all non-algorithm EFEs, CPU time, or processor cycles by instrumenting only the test program and (via stubs) the problem-defining routines that we supplied. If an implementation to be tested does things *other* than carry out the steps of the algorithm, such as printing status reports, then it too must be instrumented so that those activities are excluded from the measured effort. This is possible only if the source code can be modified.

In `eacpu.m` and `eacyc.f` we bracketed the code segments to be measured with invocations of `cputime()` or `GETCYC`, and added statements to increment `talg` by `u2-u1` or `CYALG` by `CY2-CY1`. This way of instrumenting the code assumes that there are exactly two categories of computational effort, algorithm and non-algorithm. In some performance evaluations it is desirable to partition effort into more than two categories so that, for example, the work of the updates can be compared to the work of evaluating functions, gradients, and Hessians. We have also assumed that `cputime()` and `GETCYC` return their outputs instantaneously, but executing either routine actually takes some computational effort. In practice it is both more convenient and more accurate to encapsulate the effort-measurement process in a routine that corrects for its own overhead and simplifies the accounting of effort to different categories. For example, the `TIMER` routine described in [100, §15.1.4], which returns overhead-corrected CPU times based on cycle counts, supports a simple conceptual model of computational effort in which execution time flows continuously and is redirected by each `TIMER` call into a specified timing bin.

In a MATLAB program our source code is interpreted one statement at a time, so the calculation that is performed is precisely the one we specified. When an algorithm is implemented in a compiled language, hidden optimizations introduced by the compiler can rearrange the calculations in such a way that the algorithm actually carried out by the executable is subtly different from the one described by the source program. I mentioned in §26.1.2 that this phenomenon can invalidate our definition of precisely what the algorithm is. It can also have a disastrous effect on instrumented code, by changing what sequence of operations a measurement includes or by "factoring out" the measurement altogether. Instrumented source code must therefore be compiled using options that prevent code rearrangement.

In `eaefe.m` and `eacpu.m` we collected (effort,error) coordinates in arrays and graphed them within the test program, but in conducting a real study it is more convenient to write or redirect each set of performance results to a file. That way each algorithm can be tested separately and a different program used afterward to read the files for the algorithms to be compared and produce an error-vs-effort curve that includes them all. Sometimes a program under test finds the optimal solution exactly, so that an iterate has $\varepsilon_k = 0$ and $\mathcal{E}_k = -\infty$; that must be indicated somehow on the graph but not allowed to spoil its vertical scaling.

The measures of effort that we have considered all assume the simplest and most typical computer architecture, in which a single processor is running a single program at any given instant, in a single memory. Much current research (e.g., [129]) is focused on the development of optimization algorithms that can exploit parallel processing and distributed memory. The performance of each scalar process that makes up a parallel algorithm can be studied using the techniques discussed above. When multiple processes are run in parallel, however, other measures of algorithm quality must also be considered, including the wallclock time required to solve a problem (reducing this time is the goal of parallel processing) and how that measure of performance and the memory footprint of each process scale with the number of processors used.

26.4 Testing Environment

Algorithm performance evaluation is based on measurements made during computational experiments. The laboratory instrument that we use to make those measurements is an instrumented computer program. In the examples we have studied the **test program** consists of a main routine or **driver**, an algorithm implementation or **solver** subprogram that is invoked by the driver, and a problem definition that is invoked by the solver. To make accurate measurements of CPU time, or to measure cycle counts, all of this code must be written in a compiled programming language such as FORTRAN, C, or C++ rather than in a high-level package such as MATLAB, AMPL, or Maple. To be suitable for testing optimization software, a computing environment must therefore support the writing, compilation, and maintenance of computer programs. It needs at least a text editor, a language compiler, and a program management utility such as `make`.

A serious computational study often uses several test programs to solve multiple test problems, generating many sets of performance data to be analyzed using other programs. The various pieces of code, the experimental data, and the results of the analyses are all stored in files. To be practical a testing environment must therefore provide some way to automate the uninterrupted running of the experiments and the manipulation of the associated files.

These requirements strongly favor the Unix operating system. It provides program development tools and a way to write software for systematically managing experiments and the files they produce and consume, and it can be made to surrender control of the processor to a user program and thereby get out of the way for the duration of an experiment.

26.4.1 Automating Experiments

Suppose that three test programs are to be used to solve each of twenty test problems, and that an error-vs-effort curve is to be produced comparing the performance of the algorithms on each problem. The pieces that make up each test program are stored in separate FORTRAN source code files. What must be done to carry out this computational testing plan? If you were to do it by typing at the command prompt, your interactions with Unix might begin something like this.

```
unix[1] ftn driver1.f alg1.f prob1.f
unix[2] a.out > p1a1.e
unix[3] ftn driver2.f alg2.f prob1.f
unix[4] a.out > p1a2.e
unix[5] ftn driver3.f alg3.f prob1.f
unix[6] a.out > p1a3.e
unix[7] perfplot p1a1.e p1a2.e p1a3.e
unix[8] echo 'load "p1.gnu"' | gnuplot
:
```

Here I have assumed `ftn` is a compiler that translates each `.f` file named in its argument list and links the resulting object modules to produce an executable named `a.out`. For example, the first invocation `[1]` of `ftn` combines the driver routine for algorithm 1 with the subprogram implementing algorithm 1 and the problem definition file for problem 1. Each driver routine writes (effort,error) coordinates to its standard output, which is redirected to a file whose name encodes the problem and algorithm that were used to generate it. For example, the output of the executable that solves problem 1 using algorithm 1 is redirected `[2]` to `p1a1.e`

I have also assumed `[7]` the existence of a program named `perfplot`, which reads error-vs-effort data from the files given as its parameters and writes two output files. The first of these is a set of plotting instructions similar to the `rays.gnu` file described in §3.6.1; the second is a file similar to `rays.dat` containing the three sets of error-vs-effort data, censored if necessary to deal with points having $\mathcal{E}_k = -\infty$ (in that case commands must be added to the `.gnu` file for annotating the graph accordingly). Piping `[8]` the command `load "p1.gnu"` into `gnuplot` causes it to generate an appropriately-named `eps` file containing the error-vs-effort graph, which we could later print or include in a LaTeX 2_ε document.

So far we have run experiments for only the first of the twenty problems, so there is a lot of typing ahead. Fortunately, repetitive command sequences like this can be automated in Unix by writing a **shell script** [96] such as the one on the next page. Entering the single command `expts` at the Unix command prompt would run all of the experiments.

Depending on the computational testing plan, the shell script you write to run the experiments might be much more complicated than this one. You might need to modify and test the script repeatedly until you get it right, but because it is just text in a file that is much easier than typing a long sequence of lines perfectly at the interactive command prompt. Once the script is correct you can go to lunch while it executes, confident that the right program, problem, and data files will be used in each step.


```
#! /bin/sh
#  expts: run programs 1-3 on problems 1-20
for pr in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
    for ag in 1 2 3
    do
        ftn driver${ag}.f alg${ag}.f prob${pr}.f
        a.out > p${pr}a${ag}.e
    done
    perfplot p${pr}a1.e p${pr}a2.e p${pr}a3.e
    echo 'load "p${pr}.gnu"' | gnuplot
done
exit 0
```

26.4.2 Utility Programs

In addition to running a series of experiments, many other tasks that frequently arise in carrying out a performance evaluation project can be greatly simplified and speeded up by writing and using your own collection of utility programs.

We used the `eacyc.f` program of §26.3.4 to generate an error-vs-effort curve for the `ek1` problem, but it could just as easily be used to study any other inequality-constrained problem for which we have a problem definition file. All we need to do is replace `ek1.f` by the other problem definition in the Unix command we use to compile the program.

We used `ek1.f` in preparing an executable of `eacyc.f`, but it could just as easily be linked into programs that answer other questions about the problem. Is a certain point feasible? Does it satisfy the KKT conditions? What is its objective value? We could also link `ek1.f` with drivers and solvers implementing other algorithms. Of course the same programs that do these calculations for `ek1` can do them for any problem if we link in the right definition. The task of building an executable that combines a given test problem with a given utility or driver and algorithm implementation can itself be automated using a shell script.

In §26.2.2, I outlined some complicated rules for constructing bounds. These could be used by a program that links to a problem definition but ignores the `XL` and `XH` vectors given there. For each variable j it could ask the user whether x_j^L, x_j^H, or x_j^0 is known, and use whatever values are given to compute new bounds from the appropriate equation.

What is the lowest $\mathcal{E}_k$ achieved by a given algorithm on a given problem? This can be discovered by examining the appropriate `.e` file produced in an experiment. Among these lowest errors achieved by the given algorithm across the whole set of test problems, which is the highest? The answers to statistical questions like this can be obtained by examining all of the `.e` files with a program, written in a compiled language, that is run on each file in turn by a shell script. The `perfplot` program that I envisioned for the example of §26.4.1 is another utility of this sort.

In reporting statistical results it is often appropriate to include tables of values (see §26.5.1). These are tedious to typeset and to populate with the right numbers, so it can be worth the trouble to write a program that gathers or calculates the entries and generates LaTeX source text for setting the tables.

26.5 Reporting Experimental Results

As Richard Hamming famously sermonized [166, p3], "The purpose of computing is insight, not numbers." When we use numerical optimization to study a practical problem, the results we get are already once removed from the application; when we use computational experiments to study the numerical algorithm itself, our measurements are separated from reality by an additional layer of abstraction. How can we summarize and interpret a deluge of observational data in ways that lead to useful insights about the algorithms we tried?

26.5.1 Tables

To compare the behavior of algorithms when they are used to solve a single problem, only an error-vs-effort curve will do. But one such picture provides too little information to say which method works best in general, and twenty such pictures (a lot for any journal to publish) would provide too much information for a reader to assimilate just by looking at them. To comprehend the whole portfolio of results from a computational study it is necessary to summarize them. One way to do that is in tables; these are the standard types.

algorithm	LRCSE level $\mathcal{E}$ −2	−4	−8	−12
A	0.0	0.1	0.2	0.0
B	0.1	0.2	0.3	0.5
C	0.9	0.7	0.5	0.0

fraction of problems solved first

algorithm	LRCSE level $\mathcal{E}$ −2	−4	−8	−12
A	1.0	1.0	1.0	0.0
B	1.0	1.0	0.9	0.5
C	1.0	0.9	0.8	0.0

fraction of problems solved

The table on the left shows that algorithm C usually achieves error levels down to −8 more quickly than the other algorithms, but neither A nor C ever achieves an error level of −12 while B reaches that level on half of the problems. These results suggest that C is most efficient but B is most accurate.

The table on the right shows that A solves more of the problems to −8 than either of the other algorithms. If all three should have been able to solve all of the problems that were used in computing these statistics, this result suggests that A is the most reliable of the algorithms down to that error level. If the problems lack some property necessary to prove convergence of the algorithms (e.g., if these are ellipsoid algorithm variants but none of the problems is a convex program) then the result suggests that A is the most robust.

Depending on the goals of the study it might be appropriate to table, for each algorithm, other attributes such as

- its best possible accuracy, the lowest error level attained on any problem;
- its **sensitivity** to imprecisions in the calculation of function and derivative values;
- its **stability**, whether it stays at $\mathbf{x}^\star$ if that is used as the starting point [98, p65].

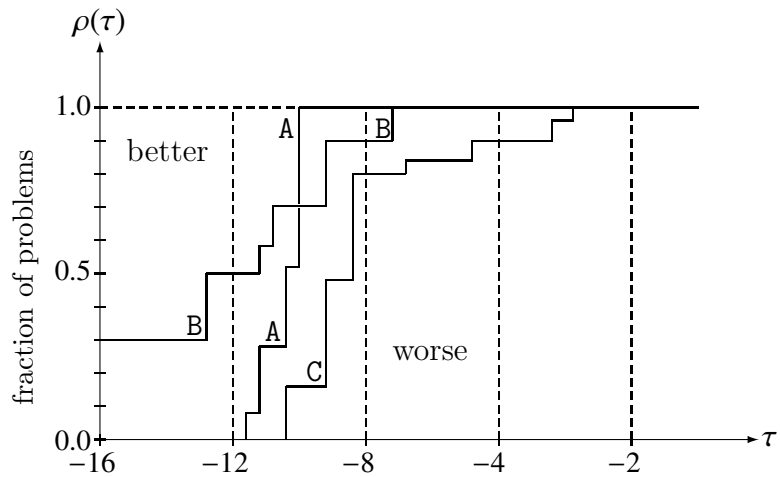
26.5.2 Performance Profiles

Another way to summarize results over the whole set of test problems is by using performance profiles [44] [137, §5]. A **performance profile** is a cumulative distribution function $\rho_s(\tau)$ for a **performance metric** $f_{p,s}$ of algorithm s over the problems p in the test set.

Above I suggested tabulating, for each algorithm, the lowest error level it attained on *any* problem. A more complete picture of ultimate accuracy can be had by plotting $\rho_s(\tau)$ for that performance metric (but see [68]). If we let

$$\begin{aligned} f_{p,s} &= \text{lowest } \mathcal{E} \text{ attained by algorithm } s \text{ on problem } p \\ \rho_s(\tau) &= \frac{\text{number of problems having } f_{p,s} \le \tau}{\text{number of problems in the test set}} \in [0,1] \end{aligned}$$

then we can plot $\rho(\tau)$ as a function of τ like this.



Algorithm `B` is most likely to work if we require $\mathcal{E} < -10$; otherwise we should use `A`.

Other performance metrics require more subtle analysis. For example, if we let

$$\begin{aligned} f_{p,s} &= \text{effort for algorithm } s \text{ to reach its lowest } \mathcal{E} \text{ on problem } p \\ \hat{f}_{p,s} &= \text{effort for algorithm } s \text{ to reach } \textbf{reference error } \mathcal{E} = -3 \text{ on problem } p \end{aligned}$$

then the **performance ratio**

$$r_{p,s} = \frac{f_{p,s}}{\min\limits_s \hat{f}_{p,s}}$$

is a dimensionless number normalized for the difficulty of problem p, and the performance profile is

$$\rho_s(\tau) = \frac{\text{number of problems having } \log_{10}(r_{p,s}) \le \tau}{\text{number of problems in the test set}}.$$

26.5.3 Publication

In §26.2, I advocated sharing the test problems used in every study along with the algorithm implementations that are tested. In order for other people to be able to confirm the results through independent replication of the experiments, it is also necessary for them to know the details of the computing environment that you used. This includes

- the processor chip,
- the operating system,
- the language compilers and options,
- the algorithm parameter settings, and
- if CPU time or cycle counts were the measure of effort, any precautions you took to ensure that the measurements were accurate and free of noise.

For your experimental results to be publishable it should at least be possible for you yourself to replicate them. If CPU time or processor cycles are the measure of effort, repeat the experiments to provide an estimate of the variability in those measurements. If an algorithm fails on some problems, explain why.

Computational studies are as difficult to publish as they are to conduct. Some journal editors and many anonymous referees dismiss "experimental mathematics" as a last resort of incompetents, and recoil from its unhygienic contact with actual computing; others have had bitter experience with algorithm evaluations that were badly done, with which the literature is unfortunately replete. If your paper is accepted it will probably be on condition that you shorten it; there is never enough space to tell the whole story. Publish a summary of your findings, citing an unabridged report that interested readers can easily obtain.

Computational comparisons are perilous when, in the process of drawing contrasts between *algorithms,* they reveal shortcomings of the *implementations* that are tested. An algorithm might be public property, but every implementation has an author whose feelings (and perhaps tenure case) are at stake. If you find some flaw in another person's work report it to the person privately, and when you cannot avoid printing bad news do so as gently as possible. Science often progresses through public discussion, but argument should always be for the sake of getting to the truth rather than for the sake of humiliating your competition [178, §5:20]. Label your speculations to distinguish them from supported conclusions, and remember that only very limited claims can be made about proprietary codes.

Performance (in all its aspects) is sufficiently important that I have devoted many pages to techniques for evaluating it, but other factors also affect the utility of an algorithm. A publication reporting your findings will be most useful to other practitioners if it also mentions how to get the implementations you tested, how easy you found the software to install and use, and any practical advice you can offer based on your experience.

26.6 Exercises

26.6.1 [E] I claimed in §26.0 that the performance of nonlinear optimization algorithms actually matters. (a) List the aspects of algorithm performance that are mentioned in this Chapter. (b) Explain *why* performance matters. Is speed the only aspect that matters?

26.6.2 [E] Why is it difficult to predict the performance of an optimization algorithm by analyzing it mathematically?

26.6.3 [E] (a) Developers and users of optimization algorithms often conduct informal computational experiments. Why do they do that? (b) A few of them conduct computational studies that are much more formal, careful, and difficult. Why do they do that?

26.6.4 [E] What is the logical basis or fundamental assumption of computational testing? What role do computer programs play?

26.6.5 [E] List three important issues that arise in the experimental study of optimization methods.

26.6.6 [E] Explain the difference between an algorithm and a computer program that implements the algorithm. What are *invariant properties*, and how can they be used to specify an algorithm?

26.6.7 [E] Explain how the algorithm definition we adopt affects the tradeoff between the generality and the strength of the conclusions that we can draw about the algorithm from observations of an implementation. In how much detail should an algorithm be specified for the purposes of computational testing?

26.6.8 [P] Newton descent has second-order convergence, but computing Hessians and finding the Newton direction take a lot of work. (a) Describe experiments whose results can be used to determine whether Newton descent is really faster than steepest descent. (b) Present a specification of each algorithm that is appropriate to this study. (c) Carry out your test plan and explain your findings.

26.6.9 [E] How can a computational experiment be structured to minimize the effects of (a) differences in algorithm implementation; (b) differences in algorithms?

26.6.10 [E] What is *convenience code*, and how can it be excluded from measurements of computational effort?

26.6.11 [E] How does the *reliability* of an algorithm differ from its *robustness*?

26.6.12 [E] What role do the adjustable parameters of an algorithm implementation play in computational testing?

26.6.13 [E] How should the test problems be chosen for a computational study? How should the starting points be determined?

26.6.14 [E] What is the function of a test problem *catalog*? What attributes of a test problem should be cataloged? Which are best specified in a problem definition file, and which in a separate catalog file?

26.6.15 [E] Why is it necessary to validate test problems before using them in a computational study?

26.6.16 [E] Why is it important for a test problem's bounds to be determined in an unbiased way? What requirements should be satisfied by a problem's catalog bounds?

26.6.17 [H] Three examples are used in §26.2.2 to illustrate how limits on the x_j can be deduced from a problem statement. Use the formulas given there to compute catalog bounds for each problem.

26.6.18 [H] In §24.3.1 we used these variable bounds for problem `ek1`.

$$\begin{aligned} \mathbf{x}^{\mathrm{H}} &= \left[18 + 9/\sqrt{2},\ 21 + 13/\sqrt{2}\right]^{\mathsf{T}} \\ \mathbf{x}^{\mathrm{L}} &= \left[18 - 9/\sqrt{2},\ 21 - 13/\sqrt{2}\right]^{\mathsf{T}} \end{aligned}$$

Are these the tightest bounds you can deduce from the constraints of the problem? If not, find tighter bounds.

26.6.19 [H] Suppose that a nonlinear program includes the constraints

$$\begin{aligned} 4t_3t_5^{-1} + 2t_3^{-0.71}t_5^{-1} + 0.0588t_3^{-1.3}t_7 - 1 &< 0 \\ t_j &> 0, \quad j = 1 \ldots 8. \end{aligned}$$

Show how these inequalities can be used to establish the lower bound $t_5 > 2.666975697132930$.

26.6.20 [P] Suppose that a nonlinear program includes the constraint

$$e^{-x_1} + x_1^2 + x_2^2 \leq 15.$$

Show how this inequality can be used to establish the upper bound $x_2 \leq 3.764680062617868$.

26.6.21 [P] Write a program that gets $\mathbf{x}^\star$ for a test problem, prompts the user for each x_j^L, x_j^H, and x_j^0, and then uses the appropriate formula from §26.2.2 to find catalog bounds. (a) Use MATLAB. (b) Use FORTRAN or another compiled language of your choice.

26.6.22 [H] Of the eight formulas given in §26.2.2 for computing catalog bounds, which can produce bounds that exclude the optimal point? If that happens, how can the bounds be adjusted to include $\mathbf{x}^\star$?

26.6.23 [H] The catalog entry of §28.7.2 for `rb` and the catalog entry of §28.7.4 for `gns` each specify a starting point $\mathbf{x}^0$ that is *not* the midpoint of the catalog bounds. I did this so that I could use the bounds to delimit the contour plots in §9.1 and §10.4 with starting points that are not centered in those pictures. (a) Use the appropriate algorithm from §26.2.2 to construct bounds symmetric about $\mathbf{x}^0$ for each of these problems. (b) Use the formula $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}})$ to find a starting point $\mathbf{x}^0$ that is centered in the catalog bounds for each

of these problems. (c) To ensure fairness in computational testing we have adopted the convention that $\mathbf{x}^0$ should be the midpoint of the bounds. If each of these problems is to be used in a test program, what should be changed, its starting point or its bounds?

26.6.24 [H] In explaining the idea of a restricted-steplength algorithm in §17.1 I found it convenient to use two different starting points $\bar{\mathbf{x}}^0 = [2.5, 0.3]^\top$ and $\hat{\mathbf{x}}^0 = [1, 0.6]^\top$ for h35 (see §28.7.18), neither of which is the starting point $\mathbf{x}^0 = [2, 0.2]^\top$ given in the original problem statement [80, p122,401]. (a) Which starting point is the midpoint of the catalog bounds given in §28.7.18? (b) Use the appropriate algorithm from §26.2.2 to construct bounds symmetric about $\mathbf{x}^0$. (c) If this problem is to be used in a test program, what starting point and bounds should be used?

26.6.25 [E] Research articles sometimes compare algorithms by stating the number of iterations each used to solve a particular problem or by plotting graphs of distance error e_k versus the iterations k they used in solving the problem. (a) Explain why neither of these comparisons is very informative. (b) What interesting algorithm property *can* be deduced from a graph of e_k/e_0 versus k? (c) Explain how e_k/e_0 can be misleading when used as a measure of solution error in comparing algorithms. (d) What is the advantage of using an error-vs-effort curve, rather than graphs of e_k/e_0 versus k, in comparing algorithms?

26.6.26 [H] What is the definition of combined solution error ε, and what are its desirable properties? Why can't it be used in studying a problem that lacks a constraint qualification? Does it have other drawbacks?

26.6.27 [E] What is the definition of LRCSE? What is the numerical value of $\mathcal{E}_0$, and why?

26.6.28 [E] In §26.3.2 we assumed that a gradient evaluation requires about n times as much work as a function evaluation and a Hessian evaluation requires about $\frac{1}{2}n(n+1)$ times as much. (a) What rationale was given for using these multiples? (b) What multiples would be appropriate if central difference approximations were used to compute gradients and Hessians?

26.6.29 [E] What is a *stub routine,* and why might we use one?

26.6.30 [E] Why is an error-vs-effort curve always a square wave?

26.6.31 [P] The `eaefe.m` program of §26.3.2 plots an error-vs-effort curve for `ea.m` when it is used to solve the `ek1` problem. (a) Revise the `nlpin.m` routine of §21.3.1 to make it serially reusable. Hint: this involves making the starting value of `mu` an input parameter and returning its final value as `mustar`, and making the loop limit an input parameter `kmax` rather than the fixed number `52`. (b) Enlarge the `eaefe.m` program to also plot, on the same set of axes, an error-vs-effort curve for `nlpin.m` when it is used to solve the `ek1` problem. Hint: you will need to write a stub routine `ek1hefe.m` to update `NHE` before each Hessian evaluation. (c) Run your program and interpret the error-vs-effort curve that it produces.

26.6.32 [H] In using EFE we assume that each function value, gradient component, or Hessian component takes the same amount of work. Is this true for the ek1 problem? What are the possible sequences of function and gradient evaluations that might be performed in an iteration of the ellipsoid algorithm when solving that problem?

26.6.33 [E] When is it reasonable to assume that most of the effort required to solve a nonlinear program is spent in evaluating functions, gradients, and Hessians? Why is this assumption often *un*reasonable?

26.6.34 [E] (a) What is the difference between wallclock and CPU time? How is it possible in MATLAB to measure (b) wallclock time; (c) CPU time. (d) With what precision can MATLAB measure CPU time on your computer?

26.6.35 [E] (a) Why does the number of EFEs used by an algorithm underestimate the effort it requires to solve a problem? (b) Why does the CPU time used by a test program overestimate the CPU time used by the algorithm under test?

26.6.36 [E] How is it possible to avoid timing convenience code? Why is this difficult to do in practice?

26.6.37 [H] The GETCYC subroutine described in §26.3.4 returns the current cycle of the processor clock. (a) Explain how it can be used to count the clock cycles that elapse in performing a given sequence of program statements. (b) How is it possible for a compiler to affect this measurement? (c) If the code is executing on a processor with a clock speed of 2 GHz, what interval of time corresponds to each clock cycle? (d) Is cycle counting a good way to measure CPU time? Explain.

26.6.38 [E] Describe the advantages and drawbacks of using the following measures for computational effort; (a) iteration count k; (b) equivalent function evaluations EFE; (c) CPU time; (d) processor cycle count.

26.6.39 [P] Random leakage of non-algorithm effort into cycle count measurements always makes the intervals look longer than they really are. (a) Describe in detail how this noise could be filtered out of the measurements made in eacyc.f. (b) Revise eacyc.f to implement your plan. (c) Run the resulting test program on a machine that you are also using for other tasks, and show that the resulting contamination of the interval measurements is effectively removed by your filtering scheme.

26.6.40 [H] By instrumenting a program we can avoid timing (or counting the cycles used by) convenience code. Would it be useful to adopt as a definition of what the algorithm is that "the algorithm is what gets timed"? Explain.

26.6.41 [H] In the early days of mathematical programming, to permit the comparison of effort measurements made on different computers CPU times were sometimes expressed as multiples of a **standard timing unit** [28, Appendix III], the time required to invert a certain 40×40 matrix ten times. This turned out not to work very well [80, p368-369]. Can you think of some possible reasons why?

26.6.42 [P] The problem definition file described in §26.3.5 identifies the `ek1` test problem `NGC29`. (a) Write a problem definition file for the test problem `NGC35`. (b) (historical research) I named the variable containing a test problem's number `NGC`, for New General Catalog. In what field of science was this acronym originally used?

26.6.43 [E] Why is it advantageous to define a nonlinear programming test problem by constructing a problem definition file for it? In the problem definition file of §26.3.5, why are the vectors `XL`, `XH`, `XSTAR`, and `LAMBDA` dimensioned 50 elements long, independent of the number of variables or constraints in the problem?

26.6.44 [E] What is involved in making CPU time or clock cycle measurements when testing an algorithm whose implementation (a) is not serially reusable; (b) does things other than perform the steps of the algorithm?

26.6.45 [P] Each invocation of the MATLAB function `cputime()` itself consumes some CPU time, though far less than its resolution. Write a MATLAB program to measure this overhead.

26.6.46 [E] Describe two measures of quality that are important for a parallel algorithm.

26.6.47 [E] What parts make up a *test program* for experimenting with a nonlinear programming algorithm?

26.6.48 [P] Write a `perfplot` program that reads (effort,error) coordinates from each of several `.e` files and writes two output files. One output file should contain plotting instructions for `gnuplot` and the other should contain the multiple data sets separated by blank lines. An input `LRCSE` value of $-\infty$ should be modified by the program so that the graph drawn by `gnuplot` descends to the bottom of the frame and is marked with an arrow to show that it is a zero-error point. The resultant scaling of the vertical axis should be such that most of the graph is filled by the parts of the curve that have nonzero error values.

26.6.49 [E] In a Unix environment, how can a repetitive command sequence be automated?

26.6.50 [E] Describe one utility program that might be handy in carrying out a performance evaluation project.

26.6.51 [E] Two standard types of summary table are described in §26.5.1. What are they?

26.6.52 [P] (a) What is a *performance profile*? (b) Write a program that reads a `.e` file of (effort,error) coordinates and writes out the coordinates of a performance profile for the lowest error level attained, as described in §26.5.2.

26.6.53 [E] List some details of the computing environment that should be mentioned in reporting the results of computational experiments with algorithms for nonlinear programming. Why can only very limited claims be made about codes that are proprietary?

26.6.54 [H] Occasionally an algorithm developer finds a method whose implementation turns out to be in some way superior to a widely-respected solver. Delighted by his surprising good fortune, he might gratify his ego by presenting the results in a way that places more emphasis on the shortcomings of the other code than on the merits of his own. Explain why this is always a bad idea, and suggest an alternative way of reporting such a discovery.

26.6.55 [P] In the `eaefe.m` and `eacpu.m` programs of §26.3.3, I initialized `Qk=[80,0;0,169]` for the test problem `ek1`. But in §24.3.1 we found from the catalog bounds for that problem a starting ellipsoid that has

$$\mathbf{Q}_0 = \begin{bmatrix} 81 & 0 \\ 0 & 169 \end{bmatrix}$$

so the results we obtained here are not precisely what they should have been. (`a`) Correct the mistake and rerun the experiments. Do the detailed observations change? Do the conclusions change? (`b`) In running computational experiments and reporting their results, how important do you think it is to avoid little mistakes of this sort? Should the discovery of such a mistake warrant the withdrawal of a research paper that has already been published? (`c`) Is the initial `Qk` computed correctly in the `eacyc.f` program of §26.3.4? What object lesson can you draw from that?

27

`pivot`: A Simplex Algorithm Workbench

In §2.7, I introduced the `pivot` utility as a hypothetical program defined abstractly by the user's manual in §27.1. It assumes that the standard-form linear program

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & d + \mathbf{c}^\top\mathbf{x} & & \\ \text{subject to} & A_1\mathbf{x} & = & b_1 \\ & A_2\mathbf{x} & = & b_2 \\ & \vdots & \vdots & \vdots \\ & A_m\mathbf{x} & = & b_m \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

is represented by the $(m+1)\times(n+1)$ tableau

$$\mathbf{T} = \begin{array}{c|c|c|c|c|c|} & x_1 & x_2 & x_3 & \dots & x_n \\ \hline -d & \multicolumn{5}{c|}{\mathbf{c}^\top} \\ \hline \mathbf{b} & \multicolumn{5}{c|}{\mathbf{A}} \\ \hline \end{array}$$

upon which we will perform various operations. Among these operations the most important in applying the simplex method is the pivot, which I described in §2.3 like this.

- We are given $h \in \{1\dots m\}$, the index in $\mathbf{A}$ of the pivot row, and $p \in \{1\dots n\}$, the index in $\mathbf{A}$ of the pivot column, specifying a pivot element $a_{hp} \neq 0$.
- We divide the pivot row of the tableau by the pivot element. This makes the pivot element equal to 1.
- We add multiples of the resulting pivot row to the other rows of the tableau to get zeros elsewhere in the pivot column.

Because the simplex method involves pivots only on elements of the constraint matrix $\mathbf{A}$, the indexing scheme used in this description makes the objective row correspond to $h = 0$ and the constant column correspond to $p = 0$. In pivoting on a computer it is more convenient to talk about the whole tableau $\mathbf{T}$ rather than just its $\mathbf{A}$ part, so here we will index the rows by `i` $= h+1$ and the columns by `j` $= p+1$. Then the objective is row `i = 1` and the constant column is column `j = 1`. We will call the number of tableau columns `n` $= n+1$ and the number of tableau rows `m` $= m+1$,

27.1 Commands

Each **command** of the `pivot` program is described on a separate page of this Section, and the pages are arranged in alphabetical order by command name.

Each page begins with a command **prototype** showing the full command name in `vertical` typewriter font and the command's **parameters**, if it has any, in *`slanting`* typewriter font. The initial letter or letters of the command name are capitalized to show the **minimum unambiguous abbreviation** that can be entered to give the command. Parameters appearing in brackets *`[ ]`* are omitted in some forms of the command; whether or not the parameters are used, the brackets themselves should never be included in the command. After each command prototype comes a more thorough description of the command, including any limits on the parameter values. Then there is a session excerpt illustrating the use of the command. At the bottom of the page there might be further information or advice about using the command.

The `help` and `stop` commands have **aliases** which are described on their own pages, each of which also lists the other names for the command.

The names that are used in the command prototypes to represent parameters should be replaced by either numerical or character values as appropriate. The multiplier *`s`* of the `scale` command is a floating-point number, as are the link cost and supply-minus-demand values that you are prompted to enter by `gnf` and the tableau elements that you are prompted to enter by `insert`. The examples in the table below show some acceptable ways to specify these floating-point values.

input	value represented
`0`	0.0
`-0.`	0.0
`0.0`	0.0
`-0.e0`	0.0
`+6`	6.0
`-6.023`	−6.023
`6.023E23`	6.023×10^{23}
`-0.004`	−0.004
`4e-3`	0.004
`-4.0E+02`	−400.0

The examples used in the command descriptions (like most of the linear programs discussed elsewhere in the text) have starting data that happen to be small whole numbers, but all `REAL*8` values [100, §4] conforming to the IEEE floating-point standard [84] are acceptable to the `pivot` program as real-number data.

All of the command parameters that are not floating-point numbers are either integers, which should be entered without a decimal point, or character strings, which should be entered verbatim, without quotation marks. A zero first tableau index denotes all of the rows, a zero second index all of the columns.

Append *newrows newcols*

Resize the tableau by adding *newrows* rows at the bottom or *newcols* columns at the right, or both.

```
< list

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.

< append 1

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.
   0.  0.  0.  0.  0.  0.  0.  0.

< append 0 1

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.  0.
  80.  0.  0.  4.  4.  1. -1.  1.  0.
 110.  0.  1. -1.  1.  3.  1.  0.  0.
  20.  1.  0.  2.  3. -4.  2.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.

< append 2 3

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.  0.  0.  0.  0.
  80.  0.  0.  4.  4.  1. -1.  1.  0.  0.  0.  0.
 110.  0.  1. -1.  1.  3.  1.  0.  0.  0.  0.  0.
  20.  1.  0.  2.  3. -4.  2.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
```

The resulting tableau cannot have more than 30 rows or 40 columns.

Clear *[i j]*

Set the tableau, or a row of entries, or a column of entries, or a single entry, to zero.

The row index *i* must be in the range [0...m] and the column index *j* must be in the range [0...n]. If neither *i* nor *j* is 0, the (*i*, *j*)'th element is set to 0; if *i* is zero, the entire *j*'th column is set to zero; if *j* is zero, the entire *i*'th row is set to zero. If both *i* and *j* are zero or omitted, the entire tableau is set to zero.

```
< list

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.

< clear 3 5

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  0.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.

< clear 2 0

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
   0.  0.  0.  0.  0.  0.  0.  0.
 110.  0.  1. -1.  0.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.

< clear 0 6

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  0.  5.  0.
   0.  0.  0.  0.  0.  0.  0.  0.
 110.  0.  1. -1.  0.  0.  1.  0.
  20.  1.  0.  2.  3.  0.  2.  0.

< clear
> OK to zero out the entire tableau? yes

     x1  x2  x3  x4  x5  x6  x7
 0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.  0.  0.
```


DElete *i* *j*

Resize the tableau by removing one row or one column.

The row index *i* must be in the range [0...m] and the column index *j* must be in the range [0...n]. Either *i* or *j* must be zero, but not both. If *j* is zero the entire *i*'th row is removed; if *i* is zero the entire *j*'th column is removed.

```
< list

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.

< delete 2 0

       x1  x2  x3  x4  x5  x6  x7
   0.  0.  0. -2.  7.  2.  5.  0.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.

< delete 0 3

       x1  x3  x4  x5  x6  x7
   0.  0. -2.  7.  2.  5.  0.
 110.  0. -1.  1.  3.  1.  0.
  20.  1.  2.  3. -4.  2.  0.
```

Permission is asked before deleting the objective row or the constant column. The result tableau cannot have fewer than 2 rows or 2 columns.

DIgits *[d]*

Report display precision, or reset display precision to *d* significant digits.

If the parameter is omitted, the current display precision is reported. If a new precision *d* is specified it must be in the range [1...16], or *. If * is used the precision is reset to its default value of 8 significant digits; otherwise it is reset to *d* significant digits.

```
< list

           x1         x2  x3  x4         s1         s2  s3
 2290.9091 -6.8181818  0.  0.  60.909091  4.0909091  0.  27.272727
    1.8182  0.3636364  0.  1.   0.818182  0.1818182  0.  -0.454545
    4.5455 -0.5909091  0.  0.  -1.454545 -0.0454545  1.  -0.636364
   14.5455  0.4090909  1.  0.   0.545455 -0.0454545  0.   0.363636

< digits
> Display precision is set to  8 digits.
< digits 6
> Display precision is set to  6 digits.
< list

          x1       x2  x3  x4       s1       s2  s3
 2290.91 -6.81818  0.  0.  60.9091  4.09091  0.  27.2727
    1.82  0.36364  0.  1.   0.8182  0.18182  0.  -0.4545
    4.55 -0.59091  0.  0.  -1.4545 -0.04545  1.  -0.6364
   14.55  0.40909  1.  0.   0.5455 -0.04545  0.   0.3636

< digits 12
> Display precision is set to 12 digits.
< list

               x1             x2  x3  x4             s1             s2  s3
 2290.90909091 -6.81818181818  0.  0.  60.9090909091  4.09090909091  0.  27.2727272727
    1.81818182  0.36363636364  0.  1.   0.8181818182  0.18181818182  0.  -0.4545454545
    4.54545455 -0.59090909091  0.  0.  -1.4545454545 -0.04545454545  1.  -0.6363636364
   14.54545455  0.40909090909  1.  0.   0.5454545455 -0.04545454545  0.   0.3636363636
```

This command sets the *maximum* precision used by `list`. If the current display width (defaulted to the screen width or set using `margin`) is too narrow to fit the tableau at the current precision (defaulted to 8 digits or set using `digits`) fewer digits are used so that the tableau fits in that width without linewraps.

DUal

Replace the current tableau by a tableau corresponding to the dual of the linear program the current tableau represents.

The current tableau must have a basis. First its basic columns are moved to the right and its constraint rows are rearranged, if necessary, to make those columns the $m \times m$ identity matrix. This is the tableau that is saved for restoration by the `undo` command. Then $\mathbf{A}$, $\mathbf{c}^\top$, and $\mathbf{b}$ are extracted from the tableau assuming it represents the primal problem of the standard dual pair. Finally the dual tableau is constructed using $\mathbf{A}^\top$, $\mathbf{b}^\top$, and $\mathbf{c}$. The row dimension `m` of the tableau is changed from $m+1$ to $n+1$, and the column labels are changed to `y1`...`ym`,`w1`...`wn`. Using the command twice to find the dual of the dual returns the starting tableau only if its identity columns were in order on the right.

```
< read brewery.tab
Reading the tableau...
...done.

       x1   x2    x3   x4    x5  x6  x7
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< dual

       y1    y2   y3   w1  w2  w3  w4
   0.  160.  50.  60.  0.  0.  0.  0.
 -90.   -7.  -1.  -2.  1.  0.  0.  0.
-150.  -10.  -3.  -4.  0.  1.  0.  0.
 -60.   -8.  -1.  -1.  0.  0.  1.  0.
 -70.  -12.  -1.  -3.  0.  0.  0.  1.

< dual;
< names x1 x2 x3 x4 s1 s2 s3

       x1   x2    x3   x4    s1  s2  33
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.
```

Here the initial tableau represents standard form for the `brewery` problem, the dual tableau the standard form of its dual, and the final tableau the dual of that dual. The program has no way of knowing that the middle tableau is a dual, so the second invocation of `dual` cannot by itself supply column labels appropriate to the primal.

Every

Toggle the switch that prohibits pivots in the constant column or objective row.

The simplex algorithm never pivots in the constant column or objective row of the tableau, so by default the program prohibits pivots there. If the program is used for other purposes it might make sense to pivot everywhere, so `every` is provided to enable or disable such pivots.

```
< tableau 3 6
< i
T( 1, 1)... = 1 2 -1 1 0 0
T( 2, 1)... = 2 1 0 0 1 0
T( 3, 1)... = -1 1 2 0 0 1

 1.  2. -1.  1.  0.  0.
 2.  1.  0.  0.  1.  0.
-1.  1.  2.  0.  0.  1.

< pivot 1 1
> Cannot pivot in the constant column.
>
< every
> Pivots will be allowed everywhere.
< pivot 1 1

 1.  2. -1.  1.  0.  0.
 0. -3.  2. -2.  1.  0.
 0.  3.  1.  1.  0.  1.

< pivot 2 2

 1.  0.  0.3333333 -0.3333333  0.6666667  0.
 0.  1. -0.6666667  0.6666667 -0.3333333  0.
 0.  0.  3.0000000 -1.0000000  1.0000000  1.

< pivot 3 3

 1.  0.  0. -.22222222 0.55555556 -.11111111
 0.  1.  0. 0.44444444 -.11111111 0.22222222
 0.  0.  1. -.33333333 0.33333333 0.33333333
```

Here `pivot` is used to invert a matrix by appending the identity and pivoting to make the original matrix columns the identity columns (see [20, p280-281]).

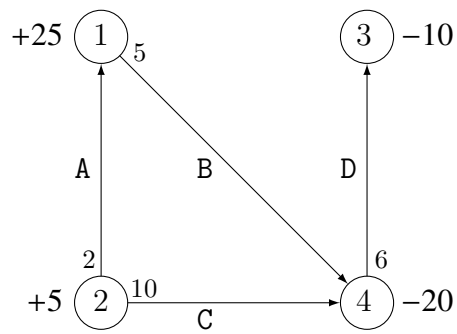
$$\begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & 0 \\ -1 & 1 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} -\frac{2}{9} & \frac{5}{9} & -\frac{1}{9} \\ \frac{4}{9} & -\frac{1}{9} & \frac{2}{9} \\ -\frac{3}{9} & \frac{3}{9} & \frac{3}{9} \end{bmatrix} = \begin{bmatrix} -0.2\bar{2} & 0.5\bar{5} & -0.1\bar{1} \\ 0.4\bar{4} & -0.1\bar{1} & 0.2\bar{2} \\ -0.3\bar{3} & 0.3\bar{3} & 0.3\bar{3} \end{bmatrix}$$

Gnf *links nodes*

Prompt for the data of a general network flow problem and construct the associated simplex tableau.

The session below illustrates the use of `gnf` to construct a simplex tableau for the general network flow problem pictured at the right. In doing this the program follows the sign conventions of §6.0.

```
< gnf 4 4
 link from-node to-node cost
 ---- --------- ------- ----
    A 2         1       2
    B 1         4       5
    C 2         4       10
    D 4         3       6
 node supply-demand
 ---- -------------
    1 25
    2 5
    3 -10
    4 -20

       x21 x14 x24  x43
   0.  2.  5.  10.  6.
 -25.  1. -1.   0.  0.
  -5. -1.  0.  -1.  0.
  10.  0.  0.   0.  1.
  20.  0.  1.   1. -1.

< solve

       x21 x14 x24 x43
-220.  0.  0.  3.  0.
   5.  1.  0.  1.  0.
  30.  0.  1.  1.  0.
  10.  0.  0.  0.  1.
   0.  0.  0.  0.  0.
```



After constructing the simplex tableau I used `solve` to obtain the optimal shipping schedule $x_{21} = 5$, $x_{14} = 30$, $x_{24} = 0$, and $x_{43} = 10$.

At any input prompt, insertion can be interrupted by replying with the end-of-file signal "control-D," which appears on the screen as `^D` but is produced by holding down the CNTL key while pressing the `d` key. This causes the network problem to be abandoned, and the current tableau is left unchanged.

Help *[command]*

ALIAS: ?

If no parameter is given, list the commands; otherwise display the lines in the file `pivot.help` describing the given command.

```
< help
> commands: STOP QUIT HELP ?
> commands: TABLEAU NAMES INSERT LIST APPEND DELETE SWAP
>           EVERY PIVOT CLEAR WRITE READ UNDO MARGIN RATIOS
>           GNF DIGITS SOLVE UNSOLVE SCALE ITERS DUAL
>
< help help
> Help [command]
> if no parameter is given, list the commands
> otherwise copy help file lines describing the given command
> [] denote optional parameters and should not be typed
> capital letters give minimum unambiguous abbreviation
> input lines beginning * are ignored as comments
> alias: ?
>
< help pivot
> Pivot i j
> pivots on tableau element (i,j)
> i must be an integer in the range 1...m
> j must be an integer in the range 1...n
> if a pivot on element (i,j) would cause an overflow, an
>    error message is written and the pivot is not performed
> to pivot in column 1 or row 1, issue the Every command first
> the new tableau is listed after each pivot
>
```

The built-in help is meant to jog the user's memory rather than to take the place of this manual. In addition to telling how to use the `help` command, `help help` provides some general advice about how to interact with the program. To explain a command, `help` just copies lines from the file `pivot.help` (see §27.2).

INsert *[i j]*

Prompt for and read tableau element values.

The row index *i* must be in the range [0...m] and the column index *j* must be in the range [0...n]. If neither *i* nor *j* is zero, the (*i*,*j*)'th element is read; if *i* is zero, the entire *j*'th column is read; if *j* is zero, the entire *i*'th row is read; if both *i* and *j* are zero, or omitted, the entire tableau is read row by row.

```
< t 4 8
< insert
T( 1, 1)... = 0 0 0 -2 7 2 5 0 80 0 0 4 4 1
T( 2, 7)... = -1 1
T( 3, 1)... = 110 0 1 -1 1 3 1 0
T( 4, 1)... = 20 1 0 2 3
T( 4, 6) = ^D
> insertion interrupted

   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3.  0.  0.  0.

< in 0 6
     ( 6)
( 1) 2
( 2) 1
( 3) 3
( 4) -4

   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  0.  0.

< in 4 7
T( 4, 7) = 2

   0.  0.  0. -2.  7.  2.  5.  0.
  80.  0.  0.  4.  4.  1. -1.  1.
 110.  0.  1. -1.  1.  3.  1.  0.
  20.  1.  0.  2.  3. -4.  2.  0.
```

The example shows that insertion can be interrupted by replying to the prompt with the end-of-file signal "control-D," which appears on the screen as ^D but is produced by holding down the CNTL key while pressing the d key.

If a tableau element you enter is not recognized as a number, your computer will signal the error by beeping if it can.

ITers *[kmax [kprint]]*

Report or set options for the `solve` command.

Sometimes (as in studying degenerate problems) it is useful to limit the number of phase-2 pivots performed by the `solve` command in solving each phase-1 subproblem and in finding a final form. Without parameters `iters` reports the current limit `kmax`. If a new value of *`kmax`* is specified, it must be a positive integer. If a tableau has been defined, this routine also reports the theoretical maximum number of iterations required by the simplex algorithm for the given `n` and `m`. If that number is greater than the largest `INTEGER*4`, the largest `INTEGER*4` is printed for comparison; if it is less then the value set for *`kmax`* can be no larger than the theoretical maximum. When the `pivot` program starts it sets `kmax=60`.

Sometimes it is interesting to know the pivot positions chosen by `solve`, though it is seldom desirable to let this output fill the screen. Without parameters this command reports the current limit *`kprint`* on pivot positions to be reported. If a new value of *`kprint`* is specified it must be a nonnegative integer no larger than `kmax`. When the `pivot` program starts it sets `kprint=0` so that no pivot positions are reported.

In the session excerpted below, `solve` attempts the solution of a problem that cycles, so convergence is never achieved. After the initial pivots at `(2,2)`, `(3,3)`, `(4,4)` the sequence `(2,5) (3,6) (2,7) (3,8) (2,2) (3,3)` repeats until the iteration limit is met.

```
< read cycle.tab;
Reading the tableau...
...done.

< iters 300 12
> n!/(n-m)! possible bases:        210
> SOLVE iteration limit:            60
>          now reset to:           210
> SOLVE reporting limit:             0
>          now reset to:            12
>
< solve;
> pivoting at (  2,  2)
> pivoting at (  3,  3)
> pivoting at (  4,  4)
> pivoting at (  2,  5)
> pivoting at (  3,  6)
> pivoting at (  2,  7)
> pivoting at (  3,  8)
> pivoting at (  2,  2)
> pivoting at (  3,  3)
> pivoting at (  2,  5)
> pivoting at (  3,  6)
> pivoting at (  2,  7)
> pivot limit of       210 met
```


List *[i j]*

Print tableau element values on the screen.

The row index *i* must be in the range [0...m] and the column index *j* must be in the range [0...n]. If neither *i* nor *j* is zero, the (*i*,*j*)'th element is printed; if *i* is zero, the entire *j*'th column is printed; if *j* is zero, the entire *i*'th row is printed; if both *i* and *j* are zero, or omitted, the entire tableau is printed.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< list 0 3

 x2
-150.
  10.
   3.
   4.

< list 3 0

      x1  x2  x3  x4  s1  s2  s3
 50.  1.  3.  1.  1.  0.  1.  0.

< list 3 3

  0.300000000000000D+01
```

A single element is printed with full precision. Otherwise the program tries to display only as many digits as necessary, never more than the number set using `digits`, and never so many that the lines of the tableau wrap in the display width set by `margin`.

Sometimes the result of a floating-point calculation is a very small number that is not exactly zero. If a tableau entry is not exactly zero but is less than 10^{-6} times the largest entry in the tableau, it is displayed as `+0` or `-0` to show its sign.

If the requested output cannot be made to fit when displayed in tableau form but the display width is set to `75` or greater, the rows are printed at full `REAL*8` precision, 3 values to a line. If the display width is less than `75`, the elements are printed at full precision in a single column.

Margin *[w]*

Report or set the display width used by `list`.

If `w` is omitted or zero, report the assumed display width. If `w` is greater than zero, reset the assumed display width to `w` characters. If `w` is `*`, reset the assumed display width to the actual screen width.

```
< digits 12
> Display precision is set to 12 digits.
< margin *
> Resetting display width to starting screen size of 114 columns.
< list

               x1             x2  x3  x4             s1             s2  s3
 2290.90909091 -6.81818181818  0.  0.  60.9090909091  4.09090909091  0.  27.2727272727
    1.81818182  0.36363636364  0.  1.   0.8181818182  0.18181818182  0.  -0.4545454545
    4.54545455 -0.59090909091  0.  0.  -1.4545454545 -0.04545454545  1.  -0.6363636364
   14.54545455  0.40909090909  1.  0.   0.5454545455 -0.04545454545  0.   0.3636363636

< margin 75
> Resetting display width to  75 columns.
< list

            x1          x2  x3  x4           s1            s2  s3
 2290.90909 -6.81818182  0.  0.  60.90909091  4.0909090909  0.  27.27272727
    1.81818  0.36363636  0.  1.   0.81818182  0.1818181818  0.  -0.45454545
    4.54545 -0.59090909  0.  0.  -1.45454545 -0.0454545455  1.  -0.63636364
   14.54545  0.40909091  1.  0.   0.54545455 -0.0454545455  0.   0.36363636
```

No tableau with `n` columns can be printed in less than 4`n` characters, so if you set a margin narrower than that `margin` writes a warning. A margin of 4`n` characters is enough only if each entry is in the interval $(\frac{1}{10}, 10)$ so a margin that does not elicit the warning still might not be wide enough to allow printing the tableau with one row on each output line.

Names *[x1 x2 x3 ...]*

Set or unset tableau column labels.

If no parameter is given, this command resets the tableau column labels to blank. If labels are given they are used by `list` in displaying the tableau.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< names porter stout lager ipa

       por  sto   lag  ipa
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< names

   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.
```

Column labels may be 1, 2, or 3 characters wide; if a wider label is given only its first 3 characters are used. If more labels are given than there are variable columns in the tableau, the trailing extra labels are ignored. The program does not provide any way to label the constant column or the rows of the tableau.

Pivot *i j*

Pivot on tableau element (*i*,*j*).

The row index *i* must be in the range `[1...m]` and the column index *j* must be in the range `[1...n]`. If element (*i*, *j*) is zero or small enough that pivoting there would cause an overflow, an error message is written and the pivot is not performed; otherwise the pivot is performed on the whole tableau. The new tableau is listed after each pivot.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< pivot 4 6
> Cannot pivot on a zero element.
>
< pivot 4 3

        x1    x2  x3     x4     s1  s2  s3
 2250. -15.0  0. -22.50  42.50  0.  0.  37.50
   10.   2.0  0.   5.50   4.50  1.  0.  -2.50
    5.  -0.5  0.   0.25  -1.25  0.  1.  -0.75
   15.   0.5  1.   0.25   0.75  0.  0.   0.25

< p 2 4

            x1         x2  x3  x4         s1         s2  s3
 2290.9091 -6.8181818  0.  0.  60.909091  4.0909091  0.  27.272727
    1.8182  0.3636364  0.  1.   0.818182  0.1818182  0.  -0.454545
    4.5455 -0.5909091  0.  0.  -1.454545 -0.0454545  1.  -0.636364
   14.5455  0.4090909  1.  0.   0.545455 -0.0454545  0.   0.363636
```

To pivot in column 1 or row 1, issue the `Every` command first.

Quit

Stop the program, returning control to the operating system.

ALIASES: `STop`, `^D`

```
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
< quit
> STOP
```

In this example the user did no work, but the program quits the same way, without asking for confirmation, even if you have done hours of work and stand to lose some precious result by stopping the program. You can save the current tableau for future use by issuing a `write` command before you quit.

RAtios *i* *j*

Report row or column ratios.

The row index *i* must be in the range [0...m] and the column index *j* must be in the range [0...n]. Either *i* or *j* must be zero, but not both.

If *i* is zero, report the row ratios

$$\frac{T_{k,1}}{T_{k,j}}, \quad k = 1 \ldots m + 1.$$

If *j* is zero, report the column ratios

$$\frac{T_{1,k}}{T_{i,k}}, \quad k = 1 \ldots n + 1.$$

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< ratios 0 5
row ratio
  1 -0.000000E+00
  2  1.333333E+01
  3  5.000000E+01
  4  2.000000E+01

< ratios 3 0
col ratio
  1  0.000000E+00
  2 -9.000000E+01
  3 -5.000000E+01
  4 -6.000000E+01
  5 -7.000000E+01
  6           NaN
  7  0.000000E+00
  8           NaN
```

The command `ratios 0 5` finds the row ratios for column 5 (the x_4 column), which are $\frac{0}{-70}$, $\frac{160}{12}$, $\frac{50}{1}$, and $\frac{60}{3}$. The command `ratios 3 0` finds the column ratios for row 3 (the second constraint row), which are $\frac{0}{50}$, $\frac{-90}{1}$, $\frac{-150}{3}$, $\frac{-60}{1}$, $\frac{-70}{1}$, $\frac{0}{0}$, $\frac{0}{1}$, and $\frac{0}{0}$. The divisions of zero by zero yield the bit pattern for "not a number" as specified in the IEEE standard for floating-point arithmetic (see [100, §4.3] and §28.3.3), which prints as NaN.

REad *filename*

Read a new tableau from a specified file.

This command prompts for the name of a text file, opens the file, and reads the description of a new tableau. The format of the file is illustrated by this example.

```
# brewery problem
4 8
    x1   x2  x3  x4 s1 s2 s3
0  -90 -150 -60 -70  0  0  0
160  7   10   8  12  1  0  0
50   1    3   1   1  0  1  0
60   2    4   1   3  0  0  1
```

The first line of this file is a comment and is ignored by the program. You can use comments wherever you like; the first `#` or `*` on a line, and any text to its right, are ignored. The second line says that the tableau has 4 rows and 8 columns. The third line says that the variable columns (the rightmost 7 columns) have labels `x1`, `x2`, `x3`, `x4`, `s1`, `s2`, and `s3`. The last 4 lines of the file contain the tableau elements.

If you don't want to specify any column labels, leave the second line blank (but don't leave it out). The row and column counts must be integers, but the tableau elements are read in free format so any reasonable way of stating the values is acceptable (`1.5E2` would be as good as `150`). In this example the numbers are neatly spaced so that it is easy to read the tableau when looking in the file with an editor, but extra blanks are ignored in reading the data so the spacing within a line does not matter to the program.

```
< read brewery.tab
> OK to abandon the previous tableau? yes
Reading the tableau...
...done.

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.
```

If the input file does not exist or cannot be read, an error message is written and the previous tableau is restored. If the input file is in a different directory you can give its full path name (but the `read` command, including the file name, cannot be more than 80 characters long). I have adopted the convention of giving tableau files a `.tab` extension, but the program does not care how you name the file.

SCale *i j s*

The row index *i* must be in the range [0...m] and the column index *j* must be in the range [0...n]; the scale factor *s* can be any floating-point value. If neither *i* nor *j* is zero, the (*i*,*j*)'th tableau element is multiplied by the scale factor *s*. If *i* is zero, the entire *j*'th column is scaled; if *j* is zero, the entire *i*'th row is scaled; if *i* and *j* are both zero or omitted, the entire tableau is scaled.

```
< list

     x1  x2  x3  x4  x5  x6  x7
 0. -8.  6.  2.  0. -7.  5.  0.
-1.  0. -3.  0.  8.  6. -4.  3.
-2. -9.  7.  0. -5.  0.  0. -9.
 3. -6.  0.  1. -7.  4. -6.  5.
 4.  9. -5.  0.  0.  3.  9.  4.
 1.  0. -1.  0.  3.  9.  5. -2.

< scale 2 0 -1

     x1  x2  x3  x4  x5  x6  x7
 0. -8.  6.  2.  0. -7.  5.  0.
 1.  0.  3.  0. -8. -6.  4. -3.
-2. -9.  7.  0. -5.  0.  0. -9.
 3. -6.  0.  1. -7.  4. -6.  5.
 4.  9. -5.  0.  0.  3.  9.  4.
 1.  0. -1.  0.  3.  9.  5. -2.

< scale 0 0 3.14159

            x1         x2         x3       x4         x5         x6         x7
  0.000000 -25.132720  18.849540  6.28318   0.000000 -21.991130  15.707950   0.000000
  3.141590   0.000000   9.424770  0.00000 -25.132720 -18.849540  12.566360  -9.424770
 -6.283180 -28.274310  21.991130  0.00000 -15.707950   0.000000   0.000000 -28.274310
  9.424770 -18.849540   0.000000  3.14159 -21.991130  12.566360 -18.849540  15.707950
 12.566360  28.274310 -15.707950  0.00000   0.000000   9.424770  28.274310  12.566360
  3.141590   0.000000  -3.141590  0.00000   9.424770  28.274310  15.707950  -6.283180
```

First the second row of the tableau is multiplied through by -1, then the entire tableau is multiplied by an approximation of π. If the scale factor *s* is zero, the `clear` command is used to zero out the specified tableau elements.

SOlve *[filename]*

Use the simplex algorithm to pivot the current tableau, or the tableau specified in the file *filename*, to a final form. If a tableau file is specified, it must conform to the format described in the manual page for REad.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< solve

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375

< solve brewery.tab

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375
```

The first tableau is the same as T0 in §2.2 and the others are (except for a row permutation) the same as T3c in §2.4.3.

The iters command can be used to change the limit on phase-2 pivots performed by solve from its default value of 60 and to make it display the pivot positions that it uses. If solve reaches its iteration limit without finding a final form (see §2.5) a message is written.

STop

Stop the program.

ALIASES: `Quit`, `^D`

This command stops the program and returns control to the operating system.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< pivot 2 2

            x1  x2         x3         x4         s1         s2  s3
 2057.1429  0. -21.428571  42.857143  84.285714  12.857143  0.  0.
   22.8571  1.   1.428571   1.142857   1.714286   0.142857  0.  0.
   27.1429  0.   1.571429  -0.142857  -0.714286  -0.142857  1.  0.
   14.2857  0.   1.142857  -1.285714  -0.428571  -0.285714  0.  1.

< stop
> STOP
unix[123]
```

If you want to save the current tableau so that you can resume working with it in a subsequent session, use the `write` command before `stop`.

SWap *r1 r2 [c1 c2]*

Exchange tableau row *r1* with row *r2* and/or tableau column *c1* with column *c2*.

If only columns are to be exchanged, make *r1* and *r2* both zero; if only rows are to be exchanged, omit *c1* and *c2* or make them both zero.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< swap 2 3

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< swap 0 0 3 4

       x1   x3   x2    x4    s1  s2  s3
   0. -90. -60. -150.  -70.  0.  0.  0.
  50.   1.   1.    3.    1.  0.  1.  0.
 160.   7.   8.   10.   12.  1.  0.  0.
  60.   2.   1.    4.    3.  0.  0.  1.

< swap 2 4 2 8

       s3  x3   x2    x4    s1  s2  x1
   0.  0. -60. -150.  -70.  0.  0. -90.
  60.  1.   1.    4.    3.  0.  0.   2.
 160.  0.   8.   10.   12.  1.  0.   7.
  50.  0.   1.    3.    1.  0.  1.   1.
```

When columns are swapped their labels are swapped too. Permission is asked before swapping the objective row or the constant column.

Tableau *m n*

Define a new tableau having *m* rows and *n* columns. The number of rows *m* must be in the range [2...30] and the number of columns *n* must be in the range [2...40]. All of the entries in the new tableau are set to zero.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< tableau 3 4
> OK to abandon the previous tableau? yes
< list

 0.  0.  0.  0.
 0.  0.  0.  0.
 0.  0.  0.  0.

< tableau 2 3
< list

 0.  0.  0.
 0.  0.  0.
```

Permission is asked before replacing a previous tableau, unless the previous tableau is all zeros. Because a new tableau is all zeros it is seldom useful to see it, so `tableau` does not list it. The limits of 30 rows and 40 columns are sufficient to define tableaus that are practical to manipulate by hand or likely to be encountered in a course based on §1-§7 of this text. Larger problems should be studied using production linear programming software.

UNDo

Restore the tableau to its most recent previous state.

Before any operation that changes the numerical entries in the current tableau, it is saved as the "previous" tableau, unless it is all zeros. The `undo` command exchanges the current tableau for the previous tableau.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< pivot 3 5

        x1   x2    x3   x4  s1  s2    s3
 3500. -20.   60.  10.  0.  0.   70.  0.
 -440.  -5.  -26.  -4.  0.  1.  -12.  0.
   50.   1.    3.   1.  1.  0.    1.  0.
  -90.  -1.   -5.  -2.  0.  0.   -3.  1.

< undo

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< undo

        x1   x2    x3   x4  s1  s2    s3
 3500. -20.   60.  10.  0.  0.   70.  0.
 -440.  -5.  -26.  -4.  0.  1.  -12.  0.
   50.   1.    3.   1.  1.  0.    1.  0.
  -90.  -1.   -5.  -2.  0.  0.   -3.  1.
```

The `undo` command goes back one step, even if the operation being undone changed the tableau very little (for example, if `insert` was used to change one element). Two consecutive `undo` commands restore the tableau to what it was before the first `undo`, so this command can undo only a single command. `Undo` can exactly reverse the effect of `solve`, while `unsolve` might not.

UNSolve

Restore the tableau to a maximally suboptimal state.

A sequence of minimum-ratio pivots is performed, each in the column having the *most* positive cost entry, until all of the cost entries are *nonpositive.*

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< solve

         x1  x2  x3      x4      s1    s2  s3
 2325.0  0.  0.  18.750  76.250  7.50  0.  18.750
    5.0  1.  0.   2.750   2.250  0.50  0.  -1.250
   12.5  0.  1.  -1.125  -0.375 -0.25  0.   0.875
    7.5  0.  0.   1.625  -0.125  0.25  1.  -1.375

< unsolve

       x1   x2    x3   x4    s1  s2  s3
  -0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  60.   2.    4.   1.    3.  0.  0.  1.
  50.   1.    3.   1.    1.  0.  1.  0.
```

If the starting tableau has some cost coefficients positive, as it will if it is in optimal form, this command finds a tableau from which the simplex method might have started. That tableau is not unique, so `solve` followed by `unsolve` does not necessarily yield the original tableau (as in this example, where the final tableau has its rows permuted from the original).

Write *filename*

Save a description of the current tableau in `filename`.

```
< list

     x1  x2  s1  s2
 0.  1.  1.  0.  0.
 0. -1.  1.  1.  0.
-2. -1. -1.  0.  1.

< solve

     x1  x2  s1   s2
-2.  0.  0.  0.0  1.0
 1.  1.  0. -0.5 -0.5
 1.  0.  1.  0.5 -0.5

< write mulopt.tab
Writing the tableau...
...done.
< read mulopt.tab
> OK to abandon the previous tableau? yes
Reading the tableau...
...done.

     x1  x2  s1   s2
-2.  0.  0.  0.0  1.0
 1.  1.  0. -0.5 -0.5
 1.  0.  1.  0.5 -0.5
```

The file `mulopt.tab`, written and then read in the example, is listed below.

```
  3  5
                                        x1                      x2                      s1                      s2
 -2.0000000000000000D+00  0.0000000000000000D+00  0.0000000000000000D+00  0.0000000000000000D+00  1.0000000000000000D+00
  1.0000000000000000D+00  1.0000000000000000D+00  0.0000000000000000D+00 -5.0000000000000000D-01 -5.0000000000000000D-01
  1.0000000000000000D+00 -0.0000000000000000D+00  1.0000000000000000D+00  5.0000000000000000D-01 -5.0000000000000000D-01
```

The first line says that the tableau has 3 rows and 5 columns. The second line says that the variable columns have labels `x1`, `x2`, `s1`, and `s2`. The last 3 lines of the file contain the tableau elements; because they are written at full precision, these lines and the labels line are 24`n` characters long.

If the tableau has no column labels, `write` makes the second line of the file a blank line. If the output file already exists, `write` asks permission before overwriting it. If the output file is in a different directory you can give its full path name (but the `write` command, including the file name, cannot be more than 80 characters long). I have adopted the convention of giving tableau files a `.tab` extension, but the program does not care how you name the file.

? *[command]*

ALIAS: `help`

If no parameter is given, list the commands; otherwise display the lines in the file `pivot.help` describing the given command.

```
< ?
> commands: STOP QUIT HELP ?
> commands: TABLEAU NAMES INSERT LIST APPEND DELETE SWAP
>           EVERY PIVOT CLEAR WRITE READ UNDO MARGIN RATIOS
>           GNF DIGITS SOLVE UNSOLVE SCALE ITERS DUAL
>
< ? ?
? [command]
if no parameter is given, list the commands
otherwise copy help file lines describing the given command
[] denote optional parameters and should not be typed
capital letters give minimum unambiguous abbreviation
input lines beginning * are ignored as comments
alias: Help
>
< ? pivot
> Pivot i j
> pivots on tableau element (i,j)
> i must be an integer in the range 1...m
> j must be an integer in the range 1...n
> if a pivot on element (i,j) would cause an overflow, an
>    error message is written and the pivot is not performed
> to pivot in column 1 or row 1, issue the Every command first
> the new tableau is listed after each pivot
>
```

The built-in help is meant to jog the user's memory rather than to take the place of this manual. In addition to telling how to use the `?` command, `? ?` provides some general advice about how to interact with the program.

27.2 Installing the pivot Program

It is easy to perform one pivot on a small tableau by hand, but pivoting repeatedly or in a large tableau is tedious and error-prone so it is very helpful to have a computer do the arithmetic. I mentioned in §2.7 having written an implementation of the utility described in the previous Section, and here when I refer to "the `pivot` program" I will mean that actual code rather than the abstraction. This Section tells how you can download the actual code and install it on your computer.

The `pivot` program is designed to be used in a Unix terminal window, so first you will need access a computer that runs some version of the Unix operating system. If your computer runs the Windows operating system you can install the `cygwin` Unix emulator as an application. If your computer is an Apple running the Mac OS-X operating system you can open a terminal window to get the command-line interface required for `pivot`. A third possibility is to use a personal computer on which Linux is installed as the only operating system, or as a virtual machine under Windows, or as an alternative to Windows that you select when you boot the computer. Extensive tutorial information about Unix is available on the web, and excellent introductory textbooks are published inexpensively by O'Reilly (`www.ora.com`), but once the `pivot` program is installed most of its features can be used without knowing anything about Unix.

The `pivot` program is written in Classical FORTRAN [100] and distributed as source code, so on your Unix machine you will need to use a suitable compiler such as `gfortran` to build an executable.

At the time you install `cygwin` or Linux it is possible to specify that `gfortran` be included. To install a FORTRAN compiler on an Apple computer you can open a Unix terminal window, install `Homebrew`, and then enter `brew gcc` to install Xcode, the command line tools, `gcc`, and `gfortran`. These instructions are necessarily somewhat vague because the technical details change from one platform to another and from moment to moment; if you need help consult relevant web pages or an experienced colleague.

27.2.1 Building the Executable

During the five years that this book was in preparation, the `pivot` program went through several versions so that bugs could be fixed and new features added in response to feedback from users (this is evident from the different version numbers appearing in `pivot` sessions throughout the book). The version described here has the attributes listed in the table to the right.

version number	4.4
release date	24 Jul 18
source code `pivot44.f`	
application-specific routines	33
general-purpose routines	29
non-comment lines	2864
comment lines	2964
file size in bytes	173412
executable program `pivot44`	
file size in bytes	275743
virtual memory size in bytes	4120576

A single file pivot44.f containing a concatenation of all the source routines can be downloaded free from the publisher's web site and compiled using this Unix command.

```
gfortran -fno-automatic -fno-range-check -o pivot44 pivot44.f
```

In OS-X and Linux this produces an executable named pivot44; in cygwin the executable is named pivot44.exe instead.

27.2.2 Other Files

You should consider also downloading a few other files from the publisher's web site.

The pivot.help file, if it is present in your home directory, is used by the pivot program's help command to explain the program's other commands.

The file named .bashrc, if it is present in your home directory, is used by Unix to put the current directory in your path to executables and to export window dimensions as shell variables that can be used by the pivot program in formatting its output. Having this file will make your interactions with Unix and the pivot program slightly more graceful. If you have already customized your .bashrc file you can modify it rather than replacing it.

The pivotprint shell script described in §27.3.3 can be used to simplify capturing your conversation with the pivot program for printing or inclusion in a document. To use it you must also install, by compiling from source, the utility program fixscript that it invokes.

27.3 Running the pivot Program

Once you have installed the pivot program on your computer, you can invoke it in a Unix or cygwin terminal window by entering its name at the **Unix command prompt** and then pressing ENTER.

```
unix[1] pivot
> This is PIVOT, Unix version 4.4
> For a list of commands, enter HELP.
>
<
```

In this example unix[1] is the Unix command prompt; the precise appearance of the command prompt might be different on your computer. If your current directory is not in your path to executables, you might need to type ./pivot instead of pivot, to tell Unix that the program is in this directory. When the program starts, it writes the greeting shown above to tell you the version number and to remind you that you can find out about the commands by using help (as described in §27.3.2 below).

27.3.1 Using the Command-Line Interface

The pivot program makes no use of the mouse or of the function keys on your computer; you interact with the program by entering commands and responding to prompts.

The program writes output on your screen in the Unix window. When it is ready for you to enter a command, the prefix character appearing in the first column of the display changes to the **`pivot` command prompt** `<`. Messages that are printed by the program are prefixed by `>`, so when you look at the printout of a session you can tell what you typed and what the program typed. Some outputs of the program, such as the current tableau that is written by `list`, have no prefix character.

If you type a command the program doesn't recognize, it will tell you and prompt for another command.

```
< hello
> Ignored; unknown command.
<
```

You cannot damage the program or your computer by typing a wrong command. You can put extra spaces at the beginning of a command if you like. The total length of a command line, including any leading blanks, can't be more than 80 characters. If you enter a `*` or `#`, it and anything to its right are ignored by the program, so you can type comments to annotate your session. You can insert blank lines by just pressing RETURN at the `pivot` command prompt. Entering an exclamation point `!` repeats the previous command.

```
< * this is a comment
< quit # this is also a comment
> STOP
unix[2]
```

If a command normally prints the resulting tableau, you can suppress that output by appending `;` to the command.

```
< pivot 2 3;
<
```

To stop the program enter `quit` or `stop`, or send the end-of-file signal "control-D," which appears on the screen as `^D` but is produced by holding down the CNTL key while pressing the `d` key. Stopping the program discards any work you did and returns you to the Unix command prompt. If you run the program again it will not remember that you ran it before.

27.3.2 Using the Built-In Help

A command that is often useful to beginning users is `help`. If entered without a parameter, it produces a list of the command names.

```
< help
> commands: STOP QUIT HELP ?
> commands: TABLEAU NAMES INSERT LIST APPEND DELETE SWAP
>           EVERY PIVOT CLEAR WRITE READ UNDO MARGIN RATIOS
>           GNF DIGITS SOLVE UNSOLVE SCALE ITERS DUAL
>
<
```


You can get a brief synopsis of a particular command once you know its name.

```
< help tableau
> Tableau m n
> defines a new tableau with m rows and n columns
> m must be an integer in the range 1...30
> n must be an integer in the range 1...40
> the new tableau is set to all zeros
>
<
```

The response gives a command prototype, tells what the command does, and provides the minimum information you need to use the command. Here the command prototype `Tableau m n` shows by the capitalization of its first letter that the shortest abbreviation you can use for the command is the single letter `t` or `T` (the case of commands does not matter). It also shows that the command requires two numerical parameters `m` and `n`, in that order. The description explains what the parameters mean and what the command does. The `help` command is meant only to jog your memory; for complete information about a command you should consult the appropriate page in §27.1 of this manual (or examine the source code).

27.3.3 Printing the Screen

Students often want to print their interactions with the `pivot` program on paper or save them in a file for inclusion in a document.

One way to capture the dialog is to cut it from the terminal screen after you have run the program and paste the text into a file using an editor such as `vi` or Notepad. To use cut-and-paste in `cygwin` you must be running the X-windows version, so select that version when you start. In a real Unix environment you can use `lpr` to print the file, but in `cygwin` you must use the print function of Notepad.

A more convenient way of capturing the dialog is to use the Unix `script` utility to make a typescript of your terminal session (`man script` will show you all of its options). Typing `script -c pivot` at the Unix command prompt will run the `pivot` program as usual, but when you stop the program you will find that `script` has generated a new file named `typescript` containing a transcript of your session. You can print `typescript` by using `lpr` in Unix or by using Notepad in `cygwin`.

The `script` command includes in the `typescript` file everything that is input or output, including linefeeds and backspaces. Before you can include the file in a document these unprintable characters must be removed. You can clean up the `typescript` file by hand using an editor, or use the `fixscript` program to do it. Typing

```
fixscript < typescript > session
```

at the Unix command prompt will generate a laundered version of `typescript` in `session`. The shell script `pivotprint`, which is listed at the top of the next page, runs pivot under the control of `script` and invokes `fixscript` on the output to produce a `session` file.


```
#! /bin/sh
#  pivotprint: run pivot, capturing the conversation in "session"

rm -f typescript
script -c pivot
rm -f session
fixscript < typescript > session
rm -f typescript
exit 0
```

In the terminal session below, I used `pivotprint` to run the `pivot` program and capture its output (here all I did in `pivot` was issue the `help` command). Then I used the Unix `more` program to copy the contents of the file `session` to the screen.

```
unix[3] pivotprint
Script started, file is typescript
> This is PIVOT, Unix version 4.0
> For a list of commands, enter HELP.
>
< help
> commands: STOP QUIT HELP ?
> commands: TABLEAU NAMES INSERT LIST APPEND DELETE SWAP
>           EVERY PIVOT CLEAR WRITE READ UNDO MARGIN RATIOS
>           GNF DIGITS SOLVE UNSOLVE SCALE ITERS DUAL
>
< quit
> STOP
Script done, file is typescript
unix[4] more session
Script started on Fri 29 May 2015 11:20:50 AM EDT
> This is PIVOT, Unix version 4.0
> For a list of commands, enter HELP.
>
< help
> commands: STOP QUIT HELP ?
> commands: TABLEAU NAMES INSERT LIST APPEND DELETE SWAP
>           EVERY PIVOT CLEAR WRITE READ UNDO MARGIN RATIOS
>           GNF DIGITS SOLVE UNSOLVE SCALE ITERS DUAL
>
< quit
> STOP

Script done on Fri 29 May 2015 11:20:53 AM EDT
```

27.4 Exercises

27.4.1 [E] Can the `pivot` sessions that are shown throughout this book be understood *without* installing and using the `pivot` program as described in §27.2? Explain.

27.4.2 [E] The `pivot` utility described in §27.1 and the implementation described in §27.2 use `m` and `n` for the dimensions of the tableau and `i` and `j` for the indices of elements in the tableau. How are these variables related to m, the number of constraints in the linear program, n, the number of variables, h, the row index of an element in $\mathbf{A}$, and p, the column index of an element in $\mathbf{A}$?

27.4.3 [E] Where in this Chapter can you find a description of the `digits` command? Describe the structure of its manual page.

27.4.4 [E] In the manual of §27.1, some command prototypes show parameters enclosed in square brackets. What does this indicate? In typing such a command at the `pivot` command prompt, should the brackets be included?

27.4.5 [E] Of what use is a command's *minimum unambiguous abbreviation*? Which commands of the `pivot` program have *aliases*?

27.4.6 [E] In using `pivot`, instructions to the program and data about the problem under study are provided by means of command parameters and responses to prompts. Which of the command parameters and prompt responses accepted by the program are (`a`) floating-point numbers; (`b`) integer numbers; (`c`) character strings? (`d`) When supplying a character-string parameter to the program, should you enclose the string in single `'` or double `"` quotes? Explain.

27.4.7 [H] Which commands of the `pivot` program require a tableau already to have been defined?

27.4.8 [P] Why does the command `delete 1 2` elicit an error message from `pivot`? What is the message?

27.4.9 [E] In the pivot command `di 5` what does the `5` mean?

27.4.10 [E] If the command `help list` fails to elicit a description of the `list` command, what might be the reason?

27.4.11 [E] How can you limit the number of iterations that `pivot` performs in solving a linear program? How can you find out what pivot positions the `solve` command chooses?

27.4.12 [E] What is the effect of sending `^D` in response to a prompt for tableau elements from the `insert` command?

27.4.13 [E] If a tableau element is printed as `+0`, what is its value?

27.4.14 [E] Explain the difference between `margin`, `margin *`, and `margin 75`.

27.4.15 [E] What effect does the command `names` have?

27.4.16 [H] Give three possible reasons why the command `pivot 4 6` might not cause a pivot to be performed.

27.4.17 [E] Normally the `pivot` program prevents you from pivoting in the first row or column of the tableau. (`a`) Why does it do that? (`b`) How can you make it *not* do that?

27.4.18 [E] If you are solving a linear program by using `pivot` to perform a sequence of minimum-ratio pivots and you are about to pivot in column 4, how can you use the program to find the row ratios b_i/a_{i4}?

27.4.19 [E] If a row or column ratio is 0/0, what result does the `pivot` program report?

27.4.20 [E] Describe the format that a tableau file must have if it is to be read by the `read` command.

27.4.21 [E] If you issue the command `scale 2 3 4` what effect will it have on the current tableau?

27.4.22 [H] If the file `problem.tab` specifies a starting tableau in the format necessary for `read`, how can you solve the linear program with the `pivot` program by using the smallest number of commands?

27.4.23 [E] Suppose you have been using the `pivot` program to study a linear program, but now you want to go to lunch. How can you save the current tableau and resume your work later?

27.4.24 [P] In the following `pivot` session, what will be the tableau resulting from the `swap` command? (a) Predict what will happen *before* you try it. (b) Use the program to confirm your prediction.

```
< list

       x1   x2    x3   x4    s1  s2  s3
   0. -90. -150. -60.  -70.  0.  0.  0.
 160.   7.   10.   8.   12.  1.  0.  0.
  50.   1.    3.   1.    1.  0.  1.  0.
  60.   2.    4.   1.    3.  0.  0.  1.

< swap 1 2 3 4
```

27.4.25 [H] What are the smallest and largest tableaus that can be stored by the `pivot` program? How can you increase the limits this Classical FORTRAN program [100, §5.5] imposes on the maximum size of a tableau?

27.4.26 [E] If you make a mistake using the `pivot` program, how can you fix it?

27.4.27 [E] What does the `pivot` program do to "unsolve" a linear program?

27.4.28 [E] In naming a tableau file for use with the `pivot` program, what filename extension must you use?

27.4.29 [E] What does the `pivot` command `? ?` do?

27.4.30 [E] Describe the computing environment that is needed to install and use the `pivot` program.

27.4.31 [E] What release of the `pivot` program is described in §27.2? Why do the `pivot` sessions reproduced in this book show that different versions of the program were used?

27.4.32 [E] What Unix command can be used to compile version 4.4 of the `pivot` program? Where can you get the file `pivot44.f`?

27.4.33 [E] Why might you want to place the file `pivot.help` in your home directory?

27.4.34 [E] How does the `pivot` program interact with its user? How can you tell that it is ready for you to enter a command? What alphabetic case must you use when you type a command to the program? How can you repeat the previous command?

27.4.35 [H] Which `pivot` commands print a result tableau? How can you keep that from happening? Why might you want to keep that from happening?

27.4.36 [P] Explain how to capture your `pivot` session in a file. Run the program in such a way that you do that, and print the file that results.

28

Appendices

As I mentioned in §0.2.1, this book assumes that you already have some prior knowledge of undergraduate mathematics, numerical methods, and computer programming. In each of the few places where I worried that I had assumed *too much,* I referred you here for a brief review of some particular topic. Sections 28.1–28.4 are specific to those needs and thus far from exhaustive; if I have guessed wrong again and neglected to explain some idea that is missing from your background, please accept my apology and consult other references including [3], [20], [30], [50], [60], [67], [77], [87], [100], [110], [146], [147], [148], [149], and [150].

Sections 28.5–28.8 catalog the named optimization problems used in the text.

28.1 Calculus

The calculus that I have assumed you know quite well includes the concept of a limit, the definition of a derivative, and how to calculate the derivatives of functions of one or several variables. The topics discussed in this Section are also essential background, about some of which you might like to be reminded.

28.1.1 Extrema of a Function of One Variable

Elementary courses introduce the idea that the local extrema of a differentiable function occur where the slope of its graph is zero. In the graph of this function [3, p265]

$$y = 13x^6 + 14x^5 - 70x^4 - 90x^3 + 250,$$

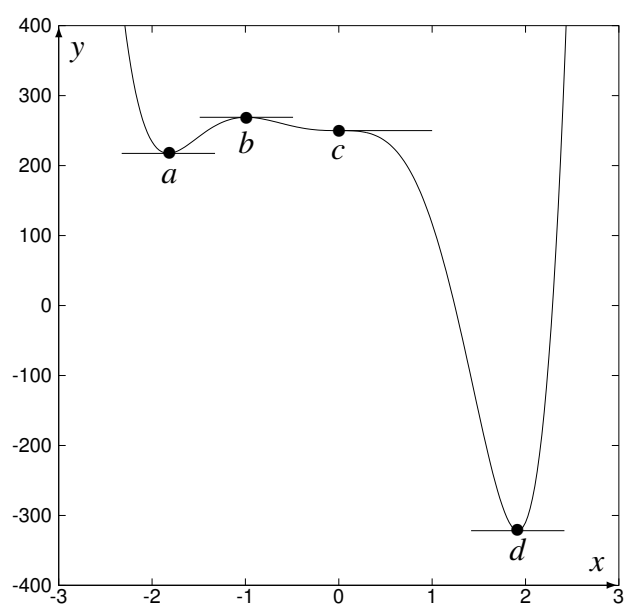
shown on the right, the derivative

$$y' = 78x^5 + 70x^4 - 280x^3 - 270x^2$$

is zero in the indicated places. These points can be classified [146, §4.4] by using the sign of the second derivative

$$y'' = 390x^4 + 280x^3 - 840x^2 - 540x$$

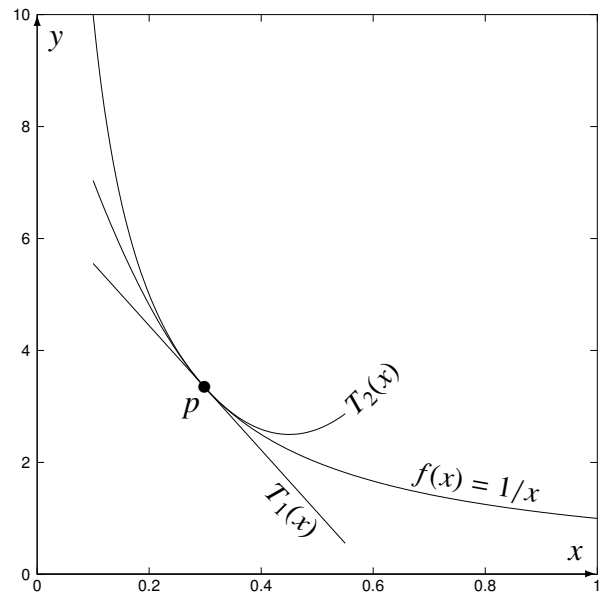
as shown in the table on the next page.



p	x_p	$y''(x_p)$	classification
a	-1.825	$812.62 > 0$	minimum
b	-0.989	$-185.45 < 0$	maximum
c	0	$0 = 0$	inflection
d	1.917	$3117.6 > 0$	minimum

If we think of increasing x to move along the curve from left to right, the slope of the tangent line is initially negative but increases through zero at point a and then to a positive value, so at the first local minimum the derivative is *increasing* and the second derivative is *positive.* Soon the slope decreases, reaching zero at point b and then becoming negative, so at point b the second derivative is *negative.* At point c the slope is changing from increasing to decreasing, so there the second derivative is *zero.*

28.1.2 Taylor's Series for a Function of One Variable



The graph on the right shows the function $f(x) = 1/x$, along with linear and quadratic approximations at the point $p = (a, 1/a)$ with $a = \frac{3}{10}$. The linear function is the straight line tangent to the curve at p,

$$T_1(x; a) = f(a) + f'(a)(x - a)$$

while the quadratic function,

$$T_2(x; a) = f(a) + f'(a)(x - a) + \tfrac{1}{2}f''(a)(x - a)^2$$

matches both the slope and the curvature of $f(x)$ at that point. From the picture it is clear that as we move away from p the error in the linear approximation grows more quickly than the error in the quadratic approximation.

We can make a more precise approximation by including more terms of the **Taylor's series expansion** [149, §10.9] [148, §5.2.2]

$$T_\infty(x; a) = f(a) + f'(a)(x - a) + \cdots = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)(x - a)^k}{k!}$$

where $f^{(k)}$ is the k'th derivative of $f(x)$. In our example $f(x) = 1/x$ so $f^{(k)}(x) = (-1)^k k! x^{-(k+1)}$ and

$$T_\infty(x; a) = \sum_{k=0}^{\infty} (-1)^k a^{-(k+1)}(x - a)^k = \frac{1}{a} + \frac{1}{a}\left(\frac{-(x - a)}{a}\right) + \frac{1}{a}\left(\frac{-(x - a)}{a}\right)\left(\frac{-(x - a)}{a}\right) + \cdots$$

This is a geometric series with first term $1/a$ and ratio $r = -(x - a)/a$, and if $|r| < 1$ or $0 < x < 2a$ it converges to $T_\infty = (1/a)/(1 - r) = 1/x$.

28.1.3 The Gradient of a Quadratic Form

Some properties of quadratic functions are discussed in §14.7; elsewhere we have had occasion to compute the gradient. For example, if

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{x}^\top \mathbf{Q}\mathbf{x} \\ &= [x_1, x_2]\begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1^2 q_{11} + x_1 x_2 q_{21} + x_1 x_2 q_{12} + x_2^2 q_{22} \end{aligned}$$

then we find

$$\begin{aligned} \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \end{bmatrix} &= \begin{bmatrix} 2x_1 q_{11} + x_2 q_{21} + x_2 q_{12} \\ 2x_2 q_{22} + x_1 q_{21} + x_1 q_{12} \end{bmatrix} = \begin{bmatrix} 2q_{11} & q_{12} + q_{21} \\ q_{12} + q_{21} & 2q_{22} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \left(\begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix} + \begin{bmatrix} q_{11} & q_{21} \\ q_{12} & q_{22} \end{bmatrix} \right) \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \end{aligned}$$

and in general $\nabla f(\mathbf{x}) = (\mathbf{Q} + \mathbf{Q}^\top)\mathbf{x}$ where $\mathbf{Q}^\top$ is the transpose (see §28.2.2) of $\mathbf{Q}$.

If $\mathbf{Q}$ is symmetric so that $q_{ij} = q_{ji}$ (as is always the case for the Hessian matrix of a function with continuous second partials) then $\mathbf{Q} = \mathbf{Q}^\top$ and $\nabla f(\mathbf{x}) = 2\mathbf{Q}\mathbf{x}$. If in addition $\mathbf{Q} = \mathbf{I}$, so that $f(\mathbf{x}) = \mathbf{x}^\top\mathbf{x}$, then $\nabla f(\mathbf{x}) = 2\mathbf{x}$.

If the quadratic form is a two-norm (see §10.6.3) then

$$f(\mathbf{x}) = \|\mathbf{x}\| = +\sqrt{\mathbf{x}^\top\mathbf{x}} = (x_1^2 + \cdots + x_n^2)^{\frac{1}{2}}$$

and if $\mathbf{x}^\top\mathbf{x} \neq 0$ we find using the chain rule that

$$\frac{\partial f}{\partial x_j} = \tfrac{1}{2}(x_1^2 + \cdots + x_n^2)^{-\frac{1}{2}}(2x_j) = \frac{x_j}{(x_1^2 + \cdots + x_n^2)^{\frac{1}{2}}}$$

so $\nabla f(\mathbf{x}) = \mathbf{x}/\|\mathbf{x}\|$; the gradient of the two-norm of $\mathbf{x}$ is a unit vector in the direction of $\mathbf{x}$.

28.2 Linear Algebra

The linear algebra that I have assumed you know quite well includes the definition of a matrix as a rectangular array of numbers and of a vector as a matrix having one row or one column, as illustrated below. The topics discussed in this Section are also essential background, about some of which you might like to be reminded.

element $a_{23} = 4$

$$\text{matrix } \mathbf{A} = \underbrace{\begin{bmatrix} -3 & 2 & 1 & 7 \\ 9 & 5 & 4 & -1 \\ 2 & -6 & 8 & 3 \end{bmatrix}}_{n = 4 \text{ columns}} \Bigg\} m = 3 \text{ rows}$$

$$\text{row vector } \mathbf{r}_1 = [-3, 2, 1, 7]$$

$$\text{column vector } \mathbf{c}_3 = \begin{bmatrix} 1 \\ 4 \\ 8 \end{bmatrix}$$

28.2.1 Matrix Arithmetic

Matrices having dimensions that permit a given arithmetic operation to be performed upon them are said to be **conformable** for that operation.

The sum or difference of two matrices **A** and **B** having the same dimensions is the matrix **C** having those dimensions, each of whose elements $c_{ij} = a_{ij} \pm b_{ij}$ is the sum or difference of the corresponding elements in **A** and **B**, as illustrated by these examples.

$$\begin{bmatrix} 4 & 6 & 1 \\ 5 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 5 & 8 & 4 \\ 9 & 7 & 9 \end{bmatrix} \qquad \begin{bmatrix} 4 & 6 & 1 \\ 5 & 2 & 3 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 4 & -2 \\ 1 & -3 & -3 \end{bmatrix}$$

The product $\mathbf{AB} = \mathbf{C}$ of two matrices $\mathbf{A}_{m\times n}$ and $\mathbf{B}_{n\times p}$ is the matrix $\mathbf{C}_{m\times p}$ whose (i, j)'th element is

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

In calculating the matrix product on the left below [3, p492-493] I have shown in the middle matrix the expansion of this sum for each c_{ij}.

$$\begin{bmatrix} \boxed{1 \quad 3 \quad 1 \quad 0} \\ -1 \quad 2 \quad 0 \quad -1 \\ 3 \quad 5 \quad -2 \quad 4 \end{bmatrix} \begin{bmatrix} \boxed{\begin{matrix} 9 \\ 2 \\ 0 \\ 6 \end{matrix}} & \begin{matrix} 5 \\ 3 \\ -5 \\ 1 \end{matrix} \end{bmatrix} = \begin{bmatrix} 1\times9 + 3\times2 + 1\times0 + 0\times6 & 1\times5 + 3\times3 + 1\times(-5) + 0\times1 \\ -1\times9 + 2\times2 + 0\times0 + (-1)\times6 & -1\times5 + 2\times3 + 0\times(-5) + (-1)\times1 \\ 3\times9 + 5\times2 + (-2)\times0 + 4\times6 & 3\times5 + 5\times3 + (-2)\times(-5) + 4\times1 \end{bmatrix} = \begin{bmatrix} \boxed{15} & 9 \\ -11 & 0 \\ 61 & 44 \end{bmatrix}$$

If you find it easier to remember words and pictures than formulas, think of computing c_{ij} by multiplying each element in the i'th row of **A** by the corresponding element in the j'th column of **B** and then adding up the results. The calculation of c_{11} in the example by that method looks like this.

$$\begin{matrix} \boxed{1 \quad 3 \quad 1 \quad 0} \\ \times \quad \times \quad \times \quad \times \\ \boxed{9 \quad 2 \quad 0 \quad 6} \\ \| \quad \| \quad \| \quad \| \\ 9 + 6 + 0 + 0 = \boxed{15} \end{matrix}$$

For the product **AB** to be conformable the number of columns in **A** and the number of rows in **B** must both be n; for the product **BA** also to be conformable the number of columns in **B** and the number of rows in **A** must be equal, so $p = m$. When the products **AB** and **BA** are both defined, they are usually *not* equal; matrix multiplication is *not* commutative.

Often in this book a system of linear algebraic equations is represented in matrix-vector form. For example, the system on the left below can be written as $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ have the values shown on the right.

$$\begin{aligned} 1x_1 + 3x_2 + 1x_3 + 0x_4 &= 15 \\ -1x_1 + 2x_2 + 0x_3 - 1x_4 &= -11 \\ 3x_1 + 5x_2 - 2x_3 + 4x_4 &= 61 \end{aligned} \qquad \mathbf{A} = \begin{bmatrix} 1 & 3 & 1 & 0 \\ -1 & 2 & 0 & -1 \\ 3 & 5 & -2 & 4 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 15 \\ -11 \\ 61 \end{bmatrix}$$

One solution to this system is $x_1 = 9$, $x_2 = 2$, $x_3 = 0$, $x_4 = 6$ because (as we found above)

$$\mathbf{Ax} = \begin{bmatrix} 1 & 3 & 1 & 0 \\ -1 & 2 & 0 & -1 \\ 3 & 5 & -2 & 4 \end{bmatrix} \begin{bmatrix} 9 \\ 2 \\ 0 \\ 6 \end{bmatrix} = \begin{bmatrix} 15 \\ -11 \\ 61 \end{bmatrix} = \mathbf{b}.$$

28.2.2 The Transpose of a Matrix

The **transpose** of a matrix $\mathbf{A}_{m\times n}$ having elements a_{ij} is the matrix $\mathbf{A}^{\mathsf{T}}_{n\times m}$ having elements a_{ji}. Thus the rows of $\mathbf{A}^{\mathsf{T}}$ are the columns of $\mathbf{A}$ and the rows of $\mathbf{A}$ are the columns of $\mathbf{A}^{\mathsf{T}}$. For example,

$$\begin{bmatrix} 1 & 3 & 1 & 0 \\ -1 & 2 & 0 & -1 \\ 3 & 5 & -2 & 4 \end{bmatrix} \qquad \text{has the transpose} \qquad \begin{bmatrix} 1 & -1 & 3 \\ 3 & 2 & 5 \\ 1 & 0 & -2 \\ 0 & -1 & 4 \end{bmatrix}.$$

If a matrix is square then transposing it reflects its elements about the **diagonal** running from the upper left corner to the lower right corner.

$$\begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} 1 & 3 & 7 \\ 2 & 4 & 8 \\ 5 & 6 & 9 \end{bmatrix} \quad \text{diagonal elements}$$

A square matrix that is equal to its transpose is said to be **symmetric**. The matrices below are symmetric, so each is its own transpose. The symmetric matrix on the right, the 3×3 **identity matrix**, is also a **diagonal matrix**.

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 4 & 5 & 6 \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 4 & 5 & 6 \end{bmatrix} \qquad \mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\mathsf{T}}$$

The transpose of a row vector is the column vector having the same elements, and the transpose of a column vector is the row vector having the same elements;

$$\text{if} \quad \mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{then} \quad \mathbf{x}^{\mathsf{T}} = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad \text{and } (\mathbf{x}^{\mathsf{T}})^{\mathsf{T}} = \mathbf{x}.$$

28.2.3 Inner and Outer Products

Two special cases of matrix multiplication are of special interest.

If $\mathbf{a}$ and $\mathbf{b}$ are column vectors both of length n then $\mathbf{a}^\top$ is a row vector, the product $\mathbf{a}^\top\mathbf{b}$ is conformable, and

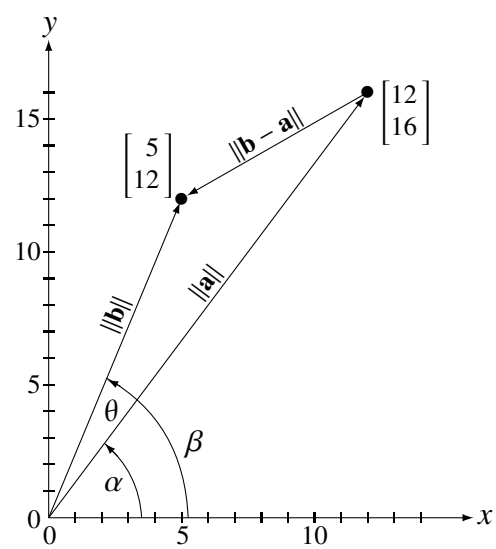
$$\mathbf{a}^\top\mathbf{b} = a_1b_1 + \cdots + a_nb_n$$

is a scalar called the **inner product** or **dot product** of the two vectors. Here is an $n = 2$ example.

$$\mathbf{a} = \begin{bmatrix} 12 \\ 16 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 5 \\ 12 \end{bmatrix} \qquad \mathbf{a}^\top\mathbf{b} = \begin{bmatrix} 12 & 16 \end{bmatrix}\begin{bmatrix} 5 \\ 12 \end{bmatrix} = 12 \times 5 + 16 \times 12 = 252$$

The dot product can also be calculated from the lengths of the vectors and the angle between them. The graph to the right [147, p98-99] shows $\mathbf{a}$ and $\mathbf{b}$ as arrows making angles α and β with the x axis and separated by the angle $\theta = \beta - \alpha$. Using the law of cosines we find [146, Theorem 11.14] for the triangle in the figure

$$\begin{aligned} \|\mathbf{b}-\mathbf{a}\|^2 &= \|\mathbf{b}\|^2 + \|\mathbf{a}\|^2 - 2\,\|\mathbf{b}\|\,\|\mathbf{a}\|\,\cos(\theta) \\ (\mathbf{b}-\mathbf{a})^\top(\mathbf{b}-\mathbf{a}) &= \mathbf{b}^\top\mathbf{b} + \mathbf{a}^\top\mathbf{a} - 2\,\|\mathbf{b}\|\,\|\mathbf{a}\|\,\cos(\theta) \\ \mathbf{b}^\top\mathbf{b} - 2\mathbf{a}^\top\mathbf{b} + \mathbf{a}^\top\mathbf{a} &= \mathbf{b}^\top\mathbf{b} + \mathbf{a}^\top\mathbf{a} - 2\,\|\mathbf{b}\|\,\|\mathbf{a}\|\,\cos(\theta) \\ \mathbf{a}^\top\mathbf{b} &= \|\mathbf{a}\|\,\|\mathbf{b}\|\,\cos(\theta). \end{aligned}$$



The vectors in the example given above have lengths $\|\mathbf{b}\| = \sqrt{5^2 + 12^2} = 13$ and $\|\mathbf{a}\| = \sqrt{12^2 + 16^2} = 20$, and the angle between them is

$$\theta = \beta - \alpha = \arccos\left(\frac{5}{13}\right) - \arccos\left(\frac{12}{20}\right) = 0.24871 \text{ rad.}$$

Then $\|\mathbf{a}\|\,\|\mathbf{b}\|\,\cos(\theta) = 20 \times 13 \times 0.96923 = 252$.

The **outer product** of the vectors in the example is an $n \times n$ matrix.

$$\mathbf{a}\mathbf{b}^\top = \begin{bmatrix} 12 \\ 16 \end{bmatrix}\begin{bmatrix} 5 & 12 \end{bmatrix} = \begin{bmatrix} 12 \times 5 & 12 \times 12 \\ 16 \times 5 & 16 \times 12 \end{bmatrix} = \begin{bmatrix} 60 & 144 \\ 80 & 192 \end{bmatrix}$$

An outer product matrix always has a rank of one [147, p70]; the first row of this result is 12 times $\mathbf{b}^\top$ and the second row is 16 times $\mathbf{b}^\top$, so the second row is $\frac{16}{12} = \frac{4}{3}$ times the first and the rows are not independent (see §28.2.4). The outer product of a vector with itself is a symmetric rank-one matrix.

$$\mathbf{a}\mathbf{a}^\top = \begin{bmatrix} 12 \\ 16 \end{bmatrix}\begin{bmatrix} 12 & 16 \end{bmatrix} = \begin{bmatrix} 144 & 192 \\ 192 & 256 \end{bmatrix}.$$

28.2.4 Linear Independence

The vectors $\mathbf{v}_j \in \mathbb{R}^n$ in a set $\mathbb{V} = \{\mathbf{v}_1 \dots \mathbf{v}_p\}$ are **linearly independent** if and only if

$$\text{the sum} \quad c_1\mathbf{v}_1 + \cdots + c_p\mathbf{v}_p \quad \text{is nonzero unless} \quad c_1 = \cdots = c_p = 0.$$

In other words [147, §2.3] if every nontrivial linear combination of the $\mathbf{v}_j$ is nonzero then the vectors are linearly independent. If any of the vectors $\mathbf{v}_j$ is zero then the set *cannot* be linearly independent. If, say, $\mathbf{v}_1 = \mathbf{0}$ then we could choose $c_1 = 1$ and set all the other $c_j = 0$ so that $c_1\mathbf{v}_1 + \cdots + c_p\mathbf{v}_p = \mathbf{0}$ even though not all of the coefficients are zero. If $p > n$ so that there are more vectors than there are coordinate directions, then at least one vector must be a linear combination of the others and the set also cannot be linearly independent. The **rank** of a matrix is the number of its rows that are linearly independent.

28.2.5 Matrix Inversion

If $\mathbf{A}$ is a square matrix and there exists a square matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, then $\mathbf{A}$ is said to be **nonsingular** and $\mathbf{A}^{-1}$ is called its **inverse matrix**. Matrix algebra often involves the symbolic manipulation of inverses (see §28.2.6). Although it is never necessary to evaluate a matrix inverse numerically [100, Exercise 6.8.6] [87, §1.14], it is sometimes convenient to do so analytically by using this definition [147, p163].

$$\mathbf{A}^{-1} = \frac{\operatorname{adj}(\mathbf{A})}{\det(\mathbf{A})}$$

Here $\det(\mathbf{A})$ is the determinant (see §11.4.1) and $\operatorname{adj}(\mathbf{A})$ is the **adjoint matrix**. The adjoint matrix can be found from the cofactors of $\mathbf{A}$, which are signed minors. To see how, consider the problem of finding $\mathbf{A}^{-1}$ when $\mathbf{A}$ is this nonsingular matrix [20, p278].

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & 0 \\ -1 & 1 & 2 \end{bmatrix}$$

If we construct a submatrix by deleting row r and column s from $\mathbf{A}$, the determinant of that submatrix is a **minor** that we will call δ_{rs} and the corresponding **cofactor** is $c_{ij} = (-1)^{r+s}\delta_{rs}$. For example, if $r = 2$ and $s = 3$ we have

$$\delta_{23} = \begin{vmatrix} 1 & 2 \\ -1 & 1 \end{vmatrix} = 1 \times 1 - (-1) \times 2 = 3 \quad \text{and} \quad c_{23} = (-1)^{2+3}\delta_{23} = (-1) \times 3 = -3.$$

Repeating the calculation for the other 8 pairs (r, s) yields this **cofactor matrix**.

$$\mathbf{C} = \begin{bmatrix} 2 & -4 & 3 \\ -5 & 1 & -3 \\ 1 & -2 & -3 \end{bmatrix}$$

The adjoint matrix is the transpose of the cofactor matrix, and dividing it by $\det(\mathbf{A}) = -9$ yields the inverse.

$$\operatorname{adj}(\mathbf{A}) = \mathbf{C}^\top = \begin{bmatrix} 2 & -5 & 1 \\ -4 & 1 & -2 \\ 3 & -3 & -3 \end{bmatrix} \qquad \mathbf{A}^{-1} = \tfrac{1}{-9}\begin{bmatrix} 2 & -5 & 1 \\ -4 & 1 & -2 \\ 3 & -3 & -3 \end{bmatrix} = \begin{bmatrix} -\frac{2}{9} & \frac{5}{9} & -\frac{1}{9} \\ \frac{4}{9} & -\frac{1}{9} & \frac{2}{9} \\ -\frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

To verify that this is the inverse we can show that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$.

$$\begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & 0 \\ -1 & 1 & 2 \end{bmatrix}\begin{bmatrix} -\frac{2}{9} & \frac{5}{9} & -\frac{1}{9} \\ \frac{4}{9} & -\frac{1}{9} & \frac{2}{9} \\ -\frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} = \begin{bmatrix} -\frac{2}{9} & \frac{5}{9} & -\frac{1}{9} \\ \frac{4}{9} & -\frac{1}{9} & \frac{2}{9} \\ -\frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}\begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & 0 \\ -1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finding the adjoint analytically for an arbitrary 2×2 matrix yields a convenient formula for the inverse [147, p163].

$$\text{if} \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad \text{is nonsingular then} \quad \mathbf{B}^{-1} = \frac{\begin{bmatrix} b_{22} & -b_{12} \\ -b_{21} & b_{11} \end{bmatrix}}{b_{11}b_{22} - b_{21}b_{12}}.$$

28.2.6 Matrix Identities

In performing algebraic manipulations involving matrices and vectors it is essential that the variables be conformable for the operations indicated; systematically check that each expression you write describes a calculation that can actually be performed, and remember that $\mathbf{AB}$ is almost never equal to $\mathbf{BA}$. Often it is convenient to make use of the following identities, each of which assumes that the indicated operations are possible.

$$\begin{aligned}
\mathbf{A} + \mathbf{B} &= \mathbf{B} + \mathbf{A} \\
\mathbf{C} + (\mathbf{A} + \mathbf{B}) &= (\mathbf{C} + \mathbf{A}) + \mathbf{B} \\
\mathbf{C}(\mathbf{A} + \mathbf{B}) &= \mathbf{CA} + \mathbf{CB} \\
(\mathbf{A} + \mathbf{B})\mathbf{C} &= \mathbf{AC} + \mathbf{BC} \\
\mathbf{A}(\mathbf{BC}) &= (\mathbf{AB})\mathbf{C} \\
(\mathbf{A}^\top)^\top &= \mathbf{A} \\
(\mathbf{A} + \mathbf{B})^\top &= \mathbf{A}^\top + \mathbf{B}^\top \\
(\mathbf{AB})^\top &= \mathbf{B}^\top\mathbf{A}^\top \\
\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} &= \mathbf{I} \\
(\mathbf{A}^\top)^{-1} = (\mathbf{A}^{-1})^\top &= \mathbf{A}^{-\top} \\
(\mathbf{AB})^{-1} &= \mathbf{B}^{-1}\mathbf{A}^{-1} \\
(\mathbf{AB})^{-\top} &= \mathbf{A}^{-\top}\mathbf{B}^{-\top}
\end{aligned}$$

28.3 Numerical Computing

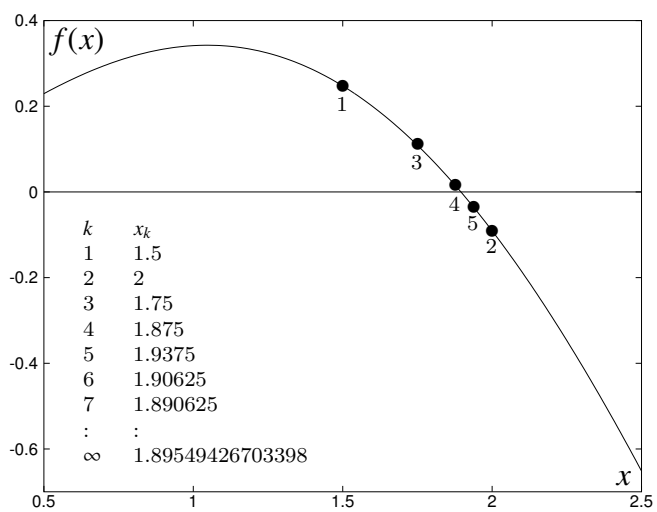
The numerical computing that I have assumed you know quite well includes these ideas:

- many mathematical problems of practical importance have no closed-form analytic solution;
- sometimes the answer to such a problem can be approximated with increasing accuracy by an **algorithm** that iteratively repeats a sequence of arithmetic and logical operations;
- when a numerical algorithm is implemented in a computer program the iterative repetition of its calculations is accomplished by using a **loop** to transfer control from the end of the process back to the first step;
- computers represent real values by **floating-point** numbers that have limited range and precision;
- for a given problem one algorithm might run faster than another or produce more accurate results.

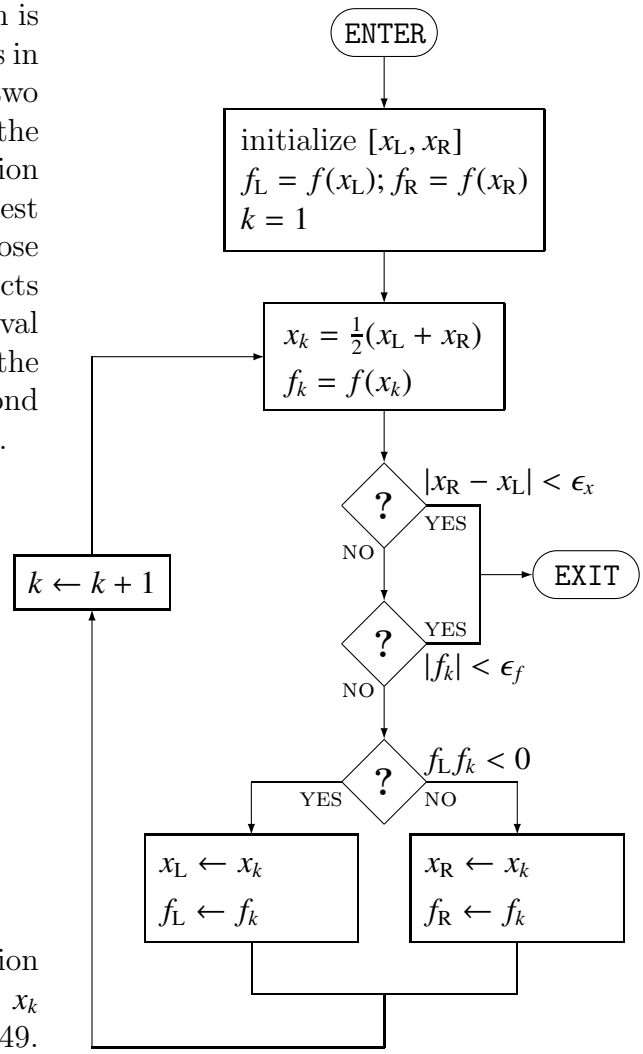
The topics discussed in this Section are also essential background, about some of which you might like to be reminded.

28.3.1 Finding a Root with Bisection

The positive value of x for which $\sin(x) = \frac{1}{2}x$ is not given by any algebraic formula, but it can be approximated numerically [100, §0.1]. In the graph of $f(x) = \sin(x) - \frac{1}{2}x$ below, $f(\frac{1}{2}) > 0$ and $f(2\frac{1}{2}) < 0$ so $f(x)$ crosses zero in the interval $[x_L, x_R] = [\frac{1}{2}, 2\frac{1}{2}]$. At the interval midpoint $x_1 = \frac{1}{2}(x_L + x_R)$ the function value is positive, so the zero must fall between x_1 and x_R. Letting $x_L \leftarrow x_1$ yields a new interval, half as wide as the old one, still containing the root. Repeating the steps of finding the midpoint x_k, finding the sign of the function there, and replacing the appropriate endpoint by the midpoint leads to the sequence of x_k listed inside the graph; the first 5 iterates are numbered on the curve. The algorithm converges to the given x_∞, at which point $f(x) \approx 7.8 \times 10^{-16}$ so that $\sin(x)$ is very close to $\frac{1}{2}x$.



A more precise definition of the algorithm is given in this **flowchart**. An **iteration** begins in the second box with finding x_k and f_k. Then two **convergence tests** stop the calculations if the interval becomes shorter than ϵ_x or the function value gets within ϵ_f of zero. The bottom test uses the product $f_L \times f_k$ to determine if those function values have the same sign, and directs the flow of control to update the correct interval endpoint. The arrow from the bottom of the flowchart through incrementing k to the second box is the loop that repeats the calculations.




```
format long; epsx=1e-16; epsf=1e-16;
xl=0.5; xr=2.5;
fl=sin(xl)-0.5*xl; fr=sin(xr)-0.5*xr;
for k=1:100
    xk=0.5*(xl+xr)
    fk=sin(xk)-0.5*xk;
    if(abs(xr-xl) < epsx) break; end
    if(abs(fk) < epsf) break; end
    if(fl*fk < 0)
       xr=xk;
       fr=fk;
    else
       xl=xk;
       fl=fk;
    end
end
```

The MATLAB program is a verbatim translation of the flowchart into code. It produces the x_k that I listed above, reaching x_∞ at iteration 49.

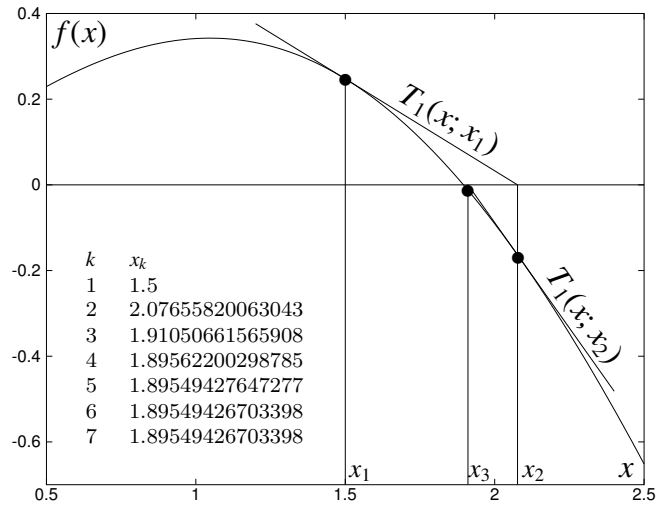
28.3.2 Finding a Root with Newton's Method

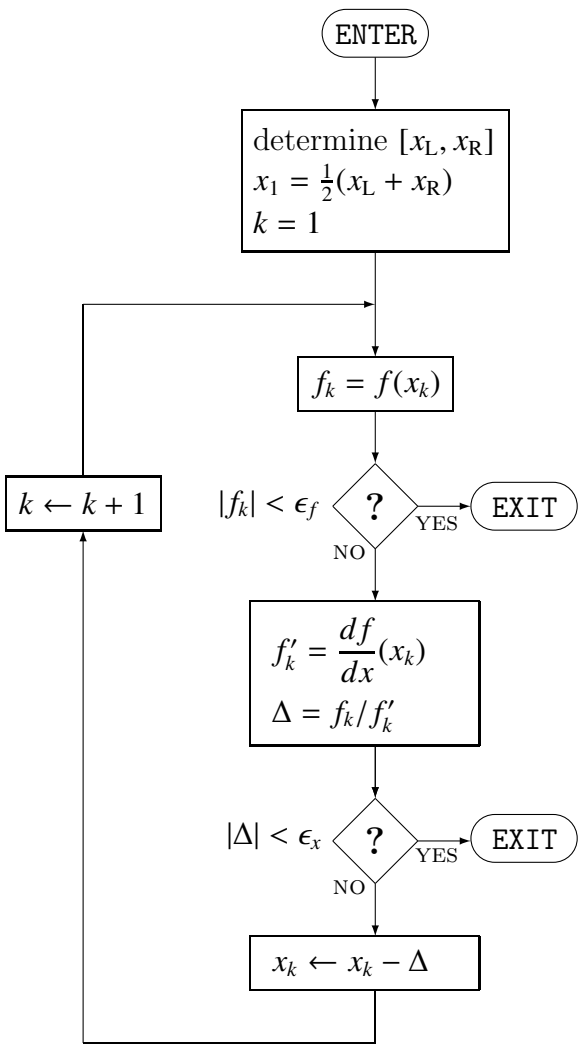
Above we found, by inspection of the graph, a starting interval $[\frac{1}{2}, 2\frac{1}{2}]$ for bisection and began that algorithm at the midpoint $x_1 = 1\frac{1}{2}$. A first-order Taylor series approximation (see §28.1.2) at x_1 predicts that $f(x) = \sin(x) - \frac{1}{2}x$ will cross zero where

$$\begin{aligned} T_1(x; x_1) = f(x_1) + f'(x_1)(x - x_1) &= 0 \\ \text{or} \quad x &= x_1 - \frac{f(x_1)}{f'(x_1)} \quad \text{provided } f'(x_1) \neq 0. \end{aligned}$$

Thus, in the graph on the next page $x_2 = x_1 - \dfrac{\sin(x_1) - \frac{1}{2}x_1}{\cos(x_1) - \frac{1}{2}} \approx 2.0766$

We can get a better approximation to $f(x)$ near the root by updating the Taylor's series to $T_1(x; x_2)$. When we use that tangent line to find x_3, the point is so close to the zero-crossing of the graph that the error in the approximation to the root is barely discernible. Repeating the process yields the root estimates x_k in the table, which converge after only six iterations to the same x_∞ that we found using bisection.





The algorithm we have been using is called **Newton's method** [4, §2.7], which is described more precisely by the flowchart on the left and implemented in this MATLAB code.

```
format long; epsx=1e-16; epsf=1e-16;
xl=0.5; xr=2.5;
xk=0.5*(xl+xr)
for k=1:10
    fk=sin(xk)-0.5*xk;
    if(abs(fk) < epsf) break; end
    fkp=cos(xk)-0.5;
    delta=fk/fkp;
    if(abs(delta) < epsx) break; end
    xk=xk-delta
end
```

I found x_1 as the midpoint of starting bounds, but this algorithm does not update the bounds and any x_1 that is close enough to x_∞ can be used. Picking an x_1 that is *not* close enough to x_∞, such as $x_1 = \frac{1}{2}$, makes the algorithm converge to the wrong root. Thus, although Newton's method is much faster than bisection when it works, it fails more often. To achieve second-order convergence (see §9.2) Newton's method uses both the function value $f(x_k)$ and its derivative $f'(x_k)$, so while it usually takes far fewer iterations than bisection each iteration usually takes more work.

28.3.3 Floating Point Arithmetic

When we write a MATLAB program all of the values it computes (bizarrely including those that should never have fractional parts, such as array indices and loop counters) are processed as `REAL*8` floating-point values [50]. The mathematical set of reals $\mathbb{R}^1$ has an infinity of members including every whole number, every fraction, and every irrational including all of the transcendentals. In dramatic contrast, the floating point numbers [100, §4.2] are a finite set of rational fractions. By their construction they have limited range and represent most real values only approximately, so that very small ones **underflow** to zero and common ones like 0.1 cannot be represented exactly. Because of these properties of floating-point numbers, almost all of the calculations we perform with them are at least slightly wrong; indeed, the discipline of numerical analysis was in its early days devoted almost entirely to figuring out just *how* wrong floating-point calculations are likely to be.

Whole books have been written [84] [125] about the floating-point number system, but of the myriad technical details they discuss only two are of immediate concern in this book.

Models of roundoff error (see §25.6.4 for one example) often make use of a quantity called the **unit roundoff**, which is $u = 2^{-53} = 1.110223024625157 \times 10^{-16}$. The unit roundoff is the largest number which, when added to 1, is sure to produce a result that still rounds to 1 (depending on the rounding rule that is in effect it might be possible to add a slightly larger number and still have the sum round to 1). Some authors [100, p436-437] [125, p14 note 7] call the unit roundoff machine epsilon, while MATLAB and other authors [5, p614] call *twice* the unit roundoff machine epsilon. In this book I have adopted the MATLAB convention that **machine epsilon** is $2u$.

In floating-point implementations that conform to the IEEE standard [84], the result of an impossible calculation such as $\arcsin(2)$ is assigned a special bit pattern called **not a number** [100, §4.7] [125]. This bit pattern does not represent a numerical value but is reported by MATLAB as `NaN` to alert the user that an error has occurred. Graceful programs issue meaningful diagnostics and resign, rather than attempting a meaningless calculation.

28.4 MATLAB Programming Conventions

I have assumed that when you began reading this book you already knew at least a little about computer programming in some procedural language, and that you had at least observed others using base MATLAB (exclusive of optional components such as the optimization toolbox). Numerical computation, mathematical analysis, and the organization of ideas in prose are all important in solving optimization problems, so throughout I have tried to encourage the development of your coding skills along with your knowledge of theory and your eloquence in exposition. My goal of instilling technical fluency has been achieved if after reading the book you find it natural to move between words, formulas, and code.

To make the example programs easy to understand and learn from, I adopted the coding conventions described below.

28.4.1 Control Structures

MATLAB provides terse constructs that maximize the efficiency of its vector and matrix calculations, but I have instead used verbose constructs that maximize the obviousness and simplicity of the code. In particular I have used `for` in preference to `while` so that the loop iteration control mechanism is explicit [100, 13.5.4] and `break` or `continue` so that transfers of control are as explicit as they can be (MATLAB has no command to branch).

```
for j=1:n
    :
    :
    if(d(j) == 0) continue; end
    :
    :
    :
    if(norm(g) <= epz) break; end
    :
    :
end
```

```
for j=1:n
    :
    if(d(j) == 0)
       continue
    end
    :
    if(norm(g) <= epz)
       break
    end
    :
end
```

Here `continue` means skip the rest of the loop body and advance to the next iteration, while `break` means exit the loop through its `end` statement. The tests on the right above are equivalent to those on the left, but to save space I usually used the short form except when there was more than one alternative as on the left below. Sometimes I used `switch`.

```
% from em.m
if(i == 0)
   f=fcn(x,0)+pn*t'*ones(m,1);
elseif(i == 1)
   f=-t(i);
else
   f=fcn(x,(i-m))-t(i-m);
end
```

```
% from sqp1.m
function f=sqp1(x,i)
  switch(i)  %  prepare to distinguish cases
    case 0   %  do this if i=0
      f=exp(x(1)-1)+exp(x(2)+1);
    case 1   %  do this if i=1
      f=x(1)^2+x(2)^2-1;
    end
end
```

These excerpts involve functions named `fcn` and `sqp1`. Many of the algorithm implementations discussed in the text find natural expression in terms of subprograms, and where possible I used them to clarify the code.

28.4.2 Variable Names

It is good style to use descriptive names for variables and functions [100, §12.4.2] but this is tricky in MATLAB because many of the names that might occur to you already have default meanings, and changing those can have unexpected consequences. Before choosing a name for a variable or function, you can see if it already means something to MATLAB by using the help command. Here it shows that `gama` is a safe name for a function of your own.

```
octave:1> help gama
error: help: 'gama' not found
octave:1> help gamma
'gamma' is a built-in function
:
```


The table below lists a few examples, mostly selected from the index of the Octave manual [50, p781-793], of names that might seem perfect for describing the variables and functions in your program but which have already been preempted by MATLAB.

name	default meaning in MATLAB
`arg`	return the angle of a complex number
`beta`	return a value of the β function
`columns`	return the number of columns in a matrix
`diff`	return a vector of first differences
`eps`	return machine epsilon
`flag`	create a colormap
`gamma`	return a value of the Γ function
`hess`	return the Hessenberg decomposition of a matrix
`i`,`I`	return $\sqrt{-1}$ for mathematicians
`j`,`J`	return $\sqrt{-1}$ for electrical engineers
`kappa`	return Cohen's kappa coefficient
`length`	return the greater number of rows or columns in a matrix
`mean`	return the algebraic average of data elements
`nnz`	return the number of nonzero elements in a matrix
`orth`	return an orthonormal basis
`prod`	return the product of array elements along a dimension
`quad`	return the value of a definite integral
`rows`	return the number of rows in a matrix
`sum`	return a sum of matrix elements
`type`	display the definition of each name referring to a function
`union`	return the union of two sets
`var`	return the variance of a data set
`which`	display the type of an object
`xlim`	set the limits of the x-axis for a plot
`ylim`	set the limits of the y-axis for a plot
`zeta`	return a value of the Riemann zeta function

A name that MATLAB has already given a default meaning can be repurposed; in the programs that appear in this book I have always used `i` and `j` for array indices and loop counters rather than for $\sqrt{-1}$, and I have occasionally used several other names to mean something different from their preassigned meanings. If a function of your own has a name that MATLAB has already used, you must set the program's `--path` option to the directory containing your definition and refrain from also using the built-in function in your program (either explicitly, or implicitly by inadvertently invoking another MATLAB routine that uses it). It can of course be confusing to have two functions with the same name, even if you are sure that MATLAB is finding the one that you wrote.

Some variable names that do *not* have a preassigned meaning in MATLAB I have usually used to refer to particular things, and those appearing most frequently are listed below.

name	usual meaning in this text
`alpha`	a step length α (typically in a line search)
`astar`	an optimal step length $\alpha^\star$
`d`	a direction vector $\mathbf{d}$
`err`	the amount by which an iterate is in error
`f`	a function value
`fcn`	a pointer to a function routine
`fr`	a record objective value
`g`	a gradient vector $\mathbf{g}$
`grd`	a pointer to a gradient routine
`H`	a Hessian matrix $\mathbf{H}$
`hsn`	a pointer to a Hessian routine
`i`	an index on functions or on matrix rows
`ip`	the row index of a pivot
`j`	an index on variables or matrix columns
`jp`	the column index of a pivot
`k`	an index on the iterations of an algorithm
`kmax`	an iteration limit
`kp`	`k+1` for MATLAB, which does not permit 0 subscripts
`m`	number of constraints
`nm`	number of Hessian modifications performed
`n`	number of variables
`p, s, t`	indices
`rc`	a subprogram return code
`S`	an LP basis vector
`T`	an LP tableau
`tol`	a convergence tolerance
`x`	a vector $\mathbf{x}$ of decision variables
`xbar, xhat`	particular values $\bar{\mathbf{x}}$, $\hat{\mathbf{x}}$ of $\mathbf{x}$
`xh`	a vector of upper bounds $\mathbf{x}^{\mathrm{H}}$
`xk`	an iterate $\mathbf{x}^k$
`xl`	a vector of lower bounds $\mathbf{x}^{\mathrm{L}}$
`xr`	a record point $\mathbf{x}^r$
`xstar`	an optimal vector $\mathbf{x}^\star$
`xzero`	a starting point $\mathbf{x}^0$
`z`	an objective value being minimized
`Z`	a nullspace basis matrix $\mathbf{Z}$
prob`.m`	routine returns function values for problem *prob*
prob`g.m`	routine returns gradient vectors for problem *prob*
prob`h.m`	routine returns Hessian matrices for problem *prob*

28.4.3 Iteration Counting

The algorithms discussed in this book vary in detail, but they all have the same basic structure: starting from $\mathbf{x}^0$ repeat some iterative calculation some maximum number of times or until a convergence criterion is satisfied first. It is easy to implement this scheme in MATLAB using a `for` loop and `if-then-else`. The code on the left illustrates one natural approach.

```
function [xstar,k]=countk(xzero,kmax)
  x=xzero;
  for k=0:kmax
      if(close enough)
         break
      else
         x=update(x);
      end
  end
  xstar=x;
end
```

In the `countk.m` routine, `k` is an index on the iterates $\mathbf{x}^0$, $\mathbf{x}^1$, ... and `kmax` is the index of the final iterate that will be generated if convergence is not attained. There is always one more iterate (namely $\mathbf{x}^0$) than there are iterations, so `kmax` is also the number of iterations (updates to `x`) that will be performed if convergence is not attained. Whether the routine returns because the convergence criterion is met (which might happen at `x=xzero`) or because `kmax` iterations have been completed, the `xstar` returned along with `k` is $\mathbf{x}^k$.

For our purposes this elegant way of counting the iterates and iterations of an algorithm unfortunately has one little infelicity. Often we want to invoke a serially-reusable MATLAB function repeatedly in a loop, having it perform a single iteration each time as described in §10.6.1. That way we can study how the method works without cluttering up the algorithm code with statements to save the iterates, draw graphs, and so on. To invoke `countk.m` in a loop so that it performs one iteration at a time we need code like this.

```
x=xzero;
for p=1:pmax
    [xstar,k]=countk(x,0);
    x=xstar;
end
```

To get a single iteration it is necessary to pass `kmax=0` to `countk.m`, so in this context `kmax` is *one fewer than* the maximum number of iterations that are to be done. At each iteration of the loop over `p`, `countk.m` returns `k=0`, which is likewise *one fewer than* the single iteration (update to `x`) that it did if convergence was not attained.

The need to think of `k` and `kmax` differently in the algorithm code and in the driver program is potentially quite confusing. In an effort to make the single-iteration use of a routine like `countk.m` more intuitive, I have tried to consistently follow the alternative indexing scheme illustrated at the top of the next page.


```
function [xstar,kp]=countkp(xzero,kmax)
  x=xzero;
  for kp=1:kmax
      if(close enough)
         break
      else
         x=update(x);
      end
  end
  xstar=x;
end
```

Here `kmax` is again the index of the final new iterate that will be generated if convergence is not attained. Now, however, the index `kp` counts the iterations (updates to `x`) that are performed if convergence is not attained, rather than the iterates (which start with $\mathbf{x}^0$, not $\mathbf{x}^1$). If convergence is not attained this routine returns `xstar` = $\mathbf{x}^{\text{kmax}}$ and `kp=kmax`, and the iterates are $\mathbf{x}^0$, $\mathbf{x}^1$, ..., $\mathbf{x}^{\text{kmax}}$.

If convergence *is* attained, then the number of updates that were made to `x` is `kp-1` so that is how many iterations were used. For example, if `kmax=10` and the algorithm returns with `kp=3`, the statements were executed in this sequence: `1 2 3(kp=1) 4 6 7(update x) 8 9 3(kp=2) 4 6 7(update x) 8 9 3(kp=3) 4 5 10 11`. There were two updates to `x`, so the `xstar` that is returned is $\mathbf{x}^2$, and the problem was solved in two iterations. If `xzero` satisfies the convergence criterion, the routine returns `xstar=xzero` and `kp=1` so the problem was solved in `kp-1=0` iterations. If k is the index of the iterate that is returned in `xstar`, then if convergence is attained k = `kp-1` or `kp` = $k+1$ (the name `kp` is meant to suggest k plus one).

In the program below we ask for `kmax=1` more iteration to be performed in each invocation of `countkp`, and each time `countkp` returns it reports that `kp=1` iteration was performed. When exercising a routine in this way we typically set the convergence tolerance so that convergence is never attained, so `kp=1` corresponds to one update of `x`.

```
x=xzero
xsave(1)=x
isave(1)=1
for p=1:pmax
    [xstar,kp]=countkp(x,1)
    xsave(p+1)=xstar
    isave(p+1)=p
    x=xstar
end
plot(psave,xsave)
```

Another potential source of confusion in the counting of iterations arises from the fact that MATLAB unhelpfully prohibits zero array subscripts. The code above saves `xzero` in `xsave(1)` rather than in `xsave(0)`, and subsequent iterates in `xsave(p+1)` rather than in `xsave(p)`. Several mathematicians in my acquaintance covet the convenience of MATLAB but use FORTRAN instead simply to avoid being confused by this trifling quirk. I myself have better reasons (see §0.2.3) to prefer FORTRAN over MATLAB for production code.

28.5 Linear Programs Used in the Text

For each named linear programming example I have shown below an initial tableau for the minimization and whatever primal and dual solutions the problem has.

28.5.1 twoexams

	x_1	x_2	s_1	s_2	s_3	s_4	s_5
40	−12	−10	0	0	0	0	0
−20	−12	0	1	0	0	0	0
−60	0	−10	0	1	0	0	0
12	1	1	0	0	1	0	0
60	12	0	0	0	0	1	0
100	0	10	0	0	0	0	1

$$\begin{aligned}\left[\mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= [5, 7 \mid 40, 10, 0, 0, 30]\\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0, 0, 10, \tfrac{1}{6}, 0]\\ \text{primal } z^\star &= -170\end{aligned}$$

28.5.2 brewery

	x_1	x_2	x_3	x_4	s_1	s_2	s_3
0	−90	−150	−60	−70	0	0	0
160	7	10	8	12	1	0	0
50	1	3	1	1	0	1	0
60	2	4	1	3	0	0	1

$$\begin{aligned}\left[\mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= [5, 12\tfrac{1}{2}, 0, 0 \mid 0, 7\tfrac{1}{2}, 0]\\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [7\tfrac{1}{2}, 0, 18\tfrac{3}{4} \mid 0, 0, 18\tfrac{3}{4}, 76\tfrac{1}{4}]\\ \text{primal } z^\star &= -2325\end{aligned}$$

This problem is modeled after, but different from, the brewery problem discussed in [3].

28.5.3 paint

	$x1$	$x2$	$s1$	$s2$	$s3$	$s4$
0	−114	−162	0	0	0	0
1500	5	3	1	0	0	0
2520	7	9	0	1	0	0
1200	2	4	0	0	1	0
0	−2	3	0	0	0	1

$$\begin{aligned}\left[\mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= [193\tfrac{11}{13}, 129\tfrac{3}{13} \mid 143\tfrac{1}{13}, 0, 295\tfrac{5}{13}, 0]\\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0, 17\tfrac{1}{13}, 0, 2\tfrac{10}{13} \mid 0, 0]\\ \text{primal } z^\star &= -43033\tfrac{11}{13}\end{aligned}$$

28.5.4 shift

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
−3	−1	0	0	0	0	0	0	−1	1	0	0	0	0	0	0	0
−6	−1	−1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
−14	0	−1	−1	0	0	0	0	0	0	0	1	0	0	0	0	0
−18	0	0	−1	−1	0	0	0	0	0	0	0	1	0	0	0	0
−16	0	0	0	−1	−1	0	0	0	0	0	0	0	1	0	0	0
−14	0	0	0	0	−1	−1	0	0	0	0	0	0	0	1	0	0
−12	0	0	0	0	0	−1	−1	0	0	0	0	0	0	0	1	0
−6	0	0	0	0	0	0	−1	−1	0	0	0	0	0	0	0	1

$$\begin{aligned}\left[\mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= [3, 4, 10, 8, 8, 6, 6, 0 \mid 0, 1, 0, 0, 0, 0, 0, 0]\\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [1, 0, 1, 0, 1, 0, 1, 0 \mid 0, 0, 0, 0, 0, 0, 0]\\ \text{primal } z^\star &= 45\end{aligned}$$

28.5.5 chairs

	s_1	s_2	s_3	a_1	a_2	a_3	f_1	f_2	f_3	u_1	u_2	u_3	x_1	x_2	x_3
	0	0	−120	0	0	−120	−300	−300	−180	−120	−120	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	0	0	0	0	−50	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	−50	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	−50
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	−1	0	0	0	0	0	1	0	0	1	0	0	0	0	0
0	0	−1	0	0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	−1	0	0	0	0	0	1	0	0	1	0	0	0
200	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
200	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
100	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	−1	1	0	−1	0	0	1	0	0	1	0	0	0	0	0
0	0	−1	1	0	−1	0	0	1	0	0	1	0	0	0	0
0	0	0	1	0	0	1	0	0	−1	0	0	−1	0	0	0

y_1	y_2	y_3	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
−25	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	−25	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	−25	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$$\left[\mathbf{x}^{\star\top} \mid \mathbf{y}^{\star\top} \mid \mathbf{u}^{\star\top} \mid \mathbf{s}^{\star\top} \mid \mathbf{a}^{\star\top} \mid \mathbf{f}^{\star\top}\right] = [4,4,0 \mid 4,8,8 \mid 0,0,0 \mid 100,200,200 \mid 200,200,0 \mid 100,200,200]$$

$$\left[\mathbf{y}^{\star\top} \,\middle|\, \mathbf{w}^{\top\star}\right] = [0,0,0,0,0,0,0,0,0,180,0,0,0,0,0,0,0,0 \mid 0,0,0,0,0,0,0,0,0,300,300,300,300,300]$$

$$\text{primal } z^\star = -150000$$

28.5.6 `pumps`

	t	x_A	x_B	s_1	s_2	s_3
	1	0	0	0	0	0
0	−1	1	0	1	0	0
0	−1	0	1	0	1	0
16	0	2	8	0	0	1
60	0	12	20	0	0	0

$$\begin{aligned}\left[t^\star \mid \mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= \left[\tfrac{20}{7} \mid \tfrac{20}{7}, \tfrac{9}{7} \mid 0, \tfrac{11}{7}, 0\right]\\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= \left[0,0,0,0 \mid 1, 0, \tfrac{5}{14}\right]\\ \text{primal } z^\star &= \tfrac{20}{7}\end{aligned}$$

28.5.7 `bulb`

	a^+	a^-	b	u_2	v_2	u_3	v_3	u_4	v_4	u_5	v_5
	0	0	0	1	1	1	1	1	1	1	1
2.5	+10	−10	3.162277660168379	1	−1	0	0	0	0	0	0
5.3	+50	−50	7.071067811865475	0	0	1	−1	0	0	0	0
7.4	+90	−90	9.486832980505138	0	0	0	0	1	−1	0	0
8.5	+120	−120	10.95445115010332	0	0	0	0	0	0	1	−1

$$\begin{aligned}\left[\mathbf{u}^{\star\top} \mid \mathbf{v}^{\star\top} \mid \mathbf{a}^\star \mid b^\star\right] &= [0,0,0.012644760248259,0 \mid 0,0.238279533746637,0 \mid\\ &\quad -0.001877412670804 \mid 0.796506315189896]\\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0,0,0,0 \mid 0,0.450646905677266,1.549353094322734,2,2,\\ &\quad 1.379112757860228,0.620887242139772]\\ \text{primal } z^\star &= -0.250924293994896\end{aligned}$$

In the original formulation the variable a is unconstrained in sign so in standard form it is represented as $a = a^+ - a^-$ where a^+ and a^- are nonnegative (see §2.9.3).

28.5.8 `unbd`

	x_1	x_2	x_3	x_4	x_5
−9	0	0	−2	1	0
3	0	0	−1	2	1
1	1	0	0	1	0
5	0	1	−4	1	0

$$\begin{aligned}\mathbf{x}^\star &= \lim_{t\to\infty}[1,\ 5+4t,\ t,\ 0,\ 3+t]^\top\\ \text{the dual} &\ \text{is}\ \text{infeasible}\\ \text{primal } z^\star &= \lim_{t\to\infty}(9-2t) = -\infty\end{aligned}$$

This problem is similar to, but different from, the one discussed in [3, p48-49].

28.5.9 `infea`

	x_1	x_2	x_3	x_4
2	0	0	−3	8
1	0	1	5	−1
4	0	0	0	0
−7	1	0	2	6

the primal is infeasible
the dual is unbounded
primal $z^\star$ is not defined

This tableau is in both infeasible form 1 and infeasible form 2 (see §2.5.3).

28.5.10 sf1

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	−8	6	2	0	−7	5	0
−1	0	−3	0	8	6	−4	3
−2	−9	7	0	−5	0	0	−9
3	−6	0	1	−7	4	−6	5
4	9	−5	0	0	3	9	4
1	0	−1	0	3	9	5	−2

$$\begin{aligned} \mathbf{x}^{\star\top} &= \left[\tfrac{235}{153}, \tfrac{66}{17}, 0, 0, \tfrac{47}{51}, 0, \tfrac{29}{17}\right] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= \left[0, 0, 0, 0 \mid \tfrac{13}{15}, \tfrac{998}{45}, \tfrac{284}{45}\right] \\ \text{primal } z^\star &= \tfrac{41}{9} \end{aligned}$$

This tableau has a redundant row.

28.5.11 sf2

	x_1	x_2	x_3	x_4	x_5	x_6
0	0	0	4	−1	2	0
−15	0	0	−1	1	−1	1
−8	1	0	0	−1	0	0
−5	0	1	−1	3	−2	0

$$\begin{aligned} \mathbf{x}^{\star\top} &= [0, 17, 0, 8, 23, 0] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= \left[0, 0, 0 \mid 1, 2, 2\right] \\ \text{primal } z^\star &= -38 \end{aligned}$$

28.5.12 graph

	x_1	x_2	s_1	s_2	s_3	s_4
0	−2	−1	0	0	0	0
6	1	$\frac{6}{5}$	1	0	0	0
2	1	−1	0	1	0	0
3	1	0	0	0	1	0
5	0	1	0	0	0	1

$$\begin{aligned} \left[\mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= \left[3, \tfrac{5}{2} \mid 0, \tfrac{3}{2}, 0, \tfrac{5}{2}\right] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= \left[0, 0, 0, 0 \mid \tfrac{5}{6}, \tfrac{7}{6}\right] \\ \text{primal } z^\star &= \tfrac{17}{2} \end{aligned}$$

This problem is modeled after the first example in [3, §4.1].

28.5.13 pm

	x_1	x_2	x_3	x_4
−3	0	1	0	−2
3	1	1	0	1
2	0	−4	1	2

$$\begin{aligned} \mathbf{x}^{\star\top} &= \left[0, \tfrac{4}{5}, 0, \tfrac{11}{5}\right] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= \left[0, 0, \mid \tfrac{4}{5}, \tfrac{3}{5}\right] \\ \text{primal } z^\star &= -\tfrac{3}{5} \end{aligned}$$

28.5.14 cycle

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	0	0	0	$-\frac{3}{4}$	20	$-\frac{1}{2}$	6
0	1	0	0	$\frac{1}{4}$	−8	−1	9
0	0	1	0	$\frac{1}{2}$	−12	$-\frac{1}{2}$	3
1	0	0	1	0	0	1	0

$$\begin{aligned} \mathbf{x}^{\star\top} &= \left[\tfrac{3}{4}, 0, 0, 1, 0, 1, 0\right] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= \left[0, \tfrac{3}{2}, \tfrac{5}{4} \mid 0, 2, 0, \tfrac{21}{2}\right] \\ \text{primal } z^\star &= -\tfrac{17}{4} \end{aligned}$$

28.5.15 in1

	x_1	x_2	s_1	s_2
0	1	1	0	0
1	−1	1	1	0
1	1	0	0	1

$$\begin{aligned} \left[\mathbf{x}^{\star\top} \mid \mathbf{s}^{\star\top}\right] &= [0,0 \mid 1,1] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0,0 \mid 1,1] \\ \text{primal } z^\star &= 0 \end{aligned}$$

28.5.16 nf1

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	−8	6	2	0	−7	5	0
−1	0	−3	0	8	6	−4	3
−2	−9	7	0	−5	0	0	−9
3	−6	0	1	−7	4	−6	5
4	9	−5	0	0	3	9	4
1	0	−1	0	3	9	5	−2

$$\begin{aligned} \mathbf{x}^{\star\top} &= [20,15,15,10,0,0,0,0,15,0] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0,0,0,0,0 \mid 12,18,22,35,4] \\ \text{primal } z^\star &= 915 \end{aligned}$$

This tableau has a redundant row.

28.5.17 nf2

	x_{14}	x_{15}	x_{16}	x_{24}	x_{25}	x_{26}	x_{34}	x_{35}	x_{36}
0	2	4	3	1	5	2	1	1	6
20	1	1	1	0	0	0	0	0	0
20	0	0	0	1	1	1	0	0	0
20	0	0	0	0	0	0	1	1	1
10	1	0	0	1	0	0	1	0	0
25	0	1	0	0	1	0	0	1	0
25	0	0	1	0	0	1	0	0	1

$$\begin{aligned} \mathbf{x}^{\star\top} &= [10,5,5,0,0,20,0,20,0] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0,0,0,0,0 \mid 0,2,2,6] \\ \text{primal } z^\star &= 115 \end{aligned}$$

This tableau has a redundant row.

28.5.18 nf3

	x_{14}	x_{15}	x_{16}	x_{24}	x_{25}	x_{26}	x_{34}	x_{35}	x_{36}
0	9	3	1	2	3	7	3	1	1
10	1	1	1	0	0	0	0	0	0
15	0	0	0	1	1	1	0	0	0
10	0	0	0	0	0	0	1	1	1
10	1	0	0	1	0	0	1	0	0
5	0	1	0	0	1	0	0	1	0
20	0	0	1	0	0	1	0	0	1

$$\begin{aligned} \mathbf{x}^{\star\top} &= [0,0,10,10,5,0,0,0,10] \\ \left[\mathbf{y}^{\star\top} \mid \mathbf{w}^{\star\top}\right] &= [0,0,0,0,0 \mid 9,2,4,3] \\ \text{primal } z^\star &= 55 \end{aligned}$$

This tableau has a redundant row.

28.6 Integer Linear Programs Used in the Text

For each named integer linear programming example I have repeated below the analytic statement of the problem and given its solution.

28.6.1 brewip

$$\begin{array}{lrcrcrcrcr}
\underset{\mathbf{x}\in\mathbb{Z}^4}{\text{minimize}} & -90x_1 & - & 150x_2 & - & 60x_3 & - & 70x_4 & & \\
\text{subject to} & 7x_1 & + & 10x_2 & + & 8x_3 & + & 12x_4 & \leq & 160 \\
 & 1x_1 & + & 3x_2 & + & 1x_3 & + & 1x_4 & \leq & 50 \\
 & 2x_1 & + & 4x_2 & + & 1x_3 & + & 3x_4 & \leq & 60 \\
 & & & & & & & x_j & \geq & 0 \quad \text{and integer, } j = 1\ldots4
\end{array}$$

$$\begin{aligned}
\mathbf{x}^\star &= [4, 13, 0, 0]^\top \\
z^\star &= -2310
\end{aligned}$$

28.6.2 spear

$$\begin{array}{lrcrcr}
\underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 & - & x_2 & & \\
\text{subject to} & -13x_1 & + & 14x_2 & \leq & 14 \\
 & 15x_1 & - & 14x_2 & \leq & 0 \\
 & & & x_j & \geq & 0 \quad \text{and integer, } j = 1\ldots2
\end{array}$$

$$\begin{aligned}
\mathbf{x}^\star &= [0, 1]^\top \\
z^\star &= -1
\end{aligned}$$

This problem is modeled after the example in [3, §8.1].

28.6.3 bb1

$$\begin{array}{lrcrcr}
\underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 - 3x_2 & & & & \\
\text{subject to} & -x_1 + & x_2 & \leq & 2 & \\
 & x_1 + & x_2 & \leq & 6\tfrac{1}{2} & \\
 & & x_j & \geq & 0 & \text{and integer, } j = 1\ldots2
\end{array}$$

$$\begin{aligned}
\mathbf{x}^\star &= [2, 4]^\top \\
z^\star &= -14
\end{aligned}$$

28.6.4 bb2

$$\begin{array}{lrcr}
\underset{\mathbf{x}\in\mathbb{Z}^3}{\text{minimize}} & -4x_1 - 5x_2 - x_3 & & \\
\text{subject to} & 3x_1 + 2x_2 & \leq & 10 \\
 & x_1 + 4x_2 & \leq & 11 \\
 & 3x_1 + 3x_2 + x_3 & \leq & 13 \\
 & x_j & \geq & 0 \quad \text{and integer, } j = 1\ldots3
\end{array}$$

$$\begin{aligned}
\mathbf{x}^\star &= [2, 2, 1]^\top \\
z^\star &= -19
\end{aligned}$$

28.6.5 `bb3`

$$
\begin{array}{lrcll}
\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -x_2 & & & \\
\text{subject to} & -x_1 + x_2 & \leq & 0 & \\
& x_1 + x_2 & \leq & 7 & \\
& x_j & \geq & 0 & \text{and integer, } j = 1\ldots 2
\end{array}
$$

$$
\begin{aligned}
\mathbf{x}^{\star 1} &= [3,3]^\top \\
\mathbf{x}^{\star 2} &= [4,3]^\top \\
z^\star &= -3
\end{aligned}
$$

28.6.6 `bb4`

$$
\begin{array}{lrcll}
\underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} & -x_1 + x_2 & & & \\
\text{subject to} & x_1 - x_2 & \leq & 3 & \\
& x_2 & \leq & 3\tfrac{1}{3} & \\
& x_j & \geq & 0 & \text{and integer, } j = 1\ldots 2
\end{array}
$$

$$
\begin{aligned}
\mathbf{x}^{\star 1} &= [3,0]^\top \\
\mathbf{x}^{\star 2} &= [4,1]^\top \\
\mathbf{x}^{\star 3} &= [5,2]^\top \\
\mathbf{x}^{\star 4} &= [6,3]^\top \\
z^\star &= -3
\end{aligned}
$$

The optima $\mathbf{x}^{\star 2}$ and $\mathbf{x}^{\star 3}$ are invisible to the branch-and-bound algorithm of §7.4.

28.6.7 `bb5`

$$
\begin{array}{lrcr}
\underset{\mathbf{x}\in\mathbb{Z}^6}{\text{minimize}} & 2x_1 + 2x_2 + 4x_3 + 7x_4 + 8x_5 + 9x_6 & = & z(\mathbf{x}) \\
\text{subject to} & -5x_1 + 3x_2 - 2x_3 + 3x_4 + x_5 - 2x_6 & \leq & 5 \\
& x_1 - 2x_3 - x_4 - 3x_5 + 3x_6 & \leq & 1 \\
& -x_1 - 2x_2 + x_3 - x_4 + 5x_5 + x_6 & \leq & -3 \\
& x_1, x_2, x_3, x_4, x_5, x_6 & \in & \{0,1\}
\end{array}
$$

$$
\begin{aligned}
\mathbf{x}^\star &= [1,1,0,0,0,0]^\top \\
z^\star &= 4
\end{aligned}
$$

28.7 Nonlinear Programs Used in the Text

For each named nonlinear programming example I have given below an algebraic statement of the standard-form problem, bounds $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ on the variables from which a starting point $\mathbf{x}^0 = \frac{1}{2}(\mathbf{x}^{\mathrm{L}} + \mathbf{x}^{\mathrm{H}})$ can be computed if none is given, the optimal solution $\mathbf{x}^\star$ and, if the problem has constraints, its optimal KKT multipliers $\boldsymbol{\lambda}^\star$.

28.7.1 garden

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = -x_1x_2 & & \\ \text{subject to} & 2x_1 + x_2 - 40 & \leq & 0 \\ & x_2 - 30 & \leq & 0 \\ & -x_1 & \leq & 0 \\ & -x_2 & \leq & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0,0]^\top \quad \mathbf{x}^\star = [10,20]^\top \quad \mathbf{x}^{\mathrm{H}} = [40,40]^\top \quad f_0(\mathbf{x}^\star) = 200 \quad \boldsymbol{\lambda}^\star = [10,0,0,0]^\top$$

28.7.2 rb

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f(\mathbf{x}) \;=\; 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

$$\mathbf{x}^{\mathrm{L}} = [-2,-1]^\top \quad \mathbf{x}^0 = [-1.2,1]^\top \quad \mathbf{x}^\star = [1,1]^\top \quad \mathbf{x}^{\mathrm{H}} = [2,2]^\top \quad f(\mathbf{x}^\star) = 0$$

This problem is from [135].

28.7.3 gpr

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f(\mathbf{x}) & = & e^{u^2} + \sin^4(v) + \frac{1}{2}w^2 \\ \text{where} & u & = & \frac{1}{2}(x_1^2 + x_2^2 - 25) \\ & v & = & 4x_1 - 3x_2 \\ & w & = & 2x_1 + x_2 - 10 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [2,3]^\top \quad \mathbf{x}^\star = [3,4]^\top \quad \mathbf{x}^{\mathrm{H}} = [4,5]^\top \quad f(\mathbf{x}^\star) = 1$$

This problem is from [66, p572-574].

28.7.4 gns

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f(\mathbf{x}) \;=\; 4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1$$

$$\mathbf{x}^{\mathrm{L}} = [-2,-2]^\top \quad \mathbf{x}^0 = [2,2]^\top \quad \mathbf{x}^\star = [\tfrac{3}{4},-\tfrac{3}{4}]^\top \quad \mathbf{x}^{\mathrm{H}} = [3,3]^\top \quad f(\mathbf{x}^\star) = -\tfrac{9}{8}$$

This problem is from [4, Exercise 2.1].

28.7.5 arch1

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = (x_1 - 1)^2 + (x_2 - 1)^2 & & \\ \text{subject to} & 4 - (x_1 - 2)^2 - x_2 & = & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0,0]^\top \quad \mathbf{x}^\star = [0.327018352145058, 1.201132405940562]^\top \quad \mathbf{x}^{\mathrm{H}} = [4,4]$$
$$f_0(\mathbf{x}^\star) = 0.493358543068992 \quad \lambda^\star = 0.402264811881125$$

28.7.6 hill

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) = x_1^2 + x_2^2 + x_3^2 & \\ \text{subject to} & 4 - \frac{1}{9}x_1^2 - x_3 & = \ 0 \\ & 4 - \frac{4}{9}(4 - x_2)^2 - x_3 & = \ 0 \end{array}$$

$\mathbf{x}^{\mathrm{L1}} = [3, 2, 2]^\top$ $\quad \mathbf{x}^{\star 1} = [+3.23137107379720, 2.38431446310140, 2.83980455371408]^\top$ $\quad \mathbf{x}^{\mathrm{H1}} = [4, 3, 3]^\top$

$\mathbf{x}^{\mathrm{L2}} = [-4, 2, 2]^\top$ $\quad \mathbf{x}^{\star 2} = [-3.23137107379720, 2.38431446310140, 2.83980455371408]^\top$ $\quad \mathbf{x}^{\mathrm{H2}} = [-3, 3, 3]^\top$

$f_0(\mathbf{x}^{\star 1}) = f_0(\mathbf{x}^{\star 2}) = 24.1912043788230$ $\quad \boldsymbol{\lambda}^\star = [9, -3.32039089257184]^\top$

28.7.7 one23

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^3}{\text{minimize}} & f_0(\mathbf{x}) = x_1 + x_2^2 + x_3^3 & \\ \text{subject to} & x_1 + x_2 + x_3 - 1 & = \ 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = \left[-\tfrac{1}{2}, -1, 0\right]^\top \quad \mathbf{x}^\star = \left[\tfrac{1}{2} - \sqrt{\tfrac{1}{3}},\ \tfrac{1}{2},\ \sqrt{\tfrac{1}{3}}\right]^\top \quad \mathbf{x}^{\mathrm{H}} = \left[\tfrac{1}{2}, 1, 1\right]^\top$$
$$f_0(\mathbf{x}^\star) = 0.365099820540249 \qquad \lambda^\star = -1$$

This problem's other Lagrange point, $\bar{\mathbf{x}} = \left[\frac{1}{2} + \sqrt{\frac{1}{3}},\ \frac{1}{2},\ -\sqrt{\frac{1}{3}}\right]^\top$, is a maximizing point with $f_0(\bar{\mathbf{x}}) = 1.13490017945975$.

28.7.8 arch2

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = (x_1 - 1)^2 + (x_2 - 1)^2 & \\ \text{subject to} & 4 - (x_1 - 2)^2 - x_2 & \leq \ 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = [0.327018352145058, 1.201132405940562]^\top \quad \mathbf{x}^{\mathrm{H}} = [4, 4]$$
$$f_0(\mathbf{x}^\star) = 0.493358543068992 \quad \lambda^\star = 0.402264811881125$$

28.7.9 arch3

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = (x_1 - 1)^2 + (x_2 - 1)^2 & \\ \text{subject to} & 4 - (x_1 - 2)^2 - x_2 & \geq \ 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = [1, 1]^\top \quad \mathbf{x}^{\mathrm{H}} = [4, 4]$$
$$f_0(\mathbf{x}^\star) = 0 \quad \lambda^\star = 0$$

28.7.10 arch4

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = (x_1-1)^2 + (x_2-1)^2 & \\ \text{subject to} & 4-(x_1-2)^2 - x_2 & \leq 0 \\ & \frac{13}{8} + \frac{1}{4}x_1 - x_2 & \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0,0]^\top \quad \mathbf{x}^\star = [\tfrac{1}{2}, \tfrac{7}{4}]^\top \quad \mathbf{x}^{\mathrm{H}} = [4,4]$$

$$f_0(\mathbf{x}^\star) = \tfrac{13}{16} \quad \boldsymbol{\lambda}^\star = \left[\tfrac{5}{22}, \tfrac{14}{11}\right]^\top$$

28.7.11 moon

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -(x_1-3)^2 - x_2^2 & \\ \text{subject to} & x_1^2 + x_2^2 - 1 & \leq 0 \\ & -(x_1+2)^2 - x_2^2 + 4 & \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-6,-2]^\top \quad \mathbf{x}^{\star 1} = [-1/4, +\sqrt{15/16}]^\top \quad \mathbf{x}^{\star 2} = [-1/4, -\sqrt{15/16}]^\top \quad \mathbf{x}^{\mathrm{H}} = [2,6]^\top$$

$$f(\mathbf{x}^\star) = -\tfrac{23}{2} \quad \boldsymbol{\lambda}^\star = \left[\tfrac{5}{2}, \tfrac{3}{2}\right]^\top$$

28.7.12 cq1

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & -x_1 & \\ \text{subject to} & x_2 - (1-x_1)^3 & \leq 0 \\ & -x_2 & \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-2,-2]^\top \quad \mathbf{x}^\star = [1,0]^\top \quad \mathbf{x}^{\mathrm{H}} = [2,4] \quad f_0(\mathbf{x}^\star) = -1 \quad \boldsymbol{\lambda}^\star \text{ is undefined}$$

This problem has no constraint qualification.

28.7.13 cq2

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & (x_1-1)^2 + (x_2-1)^2 & \\ \text{subject to} & x_2 & \leq 0 \\ & -x_2 & \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-2,-1]^\top \quad \mathbf{x}^\star = [1,0] \quad \mathbf{x}^{\mathrm{H}} = [2,3]^\top \quad f_0(\mathbf{x}^\star) = 1 \qquad \lambda_1^\star \geq 2; \;\; \lambda_2^\star = \lambda_1^\star - 2$$

The gradients of the active constraints are not linearly independent, so λ_1 and λ_2 are not uniquely determined. However, the cone of tangents $\mathbb{T}$ is equal to the cone of feasible directions $\mathbb{F}$, so a constraint qualification is satisfied.

28.7.14 cq3

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & x_1 & & \\ \text{subject to} & x_2 - \frac{1}{2} + (x_1 - 1)^2 & \leq & 0 \\ & -x_2 - \frac{1}{2} + (x_1 - 1)^2 & \leq & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-1, -1]^\top \quad \mathbf{x}^\star = [1 - 1/\sqrt{2}, 0]^\top \quad \mathbf{x}^{\mathrm{H}} = [3, 1]^\top$$
$$f_0(\mathbf{x}^\star) = 1 - 1/\sqrt{2} \qquad \boldsymbol{\lambda}^\star = [\sqrt{2}/4,\ \sqrt{2}/4]^\top$$

28.7.15 branin

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & 2x_1^2 - \frac{21}{20}x_1^4 + \frac{1}{6}x_1^6 + x_1x_2 + x_2^2 & & \\ \text{subject to} & -x_1 + 1 & \leq & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = [1.7475523472644516, -0.87377617567992016]^\top \quad \mathbf{x}^{\mathrm{H}} = [2, 2]^\top$$
$$f_0(\mathbf{x}^\star) = 0.29863844223685942 \quad \lambda^\star = 0$$

This is Branin's three-hump camel-back problem from [19], but with an added constraint. The objective has another local minimum at $-\mathbf{x}^\star$ with $f_0(-\mathbf{x}^\star) = f_0(\mathbf{x}^\star)$, and a unique global minimum at $\hat{\mathbf{x}} = [0, 0]^\top$ with $f_0(\hat{\mathbf{x}}) = 0$; both of these points violate the constraint, though it is inactive at the optimal point.

28.7.16 hearn

$$\begin{array}{rl} \underset{\mathbf{x}\in\mathbb{X}}{\text{minimize}} & f_0(\mathbf{x}) = \dfrac{(1 - x_2)^2}{2x_1} + \dfrac{(2 - x_1)^2}{2x_2} + 5x_1 + 4x_2 + \dfrac{1}{2} \\ \text{where} & \mathbb{X} = \left\{\mathbf{x} \in \mathbb{R}^2 \,\middle|\, x_1 > 0, x_2 > 0\right\} \cup [0, 1]^\top \cup [2, 0]^\top \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = [0, 1]^\top \quad \mathbf{x}^{\mathrm{H}} = [0.05, 1.80]^\top \quad f_0(\mathbf{x}^\star) = \tfrac{13}{2}$$

The objective value cannot be calculated at $\mathbf{x}^\star$, so the nonlinear programming model breaks down at the optimal point and the problem is ill-posed.

28.7.17 nset

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & (x_1 - \frac{1}{2})^2 + x_2^2 & & \\ \text{subject to} & \cos(x_1) + x_2 & \leq & 0 \\ & \frac{1}{2}(x_1 - \frac{1}{4})^2 - x_2 - 1\frac{1}{4} & \leq & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-2, -6]^\top \quad \mathbf{x}^\star = [0.967281605376012; -0.567539804600159]^\top \quad \mathbf{x}^{\mathrm{H}} = [6, 2]^\top$$
$$f_0(\mathbf{x}^\star) = 0.540453528528370 \quad \boldsymbol{\lambda}^\star = [1.135079609200316, 0]^\top$$

28.7.18 `h35`

$$\begin{array}{rrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f(\mathbf{x}) & = & v_1^2 + v_2^2 + v_3^2 \\ \text{where} & v_t & = & c_t - x_1(1 - x_2^t), \quad t = 1, 2, 3 \\ & c_1 & = & 1.5 \\ & c_2 & = & 2.25 \\ & c_3 & = & 2.625 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = \left[3, \tfrac{1}{2}\right]^\top \quad \mathbf{x}^{\mathrm{H}} = \left[5, \tfrac{3}{5}\right]^\top \quad f(\mathbf{x}^\star) = 0$$

This problem is adapted from [80, p122,431], which specifies a starting point $\mathbf{x}^0 = [2, 0.2]^\top$ different from the midpoint of these bounds.

28.7.19 `bss1`

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \quad f_0(\mathbf{x}) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$$

$$\mathbf{x}^{\mathrm{L}} = [-2, 0]^\top \quad \mathbf{x}^\star = [2, 1]^\top \quad \mathbf{x}^{\mathrm{H}} = [2, 6]^\top \quad f(\mathbf{x}^\star) = 0$$

This problem is from [1, §8.6.4].

28.7.20 `p1`

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = -x_1 x_2 \\ \text{subject to} & x_1 + 2x_2 - 4 = 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = [2, 1]^\top \quad \mathbf{x}^{\mathrm{H}} = [8, 8]^\top \quad z^\star = -2 \quad \lambda^\star = 1$$

This problem is [5, Example 16.5].

28.7.21 `p2`

$$\begin{array}{rrcl} \text{minimize} & f_0(\mathbf{x}) & = & (x_1 - 2)^4 + (x_1 - 2x_2)^2 \\ \text{subject to} & x_1^2 - x_2 & = & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}^\star = [0.945582993415968, 0.894127197437503]^\top \quad \mathbf{x}^{\mathrm{H}} = [2, 4]^\top$$
$$f_0(\mathbf{x}^\star) = 1.94618371044280 \qquad \lambda^\star = 3.37068560583616$$

This problem is [1, Example 9.2.3].

28.7.22 `b1`

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = x_1 - 2x_2 \\ \text{subject to} & \begin{array}{rcl} -x_1 + x_2^2 - 1 & \leq & 0 \\ -x_2 & \leq & 0 \end{array} \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-2, -2]^\top \quad \mathbf{x}^\star = [0, 1]^\top \quad \mathbf{x}^{\mathrm{H}} = [3, 3]^\top \quad f_0(\mathbf{x}^\star) = -2 \quad \boldsymbol{\lambda}^\star = [1, 0]^\top$$

This problem is [4, Example 16.1].

28.7.23 b2

$$\begin{array}{lrcl} \text{minimize} & f_0(\mathbf{x}) & = & (x_1-2)^4+(x_1-2x_2)^2 \\ \text{subject to} & x_1^2-x_2 & \leq & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0,0]^\top \quad \mathbf{x}^\star = [0.945582993415968, 0.894127197437503]^\top \quad \mathbf{x}^{\mathrm{H}} = [2,4]^\top$$
$$f_0(\mathbf{x}^\star) = 1.94618371044280 \qquad \lambda^\star = 3.37068560583616$$

This problem is [1, Example 9.4.4].

28.7.24 ep1

$$\begin{array}{ll} \underset{x\in\mathbb{R}^1}{\text{minimize}} & f_0(x) = x^2 \\ \text{subject to} & 1-x \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = 0 \quad \mathbf{x}^\star = 1 \quad \mathbf{x}^{\mathrm{H}} = 4 \quad f_0(\mathbf{x}^\star) = 1 \quad \lambda^\star = 2$$

28.7.25 ep2

$$\begin{array}{ll} \underset{x\in\mathbb{R}^2}{\text{minimize}} & f_0(x) = x_1^2+x_2^2 \\ \text{subject to} & 2-x_1-x_2 \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0,0]^\top \quad \mathbf{x}^\star = [1,1]^\top \quad \mathbf{x}^{\mathrm{H}} = [4,4]^\top \quad f_0(\mathbf{x}^\star) = 2 \quad \lambda^\star = 2$$

28.7.26 al2

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(x) = -x_1-x_2 \\ \text{subject to} & x_1^2+x_2^2-2 = 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [0,0]^\top \quad \mathbf{x}^\star = [1,1]^\top \quad \mathbf{x}^{\mathrm{H}} = [4,4]^\top \quad f_0(\mathbf{x}^\star) = -2 \quad \lambda^\star = \tfrac{1}{2}$$

This problem is [5, Example 17.1].

28.7.27 al1

$$\begin{array}{ll} \underset{x\in\mathbb{R}^1}{\text{minimize}} & f_0(x) = -x \\ \text{subject to} & \dfrac{1}{x}-1 = 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = -\tfrac{1}{2} \quad \mathbf{x}^\star = 1 \quad \mathbf{x}^{\mathrm{H}} = \tfrac{3}{2} \quad f_0(\mathbf{x}^\star) = -1 \quad \lambda^\star = -1$$

28.7.28 admm

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) = x_1^2 + x_2^2 + x_3^2 + x_4^2 & & \\ \text{subject to} & 3x_1 - x_2 - 2x_3 - x_4 + 1 & = & 0 \\ & -4x_1 + x_2 + 5x_3 + 2x_4 - 3 & = & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = \left[-\tfrac{63}{65}, -\tfrac{99}{65}, -\tfrac{378}{65}, -\tfrac{99}{65}\right]^{\mathsf{T}} \quad \mathbf{x}^\star = \left[\tfrac{7}{65}, -\tfrac{9}{65}, \tfrac{42}{65}, \tfrac{11}{65}\right]^{\mathsf{T}} \quad \mathbf{x}^{\mathrm{H}} = \left[\tfrac{77}{65}, \tfrac{81}{65}, \tfrac{462}{65}, \tfrac{121}{65}\right]^{\mathsf{T}}$$

$$f_0(\mathbf{x}^\star) = \tfrac{31}{65} \qquad \boldsymbol{\lambda}^\star = \left[-\tfrac{58}{65}, -\tfrac{8}{13}\right]^{\mathsf{T}}$$

Using $\mathbf{x}^\star$ and the starting point $\mathbf{x}^0 = [0, 0, 0, 0]^{\mathsf{T}}$ given in §20.3, I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the method described in §26.2.2 for case 1. Because the linear equations defining the feasible set have whole-number coefficients and the optimal point is the feasible vertex nearest the origin, its coordinates are rational fractions.

28.7.29 ek1

$$\begin{array}{lr} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = (x_1 - 20)^4 + (x_2 - 12)^4 \\ \text{subject to} & 8e^{(x_1-12)/9} - x_2 + 4 \le 0 \\ & 6(x_1 - 12)^2 + 25x_2 - 600 \le 0 \\ & -x_1 + 12 \le 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = \left[18 - \tfrac{9}{\sqrt{2}}, 21 - \tfrac{13}{\sqrt{2}}\right]^{\mathsf{T}} \quad \mathbf{x}^\star = [15.629490902306340, 15.973768617852247]^{\mathsf{T}} \quad \mathbf{x}^{\mathrm{H}} = \left[18 + \tfrac{9}{\sqrt{2}}, 21 + \tfrac{13}{\sqrt{2}}\right]^{\mathsf{T}}$$

$$f_0(\mathbf{x}^\star) = 614.21209720340380 \qquad \boldsymbol{\lambda}^\star = [250.99653438461144, 0, 0]^{\mathsf{T}}$$

This problem is from [3, p315-320].

28.7.30 qp1

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 + 2x_3^2 + 2x_4^2 + x_1x_4 + x_2x_3 \\ \text{subject to} & & & 3x_1 - x_2 - 2x_3 - x_4 + 1 = 0 \\ & & & -4x_1 + x_2 + 5x_3 + 2x_4 - 3 = 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = \left[-\tfrac{1928}{89}, -\tfrac{4485}{89}, -\tfrac{540}{89}, -\tfrac{130}{89}\right]^{\mathsf{T}} \quad \mathbf{x}^\star = \left[-\tfrac{3}{89}, -\tfrac{41}{89}, \tfrac{54}{89}, \tfrac{13}{89}\right]^{\mathsf{T}} \quad \mathbf{x}^{\mathrm{H}} = \left[\tfrac{1572}{89}, \tfrac{3595}{89}, \tfrac{540}{89}, \tfrac{130}{89}\right]^{\mathsf{T}}$$

$$f_0(\mathbf{x}^\star) = \tfrac{63}{89} \qquad \boldsymbol{\lambda}^\star = \left[-\tfrac{105}{89}, -\tfrac{77}{89}\right]^{\mathsf{T}}$$

The starting point $\mathbf{x}^0 = [-2, -5, 0, 0]^{\mathsf{T}}$ and exact optimal point are given in §22.1. Using them I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 1. To find $\boldsymbol{\lambda}^\star$ I used the

KKT conditions for the problem, which reduce to the following system of linear algebraic equations.

$$\begin{bmatrix} 2 & 0 & 0 & 1 & 3 & -4 \\ 0 & 2 & 1 & 0 & -1 & 1 \\ 0 & 1 & 4 & 0 & -2 & 5 \\ 1 & 0 & 0 & 4 & -1 & 2 \\ 3 & -1 & -2 & -1 & 0 & 0 \\ -4 & 1 & 5 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 3 \end{bmatrix}$$

This system also yields $\mathbf{x}^\star$, and because the coefficients in the linear system are whole numbers its solution components are rational fractions.

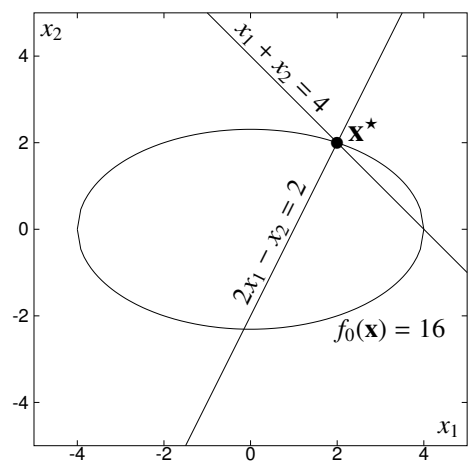
28.7.31 qp2

$$\begin{array}{ll} \underset{x_1,\, x_2}{\text{minimize}} & f_0(\mathbf{x}) = x_1^2 \\ \text{subject to} & x_1 = 1 \end{array}$$

$$\mathbf{x}^\star = [1, x_2]^\top \text{ for any } x_2 \in \mathbb{R}^1 \quad f_0(\mathbf{x}^\star) = 1 \quad \lambda^\star = -2$$

28.7.32 qp3

The feasible set for this problem is the single point $\mathbf{x}^\star$. I solved the KKT conditions analytically to find $\boldsymbol{\lambda}^\star$.



$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = x_1^2 + 3x_2^2 & \\ \text{subject to} & x_1 + x_2 - 4 & = 0 \\ & 2x_1 - x_2 - 2 & = 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-5, -5]^\top \quad \mathbf{x}^\star = [2, 2]^\top \quad \mathbf{x}^{\mathrm{H}} = [5, 5]^\top$$

$$f_0(\mathbf{x}^\star) = 16 \quad \boldsymbol{\lambda}^\star = \left[-\tfrac{28}{3}, \tfrac{8}{3}\right]^\top$$

28.7.33 qp4

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 + 2x_3^2 + 2x_4^2 + x_1x_4 + x_2x_3 \\ \text{subject to} & & & 3x_1 - x_2 - 2x_3 - x_4 + 1 \leq 0 \\ & & & -4x_1 + x_2 + 5x_3 + 2x_4 - 3 \leq 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = \left[\tfrac{1}{40}, \tfrac{1}{260}, \tfrac{3}{520}, \tfrac{5}{520}\right]^\top \quad \mathbf{x}^\star = \left[\tfrac{1}{4}, \tfrac{1}{26}, \tfrac{3}{52}, \tfrac{5}{52}\right]^\top \quad \mathbf{x}^{\mathrm{H}} = \left[\tfrac{5}{2}, \tfrac{5}{13}, \tfrac{15}{26}, \tfrac{25}{26}\right]^\top \quad f_0(\mathbf{x}^\star) = \tfrac{7}{104} \quad \boldsymbol{\lambda}^\star = \left[\tfrac{7}{52}, 0\right]^\top$$

This problem is identical to qp1 except that the constraints are inequalities. I used qpin.m to find $\mathbf{x}^\star$ and $\boldsymbol{\lambda}^\star$, confirming that the first constraint is tight and the second is slack. The optimal point and multipliers can also be found from the KKT conditions of the equality-constrained problem, which reduce to the following system of linear algebraic equations.

$$\begin{bmatrix} 2 & 0 & 0 & 1 & 3 \\ 0 & 2 & 1 & 0 & -1 \\ 0 & 1 & 4 & 0 & -2 \\ 1 & 0 & 0 & 4 & -1 \\ 3 & -1 & -2 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \lambda_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix}$$

Because the coefficients in the linear system are whole numbers, the solution is rational fractions. Using $\mathbf{x}^\star$ I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 0.

28.7.34 qp5

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) & = & x_1^2 + x_2^2 - x_1x_2 - 12x_1 + 3x_2 \\ \text{subject to} & -x_1 + x_2 - 6 & \leq & 0 \\ & 2x_1 + x_2 - 3 & \leq & 0 \\ & \frac{1}{2}x_1 - x_2 - 10 & \leq & 0 \\ & -\frac{2}{3}x_1 - x_2 - 7 & \leq & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = \left[\tfrac{3}{7}, -\tfrac{472}{7}\right]^\top \quad \mathbf{x}^\star = \left[\tfrac{33}{14}, -\tfrac{12}{7}\right]^\top \quad \mathbf{x}^{\mathrm{H}} = \left[\tfrac{33}{7}, \tfrac{508}{7}\right]^\top$$

$$f_0(\mathbf{x}^\star) = -\tfrac{585}{28} \qquad \boldsymbol{\lambda}^\star = \left[0, \tfrac{39}{14}, 0, 0\right]^\top$$

I used qpin.m to find $\mathbf{x}^\star$ and $\boldsymbol{\lambda}^\star$, confirming that only the second constraint is tight. The optimal point and multipliers can also be found from the KKT conditions of the equality-constrained problem, which reduce to the following system of linear algebraic equations.

$$\begin{bmatrix} 2 & -1 & 2 \\ -1 & 2 & 1 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 12 \\ -3 \\ 3 \end{bmatrix}$$

Because the coefficients in the linear system are whole numbers, the solution is rational fractions. Using $\mathbf{x}^\star$ and the starting point $\mathbf{x}^0 = \left[\frac{18}{7}, -\frac{61}{7}\right]^\top$ given in §22.2.1, I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 1.

28.7.35 rnt

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) & = & (x_1+x_4)^4+(x_2+x_3)^2 \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \begin{bmatrix} 3x_1 - x_2 - 2x_3 - x_4 \\ -4x_1 + x_2 + 5x_3 + 2x_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \mathbf{b} \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-21,-49,-6,-1]^\top \quad \mathbf{x}^\star = \left[-\tfrac{1}{10},-\tfrac{3}{5},\tfrac{3}{5},\tfrac{1}{10}\right]^\top \quad \mathbf{x}^{\mathrm{H}} = [17,39,6,1]^\top$$
$$f_0(\mathbf{x}^\star) = 0 \qquad \boldsymbol{\lambda}^\star = [0,0]^\top$$

This problem has the same constraints as qp1. Because of the form of its objective function, $x_4^\star = -x_1^\star$, $x_3^\star = -x_2^\star$, and $f_0(\mathbf{x}^\star) = 0$ for all right-hand side vectors $\mathbf{b}$. This makes $\boldsymbol{\lambda}^\star = \mathbf{0}$ even though the constraints are both satisfied with equality. Using $\mathbf{x}^\star$ and the starting point $\mathbf{x}^0 = [-2,-5,0,0]^\top$ given in §22.3, I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 1.

28.7.36 grg2

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = (x_1-8)^2+x_2^2 & & \\ \text{subject to} & \tfrac{1}{20}x_1^2 + x_2 - 5 & = & 0 \end{array}$$

$$\mathbf{x}^{\mathrm{L}} = [-67.149,-32.938]^\top \quad \mathbf{x}^\star = [8.91488339968883, 1.02624269849762]^\top \quad \mathbf{x}^{\mathrm{H}} = [71.149, 42.538]$$
$$f_0(\mathbf{x}^\star) = 1.89018571124588 \qquad \lambda^\star = -2.05248539699525$$

Using $\mathbf{x}^\star$ and the starting point $\mathbf{x}^0 = [2, \frac{24}{5}]^\top$ given in §23.1.2, I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 1. The Lagrange conditions for the problem require that $\lambda^3 + 50\lambda^2 + 800\lambda + 1440 = 0$, which I solved numerically for $\lambda^\star$.

28.7.37 grg4

$$\begin{array}{lrcl} \underset{\mathbf{x}\in\mathbb{R}^4}{\text{minimize}} & f_0(\mathbf{x}) = x_1^2 + x_2 + x_3^2 + x_4 & & \\ \text{subject to} & x_1^2 + x_2 + 4x_3 + 4x_4 - 4 & = & 0 \\ & -x_1 + x_2 + 2x_3 - 2x_4^2 + 2 & = & 0 \end{array}$$

$$\begin{aligned} &\mathbf{x}^{\mathrm{L}} = [-5,-39.75208185513982,-11.65942549594963,-6.09640503216468]^\top \\ &\mathbf{x}^\star = [-0.5,-4.824791814486018, 1.534057450405037, 0.609640503216468]^\top \\ &\mathbf{x}^{\mathrm{H}} = [5, 23.75208185513982, 17.65942549594963, 6.09640503216468]^\top \\ &f_0(\mathbf{x}^\star) = -1.61181905012635 \quad \boldsymbol{\lambda}^\star = [-0.534057450405037, -0.465942549594963]^\top \end{aligned}$$

The starting point $\mathbf{x}^0 = [0,-8,3,0]^\top$ given in §23.1.2, which comes from [3, p313], happens to satisfy the constraints. Using it and $\mathbf{x}^\star$ I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure

described in §26.2.2 for case 1. The Lagrange conditions for this problem require that $16\lambda_1^3 + 83\lambda_1^2 + 116\lambda_1 + 41 = 0$, which I solved numerically for $\lambda_1^\star$; they also require $\lambda_2 = -\lambda_1 - 1$, which I used to find $\lambda_2^\star$.

28.7.38 `sqp1`

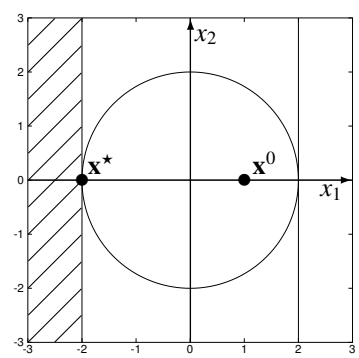
$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = e^{x_1-1} + e^{x_2+1} \\ \text{subject to} & x_1^2 + x_2^2 - 1 \;=\; 0 \end{array}$$

$$\begin{array}{l} \mathbf{x}^{\mathrm{L}} = [-8.36709035275112, -18.64716470209894]^\top \\ \mathbf{x}^\star = [-0.263290964724888, -0.964716470209894]^\top \\ \mathbf{x}^{\mathrm{H}} = [6.36709035275112, 20.64716470209894]^\top \\ f_0(\mathbf{x}^\star) = 1.31863544493956 \quad \lambda^\star = 0.536900432125476 \end{array}$$

Using $\mathbf{x}^\star$ and the starting point $\mathbf{x}^0 = [-1, 1]^\top$ given in §23.2.0, I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 1.

28.7.39 `incon`

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} & f_0(\mathbf{x}) = & x_1^2 + x_2^2 \\ \text{subject to} & x_1 - 1 & \leq \; 0 \\ & -x_1^2 + 4 & \leq \; 0 \end{array}$$



$$\mathbf{x}^{\mathrm{L}} = [-29, -20]^\top \quad \mathbf{x}^\star = [-2, 0]^\top \quad \mathbf{x}^{\mathrm{H}} = [31, 20]^\top$$

$$f_0(\mathbf{x}^\star) = 4 \quad \boldsymbol{\lambda}^\star = [0, 1]^\top$$

The constraints of this problem come from [5, p535]. Using $\mathbf{x}^\star$ and the starting point $\mathbf{x}^0 = [1, 0]^\top$ given in §23.2.4, I found $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ by the procedure described in §26.2.2 for case 1. To find $\boldsymbol{\lambda}^\star$ I solved the KKT conditions for the problem.

28.7.40 `egg`

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}} \;\; f_0(\mathbf{x}) = e^{(x_1-2)^2}\Gamma(x_2) \qquad \text{where} \qquad \Gamma(t) = \int_0^\infty y^{t-1}e^{-y}dy$$

$$\mathbf{x}^\star = [2, 1.46163214498002]^\top \qquad f_0^\star = 0.885603194410889$$

To determine $x_2^\star$ with $x_1^\star \equiv 2$, I used `gradcd.m` and bisection to find the zero of $\partial\Gamma(2, x_2)/\partial x_2$.

28.7.41 `big`

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & f_0(\mathbf{x}) = \sum_{j=1}^{n} a_j(x_j-1)^2 \\ \text{subject to} \quad & \min\left(\frac{1}{a_j}, a_j\right) - x_j \le 0, \quad j = 1\ldots n \\ & x_j - \max\left(\frac{1}{a_j}, a_j\right) \le 0, \quad j = 1\ldots n. \end{aligned}$$

for $a = [\ \ 2, 3]^\top$: $\mathbf{x}^{\mathrm{L}} = \left[\frac{1}{2}, \frac{1}{3}\right]^\top$ $\mathbf{x}^\star = [1, 1]^\top$ $\mathbf{x}^{\mathrm{H}} = [2, 3]^\top$ $f_0(\mathbf{x}^\star) = 0$ $\boldsymbol{\lambda}^\star = [0, 0, 0, 0]^\top$

for $a = [-3, 3]^\top$: $\mathbf{x}^{\mathrm{L}} = \left[-3, \frac{1}{3}\right]^\top$ $\mathbf{x}^\star = [-3, 1]^\top$ $\mathbf{x}^{\mathrm{H}} = \left[-\frac{1}{3}, 3\right]^\top$ $f_0(\mathbf{x}^\star) = -48$ $\boldsymbol{\lambda}^\star = [-16, 0, 0, 0]^\top$

In general,

$$x_j^\star = \begin{cases} 1 & \text{if } a_j > 0 \\ \min\left(a_j, 1/a_j\right) & \text{if } a_j < 0. \end{cases}$$

28.8 Integer Nonlinear Program Used in the Text

For the single named integer nonlinear programming example, I have given below an algebraic statement in standard form, bounds $\mathbf{x}^{\mathrm{L}}$ and $\mathbf{x}^{\mathrm{H}}$ on the variables, and the optimal integer points.

28.8.1 `inlp`

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{Z}^2}{\text{minimize}} \quad & f_0(\mathbf{x}) = (x_1-4)^2 + (x_2 - 2\tfrac{1}{2})^2 \\ \text{subject to} \quad & (x_1-2)^2 + (x_2-4) \le 0 \\ & -x_1 \le 0 \text{ and integer} \\ & -x_2 \le 0 \text{ and integer} \end{aligned}$$

$$\mathbf{x}^{\mathrm{L}} = [0, 0]^\top \quad \mathbf{x}_{\mathrm{IP}}^{\star 1} = [3, 2]^\top \quad f_0(\mathbf{x}_{\mathrm{IP}}^{\star 1}) = \tfrac{5}{4} = f_0(\mathbf{x}_{\mathrm{IP}}^{\star 2}) \quad \mathbf{x}_{\mathrm{IP}}^{\star 2} = [3, 3]^\top \quad \mathbf{x}^{\mathrm{H}} = [4, 4]^\top$$

28.9 Exercises

A few of these problems assume a knowledge of material from other Chapters.

28.9.1 [E] This Chapter includes some background information about undergraduate mathematics, numerical methods, and computer programming. (a) Is the survey that it provides of these subjects exhaustive, superficial, or focused on specific needs? Explain. (b) Where can you find additional background information on these subjects? (c) What else is in this Chapter?

28.9.2 [E] What topics in calculus have I assumed you know quite well, so that they do not need to be reviewed in this Chapter?

28.9.3 [E] Suppose that $f(x)$ is a twice-differentiable scalar function of the scalar variable x. (a) Where might we find its local minima? Why? (b) How can its second derivative be used to classify the points where its first derivative is zero? (c) Show that in the example of §28.1.1 $y'(x_d) = 0$. (d) In the example we classified point d as a local minimum by computing $y''(x_d) \approx 3117.6 > 0$. Explain how inspection of the graph reveals that the slope of the curve is increasing at that point.

28.9.4 [P] The function $f(x) = 3x^3 + 2x^2 - x + 1$ is twice-differentiable. (a) Find its one local minimum and its one local maximum. (b) Is $x = -\frac{2}{9}$ an inflection point?

28.9.5 [P] Suppose we want to approximate the function $f(x) = \sin(x)$ in the vicinity of $x = \pi$. (a) Construct a linear approximation $T_1(x;\pi)$. (b) Construct a quadratic approximation $T_2(x;\pi)$. (c) Write a MATLAB program that plots on one set of axes $e_1 = f(x) - T_1(x;\pi)$ and $e_2 = f(x) - T_2(x;\pi)$ over the interval $x \in [0, 2\pi]$. (d) Compute the area between the curves of $f(x)$ and $T_1(x;\pi)$ and the area between the curves of $f(x)$ and $T_2(x;\pi)$ over that interval. Over what range of x do you think these approximations might actually be useful? (e) Construct the Taylor series expansion $T_\infty(x;\pi)$ of $f(x)$. Do you recognize this as the power series for $\sin(x)$?

28.9.6 [H] Suppose that $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{Q}\mathbf{x}$ where $\mathbf{x} \in \mathbb{R}^2$ and

$$\mathbf{Q} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

(a) Compute $\nabla f(\mathbf{x})$. (b) Verify that the components of your answer are the partial derivatives of $f(x) = x_1^2 + 5x_1x_2 + 4x_2^2$.

28.9.7 [H] Show that $\nabla(\mathbf{x}^\top\mathbf{x}) = 2\nabla\left(\sqrt{\mathbf{x}^\top\mathbf{x}}\right)$.

28.9.8 [E] What topics in linear algebra have I assumed you know quite well, so that they do not need to be reviewed in this Chapter?

28.9.9 [E] What does it mean to say that two matrices are *conformable* (a) for addition? (b) for multiplication?

28.9.10 [H] For the matrices $\mathbf{A}$ and $\mathbf{B}$ below [147, p16] compute the matrix products (a) $\mathbf{AB}$; (b) $\mathbf{BA}$.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix}$$

28.9.11 [H] What properties of two matrices $\mathbf{A}$ and $\mathbf{B}$ are sufficient to ensure that $\mathbf{AB} = \mathbf{BA}$?

28.9.12 [P] The following system of linear algebraic equations has a unique solution.

$$\begin{aligned} x_1 + 3x_2 + 5x_3 &= -2 \\ 2x_1 \qquad - 4x_3 &= 7 \\ -6x_1 + x_2 - 8x_3 &= 0 \end{aligned}$$

(a) Write the system in matrix-vector form. (b) Use matrix multiplication to prove that $x_1 = \frac{195}{134}$, $x_2 = \frac{37}{67}$, and $x_3 = -\frac{137}{134}$ solve the linear system. (c) Use the MATLAB backslash operator \ to obtain this solution.

28.9.13 [H] Compute the transpose of each matrix below.

$$\text{(a)} \begin{bmatrix} 4.2 & -9.7 & 3.1 & 5.0 \\ 2.1 & 6.6 & -1.7 & 8.3 \end{bmatrix} \qquad \text{(b)} \begin{bmatrix} 2 & 4 & 6 \\ 4 & 5 & 1 \\ 6 & 1 & 7 \end{bmatrix} \qquad \text{(c)} \ [1, 2, 3]^\top$$

28.9.14 [E] What makes a matrix (a) symmetric? (b) diagonal? (c) the identity matrix?

28.9.15 [H] If $\mathbf{a}^\top = [1, 2, 3]$ and $\mathbf{b}^\top = [4, 5, 6]$ compute (a) the *inner product* $\mathbf{a}^\top\mathbf{b}$; (b) the inner product $\mathbf{b}^\top\mathbf{a}$; (c) the *outer product* $\mathbf{a}\mathbf{b}^\top$; (d) the outer product $\mathbf{b}\mathbf{a}^\top$. (d) What sort of product is $\mathbf{ab}$?

28.9.16 [H] Why is the outer product of two vectors a matrix of rank one? Why is the outer product of a vector with itself a symmetric matrix? When is it an identity matrix?

28.9.17 [H] What is the dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ if the angle between them is (a) 0°; (b) 90°.

28.9.18 [H] If $\mathbf{x} \in \mathbb{R}^2$ has length 3.5, $\mathbf{y} \in \mathbb{R}^2$ has length 5.2, and $\mathbf{x}^\top\mathbf{y} = 12$, what must be the angle θ between the two vectors?

28.9.19 [H] Are the vectors $\mathbf{v}_1 = [1, 2, 3]^\top$ and $\mathbf{v}_2 = [4, 5, 6]^\top$ *linearly independent*? If so, prove it; if not, what must c_1 and c_2 be so that $c_1\mathbf{v}_1 + c_2\mathbf{v}_2 = \mathbf{0}$?

28.9.20 [E] Why can't a set of vectors that includes the zero vector be linearly independent?

28.9.21 [H] Show that if $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are any three vectors in $\mathbb{R}^2$, then scalars a and b can be found such that $a\mathbf{x} + b\mathbf{y} = \mathbf{z}$. What does this imply about the linear independence of the three vectors?

28.9.22 [P] This matrix has three rows, but its rank is only 2.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

(a) Use the MATLAB command `rank(A)` to confirm that its rank is 2. (b) Find scalars a and b such that $a[1, 2, 3] + b[4, 5, 6] = [7, 8, 9]$. (c) What is implied by the fact that this is possible?

28.9.23 [H] Prove that if $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ then $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$.

28.9.24 [H] Can a singular matrix have an inverse? If so, write down a singular matrix that has an inverse; if not, write down a singular matrix and show that it cannot have an inverse.

28.9.25 [P] Consider the following matrix.

$$\mathbf{A} = \begin{bmatrix} -\frac{2}{9} & \frac{5}{9} & -\frac{1}{9} \\ \frac{4}{9} & -\frac{1}{9} & \frac{2}{9} \\ -\frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

(a) Find the *cofactor matrix* $\mathbf{C}$ corresponding to $\mathbf{A}$. (b) Find the *adjoint matrix* corresponding to $\mathbf{A}$. (c) Find the determinant of $\mathbf{A}$. (d) Find the inverse $\mathbf{A}^{-1}$. (e) Confirm that you have found the inverse by showing that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. (f) Write a MATLAB routine `adj(A)` that returns the adjoint matrix corresponding to `A`.

28.9.26 [P] The inverse of a nonsingular 2×2 matrix $\mathbf{B}$ can be found from a simple formula. (a) State the formula. (b) Use the formula to find the inverse of

$$\mathbf{B} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

(c) Write a MATLAB routine `twoinv(B)` that uses the formula to compute the inverse of its 2×2 matrix argument `B`. What does your routine do if `B` is singular?

28.9.27 [H] In §28.2.6 I stated several identities concerning matrix inverses, which assume that each matrix being inverted is square and nonsingular. Which of them make sense only if the matrices $\mathbf{A}$ and $\mathbf{B}$ are *both* square?

28.9.28 [E] What notation is used in this book to represent the transpose of an inverse matrix? Why can the same notation be used for the inverse of a matrix transpose?

28.9.29 [P] In §28.2.6, I claimed that $(\mathbf{A}^{\top})^{-1} = (\mathbf{A}^{-1})^{\top}$. (a) Use MATLAB to confirm this claim for several random square matrices of different sizes. (b) Prove that the claim is true in general.

28.9.30 [P] In §28.2.6 I claimed that $(\mathbf{A}\mathbf{B})^{-\top} = \mathbf{A}^{-\top}\mathbf{B}^{-\top}$. (a) Use MATLAB to confirm this claim for several random square matrices of different sizes. (b) Prove that the claim is true in general.

28.9.31 [H] Prove that $(\mathbf{A}\mathbf{B})^{\top} = \mathbf{B}^{\top}\mathbf{A}^{\top}$.

28.9.32 [H] Prove that $(\mathbf{A}\mathbf{B})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.

28.9.33 [E] What ideas from numerical computing have I assumed you know quite well, so that they do not need to be reviewed in this Chapter?

28.9.34 [E] Describe one mathematical problem of practical importance that does not have a closed-form analytic solution.

28.9.35 [E] How does a computer program that implements an iterative algorithm repeat the sequence of arithmetic and logical operations until a sufficiently precise answer is obtained?

28.9.36 [E] What are *floating-point* calculations? Are they exact?

28.9.37 [E] Describe a class of problems that can be solved using more than one numerical algorithm. Are the algorithms equally fast? Are they equally accurate? Are they equally likely to give the right answer?

28.9.38 [H] If $f(a) < 0$ and $f(b) > 0$, what property must $f(x)$ have to ensure that its value is zero at some $x \in (a, b)$? What property must $f(x)$ have to ensure that its value is zero at *exactly* one $x \in (a, b)$?

28.9.39 [E] Describe in words the idea of the *bisection* algorithm for finding a root of $f(x) = 0$.

28.9.40 [E] What is a *convergence test,* and why might we use one?

28.9.41 [E] The product $f_L \times f_k$ is negative if f_L and f_k are of opposite sign or positive if they are of the same sign. What happens in the bisection algorithm if one or the other value is exactly zero?

28.9.42 [E] Describe in words the idea of *Newton's method* for solving $f(x) = 0$. What are its advantages over bisection? What are its drawbacks when compared to bisection?

28.9.43 [E] What happens if you start Newton's method too far from the root you are trying to find?

28.9.44 [E] How does MATLAB store integers such as loop counters and array indices?

28.9.45 [E] Why are floating-point calculations usually not perfectly precise? What is the definition of *machine epsilon,* and what is its numerical value? What is a `NaN`, and how can they be avoided?

28.9.46 [P] Write a MATLAB program that approximates the value of machine epsilon.

28.9.47 [P] Write a MATLAB program that generates a `NaN`.

28.9.48 [E] What experience with numerical computing did I assume you had as you began reading this book? What level of fluency with numerical computation do I hope you will have reached by the time you finish reading it?

28.9.49 [E] What MATLAB control structures have I used in this book? Where are `continue` and `break` useful, and what is the difference between them?

28.9.50 [E] What are the two forms of the MATLAB `if` statement, and in what circumstances have I used each?

28.9.51 [E] Explain how the MATLAB `switch` statement works. In the code excerpt from `sqp1.m` reproduced in §28.4.1, what happens if the routine is entered with `i=2`?

28.9.52 [H] Many programming environments provide a small number of functions that are built-in and thus always present (such as square root) and expect other functions to be accessed only after their individual definitions have been extracted from a library specified by the programmer. In base MATLAB and its work-alike Octave, a vast legion of functions are built-in. What are the advantages of this design choice? Does it have any drawbacks?

28.9.53 [E] How can you tell whether a name is already in use for a MATLAB function or variable? What happens if you use one of those many names to mean something else?

28.9.54 [E] In the MATLAB programs listed in this book, what does the variable `i` usually denote? What is its default meaning in MATLAB?

28.9.55 [H] Two different schemes are described in §28.4.3 for coding the implementation of an iterative algorithm. Explain how the first scheme works if (`a`) convergence is attained at $\mathbf{x}^0$; (`b`) convergence is attained at a later iteration but before the iteration limit is met; (`c`) the iteration limit is met without convergence being attained. What values are returned for `xstar` and `k` in each case?

28.9.56 [H] Two different schemes are described in §28.4.3 for coding the implementation of an iterative algorithm. (`a`) Why is the first scheme ill-suited for repeated invocation in a loop to perform one iteration at a time? Explain how the second scheme works if (`b`) convergence is attained at $\mathbf{x}^0$; (`c`) convergence is attained at a later iteration but before the iteration limit is met; (`d`) the iteration limit is met without convergence being attained. What values are returned for `xstar` and `k` in each case?

28.9.57 [P] The bisection algorithm described in §28.3.1 and the Newton's method algorithm described in §28.3.2 both increment k. (`a`) Do they count iterations in either of the ways discussed in §28.4.3? (`b`) Reimplement the bisection algorithm as a serially-reusable MATLAB function `[xstar,kp]=bisect(fcn,xh,xl,epsx,epsf,kmax)` that can be invoked in a loop to perform one iteration of the algorithm at a time. (`c`) Write a program to invoke `bisect` repeatedly in a loop and use it to print out each iterate x^k produced by the algorithm.

28.9.58 [E] Each linear program description in §28.5 gives the optimal objective value for the *primal* problem. How can you get the optimal objective value for the *dual*?

28.9.59 [H] If a nonsingular system of linear algebraic equations has coefficients that are whole numbers, the components of its solution vector are rational fractions. (`a`) Why? (`b`) Given the decimal expansion of a rational fraction, how can you find the rational fraction?

28.9.60 [H] Explain why the `rnt` problem (see §28.7.35) has $f_0(\mathbf{x}^\star) = 0$ for all right-hand side vectors $\mathbf{b}$. Why does this make $\boldsymbol{\lambda}^\star = [0, 0]^\top$?

28.9.61 [P] The structure of the `big` problem allowed us to deduce in §25.7.4 that

$$x_j^\star = \begin{cases} 1 & \text{if } a_j > 0 \\ \min\left(a_j, 1/a_j\right) & \text{if } a_j < 0. \end{cases}$$

(a) What are the corresponding variable bounds? (b) What are the corresponding KKT multipliers?

29

Bibliography

If you encountered a citation in the text and want to look up the reference, find the entry with the given number. For example, the citation [1] refers to the first entry below, the textbook by Bazaraa et al.

If you have a particular work in mind and want to check whether it is used as a reference or find the number by which it is cited, scan for its author or title. To make this easy, the entries are sorted into three categories and are listed alphabetically by *author's name* within each category. Documents authored by an organization, or containing no attribution of authorship, are alphabetized by the *most significant words* in the title.

Some of the entries include annotations in *slanting type.* The internet addresses that are given in a few of the entries (and elsewhere in the book) were valid when I used them but might have changed since then.

29.1 Suggested Reading

This category lists basic works that are relevant in a general way to mathematical programming, and which I recommend in their entirety for further study.

[1] **Bazaraa, Mokhtar S.**, **Sherali, Hanif D.**, and **Shetty, C. M.**, *Nonlinear Programming: Theory and Algorithms,* Third Edition, John Wiley & Sons, 2006. *The indispensable reference on nonlinear programming theory, long on convex analysis and thus not easy reading but well worth the effort of careful study. The typesetting of this edition leaves much to be desired.*

[2] **Bertsekas, Dimitri P.,** *Nonlinear Programming,* Third Edition, Athena Scientific, 2016. *A "comprehensive, and rigorous account of nonlinear programming... up to date with recent research progress..." Also not for the faint of heart, but breathtaking in scope and informed by an awareness of engineering applications.*

[3] **Ecker, J. G.** and **Kupferschmid, Michael**, *Introduction to Operations Research,* Reprint Edition, Krieger Publishing Company, 2004. *An easy introduction to Chapters 1-7, 11, 15-16, and 24 of the present book, plus chapters on queueing, inventory, and simulation. The present book's treatment of linear programming is based on the approach taken in this book, which was originally developed by Joe Ecker.*

[4] **Griva, Igor**, **Nash, Stephen G.**, and **Sofer, Ariela**, *Linear and Nonlinear Optimization,* Second Edition, SIAM, 2009. *A thorough survey favored by students, acces-*

sible and a pleasure to read, with many numerical examples. This edition has the best cover art ever.

[5] **Nocedal, Jorge** and **Wright, Stephen J.**, *Numerical Optimization,* Second Edition, Springer, 2006. *A widely-taught and authoritative survey, also very thorough, comparable in rigor to [2] but easier to read.*

29.2 Technical References

This category lists other references about optimization and related technical subjects. Many of these books and papers are also well worth reading in their entirety, but they are cited in the text only as authority for specific claims made there or as sources of additional information about particular topics.

[6] **Abramowitz, Milton** and **Stegun, Irene A.**, *Handbook of Mathematical Functions,* Dover, 1970.

[7] **Abu-Mostafa, Yaser S.**, **Magdon-Ismail, Malik**, and **Lin, Hsuan-Tien**, *Learning From Data: A Short Course,* AMLbook.com, 2012.

[8] **Apostol, Tom M.**, *Mathematical Analysis,* Second Edition, Addison-Wesley, 1975.

[9] **Audet, Charles**, **Hansen, Pierre**, and **Messine, Frédéric**, "The Largest Small Octagon," *Journal of Combinatorial Theory,* Series A, 98 46-59, 2002. *Constraints 4 and 5 in the statement of the nonlinear program contain sign reversals in six terms, which I have corrected in Exercise 25.8.11. For a class project in 2004, Zheng Yuan used symmetry arguments to generalize the results of this paper and find the decagon of maximum area.*

[10] **Balinski, M. L.**, "A Competitive (Dual) Simplex Method for the Assignment Problem," *Mathematical Programming* 34: 125-141, 1986.

[11] **Beale, E. M. L.**, "Cycling in the Dual Simplex Algorithm," *Naval Research Logisics Quarterly* 2:4 269-276, December 1955. *The example discussed in §4.5 and attributed by many authors to Beale is actually the dual of the problem he suggests in this paper.*

[12] **Beightler, Charles S.** and **Phillips, Donald T.**, *Applied Geometric Programming,* John Wiley & Sons, 1976.

[13] **Bellman, Richard**, *Dynamic Programming,* Princeton University Press, 1957.

[14] **Bennett, Kristin P.**, *Classnotes,* Computational Optimization `MATP-4820/6610`, Rensselaer Polytechnic Institute, spring 2015.

[15] **Bertsekas, Dimitri P.** and **Tseng, Paul**, "The RELAX Codes for Linear Minimum Cost Network Flow Problems," *Annals of Operations Research* 13:1 125-190, December 1988.

[16] **Bland, Robert G.**, "New Finite Pivoting Rules for the Simplex Method," *Mathematics of Operations Research* 2:2, May 1977.

[17] **Boyd, Stephen**, **Parikh, Neal**, **Chu, Eric**, **Peleato, Borja**, and **Eckstein, Jonathan**, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning* 3:1 1-122, 2011.

[18] **Bracken, Jerome** and **McCormick, Garth P.**, *Selected Applications of Nonlinear Programming,* John Wiley & Sons, 1968.

[19] **Branin, F. H.**, "Widely Convergent Method for Finding Multiple Solutions of Simultaneous Nonlinear Equations," *IBM Journal of Research and Development* 16: 504-522, 1972.

[20] **Burden, Richard L.**, **Faires, J. Douglas**, and **Reynolds, Albert C.**, *Numerical Analysis,* Second Edition, Prindle, Weber & Schmidt, 1981.

[21] **Calingaert, Peter**, *Assemblers, Compilers, and Program Translation,* Computer Science Press, 1979.

[22] **Cauchy, A.**, "Méthode générale pour la résolution des systèmes d'équations simultanées," *Compte Rendu à l'Académie des Sciences* 25 536-538, 18 October 1847.

[23] **Cecchini, Mark**, **Ecker, Joseph**, **Kupferschmid, Michael**, and **Leitch, Robert**, "Solving Nonlinear Principal-Agent Problems using Bilevel Programming," *European Journal of Operational Research* 230:2 364-373, 2013.

[24] **Chen, S.**, **Donoho, D. L.**, and **Saunders, M. A.**, "Atomic Decomposition by Basis Pursuit," *SIAM Journal of Scientific Computing* 20:1 33-61, 1999.

[25] **Charnes, A.** and **Cooper, W. W.**, *Management Models and Industrial Applications of Linear Programming,* two volumes, John Wiley & Sons, 1961. *This iconic tome from the dawn of mathematical programming could serve to define the term "venerable." Its gentle introduction, assuming only high-school algebra as prerequisite, might strike the jaded modern as childlike in its earnest simplicity, but students who want to start learning the subject at its very beginning will find here much more than charm.*

[26] **Chatterjee, Samprit** and **Price, Bertram**, *Regression Analysis by Example,* John Wiley, 1977. *In their equation (8.12) the diagonal terms should be multiplied by* $r_{11} \ldots r_{pp}$. *They assume regression data have been transformed to make* $\beta_0 = 0$.

[27] **Cheney, Margaret** and **Borden, Brett**, *Fundamentals of Radar Imaging,* SIAM, 2009.

[28] **Colville, A. R.**, *A Comparative Study on Nonlinear Programming Codes,* New York Scientific Center Report 320-2949, International Business Machines, 1968.

[29] **Conley, William,** *Computer Optimization Techniques,* Petrocelli Books, 1980. *A gallant defense of Monte Carlo optimization.*

[30] **Conte, S. D.** and **de Boor, Carl**, *Elementary Numerical Analysis: An Algorithmic Approach,* Third Edition, McGraw-Hill, 1980.

[31] **Cornwell, L. W.**, **Hutchison, P. A.**, **Minkoff, M.**, and **Schultz, H. K.**, *Test Problems for Constrained Nonlinear Mathematical Programming Algorithms,* Technical Memorandum No. 320, Applied Mathematics Division, Argonne National Laboratory, 1978.

[32] **Courant, R.**, "Variational methods for the solution of problems of equilibrium and vibrations," *Bulletin of the American Mathematical Society* 49: 1-23, 1943.

[33] **Covey, David**, *Parallel Ellipsoid Methods for Nonlinear Programming,* PhD Thesis, Rensselaer Polytechnic Institute, May 1989.

[34] **Crowder, Harlan**, **Dembo, Ron S.**, and **Mulvey, John M.**, "On Reporting Computational Experiments with Mathematical Software," *ACM Transactions on Mathematical Software* 5:2 192-203, June 1979.

[35] **Dantzig, George B.**, *Linear Programming and Extensions,* Princeton University Press, 1963. *The foundational text of linear programming, including a detailed history of the discipline.*

[36] **Dantzig, George B.**, "Remarks on the Occasion of the Bicentennial Conference on Mathematical Programming," *NBS Special Publication 502: Computers and Mathematical Programming* 1-3, February 1978.

[37] **Dantzig, George B.**, "Khachian's Algorithm: a Comment," SIAM News 13 1,4, 1980.

[38] **Dantzig, George B.**, **Orden, A.**, and **Wolfe, Philip**, "The Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints," *Pacific Journal of Mathematics* 5 183-195, 1955.

[39] **Davenport, Mark A.**, **Duarte, Marco F.**, **Eldar, Yonina C.**, and **Kutyniok, Gitta**, "Introduction to Compressed Sensing," Chapter 1 of *Compressed Sensing: Theory and Applications,* Cambridge University Press, 2012.

[40] **Davidon, W. C.**, *Variable metric method for minimization*, Technical Report ANL-5990 (revised), Argonne National Laboratory, 1959.

[41] **Dembo, R. S.,** "GGP — A Program for Solving Generalized Geometric Programming Problems — User's Manual," Chemical Engineering Report 72/59, Technion, 1972.

[42] **Dembo, R. S.** and **Mulvey, J. M**, *On the Analysis and Comparison of Mathematical Programming Algorithms and Software,* Harvard Business School HBS 76-19, 1976, later published in *Computers and Mathematical Programming,* Special Publication #502, National Bureau of Standards, 1978.

[43] **Dempe, S.**, *Foundations of Bilevel Programming,* Kluwer, 2002.

[44] **Dolan, Elizabeth D.** and **Moré, Jorge J.**, "Benchmarking optimization software with performance profiles," *Mathematical Programming* A 91 201-213, 2002.

[45] **Donoho, David L.**, "Compressed Sensing," *IEEE Transactions on Information Theory* 52:4, April 2006.

[46] **Duffin, Richard J.**, **Peterson, Elmor L.**, and **Zener, Clarence**, *Geometric Programming — Theory and Application,* John Wiley & Sons, 1967.

[47] **Dziuban, Stephen T.**, *Ellipsoid Algorithm Variants in Nonlinear Programming,* PhD Thesis, Rensselaer Polytechnic Institute, August 1983.

[48] **Eason, E. D.** and **Fenton, R. G.**, *Testing and Evaluation of Numerical Methods for Design Optimization,* Technical Publication 7204, Department of Mechanical Engineering, University of Toronto, September 1972.

[49] **Eason, Ernest D.** and **Padmanaban, Jeya**, "Engineering Problems for Evaluating Nonlinear Programming Codes," XI International Symposium on Mathematical Programming, Bonn, Germany, 23-27 August 1982. *They introduce the characterization of problems as class–1 or class–2; to avoid confusion with other uses of the word "class" I have referred to these categories as type–1 and type–2.*

[50] **Eaton, John W.**, **Bateman, David**, and **Hauberg, Søren**, *GNU Octave*, Edition 3 for Octave version 3.6.1, Free Software Foundation, 2011.

[51] **Ech-cherif, A.**, **Ecker, J. G.**, and **Kupferschmid, Michael**, "A Numerical Investigation of Rank-Two Ellipsoid Algorithms for Nonlinear Programming," *Mathematical Programming* 43 87-95, 1989.

[52] **Ecker, Joseph G.** and **Kupferschmid, Michael**, "A computational comparison of the ellipsoid algorithm with several nonlinear programming algorithms," *SIAM Journal on Control and Optimization* 23 657-674 1985.

[53] **Ecker, Joseph G.**, *Classnotes,* Computational Optimization `MATP-4820/6610`, Rensselaer Polytechnic Institute, spring 2005.

[54] **Eckstein, J.** and **Bertsekas, D. P.**, "On the Douglas-Rachford Splitting Method and the Proximal Point Algorithm for Maximal Monotone Operators," *Mathematical Programming* 55: 293-318, 1992.

[55] **Edmonds, Jack**, "Paths, Trees, and Flowers," *Canadian Journal of Mathematics* 17 449-467, 1965; also "Optimum Branchings," *Journal of Research of the National Bureau of Standards* 71B 233-240, 1967.

[56] **Fang, Shu-Chern** and **Puthenpura, Sarat**, *Linear Optimization and Extensions: Theory and Algorithms,* Prentice-Hall, 1993.

[57] **Fiacco, Anthony V.** and **McCormick, Garth P.**, *Nonlinear Programming: Sequential Unconstrained Minimization Techniques,* John Wiley & Sons, 1968. *This book provides an extensive historical survey as its §1.2.*

[58] **Fisher, Marshall L.**, "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science* 50:12 supplement 1861-1871, December 2004.

[59] **Fletcher, R.**, *Practical Methods of Optimization: Volume 1, Unconstrained Optimization,* John Wiley & Sons, 1980.

[60] **Forsythe, George E.**, **Malcolm, Michael A.**, and **Moler, Cleve B.**, *Computer Methods for Mathematical Computations,* Prentice-Hall, 1977.

[61] **Fourer, Robert**, **Gay, David M.**, and **Kernighan, Brian W.**, *AMPL: A Modeling Language for Mathematical Programming,* `www.ampl.com/BOOK/download.html`, 2003.

[62] **Garfinkel, Robert S.** and **Nemhauser, George L.**, *Integer Programming,* John Wiley & Sons, 1972.

[63] **Gass, Saul I.**, *Linear Programming: Methods and Applications,* Fifth Edition, McGraw-Hill, 1985.

[64] **Geoffrion, A. M.**, "Integer Programming by Implicit Enumeration and Balas' Method," *SIAM Review* 7:2 178-190, April 1967.

[65] **Glassey, C. R.** and **Gupta, V. K.**, "A Linear Programming Analysis of Paper Recycling," *Management Science* 20: 392-408, 1974.

[66] **Goldstein, A. A.** and **Price, J. F.**, "On Descent from Local Minima," *Mathematics of Computation* 25:115 569-574, July 1971.

[67] **Golub, Gene H.** and **Van Loan, Charles F.**, *Matrix Computations,* Second Edition, Johns Hopkins University Press, 1989.

[68] **Gould, Nicholas** and **Scott, Jennifer**, "A Note on Performance Profiles for Benchmarking Software," *ACM Transactions on Mathematical Software* 43:2, Article 15, August 2016.

[69] **Gradshteyn, I. S.** and **Ryzhik, I. M.**, *Table of Integrals, Series, and Products,* Academic Press, 1965.

[70] **Greenberg, Harold**, *Integer Programming,* Academic Press, 1971.

[71] **Greenberg, Harvey J.**, *Myths and Counterexamples in Linear Programming,* `hjgreenberg@gmail.com`, 20 Feb 2010.

[72] **Greene, Daniel H., and Knuth, Donald E.**, *Mathematics for the Analysis of Algorithms,* Second Edition, Birkhäuser, 1982.

[73] **Grötschel, M.**, **Lovász, L.**, and **Schrijver, A.**, *Geometric Algorithms and Combinatorial Optimization,* Springer, 1985.

[74] **Hadley, G.**, *Nonlinear and Dynamic Programming,* Addison-Wesley, 1964.

[75] **Hayes, Brian**, "The Best Bits," Computing Science, *American Scientist* 97: 276-280, July-August 2009.

[76] **Hearn, Donald W.** and **Randolph, W. D.**, *Dual Approaches to Quadratically Constrained Quadratic Programming,* Research Report 73-15, Industrial and Systems Engineering Department, University of Florida, 1973.

[77] **Heath, Michael T.**, *Scientific Computing: An Introductory Survey,* McGraw-Hill, 1996.

[78] **Hestenes, Magnus R.**, *Optimization Theory: The Finite Dimensional Case,* John Wiley, 1975. *Hestenes was William Karush's PhD thesis advisor.*

[79] **Hillier, Frederick S.** and **Lieberman, Gerald J.**, *Introduction to Operations Research,* Holden-Day, 1980.

[80] **Himmelblau, David M.**, *Applied Nonlinear Programming,* McGraw-Hill, 1972.

[81] **Hock, W.** and **Schittkowski, K.**, *Test Examples for Nonlinear Programming Codes,* Springer-Verlag, New York, 1981.

[82] **Hoffman, A. J.**, "Cycling in the Simplex Algorithm," *Report No. 2974,* National Bureau of Standards, 1953.

[83] **Horowitz, Ellis** and **Sahni, Sartaj**, *Fundamentals of Data Structures,* Computer Science Press, 1976.

[84] *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, 12 August 1985.

[85] **Jackson, Richard H. F.**, **Boggs, Paul T.**, **Nash, Stephen G.**, and **Powell, Susan**, "Guidelines for reporting results of computational experiments: report of the ad hoc committee," *Mathematical Programming* 49 413-426 1990/1991.

[86] **Jaderberg, Max**, **Czarnecki, Wojciech M.**, **Dunning, Iain**, **Marris, Luke**, **Lever, Guy**, **Castañeda, Antonio Garcia**, **Beattie, Charles**, **Rabinowitz, Neil C.**, **Morcos, Ari S.**, **Ruderman, Avraham**, **Sonnerat, Nicholas**, **Green, Tim**, **Deason, Louise**, **Leibo, Joel Z.**, **Silver, David**, **Hassabis, Demis**, **Kavukcuoglu, Koray**, and **Graepel, Theore**, "Human-level performance in 3D multiplayer games with population-based reinforcement learning," *Science* 364:6443 859-864, 31 May 2019.

[87] **Jennings, Alan,** *Matrix Computation for Engineers and Scientists,* John Wiley, 1977.

[88] **Johnson, Eric C.**, *A Parallel Decomposition Algorithm for Constrained Nonlinear Optimization,* PhD Thesis, Rensselaer Polytechnic Institute, July 2001.

[89] **Karmarkar, N.**, "A New Polynomial-Time Algorithm for Linear Programming," *Combinatorica* 4: 373-395, 1984.

[90] **Karush, William,** *Minima of Functions of Several Variables with Inequalities as Side Conditions,* S.M. Thesis, Department of Mathematics, University of Chicago, December 1939.

[91] **Kelly, Terrence K.** and **Kupferschmid, Michael**, "Numerical Verification of Second-Order Sufficiency Conditions for Nonlinear Programming," Classroom Notes, *SIAM Review* 40:2 310-314, June 1998.

[92] **Khachiyan, L. G.**, "A Polynomial Algorithm in Linear Programming," *Doklady Akademii Nauk SSSR* 244 1093-1096, 1979 *translated from the Russian in Soviet Mathematics Doklady* 20 191-194, 1979.

[93] **Klee, Victor** and **Minty, George J.**, "How Good is the Simplex Method?" *Inequalities-III* 159-175, Academic Press, 1972.

[94] **Knuth, Donald E.**, *The Art of Computer Programming: Volume 1/Fundamental Algorithms,* Second Edition, Addison-Wesley, 1973.

[95] **Knuth, Donald E.**, *The Art of Computer Programming: Volume 3/Sorting and Searching,* Second Printing, Addison-Wesley, 1973.

[96] **Kochan, Stephen G.** and **Wood, Patrick H.**, *Unix Shell Programming*, Revised Edition, Hayden Books, 1990.

[97] **Kuhn, H. W.** and **Tucker, A. W.**, "Nonlinear Programming," *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability* 481-492, University of California Press, 1951.

[98] **Kupferschmid, Michael**, *An Ellipsoid Algorithm for Convex Programming,* PhD Thesis, Rensselaer Polytechnic Institute, July 1981. *I apologize for the adolescent pomposity, clumsy mechanics, and numerous typographical errors that pervade this thesis.*

[99] **Kupferschmid, Michael** and **Ecker, J. G.**, "A Note on the Solution of Nonlinear Programming Problems with Imprecise Function and Gradient Values," *Mathematical Programming Study* 31 129-138, 1987.

[100] **Kupferschmid, Michael**, *Classical FORTRAN,* Second Edition, CRC Press, 2009.

[101] **Kupferschmid, Michael**, *Computing Fourier Transforms,* Department of Mathematical Sciences, Rensselaer Polytechnic Institute, 2012. *This was the textbook for the course Fast Fourier Transforms,* `MATH-4961`*, in spring 2013.*

[102] **Lan, G.**, "An optimal method for stochastic composite optimization," *Mathematical Programming* 133:1 365-397, 2012.

[103] **Lasdon, Leon S.**, *Optimization Theory for Large Systems,* McMillan, 1970. *A comprehensive exposition of techniques for large-scale mathematical (especially linear) programs, including appendices of more general interest about convex functions and their conjugates and about subgradients and directional derivatives of convex functions.*

[104] **Levenberg, K.**, "A method for the solution of certain nonlinear problems in least squares," *Quarterly of Applied Mathematics* 2: 164-168, 1944.

[105] **Lin, C. C.** and **Segal, L. A.**, *Mathematics Applied to Deterministic Problems in the Natural Sciences,* Macmillan, 1974.

[106] **Linz, Peter**, *Theoretical Numerical Analysis: An Introduction to Advanced Techniques,* John Wiley & Sons, 1979.

[107] **Luenberger, David G.**, *Introduction to Linear and Nonlinear Programming,* Second Edition, Addison-Wesley, 1989.

[108] **Mangasarian, Olvi L.**, *Nonlinear Programming,* McGraw-Hill, 1969.

[109] **Mangasarian, Olvi L.**, "Duality in Nonlinear Programming," *Quarterly of Applied Mathematics* 20: 300-302, 1962. *This article contains the converse duality theorem rephrased in [5, Theorem 12.13].*

[110] **Marlow, W. H.**, *Mathematics for Operations Research,* John Wiley & Sons, 1978.

[111] **Marquardt, Donald W.**, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM Journal* 11:2, June 1963.

[112] **Miele, A.** and **Gonzalez, S.**, "On the Comparative Evaluation of Algorithms for Mathematical Programming Problems," *Nonlinear Programming 3,* Olvi L. Magasarian, Robert E. Meyer, and Stephen M. Robinson, Eds., Academic Press, 1978.

[113] **Mitchell, John E.**, "Branch-and-Cut Algorithms for Combinatorial Optimization Problems," *Handbook of Applied Optimization* 65-77, 2002.

[114] **Mitchell, John E.**, *Classnotes,* Linear and Conic Optimization `MATP-6640/ISYE-6770`, Rensselaer Polytechnic Institute, spring 2018.

[115] **Mohrmann, Kelly Bean**, *Algorithms for Hard Nonlinear Programs,* PhD Thesis, Rensselaer Polytechnic Institute, 1993.

[116] **Mood, Alexander M.**, **Graybill, Franklin A.**, and **Boes, Duane C.**, *Introduction to the Theory of Statistics,* Third Edition, McGraw-Hill, 1963.

[117] **Moré, Jorge J.** and **Wright, Stephen J.**, *Optimization Software Guide,* SIAM, 1993. *These are notes for a short course that was presented at two SIAM conferences in 1992, so they reflect the state of numerical optimization software at that time. The first Part distinguishes several types of optimization problem and for each type lists several suitable packages. The second Part provides for each package a one-page description of the areas covered, the basic algorithms employed, the computing environment required, a contact address for obtaining the software, and sometimes citations to relevant literature. The algorithm descriptions are unfortunately very terse. Many of the packages discussed appear to be research codes, while others are commercial products.*

[118] *MPI: A Message-Passing Interface Standard,* Message Passing Interface Forum, University of Tennessee, 1994.

[119] **Nagel, Ernest** and **Newman, James R.**, *Gödel's Proof,* New York University Press, 1958.

[120] **Nash, J. C.**, *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation,* John Wiley & Sons, 1979.

[121] **Nelder, J. A.** and **Mead, R.**, "A simplex method for function minimization," *Computer Journal* 7 308-313, 1965. *It is an unfortunate accident of history that this method for unconstrained nonlinear optimization came to be known as "simplex search" even though it has nothing to do with the simplex algorithm for linear programming.*

[122] **Nesterov, Y. E.**, "Smooth minimization of non-smooth functions," *Mathematical Programming* 103:1 127-152, 2015.

[123] **Neter, John** and **Wasserman, William**, *Applied Linear Statistical Models: Regression, Analysis of Variance, and Experimental Designs,* Irwin, 1974.

[124] **Nocedal, Jorge** and **Wright, Stephen J.**, *Numerical Optimization,* Springer, 1999. *This is the first edition of [5].*

[125] **Overton, Michael L.**, *Numerical Computing with IEEE Floating Point Arithmetic,* SIAM, 2001.

[126] **Pardalos, P. M.** and **Rosen, J. B.**, *Constrained Global Optimization: Algorithms and Applications,* Lecture Notes in Computer Science 268, Springer, 1987.

[127] **Pedersen, Joseph**, *Transshipment in General Networks,* independent project in `MATP-4700/ISYE-4770` Mathematical Models of Operations Research, Rensselaer Polytechnic Institute, fall 2011.

[128] **Pedroso, Moacir**, *Hybrid Ellipsoid-Sequential Quadratic Programming Algorithms,* PhD Thesis, Rensselaer Polytechnic Institute, August 1985.

[129] **Peng, Zhimin**, **Xu, Yangyang**, **Yan, Ming**, and **Yin, Wotao**, "ARock: an Algorithmic Framework for Asynchronous Parallel Coordinate Updates," arXiv: 1506.02396v5, 27 May 2016.

[130] **Polak, E.**, *Computational Methods in Optimization: A Unified Approach,* Academic Press, 1971.

[131] **Powell, M. J. D.**, "Some global convergence properties of a variable metric algorithm for minimization without exact line searches," *Nonlinear Programming, SIAM-AMS Proceedings, Volume IX*, SIAM, 1976.

[132] **Press, William H.**, **Teukolsky, Saul A.**, **Vettering, William T.**, and **Flannery, Brian P.**, *Numerical Recipes in FORTRAN: The Art of Scientific Computing,* Second Edition, Cambridge University Press, 1992.

[133] **Reinfeld, Nyles V.** and **Vogel, William R.**, *Mathematical Programming,* Prentice-Hall, 1958.

[134] **Rohn, J.**, "Solving Systems of Linear Interval Equations," *Reliability in Computing: The Role of Interval Methods in Scientific Computing,* Ramon E. Moore, Ed., Academic Press, 1988.

[135] **Rosenbrock, H. H.**, "An Automatic Method for Finding the Greatest and Least Value of a Function," *Computer Journal* 3 175, 1960.

[136] **Rudin, Walter**, *Principles of Mathematical Analysis,* Third Edition, McGraw-Hill, 1976.

[137] **Rugenstein, Edgar K.** and **Kupferschmid, Michael**, "Active set strategies in an ellipsoid algorithm for nonlinear programming," *Computers & Operations Research* 31 941-962, 2004.

[138] **Russell, Edward J.**, "Extension of Dantzig's Algorithm to Finding an Initial Near-Optimal Basis for the Transportation Problem," *Operations Research* 17 187-191, 1969.

[139] **Sandgren, Eric**, *The Utility of Nonlinear Programming Algorithms,* PhD Thesis, Purdue University, 1977.

[140] **Schittkowski, Klaus**, *Nonlinear Programming Codes: Information, Tests, Performance,* Lecture Notes in Economics and Mathematical Systems 183, Springer-Verlag, 1980.

[141] **Shah, Sharmila**, *An Ellipsoid Algorithm for Equality-Constrained Nonlinear Programs*, PhD Thesis, Rensselaer Polytechnic Institute, August 1998.

[142] **Shah, Sharmila**, **Mitchell, John E.**, and **Kupferschmid, Michael**, "An ellipsoid algorithm for equality-constrained nonlinear programs," *Computers & Operations Research* 28 85-92, 2001.

[143] **Shor, N. Z.**, "Cut-Off Method With Space Extension in Convex Programming Problems," *Cybernetics* 13 94-96, 1977.

[144] **Sipser, Michael**, *Introduction to the Theory of Computation,* PWS Publishing, 1997. *"What are the fundamental capabilities and limitations of computers?" This book is a delightful introduction to the various answers that computer science provides.*

[145] **Spivey, W. Allen** and **Thrall, Robert M.**, *Linear Optimization,* Holt, Rinehart and Winston, 1970.

[146] **Stewart, James,** *Calculus: Early Transcendentals,* Second Edition, Brooks/Cole, 1991.

[147] **Strang, Gilbert**, *Linear Algebra and Its Applications,* Academic Press, 1976.

[148] **Strichartz, Robert S.**, *The Way of Analysis,* Jones and Bartlett, 2000.

[149] **Thomas, George B.**, **Weir, Maurice D.**, and **Haas, Joel**, *Thomas' Calculus: Early Transcendentals,* Pearson, 2011.

[150] **Trefethen, Lloyd N.** and **Bau, David**, *Numerical Linear Algebra,* SIAM, 1997.

[151] **Wagner, Harvey M.**, *Principles of Operations Research With Applications to Managerial Decisions,* Prentice-Hall, 1969.

[152] **Wagner, Harvey M.**, "Linear Programming Techniques for Regression Analysis," *Journal of the American Statistical Association* 54:285 206-212, March 1959. *This paper popularized least absolute-value-regression, but it cites even earlier work suggesting the idea.*

[153] **Walpole, Ronald E.** and **Myers, Raymond H.**, *Probability and Statistics for Engineers and Scientists,* Second Edition, Macmillian, 1978.

[154] **Wilkinson, J. H.**, *Rounding Errors in Algebraic Processes,* Dover, 1994.

[155] **Wilde, Douglass J.**, *Optimum Seeking Methods,* Prentice-Hall, 1964.

[156] **Wilde, Douglass J.** and **Beightler, Charles S.**, *Foundations of Optimization,* Prentice-Hall, 1967.

[157] **Wolfe, Philip**, "Convergence conditions for ascent methods," *SIAM Review* 11 226-235, 1969.

[158] **Wolfe, Philip**, "An extended simplex method," *Notices of the American Mathematical Society* 9:4 308, August 1962. *This is the brief abstract of paper 592-78, which was accepted to the Society's Supplemental Program #12.*

[159] **Wolfe, Philip**, "A Technique for Resolving Degeneracy in Linear Programming," Report RM-2995-PR, Rand Corporation, 1962.

[160] **Xu, Yangyang**, "Asynchronous parallel primal-dual block update methods," arXiv: 1705.06391v1, 18 May 2017.

[161] **Zangwill, Willard I.**, *Nonlinear Programming: A Unified Approach,* Prentice-Hall, 1969.

[162] **Zoutendijk, G.**, *Methods of Feasible Directions: A Study in Linear and Non-linear Programming,* Elsevier, 1960.

29.3 Other References

This category lists publications whose nontechnical content is cited in the text as authority for specific claims made there or to provide cultural context. Some of them also contain interesting mathematics.

[163] **Cardano, Gerolamo**, *Ars Magna: The Rules of Algebra,* reprint edition translated by T. Richard Witmer, Dover, 1993.

[164] **Cottle, Richard W.**, "William Karush and the KKT Theorem," *Documenta Mathematica,* Extra Volume ISMP 255-269, 2012.

[165] **Ezrachi, Ariel** and **Stuke, Maurice E.**, *Virtual Competition: The Promise and Perils of the Algorithm-Based Economy,* Harvard University Press, 2016.

[166] **Hamming, R. W.**, *Numerical Methods for Scientists and Engineers,* Second Edition, Dover, 1986.

[167] **Hutson, Matthew**, "Has artificial intelligence become alchemy?" *Science* 360:6388 478, 04 May 2018.

[168] **Lemaréchal, Claude**, "Cauchy and the Gradient Method," *Documenta Mathematica,* Extra Volume ISMP 251-254, 2012. *As described by Lemaréchal, Cauchy's original paper [22] proposed using steepest descent to minimize a sum of squares, as a way of solving simultaneous nonlinear algebraic equations arising in astronomical calculations.*

[169] **Macdiarmid, Jennie I.**, **Kyle, Janet**, **Horgan, Graham W.**, **Loe, Jennifer**, **Fyfe, Claire**, **Johnstone, Alexandra**, and **McNeill, Geraldine**, "Sustainable diets for the future: can we contribute to reducing greenhouse gas emissions by eating a healthy diet?" *The American Journal of Clinical Nutrition* 96:3 632-639, 01 August 2012.

[170] **McNutt, Marcia**, "Taking on TOP," Editorial, *Science* 352: 1147, 03 June 2016. *TOP is an acronym for Transparency and Openness Promotion, a set of eight standards adopted by "more than 500 journals." They require "the citation of all . . . program code . . . used in a given study."*

[171] **O'Neil, Cathy**, *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*, Crown, 2016.

[172] **O'Neil, Cathy**, "Life in the age of the algorithm," Book Review, *Science* 355: 137, 13 January 2017.

[173] **Polya, G.**, *Induction and Analogy in Mathematics,* Princeton University Press, 1954. *The second volume of this delightful series is Patterns of Plausible Inference.*

[174] **Polya, G.**, *Mathematical Discovery: On understanding, learning, and teaching problem solving,* two volumes, John Wiley & Sons, 1962.

[175] **Raymond, Eric**, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary,* O'Reilly, 2001.

[176] **Stodden, Victoria**, **McNutt, Marcia**, **Bailey, David H.**, **Deelman, Ewa**, **Gil, Yolanda**, **Hanson, Brooks**, **Heroux, Michael A.**, **Ionnidis, John P. A.**, and **Taufer, Michela**, "Enhancing reproducibility for computational methods," *Science* 354:6317 09 December 2016. *This article concedes that "It may not be possible to fully disclose... proprietary software such as MATLAB" but advocates that wherever possible authors "use Open Licensing when publishing digital scholarly objects."*

[177] **Thomas, Philip S.**, **Castro da Silva, Bruno**, **Barto, Andrew G.**, **Giguere, Stephen**, **Brun, Yuriy**, and **Brunskill, Emma**, "Preventing undesirable behavior of intelligent machines," *Science* 366:6468 22 November 2019.

[178] **Zlotowitz, Meir** and **Scherman, Nosson**, *Pirkei Avos: Ethics of the Fathers,* Second Edition, Mesorah Publications, February 2013.

30

Index

This book has three Indices that you can use to navigate the text, understand the notation, and find the references. The Subject Index and Symbol Dictionary will be of special interest if you are reading the Chapters out of order, while the Bibliography Citations might be useful if you are further exploring some topic in the cited literature.

30.1 Subject Index

Key words in the text appear in **bold** type at their first or defining use. This Index lists pages on which key words appear in the sense of their technical definitions, and also pages on which the text mentions important ideas that are not described by a key word. Some entries are shortened by using abbreviations from the table below.

abbreviation	meaning
LP	linear program[ming]
IP	integer program[ming]
DP	dynamic program[ming]
GRG	generalized reduced gradient
QP	quadratic program[ming]
SQP	sequential quadratic program[ming]
NLP	nonlinear program[ming]
PD	positive definite
KKT	Karush-Kuhn-Tucker
OLS	ordinary-least-squares
LAV	least-absolute-value
SVM	support vector machine[s]

If you look for an Index entry but find that it is missing, please let me know so that I can include it in a future edition of the book.

30.2 Symbol Dictionary

The undergraduate mathematics that I have assumed you already know includes the standard notation of algebra and calculus, including the locutions shown below.

$=$	equal
$\equiv$	equivalent
$\neq$	unequal
$\leq$	less or equal
$\geq$	greater or equal
$<$	less
$>$	greater
$\gg$	much greater
$\propto$	proportional
$\delta\mathbf{x}$	small difference in $\mathbf{x}$
$\Rightarrow$	implication
$\Leftrightarrow$	if and only if
$(\bullet)$, $[\bullet]$	grouping

$\{\bullet\}$	set
$\mid$	such that
$\in$	membership
$\cap$	intersection
$\cup$	union
$\setminus$	difference
$\subseteq$	subset
$\subset$	proper subset
$\varnothing$	empty set
∂	set boundary
∞	infinity
$\forall$	for all
$\lim_{a\to b}$	limit

$a+b$	add
$a-b$	subtract
$a \pm b$	symmetric range
$ab = a \cdot b = a \times b$	multiply
$a \div b = a/b = \frac{a}{b}$	divide
a^b or e^x	power
$\sqrt{a} = a^{\frac{1}{2}}$	root
$n! = 1 \cdot 2 \cdot \cdots \cdot n$	factorial
$\ln(x)$	natural logarithm
$\lg(x)$	base-2 logarithm
$\log_{10}(x)$	common logarithm
$\lfloor\bullet\rfloor$	floor
$\lceil\bullet\rceil$	ceiling

Other standard notations are reviewed or illustrated in §28.1 and §28.2.

Some standard notations are used in a consistent way throughout the book, and those are listed in §0.2.5. For example, vectors are denoted by lower-case boldface letters such as $\mathbf{v}$ and sets are named using an outline font as in $\mathbb{R}^n$. The two Hebrew letters that I have used, ℸ and ℷ, are also mentioned there just because you might not have seen them before.

Some variable names and other symbols are used repeatedly to mean the same thing. For example, $\mathbf{x}$ is almost always a vector of decision variables, $\mathbb{X}$ is almost always the set of all feasible $\mathbf{x}$ vectors, and $\mathbf{x}^\star$ is almost always an optimal point. Sometimes a name means, depending on the context in which it used, one of only a few different things. For example, $\mathbf{G}$ is an approximation to the Hessian inverse throughout Chapter 13 but a transformed ellipsoid matrix throughout Chapter 24. This Index shows some of these usual meanings along with the page on which each first appears.

c, convergence constant, 339

$\mathbf{d}$, a descent direction, 369
$\mathbf{d}$, limiting direction of a chord, 520
argmin $f(\alpha)$, value of α where f is minimized, 356
`det(•)`, determinant of a scalar, 380

$\mathbb{E}$, an ellipsoid, 775
ε, relative error, 819
$\mathcal{E}$, log relative combined solution error, 861
$\mathscr{E}$, expected value operator, 311
ξ_i, classification error, 326
$\boldsymbol{\xi}$, subgradient vector, 378
e_k, error in iterate k, 339
ϵ, descent method convergence tolerance, 356
epi (f), the epigraph of f, 375

$\mathbb{F}$, cone of feasible directions, 520
$f_{p,s}$, performance metric, 877

$\mathbf{G}$, quasi-Newton approximation to Hessian inverse, 439
$\mathbf{G}$, transformed ellipsoid matrix, 785
Γ, the gamma function, 820
γ, weighting factor in Hessian modification, 425

$\mathbb{H}$, a hyperplane, 775
h, increment in definition of a derivative, 398
h, index of pivot row in $\mathbf{A}$, 59

i, index on constraints, 56
`i`, index on tableau rows, 885

$\mathbf{J}$, Jacobian matrix, 674
j, index on variables, 56
`j`, index on tableau columns, 885

k, iteration of an optimization method, 338
κ, condition number of a matrix, 363
κ, the constant determining a hyperplane, 782

$\mathcal{L}$, Lagrangian, 295
λ, Lagrange multiplier, 485
λ, an eigenvalue, 384
λ, bias in ridge regression, 311
$\boldsymbol{\lambda}$, KKT multiplier vector, 513, 944

m, number of constraints, 56
`m`, number of tableau rows, 885
min $f(\alpha)$, minimum value of f, 356
μ, barrier multiplier, 605
μ, penalty multiplier, 581
μ, sufficient decrease parameter in Wolfe line search, 405
$\boldsymbol{\mu}$, Lagrange multipliers in quadratic subproblem, 755

$\mathcal{N}_\varepsilon(\bullet)$, neighborhood, 344
n, number of variables, 56
`n`, number of tableau columns, 885
n-choose-m, combinations, 45
η, curvature condition parameter in Wolfe line search, 406

30.3 Bibliography Citations

If you encounter a literature citation and find the reference helpful, you might like to know where else in this book that reference is cited. Each entry in this Index shows a reference number and the pages on which it is cited. For example, reference [1] is cited on each of the pages listed after its number, while reference [6] is cited on page 820 only.

