Lecture Notes
Introduction to Fortran 95 and Numerical Computing
A Jump-Start for Scientists and Engineers

Adrian Sandu
Computer Science Department, Michigan Technological University

August 23, 2001

Acknowledgements

The present lecture notes are based in part on the following excellent books:

"Fortran 90 Programming" by Ellis, Philips and Lahey, Addison-Wesley publishing company, 1994;

"Fortran 90 Course Notes" by A.C. Marshall, U. Liverpool, 1997;

"Elementary numerical analysis" by K.E. Atkinson, 2nd edition, John Wiley & Sons, Inc, 1993.

Contents

Chapter 1

A quick tour of Fortran 95

1.1 Program Form

The main program unit begins with the keyword program (optional), followed by the program name (also otional); it ends with the keyword end (required), followed by program Name_Prog (optional).

```
        program <name>
            declarations
            executable code
        end [ program <name> ]
```

Open a file *hello.f90*. Type the following "hello world" program:

```
! my first Fortran program
program hello
  print *, 'Hello World!'
end program hello
```

Compile it with **f90 hello.f90**. Run the resulting code with **a.out**.

1.2 Free vs. Fixed Formats

In Fortran 95 the code layout is obeys the following rules, which describe the "*free source form*".

- statements can begin in any column;
- multiple statements on a line are allowed; they have to be separated by semicolon **";"**
- an exclamation mark "**!**" in any column is the beginning of a comment; the rest of the line is ignored by the compiler;
- a statement can be continued on the following line by appending a "**&**" sign on the current line.

For example, the above code fragment could be written in free source form as

```
! This is free-form
temp = x; x = y; y = temp ! Swap x and y
WRITE(6,*) 'x and y are =',  &
           x,y          ! Print x and y
```

For back-compatibility with Fortran 77, Fortran 90 accepts a "*fixed source form*". In short, the fixed source form requirements are:

- all lines of code start in the 7th (or higher) column; the first 6 columns are reserved for labels, continuation characters, etc.
- each statement is written on a separate line (i.e., statements are separated by newlines); a statement can be continued on the next line by writing (any) character in the 5th column of the new line;
- comment lines start with a C in the first column.

Fixed format was born in 1950's, when each line of code was punched on a paper card ("punched card"). For debugging and maintaining the code it was clearly easier to have one statement per card. When running, a punched card had to be aligned, checked if it is a new statement or a continuation of the last, if it is a comment, etc. - lots of information stored in the first 6 columns. (A million-line code, quite common today, on punched cards, needed a truck to be carried from one computer to another).

For example, a fragment of a Fortran 90 program in fixed source form (in fact, Fortran 77 program) can be:

```
C---This-is-fixed-form
C---Swap x and y
      temp = x
      x = y
      y = temp
C---Print x and y
      WRITE(6,*) 'x and y are =',
    *            x,y
```

Our compiler (f90) will consider the program to be in fixed source format if the source file has the extension " .f" (for example, my_program.f). The compiler will consider the program to be in free source format if the source file has the extension " .f90" (for example, my_program.f90). Note that a free format source in a .f file may result in compilation errors.

1.3 Declaration of Variables

1.3.1 Memory Organization. Variables

Discuss how memory is organized

Variables correspond to storage locations in memory. They are denoted by *identifiers*, which obey the following restrictions:

- contain up to 31 characters;
- the first character must be a letter;
- names are case-insensitive (no distiction between upper and lower case).

At the beginning of the program, we declare the types of the variables. Declarations reserve the necessary number of bytes in memory and "bind" the variable name (identifier) to the address of the reserved memory space; the content of the reserved bytes will hold the variable's value. There are 6 intrinsic (default) data types in Fortran: character, logical, integer, real (single precision), double precision and complex. The last four types are refferred to as *numeric data types.*

1.3.2 Integers.

Declaration:

```
INTEGER I, J, K  or
INTEGER :: I, J, K
```

An integer can be declared and initialized with

```
INTEGER :: I = 1
```

Normally, when the program is loaded in memory, the contents of the declared variables are undefined (some compilers may set them to 0 by default). With initialization, the content of the variables when loading is set to the prescribed values.

An integer constant (whose value cannot be changed later in the program) can be declared with

```
INTEGER MAXIMUM
PARAMETER (MAXIMUM=340)
```

or with

```
INTEGER, PARAMETER :: MAXIMUM=340
```

The plain form of declaration, and the two-statement parameter declaration come from F77, and is legal in F90 also. The double colon "**::**" form is specific to F90. Note that we *have to use* the double colon form whenever more than one attribute is given for the variable (e.g., when is integer and parameter, i.e. constant) or when the variable is initialized.

1.3.3 Characters.

Declaration:

```
CHARACTER C or
CHARACTER :: C
```

For a string of characters, we can specify the length and (optionally) initialize it as

```
CHARACTER(LEN=7) :: LOGIN
```

Again we see that, if we want to specify more than one attribute (here, character and length) the double colon form is needed.

We can have constant (PARAMETER) characters/strings, in which case we use the declaration

```
CHARACTER(LEN=8), PARAMETER :: LOGIN="johndoe"
```

(the F77 two-statement form is also acceptable). Here the initialization value is 6-character long, so it will be padded by 2 blanks to fit the declared length. Similarly, if the initialization value was too long, it would have been truncated to the declared length.

Alternatively, we may let the character variable assume the length of the initialization string with

```
CHARACTER(LEN=*), PARAMETER :: LOGIN="johndoe", PASSWORD="michigantech"
```

Here LOGIN will be 7-character, and PASSWORD 12-character long.

The following equivalent form of declaration is also accepted (for back-compatibility with F77):

```
CHARACTER*8 :: LOGIN
```

The LEN attribute can be overriden by a ∗ attribute in a string declaration as follows:

```
CHARACTER(LEN=8) :: LOGIN, PASSWORD*12
```

Here *LOGIN* is a string of 8 characters, but PASSWORD is a string of 12 characters.

Note that *a string is a scalar*; in particular it is NOT an array of characters. For example, it is possible to declare a 10×10 matrix whose elements are 6-character long strings:

```
CHARACTER(LEN=6), DIMENSION(10,10) :: A
```

A string can be split accross lines by adding an ampersand & both at the end of the current line and at the beginning of the next. For example

```
"michig&
\&antech"
```

1.3.4 Reals.

The declarations

```
REAL X, Y, PI or
REAL :: X, Y, PI
```

state that X, Y and PI are single precision floating point variables.

A real parameter (whose value cannot be changed subsequently) may be declared with

```
REAL PI
PARAMETER (PI=3.141592)
```

or with

```
REAL, PARAMETER :: PI=3.141592
```

The double colon form is needed for more than one attribute (see above), or for declarations plus initializations

```
REAL :: X = 21.5}
```

Application: circle area and perimeter.

```
PROGRAM first
IMPLICIT NONE ! disable implicit d.
REAL::R,Pi,a,p
Pi=3.1415926
PRINT*, 'Please give radius:'
READ*, R
a=Pi**R ! ** is exponentation
p=2.0*Pi*R
PRINT*, 'Area=',a,' Perimieter=',p
END PROGRAM first
```

1.3.5 Double Precision.

The declarations

```
DOUBLE PRECISION X, Y, PI or
DOUBLE PRECISION :: X, Y, PI
```

state that X, Y and PI are double precision floating point variables. A similar discussion as for REAL type holds.

1.3.6 Complex

Fortran allows for complex numbers also. They are declared as

```
COMPLEX Z, W or
COMPLEX :: Z, W
```

A Fortran complex variable is (and is stored as) a pair of real (single precision floating point) variables (the real and the imaginary part, of course). For example, to declare and initialize the complex constant $2 + 3\mathbf{i}$ we use

```
COMPLEX, PARAMETER :: Z=(2.0,3.0)
```

1.3.7 Logical.

Logical (Boolean) variables are declared as

```
LOGICAL Q, P or
LOGICAL :: Q, P
```

They can take only (one of) two possible values: true or false (written, in Fortran notation, .TRUE. and .FALSE.). A declaration with initialization may look like

```
LOGICAL :: Q=.TRUE., P=.FALSE.
```

1.3.8 General Form of Declarations

The general form a F90 declaration is

$$\langle type\rangle\,[,\langle attribute-list\rangle]\ \ ::\ \ \&$$
$$[,\langle variable\rangle[=\langle value\rangle]]$$

$< attribute - list >$ contains attributes like PARAMETER, SAVE, INTENT, POINTER, TARGET, DIMENSION, etc. Any object may be given any number of attributes, provided they are compatible with each other.

1.3.9 Implicit declarations

In Fortran implicit declarations are allowed. Suppose we did not declare the variables I, J, X, Y but used them somewhere in the program. The Fortran compiler will not complain; rather, it will automatically declare I, J as integers and X, Y as reals. The rule is that undeclared variables which have the first letter I, J, K, L, M or N are considered INTEGER-s, and undeclared variables which start in A through H and O through Z are considered REAL-s. The automatic declarations based on implicit types are called implicit declarations. Some fourty years ago programmers found it cumbersome to explicitly declare all the variables all the time !

In F90 implicit declarations are permitted, but undesirable. In general, their use is a very bad programming habit, as it can mask programming errors, and can negatively impact future software development and maintainance. For example, a misspelling of a variable name will result in a new variable declaration, which can be further assigned etc, with the user being totally unaware.

An example (from A.C. Marshall) is

```
   do30i = 1.100
      <statements>
30 CONTINUE
```

Instead of a DO loop, because of the misprints, we will end up with a new real variable, do30i.

In consequence, we will always disable the implicit declarations by placing the command

```
IMPLICIT NONE
```

as the first line after any USE statements (i.e. before the declarations sequence). With this command in place, the existence of variables that are not explicitly declared will lead to a copilation error.

1.4 Assignment

An expression of the form

```
  Z = Z + 2.0
```

fetches the value of Z from memory, adds 2.0, and stores the result at the same memory location Z. In short, $Z_{\rm new} = Z_{\rm old} + 2.0$. Note that the assignment = has a totally different meaning than mathematical equality (here, the mathematical relation $Z = Z + 2.0$ is an equation without solution).

1.5 Intrinsic Numerical Operations

NUMERIC_TYPE :: a,b
NUMERIC_TYPE :: [a] ⟨*numerical_operator*⟩ b

Fortran, like any other language, defines several operators that act on numerical type variables. The addition, subtraction, multiplication, division and exponentiation operators are denoted

$$+\,, \quad -\,, \quad *\,, \quad /\,, \quad \text{and} \quad **$$

respectively. Nothe that addition and subtraction can be monadic (e.g. $+2.1$ or -2.1) or dyadic (e.g. $2.1 - 3.4$) operators. Also note that we can raise a positive real number to a real power, e.g. $2.3 ** 3.14$, but not $(-2.3) ** 3.14$.

In arithmetic expressions different numerical types can be used (will see later), but we usually cannot mix numerical and character, or numerical and logical variables.

1.6 Literal Constants

We can use value constants *directly* in the statements; such values are called *literal constants*.

For example, a number can be raised to an integer power

```
Y = X**4 ; Z = X**(−6)
```

Both exponent values 4, −6 are written directly in the source text.

For real constants, we need to use either decimal point notation or scientific notation (similar to floating point notation base 10: we have a mantissa, followed by an exponent of 10; the expenent is preceded by E)

```
Y = X + 21.54   or     Y = X + 2.154E+1
```

For double precision constants, we always use scientific notation, but now the exponent marker E is replaced by D, from double:

```
Y = X + 21.54D0 or  Y = X + 2.154D+1
```

For complex literal constants we need to specify a (real part, imaginary part) pair. To assign the number $Z = 2.25 + 4.1\mathbf{i}$ we do

```
Z  = ( 2.25, 4.1 )
```

Logical constants can take one of two values, written as

```
Q = .TRUE.  or     Q = .FALSE.
```

Finally, character literal constants are delimited by single or double quotes

```
C = 'A'   or      C = "A"
```

If the delimiter is part of the string, the convention is to write it twice inside the string, for example

```
C = 'O''Malley'
```

1.7 Relational Operators

numerical_type :: a,b
logical :: a < *relational_operator* > b

F77/F90	F90	Meaning
.GT.	>	greater than
.GE.	>=	g.t. or equal to
.LE.	<=	l.t. or equal to
.LT.	<	less than
.NE.	/ =	not equal to
.EQ.	==	equal to

Relational operators compare the values of two numerical operands and deliver a logical result (.TRUE. or .FALSE.). Note that for complex operands only .EQ. and .NE. can be used.

Example.

a = 12.0 if (a.GE.10.0) then ...

The expression a .GE. 10 evaluates to .TRUE.

1.8 Intrinsic Logical Operations

logical :: a,b
logical :: [a] < *logical_operator* > b

F90	Meaning
.NOT.	monadic logical negation
.AND.	logical AND (T.AND.T=T, else F)
.OR.	logical OR (F.OR.F=F, else T)
.EQV.	true, if both operands have same value
.NEQV.	true, if operands have different values

Example.

```
REAL :: a, b
LOGICAL :: q1, q2
q1 = (.NOT.q2) .AND. (a.GT.b)}
```

1.9 Intrinsic Character Operations

1.9.1 Substrings

```
CHARACTER(LEN=*), PARAMETER :: school="michigantech"
```

Expression	Value
school(2:4)	"ich"
school(:8)	"michigan"
school(9:)	"tech"
school(4)	error (string is scalar)
school(4:4)	"h" (correct form)

1.9.2 Concatenation

```
 CHARACTER(LEN=*), PARAMETER :: &
        state="michigan", NAME="tech"
 CHARACTER(LEN=12) :: school
 school = state // NAME
! produces "michigantech"
```

1.10 Operator Precedence

user-defined monadic	highest
**	(tightest binding)
*, /	.
monadic +, −	.
dyadic +, −	.
//	.
relational operators (.GT. etc.)	.
.NOT.	.
.AND.	.
.OR.	.
.EQV., .NEQV.	(weakest binding)
user-defined dyadic	lowest

For multiple same-level operators evaluation is done *left to right* (except for $**$).

Paranthesis can alter the order of evaluation, and are recommended anyway for clarity. Sometimes paranthesis can make subtle differences, e.g.

$$A/B * C \neq A/(B * C)$$

or

$$\begin{array}{l} A = 2 * *29; \\ B = 1.999999 * 2 * *30; \\ C = 1.999998 * 2 * *30; \\ X = A + B - C \ ! \ X = \infty, \text{ (overflow)} \\ Y = A + (B - C) \ ! \ Y = \text{correct value} \end{array}$$

Homework 0.1 *Add all possible paranthesis to indicate the order of evaluation for*

$$.NOT.A.AND.B.EQV.C.OR.D.AND.E.OR.x.GT.y.AND.y.EQ.z$$

1.11 Intrinsic Functions

The most widely used functions are part of the Fortran system (are "intrinsic" to Fortran). This means that the each Fortran system provides high quality implementations of these functions. We can call them without linking to any mathematical library. Give some examples here.

1.12 Controlling the flow of the program

1.12.1 Conditional execution

IF Blocks permit conditional execution of a part of program.

Syntax:

```
[⟨name⟩ :]   IF(⟨logical_expr⟩ ) THEN
             ⟨then_block⟩
[· · ·
             ELSEIF(⟨logical_expr⟩ ) THEN    [⟨name⟩]
             ⟨elseif_block⟩
· · ·]
[
             ELSE                            [⟨name⟩]
             ⟨else_block⟩
]
             END IF                          [⟨name⟩]
```

Both ELSEIF and ELSE are optional. There can be any number of ELSEIF branches.

First the IF-s ⟨*logical_expr*⟩ession is evaluated, and if .TRUE. the statements in the ⟨*then_block*⟩ are executed and control is then transferred to the first statement following END IF. If .FALSE., the ELSEIF-s ⟨*logical_expr*⟩essions are evaluated succesively, until the first one is found to hold .TRUE. Then the corresponding ⟨*elseif_block*⟩ statements are executed.

If none of the ELSEIF-s ⟨*logical_expr*⟩essions is found .TRUE., the ELSE branch is taken (if present). If the ELSE branch is not present the control is transferred to the first instruction after END IF.

1.12.2 Examples

(Discuss control flow diagram with rombs and rectangles)

```
IF (i .gt. 17) THEN
   PRINT*, "i > 17 !"
END IF
```

```
IF (i .gt. 17) THEN
   PRINT*, "i > 17 !"
ELSE
   PRINT*, "i <= 17 !"
END IF
```

```
IF (i .gt. 17) THEN
   PRINT*, "i > 17 !"
ELSEIF (i .eq. 17) THEN
   PRINT*, "i = 17 !"
```

```
ELSEIF (i .eq. 16) THEN
  PRINT*, "i = 16 !"
ELSE
  PRINT*, "i < 16 !"
END IF
```

Example of IF block use: solving a quadratic equation.

```
PROGRAM quadratic_eqn_1
IMPLICIT NONE
REAL::a,b,c ! coeff of the quadratic eqn.
REAL::d ! determinant
REAL::x1,x2 ! solutions , if real (cases I, II)
REAL::xre,xim ! real and imaginary parts of solutions (case III)
!read in the coefficients
PRINT*, 'Please give quadr. eqn. coeff . a, b, c:'
READ*,a,b,c
d=b**2-4.0*a*c
! check the cases and treat them seperate
IF (d.GT.0.0) THEN
    x1=(-b-SQRT(d))/(2.0*a)
    x2=(-b+SQRT(d))/(2.0*a)
    PRINT*,'The eqn. has two disctinct REAL roots: ', &
    'x1=',x1,' x2=',x2
ELSE IF(d.EQ.0.0) THEN
    x1=-b/(2.0*a)
    PRINT*, 'The eqn. has two equal roots: ', &
    'x1=x2=',x1
ELSE ! d<0
    xre=-b/(2.0*a)
    xim=SQRT(-d)/(2.0*a)
    PRINT*, 'The eqn. has two COMPLEX-conjugate roots: ',&
    xre,'+/-',xim,'i'
END IF
END PROGRAM quadratic_eqn_1
```

Example ofcomplex numbers use: solving a quadratic equation, Solution no. 2.

```
PROGRAM quadratic_eqn_2
IMPLICIT NONE
REAL::a,b,c
COMPLEX::d,x1,x2,sd
!read in the coefficients
PRINT*,'Please give a, b, c:'
READ*,a,b,c
!compute discriminant
d=b**2-4*a*c
!sqrt(d): since d is complex, sqrt is complex
sd=SQRT(d)
! compute roots:
x1=(-b+sd)/(2.0*a)
x2=(-b-sd)/(2.0*a)
PRINT*,'Roots are'
PRINT*,'X1=',x1
```

```
PRINT*,'X2=',x2
END PROGRAM quadratic_eqn_2
```

1.12.3 Repetitive action

DO Loops permit repetitive action.

Syntax:

$$\boxed{\begin{array}{l}\text{DO } \langle DO\_var\rangle = \langle n\_expr_1\rangle, \langle n\_expr_2\rangle [, \langle n\_expr_3\rangle] \\ \quad \langle exec\_stmts\rangle \\ \text{END DO}\end{array}}$$

The loop can be named and the body can contain EXIT or CYCLE statements.

The loop is worked out as follows

1. Evaluate numerical expressions $\langle n\_expr_1\rangle$, $\langle n\_expr_2\rangle$ and, if present, $\langle n\_expr_3\rangle$. If not present, $\langle n\_expr_3\rangle$ is assumed to have the default value = 1.
2. Initialize the Do variable, $DO\_var \leftarrow \langle n\_expr_1\rangle$.
3. The iteration count is computed with the formula

$$\#_{\rm it} = \max\left(\lfloor \frac{\langle n\_expr_2\rangle - \langle n\_expr_1\rangle \langle n\_expr_3\rangle}{\langle n\_expr_3\rangle} \rfloor, 0 \right)$$

Note that both the number of iterations $\#_{\rm it}$ and the stride $\langle expr_3\rangle$ are evaluated **before** execution of the loop begins. Subsequent value changes will have no influence on the iteration process. If $\#_{\rm it} > 0$ the execution cycle starts. At the end of each cycle, the iteration counter is decremented by 1

$$\#_{\rm it} \leftarrow \#_{\rm it} - 1$$

and the DO variable (DO_var) is modified by adding the stride

$$DO\_var \leftarrow DO\_var + \langle n\_expr_3\rangle$$

The iteration continues if $\#_{\rm it} > 0$, otherwise exits. As a consequence, at the end of the iterations, DO_var might assume a value different from $\langle n\_expr_2\rangle$.

Modifying the DO_var inside the body loop is prohibited (and results in a compilation error).

For example, a missing stride $\langle expr_3\rangle$ is set by default to 1. The code

```
DO i=1,9
   PRINT*, i
END DO
```

will execute the loop exactly 11 times, and will produce the output $1, 2, 3, \ldots 9$.

```
DO i=1,9,2
   PRINT*, i
END DO
```

will produce the output $1, 3, 5, \ldots 9$.

A negative stride is possible, the code

```
DO  i=9,1,-2
   PRINT*, i
END DO
```

will produce the output $9, 7, 5 \ldots 1$.

The loop

```
DO i=9,1,1
   PRINT*, i
END DO
```

will produce no output since initially $\#_{\rm it} = 0$. Also, the loop

```
DO i=1,9,-1
   PRINT*, i
END DO \\
```

will produce no output since initially $\#_{\rm it} = \max(-8, 0) = 0$.

EXIT statement finishes the innermost loop; however, with named do loops, it can exit the indicated loop, regardless of its nesting level. Similarly, CYCLE outer cycles back to the outer loop, whereas a plain CYCLE would have cycled back to the inner loop.

```
outer: DO i=1,9
  inner: DO j=i,9
    PRINT*, "before: ",i, j
    IF (j .gt. 3) CYCLE outer ! go to outer: do
    PRINT*, "before: ",i, j
    IF (j .gt. 6) EXIT outer ! go to outer: do
    PRINT*, "after: ",i,j
  END DO inner
END DO outer
```

1.13 Application to DO loops: Fahrenheit to Celsius

Write a program that converts the temperature from Fahrenheit degrees to Celsius degrees (centigrades). The steps are:

1. Problem formulation
2. Algorithm
3. Implementation
4. Run, and if the result is wrong, loop back.

```
PROGRAM temp_conv
IMPLICIT NONE
INTEGER::n    ! no. of temperatures to convert
```

```
REAL::f,c       ! temp. values in F,C
INTEGER::i      ! loop counter
PRINT*,'No. of conversions='
READ*,n
DO i=1,n
PRINT*,'Fahrenheit temp. no ',i,'='
READ*,f
c=9.0/5.0*(f-32)
PRINT*,f,' F is ',c,' C'
END DO
END PROGRAM temp_conv
```

1.14 Problems

1. write a program that reads in the radius and computes the area of a circle;
2. extend the program to test whether the area is negative; if so just STOP. This introduces IF blocks, and logical expressions;
3. extend the program to read 5 radii; this introduces a DO loop and integer variables; CYCLE and EXIT.
4. extend the code to first read all 5 radii and then compute all 5 areas; this introduces arrays. The dimension (N=5) is a PARAMETER.

1.15 Input and Output

1.15.1 Standard (keyboard/console)

```
READ*, R
PRINT*, V
PRINT*, 'R = ', R  or
PRINT*, "R = ", R
```

1.15.2 File I/O

Opening files

In Fortran, files are designated by *unit number*, which is an integer number. Values from 0 to 6 are reserved for standard I/O, standard error, etc. Usually user-defined files have unit numbers of 10 and above.

Sometimes we need to read from/ write into user defined files. In order for Fortran to be aware of the existence of user-defined files, we need to OPEN them (this is somehow similar to variable declaration).

For example, the statement

```
OPEN( UNIT=10, FILE='DATA.txt', ACTION='READ')
```

will open the file named 'data.txt'; the mode declaration states that this is an input (read-only) file; the statement will allocate unit number 10 to the file 'data.txt'; from now on, all READ(10,*) will read in data from this file.

The statement

```
OPEN( UNIT=11, FILE='results.txt', ACTION='WRITE')\END{tabular}
```

will open the file named 'results.txt'; if not present, will create one; the mode declaration states that this is an output (writeable) file; the statement will allocate unit number 11 to the file 'results.txt'; from now on, all WRITE(11,*) will write data into this file. The list of arguments must be of standard intrinsic types (unless explicit format is used). Each WRITE statement puts the arguments on a new line; non-advancing WRITE is possible, but we need to use formats. Note that the statement, as is, will wipe out the content of results.txt, should it previously exist.

It is possible to declare action='readwrite' for files which are both I/O.

Also not recommended, the two statements above can be abbreviated as

```
OPEN( 10, 'DATA.txt', 'READ')
OPEN( 11, 'results .txt ', ' WRITE')
```

The opened files may be closed with

```
CLOSE(10)
CLOSE(11)
```

The unit number can then be re-used (for a different file) in a subsequent OPEN statement.

Read and Write

The file read function has two parameters

```
READ( UNIT=UNIT−no, FMT=FORMAT−label ) [ list of arguments ]
```

Each parameter can be replaced by a ∗, with the meaning "default unit" or "default format". We will talk later about formats.

The call

```
READ(5,∗) R, V
```

reads two reals from unit number 5 in default format. Unit 5 is predefined, and is the standard input (keyboard, unless redirected). Thus, an equivalent formulation is

```
READ(∗,∗) R, V
```

The file write function is very similar,

```
WRITE( UNIT\_no, FORMAT label ) [ list  of  arguments ]
```

Each parameter can be replaced by a ∗ (defaults). The call

```
WRITE(6,∗) R, V
```

writes two reals onto unit number 6 in default format. Unit 6 is predefined, and is the standard output (monitor, unless redirected). Thus, an equivalent formulation is

```
WRITE(*,*) R, V
```

The list of arguments must be of standard intrinsic types (unless explicit format is used). Each READ statement reads the arguments from a new line; non-advancing READ is possible, but we need to use formats.

1.15.3 Application

Modify the Fahrenheit to Celsius program to read the temperature ($^\circ F$) from one file, and output it to another file ($^\circ C$).

```
PROGRAM temp_conv
IMPLICIT NONE
INTEGER::i,n
REAL::f,c
INTEGER::IN=10,OUT=11 ! unit numbers
  !open the files
OPEN(UNIT=IN,FILE='fahr.DATA',ACTION='READ')
OPEN(UNIT=OUT,FILE='celsius.DATA',ACTION='WRITE')
  !read and write how many temperatures
READ(UNIT=IN,FMT=*) n
WRITE(UNIT=OUT,FMT=*) n
DO i=1,n
READ(UNIT=IN,FMT=*) f
c=9.0/5.0*(f-32)
WRITE(UNIT=OUT,FMT=*) c
PRINT*,i,' F=',f,' is C=',c
END DO
END PROGRAM temp_conv
```

The open, close, read and write functions can be checked if they performed ok using the IOSTAT parameter. In our case we check whether end-of-file was reached and if so we exit the loop. This way there is no need to have the number of temperatures as the first number in the file.

```
PROGRAM temp_conv_3
IMPLICIT NONE
REAL::f,c
INTEGER::iostatus
!open the data file , F
OPEN(UNIT=10,FILE='fahr2.DATA',ACTION='READ',IOSTAT=iostatus)
!check if the file was opened properly
IF(iostatus .NE.0) THEN
PRINT*,'DATA FILE could not be OPENED'
STOP ! terminates the program
END IF
!open the result file
OPEN(UNIT=11,FILE='celsius2.DATA',ACTION='WRITE',IOSTAT=iostatus)
!check if the file opened properly
IF(iostatus .NE.0) THEN
PRINT*,'Output FILE cannot be OPENED'
STOP
```

```
END IF
DO
READ(UNIT=10,FMT=*,IOSTAT=iostatus) f
!check if this is a valid read
IF(iostatus .NE.0) THEN
PRINT*,'END of FILE reached'
EXIT
END IF
c=5.0/9.0*(f-32)
WRITE(UNIT=11,FMT=*) c
END DO
END PROGRAM temp_conv_3
```

1.16 Arrays

Arrays are collection of elements of similar type. This data structure allows to implement mathematical objects like vectors, matrices and tensors.

To declare an array with 3 real elements we use

```
  REAL, DIMENSION(3) :: v or
  REAL :: v(3)
  REAL v(3)
```

Particular entries of the array can be accessed with an index, running from 1 through the number of entries.

```
v(1)=1.0; v(2)=2.0; v(3)=3.0
```

One can assign values to all elements of v at once using an *array constructor*. A constructor is a list of scalar values delimited by (/ ... /). For example the same initialization of v can be achieved using

```
v  = (/ 1.0, 2.0, 3.0 /)
```

An even more compact initialization is achieved using an implicit do loop:

```
v = (/ (i , i=1,3) /)
```

In Fortran jargon the one-dimensional object v is called a rank-1 array (its entries are accessed using a single index). One can define multi-dimensional arrays as well. A rank-2 array can be defined for example as

```
  REAL, DIMENSION(2,3) :: A or
  REAL :: A(2,3)
  REAL A(2,3))
```

A particular element is now accessed using two indices: A(i,j) where $1 \leq i \leq 2$ and $1 \leq j \leq 3$.

One can initialize a rank-n array using constructors. The list of entries in a constructor is one-dimensional; we need to map this one-dimensional list onto the k-dimensional array. This is done using the intrinsic function reshape. For example, we can initialize the 6 entries of A using

```
A = reshape   ( (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /), (/ 2, 3 /) )
```

The first argument of the reshape function is the constructor, the second is another constructor containing the shape of A. Reshape initializes the entries of A in column-wise order; the result of the above operation is

$$A = \left[\begin{array}{ccc} 1.0 & 3.0 & 5.0 \\ 2.0 & 4.0 & 6.0 \end{array} \right] .$$

Arrays can be used as arguments for input/output purposes. For example

```
READ*, A
```

will (expect to) read 6 numbers from the console and assign the values to elements of A in column-wise order. Similarly,

```
PRINT*, A
```

prints the elements of A one after another in column-wise order.

1.16.1 Application

Vector times matrix multiplication.

```
PROGRAM mvproduct
IMPLICIT NONE
REAL, DIMENSION(3)  ::  V
REAL, DIMENSION(2,3) :: A
REAL,DIMENSION(2)   ::  W
INTEGER :: i,j

PRINT*,'please give V(1:3)'
READ*,V

PRINT*,'Please give A (1:2,1:3)';
READ*,a
! The list of numbers from console is 1-dimensional, and is
! reshaped by the read function to the shape of A in column-wise order.

! compute the matrix*Vector product
DO i=1,2       ! for each entry of w, compute w(i)
  w(i)=0.0
  DO j=1,3
    w(i)=w(i)+a(i,j)*V(j)
  END DO
END DO
!print eVerything.
PRINT*,'A='
DO i=1,2
    PRINT*,(A(i,j),j=1,3) ! use implicit do loop to print all elements in row i
END DO
PRINT*,'V=',V
PRINT*,'W=',w
!
END PROGRAM mvproduct
```

Chapter 2

The Building Blocks of a Fortran Application

2.1 Program Units

An F95 application will usually consist of several *program units*. One of them is the *main program unit*, where the execution of the program starts, and where, usually, it ends if everything went all right with the computation. The main program unit is delimited by the following keywords:

```
PROGRAM ⟨program_name⟩
      ⋮
END PROGRAM pname
```

The other four types of program units defined in F95 are *external functions*, *external subroutines*, *modules* and *block data units*. The beginnings and ends of these units are designated by the following syntaxes, respectively:

```
FUNCTION ⟨function_name⟩
      ⋮
END FUNCTION ⟨function_name⟩

SUBROUTINE ⟨subroutine_name⟩
      ⋮
END SUBROUTINE ⟨subroutine_name⟩

MODULE ⟨module_name⟩
      ⋮
END MODULE ⟨module_name⟩

BLOCK DATA ⟨block_data_name⟩
      ⋮
END BLOCK DATA ⟨block_data_name⟩
```

As a general principle, the programming task is divided into smaller subtasks, which are then individually solved. Different program units can implement different subtasks; programming units are standalone, in the sense that they do not depend upon implementation details in the body of other program units; the units communicate to each other via specific interfaces, so one unit only needs to be aware of the interfaces to other program units.

2.2 External Procedures

External procedures

- are parametrised blocks of code that perform a specific task; this section of code is written once, and can be reffered to as many times as needed;
- are "independent" units which may be compiled separately;
- communicate with other program units via arguments, returned values (functions only), and global variables (to be discussed another lecture);
- can be used as arguments in other procedure calls.

Advantages

- avoid code duplication (when same task appears several times in our program);
- lead to modularization of programs. The general principle of dividing the problem into subproblems, and managing the subproblems separately, can be naturally implemented using procedures;
- make software reusable (pieces can be used later in a different context);

In F95 there are 2 types of external procedures,

1. functions, and
2. subroutines.

2.3 External Functions

The syntax of a function declaration is

```
[⟨type⟩]  FUNCTION⟨fcn_name⟩([⟨formal(dummy)_args⟩])
          ⟨declare formal(dummy)_args⟩
          ⟨declare local objects⟩
          ...
          ⟨executable statements⟩
          END[FUNCTION[⟨fcn_name⟩]]
```

For example, suppose we want to declare a function that receives the three coordinates of a cartesion vector x,y,z and returns the euclidian norm of the vector, $r = \sqrt{x^2 + y^2 + z^2}$ (this is also called the 2-norm). The arguments x,y,z are REAL, and so is the returned value r.

The declaration could then be

```
! function that computes norm 2   real=type of the returned value
REAL FUNCTION norm_2(x,y,z) !x,y,z are "dummy" arguments
    IMPLICIT NONE !scope=body of the function
    REAL::x,y,z  !dummy args declaration
    norm_2=SQRT(x**2+y**2+z**2)
                          !function name behaves like a variable
                          !holding the return value
END FUNCTION norm_2

!function for norm 1
REAL FUNCTION norm_1(x,y,z)
    IMPLICIT NONE
    REAL::x,y,z
    norm_1=abs(x)+abs(y)+abs(z)
END FUNCTION norm_1

!function for norm infinity
REAL FUNCTION norm_inf(x,y,z)
    IMPLICIT NONE
    REAL::x,y,z
    norm_inf=max(abs(x),abs(y),abs(z))
END FUNCTION norm_inf

!the main program
PROGRAM norms
    IMPLICIT NONE
    REAL::a,b,c
    REAL,EXTERNAL::norm_1,norm_2,norm_inf
    PRINT*,'Please give coordinates a,b,c.'
    READ*,a,b,c
    PRINT*,'The 2-norm= ',norm_2(a,b,c) !a,b,c are "actual" args
    PRINT*,'The 1-norm= ',norm_1(a,b,c)
    PRINT*,'The inf-norm= ',norm_inf(a,b,c)
END PROGRAM norms
```

Note that the function name ⟨*fcn_name*⟩ (`norm_2`) behaves like a variable; when the function terminates, this variable holds the result of the function (the "return value" of the function). The type of function is the type of the returned result, i.e. the type of the result variable (`norm2`). This type is declared explicitly as a prefix to the function name; in our example,

```
REAL FUNCTION norm2(x,y,z).
```

Alternatively, this declaration can be mixed with the other declarations, for example

```
FUNCTION norm2(x,y,z)
IMPLICIT NONE
REAL :: x, y, z, norm2
```

Either form is valid, and one declaration should always be given (otherwise the compiler will signal an error).

The variables `x,y,z` are called *formal (dummy) arguments*. They hold the input data for the function. When the function is invoked, they will be replaced by *actual* values.

The calling program also declares the type of the function, padded with the `EXTERNAL` attribute. For example, the calling program might read in the coordinates and print the 2-norm of the vector:

```
PROGRAM print_norm
IMPLICIT NONE
REAL, EXTERNAL :: norm2
REAL :: a, b, c
PRINT*, 'input~3~coordinates:'
READ*,a,b,c
PRINT*,'the norm is',norm2(a,b,c)
END PROGRAM print_norm
```

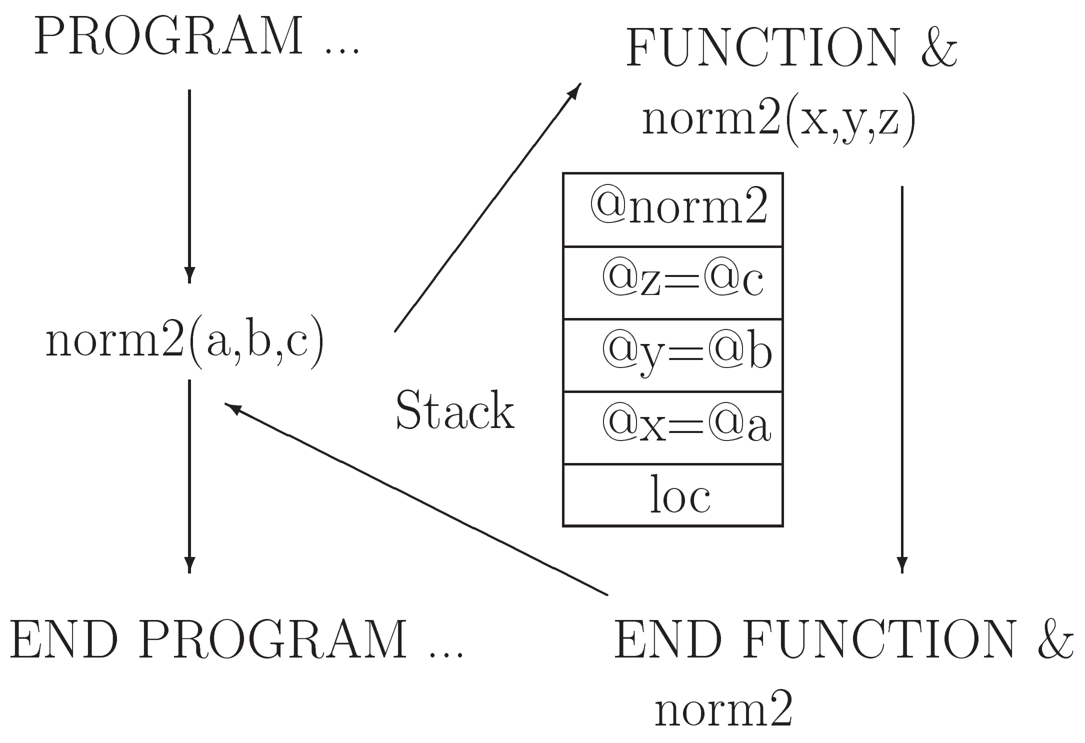
The declaration `REAL, EXTERNAL :: norm2` tells the main program that `norm2` is an external function which returns a `REAL` result. Of course, the compiler can figure out itself that `norm2` is a function by seeing the name `norm2` followed by paranthesis and the list of arguments at the place of call. Therefore, the `EXTERNAL` attribute is not really necessary here, and we can simplify the declaration to `REAL :: norm2`. However, it is good programming practice to have the `EXTERNAL` attribute, and I advise you to keep it whenever external functions (or procedures) are used. In passing, we note that there are instances when the name of the external function appears without the argument list - e.g. when the function name itself is an argument in another function call. In these instances, the `EXTERNAL` attribute is mandatory, since the compiler cannot distinguish between a function name and a variable name. Again, I advise you to give the `EXTERNAL` attribute all the time: the code is more readable, and you do not have to remember the detailed requirements for using this attribute.

The argument passing mechanism uses stacks. A stack is a memory structure which can be accessed (for both writting (push) and reading (pop)) from the top only. An example of a stack is: [see picture].

2.4 Actual arguments, formal (dummy) arguments and local variables

The function is invoked by its name, followed by a list of *actual arguments*. When the call is issued, in a special memory structure, called the *program stack*, 5 new locations are reserved; the addresses of the 3 arguments `a,b,c` are pushed, in this order, into the stack. The 4th location contains the address of the real variable `norm2`, which was declared in the main program and is expected to hold the result of the function calculation. The 5th location is dedicated to the local variable `loc`, about which we will talk later. The control is then transferred from the caller (here, the main program) to the called function `norm2`. The function "sees" the stack; from the function perspective, on the stack are (top to bottom): result variable address, argument `z` address, argument `y` address, and argument `x` address. The formal (dummy) names `x,y,z` are therefore just aliases for the actual variable names, to be used inside the function body (we say that, when the funcion is called, formal (dummy) variables are replaced by actual variables). It is therefore required that *the order in which the actual arguments are supplied is precisely the order of formal (dummy) arguments in the function declaration*. Moreover, *the type of n^{th} actual argument should coincide exactly with the type its omologue (n^{th}) formal (dummy) argument*.

The norm calculation is performed and the result stored in `norm2` (the function knows the address of this variable also!). At this point the control is returned to the main program. Here the function call `norm2(a,b,c)` is simply replaced by the value found in the result variable `norm2`, and the computations in the main program proceed further.



As presented here, the argument passing is said to be done "by reference", since references to, rather than values of arguments are put on the stack. Note that, in practice, the situation is more complicated; the actual parameter-passing mechanism can be somewhat different than what we show here. This is the case, for example, when the function is invoked with literal constant arguments (perfectly legal in F95)

```
PRINT*, norm2(1.0, 2.0, 3.0)
```

Nevertheless, the above discussion is useful to understand the "inner logic" of the F95 function declaration and function call syntax.

The variable `loc` is called a *local variable.* Its scope is the body of the function. When the function is called, space is allocated on the stack for `loc`; at this point the memory location does not carry any useful information. When the control returns to the main program (the caller), the stack pointer is reset to its previous value, wiping out all the information on stack; `loc` is therefore destroyed when the function is exited. `loc` does not exist between function calls. Obviously, a new function call means a new incarnation of `loc`, unrelated to the previous one; in particular, `loc` cannot remember its value from the previous call.

Note that `IMPLICIT NONE` should be used in any function declaration just as it is used in the main program.

2.4.1 Implicit Interfaces

Implicit interfaces are the old, F77 way of communication. Separate procedures are desired to be as independent as possible from one another (so that they can be written, compiled and maintained independently); in particular, a procedure is totally ignorant of how a different procedure looks like.

Consider our norm example. The main program knows that `norm2` is an external function that returns a real value, but it has no information about the arguments.

Special problems arise when procedures call each other; the caller has very limited knowledge on the called procedure; it passes a list of actual argument addresses to the called procedure; the called procedure retrieves this addresses in order, and assumes they are the addresses of the formal (dummy) arguments.

The disadvantage of this approach is that the compiler is able to perform only limited consistency checks. In F77, it is easy to make mistakes undetectable by the compiler when working with argument lists.

Consider, for example, the following subroutine

```
! function that computes norm 2   real=type of the returned value
REAL FUNCTION norm_2(x,y,z) !x,y,z are "dummy" arguments
    IMPLICIT NONE !scope=body of the function
    REAL::x,y,z  !dummy args declaration
    norm_2=SQRT(x**2+y**2+z**2)
                          !function name behaves like a variable
                          !holding the return value
END FUNCTION norm_2

!function for norm 1
REAL FUNCTION norm_1(x,y,z)
    IMPLICIT NONE
    REAL::x,y,z
    norm_1=abs(x)+abs(y)+abs(z)
END FUNCTION norm_1

!function for norm infinity
REAL FUNCTION norm_inf(x,y,z)
    IMPLICIT NONE
    REAL::x,y,z
    norm_inf=max(abs(x),abs(y),abs(z))
END FUNCTION norm_inf

!the main program
PROGRAM norms
    IMPLICIT NONE
    REAL::a,b,c
    REAL,EXTERNAL::norm_1,norm_2,norm_inf
    PRINT*,'Please give coordinates a,b,c.'
    READ*,a,b,c
    PRINT*,'The 2-norm= ',norm_2(a,b,c) !a,b,c are "actual" args
    PRINT*,'The 1-norm= ',norm_1(a,b,c)
    PRINT*,'The inf-norm= ',norm_inf(a,b,c)
END PROGRAM norms
```

The accidental switch of arguments cannot be detected by the compiler and produces most curious results.

2.5 Subroutines

An external subroutine a program unit that does all the communication with the rest of the world via arguments (and global variables); it does not return a value. The syntax of a subroutine declaration is

```
SUBROUTINE⟨sbr_name⟩([⟨formal(dummy)_args⟩])
⟨declare formal(dummy)_args⟩
⟨declare local objects⟩
...
⟨executable statements⟩
END[SUBROUTINE[⟨sbr_name⟩]]
```

For example, suppose we want to declare a subroutine that receives the three coordinates of a cartesion vector `x,y,z` and computes the 2-norm of the vector, $r = \sqrt{x^2 + y^2 + z^2}$ (this is also called the 2-norm). The arguments `x,y,z` are `REAL`, and so is the computed value `r`.

The declaration could then be

```
SUBROUTINE norm2s(x,y,z,r)
 IMPLICIT NONE
 REAL :: x,y,z,r
 r = sqrt(x**2+y**2+z**2)
END SUBROUTINE norm2s
```

The calling program invokes the subroutine using

```
CALL norm2s(a,b,c,d)
```

The `CALL` statement builds the argument stack and passes control to the subroutine; when `END SUBROUTINE` is encountered, the control is returned to the calling program, at the first statement after `CALL`. Unlike functions, subroutines communicate with the calling program via arguments (or global variables) only - they do NOT return a value associated with the subroutine's name.

For example, the main program might read in the coordinates and print the 2-norm of the vector:

```
PROGRAM print_norm
IMPLICIT NONE
EXTERNAL :: norm2s
REAL :: a, b, c, d
PRINT*, "input~3~coordinates:"
READ*,a,b,c
CALL norm2s(a,b,c,d)
PRINT*,"the norm is",d
END PROGRAM print_norm
```

The declaration `EXTERNAL :: norm2s` tells the main program that `norm2` is an external subroutine (we do NOT have any type declaration associated with the name!). Again, in this particular context the declaration is superfluos, but, since it will be mandatory in other contexts, and since it improves readability, I strongly recommend its use.

The actual arguments `a,b,c,d` replace the formal (dummy) arguments `x,y,z,r` (in this order) at the time of call. The type and order of actual arguments should match exactly the type and order of their omologue formal (dummy) arguments.

An `IMPLICIT NONE` statement should be given in each subroutine, just as it is given in the main program.

2.6 Comments

Procedures for most common programming tasks have already been written and are available for us to use (many times for free!). One example is the set of intrinsic functions in F95. Another example are the standard libraries for numerical computations. `BLAS` (Basic Linear Algebra Subroutines) is a collection of modules that perform matrix and vector calculations. `LAPACK` is a complete set of subroutines for solving numerical linear algebra problems.

Any time we have a programming task, and need to write a piece of code, it is advisable to check whether a standard subroutine, in a standard collection, performs the same task for us. This cuts down programming time; cuts down the likelihood of programming errors; usually, enhances performance; and, of course, enhances software maintainability.

If we are to write our own procedures from scratch, here is some expert (and free!) advice, from A.C. Marshall:

1. keep the procedures under 50 lines long, for a simple control structure;
2. make procedures as flexible as possible, to allow for software reuse;
3. pass as many variables as possible as actual arguments, and rely as little as possible on global storage or host association;
4. give procedures meaningful names, and plenty of comments;
5. there is absolutely no point in reinventing the wheel - use standard libraries whenever they offer subroutines which solve the required task.

2.7 Modules

Modules are special program units, delimited by the following syntax:

```
MODULE ⟨module_name⟩
   ⋮
END MODULE ⟨module_name⟩
```

A module contains only declarations of different entities; it can contain no executable statements. Other program units (functions, subroutines, main program or even other modules) can attach the module, and by doing so they can "see" (have access to) the entities declared in the module.

Therefore, the functionality of a module is to declare objects and to make them available to other program units. In this regard, they are different from procedures (which are supposed to perform some computations, some I/O, etc).

We can attach a module to a program unit by inserting

```
USE ⟨module_name⟩
```

as the first statement in the program unit (right after the header, and right before any declaration).

Jargon: when a program unit `USE`s a module, it has access to the (public) objects declared in the module; we say that the module entities are visible within that program unit by *use-association*.

Note that, in particular, modules can be attached to other modules, which in their turn can be attached to other modules etc. In this situation we have to be careful not to create a circular definition, which is of course prohibited. An example of a circular, hence mistaken definition, is: module 2 uses module 1, module 3 use module 2 and module 1 uses module 3.

What we said here is not the whole story with modules: it suffices for now, but we will study more in the future.

2.8 Scope of Variables

The scope of an entity is the part of the program where the entity is visible. The main program, a function or a procedure are scoping units; an entity declared inside a scoping unit is visible throughout the unit.

In particular, a variable declared in the main program or in a procedure has the body of the program/pocedure as its scope.

A variable is local (has local scope) if its scope is that of a scoping unit.

A variable whose scope is the entire program has global scope.

Some variables can be seen by multiple program units - by extension we will also talk about global scope. Global storage (global scope variables) can be implemented using modules or common blocks. This is discussed later in this chapter.

2.9 Lifetime of Variables

A variable is "alive" when its name is associated with a memory location. A variable is "alive" during the execution of the program unit that declared it:

- variables declared in the main program live throughout the execution of our application;
- local variables (declared inside a function or subroutine and which are not formal (dummy) arguments or global variables) are automatically allocated each time the procedure is called (allocation on function stack) and automatically deallocated when the procedure returns. They are called automatic variables. Their value is destroyed when the procedure returns; each call means a new variable is created.

SAVE attribute makes the variable static. It is allocated in a static part of the memory and is never deallocated (during the execution of our program). In particular static variables preserve their value between calls. Here is an example of a subroutine which counts how many times it has been called. The variable icount is static due to the initialization.

```
! function that computes norm 2   real=type of the returned value
REAL FUNCTION norm_2(x,y,z) !x,y,z are "dummy" arguments
   IMPLICIT NONE !scope=body of the function
   REAL::x,y,z  !dummy args declaration
   norm_2=SQRT(x**2+y**2+z**2)
                    !function name behaves like a variable
                    !holding the return value
END FUNCTION norm_2

!function for norm 1
REAL FUNCTION norm_1(x,y,z)
   IMPLICIT NONE
```

```
    REAL::x,y,z
    norm_1=abs(x)+abs(y)+abs(z)
END FUNCTION norm_1

!function for norm infinity
REAL FUNCTION norm_inf(x,y,z)
    IMPLICIT NONE
    REAL::x,y,z
    norm_inf=max(abs(x),abs(y),abs(z))
END FUNCTION norm_inf

!the main program
PROGRAM norms
    IMPLICIT NONE
    REAL::a,b,c
    REAL,EXTERNAL::norm_1,norm_2,norm_inf
    PRINT*,'Please give coordinates a,b,c.'
    READ*,a,b,c
    PRINT*,'The 2-norm= ',norm_2(a,b,c) !a,b,c are "actual" args
    PRINT*,'The 1-norm= ',norm_1(a,b,c)
    PRINT*,'The inf-norm= ',norm_inf(a,b,c)
END PROGRAM norms
```

Instead of initializing it, one can make the variable ilocal static using

```
INTEGER, save :: ilocal
save ilocal
```

A single SAVE statement not followed by a list of variables has the effect of declaring static(saving) all local entities in a program unit that can be saved (note that formal (dummy) arguments and automatic arrays cannot be saved).

2.10 Global Storage

We have seen so far that procedures communicate with the outside world via *arguments* and via *returned values* (functions); all other variables declared are local, their scope is the body of the procedure only.

This fact has the advantage that procedures can be written without any knowledge of the program unit from which they will be called - all that is required is that the interface is known.

However, for large programs, this will lead to very long argument lists, which can be a serious drawback when programming (how about writing a routine with 10,000 arguments, and then calling it from 1,000 diferent places with different actual arguments?). It is therefore necessary to have a mechanism for global storage, that is, to have variables that are visible from different program units, without being included in the argument lists.

In F95, modules provide this extra mechanism of communication. Variables declared in a module are visible within all procedures and programs which `USE` that particular module.

2.10.1 Circle Example

We want to write two functions, which calculate the perimeter and the area of a circle, respectively. Both functions need the value of π. We define this value in a module:

```
MODULE define_pi
 IMPLICIT NONE
 REAL, PARAMETER :: pi = 3.1415926
END MODULE define_pi
```

By USE-ing the module in both functions, we make sure that they see the same value of π. The main program reads in the value of the radius and prints the perimeter and the area of the circle.

```
REAL FUNCTION perimeter(r)
USE define_pi
  IMPLICIT NONE
  REAL :: r
  perimeter = 2.0*pi*r
END FUNCTION perimeter
```

```
REAL FUNCTION area(r)
USE define_pi
  IMPLICIT NONE
  REAL :: r
  area = pi*r*r
END FUNCTION area
```

```
PROGRAM circle
  IMPLICIT NONE
  REAL :: r
  REAL, EXTERNAL :: perimeter, area
  PRINT*, 'radius ='
  READ*, r
  PRINT*,'area = ',area(r)
  PRINT*,'perimeter = ',perimeter(r)
END PROGRAM circle
```

2.10.2 Taylor Approximation of the Exponential

Consider a program which compares the Taylor approximation of the exponential with the intrinsic function.

We define a module Coeff which declares the order n of the approximation; the maximal order allowed n_max=10 and a vector b of coefficients for the Taylor polynomial. Note that, in the definition, we include specifically the range to be 0:n_max; this means that b will contain n_max+1 elements, indexed from 0 to n_max.

```
MODULE coeff
  INTEGER :: n
  INTEGER, PARAMETER :: n_max = 10
  REAL, DIMENSION(0:n_max) :: b
END MODULE coeff

PROGRAM approx
USE coeff
  IMPLICIT NONE
  REAL :: x
  INTEGER :: i
  EXTERNAL taylor_exp
  REAL, EXTERNAL :: eval
!
  PRINT*, "please input order (n <= 10)"
  READ*, n
  n = min(n, n_max)
  CALL taylor_exp
!
  DO i=-3,3
    x= 2.0**i
    PRINT*, x,") exp=",exp(x), &
```

```
            "; taylor=", eval(x)
  END DO
END PROGRAM approx

SUBROUTINE taylor_exp
!  calculate the first n coefficients
!   in the taylor approx. of exp
USE coeff
  IMPLICIT NONE
  INTEGER :: i
  b(0) = 1.0
  DO i=1,n
    b(i) = b(i-1)/REAL(i)
  END DO
END SUBROUTINE taylor_exp

REAL FUNCTION eval(x)
! evaluate the order n
! polyn. with coefficients b(i)
USE coeff
  IMPLICIT NONE
  REAL, INTENT(IN) :: x
  INTEGER :: i
  eval = b(n)
  DO i = n-1,0,-1
    eval = b(i)+x*eval
  END DO
END FUNCTION eval
```

The subroutine `Taylor_exp` `USE`s the module `Coeff`. As a consequence, it has access to all three global variables n, n_max and `b`. Note that this subroutine has no arguments, which is legal in F90, but does all the communication with the ouside world via the global variables in the module `Coeff`. `Taylor_exp` calculates the first `n+1` coefficients in the Taylor series for the exponential function, and stores them in `b(0)` through `b(n)`.

The function `Eval` has just one input argument, `x`. It also `USE`s the module `Coeff`, hence it "sees"n, n_max and `b`. The function evaluates the value of the polynomial

$$b(0) + b(1) \cdot x + \ldots + b(n) \cdot x^n \ ,$$

and returns this value to the calling program. It is easy to notice that a nested form evaluation algorithm is used.

The main program also `USE`s `Eval`. Because of this, the variables in `Eval` exist as long as the program runs; they are effectively static variables (we will discussed in the future about this). The subroutine `Taylor_exp`, the function `Eval` and the main program all have access to n, n_max and `b`; any of them can read and write any of these variables; in this sense, n, n_max and `b` are called global variables. We say that the scope of n, n_max and `b` includes the main program, the subroutine and the function.

The main program reads in the desired order of approximation `n`; then, sets it to `n_max` if it is larger than this maximal value. When called, the subroutine `Taylor_exp` fills in `b(0) ... b(n)` with the coefficients of the $\mathtt{n}^{\mathtt{th}}$ Taylor polynomial. Finally, `Eval` is called several times, with different arguments, and the results of the intrinsic function and this approximation are printed together, for comparison. Note that, once the coefficients `b(0) ... b(n)` have been calculated and stored, they can be subsequently used to obtain any number of approximate values `Eval(x)`.

Homework 0.2 *Write a module and two subroutines* PUSH *and* POP *to simulate a stack; the module will allocate an 0:N_MAX array, the memory space for the stack, and will contain a stack pointer (an integer to keep track of the current position in the stack).*

2.11 F77 Global Storage. Storage Association.

The default global storage facility in F77 are the COMMON blocks. It is likely that we will encounter them in older software; however, we encourage the use of modules, not COMMON blocks, whenever global storage is required.

The above version of our example program can be reformulated with COMMON blocks as follows:

```
PROGRAM approx
  INTEGER :: n
  REAL, DIMENSION(0:10) :: b
  COMMON /coeff/ n, b
  PRINT*, "please input order (n <= 10)"
  READ*, n
  n = min(n, 10)
  CALL taylor_exp
!
  DO i=-3,3
    x= 2.0**i
    PRINT*, x,") exp=",exp(x), &
             "; taylor=", eval(x)
  END DO
END PROGRAM approx

SUBROUTINE taylor_exp
!  calculate the first
!  n coefficients in
!  the taylor approx. of exp
  INTEGER :: n
  REAL, DIMENSION(0:10) :: b
  COMMON /coeff/ n, b
!
  INTEGER :: i
  b(0) = 1.0
  DO i=1,n
    b(i) = b(i-1)/REAL(i)
  END DO
END SUBROUTINE taylor_exp

REAL FUNCTION eval(x)
! evaluate the order n
! polyn. with coefficients b(i)
  INTEGER :: n
  REAL, DIMENSION(0:10) :: b
  COMMON /coeff/ n, b
!
  REAL, INTENT(IN) :: x
  INTEGER :: i
  eval = b(n)
  DO i = n−1,0,−1
    eval = b(i)+x∗eval
  END DO
END FUNCTION eval
```

A common block declaration consists by the keyword `COMMON`, followed by the common block's name ⟨`cb_name`⟩ (included between slashes); common blocks are recognized by their names within all program units, i.e. `COMMON block` names are, by default, global names (they have to differ from any program unit name). In the declaration, the name is followed by the list of variables stored in the `COMMON block`.

$$COMMON/\langle cb\_name\rangle/\ var_1, var_2, var_3, \ldots$$

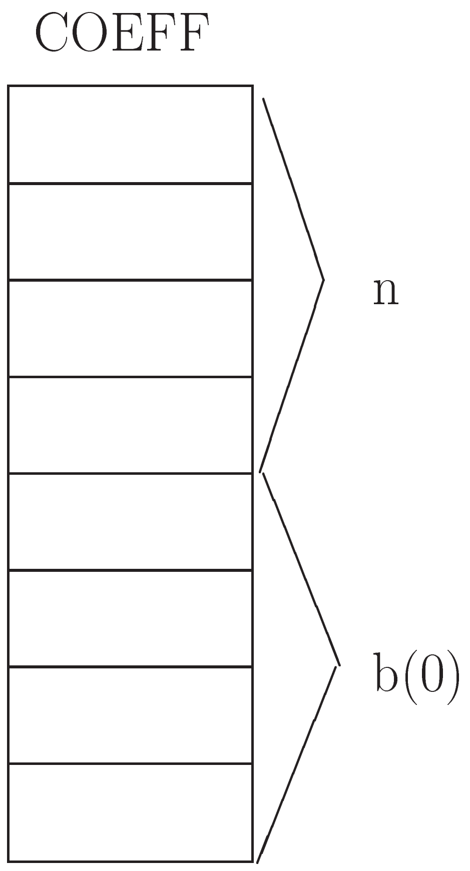
All program units that invoke the

```
COMMON /coeff/ ...
```

statement, for example, will have access to the common block `/coeff/`'s variables. Since we can have variables shared by multiple units, common blocks are a mean of implementing global storage. Note that an argument of a function or subroutine cannot be simultaneously a common block variable in the same procedure.

Physically, a common block is a contiguous zone of memory (a "block" of memory) in which succesive chunks of bytes are allocated to succesive variables (i.e. to the variables specified in the definition of the `COMMON block`). Specifically, `INTEGER`s, `REAL`s and LOGICALs are allocated 1 storage unit in the block, `DOUBLE PRECISION` and `COMPLEX` variables are given 2 storage units (1 unit is usually 4 bytes, but depends on the implementation). Characters are considered to have a different storage unit, incompatible with the numerical storage unit; they are therefore incompatible with the numerical types; chracter and numerical variables cannot be part of the same `COMMON block`.

The memory `block` can be seen by any program unit which includes its declaration, being therefore `COMMON` to several program units.



This storage scheme is also the main weakness of `COMMON blocks`. The reason is that the names of variables $\texttt{var}_1, \texttt{var}_2 \ldots$ are local to the program unit which declares them, while the `COMMON block` name is global. Thus, different program units can access the same common block, but can refer to the common block variables with different (local) names. Of course, what counts is the relative position of a variable within the block - the name is just a (local) alias for this position. Therefore, the types and the order of common block variables have to be the same, but their names can differ (pretty much like the list of formal (dummy) vs. the list of actual arguments).

For example, consider the following subroutine which prints the first two elements of the `COMMON block COEFF`.

```
SUBROUTINE p2
  INTEGER :: n, p
  COMMON /coeff/ n, p
  PRINT*, n, p
END SUBROUTINE p2
```

The compiler cannot check that types of the succesive variables in `COMMON blocks` match for different common block definitions; it will just check that local definitions are consistent. Now, our intent was to have first an integer (`n`), then a real number `b(0)`; by mistake, we declared both integers, and the compiler cannot help us; in consequence, the 32 bits of the real number `b(0)` will be interpreted as the 32 bits of a signed integer (in two's complement) `p`; instead of `1.0` we obtain `1065353216`.

In addition, common blocks containing data of heterogeneous(different) types may lead to memory missalignments.

2.12 BLOCK DATA statements

To initialize the variables in a `COMMON BLOCK` we can use the `BLOCK DATA` construct. For example,

```
BLOCK DATA coeff_init
  INTEGER :: n
  REAL, DIMENSION(0:10) :: b
  COMMON /coeff/ n, b
  DATA n /10/
  DATA b /1.0000000000e+00, 1.0000000000e+00, 0.5000000000e+00, &
          0.1666666667e+00, 4.1666666667e−02, 8.3333333333e−03, &
          1.3888888889e−03, 1.9841269841e−04, 2.4801587302e−05, &
          2.7557319224e−06, 2.7557319224e−07/
END BLOCK DATA coeff_init
```

initializes the elements of the (`coeff`) `COMMON` block variables to `n=1`, `x(1) = 3.0`, `x(2) = 5.0`. Note that a `BLOCK DATA` construct includes the `COMMON` block statement, and the definitions of the `COMMON` block variables; also, it contains the `DATA` statements, which initialize the variables with the prescribed values (i.e. at compile time the allocated memory slots for the specified variables are written with the given initial values).

The statement

```
save / coeff /
```

makes the common block static storage; it will be kept in memory for the whole duration of the current execution (in practice most Fortran systems will automatically make all common blocks static, but we should not rely on this!).

2.13 Include files

To avoid the problems of inconsistent common block declarations it is common practice to place the common blocks and their variable declarations in a file, and then to include the file in different program units which use the common block. For example, the file *coeff.cmm* may contain

```
  INTEGER :: n
  REAL, DIMENSION(0:10) :: b
  COMMON /coeff/ n, b
```

and we may use

```
  INCLUDE 'coeff.cmm'
```

This is equivalent to inserting (rewritting) the content of *coeff.cmm* at each place where the file is included. We therefore avoid repeating the same declarations and are sure of the consistency.

2.14 More on Storage Association

Fortran allows for a special, unnamed common block, called the `blank COMMON block`. It is declared as

```
COMMON // var_1, var_2 , ..., var_n
```

Blank common block is useful when most, or a large number of routines, need to share same variables. Some special features of the blank common block are

- contained data is automatically static,and
- it cannot be initialized in a `BLOCK DATA` statement;
- the compiler is more permissive (e.g. it allows the blank common block to be invoked with a different number of variables in different program units).

In F77 several entities (scalar or array) can be associated with the same memory location, using

```
EQUIVALENCE ( var_1, var_2 , ..., var_n )
```

All the variables in the list are stored in the same memory locations, more exactly, their storage spaces start at the same address in the memory.

`EQUIVALENCE` is usually used in conjunction with large common blocks, to identify the parts of the block for the current procedure. For example, a common block `/VERY_LONG/` can store 28 real arrays, containing a total of 15,421 real elements

```
COMMON /very_long/ a(7311), b(121), ..., z(1272)
```

Suppose in a procedure only the second array, `B(121)` is used; we can use the declaration

```
REAL b(121)
COMMON /very_long/ blk(15421)
EQUIVALENCE (b, blk(7312))
```

(we employed F77 syntax on purpose).

Sometimes the results of `EQUIVALENCE` statements are hard to understand. For example

```
REAL x, y(2), z
COMPLEX c
EQUIVALENCE (x,y,c), (y(2), z)
c = cmplx(1.0,2.0)
```

has the effect of setting `y(1)` $\equiv$ `x` $\equiv$ `REAL(c) = 1.0` and `y(2)` $\equiv$ `z` $\equiv$ `AIMAG(c) = 2.0`.

Chapter 3

More on Flow Control

3.1 Named IF and DO Blocks

Naming `IF` blocks is mainly cosmetic, but is useful to improve readability when nested `IF` blocks are used (nesting can go to any depth!). For example

```
lev0: IF (a.gt.b) THEN
        PRINT*, "a is larger"
      ELSEIF (a.lt.b) THEN  lev0
lev1:  IF (b.gt.c) THEN
         PRINT*, "b is larger"
        ELSEIF (b.lt.c) THEN lev1
lev2:     IF (c.gt.d) THEN
            PRINT*, "c is larger"
          END IF lev2
        END IF  lev1
      END IF    lev0
```

A similar discussion for DO loops.

3.2 The IF Statement

Is a shorter form of an `IF` block, when neither the `ELSEIF`s nor the `ELSE` branches are present. Syntax:

```
IF (⟨logical_expression⟩) statement
```

If the ⟨*logical_expr*⟩ession is `.TRUE.` then the (single) ⟨*statement*⟩ is executed; if `.FALSE.`, then control is just passed further. Note that the single ⟨*statement*⟩ is usually a very powerfull instruction, like `EXIT` or `GOTO`.

Homework 0.3 *Write a program that reads 3 lengths (3 real numbers) and reports back if they can define a triangle; if yes, print a message if the defined triangle is equilateral, isosoles, or arbitrary.*

3.3 More on DO Loops

The general syntax of a do loop is:

```
[⟨name⟩ :]DO
   ⟨exec_stmts⟩
END DO [⟨name⟩ :]
```

The body of the loop (⟨*exec_stmts*⟩) is delimited by `DO` and `ENDDO` keywords. A name can be associated with the DO loop. ⟨*exec_stmts*⟩ are executed as many times as required. As it stands the loop will cycle indefinitely; we have to append the loop with the proper mechanism to control the number of iterations.

3.3.1 Conditional EXIT

```
DO
  ⟨exec_stmts_1⟩
  IF ( ⟨logical_expr⟩) EXIT
  ⟨exec_stmts_2⟩
END DO
```

The ⟨*logical_expr*⟩ession is evaluated at every sweep through the loop. If true, `EXIT` is executed, which (of course!) exits the loop and transfers control to the first statement after `END DO`. Usually, `EXIT` is in either the first or the last statement in the body of the loop, for improved readability. An `EXIT` statement outside the body of a loop is an error.

Conditional EXIT loops are useful when we want the input data to control the number of iterations. For example, the Fahrenheit to Celsius program can read and convert temperatures until an unrealistic temperature, say -1000 or less, is read in. Try this example!

3.3.2 Conditional CYCLE

```
DO
  ⟨exec_stmts_1⟩
  IF ( ⟨logical_expr⟩) CYCLE
  ⟨exec_stmts_2⟩
END DO
```

If ⟨*logical_expr*⟩ession is true, `CYCLE` is executed; it forcess control to bypass ⟨*exec_stmts*$_2$⟩ and to jump to the `DO` statement; the loop will then start the next iteration. A `CYCLE` statement outside the body of a loop is an error.

For example, the Fahrenheit to Celsius program can skip the conversion of the temperatures that are unreasonably high, say 1000 or above. Try this example!

3.3.3 Exit and Cycle with named loops

`EXIT` statement finishes the innermost loop; however, with named do loops, it can exit the indicated loop, regardless of its nesting level. Similarly, `CYCLE outer` cycles back to the outer loop, whereas a plain `CYCLE` would have cycled back to the inner loop.

```
outer: DO i=1,9
```

```
inner : DO j=i,9
  PRINT*, "before: ",i, j
  IF (j > 3) CYCLE outer ! go to outer: do
  PRINT*, "before: ",i, j
  IF (j > 6) EXIT outer ! go to outer: do
  PRINT*, "after: ",i,j
END DO inner
END DO outer
```

3.3.4 Initial Test (DO...WHILE) Loop

Syntax:

```
DO WHILE ⟨logical_expr⟩
   ⟨exec_stmts⟩
END DO
```

The ⟨*logical_expr*⟩ession is evaluated at the beginning of every iteration; the loop executes only if it holds `.TRUE.` Clearly, the operands of the ⟨*logical_expr*⟩ession need to be modified during execution (otherwise the loop will either not execute or continue forever). Initial test loops are standard in most programming languages and have numerous applications.

The DO...WHILE LOOP can be replaced with the functionally equivalent construction: DO; IF<>EXIT ... END DO.

3.4 SELECT CASE

Syntax:

```
[⟨name⟩ :]  SELECT CASE (⟨case_expr⟩ )
[· · ·
            CASE (⟨case_selector⟩)          [⟨name⟩]
            ⟨exec_stmts⟩
· · ·]
[
            CASE DEFAULT                    [⟨name⟩]
            ⟨exec_stmts⟩
]
            END SELECT                      [⟨name⟩]
```

- ⟨*case_expr*⟩ must be a scalar of type INTEGER, LOGICAL or CHARACTER;
- ⟨*case_selector*⟩ can be a single value (`.TRUE.`) or a range (12 : 16); one cannot use an expression as case selector.
- ⟨*case_expr*⟩ is evaluated and compared to the ⟨*case_selector*⟩s, in order; when a match is found, the branch is taken and the corresponding ⟨*exec_stmts*⟩ executed. If no ⟨*case_selector*⟩ matches then the `CASE DEFAULT` branch is executed, if present.
- At the end of ⟨*exec_stmts*⟩ in the selected branch the control is passed to the first statement following `END SELECT`.

A CASE SELECT function can be implemented with an IF ... ELSEIF... ENDIF construction; however, the former needs a different expression to evaluate at each branch, and is therefore less efficient. Example:

```
SELECT CASE (i)
  CASE (:0)    ; PRINT*, "i<=0"
  CASE (1)     ; PRINT*, "i=1"
  CASE DEFAULT; PRINT*, "i>=2"
END SELECT
```

```
PROGRAM season
  IMPLICIT NONE
  INTEGER::month
  PRINT*,'Give month'
  READ*,month
  SELECT CASE (month)
  CASE (12,1,2)
    PRINT*,'Month ',month,' is IN winter'
  CASE(3:5)    ! this is range from 3 to 5
    PRINT*,'Month ',month,' is IN spring'
  CASE(6:8)
    PRINT*,'Month ',month,' is IN summer'
  CASE(9:11)
    PRINT*,'Month ',month,' is IN  fall '
  CASE DEFAULT
    PRINT*,'invalid month: ',month
  END SELECT
END PROGRAM season
```

3.5 Exceptional Situations

All the control flow constructs described so far enter the construct at only one place (IF, DO or SELECT CASE) and exit the construct at only one place also (END IF, END DO or END SELECT respectively). This enables the programmer to easily control the logic of the code.

In some situations, it is convenient to have the possibility to exit the constructs at different places; for example, if an error in the data was detected, measures have to be taken right away, even if we are in the middle of several multiple nested DO loops.

Fortran provides 3 different means to achieve this.

3.5.1 STOP

The statement STOP immediately terminates the execution of the program. It is used when the program has detected an unrecoverable error.

3.5.2 RETURN

RETURN is called from a subroutine, causing its execution to be terminated immediately and transferring the control back to the caller.

3.5.3 GO TO

Syntax:

$\boxed{\texttt{GOTO}\ \langle \textit{stmt\_label} \rangle}$ or $\boxed{\texttt{GO TO}\ \langle \textit{stmt\_label} \rangle}$

A statement label ($\langle$*stmt_label*$\rangle$) is a number of up to 5 digits (1 . . . 99999), written in column 1 through 5 (this is inherited from the fixed form). It is separated from the statement it labels by at least 1 blank.

For example,

```
12345 PRINT*, "Error"
001   CONTINUE
```

The statement `GOTO`$\langle$*stmt_label*$\rangle$ immediately transfers control to the statement labeled by $\langle$*stmt_label*$\rangle$.

```
IF (A .EQ. 0) GO TO 12345
........
12345 PRINT*, "Error: A=0"
! Begin the recover-from-error strategy
```

The $\langle$*stmt_label*$\rangle$ can be an integer expression, evaluated at run time.

Except when needed to recover from error situations, we will avoid the use of `GO TO`, since it leads to unstructured programs.

"Experience over many years has shown [GOTO statements] to be the single biggest cause of bad programming habits and consequent programming errors." (T. Ellis, I. Philips, T. Lahey).

Chapter 4

Computer Representation of Numbers and Computer Arithmetic

4.1 Binary numbers

In the decimal system, the number 107.625 means

$$107.625 = 1 \cdot 10^2 + 7 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3} \ .$$

Such a number is the sum of terms of the form {a digit times a different power of 10} - we say that 10 is the *basis* of the decimal system. There are 10 digits (0,...,9).

All computers today use the *binary system*. This has obvious hardware advantages, since the only digits in this system are 0 and 1. In the binary system the number is represented as the sum of terms of the form {a digit times a different power of 2}. For example,

$$\begin{aligned} (107.625)_{10} &= 2^6 + 2^5 + 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} \\ &= (1101011.101)_2 \ . \end{aligned}$$

Arithmetic operations in the binary system are performed similarly as in the decimal system; since there are only 2 digits, 1+1=10.

$$\begin{array}{rcccccc} & & 1 & 1 & 1 & 1 & 0 \\ + & & & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \qquad \qquad \begin{array}{rcccccc} & & & & 1 & 1 & 1 \\ \times & & & & 1 & 1 & 0 \\ \hline & & & & 0 & 0 & 0 \\ & & & 1 & 1 & 1 & \\ & & 1 & 1 & 1 & & \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 \end{array}$$

Decimal to binary conversion. For the integer part, we divide by 2 repeatedly (using integer division); the remainders are the successive digits of the number in base 2, from least to most significant.

Quotients	107	53	26	13	6	3	1	0
Remainders		1	1	0	1	0	1	1

For the fractional part, multiply the number by 2; take away the integer part, and multiply the fractional part of the result by 2, and so on; the sequence of integer parts are the digits of the base 2 number, from most to least significant.

Fractional	0.625	0.25	0.5	0
Integer		1	0	1

Octal representation. A binary number can be easily represented in base 8. Partition the number into groups of 3 binary digits ($2^3 = 8$), from decimal point to the right and to the left (add zeros if needed). Then, replace each group by its octal equivalent.

$$(107.625)_{10} = (\boxed{1}\boxed{101}\boxed{011}.\boxed{101})_2 = (153.5)_8$$

Hexadecimal representation. To represent a binary number in base 16 proceed as above, but now partition the number into groups of 4 binary digits ($2^4 = 16$). The base 16 digits are 0,...,9,A=10,...,F=15.

$$(107.625)_{10} = (\boxed{0110}\boxed{1011}.\boxed{1010})_2 = (6B.A)_{16}$$

1. Convert the following binary numbers to decimal, octal and hexa: 1001101101.0011, 11011.111001;
2. Convert the following hexa numbers to both decimal and binary: 1AD.CF, D4E5.35A;
3. Convert the following decimal numbers to both binary and hexa: 6752.8756, 4687.4231.

4.2 Memory

The data and the programs are stored in *binary format* in computer's memory. Memory is organized in *bytes*, where **1 byte = 8 binary digits**. In practice we use multiples of byte.

1 Kb	1024 bytes	2^{10} bytes
1 Mb	1024 Kb	2^{20} bytes
1 Gb	1024 Mb	2^{30} bytes

There are several physical memories in a computer; they form a *memory hierarchy*. Note that the physical chips for cache memory use a different technology than the chips for main memory; they are faster, but smaller and more expensive. Also, the disk is a magnetic storage media, a different technology than the electronic main memory; the disk is larger, cheaper but slower.

Memory Type	Size	Access time
Registers	8 bytes	1 clock cycle
Cache, Level 1	126 Kb - 512 Kb	1 ns
Cache, Level 2	512 Kb - 8 Mb	10 ns
Main memory	8 Mb - 2 Gb	60 ns
Hard drive	2 Gb - 40 Gb	10 ms

4.2.1 Characters in Memory

Characters are letters of the alphabet, both upper and lower case, punctuation marks, and various other symbols. In the ASCII convention (American Standard Code for Information Interchange) one character uses 7 bits. (there are at most $2^7 = 128$ different characters representable with this convention). As a consequence, *each character will be stored in exactly one byte of memory.*

Homework 0.4 *Implement the following program*

```
PROGRAM test_char
CHARACTER a, b
a='s'
WRITE(6,*) 'Please input b:'
READ*, b
WRITE(6,*) a,b
STOP
END
```

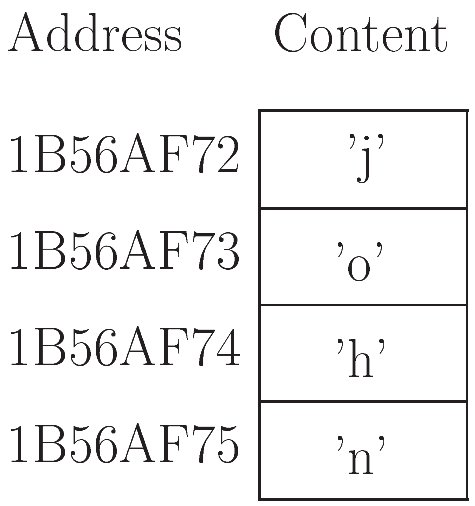
Note how characters are declared and initialized. Run the program successfully.

4.2.2 The Memory Model

When programming, we think of the main memory as a long sequence of bytes. Bytes are numbered sequentially; each byte is designated by its number, called the *address*.

For example, suppose we have a main memory of 4 Gb; there are 2^{32} bytes in the memory; addresses ranging from $0 ... 2^{32}-1$ can be represented using 32 bits (binary digits), or (equiv.) by 8 hexa digits.

Suppose we want to store the string "john". With one character per byte, we need 4 successive memory locations (bytes) for the string. Each memory location has an *address* and a *content*.



When we declare a variable, the corresponding number of bytes is reserved in the memory; the name of the variable is just an alias for the address of the first byte in the storage.

4.3 Reprentation of Signed Integers

m binary digits (bits) of memory can store 2^m different numbers. They can be positive integers between $\boxed{00...00} = (0)_{10}$ and $\boxed{11...11} = (2^m-1)_{10}$. For example, using $m=3$ bits, we can represent any integer between 0 and 7.

If we want to represent signed integers (i.e. both positive and negative numbers) using m bits, we can use one of the following methods:

- *Sign/Magnitude representation.* Reserve the first bit for the signum (for example, let 0 denote positive numbers, and 1 negative numbers); the other $m-1$ bits will store the magnitude (the absolute value) of the number. In this case the range of numbers represented is $-2^{m-1}+1$ to $+2^{m-1}-1$. With $m=3$ there are 2 bits for the magnitude, different possible magnitudes, between 0 and 127; each of these can have a positive and negative sign. Note that with this representation we have both positive and negative zero. If we make the convention that the sign bit is 1 for negative numbers we have

Number_{10}	$([\text{S}]\text{M})_2$
-3	[1]11
-2	[1]10
-1	[1]01
-0	[1]00
+0	[0]00
+1	[0]01
+2	[0]10
+3	[0]11

- *Two's complement representation.* All numbers from -2^{m-1} to $+2^{m-1}-1$ are represented by the smallest positive integer with which they are congruent modulo 2^m. With $m=3$, for example, we have

Number_{10}	$(2\text{C})_{10}$	$(2\text{C})_2$
-4	4	100
-3	5	101
-2	6	110
-1	7	111
0	0	000
1	1	001
2	2	010
3	3	011

  Note that the first bit is 1 for negative numbers, and 0 for nonnegative numbers.

- *Biased representation.* A number $x \in [-2^{m-1}, 2^{m-1}-1]$ is represented by the positive value $\bar{x} = x + 2^{m-1} \in [0, 2^m-1]$. Adding the *bias* 2^{m-1} gives positive results.

Number_{10}	$(\text{biased})_{10}$	$(\text{biased})_2$
-4	0	000
-3	1	001
-2	2	010
-1	3	011
0	4	100
1	5	101
2	6	110
3	7	111

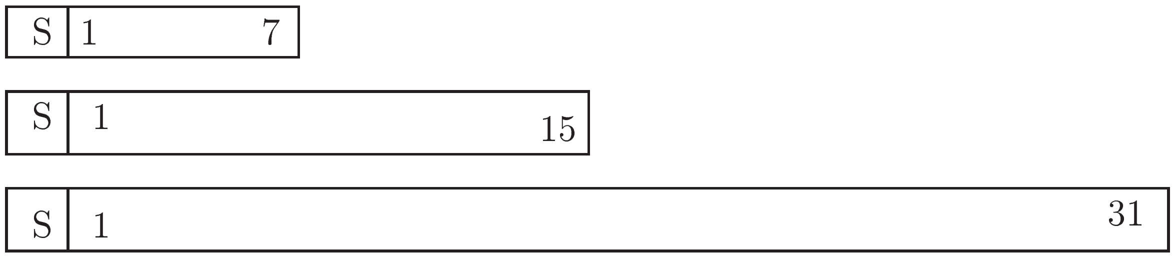
The first bit is 0 for negative numbers, and 1 for nonnegative numbers.

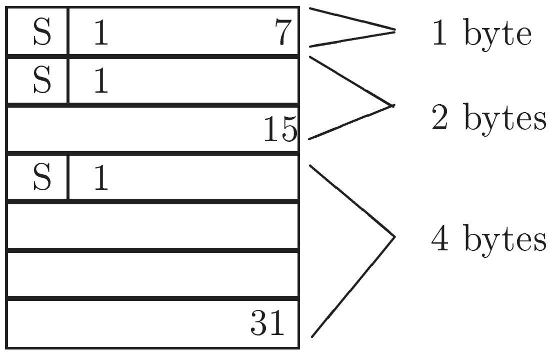
4.3.1 Integers in Memory

One byte of memory can store $2^8 = 256$ different numbers. They can be positive integers between $\boxed{00000000} = (0)_{10}$ and $\boxed{11111111} = (255)_{10}$.

For most applications, one byte integers are too small. Standard data types usually reserve 2, 4 or 8 successive bytes for each integer. In general, using p bytes ($p = 1, 2, 4, 8$) we can represent integers in the range

Unsigned integers:	0	$\cdots$	$2^{8p} - 1$
Signed integers:	-2^{8p-1}	$\cdots$	$2^{8p-1} - 1$





Homework 0.5 *Compute the lower and upper bounds for signed and unsigned integers representable with $p = 2$ and with $p = 4$ bytes.*

Homework 0.6 *Write a Fortran program in which you define two integer variables m and i. Initialize m to 2147483645. Then read i and print out the sum $m + i$.*

```
PROGRAM test_int
IMPLICIT NONE
INTEGER :: m,i
m = 2147483645
DO i=1,10
  PRINT*,'i=',i ,'.  m+i=',m+i
END DO
END PROGRAM test_int
```

Run the program several times, with i = 1,2,3,4,5.

1. *Do you obtain correct results ? What you see here is an example of* integer overflow. *The result of the summation is larger than the maximum representable integer.*
2. *What exactly happens at integer overflow ? In which sense are the results inaccurate ?*
3. *How many bytes does Fortran use to represent integers ?*

4. *Modify the program to print -m-i and repeat the procedure. What is the minimum (negative) integer representable ?*

4.3.2 Note.

Except for the overflow situation, the result of an integer addition or multiplication is always exact (i.e. the numerical result is exactly the mathematical result).

4.4 Floating-Point Numbers

For most applications in science and engineering integer numbers are not sufficient; we need to work with real numbers. Real numbers like π have an infinite number of decimal digits; there is no hope to store them exactly. On a computer, floating point convention is used to represent (approximations of) the real numbers. The design of computer systems requires in-depth knowledge about FP. Modern processors have special FP instructions, compilers must generate such FP instructions, and the operating system must handle the exception conditions generated by these FP instructions.

We will now illustrate the floating point representation in base 10. Any decimal number x can be *uniquely* written as

$x = \sigma \cdot m \cdot 10^e$		
σ	+1 or -1	sign
m	$1 \le m < 10$	mantissa
e	integer	exponent

For example

$$107.625 = +1 \cdot 1.07625 \cdot 10^2 \ .$$

If we did not impose the condition $1 \le m < 10$ we could have represented the number in various different ways, for example

$$(+1) \cdot 0.107625 \cdot 10^3 \ \text{or} \ (+1) \cdot 0.00107625 \cdot 10^5 \ .$$

When the condition $1 \le m < 10$ is satisfied, we say that the mantissa *is normalized.* Normalization guarantees that

1. the FP representation is unique,
2. since $m < 10$ there is exactly one digit before the decimal point, and
3. since $m \ge 1$ the first digit in the mantissa is nonzero. Thus, none of the available digits is wasted by storing leading zeros.

Suppose our storage space is limited to 6 decimal digits per FP number. We allocate 1 decimal digit for the sign, 3 decimal digits for the mantissa and 2 decimal digits for the exponent. If the mantissa is longer we will chop it to the most significant 3 digits (another possibility is rounding, which we will talk about shortly).

σ	MMM	EE

Our example number can be then represented as

$$\underbrace{\boxed{+1}}_{\sigma}\ \underbrace{\boxed{107}}_{m}\ \underbrace{\boxed{+2}}_{e}$$

A floating point number is represented as (sign, mantissa, exponent) with a limited number of digits for the mantissa and the exponent. The parameters of the FP system are $\beta = 10$ (the basis), $d_m = 3$ (the number of digits in the mantissa) and $d_e = 2$ (the number of digits for the exponent).

Most real numbers cannot be exactly represented as floating point numbers. For example, numbers with an infinite representation, like $\pi = 3.141592\ldots$, will need to be "approximated" by a finite-length FP number. In our FP system, π will be represented as

$$\boxed{+}\boxed{314}\boxed{00}$$

Note that the finite representation in binary is different than finite representation in decimal; for example, $(0.1)_{10}$ has an infinite binary representation.

In general, the FP representation $f\ell(x)$ is just an approximation of the real number x. The *relative error* is the difference between the two numbers, divided by the real number

$$\delta = \frac{x - f\ell(x)}{x} \ .$$

For example, if $x = 107.625$, and $f\ell(x) = 1.07 \times 10^2$ is its representation in our FP system, then the relative error is

$$\delta = \frac{107.625 - 1.07 \times 10^2}{107.625} \approx 5.8 \times 10^{-3}$$

Another measure for the approximation error is the number of *units in the last place*, or `ulps`. The error in `ulps` is computed as

$$err = |x - f\ell(x)| \times \beta^{d_m - 1 - e} \ .$$

where e is the exponent of $f\ell(x)$ and d_m is the number of digits in the mantissa. For our example

$$err = |107.625 - 1.07 \times 10^2| \times 10^{3-1-2} = 0.625\texttt{ulps} \ .$$

The difference between relative errors corresponding to 0.5 `ulps` is called the *wobble factor*. If $x - f\ell(x)$ $= 0.5$ `ulps` and $f\ell(x) = m.mmm\cdots m \times \beta^e$, then $x - f\ell(x) = \left(\beta/2 \times \beta^{-dm}\right) \times \beta^e$, and since $\beta^e \le x < \beta^{e+1}$ we have that

$$\frac{1}{2} \times \beta^{-d_m} \le \frac{x - f\ell(x)}{x} = 0.5\ \texttt{ulps} \le \frac{\beta}{2} \times \beta^{-d_m}$$

If the error is n ulps, the last $\log_\beta n$ digits in the number are contaminated by error. Similarly, if the relative error is δ, the last $\log_\beta\left(2\,\delta \times \beta^{1-d_m}\right)$ digits are in error.

With normalized mantissas, the three digits $\boxed{m_1 m_2 m_3}$ always read $m_1.m_2 m_3$, i.e. the decimal point has fixed position inside the mantissa. For the original number, the decimal point can be floated to any position in the bit-string we like by changing the exponent.

We see now the origin of the term *floating point*: the decimal point can be floated to any position in the bit-string we like by changing the exponent.

With 3 decimal digits, our mantissas range between $1.00, \ldots, 9.99$. For exponents, two digits will provide the range $00, \ldots, 99$.

Consider the number 0.000123. When we represent it in our floating point system, we lose all the significant information:

$$\underbrace{\boxed{+1}}_{\sigma}\ \underbrace{\boxed{000}}_{m}\ \underbrace{\boxed{00}}_{e}$$

In order to overcome this problem, we need to allow for negative exponents also. We will use a *biased representation*: if the bits $e_1 e_2$ are stored in the exponent field, the actual exponent is $e_1 e_2 - 49$ (49 is called *the exponent bias*). This implies that, instead of going from 00 to 99, our exponents will actually range from -49 to $+50$. The number

$$0.000123 = +1 \cdot 1.23 \cdot 10^{-4}$$

is then represented, with the biased exponent convention, as

$$\underbrace{\boxed{+1}}_{\sigma}\ \underbrace{\boxed{123}}_{m}\ \underbrace{\boxed{45}}_{e}$$

What is the maximum number allowed by our toy floating point system? If $m = 9.99$ and $e = +99$, we obtain

$$x = 9.99 \cdot 10^{50} \ .$$

If $m = 000$ and $e = 00$ we obtain a representation of ZERO. Depending on σ, it can be $+0$ or -0. Both numbers are valid, and we will consider them equal.

What is the minimum positive number that can be represented in our toy floating point system? The smallest mantissa value that satisfies the normalization requirement is $m = 1.00$; together with $e = 00$ this gives the number 10^{-49}. If we drop the normalization requirement, we can represent smaller numbers also. For example, $m = 0.10$ and $e = 00$ give 10^{-50}, while $m = 0.01$ and $e = 00$ give 10^{-51}.

The FP numbers with exponent equal to ZERO and the first digit in the mantissa also equal to ZERO are called subnormal numbers.

Allowing subnormal numbers improves the resolution of the FP system near 0. Non-normalized mantissas will be permitted only when $e = 00$, to represent ZERO or subnormal numbers, or when $e = 99$ to represent special numbers.

Example (D. Goldberg, p. 185, adapted): Suppose we work with our toy FP system and do not allow for subnormal numbers. Consider the fragment of code

```
IF (x ≠ y) THEN z=1.0/(x-y)
```

designed to "guard" against division by 0. Let $x = 1.02 \times 10^{-49}$ and $y = 1.01 \times 10^{-49}$. Clearly $x \neq y$ but, (since we do not use subnormal numbers) $x \ominus y = 0$. In spite of all the trouble we are dividing by 0! If we allow subnormal numbers, $x \ominus y = 0.01 \times 10^{-49}$ and the code behaves correctly.

Note that for the exponent bias we have chosen 49 and not 50. The reason for this is self-consistency: the inverse of the smallest normal number does not overflow

$$x_{min} = 1.00 \times 10^{-49} \ , \quad \frac{1}{x_{\min}} = 10^{+49} < 9.99 \times 10^{50} = x_{\max} \ .$$

(with a bias of 50 we would have had $1/x_{min} = 10^{50} > 9.99 \times 10^{+49} = x_{\max}$).

Similar to the decimal case, any binary number x can be represented

$x = \sigma \cdot m \cdot 2^e$		
σ	+1 or -1	sign
m	$1 \le m < 2$	mantissa
e	integer	exponent

For example,

$$1101011.101 = +1 \cdot 1.101011101 \cdot 2^6 \ . \tag{4.1}$$

With 6 binary digits available for the mantissa and 4 binary digits available for the exponent, the floating point representation is

$$\underbrace{\boxed{+1}}_{\sigma} \ \underbrace{\boxed{\mathbf{1}10101}}_{m} \ \underbrace{\boxed{0110}}_{e} \tag{4.2}$$

When we use normalized mantissas, the first digit is always nonzero. With binary floating point representation, a nonzero digit is (of course) 1, hence the first digit in the normalized binary mantissa is always 1.

$$1 \le x < 2 \to (x)_2 = \mathbf{1}.m_1 m_2 m_3 \ldots$$

As a consequence, it is not necessary to store it; we can store the mantissa starting with the second digit, and store an extra, least significant bit, in the space we saved. This is called *the hidden bit technique*.

For our binary example (4.2) the leftmost bit (equal to 1, of course, showed in bold) is redundant. If we do not store it any longer, we obtain the hidden bit representation:

$$\underbrace{\boxed{+1}}_{\sigma} \ \underbrace{\boxed{10101\mathbf{1}}}_{m} \ \underbrace{\boxed{0110}}_{e} \tag{4.3}$$

We can now pack more information in the same space: the rightmost bit of the mantissa holds now the 7th bit of the number (4.1) (equal to 1, showed in bold). This 7th bit was simply omitted in the standard form (4.2). Question: Why do we prefer

4.5 The IEEE standard

The IEEE standard regulates the representation of binary floating point numbers in a computer, how to perform consistently arithmetic operations and how to handle exceptions, etc. Developed in 1980's, is now followed by virtually all microprocessor manufacturers.

Supporting IEEE standard greatly enhances programs portability. When a piece of code is moved from one IEEE-standard-supporting machine to another IEEE-standard-supporting machine, the results of the basic arithmetic operations (+,-,*,/) will be identical.

4.5.1 Floating Point Types

The standard defines the following FP types:

Single Precision. (4 consecutive bytes/ number).

$$\boxed{\pm | e_1 e_2 e_3 \cdots e_8 | m_1 m_2 m_3 \cdots m_{23}}$$

Useful for most short calculations.

Double Precision. (8 consecutive bytes/number)

$$\boxed{\pm|e_1e_2e_3\cdots e_{11}|m_1m_2m_3\cdots m_{52}}$$

Most often used with scientific and engineering numerical computations.

Extended Precision. (10 consecutive bytes/number).

$$\boxed{\pm|e_1e_2e_3\cdots e_{15}|m_1m_2m_3\cdots m_{64}}$$

Useful for temporary storage of intermediate results in long calculations. (e.g. compute a long inner product in extended precision then convert the result back to double) There is a single-extended format also. The standard suggests that implementations should support the extended format corresponding to the widest basic format supported (since all processors today allow for double precision, the double-extended format is the only one we discuss here). Extended precision enables libraries to efficiently compute quantities within 0.5 `ulp`. For example, the result of `x*y` is correct within 0.5 `ulp`, and so is the result of `log(x)`. Clearly, computing the logarithm is a more involved operation than multiplication; the log library function performs all the intermediate computations in extended precision, then rounds the result to single or double precision, thus avoiding the corruption of more digits and achieving a 0.5 `ulp` accuracy. From the user point of view this is transparent, the log function returns a result correct within 0.5 `ulp`, the same accuracy as simple multiplication has.

4.5.2 Detailed IEEE representation

(for single precision standard; double is similar)

$$\boxed{\pm|e_1e_2e_3\cdots e_8|m_1m_2m_3\cdots m_{23}}$$

Signum. "±" bit = 0 (positive) or 1 (negative).

Exponent. Biased representation, with an exponent bias of $(127)_{10}$.

Mantissa. Hidden bit technique.

$e_1e_2e_3\cdots e_8$	Numerical Value
$(00000000)_2 = (0)_{10}$	$\pm(0.m_1\ldots m_{23})_2 \times 2^{-126}$ (ZERO or subnormal)
$(00000001)_2 = (1)_{10}$	$\pm(1.m_1\ldots m_{23})_2 \times 2^{-126}$
$\cdots$	$\cdots$
$(01111111)_2 = (127)_{10}$	$\pm(1.m_1\ldots m_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.m_1\ldots m_{23})_2 \times 2^{1}$
$\cdots$	$\cdots$
$(11111110)_2 = (254)_{10}$	$\pm(1.m_1\ldots m_{23})_2 \times 2^{+127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $m_1\ldots m_{23} = 0$ NaN otherwise

Note that $-e_{min} < e_{max}$, which implies that $1/x_{min}$ does not overflow.

IEEE Format	$E_{\min}$	$E_{\max}$
Single Prec.	-126	+127
Double Prec.	-1,022	+1,023
Extended Prec.	-16,383	+16,383

Table 4.1: IEEE floating point number exponent ranges

4.5.3 Number range

The range of numbers represented in different IEEE formats is summarized in Table 4.5.3.

4.5.4 Precision

To define the *precision of the FP system*, let us go back to our toy FP representation (2 decimal digits for the exponent and 3 for the mantissa).

We want to add two numbers, e.g.

$$1 = 1.00 \times 10^0 \text{ and } 0.01 = 1.00 \times 10^{-2} .$$

In order to perform the addition, we bring the smaller number to the same exponent as the larger number *by shifting right the mantissa*. For our example,

$$1.00 \times 10^{-2} = 0.01 \times 10^0 .$$

Next, we add the mantissas and normalize the result if necessary. In our case

$$1.00 \times 10^0 + 0.01 \times 10^0 = 1.01 \times 10^0 .$$

Suppose now we want to add

$$1 = 1.00 \times 10^0 \text{ and } 0.001 = 1.00 \times 10^{-3} .$$

For bringing them to the same exponent, we need to shift right the mantissa 3 positions, and, due to our limited space (3 digits) we lose all the significant information. Thus

$$1.00 \times 10^0 + 0.00[1] \times 10^0 = 1.00 \times 10^0 .$$

We can see now that this is a limitation of the FP system due to the storage of only a finite number of digits.

The precision of the floating point system (the "machine precision") is the smallest number ϵ for which $1 + \epsilon > 1$.

For our toy FP system, it is clear from the previous discussion that $\epsilon = 0.01$.

If the relative error in a computation is $p\epsilon$, then the number of corrupted decimal digits is $\log_{10} p$.

In (binary) IEEE arithmetic, the first single precision number larger than 1 is $1 + 2^{-23}$, while the first double precision number is $1 + 2^{-52}$. For extended precision there is no hidden bit, so the first such number is $1 + 2^{-63}$. You should be able to justify this yourselves.

If the relative error in a computation is $p\epsilon$, then the number of corrupted binary digits is $\log_2 p$.

Remark: We can now answer the following question. Signed integers are represented in two's complement. Signed mantissas are represented using the sign-magnitude convention. For signed exponents the standard uses a biased representation. Why not represent the exponents in two's complement, as we do for the signed

IEEE Format	Machine precision (ϵ)	No. Decimal Digits
Single Prec.	$2^{-23} \approx 1.2 \times 10^{-7}$	7
Double Prec.	$2^{-52} \approx 1.1 \times 10^{-16}$	16
Extended Prec.	$2^{-63} \approx 1.1 \times 10^{-19}$	19

Table 4.2: Precision of different IEEE representations

integers? When we compare two floating point numbers (both positive, for now) the exponents are looked at first; only if they are equal we proceed with the mantissas. The biased exponent is a much more convenient representation for the purpose of comparison. We compare two signed integers in greater than/less than/ equal to expressions; such expressions appear infrequently enough in a program, so we can live with the two's complement formulation, which has other benefits. On the other hand, any time we perform a floating point addition/subtraction we need to compare the exponents and align the operands. Exponent comparisons are therefore quite frequent, and being able to do them efficiently is very important. This is the argument for preferring the biased exponent representation.

Homework 0.7 *Consider the real number $(0.1)_{10}$. Write its single precision, floating point representation. Does the hidden bit technique result in a more accurate representation?*

Homework 0.8 *What is the gap between 1024 and the first IEEE single precision number larger than 1024?*

Homework 0.9 *Let $x = m \times 2^e$ be a normalized single precision number, with $1 \leq m < 2$. Show that the gap between x and the next largest single precision number is*

$$\epsilon \times 2^e \ .$$

Homework 0.10 *The following program adds $1 + 2^{-p}$, then subtracts 1. If $2^{-p} < \epsilon$ the final result will be zero. By providing different values for the exponent, you can find the machine precision for single and double precision. Note the declaration for the simple precision variables ("real") and the declaration for double precision variables ("double precision"). The command $2.0**p$ calculates 2^p (** is the power operator). Also note the form of the constants in single precision (2.e0) vs. double precision (2.d0).*

```
PROGRAM test_precision
REAL a
DOUBLE PRECISION b
INTEGER p
PRINT*, 'please provide exponent'
READ*, p
a = 1.e0 + 2.e0**(-p)
PRINT*, a-1.e0
b = 1.d0 + 2.d0**(-p)
PRINT*, b-1.d0
END PROGRAM test_precision
```

Run the program for values different of p ranging from 20 to 60. Find experimentally the values of ϵ for single and for double precision.

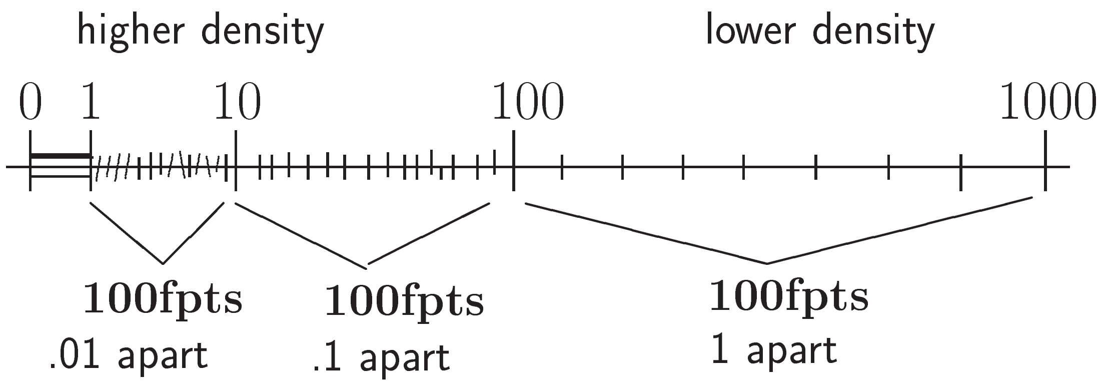
4.6 The Set of FP Numbers

The set of all FP numbers consists of

$$\text{FP} = \{\pm 0, \text{all normal}, \text{all subnormal}, \pm\infty\} \ .$$

Because of the limited number of digits, the FP numbers are *a finite set*. For example, in our toy FP system, we have approximately $2 \cdot 10^5$ FP numbers altogether.

The FP numbers are not uniformly spread between min and max values; they have a high density near zero, but get sparser as we move away from zero.



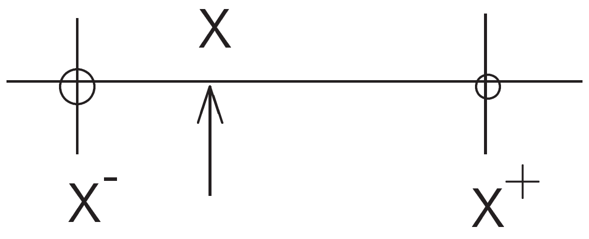
For example, in our FP system, there are 90 points between 1 and 10 (hence, the gap between 2 successive numbers is 0.01). Between 10 and 100 there are again 90 FP numbers, now with a gap of 0.1. The interval 100 to 1000 is "covered" by another 90 FP values, the difference between 2 successive ones being 1.0.

In general, if $m \times 10^e$ is a normalized FP number, with mantissa $1.00 \le m < 9.98$, the very next FP number representable is $(m + \epsilon) \times 10^e$ (please give a moment's thought about why this is so). In consequence, the gap between $m \times 10^e$ and the next FP number is $\epsilon \times 10^e$. The larger the floating point numbers, the larger the gap between them will be (the machine precision ϵ is a fixed number, while the exponent e increases with the number).

In binary format, similar expressions hold. Namely, the gap between $m \times 2^e$ and its successor is $\epsilon \times 2^e$.

4.7 Rounding - up or down

It is often the case that we have a real number X that is not exactly a floating point number: X falls between two consecutive FP numbers X^- and X^+.



In order to represent X in the computer, we need to approximate it by a FP number. If we choose X^- we say that we *rounded X down*; if we choose X^+ we say that we *rounded X up*. We can choose a different FP number also, but this makes little sense, as the approximation error will be larger than with $X^{\pm}$. For example, $\pi = 3.141592\ldots$ is in between $\pi^- = 3.14$ and $\pi^+ = 3.15$. π^- and π^+ are successive floating point numbers in our toy system.

We will denote $f\ell(X)$ the FP number that approximates X. Then

$$f\ell(X) = \begin{cases} X^- \ , & \text{if rounding down,} \\ X^+ \ , & \text{if rounding up.} \end{cases}$$

Obviously, when rounding up or down we have to make a certain representation error; we call it **the roundoff (rounding) error**.

The relative roundoff error, δ, is defined as

$$\delta = \frac{f\ell(X) - X}{X} \ .$$

This does not work for $X = 0$, so we will prefer the equivalent formulation

$$f\ell(X) = X \cdot (1 + \delta) \ .$$

What is the largest error that we can make when rounding (up or down)? The two FP candidates can be represented as $X^- = m \times 2^e$ and $X^+ = (m + \epsilon) \times 2^e$ (this is correct since they are successive FP numbers). For now suppose both numbers are positive (if negative, a similar reasoning applies). Since

$$|f\ell(X) - X| \le |X^+ - X^-|, \text{ and } X \ge X^- \ ,$$

we have

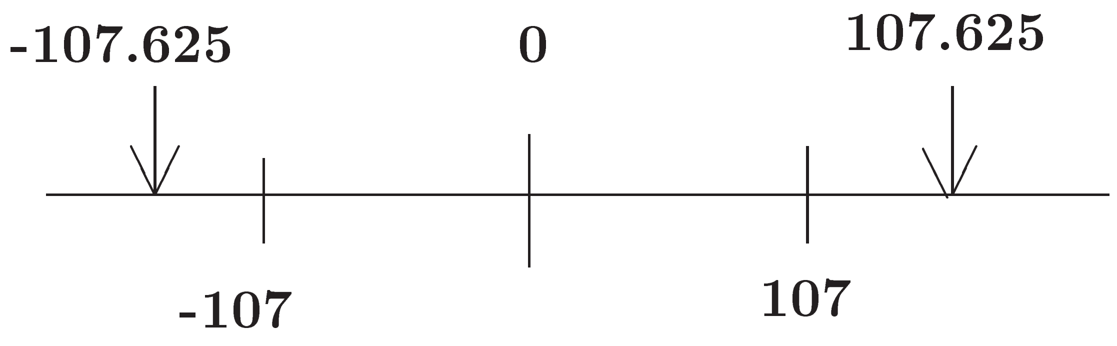
$$|\delta| \le \frac{|X^+ - X^-|}{X^-} = \frac{\epsilon \times 2^e}{m \times 2^e} \le \epsilon.$$

Homework 0.11 *Find an example of X such that, in our toy FP system, rounding down produces a roundoff error $\delta = \epsilon$. This shows that, in the worst case, the upper bound ϵ can actually be attained.*

Now, we need to choose which one of X^+, X^- 'better" approximates X. There are two possible approaches.

4.8 Chopping

Suppose $X = 107.625$. We can represent it as $\boxed{+1}\boxed{107}\boxed{+2}$ by simply discarding ("chopping") the digits which do not fit the mantissa format (here the remaining digits are 625). We see that the FP representation is precisely X^-, and we have $0 \le X^- < X$. Now, if X was negative, $X = -107.625$, the chopped representation would be $\boxed{-1}\boxed{107}\boxed{+2}$, but now this is X^+. Note that in this situation $X < X^+ \le 0$. In consequence, with chopping, we choose X^- if $X > 0$ and X^+ is $X < 0$. In both situations the floating point number is closer to 0 than the real number X, so chopping is also called *rounding toward 0.*



Chopping has the advantage of being very simple to implement in hardware. The roundoff error for chopping satisfies

$$-\epsilon < \delta_{\rm chopping} \le 0 \ .$$

For example:

$$X = 1.00999999\ldots \Rightarrow f\ell(X)_{\rm chop} = 1.00$$

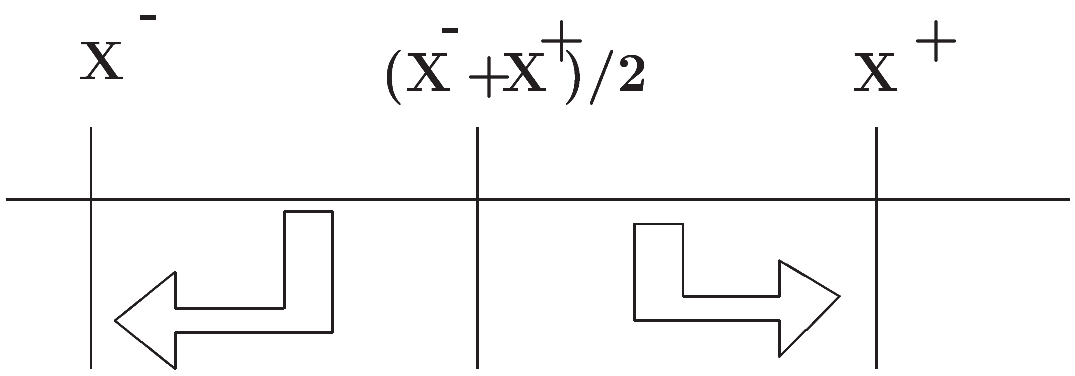
and

$$\delta = \frac{f\ell(X) - X}{X} = \frac{-0.0099...}{1.00999...} = -0.0099 \approx -0.01 = \epsilon \ .$$

4.9 Rounding to nearest.

This approximation mode is used by most processors, and is called, in short *"rounding"*. The idea is to choose the FP number (X^- or X^+) which offers the best approximation of X:

$$f\ell(X) = \begin{cases} X^- \ , & \text{if } X^- \le X < \frac{X^+ + X^-}{2}, \\ X^+ \ , & \text{if } \frac{X^+ + X^-}{2} < X \le X^+. \end{cases}$$



The roundoff for the "round to nearest" approximation mode satisfies

$$-\frac{\epsilon}{2} \le \delta_{\rm rounding} \le \frac{\epsilon}{2} \ .$$

The worst possible error is here half (in absolute magnitude) the worst-case error of chopping. In addition, the errors in the "round to nearest" approximation have both positive and negative signs. Thus, when performing long computations, it is likely that positive and negative errors will cancel each other out, giving a better numerical performance with "rounding" than with "chopping".

There is a fine point to be made regarding "round to nearest" approximation. What happens if there is a tie, i.e. if X is precisely $(X^+ + X^-)/2$? For example, with 6 digits mantissa, the binary number $X = 1.0000001$ can be rounded to $X^- = 1.000000$ or to $X^+ = 1.000001$. In this case, the IEEE standard requires to choose the approximation with an even last bit; that is, here choose X^-. This ensures that, when we have ties, half the roundings will be done up and half down.

The idea of rounding to even can be applied to decimal numbers also (and, in general, to any basis). To see why rounding to even works better, consider the following example. Let $x = 5 \times 10^{-2}$ and compute $((((1 \oplus x) \ominus x) \oplus x) \ominus x)$ with correct rounding. All operations produce exact intermediate results with the fourth digit equal to 5; when rounding this exact result, we can go to the nearest even number, or we can round up, as is customary in mathematics. Rounding to nearest even produces the correct result (1.00), while rounding up produces 1.02.

An alternative to rounding is *interval arithmetic*. The output of an operation is an interval that contains the correct result. For example $x \oplus y \in [\underline{z}, \bar{z}]$, where the limits of the interval are obtain by rounding down and up respectively. The final result with interval arithmetic is an interval that contains the true solution; if the interval is too large to be meaningful we should repeat the calculations with a higher precision.

Homework 0.12 *In IEEE single precision, what are the rounded values for* $4+2^{-20}$, $8+2^{-20}$,$16+2^{-20}$,$32+2^{-20}$,$64+2^{-20}$. *(Here and from now "rounded" means "rounded to nearest".)*

In conclusion, real numbers are approximated and represented in the floating point format. The IEEE standard recognizes four approximation modes:

1. Round Up;

2. Round Down;

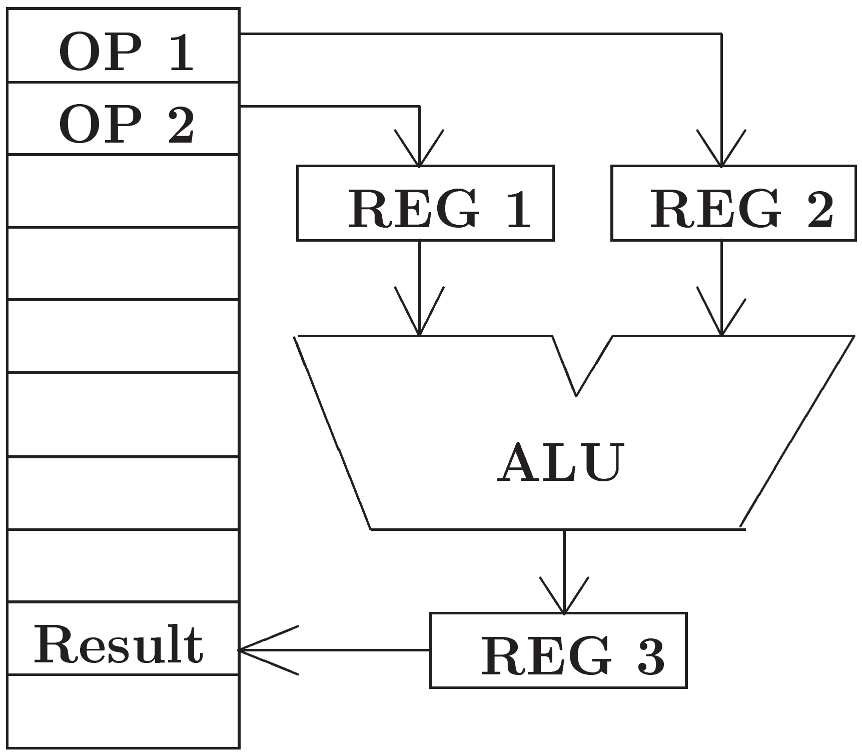
3. Round Toward Zero;

4. Round to Nearest (Even).

Virtually all processors implement the ("round to nearest") approximation. From now on, we will call it, by default, *"rounding"*. Computer numbers are therefore accurate only within a factor of $(1 \pm \epsilon/2)$. In single precision, this gives 1 ± 10^{-7}, or about 7 accurate decimal places. In double precision, this gives 1 ± 10^{-16}, or about 16 accurate decimal digits.

4.10 Arithmetic Operations

To perform arithmetic operations, the values of the operands are loaded into registers; the Arithmetic and Logic Unit (ALU) performs the operation, and puts the result in a third register; the value is then stored back in memory.



The two operands are obviously floating point numbers. The result of the operation stored in memory must also be a floating point number.

Is there any problem here? Yes! Even if the operands are FP numbers, the result of an arithmetic operation may not be a FP number.

To understand this, let us add two floating point numbers, a=$\boxed{9.72}\boxed{01}$ (97.2) and b=$\boxed{6.43}\boxed{00}$ (6.43), using our toy FP system. To perform the summation we need to align the numbers by shifting the smaller one (6.43) to the right.

$$\begin{array}{rccccc} 9. & 7 & 2 & & & 01 \\ 0. & 6 & 4 & 3 & & 01 \\ \hline 10. & 3 & 6 & 3 & & 01 \end{array}$$

The result (103.63) is not a floating number. We can round the result to obtain $\boxed{1.04}\boxed{02}$ (104).

From this example we draw a first useful conclusion: *the result of any arithmetic operation is, in general, corrupted by roundoff errors.* Thus, the arithmetic result is different from the mathematical result.

If a, b are floating point numbers, and $a + b$ is the result of mathematical addition, we will denote by $a \oplus b$ the computed addition.

The fact that $a \oplus b \neq a + b$ has surprising consequences. Let c=$\boxed{9.99}\boxed{-1}$ (0.999). Then

$$(a \oplus b) \oplus c = \boxed{1.04}\boxed{02}(104),$$

while

$$a \oplus (b \oplus c) = \boxed{1.05}\boxed{02}(105)$$

(you can readily verify this). Unlike mathematical addition, computed addition is *not associative*!

Homework 0.13 *Show that computed addition is commutative, i.e.* $a \oplus b = b \oplus a$.

4.11 IEEE Arithmetic

The IEEE standard specifies that the result of an arithmetic operation (+,-,*,/) must be computed exactly and then rounded to nearest. In other words,

$$\begin{aligned} a \oplus b &= f\ell(a+b) \\ a \ominus b &= f\ell(a-b) \\ a \otimes b &= f\ell(a \times b) \\ a \oslash b &= f\ell(a/b)\ . \end{aligned}$$

The same requirement holds for square root, remainder, and conversions between integer and floating point formats: compute the result exactly, then round.

This IEEE convention completely specifies the result of arithmetic operations; operations performed in this manner are called *exactly*, or *correctly rounded*. It is easy to move a program from one machine that supports IEEE arithmetic to another. Since the results of arithmetic operations are completely specified, all the intermediate results should coincide to the last bit (if this does not happen, we should look for software errors!).

(Note that it would be nice to have the results of transcendental functions like *exp(x)* computed exactly, then rounded to the desired precision; this is however impractical, and the standard does NOT require correctly rounded results in this situation.)

Performing only correctly rounded operations seems like a natural requirement, but it is often difficult to implement it in hardware. The reason is that if we are to find first the exact result we may need additional resources. Sometimes it is not at all possible to have the exact result in hand - for example, if the exact result is a periodic number (in our toy system, 2.0/3.0 = 0.666...).

4.12 The Guard Digit

Is useful when subtracting almost equal numbers. Suppose $a = (1.0)_2 \times 2^0$ and $b = (1.11\ldots1)_2 \times 2^{-1}$, with 23 1's after the binary point. Both a and b are single precision floating point numbers. The mathematical result is $a - b = (1.0)_2 \times 2^{-24}$. It is a floating point number also, hence the numerical result should be identical to the mathematical result, $a \ominus b = f\ell(a-b) = a - b$.

When we subtract the numbers, we align them by shifting b one position to the right. If computer registers are 24-bit long, then we may have one of the following situations.

1. Shift b and "chop" it to single precision format (to fit the register), then subtract.

$$\begin{array}{rrcr} & 1.000 & \ldots & 000 \\ - & 0.111 & \ldots & 111 \\ \hline & 0.000 & \ldots & 001 \end{array}$$

The result is 2^{-23}, twice the mathematical value.

2. Shift b and "round" it to single precision format (to fit the register), then subtract.

$$\begin{array}{rrcr} & 1.000 & \ldots & 000 \\ - & 1.000 & \ldots & 000 \\ \hline & 0.000 & \ldots & 000 \end{array}$$

The result is 0, and all the meaningful information is lost.

3. Append the registers with an extra **guard bit**. When we shift b, the guard bit will hold the 23rd 1. The subtraction is then performed in 25 bits.

$$\begin{array}{rllll} & 1.000 & \ldots & 000 & [0] \\ - & 0.111 & \ldots & 111 & [1] \\ \hline & 0.000 & \ldots & 000 & [1] \end{array}$$

The result is normalized, and is rounded back to 24 bits. This result is 2^{-24}, precisely the mathematical value. Funny fact: Cray supercomputers lack the guard bit. In practice, many processors do subtractions and additions in extended precision, even if the operands are single or double precision. This provides effectively 16 guard bits for these operations. This does not come for free: additional hardware makes the processor more expensive; besides, the longer the word the slower the arithmetic operation is.

The following theorem (see David Goldberg, p. 160) shows the importance of the additional guard digit. *Let x, y be FP numbers in a FP system with β, d_m, d_e;*

- *if we compute $x - y$ using d_m digits, then the relative rounding error in the result can be as large as $\beta - 1$ (i.e. all the digits are corrupted!).*
- *if we compute $x - y$ using $d_m + 1$ digits, then the relative rounding error in the result is less than 2ϵ.*

Note that, although using an additional guard digit greatly improves accuracy, it *does not* guarantee that the result will be exactly rounded (i.e. will obey the IEEE requirement). As an example consider $x = 2.34 \times 10^2$, $y = 4.56$ in our toy FP system. In exact arithmetic, $x - y = 229.44$, which rounds to $f\ell(x-y) = 2.29 \times 10^2$. With the guard bit arithmetic, we first shift y and chop it to 4 digits, $\hat{y} = 0.045 \times 10^2$. Now $x - \hat{y} = 2.295 \times 10^2$ (calculation done with 4 mantissa digits). When we round this number to the nearest (even) we obtain 2.30×10^2, a value different from the exactly rounded result.

However, by introducing a second guard digit and a third, "sticky" bit, the result is the same as if the difference was computed exactly and then rounded (D.Goldberg, p. 177).

4.13 Special Arithmetic Operations

4.13.1 Signed zeros

Recall that the binary representation 0 has all mantissa and exponent bits zero. Depending on the sign bit, we may have $+0$ or -0. Both are legal, and they are *distinct*; however, if $x = +0$ and $y = -0$ then the comparison (x.EQ.y) returns `.TRUE.` for consistency.

The main reason for allowing signed zeros is to maintain consistency with the two types of infinity, $+\infty$ and $-\infty$. In IEEE arithmetic, $1/(+0) = +\infty$ and $1/(-0) = -\infty$. If we had a single, unsigned 0, with $1/0 = +\infty$, then $1/(1/-\infty) = 1/0 = +\infty$, and not $-\infty$ as expected.

There are other good arguments in favor of signed zeros. For example, consider the function $\tan(\pi/2 - x)$, discontinuous at $x = 0$; we can consistently define the result to be $\mp\infty$ based on the signum of $x = \pm 0$.

Signed zeros have disadvantages also; for example, with $x = +0$ and $y = -0$ we have that $x = y$ but $1/x \neq 1/y$!

$(a < b)$.OR.$(a = b)$.OR.$(a > b)$	True, if a, b FP numbers False, if one of them NaN
$+0 = -0$	True
$+\infty = -\infty$	False

Table 4.3: IEEE results to comparisons

4.13.2 Operations with ∞

The following operations with infinity are possible:

a/∞	$=$	$\begin{cases} 0, & \text{a finite} \\ \text{NaN}, & a = \infty \end{cases}$
$a * \infty$	$=$	$\begin{cases} \infty\ , & a > 0\ , \\ -\infty\ , & a < 0\ , \\ NaN, & a = 0\ . \end{cases}$
$\infty + a$	$=$	$\begin{cases} \infty\ , & a\text{finite}\ , \\ -\infty\ , & a = \infty\ , \\ \text{NaN}, & a = -\infty\ . \end{cases}$

4.13.3 Operations with NaN

Any operation involving NaN as (one of) the operand(s) produces NaN. In addition, the following operations "produce" NaN: $\infty + (-\infty)$, $0 * \infty$, $0/0$, ∞/∞, $\sqrt{-|x|}$, x *modulo* 0, ∞ *modulo* x.

4.13.4 Comparisons

The IEEE results to comparisons are summarized in Table 4.13.4.

4.14 Arithmetic Exceptions

One of the most difficult things in programming is to treat exceptional situations. It is desirable that a program handles exceptional data in a manner consistent with the handling of normal data. The results will then provide the user with the information needed to debug the code, if an exception occurred. The extra FP numbers allowed by the IEEE standard are meant to help handling such situations.

The IEEE standard defines 5 exception types: division by 0, overflow, underflow, invalid operation and inexact operation.

4.14.1 Division by 0

If a is a floating point number, then IEEE standard requires that

$$a/0.0 = \begin{cases} +\infty\ , & \text{if } a > 0\ , \\ -\infty\ , & \text{if } a < 0\ , \\ \text{NaN}\ , & \text{if } a = 0. \end{cases}$$

If $a > 0$ or $a < 0$ the ∞ definitions make mathematical sense. Recall that $\pm\infty$ have special binary representations, with all exponent bits equal to 1 and all mantissa bits equal to 0.

If $a = 0$, then the operation is $0/0$, which makes no mathematical sense. What we obtain is therefore invalid information. The result is the **"Not a Number"**, in short **NaN**. Recall that NaN also have a special binary representation. NaN is a red flag, which tells the user that something wrong happened with the program. ∞ may or may not be the result of a bug, depending on the context.

4.14.2 Overflow

Occurs when the result of an arithmetic operation is finite, but larger in magnitude than the largest FP number representable using the given precision. The standard IEEE response is to set the result to $\pm\infty$ (round to nearest) or to the largest representable FP number (round toward 0). Some compilers will trap the overflow and abort execution with an error message.

Example (Demmel 1984, from D. Goldberg, p. 187, adapted): In our toy FP system let's compute

$$\frac{2\times 10^{23} + 10^{23}\,\mathbf{i}}{2\times 10^{25} + 10^{25}\,\mathbf{i}}$$

whose result is 1.00×10^{-2}, a "normal" FP number. A direct use of the formula

$$\frac{a + b\,\mathbf{i}}{c + d\,\mathbf{i}} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}\,\mathbf{i}$$

returns the result equal to 0, since the denominators overflow. Using the scaled formulation

$$\xi = \frac{d}{c}\ ; \quad \frac{a + b\,\mathbf{i}}{c + d\,\mathbf{i}} = \frac{a + b\xi}{c + d\xi} + \frac{b - a\xi}{c + d\xi}\,\mathbf{i}$$

we have $\xi = 0.5$, $(a + b\xi)/(c + d\xi) = (2.5\times 10^{23})/(2.5\times 10^{25}) = 0.01$ and $b - a\xi = 0$.

Sometimes overflow and infinity arithmetic mat lead to curious results. For example, let $x = 3.16\times 10^{25}$ and compute

$$\frac{x^2}{/}(x + 1.0\times 10^{23})^2 = 9.93\times 10^{-1}$$

Since the denominator overflows it is set to infinity; the numerator does not overflow, therefore the result is 0!. If we compute the same quantity as

$$\left(\frac{x}{x + 1\times 10^{23}}\right) = \left(\frac{3.16}{3.17}\right) = 0.99$$

we obtain a result closer to the mathematical value.

4.14.3 Underflow

Occurs when the result of an arithmetic operation is smaller than the smallest *normalized* FP number which can be stored. In IEEE standard the result is a *subnormal* number ("gradual" underflow) or 0, if the result is small enough. Note that subnormal numbers have fewer bits of precision than normalized ones, so using them may lead to a *loss of accuracy*. For example, let

$$x = 1.99\times 10^{-40}\ , \quad y = 1.00\times 10^{-11}\ , \quad z = 1.00\times 10^{+11}\ ,$$

and compute $t = (x \otimes y) \otimes z$. The mathematical result is $t = 1.99\times 10^{-40}$. According to our roundoff error analysis, we expect the calculated t to satisfy

$$\hat{t}_{\text{expected}} = (1 + \delta)t_{\text{exact}}\ , \quad |\delta| \approx \epsilon\ ,$$

IEEE Exception	Operation Result
Invalid Operation	NaN
Division by 0	$\pm\infty$
Overflow	$\pm\infty$ (or FPmax)
Underflow	0 or subnormal
Precision	rounded value

Table 4.4: The IEEE Standard Response to Exceptions

where the bound on delta comes from the fact that we have two floating point multiplications, and (with exact rounding) each of them can introduce a roundoff error as large as the half the machine precision $|\delta_\otimes| \le \epsilon/2$:

$$\begin{array}{rcl} x \otimes y &=& (1+\delta_\otimes^1)(x \times y) \\ (x \otimes y) \otimes z &=& (1+\delta_\otimes^2)\,[(x \otimes y) \times z] \\ &=& (1+\delta_\otimes^2)(1+\delta_\otimes^1)\,[x \times y \times z] \\ &\approx& (1+\delta_\otimes^1+\delta_\otimes^2)\,[x \times y \times z] \\ &\le& (1+\epsilon)\,[x \times y \times z] \end{array}$$

Since in our toy system $\epsilon = 0.01$, we expect the computed result to be in the range

$$\hat{t}_{\text{expected}} \in [(1-2\epsilon)t_{\text{exact}},\ (1+2\epsilon)t_{\text{exact}}] = [1.98 \times 10^{-40},\ 2.00 \times 10^{-40}]\ .$$

However, the product $x \otimes y = 1.99 \times 10^{-51}$ underflows, and has to be represented by the subnormal number 0.01×10^{-49}; when multiplied by z this gives $\hat{t} = 1.00 \times 10^{-40}$, which means that the relative error is almost 100 times larger than expected

$$\hat{t} = 1.00 \times 10^{-40} = (1+\hat{\delta})t_{\text{exact}}\ , \quad \hat{\delta} = 0.99 = 99\epsilon\ !$$

4.14.4 Inexact

Occurs when the result of an arithmetic operation is inexact. This situation occurs quite often!

4.14.5 Summary

The IEEE standard response to exceptions is summarized in Table 4.14.5.

4.15 Flags and Exception Trapping

Each exception is signaled by setting an associate status flag; the flag remains set until explicitly cleared. The user is able to read and write the status flags. There are 5 status flags (one for each possible exception type); in addition, for each flag there is a trap enable bit (see below), and there are 4 rounding modes bits. If the result of a computation is, say, $+\infty$, the flag values help user decide whether this is the effect of an overflow or is a genuine infinity, like 1/0.

The programmer has the option to

- *Mask the exception.* The appropriate flag is set and the program continues with the standard response shown in the table;

- *Trap the exception.* Occurrence of the exception triggers a call to a special routine, the trap handler. Trap handlers
  - receive the values of the operands which lead to the exception, or the result;
  - clear or set the status flag; and
  - return a value that will be used as the result of the faulty operation.

Using trap handler calls for each inexact operation is prohibitive.

For overflow/underflow, the argument to the trap handler is the result, with a modified exponent (the "wrapped-around" result). In single precision the exponent is decreased/increased by 192, and in double precision by 1536, followed by a rounding of the number to the corresponding precision.

Trap handlers are useful for backward compatibility, when an old code expects to be aborted if exception occurs. Example (from D. Goldberg, page 189): without aborting, the sequence

$$\texttt{doSuntil}(\texttt{x} >= 100)$$

will loop indefinitely if `x` becomes NaN.

4.16 Systems Aspects, from D. Goldberg, p. 193

The design of computer systems requires in-depth knowledge about FP. Modern processors have special FP instructions, compilers must generate such FP instructions, and the operating system must handle the exception conditions generated by these FP instructions.

4.16.1 Instruction Sets

It is useful to have a multiplication of single precision operands (p mantissa digits) that returns a double precision result (2p mantissa digits). All calculations require occasional bursts of higher precision.

4.16.2 Ambiguity

A language should define the semantics precisely enough to prove statements about the programs. Common points of ambiguity:

- x=3.0/10.0 FP number, it is usually not specified that all occurrences of 10.0*x must have the same value.
- what happens during exceptions.
- interpretation of parenthesis.
- evaluation of subexpressions. If `x` real and m,n integers, in the expression `x+m/n` is the division integer or FP? For example, we can compute all the operations in the highest precision present in the expression; or we can assign from bottom up in the expression graph tentative precisions based on the operands, and then from top down assign the maximum of the tentative and the expected precision.
- defining the exponential consistently. Ex: (-3)**3 = -27 but (-3.0)**(3.0) is problematic, as it is defined via logarithm. Goldberg proposes to consider $f(x) \to a$, $g(x) \to b$ as $x \to 0$. If $f(x)^{g(x)} \to c$ for all f, g then $a^b = c$. For example, $2^\infty = \infty$, but $1^\infty = NaN$ since $1^{1/x} \to 1$ but $(1-x)^{1/x} \to e^{-1}$.

4.16.3 Programming Language Features

The IEEE standard says nothing about how the features can be accessed from a programming language. There is usually a mismatch between IEEE-supporting hardware and programming languages. Some capabilities, like exactly rounded square root, can be accessed through a library of function calls. Others are harder:

- The standard requires extended precision, while most languages define only single and double.
- Languages need to provide subroutines for reading and writing the state (exception flags, enable bits, rounding mode bits, etc).
- Cannot define $-x = 0 - x$ since this is not true for $x = +0$;
- NaN are unordered, therefore when comparing 2 numbers we have $<, >, =, unordered$.
- The precisely defined IEEE rounding modes may conflict with the programming language's implicitly-defined rounding modes or primitives.

4.16.4 Optimizers

Consider the following code for estimating the machine ϵ

```
eps = 1.0; do eps=0.5*eps; while (eps+1 > 1);
```

If the compiler "optimizes" `(eps+1 > 1)` to `(eps > 0)` the code will compute the largest number which is rounded to 0.

Optimizers should be careful when applying mathematical algebraic identities to FP variables. If, during the optimization process, the expression $x+(y+z)$ is changed to $(x+y)+z$, the meaning of the computation is different.

Converting constants like $1.0E-40*x$ from decimal to binary at compile time can change the semantic (a conversion at run time obeys the current value of the IEEE rounding modes, and eventually raise the inexact and underflow flags).

Semantics can be changed during common subexpression elimination. In the code

```
C = A*B; RndMode = Up; D = A*B;
```

$A*B$ is not a common subexpression, since it is computed with different rounding modes.

4.16.5 Exception Handling

When an operation traps, the conceptual model is that everything stops and the trap handler is executed; since the underlying assumption is that of serial execution, traps are harder to implement on machines that use pipelining or have multiple ALU. Hardware support for identifying exactly which operation did trap may be needed. For example,

```
x = y*z; z = a+b;
```

both operations can be physically executed in parallel; if the multiplication traps, the handler will need the value of the arguments `y` and `z`. But the value of `z` is modified by the addition, which started in the same time as the multiply and eventually finished first. IEEE supporting systems must either avoid such a situation, or provide a way to store `z` and pass the original value to the handler, if needed.

4.17 Long Summations

Long summations have a problem: since each individual summation brings an error of 0.5 ulp in the partial result, the total result can be quite inaccurate. Fixes

- compute the partial sums in a higher precision;
- sort the terms first;
- use Kahan's formula.

4.18 Tipical pitfalls with floating point programs

All numerical examples in this section were produced on an Alpha 21264 workstation. On other systems the results may vary, but in general the highlighted problems remain the same.

4.18.1 Binary versus decimal

Consider the code fragment

```
PROGRAM test
  REAL :: x=1.0E-4
  PRINT*, x
END PROGRAM test
```

We expect the answer to be $1.0E-4$, but in fact the program prints $9.9999997E-05$. Note that we did nothing but store and print! The "anomaly" comes from the fact that 0.0001 is converted (inexactly) to binary, then the stored binary value is converted back to decimal for printing.

4.18.2 Floating point comparisons

Because of the inexactities, it is best to avoid strict equality when comparing floating point numbers. For the above example, the code

```
IF ( (1.0 E+8*x**2) == 1.0 ) THEN
   PRINT*, 'Correct'
END IF
```

should print `''Correct''`, but does not, since the left expression is corrupted by roundoff. The right way to do floating point comparisons is to define the epsilon machine, `eps`, and check that the magnitude of the difference is less than half epsilon times the sum of the operands:

```
epsilon = 1.0E-7
w = 1.0E+8 * x**2
IF ( abs(w-1.0) .LE. 0.5*epsilon*( abs(w)+abs(1.0) ) ) THEN
   PRINT*, 'Correct'
END IF
```

This time we allow small roundoff's to appear, and the program takes the right branch.

In the following example the branch correct is taken:

```
PROGRAM quiz_2b
  IMPLICIT NONE
  REAL :: x
  x = 1.0/2.0
  IF ( (2.0*x) .eq . 1.0 )  THEN
     PRINT*, 'Correct'
  ELSE
     PRINT*, 'Funny'
  END IF
END PROGRAM quiz_2b
```

while in the next the branch incorrect is taken:

```
PROGRAM quiz_2a
  IMPLICIT NONE
  REAL :: x
  x = 1.0/3.0
  IF ( (3.0*x) .eq . 1.0 )  THEN
     PRINT*, 'Correct'
  ELSE
     PRINT*, 'Funny'
  END IF
END PROGRAM quiz_2a
```

4.18.3 Funny conversions

Sometimes the inexactness in floating point is uncovered by real to integer conversion, which by Fortran default is done using truncation. For example the code

```
PROGRAM test
  REAL :: x = 1.0E-4
  INTEGER :: i
  i = 10000*x
  PRINT *, i
END PROGRAM test
```

produces a stunning result: the value of i is 0, not 1!

Another problem appears when a single precision number is converted to double precision. This does not increase the accuracy of the number. For example the code

```
PROGRAM test
  REAL :: x = 1.234567
  DOUBLE PRECISION :: y = 0.0D0
  y = x
  PRINT *, 'X =',x,'   Y =',y
END PROGRAM test
```

produces the output

```
 X=  1.234567         Y=  1.23456704616547
```

The explanation is that, when converting single to double precision, register entries are padded with zeros in the binary representation. The double precision number is printed with 15 positions and the inexactity shows up. (if we use a formatted print for `x` with 15 decimal places we obtain the same result). In conclusion, we should only print the number of digits that are significant to the problem.

4.18.4 Memory versus register operands

The code

```
        DATA a /3.0/, b /10.0/
        DATA x /3.0/, y /10.0/
        z = (y/x)−(b/a)
        CALL ratio(x,y,a1)
        CALL ratio(a,b,a2)
        CALL sub(a2,a1,c)
        PRINT*, z−c
```

may produce a nonzero result. This is so because `z` is computed with register operands (and FP registers for Pentium are in extended precision, 80 bits) while for `c` the operands `a` and `b` are stored in the memory. (note that the Alpha compiler produces zero).

4.18.5 Cancellation ("Loss-of Significance") Errors

When subtracting numbers that are nearly equal, the most significant digits in the operands match and cancel each other. This is no problem if the operands are exact, but in real life the operands are corrupted by errors. In this case the cancellations may prove catastrophic.

For example, we want to solve the quadratic equation

$$a\,x^2 + b\,x + c = 0 \ ,$$

where all the coefficients are FP numbers

$$a = 1.00 \times 10^{-3},\ b = 1.00 \times 10^{0},\ c = 9.99 \times 10^{-1} \ ,$$

using our toy decimal FP system and the quadratic formula

$$r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \ .$$

The true solutions are $r_1 = -999$, $r_2 = -1$. In our FP system $b^2 = 1.00$, $4ac = 3.99 \times 10^{-3}$, and $b^2 - 4ac = 1.00$. It is here where the cancellation occurs! Then $r_1 = (-1-1)/(2 \times 10^{-3}) = -1000$ and $r_2 = (-1+1)/(2 \times 10^{-3}) = 0$. If the error in r_1 is acceptable, the error in r_2 is 100%!

The same happens in single precision:

```
REAL :: a=1.0, b=-1.0E+8, c=9.999999E+7
d  = sqrt(b**2-4.0*a*c)
r1 = (-b+d)/(2.0*a)
r2 = (-b-d)/(2.0*a)
e2 = (2.0*c)/(-b+d)
```

The exact results are -1 and $-c$, and we expect the numerical results to be close approximations. We have $b**2 = 1.0E+16$ and $4ac = 3.9999997E+08$; due to cancellation errors the computed value of d is $d = 1.0E+8$. Then $r1 = 0$ and $r2 = 1.0E+8$.

With

```
a=1.0E-3,b=-9999.999,c=-1.0E+4
```

The exact results are -1 and $1.0E+7$. d is calculated to be $1.d+4$, and $b-d$ suffers from cancellation errors. The numerical roots are $1.0E+07$ (exact!) and -0.9765624 (about 2.5% relative error, much higher than the expected $1.0e-7$!).

To overcome this, we might avoid the cancellation by using mathematically equivalent formulas:

$$e_{1,2} = \frac{2c}{-b \mp \sqrt{b^2-4ac}} \ .$$

With this formula, $r_2 = (2c)/(-2) = -9.99 \times 10^{-1}$, a much better approximation. For the second example $e2 = 0.9999999$. For the third example the root is -1 (exact).

4.18.6 Insignificant Digits

Consider the Fortran code

```
PROGRAM test
  REAL :: x=100000.0, y=100000.1, z
  z = y-x
  PRINT*, 'z=',z
END PROGRAM test
```

We would expect the output

$$Z = 0.1000000$$

but in fact the program prints (on Alpha ...)

$$Z = 0.1015625$$

Since single precision handles about 7 decimal digits, and the subtraction $z = y - x$ cancels the most significant 6, the result contains only one significant digit. The appended garbage `15625` are insignificant digits, coming from the inexact binary representation of `x` and `y`. Beware of convincing-looking results!

4.18.7 Order of Operations Matters

Mathematically equivalent expressions may give different values in floating point, depending on the order of evaluation. For example

```
PROGRAM test
 REAL :: x=12345.6, y=45678.9, z=98765432.1
```

```
  REAL :: w1, w2
  w1 = x*y/z
  w2 = y*(x*(1.0/z))
  PRINT*, w1−w2
END PROGRAM test
```

Mathematically, the difference between `w1` and `w2` should be zero, but on Alpha ... it is about $-4.e-7$.

4.19 Integer Multiplication

As another example, consider the multiplication of two single-precision, FP numbers.

$$(m_1 \times 2^{e_1}) \cdot (m_2 \times 2^{e_2}) = (m_1 \cdot m_2) \times 2^{e_1+e_2} \ .$$

In general, the multiplication of two 24-bit binary numbers $(m_1 \cdot m_2)$ gives a 48-bit result. This can be easily achieved if we do the multiplication in double precision (where the mantissa has 53 available bits), then round the result back to single precision.

However, if the two numbers to be multiplied are double-precision, the exact result needs a 106-bit long mantissa; this is more than even extended precision can provide. Usually, multiplications and divisions are performed by specialized hardware, able to handle this kind of problems.

4.20 Homework

Homework 0.14 *The following program computes a very large FP number in double precision. When assigning* `a=b`*, the double precision number (b) will be converted to single precision (a), and this may result in overflow. Compile the program with both Fortran90 and Fortran77 compilers. For example,* `f90 file.f -o a90.out` *and* `f77 file.f -o a77.out`*. Run the two versions of the program for $p = 0, 0.1, 0.01$. Does the Fortran90 compiler obey the IEEE standard? For which value the single precision overflow occurs? How about the Fortran77 compiler? Note that, if you do not see* `Normal End Here !` *and* `STOP` *the program did not finish normally; trapping was used for the FP exception. If you see them, masking was used for the exception and the program terminated normally. Do the compilers use masking or trapping?*

```
  PROGRAM test_overflow
    REAL :: a, p
    DOUBLE PRECISION :: b
    PRINT*, 'please provide p:'
    READ*, p
    b = (1.99d0+p)*(2.d0**127)
    PRINT*, b
    a = b
    PRINT*, a
    PRINT*, 'normal END here !'
  END PROGRAM test_overflow
```

Homework 0.15 *The following program computes a small FP number (2^{-p}) in single and in double precision. Then, this number is multiplied by 2^p. The theoretical result of this computation is 1. Run the code for $p = 120$ and for $p = 150$. What do you see? Does the Fortran90 compiler obey the IEEE standard? Repeat the compilation with the Fortran77 compiler, and run again. Any differences?*

```
  PROGRAM test_underflow
    REAL :: a,b
    DOUBLE PRECISION :: c,d
    INTEGER :: p
    PRINT*, 'please provide p'
    READ*, p
    c = 2.d0**(-p)
    d = (2.d0**p)*c
    PRINT*, 'DOUBLE PRECISION: ', d
    a = 2.e0**(-p)
    b = (2.d0**p)*a
    PRINT*, 'single PRECISION: ', b
    PRINT*, 'normal END here !'
  END PROGRAM test_underflow
```

Homework 0.16 *The following program performs some messy calculations, like division by 0, etc. Compile it with both Fortran90 and Fortran77 compilers, and run the two executables. What do you see? Any differences?*

```
PROGRAM test_arit
    REAL :: a, b, c, d
    c = 0.0
    d = -0.0
    PRINT*, 'c=',c,'  d=',d
    a = 1.0/c
    PRINT*, a
    b = 1.0/d
    PRINT*, 'a=',a,'  b=',b
    PRINT*, 'a+b=',a+b
    PRINT*, 'a-b=',a-b
    PRINT*, 'a/b=',a/b
END PROGRAM test_arit
```

Chapter 5

Applications Part I.

5.1 The Fibonacci Sequence

The Fibonacci sequence is defined by

$$x_0 = 1,\ x_1 = 1,\ x_2 = 2,\ ,x_3 = 3,\ \ldots\ x_{i+1} = x_i + x_{i-1}, \ldots$$

One characteristic of the sequence is that the ratio $(x_{i+1}\, x_{i-1})/x_i^2$ tends to the Golden Ration

$$\frac{x_{i+1}\, x_{i-1}}{x_i^2}\ \overset{\Longrightarrow}{i \to \infty}\ \frac{1+\sqrt{5}}{2}\ .$$

We want to write a program that computes the first 50 terms in the Fibonacci sequence; at each step we check whether the above ratio is sufficiently close to the Golden Ratio; and if it is, we print a message and end the computations.

The implementation follows.

```
PROGRAM fibonacci
!
  IMPLICIT NONE
  REAL :: x0=1.0, x1=1.0, x2
  REAL :: ratio , golden, tol=1.e-6
  INTEGER :: i , n_max=50
!
  golden = (1.0+sqrt(5.0))/2.0
!
  DO i=2,n_max
    x2 = x0+x1
    PRINT *,"x",i," = ",x2
    ratio = x2/x1
    IF ( abs(ratio-golden) .lt . tol ) THEN
      PRINT *,"ratio did converge at i = ",i
      EXIT
    END IF
    x0=x1; x1=x2
  END DO
!
```

```
  IF ( i .gt. n_max ) THEN
     PRINT *,"ratio did not converge, i = ",i
  END IF
!
END PROGRAM fibonacci
```

During each sweep through the DO loop we compute and print the new element in the sequence, x2. Then, we calculate the x2/x1 ratio, and compare it to the golden ration; if they are suffieciently close we exit the loop. If the loop is terminated normally, the counter is n_max+1, and in this case the ratio did not converge. At the end of each sweep we prepare for the next iteration by assigning x1, x2 to x0,x1).

5.2 Summation Formulas

Consider the summation

$$s = \sum_{i=1}^{n} x_i$$

On a computer, the direct evaluation

```
s = 0.0
DO i=1,n
 s = s + x(i)
END DO
```

leads to the succesive computation of the partial sums $s_1 = x_1$, $s_2 = s_1 \oplus x_2$, ..., $s_i = s_{i-1} \oplus x_i$. Each step is corrupted by roundoff error

$$s_i = s_{i-1} \oplus x_i = (s_{i-1} + x_i)\,(1+\delta_i) \ ,$$

and these errors accumulate from one step to another. In our case, the effect of δ_i is present in the $i+1^{\text{st}}$ step

$$\begin{aligned} s_{i+1} &= s_i \oplus x_{i+1} = (s_i + x_{i+1})\,(1+\delta_{i+1}) \\ &\approx (s_i + s_{i-1} + x_i)\,(1 + \delta_{i+1} + \delta_{i+1}) \end{aligned}$$

(we have ignored the terms containing $\delta_i\delta_{i+1}$).

In general, the computed sum is

$$s_n = \sum_{i=1}^{n} x_i \left(1 + \sum i = 1^n \delta_i\right)$$

In general, the relative order can be as large as $n\,\epsilon$; in single precision $\epsilon \sim 10^{-7}$, so computing a long sum of $n \sim 10^7$ terms may give a meaningless result.

To minimize the accumulation of roundoff, we consider 2 strategies. First, we try to minimize δ_i at each step. For this, the partial sum s_i and the element x_i must be of comparable size; therefore, sorting the terms x_i in increasing order before summation may help to reduce the individual roundoffs.

The second strategy is more elaborate, and is called the *Kahan summation algorithm*. The idea is to account for the errors at each step and to add the proper corrections as we go along. At step i we have

$$s_i = s_{i-1} \oplus x_i = (s_{i-1} + x_i)\,(1+\delta_i) \ ;$$

the error made is $(s_{i-1}+x_i)\,\delta_i$; we estimate this roundoff error by subtracting s_{i-1} and then x_i from the result

$$s_i = s_{i-1} \oplus x_i c = (s_i \ominus s_{i-1}) \ominus x_i$$

To correct the result, we need to subtract the error c from the sum. Since c is a small number, and the partial sum s_i can be quite large, in general applying the correction directly to s_i is useless: $s_i \ominus c = s_i$! However, we can apply the correction to the next element x_{i+1} (which can be much smaller). The next step reads

$$\begin{aligned} y &= x_{i+1} \ominus c \\ s_{i+1} &= s_i \oplus y \end{aligned}$$

Below, we give an example of a program which evaluates the sum

$$s = 1 + \left(\frac{1}{\text{base}} + \frac{1}{\text{base}} + \ldots\right) + \ldots + \left(\frac{1}{\text{base}^{\text{p\_max}}} + \frac{1}{\text{base}^{\text{p\_max}}} + \ldots\right)$$

where each paranthesis $\left(\frac{1}{\text{base}^{\text{p}}} + \ldots\right)$ contains exactly base^{p} terms, and therefore evaluates to 1; the exact result is $s = p\_max + 1$.

Note the following characteristics of the implementation. They give a flavor of how to implement large programs.

- each major function (terms initialization, computation of the sum using sorted up, sorted down and Kahan formula, and printing of the results) is handled by a different procedure. Each procedure is written and tested independently.
- The main program just calls the individual procedures in the appropriate order; it is therefore easy to keep track of the program flow by simply reading the main.
- Global data (summation terms, and sum values) are defined in a module, which is used in all the subroutines.

```
! Long summations example
! different algorithms for performing summation

MODULE terms
  save
!     Numeration basis
  INTEGER, PARAMETER :: base=10
!     sum = 1 + base*(.) + ... + base**p_max*(.)
  INTEGER, PARAMETER :: p_max=7
!     Number of terms in the sum
  INTEGER, PARAMETER :: n_max=(base**(p_max+1)-1)/(base-1)
  REAL, DIMENSION(n_max) :: x
  REAL :: exact, kahan, sortu, sortd
END MODULE terms

SUBROUTINE init
  USE terms
  IMPLICIT NONE
  INTEGER :: i, j, k=0
!
  DO i = 0,p_max
```

```
    DO j = 1,base**i
      x(k+j) = 1.0/base**(i)
    END DO
    k = k+base**i
  END DO
!
END SUBROUTINE init

SUBROUTINE sum_exact
!
! the exact value of the sum
!
  USE terms
  IMPLICIT NONE
  exact = p_max+1
END SUBROUTINE sum_exact

SUBROUTINE sum_sortd
!
! Performs normal summation,
! with the terms sorted in decreasing order
!
  USE terms
  IMPLICIT NONE
  INTEGER :: i
  sortd = 0.0
  DO i = 1,n_max
    sortd = sortd + x(i)
  END DO
END SUBROUTINE sum_sortd

SUBROUTINE sum_sortu
!
! Performs normal summation,
! with the terms sorted in increasing order
!
  USE terms
  IMPLICIT NONE
  INTEGER :: i
  sortu = 0.0
  DO i = n_max,1,-1
    sortu = sortu + x(i)
  END DO
END SUBROUTINE sum_sortu

SUBROUTINE sum_kahan
!
! Performs the summation using Kahan's algorithm
!
  USE terms
  IMPLICIT NONE
  INTEGER :: i
  REAL :: c, y, s
  kahan = x(1)
  c = 0.0
  DO i = 2,n_max
    y = x(i) - c
    s = kahan + y
    c = (s-kahan) - y
    kahan = s
  END DO
END SUBROUTINE sum_kahan

SUBROUTINE print_results
!
! Prints the  setting , the  results  and their  accuracies
!
  USE terms
  IMPLICIT NONE
  PRINT *, "basis = ",base,", maximal power = ",p_max
  PRINT *, "no. of terms in the sum = ", n_max
  PRINT *, "sort down: ",sortd," error = ",(sortd-exact)/exact
  PRINT *, "sort up: ",sortu," error = ",(sortu-exact)/exact
  PRINT *, "kahan:   ",kahan," error = ",(kahan-exact)/exact
END SUBROUTINE print_results

PROGRAM sum
  USE terms
  IMPLICIT NONE
!
  CALL init
  CALL sum_exact
  CALL sum_sortd
  CALL sum_sortu
  CALL sum_kahan
  CALL print_results
!
END PROGRAM sum
```

The results of this program show clearly the advantage of Kahan summation.

```
! BASIS = 10,  Maximal Power = 7
! No. of terms in the Sum = 11111111
! SORT DOWN: 6.95631695  Error = -0.130460382
! SORT UP:   8.01876831  Error =  2.346038818E-3
! KAHAN:     8.          Error =  0.E+0
```

Chapter 6

Intrinsic Functions

F90 offers efficient implementations for some of the most frequently performed tasks in programming. These implementations are available as functions, built the F90 language, and named ''intrinsics''. For example, taking absolute value can be done by calling the intrinsic function ABS(x). Note that ABS is built in the F90 language (and therefore it is not necessary to include any special header, or link the program to a mathematical library to perform ABS).

6.1 Generic vs. Specific Functions

F90 intrinsic functions are usually *generic*; this means that they can accept arguments of different types and return a value appropriate for that type of argument. For example, ABS(x) accepts an INTEGER, REAL, DOUBLE PRECISION or COMPLEX argument x, and the returned value is of the same type as x. In addition to the *generic name*, there are also *specific names* which depend on the argument type. For example, IABS, ABS, DABS, CABS are the absolute value functions for INTEGER, REAL, DOUBLE PRECISION and COMPLEX type arguments.

Why does F90 allow for both generic and specific names? First, generic names are easier to remember and use. When we write

```
abs(x)
```

we understand that the compiler checks the type of the argument x. If it is is integer, then the compiler will substitute ABS (the generic name) with a call to IABS (the specific name); and similar for other types.

However, there are situations when the name of the function is not followed by arguments. For example, we can write a user-defined plot subroutine, which takes an intrinsic function f(x), evaluates it at $x = -1.0, \ldots, 1.0$ and plots the graph (x,f(x)). We will invoke our subroutine with, say,

```
CALL my_plot( abs )
```

to plot (x,ABS(x)) for x between -1 and 1, and will invoke it as

```
CALL my_plot( sin )
```

to plot (x,SIN(x)) for x between -1 and 1. F90 allows the use of function names (ABS, SIN) as arguments in (another) procedure call. In these instances, the intrinsic function name is not followed by arguments. The compiler does not know which absolute value function to use (will ABS be called with integer, real or complex arguments in MY_PLOT? What if MY_PLOT is in a different file, which is compiled separately?) In consequence, it is the user's responsability to replace the generic name with the secific one when the intrinsic function name is a procedure argument.

6.2 Numerical Type Conversions

In what follows we will use the short notation I,R,D,C for INTEGER, REAL, DOUBLE PRECISION and COMPLEX.

- INT(x); converts a R, D or C argument (x) to the corresponding INTEGER approximation using ''round towards 0'' (dumps the fractional part). For a complex argument, INT(real part) is taken.
- CEILING(x); converts a R or D argument (x) to the corresponding INTEGER approximation using ''round up''.
- FLOOR(x); converts a R or D argument (x) to the corresponding INTEGER approximation using ''round down''.
- NINT(x); converts a R or D argument (x) to the nearest INTEGER (using ''round to nearest'').

For example,

$$\begin{array}{lcrclcr} \texttt{INT}(1.7) & \equiv & 1 & ; & \texttt{INT}(-1.7) & \equiv & -1 \\ \texttt{FLOOR}(1.7) & \equiv & 1 & ; & \texttt{FLOOR}(-1.7) & \equiv & -2 \\ \texttt{CEILING}(1.7) & \equiv & 2 & ; & \texttt{CEILING}(-1.7) & \equiv & -1 \\ \texttt{NINT}(1.7) & \equiv & 2 & ; & \texttt{NINT}(-1.7) & \equiv & -2 \end{array}$$

- REAL(k); converts a I or D argument (k) to the corresponding REAL approximation. Selects the real part of a C argument.
- AIMAG(z); selects the imaginary part of a C argument.
- DBLE(x); converts an INTEGER or REAL argument (x) to the corresponding DOUBLE PRECISION approximation.
- CMPLX(x,y); returns the complex value x + $\mathbf{i}\cdot$y. The arguments x,y are usually REAL, but can also be INTEGER or DOUBLE PRECISION, in which case the returned value is REAL(x) + $\mathbf{i}\cdot$REAL(y). CMPLX(x) returns the complex value REAL(x) + $\mathbf{i}\cdot 0$.

For example,

```
INTEGER :: i=2
REAL ::  t=1.7
DOUBLE PRECISION :: x=1.7d0,y=2.8d0
COMPLEX :: z=(1.7,2.8)
PRINT*, x, REAL(x) ! output = 1.7, 1.70000005
PRINT*, REAL(z), aimag(z) ! output = 1.70000005, 2.79999995
PRINT*, dble(t) ! output = 1.7000000476837158
PRINT*, cmplx(i), cmplx(z) ! output = (2.,0.e+0), (1.70000005,2.79999995)
```

Note the errors in double to real and real to double conversions, as well as the representation errors.

6.3 Numerical Intrinsic Functions

We will give the *generic* names for the functions, and mention the *specific* names if appropriate. Only most important such functions are given here.

- ABS; absolute value. I,R,D,C argument, and same type return value. For example,

  $$\begin{array}{lcl} \texttt{ABS}(-1) \equiv 1 & ; & \texttt{ABS}(-1.0) \equiv 1.0 \\ \texttt{ABS}(-1.\texttt{D0}) \equiv 1.D0 & ; & \texttt{ABS}(\texttt{CMPLX}(-3.0, 4.0)) \equiv 5.0 \ . \end{array}$$

  Specific names are IABS, ABS, DABS, CABS.

- AINT(x): truncates to a whole R (D) number. ANINT(x): nearest whole R (D) number. They are similar to INT and NINT, except that the returned result is R or D, depending on the argument type.

  $$\begin{array}{lcl} \texttt{AINT}(-1.7) \equiv -1.0 & ; & \texttt{ANINT}(-1.7) \equiv -2.0 \\ \texttt{AINT}(-1.7\texttt{D0}) \equiv -1.7D0 & ; & \texttt{ANINT}(-1.7\texttt{D0}) \equiv 2.D0 \ . \end{array}$$

  Specific names for D: DINT, DNINT.

- MAX(x,y,z,...), MIN(x,y,z,...): maximum or minimum from a list of arguments. Any number of arguments is allowed, but all of them need to be of the same type (I, R or D). Some specific names are MAX0 (I arguments, I result), AMAX1 (R arguments, R result), DMAX1 (D arguments, D result).

- MOD(i,j) is the remainder of the division i/j. MODULO(i,j) is i mod j. For example,

  $$\begin{array}{lcl} \texttt{MOD}(7,3) \equiv 1 & ; & \texttt{MODULO}(7,3) \equiv 1 \\ \texttt{MOD}(-7,3) \equiv -1 & ; & \texttt{MODULO}(-7,3) \equiv 2 \ . \end{array}$$

  Specific names are AMOD, DMOD.

- SIGN(x,y) returns $signum(y)*abs(x)$ (transfers the signum of second argument to the first). Specific names: ISIGN, DSIGN.

- FRACTION(x), EXPONENT(x): the mantissa and the exponent in the *binary* representation of a R or D argument x. For example,

  $$\begin{array}{l} \texttt{FRACTION}(3.141592) \equiv 0.785398 \\ \texttt{EXPONENT}(3.141592) \equiv 2 \ . \end{array}$$

- CONJG(z), complex conjugate of a complex number.

6.4 Mathematical Intrinsic Functions

- SIN(x), COS(x), TAN(x): mathematical sine, cosine and tangent. R or D arguments, meaning *angle expressed in radians*. Specific names: DSIN, CSIN, DCOS, CCOS, DTAN.

- ASIN(x), ACOS(x), ATAN(x): mathematical arc sine, arc cosine and arc tangent. R or D arguments (for ASIN, ACOS: $|x| \leq 1$). Result in radians. Specific names: DASIN, DACOS, DATAN.

- SINH(x), COSH(x), TANH(x): mathematical hyperbolic sine, hyperbolic cosine and hyperbolic tangent. R or D arguments. Specific names: DSINH, DCOSH, DTANH.

- LOG(x), LOG10(x), EXP(x), SQRT(x): natural logarithm, base 10 logarithm, exponential, square root. R,D,C arguments (with some restrictions). Specific names: DLOG, CLOG, DEXP, CEXP, DSQRT, CSQRT.

6.5 Examples

Write a program for polar to cartesian coordinate transformation.

Write a program to solve a quadratic equation.

Write a program to approximate the exponential by Taylor polynomials with a prescribed tolerance, but using an order lower than or equal to a prescribed value (when the order needed to satisfy a certain accuracy is too high, we stop at the prescribed maximal order and ignore the tolerance requirement). Check program's accuracy and speed against the intrinsic function EXP.

6.6 Mixing Types

6.6.1 Mixed-Type Expressions

Operations in ALU take place between operands of the same type. When mixed type operands appear in the same expression, it is up to the compiler to convert them to the proper, common type.

Default types obey the ordering

INTEGER	*weakest*
REAL	
DOUBLE PRECISION	
COMPLEX	*strongest*

In a mixed expression the weaker type operands are promoted to the stronger type; the result is of the stronger type.

For example, the mixed-type expression

```
i*r*dp*c
```

is evaluated left-to right (all ∗ have the same precedence). For the first ∗ the integer i is converted to real, and the result i*r is real; this result is then converted to double precision, and (i∗r)∗dp is a double precision result; finally, this result in its turn is converted to complex to be able to calculate ((i∗r)∗dp)∗c; the returned result is complex.

6.6.2 Mixed-Type Assignement

In an assignment

```
{\tt LHS=RHS},
```

where LHS and RHS have different types, the RHS expression is evaluated first using RHS type arithmetic; the result is then converted to the type of LHS.

Explicit type conversions are to be preffered to automatic ones. For example, the real to integer conversion is automatically done by truncation toward 0 (similar to INT intrinsic).

6.6.3 Integer Division

A word of caution needs to be said when the expression involves an integer division. Division of two integers produces an integer result, obtained by rounding toward zero the real result. For example,

$$\texttt{19/10} \equiv \texttt{1} \ , \quad \texttt{-19/10} \equiv \texttt{-1}.$$

Chapter 7

Input and Output.

7.1 List-Directed Input and Output

List directed input is read in from the input stream (for now, the standard input device, usually the keyboard) using

```
READ *, input_list
```

List directed output sends the values of the output list to the output stream (for now, the standard output, usually the display):

```
PRINT *, output_list
```

Conceptually, an input (or output) *stream* is a long sequence of characters, grouped in records; usually, one record = one line, and is ended by the end-of-line character; (it is possible to specify records of arbitrary length, however, they will be ended by the end-of-record character). The characters in a record are read (or written) in sequential order; each stream has a pointer which indicates the current position in the stream; the next read bring in the character whose position in the stream is specified by the stream pointere (next write puts a character in the stream, at the position indicated by the pointer).

To illustrate this, suppose we want a program that reads in the radii of 3 circles from the keyboard, and prints out the perimeters and the areas of the circles. This program might be

```
PROGRAM io
  REAL :: r1,r2,r3,a1,a2,a3,p1,p2,p3
  REAL, PARAMETER :: pi=3.1415926
  PRINT *,"give 3 radii"
  READ *, r1, r2, r3
  p1 = 2.0*pi*r1; p2 = 2.0*pi*r2; p3 = 2.0*pi*r3
  a1 = pi*r1*r1; a2 = pi*r2*r2; a3 = pi*r3*r3
  PRINT *,"areas = ",a1,a2,a3
  PRINT *,"perimeters = ",p1,p2,p3
END PROGRAM io
```

The first PRINT * statement sends the string ‘‘Give 3 radii’’ to the standard output device (presumably the display). The star * following the PRINT means that the output data is the edited using the implicit format.

The READ * statement reads 3 numerical values from the keyboard, and assigns them to r1, r2 and r3. When providing the numbers, the user can separate them by blanks or commas,

```
1.1  2.2  3.3   or   1.1, 2.2, 3.3
```

The end of the list is either the end of line character or a slash (/); therefore, if we enter

```
1.1  2.2 <EOL>   or    1.1 2.2 / 3.3
3.3
```

in either case the first 2 values will be read in and the 3rd will be ignored; r3 will be left unmodified by the READ statement.

The last PRINT * statement outputs the values of the three areas,

```
Areas = 3.80132723, 15.2053089, 34.2119408
```

Note that numerical values are rounded to 8 fractional digits and displayed, the default format of the current system. If we want to have more control over the form of the displayed quantities, we need to use formatted output; similarly, formatted input gives control over the read in data.

7.2 Formats

A console READ with format has the form

```
READ "(editor_descriptor_list)" input_list
```

(note that the star has been replaced by the actual format, that is, by string containing a list of editor descriptors, between paranthesis). The values for the input_list variables will be read in succesively, in the format specified by the corresponding descriptor from the editor_descriptor_list.

For example, we can replace the READ *, r1, r2, r3 statement by

```
READ "(F2.1,F2.1,F2.1)",r1,r2,r3
```

The Fw.d descriptor means that the next w characters of the input stream represent a Floating point number; if a decimal point is not present in the data, then the last d characters are the digits of the fractional part.

Therefore, an input line of the form

```
112233
```

will be interpreted as follows. The first edit descriptor in the list is F2.1; therefore, the first 2 characters (11) are read in from the input stream, and considered to represent a floating point number; since there is no decimal point in the data, the last character is the fractional part, therefore the value is 1.1 and is assigned to r1. At this time we are done with the first edit descriptor in the list, and we move to the second (also F2.1). The next 2 characters (22) are read in, and similarly, they are resolved to represent the floating point number 2.2, value which is assigned to r2. Similar treatment applies to the last edit descriptor.

Suppose now our input line has the easy-to-read form

```
1.1, 2.2, 3.3
```

(we have decimal points, and two consecutive numbers are separated by a comma and a blank). Then, the numbers are read in correctly if we use the following statement

```
READ "(F3.1,2X,F3.1,2X,F3.1)",r1,r2,r3
```

The first edit descriptor is F3.1; therefore, the first 3 characters (1.1) represent a floating point value, which is assigned to r1; since the data contains a decimal point, we ignore the edit specification of the fractional part. The edit descriptor nX skips the next n characters; therefore, the 2X descriptor will lead to ignoring the comma and the blank. Next edit desciptor, F3.1 will read in 3 characters, 2.2, which represent a real value and which is assigned to r2; etc.

Slashes in the format position the input stream pointer at the beginning of the next record (here, the next line); for example, if we read

```
READ "(F3.1,2X,F3.1,/,F3.1)",r1,r2,r3
```

and the input from the console is

```
1.1, 2.2, 3.3 <CR> \\
4.4
```

the assigned values are r1=1.1, r2=2.2, r3=4.4.

The formatted console PRINT has the form

```
PRINT "(editor_descriptor_list)" output_list
```

The values of the variables in the output_list will be printed out succesively, each one obeying the format imposed by the corresponding descriptor from the editor_descriptor_list.

We want to print the 3 computed areas; since the radii have just one fractional digit, it is probably sufficient to have the results rounded to 3 fractional digits.

We can use the following statement

```
PRINT "(A8,F7.3,2X,F7.3,2X,F7.3)","Areas = ",a1,a2,a3
```

The Aw edit desciptor specifies CHARACTER data of width w. The first object in the output list is the string "Areas = "; the first edit descriptor is A8, therefore the first 8 characters in the output stream will be Areas = . If we want to print the full string argument, without counting for its length, we can use the plain descriptor A (without any width specification).

The second object in the output list is the REAL a1, and it will be printed in a format specified by the second edit descriptor (F7.3; the number will be displayed on 7 characters, with 3 digits for the fractional part (the value of a1 will be *rounded* to 3 digits after the decimal point); Since there are 7 characters, we use 3 for the fractional part and one for the decimal point, the integer part will be displayed on the remaining 3 characters. The output is

```
Areas =  3.801  15.205  34.212
```

Note that if we initialize r1=20. (possible with 3 characters), the area a1=1256.637, a number whose integer part contains 4 digits; this number cannot be represented in the F7.3 format, and the produced output is ******* (7 stars).

In general, it is recommended that we allow a generous width w when printing numbers in the Fw.d format; if we allow for more places than the number needs, the representation will be aligned to the right, with blanks inserted in the leftmost positions (see 3.801 above). It is also recommended that The number of digits of the fractional part should be in tune with the number of accurate digits produced by the computation. For example, if we do the computations in single precision we have at most 7 accurate digits in the results; it is pointless to use a F20.10 format, which would print out 3 extra digits containing ‘‘garbage’’.

The 3rd descriptor is 2X; this will print 2 blanks in the output stream. From here on the behavior should be clear: print a2 in the F7.3 format, then insert 2 spaces, then print a3 in the same format.

The Fw.d format prints REAL values in the decimal point notation (integer part, point, decimal part). Alternatively, we can choose to print numbers in the exponent form. For this, we use the descriptor Ew.d, where w is the total number of characters (the width of the representation), and d is the number of decimal places in the normalized number; note that in this case d truly represents the number of digits printed. For example, the statement

```
PRINT "(a,e10.3,2x,e10.3,2x,e10.3)","areas = ",a1,a2,a3
```

produces the result

```
Areas = 0.380E+01  0.152E+02  0.342E+02
```

Note that the characters ., E, + are counted toward the total specified width of 10 places.

Let us mention that groups of descriptors can be repeated by enclosing the group between parantheses, and prefixing this expression with the number of repeats. For example, the print example above can be written in a shorter form as

```
PRINT "(a,3(e10.3,2x))","areas = ",a1,a2,a3
```

The string "Areas = " can be embedded as a constant in the editor descriptor list; the following statement produces the same output as the statement above

```
PRINT "('Areas = ',3(E10.3,2X))",a1,a2,a3
```

(note the interplay between double and single quotes in delimiting a string within a string).

Slashes in the format move the output stream pointer to the next record (line).

```
PRINT "('Areas = ',//,3(F7.3,/))",a1,a2,a3
```

produces the output

```
Areas =

  3.801
 15.205
 34.212
```

If the number of items in the output_list is greater than the number of edit descriptors in the format, the edit descriptor list is ‘‘wrapped around’’ and read again from the beginning. The statement

```
PRINT "('Areas = ',(F7.3,2X))",a1,a2,a3
```

produces the output

```
Areas =  3.801
Areas = 15.205
Areas = 34.212
```

7.2.1 Editor Descriptors

The most important edit descriptors in F90 are given below. This table is not comprehensive.

Iw	w places of integer data
Fw.d	w places of real data (in decimal point notation) including d decimal places
Ew.d	w places of real data (in scaled number-exponent notation) including d decimal places
Lw	w places of LOGICAL data
Aw	w places of CHARACTER data
A	READ: n places of CHARACTER data, enough to fill the input list item WRITE: print all characters of the string
wX	skip w places (for READ), or insert w spaces (for PRINT, WRITE)
Tn	move I/O stream pointer to position n
TLn	move I/O stream pointer n positions left
TRn	move I/O stream pointer n positions right

7.2.2 Example: READ Integer Data

Consider a read statement that initializes three INTEGER variables, i, j and k. The input string is

```
123456789
```

Depending on the format, the string can be interpreted differently; below are several examples of editor lists, and the resulting values of the variables i, j and k.

Statement	i	j	k
READ "(I2,I3,I4)",i,j,k	12	345	6789
READ "(I2,2X,I2,2X,I2)",i,j,k	12	56	9
READ "(I2,I3,T1,I4)",i,j,k	12	345	1234
READ "(I2,TR2,I3,TL3,I4)",i,j,k	12	567	5678

Note that Tw moves the input stream pointer to the w^{th} position of the current record/ current input line; TLw and TRw move the input stream pointer w positions to the left and right, respectively, inside the current record/ current input line.

7.2.3 Example: READ Character Data

Consider the length 10 character variables

```
CHARACTER(LEN=10) :: C1, C2
```

which are initialized by a READ statement. The input string is

```
michigantech
```

Depending on the format, the values assigned to A, B are different.

```
Statement                  C1          C2
READ "(A8,A4)",C1,C2    michigan◇◇    tech◇◇◇◇◇◇
READ "(A12,A12)",C1,C2  chigantech  ◇◇◇◇◇◇◇◇◇◇
READ "(A,A)",C1,C2      michigante    ch◇◇◇◇◇◇◇◇
```

(we denote the blank spaces by ◇). In case 1 the first 8 characters are read in C1 (which is padded with 2 additional blanks), and the remaining 4 characters are read in C2 (which is padded with additional 6 blanks). In case 2 the required width (12) is greater than the length of the string; the rightmost 10 characters are stored in C1, and there is no input left for C2 (which is therefore not changed). In case 3 we do not specify any width; the plain A descriptor reads in as many characters as needed to fill the variable (here, the first 10 characters fill in C1).

7.3 File I/O

So far we have considered only terminal I/O. Many times programs read large sets of input data from data files, and write large amounts of output results to another files; these results can be subsequently used by other programs, which, for example, analyze them. File data are permanent, in the sense that they remain after the program who produced them finishes.

7.3.1 Records and Files

A *record* is a sequence of characters and values; a sequence of records form a *file*. You can think of a file as a typed page, with the records being the lines of text.

A file can have sequential access (records are read one after the other) or random access (any record, specified by its number, can be accessed any time). The last record in a file is of special type, called the *endfile record* (it allows the automatic detection of the end of the file during reading).

In this part we talk about external files, stored on external medium - like the hard disk.

A program can read or write formatted or unformatted records.

Formatted records

- are composed of ASCII characters only
- information is in human-readable form
- when writing a formatted record, numbers are converted from internal, binary representation, to decimal, according to the edit descriptor specification. This conversion introduces roundoff, computational overhead, and results in a larger file.
- formatted data files are portable.

The following statement writes one formatted record; the format is explicitly given (F7.3)

```
WRITE (UNIT=10, FMT="f7.3") r1
```

Unformatted records

- store data in the binary form, as represented internally by the computer.
- information is not human-readable
- there is no roundoff, no computational overhead, and results in a smaller file.
- unformatted data files may not be portable, since the internal representation may be different on different computer systems; however, they are useful to temporarily store intermediate results during one execution.

The following statement writes one unformatted record; no format is given

```
WRITE (UNIT=10) r1
```

Unformatted data can be retrieved by unformatted read; the following statement reads one record; again, no format is specified.

```
READ (UNIT=10) x1
```

Note that unformatted files may not be portable from one system to another.

7.3.2 OPEN

F90 allows a number of different files (streams) to be connected to the program for both reading and writing. Before being used by a program, a file must be connected to a *logical unit* (the F90 program operates with logical units, designated by positive numbers, usually between 1 and 100).

Connecting a file to a logical unit is done using an OPEN statement.

```
OPEN (  UNIT= integer,         logical unit number
        FILE= ´filename',      name of the file
        STATUS='status',       'status' = 'OLD' looks for existing file
                               'NEW' creates the file, error if it already exists
                               'UNKNOWN' (is the default when status not specified)
                               ≡ (NEW if file exists)/(OLD if it doesn't)
                               'REPLACE' to override existing file
                               'SCRATCH' unnamed file, deleted at closing
        FORM='mode',           'mode' = 'FORMATTED' or 'UNFORMATTED'
        ACTION='action',       'READ' (only), 'WRITE' (only) or
                               'READWRITE' (default if ACTION missing)
        POSITION='position'    when opening, position the file pointer at
                               the beginning 'REWIND', the end 'APPEND', where it is ASSIS (default)
        ERR=label,             control is transferred to label if
                               error when opening the file
        ACCESS='access'        'SEQUENTIAL', (line by line), or
                               'DIRECT' (access individual tagged records)
        RECL=integer           record length for direct access.
)
```

Note that not all the above arguments need to be explicitly given.

Some units are *preconnected*, i.e. they are available without explicitly opening them. For example, the default input is usually unit 5 (in some cases unit 1), and the default output is usually unit number 6 (sometimes 2).

To unattach an opened file from the corresponding unit number when we are done we use

```
CLOSE(UNIT=10,IOSTAT=ierr)
```

(this also inserts an end-of-file character when writting)

7.3.3 READ

READ(	UNIT=*int-expression*	UNIT=5 or UNIT=* for the default input unit
	FMT=*format*	FMT=*''(edit_descriptor_list)''*
		FMT=* is the implicit format
		FMT=*label* points to a labeled FORMAT statement (see below)
	IOSTAT=*int-variable*	*int-variable* holds a return code,
		set when READ completed; 0 if OK, negative if end-of-file
	ERR=*label*,	control is transferred to *label* if
		error when reading the file
	END=*label*,	control is transferred to *label* if
		end-of-file encountered
	EOR=*label*,	control is transferred to *label* if
		end-of-record encountered (non-advancing only)
	ADVANCE=*'mode'*,	'YES', the default, means that each
		READ starts a new record; 'NO' for non-advancing I/O.
	REC=*int-expression*,	record no for direct access
	SIZE=*int-var*	*int-var* holds the number of characters read
		for non-advancing READ
) input_list		

7.3.4 READ example

The statements

```
   READ(UNIT=10,FMT=20) a, b, c
20 FORMAT(3(f7.2))
```

are equivalent to

```
READ(UNIT=10,FMT=''(3(f7.2))'') a, b, c
```

Also, the following are equivalent statements

```
READ(UNIT=5,FMT=*) a, b, c
READ(UNIT=*,FMT=*) a, b, c
READ *, a, b, c
```

7.3.5 WRITE

```
WRITE (            UNIT=int-expression  same as for READ
                   FMT=format
same as for READ   IOSTAT=int-variable  int-variable holds a return code, 0 if writing OK
                   ERR=label,           control is transferred to label if
                                        error when reading the file
                   ADVANCE='mode',      'YES', the default, means that each WRITE
                                        starts a new line/record; 'NO' for non-advancing I/O.
) output_list
```

7.3.6 FORMAT

In a READ or WRITE statement we can give the *format* explicitly as an argument, or we can indicate a label where a FORMAT statement resides.

For example, the statement

```
WRITE(UNIT=6,FMT=''('Areas = ',3(F7.3,2X))'') a1, a2, a3
```

can be written, equivalently, as

```
   WRITE(UNIT=6,FMT=''(a,3(F7.3,2x))'') a1, a2, a3
10 FORMAT( 'areas = ',3(F7.3,2x) )
```

7.4 File Positioning

We can set the position of the file pointer using special instructions.

```
REWIND(UNIT=10)
```

positions the file pointer to the beginning of the specified unit. Note that, writing a record in a sequential file destroys all subsequent information, so by rewinding and writing a file we lose all the data.

```
BACKSPACE(UNIT=10, IOSTAT=ierr)
```

positions the file pointer back one record.

```
ENDFILE(UNIT=10)
```

inserts the endfile record in the file, and keeps the file open, with the file pointer positioned on the endfile record.

7.5 Writing to a string

character(len=5) :: c write(c,10) [list]

Example: opening files with names file01, file02.m, .., file22.m

Chapter 8

Arrays

8.1 Declaring Arrays

Arrays are collections of same-type elements; individual elements are accessed by subscripting the array.

Consider A $\in \Re^6$ be a vector of 6 elements,

$$\mathtt{A} = [a_1 \;\; a_2 \;\; a_3 \;\; a_4 \;\; a_5 \;\; a_6]$$

and B $\in \Re^{2\times 3}$ be a 2×3 matrix,

$$\mathtt{B} = \left[\begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{array} \right]$$

Suppose each element a_i, b_{ij} is exactly represantable in floating point (i.e. a_i, b_{ij} are REAL numbers). Instead of working with 6+6=12 different REAL variables, we can treat A and B in their entirety as F90 variables; the following are valid F90 declarations:

```
REAL, DIMENSION(6) :: a
REAL, DIMENSION(2,3) :: b
REAL :: a(6), b(2,3) ! shorthand form
```

8.2 Memory Storage

Each element of A and B is a REAL number, and requires 4 bytes in the memory. The full A and B have 6 components, hence they require 24 bytes storage.

In F77 A it is stored in a *contiguous* 24-byte region in the memory as follows: A(1) in bytes 1 through 4, A(2) in bytes 5 through 8, ..., A(6) in bytes 20 through 24.

B is also stored in a *contiguous* 24-byte memory region, but now we have to be careful: the memory model is one-dimensional, while B is a two-dimensional array. F90 stores the first column of B first, the second column next and the third column last. In more detail, B(1,1) is stored in bytes 1 through 4, B(2,1) in bytes 5 through 8, B(1,2) in bytes 9 through 12, B(2,2) in bytes 13 through 16, B(1,3) in bytes 17 through 20 and B(2,3) in bytes 20 through 24.

We say in short that F77 uses *storage association*.

Unlike F77, F90 does not specify how arrays are to be organised in memory.

- Advantage. This gives flexibility and allows portability over different architectures (e.g. on a parallel computer the compiler may decide to spread a large array over 16 different memories - this does not violate the F90 standard; the same program may run efficiently on a single processor machine also). Consider for example a 4-processor machine. We need to execute a *saxpy* operation with *a* scalar, *x(1:1024),y(1:1024),z(1:1024)* vectors:

$$z = a * x + y$$

  For efficiency the vectors are stored as follows: $x(1:256)$ on processor 1, $x(257:512)$ on processor 2, $x(513:768)$ on processor 3, $x(769:1024)$ on processor 4. This is an example of *data-parallel* programming, where the same operations are executed by different processors on different data sets.

- Disadvantage. It is common practice to write C++ programs which use Fortran numerical libraries, and viceversa, to call a-different-language library functions in a Fortran program. Consider a F90 routine with an array argument; it is very difficult to call it from C++ (or from another language), since the storage scheme needs to be known in order to properly pass the array argument.

For input-output purposes an ordering of the elements is needed, and this is the colum-wise ordering (F77-similar). This ordering does not imply anything about the storage scheme. For example, if

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

the statement

```
PRINT*, b
```

will produce the output

```
1., 4., 2., 5., 3., 6.
```

Similarly,

```
READ*, b
```

will read in 6 numbers, the elements of B in column-wise order.

8.3 Array Attributes

In F90 the following terminology is used:

- *Size* = total number of elements. Both A and B have size = 6.
- *Rank* = number of dimensions. In F90 the rank should be less than or equal to 7. A has rank 1, B has rank 2.
- *Extent* = number of elements in each dimension. A has extent 6, B has extents 2 and 3. Note that the extent can be ZERO (in this case the expressions involving the array are simply ignored).

- *Bounds* = upper and lower limits of indices; by default, they are taken *1:Extent*. E.g: A index varies between 1 and 6, while B indices vary between 1 and 2, and 1 and 3 respectively. These bounds can be changed using declarations of the form

```
REAL, DIMENSION(-2:3)    :: a
REAL, DIMENSION(-10:-9,5:7) :: b
REAL :: a(-2:3), b(-10:-9,5:7) ! shorthand form
```

- *Shape* = rank and extents. E.g: A has shape (/6/) and B has shape (/2,3/).

F90 provides a number of intrinsic functions that ''inquire'' the arrays about their attributes and status. These intrinsics are useful inside procedure bodies; for example we can obtain the dimensions (extents) of an input array argument without explicitly requiring them in the argument list. The inquiry intrinsic functions may return the lower and the upper subscript bounds (LBOUND and UBOUND), the shape (SHAPE) and the size (SIZE) of an array.

For example

Code	Means	Value
REAL, DIMENSION(−10 : −9, 5 : 7) :: B		
LBOUND(B, 1)	Lower Bound, dim 1	−10
LBOUND(B, 2)	Lower Bound, dim 2	5
LBOUND(B)	All Lower Bounds (array)	(/ − 10, 5/)
UBOUND(B, 1)	Upper Bound, dim 1	−9
UBOUND(B, 2)	Upper Bound, dim 2	7
UBOUND(B)	All Upper Bounds (array)	(/ − 9, 7/)
SHAPE(B)	Rank and extents	(/2, 3/)
SIZE(B, 1)	Extent in dim 1	2
SIZE(B, 2)	Extent in dim 2	3
SIZE(B)	No of elements	6

l

8.4 Subscripting

```
REAL, DIMENSION(6) :: A
REAL, DIMENSION(2,3) :: B, C
B = C
```

One can select elements of an array using subscripts. For example, A(2) and B(2,2) are REAL variables, denoting a_2 and b_{22} respectively. The array subscripts must be integer numbers, integer variables, or expressions returning an integer result.

The statement

```
A(2) = 0.0
B(2,2) = 1.0
```

sets sets a_2 to 0 and b_{22} to the value 1.

We can reference the whole array by its name. For example, the statement

```
B = 1.0
```

will set all the elements in B to 1.

We can select *array sections* using the "**:**" selector. For example, we can set a_1, a_2, a_3 to 0 and $b_{11}, b_{12}, b_{21}, b_{22}$ to 1 using

```
A(1:3) = 0.0
B(1:2,1:2) = 1.0
```

8.5 Conformance

Two arrays are *conformable* if they have the same shape.

All arrays used directly in an expression must be conformable. For example,

```
REAL, DIMENSION(6) :: A
REAL, DIMENSION(2,3) :: B, C
B = C
```

is valid since B, C are conformable (the statement will just copy C into B element-by-element multiplication). However, the statement

```
A = C
```

is illegal, since A and C have different shapes.

Note that scalars are conformal to any array (they are considered to be an array with all elements equal to the scalar). For example,

```
A=2.0 ; C=3.0
```

will fill A with 2's, and C with 3's.

8.6 Array Expressions

In F90 it is possible to build expressions using intrinsic operators and elemental intrinsic functions acting on arrays as variables. All arrays in an array expression must conform. The convention is that intrinsic operations are performed element-by-element (similarly, intrinsic functions applied to arrays act elementwise).

For example,

```
REAL, DIMENSION(2,3) :: B, C, D
B = 3.0; C = 4.0
D = sqrt(B**2 + C**2)
```

will produce a matrix D full of 5's. The meaning of the last (array) statement is that of 6 elemental statements

$$\left\{\begin{array}{l} \mathtt{D}(1,1) = \mathtt{SQRT}(\mathtt{B}(1,1) * *2 + \mathtt{C}(1,1) * *2) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \mathtt{D}(2,3) = \mathtt{SQRT}(\mathtt{B}(2,3) * *2 + \mathtt{C}(2,3) * *2) \end{array}\right.$$

Note that all six elemental statements are, conceptually, performed in parallel (this may or may not be true, depending on the architecture). In contradistinction, a nested pair of DO loops would produce the same result evaluating the elements of D in a predefined order.

In conclusion, array expressions are evaluated element by element, with no particular ordering implied; conceptually, all elemental operations are performed simultaneously.

8.7 Array Sections

We can select sections of an array using the following subscript-triplet

$$[\langle\texttt{bound}_1\rangle] : [\langle\texttt{bound}_2\rangle][: \langle\texttt{stride}\rangle]$$

The subscripts of the array section start at $\langle bound_1\rangle$, are incremented by $\langle stride\rangle$ and stop at or before $\langle bound_2\rangle$. The selected subscript values are similar to the values of the DO-loop variable i generated by the statement

$$\texttt{DO i} = \langle\texttt{bound}_1\rangle, \langle\texttt{bound}_2\rangle, \langle\texttt{stride}\rangle$$

The bounds and the stride must be integer-valued expessions; if missing, they are assumed to take implicit values as follows:

- missing $\langle\texttt{bound}_1\rangle$: $bound_1$ = subscript's lower bound, as declared.
- missing $\langle\texttt{bound}_2\rangle$: $bound_2$ = subscript's upper bound, as declared.
- missing $\langle\texttt{stride}\rangle$: $stride = 1$.

For example, let

$$\texttt{REAL,DIMENSION}(-2:6) :: \texttt{A}$$

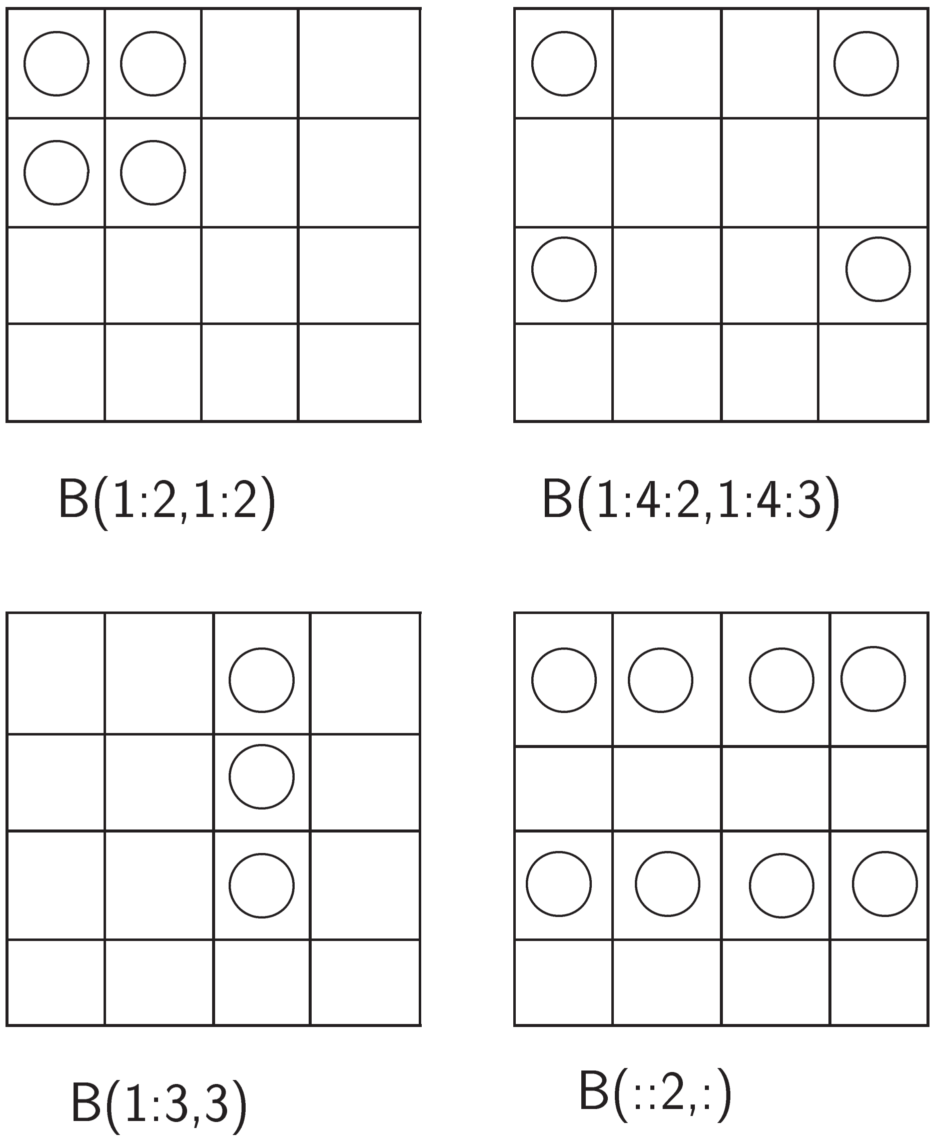
be the array $[a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, a_4, a_5, a_6]$. Note that the subscript's lower bound is -1, and upper bound is 6. Then the following sections can be selected

A(1:3)	$[a_1, a_2, a_3]$
A(1:3:2)	$[a_1, a_3]$
A(1:4:2)	$[a_1, a_3]$
A(3:1)	zero sized section
A(3:1:-1)	$[a_3, a_2, a_1]$
A(:)	whole array
A(4:)	$[a_4, a_5, a_6]$
A(:2)	$[a_{-2}, a_{-1}, a_0, a_1, a_2]$
A(::4)	$[a_{-2}, a_2, a_6]$
A(2:2)	$[a_2]$
A(2)	scalar element a_2

Note that a section of the array is an array itself.

As a second example, consider

```
REAL, DIMENSION(4,4) :: B
```



8.8 RESHAPE intrinsic function

This intrinsic function changes the shape of an array to the desired form.

The syntax is

$$\texttt{RESHAPE}(\langle\texttt{array}\rangle, \langle\texttt{shape}\rangle[, \langle\texttt{pad}\rangle][, \langle\texttt{order}\rangle])$$

For example,

$$\texttt{RESHAPE}((/1,2,3,4/),(/2,2/)) \Longrightarrow \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

If the constructor list is too short, elements from the ⟨pad⟩ are used to pad the result:

$$\texttt{RESHAPE}((/1,2,3/),(/2,2/),(/0/)) \Longrightarrow \begin{bmatrix} 1 & 3 \\ 2 & 0 \end{bmatrix}$$

(if the constructor list is longer than the size specified by ⟨*shape*⟩, the first *size* elements will be used for the array, and the rest will be ignored).

The column-wise order can be changed (the subscripts vary in the order specified by ⟨order⟩):

$$\texttt{RESHAPE}((/1,2,3/),(/2,2/),(/0/),(/2,1/)) \Longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$

8.9 Array Constructors

Have the form

$$(/ \ \texttt{scalar}_1, \ \texttt{scalar}_2, \ \ldots \ , \ \texttt{scalar}_n \ /)$$

The list of scalars are placed into the array in order. The size of the constructor must equal the size of the array.

```
REAL, DIMENSION(6) :: A, B
REAL, DIMENSION(2,3) :: C
A = (/1.0,2.0,3.0,4.0,5.0,6.0/)
B = (/ (i, i=1,6) /)
C = RESHAPE( (/1.0,2.0,3.0,4.0,5.0,6.0/), (/2,3/) )
```

The B definition contains an implied DO loop (the values of i generated by the construct ‘‘DO i=1,6’’).

Only one dimensional constructors are allowed; to initialize higher rank matrices, we need to explicitly RESHAPE the constructor to the desired shape.

We can use constructors to initialize arrays, for example

```
INTEGER, DIMENSION(6) :: A = (/ (i, i=1,6) /)
REAL, DIMENSION(2,3), PARAMETER :: &
          C = reshape( (/1.,2.,3.,4.,5.,6./),(/2,3/))
```

8.10 Allocatable Arrays

In F90 temporary arrays can be dynamically created, used, and then discarded at will; the necessary memory is grabbed, used, and then put back (dynamic heap storage).

The declaration

```
REAL, DIMENSION(:), ALLOCATABLE :: A
```

states that A is a temporary matrix; no memory is allocated at the time of declaration; the size of A is irrelevant at this point, and the precise specification of the shape is deferred until the allocation point. In F90 jargon, A is said to be a *deferred-shape array*; in contrast, the ‘‘standard’’ declaration

```
REAL, DIMENSION(5,5) :: B
```

makes B an *explicit-shape* array.

To reserve memory for A, we use

```
ALLOCATE( A(-1:3,2:5), STAT=ierr )
```

Upon completion of the function, the integer variable ierr reports the success of the allocation process: ier $= 0$ if everything is ok, and ierr $\neq 0$ if the allocation request was denied.

The intrinsic inquiry function

```
allocated( A )
```

returns a logical result: .TRUE. if A is allocated, and .FALSE. otherwise.

Heap storage is reclaimed using

```
IF( allocated(A) ) DEALLOCATE( A, STAT=ierr )
```

(this deletes the array from memory and returns the space as free space in the heap). ierr reports the succes of the dealloction function. Note that we check first if A was allocated.

There is always an overhead in managing dynamic arrays; therefore, if the size of the array is known, the memory restrictions are not tight, or the arrays are needed during most of the program's lifetime, one should preffer explicit-shape arrays.

Attention: a deffered-shape array allocated in a procedure should be deallocated upon return - otherwise the memory space becomes inaccesible for the rest of the program's life (an inaccessible memory chunk is called ‘‘garbage’’).

An allocatable array in a procedure can be SAVEd. For example,

```
SUBROUTINE ...
REAL, ALLOCATABLE, DIMENSION(:,:), save :: A
...
IF (.not.allocated(a)) ALLOCATE(A(10,10)
```

A is allocated during the first call to the subroutine; after the procedure termination, A will remain allocated and its value preserved becaude of the SAVE attribute; following calls to the subroutine will ‘‘see’’ that A was already allocated and will work with the old values.

Homework 0.17 *In a loop, allocate arrays of increasing size, and check the allocation status; exit the loop when the allocation request is refused. What is the largest size of a succesfully-allocated array? Do not forget to deallocate the current array before the new allocation at the next iteration.*

8.11 Masked Assignment

Assignments can be executed for noregular sections of the arrays only. For example the assignments can be selectively executed only for those elements which satisfy a certain condition. This is accomplished using a WHERE construct.

Suppose for example we have three real matrices:

$$A = \begin{bmatrix} 10. & 10. \\ 10. & 10. \end{bmatrix} ,B = \begin{bmatrix} 1. & 3. \\ 5. & 9. \end{bmatrix} ,C = \begin{bmatrix} 0. & 2. \\ 4. & 6. \end{bmatrix} .$$

We want to change the matrix A such that $a_{ij} = b_{ij}/c_{ij}$; if $c_{ij} = 0$ then we skip the division and let the old value of a_{ij} unchanged.

The form of the construct is

```
WHERE (C /= 0)
  A = B/C
END WHERE
```

The argument of WHERE is evaluated first. This argument is a matrix of logical elements, and is called the *mask*. The mask must conform to the implied shape of each assignment in the body. In our example

$$\text{C.NE.0} = \left[\begin{array}{ll} \text{.FALSE.} & \text{.TRUE.} \\ \text{.TRUE.} & \text{.TRUE.} \end{array}\right]$$

(all the elements of the mask are, conceptually, evaluated in parallel).

Next, the statements in the body of the WHERE construct are evaluated succesively. The matrix assignment C = A/B is performed element by element, but only for those elements (i, j) for which $mask_{ij} =$.TRUE.; in our example, the result of the masked assignment is

$$A = \left[\begin{array}{ll} 10. & 1.5 \\ 1.25 & 1.5 \end{array}\right] .$$

The complete form of the WHERE construct includes an ELSEWHERE branch as well:

```
WHERE (C /= 0)
  A = B/C
elsewhere
  A = -1.0
END WHERE
```

The ELSEWHERE assignments are performed for those elements (i, j) for which $mask_{ij} =$.FALSE.; the result of our example is

$$A = \left[\begin{array}{ll} -1.. & 1.5 \\ 1.25 & 1.5 \end{array}\right] .$$

Note: a short form is possible; similar to the IF statement, there is a WHERE statement which reads

```
WHERE (C /= 0) A = B/C
```

WHERE statements cannot be nested. The execution first evaluates the *mask* (a matrix of .TRUE./.FALSE. depending on the elemental value of the condition) then executes the assignments for the .TRUE. positions (the WHERE block), then execute the assignments for the .FALSE. positions (the ELSEWHERE block).

8.12 Vector-valued Subscripts

Are useful for building irregular sections of arrays (when subscripts do not follow a linear pattern).

For example,

```
INTEGER, DIMENSION(2) :: V=(/4,2/)
INTEGER, DIMENSION(4) :: W=(/3,2,4,1/)
REAL, DIMENSION(10,10) :: A
```

A(V,W) is the irregular 2×4 section of the matrix resulting from intersecting rows $V(i)$ with columns $W(j)$.

If $A = [1, 2, 3, 4, 5, 6]$, $B = [10, 11, 12]$, $V = [3, 5, 3]$, then $A(V) = [3, 5, 3]$; $B = A(V)$ is OK and will produce $B = [3, 5, 3]$, but $A(V) = B$ should be avoided since the subscript 3 is repeated ($A(3)$ is assigned twice, and since no order can be guaranteed in the parallel assignment, the result is not well defined).

Vector subscripting basically performs index indirection (of one, two or more nested levels) and is therefore *very inefficient*; it should not be used unless really necessary.

8.13 Homework

Homework 0.18 *Consider the matrices A = [.1 .2;.4 .6], B=[.5 .9; .7 .8]. Initialize A with an array constructor, then Read in 4 numbers for B, then compute the expression C=sin(A)+cos(B). Using inquiry intrinsic functions print the total no of elements in C, the shape of C, the upper and lower bounds for A, B, C.*

.

Homework 0.19 *Write a subroutine that takes an integer matrix and negates all the odd elements (Hint: use* WHERE*).*

8.14 Vector and Matrix Multiplication Intrinsics

If A, B are conform, dimension one arrays, their dot product is computed by calling the function DOT_PRODUCT; the result is, of course, a scalar DP

```
DP = dot_product(A,B)
```

The meaning is

Type	DP
REAL	$A(1)*B(1) + A(2)*B(2) + \ldots$
DOUBLE	$A(1)*B(1) + A(2)*B(2) + \ldots$
COMPLEX	$CONJG(A(1))*B(1) + CONJG(A(2)*B(2)) + \ldots$
LOGICAL	$A(1).AND.B(1).OR.A(2).AND.B(2) + \ldots$

For two matrices C and D (at least one has dimension 2) their matrix-matrix product is given by

```
E = matmul(C,D)
```

The result is a matrix. The shapes of C, D, E have to obey the usual mathematical relation for matrix multiplication. Note that the matrix product MATMUL(C,D) is not equal to the element-by-elemen product C*D.

Homework 0.20 *Consider the matrix*

$$A = \begin{bmatrix} 1 & 8 & 3 & 9 \\ 8 & 3 & 0 & -3 \\ -2 & -2 & 1 & 9 \\ 5 & -3 & 2 & -1 \end{bmatrix}$$

Write a small program to compute and print

1. *A* 4×4 REAL *matrix* B *with random elements. Such a matrix can be created using*

   CALL RANDOM_NUMBER(B) .

2. MATPROD(A, B)
3. MATPROD(A, B) - A*B
4. DOT_PROD(first_row_A, third_column_B)

8.15 Maximum and Minimum Intrinsics

```
max( a1, a2, a3, ...)      min( a1, a2, a3, ...)
```

return the maximum (minimum) value over a list of arguments.

The arguments $a_1, a_2, \ldots$ can be (conformal) arrays; the result of MAX (MIN) is then an array of the same shape and size as the arguments, containing element by element maxima (minima).

Homework 0.21 *Write a small program to calculate* MAX *and* MIN *of*

- MAX *and* MIN(1,3 2,5,3).
- MAX *and* MIN(5, (/2,6,1,6,3,2/),(/7,3,9,1,6,3/))

The maximum value among the elements of an array A can be obtained with

```
maxval(A)
```

The location of the first element which has this maximum value is given by

```
maxloc(A)
```

If the rank of the matrix A is r, the returned result is an r-dimensional array, containing the values of the subscripts (i.e. the location) of the maximum value element.

We can take the maximum values along a specified dimension, using

```
maxval(A, DIM=d)
```

When we use the DIM argument, the result of MAXVAL is a rank r-1 array, with the maximum values in other dimensions.

For example, if A is a 2-dimensional matrix, using d=1 computes the maxima in each column, and d=2 computes the maxima in each row. The results are rank 1 arrays (vectors).

Finally, we can use the masked form of MAXVAL, for example

```
maxval(A, MASK=A<4)
```

The survey is only performed on elements of A which correspond to .TRUE. elements of the mask; here the maximum is taken over the elements of A which are less than 4.

Note that DIM and MASK arguments can be used simultaneously.

The functions

$$\mathtt{MINLOC(A)}\ ,\quad \mathtt{MINVAL(A[,MASK=m][,DIM=d])}\ ,$$

return the position of the minimal element in an array, and the minimal value respectively; their use is similar to the use of MAXLOC, MAXVAL.

Homework 0.22 *Consider the matrices*

$$A = \begin{bmatrix} 1 & 8 & 3 & 9 \\ 8 & 3 & 0 & -3 \\ -2 & -2 & 1 & 9 \end{bmatrix} B = \begin{bmatrix} 4 & 5 & 2 & 9 \\ 5 & -3 & 2 & -1 \\ 9 & -2 & 7 & -5 \end{bmatrix}$$

Write a small program that allocates A, B, computes and prints the following, and deallocates A, B before exit.

```
maxloc(A, mask=B.LT.0)
maxloc(A, dim=1)
maxloc(A, dim=2)
maxloc(A, mask=B.LT.0, dim=1)
maxloc(A, mask=B.LT.0, dim=2)
minloc(B, dim=1)
minloc(B, dim=2)
```

8.16 Array Reduction Intrinsics

Reduction functions act on a rank r array A, and return a result of a smaller rank. If the argument DIM is absent the result is a scalar, if DIM is present the result is an array of rank r-1.

• SUM(A [,DIM=d] [,MASK=m]) returns the sum of array elements, along an optionally specified dimension under an optionally specified mask.

• SUM(A [,DIM=d] [,MASK=m]) returns the sum of array elements, along an optionally specified dimension under an optionally specified mask.

• PRODUCT(A [,DIM=d] [,MASK=m]) returns the product of array elements, along an optionally specified dimension under an optionally specified mask.

• ALL(MASK [,DIM=d]) returns .TRUE. if all entries in the logical array MASK are .TRUE.; the operation can be performed along a certain dimension. (Note: returns .TRUE. for zero-sized arrays)

• ANY(MASK [,DIM=d]) returns .TRUE. if at least one entry in the logical array MASK is .TRUE.; the operation can be performed along a certain dimension. (Note: returns .FALSE. for zero-sized arrays)

• COUNT(MASK [,DIM=d]) returns the number of .TRUE. elements in the logical array MASK; the operation can be performed along a certain dimension. (Note: returns 0 for zero-sized arrays)

Homework 0.23 *With*

$$A = \begin{bmatrix} 1 & 8 & 3 & 9 \\ 8 & 3 & 0 & -3 \\ -2 & -2 & 1 & 9 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 5 & 2 & 9 \\ 5 & -3 & 2 & -1 \\ 9 & -2 & 7 & -5 \end{bmatrix}$$

write a small program to compute and print

1. COUNT(B.LT.0)
2. COUNT(B.LT.0, DIM=1)
3. COUNT(B.LT.0, DIM=2)
4. SUM(A)
5. SUM(A, DIM=1), SUM(A, DIM=2)
6. PROD(A)
7. PROD(A, DIM=1), PROD(A, DIM=2)
8. ALL(A.NE.B), ANY(A.NE.B)

Homework 0.24 *Write three functions that have one input argument, a rank 2 array (a matrix)* A, *and return*

1. *the Frobenius norm of the matrix* $\sqrt{\sum_{i=1}^n \sum_{j=1}^n a_{ij}^2}$;
2. *the 1-norm of the matrix* $\max_i \sum_{j=1}^n |a_{ij}|$;
3. *the* ∞*-norm of the matrix* $\max_j \sum_{i=1}^n |a_{ij}|$.

8.17 Array Arguments

In F90 arrays can be passed as procedure arguments. There are several points to be made regarding how the shape and the size of an array argument is consistently passed to the procedure.

Consider for example the subroutine COMMUT that computes the ''commutator'' of two matrices A and B,

$$C = AB - BA$$

8.17.1 Explicit Shape Arrays

One possible definition of the subroutine is

```
SUBROUTINE commut(A,B,C)
  IMPLICIT NONE
  REAL, DIMENSION(10,10), INTENT(IN) :: A, B
  REAL, DIMENSION(10,10), INTENT(OUT) :: C
  C = matmul(A,B)-matmul(B,A)
END SUBROUTINE commut
```

The shape of the dummy array arguments A,B,C is explicitly declared: they are 10×10 matrices. When calling the subroutine, the sizes and shapes of the actual arguments must of course conform to the sizes and shapes of dummy arguments; therefore, the routine can only compute the commutator of 10×10 matrices. The explicit shape approach is very inflexible, and usually avoided.

8.17.2 Assumed Shape Arrays

We want the routine to work for square arrays of any dimension. To allow this, the recommended method in F90 is to declare dummy array arguments as *assumed shape arrays*:

```
SUBROUTINE commut(A,B,C)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) :: A, B
  REAL, DIMENSION(:,:), INTENT(OUT) :: C
  C = matmul(A,B)-matmul(B,A)
END SUBROUTINE commut
```

An assumed-shape declaration must have the same type and rank as the actual argument declaration.

When the subroutine is called, the dummy arrays A,B,C *assume* the shape of the associated actual arguments. The actual argument can be an array, or an array section, but needs to have an explicit shape. An assumed-shape array as actual argument lacks the bound/extent information and cannot be passed on to a further procedure. Note that the program units which call procedures with assumed shape arguments need to have explicit interfaces available at the point of call (we will iscuss later about this).

Note that, whenever an external subroutine uses assumed-shape array arguments, the calling program must contain an *interface* of the respective procedure. This allows the compiler to prepare the necessary information when passing array arguments. An interface is placed before local declarations; it contains the subroutine headers and argument declarations. The syntax can be viewed in the following example:

```
INTERFACE
 SUBROUTINE commut(A,B,C)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) :: A, B
  REAL, DIMENSION(:,:), INTENT(OUT) :: C
 END SUBROUTINE commut
END INTERFACE
```

8.17.3 Automatic Arrays

Some temporary arrays can depend on dummy arguments; these arrays are truly automatic, they are created and destroyed with each invocation of the procedure. Automatic arrays are not dummy arguments; they are declared as explicit-shape arrays are, except that the size is depends oninput arguments - it is therefore known at run time only. Traditionally, such arrays are used for workspace. For example, two automatic arrays P1, P2 can be declared to conform to A:

```
SUBROUTINE commut(A,B,C)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) :: A, B
  REAL, DIMENSION(:,:), INTENT(OUT) :: C
  REAL, DIMENSION(SIZE(A,1),SIZE(B,2)) :: P1
  REAL, DIMENSION(SIZE(B,1),SIZE(A,2)) :: P2
  P1 = matmul(A,B)
  P2 = matmul(B,A)
  C = P1 - P2
END SUBROUTINE commut
```

When using automatic arrays we have to include an interface in the calling program.

In F77 we had to explicitly pass the size of A in the argument list, and use this dummy variable to declare P1,P2:

```
SUBROUTINE commut(A,B,C,N)
REAL, DIMENSION(N,N) :: P1,P2}
```

Note that automatic arrays cannot have the SAVE attribute - the reason being that they can have different shapes and sizes during different procedure calls.

8.17.4 F77's Assumed-Size Arrays

An assumed-size array may only appear as a dummy argument of a procedure; in its declaration, all the extents except the last one must be given explicitly, and the last one is specified by $*$.

For example, the calling program declares

```
REAL, DIMENSION(50) :: Z
CALL test(Z)
```

and the subroutine header is

```
SUBROUTINE test(Z)
REAL, DIMENSION(*) :: Z
```

The idea in F77 was that Z was allocated in the calling program; the only thing the subroutine needs to know is where the array Z starts; all other elements are found in subsequent memory locations.

8.18 Array-Valued Functions

In F90 we can have functions that return array results. If the input arguments are assumed-shape, the result must be automatic. For example, we can reformulate the commutator as an array-valued function:

```
FUNCTION commut(A,B)
IMPLICIT NONE
REAL, DIMENSION(:,:), INTENT(IN) :: A,B
REAL, DIMENSION(SIZE(a,1),SIZE(a,2)) :: COMMUT
  IF( (SIZE(A,1)/=SIZE(A,2)) .or. &
      (SIZE(A,2)/=SIZE(B,1)) .or. &
      (SIZE(B,1)/=SIZE(B,2))      ) THEN
   PRINT*, 'Error: input matrices not conformal'
   RETURN
END IF
commut =  matmul(A,B)-matmul(B,A)
END FUNCTION commut
```

Here we check the compatibility of the input arguments.

Note that declaring the result (COMMUT) as an assumed-shape array does not work, since it is not associated with any actual argument; we have to declare COMMUT explicitly with proper dimensions that depend on the size of dummy arguments - in other words, COMMUT is declared as an automatic array.

Chapter 9

More on Procedures

9.1 Recall F90 Program Units

- main PROGRAM = the unit where the execution begins, and where it eventually returns before termination.
- MODULE = ‘‘packs’’ procedures and declarations
- Procedures
  - (External) SUBROUTINE = parametrized sequence of code;
  - (External) FUNCTION = parametrized sequence of code that returns a result in the function name.

9.2 Side Effects

A notorious characteristic of bad software design are the side effects. Side effects may happen when a function or subroutine modifies the values of their input arguments, the result being a hard-to-control program logic.

For example, suppose we want a function norm2w (w stands for wrong!) which computes the 2-norm of the vector, then projects the vector on the (y,z) plane by setting its x-coordinate to zero.

At some point in the program we have 2 vectors which have the same (x,y) coordinates, but a different z coordinate, $[a,b,c]$ and $[a,b,d]$. To fix the ideas, suppose $a=3$, $b=1$, $c=7$, and $d=2$. Our intent is to compute the norm of the first vector ($\sqrt{a^2+b^2+c^2} \approx 7.6811$) plus the norm of the (y,z) projection of the second vector ($\sqrt{0^2+b^2+d^2} \approx 2.4495$). The correct result is ≈ 10.1306.

We write the following line of code:

```
y = norm2w(a,b,c) + norm2w(a,b,d)
```

When the first call is executed, $\sqrt{a^2+b^2+c^2}$ is computed, then a is set to 0; at the second call, $\sqrt{0^2+b^2+c^2}$ is computed, and a is set again to zero; overall, we get the correct result.

Now, the trouble is that we do not know that norm2w(a,b,c) is evaluated first, and norm2w(a,b,d) second. Except for operator precedence, F95 standard says nothing about the order of argument evaluation. The compiler may choose to evaluate the right operand first (i.e. norm2w(a,b,d)), put the result in temporary storage, evaluate the left operand next (i.e., norm2w(a,b,c)) and then perform the addition. Now the result is ≈ 10.8127, clearly different than what we had in mind.

9.3 Argument Intent

In order to increase robustness to side-effects, and to facilitate efficient compilation, dummy arguments in the procedure declaration can be tagged with an intent attribute:

- INTENT(IN) Input only arguments; they cannot be written inside the procedure (or the compiler will complain);
- INTENT(OUT) Output only arguments; they cannot be read until they are written inside the procedure (assume they come in with garbage); also, if they are not assigned at all in the procedure body, a compilation error will result;
- INTENT(INOUT) Both input and output; assumed to hold valid data at entry, but this value can be modified by the procedure and passed back to the calling program.

For example, in our norm2s subroutine we may declare

```
REAL, INTENT(IN) :: x,y,z
REAL, INTENT(OUT) :: r
```

Also, the norm2 function declarations may be

```
REAL, INTENT(IN) :: x,y,z
```

9.4 Pure Functions

A pure function is free of side effects. A function is pure if all of the following three requirements are fullfilled:

- all arguments have the explicit attribute INTENT(IN);
- the function does not modify any global variables, and
- it does not perform I/O.

We can declare a function to be pure, and the compiler will check the above conditions. Example:

```
pure REAL FUNCTION norm_2(x, y, z)
IMPLICIT NONE
  REAL, INTENT(IN) :: x,y,z
  norm2 = sqrt(x*x+y*y+z*z)
END FUNCTION norm_2
```

9.5 Internal Procedures

In F90 program units contain declarations, executable statements, but may also contain *internal procedures*. A program unit that contains internal procedures is called *the host*. Internal procedures

- may be called only from within the host;
- have access to all the host declarations;
- can override host declarations;
- may not contain further internal procedures;
- are separated from the host by a CONTAINS statement, for example

```
PROGRAM ⟨name⟩
  CALL SUB1(···)
CONTAINS
    SUBROUTINE sub1(···)
    .....................
    END SUBROUTINE sub1
END PROGRAM ⟨name⟩
```

9.6 Modules Hosting Procedures

```
MODULE example
! Type Definitions
! Global Data
...
CONTAINS
    SUBROUTINE level_1
    ...
    CONTAINS
      SUBROUTINE level_2
      ...
      END SUBROUTINE level_2
  END SUBROUTINE level_1
END MODULE example
```

Procedures, together with global data, are visible by use-association (i.e. are visible in the program units that USE the module). In modules, one level of nesting is allowed for contained procedures (in contradistinction to program or procedure hosts, where nesting is not allowed). The second-level routines are visible from the first-level routines, and only the first-level routines are visible from the unit that USEs the module. In F95 an indefinite number of nesting levels is allowed.

9.7 Scope

The scope of an entity is the range of a program within which the entity is visible and accesible.

- Local scope: the entity is visible within the program unit in which it is defined.
- When the declaring program CONTAINS functions (and/or subroutines), the declared entity can be made available to the hosted procedures via
  - Host association: An entity defined in a program unit is visible within all procedures contained in that program unit.
  - Argument association. An entity defined in a program unit is made available to a contained function or subroutine by being passed as argument.
- Global scope: the entity has to be declared in a module; then it is visible in each program unit which USEs that module.

9.8 Scope example

```
PROGRAM test_scope
IMPLICIT NONE
INTEGER :: i=1, j=2, k=3, glb=4, loc=5
!
PRINT*,"(1) main: i=",i,", j=",j, &
   ", k=",k,", glb=",glb,", loc=",loc
CALL my_sum(i,j,k)
PRINT*,"(2) main: i=",i,", j=",j, &
   ", k=",k,", glb=",glb,", loc=",loc
!
CONTAINS
!
  SUBROUTINE my_sum(a,b,c)
  INTEGER, INTENT(IN) :: a,b
  INTEGER, INTENT(OUT) :: c
  INTEGER :: loc=7
  PRINT*,"(1) my_sum: i=",i,", j=",j, &
    ", k=",k,", glb=",glb,", loc=",loc, &
    ", a=",a,", b=",b,", c=",c
  c = a**2 + b**2
!
! c = i**2 + j**2 is ok, same result,
!     since we read i, j in <my_sum>
! k = a**2 + b**2 is illegal since
!     we cannot write k;
!
  glb = 6  ! seen by main (global)
  loc = 8  ! NOT seen by main (local)
!
  PRINT*,"(2) my_sum: i=",i,", j=",j, &
    ", k=",k,", glb=",glb,", loc=",loc, &
    ", a=",a,", b=",b,", c=",c
  END SUBROUTINE my_sum
!
END PROGRAM test_scope
```

In the subroutine my_sum the following variables are visible

- a,b,c by argument association
- i,j,k,glb by host association (global variables)
- loc locally owned by my_sum (no relation to m in main).

The results produced by the above program are

```
(1) main:   i=1, j=2, k=3, glb=4, loc=5
(1) my_sum: i=1, j=2, k=3, glb=4, loc=7, a=1, b=2, c=3
(2) my_sum: i=1, j=2, k=5, glb=6, loc=8, a=1, b=2, c=5
(2) main:   i=1, j=2, k=5, glb=6, loc=5
```

glb is a global variable; its declaration in the main program is valid within my_sum. In consequence, the modification glb = 6 is visible in the main program also (even after the function ends).

loc is declared in main, but is also re-declared as a local variable in my_sum. This local re-declaration overrides the global declaration within the body of the function. Therefore, loc is a local variable of the function my_sum (and bears no relationship with the loc in main - they're like John from California and John from Wisconsin). The modification loc = 8 is seen locally only; when the function terminates and control is returned to main program, loc is restored to the value it had before the function call (this value, 5, is local to the main program).

Variables i,j,k are in a special situation: they are the actual arguments when calling my_sum (and therefore are visible by argument association under the dummy names a,b,c) and are in the same time visible by host association (under their original names i,j,k). In order to prevent uncontrollable side-effects, F90 syntax allows us to read i,j,k inside my_sum, but forbids us to modify them (they behave like dummy arguments with INTENT(IN)). If we want to modify k, for example, we have to use the equivalent dummy (argument) name c.

The hosted functions and subroutines inherit not only the declared variables but also the IMPLICIT NONE declaration.

Finally, note that internal procedures cannot be used as arguments (only INTRINSIC and EXTERNAL procedures can, and they need to be declared with the proper attribute).

9.9 Procedure Interfaces

A procedure interface consists of all the information needed for the caller-callee communication; such information must discriminate between a function or a subroutine and should provide the types, number and order of arguments (such that the actual argument list must the dummy argument list).

9.9.1 More on Implicit Interfaces

Implicit interfaces are the old, F77 way of communication. Separate procedures are desired to be as independent as possible from one another (so that they can be written, compiled and maintained independently); in particular, a procedure is totally ignorant of how a different procedure looks like.

Special problems arise when procedures call each other; the caller has very limited knowledge on the called procedure; it passes a list of actual argument addresses to the called procedure; the called procedure retrieves this addresses in order, and assumes they are the addresses of the dummy arguments.

The disadvantage of this approach is that the compiler is able to perform only limited consistency checks. In F77, it is easy to make mistakes undetectable by the compiler when working with argument lists.

Consider, for example, the following F77 subroutine

```
      SUBROUTINE BubbleSort(iv,n,how)
      INTEGER n, how, iv(*)
      INTEGER i, j, tmp, iperm
      iperm = 0
      DO i=1,n
        DO j=i,n
          IF ( how.eq.-1 .and. iv(i).lt.iv(j) .or.
     &         how.eq.1 .and. iv(i).gt.iv(j) ) THEN
            tmp   = iv(i); iv(i) = iv(j); iv(j) = tmp
            iperm = iperm + 1
          END IF
        END DO
      END DO
      how = iperm
      END
```

The subroutine receives a vector of integers, iv of dimension n (here the star means assumed size, that is the routine receives the address of the first aelement and is able to handle the whole vector from here). The routine returns the vector iv with elements sorted in increasing order (if how=+1), or in decreasing order (if how=-1); upon return the variable how will contain the number of swaps performed by the bubble algorithm.

The main program can be

```
      PROGRAM impint
      INTEGER n
      PARAMETER (n=4)
      INTEGER iv(n), iw, how
      DATA iv /3,1,4,2/
      how = -1
      CALL BubbleSort(iv, n, how)
      PRINT *, 'sorted decreasingly: ',(iv(i),i=1,n),' how:', how
      END
```

and produces the output

```
Sorted Decreasingly:     4   3   2   1  How:  3
```

Now, if we mistype

```
      hoe = -1
      CALL sort(iv, n, hoe)
```

the variable hoe will be considered a real variable; the bytes starting at that memory address will be considered as representing an integer (-1082130432) by the subroutine; since it is not $+1$ or -1, the output is 3 1 4 2. Note that, since the main program knows nothing about the subroutine sort, the compiler cannot detect the inconsistency of using a real actual argument hoe when the third dummy argument how is of type integer.

Suppose we call

```
    CALL sort(iv, n, -1)
```

This call is fine for input purposes, the third argument being a constant of type integer; the problem is that the third argument (how) is intended to return the number of swaps, and is written inside the procedure; this produces a segmentation fault (it is anyway clear that we cannot write the constant -1). Again, the compiler cannot help us because of its limited knowledge of the function.

The call

```
    CALL sort(iv, n)
```

is also fine with the compiler; obviously, the dummy argument how will not be initialized (in fact it takes a garbage-value). Similarly, we can include extra arguments without bothering the compiler

```
    CALL sort(iv, n, how, extra)
```

(but we can trick ourselves if we want to read the value of extra upon return).

Also, if iw is an integer scalar, the call

```
    CALL sort(iw, n, how)
```

is permitted by the compiler; since F77 assumes that the entries of iv are stored sequentially into memory, the subroutine will "work" with the memory locations following those that contain iw; if we are lucky we get a segmentation error; if not, other data can be messed up - without any warning - and we get meaningless results.

9.9.2 Explicit Interfaces

The above examples show clearly that it is important to provide the caller with essential information about the called procedure's arguments; this information must include the types, number and order of dummy arguments.

If, in the above examples, the main program CONTAINS the function sort, all the faulty examples above will result in compilation errors. We say that internal procedures have an explicit interface with the host. This means that the host (main program) has all the information needed about the procedure's argument list.

Explicit interfaces provide the compiler with all the needed information to

- perform consistency checks (actual vs. dummy argument lists);
- better optimize the code;
- communicate enough info from the caller to procedures ar runtime.

In general, procedures have an explicit interface with the calling program (the calling program has complete knowledge about the procedure's argument list) in the following situations:

1. the procedure is intrinsic to Fortran (e.g. SIN, EXP, MAX, etc.)

2. the procedure is internal, and the calling program is its host;

3. the procedure is internal, and a module is its host; the calling program USEs the host module;

4. the calling program has a block interface construct (to be discussed next);

5. a module has a block interface construct, and the calling program USEs this module.

9.9.3 Explicit Interfaces for Internal Procedures

In F90, the above program may be

```
PROGRAM impint
  IMPLICIT NONE
  INTEGER, PARAMETER :: n=4
  INTEGER, DIMENSION(n) :: iv=(/3,1,4,2/)
  INTEGER :: iw, how
  how = -1
  CALL sort(iv, n, how)
  PRINT *, iv(1:n), how
CONTAINS
  SUBROUTINE sort(iv,n,how)
  INTEGER, DIMENSION(:), INTENT(IN) :: iv(:)
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(INOUT) :: how
  ...................
  END SUBROUTINE sort
END PROGRAM impint
```

In this case the interface of sort with the main program is explicit.

9.9.4 Explicit Interfaces for External Procedures

In F90 ‘‘it is possible, often essential and wholly desirable’’ to provide explicit block interfaces whenever external procedures are used.

An interface declaration of an external procedure is initiated by INTERFACE statement and terminated by END INTERFACE. The declaration specifies the attributes of the dummy arguments and the procedure - it is in fact the whole procedure, without the local declarations and the executable code. An interface that is part of the declarations sequence is called ‘‘explicit’’.

For example, consider the sorting function. If the function is internal to the main program we have an explicit interface, but we lose modularity. With block interface, the routine can be in a different place (different file, or different library) and still have explicit interface.

```
PROGRAM impint
IMPLICIT NONE
INTERFACE
  SUBROUTINE sort(iv,n,how)
  INTEGER, DIMENSION(:), &
  INTENT(IN) ::  iv(:)
  INTEGER, INTENT(IN) ::  n
  INTEGER, INTENT(INOUT) ::  how
  END SUBROUTINE sort
END INTERFACE
INTEGER, PARAMETER ::  n=4
INTEGER, DIMENSION(n) ::  iv
INTEGER ::  iw, how
iv=(/3,1,4,2/)
how = -1
CALL sort(iv, n, how)
PRINT *, iv(1:n), how
END PROGRAM impint
```

```
SUBROUTINE sort(iv,n,how)
INTEGER, DIMENSION(:), &
INTENT(IN) ::  iv(:)
INTEGER, INTENT(IN) ::  n
INTEGER, INTENT(INOUT) ::  how
INTEGER ::  i, j, tmp, iperm
iperm = 0
DO i=1,n
DO j=i,n
  IF (how.EQ.-1.AND.iv(i).LT.iv(j).OR.&
  how.EQ.1.AND.iv(i).GT.iv(j)) THEN
  tmp = iv(i); iv(i) = iv(j)
  iv(j) = tmp; iperm=iperm+1
  END IF
END DO
END DO
how = iperm
END SUBROUTINE sort
```

If the subroutine sort is called from different program units, then a copy of the bloc interface should be included in the declaration sequence of each caller. If at a later time we decide to modify the interface, all these copies need to be updated. A way to circumvent these drawbacks is to write the block interface once, in a module, then to USE the module in all units that call the subroutine. For example, we may have

```
PROGRAM impint
USE sort_interf
IMPLICIT NONE
INTEGER, PARAMETER ::  n=4
INTEGER, DIMENSION(n) ::  iv
INTEGER ::  iw, how
iv=(/3,1,4,2/); how = -1
CALL sort(iv, n, how)
PRINT *, iv(1:n), how
END PROGRAM impint
```

```
MODULE sort_interf
INTERFACE
 SUBROUTINE sort(iv,n,how)
 INTEGER, DIMENSION(:), &
 INTENT(IN)::  iv(:)
 INTEGER, INTENT(IN) ::  n
 INTEGER, INTENT(INOUT)::  how
 END SUBROUTINE sort
END INTERFACE
END MODULE sort_interf
```

When the compiler performs the consistency checks for the argument list, it verifies that the actual arguments match (in number, type and order) the dummy arguments provided by the block interface declaration (and not the dummy arguments of the procedure itself!). Therefore, if there is a mismatch between the interface arguments and the procedure arguments, the compiler is fed wrong information, and a run time error may occur.

A direct way to provide an explicit interface for a procedure, without writing a block interface, is to write a module which *hosts the procedure itself*, then USE this module in all the program units that call the procedure. For our example, this can be achieved as follows.

```
PROGRAM impint
USE sort_code
IMPLICIT NONE
INTEGER, PARAMETER ::  n=4
INTEGER, DIMENSION(n) ::  iv
INTEGER ::  iw, how
iv=(/3,1,4,2/); how = -1
CALL sort(iv, n, how)
PRINT *, iv(1:n), how
END PROGRAM impint
```

```
MODULE sort_code
CONTAINS
 SUBROUTINE sort(iv,n,how)
 INTEGER, DIMENSION(:), &
 INTENT(IN)::  iv(:)
 INTEGER, INTENT(IN) ::  n
 INTEGER, INTENT(INOUT)::  how
................................
 END SUBROUTINE sort
END MODULE sort_code
```

Consider the TEST_SCOPE program, but now with MY_SUM an explicit procedure. We declare the interface as follows:

```
program test_scope
implicit none
interface
   subroutine my_sum(a,b,c)
    integer, intent(in) ::  a,b
    integer, intent(out) ::  c
   end subroutine my_sum
end interface
integer ::  i=1, j=2, k=3, glb=4, loc=5
...........................................
call my_sum(i,j,k)
...........................................
end program test_scope
```

Since

- internal procedures have visible interfaces, as they are build within the host;
- intrinsic procedures have explicit interfaces built in the language;

we need to write explicit interfaces for external procedures only.

Note that an explicit interface and the EXTERNAL attribute cannot be used simultaneously; this is no problem, since INTERFACEd procedures can be used as actual arguments.

9.10 Required Interfaces

Explicit interfaces are mandatory in the following situations.

1. the external procedure has dummy arguments that are assumed-shape arrays, pointers or targets. The compiler needs to figure out the rank, type and bounds of the array that need to be passed to the routine; pointers or targets. The compiler needs to figure out types and attributes that are to be transmitted to the procedure;
2. the external procedure has optional arguments; the compiler need to know the names of the arguments to figure out the correct association;
3. the external function returns an array or pointer valued result; the compiler needs to pass back info in a different form than usual;

4. the external character function contains an inherited LEN=* specifier; the compiler needs to know to pass string length info to and from the procedure;

5. the reference is a call to a generic name (extra info required);

6. the reference has a keyword argument (same reasons as for the optional case);

7. the reference is a defined assignment (extra info required);

8. the function reference is a call to a defined operator (extra info required);

9.11 Keyword Arguments

Normal argument correspondence is performed by position (n^{th} actual argument corresponds to n^{th} dummy argument). F90 allows us to circumvent this ‘‘fixed form’’ limitation. The arguments can be specified in any order, provided they are labeled by appropriate *keywords* at the site of call. A keyword is the name of the dummy variable in the procedure declaration (followed by an = sign); using keywords is necessary for the compiler to resolve the argument correspondence, if they are given out of order. Moreover, if a procedure has a long list of similar type arguments (say, 10 integer arguments) the use keywords greatly improves readability.

For example, the ‘‘Polar-to-Cartesian’’ subroutine

```
SUBROUTINE polar2cart(r, theta, x, y)
  REAL, INTENT(IN) :: r, theta
  REAL, INTENT(OUT) :: x, y
   ...
END SUBROUTINE polar2cart
```

can be called as

```
CALL polar2cart(1.0, 30.0, c1, c2)
```

or as

```
CALL polar2cart(r=1.0, theta=30.0, x=c1, y=c2)
```

or as

```
CALL polar2cart(x=c1, y=c2, r=1.0, theta=30.0)
```

9.12 Optional Arguments

Consider the following function

```
my_exp(x, tol, n_max)
```

which uses a Taylor polynomial (of order at most n_max) to approximate the value of EXP(x) within tol. We know that higher order Taylor polynomials approximate EXP better; the function chooses the optimal order for which the approximation error is less than tol; however, if tol is too small, a very large order of the approximating polynomial may be needed; in this situation the polynomial of order n_max is used, even if the error is larger than tol.

When we call the function we have to specify the argument and the desired tolerance, and the maximal order, for example

```
Y = my_exp(x, 1.E-6, 10)
```

This is clumsy, since most often we use, say, tol = 1.E-6 and n_max = 10; we would like to omit the last arguments when calling the function (and use implicitly the default values); but also retain the possibility of changing the defaults to some other values now and then.

We can do this in F90 by specifying the tol and n_max as optional arguments.

```
REAL FUNCTION my_exp(x, tol, n_max)
IMPLICIT NONE
REAL, INTENT(IN) :: x
REAL, INTENT(IN), OPTIONAL :: tol
REAL :: err_max
INTEGER, INTENT(IN), OPTIONAL :: n_max
INTEGER :: order_max
...
 IF( present(tol) ) THEN
    err_max = tol
 ELSE
   err_max = 1.e-6
 END IF
 IF( present(n_max) ) THEN
   order_max = n_max
 ELSE
   order_max = 10
 END IF
...
END FUNCTION my_exp
```

The OPTIONAL parameter allow tol and n_max to be omitted when the function is called. For example,

```
y = my_exp(x)
```

is a valid call and signifies that tol, n_max take the default values 1.E-6 and 10.

In the function body, the intrinsic function PRESENT(tol) returns .TRUE. if the current function call explicitly had tol as argument, and .FALSE. if tol was omitted. This distinction allows us to know when the optional arguments are not present and replace them by their default values.

Note that, when an optional argument is missing, the space on the stack associated with it is not allocated; in consequence missing optional arguments ‘‘do not exist’’ in the function body and we cannot read and write them (we can only test their existence with PRESENT()). For example the statement

```
IF( .not.present(tol))~ tol=1.e-6
```

is wrong, since it writes a variable that has not been allocated.

Finally, let us mention that only some of the optional arguments might be missing. For example

```
y = my_exp(x, 1.e-8)
```

is correct (tol=1.E-8 and n_max assumes the default value, 10). However, the statement

```
y = my_exp(x,12)
```

is incorrect, and , in particular, does NOT mean default tol and n_max=12. Whenever an optional argument is missing, the following arguments must be tagged with the proper keywords (otherwise the compiler has no way of knowing which argument is which). The correct call for default tol and n_max=12 is

```
y = my_exp(x,n_max=12)
```

In general it is always good practice to use keywords whenever the optional arguments are explicitly present in the procedure call.

Some built in functions (OPEN, READ, WRITE) have optional arguments and keywords - remember?

9.13 Recursive Functions

Recursion = procedures call themselves (either directly or indirectly). This is a powerful, but dangerous feature: if used incorrectly, the efficiency, and sometimes the meaning of the algorithm may suffer.

The declaration of a recursive function may be

```
INTEGER RECURSIVE FUNCTION fact(n) RESULT(n_fact)
```

The RESULT keyword specifies a variable where the result of the function is stored; this is necessary since the name of the function cannot be used to return the result for efficient implementation reasons. The value of dummy argument N_fact will be returned to the caller under the name fact; clearly, the type of the function (fact) must coincide with the type of the returned variable (N_fact); it is enough to declare only one of them. Above, we declared fact to be INTEGER, and this automatically declares N_fact INTEGER. Alternatively, we can declare N_fact, and this will automatically extend to fact (see example below).

In our previous lectures on functions we learned that, by default, the name of the return variable coincides with the name of the function. With any function (recursive or not), it is possible to rename the return variable using the statement RESULT(ret_var) in the function header.

The full example of the declaration and use of factorial function:

```
RECURSIVE FUNCTION fact(n) RESULT(n_fact)
 INTEGER, INTENT(IN) :: n
 INTEGER:: n_fact
 IF (n > 0) THEN
   n_fact = n*fact(n-1)
 ELSE
   n_fact = 1
 END IF
END FUNCTION fact
```

The function repeatedly calls itself, with decreased n, until the argument is 0, when the recursion begins to unwind.

9.14 Recursive Subroutines

In F90 subroutines can also be recursive. The factorial example can be implemented using a subroutine, as follows:

```
RECURSIVE SUBROUTINE factorial(n,n_fact)
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(OUT):: n_fact
  IF (n > 0) THEN
   CALL factorial(n-1,n_fact)
   n_fact = n*n_fact
  ELSE
   n_fact=1
  END IF
END SUBROUTINE factorial
```

Chapter 10

Parametrised Intrinsic Types

10.1 Rationale

Fortran 77 programs had some portability problems steming from the fact that the precision of data types is different on different processors. For example, a REAL variable is represented on 4 bytes on a UltraSparc processor, but is represented on 8 bytes on a Cray processor. Therefore, an F77 program that works all right on Cray may fail on UltraSparc, due to insuficient accuracy.

F90 addresses this portability problem using parametrised intrinsic types. The idea is that a processor can support different INTEGER, REAL COMPLEX, CHARACTER and LOGICAL representations; they are usually described in the compiler's manual. F90 allows to explicitly select a specific representation of a data type using a numerical parameter, called data types's **kind**.

Since several kinds of, say, REALs are allowed, we may ask how do we operate with them in a mixed expression, and how do we use them as procedure arguments. In mixed expressions lower accuracy kind terms are promoted to higher accuracy kinds; actual and dummy procedure arguments must match in both type and kind.

10.2 Parametrised Integers

Consider the following declarations

```
INTEGER(KIND=1) :: i1
INTEGER(KIND=2) :: i2
INTEGER(KIND=4) :: i4
INTEGER(KIND=8) :: i8
```

The variables i1, i2, i4, i8 are all of type INTEGER, but they have different KINDs; this means they will be stored using a different scheme. The presence of the numerical parameter KIND distinguishes between different kinds of integers; the exact meaning of a specific KIND parameter value is processor dependent.

For example, on an Alpha 21264 processor, i1 can hold numbers between -128 and $+127$; similarly, i2 can hold values between -32768 and 32767, i3 between -2147483647 and 2147483647, and i4 between -9223372036854775808 and 9223372036854775807; on Alpha 21264 KIND=p means storing the integer on p bytes; allowed values are p=1,2,4,8.

Numerical constants of a specific kind can be specified by appending an underscore and the kind number, for example

```
i1 = 39_1; i8 = 39876_8
```

In order to make the code completely portable, and to work with kind numbers transparently, we can specify the range of integer values. The function

```
selected_int_kind(2)
```

returns the minimal kind number able to hold integers in the range $(-10^2, 10^2)$. Therefore, we request the accuracy needed by the program, and let the compiler decide which kind number, on the processor at hand, satisfies this. One elegant use of the select function may look like

```
INTEGER, PARAMETER :: short = selected_int_kind(2), &
                     medium = selected_int_kind(3), &
                     long   = selected_int_kind(8), &
                     huge   = selected_int_kind(16)
INTEGER(KIND=short) :: i1
INTEGER(KIND=medium) :: i2
INTEGER(KIND=long :: i4
INTEGER(KIND=huge) :: i8
i1 = 39_short; i8 = 39876_huge
```

If there is no available kind that can accomodate the required range $(-10^p, 10^p)$, the function SELECTED_INT_KIND(p) returns -1 and we get an error at compile time.

10.3 Parametrized Reals

Since a floating point number is represented by mantissa and exponent, we need to select two parameters

P the precision, giving the number of decimal digits allowed in the mantissa, and

R the range of the decimal exponent.

The selector function

```
selected_real_kind(P=7, R=12)
```

returns a kind value that supports numbers with 7 decimal digits and exponent magnitudes in the range $(10^{-12}, 10^{+12})$.

Consider the following declarations

```
INTEGER, PARAMETER :: single = selected_real_kind(p=6, r=37), &
                     DOUBLE = selected_real_kind(p=14,r=307), &
                     quad   = selected_real_kind(p=33,r=4931)
REAL(KIND=single) :: x
REAL(KIND=DOUBLE) :: y
REAL(KIND=quad) :: z
```

On an Alpha 21264 processor the kind values are single = 4, double = 8, quad = 16. The variables x, y, z are of type REAL, but have different kinds; using parametrized declarations we can work at the level of accuracy desired. Since the kinds single, double and quad are defined via the selector function, we can count on the precision and range desired, regardless of the machine. The code is therefore portable.

10.4 Kind Functions

The function KIND(var) returns the kind of the variable var. For example, the code

```
INTEGER(KIND=4) :: i4
k = KIND(i4)
```

returns the value k = 4.

Intrinsic types have a predetermined kind value. With the KIND function we can inquire about the default kinds,

```
INTEGER :: i
REAL :: x
DOUBLE PRECISION :: y
ki = KIND(i); kx = KIND(x); ky = KIND(y)
\END{stlisting}
%
On Alpha 21264 the results are {\tt ki} = 4, {\tt kx = 4} and
{\tt ky = 8}.

All TYPE conversion functions have an OPTIONAL {\tt KIND} argument
for specifying the KIND of the RESULT (recall that the TYPE of the RESULT
is given by the FUNCTION itself).
For example,
%
\begin{lstlisting}{}
REAL :: x
INTEGER(KIND=huge) :: i8
i8 = int(x,KIND=huge) ! or i8 = int(x,kind=kind(i8))
```

10.5 Mixed KIND Expressions

In the expressions that contain mixed kind operands, lower accuracy operands are promoted to the higher accuracy kind. Also, the RHS is converted to the kind of LHS before assigning (this might create problems if the RHS has higher accuracy than LHS).

10.6 Parametrised Logical Vaiables

Logical variables can be represented differently. For example, on Alpha 21264, the declaration

```
LOGICAL :: l
k = KIND(l)
```

gives k=4 (which means that 4 bytes are used to store l). Since logical variables take only one of two possible values, it makes sense to try to store them using less memory; for example,

```
LOGICAL(KIND=1) :: l1
LOGICAL(KIND=2) :: l2
LOGICAL(KIND=4) :: l4
l1 = LOGICAL(ll,KIND=1); l2 = LOGICAL(ll,KIND=2)
```

l1 is stored on 1 byte, and l2 is stored on 2 bytes; conversion from one representation to another is done via the LOGICAL function, with the KIND= argument.

10.7 Parametrised Character

Different character sets can be selected by different kind values (US, greek, arabic, etc) The values are compiler-dependent, and not all systems support all character sets.

```
CHARACTER( KIND=character_set,LEN=10) :: greek
```

10.8 Kinds and Procedures

Actual arguments must *exactly match* the dummy arguments in

- Type,
- Kind, and
- Rank (for array arguments).

Therefore, whenever we use parametrised declarations, we have to be consistent throughout the program. A good practice is to define the kind parameters in a module, and use it in all program units. Note that, since the intrinsic default types are processor dependent, a truly portable program needs to use parametrized declarations.

With generic interfaces (to be studied later), each kind value needs its own procedure declaration; we should not mix intrinsic and parametrized types in a generic interface, since on some systems the intrinsic kind can overlap one defined kind which results in an error in the interface.

Chapter 11

Derived Types

11.1 Defininig Derived Types

Compound entities (similar to C ‘‘structures’’, or to Pascal ‘‘records’’) can be defined in F90; they are known as *derived types*.

For example, the three real coordinates of a 3D Point can be packed into a single variable. First, we define a new type in a derived-type statement

```
TYPE Point
  REAL :: x,y,z
END TYPE Point
```

An object of type Point can be declared in a type declaration statement

```
TYPE(Point) :: A, B
```

To select individual components of a derived type object, we use the % operator; for example

```
A%x = 1.0
A%y = 2.0
A%z = 3.0
```

assigns the values 1,2,3 to the individual components (coordinates) of A.

As an alternative to component-by-component assignment, it is possible to use a *derived type constructor* to assign values to the whole object. The derived type constructor is the type name followed by a paranthesised list of values, which will be assigned to the individual components. For example, the coordinate assignment of A can be solved using

```
A = Point( 1.0, 2.0, 3.0 )
```

Assignment between two objects of the same derived type is intrinsically defined (and is equivalent to component-by-component assignment). For example, the statement

```
B = A
```

has the effect of setting the x,y,z components of pt_B to 1,2 and respectively 3.

Note that, since F90 does not imply any form of storage association, there is no reason to suppose that objects of type Point occupy 3 contiguous REAL storage locations.

A new derived type can contain another derived type as one of its components; the derived type of the components must have already been declared or must be the type currently being declared. As an example of ''supertype'' consider

```
TYPE Sphere
  TYPE(Point) :: center
  REAL :: radius
END TYPE Sphere
TYPE(sphere) :: bubble
bubble%radius = 1.0
bubble%center%x = 0.2
bubble%center%y = 0.4
bubble%center%z = 0.6
bubble = Sphere( Point(0.2,0.4,0.6), 1.0 )
```

Finally, derived objects can be used in I/O statements similarly to the intrinsic objects. The statement

```
PRINT*, bubble
```

is equivalent to

```
PRINT*, bubble%center%x, bubble%center%y, bubble%center%z, bubble%radius
%
```

11.2 Arrays and Derived Types

◊ In F90 is possible to have an array of derived-type objects. For example

```
TYPE(Point), DIMENSION(4) :: tetrahedron
```

◊ It is also possible for a derived type to contain array components.

```
TYPE Pnt
 REAL, DIMENSION(3) :: x
END TYPE Pnt
TYPE Volume
  TYPE(Point), DIMENSION(4) :: tetrahedron
  INTEGER :: label
END TYPE Volume
TYPE(Volume), DIMENSION(100) :: diamond
```

The diamond is an object of type (''which has a'') Volume. Geometrically, the diamond has many facets, and we can conceptually ''create'' it (in a computer graphics program) by adjoining a number of tetrahedra. Each tetrahedron is described by its four corner points, and each corner point is given by its set of cartesian coordinates x_1,x_2 and x_3.

◊ We can reffer to a specific coordinate of one of the node points. For example

```
diamond(5)%tetrahedron(2)%x(1)
```

means 5th tetrahedron, 2nd node point, 1st coordinate.

We can also reffer to a subsection of the array component, provided that there is only one non-scalar index in the reference. For example,

```
diamond(:)%tetrahedron(2)%x(1)
diamond(5)%tetrahedron(:)%x(1)
diamond(5)%tetrahedron(2)%x(:)
```

are all correct, however the form

```
diamond(:)%tetrahedron(:)%x(1)
diamond(5)%tetrahedron(:)%x(:)
diamond(:)%tetrahedron(2)%x(:)
```

are incorrect, since *we can only section at most one component at a time.*

11.3 Derived Types and Procedures

◊ In F90 derived type objects can be passed as arguments in much the same manner intrinsic objects do. They can be given attributes (OPTIONAL, INTENT, dimension, SAVE, ALLOCATABLE, etc) However, some care needs to be exercised, as discussed below.

Consider the following sequence of code:

```
PROGRAM dist
IMPLICIT NONE
REAL, EXTERNAL :: distance
TYPE Point
  REAL :: x,y,z
END TYPE Point
TYPE(Point) ::a,b
a = point(1.,2.,3.)
b = point(4.,6.,8.)
PRINT*, distance(a,b)
END PROGRAM dist
!
REAL FUNCTION distance(a,b)
TYPE Point
  REAL :: x,y,z
END TYPE Point
TYPE(Point), INTENT(IN) :: a,b
distance = sqrt( (a\%x-b\%x)**2 + &
  (a%y-b%y)**2 + (a%z-b%z)**2 )
END FUNCTION distance \\
```

Everything seems fine, but in reality it is not, because of the following. The type Point definition in the main program and the type Point definition in the distance function *declare two different types*, with two different scopes (the definitions behave the same way two different local variable declarations would behave). Recall that the compiler represents the derived types however it sees fit, and it can happen that the main program type(Point) variables have a different storage scheme than the distance function's type(Point) variables.

The solution to this ''problem'' is to make the type definition accessible by either

- USE association (the type is defined in a module, which is then used by both the main program and the distance function), or
- Host association (the distance function is contained in the main program).

As a general rule, it is therefore preferable to always encapsulate type definitions in modules, and make them visible by USE association. For example, the module

```
MODULE Pnt
TYPE Point
 REAL :: x,y,z
END TYPE Point
END MODULE Pnt
```

can be used in both the main program and external function.

♢ Functions can return results of an arbitrary defined type. For example, consider the type of ''double precision complex numbers'', and the function that adds two double precision complex numbers returns a double precision complex result

```
MODULE dpc
 TYPE DP_Complex
  DOUBLE PRECISION :: re, im
 END TYPE DP_Complex
CONTAINS
 FUNCTION DPC_Add(a,b)
 TYPE(DP_Complex), INTENT(IN) :: a,b
 TYPE(DP_Complex) :: DPC_Add
  DPC_Sum%re = a%re + b%re
  DPC_Sum%im = a%im + b%im
 END FUNCTION DPC_Add
END MODULE dpc
```

Chapter 12

Pointers and Targets

12.1 Pointers and Targets

Unlike C pointers, F90 pointers are much less flexible and more highly optimized. The space to which a F90 *pointer* ‘‘points’’ is called a *target*.

Several restrictions

- F90 pointers are strongly typed: e.g. a pointer to a REAL, scalar target may not point to any other data type; similarly, a pointer to a REAL 1D array cannot point to a REAL 2D array;
- any variable that is pointed at must have the TARGET attribute;
- pointers are automatically dereferenced (the pointer name and the target name reffer to the same memory location);
- it is not possible to print out the address of the target of a pointer.

Benefits: provide a more flexible alternative to allocatable arrays, and allow the creation and manipulation of linked lists, etc.

12.2 Pointer Declaration

Include POINTER attribute. For example:

```
REAL, POINTER :: PtoR, PtoR2
```

PtoR, PtoR2 are pointers to a scalar, REAL target.

```
REAL, DIMENSION(:,:), POINTER :: PtoA
```

PtoA is pointer to a rank 2, REAL array of REALs.

Note that

- the declaration fixes the type, kind and rank of any possible target;

- pointers to arrays are declared using deffered-shape specification;
- rank of the target is fixed, but actual shape may vary;

POINTER attribute is incompatible with ALLOCATABLE or PARAMETER.

12.3 Target Declaration

```
REAL,TARGET :: x, y
REAL, DIMENSION(5,3),TARGET :: a, b
REAL, DIMENSION(4,7),TARGET :: c, d
```

x,y may become associated with PtoR, while a,b,c,d may become associated with PtoA.

The TARGET attribute is defined solely for optimization purposes. The compiler can assume that any nonpointer object not explicitly declared as a TARGET is only reffered to by its original name.

12.4 Pointer Assignment

Is a form of referring an object by different names - the pointer and its target reffer to the same space in memory. Pointer assignment takes place between a pointer variable and a target variable, or between two pointer variables.

```
PtoR => y
```

Pointer PtoR is associated with the target y, i.e. PtoR becomes an alias for y.

```
PtoA => b
```

Pointer PtoA is associated with the target b, i.e. PtoA becomes an alias for b.

```
PtoR2 => PtoR
```

Pointer PtoR2 is associated with the target of the pointer PtoR, i.e. is associated with y; now both PtoR and PtoR2 are aliases for y. This statement is correct since all pointer variables implicitly have the TARGET attribute (PtoR is here a target).

Note the difference between ‘‘pointer asignment’’ (=>, which makes the pointer and the target variables reference the same space) and ‘‘normal assignment’’ (=, which alters the value in the space reffered to by the LHS).

For example,

```
x=3.0
PtoR => y ! pointer assignment
PtoR = x !  y = x
```

The last statement effectively sets y to 3. Pointers in a ‘‘normal assignment’’ are automatically dereferenced; thus, PtoR is effectively an alias for y.

12.5 Association with Arrays

An array pointer may be associated with the whole target array, or with a regular section of the target (as long as the section has the correct rank).

For example, let

```
a = reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/))
```

and define

```
PtoA => a(1:3:2,2:3)
```

The vector assignment is valid, and we have that SHAPE(PtoA) = (/2,2/), SIZE(PtoA)=4; PtoA(1,1)=a(1,2)=4 and PtoA(2,1)=a(3,2)=6 etc.

We can associate pointers with pointers as follows:

```
PtoR\, => PtoA(2,2)
```

This means that PtoR, PtoA(2,2), a(3,3) are all aliases of the same memory space.

The pointer assignment

```
PtoA => a(1:3,2)
```

is not valid, since the rank of the target is not 2.

An array pointer cannot be associated with a vector-subscripted array section. For example, the following is invalid:

```
v = (/2,3,1/); PtoA => a(v,v)
```

12.5.1 Example: Array swapping

This example shows the usufulness of pointers. Suppose we want to swap 2 large arrays. The direct code will involve creating an extra array and copying a large amount of data 3 times:

```
REAL, DIMENSION(1000,1000):: A, B, Tmp
Tmp=A; A=B; B=Tmp ! 3 array copies involved
```

If we decide to work with pointers to arrays in the code swapping will only involve changing pointer values, which is considerably more efficient:

```
REAL, DIMENSION(1000,1000) :: mat1, mat2
REAL, DIMENSION(:,:), POINTER :: a, b, tmp
a=>mat1; b=> mat2      ! initial
tmp=>a; ! tmp points to the target of a, i.e. to mat1
a=>b;   ! a points to the target of b, i.e. to mat2
b=>tmp  ! b points to the target of tmp, i.e. to mat1
```

or even better:

```
REAL, DIMENSION(1000,1000), TARGET :: MatA, MatB
REAL, DIMENSION(:,:), POINTER :: A, A
A=>MatA; B=>MatB  ! normal
A=>MatB; B=>MatA  ! swapped
```

12.6 Status of a Pointer

Pointer Status:

- Undefined (initial status);
- Associated (the pointer has a target);
- Disassociated (the pointer is defined but has no target - ‘‘Null’’ value).

For example, the declaration

REAL, POINTER :: PtoR

leaves PtoR in *undefined* association status. We can bind the pointer to a target

PtoR => x

which changes the pointer status to associated. Finally, we can break the association with the target (without setting PtoR to point to a different target) by

```
NULLIFY(PtoR)
```

PtoR has now the ‘‘Null’’ values, and its status is *disassociated*.

The association status may be tested using the logical intrinsic function

```
associated(PtoR)
```

which returns .TRUE. if the pointer is associated, .FALSE. if not associated, or undefined if the pointer itself is undefined (because of this, it is a good idea to NULLIFY all pointers right after the declarations).

We can test the target of the pointer,

```
associated(PtoR,x)
```

which returns .TRUE. if the pointer is defined and associated to the specific target x, and .FALSE. otherwise.

The function is useful to avoid deallocation errors

```
IF ( associated(PtoA) ) DEALLOCATE(PtoA, STAT=ierr)
```

12.7 Dynamic Targets

The ALLOCATE statement can reserve space to be the target of a pointer; in this case the pointer is not so much an alias of another variable, but a reference to an unnamed part of the heap storage.

For example,

```
ALLOCATE(PtoR,STAT=ierr)
ALLOCATE(PtoA(n*n,2*k-1),STAT=ierr)
```

allocate new space for a single real, and then for a rank 2 array. If PtoA, PtoR were previously associated with other targets, those associations are broken (overwritten) by the ALLOCATE statements.

The deallocation statement

```
DEALLOCATE(PtoR,STAT=ierr)
```

breaks the connection between the pointer and its target and returns the freed space to the heap. The pointer remains in disassociated status.

We cannot use a pointer deallocate statement for objects that were not created by pointer allocation (e.g. allocate an allocatable matrix and associate a pointer to it, cannot just deallocate the pointer).

Note that nullifying the ‘‘allocated’’ pointer is possible

```
NULLIFY(PtoR)
```

but is not recommended; nullification breaks the pointer-target connection, but does not free the target storage; this space will become inaccessible until the program terminates, unless it is pointed to by at least one more pointer.

12.8 Pointers to Arrays vs Allocatable Arrays

We can create space dynamically using allocatable arrays, or using pointer allocation. In general, allocatable arrays are more efficient, while pointer allocation is more flexible.

There are two main restrictions imposed upon allocatable arrays which do not apply to pointer arrays:

- ALLOCATABLE arrays cannot be used as components of derived types (to be discussed later);
- unallocated ALLOCATABLE arrays cannot be pased as actual arguments to procedures.

For example,

```
PROGRAM ptralloc
IMPLICIT NONE
REAL, DIMENSION(:), ALLOCATABLE :: a
REAL, DIMENSION(:), POINTER :: b
  ALLOCATE(a(4))
  CALL t1(a)
  PRINT*, a
  CALL t2(b)
  PRINT*, b
CONTAINS
  SUBROUTINE t1(a)
    REAL, DIMENSION(:) :: a
    a = reshape((/1.,2.,3.,4./),(/4/))
  END SUBROUTINE t1
  SUBROUTINE t2(b)
    REAL, DIMENSION(:), POINTER :: b
    ALLOCATE(b(4));
    b = reshape((/5.,6.,7.,8./),(/4/))
  END SUBROUTINE t2
END PROGRAM ptralloc
```

12.9 Pointers and Procedures

Pointers can be passed as actual and dummy arguments in much the same way as non-pointer variables (dummies and actuals must match in type, kind and rank). Note that a POINTER variable cannot have the INYTENT attribute.

If a pointer or target is used as a dummy argument, then an explicit interface is required at the place of call. The reason is that a pointer argument can be interpreted in two ways:

- immediately dereference and pass the target (the dummy argument does not have the POINTER attribute);
- pass the pointer, so that it can be manipulated as a pointer in the procedure (the dummy argument has the POINTER attribute);

In the folowing example Pint2 is dereferenced before being passed to Beer, Pint1 is not:

```
PROGRAM Brew
  INTEGER, POINTER :: Pint1, Pint2
  CALL Beer(Pint1,Pint2)
  ...
CONTAINS
  SUBROUTINE Beer(arg1,arg2)
    INTEGER, POINTER :: arg1
    INTEGER, INTENT(IN) :: arg2
    ...
  END SUBROUTINE Beer
END PROGRAM Brew
```

Pint1 in the calling unit and arg1 in Beer reffer to the same space; however, if an unassociated pointer is the actual argument, and is associated in the procedure, it is not guaranteed that the pointer will still be associated after return.

12.10 Pointer Valued Functions

F90 allows functions to be pointer-valued. For example,

```
FUNCTION largest(a,b)
IMPLICIT NONE
INTEGER,TARGET,INTENT(IN) :: a,b
INTEGER,POINTER :: largest
IF (a.gt.b) THEN
 largest => a
ELSE
 largest => b
END IF
END FUNCTION largest
```

Using the generalized pointer assignment form

```
POINTER => ptr_expression
```

(ptr_expression returns a pointer result), the function name must identify a target by being the LHS of a pointer assignment

```
INTEGER, POINTER :: greater
greater => largest(a,b)
```

The interface of an external pointer valued function must always be explicit.

12.11 Pointers and Derived Types

♢ F90 allows pointers to derived types

```
TYPE(DP_Complex), TARGET :: a
TYPE(DP_Complex), POINTER :: Ptr_a
Ptr_a => a
```

♢ Derived types may contain pointer components

```
TYPE Student
 INTEGER :: id
 CHARACTER,DIMENSION(:),POINTER :: NAME
END TYPE Student
```

When the derived type contains pointer components, the structure asignment works as follows. Consider the code sequence

```
TYPE(Student) :: stud1, stud2
stud1 = stud2
```

The meaning is that

```
stud1%id = stud2%id    ! copy
stud1%NAME => stud2%NAME ! pointer assignment
```

Although it is not possible to have ALLOCATABLE arrays as components of derived types, we can still work with student names of different lengths using pointer components. Here the component name is a pointer to a dynamically sized 1-D array of characters. We use it as follows:

```
TYPE(Student) :: leader
leader%id = 1234
ALLOCATE(leader%NAME(8))
leader%NAME = (/"J","o","h","n"," ","D","o","e"/)
```

12.12 Example: Linked Lists

Pointer components may ‘‘point to’’ objects of any intrinsic type, any derived type *previously defined* or to the type being currently defined. Note that it is not possible to have a target type which has yet to be defined.

It is therefore possible to construct linked lists using derived types and pointer components. For example, objects of the type Node contain a value, an id number, a pointer to the previous node and one to the next node in the double linked list.

```
MODULE CircularList
IMPLICIT NONE
TYPE node
  INTEGER :: id
  REAL :: value
  TYPE(node), POINTER :: prev, next
END TYPE node
!
CONTAINS
!
   FUNCTION BuildCircularList(n)
     IMPLICIT NONE
     INTEGER, INTENT(IN) :: n
     TYPE(node), POINTER :: BuildCircularList
     INTEGER :: i, ierr
     TYPE(node), POINTER :: head, temp, curr
     !
     ALLOCATE(temp,STAT=ierr)
     IF (ierr.ne.0) THEN
       PRINT*, "allocation error"; RETURN
     END IF
     head%id = 1; head%value = 0.0
     !
     curr => head
     DO i=2,n
        ALLOCATE(temp,STAT=ierr)
        IF (ierr.ne.0) THEN
           PRINT*, "allocation error"; RETURN
        END IF
        temp%id = i; temp%value = 0.0
        curr%next => temp; temp%prev => curr; curr => temp
     END DO
     ! the next 2 lines circularly " close " the list
     curr%next => head; head%prev => curr
     BuildCircularList => head
  END FUNCTION BuildCircularList
!
END MODULE CircularList
```

```
PROGRAM test
USE CircularList
IMPLICIT NONE
  TYPE(node), POINTER :: f, b
  f => BuildCircularList(3)
  b => f
  DO i=1,7
    PRINT*, f%id, f%value, b%id, b%value
    f => f%next; b => b%prev
  END DO
END PROGRAM test
```

12.13 Arrays of Pointers

Despite the F90 syntax restrictions, it is possible to create de facto arrays of pointers. For example, the sequence

```
TYPE iptr
 INTEGER, POINTER :: comp
END TYPE iptr
TYPE(iptr), DIMENSION(100) :: I
```

declares a 100 dimensional vector whose components are pointers to integers. It is not possible to select sections of the I array

```
I(10)%comp    ! valid
I(10:20)%comp ! invalid
```

We can define an array of pointers to integer arrays

```
TYPE IAptr
 INTEGER, DIMENSION(:), POINTER :: comp
END TYPE Iptr
TYPE(IAptr), DIMENSION(100) :: IA
```

Each pointer component can then be made to point to an unnamed space in the heap,

```
ALLOCATE(IA(1)%comp(20), STAT=ierr)
```

or can be made to point to an existing array

```
INTEGER, TARGET :: JA(10)
ia(2)%comp => JA
```

Note that ALLOCATABLE arrays cannot be components of derived types.

12.14 Example: Sorting Vectors

Given N vectors of measurements (of different dimensions), write a program that sorts the set of vectors according to their mean; several vectors of equal mean are sorted according to their variance.

To represent each set of measurements, we choose the structure

```
TYPE meas_set
   REAL, DIMENSION(:), POINTER :: DATA
   INTEGER :: dim
   REAL :: mean, dev
END TYPE meas_set
```

where the number of measured values in the set, dim, may vary; data holds the measured values, mean and dev the mean value and the standard deviation of the set of measurements.

Suppose we have a total of N_Meas data sets. For reasons that will become clear later, we want to work with pointers to objects of type Meas_Set; thus, we construct

```
TYPE ptr_meas_set
   TYPE(meas_set), POINTER :: v
END TYPE ptr_meas_set
TYPE(ptr_meas_set), DIMENSION(n_meas) :: mx
TYPE(ptr_meas_set) :: tmp
```

Each measurement set will correspond to one entry in the vector mx.

We can read in the N_Meas measurement sets as follows. If there are still measured data sets to be read in, we allocate an object of the type Ptr_Meas_Set, then read in the number of points dim, allocate the vector data of dimension dim, read the measured values and store them in data.

```
DO i=1,n_meas
  ALLOCATE( mx(i)%v )
  IF (ierr.ne.0) .... ! some action
  READ(10,10) mx(i)%v%dim ! no. of data points
  ALLOCATE( mx(i)%v%DATA(mx(i)%v%dim), STAT=ierr) ! allocate data vector
  IF (ierr.ne.0) .... ! some action
  READ(10,10) mx(i)%v%DATA ! read data
END DO
```

After creating the storage and reading in the data, we can compute the mean and the deviation for each data set

```
DO i=1,n_meas
! compute the mean
  mx(i)%v%mean = 0.0
  DO j=1,mx(i)%v%dim
    mx(i)%v%mean = mx(i)%v%mean + mx(i)%v%DATA(j)
  END DO
  mx(i)%v%mean = mx(i)%v%mean/mx(i)%v%dim
! compute the deviation
  mx(i)%v%dev = 0.0
  DO j=1,mx(i)%v%dim
    mx(i)%v%dev = mx(i)%v%dev + (mx(i)%v%DATA(j)-mx(i)%v%mean)**2
  END DO
  mx(i)%v%dev = sqrt(mx(i)%v%dev/mx(i)%v%dim)
END DO
```

Now we need to sort the measured data in increasing mean order. Since the sorting algorithms move the data around, and we want to avoid repeated copying of large measurement vectors, we introduced the array of pointers mx. Interchanging to elements of mx means interchanging two pointer values; this is a cheap operation, when compared to interchanging two 10,000-elements vectors.

The simplest sorting algorithm might look like

```
DO k=1,n_meas
   DO i=1,n_meas-1
     IF( mx(i)%v%mean > mx(i+1)%v%mean ) THEN
         tmp = mx(i+1)
         mx(i+1) = mx(i)
          mx(i) = tmp
     END IF
   END DO
END DO
```

Chapter 13

Elements of object-oriented programming

This chapter needs to written.

13.1 Public and Private Entities

13.2 Derived Type Constructors

Can support default values, optional and keyword arguments. Subsequent modifications of the type internal structure can be made invisible to the user.

```
MODULE d3
 !
 TYPE coords
  PRIVATE
  REAL :: x,y,z
 END TYPE coords
 !
CONTAINS
 !
 TYPE(coords) FUNCTION init_coords(x,y,z)
  REAL, INTENT(IN), OPTIONAL :: x,y,z
  init\_coords = coords(0.0,0.0,0.0)
  IF(present(x)) init\_coords%x = x
  IF(present(y)) init\_coords%y = y
  IF(present(z)) init\_coords%z = z
 END FUNCTION init_coords
 !
 SUBROUTINE print_coords(c)
  TYPE(coords), INTENT(IN) :: c
  PRINT*, c%x,c%y,c%z
 END SUBROUTINE print_coords
 !
END MODULE d3
```

The components of a Coords type object are not visible to the user; they can only be accessed through the functions contained in the module d3. In particular, the components cannot be printed in a normal IO statement; a module procedure Print_Coords is needed for this.

13.3 Generic Procedures

- group a number of procedures with similar functionality (‘‘the overload set’’) under one name;
- the compiler decides which specific procedure to call based on the type, number, rank and kind of non-optional arguments (decision made at compile time);
- the overload set must be unambiguous.

```
MODULE gi
INTERFACE plus1
 MODULE procedure iplus1
 MODULE procedure rplus1
 MODULE procedure dplus1
END INTERFACE ! plus1
CONTAINS
INTEGER FUNCTION iplus1(x)
   INTEGER, INTENT(IN) :: x
   iplus1 = x + 1
END FUNCTION iplus1
REAL FUNCTION rplus1(x)
   REAL, INTENT(IN) :: x
   rplus1 = x + 1.0
END FUNCTION rplus1
DOUBLE PRECISION FUNCTION dplus1(x)
   DOUBLE PRECISION, INTENT(IN) :: x
   dplus1 = x + 1.0d0
END FUNCTION dplus1
END MODULE gi
```

The call plus1(2) will return an integer result, while plus1(2.0) will return a real result.

Chapter 14

Code Performance

In this section we discuss different elemenets that impact the performance of a Fortran code.

We should not perform machine-specific optimizations that destroy the portability or the readability of the code. We should rely on standard packages where available to ensure both performance and portability. An example is BLAS.

For the timings in this section we used the following machines:

- Alpha 21260 at 500 MHz, 512 MB RAM, Linux, native compiler (fort);
- Intel P-III 1 GHz, 256 MB RAM, Portland Group compiler (pgf90) and Pacific-Sierra compiler (vf90), which is a F90 to F77 translator plus Gnu g77 compiler;
- Ultra Sparc ?? with Sun's native compiler (f90).

All compilations used the default level of optimization (-O).

14.1 Measures of performance

The ultimate measure of performance is cpu time. Fortran defines several intrinsic procedures to measure time.

The intrinsic subroutine

```
system_clock( count, count_rate, count_max)
```

gives the system clock in clock ticks (count). The number of ticks per second is count_rate, and the maximum value of the tick counter is count_max (after which the tick counter resets).

A frequently used measure for performance is the number of floating point operations per second of cpu time (the flop rate).

14.2 The Computational Algorithm

The most important factor that determines performance is the computational algorithm. Two algorithms that provide similar results can be compared from the point of view of computational expense - the one that uses fewer operations is to be preffered. Of course, comparing two algorithms that do not provide similar results is meaningless.

14.2.1 Example: sorting

Sometimes better algorithms are available for the same problem. The classic example is bubblesort versus quick sort.

14.2.2 Example: the DFT versus FFT

14.3 Avoid Computations when their result is known

Best example: avoid multiplications by zero. Sparse matrices exploit this.

14.3.1 Example: multiplication of tridiagonal matrices

In this example we show how exploiting the particularities of the problem can lead to a more efficient solution.

We are given two tridiagonal matrices A, B, and we want to compute their product C.

```
INTEGER,PARAMETER::n=1024
REAL,DIMENSION(n,n) :: A,B,C
A=0.0; B=0.0
DO i=2,n-1
   A(i,i-1) = 1.0; A(i,i) = 2.0; A(i,i+1) = 1.0;
   B(i,i-1) = 2.0; B(i,i) =-4.0; B(i,i+1) = 2.0;
END DO
A(1,1)= 2.0; A(1,2)=1.0; A(n,n-1)=1.0; A(n,n)= 2.0
B(1,1)=-4.0; B(1,2)=2.0; B(n,n-1)=2.0; B(n,n)=-4.0
```

Consider first the direct code

```
c=0.0
CALL system_clock(count1,cr,cmax)
DO i=1,n
  DO j=1,n
    DO k=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    END DO
  END DO
END DO
CALL system_clock(count2,cr,cmax)
time=dble(count2-count1)/dble(cr)
PRINT*,'time=',time,' seconds.'
```

This algorithm performs n^3 multiplications and n^3 additions, a total of $2n^3$ flops. The time on Alpha is 151.5770 seconds, i.e. the rate 14.1676 Mflops/sec.

Alpha (fort)		Intel (pgf90)		Sun (f90)	
Time	Rate	Time	Rate		
151.5	14.1	11.3	188.9	61.1	31.1

Now, we do the same thing using the intrinsic matrix multiplication function,

```
c = matmul(a,b)
```

Alpha (fort)		Intel (pgf90)		Sun (f90)	
Time	Rate	Time	Rate		
146.2	14.6	13.8	155.7	5.2	405.3

Let us now notice that C is only pentadiagonal; we do not have to compute all n^2 entries, since most of them are known to have value 0. Moreover, each nonzero element of C is given by a scalar product of two vectors having at most 3 nonzero elements each. Therefore, we can avoid multiplications by zero to obtain the algorithm

```
C(1,1) = A(1,1)*B(1,1)+A(1,2)*B(2,1)
C(1,2) = A(1,1)*B(1,2)+A(1,2)*B(2,2)
C(1,3) = A(1,2)*B(2,3)
C(2,1) = A(2,1)*B(1,1)+A(2,2)*B(2,1)
C(2,2) = A(2,1)*B(1,2)+A(2,2)*B(2,2)+A(2,3)*B(3,2)
C(2,3) = A(2,2)*B(2,3)+A(2,3)*B(3,3)
C(2,4) = A(2,3)*B(3,4)
DO i=2,n-2
    C(i,i-2) = A(i,i-1)*B(i-1,i-2)
    C(i,i-1) = A(i,i-1)*B(i-1,i-1)+A(i,i)*B(i,i-1)
    C(i,i)  = A(i,i-1)*B(i-1,i)+A(i,i)*B(i,i)+A(i,i+1)*B(i+1,i)
    C(i,i+1) = A(i,i)*B(i,i+1)+A(i,i+1)*B(i+1,i+1)
    C(i,i+2) = A(i,i+1)*B(i+1,i+2)
END DO
C(n-1,n-3) = A(n-1,n-2)*B(n-2,n-3)
C(n-1,n-2) = A(n-1,n-2)*B(n-2,n-2)+A(n-1,n-1)*B(n-1,n-2)
C(n-1,n-1) = A(n-1,n-2)*B(n-2,n-1)+A(n-1,n-1)*B(n-1,n-1)+A(n-1,n)*B(n,n-1)
C(n-1,n)  = A(n-1,n-1)*B(n-1,n)+A(n-1,n)*B(n,n)
C(n,n-2)  = A(n,n-1)*B(n-1,n-2)
C(n,n-1)  = A(n,n-1)*B(n-1,n-1)+A(n,n)*B(n,n-1)
C(n,n)    = A(n,n-1)*B(n-1,n)+A(n,n)*B(n,n)
```

This algorithm performs about $38+13*(n-3)$ multiplications and additions, a huge reduction from n^3. The results are the same, but the computational time is considerably smaller (about 0.02 seconds).

14.4 Avoid redundant computations

You can save and reuse partial results.

14.5 Memory Effects

14.5.1 Cache

Insert brief explanation of memory hierarchy, cache memory, etc.

Idea: maximize the locality of data. Access data with stride 1.

14.5.2 Memory access patterns

Should preserve data locality, such that the data in cache memory is reused as much as possible. Loop ordering is therefore important for this.

```
DO j=1,n
  DO k=1,n
    DO i=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    END DO
  END DO
END DO
```

The results are summarized in the following table:

	Alpha (fort)		Intel (pgf90)		Intel (vf90)		Sun (f90)	
Loop order	Time	Rate	Time	Rate	Time	Rate		
ijk	148.4	14.5	11.3	188.9	9.20	233.4	61.1	31.1
kji	17.1	125.6	31.2	68.7	31.1	69.1	10.2	210.3
kij	345.8	6.2	212.6	10.1	209.3	10.2	138.1	15.6
jik	144.0	14.9	11.15	192.54	8.6	250.3	67.7	31.7
jki	14.02	153.1	7.0	305.82	6.9	311.2	10.78	199.1
ikj	348.1	6.2	213.1	10.1	220.7	9.73	134.1	16.0

The ‘‘winner’’ is the jki ordering in all cases. This is consistent with a column-wise storage of matrices in memory.

14.5.3 Loop Unrolling

Unroll the j-loop (level = 4)

```
DO j=1,n-3,4
  DO k=1,n
    DO i=1,n
      c(i,j)   = c(i,j)   + a(i,k)*b(k,j)
      c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
      c(i,j+2) = c(i,j+2) + a(i,k)*b(k,j+2)
      c(i,j+3) = c(i,j+3) + a(i,k)*b(k,j+3)
    END DO
  END DO
END DO
```

Alpha (fort)		Intel (pgf90)		Intel (vf90)		Sun (f90)	
Time	Rate	Time	Rate	Time	Rate		
6.1	347.8	13.2	162.5	14.5	148.4	8.4	255.3

This unrolling boosts the Alpha and Sun performance, but hurts the Intel.

Unroll the k-loop (level = 2)

```
DO j=1,n
  DO k=1,n-1,2
    DO i=1,n
      c(i,j) = c(i,j) + a(i,k)*b(k,j) + a(i,k+1)*b(k+1,j)
    END DO
  END DO
END DO
```

Alpha (fort)		Intel (pgf90)		Intel (vf90)		Sun (f90)	
Time	Rate	Time	Rate	Time	Rate		
11.7	183.2	5.3	402.0	5.9	360.3	6.7	320.0

This unrolling boosts the Intel performance.

Combining the j and k unrollings gives

Alpha (fort)		Intel (pgf90)		Intel (vf90)		Sun (f90)	
Time	Rate	Time	Rate	Time	Ratei	Time	Rate
5.23	410.4	8.14	263.6	8.51	250.5	6.0	355.6

The results of loop unrolling depend strongly on the system.

14.5.4 Reusing the data to minimize no. of fetches per computation step

14.5.5 Scalar copying

14.5.6 Blocking

14.5.7 BLAS

Standard libraries allow us to be very efficient on different architectures, since libraries are optimized specifically for them.

```
CALL SGEMM ( 'N', 'N', n, n, n, 1.0, a, n, b, n, 1.0, c, n )
```

The compiled code is linked to the library, for example

$$f90\ \ code.f90\ \ -lblas$$

Alpha (fort)		Intel (pgf90)		Sun (f90)	
Time	Rate	Time	Rate		
2.37	905.7	2.97	723.1	?	?

Chapter 15

Linear Systems of Algebraic Equations

15.1 Least Squares Fitting

Consider the following problem. We have a set of 21 data points, representing measurements of a rocket altitude during flight at 21 consecutive time moments. We know that, in an uniformly accelerated motion, the altitude as a function of time is given by

$$h(t) = at^2 + bt + c \tag{15.1}$$

From the data set

t	h(t)	t	h(t)	t	h(t)
0.0	-0.2173	0.7	3.3365	1.4	6.3312
0.1	1.0788	0.8	1.9122	1.5	6.3549
0.2	0.1517	0.9	3.0594	1.6	8.5257
0.3	0.1307	1.0	3.7376	1.7	8.7116
0.4	1.4589	1.1	3.3068	1.8	8.2902
0.5	2.9535	1.2	3.7606	1.9	11.4596
0.6	2.4486	1.3	6.6112	2.0	11.2895

we try to infer the parameters a, b, c, which define the smooth curve (16.1). In particular, the acceleration of the vehicle is $2a$, and, if m is the mass of the rocket and g is the gravitational acceleration, then we can infer the total force produced by thrusters $F = m(2a + g)$.

Note that we have 21 data points to determine 3 parameters; the data is corrupted by measurement errors (for example, the first altitude is negative!). We will use the redundancy in the data to ''smooth out'' these measurement errors; see Figure 16.1 for the distribution of data points and the parametrized curve.

At each time moment t_i, we are given the measured height h_i (in the data set); the height obtained by the formula (16.1) is $h(t_i) = at_i^2 + bt_i + c$. Therefore, the formula (16.1) ''approximates'' the measured height at time t_i with an error

$$e_i = h(t_i) - h_i = at_i^2 + bt_i + c - h_i \ .$$

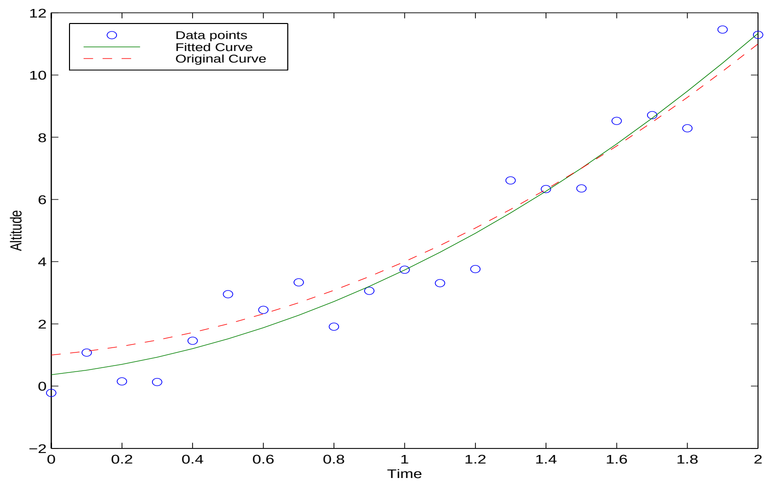


Figure 15.1: Altitude versus time: the data points and the resulting fitted quadratic curve.

We want the values a, b and c chosen such that the differences e_i between the model and the measurements are small. Therefore, we impose that the sum of squares of errors is minimized

$$g(a,b,c) = \sum_{i=1}^{21} e_i^2 = \sum_{i=1}^{21} \left(a t_i^2 + b t_i + c - h_i\right)^2$$

We recall from calculus that, when g attains its minimum value, the derivatives are equal to zero. therefore, to obtain a minimum of g, the following necessary conditions hold

$$\begin{array}{rcl} \frac{\partial g}{\partial a} &=& 2\sum_{i=1}^{21} \left(a t_i^2 + b t_i + c - h_i\right) t_i^2 &=& 0 \\ \frac{\partial g}{\partial b} &=& 2\sum_{i=1}^{21} \left(a t_i^2 + b t_i + c - h_i\right) t_i &=& 0 \\ \frac{\partial g}{\partial c} &=& 2\sum_{i=1}^{21} \left(a t_i^2 + b t_i + c - h_i\right) &=& 0 \end{array}$$

These equations form a linear system in the unknowns a, b, c, which, in matrix notation, is

$$\begin{bmatrix} \sum_{i=1}^{21} t_i^4 & \sum_{i=1}^{21} t_i^3 & \sum_{i=1}^{21} t_i^2 \\ \sum_{i=1}^{21} t_i^3 & \sum_{i=1}^{21} t_i^2 & \sum_{i=1}^{21} t_i^1 \\ \sum_{i=1}^{21} t_i^2 & \sum_{i=1}^{21} t_i^1 & \sum_{i=1}^{21} 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{21} h_i\, t_i^2 \\ \sum_{i=1}^{21} h_i\, t_i \\ \sum_{i=1}^{21} h_i \end{bmatrix} \qquad (15.2)$$

The computations give $a = 2.12$, $b = 1.2455$, $c = 0.3664$.

15.2 The problem

This presentation is devoted to solving linear systems of the form

$$\left\{ \begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & 5x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ & \vdots & & & & & & & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n \end{array} \right. \qquad (15.3)$$

The unknowns are $x_1 \cdots x_n$ and they need to satisfy *simultaneously* all n equations. Several simulaneous equations form a *system*; each equation is linear (involves only the first power of the unknowns), therefore we have a *linear system of equations*. The numbers a_{ij} are called the coefficients (a_{ij} is the coefficient of variable x_j in equation number i) and b_i are called the right hand sides.

Linear systems arise in a variety of applications in science and engineering.

15.3 Row Operations

Consider the linear system

$$\left\{\begin{array}{rcrcrcr} x_1 & + & 2x_2 & + & 3x_3 & = & 6 \\ 2x_1 & + & 5x_2 & + & x_3 & = & 8 \\ 3x_1 & + & 2x_2 & + & 4x_3 & = & 9 \end{array}\right. \tag{15.4}$$

We can see directly that the solution is $[x_1 = 1, x_2 = 1, x_3 = 1]$.

We can represent it in matrix form as

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \\ 9 \end{bmatrix}$$

or, in short,

$$A \cdot x = b$$

where A is the matrix of coefficients, b is the right hand side vector and x is the vector of unknowns.

If we multiply one equation by a constant (for example, multiply first equation by (-2)) we obtain an *equivalent system*, i.e. a system *with the same solution*. In our example, we can see that

$$\left\{\begin{array}{rcrcrcr} -2x_1 & + & (-4)x_2 & + & (-6)x_3 & = & -12 \\ 2x_1 & + & 5x_2 & + & x_3 & = & 8 \\ 3x_1 & + & 2x_2 & + & 4x_3 & = & 9 \end{array}\right.$$

has the solution $[x_1 = 1, x_2 = 1, x_3 = 1]$.

If we add two equations together, and replace the second equation by the result, we obtain an *equivalent system* also. For example, first equation plus the second give

$$0 \;+\; x_2 \;+\; -5x_3 \;=\; -4$$

and, when replacing the second equation by this result, we get

$$\left\{\begin{array}{rcrcrcr} -2x_1 & + & -4x_2 & + & -6x_3 & = & -12 \\ & & x_2 & + & -5x_3 & = & -4 \\ 3x_1 & + & 2x_2 & + & 4x_3 & = & 9 \end{array}\right.$$

This system has the same solution $[x_1 = 1, x_2 = 1, x_3 = 1]$.

In conclusion, multiplying one equation by a constant or replacing one equation by the sum of itself plus another equation lead to equivalent systems.

If we combine these two operations into a single step we conclude that *we can replace one equation by the sum of itself plus a multiple of another equation* without modifying the solution of the system. For example, multiplying the first equation of the system (15.4) by -2 and adding it to the second equation leads to the equivalent system

$$\left\{\begin{array}{rcrcrcr} x_1 & + & 2x_2 & + & 3x_3 & = & 6 \\ & & x_2 & + & -5x_3 & = & -4 \\ 3x_1 & + & 2x_2 & + & 4x_3 & = & 9 \end{array}\right. \tag{15.5}$$

15.4 The triangular form

We exploit the fact that row transformations do not change the solution of the system $Ax = b$ and transform the system succesively as follows.

In matrix notation, multiplying the first row by 2 and subtracting it from the second row leads to

$$A^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -5 \\ 3 & 2 & 4 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 6 \\ -4 \\ 9 \end{bmatrix}$$

We store the multiplier 2 as follows. Take a matrix $L = I$; since the multiplier is used to cancel A_{21}, we store it into L_{21}.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

Multiplying the first row by 3 and subtracting it from the third gives

$$A^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -5 \\ 0 & -4 & -5 \end{bmatrix} , \quad b^{(2)} = \begin{bmatrix} 6 \\ -4 \\ -9 \end{bmatrix} , \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix} .$$

Finally, by multiplying the second row by -4 and subtracting it from the third we obtain

$$A^{(3)} = U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -5 \\ 0 & 0 & -25 \end{bmatrix} , \quad b^{(3)} = \begin{bmatrix} 6 \\ -4 \\ -25 \end{bmatrix} , \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -4 & 1 \end{bmatrix} . \tag{15.6}$$

Now, all the elements in the sub-diagonal positions of $A^{(3)}$ are zero, and only the elements above the main diagonal are left. This transformed $A^{(3)}$ is in ''upper triangular form''; we usually denote it by U. In the same time, the matrix of multipliers L has only zero elements above the main diagonal; it is in ''lower triangular form'' (L stands for lower). Note that all the diagonal elements of L are 1, while the diagonal elements of U can take any values, without any restriction.

If we multiply L and U we obtain the original matrix A

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -5 \\ 0 & 0 & -25 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 2 & 4 \end{bmatrix} = A. \tag{15.7}$$

The relation $A = LU$ is called the ''LU decomposition'' of A. Since U has zeros below the main diagonal, we can use this space to store the elements of L; therefore, we can represent the LU decomposition compactly as

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & -5 \\ 3 & -4 & -25 \end{bmatrix}$$

15.5 Solving the upper triangular system

The upper triangular system

$$U\,x = b^{(3)} \ \text{(where } U = A^{(3)}\text{)} ,$$

is equivalent to the initial system

$$A\,x = b$$

in the sense that they admit the same solution x.

The upper triangular system is

$$\left\{\begin{array}{rcrcrcl} x_1 & + & 2x_2 & + & 3x_3 & = & 6 \\ & & x_2 & + & (-5)x_3 & = & -4 \\ & & & & -25x_3 & = & -25 \end{array}\right. \tag{15.8}$$

From the last equation $x_3 = 1$. We substitute this value in equation 2

$$x_2 - 5 \cdot 1 = -4$$

from which we infer that $x_2 = 1$.

Finally, we substitute the known values of x_2, x_3 in equation 1

$$x_1 + 2 \cdot 1 + 3 \cdot 1 = 6$$

and we obtain $x_1 = 1$.

In short, we start with the last equation and find x_n. Then, we move *backwards* through the equations; at each step we substitute the known values $x_{k+1} \ldots x_n$ in the current equation k, and compute the value of x_k. This process is called *backward substitution*.

15.6 General algorithms

For the LU decomposition we have

```
DO j=1,n-1
   DO i=j+1,n
      m = - A(i,j)/A(j,j)
      DO k=j+1,n
         A(i,k) = A(i,k) - m*A(j,k)
      END DO
      A(i,j) = m
   END DO
END DO
```

The number of multiplications and divisions used is

$$N_* = \sum_{j=1}^{n-1} \sum_{i=j+1}^{n} \left(1 + \sum_{k=j+1}^{n} 1 \right) \approx \frac{n^3}{3} \ .$$

Similarly, the number of sumations is $n^3/3$.

For the backward substitution the algorithm that

```
b(n) = b(n)/A(n,n)
DO j=n-1,1,-1
```

```
   DO k=j+1,n
      b(j) = b(j) - A(j,k)*b(k)
   END DO
   b(j) = b(j)/A(j,j)
END DO
```

The number of multiplications is about $n^2/2$.

15.7 Computing the determinant

Using the LU factorization

$$A = L \cdot U$$

we have that the determinant of A equals the product of the diagonal elements of U

$$\det(A) = \det(U) = U(1,1) \cdot U(2,2) \dots U(n,n) \ .$$

15.8 Pivoting

What happens with our method when applied to the system

$$A = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 2 & 4 \end{bmatrix} \ , \quad b = \begin{bmatrix} 5 \\ 8 \\ 9 \end{bmatrix} \ .$$

The unpleasant answer is that, although the system is well defined with solution $[x_1 = 1, x_2 = 1, x_3 = 1]$, our method simply fails. We try to zero $A(2,1)$ by subtracting the first row, multiplied by a suitable constant α, from the second row; since $A(1,1) = 0$, no matter what constant α we choose, the difference $A(2,1) - \alpha A(1,1) = A(2,1) \neq 0$.

We can remedy things if we change the order of the equations, such that $A(1,1) \neq 0$. Obviously, changing the order of the equations does not affect the solution. For example, interchanging equations 1 and 3 means permuting rows 1 and 3 in A, b:

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 2 & 5 & 1 \\ 0 & 2 & 3 \end{bmatrix} \ , \quad b = \begin{bmatrix} 9 \\ 8 \\ 5 \end{bmatrix} \ .$$

Note that we have two distinct choices when making $A(1,1) \neq 0$: interchange rows 1 and 3, or interchange rows 1 and 2. We will always choose the permutation that maximizes the absolute value of $A(1,1)$. The reason is that this choice minimizes the propagation of roundoff errors during the computation (a rigorous proof of this fact is beyond the scope of this class).

In our example, interchanging rows $1 \leftrightarrow 3$ makes $A(1,1) = 3$, while interchanging rows $1 \leftrightarrow 2$ makes $A(1,1) = 2$; hence we will use $1 \leftrightarrow 3$.

The process of interchanging rows in order to maximize $|A(i,i)|$ is called *partial pivoting* (''partial'' because the maximum is selected from $\{|A(i,i)|, |A(i+1,i)|, \dots, |A(n,i)|\}$, i.e. only from a ''part'' of the matrix).

15.9 Triangular form with pivoting

We bring again to a triangular form the system

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 2 & 4 \end{bmatrix} \quad b = \begin{bmatrix} 6 \\ 8 \\ 9 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

but this time we use pivoting; the $n-1$ dimensional vector P stores the row permutations, while L stores the multipliers as before.

For the first step, since the maximal element in the first column of A is $A(3,1) = 3$ we permute rows one and three, and store this permutation as $P(1) = 3$.

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 2 & 5 & 1 \\ 1 & 2 & 3 \end{bmatrix} \quad b = \begin{bmatrix} 9 \\ 8 \\ 6 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

As before, we multiply the first row by 2/3 (1/3) and subtract it from the second (third) row in order to zero the elements $A(2,1)$ and $A(3,1)$ respectively; the multipliers 2/3, 1/3 are stored in L in the appropriate positions.

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 0 & 11/3 & -5/3 \\ 0 & 4/3 & 5/3 \end{bmatrix} \quad b = \begin{bmatrix} 9 \\ 2 \\ 3 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1/3 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Now we process the second column. The maximum between $A(2,2)$ and $A(3,2)$ is $A(2,2) = 11/3$; no row permutation is necessary here, hence $P(2) = 2$. The second row is multiplied by 4/11 and subtracted from the third to get

$$U = \begin{bmatrix} 3 & 2 & 4 \\ 0 & 11/3 & -5/3 \\ 0 & 0 & 25/11 \end{bmatrix} \quad b = \begin{bmatrix} 9 \\ 2 \\ 25/11 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1/3 & 4/11 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Now the product LU gives

$$L\,U = \begin{bmatrix} 3 & 2 & 4 \\ 2 & 5 & 1 \\ 1 & 2 & 3 \end{bmatrix}$$

that is, the initial matrix A with rows 1 and 3 interchanged. In matrix language,

$$L \cdot U = \Pi \cdot A \ , \quad \Pi = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \ .$$

Note that, when pivoting is used, all the multipliers are less than or equal to 1. Compactly we can represent the LU decomposition of A as

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 2/3 & 11/3 & -5/3 \\ 1/3 & 4/11 & 25/11 \end{bmatrix} \quad , P = \begin{bmatrix} 3 \\ 2 \end{bmatrix} . \tag{15.9}$$

15.10 Operation count

The LU factorization algorithm uses $n^3/3$ multiplications and additions, while the forward/backward substitution uses $n^2/2$.

15.11 Singularity

Consider the system

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 7 & 4 \end{bmatrix} x = b = \begin{bmatrix} 6 \\ 8 \\ 14 \end{bmatrix}$$

It can be easily checked that all the following vectors are solutions to the system:

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \ , \quad x = \begin{bmatrix} -14 \\ -4 \\ 0 \end{bmatrix} \ , \quad x = \begin{bmatrix} -14t \\ -4+5t \\ t \end{bmatrix} \ , \text{ for any real } t \ .$$

Why does this system have a non-unique solution? The third row of A equals the sum of the first plus the second row, therefore A is singular.

The solution of the linear system $Ax = b$:

- A nonsingular $\Rightarrow$ there is a unique solution x;
- A singular $\Rightarrow$
  - if $b \in range(A)$ there are infinitely many solutions,
  - if $b \notin range(A)$ there are no solutions.

Singularity has to be detected and reported during the LU decomposition step. Numerically, if we apply the LU decomposition with pivoting to the example matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 7 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 7 & 4 \\ 2 & 5 & 1 \\ 1 & 2 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 7 & 4 \\ 0 & \frac{1}{3} & -\frac{5}{3} \\ 0 & -\frac{1}{3} & \frac{5}{3} \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 7 & 4 \\ 0 & \frac{1}{3} & -\frac{5}{3} \\ 0 & 0 & 0 \end{bmatrix} .$$

we end up with an upper triangular matrix U which has a zero diagonal element. When back-substituting we obtain the equation $0\,x_3 = 0$, which will hold for any value $x_3 = t$. When diagonal elements of U are zero (or very small) the system is (almost) singular.

15.12 Multiple systems with the same matrix

It is often the case that we need to solve consecutively several linear systems that share the same coefficient matrix

$$Ax^\alpha = b^\alpha \ , \quad Ax^\beta = b^\beta \ , \quad Ax^\gamma = b^\gamma \ldots$$

We can save substantial amounts of CPU time by computing the LU decomposition of the matrix A once, and then re-using it to obtain different solutions with different right hand sides.

In the standard solution, we applied to the right hand vector b all the permutations and row operations applied to A, then solved the upper tringular system

$$Ux = \text{modified\_RHS} \ .$$

To re-use the LU decomposition, we need to apply the permutations and row operations (stored in P and L respectively) to the new right hand vector b^α - that is, compute $L^{-1}\Pi c$, then solve the upper triangular system

$$Ux = \text{modified\_RHS} \ .$$

Now, all permutations are stored in P and all row operations in L, hence we can use this information directly. In matrix language,

$$Ax = c \Rightarrow \Pi A x = \Pi c \Rightarrow LUx = \Pi c \Rightarrow Ux = L^{-1}\Pi c$$

For example, the system

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \\ 3 & 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 15 \\ 19 \end{bmatrix} = c$$

has the same coefficient matrix as (15.4), but the right hand side vector is now c instead of b.

Therefore, we use the decomposition (15.9) as follows.

First, interchange rows (elements) 1 and $P(1) = 3$ in the right hand side vector

$$\begin{bmatrix} 19 \\ 15 \\ 14 \end{bmatrix}$$

Then, multiply first row by $L(2,1)$ and subtract it from the second row, and by $L(3,1)$ and subtract it from the third row, to obtain

$$\begin{bmatrix} 19 \\ 15 \\ 14 \end{bmatrix} \overset{L(2,1)=2/3}{\longrightarrow} \begin{bmatrix} 19 \\ 7/3 \\ 14 \end{bmatrix} \overset{L(2,1)=1/3}{\longrightarrow} \begin{bmatrix} 19 \\ 7/3 \\ 23/3 \end{bmatrix}$$

Note that $L(2,1)$ and $L(3,1)$ are now stored in $A(2,1)$ and $A(3,1)$ respectively.

Next, we need to interchange rows 2 and $P(2) = 2$ - nothing to do. Multiply second row by $L(3,1)$ and subtract it from the third row

$$\begin{bmatrix} 19 \\ 7/3 \\ 23/3 \end{bmatrix} \overset{L(3,1)=4/11}{\longrightarrow} \begin{bmatrix} 19 \\ 7/3 \\ 75/11 \end{bmatrix}$$

So far we have applied the permutations (P) and row transformations (L) to the new RHS vector c. It remains to solve the system

$$Ux = c$$

or

$$\begin{bmatrix} 3 & 2 & 4 \\ 0 & 11/3 & -5/3 \\ 0 & 0 & 25/11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 19 \\ 7/3 \\ 75/11 \end{bmatrix}$$

It is easy to see that $[x_1 = 1, x_2 = 2, x_3 = 3]$.

15.13 Computing the inverse of a matrix

There are some situations when computing the inverse of a matrix is needed. Based on the fact that

$$A \cdot A^{-1} = I$$

where I is the identity matrix, the inverse of A, $X = A^{-1}$, satisfies the equation

$$A \cdot X = I \ .$$

If we write explicitly the column composition of X and the identity matrix I we have

$$A \left[\begin{array}{c|c|c} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{array} \right] = \left[\begin{array}{c|c|c} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right]$$

From here we see that, columnwise, the above relation is equivalent to solving 3 systems with the same matrix A:

$$A \left[\begin{array}{c} x_{11} \\ x_{21} \\ x_{31} \end{array} \right] = \left[\begin{array}{c} 1 \\ 0 \\ 0 \end{array} \right] \ , \quad A \left[\begin{array}{c} x_{12} \\ x_{22} \\ x_{32} \end{array} \right] = \left[\begin{array}{c} 0 \\ 1 \\ 0 \end{array} \right] \ , \quad A \left[\begin{array}{c} x_{13} \\ x_{23} \\ x_{33} \end{array} \right] = \left[\begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right] \ .$$

For our example we get

$$A^{-1} = A \left[\begin{array}{r|r|r} -18/25 & 1/5 & 11/25 \\ 2/25 & 1/5 & -4/25 \\ 13/25 & -1/5 & -1/25 \end{array} \right]$$

15.14 Coding the LU decomposition

The F90 routine is given A, the coefficient matrix; it returns the LU factorization, stored compactly in A and the integer vector of permutations, or pivots, $Pivot$. Note that the content of A is overwritten by this routine.

An LU decomposition routine might look like:

```
SUBROUTINE LU_Fact(A,Pivot,Ierr)
IMPLICIT NONE
!
! The arguments:
!
! The input matrix, at exit
! will hold the LU factorization
REAL, DIMENSION(:,:), INTENT(INOUT) :: A
! Vector of permutations
INTEGER, DIMENSION(:), INTENT(OUT)   :: Pivot
! Singularity indicator, = 0 if A nonsingular,
! and = column number j if the first zero
! pivot was detected in column j
INTEGER, INTENT(OUT)   :: Ierr
!
! The local variables:
!
LOGICAL :: singular        ! Singularity flag
INTEGER :: i,j,k,n         ! DO loop variables
```

```
INTEGER, DIMENSION(:) :: p ! pivot location
REAL, DIMENSION(:) :: Tmp  ! Temporary row
REAL  :: uround            ! rounding unit
!
! Check if the argument is a square matrix
IF( size(A,1).NE.size(A,2) ) THEN
  PRINT*,"Error in Factorize: A must be square"
  RETURN
END IF
n=SIZE(A,1) ! the dimension of the matrix

ALLOCATE(Tmp(n),stat = k)
IF (k.NE.0) THEN
  PRINT*,"Error in Factorize: cannot allocate Tmp"; RETURN
END IF
!
ALLOCATE(p(n),stat = k)
IF (k.NE.0) THEN
  PRINT*,"Error in Factorize: cannot allocate p"; RETURN
END IF
!
Ierr = 0                      ! reset error indicator
singular = .FALSE.            ! reset singularity flag
uround = 1.0E-7               ! unit roundoff, set it to
                              ! 1.0D-14 for double precision
!
DO j=1,n-1  ! columns 1:n-1
!
  p=MAXLOC(ABS(A(j:n,j)))+j-1 ! Look for pivot, A(p(1),j)
  IF (p(1).NE.j) THEN         ! If pivot is non-diagonal
     Tmp(:)    = A(j,:)       ! permute rows j and p(1)
     A(j,:)    = A(p(1),:)
     A(p(1),:) = Tmp(:)
     Pivot(j) = p(1)          ! Save the pivot position
  ELSE
     Pivot(j) = j             ! Save the pivot position
  END IF
!
! If A(j,j)=0 then the matrix is singular
! uround is the rounding unit (machine epsilon)
  IF ( ABS(A(j,j)) .LT. uround  ) THEN
    Ierr = j                  ! Singularity Position
    singular = .TRUE.         ! Singularity Flag
    exit                      ! the ‘DO j’ loop
  END IF
!
  DO i=j+1,n                  ! rows to be processed
     A(i,j) = A(i,j)/A(j,j)   ! Store the multiplier
     A(i,j+1:n) = A(i,j+1:n) - A(i,j)*A(j,j+1:n)
  END DO          ! Row loop
!
END DO            ! Column loop
!
```

```
! If A(n,n)=0 then the matrix is singular
! uround is the rounding unit (machine epsilon)
IF ( abs(A(n,n)) .LT. uround  ) THEN
    Ierr = n                  ! Singularity Flag
    singular = .TRUE.
END IF
!
IF (allocated(Tmp)) DEALLOCATE(Tmp)
!
IF (.NOT. singular) THEN
   Ierr = 0
END IF
```

15.15 Coding the forward-backward substitution

The routine accepts the LU factorization of the coeficient matrix, stored compactly in A, the vector of pivots $Pivot$ and a right hand side vector b. It returns the solution of the system stored in the vector b (i.e. b is overwritten with the solution).

A sample substitution routine might be

```
! Check if the arguments match
IF ( size(A,1).NE.size(b,1) ) THEN
  PRINT*,"Error in solve: A and b must match"
  RETURN
END IF
n=SIZE(A,1) ! the dimension of the matrix
!
! Permute the vector b,   b <- Pb
!
DO j=1,n-1  ! columns 1:n-1
!
  IF (Pivot(j).NE.j) THEN     ! If pivot is non-diagonal
     Tmp         = b(j)       ! permute elements j and pivot(j)
     b(j)        = b(Pivot(j))
     b(Pivot(j)) = Tmp
  END IF
!
END DO

!
!  Forward Substitution,  y <- L\b
!
DO i=2,n
  b(i) = b(i) - DOT_PRODUCT( A(i,1:i-1), b(1:i-1) )
END DO
!
!
!  Backward Substitution, x <- U\y
!
b(n) = b(n)/A(n,n)
```

```
DO i=n-1,1,-1                ! elements to be processed
  b(i) = (b(i) - DOT_PRODUCT(A(i,i+1:n),b(i+1:n)))/A(i,i)
END DO
```

15.16 LAPACK and BLAS.

The best known librarys for linear algebra calculations is LAPACK (Linear Algebra PACKage). LAPACK uses the Basic Linear Algebra Subroutines (BLAS).

Factorization

The single precision LAPACK factorization function is sgetrf (single precision, general matrix, triangular factorization).

It is called by

$$\texttt{sgetrf}(\texttt{m}, \texttt{n}, \texttt{A}, \texttt{lda}, \texttt{Ipiv}, \texttt{Info})$$

The arguments are

m (input) Number of rows of A;

n (input) Number of columns of A;

A (input) The matrix A; and (output) the L and U factors;

lda (input) leading dimension of matrix A.

Ipiv (output) n-dimensional integer vector of permutations;

Info (output) the status of factorization. 0=success, $i > 0$ means singularity - $U(i,i) = 0$, $j < 0$ means incorrect j-th argument.

Substitution.

The single precision LAPACK substitution function is sgetrs (single precision, general matrix, triangular substitution).

It is called by

$$\texttt{sgetrs}(\texttt{Trans}, \texttt{n}, \texttt{nrhs}, \texttt{A}, \texttt{lda}, \texttt{Ipiv}, \texttt{b}, \texttt{ldb}, \texttt{Info})$$

The arguments are

Trans (input) $'N'$ means no transpose (solve for $Ax = b$); $'T'$ means transpose (solve for $A^Tx = b$);

n (input) Number of equations;

nrhs (input) Number of right hand sides;

A (input) The the L and U factors of matrix A as given by sgetrf;

lda (input) leading dimension of matrix A.

Ipiv (input) n-dimensional integer vector of permutations, as given by sgetrf;

b (input) the right hand side vector; and (output) the solution x. For multiple right hand sides b is a $n \times nrhs$ matrix.

ldb (input) leading dimension of vector b.

Info (output) the status of factorization. 0=success, $i > 0$ means singularity - $U(i,i) = 0$, $j < 0$ means incorrect j-th argument.

Compilation and Linkage.

We need to link our object code to the lapack and blas libraries

f90 file.f90 $-$lm $-$llapack $-$lblas

15.17 Homework

15.17.1 LU decomposition

Write a routine that computes the LU decomposition of a matrix with partial pivoting.

Input arguments:

- A, a square $n \times n$ matrix.

Output arguments:

- A, containing the compact storage of L and U; the input value of A is overwritten.
- $Pivot$, an integer $n-1$ dimensional vector, containing the pivot positions (i.e. the permutations).
- $Ierr$, an integer, indicating the column where a zero pivot was found (which means that the matrix is singular). $Ierr = 0$ if no zero pivots were found (nonsingular matrix).

The header and the declarations of this routine should be

```
subroutine LU_Fact(A,Pivot,Ierr)
implicit none
! The input matrix, at exit
! will hold the LU factorization
real, dimension(:,:), intent(inout) :: A
! Vector of permutations
integer, dimension(:), intent(out)   :: Pivot
```

```
! Singularity indicator, = 0 if A nonsingular,
! and = column number j if the first zero
! pivot was detected in column j
integer, intent(out)   :: Ierr
...................................
```

15.17.2 Back-substitution

Write a routine that solves a system after knowing its LU decomposition. Input arguments:

- A, a compact LU decomposition, as computed by the routine at point 1.
- $Pivot$, teh integer vector of permutations, as computed by the routine at point 1.
- b, a right hand vector.

Output arguments:

- b, containing the solution of the system; the input value of b is overwritten.

The header and the declarations of this routine should be

```
subroutine FB_Subst(A,b,Pivot)
implicit none
! The LU factorization, as given by LU_Fact
real, intent(in), dimension(:,:) :: A
! The RHS vector
real, intent(inout), dimension(:) :: b
! Vector of permutations, as given by LU_Fact
integer, dimension(:), intent(in)   :: Pivot
...................................
```

The routines LU_Fact and FB_Subst will both be submitted in the same file, *lin_solve.f90*.

15.18 Tests

Test your routines on the following systems:

Two systems with the same A.

```
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
A=  0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
```

```
      2.7912           6.2328
      2.7679           9.0263
b =   3.2772  and  c = 11.1428
      2.4658           6.0341
      2.5448           6.5785
```

The main program and all additional routines for this example will be submitted in a file *lin_twosys.f90*.

A singular system.

The following system is singular; apparently the solution is $[1,1,1,1]$, but your decomposition routine should signal singularity by encountering very small pivots.

```
    0.1934   0.1509   0.8537   0.8216        2.0196
A=  0.6822   0.6979   0.5936   0.6449    b=  2.6186
    0.8756   0.8488   1.4473   1.4665        4.6382
    0.2481   0.2646   0.6752   0.8198        2.0077
```

The main program and all additional routines for this example will be submitted in a file *lin_singular.f90*.

The Hilbert matrix

The $n \times n$ Hilbert matrix is defined as

$$A(i,j) = \frac{1}{i+j-1} \ , \qquad 1 \le i,j \le n \ ,$$

and we construct b such that the solution of the system $Ax = b$ has all the components equal to 1.

$$b(i) = \sum_{j=1}^{n} \frac{1}{i+j-1} \ , \quad 1 \le i \le n \ .$$

Write a routine that generates A and b for given n.

Find the inverse A^{-1} for $n = 2, 5$. What can you say about the errors in the solution? (subtract the exact solution from the numerical solution obtained).

The exact inverses are

```
  4    -6
 -6    12
```

```
    25       -300       1050      -1400        630
  -300       4800     -18900      26880     -12600
  1050     -18900      79380    -117600      56700
 -1400      26880    -117600     179200     -88200
   630     -12600      56700     -88200      44100
```

Solve the system $Ax = b$ for $n = 10, 30$. What can you say about the errors in the solution? (subtract the exact solution $[1, 1, \ldots 1]$ from the numerical solution obtained).

This is an example of an ill-conditioned system.

The main program and all additional routines for this example will be submitted in a file *lin_hilbert.f90*.

Chapter 16

Linear Least Squares

16.1 Least Squares Data Fitting

Consider the following problem. We have a set of 21 data points, representing measurements of a rocket altitude during flight at 21 consecutive time moments. We know that, in an uniformly accelerated motion, the altitude as a function of time is given by

$$h(t) = at^2 + bt + c \tag{16.1}$$

From the data set

t	h(t)	t	h(t)	t	h(t)
0.0	-0.2173	0.7	3.3365	1.4	6.3312
0.1	1.0788	0.8	1.9122	1.5	6.3549
0.2	0.1517	0.9	3.0594	1.6	8.5257
0.3	0.1307	1.0	3.7376	1.7	8.7116
0.4	1.4589	1.1	3.3068	1.8	8.2902
0.5	2.9535	1.2	3.7606	1.9	11.4596
0.6	2.4486	1.3	6.6112	2.0	11.2895

we try to infer the parameters a, b, c, which define the smooth curve (16.1). In particular, the acceleration of the vehicle is $2a$, and, if m is the mass of the rocket and g is the gravitational acceleration, then we can infer the total force produced by thrusters $F = m(2a + g)$.

Note that we have 21 data points to determine 3 parameters; the data is corrupted by measurement errors (for example, the first altitude is negative!). We will use the redundancy in the data to ‘‘smooth out’’ these measurement errors; see Figure 16.1 for the distribution of data points and the parametrized curve.

At each time moment t_i, we are given the measured height h_i (in the data set); the height obtained by the formula (16.1) is $h(t_i) = at_i^2+bt_i+c$. Therefore, the formula (16.1) ‘‘approximates’’ the measured height at time t_i with an error

$$e_i = h(t_i) - h_i = at_i^2 + bt_i + c - h_i \ .$$

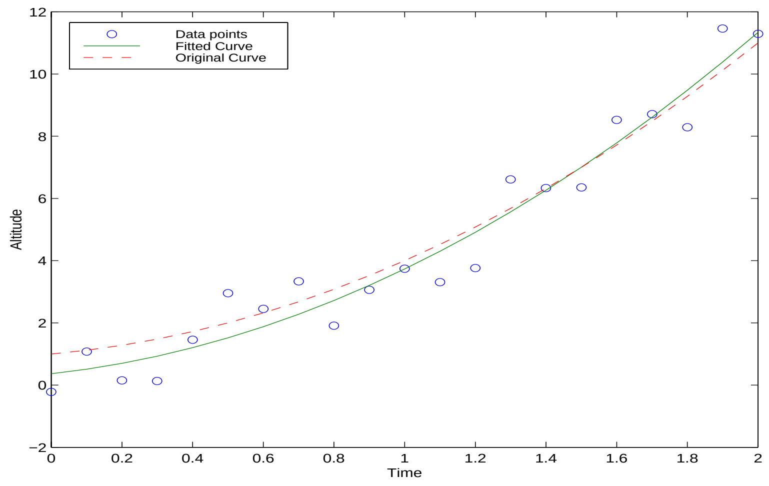


Figure 16.1: Altitude versus time: the data points and the resulting fitted quadratic curve.

We want the values a, b and c chosen such that the differences e_i between the model and the measurements are small. Therefore, we impose that the sum of squares of errors is minimized

$$g(a,b,c) = \sum_{i=1}^{21} e_i^2 = \sum_{i=1}^{21} \left(at_i^2 + bt_i + c - h_i\right)^2$$

We recall from calculus that, when g attains its minimum value, the derivatives are equal to zero. therefore, to obtain a minimum of g, the following necessary conditions hold

$$\begin{array}{rcccl} \frac{\partial g}{\partial a} & = & 2\sum_{i=1}^{21} \left(at_i^2 + bt_i + c - h_i\right) t_i^2 & = & 0 \\ \frac{\partial g}{\partial b} & = & 2\sum_{i=1}^{21} \left(at_i^2 + bt_i + c - h_i\right) t_i & = & 0 \\ \frac{\partial g}{\partial c} & = & 2\sum_{i=1}^{21} \left(at_i^2 + bt_i + c - h_i\right) & = & 0 \end{array}$$

These equations form a linear system in the unknowns a, b, c, which, in matrix notation, is

$$\begin{bmatrix} \sum_{i=1}^{21} t_i^4 & \sum_{i=1}^{21} t_i^3 & \sum_{i=1}^{21} t_i^2 \\ \sum_{i=1}^{21} t_i^3 & \sum_{i=1}^{21} t_i^2 & \sum_{i=1}^{21} t_i^1 \\ \sum_{i=1}^{21} t_i^2 & \sum_{i=1}^{21} t_i^1 & \sum_{i=1}^{21} 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{21} h_i\, t_i^2 \\ \sum_{i=1}^{21} h_i\, t_i \\ \sum_{i=1}^{21} h_i \end{bmatrix} \qquad (16.2)$$

The computations give $a = 2.12$, $b = 1.2455$, $c = 0.3664$.

16.2 The general problem formulation

We have a set of data points (t_i, y_i) for $i = 1, \cdots, m$ which, for example, represent time moments and measurements. We want to find a relationship (function) that gives $y(t)$. For this, we consider a set of n predefined functions $\{\Phi_0(t), \cdots, \Phi_n(t)\}$ and claim that the relationship is of the form

$$y(t) = f(t) = \sum_{i=0}^{n-1} a_i \Phi_i(t) \ , \qquad (16.3)$$

where the parameters a_i are unknown for the moment and will be determined based on the data. Usually the number of parameters is much smaller than the number of data pairs,

$$n \ll m \ .$$

For example, if $n = 3$ and we use polynomial functions $\Phi_0(t) = 1$, $\Phi_1(t) = t$, $\Phi_2(t) = t^2$ we recover the previous example, where we fitted a quadratic function; in this example $m = 21 \gg n = 3$.

To determine $\{a_i\}$ we insert the data in the relation (16.3) and obtain the following equation

$$\begin{aligned}
y_1 &= \sum_{i=0}^{n-1} a_i\Phi_i(t_1) = a_0\Phi_0(t_1) + \cdots + a_{n-1}\Phi_{n-1}(t_1) \ , \\
y_2 &= \sum_{i=0}^{n-1} a_i\Phi_i(t_2) = a_0\Phi_0(t_2) + \cdots + a_{n-1}\Phi_{n-1}(t_2) \ , \\
&\vdots \\
y_m &= \sum_{i=0}^{n-1} a_i\Phi_i(t_m) = a_0\Phi_0(t_m) + \cdots + a_{n-1}\Phi_{n-1}(t_m) \ .
\end{aligned}$$

Denoting

$$\Phi = \begin{bmatrix} \Phi_0(t_1) & \cdots & \Phi_{n-1}(t_1) \\ \vdots & \ddots & \vdots \\ \Phi_0(t_m) & \cdots & \Phi_{n-1}(t_m) \end{bmatrix} , y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} , a = \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} \ ,$$

the equations become

$$\Phi a = y \ .$$

This system has m rows (equations) and only n columns (unknowns), with $m \gg n$. Therefore we can only find a solution in an approximative sense.

16.3 The Calculus Approach

The residual (error) made is

$$r = y - \Phi a = \begin{bmatrix} y_1 - a_0\Phi_0(t_1) \ldots - a_{n-1}\Phi_{n-1}(t_1) \\ \vdots \\ y_m - a_0\Phi_0(t_m) \ldots - a_{n-1}\Phi_{n-1}(t_m) \end{bmatrix}$$

We try to find the vector of parameters a such that we minimize

$$\begin{aligned}
g(a) &= \sum_{i=1}^{m} \{y_i - a_0\Phi_0(t_i) - \cdots - a_{n-1}\Phi_{n-1}(t_i)\}^2 \\
&= r^T r = y^T y - 2a^T\Phi^T y + a^T\Phi^T\Phi a \ .
\end{aligned}$$

As in Calculus, for finding the minimum we take the derivative w.r.t. a and set it to zero,

$$\frac{\partial g}{\partial a} = -2\Phi^T y + 2\Phi^T\Phi a = 0$$

which is equivalent with the following *system of normal equations*

$$\Phi^T\Phi a = \Phi^T y \ .$$

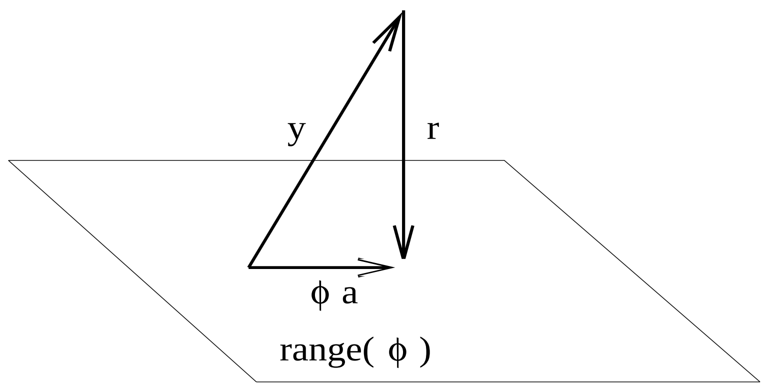


Figure 16.2: The residual has minimum norm when it is orthogonal to the range(Φ); from the orthogonality relation one can determine the least squares solution vector a.

16.4 Orthogonality

We want a such that $r \perp range(\Phi)$. This means that:

$$\Phi^T r = 0 \implies \Phi^T (y - \Phi a) = 0 \implies \Phi^T \Phi a = \Phi^T y \ .$$

16.5 Augmented system

The relations:

$$y = r + \Phi a \ , \quad \Phi^T r = 0 \ ,$$

can be written as

$$\begin{bmatrix} I & \Phi \\ \Phi^T & 0 \end{bmatrix} \begin{bmatrix} r \\ a \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix}$$

which is called the *augmented* system.

16.6 Matlab

In matlab the lsline command fits a line through a set of plotted data points. The overdetermined system $\Phi a = y$ can be solved in a least squares sense by simply $a = \Phi \ y$. The command p=polyfit(t,y,n) gives the coefficients of the degree n polynomial which fits the (t_i, y_i) data points in a least squares sense. To calculate the values of this polynomial at points x we use polyval(p,x).

16.7 Algorithm for polynomial least squares fitting

Given the m data points (t_i, y_i) for $1 \le i \le m$ (m is large), we want to find a degree n polynomial that fits the data in the least squares sense. This polynomial is

$$f(t) = a_0 + a_1 t + \cdots + a_n t^n \ .$$

To determine the $n+1$ coefficients $a_0 \cdots a_n$ we will use the method of normal equations. For this we build the $m \times (n+1)$ matrix Φ

$$\Phi = \begin{bmatrix} 1 & t_1 & \cdots & t_1^{n-1} & t_1^n \\ 1 & t_2 & \cdots & t_2^{n-1} & t_2^n \\ \vdots & & \ddots & & \vdots \\ 1 & t_m & \cdots & t_m^{n-1} & t_m^n \end{bmatrix} \;, \quad (\Phi)_{ij} = t_i^{j-1} \quad 1 \le i \le m, \quad 1 \le j \le n+1 \;,$$

and the vector of measurements y

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \;.$$

Φ has a special form and is called a Vandermonde matrix.

The system of normal equations is

$$Pa = b \;, \quad P = \Phi^T \Phi \;, \quad b = \Phi^T y \;, \quad a = \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} \;.$$

To determine the matrix P we apply the matrix multiplication formula:

$$P_{ij} = \sum_{k=1}^{m} \left(\Phi^T\right)_{ik} (\Phi)_{kj} = \sum_{k=1}^{m} (\Phi)_{ki} (\Phi)_{kj} = \sum_{k=1}^{m} (t_k)^{i-1} (t_k)_{j-1} = \sum_{k=1}^{m} t_k^{i+j-2} \;.$$

Clearly, $P_{ij} = P_{ji}$ so we need to determine only the elements of the upper triangle of P.

For the right hand side vector, we have

$$b_i = \sum_{k=1}^{m} \left(\Phi^T\right)_{ik} y_k = \sum_{k=1}^{m} (\Phi)_{ki} y_k = \sum_{k=1}^{m} t_k^{i-1} y_k \;, \quad 1 \le i \le n \;.$$

```
! Compute P
DO i=1,n+1
   DO j=1,i
      P(i,j) = 0.0
      ipower=i+j-2
      DO k=1,m
         P(i,j) = P(i,j) + t(k)**ipower
      END DO
      P(j,i) = P(i,j)
   END DO
END DO

! Compute b
DO i=1,n+1
    b(i) = 0.0
    DO k=1,m
       b(i) = b(i) + t(k)**(i-1)*y(k)
    END DO
END DO

! Solve the system Pa=b etc.
```

16.8 Note

In practice a much better numerical method for least squares approximations is the QR decomposition of Φ.

Chapter 17

Nonlinear Equations

17.1 The Newton Method for Nonlinear Equations

The Newton method is useful for solving nonlinear equations of the form

$$f(x) = 0 \ .$$

More exactly, given x_0, an initial, rough guess of the solution, the method builds a sequence of points

$$x_0 \ , x_1 \ , \ldots \ x_k \ , x_{k+1} \ , \ldots$$

which converge to the true solution of the problem

$$\lim_{k\to\infty} x_k = x_* \ ,$$

where

$$f(x_*) = 0 \ .$$

We start with an intial guess x_0. $f(x_*)$ can be expanded in Taylor series around the guessed solution x_0

$$0 = f(x_*) = f(x_0) + f'(x_0)(x_* - x_0) + \frac{1}{2}f''(x_0)(x_* - x_0)^2 + \cdots$$

Our purpose is to derive a better approximation for x_*. From the above formula we have

$$-f(x_0) = f'(x_0)(x_* - x_0) + \frac{1}{2}f''(x_0)(x_* - x_0)^2 + \cdots \quad \Longrightarrow \quad x_* = x_0 - \frac{f(x_0)}{f'(x_0)} - \frac{f''(x_0)}{2f'(x_0)}(x_* - x_0)^2 + \cdots$$

In this exact formula the higher order terms of the right hand side depend on the unknown x_*. We simply ignore them to arrive at the approximate relation

$$x_* \approx x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The obtained x_1 is (hopefully) a better approximation to x_* than the initial guess x_0 was. To obtain an even better approximation we repeat the procedure with x_1 the ''guess'' and arrive at

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

It is clear now that we can repeat the steps as many times as we need, untill x_n is sufficiently close to x_*.

The sequence of succesive approximations is built recursively using Newton's formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad , \quad k = 0, 1, 2, \ldots \tag{17.1}$$

From the formula we infer that

- each iteration requires one evaluation of the function f and one evaluation of its derivative f';
- the iteration step cannot be performed if $f'(x_k) = 0$.

Usually, if $f'(x_*) \neq 0$ and x_0 sufficiently close to x_* then Newton iterations converge very fast.

The order of convergence of an iterative procedure is the largest number p for which

$$|x_* - x_{k+1}| \leq C|x_* - x_k|^p$$

(here C is some constant which does not depend on k). The larger p is the faster the method converges.

For Newton's method $p = 2$; we say that *Newton's method converges quadratically.* To see this we develop $f(x_*)$ in Taylor series around x_k:

$$\begin{aligned}
f(x_*) &= f(x_k) + f'(x_k)(x_* - x_k) + \frac{1}{2}f''(x_k)(x_* - x^k)^2 + \ldots \\
&\quad \{\texttt{since } f(x_*) = 0\} \\
0 &= f(x_k) + f'(x_k)(x_* - x_k) + \frac{1}{2}f''(x_k)(x_* - x^k)^2 + \ldots \\
-f(x_k) &= f'(x_k)(x_* - x_k) + \frac{1}{2}f''(x_k)(x_* - x^k)^2 + \ldots \\
-\frac{f(x_k)}{f'(x_k)} &= x_* - x_k + \frac{f''(x_k)(x_* - x^k)^2}{2f'(x_k)} + \ldots \\
x_k - \frac{f(x_k)}{f'(x_k)} &= x_* + \frac{f''(x_k)}{2f'(x_k)}(x_* - x^k)^2 + \ldots \\
&\quad \{\texttt{since } x_{k+1} = x_k - f(x_k)/f'(x_k)\} \\
x_{k+1} &= x_* + \frac{f''(x_k)}{2f'(x_k)}(x_* - x^k)^2 + \ldots \\
x_{k+1} - x_* &= \frac{f''(x_k)}{2f'(x_k)}(x_* - x^k)^2 + \ldots \\
|x_{k+1} - x_*| &\leq \max_k \left\{ \left| \frac{f''(x_k)}{2f'(x_k)} \right| \right\} |x_* - x^k|^2 = C|x_* - x^k|^2
\end{aligned}$$

17.1.1 Order of convergence

To check experimentally that the order of convergence equals p we form the error ratios and see that they converge to a constant:

$$\frac{|x_* - x_{k+1}|}{|x_* - x_k|^p} \overset{\longrightarrow}{k \to \infty} Const.$$

Check this for the example

$$f(x) = x^4 - 9x^3 - 17x^2 + 225x - 200 \ , \quad x_* = 1 \ , \quad x_0 = 0 \ .$$

$$f(x) = x^5 - 7x^4 + 24x^3 - 48x^2 + 55x - 25 \ , \quad x_* = 1 \ , \quad x_0 = 0 \ .$$

17.2 The Secant Method

The sequence of succesive approximations is built recursively using the formula

$$x_{k+1} = x_k - \frac{f(x_k)}{s_k} \ , \quad k = 0, 1, 2, \ldots \tag{17.2}$$

With Newton's method $s_k = f'(x_k)$.

In order to avoid the (possibly expensive) derivative computation at each step, we use

$$s_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \ .$$

The order of convergence for the secant method is

$$p = \frac{1+\sqrt{5}}{2} \approx 1.6$$

(the proof is rather cumbersome and we skip it here). Since $1 < p < 2$ we say that *the secant method converges superlinearly*.

17.3 The Modified Newton Method

Again, the idea is to reduce the number of derivative computations without hurting the convergence of the method.

Modified Newton uses formula (17.2) with

$$s_k = f'(x_0) \ \text{,for all} \quad k \ .$$

that is

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_0)} \ , \quad k = 0, 1, 2, \ldots \tag{17.3}$$

This means that only one derivative computation is required, regardless of the number of iterations involved.

If after, say, m iterations the speed of convergence decreases, we need to update the derivative; set

$$s_k = f'(x_m) \ \text{,for all future} \quad k \ .$$

and continue with

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_m)} \ , \quad k = m, m+1, m+2, \ldots \tag{17.4}$$

Modified Newton is a method extensively used in scientific and engineering computations.

To obtain the order of convergence, we do the following calculations:

$$\begin{aligned}
f(x_*) &= f(x_k) + f'(x_k)(x_* - x_k) + \ldots \\
&\quad \{\texttt{since } f(x_*) = 0\} \\
0 &= f(x_k) + f'(x_k)(x_* - x_k) + \ldots \\
-f(x_k) &= f'(x_k)(x_* - x_k) + \ldots \\
-\frac{f(x_k)}{f'(x_0)} &= \frac{f'(x_k)}{f'(x_0)}(x_* - x_k) + \ldots \\
x_k - \frac{f(x_k)}{f'(x_0)} &= x_k + \frac{f'(x_k)}{f'(x_0)}(x_* - x_k) + \ldots \\
&\quad \{\texttt{since } x_{k+1} = x_k - f(x_k)/f'(x_0)\} \\
x_{k+1} &= x_k + \frac{f'(x_k)}{f'(x_0)}(x_* - x_k) + \ldots \\
x_{k+1} - x_* &= \left(1 - \frac{f'(x_k)}{f'(x_0)}\right)(x_k - x_*) + \ldots \\
|x_{k+1} - x_*| &\le \max_k \left\{ \left| 1 - \frac{f'(x_k)}{f'(x_0)} \right| \right\} |x_* - x^k| \\
&= C|x_* - x^k|
\end{aligned}$$

This means that $p = 1$ and we say that *the Modified Newton method converges linearly*. Of course, for convergence we need $C < 1$.

17.4 Fixed point iterations

The example

$$f(x) = x^5 - 7x^4 + 24x^3 - 48x^2 + 55x - 25 \ = 0 \ .$$

can be witten as

$$x = \frac{-x^5 + 7x^4 - 24x^3 + 48x^2 + 25}{55}$$

which can be transformed in the iterations:

$$x_{k+1} = \frac{-x_k^5 + 7x_k^4 - 24x_k^3 + 48x_k^2 + 25}{55}$$

In general the relation $f(x) = 0$ is rewritten as $x = g(x)$ and we use the iterations $x_{k+1} = g(x_k)$. These iterations converge if $|g'| < 1$.

For linearly convergent iterations we define the rate C as the limit

$$\frac{|x_* - x_{k+1}|}{|x_* - x_k|} \overset{\longrightarrow}{k \to \infty} Const.$$

17.5 The Newton Method for Systems

The Newton method is useful for solving nonlinear systems of equations of the form

$$\left\{\begin{array}{rcl} f_1(x_1,x_2,\ldots,x_n) & = & 0 \\ f_2(x_1,x_2,\ldots,x_n) & = & 0 \\ & \vdots & \\ f_n(x_1,x_2,\ldots,x_n) & = & 0 \end{array}\right. \tag{17.5}$$

There are n independent variables $x_1,x_2,\ldots,x_n$; in vector notation, these variables are the entries of the vector variable $\vec{x}$

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

The system is defined by n functions $f_1,f_2,\ldots,f_n$ (each f_i is a function of the n variables $x_1,x_2,\ldots,x_n$). In vector notation,

$$\vec{f}(\vec{x}) = \begin{bmatrix} f_1(x_1,x_2,\ldots,x_n) \\ f_2(x_1,x_2,\ldots,x_n) \\ \vdots \\ f_n(x_1,x_2,\ldots,x_n) \end{bmatrix}$$

Therefore, the system of equations (17.5) can be written compactly as

$$\vec{f}(\vec{x}) = \vec{0} \ .$$

We want to find *a solution of the system*, i.e. a set of values $\vec{x}_1^*,\ldots,\vec{x}_n^*$ such that all the equations are simultaneously satisfied

$$\vec{f}(\vec{x}^*) = \begin{bmatrix} f_1(x_1^*,x_2^*\ldots,x_n^*) \\ f_2(x_1^*,x_2^*\ldots,x_n^*) \\ \vdots \\ f_n(x_1^*,x_2^*\ldots,x_n^*) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0} \ .$$

Newton's method builds a sequence of points

$$\vec{x}^0 \ , \ \vec{x}^1 \ , \ \ldots \ \vec{x}^k \ , \ \vec{x}^{k+1} \ , \ldots$$

which converge to the true solution of the problem

$$\lim_{k\rightarrow\infty} \vec{x}^k = \vec{x}^* \ .$$

In one dimension, the sequence is built using Newton's formula

$$x^{k+1} = x^k - \left(f'(x^k)\right)^{-1} f(x^k) \ , \quad k = 0,1,2,\ldots \tag{17.6}$$

For n-dimensional problems, this formula generalizes to

$$\vec{x}^{k+1} = \vec{x}^k - \left(F(x^k)\right)^{-1} \vec{f}(x^k) \ , \quad k = 0,1,2,\ldots \tag{17.7}$$

The place of the derivative f' is now taken by the *Jacobian* matrix F defined as

$$F(\vec{x}) = \left(\frac{\partial f_i}{\partial x_j}\right)_{1\le i,j\le n} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Let

$$\vec{\delta}^k = \vec{x}^{k+1} - \vec{x}^k \ , \quad \vec{x}^{k+1} = \vec{x}^k + \vec{\delta}^k \ .$$

The Newton formula (17.7) can be written as

$$F(\vec{x}^k)\vec{\delta}^k = -\vec{f}(\vec{x}^k) \ . \tag{17.8}$$

This is a system of linear equations; F is a $n\times n$ matrix, $\vec{\delta}$ and and $\vec{f}$ are n-dimensional vectors.

One step of the Newton method (say, the k^{th} step) proceeds as follows:

0 We have $\vec{x}^k$ available, and want to compute the next iterate, $\vec{x}^{k+1}$;

1 Evaluate the vector function $\vec{f}(\vec{x}^k)$ (we evaluate each component function $f_1 \ldots f_n$ individually);

2 Evaluate the derivative matrix $F(\vec{x}^k)$ (each entry $\partial f_i/\partial \vec{x}^j$ is a function of $\vec{x}^1 \ldots \vec{x}^n$ and need to be evaluated individually);

3 Solve the system (17.8). For this, we need to

  3.1 compute the LU decomposition (with pivoting) of the matrix $F(\vec{x}^k)$;

  3.2 apply the back-substitution algorithm to the right hand side $-\vec{f}(\vec{x}^k)$, to obtain the solution $\vec{\delta}^k$;

  This step cannot be performed if $F(\vec{x}^k)$ is singular; if this happens we need to terminate the computations.

4 Compute the next iterate as $\vec{x}^{k+1} = \vec{x}^k + \vec{\delta}^k$.

Note that one Newton step is very expensive. We have to evaluate $\mathcal{O}(n^2)$ functions (the entries of $F(\vec{x})$) and we need to calculate the LU decomposition of F each step.

17.6 The Modified Newton Method for Systems

The idea is to reduce the number of derivative evaluations and the number of LU decompositions without hurting the convergence of the method.

The modified Newton formula is

$$\vec{x}^{k+1} = \vec{x}^k - A^{-1}\vec{f}(x^k) \ , \quad k = 0,1,2,\ldots \tag{17.9}$$

with

$$A = F(x^0) \ .$$

The modified Newton step can be equivalently written as

$$A\vec{\delta}^k = -\vec{f}(\vec{x}^k) \ , \quad \vec{x}^{k+1} = \vec{x}^k + \vec{\delta}^k \ . \qquad (17.10)$$

This means that only one Jacobian ($A = F(\vec{x}^0)$ computation is required, and only one LU decomposition is performed, regardless of the number of iterations involved. All the linear systems (17.10) share the same ''coefficient'' matrix A, and therefore we can re-use the same LU decomposition.

With modified Newton, we start with computing the matrix $A = F(\vec{x}^0$ and its LU decomposition (with pivoting!). The algorithm does not work if $F(\vec{x}^0)$ is singular; if this happens we need to terminate the computations.

One step of the Modified Newton algorithm follows.

0 We have $\vec{x}^k$ available, and want to compute the next iterate, $\vec{x}^{k+1}$;

1 Evaluate the vector function $\vec{f}(\vec{x}^k)$ (we evaluate each component function $f_1 \ldots f_n$ individually);

2 Solve the system (17.10). For this, we use the available LU decomposition of A. Apply the back-substitution algorithm to the right hand side $-\vec{f}(\vec{x}^k)$, to obtain the solution $\vec{\delta}^k$;

3 Compute the next iterate as $\vec{x}^{k+1} = \vec{x}^k + \vec{\delta}^k$.

If after, say, m iterations the speed of convergence decreases, we need to update the derivative; set

$$A = \vec{f}(\vec{x}^m) \ ,$$

and continue with (17.9) for $k \geq m$.

From the third iteration on we will monitor the progress of the iterations. For a good starting point, if the iterations proceed all right, we have

$$\|\vec{x}^{k+1} - \vec{x}^k\| \leq \theta_k \|\vec{x}^k - \vec{x}^{k-1}\|$$

Estimate

$$\theta_k \approx \frac{\|\vec{x}^{k+1} - \vec{x}^k + \epsilon\|}{\|\vec{x}^k - \vec{x}^{k-1}\epsilon\|}$$

(with $\epsilon \approx 10^{-8}$ to guard against division by zero). If $\theta_k > 0.9$ then the progress of the iterations is not satisfactory; with modified Newton we have to update the derivative,

$$A = F(\vec{x}^{k+1})$$

and continue with the iterations.

If $\theta_k > 1$ for 3 or 4 consecutive steps the iteration is likely to diverge, and a better starting point should be chosen; print the proper message and exit gracefully.

Chapter 18

Polynomial Interpolation

18.1 Example

Given the $m+1$ data points (t_i, y_i) for $0 \le i \le m$, we want to find a degree m polynomial that ‘‘passes through’’ all the data points.

18.2 Direct algorithm for polynomial interpolation

Given the $m+1$ data points (t_i, y_i) for $0 \le i \le m$, we want to find a degree m polynomial that ‘‘passes through’’ all the data points. This polynomial is

$$p(t) = a_0 + a_1 t + \cdots + a_m t^m \ .$$

To determine the $m+1$ coefficients $a_0 \cdots a_m$ we write the interpolating conditions which require the value of the polynomial in t_i be equal to y_i

$$\begin{array}{lcl} p(t_0) = y_0 & \Longrightarrow & a_0 + a_1 t_0 + \cdots a_m t_0^m = y_0 \\ p(t_1) = y_1 & \Longrightarrow & a_0 + a_1 t_1 + \cdots a_m t_1^m = y_1 \\ & \vdots & \\ p(t_m) = y_m & \Longrightarrow & a_0 + a_1 t_m + \cdots a_m t_m^m = y_m \end{array}$$

These relation can be formulated as an $(m+1) \times (m+1)$ linear system:

$$\Phi\, a = y \ ,$$

with the system matrix Φ (it has a special form and is called a Vandermonde matrix)

$$\Phi = \left[\begin{array}{ccccc} 1 & t_0 & \cdots & t_0^{m-1} & t_0^m \\ 1 & t_1 & \cdots & t_1^{m-1} & 1_2^m \\ \vdots & & \ddots & & \vdots \\ 1 & t_m & \cdots & t_m^{m-1} & t_m^m \end{array} \right] \ , \quad (\Phi)_{ij} = t_{i-1}^{j-1} \quad 1 \le i,j \le m+1 \ .$$

The vector of measurements y and the vector of polynomial coefficients a are

$$y = \left[\begin{array}{c} y_1 \\ \vdots \\ y_m \end{array} \right] \ , \quad a = \left[\begin{array}{c} a_0 \\ \vdots \\ a_m \end{array} \right] \ .$$

18.3 Nested polynomial evaluation

We want to evaluate the value of the polynomial p at a certain point z. Using the ''nested'' factorization

$$\begin{aligned} p(z) &= a_0 + a_1 z + a_2 z^2 + \cdots + a_{m-2} z^{m-2} + a_{m-1} z^{m-1} + a_m z^m \\ &= a_0 + z\left(a_1 + z\left(\cdots z\left(a_{m-2} + z\left(a_{m-1} + z a_m\right)\right)\cdots\right)\right) \end{aligned}$$

the value $p(z)$ can be evaluated in only m multiplications and m additions: we initialize $p = a_m$ and for every i from $m-1$ to 0 we update $p^{\rm new} = a_i + z p^{\rm old}$.

18.4 Lagrange form of the interpolation polynomial

18.5 Newton form of the interpolation polynomial

18.6 Reconstructing functions

Sometimes not possible, see Runge's function. Between the interpolating points the polynomial ocillates wildely.

18.7 Chebyshev points

Equidistant n points in $[-1,1]$ are

$$t_k = -1 + 2\frac{k-1}{n-1} \ , \quad k = 1, \cdots, n \ .$$

Chebyshev points are

$$t_k = -\cos\left(\frac{(2k-1)\pi}{2n}\right) \ , \quad k = 1, \cdots, n \ .$$

Chapter 19

Numerical Integration

19.1 The Numerical Integration Problem

We have learned in Calculus how to compute the definite integral

$$\int_a^b f(x)dx$$

where $f(x)$ is a continuous function. If $F(x)$ is the *antiderivative* of $f(x)$, then the Fundamental Theorem of Calculus states that

$$\int_a^b f(x)dx \quad = \quad F(b) - F(a) \ .$$

In most practical situations, however, the antiderivative is not available. For example, the Gaussian integral

$$\int_0^1 e^{-x^2} dx$$

cannot be evaluated analytically, since the antiderivative of e^{-x^2} is not an elementary function.

The purpose of this project is to compute numerically the definite integral $\int_a^b f(x)dx$; in other words, we want an algorithm (that will translate in a piece of code) to compute an approximation of the integral, up to the desired accuracy.

In Calculus we learned that the integral is a limit of *Riemann sums*. On a computer, we also approximate the integral by a sum. We consider the set of *node points* $\{x_i\}$

$$a = x_0 < x_1 < x_2 < \ldots < x_{n-1} < x_n = b$$

and approximate the integral by the sum

$$\int_a^b f(x)dx \quad \approx \quad \sum_{i=0}^{n} w_i f(x_i) \ .$$

There are many possible, meaningfull choices for the node points values x_i and the values of the weights w_i; different sets of values define different integration methods.

In what follows we focus on two simple algorithms: the trapezoidal and the Simpson method. All the methods discussed here use an equidistant set of node points. More precisely, if we define the *step*

$$h = \frac{b-a}{n}$$

the node points are

$$x_i = a + i\,h \ , \quad i = 0, 1, \ldots, n \ .$$

19.2 The Trapezoidal Method

This method was extensively discussed in class. The integral is approximated by the sum

$$T_n = \frac{h}{2} \left(f(x_0) + 2\,f(x_1) + \ldots + 2\,f(x_{n-1}) + f(x_n) \right)$$

Note that the weights are

$$w_i = \left\{ \begin{array}{lll} h/2 & \text{for} & i = 0 \text{ and } i = n \\ h & \text{for} & 1 \le i \le n-1 \end{array} \right.$$

19.3 The Simpson Method

The Simpson method uses quadratic approximations of the function on each subinterval $[x_{2j}, x_{2j+2}]$ (three function values, at x_{2j},x_{2j+1} and x_{2j+2} are used to determine uniquely the quadratic polynomial). We add up all the integrals of the quadratic polynomials on all the subintervals, to obtain an approximation of the function integral over $[a,b]$. This approximation is (the Simpson formula)

$$S_n = \frac{h}{3} \left(f(x_0) + 4\,f(x_1) + 2\,f(x_2) + 4\,f(x_3) + 2\,f(x_4) \ldots + f(x_n) \right)$$

Note that the weights are

$$w_i = \left\{ \begin{array}{lll} h/3 & \text{for} & i = 0 \text{ and } i = n \\ 2h/3 & \text{for} & i \text{ even}, \ 1 \le i \le n-1 \\ 4h/3 & \text{for} & i \text{ odd}, \ 1 \le i \le n-1 \end{array} \right.$$

An efficient implementation of the algorithm that computes S_n for different values of n might use 3 variables:

$$\begin{array}{rcl} \mathsf{sum\_end_n} & = & \mathsf{f(a) + f(b)} \\ \mathsf{sum\_odd_n} & = & \sum_{1 \le i < n, \ i \ \mathrm{odd}} \mathtt{f}(\mathtt{x_i}) \\ \mathsf{sum\_even_n} & = & \sum_{1 \le i < n, \ i \ \mathrm{even}} \mathtt{f}(\mathtt{x_i}) \end{array}$$

The integral is

$$\mathsf{S_n = h * (sum\_end_n + 4.d0 * sum\_odd_n + 2.d0 * sum\_even_n)/3.d0}$$

To implement the adaptive Simpson method, we need to compute $\mathsf{S_{2n}}$ efficiently, re-using the function values that were calculated for $\mathsf{S_n}$. The key is to observe that, when doubling the number of subintervals, the old node points $\mathtt{x_0, x_1, \ldots, x_n}$ are a subset of the new node points become $\bar{\mathtt{x_0}}, \bar{(}\mathtt{x_2}, \ldots, \bar{x_n}$ are a subset of the node points $\mathtt{x_0, x_1, \ldots, x_n}$.

19.4 Example.

The Gaussian integral

$$\int_0^1 e^{-x^2} dx = 0.74682413281243$$

was computed with 14 accurate digits.

The results are given below.

SINGLE PRECISION				
Tol	Trapezoidal		Simpson	
	error	n	error	n
0.1E+01	-.2E-01	2^1	0.5E-03	2^1
0.1E+00	-.2E-01	2^1	0.4E-04	2^2
0.1E-01	-.1E-02	2^3	0.4E-04	2^2
0.1E-02	-.3E-03	2^4	0.4E-04	2^2
0.1E-03	-.2E-04	2^6	0.3E-05	2^3
0.1E-04	-.1E-05	2^8	0.2E-06	2^4
0.1E-05	-.2E-06	2^9	-.8E-07	2^5
0.1E-06	-.2E-06	2^{10}	0.0E+00	2^6
0.1E-07	-.1E-04	0	0.8E-07	2^8
0.1E-08	-.1E-04	0	0.8E-07	2^8
0.1E-09	-.1E-04	0	0.8E-07	2^8
0.1E-10	-.1E-04	0	0.8E-07	2^8
0.1E-11	-.1E-04	0	0.8E-07	2^8

DOUBLE PRECISION				
TOL	Trapezoidal		Simpson	
	error	n	error	n
0.1E+01	-.2E-01	2^1	0.5E-03	2^1
0.1E+00	-.2E-01	2^1	0.4E-04	2^2
0.1E-01	-.1E-02	2^3	0.4E-04	2^2
0.1E-02	-.3E-03	2^4	0.4E-04	2^2
0.1E-03	-.2E-04	2^6	0.3E-05	2^3
0.1E-04	-.1E-05	2^8	0.2E-06	2^4
0.1E-05	-.3E-06	2^9	0.1E-07	2^5
0.1E-06	-.2E-07	2^{11}	0.7E-09	2^6
0.1E-07	-.1E-08	2^{13}	0.7E-09	2^6
0.1E-08	-.3E-09	2^{14}	0.4E-10	2^7
0.1E-09	-.2E-10	2^{16}	0.3E-11	2^8
0.1E-10	-.1E-11	2^{18}	0.2E-12	2^9
0.1E-11	-.3E-12	2^{19}	0.6E-14	2^{10}

Aside from this presentation, we compare the results above with the results obtained via *Gaussian Quadrature* integration formula. This method gives excellent results with only several node points; it is therefore much more efficient than both trapezoidal and Simpson rules.

Gauss Quadrature	
error	n
-3.0E-4	2
-1.2E-5	3
4.4E-7	4
-8.0E-9	5

19.5 Homework

Write two subroutines that compute the integral

$$\int_a^b f(x)dx$$

by Trapezoidal and Simpson methods respectively.

The user will provide the function f, the interval endpoints a and b, and the tolerance tol.

The routines will return T_n, and S_n respectively. The number of node points n should be a power of 2 ($n = 2^k$). n is automatically selected by the routine such that the numerical result approximates the true integral within tol ($|\mathrm{T_n} - \int_\mathrm{a}^\mathrm{b} \mathrm{f(x)dx}| \leq$ tol, and $|\mathrm{S_n} - \int_\mathrm{a}^\mathrm{b} \mathrm{f(x)dx}| \leq$ tol respectively). The routines will also return ierr, a state variable. ierr=0 means that the integration did not succeed; if the integration was successfull, then ierr returns the number of node points n.

The trapezoidal routine header and the declarations part should look like

```
subroutine trap(f,a,b,tol,T,ierr)
implicit none
interface
  real function f(x)
   real, intent(in) ::  x
  end function f
end interface
real, intent(in) ::  a,b,tol
real, intent(out) ::  T
integer, intent(out) ::  ierr
```

The Simpson routine header and the declarations part should look like

```
subroutine simpson(f,a,b,tol,S,ierr)
implicit none
interface
  real function f(x)
   real, intent(in) ::  x
  end function f
end interface
real, intent(in) ::  a,b,tol
real, intent(out) ::  S
integer, intent(out) ::  ierr
```

You need to insert detailed comments about the meaning of input, output and local variables, and about the computational algorithm.

Both routines will reside in the same file, named *integration.f90*.

19.6 Test Problem.

Use your routines to compute the Gaussian integral

$$\int_0^1 e^{-x^2} dx = 0.74682413281243$$

whose value (computed with 14 accurate digits) is shown above.

The results are given below.

SINGLE PRECISION				
Tol	Trapezoidal		Simpson	
	error	n	error	n
0.1E+01	-.2E-01	2^1	0.5E-03	2^1
0.1E+00	-.2E-01	2^1	0.4E-04	2^2
0.1E-01	-.1E-02	2^3	0.4E-04	2^2
0.1E-02	-.3E-03	2^4	0.4E-04	2^2
0.1E-03	-.2E-04	2^6	0.3E-05	2^3
0.1E-04	-.1E-05	2^8	0.2E-06	2^4
0.1E-05	-.2E-06	2^9	-.8E-07	2^5
0.1E-06	-.2E-06	2^{10}	0.0E+00	2^6
0.1E-07	-.1E-04	0	0.8E-07	2^8
0.1E-08	-.1E-04	0	0.8E-07	2^8
0.1E-09	-.1E-04	0	0.8E-07	2^8
0.1E-10	-.1E-04	0	0.8E-07	2^8
0.1E-11	-.1E-04	0	0.8E-07	2^8

DOUBLE PRECISION				
TOL	Trapezoidal		Simpson	
	error	n	error	n
0.1E+01	-.2E-01	2^1	0.5E-03	2^1
0.1E+00	-.2E-01	2^1	0.4E-04	2^2
0.1E-01	-.1E-02	2^3	0.4E-04	2^2
0.1E-02	-.3E-03	2^4	0.4E-04	2^2
0.1E-03	-.2E-04	2^6	0.3E-05	2^3
0.1E-04	-.1E-05	2^8	0.2E-06	2^4
0.1E-05	-.3E-06	2^9	0.1E-07	2^5
0.1E-06	-.2E-07	2^{11}	0.7E-09	2^6
0.1E-07	-.1E-08	2^{13}	0.7E-09	2^6
0.1E-08	-.3E-09	2^{14}	0.4E-10	2^7
0.1E-09	-.2E-10	2^{16}	0.3E-11	2^8
0.1E-10	-.1E-11	2^{18}	0.2E-12	2^9
0.1E-11	-.3E-12	2^{19}	0.6E-14	2^{10}

Aside from the project, compare the results above with thre results obtained via *Gaussian Quadrature* integration formula. This method gives excellent results with only several node points; it is therefore much more efficient than both trapezoidal and Simpson rules.

Gauss Quadrature	
error	n
-3.0E-4	2
-1.2E-5	3
4.4E-7	4
-8.0E-9	5

Chapter 20

Piecewise Polynomial Interpolation. Splines.

20.1 The Piecewise Interpolation Problem

Given the $m+1$ data points (t_i, y_i) for $0 \leq i \leq m$, we want to find continuous function that ''passes through'' all the data points.

One possibility is to compute a degree m polynomial that passes through each of the the $(m{+}1)$ data points. Such a polynomial exists and can be uniquely determined; fitting a single polynomial to a large number of data points can lead to insatisfactory results. We have seen however that between the interpolation points the polynomial oscillates, and these oscillations grow larger for higher order polynmials.

We want to use a low order polynomial that interpolates several points in the data set. On each subinterval we will have a *different* such polynomial; thus we use piecewise polynomial interpolation.

For our data set the interpolating function is

$$f(t) = \begin{cases} P_1(t) \ , & \texttt{for } t_0 \leq t \leq t_1 \\ \vdots & \\ P_i(t) \ , & \texttt{for } t_{i-1} \leq t \leq t_i \\ P_{i+1}(t) \ , & \texttt{for } t_i \leq t \leq t_{i+1} \\ \vdots & \\ P_m(t) \ , & \texttt{for } t_{m-1} \leq t \leq t_m \end{cases}$$

P_i are polynomial of the degree wanted. Since we want $f(t)$ to be continuous we require that

$$P_i(t_i) = P_{i+1}(t_i) \ , \quad i = 1 \cdots m-1 \ .$$

20.2 Piecewise linear interpolation

On each subinterval $[t_{i-1}, t_i]$ we have a linear polynomial

$$P_i(t) = y_{i-1} + \frac{t - t_{i-1}}{t_i - t_{i-1}} (y_i - y_{i-1}) \ , \quad i = 0 \cdots m \ .$$

20.3 Spline interpolation

On each subinterval $[t_{i-1}, t_i]$ we have a cubic polynomial $P_i(t)$. To obtain a very smooth, good-looking curve with minimal unwanted oscillations we impose that the function $f(t)$ is twice continuously differentiable. This means that we impose the continuity, countinuous first derivative and continuous second derivative conditions at all intermediate points:

$$P_i(t_i) = P_{i+1}(t_i) \ , \quad P_i'(t_i) = P_{i+1}'(t_i) \ , \quad P_i''(t_i) = P_{i+1}''(t_i) \ , \quad i = 1 \cdots m-1 \ . \tag{20.1}$$

The length of the i^{th} interval is

$$h_i = t_i - t_{i-1} \ , i = 1, \cdots, m \ .$$

The piecewise cubic polynomials have the form:

$$\begin{aligned} P_i(t) &= a_{i-1} \frac{(t_i - t)^3}{6h_i} + a_i \frac{(t - t_{i-1})^3}{6h_i} \\ &\quad + \left(y_{i-1} - \frac{a_{i-1} h_i^2}{6} \right) \frac{t_i - t}{h_i} + \left(y_i - \frac{a_i h_i^2}{6} \right) \frac{t - t_{i-1}}{h_i} \ , \quad i = 1 \cdots m \ , \quad t_{i-1} \le t \le t_i \ . \end{aligned}$$

The parameters a_i will be calculated such that the continuity conditions (20.1) are satisfied.

20.3.1 Interpolation and Continuity.

We compute the values at t_{i-1} and t_i to see that the interpolation condition is automatically satisfied:

$$\begin{aligned} P_i(t_{i-1}) &= a_{i-1} \frac{(t_i - t)^3}{6h_i} + \left(y_{i-1} - \frac{a_{i-1} h_i^2}{6} \right) \frac{t_i - t}{h_i} \\ &= a_{i-1} \frac{h_i^2}{6} + \left(y_{i-1} - \frac{a_{i-1} h_i^2}{6} \right) \\ &= y_{i-1} \\ P_i(t_i) &= a_i \frac{(t_i - t_{i-1})^3}{6h_i} + \left(y_i - \frac{a_i h_i^2}{6} \right) \frac{t - t_{i-1}}{h_i} \\ &= a_i \frac{h_i^2}{6} + \left(y_i - \frac{a_i h_i^2}{6} \right) \\ &= y_i \end{aligned}$$

The interpolating relation

$$P_i(t_i) = P_{i+1}(t_i) = y_i$$

means also that the function $f(t)$ is continuous.

20.3.2 Same slope condition.

The first derivative is

$$\begin{aligned} P_i'(t) &= -a_{i-1}\frac{(t_i-t)^2}{2h_i} + a_i\frac{(t-t_{i-1})^2}{2h_i} \\ &\quad -\left(y_{i-1}-\frac{a_{i-1}h_i^2}{6}\right)\frac{1}{h_i} + \left(y_i - \frac{a_ih_i^2}{6}\right)\frac{1}{h_i} \ , \quad i=1\cdots m \ , \quad t_{i-1}\le t\le t_i \ . \end{aligned}$$

At the end points

$$\begin{aligned} P_i'(t_{i-1}) &= -a_{i-1}\frac{(t_i-t_{i-1})^2}{2h_i} - \left(y_{i-1}-\frac{a_{i-1}h_i^2}{6}\right)\frac{1}{h_i} + \left(y_i - \frac{a_ih_i^2}{6}\right)\frac{1}{h_i} \ , \\ &= -\frac{a_{i-1}h_i}{3} + \frac{y_i-y_{i-1}}{h_i} - \frac{a_ih_i}{6} \\ P_i'(t_i) &= a_i\frac{(t_i-t_{i-1})^2}{2h_i} - \left(y_{i-1}-\frac{a_{i-1}h_i^2}{6}\right)\frac{1}{h_i} + \left(y_i - \frac{a_ih_i^2}{6}\right)\frac{1}{h_i} \\ &= \frac{a_ih_i}{3} + \frac{y_i-y_{i-1}}{h_i} + \frac{a_{i-1}h_i}{6} \quad i=1\cdots m \ , \quad t_{i-1}\le t\le t_i \ . \end{aligned}$$

where we used the relation $h_i=t_i-t_{i-1}$ several times. Changing $i\to i+1$ in the first relation above gives

$$P_{i+1}'(t_i) = -\frac{a_ih_{i+1}}{3} + \frac{y_{i+1}-y_i}{h_{i+1}} - \frac{a_{i+1}h_{i+1}}{6}$$

The ‘‘same slope’’ condition at all intermediate points

$$P_i'(t_i) = P_{i+1}'(t_i) \ , \quad \text{for intermediate points } i=1,\cdots,m-1 \ .$$

translates to

$$\frac{a_ih_i}{3} + \frac{y_i-y_{i-1}}{h_i} + \frac{a_{i-1}h_i}{6} = -\frac{a_ih_{i+1}}{3} + \frac{y_{i+1}-y_i}{h_{i+1}} - \frac{a_{i+1}h_{i+1}}{6} \ , \quad i=1,\cdots,m-1$$

or, after some manipulations

$$\frac{h_i}{6}a_{i-1} + \frac{h_i+h_{i+1}}{3}a_i + \frac{h_{i+1}}{6}a_{i+1} = \frac{y_{i+1}-y_i}{h_{i+1}} - \frac{y_i-y_{i-1}}{h_i} \ , \quad i=1,\cdots,m-1. \tag{20.2}$$

These are $m-1$ relations to determine the $m+1$ unknown coefficients $a_0,\cdots,a_m$. Clearly, we need two more relations in order to arrive at a unique solution. Depending on these extra two relations, we obtain splines of different flavours (natural, B-splines, not-a-knot).

For ‘‘natural spline’’ the extra conditions are

$$a_0=0 \ , \quad a_m=0 \ .$$

For the remaining unknown values $a_1\cdots a_{m-1}$ the relations (20.2) can be written as a linear system

$$\begin{bmatrix} \frac{h_1+h_2}{3} & \frac{h_2}{6} & 0 & \cdots & \cdots & 0 & 0 \\ \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{6} & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots \\ \vdots & 0 & \frac{h_i}{6} & \frac{h_i+h_{i+1}}{3} & \frac{h_{i+1}}{6} & 0 & \vdots \\ \vdots & \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & \frac{h_{m-2}}{6} & \frac{h_{m-2}+h_{m-1}}{3} & \frac{h_{m-1}}{6} \\ 0 & 0 & \cdots & \cdots & 0 & \frac{h_{m-1}}{6} & \frac{h_{m-1}+h_m}{3} \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_i \\ \vdots \\ a_{m-1} \end{bmatrix} = \begin{bmatrix} \frac{y_2-y_1}{h_2} - \frac{y_1-y_0}{h_1} \\ \vdots \\ \frac{y_{i+1}-y_i}{h_{i+1}} - \frac{y_i-y_{i-1}}{h_i} \\ \vdots \\ \frac{y_m-y_{m-1}}{h_m} - \frac{y_{m-1}-y_{m-2}}{h_{m-1}} \end{bmatrix} \tag{20.3}$$

The matrix of this system is *tridiagonal* and the system can be solved very efficiently.

20.3.3 Same curvature condition.

The second derivative is

$$P_i''(t) \;=\; a_{i-1}\frac{t_i - t}{h_i} + a_i\frac{t - t_{i-1}}{h_i} \quad , \quad i = 1\cdots m \;\; , \quad t_{i-1} \le t \le t_i \;\; .$$

At the end points we have

$$P_i''(t_{i-1}) = a_{i-1} \;\; , \quad P_i''(t_i) = a_i \;\; .$$

Shifting again the index in the first relation $i \to i+1$ gives

$$P_{i+1}''(t_i) = a_i \quad \Longrightarrow \quad P_{i+1}''(t_i) = P_i''(t_i) \;\; ,$$

therefore the same curvature condition is also satisfied.

20.4 Example

Consider the seven-point data set

t	y
-0.97492791218182	0.09519417591355
-0.78183148246803	0.14059547049600
-0.43388373911756	0.34691493861656
-0.00000000000000	1.00000000000000
0.43388373911756	0.34691493861656
0.78183148246803	0.14059547049600
0.97492791218182	0.09519417591355

t_k are the seven Chebyshev points

$$t_k = -\cos\left(\frac{(2k-1)\pi}{14}\right) \quad , \quad k = 1, \cdots 7 \;\; .$$

and y_k the corresponding value of the Runge function

$$r(t) = \frac{1}{1+10t^2} \;\; , \quad y_k = r\left(t_k\right) \;\; .$$

Fitting a degree-6 polynomial through the seven point data set, using piecewise linear and spline interpolants are shown in Figure 20.4.

20.5 Homework

This project should be done in DOUBLE PRECISION.

Write a subroutine spline that has:

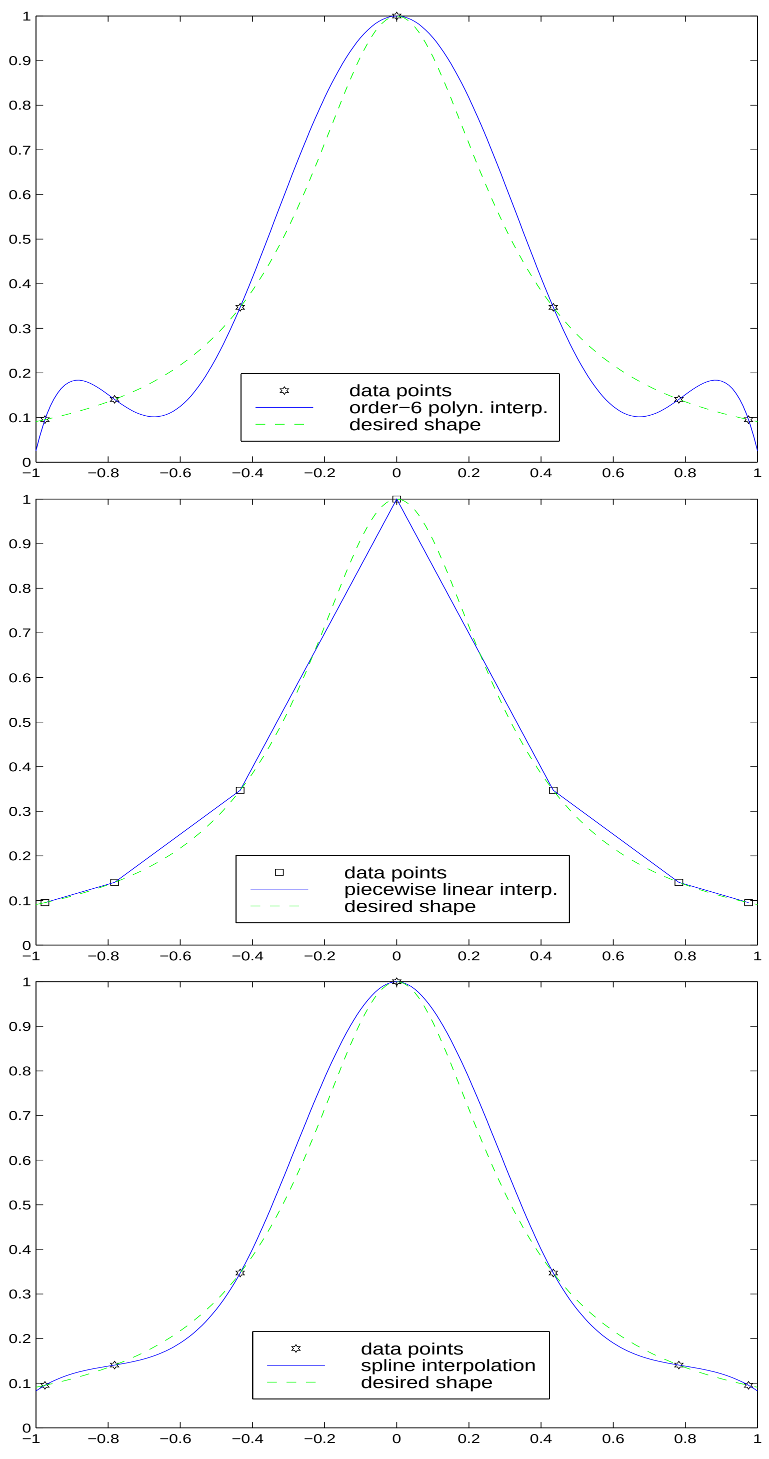


Figure 20.1: Different interpolants for the data set. Polynomial interpolant is order 6 (upper plot) and oscillates. Piecewise linear interplation (middle) is non-smooth, while spline interpolation (lower plot) is smooth and approximates the function very well.

20.5.1 Input.

- t and y, (m+1)-dimensional vectors of data.
- x, an arbitrary (k)-dimensional vector of abscissa values.

20.5.2 Output.

- v the (k)-dimensional vector of spline interpolant values, v(i) = S(x(i)).

The interface of this routine should be

```
subroutine spline(t,y,x,v)
  implicit none
  double precision, intent(in),  dimension(:) :: t,y,x
  double precision, intent(out), dimension(:) :: v
end subroutine spline
```

You will need to compute the parameters a_i first. Use natural splines. For the solution of the resulting tridiagonal system. use LAPACK's function dgtsv. Be very careful on the data vectors needed (we do not use the full matrix, only the 3 diagonals as vectors).

Go to *http://www.netlib.org/lapack/double/dgtsv.f* for more information on how to use this function.

```
      SUBROUTINE DGTSV( N, NRHS, DL, D, DU, B, LDB, INFO )
*
*     .. Scalar Arguments ..
      INTEGER            INFO, LDB, N, NRHS
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   B( LDB, * ), D( * ), DL( * ), DU( * )
*     ..
*
*  Purpose
*  =======
*
*  DGTSV  solves the equation
*
*     A*X = B,
*
*  where A is an n by n tridiagonal matrix, by Gaussian elimination with
*  partial pivoting.
*
*  Note that the equation  A'*X = B  may be solved by interchanging the
*  order of the arguments DU and DL.
*
*  Arguments
*  =========
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
```

```
*
*  NRHS    (input) INTEGER
*          The number of right hand sides, i.e., the number of columns
*          of the matrix B.  NRHS >= 0.
*
*  DL      (input/output) DOUBLE PRECISION array, dimension (N-1)
*          On entry, DL must contain the (n-1) sub-diagonal elements of
*          A.
*
*          On exit, DL is overwritten by the (n-2) elements of the
*          second super-diagonal of the upper triangular matrix U from
*          the LU factorization of A, in DL(1), ..., DL(n-2).
*
*  D       (input/output) DOUBLE PRECISION array, dimension (N)
*          On entry, D must contain the diagonal elements of A.
*
*          On exit, D is overwritten by the n diagonal elements of U.
*
*  DU      (input/output) DOUBLE PRECISION array, dimension (N-1)
*          On entry, DU must contain the (n-1) super-diagonal elements
*          of A.
*
*          On exit, DU is overwritten by the (n-1) elements of the first
*          super-diagonal of U.
*
*  B       (input/output) DOUBLE PRECISION array, dimension (LDB,NRHS)
*          On entry, the N by NRHS matrix of right hand side matrix B.
*          On exit, if INFO = 0, the N by NRHS solution matrix X.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B.  LDB >= max(1,N).
*
*  INFO    (output) INTEGER
*          = 0: successful exit
*          < 0: if INFO = -i, the i-th argument had an illegal value
*          > 0: if INFO = i, U(i,i) is exactly zero, and the solution
*               has not been computed.  The factorization has not been
*               completed unless i = N.
*
```

Recall that the compilation is done with

```
f90 spl.f90 −llapack −lblas
```

After you found a_i's you can start to compute the interpolating values v.

- For every x(ℓ) find the interval where it lies, $t_{i-1} \leq$ x(ℓ) $\leq t_i$, and compute the value of the respective polynomial v(ℓ)= P_i(x(ℓ));
- If x(ℓ)$< t_0$ then the point is outside left the interpolating interval and we simply return y_0 for the value of the interpolating polynomial at that point.
- If x(ℓ)$> t_m$ then the point is outside right the interpolating interval and we simply return y_m for the value of the interpolating polynomial at that point.

20.5.3 Test

Test your code on the data set:

```
     t      y
  --------------
   -1.0    0
   -0.5    2.0
    0      0
    0.5   -4.0
    1.0    1.0
```

Use $x = -1 : 0.01 : 1$. Find v. Save [t,y] and [x,v] in ascii files.

In MATLAB load the result files and plot the data and the obtained values:

```
plot(t,y,'o',x,v)
legend('Data','Spline')
print -deps spl.eps
```

20.5.4 Note

In the theory the data points are counted (t_0, y_0) through (t_m, y_m). Inside the program you will need to count the data points from 1 to m. This can be in principle avoided by specifying the subscript range for the vectors.

20.5.5 Submit

a single file spl.f90 that contains all the routines needed to run the spline interpolation (of course, without the LAPACK routine).

20.5.6 Test Driver

```
program test_spline
implicit none
interface
  subroutine spline(t,y,x,v)
    implicit none
    double precision, intent(in),  dimension(:) :: t,y,x
    double precision, intent(out), dimension(:) :: v
  end subroutine spline
end interface
integer, parameter :: ndata=5
integer, parameter :: nplot=201
double precision, dimension(ndata) :: t = (/-1.d0,-.5d0,0.d0,0.5d0,1.0d0/)
double precision, dimension(ndata) :: y = (/0.d0,2.d0,0.d0,-4.d0,1.d0/)
double precision, dimension(nplot) :: x=(/ (-1.d0+(i-1)*1.0D-2, i=1,nplot) /)
double precision, dimension(nplot) :: v
integer :: i
```

```
!
call spline(t,y,x,v)
!
open(10, file='Data.m',action='WRITE')
open(20, file='Results.m',action='WRITE')
!
do i=1,ndata
  write(10,FMT='(F24.16,4X,F24.16)') t(i), y(i)
end do
!
do i=1,nplot
  write(20,FMT='(F24.16,4X,F24.16)') x(i), v(i)
end do
!
close(10)
close(20)
!
end program test_spline
```

Results of this example are shown in the next Figure.

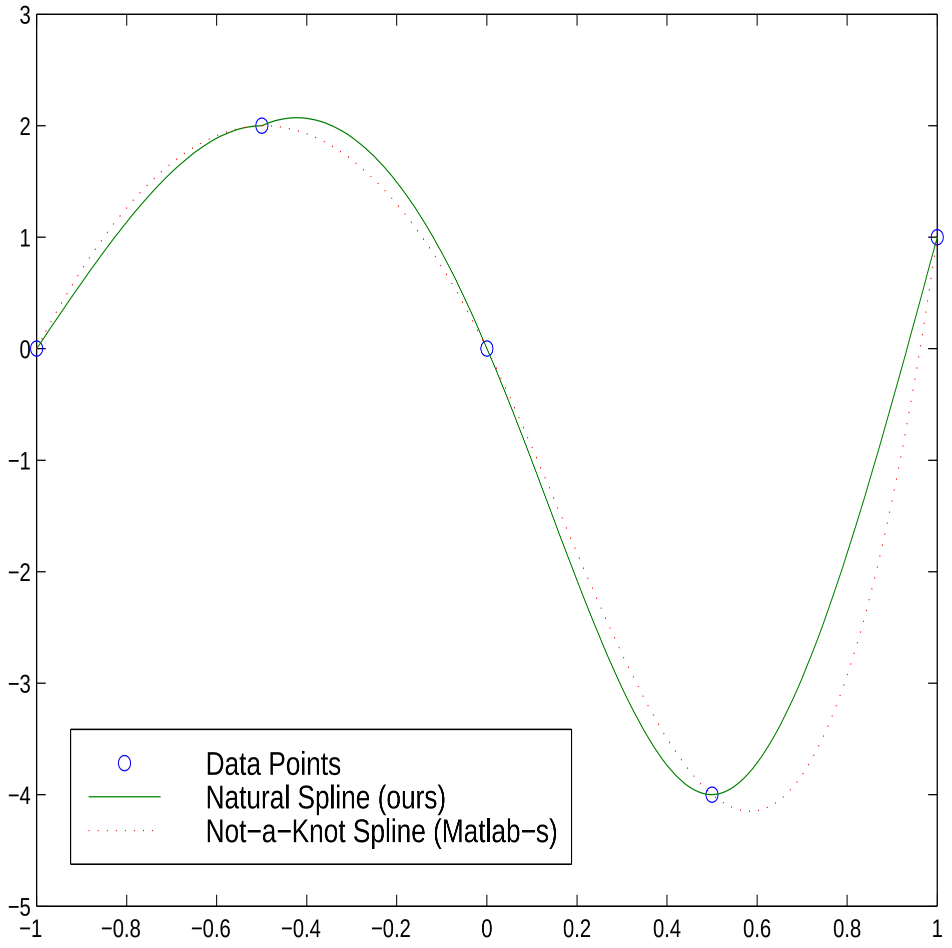


Figure 20.2: The spline results for the test driver.

Chapter 21

Taylor Polynomials

21.1 The Taylor Polynomial

Most mathematical functions (exp, sin, cos, etc) cannot be in general evaluated exactly. By hand, and on a computer as well, we usually calculate some approximations to these functions. The approximate functions need to be easy to evaluate, and have to provide values which are close enough to the values of the original function. One of the most convenient ways to compute functions is to approximate them by polynomials.

A Taylor polynomial of degree n is constructed to mimic the behavior of $f(x)$ near a point $x = a$. If p_n is a degree n Taylor polynomial,

$$p_n(x) = b_0 + b_1\,(x-a) + b_2\,(x-a)^2 + \ldots + b_n\,(x-a)^n$$

it has $n+1$ coefficients a_0 through a_n.

We have to choose the $n+1$ coefficients such that $p_n(x) \approx f(x)$ for $x \approx a$. For this, it is natural to require that the value of the polynomial (and the values of its first n derivatives) coincides with the value of the function (and the values of function's first n derivatives, respectively) for $x = a$. This means that $p_n(a) = f(a)$, and since $b_0 = p_n(a)$ we have determined b_0. This one condition is clearly not sufficient to determine all $n{+}1$ coefficients of the polynomial. An extra condition, that forces more resemblance between p_n and f when $x \approx a$ is $p_n'(a) = f'(a)$. Since $b_1 = p_n'(a)$, we have determined the second coefficient also. If $n = 1$ this is enough; if not, we continue imposing conditions on higher derivates of p_n, until we have $n+1$ conditions for the $n+1$ coefficients:

$$\begin{aligned} p_n(a) &= f(a) \\ p_n'(a) &= f'(a) \\ &\cdots \\ p_n^{(n)}(a) &= f^{(n)}(a) \end{aligned}$$

To fix the ideas, let $f(x) = e^x$, $a = 0$ and $n = 1$. Then

$$p_1(x) = b_0 + b_1\,(x-a)$$

has two unknown coefficients, a_0 and a_1. We therefore use the conditions

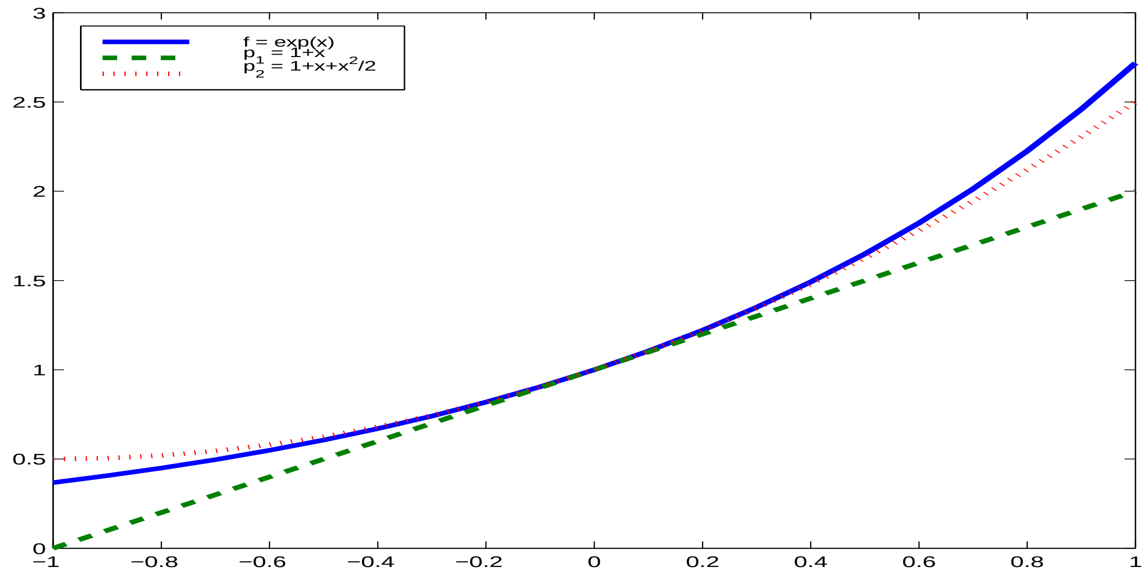
$$b_0 = p_1(a) = f(a) = e^0 = 1$$

$$b_1 = p_1'(a) = f'(a) = e^0 = 1$$

Hence, $p_1(x) = 1 + x$.

For $n = 2$, we add the third condition $p_2'''(0) = (e^x)'''(0)$, which reads $2\, b_2 = 1$, or $b_2 = 1/2$. Thus the quadratic approximation of e^x near $a = 0$ is

$$p_2(x) = 1 + x + \frac{x^2}{2} \; .$$



In the general case, if is easy to see that

$$p_n(x) = \sum_{i=0}^{n} \frac{(x-a)^j}{j!} f^{(j)}(a)$$

(to check this take the first several derivatives of p_n and evaluate them at $x = a$).

In order to be able to write this formula, we need of course to assume that f is at least n times continuously differentiable; we will actually assume $f \in \mathbf{C}^{n+1}$ for the purpose of writing an error estimate also.

For $f \in \mathbf{C}^{\infty}$ (f has derivatives of any order) we can increase n indefinitely; the approximations p_n get better and better, and, in the limit, they will coincide with f. In the limit the summation becomes infinite, and we obtain the **Taylor series** expansion

$$f(x) = \sum_{i=0}^{\infty} \frac{(x-a)^j}{j!} f^{(j)}(a)$$

Question: to define p_n we have to evaluate f, together with several of its derivatives. Aren't we better off just evaluating f, and forgetting about this whole approximation stuff? The answer is that, indeed, we have to evaluate $f, f', \ldots f^{(n)}$ at *a once*. Then p_n, once constructed, will be a cheap approximation for thousands of future calls. Moreover, f and its derivatives are evaluated at one point a, while $p_n(x)$ is a good approximation of $f(x)$ for an entire interval $x \in [a - \ell, a + \ell]$.

Homework 0.25 *Produce a general formula for the Taylor polynomial* $p_n(x)$*, when* $a = 0$ *and* $f(x)$ *is*

$$1)\ \sqrt{1+x}\ , \quad \textit{and} \quad 2)\ (1+x)^{1/3}\ .$$

Homework 0.26 *For different a's we have diferent p_n's. For example, let*

$$f(x) = 1 + 2x + 3x^2 + 4x^3$$

Thus f is a polynomial, and $p_3(x) = f(x)$ for $a = 0$. Now, the question is to find $\bar{p}_3(x)$ when $a = 1$, i.e.

$$\bar{p}_3(x) = b_0 + b_1(x-1) + b_2(x-1)^2 + b_3(x-1)^3$$

21.2 Taylor Remainder

In practice, given a function f and a desired accuracy ϵ, we would like know which degree n to choose for the Taylor polynomial approximation, such that

$$|f(x) - p_n(x)| \le \epsilon \ .$$

In other words, the error made when replacing f by p_n should not exceed ϵ. Higher degree Taylor polynomials provide better approximations, but are more and more expensive to compute. Can we find the minimal order n ($\equiv$ cheapest to compute p_n) for which the approximation error is small enough ?

The following formula from Calculus is exactly what we need to estimate the error.

Let $f(x)$ have $n+1$ continuous derivatives for $a - \ell \le x \le a + \ell$. Construct the Taylor polynomial p_n that approximates f around a. The difference between the function value and the polynomial value $R_n(x) = f(x) - p_n(x)$ is

$$R_n(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(c), \quad \alpha \le x \le \beta,$$

where c is an unknown number between a and x. $R_n(x)$ is therefore the approximation error; it is called the **Taylor remainder**.

To prove the formula, we use repeated integrations by parts.

$$\begin{aligned} f(x) &= f(a) + \int_a^x f'(t)dt \\ &= f(a) - \int_a^x (a-t)' f(t)dt \\ &= f(a) + (a-t)f'(t)|_a^x + \int_a^x \left(\frac{(t-a)^2}{2}\right) f'(t)dt \\ &= f(a) + (x-a)f'(a) + \frac{(t-a)^2}{2} f''(t)|_a^x + \dots \end{aligned}$$

The remainders can then be brought from the integral form to the standard form using the mean value theorem

$$\int_a^x \left(\frac{(t-a)^2}{2}\right) f'(t) = \frac{(x-a)^2}{2} f'(c)$$

where c is a number between a and x.

For example, let $f(1) = e^x$ and $a = 0$. We want to find the Taylor polynomial p_n which approximates $f(x)$ within 10^{-3} for $x = 1$. The problem can be formulated as follows: find n such that

$$|R_n(1)| \le 10^{-3} \ .$$

According to the Taylor remainder formula

$$R_n(1) = \frac{(1-0)^{n+1}}{(n+1)!} e^c, \text{ with } 0 \le c \le 1.$$

Since $e^c \le e < 3$ for $0 \le c \le 1$ the following inequality (error bound) holds:

$$|R_n(1)| \le \frac{3}{(n+1)!} \ .$$

It is therefore enough to find n such that

$$\frac{3}{(n+1)!} \le 10^{-3}$$

$n = 7$ satisfies this and, therefore, p_7 is the desired approximation.

When computing functions, or other mathematical objects (e.g. integrals) we are forced to make certain algorithmic approximations. In our case, we cannot evaluate the infinite Taylor series; we need to truncate it, i.e. to stop the computations after a finite number of terms. The resulting error $R_n(x)$ is a first example of *truncation error* in numerical computing. Truncation is a second source of numerical errors, after roundoff. While roundoff is due to inexact computer arithmetic, truncation errors are due to ''inexact mathematical formulas'', i.e. to the algorithmic approximations we make in order to keep the computations feasible.

"Big Oh" Notation.

If the absolute value of the function $\alpha(x)$ is bounded by a constant times h^k as $h \to 0$, we say that $\alpha = \mathcal{O}(h^k)$ (''α is *Oh* of h^k''); this means that α goes to zero as fast as, or faster than the $k^{\rm th}$ power of h, as $h \to 0$:

$$\alpha(x) = \mathcal{O}(h^k) \quad \Leftrightarrow \quad |\alpha(x)| \le C\,|h^k| \text{ as } h \to 0 \ .$$

If we want to approximate the function $f(x)$ by the Taylor polynomial $p_n(x)$ throughout the interval $x \in [a-h, a+h]$ we make the following approximation error

$$\begin{aligned} |f(x) - p_n(x)| &= \frac{|x-a|^{n+1}}{(n+1)!} f^{(n+1)}(c_x) \\ &\le \frac{h^{n+1}}{(n+1)!} \max_{a-h \le c_x \le a+h} |f^{(n+1)}(c_x)| \end{aligned}$$

In other words, the Taylor polynomial approximation error within $[a-h, a+h]$ is

$$|f(x) - p_n(x)| = \mathcal{O}(h^{n+1}) \ .$$

Homework 0.27 *How many terms do we need to consider in the Taylor polynomial approximation of $1/(1-x)$ around $a = 0$, such that for $-0.1 \le x \le 0.1$ the relative truncation error (i.e. the Taylor remainder divided by the function value) is less than 100 times the roundoff error? The computations are done in single precision, $\epsilon \approx 10^{-7}$. Hint: we want to find n such that*

$$\max_{-0.1 \le x \le 0.1} \left| \frac{R_n(x)}{1/(1-x)} \right| \le 100\epsilon \ .$$

Homework 0.28 *Write a computer code that approximates π to 10^{-5} based on the following steps.*

1. *Find the $n^{\rm th}$ Taylor polynomial with the remainder R_n for $f(x) = 1/(1+x^2)$ around $a = 0$.*
2. *Find the $n^{\rm th}$ Taylor polynomial p_n with the remainder R_n for $f(x) = arctan(x)$, using the fact that*
$$arctan(x) = \int_0^x \frac{dt}{1+t^2} \ .$$
3. *Use $\pi = 4arctan(1) \approx 4p_n(1)$.*

21.3 Important functions

In what follows $a = 0$.

$$\begin{aligned}
e^x &= \sum_{i=0}^{n} \frac{x^n}{n!} + \frac{x^{n+1}}{(n+1)!} e^c \\
\sin(x) &= \sum_{i=1}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!} + (-1)^{n+1} \frac{x^{2n+3}}{(2n+3)!} \cos(c) \\
\cos(x) &= \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!} + (-1)^{n+1} \frac{x^{2n+2}}{(2n+2)!} \cos(c) \\
\frac{1}{1-x} &= \sum_{i=0}^{n} x^i + \frac{x^{n+1}}{1-x} \\
(1+x)^\alpha &= \sum_{i=0}^{n} \left(\begin{array}{c} \alpha \\ i \end{array} \right) x^i + \left(\begin{array}{c} \alpha \\ n+1 \end{array} \right) x^{n+1} (1+c)^{\alpha-n-1} \\
&\quad \text{where} \\
\left(\begin{array}{c} \alpha \\ i \end{array} \right) &= \frac{\alpha(\alpha-1)\cdots(\alpha-i+1)}{i!} \\
&\quad \text{and} \\
i! &= i(i-1)(i-2)\cdots 1
\end{aligned}$$

21.4 Applications

Write a program to plot

$$f(x) = \frac{\log(1+x)}{x} \quad \text{for} \quad -0.5 \le x \le 0.5.$$

The trouble is that, when $x \approx 0$, $f(x) \approx 0/0$. We can overcome this problem with Taylor approximating polynomials:

$$\log(1+x) \approx \sum_{i=0}^{n} \frac{(-1)^i}{i+1} x^{i+1}$$

and therefore

$$\frac{\log(1+x)}{x} \approx \sum_{i=0}^{n} \frac{(-1)^i}{i+1} x^i \ .$$

The Taylor approximation is well defined (even for $x = 0$).

The second problem requires to accurately evaluate

$$f(x) = \frac{e^x - 1}{x}$$

for $|x| \le 10^{-4}$. We see that

$$\lim_{x\to 0} \frac{e^x - 1}{x} = 1.$$

Now, for $x = 10^{-6}$, in single precision arithmetic,

$$x = 10^{-6}, \quad e^x = 1.00000095$$

and

$$\frac{e^x - 1}{x} = 0.9536...$$

Clearly this result is corrupted by a large error. The source is the subtraction of two almost equal numbers, e^x and 1. Most of the significant digits will cancel out (in our case, the first 7 significant digits)

$$1.00000095 - 1.00000000 = 0.95 \times 10^{-6}$$

This is called a *loss of significance error* (or *cancellation* error). Now, we need to divide the small and inacurate number 0.95×10^{-6} by the small number 10^{-6}, which is in fact a multiplication by 10^6. The errors of order 10^{-7} will become now errors of order 10^{-1}.

$$\frac{0.95 \times 10^{-6}}{10^{-6}} = 0.95$$

Hence, by this division to a small number, the errors migrated from the 7th digit after the decimal point to the first digit before the decimal point (which is 0, instead of 1).

Using a Taylor polynomial of order 5,

$$\frac{e^x - 1}{x} \approx 1 + \frac{x}{2} + \frac{x^2}{6} + \frac{x^3}{24} + \frac{x^4}{120} + \frac{x^5}{840}$$

we can overcome the loss of significance error. Besides, this Taylor approximation is well defined (and equal to 1) when $x = 0$. For $x = 10^{-6}$ the Taylor approximation value is 1.00000048, correct in the first 7 digits.

21.5 Polynomial Evaluation

We discuss now the computational cost for evaluating the Taylor polynomial value at some point x

$$p_n(x) = b_0 + b_1 x + \ldots b_{n-1} x^{n-1} + b_n x^n \ .$$

When evaluating $x{*}{*}k$ the compiler (usually) translates it to $k{-}1$ multiplications. Thus, the naive algorithm

```
p = b(0)
do i=1,n
  p = p + b(i)*x**i
end do
```

will require n additions and $1+2+\ldots n$, i.e. $n(n+1)/2$ multiplications.

A better alternative is to save the computed power of x from one iteration to the next. Since $x^{i+1} = x \cdot x^i$, iteration i will need just two multiplications (not i) to compute $b_i x^i$.

```
p = b(0)
powx = 1.0
do i=1,n
  powx = powx*x     ! powx = x**i
  p = p + b(i)*powx
end do
```

This second algorithm needs n additions and $2n$ multiplications.

We can do even better than this, by rewriting the polynomial in *nested form*

$$p_n(x) = b_0 + x\left(b_1 + x\left(\ldots\left(b_{n-2} + x\left(b_{n-1} + b_n x\right)\right)\ldots\right)\right)$$

We have to start with the last term and loop back to the first. The algorithm goes as follows

```
p = b(n)
do i=n-1,0,-1
  p = b(i) + x*p
end do
```

and requires n additions and only n multiplications.

Homework 0.29 *Let $f(x) = cos(x)$ and $a = 0$. Show how to evaluate $p_n(x)$ efficiently, using nested multiplication.*

21.6 Sample Program

The following program calculates the order n of the Taylor polynomial which approximates e^x within TOL, for a given X. Note that P is the sum of the series up to $n+1$.

```
PROGRAM TAYLOR
!
! APPROXIMATES EXP(X) BY A TAYLOR
! POLYNOMIAL OF DEGREE AT MOST 100.
! THE DEGREE IS COMPUTED S.T.
! EXP(X)-P(X) <= TOL
!
IMPLICIT NONE
REAL :: X, P, R, TERM, TOL, E, ERR
INTEGER :: I
INTEGER, PARAMETER :: N_MAX = 100
!
PRINT*,"X="
READ*,  X
```

```
PRINT*,"TOLERANCE="
READ*,  TOL
!
E    = EXP(X)    ! TRUE FUNCTION VALUE
ERR  = TOL/E     ! NORMALIZED ERROR
!
TERM = 1.0
P    = 1.0
DO I=1,N_MAX
  TERM = TERM*X/REAL(I)
  P = P + TERM
  IF (TERM .LT. ERR) EXIT
END DO
!
PRINT*,"ORDER=",N-1,". TRUE ERROR=",(E-P)
!
END PROGRAM TAYLOR
```

21.7 homework

21.7.1 Taylor polynomials

Any sufficiently smooth function $f(x)$ can be approximated the n^{th} order Taylor polynomial $p_n(x)$ for $x \in [a-h, a+h]$

$$f(x) = p_n(x) + R_n(x)$$
$$p_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(a)}{i!}(x-a)^i$$

The term R_n is called the remainder, and measures the error of the Taylor approximation.

For example, if

$$f(x) = \cos(x) \ , \quad a = 0 \ , \quad h = \pi$$

then $cos(x)$ can be approximated along the interval $x \in [-\pi, \pi]$ by the following Taylor polynomial:

$$\begin{aligned} \cos(x) &= p_n(x) + R_n(x) \\ p_n(x) &= \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!} \\ R_n(x) &= (-1)^{n+1} \frac{x^{2n+2}}{(2n+2)!} \cos(c) \end{aligned}$$

The quantity c in the definition of R_n is a number between 0 and x.

When we replace $cos(x)$ by $p_n(x)$ we make an error of $R_n(x)$. An upper bound for this error is obtained as follows

$$\begin{aligned} |R_n(x)| &= \left|(-1)^{n+1} \frac{x^{2n+2}}{(2n+2)!} \cos(c)\right| \\ &= \left|\frac{x^{2n+2}}{(2n+2)!} \cos(c)\right| \\ &\leq \left|\frac{x^{2n+2}}{(2n+2)!}\right| \quad \text{(since for all } c,\ cos(c) \leq 1) \end{aligned}$$

We know that the factorial grows much faster than the exponential; in the above formula, when n increases, the denominator grows faster than the numerator. Therefore it is clear that when n (the order of the Taylor polynomial) increases, R_n decreases (that is, the higher the order the better we approximate the function).

21.7.2 The project

Write a function my_cos that approximates $cos(x)$ by a Taylor polynomial. The subroutine should work for $x \in [-\pi, \pi]$ (this is sufficient, since the interval covers one period of the cosine).

The function should take two arguments, x and tol, and should return the value of $p_n(x)$. The order of the Taylor polynomial n is determined inside the function such that

$$|R_n(x)| \leq \frac{x^{2n+2}}{(2n+2)!} \leq tol \ \ .$$

Before returning, the function should print the order of the polynomial determined.

The maximal order allowed for the Taylor polynomial is 100. If we did not succed to properly approximate $cos(x)$ with polynomials of order up to 100, then probably something is wrong (e.g. the required tol is smaller than the machine epsilon, etc). In this situation the function prints an error message and returns (the returned value is $p_{100}(x)$.

The main program should compute my_cos(x) and cos(x) and compare their values for $tol = 1.e{-}3, 1.e-6, 1.e-9$; for each tolerance use $x = k \cdot \pi/10$ with $k = 1, 2, \ldots 10$.