Computing Fourier Transforms

Michael Kupferschmid

Contents

0 Introduction

After I was invited to participate in some research projects involving radar, it took me only a little while to realize that I would need to know how to compute Fourier transforms. They play a central role in signal processing generally and in the processing of radar signals in particular. Several friends who are expert in the use of Fourier transforms reassured me that the algorithms for computing them are "easy," so I was relaxed and confident when I began my casual study of the FFT.

The first book I consulted [10] gave a clear explanation of recursive decomposition (see §5.1 of this paper), which is widely advertised as the basic idea underlying the FFT algorithm, and then immediately presented an implementation in code that I found completely baffling. I searched other books in hopes of finding a less magical or more complete explanation, but I came across only one other numerical analysis text that discusses the topic at all [6] and found it also too rabbit-out-of-a-hat for my taste.

Surprised by my inability to grasp an allegedly simple computational technique, I turned more of my attention to the project and began this paper as a way of organizing my thoughts and discoveries. One virtue of this approach is that it has let me adopt a consistent notation, eliminating one source of confusion I experienced in switching between references. A textbook[1] on digital signal processing [9] helped a great deal, but buried what I needed amid related topics and analyzed the FFT algorithm in *too much* detail for my purposes. Thinking that others might benefit from my experience in trying to master this material, I continued writing with the aim of explaining the topic in a way that is useful to me, and set the objective of understanding the algorithm well enough to be able to code my own implementation.

Having now achieved that objective, I can state categorically that I find *nothing* about this topic "easy" (except maybe typing the `fft` command into MATLAB). Recursive decomposition is *not* enough to reduce the complexity of the DFT calculation from N^2 to $N \log_2(N)$, and the code I examined first can be understood only with reference to a signal-flow graph formulation of the problem. Rearranging the input data into bit-reversed-index order is subtle and tricky, and numerous smaller details make understanding the innards of the calculation "hard," at least for me. Now I know why the topic is so seldom discussed in textbooks about numerical methods, and why it was never mentioned in any of the numerical methods courses I took!

I am hopeful that some readers really will find all of this easier than I do, and that they will tell me where I have gotten things wrong or given a complicated explanation when a simpler one would do. This is a work in progress, so corrections will be immediately useful and most welcome. I apologize to any who find my baby steps too small or the level of detail I have used too fine.

[1]Many thanks to Dr. Matt Ferrara for pointing this book out to me.

1 The Fourier Transform

These notes adopt the convention[2] that the **Fourier transform** $\mathcal{F}$ of a real or complex scalar function $f(x)$ is the complex scalar function

$$\mathcal{F}\{f(x)\} = F(\nu) = \int_{-\infty}^{+\infty} f(x)e^{i2\pi\nu x}dx$$

where, here and throughout, $i = \sqrt{-1}$. In image-processing applications $f(x)$ is often referred to as a **waveform** or **signal**, the real scalar x typically represents a distance (measured in, say, meters) and ν is spatial frequency (in cycles per meter). There are functions $f(x)$ for which this integral does not exist, but if the transform does exist then $f(x)$ can be recovered using the inverse operation,

$$\mathcal{F}^{-1}\{F(\nu)\} = f(x) = \int_{-\infty}^{+\infty} F(\nu)e^{-i2\pi\nu x}d\nu.$$

The complex value $F(\nu)$ has a **magnitude** and a **phase** that are defined like this.

$$\begin{aligned} |F(\nu)| &= \sqrt{\text{Re}\{F(\nu)\}^2 + \text{Im}\{F(\nu)\}^2} \\ \phi\{F(\nu)\} &= \arctan\left(\frac{\text{Im}\{F(\nu)\}}{\text{Re}\{F(\nu)\}}\right) \end{aligned}$$

The Fourier transform is the limit of a sum of complex sinusoidal functions of x, so $|F(\nu)|$ is the energy in each frequency component of $f(x)$ per unit of frequency, and $\phi\{F(\nu)\}$ is the phase angle of each frequency component. Some functions have Fourier transforms that are identically zero outside a finite range of frequencies, while others contain components at all frequencies.

Here are some useful properties of the Fourier transform [3, page 472ff]. The symbol $*$ denotes convolution.

$$\begin{aligned} \mathcal{F}\{f(x-\alpha)\} &= e^{i2\pi\nu\alpha}\mathcal{F}\{f(x)\} \\ \mathcal{F}\{f_1(x)+f_2(x)\} &= \mathcal{F}\{f_1(x)\}+\mathcal{F}\{f_2(x)\} \\ \mathcal{F}\{\alpha f(x)\} &= \alpha\mathcal{F}\{f(x)\} \\ \mathcal{F}\{f(\alpha x)\} &= \frac{1}{\alpha}F\left(\frac{\nu}{\alpha}\right) \\ \mathcal{F}\{f_1(x)*f_2(x)\} &= \mathcal{F}\{f_1(x)\}\mathcal{F}\{f_2(x)\} \\ f(x)\text{ even} &\Rightarrow \mathcal{F}\text{ real} \\ f(x)\text{ odd} &\Rightarrow \mathcal{F}\text{ imaginary} \end{aligned}$$

[2]These definitions of $\mathcal{F}$ and $\mathcal{F}^{-1}$ agree (after name changes) with those of [9], [10], and many other books that discuss Fourier transforms in general. For good reasons specific to the radar application in particular, [2, page xvi] uses the opposite convention that the forward transform has the negative exponent. I regret any inconvenience arising from this difference, or from my use of i rather than j to represent $\sqrt{-1}$.

The inverse transform has similar properties, leading to these additional relationships.

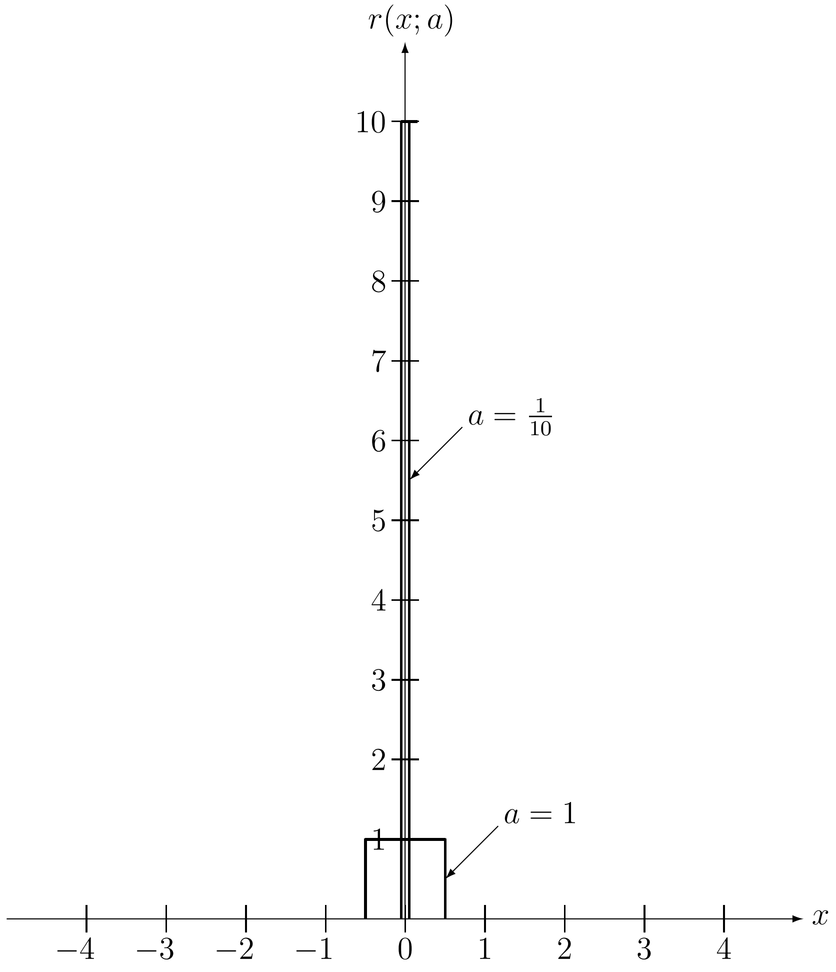
$$\begin{aligned}
\mathcal{F}\{e^{-i2\pi\nu\alpha}f(x)\} &= F(\nu-\alpha)\\
\mathcal{F}\{f_1(x)f_2(x)\} &= F_1(\nu)*F_2(\nu)\\
f(x)\text{ real} &\Rightarrow \text{Re}\{F(\nu)\}\text{ even},\ \text{Im}\{F(\nu)\}\text{ odd}
\end{aligned}$$

Here are the Fourier transforms of a few functions that are important in signal processing. The sinc function is defined as $\text{sinc}(x) = \sin(x)/x$.

function $\bullet$	Fourier transform $\mathcal{F}\{\bullet\}$
$r(x;a) = \begin{cases} \frac{1}{a} & -\frac{a}{2} \le x \le +\frac{a}{2} \\ 0 & \text{elsewhere} \end{cases}$	$\text{sinc}(\pi a\nu)$
$\text{sinc}(x)$	$r(\nu;\frac{1}{\pi})$
$\delta(x) = \lim_{a\to 0} r(x;a)$	1
1	$\delta(\nu)$
$\cos(x)$	$\frac{1}{2}\delta(\nu+\frac{1}{2\pi})+\frac{1}{2}\delta(\nu-\frac{1}{2\pi})$
$\sin(x)$	$i\frac{1}{2}\delta(\nu+\frac{1}{2\pi})-i\frac{1}{2}\delta(\nu-\frac{1}{2\pi})$

The rectangular pulse $r(x;a)$ is pictured on the next page. As a is decreased, the pulse described by $r(x;a)$ gets narrower and taller, always with an area of 1. In the table, the Dirac delta function or **unit impulse** $\delta(x)$ is defined as the limit of $r(x;a)$ as $a \to 0$, so we are to imagine continuing the process shown in the picture until $\delta(x)$ is a pulse of zero width and infinite height. That makes it not really a function at all but a "distribution" or "singularity function" [3, pages 130-135] with the following properties [8, page 176].

$$\begin{aligned}
\delta(x) &= 0 \quad \text{for} \quad x \neq 0\\
\int_{0^-}^{0^+} \delta(x)dx &= 1\\
\int_{-\infty}^{+\infty} f(x)\delta(x-b)\,dx &= f(b)
\end{aligned}$$

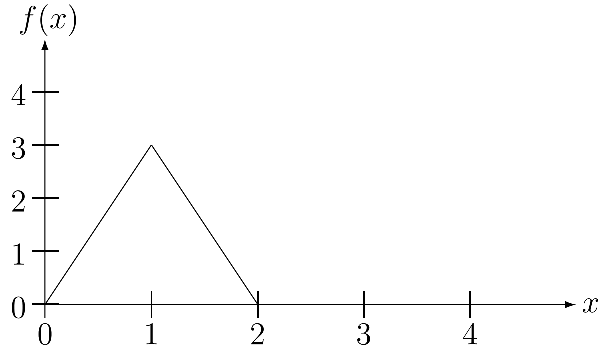


$r(x;a)$
x
$a = \frac{1}{10}$
$a = 1$
10
9
8
7
6
5
4
3
2
1
-4
-3
-2
-1
0
1
2
3
4

2 Evaluation by Calculus

When $f(x)$ is defined by a formula it might be possible to evaluate the Fourier transform integral in closed form. For example, consider the function on the left, which is graphed on the right.

$$f(x) = \begin{cases} 3x & 0 \le x \le 1 \\ 6-3x & 1 \le x \le 2 \\ 0 & \text{elsewhere} \end{cases}$$



Using the definition of the Fourier transform, we find

$$\begin{aligned}
\mathcal{F}\{f(x)\} &= \int_{-\infty}^{+\infty} f(x) e^{i2\pi\nu x} dx \\
&= \int_0^1 3x e^{i2\pi\nu x} dx + \int_1^2 (6-3x) e^{i2\pi\nu x} dx \\
&= 3\int_0^1 x e^{i2\pi\nu x} dx + 6\int_1^2 e^{i2\pi\nu x} dx - 3\int_1^2 x e^{i2\pi\nu x} dx.
\end{aligned}$$

Each of these integrals can be evaluated analytically as follows.

$$\begin{aligned}
\int_0^1 x e^{i2\pi\nu x} dx &= \frac{1}{(i2\pi\nu)^2}\left[e^{i2\pi\nu 1}(i2\pi\nu 1 - 1) - e^{i2\pi\nu 0}(i2\pi\nu 0 - 1)\right] \\
&= \frac{1}{(i2\pi\nu)^2}\left[e^{i2\pi\nu}(i2\pi\nu - 1) + 1\right] \\
\int_1^2 e^{i2\pi\nu x} dx &= \frac{1}{i2\pi\nu}\left[e^{i2\pi\nu 2} - e^{i2\pi\nu 1}\right] \\
&= \frac{1}{(i2\pi\nu)^2}\left[e^{i4\pi\nu}(i2\pi\nu) - e^{i2\pi\nu}(i2\pi\nu)\right] \\
\int_1^2 x e^{i2\pi\nu x} dx &= \frac{1}{(i2\pi\nu)^2}\left[e^{i2\pi\nu 2}(i2\pi\nu 2 - 1) - e^{i2\pi\nu 1}(i2\pi\nu 1 - 1)\right] \\
&= \frac{1}{(i2\pi\nu)^2}\left[e^{i4\pi\nu}(i4\pi\nu - 1) - e^{i2\pi\nu}(i2\pi\nu - 1)\right]
\end{aligned}$$

These results are combined at the top of the next page.

$$\begin{aligned}
\mathcal{F}\{f(x)\} &= \frac{1}{(i2\pi\nu)^2}\Big[3e^{i2\pi\nu}(i2\pi\nu-1)+3+6e^{i2\pi\nu 2}(i2\pi\nu)-6e^{i2\pi\nu}(i2\pi\nu) \\
&\qquad -3e^{i2\pi\nu 2}(i2\pi\nu 2-1)+3e^{i2\pi\nu}(i2\pi\nu-1)\Big] \\
&= \frac{1}{(i2\pi\nu)^2}\left[3+e^{i4\pi\nu}(6(i2\pi\nu)-3(i4\pi\nu-1))+e^{i2\pi\nu}(-6(i2\pi\nu)+6(i2\pi\nu-1))\right] \\
&= \frac{1}{(i2\pi\nu)^2}\left[3+3e^{i4\pi\nu}-6e^{i2\pi\nu}\right] \\
&= \frac{1}{(-4\pi^2\nu^2)}\left[3+3\left(\cos(4\pi\nu)+i\sin(4\pi\nu)\right)-6\left(\cos(2\pi\nu)+i\sin(2\pi\nu)\right)\right] \\
F(\nu) &= -\frac{3+3\cos(4\pi\nu)-6\cos(2\pi\nu)}{4\pi^2\nu^2}-i\times\frac{3\sin(4\pi\nu)-6\sin(2\pi\nu)}{4\pi^2\nu^2}
\end{aligned}$$

Both fractions are 0/0 when $\nu = 0$, but their values can be found using L'Hospital's rule as follows.

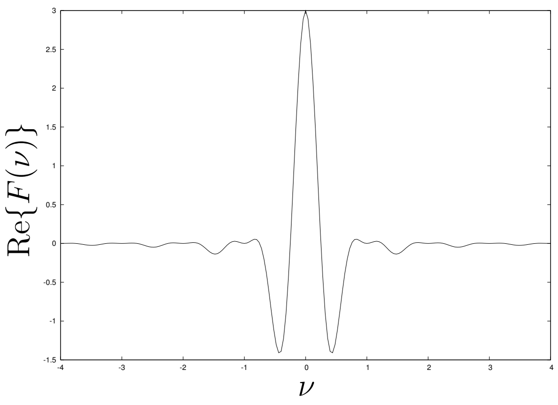
$$\begin{aligned}
\lim_{\nu\to 0}\text{Re}\{F(\nu)\} &= \lim_{\nu\to 0} -\frac{-3\sin(4\pi\nu)4\pi+6\sin(2\pi\nu)2\pi}{8\pi^2\nu} \\
&= \lim_{\nu\to 0} -\frac{-3\cos(4\pi\nu)(4\pi)^2+6\cos(2\pi\nu)(2\pi)^2}{8\pi^2} \\
&= -\frac{-3(16\pi^2)+6(4\pi^2)}{8\pi^2} = -\frac{-24}{8} = 3 \\
\lim_{\nu\to 0}\text{Im}\{F(\nu)\} &= \lim_{\nu\to 0} -\frac{3\cos(4\pi\nu)4\pi\nu-6\cos(2\pi\nu)2\pi\nu}{8\pi^2\nu} \\
&= \lim_{\nu\to 0} -\frac{-3\sin(4\pi\nu)(4\pi)^2+6\sin(2\pi\nu)(2\pi)^2}{8\pi^2} \\
&= 0
\end{aligned}$$

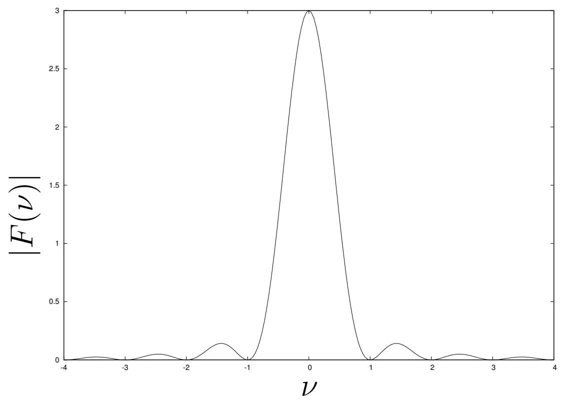
Finally, we find that the Fourier transform of the given pulse is

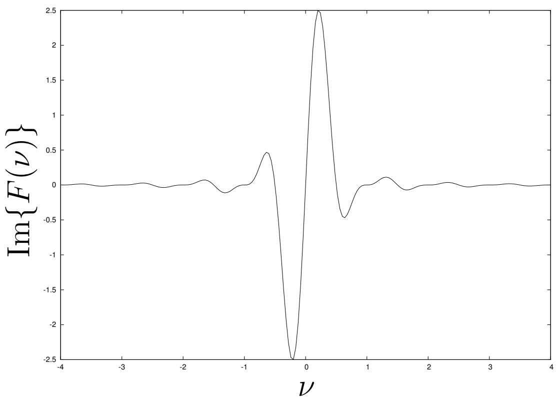
$$F(\nu) = \begin{cases} 3 \;+\; i\times 0 & \nu = 0 \\ -\dfrac{3+3\cos(4\pi\nu)-6\cos(2\pi\nu)}{4\pi^2\nu^2} \;-\; i\times\dfrac{3\sin(4\pi\nu)-6\sin(2\pi\nu)}{4\pi^2\nu^2} & \text{otherwise.} \end{cases}$$

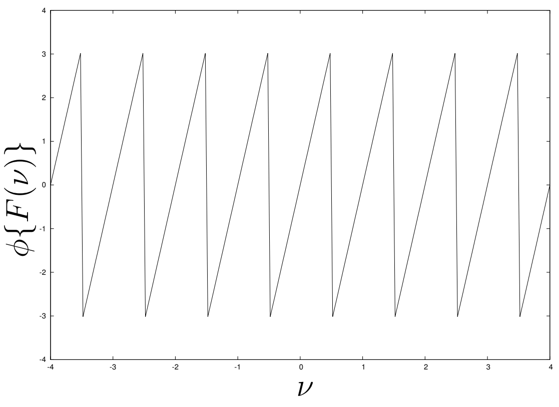
The following page shows graphs of the real and imaginary parts of $F(\nu)$, and of its magnitude and phase, as functions of ν.

I will use the pulse function that we have just transformed analytically to test the other methods discussed in this paper for computing the Fourier transform, all of which do so numerically.









3 Evaluation by Numerical Quadrature

If $f(x)e^{i2\pi\nu x}$ has no antiderivative, we might consider approximating the definite integral

$$F(\nu) = \int_{-\infty}^{+\infty} f(x)e^{i2\pi\nu x}dx$$

at each value of ν by some standard algorithm for numerical quadrature, such as Simpson's Rule [1, page 200]:

$$\int_a^b g(x)dx \approx \frac{h}{3}[g(a) + 4g(a+h) + 2g(a+2h) + 4g(a+3h) + \cdots + 4g(b-h) + g(b)].$$

The next page lists a Fortran subprogram that uses Simpson's Rule to approximate the integral of a complex function $g(x)$. Of course we can also use it on functions $g(x)$ that *do* have closed-form integrals, and compare its results to those obtained analytically.

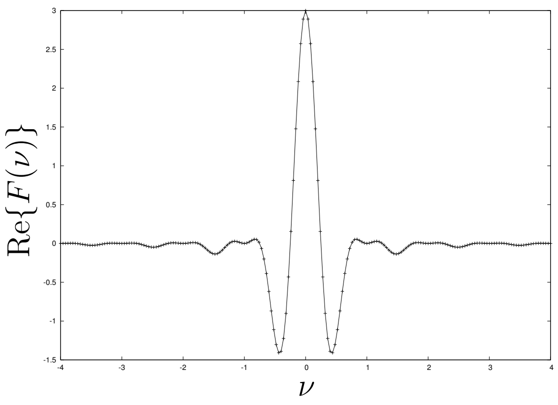
```
C
      FUNCTION ZIMPSN(G,A,B)
C     This routine approximates the definite integral
C
C                    _B
C                   /
C     ZIMPSN =     /  g(x) dx
C               A_/
C
C     using Simpson's 1/3 rule.
C
C     variable  meaning
C     --------  -------
C     A         lower limit of integration
C     B         upper limit of integration
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     G         function subprogram returns integrand value
C     H         step width
C     L         index on terms
C     MOD       Fortran function for remainder of INTEGER*4 division
C     N         number of steps
C     SUM       sum of coordinates in Simpson's formula
C     X         variable of integration at a point in the sum
C     Y         value of integrand function at X
C
C     formal parameters
      COMPLEX*16 ZIMPSN
      EXTERNAL G
      REAL*8 A,B
C
C     local variables
      REAL*8 H,X
      COMPLEX*16 Y,G,SUM
      INTEGER*4 N/100/
C
C ------------------------------------------------------------------
C
C     include the terms on the ends
      SUM=G(A)+G(B)
C
C     find the step length
      H=(B-A)/DFLOAT(N)
C
C     include the terms in the middle
      DO 2 L=1,N-1
           X=A+H*DFLOAT(L)
           Y=G(X)
           IF(MOD(L,2).EQ.0) THEN
C             L is even
              SUM=SUM+(2.D0,0.D0)*Y
           ELSE
C             L is odd
              SUM=SUM+(4.D0,0.D0)*Y
           ENDIF
    2 CONTINUE
      ZIMPSN=SUM*DCMPLX(H/3.D0,0.D0)
      RETURN
      END
```

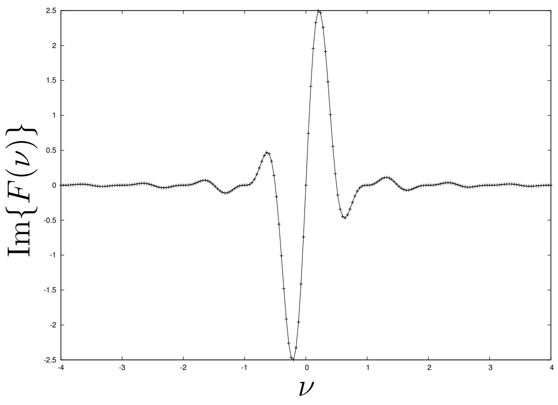
Here is some code that invokes the integrator to approximate the Fourier transform of the pulse function we considered in §2.

```
      COMPLEX*16 F,ZIMPSN
      EXTERNAL G
      COMMON /DATA/ NU
      REAL*8 NU
C     OPEN(UNIT=1,FILE='ft.real')
C     OPEN(UNIT=2,FILE='ft.imag')
      DO 1 L=1,201
           NU=0.04D0*DFLOAT(L-101)
           F=ZIMPSN(G,0.D0,2.D0)
           WRITE(1,901) NU,DREAL(F)
           WRITE(2,901) NU,DIMAG(F)
  901      FORMAT(2(1X,1PE13.6))
    1 CONTINUE
      STOP
      END
C
      FUNCTION G(X)
      COMPLEX*16 G,I/(0.D0,1.D0)/
      REAL*8 X
      REAL*8 TWOPI/6.2831853071795865D0/
      COMMON /DATA/ NU
      REAL*8 NU
      IF(X.LT.0.D0) THEN
         G=(0.D0,0.D0)
         RETURN
      ENDIF
      IF(X.GE.0.D0 .AND. X.LT.1.D0) THEN
         G=DCMPLX(3.D0*X,0.D0)*CDEXP(I*DCMPLX(TWOPI*NU*X))
         RETURN
      ENDIF
      IF(X.GE.1.D0 .AND. X.LT.2.D0) THEN
         G=DCMPLX(6.D0-3.D0*X,0.D0)*CDEXP(I*DCMPLX(TWOPI*NU*X))
         RETURN
      ENDIF
      IF(X.GE.2.D0) THEN
         G=(0.D0,0.D0)
         RETURN
      ENDIF
      END
```

The function `ZIMPSN` invokes the function `G` to compute the integrand at different values of `X`. Because $g(x) = f(x)e^{i2\pi\nu x}$ also depends on ν, the value of `NU` at which the transform is being evaluated is passed from the main program to `G` through common block `/DATA/`.

When the main and subprograms are compiled together and run, the output is two files containing the numerical integrator's approximations to $\text{Re}\{F(\nu)\}$) and $\text{Im}\{F(\nu)\}$) at the 201 values of ν produced by the main program. When these points are plotted on top of the true curves for $\text{Re}\{F(\nu)\}$ and $\text{Im}\{F(\nu)\}$ displayed in §1, we get the graphs at the top of the next page.



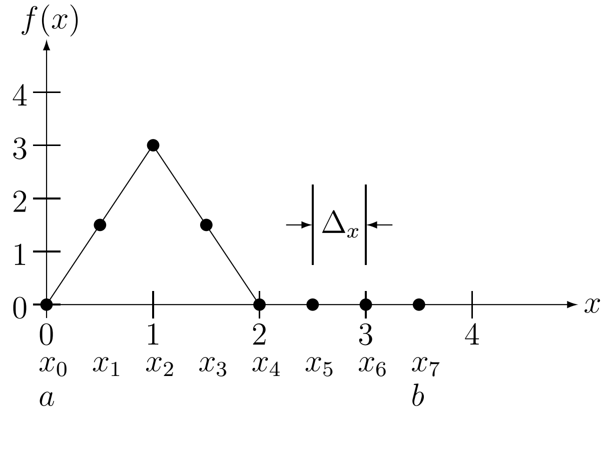


The numerical approximations can be seen to agree quite well with the analytic results we found earlier. When $f(x)$ can be described by formulas (here coded into the function `G`) a numerical integrator such as `ZIMPSN` can obtain values of the Fourier transform integrand at arbitrary points x, and that makes it possible to evaluate $\mathcal{F}\{f(x)\}$ accurately.

Unfortunately, in image-processing applications $f(x)$ is seldom described by formulas, so its value might not be available at every x where an integrator like `ZIMPSN` would like to know it. Also, it is sometimes necessary to compute the transform in nearly real time, and that might rule out extravagant floating-point calculations like those we have used here.

4 Approximation by Discrete Fourier Transform

Often $f(x)$ is known only at uniformly-spaced discrete values of x. As mentioned in §1, $\mathcal{F}\{f(x-\alpha)\} = e^{i2\pi\nu\alpha}\mathcal{F}\{f(x)\}$, so we can assume the first discrete value of x to be zero and afterwards multiply the transform by $e^{-i\nu\alpha}$ if necessary. Then $f(x)$ can be represented by the N consecutive sampled function values $f_k = f(x_k)$, where $x_k = k\Delta_x$ and $k = 0, \ldots, N-1$. The number Δ_x has the same units as x and is called the **sampling interval**; its reciprocal is the **sampling rate**. If x and Δ_x are measured in, say, meters, then the sampling rate $1/\Delta_x$ is a spatial frequency measured in samples per meter. The graph below shows the pulse of §2 sampled at $N = 8$ points, with $\Delta_x = (b-a)/(N-1) = 3\frac{1}{2}/7 = \frac{1}{2}$.



In radar applications the function to be transformed is typically assumed to be zero outside some fixed domain, such as $x \in [a,b]$ in the graph above,[3] so increasing the number of samples makes them closer together rather than widening the domain. If we double the number of samples we make $\Delta_x = 3\frac{1}{2}/15 = \frac{7}{30}$ and increase the sampling rate from 2 to $\frac{30}{7}$.

4.1 The Forward DFT

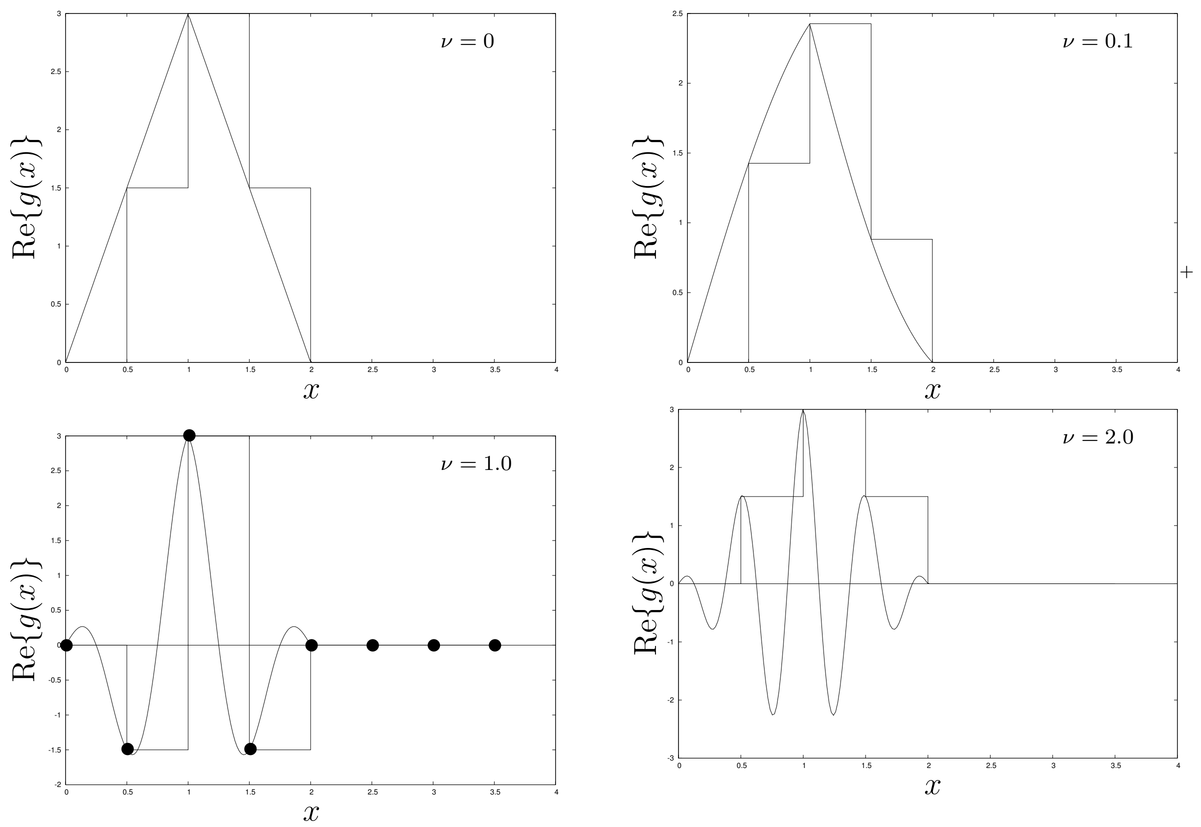
The Fourier transform integral is the limit of a Riemann sum, so we can write $F(\nu)$ as

$$F(\nu) = \int_{x=0}^{x=b} f(x)e^{i2\pi\nu x}dx = \lim_{\substack{N\to\infty \\ \Delta_x\to 0}} \sum_{k=0}^{N-1} f_k e^{i2\pi\nu x_k}\Delta_x \quad \text{where} \quad \Delta_x = \frac{b}{N-1}$$

and use the samples f_k at the points x_k in a finite sum to approximate the integral like this.

$$F(\nu) \approx \sum_{k=0}^{N-1} f_k e^{i2\pi\nu x_k}\Delta_x = \sum_{k=0}^{N-1} g(x_k)\Delta_x \quad \text{where} \quad g(x) = f(x)e^{i2\pi\nu x}$$

The real part of the integrand function $g(x)$ is plotted below for four values of ν, along with the rectangles of the finite-sum approximation for $N = 8$.



[3]In the signal-processing literature one often finds jargon such as "$f(x)$ is supported in $[a,b]$" or "$f(x)$ has support on $[a,b]$" which just means "$f(x)$ is zero outside the interval $[a,b]$."

When $\nu = 0$, $g(x) = f(x)e^{i2\pi(0)x} = f(x)$. The rectangle-rule approximation happens to equal the exact area under the pulse, $\frac{1}{2} \times 2 \times 3 = 3$, and the formula yields

$$F(0) \approx \tfrac{1}{2}\sum_{k=0}^{7} f_k = \tfrac{1}{2}(0 + 1\tfrac{1}{2} + 3 + 1\tfrac{1}{2} + 0 + 0 + 0 + 0) = 3.$$

When $\nu = 0.1$ symmetry suggests that the approximation is also exact, and adding up the rectangle areas gives about $1.4 \times 0.5 + 2.4 \times 0.5 + 0.8 \times 0.5 = 2.3$. Using the formula confirms that this rough graphical estimate is plausible.

$$\begin{aligned}
e^{i2\pi(0.1)(0.5)} &= \cos(0.1\pi) + i\sin(0.1\pi) \approx 0.95 + 0.31i \\
e^{i2\pi(0.1)(1.0)} &= \cos(0.2\pi) + i\sin(0.2\pi) \approx 0.81 + 0.59i \\
e^{i2\pi(0.1)(1.5)} &= \cos(0.3\pi) + i\sin(0.3\pi) \approx 0.59 + 0.81i \\
F(0.1) &\approx \tfrac{1}{2}\left(1.5e^{i2\pi(.05)} + 3.0e^{i2\pi(0.10)} + 1.5e^{i2\pi(0.15)}\right) \approx 2.37 + 1.72i
\end{aligned}$$

When $\nu = 1.0$, it is not so easy to decide whether the area under the rectangles is equal to the area under the curve, but if they differ it is not by much. (The sample dots are referred to in §4.5.) When $\nu = 2.0$, however, the rectangle-rule approximation clearly overestimates the net area under the curve, which is by inspection about zero.

In the formula

$$F(\nu) \approx \Delta_x \sum_{k=0}^{N-1} f_k e^{i2\pi\nu x_k}$$

it is the sum that is referred to as the **discrete Fourier transform** or DFT, so it is the DFT multiplied by Δ_x that we have been using to approximate the continuous Fourier transform $F(\nu)$. The sum can be calculated for any value of ν, but it is convenient in defining the inverse DFT to evaluate the **forward DFT**

$$F_n = \sum_{k=0}^{N-1} f_k e^{i2\pi\nu_n x_k}$$

at uniformly-spaced frequencies $\nu_n = n/(N\Delta_x)$, where $n = 0 \ldots N-1$. Sometimes that whole set of F_n is referred to loosely as the DFT of the set of samples f_k. The sequence $f_0 \ldots f_8 = [0, 1.5, 3, 1.5, 0, 0, 0, 0]$ that we considered above is said to have the DFT $F_0 \ldots F_8 = [6 + 0i,\ 0 + 5.12i,\ -3 + 0i,\ 0 - .879i,\ 0 + 0i,\ 0 + .879i,\ -3 + 0i,\ 0 - 5.12i]$.

Using

$$\nu_n x_k = \frac{n}{N\Delta_x} \times k\Delta_x = \frac{nk}{N}$$

we can rewrite the definition of the DFT as

$$F_n = \sum_{k=0}^{N-1} f_k e^{i2\pi nk/N} \qquad n = 0 \ldots N-1.$$

In §2 and §3 we plotted $F(\nu)$ for negative as well as positive values of ν, but this set of F_n includes values only at zero and positive frequencies. Notice, however, that

$$\begin{aligned} e^{i2\pi k(N-n)/N} &= e^{i2\pi k} \times e^{-i2\pi kn/N} \\ &= [\cos(2\pi k) + i \sin(2\pi k)] \times e^{-i2\pi kn/N} \\ &= e^{i2\pi k(-n)/N} \end{aligned}$$

Thus F_n is periodic in n with period N and $F_{N-n} = F_{-n}$ so we can get values of the transform at negative frequencies from those we find at positive ones. For example, if $N = 8$ then $F_{-4} = F_{+4}$ and $F_{-3} = F_{+5}$ and $F_{-2} = F_{+6}$ and $F_{-1} = F_{+7}$. For the moment we will view the results of the DFT calculation as being in order for the ν values in the top sequence listed below.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	$\frac{+1}{8\Delta_x}$	$\frac{+2}{8\Delta_x}$	$\frac{+3}{8\Delta_x}$	$\frac{+4}{8\Delta_x}$	$\frac{+5}{8\Delta_x}$	$\frac{+6}{8\Delta_x}$	$\frac{+7}{8\Delta_x}$
0	$\frac{+1}{8\Delta_x}$	$\frac{+2}{8\Delta_x}$	$\frac{+3}{8\Delta_x}$	$\frac{\pm 4}{8\Delta_x}$	$\frac{-3}{8\Delta_x}$	$\frac{-2}{8\Delta_x}$	$\frac{-1}{8\Delta_x}$
F_0	F_1	F_2	F_3	$F_{\pm 4}$	F_{-3}	F_{-2}	F_{-1}

However, in some settings it is useful to think of some of the F_n as corresponding to negative frequencies, and then the results are in the order of the bottom sequence.

4.2 The Inverse DFT

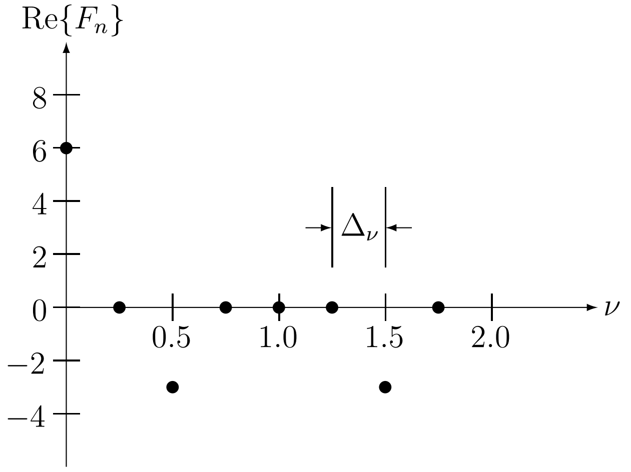
The real part of the unshifted 8-point DFT of our pulse waveform is graphed at the top of the next page. We can use the points (ν_n, F_n) in a Riemann sum to approximate the inverse Fourier transform integral, like this.

$$f(x) = \int_{\nu=-\infty}^{\nu=+\infty} F(\nu) e^{-i2\pi\nu x} d\nu \approx \sum_{n=0}^{N-1} (\Delta_x F_n) e^{-i2\pi\nu_n x} \Delta_\nu = \Delta_x \Delta_\nu \sum_{n=0}^{N-1} F_n e^{-i2\pi\nu_n x}$$

where Δ_ν is the spacing in frequency between the points of the transform.

But $\Delta_\nu = 1/(N\Delta_x)$, so $\quad f_k = \dfrac{1}{N} \displaystyle\sum_{n=0}^{N-1} F_n e^{-i2\pi\nu_n x_k} \qquad k = 0 \ldots N-1$

or, again using the fact that $\nu_n x_k = nk/N$, $\quad f_k = \dfrac{1}{N} \displaystyle\sum_{n=0}^{N-1} F_n e^{-i2\pi nk/N} \qquad k = 0 \ldots N-1.$



This is the **inverse DFT**. Unlike the direct DFT, the inverse recovers the given sampled data *without further scaling* and *exactly.* For example,

$$\begin{aligned}
f_0 &= \tfrac{1}{8}\sum_{n=0}^{7} F_n \\
&= \tfrac{1}{8}[(6+0-3+0+0+0-3+0)+i(0+5.12+0-.879+0+.879+0-5.12)] \\
&= 0 \\
f_1 &= \tfrac{1}{8}\sum_{n=0}^{7} F_n e^{-i2\pi n/8} \\
&= \tfrac{1}{8}[F_0+F_1e^{-i\pi/4}+F_2e^{-i\pi/2}+F_3e^{-i\pi 3/4}+F_4e^{-i\pi}+F_5e^{-i\pi 5/4}+F_6e^{-i\pi 3/2}+F_7e^{-i\pi 7/4}] \\
&= \tfrac{1}{8}[6+5.12i(.707-.707i)+(-3)(-i)+(-.879i)(-.707-.707i)+(.879i)(-.707+.707i) \\
&\qquad +(-3)(i)+(-5.12i)(.707+.707i)] \\
&= \tfrac{1}{8}[12+0i]=1.5
\end{aligned}$$

It is just because of algebra that the inverse transform restores the original sequence exactly, but the effect is that the discretization error introduced by the Riemann-sum approximation in the forward transform is undone. Of course, if the samples were taken far enough apart to overlook interesting features of $f(x)$, those features will be absent from even a perfect reconstruction of the original samples. For example, if the dots in the picture of our triangular pulse had actually been connected by curves rather than by straight line segments, we could not tell that from the recovered sample points. Picking the points introduces a **sampling error** that depends on $f(x)$.

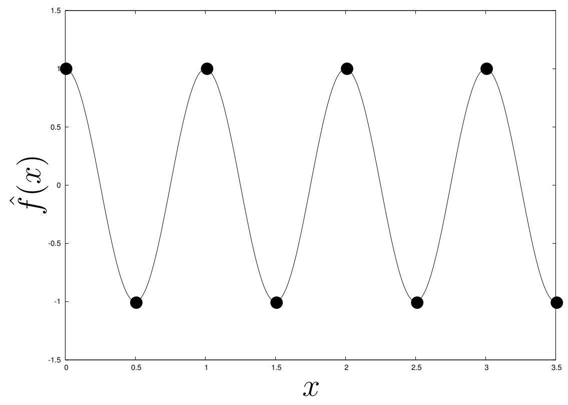
If we let the complex number $W = e^{i2\pi/N}$, the formulas above can be rewritten as

$$F_n = \sum_{k=0}^{N-1} W^{nk} f_k \qquad f_k = \frac{1}{N}\sum_{n=0}^{N-1} W^{-nk} F_n.$$

After the powers of W are computed once, it takes on the order of N^2 complex multiplications and additions to compute either the direct or inverse transform for each set of data f_k or F_n.

4.3 Trigonometric Interpolation

Another way to think about the DFT is that it prescribes a trigonometric function interpolating the given sequence of data values [5, §12.1] [6, §6.12]. For example, the sequence of $N = 8$ values $f_0 \dots f_7 = [1, -1, 1, -1, 1, -1, 1, -1]$ is plotted as points with spacing $\Delta_x = \frac{1}{2}$ in the graph below, and the points are interpolated by the curve that is drawn through them.



We can describe the sinusoidal interpolant as the real part of a finite sum of complex exponentials having certain fixed frequencies ν_n, like this.

$$\hat{f}(x) = \mathrm{Re}\{s(x)\} \quad \text{where} \quad s(x) = \sum_{n=0}^{7} c_n e^{-i2\pi\nu_n x}$$

To make the interpolant $\hat{f}(x)$ pass thru the given data points, we need only solve the algebraic equations $s(x_k) = f_k, k = 0\dots7$ for the unknown weights c_n. Using the fact that $\nu_n x_k = nk/N$ and the shorthand notation $W = e^{i2\pi/N}$ we can write

$$s(x_k) = \sum_{n=0}^{7} c_n e^{-i2\pi nk/N} = \sum_{n=0}^{7} c_n W^{-nk}$$

and then the equations we must solve are these.

$$\begin{aligned}
c_0 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 &= 1 \\
c_0 + c_1W^{-1} + c_2W^{-2} + c_3W^{-3} + c_4W^{-4} + c_5W^{-5} + c_6W^{-6} + c_7W^{-7} &= -1 \\
c_0 + c_1W^{-2} + c_2W^{-4} + c_3W^{-6} + c_4W^{-8} + c_5W^{-10} + c_6W^{-12} + c_7W^{-14} &= 1 \\
c_0 + c_1W^{-3} + c_2W^{-6} + c_3W^{-9} + c_4W^{-12} + c_5W^{-15} + c_6W^{-18} + c_7W^{-21} &= -1 \\
c_0 + c_1W^{-4} + c_2W^{-8} + c_3W^{-12} + c_4W^{-16} + c_5W^{-20} + c_6W^{-24} + c_7W^{-28} &= 1 \\
c_0 + c_1W^{-5} + c_2W^{-10} + c_3W^{-15} + c_4W^{-20} + c_5W^{-25} + c_6W^{-30} + c_7W^{-35} &= -1 \\
c_0 + c_1W^{-6} + c_2W^{-12} + c_3W^{-18} + c_4W^{-24} + c_5W^{-30} + c_6W^{-36} + c_7W^{-42} &= 1 \\
c_0 + c_1W^{-7} + c_2W^{-14} + c_3W^{-21} + c_4W^{-28} + c_5W^{-35} + c_6W^{-42} + c_7W^{-49} &= -1
\end{aligned}$$

In matrix form this linear system is

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^{-1} & W^{-2} & W^{-3} & W^{-4} & W^{-5} & W^{-6} & W^{-7} \\ 1 & W^{-2} & W^{-4} & W^{-6} & W^{-8} & W^{-10} & W^{-12} & W^{-14} \\ 1 & W^{-3} & W^{-6} & W^{-9} & W^{-12} & W^{-15} & W^{-18} & W^{-21} \\ 1 & W^{-4} & W^{-8} & W^{-12} & W^{-16} & W^{-20} & W^{-24} & W^{-28} \\ 1 & W^{-5} & W^{-10} & W^{-15} & W^{-20} & W^{-25} & W^{-30} & W^{-35} \\ 1 & W^{-6} & W^{-12} & W^{-18} & W^{-24} & W^{-30} & W^{-36} & W^{-42} \\ 1 & W^{-7} & W^{-14} & W^{-21} & W^{-28} & W^{-35} & W^{-42} & W^{-49} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

which looks daunting. Fortunately the properties of W allow the inverse of the coefficient matrix to be written in the following simple form.

$$\frac{1}{8}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^{+1} & W^{+2} & W^{+3} & W^{+4} & W^{+5} & W^{+6} & W^{+7} \\ 1 & W^{+2} & W^{+4} & W^{+6} & W^{+8} & W^{+10} & W^{+12} & W^{+14} \\ 1 & W^{+3} & W^{+6} & W^{+9} & W^{+12} & W^{+15} & W^{+18} & W^{+21} \\ 1 & W^{+4} & W^{+8} & W^{+12} & W^{+16} & W^{+20} & W^{+24} & W^{+28} \\ 1 & W^{+5} & W^{+10} & W^{+15} & W^{+20} & W^{+25} & W^{+30} & W^{+35} \\ 1 & W^{+6} & W^{+12} & W^{+18} & W^{+24} & W^{+30} & W^{+36} & W^{+42} \\ 1 & W^{+7} & W^{+14} & W^{+21} & W^{+28} & W^{+35} & W^{+42} & W^{+49} \end{bmatrix}$$

It is easy to show that the dot product of any column of the first matrix with the same row of the second yields 1, and the dot product of any column of first matrix with a *different* row of the second yields 0. For example, multiplying the second row of the coefficient matrix by the third column of its inverse should yield zero as the (2,3) element of the identity.

$$\begin{aligned} I_{2,3} &= \tfrac{1}{8}(1 + W + W^2 + W^3 + W^4 + W^5 + W^6 + W^7) \\ &= \tfrac{1}{8}(1 + e^{i2\pi/8} + e^{i2\pi 2/8} + e^{i2\pi 3/8} + e^{i2\pi 4/8} + e^{i2\pi 5/8} + e^{i2\pi 6/8} + e^{i2\pi 7/8}) \end{aligned}$$

But $e^{i2\pi/8} = -e^{i2\pi 5/8}$ and $e^{i2\pi 2/8} = -e^{i2\pi 6/8}$ and $e^{i2\pi 3/8} = -e^{i2\pi 7/8}$, so

$$I_{2,3} = \tfrac{1}{8}(1 + e^{i2\pi 4/8}) = \tfrac{1}{8}(1 + e^{i\pi}) = 0$$

as expected. With the inverse of the coefficient matrix in hand, the weights c_k can be obtained by a matrix multiplication.

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \frac{1}{8}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^{+1} & W^{+2} & W^{+3} & W^{+4} & W^{+5} & W^{+6} & W^{+7} \\ 1 & W^{+2} & W^{+4} & W^{+6} & W^{+8} & W^{+10} & W^{+12} & W^{+14} \\ 1 & W^{+3} & W^{+6} & W^{+9} & W^{+12} & W^{+15} & W^{+18} & W^{+21} \\ 1 & W^{+4} & W^{+8} & W^{+12} & W^{+16} & W^{+20} & W^{+24} & W^{+28} \\ 1 & W^{+5} & W^{+10} & W^{+15} & W^{+20} & W^{+25} & W^{+30} & W^{+35} \\ 1 & W^{+6} & W^{+12} & W^{+18} & W^{+24} & W^{+30} & W^{+36} & W^{+42} \\ 1 & W^{+7} & W^{+14} & W^{+21} & W^{+28} & W^{+35} & W^{+42} & W^{+49} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In other words,

$$c_n = \frac{1}{N}\sum_{k=0}^{N-1} W^{nk} f_k$$

But the sum is just the definition of the forward DFT! What we have found is

$$\tfrac{1}{8}\text{DFT}\{[1,-1,1,-1,1,-1,1,-1]\} = \tfrac{1}{8}[0,0,0,0,8,0,0,0]$$

and the weights c_n in our definition of the sinusoidal interpolant are just the values F_n of the forward transform scaled by the factor $1/N$. The F_n found by the forward DFT are thus scaled amplitudes of the corresponding frequency components $\nu_n = n/(N\Delta_x)$ in the signal whose samples are the f_k. In this example the only frequency component that is nonzero is $\nu_4 = 1$, so $s(x) = e^{-i2\pi(1)x} = \cos(2\pi x) - i\sin(2\pi x)$ and the sinusoid that interpolates the f_k is $\hat{f}(x) = \cos(2\pi x)$. This is the curve drawn through the points in the graph above.

4.4 Errors in the Approximation

In §4.1 (on page 15) we examined some rectangles used in the DFT to approximate the area under the Fourier transform integrand function $g(x)$ for our pulse waveform. The Riemann sum turned out to be exact when $\nu = 0$, but with only $N = 8$ samples that result might be dumb luck rather than a property of the method. When $\nu = 1.0$ the approximation was plausible, but when $\nu = 2.0$ it was useless. How good is the DFT approximation at the frequencies of the F_n if we use values of N larger than 8?

The program on the next page uses the definition of F_n to evaluate the DFT for a user-chosen number of points. In reading the program it will be helpful to know that `NN` represents N in the analysis, `N` represents n, `FF` represents F_n, and `F` represents f_k. Subprogram `FOFX`, listed below, returns values of $f(x)$ for the triangular pulse of §2.

```
C
      FUNCTION FOFX(XK)
C     This routine returns the value of the Section 2 pulse.
C
      REAL*8 FOFX,XK
C
C -----------------------------------------------------------------
C
      IF(XK.LE.0.D0) THEN
         FOFX=0.D0
      ENDIF
      IF(XK.GT.0.D0 .AND. XK.LE.1.D0) THEN
         FOFX=3.D0*XK
      ENDIF
      IF(XK.GT.1.D0 .AND. XK.LE.2.D0) THEN
         FOFX=6.D0-3.D0*XK
      ENDIF
      IF(XK.GT.2.D0) THEN
         FOFX=0.D0
      ENDIF
      RETURN
      END
```

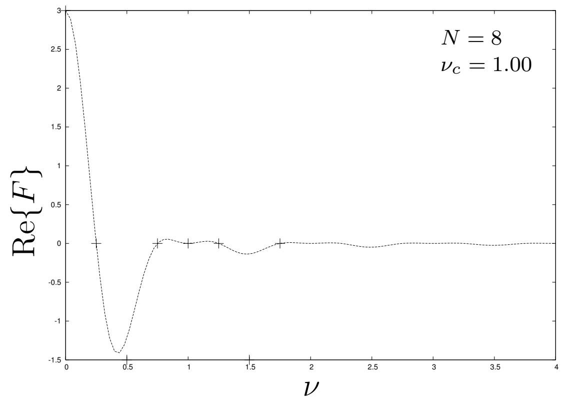
```
C
C     This program computes the discrete Fourier transform of
C     the pulse waveform using a given number of samples.
C
C     variable  meaning
C     --------  -------
C     CDEXP     Fortran function returns exp(COMPLEX*16)
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     DELTA     sampling interval
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     DIMAG     Fortran function returns imag part of a COMPLEX*16
C     DREAL     Fortran function returns real part of COMPLEX*16
C     F         the input f_k's as a COMPLEX*16s
C     FF        a point F_n in the transform
C     FOFX      function returns f(x) as a REAL*8
C     I         i, the square root of -1
C     K         index on input values
C     N         index on transform values
C     NN        number of samples
C     NU        freqency corresponding to a transform value
C     PI        the circle constant
C     W         exp(i*2*pi/N)
C     XK        a sample value of x as a REAL*8
C
      REAL*8 DELTA,FOFX,NU,PI/3.1415926535897932D0/,XK
      COMPLEX*16 F(0:63),FF,I/(0.D0,1.D0)/,W
C
C ------------------------------------------------------------------
C
C     find out how many points to use
      READ *,NN
      DELTA=3.5D0/DFLOAT(NN-1)
C
C     sample f(x)
      DO 1 K=0,NN-1
           XK=DELTA*DFLOAT(K)
           F(K)=DCMPLX(FOFX(XK),0.D0)
    1 CONTINUE
C
C     find the complex constant whose powers appear in the series
      W=CDEXP(I*DCMPLX(2.D0*PI/DFLOAT(NN),0.D0))
C
C     compute each F_n from the DFT series
      DO 2 N=0,NN-1
           FF=(0.D0,0.D0)
           DO 3 K=0,NN-1
                FF=FF+W**(N*K)*F(K)
    3      CONTINUE
           NU=DFLOAT(N)/(DELTA*DFLOAT(NN))
           WRITE(1,901) NU,DREAL(FF)
           WRITE(2,901) NU,DIMAG(FF)
  901      FORMAT(2(1X,1PE13.6))
    2 CONTINUE
      STOP
      END
```

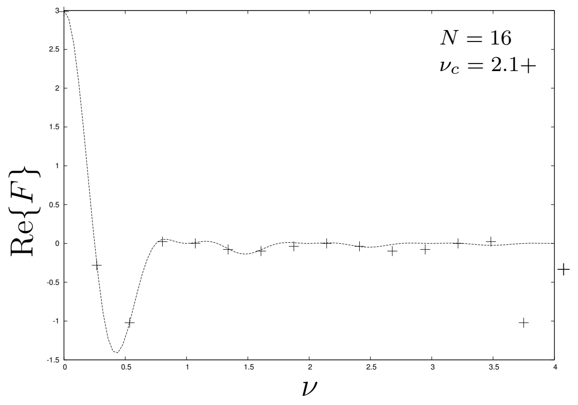
Using this program to calculate the DFT of the pulse for several values of N yields the results pictured at the bottom of the page. Each graph plots values of $\Delta_x F_n$ as points, along with the exact transform curve we found in §2. Only the real parts of $F(\nu)$ and $\Delta_x F_n$ are shown, but the agreement of their imaginary parts for different values of N follows a similar pattern.

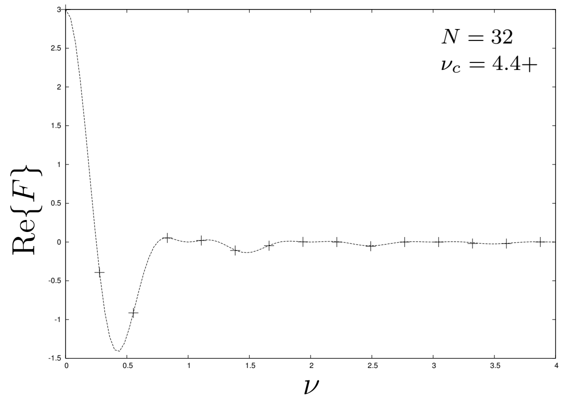
The F_n values are spaced apart in frequency by $\Delta_\nu = 1/(N\Delta_x)$, and $\Delta_x = b/(N-1)$ so

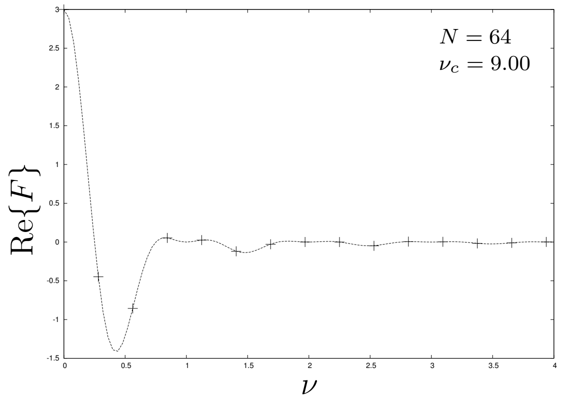
$$\Delta_\nu = \frac{1}{N\left(\frac{b}{N-1}\right)} = \left(\frac{1}{b}\right)\left(\frac{N-1}{N}\right).$$

In our example this quantity increases from 0.25 to about 0.28 as N increases from 8 to 64, so the points have almost the same spacing in all four graphs. To decrease it we would need to increase the limit of integration b.









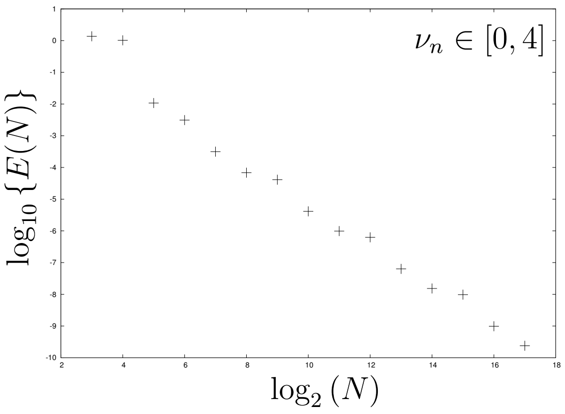
When $N = 8$, the points for $n = 0$, 1, 3, 4, 5, and 7 fall near the curve but those for $n = 2$ ($\nu = 0.5$) and 6 ($\nu = 1.5$) do not; they are hard to see at all, because they are buried in the horizontal axis. When $N = 16$ the points fall close to the curve until about $n = 10$ ($\nu \approx 2.6$) and the last two are noticeably wrong (the $n = 15$ point is slightly outside the range of the graph, at $\nu \approx 4.02$). Increasing N further yields graphs in which the points accurately approximate $F(\nu)$ over the entire frequency range shown. The critical frequency values ν_c listed in the graph legends are discussed in §4.5 below.

More samples yield narrower rectangles in the Riemann sum and more accurate approximations to the Fourier integral. The truncation error of rectangle-rule quadrature is proportional to the first power of Δ_x [4, equation 5.30b] so it might seem that the error in the DFT approximation of the Fourier transform integral would also be first order in Δ_x [10, page 577]. However, while increasing N decreases Δ_x it also changes the summand function $f_k e^{i2\pi nk/N}$ (as is evident in the graphs of $g(x)$ presented in §4.1) so the story is not so simple.[4]

For our §2 pulse waveform it is possible at least to study the error in the DFT approximation experimentally. By evaluating the DFT and the exact formula for the transform we can compute the absolute error over a given set V of frequencies as

$$E(N) = \max_{\nu_n \in V} |\Delta_x \text{Re}\{F_n\} - \text{Re}\{F(\nu_n)\}|.$$

For frequencies $\nu_n = n/(N\Delta_x)$ between zero and 4 (the range in the graphs on the previous page) and $N = 2^3 \ldots 2^{17}$ we get the graph below.



Least-squares regression on these points yields

$$\log_{10}\{E(N)\} = 1.8 - 0.68 \log_2(N).$$

Then

$$\begin{aligned} E &= 10^{1.8-0.68\log_2(N)} = (10^{1.8})(10^{-0.68})^{\log_2(N)} \\ &\approx 63(2^{-2.26})^{\log_2(N)} = 63(2^{\log_2(N)})^{-2.26} \\ &\approx 63/N^{2.26} \end{aligned}$$

[4]Following the approach used in [4, §5.2-§5.3] to analyze the approximation, this effect intrudes at the step of summing the errors committed in each subinterval to obtain a formula for the composite error. I will be grateful to anyone who shows me how to complete that derivation and obtain a formula for the error in the DFT approximation as a function of N.

For our example $\Delta_x = 3.5/(N-1) \approx 3.5/N$, so $N \approx 3.5/\Delta_x$ and

$$E \approx (63/3.5^{2.26})\Delta_x^{2.26} \approx 3.7\Delta_x^{2.26}$$

For low enough frequencies, $E \propto \Delta_x^{2.26}$ rather than to Δ_x^1, so the DFT approximation seems quite a bit better than we expected from the properties of rectangle-rule quadrature.

Increasing N decreases both **sampling error**, which results from ignoring values of $f(x)$ between the x_k, and **truncation error**, which results from approximating the Fourier integral by a Riemann sum.[5] In addition to those errors, DFT calculations are of course also subject to ordinary **roundoff errors** such as those resulting from cancellation and underflow [7, §4.3]. Roundoff errors accumulate as floating-point arithmetic is performed, so if the $E(N)$ graph were extended to large enough values of N the total error might begin to go back up as increasing roundoff became more important than decreasing truncation error. However, for our example it appears that roundoff is not a serious worry, and the inherent stability of numerical quadrature [4, page 284] suggests that might be true of DFT calculations generally.

We have just seen that if we pay attention to F_n values only up to $\nu = 4$, the DFT approximation quickly gets better as N increases. But in the graphs of $\text{Re}\{F\}$ on page 23, we noticed for small values of N that the error in the DFT approximation got bigger (though in an irregular fashion) as ν increased. How good is the DFT approximation at different frequencies for a fixed large value of N? To investigate that question we can think of the error as a function of ν_{max}, the highest frequency of the F_n we use, and find

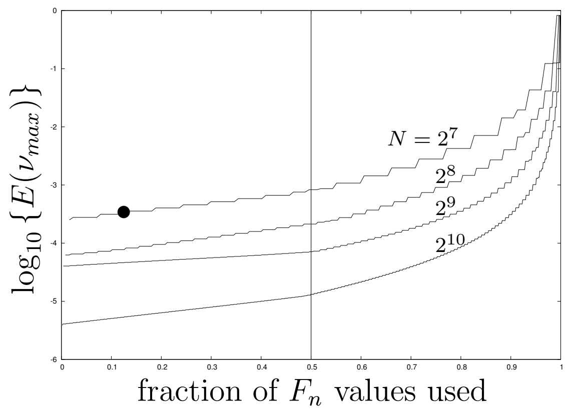
$$E(\nu_{max}) = \max_{\nu_n \in V} |\Delta_x \text{Re}\{F_n\} - \text{Re}\{F(\nu_n)\}|$$

where

$$V = \{0 \ldots \nu_{max}\}$$

and ν_{max} runs from 0 to $(N-1)/(N\Delta_x)$. The graph on the next page shows $E(\nu)$ for several different fixed values of N. To allow these curves to be compared on a single graph, the horizontal axis shows the *fraction* of the F_n values considered in computing $E(\nu)$. For example, when $N = 128$ the DFT yields $F_0 \ldots F_{127}$ corresponding to frequency components from $\nu = 0$ to $\nu = 36$. If we pay attention to only the frequency components between $\nu = 0$ and $\nu = 4$, say, we are using $\frac{1}{9} \approx 0.11$ of the F_n values. The dot marks that point on the curve for $N = 128$, where the log of $E(\nu_{max})$ is -3.5; this is the same error we found earlier at $N = 128$ when we plotted the log of $E(N)$ for $\nu \in [0, 4]$.

[5]These errors are nonetheless *different,* because by using a higher-order quadrature (such as the trapezoid rule) we could reduce the truncation error *without* changing the sampling error.



From this graph we can see that the error of the DFT approximation is much bigger in the F_n corresponding to high frequencies than it is in those corresponding to low ones. Although the big errors that we noticed in the Re(F) graphs at high frequencies appeared to go away as we increased N, that's only because we didn't look beyond $\nu = 4$. In fact, the error in *every* DFT approximation becomes large at frequencies approaching $\nu_N = (N-1)/(N\Delta_x)$ because of the phenomenon we first saw in the rectangle-rule pictures on page 15. The Fourier integrand $g(x) = f(x)e^{i2\pi\nu x}$ oscillates faster and faster as ν increases, and no matter how narrow we have made the Riemann sum rectangles (by choosing N) they end up being too wide before we reach the highest frequency in the transform (which is also determined by the N we chose). For each value of N there is a frequency beyond which the DFT approximation is useless.

Each of the curves above is a roughly straight line with shallow slope[6] for the first half of the F_n values, corresponding to results that are accurate and get only slightly worse as the frequency increases. Halfway to the highest frequency in each transform, near the **critical frequency** $\nu_c = \frac{1}{2}/\Delta_x \approx \frac{1}{2}\nu_N$, the error curves turn up sharply.

The critical frequency is $\frac{1}{2}/\Delta_x$ because that allows 2 samples per cycle of the Fourier integrand $g(x)$ [10, page 494]. One of the rectangle-rule pictures on page 15 is for the critical frequency $\nu_c = \frac{1}{2}/0.5 = 1$, and as noted there the Riemann sum still provides a plausible estimate of the area under the $g(x)$ curve. Dots mark the samples so you can convince yourself that there are 2 of them per cycle (e.g., rise from zero, fall through zero, and rise back to zero) of that integrand. For an integrand at a higher frequency, such as the one shown in the graph for $\nu = 2\nu_c = 2$ the Riemann sum approximation is grossly wrong because the 8 samples we used are too few to capture the ups and downs in $g(x)$. This is a sort of sampling error, in which interesting things happen to the function being sampled, in this case $g(x)$, in between measurements.

What fraction of the F_n values we can use depends on how much error we can tolerate in

[6]The curve for $N = 2^9$ is flatter than the others, consistent with the small decrease in error between $N = 2^8$ to $N = 2^9$ seen in the $E(N)$ graph on page 24. These are experimental data for one particular example, describing an algorithmic process in which circumstance plays a role, and apparently using 2^9 samples just happens to work well for our $f(x)$ (the curve above is nearly flat up to the critical frequency). I have omitted the first two data points from each curve because in two cases those points just happen to fall far below the third, a chance effect that would obscure the real story.

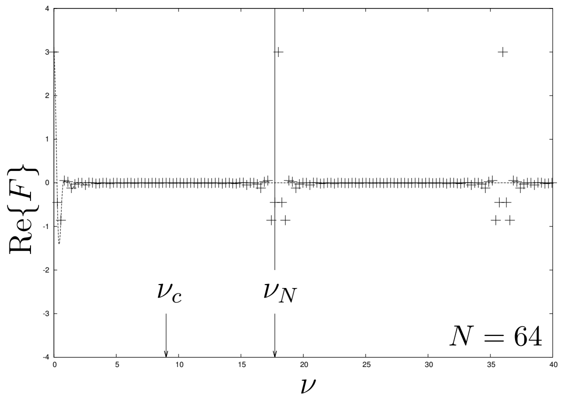
the DFT approximation. To get more accuracy we must throw away more high-frequency F_n values and be content with a narrower frequency range, or increase N so that the frequency range of the transform is wider to begin with. Doing either of those things is called **oversampling**. The Re$\{F\}$ graphs on page 23 illustrate the effect of oversampling; increasing N increases ν_c and moves the frequency window of interest (there $\nu \in [0, 4]$) farther to the left of the knee in the error curve, which improves accuracy.

4.5 Aliasing

In §4.1 we wrote the forward DFT as

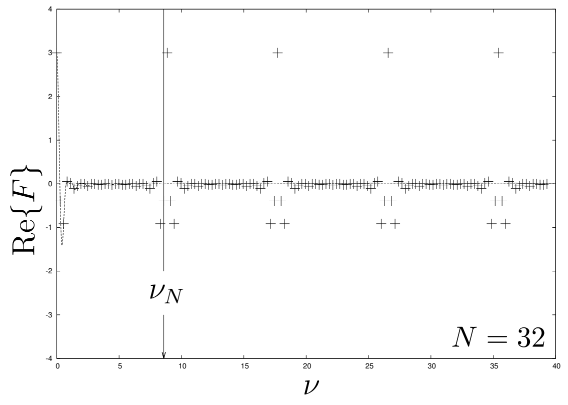
$$F_n = \sum_{k=0}^{N-1} f_k e^{i2\pi nk/N}$$

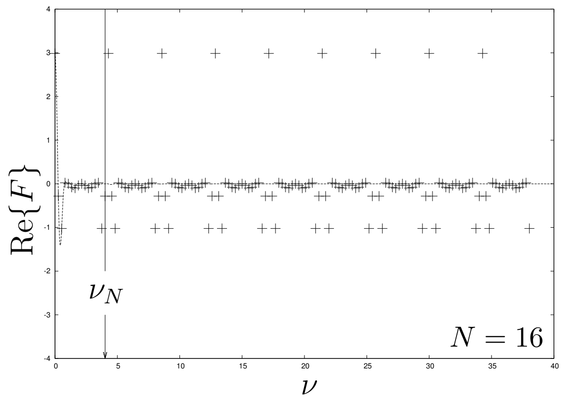
for $n = 0 \ldots N-1$, each result corresponding to a frequency $\nu_n = n/(N\Delta_x)$. However, the sum can be calculated for *any* integer n. Another way of understanding the growth of error in the DFT approximation as ν_n approaches $\nu_N = (N-1)/(N\Delta_x)$, the highest frequency in the transform, is to examine F_n values for frequencies *higher* than that. The graph below shows $\Delta_x F_n$ for our pulse waveform when $N = 64$ as n continues on past the vertical line drawn at $n = 63$ (the point representing F_{63} is buried in that line).



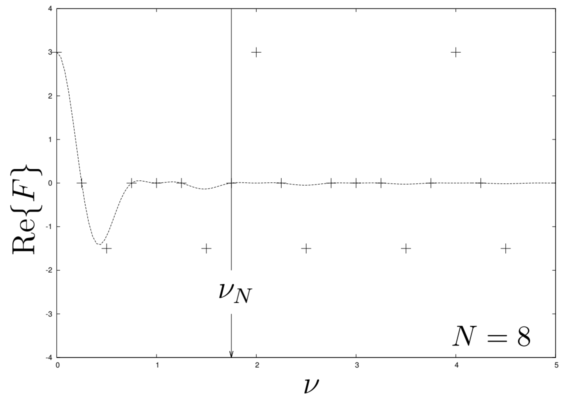
The F_n values repeat every N points (unlike the continuous transform $F(\nu)$, which is drawn as a solid curve). As mentioned in §4.1, F_n is periodic in n with period N and we can find F_n at negative frequencies from those at positive frequencies by using the fact that $F_{-n} = F_{N-n}$. Because of this periodicity, the F_n values near ν_N are also values of the transform at small *negative* frequencies. In the graph above we can see that F_{59} thru F_{63} are the same as F_{64} through F_{68} or F_{-5} through F_{-1}. It is as if the transform values F_{-5} through F_{-1} adopted the false identities or aliases F_{59} through F_{63}. This phenomenon is therefore known as **aliasing**. At the critical frequency ν_c, the first cycle of the DFT (the one that is useful for approximating $F(\nu)$) starts running into the left-hand part of the second cycle (the part that would be that repetition's negative frequency components if the repetition were centered at zero frequency). In this way the signal's power outside of the frequency range $[-\nu_c, +\nu_c]$ gets moved by the sampling process into that frequency range [10, page 495].

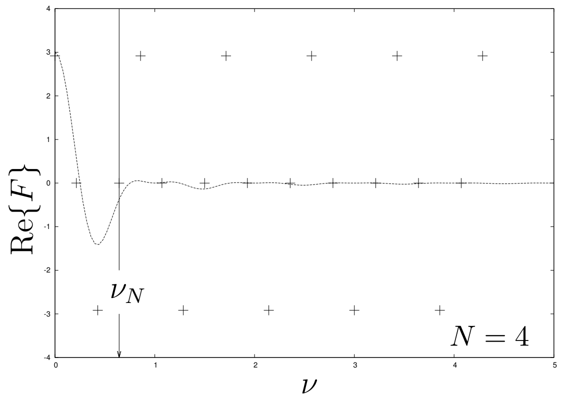
In the $N = 64$ transform of our pulse, most of the action seems to be over long before the DFT repeats, so we could just ignore the high-frequency F_n and use the others in approximating $F(\nu)$. Using fewer samples changes the DFT and reduces its period, as shown in the graphs below.





Continuing to reduce N will eventually make the big negative-frequency components of the DFT's second cycle overlap (and add to) the big positive-frequency components that we want! The gory details of this collision are shown below with an enlarged frequency scale.





Even with as few as 8 samples the DFT is of some use in approximating $F(\nu)$ below $\nu_c \approx \frac{1}{2}\nu_N$, but when $N = 4$ essentially the whole transform is destroyed by aliasing.

For the pulse of §2, $F(\nu)$ gets small at high frequencies but there is no value of ν above which it remains precisely zero. We must therefore expect that there is no value of n above which F_n remains precisely zero, so the effects of aliasing extend forever to both positive and negative frequencies. Thus *every* F_n for this pulse, including those corresponding to frequencies less than $\nu = \nu_c$, can be regarded as contaminated to some extent by aliasing. Many real signals have this property, so aliasing is often a useful way of understanding the errors in the DFT approximation.

It is because our pulse $f(x)$ is of finite duration that its transform $F(\nu)$ spans all frequencies [3, §9.4]. A signal that is *not* of finite duration can be **bandwidth limited**, so that its Fourier transform is identically zero outside some finite range of frequencies. In that case, according to the **sampling theorem** [10, page 494], $f(x)$ can be *exactly* reconstructed from its samples if the sampling rate is high enough so that the highest frequency in the signal does not exceed the critical frequency. Then there will be *no* aliasing, and we can think of representing the finite area under the transform by a Riemann sum. The Fourier transform of the sinc function $f(x) = \sin(x)/x$ is a rectangular pulse; thus each rectangle in the Riemann sum is the transform of a sinc function, and the waveform $f(x)$ must be exactly a weighted sum of sinc functions [9, equation 1.32]. This ideal result sheds some additional light on aliasing and is useful as an aid to thinking about the DFT of signals that are *approximately* bandwidth-limited, even though in practice we could never receive a signal of infinite duration.

5 DFT Evaluation by Fast Fourier Transform

The DFT is so useful that people have been working for many years on ways to speed up its computation. The current state of the art is the **fast Fourier transform** or **FFT**.

Although zero subscripts can be used in modern programming languages (including Classical FORTRAN but *not* including MATLAB) the FFT code I studied is typical of vintage-1965 programs in numbering the data x_k, function values f_k, frequencies ν_n, and transform values F_n starting with 1, so that k and n both run from 1 to N rather than from 0 to $N-1$. Rewriting the formulas for the DFT and its inverse to follow this convention we get

$$F_n = \sum_{k=1}^{N} W^{(n-1)(k-1)} f_k \qquad f_k = \frac{1}{N}\sum_{n=1}^{N} W^{-(n-1)(k-1)} F_n.$$

5.1 Recursive Decomposition

The sum for F_n can be separated into two parts, one composed of the odd-numbered terms and the other composed of the even-numbered terms. Using $W = e^{i2\pi/N}$ we can write

$$\begin{aligned}
F_n &= \sum_{k=1}^{N} e^{i2\pi(n-1)(k-1)/N} f_k \\
&= e^{i2\pi(n-1)(0)/N} f_1 + e^{i2\pi(n-1)(1)/N} f_2 + e^{i2\pi(n-1)(2)/N} f_3 + e^{i2\pi(n-1)(3)/N} f_4 + \cdots \\
&= \underbrace{\sum_{k=1}^{N/2} e^{i2\pi(n-1)2(k-1)/N} f_{2k-1}}_{\text{odd terms}} + \underbrace{\sum_{k=1}^{N/2} e^{i2\pi(n-1)(2k-1)/N} f_{2k}}_{\text{even terms}} \\
&= \sum_{k=1}^{N/2} e^{i2\pi(n-1)2(k-1)/N} f_{2k-1} + \sum_{k=1}^{N/2} e^{i2\pi(n-1)(1+2k-2)/N} f_{2k} \\
&= \sum_{k=1}^{N/2} e^{i2\pi(n-1)(k-1)2/N} f_{2k-1} + \sum_{k=1}^{N/2} e^{i2\pi(n-1)(1)/N} e^{i2\pi(n-1)(2k-2)/N} f_{2k} \\
&= \sum_{k=1}^{N/2} e^{i2\pi(n-1)(k-1)/(N/2)} f_{2k-1} + e^{i2\pi(n-1)(1)/N} \sum_{k=1}^{N/2} e^{i2\pi(n-1)(k-1)/(N/2)} f_{2k} \\
F_n &= F_n^o + W^{(n-1)} F_n^e \qquad n = 1 \ldots N.
\end{aligned}$$

We could find F_n if we knew F_n^o, the transform of just the odd-numbered f_k's in the data, and F_n^e, the transform of just the even-numbered ones. Similarly, we could find F_n^o if we knew F_n^{oo} and F_n^{oe}, and we could find F_n^e if we knew F_n^{eo} and F_n^{ee}. If we continue applying this idea, bisecting each subset of the data into odd-ordered and even-ordered terms at each step, the calculations that we are promising to do when we unwind the recursion form a binary tree. Assuming that N is a power of 2 (not just any multiple as we assumed earlier), the recursive subdivision ends with each leaf node being a single data element from the list to be transformed. The transform of a single number is the number itself.

$$F_n = \sum_{k=1}^{1} e^{i2\pi(n-1)(k-1)/1} f = f$$

To understand the recursive process it is helpful to consider an example, so suppose we start with 8 data values f_k for $k = 1 \ldots 8$. Then, for $n = 1 \ldots 8$,

$$F_n = \sum_{k=1}^{8} W_8^{(n-1)(k-1)} f_k = F_n^o + W_8^{(n-1)} F_n^e \quad \text{where} \quad W_8 = e^{i2\pi/8}$$

$$\left.\begin{aligned} F_n^o &= \sum_{k=1}^{8/2} e^{i2\pi(n-1)(k-1)/(8/2)} f_{2k-1} = \sum_{k=1}^{4} W_4^{(n-1)(k-1)} f_{2k-1} \\ F_n^e &= \sum_{k=1}^{8/2} e^{i2\pi(n-1)(k-1)/(8/2)} f_{2k} = \sum_{k=1}^{4} W_4^{(n-1)(k-1)} f_{2k} \end{aligned}\right\} \quad \text{where} \quad W_4 = e^{i2\pi/4}$$

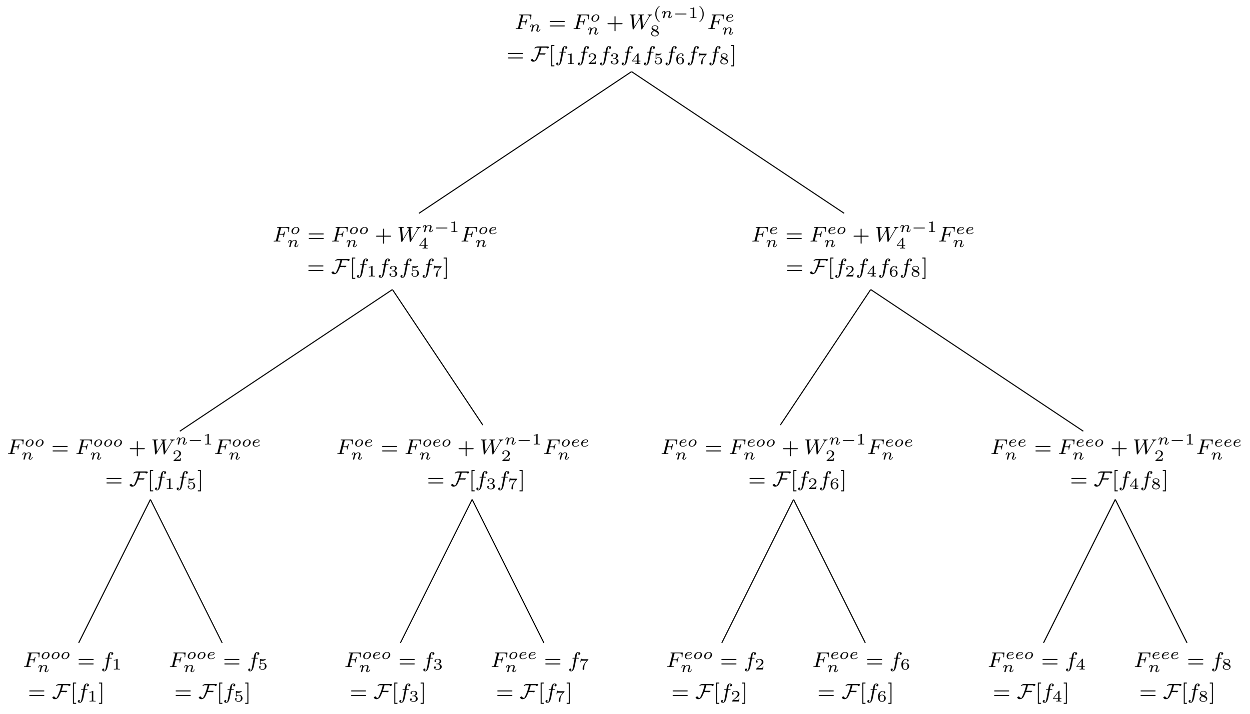
For example, when $n = 2$ we have from the formulas above

$$\begin{aligned} F_2^o &= W_4^0 f_1 + W_4^1 f_3 + W_4^2 f_5 + W_4^3 f_7 \\ F_2^e &= W_4^0 f_2 + W_4^1 f_4 + W_4^2 f_6 + W_4^3 f_8 \\ F_2 &= W_4^0 f_1 + W_4^1 f_3 + W_4^2 f_5 + W_4^3 f_7 + W_8(W_4^0 f_2 + W_4^1 f_4 + W_4^2 f_6 + W_4^3 f_8) \\ &= W_4^0 f_1 + W_8 W_4^0 f_2 + W_4^1 f_3 + W_8 W_4^1 f_4 + W_4^2 f_5 + W_8 W_4^2 f_6 + W_4^3 f_7 + W_8 W_4^3 f_8 \\ &= f_1 + W_8 f_2 + W_8^2 f_3 + W_8^3 f_4 + W_8^4 f_5 + W_8^5 f_6 + W_8^6 f_7 + W_8^7 f_8 \\ &= \sum_{k-1}^{8} W_8^{(k-1)} f_k \end{aligned}$$

where line five uses the fact that $W_4 = e^{i2\pi/4} = (e^{i2\pi/8})^2 = W_8^2$. The binary tree of calculations for this example is drawn on the next page and shows the recursive decomposition for any *single* value of n. For example, when $n = 7$ we have from the binary tree

$$\begin{aligned} F_7 &= \big((f_1 + W_2^6 f_5) + W_4^6(f_3 + W_2^6 f_7)\big) + W_8^6 \big((f_2 + W_2^6 f_6) + W_4^6(f_4 + W_2^6 f_8)\big) \\ &= f_1 + W_2^6 f_5 + W_4^6 f_3 + W_4^6 W_2^6 f_7 + W_8^6 f_2 + W_8^6 W_2^6 f_6 + W_8^6 W_4^6 f_4 + W_8^6 W_4^6 W_2^6 f_8 \\ &= f_1 + W_8^6 f_2 + W_8^{12} f_3 + W_8^{18} f_4 + W_8^{24} f_5 + W_8^{30} f_6 + W_8^{36} f_7 + W_8^{42} f_8 \\ &= \sum_{k=1}^{8} W_8^{6(k-1)} f_k \end{aligned}$$

where line three uses the observation that $W_2 = e^{i2\pi/2} = (e^{i2\pi/4})^2 = W_4^2$.


$F_n = F_n^o + W_8^{(n-1)} F_n^e = \mathcal{F}[f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8]$
$F_n^o = F_n^{oo} + W_4^{n-1} F_n^{oe} = \mathcal{F}[f_1 f_3 f_5 f_7]$
$F_n^e = F_n^{eo} + W_4^{n-1} F_n^{ee} = \mathcal{F}[f_2 f_4 f_6 f_8]$
$F_n^{oo} = F_n^{ooo} + W_2^{n-1} F_n^{ooe} = \mathcal{F}[f_1 f_5]$
$F_n^{oe} = F_n^{oeo} + W_2^{n-1} F_n^{oee} = \mathcal{F}[f_3 f_7]$
$F_n^{eo} = F_n^{eoo} + W_2^{n-1} F_n^{eoe} = \mathcal{F}[f_2 f_6]$
$F_n^{ee} = F_n^{eeo} + W_2^{n-1} F_n^{eee} = \mathcal{F}[f_4 f_8]$
$F_n^{ooo} = f_1 = \mathcal{F}[f_1]$
$F_n^{ooe} = f_5 = \mathcal{F}[f_5]$
$F_n^{oeo} = f_3 = \mathcal{F}[f_3]$
$F_n^{oee} = f_7 = \mathcal{F}[f_7]$
$F_n^{eoo} = f_2 = \mathcal{F}[f_2]$
$F_n^{eoe} = f_6 = \mathcal{F}[f_6]$
$F_n^{eeo} = f_4 = \mathcal{F}[f_4]$
$F_n^{eee} = f_8 = \mathcal{F}[f_8]$

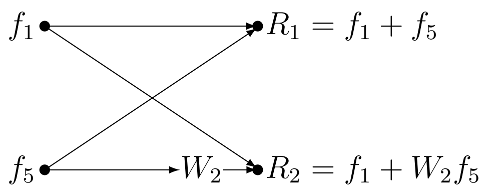
If the tree were represented explicitly using a linked list, each F_n might be found by performing a depth-first traversal sweeping from left to right, along the way doing the indicated calculation at each node and saving the result. If there are N data values the tree has

$$1 + 2 + 4 + \cdots + N/2 = \sum_{m=1}^{\log_2(N)} 2^{m-1} = \frac{2^{\log_2(N)} - 1}{2 - 1} = N - 1$$

nodes at which a calculation must be done, so the work of doing N traversals to find the whole DFT scales as N^2 (just like the naïve evaluation of the DFT discussed in §4). The storage needed for intermediate values (F_n^o, F_n^e, and so on) could be reused for each new n, so it scales as N, and because all of the f_k are used in finding each F_n the input data cannot be overwritten so another N locations would be needed to store the results.

5.2 Signal Flow Graphs

Fortunately, the special structure of the recursive decomposition can be exploited to speed things up. Each traversal of the binary tree for a given n involves many calculations that are also needed for other values of n. To take advantage of this fact we need a representation of the process that reveals in a single picture how the input data get transformed into all N output values F_n. Following [9, pages 294-297] I will use **signal-flow graphs** like the one shown on the next page.



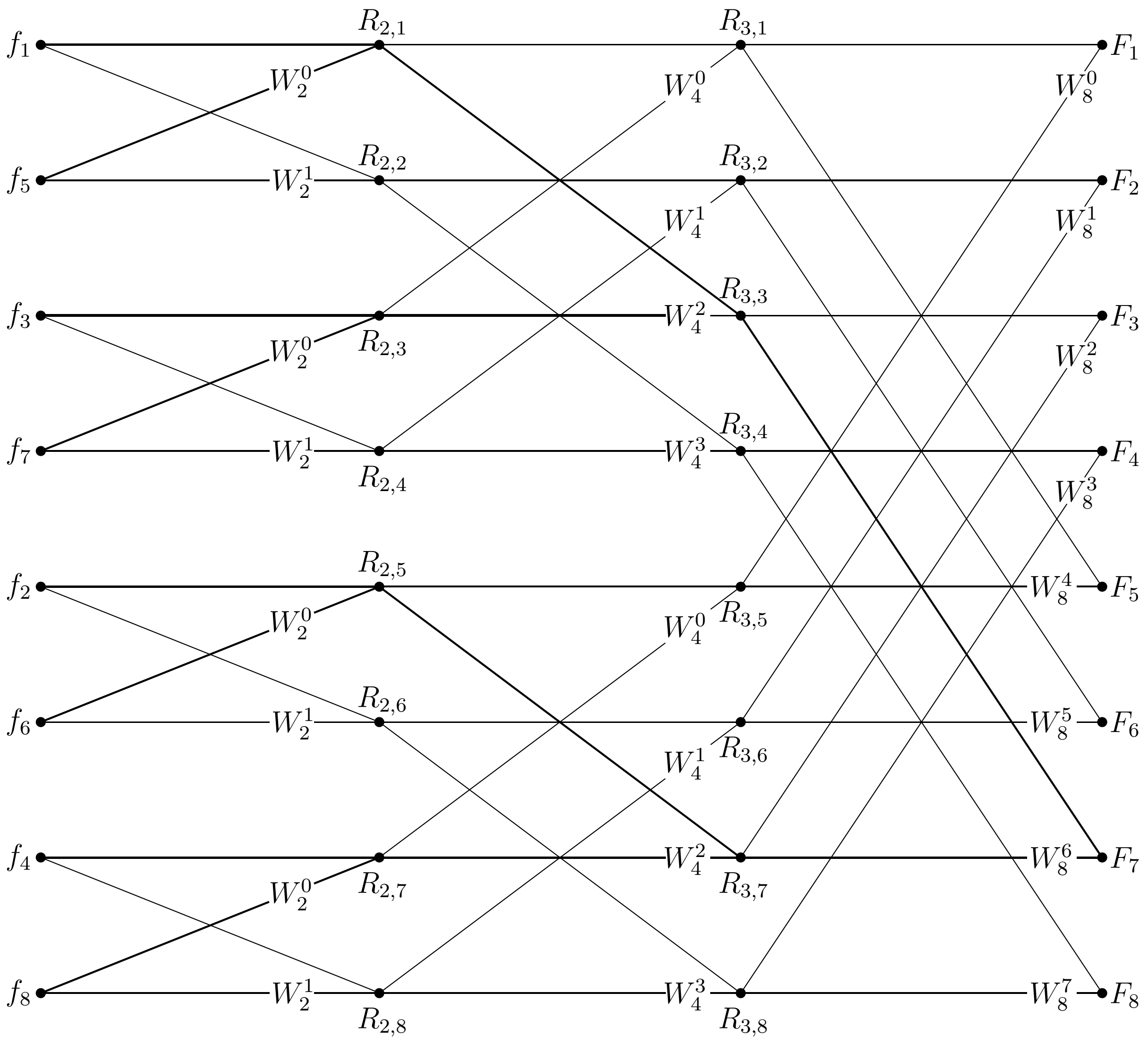
Here the input **signals** f_1 and f_5 are thought of as flowing from their nodes of origin on the left to the result nodes R_1 and R_2 on the right. On its way to R_2, f_5 gets multiplied by the **gain** W_2; unmarked links have a gain of 1. Links are connected only where there is a • dot; if they simply cross that does not interfere with either signal. At each result node the incoming signals are added. Both R_1 and R_2 correspond to the F_n^{oo} node in the binary tree, but for different values of n. From the definition of W we find $W_2 = e^{i2\pi/2} = e^{i\pi} = -1$. Thus for odd values of n (even powers of W_2) $W_2^{(n-1)} = +1$ and $F_n^{oo} = R_1$, while for even values of n (odd powers of W_2) $W_2^{(n-1)} = -1$ and $F_n^{oo} = R_2$.

A signal-flow graph is given on the next page for the whole computation of F_1 thru F_8. To see how it specifies the calculation, consider finding F_7 as we did using the binary tree. In the signal-flow graph the node labeled F_7 receives inputs of $R_{3,3}$ and $W_8^6 \times R_{3,7}$. Those nodes receive inputs from others, which in turn receive inputs from others, forming the tree drawn in darker links. Traversing that tree we find

$$\begin{aligned}
F_7 &= W_8^6 R_{3,7} + R_{3,3} \\
&= W_8^6 (W_4^2 R_{2,7} + R_{2,5}) + W_4^2 R_{2,3} + R_{2,1} \\
&= W_8^6 \left(W_4^2 (W_2^0 f_8 + f_4) + W_2^0 f_6 + f_2 \right) + W_4^2 (W_2^0 f_7 + f_3) + W_2^0 f_5 + f_1 \\
&= f_1 + W_8^6 f_2 + W_4^2 f_3 + W_8^6 W_4^2 f_4 + W_2^0 f_5 + W_8^6 W_2^0 f_6 + W_4^2 W_2^0 f_7 + W_8^6 W_4^2 W_2^0 f_8 \\
&= f_1 + W_8^6 f_2 + W_8^4 f_3 + W_8^{10} f_4 + f_5 + W_8^6 f_6 + W_8^4 f_7 + W_8^{10} f_8 \\
&= f_1 + W_8^6 f_2 + W_8^{12} f_3 + W_8^{18} f_4 + W_8^{24} f_5 + W_8^{30} f_6 + W_8^{36} f_7 + W_8^{42} f_8 \\
&= \sum_{k=1}^{8} W_8^{6(k-1)} f_k
\end{aligned}$$

In the fifth line we used $W_4^2 = e^{2i2\pi/4} = e^{4i2\pi/8} = W_8^4$, and in the sixth $W_8^8 = e^{8i2\pi/8} = (e^{i\pi})^2 = (-1)^2 = 1$. The tree drawn in darker links in the signal-flow graph is just the binary tree when $n = 7$; the other instances of the binary tree are in the signal-flow graph too, each rooted at the node where its F_n is delivered. This way of organizing the calculation is advantageous in several important ways.

First, we can find all the F_n's in parallel, and without traversing any trees. Starting with the given values of the f_k we can find $R_{2,1}$ thru $R_{2,8}$ in one **stage** of the calculation. I will number the stages $m = 1, 2, 3$ from left to right. Once the stage-1 calculation is complete we can use the $R_{2,p}$ values to find $R_{3,1}$ thru $R_{3,8}$ in stage 2. Finally, using those values we can find F_1 thru F_8 in stage 3. Stage 1 is a stack of $N/2$ 2-point FFTs like the one diagrammed in the little network flow graph above; each has 2 inputs corresponding to leaf nodes of the binary tree and 2 outputs corresponding to the possible values of W_2^{n-1} (above we saw that



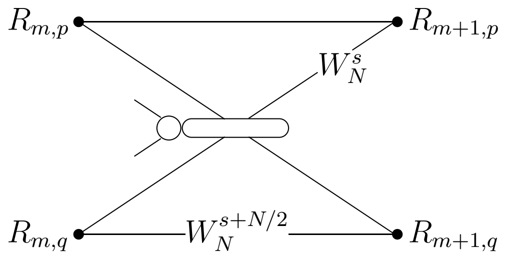
they are -1 and $+1$). Stage 2 is a stack of $N/4$ 4-point FFTs, each with 4 inputs that are outputs from the first stage and 4 outputs corresponding to the possible values of W_4^{n-1} (which happen to be 1, i, -1, and $-i$). Stage 3 is a single 8-point FFT (that is, a "stack" of $N/8 = 1$ 8-point FFTs) whose inputs are outputs from the second stage and whose outputs correspond to the 8 possible values of W_8^{n-1}. The number of stages is $\log_2(N)$, and since each stage has N nodes at which a calculation is required the work now scales as $N \log_2(N)$ rather than as N^2. This might seem a technical detail, but in practice it is what makes evaluating the DFT computationally tractable for large values of N. The table on the top of the next page compares $\log_2(N)$ to N^2 for N equal to some small powers of 2.

The second big advantage of organizing the calculation according to the signal-flow graph is that the input data for each stage are not needed by subsequent stages, so we can overwrite them with results. This means we can perform the whole calculation **in-place**, without needing *any* extra storage for the intermediate values $R_{m,p}$ or outputs F_n.

N	$N\log_2(N)$	N^2	$(N\log_2(N))/(N^2)$
2	2	4	0.500000
4	8	16	0.500000
8	24	64	0.375000
16	64	256	0.250000
32	160	1024	0.156250
64	384	4096	0.093750
128	896	16384	0.054688
256	2048	65536	0.031250
512	4608	262144	0.017578
1024	10240	1048576	0.009766
2048	22528	4194304	0.005371
4096	49152	16777216	0.002930
8192	106496	67108864	0.001587
16384	229376	268435456	0.000854
32768	491520	1073741824	0.000458

5.3 The Butterfly Calculation

In each stage, the calculation that we repeat $N/2$ times (using different numbers) is *the same,* as illustrated by the representative examples drawn on the next page with heavy lines. Here I have expressed all of the link gains as powers of W_8. In the stage-1 example, to find $R_{2,1}$ and $R_{2,2}$ we use f_1 and f_5; both inputs are needed to find each result, and neither is needed anywhere else in the calculation. Similarly in stage 2 we need $R_{2,5}$ and $R_{2,7}$ only to find $R_{3,5}$ and $R_{3,7}$, and in stage 3 we need $R_{3,2}$ and $R_{3,6}$ to find F_2 and F_6. Each such calculation involves lines from the signal-flow graph that form the shape of a butterfly, like this.

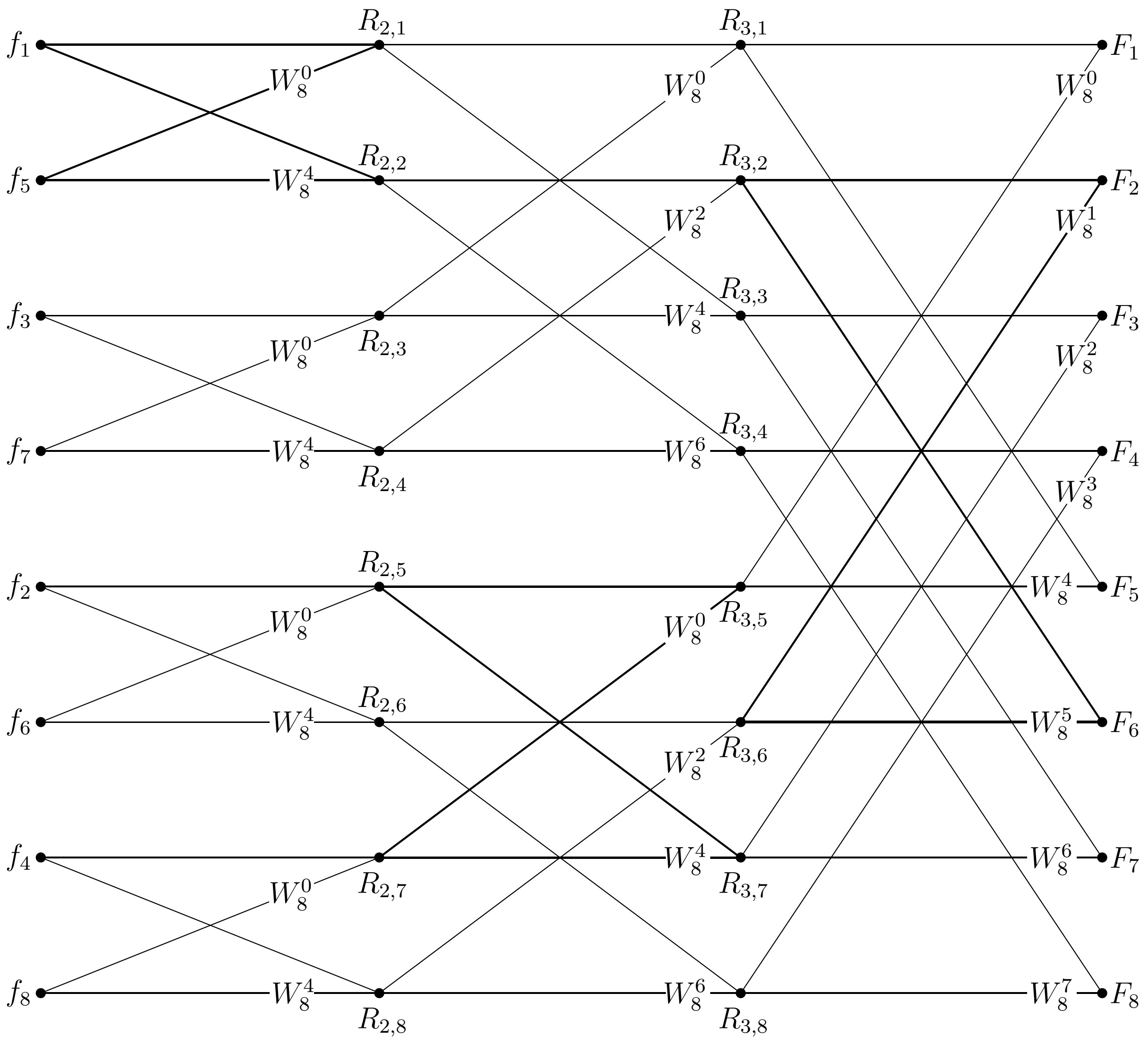


In the stage-1 example we have $m = 1$, $p = 1$, and $q = 2$ with $R_{m,p} = f_1$ and $R_{m,q} = f_5$. The gain $W_N^s = W_8^0$, so $s = 0$. In the stage-2 example, $m = 2$, $p = 5$, $q = 7$, and again $s = 0$. In the stage-3 example, $m = 3$, $p = 2$, $q = 6$, and $s = 2$. Each butterfly calculation can be expressed like this.

$$\begin{aligned} R_{m+1,q} &= R_{m,p} + W_N^{s+N/2} R_{m,q} \\ R_{m+1,p} &= R_{m,p} + W_N^s R_{m,q} \end{aligned}$$

But $W_N^{N/2} = e^{(N/2)i2\pi/N} = e^{i\pi} = -1$, so $W_N^{s+N/2} = -W_N^s$ and the calculation reduces to this.

$$\begin{aligned} R_{m+1,q} &= R_{m,p} - W_N^s R_{m,q} \\ R_{m+1,p} &= R_{m,p} + W_N^s R_{m,q} \end{aligned}$$



Thus, the final virtue of the signal-flow formulation is that the butterfly calculation consists simply of adding and subtracting the same product, so it is easy to program and fast to perform.

Doing the calculation in-place means that $R_{m+1,q}$ overwrites $R_{m,q}$, so a temporary variable is needed to remember the product. In pseudocode, what we need to do is this.

```
temp ← $W_N^s R_q$
$R_q$ ← $R_p - \text{temp}$
$R_p$ ← $R_p + \text{temp}$
```

It is important to update R_p last, because its original value is needed in finding R_q. As we work from left to right in the signal-flow graph from stage 1 thru stage $\log_2(N)$, we perform

$N/2$ butterfly calculations in each stage working from top to bottom. When we begin, the vector R contains the f_k; it gets updated in the successive stages and contains the F_n when the calculation is finished.

The program on the next page reads a sequence of 8 input values f_k (in the order they have from top to bottom in the signal-flow graph, or from left to right in the binary tree of §5.1) and prints out the resulting F_n. The lines of the listing are numbered so that I can refer to them in describing the code. Experiments confirm that this program produces output identical (modulo roundoff) to that from the naïve DFT implementation of §4. The pseudocode above translates directly into FORTRAN $\boxed{\texttt{56-58}}$ but those three statements are surrounded by code for indexing the appropriate elements of R and for computing the gains W_N^s. These operations can be related easily to the signal-flow graph, but they involve a lot of arithmetic. In the next Section we will consider less-obvious coding solutions that save on that work and allow for problems of arbitrary size.

The recursive decomposition of the DFT calculation that we have considered in this Section is called **decimation**[7] **in time**, because it forms subsequences of the input sequence $f_k = f(x_k)$ and x often represents time (though in image-processing applications it more commonly represents distance). The calculation can also be recursively decomposed by forming subsequences of the output sequence F_n, and that is called **decimation in frequency**. As discussed in [9, §6.3], to each decimation-in-time algorithm there corresponds a decimation-in-frequency algorithm obtained by interchanging the input and output sequences and reversing the directions of flow in the signal-flow graph. Decimation in frequency is more difficult to understand and provides no advantage in efficiency, so decimation in time seems to be more commonly used.

[7]In ordinary usage to decimate means to reduce by one-tenth, but here it means to decompose thru successive divisions by two.

```
C
C     This program implements the N=8 signal flow graph calculation.
C     Refer to the second signal flow graph.
C
C     variable  meaning
C     --------  -------
C     CDEXP     Fortran function returns exp(COMPLEX*16)
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     I         i, the square root of -1
C     K         index on input f values
C     L         index on groups of butterflies within a stage
C     M         index on stages in the signal flow graph
C     NB        number of butterflies in each group
C     NG        number of groups in this stage
C     P         node index at top of a butterfly
C     PI        the circle constant
C     Q         node index at bottom of a butterfly
C     R         the f's, then intermediate results, finally F's
C     S         power of W in this butterfly
C     SIGNAL    f values read in
C     T         index on butterflies in a group
C     TEMP      the intermediate in each pairwise calculation
C     W8        the constant whose powers are the link gains
C
      REAL*8 PI/3.1415926535897932D0/,SIGNAL(8)
      COMPLEX*16 I/(0.D0,1.D0)/,R(8),TEMP,W8
      INTEGER*4 P,Q,S,T
C
C -----------------------------------------------------------------
C
C     get the input f values
      READ(5,*) SIGNAL
      DO 1 K=1,8
           R(K)=DCMPLX(SIGNAL(K),0.D0)
    1 CONTINUE
C
C     find the constant whose powers are the link gains
      W8=CDEXP((I*2.D0*PI)/8.D0)
C
C     consider each stage
      DO 2 M=1,3
C          in each stage consider each group of butterflies
           NG=2**(3-M)
           DO 3 L=1,NG
C               in each group consider each butterfly
                NB=2**(M-1)
                DO 4 T=1,NB
C                    find the nodes involved in this butterfly
                     P=T+2*NB*(L-1)
                     Q=P+NB
C
C                    find the power of W to use in this butterfly
                     S=(T-1)*2**(3-M)
C
C                    perform the pairwise calculation
                     TEMP=(W8**S)*R(Q)
                     R(Q)=R(P)-TEMP
                     R(P)=R(P)+TEMP
    4           CONTINUE
    3      CONTINUE
    2 CONTINUE
C
C     report the transform values
      WRITE(6,901) R
  901 FORMAT(1PE13.6,1X,1PE13.6)
      STOP
      END
```

6 FFT Implementation

The Fourier transform $F(\nu)$ is in general a complex function of the real scalar ν, so the F_n found by the DFT are complex numbers and the data structure used throughout the signal-flow graph calculation must accommodate numbers that have both a real and an imaginary part. Although complex data types are native to both Classical FORTRAN and MATLAB (though not to C) the code I studied is typical of vintage-1965 programs in representing complex numbers by adjacent real values. Thus we find the f_k stored in a real vector of length $2N$ elements, like this.

$\mathrm{Re}(f_1)$	$\mathrm{Im}(f_1)$	$\mathrm{Re}(f_2)$	$\mathrm{Im}(f_2)$	
`DATA(1)`	`DATA(2)`	`DATA(3)`	`DATA(4)`	

Gains W and other complex quantities are similarly represented by pairs of real values. To compute the product of two complex numbers $a + bi$ and $c + di$ represented in this way, one calculates the real and imaginary parts of the product separately like this.

$$\mathrm{Re}\{(a + bi) \times (c + di)\} = ac - bd$$
$$\mathrm{Im}\{(a + bi) \times (c + di)\} = ad + bc$$

6.1 The Butterfly Calculation Revisited

There are not many ways to code the butterfly calculation itself, but one conspicuous inefficiency in the code of §5.3 is that it computes the same powers of W_N over and over. Some work could be avoided if, within each stage, the butterflies using each power of W_N were all evaluated before the next power was found. More work could be avoided by using multiplications, rather than complex exponentiations, to find the successive powers of W_N. A final improvement would be to compute the various indices by multiplications and additions rather than by integer exponentiations.

6.1.1 A Typical Butterfly Algorithm

The FORTRAN program listed on the following two pages incorporates those improvements. This code is inspired by the butterfly part of the `four1` subroutine from [10, page 501], but I have altered the order of statements, made everything double precision, used the coding style described in [7], and changed the names of several variables. Experiments confirm that this program produces output identical to that from the naïve DFT implementation of §4.

Powers of W. Referring to the first signal-flow graph (on page 34) we see that the gains there are powers of W_2 in the first stage, powers of W_4 in the second stage, and powers of W_8 in the third. In the program the variable `LMAX` takes on $\boxed{\texttt{47,59,77}}$ the values 2, 4, and 8 to determine the W that is used in each stage. The real and imaginary parts of W can then be found from

```
 1 C
 2 C     This program implements the signal flow graph calculation
 3 C     with some obvious improvements.  Refer to the first signal
 4 C     flow graph.
 5 C
 6 C     variable  meaning
 7 C     --------  -------
 8 C     DATA      the f's, then intermediate results, finally F's
 9 C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
10 C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
11 C     DSIN      Fortran function returns sine of a REAL*8
12 C     K         index on input f values
13 C     L         index on powers of W
14 C     LMAX      in the current stage, W=e^(i * TWOPI/LMAX)
15 C     N         number of data values
16 C     NN        number of (real,complex) data pairs
17 C     P         odd index of (real,imag) at top of a butterfly
18 C     PSTEP     nodes between butterflies using the same power of W
19 C     Q         odd index of (real,imag) at bottom of a butterfly
20 C     SIGNAL    f values read in
21 C     TEMPI     imaginary part of WRq
22 C     TEMPR     real part of WRq
23 C     THETA     angle corresponding to W^s
24 C     TWOPI     twice the circle constant
25 C     WI        imaginary part of W
26 C     WPI       imaginary part of W multiplier
27 C     WPR       real part of W multiplier
28 C     WR        real part of W
29 C     WTEMP     saved value of WR
30 C
31       PARAMETER(NN=8,N=2*NN)
32       REAL*8 DATA(N)
33       REAL*8 SIGNAL(NN),TEMPI,TEMPR,THETA
34       REAL*8 TWOPI/6.2831853071795865D0/,WI,WPI,WPR,WR,WTEMP
35       INTEGER*4 P,PSTEP,Q
36 C
37 C -----------------------------------------------------------------
38 C
```

$$\begin{aligned}
W &= e^{i2\pi/\mathtt{LMAX}} = \cos(2\pi/\mathtt{LMAX}) + i\sin(2\pi/\mathtt{LMAX})\\
\mathrm{Re}(W) &= \cos(2\pi/\mathtt{LMAX})\\
\mathrm{Im}(W) &= \sin(2\pi/\mathtt{LMAX})
\end{aligned}$$

The code sets θ to $2\pi/\mathtt{LMAX}$ $\boxed{54}$ and `WPI` to $\sin\theta$ $\boxed{56}$ so `WPI` is the imaginary part of $W_{\mathtt{LMAX}}$, but for the real part of the W multiplier it uses $\boxed{55}$

$$\mathtt{WPR} = -2\sin^2\left(\tfrac{1}{2}\theta\right) = -2\left(\pm\sqrt{\frac{1-\cos(\theta)}{2}}\right)^2 = -|1-\cos(\theta)| = \cos(\theta) - 1.$$

When the next power of W is calculated $\boxed{73\text{-}75}$ the real and imaginary parts of the product are adjusted to account for the extra -1 in `WPR`, like this.

$$\begin{aligned}
\mathtt{WR} &\leftarrow \mathtt{WR}*\mathtt{WPR} - \mathtt{WI}*\mathtt{WPI} + \mathtt{WR}\\
&= \mathtt{WR}\times(\cos(\theta)-1) - \mathtt{WI}\times\sin(\theta) + \mathtt{WR}\\
&= \mathtt{WR}\cos(\theta) - \mathtt{WR} - \mathtt{WI}\sin(\theta) + \mathtt{WR}\\
&= \mathtt{WR}\cos(\theta) - \mathtt{WI}\sin(\theta)
\end{aligned}$$

$$\begin{aligned}
\mathtt{WI} &\leftarrow \mathtt{WI}*\mathtt{WPR} + \mathtt{WR}*\mathtt{WPI} + \mathtt{WI}\\
&= \mathtt{WI}\times(\cos(\theta)-1) + \mathtt{WR}\times\sin(\theta) + \mathtt{WI}\\
&= \mathtt{WI}\cos(\theta) - \mathtt{WI} + \mathtt{WR}\sin(\theta) + \mathtt{WI}\\
&= \mathtt{WI}\cos(\theta) + \mathtt{WR}\sin(\theta)
\end{aligned}$$

The new value of the complex gain represented by `WR` and `WI` is then

$$\begin{aligned}
(\mathtt{WR}\cos(\theta) - \mathtt{WI}\sin(\theta)) + i\,(\mathtt{WI}\cos(\theta) + \mathtt{WR}\sin(\theta)) &= \\
(\mathtt{WR} + i\mathtt{WI}) \times (\cos(\theta) + i\sin(\theta)) &= \\
W_{\mathtt{LMAX}}^{s-1} \times e^{i\theta} &= \\
W_{\mathtt{LMAX}}^{s-1} \times e^{i2\pi/\mathtt{LMAX}} &= \\
W_{\mathtt{LMAX}}^{s-1} \times W_{\mathtt{LMAX}} &= W_{\mathtt{LMAX}}^{s}.
\end{aligned}$$

```
C     get the input f values
      READ(5,*) SIGNAL
      DO 1 K=1,NN
           DATA(2*K-1)=SIGNAL(K)
           DATA(2*K  )=0.D0
    1 CONTINUE
C
C     consider each stage
      LMAX=2
    5 IF(LMAX.GE.N) GO TO 2
C        set W^0=1
         WR=1.D0
         WI=0.D0
C
C        compute the W for this stage
         THETA=TWOPI/DFLOAT(LMAX)
         WPR=-2.D0*(DSIN(0.5D0*THETA))**2
         WPI=DSIN(THETA)
C
C        consider each power of W used in this stage
         PSTEP=2*LMAX
         DO 3 L=1,LMAX,2
C             do the butterflies with this power of W in all groups
              DO 4 P=L,N,PSTEP
                   Q=P+LMAX
                   TEMPR=WR*DATA(Q)-WI*DATA(Q+1)
                   TEMPI=WR*DATA(Q+1)+WI*DATA(Q)
                   DATA(Q)=DATA(P)-TEMPR
                   DATA(Q+1)=DATA(P+1)-TEMPI
                   DATA(P)=DATA(P)+TEMPR
                   DATA(P+1)=DATA(P+1)+TEMPI
    4         CONTINUE
C
C             find the next power of W as W*WP
              WTEMP=WR
              WR=WR*WPR-WI*WPI+WR
              WI=WI*WPR+WTEMP*WPI+WI
    3    CONTINUE
         LMAX=PSTEP
      GO TO 5
C
C     report the transform values
    2 WRITE(6,901) DATA
  901 FORMAT(1PE13.6,1X,1PE13.6)
      STOP
      END
```

Indexing. There is no explicit index on the stages; when `LMAX.GE.N` we know that we are done, and the loop $\boxed{\texttt{48-78}}$ starting with the test and ending with `GO TO 5` is free [7, page 275]. In our example, $\texttt{N} = 2 \times N = 2 * \texttt{NN} = 16$ and `LMAX` takes on the values 2 for stage 1, 4 for stage 2, 8 for stage 3, and finally 16 which makes the test $\boxed{\texttt{48}}$ succeed to end the process with a branch to $\boxed{\texttt{81}}$ statement `2` (`LMAX` never does get to be greater than `N`).

In each stage, $W_{\texttt{LMAX}}$ is set $\boxed{\texttt{50-51}}$ to $W^0_{\texttt{LMAX}} = 1 + 0i$ for the first pass thru the `3` loop $\boxed{\texttt{60-76}}$. In the first stage, that is the only value to consider so the `3` loop does only one pass. (`LMAX` is `2` and the `DO` increment is `2`, so `L` takes on only the value `1` [7, page 81].) At the end of the `3` loop, `LMAX` is doubled $\boxed{\texttt{77}}$ for the next stage.

In the first stage `LMAX=2` $\boxed{\texttt{47}}$ so `PSTEP` is `4` $\boxed{\texttt{59}}$. This is the number of data values (twice the number of nodes vertically in the signal-flow graph) between the tops of adjacent butterflies. In the first stage the `4` loop is performed for `P` values of `1`, `5`, `9`, and `13` (`13+4=17` would exceed the loop limit of `N` which is `16` $\boxed{\texttt{31}}$). Recall from the introduction to this Section that each data item occupies two locations in `DATA`, so node $R_{2,1}$ in the signal-flow graph corresponds to `DATA(1)` and `DATA(2)`, $R_{2,2}$ to `DATA(3)` and `DATA(4)`, and so on. The top of the second butterfly, $R_{2,3}$ thus has its real value at `DATA(5)`, so that is the second value of `P` produced by the `4` loop. The data element corresponding to the bottom node of the butterfly is `Q=P+LMAX` $\boxed{\texttt{63}}$. In the first stage, for example, when `P` is `1` we find $\boxed{\texttt{63}}$ that `Q=P+LMAX=1+2=3` which is the real half of the data element corresponding to the bottom of the butterfly. In the butterfly calculation $\boxed{\texttt{64-69}}$ `DATA(P+1)` and `DATA(Q+1)` are the imaginary parts of the numbers whose real parts are stored at `DATA(P)` and `DATA(Q)`. The doubling of `LMAX` $\boxed{\texttt{77}}$ and `PSTEP` $\boxed{\texttt{59}}$ as the `5` loop progresses from stage to stage ensures that the foregoing logic works in each stage subsequent to the first.

6.1.2 A Little Improvement

Instead of using $\texttt{WPR} = -2\sin^2\left(\frac{1}{2}\theta\right)$ $\boxed{\texttt{55}}$ and including the terms `+WR` $\boxed{\texttt{74}}$ and `+WI` $\boxed{\texttt{75}}$ in computing the next value of $W_{\texttt{LMAX}}$, it would be more straightforward[8] to use $\texttt{WPR} = \cos(\theta)$ and omit the corrections. This change saves some floating-point arithmetic, makes the code easier to understand and explain, and produces the same results we got before.

6.2 Input Data Rearrangement

A consequence of the recursive decomposition described in §5.1 is that the input values need to be arranged in `DATA` in a particular order. For our example with $N = 8$ data values $f_k, k = 1 \ldots 8$, they must be in the order f_1, f_5, f_3, f_7, f_2, f_6, f_4, f_8 as shown in the binary tree on page 32 and the signal-flow graph on pages 34 and 36. This ordering is of the *indices,* and has nothing to do with the data *values,* so it would be misleading to call it sorting.

[8]The author of `four1` is identified in [10, page 500] as N. M. Brenner, but no work of his is cited in [10, page 504]. I can only speculate that he chose to use the half-angle formula because it affords some advantage in the control of roundoff error or in the size of a statically-linked program (by avoiding reference to the elementary function library routine for `COS`). I will be grateful to anyone who presents me with a convincing argument why Brenner's approach might be better for either of these or some other reason.

If the indices went from 0 to $N-1$ instead (as we originally had them in §4) each index value would be one less so the corresponding order would be f_0, f_4, f_2, f_6, f_1, f_5, f_3, f_7. These indices and their binary values are listed below in the first and second columns. The third column is the binary value with the order of its bits reversed.

```
0  000  000
4  100  001
2  010  010
6  110  011
1  001  100
5  101  101
3  011  110
7  111  111
```

When the data are arranged in ascending order according to the *bit-reversed* values of their original indices, they are in just the order we need! As explained in [10, page 499], this "is because the successive subdivisions of the data into even and odd are tests of successive low-order (least significant) bits" of N. If we can somehow arrange indices by their bit-reversed values, we can arrange data indexed from 0 to $N-1$ into the right order. How might such an arrangement of indices be accomplished? Is there some way to make it work for our data, which we decided for other reasons to index from 1 to N instead? What about arranging real-imaginary number pairs stored in consecutive elements of `DATA`? To see how the basic idea can be used in the presence of these complications we need some background in bit manipulation.

6.2.1 Bit-Reversed-Index Order

A decimal integer is represented in the computer (e.g., as a FORTRAN `INTEGER*4` variable [7, page 55]) by a string of 32 bits like this one.

s	2^{30}	$\cdots$																									$\cdots$	8	4	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1

The sign bit (marked `s`) is zero for positive values, and the other bit positions correspond to powers of 2 in the positional notation for the number, so the integer represented is $16+2+1 = 19_{10} = 10011_2$. The binary representation of 20 is the bits for $16+4 = 10100_2$. Lining these numbers up suggests an algorithm for finding the next binary value.[9] There are many methods by which a list of indices might be arranged in ascending bit-reversed-index order, but the obscure algorithm I am about to describe is fast and widely used.

```
19  10011
20  10100
```

We can change the top bitstring into the bottom one by finding its least-significant or right-most `0` (in the $4 = 2^2$ spot), changing it to a `1`, and making the bits to the right of it (they

[9] I am indebted to Dr. Daryn Ramsden for helping me to understand the algorithm implemented in [10, page 501]. The code presented below is somewhat different, but it implements the same idea and my explanation is therefore based on his.

are both `1`) into `0`s. This always works, because

$$\sum_{l=1}^{L} 2^{l-1} + 1 = 2^L$$

In our case there are $L = 2$ bits in the rightmost string of `1`s, so the identity yields $2^0 + 2^1 + 1 = 2^2 = 4$ and this is the same result we would get by addition: $\texttt{011}_2 + \texttt{1}_2 = \texttt{100}_2 = 4$.

Similarly, to find the number that comes after a given number in the order they would have if their bits were reversed (without actually reversing the bits) we can find the *most* significant or leftmost `0` in the given number, change it to a `1`, and make the `1`s to the *left* of it into `0`s. The most significant `0` of `10011` is in the 2^3 bit position. Changing that bit to a `1` and making the `1` to the left of it into `0` we get `01011` = 11. Thus, when the numbers in a list are represented by 5 bits (i.e., they range from 0 thru 31) the number that follows 19 in bit-reversed ascending order is 11.

The FFT example we studied previously had 8 data values, but it will be easier to see how the process just described can be used to rearrange data into bit-reversed-index order if we consider indices larger than 8. Suppose now that we have $N = 16$ real-imaginary pairs of values in which the individual numbers are indexed 1, 2 ... 32. Then each pair can be identified by the odd index value that names the real number of the pair, so in examining the pairs we need consider only the index values 1, 3, ..., 31. These numbers have the decimal and binary values shown below, in natural order, on the left. Because they are odd they all end with a `1` bit, which we can ignore in determining the bit-reversed ascending order of the numbers, shown on the right along with the decimal numbers represented by their unreversed bits.

1	00001	10000	1
3	00011	10001	17
5	00101	10010	9
7	00111	10011	25
9	01001	10100	5
11	01011	10101	21
13	01101	10110	13
15	01111	10111	29
17	10001	11000	3
19	10011	11001	19
21	10101	11010	11
23	10111	11011	27
25	11001	11100	7
27	11011	11101	23
29	11101	11110	15
31	11111	11111	31

If the data pairs are rearranged into the order shown on the right, their original indices will be ascending in bit-reversed order. Each step in this rearrangement can be accomplished by exchanging one data pair for another, and that puts both pairs in their right places. For example, the data pair whose real value has original index 17 or `10001` can be exchanged with the one whose real value has original index 3 or `00011`, and that puts both data pairs where they belong in the new arrangement. If we ignore the final `1` in each number of the left-hand list and reverse its remaining bits, we get the trailing bits of its partner in ascending bit-reversed order. Of course, once data pair 3 has been exchanged for data pair 17, we must not exchange 17 for 3 again later in the process.

6.2.2 A Typical Bit-Reversal Algorithm

How can we generate the sequence 1, 17, 9, . . . ? The little FORTRAN program below does the job, its output reproducing the first and last columns of the table above. This code is inspired by the bit-reversal logic of the `four1` subroutine listed in [10, page 501], revised to change some variable names and to conform with the coding style used in [7].

```
      NN=2*16
      J=1
      DO 1 I=1,NN,2
           PRINT *,I,J
           MSZ=NN/2
    3      IF((MSZ.LT.2) .OR. (J.LE.MSZ)) GO TO 2
              J=J-MSZ
              MSZ=MSZ/2
           GO TO 3
    2      J=J+MSZ
    1 CONTINUE
      STOP
      END
```

Here `NN` stores the total number of data values in the list (twice the number of data pairs), `I` is a member of the list that is arranged in increasing order (as on the left above), and `J` is the corresponding index when the numbers are arranged in ascending bit-reversed order (as on the right above). To understand how the algorithm works it is necessary to keep track of the bit pattern in the auxiliary variable `MSZ`, which means "most significant zero."

For each `I`, `MSZ` begins as `NN/2` which for our example is 16 or the bit pattern `10000`. (The bit pattern for `MSZ` will always contain a single `1` if `NN` is a power of 2). Suppose that `I` has just been incremented to 15 so that `J` contains, according to the table above, 29 (printed opposite 15 but actually found in the previous pass thru the `1` loop). In that case the test fails because `MSZ=16>2` and `J=29>MSZ=16`, and control enters the `3` loop. The subtraction `J=J-MSZ` computes `11101-10000`, removing the leftmost `1` from the bit pattern for `J`. Then `MSZ` is divided by `2`, which shifts the `1` bit it contains one place to the right yielding `01000`. Now `J` is `01101` or 13, but `MSZ` is now only 8 so the test fails again. The subtraction `01101-01000` again removes the leftmost `1` bit from `J`, leaving `J` equal to `00101` or 5, and `MSZ` is divided again yielding `00100` or 4. The test fails yet again, and the subtraction `00101-00100` removes the leading 1 again, yielding `00001` for `J`. `MSZ` gets divided again to `00010` or 2. This time the test succeeds because `J=1<MSZ=2`, so the branch is taken to statement `2` and `J` gets incremented by `MSZ` producing `00001+00010=00011` or 3 (which does indeed follow 29 in the list above). The `3` loop and the statement following it perform the task of finding the next `J` in ascending bit-reversed order.

The `3` loop found the most significant `0` in `11101`, changing the `1`s to the left of it into `0`s along the way, and then the `J=J+MSZ` changed that most significant `0` to a `1`. During the process of zeroing out the `1`s to the left of the (originally) most significant `0` in `J`, the `1` bit in `MSZ` also got shifted to the right, so `J` remained greater than `MSZ` until the final leading `1` in `J` got zeroed out. The next bit in `J` was then a `0` (the leading zero that we found), in the bit position where `MSZ` now had its `1` bit, so the test for `(J.LE.MSZ)` succeeded. When the process works that is how it stops, with `J` strictly less than `MSZ`.

6.2.3 A Little Improvement

For `I=1` we want `J=1`, because that is also the first index in ascending bit-reversed order. Each `I` and its corresponding `J` are available at the *top* of the `1` loop, so each pass computes the value of `J` that will go with the *next* `I`. For `I=NN-1` we report `J=NN-1` (computed on the previous pass thru the `1` loop) because `11111` is its own bit reversal and comes last in ascending bit-reversed order. Unfortunately the flow of control then continues on down thru the body of the `1` loop, uselessly trying to calculate the value of the next `J` even though there won't be another `I` to go along with it. This time `J` starts out as the bit string `11111`, in which there is no most significant `0` to find. `J` never gets to be less than `MSZ`, so if that were the only test then in the final pass thru the `1` loop the `3` loop would go on forever. But that is also the only time when `MSZ` gets divided down to `1`, so the `3` loop can be stopped by testing for `(MSZ.LT.2)`. Thus the `(MSZ.LT.2)` test is needed only to make the `3` loop terminate on the occasion when the `1` loop is finishing its last pass. But if the `(MSZ.LT.2)` test were missing, the test `(J.LE.MSZ)` would still succeed, because when `MSZ` reaches 1 so does J. So the `(MSZ.LT.2)` test is actually not needed so long as the other test is `(J.LE.MSZ)` rather than `(J.LT.MSZ)`. The code below produces the same output as the earlier version.

```
      NN=2*16
      J=1
      DO 1 I=1,NN,2
           PRINT *,I,J
           MSZ=NN/2
    3      IF(J.LE.MSZ) GO TO 2
              J=J-MSZ
              MSZ=MSZ/2
           GO TO 3
    2      J=J+MSZ
    1 CONTINUE
      STOP
      END
```

6.2.4 A Bigger Improvement

Why do the extra work of completing the final pass thru the `1` loop at all, when the result will just be thrown away? Instead, we can reorganize the calculation as shown in the left-hand program on the next page.

In an FFT code the indices generated by this algorithm must be used to rearrange the elements of `DATA` into bit-reversed-index order, so wherever we `PRINT *, I,J` in the program above we would actually do a swap. But because the first and last data elements start out in their bit-reversed locations they don't actually need to be moved. No swap need replace the first `PRINT`, and the `1` loop can skip the final data pair. Making these changes we get the program on the right, where comments indicate the data swapping. The test for `(J.GT.I)` is to avoid swapping any pair of elements twice.

```
  NN=2*16
  J=1
  I=1
  PRINT *,I,J
  DO 1 I=3,NN-1,2
       MSZ=NN/2
3      IF(J.LT.MSZ) GO TO 2
          J=J-MSZ
          MSZ=MSZ/2
       GO TO 3
2      J=J+MSZ
       PRINT *,I,J
1 CONTINUE
  STOP
  END
```

```
      NN=2*16
      J=1
      DO 1 I=3,NN-3,2
           MSZ=NN/2
    3      IF(J.LT.MSZ) GO TO 2
              J=J-MSZ
              MSZ=MSZ/2
           GO TO 3
    2      J=J+MSZ
C
           IF(J.GT.I) THEN
C             swap DATA(J  ) with DATA(I  )
C             swap DATA(J+1) with DATA(I+1)
           ENDIF
    1 CONTINUE
      STOP
      END
```

6.3 Other Considerations

The input rearrangement and butterfly algorithms are key to any subroutine for computing the FFT, but before we can complete such a code there are a few small details to consider.

6.3.1 Computing the Inverse Transform

The discussion of §6.2.1 and the test program listed there assume that we want to calculate the direct transform, but by making only a small change to the code we can use it to find the inverse transform as well. Recall that the direct and inverse transforms are defined like this.

$$F_n = \sum_{k=0}^{N-1} W^{nk} f_k \qquad f_k = \frac{1}{N}\sum_{n=0}^{N-1} W^{-nk} F_n = \frac{1}{N}\sum_{n=0}^{N-1} (W^{-1})^{nk} F_n.$$

These calculations are the same except for the sign of the exponent and the scaling of the inverse transform by $1/N$. The code that we settled on in §6.1.2 for calculating the real and imaginary parts of W looks like this.

```
THETA=TWOPI/DFLOAT(LMAX)
WPR=DCOS(THETA)
WPI=DSIN(THETA)
```

To change the sign of the exponent we need to instead compute

$$\begin{aligned} W_{\texttt{LMAX}}^{-1} &= \left(e^{i2\pi/\texttt{LMAX}}\right)^{-1} = e^{-i2\pi/\texttt{LMAX}} \\ &= \cos\left(-2\pi/\texttt{LMAX}\right) + i \sin\left(-2\pi/\texttt{LMAX}\right) \\ &= \cos\left(-\theta\right) + i \sin\left(-\theta\right) \end{aligned}$$

Introducing an integer variable INVDIR[10] that is set to +1 for the direct transform or -1 for the inverse, the code segment for finding W can be rewritten like this.

```
      THETA=TWOPI/DFLOAT(INVDIR*LMAX)
      WPR=DCOS(THETA)
      WPI=DSIN(THETA)
```

6.3.2 Checking for Legal Values of N

The FFT algorithm that we have considered works only if N is a positive power of 2, which means that it has exactly one 1 bit. The program below checks that property to determine whether a given N is suitable.

```
 1          INTEGER*4 MZRO/Z'80000000'/
 2 C
 3 C        get the integer to test
 4        5 READ(5,901,END=1) N
 5      901 FORMAT(Z8)
 6 C
 7 C        find the positive power of 2 that N is, or that it is not one
 8          NBITS=N
 9          DO 2 NS=1,32
10               IF(NBITS.LT.0) THEN
11                  IF(NBITS.NE.MZRO) GO TO 3
12                  IF(NS.EQ.1 .OR. NS.EQ.32) GO TO 3
13                  LGN=32-NS
14                  GO TO 4
15               ENDIF
16 C
17 C             left bit is not a 1 yet; shift left
18               NBITS=2*NBITS
19        2 CONTINUE
20        3 WRITE(6,902) N
21      902 FORMAT(I11,' is not a positive power of +2')
22          GO TO 5
23 C
24 C        report the power of 2 that N is
25        4 WRITE(6,903) N,LGN
26      903 FORMAT(I11,' = 2^[',I2,']')
27          GO TO 5
28        1 STOP
29          END
```

The 2 loop uses integer multiplication $\boxed{18}$ to shift the bits of NBITS left one bit at a time until $\boxed{10}$ the leading 1 is in the sign position; then it checks $\boxed{11}$ whether all the other bits are zeros. If $\boxed{12}$ that happens before the first shift, then N was negative and illegal; if it happens before the last shift, then N was 1 and illegal. If it happens after some intermediate number of shifts, then $\boxed{13}$ we get $\log_2(N)$, called LGN in the program, for free. If it *never* happens (control falls through the bottom of the loop) then N was zero and illegal.

If an FFT routine is called repeatedly with the same value of N it is of course necessary to check that N is legal only on the *first* call. In the test program of §6.2.1 the 5 loop is

[10]The four1 routine [10, page 501] uses isign, but that is the name of a FORTRAN built-in function.

free and exits based on a test of LMAX. If we know LGN it should be possible to make this a bounded loop [7, §13.3.5] that explicitly steps through the stages of the signal-flow graph.

6.3.3 Output Ordering and Shift

Recall from §4 that because F_n is periodic in n with period N, we can if we wish think of some F_n as corresponding to negative frequencies. There we assumed that n runs from 0 to $N-1$, but ever since §5 we have numbered the results F_1 through F_N instead. This makes it no longer true that $F_{-n} = F_{N-n}$, because the modulus arithmetic works only if zero is allowed as a remainder of integer division. However, if $N = 8$ we can still think of the F_n as being in order for either sequence of ν values listed below.

$$\begin{array}{cccccccc}
F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 \\
0 & \dfrac{+1}{8\Delta} & \dfrac{+2}{8\Delta} & \dfrac{+3}{8\Delta} & \dfrac{+4}{8\Delta} & \dfrac{+5}{8\Delta} & \dfrac{+6}{8\Delta} & \dfrac{+7}{8\Delta} \\
0 & \dfrac{+1}{8\Delta} & \dfrac{+2}{8\Delta} & \dfrac{+3}{8\Delta} & \dfrac{\pm 4}{8\Delta} & \dfrac{-3}{8\Delta} & \dfrac{-2}{8\Delta} & \dfrac{-1}{8\Delta} \\
F_1 & F_2 & F_3 & F_4 & F_{\pm 5} & F_{-4} & F_{-3} & F_{-2}
\end{array}$$

If we think of the F_n as having been delivered in the order shown at the bottom above, we might like to rearrange them in increasing order of frequency like this.

$$\begin{array}{ccccccccc}
\dfrac{-4}{8\Delta} & \dfrac{-3}{8\Delta} & \dfrac{-2}{8\Delta} & \dfrac{-1}{8\Delta} & 0 & \dfrac{+1}{8\Delta} & \dfrac{+2}{8\Delta} & \dfrac{+3}{8\Delta} & \dfrac{+4}{8\Delta} \\
F_5 & F_6 & F_7 & F_8 & F_1 & F_2 & F_3 & F_4 & F_5
\end{array}$$

This operation is called a **shift**, because it amounts to a circular shift left by $N/2$ elements. If we start with the top list of F_n and exchange F_1 for F_5, they will both be in the right places for the bottom list; similarly for F_2 and F_6, for F_3 and F_7, and for F_4 and F_8. The 1 loop in the code below performs these four exchanges on a vector of integers to illustrate the idea. Because F_5 appears twice, the result vector has $N + 1$ elements and the code copies F_5 into the $N + 1$st location before doing the exchanges.

```
      PARAMETER(NN=8)
      INTEGER*4 FF(NN+1)/1,2,3,4,5,6,7,8,0/,TEMP
      WRITE(6,901) FF
  901 FORMAT(9I2)
      FF(NN+1)=FF(1+NN/2)
      DO 1 N=1,NN/2
           TEMP=FF(N)
           FF(N)=FF(N+NN/2)
           FF(N+NN/2)=TEMP
    1 CONTINUE
      WRITE(6,901) FF
      STOP
      END
```

```
 1 2 3 4 5 6 7 8 0
 5 6 7 8 1 2 3 4 5
```

When it is compiled and run the program produces the output on the right. In an FFT routine the results will of course be real-imaginary pairs rather than integers.

Our algorithm will do the arithmetic to invert a direct transform that has been shifted (ignoring its repeated $N+1$st point) but it produces a result different from the inverse of the unshifted forward transform. In our $N=8$ example, with the F_n rearranged by shifting we get for f_2 the sum on top instead of the usual DFT sum on the bottom.

$$\begin{aligned}Nf_2 &= W^0F_5+W^{-1}F_6+W^{-2}F_7+W^{-3}F_8+W^{-4}F_1+W^{-5}F_2+W^{-6}F_3+W^{-7}F_4\\ Nf_2 &= W^0F_1+W^{-1}F_2+W^{-2}F_3+W^{-3}F_4+W^{-4}F_5+W^{-5}F_6+W^{-6}F_7+W^{-7}F_8\end{aligned}$$

The results would be equal if the power of W multiplying each F_n in one sum evaluated to the same number as the power of W multiplying the same F_n in the other. However, we find

$$\begin{aligned}W^{-4} &= e^{-4i2\pi/8}=e^{-i\pi}=e^{i(-\pi)}=\cos(-\pi)+i\sin(-\pi)=-W^0\\ W^{-5} &= e^{-5i2\pi/8}=e^{-i2\pi/8}e^{-i\pi}=-W^{-1}\\ W^{-6} &= e^{-6i2\pi/8}=e^{-2i2\pi/8}e^{-i\pi}=-W^{-2}\\ W^{-7} &= e^{-7i2\pi/8}=e^{-3i2\pi/8}e^{-i\pi}=-W^{-3}\end{aligned}$$

so f_2 comes out with the right magnitude but the wrong sign. On the other hand, when we compare the shifted and unshifted sums for f_3,

$$\begin{aligned}Nf_3 &= W^0F_5+W^{-2}F_6+W^{-4}F_7+W^{-6}F_8+W^{-8}F_1+W^{-10}F_2+W^{-12}F_3+W^{-14}F_4\\ Nf_3 &= W^0F_1+W^{-2}F_2+W^{-4}F_3+W^{-6}F_4+W^{-8}F_5+W^{-10}F_6+W^{-12}F_7+W^{-14}F_8\end{aligned}$$

we find that

$$\begin{aligned}W^{-8} &= e^{-8i2\pi/8}=e^{-i2\pi}=e^{i(-2\pi)}=\cos(-2\pi)+i\sin(-2\pi)=W^0\\ W^{-10} &= e^{-10i2\pi/8}=e^{-2i2\pi/8}=W^{-2}\\ W^{-12} &= e^{-12i2\pi/8}=e^{-3i2\pi/8}=W^{-3}\\ W^{-14} &= e^{-14i2\pi/8}=e^{-4i2\pi/8}=W^{-4}\end{aligned}$$

so f_3 comes out right. For an N-point transform, the shifted and unshifted DFT sums for f_k are

$$\begin{aligned}Nf_k &= W^0F_{N/2+1}+W^{-(k-1)}F_{N/2+2}+W^{-2(k-1)}F_{N/2+3}\cdots+W^{-(N/2-1)(k-1)}F_N+W^{-(N/2)(k-1)}F_1\cdots\\ Nf_k &= W^0F_1+W^{-(k-1)}F_2+W^{-2(k-1)}F_3\cdots+W^{-(N/2-1)(k-1)}F_{N/2}+W^{-(N/2)(k-1)}F_{N/2+1}\cdots\end{aligned}$$

so for the coefficients to agree we need

$$\begin{aligned}W^{-(N/2+n-1)(k-1)} &= W^{-(n-1)(k-1)}\\ W^{-(N/2)(k-1)}W^{-(n-1)(k-1)} &= W^{-(n-1)(k-1)}\\ W^{-(N/2)(k-1)} &= 1\\ e^{-(N/2)(k-1)i2\pi/N} &= 1\\ e^{-i(k-1)\pi} &= 1\end{aligned}$$

But $e^{i\theta}=\cos(\theta)+i\sin(\theta)=1$ when θ is an even multiple of π, so if $k-1$ is even the sums agree and if it is odd they differ by a sign.

Thus we can invert a direct transform that was shifted by treating it as though it were not and then changing the sign of each even-indexed result f_k.

6.3.4 Scaling, and Limits of Integration

As discussed in §6.3.1 the signal-flow graph algorithm computes, if we make the proper choice of exponent sign, the sum for either the direct or the inverse DFT. The direct DFT is the sum when the sign is positive, but the inverse DFT is the sum when the sign is negative, *multiplied by* $1/N$. It would be counterintuitive for any routine that computes the DFT to omit this scaling, because then using the routine to transform a sequence and invert the result would *not* recover the starting sequence. So this scaling should always be done.

Of course the DFT is not the Fourier transform, either. As discussed in §4.1, it is the DFT *multiplied by* Δ_x that approximates the continuous Fourier transform $F(\nu)$.

Thus, if the direct transform is being computed, we should be allowed to either scale the result by Δ_x or leave it alone. If the inverse transform is being computed, then if the input direct transform was scaled we must scale the inverse by $1/(N\Delta_x) = \Delta_\nu$ and if the direct transform was not scaled we must scale the inverse transform by $1/N$.

The limits of integration used in the analysis of §4 do not enter into the numerical process of finding the sum for the forward or the inverse DFT, but they are needed for finding Δ_x and Δ_ν. In a more philosophical vein, the limits of integration determine the values of the ν_n that go along with the F_n and of the x_k that go along with the f_k, and that information is essential for interpreting the transform and its inverse. An FFT routine that permits scaling must provide for the limits of integration to be supplied along with the f_k or F_n input data, and must replace limits on x with the corresponding limits on ν in producing the forward transform or the limits on ν with the corresponding limits on x in producing the inverse.

We have assumed that the lower limit of integration for the forward transform is always zero, so only the upper limit of integration b need be supplied. The lower limit on the inverse transform is also zero if there is no shift, and is equal to minus the upper limit if there is a shift. Providing only b is thus sufficient in both directions.

If x goes from 0 to b and there are N points, then $\Delta_x = b/(N-1)$. The frequency range of the forward transform is from 0 to $(N-1)/(N\Delta_x)$ if unshifted, or from $-(N/2)/(N\Delta_x) = -1/(2\Delta_x)$ to $+(N/2)/(N\Delta_x) = +1/(2\Delta_x)$ if shifted.

If ν goes from $a = 0$ to $b = (N-1)/(N\Delta_x)$, then there are N points and

$$\Delta_\nu = \frac{b}{N-1} = \frac{(N-1)/(N\Delta_x)}{N-1} = \frac{1}{N\Delta_x}$$

so the x range of the inverse transform is from 0 to $(N-1)/(N\Delta_\nu) = (N-1)\Delta_x$. If ν goes from $-b = -1/(2\Delta_x)$ to $b = +1/(2\Delta_x)$, then there are $N+1$ points and

$$\Delta_\nu = \frac{b-(-b)}{N} = \frac{+1/(2\Delta_x) - (-1/(2\Delta_x))}{N} = \frac{1}{N\Delta_x}$$

once again, so the x range of the inverse transform is still from 0 to $(N-1)/(N\Delta_\nu)$.

6.3.5 User Interface

Because a primary motivation for using FFT in the first place is speed, a case can be made for sacrificing safety and convenience to save computer time. This argues for trusting the user to give N a sensible value, to scale the inverse transform by $1/N$, to scale the forward transform by Δ_x if desired, and to frequency-shift the forward transform if desired.[11] It is not always necessary to do the scalings at all, and scaling by Δ enlarges the parameter list (to pass the upper limit of integration b). Checking the parameters inside a routine adds code and takes cycles. Especially when FFT calls must be iterated in transforming data with more than one dimension, as discussed in §7, it should be possible to avoid repetitive or unnecessary work. This is true even though the convenience code accounts for only a small fraction of the cycles consumed when a problem is big enough so that execution time is a serious consideration. Scaling and shifting could be implemented in separate routines, rather than in the same routine that computes the DFT sums.

On the other hand, it is expecting quite a lot for simpleton users (such as me) to get subroutine parameters right on the first try, to code transform scalings outside the FFT routine whenever they are necessary, and to remember how to shift the output if that is desired. A library subprogram is supposed to encapsulate arcane technical details of that kind so the user can instead focus on solving the application problem. Implementing closely-related functionalities in separate routines increases cognitive load, coding effort, and the chance of making a mistake when they are all needed, and a routine for shifting and scaling could never justify its existence apart from being used with the DFT-sum calculator.

Given all of these considerations, I have included code for parameter checking, scaling, and shifting in the (single) subroutine of the next Section. However, as a concession to efficiency the routine checks that `N` is legal only once, and it lets the user specify with parameter values whether the transform is to be scaled or shifted. This design results in the calling sequence given on the next page. The parameters that come first are inputs, and do not get changed by the routine; the parameters in the middle are both inputs and outputs; and the return code is an output whose input value is ignored.

A main program for exercising `FFT` is given on page 54, which you can modify to suit your needs (but please read the **disclaimers and permissions** tab of the website from which you downloaded this paper[12]). The `FUNCTION` subprogram `FOFX`, which returns $f(x)$ for our triangular pulse, was listed in §4. The `PROMPT` and `QUERY` routines are given in [7, §10].

[11] In [10, page 500,501] all of these tasks are assigned to the user.

[12] You should also be aware that although the source code of `four1` is listed in [10] it is copyrighted and can be used legally only under license; see pages xvi-xvii of that book for stern legalese stating conditions and prices. Although I got ideas from [10], as mentioned repeatedly above, I think the code presented here is different enough not to infringe their copyright. The key algorithms in `four1`, for bit-reversed-index rearrangement and the butterfly calculation, are not original with [10] either; see [9, page 331] for one example. Many authors seem to have used the bit-reversed-index rearrangement, and as mentioned above there are few ways to code the butterfly calculation. A source-code alternative to `four1` is `FFTPACK`, which is available free from the `netlib` repository and is apparently public-domain, but it is uncommented and comes in many pieces so it is even more difficult to understand than `four1`. MATLAB is proprietary like `four1`, but its source code is not available even under license.

```
CALL FFT(N,INVDIR,SCALE,SHIFT, DATA,B, RC)
```

`N`	`INTEGER*4` number of samples, equal to a positive power of 2 shifting adds an `(N+1)`st point but leaves `N` unchanged
`INVDIR`	`INTEGER*4` flag: `+1` $\Rightarrow$ direct transform, `-1` $\Rightarrow$ inverse transform
`SCALE`	`LOGICAL*4` flag: `.FALSE.` $\Rightarrow$ no scaling by Δ; `.TRUE.` $\Rightarrow$ if `INVDIR=+1` scale F_n by Δ_x to approximate $F(\nu)$ if `INVDIR=-1` assume the input F_n values were scaled by Δ_x
`SHIFT`	`LOGICAL*4` flag: `.FALSE.` $\Rightarrow$ no frequency shift; `.TRUE.` $\Rightarrow$ if `INVDIR=+1` rearrange F_n to center spectrum on $\nu = 0$ and set `DATA(N+1)` if `INVDIR=-1` assume the input F_n values were rearranged but ignore `DATA(N+1)`
`DATA`	`COMPLEX*16` vector of length `N` (`N+1` if `SHIFT=.TRUE.`) on input, `N` values of f_k or `N` (`N+1` for `SHIFT=.TRUE.`) values of F_n on output, `N` (`N+1` for `SHIFT=.TRUE.`) values of F_n or `N` values of f_k
`B`	`REAL*8` scalar, the upper limit of integration if `INVDIR=+1`, `B` is a value of x on input, a value of ν on output if `SHIFT=.FALSE.` the spectrum goes from 0 to `B` if `SHIFT=.TRUE.` the spectrum goes from `-B` to `B` if `INVDIR=-1`, `B` is a value of ν on input, a value of x on output if `SCALE=.FALSE.`, `B` is neither used nor changed
`RC`	`INTEGER*4` return code; 0 if all went well, otherwise the sum of 1 $\Rightarrow$ `N` is not a positive integer power of 2 2 $\Rightarrow$ `INVDIR` is not `+1` or `-1` 4 $\Rightarrow$ `B` ≤ 0 if a parameter error occurs, `DATA` and `B` are left unchanged

```
C     This program exercises FFT.
C
      REAL*8 B,DELTA,FOFX,NU,X
      COMPLEX*16 DATA(257)
      INTEGER*4 RC
      LOGICAL*4 SCALE,SHIFT,QUERY
C
C ------------------------------------------------------------------
C
C     find out how many points to use
      CALL PROMPT('N=',2)
      READ *,N
      SCALE=QUERY('scale?',6)
      SHIFT=QUERY('shift?',6)
C
C     sample f(x)
      B=3.5D0
      DELTA=B/DFLOAT(N-1)
      DO 1 K=1,N
           X=DELTA*DFLOAT(K-1)
           DATA(K)=DCMPLX(FOFX(X),0.D0)
    1 CONTINUE
C
C     transform the input waveform
      INVDIR=+1
      CALL FFT(N,INVDIR,SCALE,SHIFT, DATA,B, RC)
      IF(SCALE) WRITE(6,901) B
  901 FORMAT('B=',1PE13.6)
C
C     report the transform
      IF(.NOT.SHIFT) NLIM=N
      IF(     SHIFT) NLIM=N+1
      DO 2 K=1,NLIM
           IF(.NOT.SHIFT) NU=DFLOAT(K-1)*B/DFLOAT(NLIM-1)
           IF(     SHIFT) NU=-B+DFLOAT(K-1)*2.D0*B/DFLOAT(NLIM-1)
           WRITE(1,902) NU,DREAL(DATA(K))
           WRITE(2,902) NU,DIMAG(DATA(K))
  902      FORMAT(2(1X,1PE13.6))
    2 CONTINUE
C
C     inverse transform the result
      INVDIR=-1
      CALL FFT(N,INVDIR,SCALE,SHIFT, DATA,B, RC)
      IF(SCALE) WRITE(6,901) B
C
C     report the recovered input waveform
      DO 3 K=1,N
           X=DFLOAT(K-1)*B/DFLOAT(N-1)
           WRITE(3,902) X,DREAL(DATA(K))
           WRITE(4,902) X,DIMAG(DATA(K))
    3 CONTINUE
      STOP
      END
```

6.4 A Library Routine

My implementation of the FFT algorithm is listed in the following pages. The routine begins with a standard preamble [7, §12.3.2]. `DATA` is `REAL*8` so its dimension is assumed twice that of the corresponding `COMPLEX*16` vector in the calling routine. `NPREV`, the value of `N` on the previous call, is initialized at compile time $\boxed{55}$ to the bit pattern for `-0` (if `N=-0` on input `NS=1` $\boxed{68}$ so the routine returns `RC=1`) and gets updated $\boxed{76}$ each time `N` changes between calls. `MZRO` has the same bit pattern for testing `N`, and it does not change.

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE FFT(N,INVDIR,SCALE,SHIFT, DATA,B, RC)
C     This routine computes in-place the direct or inverse fast
C     Fourier transform of the sequence in DATA.
C
C     variable  meaning
C     --------  -------
C     B         upper limit of integral for transform or inverse
C     DATA      input sequence, then its transform or inverse
C     DCOS      Fortran function returns cosine of REAL*8
C     DELTA     sampling interval or frequency interval
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     DSIN      Fortran function returns sine of a REAL*8
C     I         index on DATA elements
C     INVDIR    +1 => direct transform, -1 => inverse transform
C     J         second index on DATA elements for rearrangement
C     L         index on powers of W
C     LGN       log_2(N) = number of stages in signal flow graph
C     LMAX      size of the transforms in this stage
C     M         index on stages in the signal flow graph
C     MSZ       bit position of most significant 0 in an index value
C     MZRO      bit pattern 10000000000000000000000000000000
C     N         number of input real-imaginary data pairs
C     NBITS     N bit pattern shifted left
C     NN        number of input data items = 2*N
C     NPREV     previous value of N
C     NS        number of shifts to put N 1 bit in sign position
C     P         odd index of data pair at top of a butterfly
C     PSTEP     DATA elements between butterflies using same W power
C     Q         odd index of data pair at bottom of a butterfly
C     RC        return code; 0 => parameters make sense
C     SCALE     T => scale output or assume that input is scaled
C     SF        scale factor
C     SHIFT     T => frequency-shift result
C     TEMPI     temporary storage for real part of a DATA element
C     TEMPR     temporary storage for imag part of a DATA element
C     THETA     angle in W
C     TWOPI     twice the circle constant
C     WI        imaginary part of W
C     WPI       imaginary part of W for this stage
C     WPR       real part of W for this stage
C     WR        real part of W
C     WTEMP     temporary in finding next power of W
C
C     input parameters
      LOGICAL*4 SCALE,SHIFT
      REAL*8 DATA(*),B
      INTEGER*4 RC
C
C     local variables
      REAL*8 DELTA,SF,TEMPI,TEMPR,THETA,WI,WPI,WPR,WR,WTEMP
      REAL*8 TWOPI/6.2831853071795865D0/
      INTEGER*4 MZRO/Z'80000000'/,NPREV/Z'80000000'/,P,PSTEP,Q
C
C ------------------------------------------------------------------
```

```
 58 C
 59 C     sanity-check the input parameters
 60       RC=0
 61 C       the number of points must be a positive power of 2
 62         IF(N.NE.NPREV) THEN
 63 C          this is a new N; check it and find its lg
 64            NBITS=N
 65            DO 2 NS=1,32
 66                 IF(NBITS.LT.0) THEN
 67                    IF(NBITS.NE.MZRO) GO TO 3
 68                    IF(NS.EQ.1 .OR. NS.EQ.32) GO TO 3
 69                    LGN=32-NS
 70                    GO TO 4
 71                 ENDIF
 72                 NBITS=2*NBITS
 73     2      CONTINUE
 74     3      RC=1
 75     4      IF(RC.EQ.0) THEN
 76               NPREV=N
 77            ELSE
 78               NPREV=MZRO
 79            ENDIF
 80         ENDIF
 81 C
 82 C       the transform flag must denote either direct or inverse
 83         IF(INVDIR.NE.+1 .AND. INVDIR.NE.-1) RC=RC+2
 84 C
 85 C       the upper limit of integration was assumed positive
 86         IF(B.LE.0.D0) RC=RC+4
 87       IF(RC.NE.0) RETURN
 88       NN=2*N
 89 C
 90 C     arrange the input sequence by ascending bit-reversed index
 91       J=1
 92       DO 5 I=3,NN-3,2
 93            MSZ=N
 94     7      IF(J.LT.MSZ) GO TO 6
 95               J=J-MSZ
 96               MSZ=MSZ/2
 97            GO TO 7
 98     6      J=J+MSZ
 99            IF(J.GT.I) THEN
100               TEMPR=DATA(J)
101               TEMPI=DATA(J+1)
102               DATA(J)=DATA(I)
103               DATA(J+1)=DATA(I+1)
104               DATA(I)=TEMPR
105               DATA(I+1)=TEMPI
106            ENDIF
107     5 CONTINUE
```

The test whether N is a positive power of 2 proceeds as described in §6.3.2, and if it fails $\boxed{74}$ RC gets set to 1. In that case $\boxed{78}$ NPREV is reset to MZRO so that the test will be performed again on the next call. If N passes the test, LGN saves $\log_2$(N) $\boxed{69}$ and NPREV remembers N $\boxed{76}$ so that the test can be skipped on the next call. INVDIR $\boxed{83}$ and B $\boxed{86}$ are checked for sensible values on each call. If any of the tests fail, RC is nonzero and the routine returns

$\boxed{87}$ without doing any calculations. Otherwise $\boxed{88}$ NN is set to the total number of values, real and imaginary, in DATA. (If this call is to invert and the input sequence was shifted, it actually contains N+1 complex F_n values, so there are NN+2 numbers in DATA.)

Next $\boxed{90\text{-}107}$, the input data are rearranged using the algorithm of §6.2.4. Both real and imaginary parts must be swapped $\boxed{100\text{-}105}$.

```
108 C
109 C     use the butterfly algorithm to evaluate the signal flow graph
110       LMAX=2
111       DO 8 M=1,LGN
112 C          set W^0=1
113            WR=1.D0
114            WI=0.D0
115 C
116 C          compute the W for this stage
117            THETA=TWOPI/DFLOAT(INVDIR*LMAX)
118            WPR=DCOS(THETA)
119            WPI=DSIN(THETA)
120 C
121 C          consider each power of W used in this stage
122            PSTEP=2*LMAX
123            DO 9 L=1,LMAX,2
124 C               do the butterflies with this power of W in all groups
125                 DO 10 P=L,NN,PSTEP
126                       Q=P+LMAX
127                       TEMPR=WR*DATA(Q)-WI*DATA(Q+1)
128                       TEMPI=WR*DATA(Q+1)+WI*DATA(Q)
129                       DATA(Q)=DATA(P)-TEMPR
130                       DATA(Q+1)=DATA(P+1)-TEMPI
131                       DATA(P)=DATA(P)+TEMPR
132                       DATA(P+1)=DATA(P+1)+TEMPI
133    10           CONTINUE
134 C
135 C               find the next power of W as W*WP
136                 WTEMP=WR
137                 WR=WR*WPR-WI*WPI
138                 WI=WI*WPR+WTEMP*WPI
139     9      CONTINUE
140            LMAX=PSTEP
141     8 CONTINUE
142 C
143 C     find sampling interval and new upper limit of integration
144       IF(SCALE) THEN
145          IF(INVDIR.EQ.+1) THEN
146             DELTA=B/DFLOAT(N-1)
147             IF(     SHIFT) B=DFLOAT(N/2)/(DELTA*DFLOAT(N))
148             IF(.NOT.SHIFT) B=DFLOAT(N-1)/(DELTA*DFLOAT(N))
149          ELSE
150             IF(     SHIFT) DELTA=2.D0*B/DFLOAT(N)
151             IF(.NOT.SHIFT) DELTA=B/DFLOAT(N-1)
152             B=DFLOAT(N-1)/(DELTA*DFLOAT(N))
153          ENDIF
154          SF=DELTA
155       ELSE
156          IF(INVDIR.EQ.+1) SF=1.D0
157          IF(INVDIR.EQ.-1) SF=1.D0/DFLOAT(N)
158       ENDIF
```

With the data in ascending bit-reversed-index order the routine performs $\boxed{\texttt{109-141}}$ the butterfly calculations described in §6.1.1, improved as described in §6.1.2. Since the number of stages, $\log_2(N)$, is available $\boxed{\texttt{69}}$ in the variable `LGN`, the `8` loop over stages is now a `DO` loop.

The next stanza $\boxed{\texttt{143-158}}$ finds the scale factor `SF` by which the butterfly result must be multiplied. If scaling has not been selected $\boxed{\texttt{156-157}}$ this is 1 for the forward transform or $1/N$ for the inverse; in that case the upper limit of integration `B` is ignored and `DELTA`, the sampling interval in x or ν, is not found. If scaling has been selected $\boxed{\texttt{145-154}}$ `DELTA` and `B` are calculated as discussed in §6.3.4, and `SF` is set $\boxed{\texttt{154}}$ to `DELTA` so that the result will be multiplied by Δ_x if it is the forward transform or by Δ_ν if it is the inverse.

```
159 C
160 C     scale the output values
161       IF(SF.NE.1.D0) THEN
162          DO 11 I=1,NN,2
163                DATA(I)=SF*DATA(I)
164                DATA(I+1)=SF*DATA(I+1)
165    11    CONTINUE
166       ENDIF
167 C
168 C     shift, or compensate for having shifted before
169       IF(SHIFT) THEN
170          IF(INVDIR.EQ.1) THEN
171 C           frequency-shift the forward transform
172             DATA(NN+1)=DATA(N+1)
173             DATA(NN+2)=DATA(N+2)
174             DO 12 I=1,N,2
175                   TEMPR=DATA(I)
176                   TEMPI=DATA(I+1)
177                   DATA(I)=DATA(I+N)
178                   DATA(I+1)=DATA(I+1+N)
179                   DATA(I+N)=TEMPR
180                   DATA(I+1+N)=TEMPI
181    12       CONTINUE
182          ELSE
183 C           fix up the inverse of the frequency-shifted transform
184             DO 13 I=3,NN-1,4
185                   DATA(I)=-DATA(I)
186                   DATA(I+1)=-DATA(I+1)
187    13       CONTINUE
188          ENDIF
189       ENDIF
190       RETURN
191       END
```

If the scale factor is 1 $\boxed{\texttt{161}}$, either because we found an unscaled forward transform or because `DELTA` turned out to be 1, the scaling loop is skipped. If scaling is required, the `11` loop $\boxed{\texttt{162-165}}$ multiplies the real and imaginary parts of each result by the `SF` calculated earlier.

Finally, if the forward transform just found is to be frequency-shifted or if the forward transform just inverted began as frequency-shifted, result elements are rearranged $\boxed{\texttt{171-181}}$ or changed in sign $\boxed{\texttt{183-187}}$ as described in §6.3.3.

6.5 FFT Performance

On the first call with a correct value of `N`, finding what power of 2 it is takes in the worst case 31 tests $\boxed{\texttt{66}}$, 31 integer multiplications $\boxed{\texttt{72}}$, 3 additional tests $\boxed{\texttt{67,68,75}}$, 1 integer subtract $\boxed{\texttt{69}}$, 2 assignments $\boxed{\texttt{64,76}}$, and the overhead of the `2` loop. After the first call with a given value of `N`, the run-time penalty for testing correct input parameters amounts to 1 assignment $\boxed{\texttt{60}}$ and 4 tests $\boxed{\texttt{62,83,86,87}}$, each performed once per call. If `SCALE=.FALSE.`, the run-time penalty for having the scaling capability built into the code amounts to two tests $\boxed{\texttt{144,161}}$ each performed once per call. If `SHIFT=.FALSE.`, the run-time penalty for having the shift capability built into the code amounts to a single test $\boxed{\texttt{169}}$ performed once per call. For problems of even modest size all of this overhead is negligible compared to the work of computing the transform. For example, when `N=256` we use 120 swaps $\boxed{\texttt{100-105}}$ (720 assignments) to rearrange the data and 1024 butterfly calculations $\boxed{\texttt{111-141}}$ (4128 floating-point multiplies, 6168 floating-point adds or subtracts, 8 invocations of `DSIN` and `DCOS`, some integer arithmetic, and the overhead for several loops). To measure the CPU time spent on parameter checking, I used the program om the next page

The first set of initializations $\boxed{\texttt{3-5}}$ are for the parameter-checking process $\boxed{\texttt{19-46}}$, which is copied verbatim from `FFT` $\boxed{\texttt{59-86}}$. The loop over powers of 2 $\boxed{\texttt{16-17,47}}$ exercises the parameter-checking code 31 times for different values of `N`. The remainder of the code performs the CPU timing and reports the result.

`TIMER` is the CPU timing routine described in [7, §18.5.4]. It is enabled by setting `TIMING=.TRUE.` $\boxed{\texttt{7,8,13}}$ and making an initialization call $\boxed{\texttt{14}}$. Then CPU time is directed to timing bin `1` $\boxed{\texttt{15}}$ and the algorithm is performed repeatedly by the `1` loop. At the conclusion of the code segment under study $\boxed{\texttt{48}}$ time is redirected to the bin for non-algorithm time and $\boxed{\texttt{49}}$ the timing process is concluded. Then the CPU time in bin `1` is written from `BINCPU` $\boxed{\texttt{7,9,50-51}}$. This time erroneously includes the overhead of the `1` loop, but that is small enough to neglect.

`TIMER` works by reading the cycle counter of an Intel Pentium processor, which has a resolution of one clock cycle, but the time it measures is contaminated to some extent by the cycle consumption of concurrently-executing processes (such as UNIX™ background processes[13]) and this introduces random variations in the measurements. Long calculations tend to monopolize the processor, which reduces the importance of this noise relative to the CPU time being measured, but the parameter-checking code sequence is fast so the cycle counts for it vary quite a bit. Running the program 50 times with the processor clocking 798 MHz yielded the measurements plotted in the following histogram.

```
1.96E+04 .
2.76E+04 ......................................
3.56E+04 ..........
4.36E+04 .
5.16E+04
```

[13] The experiments reported here were conducted on an IBM T43p laptop computer running Ubuntu Linux, and the programs were compiled without optimization using the `gcc-4.3` version of `gfortran` with the following options: `-g -ffixed-form -fdollar-ok -std=legacy -Wno-tabs -fno-automatic -fbounds-check -fno-range-check`.

```
C     This program times the checking of input parameters.
C
      INTEGER*4 MZRO/Z'80000000'/,NPREV/Z'80000000'/
      INTEGER*4 RC,INVDIR/1/,P
      REAL*8 B/3.5D0/
C
      COMMON /EXPT/ TIMING,NONALG,NBIN,TOPBIN,NTMEAS,TOH,BINCPU
      LOGICAL*4 TIMING
      INTEGER*4 TOPBIN,TOH,BINCPU(2,22)
C
C -----------------------------------------------------------------
C
      TIMING=.TRUE.
      CALL TIMER(-1,1)
      CALL TIMER(1,2)
      DO 1 P=0,30
      N=2**P
C
C     sanity-check the input parameters
      RC=0
C       the number of points must be a positive power of 2
        IF(N.NE.NPREV) THEN
C          this is a new N; check it and find its lg
           NBITS=N
           DO 2 NS=1,32
                IF(NBITS.LT.0) THEN
                   IF(NBITS.NE.MZRO) GO TO 3
                   IF(NS.EQ.1 .OR. NS.EQ.32) GO TO 3
                   LGN=32-NS
                   GO TO 4
                ENDIF
                NBITS=2*NBITS
    2      CONTINUE
    3      RC=1
    4      IF(RC.EQ.0) THEN
              NPREV=N
           ELSE
              NPREV=MZRO
           ENDIF
        ENDIF
C
C       the transform flag must denote either direct or inverse
        IF(INVDIR.NE.+1 .AND. INVDIR.NE.-1) RC=RC+2
C
C       the upper limit of integration was assumed positive
        IF(B.LE.0.D0) RC=RC+4
    1 CONTINUE
      CALL TIMER(21,3)
      CALL TIMER(0,4)
      WRITE(6,901) BINCPU(1,1),BINCPU(2,1)
  901 FORMAT('CPU time = ',I1,' seconds and ',I10,' nanoseconds')
      STOP
      END
```

Throwing out the smallest and largest observation, the remaining CPU time estimates for the 31 repetitions ranged from 28823 ns to 37529 ns, with a median value of 34962 ns. To check the parameters once (such as on the first call to FFT with a given `N`) thus takes about $34962/31 \approx 1128$ nanoseconds or roughly one microsecond on an 800 MHz computer.

To assess the performance of the whole FFT implementation described in §6.4, I first wrote a DFT implementation having a similar calling sequence. That library subroutine, named `DFT`, is listed on the next two pages. In this code `NN` $\boxed{\texttt{22}}$ corresponds to the number of samples N in our analysis so that `N` $\boxed{\texttt{21}}$ can represent the index n. The samples and results are indexed internally starting with `0` $\boxed{\texttt{33,38}}$ in order to simplify the expression of the DFT algorithm, and their indices go up to `NN` rather than `NN-1` to allow (as in `FFT`) for the repeated result that is returned when the output is shifted. This code uses the built-in ability of FORTRAN to do complex arithmetic, so the data arrays and several other variables are declared `COMPLEX*16`, and the variable `I` $\boxed{\texttt{18,41}}$ is used for $\sqrt{-1}$.

The executable code begins by sanity-checking input parameters $\boxed{\texttt{46-56}}$ but instead of insisting that N be a power of 2 this routine only demands $\boxed{\texttt{49}}$ that it be greater than 1. Next it copies the input sequence from `DATA` to the vector `F` $\boxed{\texttt{37-38,59-61}}$ because the naïve DFT algorithm does not work in-place. Then $\boxed{\texttt{64}}$ it finds $W = e^{i2\pi/N}$, making the sign of the exponent negative if `INVDIR=-1` (we used the same approach in FFT $\boxed{\texttt{117}}$ on page 58).

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE DFT(NN,INVDIR,SCALE,SHIFT, DATA,B, RC)
C     This routine returns in-place the direct or inverse discrete
C     Fourier transform of the sequence in DATA.
C
C     variable  meaning
C     --------  -------
C     B         upper limit of integral for transform or inverse
C     CDEXP     Fortran function returns exp(COMPLEX*16)
C     DATA      input sequence, then its transform or inverse
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     DELTA     sampling interval
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     F         the input f_k's as a COMPLEX*16s
C     FF        a point F_n in the transform
C     I         i, the square root of -1
C     INVDIR    +1 => direct transform, -1 => inverse transform
C     K         index on input values
C     N         index on transform values
C     NN        number of samples
C     PI        the circle constant
C     RC        return code; 0 => parameters make sense
C     SCALE     T => scale output or assume that input is scaled
C     SF        scale factor
C     SHIFT     T => frequency-shift result
C     TEMP      temporary scalar used in shifting result
C     W         exp(i*2*pi/N)
C
C     formal parameters
      LOGICAL*4 SCALE,SHIFT
      COMPLEX*16 DATA(0:NN)
      REAL*8 B
      INTEGER*4 RC
C
C     automatic workspace
      COMPLEX*16 F(0:NN)
C
C     other local variables
      COMPLEX*16 FF,I/(0.D0,1.D0)/,W,TEMP
      REAL*8 DELTA,PI/3.1415926535897932D0/,SF
C
C -----------------------------------------------------------------
C
C     sanity-check the input parameters
      RC=0
C       the number of points must be positive
        IF(NN.LE.1) RC=1
C
C       the transform flag must denote either direct or inverse
        IF(INVDIR.NE.+1 .AND. INVDIR.NE.-1) RC=RC+2
C
C       the upper limit of integration was assumed positive
        IF(B.LE.0.D0) RC=RC+4
      IF(RC.NE.0) RETURN
C
C     save the input sequence
      DO 1 K=0,NN-1
           F(K)=DATA(K)
    1 CONTINUE
C
C     find the complex constant whose powers appear in the series
      W=CDEXP(I*DCMPLX(2.D0*PI/DFLOAT(INVDIR*NN),0.D0))
C
```

```
 66 C     compute each F_n from the DFT series
 67       DO 2 N=0,NN-1
 68            FF=(0.D0,0.D0)
 69            DO 3 K=0,NN-1
 70                 FF=FF+W**(N*K)*F(K)
 71     3      CONTINUE
 72            DATA(N)=FF
 73     2 CONTINUE
 74 C
 75 C     find sampling interval and new upper limit of integration
 76       IF(SCALE) THEN
 77          IF(INVDIR.EQ.+1) THEN
 78             DELTA=B/DFLOAT(NN-1)
 79             IF(     SHIFT) B=DFLOAT(NN/2)/(DELTA*DFLOAT(NN))
 80             IF(.NOT.SHIFT) B=DFLOAT(NN-1)/(DELTA*DFLOAT(NN))
 81          ELSE
 82             IF(     SHIFT) DELTA=2.D0*B/DFLOAT(NN)
 83             IF(.NOT.SHIFT) DELTA=B/DFLOAT(NN-1)
 84             B=DFLOAT(NN-1)/(DELTA*DFLOAT(NN))
 85          ENDIF
 86          SF=DELTA
 87       ELSE
 88          IF(INVDIR.EQ.+1) SF=1.D0
 89          IF(INVDIR.EQ.-1) SF=1.D0/DFLOAT(NN)
 90       ENDIF
 91 C
 92 C     scale the output values
 93       IF(SF.NE.1.D0) THEN
 94          DO 11 N=0,NN-1
 95               DATA(N)=SF*DATA(N)
 96    11    CONTINUE
 97       ENDIF
 98 C
 99 C     shift, or compensate for having shifted before
100       IF(SHIFT) THEN
101          IF(INVDIR.EQ.1) THEN
102 C           frequency-shift the forward transform
103             DATA(NN)=DATA(NN/2)
104             DO 12 N=0,NN/2-1
105                   TEMP=DATA(N)
106                   DATA(N)=DATA(N+NN/2)
107                   DATA(N+NN/2)=TEMP
108    12       CONTINUE
109          ELSE
110 C           fix up the inverse of the frequency-shifted transform
111             DO 13 N=1,NN-1,2
112                   DATA(N)=-DATA(N)
113    13       CONTINUE
114          ENDIF
115       ENDIF
116       RETURN
117       END
```

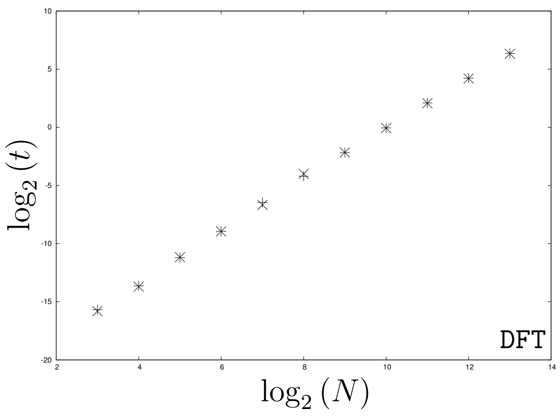
The core of the calculation is $\boxed{\texttt{66-73}}$ the evaluation of the DFT series for each output frequency. This code segment is identical to that used in the program of §4.4, except that instead of being written out the results are returned $\boxed{\texttt{72}}$ in `DATA`. This `DFT` subroutine produces the same results as the §6.4 `FFT` subroutine, within roundoff.[14]

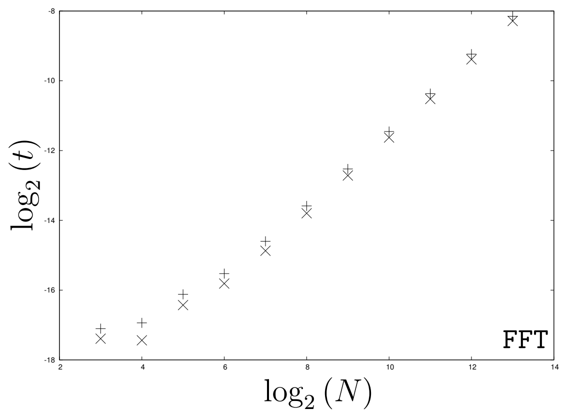
[14]As N ranges from 2^9 to 2^{13} the discrepancy between `DFT` and `FFT` ranges from 10^{-11} to 10^{-9} for $\nu < \nu_c$ and from 10^{-9} to 10^{-6} for $\nu > \nu_c$. Neither `DFT` nor `FFT` is consistently more accurate when compared to the analytic transform.

```
C
C     This program compares the CPU time used by FFT() and DFT().
C
      REAL*8 DB,FB,DELTA,FOFX,X,TPV2R8,DLOG2
      PARAMETER(NMAX=16384)
      COMPLEX*16 DDATA(NMAX+1),FDATA(NMAX+1)
      LOGICAL*4 SCALE/.FALSE./,SHIFT/.FALSE./
      INTEGER*4 P,RC,INVDIR/1/
C
C     set up for timing
      COMMON /EXPT/ TIMING,NONALG,NBIN,TOPBIN,NTMEAS,TOH,BINCPU
      LOGICAL*4 TIMING
      INTEGER*4 TOPBIN,TOH,BINCPU(2,22)
C
C ----------------------------------------------------------------
C
C     open output files
      OPEN(UNIT=1,FILE='dfttimes')
      OPEN(UNIT=2,FILE='ffttimes')
C
C     try different numbers of samples
      DO 1 P=2,13
           N=2**P
C
C          sample f(x)
           DB=3.5D0
           FB=3.5D0
           DELTA=DB/DFLOAT(N-1)
           DO 3 K=1,N
                X=DELTA*DFLOAT(K-1)
                DDATA(K)=DCMPLX(FOFX(X),0.D0)
                FDATA(K)=DCMPLX(FOFX(X),0.D0)
    3      CONTINUE
C
C          initiate to time the transforms
           TIMING=.TRUE.
           CALL TIMER(-1,1)
C
C          compute the discrete Fourier transform
           CALL TIMER(1,2)
           CALL DFT(N,INVDIR,SCALE,SHIFT, DDATA,DB, RC)
C
C          compute the fast Fourier transform
           CALL TIMER(2,4)
           CALL FFT(N,INVDIR,SCALE,SHIFT, FDATA,FB, RC)
C
C          report the times
           CALL TIMER(NONALG,5)
           CALL TIMER(0,6)
           WRITE(1,901) P,DLOG2(TPV2R8(BINCPU(1,1),1000000000))
           WRITE(2,901) P,DLOG2(TPV2R8(BINCPU(1,2),1000000000))
  901      FORMAT(I6,1X,1PE13.6)
C
C          let the processor cool off
           CALL WAIT(60,60)
    1 CONTINUE
      STOP
```

Then I used the program above to exercise `DFT` and `FFT` for different numbers of samples and report the CPU times each routine consumed.

The use of `TIMER` here is similar to what we saw above, but now bin `1` accumulates the time used by `DFT` $\boxed{40}$ and bin `2` is for `FFT` $\boxed{44}$. The measured times are stored as [second,nanosecond] two-part values [7, §18.4] so `TPV2R8` is used $\boxed{50\text{-}51}$ to convert them to `REAL*8` for output. The call to `WAIT` suspends execution of the program for 60 seconds every time another 60 seconds of CPU time has been consumed, but that does not affect the algorithm time measurements. When this program is run with the processor clocking 1995 MHz it produces the results plotted below.[15]





In these graphs the $+$ symbol denotes times t measured when the transforms are scaled and shifted, and the $\times$ symbol times measured when they are not. Least-squares regression on the `DFT` points yields $\log_2(t) = 2.22 \log_2(N) - 22.2$ so $t \propto N^{2.22}$, which is slightly worse than the arithmetic complexity of N^2 (this might be because a practical `DFT` code cannot use precomputed powers of `W` as suggested at the end of §4.2). Least-squares regression on the final 7 FFT points yields $\log_2(t) = 1.10 \log_2(N) - 22.6$ so $t \propto N^{1.10}$. As we saw in §5.2 the arithmetic complexity of the fast Fourier transform is $N \log_2(N)$, so it is not surprising[16] that we observe something slightly faster than N. The breathtaking thing about these graphs is the difference between their (logarithmic) vertical scales; when $N = 2^{13}$, `DFT` takes about $2^{6.32} \approx 80$ seconds but `FFT` is finished in just $2^{-8.15} \approx 0.0035$, and even for very small problems `FFT` is much faster.

[15]The first point in each graph is omitted because `TIMER` reported the processor at 798 MHz; then the automatic mechanism controlling processor frequency detected a compute-intensive job and increased the clock rate to 1995 MHz.

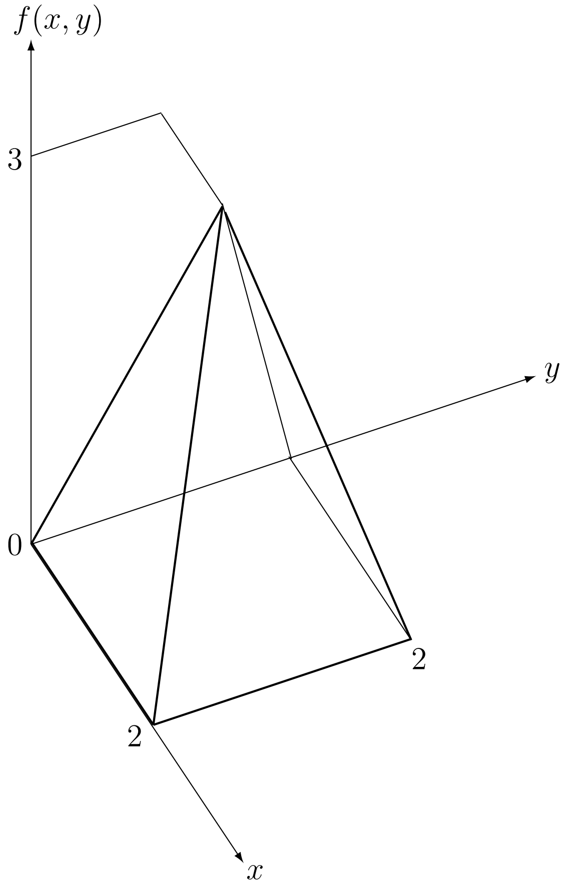
[16]What *is* surprising is that the initial curve in this graph bends the wrong way for $N \log_2(N)$. This might be an experimental artifact arising in the measurement of very brief times, but I will be grateful to anyone who provides me with a more convincing explanation.

Now we can calculate the impact on performance of the parameter checking that is done by `FFT`, which we estimated earlier takes 1128×10^{-9} seconds on a 798 MHz processor. At 1995 MHz the same number of cycles take 451.2×10^{-9} seconds, accounting for these percentages of execution time. Unless we need to find a large number of small transforms, the time spent checking parameters is negligible as claimed at the beginning of this Section.

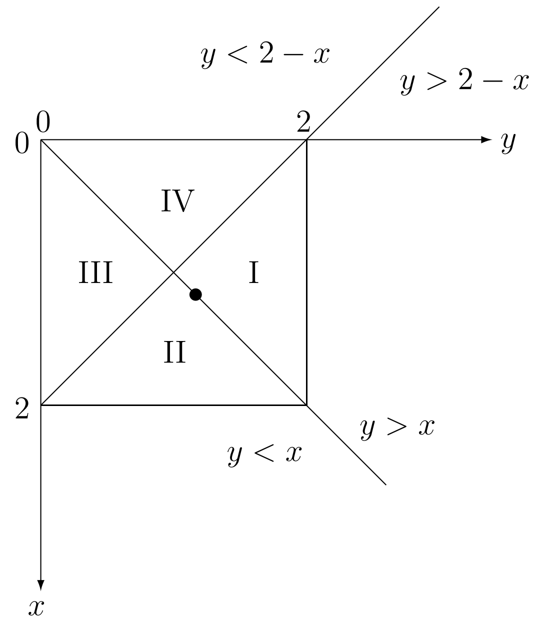
N	`FFT` time, sec	overhead, %
2^3	7.099014E-06	6.36
2^4	7.960994E-06	5.67
2^5	1.403404E-05	3.22
2^6	2.117395E-05	2.13
2^7	4.026400E-05	1.12
2^8	8.127099E-05	0.56
2^9	1.693716E-04	0.27
2^{10}	3.564531E-04	0.13
2^{11}	7.567425E-04	0.06
2^{12}	1.658639E-03	0.03
2^{13}	3.517685E-03	0.01

7 FFTs in Two Dimensions

In image processing the input signal is often a scalar function $f(x,y)$ of two variables, such as the pulse pictured on the left. The top view on the bottom right shows the conditions defining the faces of the pyramid, which lead to the function definition on the top right.



$$f(x,y) = \begin{cases} 0 & (x,y) \text{ outside pyramid base} \\ 6-3y & y \geq x, y \geq 2-x \quad \text{I} \\ 6-3x & y \leq x, y \geq 2-x \quad \text{II} \\ 3y & y \leq x, y \leq 2-x \quad \text{III} \\ 3x & y \geq x, y \leq 2-x \quad \text{IV} \end{cases}$$



The integral we used in §1 for the Fourier transform of a function of one variable generalizes to the case of two dimensions like this,

$$\mathcal{F}\{f(x,y)\} = F(\nu_x,\nu_y) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x,y)e^{i2\pi\nu_x x}e^{i2\pi\nu_y y}dx\,dy$$

and the DFT definition of §4.1 generalizes in a similar way. Sampling $f(x,y)$ at points $[(k_x-1)\Delta_x,(k_y-1)\Delta_y]^\top, k_x = 1,\ldots,N, k_y = 1,\ldots,N$ yields an $N\times N$ matrix of values f_{k_x,k_y} from which we can compute the **two-dimensional DFT** of the samples like this.

$$F_{n_x,n_y} = \sum_{k_y=1}^{N}\sum_{k_x=1}^{N} f_{k_x,k_y}W^{(n_x-1)(k_x-1)}W^{(n_y-1)(k_y-1)}$$

The outside sum doesn't depend on k_x, so we can insert parentheses like this.

$$F_{n_x,n_y} = \sum_{k_y=1}^{N} \left(\sum_{k_x=1}^{N} f_{k_x,k_y} W^{(n_x-1)(k_x-1)} \right) W^{(n_y-1)(k_y-1)}$$

If we call the sum in parentheses $G(n_x, k_y)$ then we can find the two-dimensional DFT by evaluating the following one-dimensional DFTs in sequence.

$$\begin{aligned} G_{n_x,k_y} &= \sum_{k_x=1}^{N} f_{k_x,k_y} W^{(n_x-1)(k_x-1)} \qquad n_x = 1 \ldots N, \quad k_y = 1 \ldots N \\ F_{n_x,n_y} &= \sum_{k_y=1}^{N} G_{n_x,k_y} W^{(n_y-1)(k_y-1)} \qquad n_x = 1 \ldots N, \quad n_y = 1 \ldots N \end{aligned}$$

To find G we compute N one-dimensional transforms of the columns, and to find F we compute N one-dimensional transforms of the resulting rows. If we take the upper limit of integration to be $3\frac{1}{2}$ in each direction (as in the one-dimensional example of §4) then sampling the pyramidal pulse pictured above with $N = 4$ yields the data matrix f on the left below. The one nonzero sample, at $x = 1\frac{1}{6}, y = 1\frac{1}{6}$, is shown as a dot • in the view looking down on the pyramid on the previous page, and has height $f_{2,2} = 2.5$. Following the procedure outlined above to compute the transform yields the result matrix F on the right. Each complex element is boxed with dark lines and contains the real part of the element in the upper square and the imaginary part in the lower square.

$$f = \begin{array}{c|cccc} x \backslash y & 0 & 1\frac{1}{6} & 2\frac{1}{3} & 3\frac{1}{2} \\ \hline & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline & 0 & 2.5 & 0 & 0 \\ 1\frac{1}{6} & 0 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 \\ 2\frac{1}{3} & 0 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 \\ 3\frac{1}{2} & 0 & 0 & 0 & 0 \\ \hline \end{array} \qquad F = \begin{array}{c|cccc} \nu_x \backslash \nu_y & 0.00 & 0.21 & 0.43 & 0.64 \\ \hline & 2.5 & 0 & -2.5 & 0 \\ 0.00 & 0 & 2.5 & 0 & -2.5 \\ \hline & 0 & -2.5 & 0 & 2.5 \\ 0.21 & 2.5 & 0 & -2.5 & 0 \\ \hline & -2.5 & 0 & 2.5 & 0 \\ 0.43 & 0 & -2.5 & 0 & 2.5 \\ \hline & 0 & 2.5 & 0 & -2.5 \\ 0.64 & -2.5 & 0 & 2.5 & 0 \\ \hline \end{array}$$

The axes bordering f give the coordinates of the samples in $x - y$ space, and those bordering F give the coordinates of the transform elements in $\nu_x - \nu_y$ space, to show that distance and frequency increase with row or column index in the matrix (the origin of coordinates is at the top left of the picture). With $N = 4$ and $b = 3$ we have

$$\begin{aligned}\Delta_x &= \frac{b}{N-1} = \frac{3.5}{3} = \frac{7}{6} \approx 1.167 \\ \Delta_\nu &= \frac{1}{N\Delta_x} = \frac{1}{4 \times \frac{7}{6}} = \frac{3}{14} \approx 0.214\end{aligned}$$

Although only $f_{2,2} = (2.5 + 0i)$ is nonzero, each element of the transform is $(2.5 + 0i)$, $(-2.5+0i)$, $(0+2.5i)$, or $(0-2.5i)$. In general each can have both real and imaginary parts.

7.1 DFT by Nested Sums

For simplicity I have assumed that the number of samples is the same in both directions, but they might be different. If `N_x` is the number of samples in the x direction and `N_y` is the number in the y direction, the algorithm described above for computing the two-dimensional DFT looks like this in pseudocode.

```
do k_y = 1 ... N_y
   transform the k_y'th column of f into the k_y'th column of G
enddo
do n_x = 1 ... N_x
   transform the n_x'th row of G into the n_x'th row of F
enddo
```

A library routine to perform this calculation should of course also provide the amenities we earlier built into the one-dimensional codes `FFT` and `DFT`. The `DFT2` routine listed on the following pages is a generalization of `DFT` (see pages 63-64).

The calling sequence $\boxed{4}$ now includes the dimensions `LX` and `LY` of the data matrix `F`, the sizes `NNX` and `NNY` of the transform in both directions, and the limits of integration `BX` and `BY` in both directions. The data matrix `F` is now two-dimensional $\boxed{43}$ and is accompanied by a workspace matrix `G` of the same size $\boxed{48}$. There are now two constants `WX` and `WY` $\boxed{51,72\text{-}73}$, two sampling intervals `DX` and `DY`, and two scale factors `SFX` and `SFY` $\boxed{52}$. The parameter checking is similar to that in `DFT`, but now we also make sure $\boxed{67\text{-}68}$ that the dimensions of `F` make sense for the numbers of rows and columns in use.

The pseudocode above is implemented by loops for transforming the columns $\boxed{75\text{-}82}$ and rows $\boxed{84\text{-}91}$, which evaluate the formulas given on page 69 in a straightforward way.

The scaling code $\boxed{93\text{-}134}$ resembles that used in `DFT`, but if the result is to be frequency-shifted that is done in both directions. Frequency-shifting the transforms of the columns $\boxed{139\text{-}147}$ adds a row, which must be included in frequency-shifting the transforms of the rows $\boxed{148\text{-}156}$. The inverse of a forward transform that was shifted needs to be fixed up by changing the sign of every other term (they are the odd-numbered ones here because the indices run from `0`). Doing this down each column and across each row results in a checkerboard pattern of sign changes that is performed by the nest of `DO 10` loops $\boxed{158\text{-}162}$.

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE DFT2(LX,LY,NNX,NNY,INVDIR,SCALE,SHIFT, F,BX,BY, RC)
C     This routine returns in-place the direct or inverse discrete
C     Fourier transform of the NX x NY array of numbers in the upper
C     left corner of F(LX,LY) by evaluating the nested sum.
C
C     variable  meaning
C     --------  -------
C     BX        limit of integration in the X direction
C     BY        limit of integration in the Y direction
C     CDEXP     Fortran function returns exp(COMPLEX*16)
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     DX        sampling interval in Y
C     DY        sampling interval in X
C     F         input data, then its transform or inverse
C     G         intermediate matrix
C     I         i, the square root of -1
C     INVDIR    +1 => direct transform, -1 => inverse transform
C     KX        index on input rows
C     KY        index on input columns
C     LX        first dimension of F in calling routine
C     LY        second dimension of F in calling routine
C     MOD       Fortran function for remainder of INTEGER*4 division
C     NNX       number of rows
C     NNY       number of columns
C     NX        index on output rows
C     NY        index on output columns
C     PI        the circle constant
C     RC        return code; 0 => parameters make sense
C     SCALE     T => scale output or assume that input is scaled
C     SFX       scale factor in X
C     SFY       scale factor in Y
C     SHIFT     T => frequency-shift result
C     TEMP      temporary used in shifting
C     WX        exp(i*2*pi/NNX)
C     WY        exp(i*2*pi/NNY)
C
C     formal parameters
      LOGICAL*4 SCALE,SHIFT
      COMPLEX*16 F(0:LX-1,0:LY-1)
      REAL*8 BX,BY
      INTEGER*4 RC
C
C     automatic workspace
      COMPLEX*16 G(0:LX-1,0:LY-1)
C
C     other local variables
      COMPLEX*16 I/(0.D0,1.D0)/,TEMP,WX,WY
      REAL*8 DX,DY,PI/3.1415926535897932D0/,SFX,SFY
C
C -----------------------------------------------------------------
C
```

```
C     sanity-check the input parameters
      RC=0
C       the numbers of points must be positive
        IF(NNX.LE.1 .OR. NNY.LE.1) RC=1
C
C       the transform flag must denote either direct or inverse
        IF(INVDIR.NE.+1 .AND. INVDIR.NE.-1) RC=RC+2
C
C       the upper limits of integration were assumed positive
        IF(BX.LE.0.D0 .OR. BY.LE.0.D0) RC=RC+4
C
C       the leading dimensions must be big enough
        IF(LX-1.LT.NNX .OR. LY-1.LT.NNY) RC=RC+8
      IF(RC.NE.0) RETURN
C
C     find the complex constant whose powers appear in the series
      WX=CDEXP(I*DCMPLX(2.D0*PI/DFLOAT(INVDIR*NNX),0.D0))
      WY=CDEXP(I*DCMPLX(2.D0*PI/DFLOAT(INVDIR*NNY),0.D0))
C
C     transform the columns
      DO 1 KY=0,NNY-1
      DO 1 NX=0,NNX-1
           G(NX,KY)=(0.D0,0.D0)
           DO 2 KX=0,NNX-1
                G(NX,KY)=G(NX,KY)+F(KX,KY)*WX**(NX*KX)
    2      CONTINUE
    1 CONTINUE
C
C     transform the rows
      DO 3 NX=0,NNX-1
      DO 3 NY=0,NNY-1
           F(NX,NY)=(0.D0,0.D0)
           DO 4 KY=0,NNY-1
                F(NX,NY)=F(NX,NY)+G(NX,KY)*WY**(NY*KY)
    4      CONTINUE
    3 CONTINUE
C
```

```
C     find sampling interval and new upper limit of integration
      IF(SCALE) THEN
         IF(INVDIR.EQ.+1) THEN
            DX=BX/DFLOAT(NNX-1)
            DY=BY/DFLOAT(NNY-1)
            IF(SHIFT) THEN
               BX=DFLOAT(NNX/2)/(DX*DFLOAT(NNX))
               BY=DFLOAT(NNY/2)/(DY*DFLOAT(NNY))
            ELSE
               BX=DFLOAT(NNX-1)/(DX*DFLOAT(NNX))
               BY=DFLOAT(NNY-1)/(DY*DFLOAT(NNY))
            ENDIF
         ELSE
            IF(SHIFT) THEN
               DX=2.D0*BX/DFLOAT(NNX)
               DY=2.D0*BY/DFLOAT(NNY)
            ELSE
               DX=BX/DFLOAT(NNX-1)
               DY=BY/DFLOAT(NNY-1)
            ENDIF
            BX=DFLOAT(NNX-1)/(DX*DFLOAT(NNX))
            BY=DFLOAT(NNY-1)/(DY*DFLOAT(NNY))
         ENDIF
         SFX=DX
         SFY=DY
      ELSE
         IF(INVDIR.EQ.+1) THEN
            SFX=1.D0
            SFY=1.D0
         ELSE
            SFX=1.D0/DFLOAT(NNX)
            SFY=1.D0/DFLOAT(NNY)
         ENDIF
      ENDIF
C
C     scale the output values
      IF(SFX.NE.1.D0 .OR. SFY.NE.1.D0) THEN
         DO 5 NY=0,NNY-1
         DO 5 NX=0,NNX-1
              F(NX,NY)=DCMPLX(SFX*SFY,0.D0)*F(NX,NY)
    5    CONTINUE
      ENDIF
C
```

```
C     shift, or compensate for having shifted before
      IF(SHIFT) THEN
         IF(INVDIR.EQ.1) THEN
C           frequency-shift forward transforms of columns
            DO 6 NY=0,NNY-1
                 F(NNX,NY)=F(NNX/2,NY)
                 DO 7 NX=0,NNX/2-1
                      TEMP=F(NX,NY)
                      F(NX,NY)=F(NX+NNX/2,NY)
                      F(NX+NNX/2,NY)=TEMP
    7            CONTINUE
    6       CONTINUE
C           frequency-shift forward transforms of rows
            DO 8 NX=0,NNX
                 F(NX,NNY)=F(NX,NNY/2)
                 DO 9 NY=0,NNY/2-1
                      TEMP=F(NX,NY)
                      F(NX,NY)=F(NX,NY+NNY/2)
                      F(NX,NY+NNY/2)=TEMP
    9            CONTINUE
    8       CONTINUE
         ELSE
C           fix up the inverse of the frequency-shifted transform
            DO 10 NY=0,NNY-1
            DO 10 NX=0+MOD(NY+1,2),NNX-1,2
                  F(NX,NY)=-F(NX,NY)
   10       CONTINUE
         ENDIF
      ENDIF
      RETURN
      END
```

The program on the next page uses the `FOFX2` routine listed below to calculate values of the function $f(x,y)$, and then invokes `DFT2` to compute the transform of our pyramidal pulse.

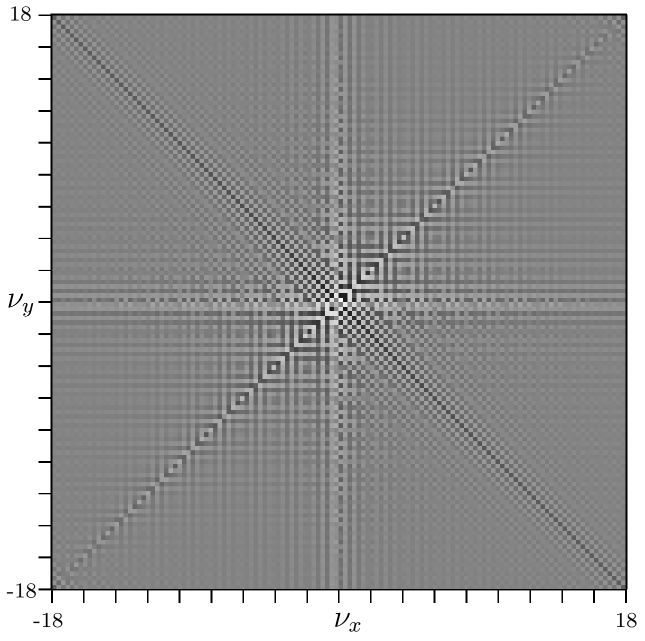
```
      FUNCTION FOFX2(X,Y)
      REAL*8 FOFX2,X,Y
C
C     return zero outside the pulse
      FOFX2=0.D0
      IF(X.LE.0.D0 .OR. X.GE.2.D0) RETURN
      IF(Y.LE.0.D0 .OR. Y.GE.2.D0) RETURN
C
C     return the height of the pyramid inside the pulse
      IF(Y.LE.X) THEN
         IF(Y .LE. 2.D0-X) THEN
C           on face III
            FOFX2=3.D0*Y
         ELSE
C           on face II
            FOFX2=6.D0-3.D0*X
         ENDIF
      ELSE
         IF(Y .LE. 2.D0-X) THEN
C           on face IV
            FOFX2=3.D0*X
         ELSE
C           on face I
            FOFX2=6.D0-3.D0*Y
         ENDIF
      ENDIF
      RETURN
      END
```

```
C
C     This program exercises DFT2.
C
      REAL*8 BX/3.5D0/,BY/3.5D0/,DNUX,DNUY,DX,DY,NUX,NUY,X,Y,FOFX2
      PARAMETER(LX=1024,LY=1024)
      COMPLEX*16 F(LX,LY)
      INTEGER*4 RC
      LOGICAL*4 SCALE,SHIFT,QUERY
C
C ------------------------------------------------------------------
C
C     find out how many points to use
      CALL PROMPT('NX=',3)
      READ(5,*,END=1) NX
      DX=BX/DFLOAT(NX-1)
      DNUX=1.D0/(DX*DFLOAT(NX))
      CALL PROMPT('NY=',3)
      READ(5,*,END=1) NY
      DY=BY/DFLOAT(NY-1)
      DNUY=1.D0/(DY*DFLOAT(NY))
C
C     sample f(x,y)
      DO 1 KX=1,NX
           X=DX*DFLOAT(KX-1)
           DO 2 KY=1,NY
                Y=DY*DFLOAT(KY-1)
                F(KX,KY)=DCMPLX(FOFX2(X,Y),0.D0)
    2      CONTINUE
    1 CONTINUE

C     find out about scaling and shifting
      SCALE=QUERY('scale?',6)
      SHIFT=QUERY('shift?',6)
C
C     direct transform
      CALL DFT2(LX,LY,NX,NY,+1,SCALE,SHIFT, F,BX,BY, RC)
C
C     report the result
      IF(SHIFT) THEN
         KXMAX=NX+1
         KYMAX=NY+1
      ELSE
         KXMAX=NX
         KYMAX=NY
      ENDIF
      DO 3 KX=1,KXMAX
           NUX=DNUX*DFLOAT(KX-1)
           IF(SHIFT) NUX=NUX-0.5D0*DNUX*DFLOAT(NX)
           DO 4 KY=1,KYMAX
                NUY=DNUY*DFLOAT(KY-1)
                IF(SHIFT) NUY=NUY-0.5D0*DNUY*DFLOAT(NY)
                WRITE(2,901) NUX,NUY,F(KX,KY)
  901           FORMAT(4(1X,1PE13.6))
    4      CONTINUE
    3 CONTINUE
      STOP
      END
```

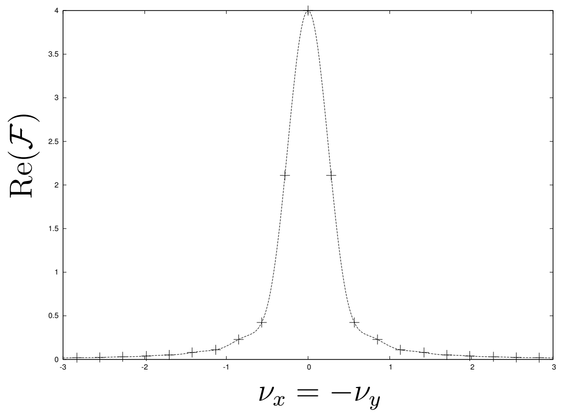
When the three routines are compiled together and run, the result is a file of lines each containing one point $[\nu_x, \nu_y, \mathrm{Re}(F_{n_x,n_y}), \mathrm{Im}(F_{n_x,n_y})]$ in the transform. If the prompts are answered with `NX=4`, `NY=4`, and no shifting or scaling, this output agrees with the F matrix pictured earlier.

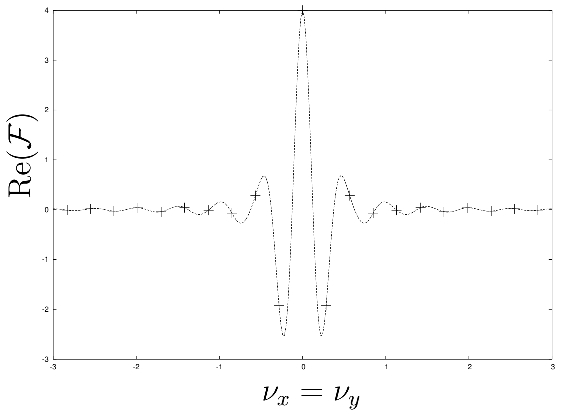
If the prompts are answered with `NX=128`, `NY=128`, and yes to shifting and scaling, the output file contains $129 \times 129 = 16641$ points. Unshifted the transform would cover spatial frequencies from 0 to $(N-1)/b = 127/3.5 \approx 36.285$ Hz, so frequency-shifted it covers frequencies from about -18 to about $+18$ Hz in each direction. To interpret this data we can show the undulations of Re(`F`) or Im(`F`) with ν_x and ν_y in a **raster plot**. In the raster plot below, the gray scale of each pixel depends[17] on the values of Re(`F`) inside that pixel's boundaries. Because we frequency-shifted the transform, $(\nu_x = 0, \nu_y = 0)$ is in the middle of the picture, and because we scaled it the values returned in matrix `F` are actually of $\Delta_x \Delta_y F \approx \mathcal{F}(f(x,y))$.



The symmetry of the raster plot suggests that the behavior of the transform might also be visualized by plotting Re(`F`) along the diagonals. The graph on the left at the top of the next page is the transform along the cut $\nu_x = -\nu_y$; the one on the right is for the other diagonal, where $\nu_x = \nu_y$.

[17]The values of Re(F) span several orders of magnitude, so to enhance the contrast of the raster image I compressed the vertical scale of the data using a log transformation of the form $|\log_a(1 + |\bullet|)| \times \mathrm{sgn}(\bullet)$, where a was chosen to map the Re(F) that is largest in absolute value to $+1$ or -1 according to its sign. This scaling distorts the shape of the function but preserves its zeros.





The points + in these plots are values from `F` and the solid curve is the continuous transform. The double integral defining the continuous transform is tedious to evaluate by hand, but it was straightforward using the symbolic algebra package Maple.[18] From the agreement of the points with the lines in these graphs, it appears the DFT provides a plausible approximation to the continuous transform, at least along those two cuts.

In two dimensions it is still expensive to calculate the DFT by adding up the sums in its definition. Now, if the number of samples in each direction is N we use $N \times N^2$ operations in transforming the columns and $N \times N^2$ more for the rows, or $2N^3$ altogether. If instead we use the FFT to transform each column and then each row, the complexity will be $2N^2 \log_2(N)$, a big improvement. The ratios in the rightmost column below are the same as those in the table on page 35 for the one-dimensional FFT, so now as then we have a strong incentive to use the FFT.

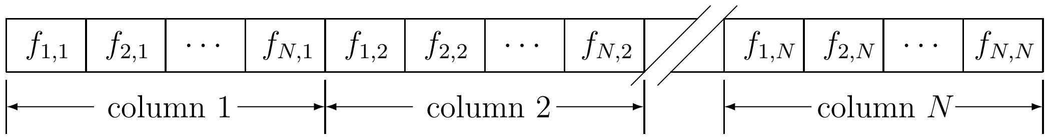
N	$2N^2 \log_2(N)$	$2N^3$	$(2N^2 \log_2(N))/(2N^3)$
2	8	16	0.500000
4	64	128	0.500000
8	384	1024	0.375000
16	2048	8192	0.250000
32	10240	65536	0.156250
64	49152	524288	0.093750
128	229376	4194304	0.054688
256	1048576	33554432	0.031250
512	4718592	268435456	0.017578
1024	20971520	2147483648	0.009766
2048	92274688	17179869184	0.005371
4096	402653184	137438953472	0.002930
8192	1744830464	1099511627776	0.001587
16384	7516192768	8796093022208	0.000854
32768	32212254720	70368744177664	0.000458

[18]After Maple had found the double integral in terms of complex exponentials, `F:=convert(F,trig)` followed by `evalc(Re(F))` and `evalc(Im(F))` yielded lengthy formulas for the real and imaginary parts as functions of ν_x and ν_y. Reminiscent of §2, these formulas are 0/0 indeterminate along the diagonals and axes of the raster plot, so I used Maple to apply L'Hospital's Rule. From the resulting definition of $F(\nu_x, \nu_y)$, I wrote a FORTRAN program to compute the values plotted in the graphs.

7.2 FFT by Transposition

We already have a library routine for computing one-dimensional FFTs, so the most obvious approach for two dimensions is to use it somehow for finding the transforms of the columns and rows in the matrix f.

FORTRAN stores arrays in column-major order [7, §11.1], so if f is $N \times N$ its elements are arranged in adjacent blocks of memory as shown below. Here each box represents *two* doublewords of memory, one for the real part of the array element and one for the imaginary part.



FORTRAN passes subprogram parameters by address [7, §6.2], so to pass a matrix column for a vector parameter it is necessary only to supply the address of the first array element in the column. The first loop in the pseudocode (on page 70) for calculating the DFT can therefore be rendered into FORTRAN like this.

```
      DO 1 KY=1,N
           CALL FFT(N,INVDIR,SCALE,SHIFT, F(1,KY),B, RC)
    1 CONTINUE
```

A FORTRAN variable is the name (which the compiler translates into the address) of a storage location in memory, so the address of the first element in, say, the second column of f is `F(1,2)`. In the code above, when `KY=2` the address of `F(1,2)` is passed for the address of the data vector that `FFT` will transform in-place.

Passing rows is harder, because the consecutive elements in each row of f are spread across the columns and are thus separated by N blocks in memory. For example, the first and second elements of the second row are $f_{2,1}$ and $f_{2,2}$, which are separated by N boxes in the picture. `FFT` assumes that consecutive elements of its input data vector are adjacent to one another, not separated by N boxes. But if we transpose f its rows become the columns of $f^{\top}$, and then we can pass those. After that loop of `FFT` calls it is necessary to transpose again, to put the elements of the transform back into the same order as the input data.

These ideas are incorporated in the routine `FFT2TR`, which is listed on the following pages. The argument list $\boxed{\texttt{4}}$ is identical to that of `DFT2`, so the two routines can be used interchangeably.

Most of the needed parameter checking is done by `FFT` each time it is called $\boxed{\texttt{66,84}}$, and any errors that are detected in that way are acted upon when that routine returns $\boxed{\texttt{67-70,85-88}}$ (see §6.5 for a discussion of the overhead incidental to this repeated parameter checking). Because the transpositions are performed in-place, `F` must be big enough to hold either the data matrix or its transpose, and this is verified in the first parameter-checking stanza $\boxed{\texttt{51-56}}$ of this routine. The second parameter-checking stanza $\boxed{\texttt{58-62}}$ ensures that if the result is to be shifted there is enough room in `F` for the extra row and column. The new return code of `RC=8` means that `F` isn't big enough.

The in-place transpositions $\boxed{\texttt{64-71,82-89}}$ work by exchanging off-diagonal elements across the diagonal, so only the one `COMPLEX*16` temporary `TEMP` is needed. In each row of the matrix to be transposed, each element `(I,J)` before the diagonal is exchanged with its mirror image element `(J,I)` on the other side of the diagonal. For example, in the second row `I=2` and the single element to the left of the diagonal, at `J=1`, gets exchanged with element `(1,2)`, the first one to the right of the diagonal in the top row. This method moves the fewest elements possible.

Scaling the column transforms in `FFT` would change the frequency content of the row transforms, and scaling the row transforms in `FFT` would use the factor for one dimension instead of two. Thus, if scaling is done it must be after the two-dimensional transform is complete, and the calls to `FFT` pass `.FALSE.` for its `SCALE` parameter. The scaling part of the code $\boxed{\texttt{99-141}}$ differs slightly from that in `DFT2` because `FFT` always does the scaling by $1/N$ when it computes an inverse transform.

Frequency-shifting the columns in the first loop of calls to `FFT` would add a row that can't be included in the second loop of calls to `FFT`, so if shifting is done that must also be after the transform is complete. Thus the calls to `FFT` pass `.FALSE.` for its `SHIFT` parameter, and `FFT2TR` includes frequency-shifting code $\boxed{\texttt{143-171}}$ like that used in `DFT2`.

To test the `FFT2TR` routine we can use the same program we used to test `DFT2`, changing only the name of the routine under test. When I compiled the resulting main program with `FFT2TR` and `FOFX2`, the program produced results identical within roundoff to those found by `DFT2` for various sizes and shapes of input matrix and for all combinations of `SCALE` and `SHIFT` values.

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE FFT2TR(LX,LY,NX,NY,INVDIR,SCALE,SHIFT, F,BX,BY, RC)
C     This routine computes in-place the direct or inverse fast
C     Fourier transform of the NX x NY array of numbers in the upper
C     left corner of F(LX,LY) by using calls to FFT and in-place
C     transpositions.
C
C     variable  meaning
C     --------  -------
C     BX        limit of integration in the X direction
C     BY        limit of integration in the Y direction
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     DX        sampling interval in X
C     DY        sampling interval in Y
C     F         data matrix
C     FFT       routine computes 1-dimensional FFT
C     I         index on rows of F
C     INVDIR    1 => direct, -1 => inverse
C     J         index on columns of F
C     LX        first dimension of F in calling routine
C     LY        second dimension of F in calling routine
C     MAX0      Fortran function returns larger of INTEGER*4s
C     MOD       Fortran function for remainder of INTEGER*4 division
C     N         size of transposition
C     NX        number of rows
C     NY        number of columns
C     RC        return code; 0 => all went well
C     RCFFT     FFT return code; 0 => all went well
C     SCALE     T => scale transform/transform was scaled
C     SFX       scale factor in X
C     SFY       scale factor in Y
C     SHIFT     T => frequency-shift transform/transform was shifted
C     TEMP      temporary for in-place transpositions
C
C     formal parameters
      LOGICAL*4 SCALE,SHIFT
      COMPLEX*16 F(LX,LY)
      REAL*8 BX,BY
      INTEGER*4 RC
C
C     local variables
      REAL*8 DX,DY,SFX,SFY
      INTEGER*4 RCFFT
      COMPLEX*16 TEMP
C
C -----------------------------------------------------------------
C
```

```
C     check that there is enough space in F for the transposes
      N=MAX0(NX,NY)
      IF(NX.LE.0 .OR. NY.LE.0 .OR. N.GT.LX .OR. N.GT.LY) THEN
         RC=8
         RETURN
      ENDIF
C
C     check that there is enough space in F for the result
      IF(SHIFT .AND. (NX+1.GT.LX .OR. NY+1.GT.LY)) THEN
         RC=8
         RETURN
      ENDIF
C
C     transform the columns
      DO 1 J=1,NY
           CALL FFT(NX,INVDIR,.FALSE.,.FALSE., F(1,J),BX, RCFFT)
           IF(RCFFT.NE.0) THEN
              RC=RCFFT
              RETURN
           ENDIF
    1 CONTINUE
C
C     transpose in-place
      DO 2 I=1,N
           DO 3 J=1,I-1
                TEMP=F(I,J)
                F(I,J)=F(J,I)
                F(J,I)=TEMP
    3      CONTINUE
    2 CONTINUE
C
C     transform the rows
      DO 4 I=1,NX
           CALL FFT(NY,INVDIR,.FALSE.,.FALSE., F(1,I),BY, RCFFT)
           IF(RCFFT.NE.0) THEN
              RC=RCFFT
              RETURN
           ENDIF
    4 CONTINUE
C
C     transpose in-place
      DO 5 I=1,N
           DO 6 J=1,I-1
                TEMP=F(I,J)
                F(I,J)=F(J,I)
                F(J,I)=TEMP
    6      CONTINUE
    5 CONTINUE
C
```

```fortran
C     find sampling intervals and new upper limits of integration
      IF(SCALE) THEN
         IF(INVDIR.EQ.+1) THEN
            DX=BX/DFLOAT(NX-1)
            DY=BY/DFLOAT(NY-1)
            IF(SHIFT) THEN
               BX=DFLOAT(NX/2)/(DX*DFLOAT(NX))
               BY=DFLOAT(NY/2)/(DY*DFLOAT(NY))
            ELSE
               BX=DFLOAT(NX-1)/(DX*DFLOAT(NX))
               BY=DFLOAT(NY-1)/(DY*DFLOAT(NY))
            ENDIF
            SFX=DX
            SFY=DY
         ELSE
            IF(SHIFT) THEN
               DX=2.D0*BX/DFLOAT(NX)
               DY=2.D0*BY/DFLOAT(NY)
            ELSE
               DX=BX/DFLOAT(NX-1)
               DY=BY/DFLOAT(NY-1)
            ENDIF
            BX=DFLOAT(NX-1)/(DX*DFLOAT(NX))
            BY=DFLOAT(NY-1)/(DY*DFLOAT(NY))
C           scaling for inverse is done inside FFT
            SFX=DX*DFLOAT(NX)
            SFY=DY*DFLOAT(NY)
         ENDIF
      ELSE
C        forward transform is not to be scaled
C        scaling for inverse is done inside FFT
         SFX=1.D0
         SFY=1.D0
      ENDIF
C
C     scale result to match Fourier transform if desired
      IF(SFX.NE.1.D0 .OR. SFY.NE.1.D0) THEN
         DO 7 I=1,NX
         DO 7 J=1,NY
              F(I,J)=DCMPLX(SFX*SFY,0.D0)*F(I,J)
    7    CONTINUE
      ENDIF
C
```

```
C     shift, or compensate for having shifted before
      IF(SHIFT) THEN
         IF(INVDIR.EQ.1) THEN
C           frequency-shift forward transforms of columns
            DO 8 J=1,NY
                 F(NX+1,J)=F(1+NX/2,J)
                 DO 9 I=1,NX/2
                      TEMP=F(I,J)
                      F(I,J)=F(I+NX/2,J)
                      F(I+NX/2,J)=TEMP
    9            CONTINUE
    8       CONTINUE
C           frequency-shift forward transforms of rows
            DO 10 I=1,NX+1
                  F(I,NY+1)=F(I,1+NY/2)
                  DO 11 J=1,NY/2
                        TEMP=F(I,J)
                        F(I,J)=F(I,J+NY/2)
                        F(I,J+NY/2)=TEMP
   11             CONTINUE
   10       CONTINUE
         ELSE
C           fix up the inverse of the frequency-shifted transform
            DO 12 J=1,NY
            DO 12 I=1+MOD(J,2),NX,2
                  F(I,J)=-F(I,J)
   12       CONTINUE
         ENDIF
      ENDIF
      RETURN
      END
```

7.3 FFT by Strides

The transpositions in the FFT2TR routine of §7.2 take extra time, and they are needed only because FFT expects its input data to be in adjacent memory blocks (pairs of doublewords). Accessing data in that way within FFT is convenient, and because that order makes minimal use of the computer's memory hierarchy [7, §15.2.7] FFT is fast. But the transpositions must access data that are *not* consecutive in storage, and the work of doing them might be avoided simply by accessing non-consecutive data in computing a two-dimensional transform. Let's call the routine that does that FFT2ST.

As explained in §7.2, if f is NX $\times$ NY the column-major array layout separates successive elements of a row by NX memory blocks. The distance of NX blocks from $f_{1,1}$ to $f_{1,2}$ (or from $f_{2,1}$ to $f_{2,2}$, etc.) is called the **stride** in memory between the elements of a row (i.e., between columns). If the starting address of a two-dimensional array F is passed for the vector DATA that we will use in FFT2ST, the elements of F will be arranged as shown in the picture on page 78, and to figure out where they are in DATA all we need to know is the stride NX. For example, the COMPLEX*16 data array might be F(4,8), in which case its elements will have the indices in the REAL*8 vector DATA shown below. Each COMPLEX*16 element of F is boxed with dark lines. Thus F(1,1) consists of DATA(1), containing the real part of the number, and DATA(2) for the imaginary part.

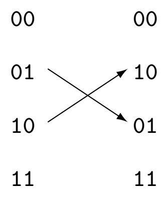
F =

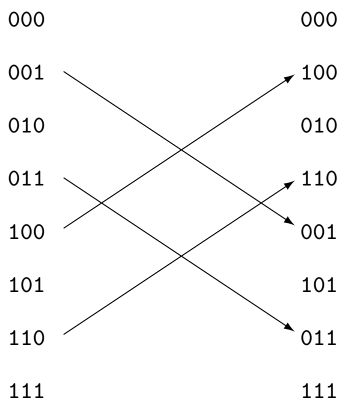
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

7.3.1 Input Data Rearrangement

Each column of F has elements that are contiguous in memory, so it is just a vector. We can put the first column into bit-reversed-index order using the algorithm from FFT, repeating each swap in the other columns to rearrange the rows of F. A similar process can then be used to put the columns into bit-reversed-index order, but the elements in each row are not contiguous so in addressing them we must account for stride. The program at the top of the next page implements this strategy. It stores NX and NY in the vector NN.

The bit-reversed reorderings of 4 and 8 indices shown on the left below (recall §6.2.1) show that in the F given above the middle pairs of rows will be swapped, the second column will be swapped with the fifth, and the fourth column will be swapped with the seventh. The program produces the output on the right below, which lists the indices of the doublewords in DATA that are swapped to rearrange the rows and columns in F. The first 8 lines swap the middle pair of rows, the next 4 swap the second column with the fifth, and the final 4 swap the fourth column with the seventh.





```
( 3, 4) <--> ( 5, 6)
(11,12) <--> (13,14)
(19,20) <--> (21,22)
(27,28) <--> (29,30)
(35,36) <--> (37,38)
(43,44) <--> (45,46)
(51,52) <--> (53,54)
(59,60) <--> (61,62)
( 9,10) <--> (33,34)
(11,12) <--> (35,36)
(13,14) <--> (37,38)
(15,16) <--> (39,40)
(25,26) <--> (49,50)
(27,28) <--> (51,52)
(29,30) <--> (53,54)
(31,32) <--> (55,56)
```

```
 1 C     This program figures out the rearrangement of rows and then
 2 C     columns so the indices are in ascending bit-reversed order.
 3 C
 4       INTEGER*4 NN(2)/4,8/,T
 5 C
 6 C     step by rows two at a time to go down a column
 7       J=1
 8       DO 1 I=3,2*NN(1)-3,2
 9            MSZ=NN(1)
10     3      IF(J.LT.MSZ) GO TO 2
11               J=J-MSZ
12               MSZ=MSZ/2
13            GO TO 3
14     2      J=J+MSZ
15            IF(J.GT.I) THEN
16 C             repeat row swaps for all columns
17               DO 4 K=I,I+2*NN(1)*NN(2)-2*NN(1),2*NN(1)
18 C                  repeat the pair of row swaps in each column
19                    T=J+(K-I)
20                    WRITE(6,991) K,K+1,T,T+1
21   991              FORMAT('(',I2,',',I2,') <--> (',I2,',',I2,')')
22     4         CONTINUE
23            ENDIF
24     1 CONTINUE
25 C
26 C     step by columns to go across a row
27       J=1
28       DO 5 I=1+2*NN(1),2*NN(1)*NN(2)-4*NN(1)+1,2*NN(1)
29            MSZ=NN(1)*NN(2)
30     7      IF(J.LT.MSZ) GO TO 6
31                J=J-MSZ
32                MSZ=MSZ/2
33            GO TO 7
34     6      J=J+MSZ
35            IF(J.GT.I) THEN
36 C             repeat column swap for all rows
37               DO 8 K=I,I+2*NN(1)-2,2
38 C                  repeat the column swap in each pair of rows
39                    T=J+(K-I)
40                    WRITE(6,991) K,K+1,T,T+1
41     8         CONTINUE
42            ENDIF
43     5 CONTINUE
44       STOP
45       END
```

Looking at the code reveals how the algorithm works in general. First consider the swapping of rows, which is carried out by the top stanza of executable statements. It processes the first column of `F` in the same way that the code we used in `FFT` processes its single column of data (compare $\boxed{\texttt{7-15}}$ here with $\boxed{\texttt{91-99}}$ on page 57). The number of elements in a column of `F` is `NN(1)`, so that variable plays the role that `NN` does in `FFT`. When the time comes to exchange rows `I` and `I+1` with rows `J` and `J+1` (as in $\boxed{\texttt{100-105}}$ on page 57) we now need to step across all the columns of `F`. To `FFT2ST`, `F` will look like one long vector `DATA`, so we must do the row exchanges by indexing into that vector. The indices this code uses for that purpose are `K` and `T`. In the first column of `F`, the first element to be exchanged is just `I`, so `K` starts $\boxed{17}$ equal to `I`. In the second column of `F`, the element corresponding to element `I` in the first column is `I` plus the stride between columns; thus, the increment of the loop over `K` $\boxed{17}$ is `2*NN(1)`. The last element in row `I` is `NN(2)-1` columns away at the index `I+2*NN(1)*NN(2)-2*NN(1)`, so that is the limit on `K`. In the first column of `F`, the element to exchange element `K` with is just element `J`, so $\boxed{19}$ `T` starts out as `J` when `K` takes on its first value of `I`. In the next column of `F`, `T` needs to be one stride farther along, which is just what happens when `K` gets incremented by that amount.

Once the rows have been put into the right order, the second stanza of code rearranges the columns. To see how it works, imagine the algorithm from `FFT` applied to the first *row* of `F`. For our example `I` must take on index values in `DATA` that we find listed across the top row of `F` on page 84. Those are `1`, `9`, `17`, `25`, `33`, `41`, `49`, and `57`, but since the first and last elements are always in the right places `I` needs to take on only the other values. There are `2*NN(1)` doublewords in the first column of `F`, so the starting value of `I` is `1+2*NN(1)` $\boxed{28}$ which for our example is `9`. The stride between columns is `2*NN(1)`, so that is the increment of the loop on `I`. The final value of `I` is the index in `DATA` of the top of the second-to-last column in `F`, so the upper limit of the `I` loop is `2*NN(1)*NN(2)-4*NN(1)+1`, which is `49` in our example. Now the highest index to be considered in the reordering process is the number of columns to be reordered times the stride (in `COMPLEX*16` values or doubleword-pairs) between them, or `NN(1)*NN(2)`, so `MSZ` begins with that value $\boxed{29}$; the remainder of the algorithm $\boxed{30\text{-}35}$ is the same as before. When the time comes to exchange column `I` with column `J` we need to step down the rows of `F`. In the first row of `F`, the first element to be exchanged is just `I`, so `K` starts $\boxed{37}$ equal to `I`. In the second row of `F`, the `COMPLEX*16` element corresponding to `I` in the first row has its first doubleword at `DATA(I+2)` because the elements of `F` are in column-major order. Thus the increment of the `K` loop is $\boxed{37}$ 2 doublewords. The last `COMPLEX*16` row in the column of `F` with `I` at its top is the one whose first doubleword is at index `I+2*NN(1)-2`, so that is the limit of the `K` loop $\boxed{37}$. In our example, if `I=33` is the index in `DATA` at the top of the column, the index in `DATA` of the first doubleword of the bottom `COMPLEX*16` in that column is `33+2*4-2=39`.

The code segments we have been discussing differ only in starting values, increments, and ending values of loops, and in their initial values of `MSZ`. The code at the top of the next page introduces variables to represent those quantities and sets them according to the dimension `IDIM` being processed, so that the same data rearrangement loops can be used for columns and rows. This program produces the same output as the earlier one. Although it is no shorter, it has the virtue that one instance of the data-rearrangement code is executed repeatedly in a loop over the dimensions. This will allow us to insert code for the butterfly calculation into that loop, in the place indicated, rather than having to repeat it after each stanza of the earlier program.

```
C     This version merges the loops by setting the appropriate
C     indexing parameters for each dimension.
C
      INTEGER*4 NN(2)/4,8/,T
C
      DO 1 IDIM=1,2
           IF(IDIM.EQ.1) THEN
              IST=3
              ILIM=2*NN(1)-3
              IINC=2
              KLIM=2*NN(1)*NN(2)-2*NN(1)
              KINC=2*NN(1)
              MSZST=NN(1)
           ELSE
              IST=1+2*NN(1)
              ILIM=2*NN(1)*NN(2)-4*NN(1)+1
              IINC=2*NN(1)
              KLIM=2*NN(1)-2
              KINC=2
              MSZST=NN(1)*NN(2)
           ENDIF
           J=1
           DO 2 I=IST,ILIM,IINC
                MSZ=MSZST
    4           IF(J.LT.MSZ) GO TO 3
                   J=J-MSZ
                   MSZ=MSZ/2
                GO TO 4
    3           J=J+MSZ
                IF(J.GT.I) THEN
C                  repeat for all rows or columns
                   DO 5 K=I,I+KLIM,KINC
                        T=J+(K-I)
                        WRITE(6,901) K,K+1,T,T+1
    5              CONTINUE
                ENDIF
    2      CONTINUE
  901      FORMAT('(',I2,',',I2,') <--> (',I2,',',I2,')')
C
C          [butterfly calculation for dimension IDIM will go here]
C
    1 CONTINUE
      STOP
      END
```

7.3.2 Butterfly Calculation

We can generalize the butterfly algorithm to two dimensions by taking the same approach we used for the data rearrangement algorithm. Thinking of the first column of F as a vector being processed by FFT, we can use the algorithm from that code and repeat each calculation across all the columns. Then, thinking of the first row of F as the vector being processed, we can revise the algorithm to take account of the stride between columns and repeat each calculation down all the rows.

The code segment at the top of the next page uses the butterfly algorithm from FFT for going down the first column of F, and propagates the calculation across all the columns. The number of stages in the signal-flow graph, LGN, is $\log_2$ (NN(1)) and will be found in the process of checking whether NN(1) is a power of 2. In FFT the limit on the P loop is NN=2*N, which corresponds to 2*NN(1) here, but the top of a butterfly could actually never be farther down the column than 2*NN(1)-3. I have omitted from the listing statements that are the same as in FFT.

```
C     this code goes down the first column of F (IDIM=1)
      LMAX=2
      DO 1 M=1,LGN
           :
           PSTEP=2*LMAX*1
           DO 2 L=1,LMAX,2
                DO 3 P=(L-1)*1+1,2*NN(1)-3,PSTEP
                     Q=P+LMAX*1
                     DO 4 K=P,P+2*NN(1)*NN(2)-2*NN(1),2*NN(1)
                          T=Q+(K-P)
                          [butterfly calculation]
    4                CONTINUE
    3           CONTINUE
                :
    2      CONTINUE
           LMAX=LMAX*2
    1 CONTINUE
```

This code finds P and Q just like FFT does, but I have written the formulas for PSTEP and P in a peculiar way, and I have separately doubled LMAX after the 2 loop rather than using PSTEP to save a multiplication as in FFT. These typographical oddities facilitate comparison with the code below for going across the first row of F. The butterfly calculation is repeated across the columns of F by the 4 loop, in which DATA(K) and DATA(T) participate in the butterfly. Elements in row P of F are separated by the column stride 2*NN(1), so that is the increment of K. The last element in row P is NN(2)-1 columns away, so the limit on K is P+2*NN(1)*NN(2)-2*NN(1). In the first column K=P and the bottom of the butterfly is at T=Q; as K increases by one column-stride with each iteration, so does T.

The code segment below modifies the butterfly algorithm of FFT to go across the first row of F, and repeats the calculation in all the rows.

```
C     this code goes across the first row of F (IDIM=2)
      LMAX=2
      DO 1 M=1,LGN
C          :
           PSTEP=2*LMAX*NN(1)
           DO 2 L=1,LMAX,2
                DO 3 P=(L-1)*NN(1)+1,2*NN(1)*NN(2)-4*NN(1)+1,PSTEP
                     Q=P+LMAX*NN(1)
                     DO 4 K=P,P+2*NN(1)-2,2
                          T=Q+(K-P)
                          [butterfly calculation]
    4                CONTINUE
    3           CONTINUE
                :
    2      CONTINUE
           LMAX=LMAX*2
    1 CONTINUE
```

Now LGN is the number of stages in the signal-flow graph for columns, $\log_2$ (NN(2)). The only other change that is needed in applying the butterfly algorithm to the first row of F is that stride must be taken into account in finding P and Q, the indices in DATA corresponding to the tops and bottoms of the butterflies.

In each stage the distance between the tops of the butterflies is now PSTEP=LMAX*2*NN(1). For example, in the signal-flow graph on page 34, in the first stage the butterflies involve adjacent samples and their tops are spaced apart by LMAX=2 samples; along the first row of F each sample heads a column, so the tops of the butterflies are spaced apart by LMAX=2 *columns* or LMAX*2*NN(1) elements in DATA. In the second stage the tops of the butterflies for each power of W are spaced apart by LMAX=4 samples; along the first row of F the tops of the butterflies are spaced apart by LMAX=4 columns or LMAX*2*NN(1) elements, and so on. This is the increment of the P loop.

In each stage we evaluate all the butterflies involving the zero'th power of the W for that stage (L=1), then all the butterflies with the first power of that W (L=3), and so on for all the powers of that W used in that stage. The first butterfly involving each power of W starts at index (L-1)*NN(1)+1 in DATA. For example, in the third stage of the signal-flow graph on page 34, LMAX=8 so L takes on the values 1, 3, 5, and 7. When L=1, the butterflies involve the zero'th power of W and the top of the first one is at DATA(1). When L=3, the butterflies involve the first power of W and the top of the first butterfly is the second sample. Along the first row of F the second sample is at the top of the second column, which has index (3-1)*NN(1)+1 in DATA. When L=5 the butterflies involve the second power of W and the top of the first butterfly is the third sample. Along the first row of F the third sample is at the top of the third column, which has index (5-1)*NN(1)+1 in DATA, and so on. Thus (L-1)*NN(1)+1 is the starting value of the P loop.

The biggest P can ever get is the index in DATA of the element at the top of the column second to last, or 2*NN(1)*NN(2)-4*NN(1)+1 (element 2*4*8-4*4+1=49 in our example), so this is the limit of the P loop.

The bottom of each butterfly is LMAX/2 samples away from its top. In the first stage of the signal-flow graph, LMAX=2 and adjacent samples are involved in the butterflies; in the second stage LMAX=4 and alternate samples are involved in the butterflies, and so on. Along the first row of F, the bottom of a butterfly is LMAX/2 *columns,* or (LMAX/2)*2*NN(1) elements in DATA, away from its top so Q=P+LMAX*NN(1).

The K loop propagates each butterfly calculation down all the rows. When we say that P is the index in DATA of the top of a butterfly, we mean that DATA(P) is the real part of that element of F and that DATA(P+1) is the imaginary part. Because the data are in column-major order, the top of the corresponding butterfly in the second row of F will be at DATA(P+2), and so on down the column, so the increment of the K loop is 2. The index of the last doubleword-pair in column P is P+2*NN(1)-2, so that is the limit on K.

The code segments we have been discussing differ only in starting values, increments, and ending values of loops, and in the multiplicative factor appearing in the formula for Q. The code below introduces variables to represent those quantities and sets them according to the dimension IDIM being processed, so that the same butterfly loops can be used for columns and rows.

```
C     This version merges the loops by setting the appropriate
C     indexing parameters for each dimension.
C
      INTEGER*4 NN(2)/4,8/,PLIM,PSTEP,P,Q,S,T
C
      DO 1 IDIM=1,2
           IF(IDIM.EQ.1) THEN
              MLT=1
              PLIM=2*NN(1)-3
              KLIM=2*NN(1)*NN(2)-2*NN(1)
              KINC=2*NN(1)
              LGN=2
           ELSE
              MLT=NN(1)
              PLIM=2*NN(1)*NN(2)-4*NN(1)+1
              KLIM=2*NN(1)-2
              KINC=2
              LGN=3
           ENDIF
C
           LMAX=2
           DO 2 M=1,LGN
                PSTEP=2*LMAX*MLT
                DO 3 L=1,LMAX,2
                     DO 4 P=(L-1)*MLT+1,PLIM,PSTEP
                          Q=P+LMAX*MLT
                          DO 5 K=P,P+KLIM,KINC
                               T=Q+(K-P)
                               WRITE(6,901) K,K+1,T,T+1
    5                     CONTINUE
    4                CONTINUE
    3           CONTINUE
                LMAX=LMAX*2
    2      CONTINUE
    1 CONTINUE
  901 FORMAT('(',I2,',',I2,') >< (',I2,',',I2,')')
      STOP
      END
```

```
( 1, 2) >< ( 3, 4)
( 9,10) >< (11,12)
(17,18) >< (19,20)
(25,26) >< (27,28)
(33,34) >< (35,36)
(41,42) >< (43,44)
(49,50) >< (51,52)
(57,58) >< (59,60)
( 5, 6) >< ( 7, 8)
(13,14) >< (15,16)
(21,22) >< (23,24)
(29,30) >< (31,32)
(37,38) >< (39,40)
(45,46) >< (47,48)
(53,54) >< (55,56)
(61,62) >< (63,64)
( 1, 2) >< ( 5, 6)
( 9,10) >< (13,14)
(17,18) >< (21,22)
(25,26) >< (29,30)
(33,34) >< (37,38)
(41,42) >< (45,46)
(49,50) >< (53,54)
(57,58) >< (61,62)
( 3, 4) >< ( 7, 8)
(11,12) >< (15,16)
(19,20) >< (23,24)
(27,28) >< (31,32)
(35,36) >< (39,40)
(43,44) >< (47,48)
(51,52) >< (55,56)
(59,60) >< (63,64)
( 1, 2) >< ( 9,10)
( 3, 4) >< (11,12)
( 5, 6) >< (13,14)
( 7, 8) >< (15,16)
(17,18) >< (25,26)
(19,20) >< (27,28)
(21,22) >< (29,30)
(23,24) >< (31,32)
(33,34) >< (41,42)
(35,36) >< (43,44)
(37,38) >< (45,46)
(39,40) >< (47,48)
(49,50) >< (57,58)
(51,52) >< (59,60)
(53,54) >< (61,62)
(55,56) >< (63,64)
( 1, 2) >< (17,18)
( 3, 4) >< (19,20)
( 5, 6) >< (21,22)
( 7, 8) >< (23,24)
(33,34) >< (49,50)
(35,36) >< (51,52)
(37,38) >< (53,54)
(39,40) >< (55,56)
( 9,10) >< (25,26)
(11,12) >< (27,28)
(13,14) >< (29,30)
(15,16) >< (31,32)
(41,42) >< (57,58)
(43,44) >< (59,60)
(45,46) >< (61,62)
(47,48) >< (63,64)
( 1, 2) >< (33,34)
( 3, 4) >< (35,36)
( 5, 6) >< (37,38)
( 7, 8) >< (39,40)
( 9,10) >< (41,42)
(11,12) >< (43,44)
(13,14) >< (45,46)
(15,16) >< (47,48)
(17,18) >< (49,50)
(19,20) >< (51,52)
(21,22) >< (53,54)
(23,24) >< (55,56)
(25,26) >< (57,58)
(27,28) >< (59,60)
(29,30) >< (61,62)
(31,32) >< (63,64)
```

When this program is run it produces the output shown to the right, which can be verified correct for our example by looking at the signal-flow graph on page 34 and the layout of DATA elements in F shown on page 84. The first butterfly listed involves the signal in F(1,1) (the real and imaginary parts of which are stored in DATA(1) and DATA(2)) and the signal in F(2,1) (stored in DATA(3) and DATA(4)). Then the same calculation is repeated across the columns of F. This pattern repeats for 3 more sets of 8 butterflies. Then, starting with (1, 2) >< (9,10) and its propagation down 3 more rows, there are a total of 12 sets of 4 butterflies for the columns.

7.3.3 Underfull Data Arrays

So far in this Section I have assumed the two-dimensional[19] `COMPLEX*16` data matrix `F` has exactly the right dimensions for the DFT we want to find. That way, when we pass its starting address for the one-dimensional `REAL*8 DATA` vector in `FFT2ST`, the data will be laid out in memory as described on page 78, with no gaps. This might seem natural, but in practice demanding that `F` be precisely the right size will be an inconvenience to the user when it is necessary to calculate two-dimensional DFTs of several sizes in one program, or when the size of a DFT is specified interactively as the program is run.[20] It would be handy if `FFT2ST` were smart enough to just skip over any unused doubleword-pairs at the ends of the columns. To make this possible we need only pass the **leading dimension** of `F` and use it in the indexing calculations for the stride between columns.[21] The array below is declared `COMPLEX*16 F(5,9)`, so its leading dimension is `5`.

F =

1	11	21	31	41	51	61	71	
2	12	22	32	42	52	62	72	
3	13	23	33	43	53	63	73	
4	14	24	34	44	54	64	74	
5	15	25	35	45	55	65	75	
6	16	26	36	46	56	66	76	
7	17	27	37	47	57	67	77	
8	18	28	38	48	58	68	78	

[19]The `fourn` routine given in [10, pages 518-519] generates all the necessary index pairs for the exchanges and butterflies in another order, by taking a fundamentally different approach that generalizes to any number of dimensions. Although I am skeptical that FFTs in more than 2 dimensions are needed very often, I will be grateful to anyone who explains to me how and why the indexing scheme used in `fourn` works.

[20]Dynamic memory allocation [7, §17.1.2] could be used to mitigate this problem, but it is bad for performance and unavailable in the Classical FORTRAN we have been using.

[21]The `DFT2` and `FFT2TR` routines receive and use the leading dimension (see [7, §7.1.2]) of `F` so that `F(1,KY)` or `F(1,NX)` will be the address of the top of a column even if the calling routine dimensions `F` bigger than it needs to be. But by always dealing with `F` as a two-dimensional array or always passing a single column at a time to `FFT`, both routines avoid any need for explicit indexing calculations involving `LX`.

But we are interested only in the upper-left `4` by `8` submatrix. The indices in `DATA` corresponding to the elements of `F` are still column-major in the whole array, so they differ from those on page 84. The final program in §7.3.1 and the final program in §7.3.2 each include an `IF-THEN-ELSE` block that sets certain loop parameters depending on which dimension `IDIM` is being processed. The code on the next page combines the two programs, merging their parameter-setting segments.

```
      INTEGER*4 NN(2)/4,8/,PLIM,PSTEP,P,Q,S,T,LD/5/
C
      DO 1 IDIM=1,2
           IF(IDIM.EQ.1) THEN
              IST=3
              IPLIM=2*NN(1)-3
              IINC=2
              KLIM=2*LD*NN(2)-2*LD
              KINC=2*LD
              MSZST=NN(1)
              MLT=1
              LGN=2
           ELSE
              IST=1+2*LD
              IPLIM=2*LD*NN(2)-4*LD+1
              IINC=2*LD
              KLIM=2*NN(1)-2
              KINC=2
              MSZST=LD*NN(2)
              MLT=LD
              LGN=3
           ENDIF
C
           J=1
           DO 2 I=IST,IPLIM,IINC
                MSZ=MSZST
    4           IF(J.LT.MSZ) GO TO 3
                   J=J-MSZ
                   MSZ=MSZ/2
                GO TO 4
    3           J=J+MSZ
                IF(J.GT.I) THEN
C                  repeat for all rows or columns
                   DO 5 K=I,I+KLIM,KINC
                        T=J+(K-I)
                        WRITE(6,901) K,K+1,T,T+1
    5              CONTINUE
                ENDIF
    2      CONTINUE
  901      FORMAT(’(’,I2,’,’,I2,’) <--> (’,I2,’,’,I2,’)’)
C
           LMAX=2
           DO 6 M=1,LGN
                PSTEP=LMAX*2*MLT
                DO 7 L=1,LMAX,2
                     DO 8 P=(L-1)*MLT+1,IPLIM,PSTEP
                          Q=P+LMAX*MLT
                          DO 9 K=P,P+KLIM,KINC
                               T=Q+(K-P)
                               WRITE(6,902) K,K+1,T,T+1
    9                     CONTINUE
    8                CONTINUE
    7           CONTINUE
                LMAX=LMAX*2
    6      CONTINUE
    1 CONTINUE
  902 FORMAT(’(’,I2,’,’,I2,’) >< (’,I2,’,’,I2,’)’)
C
      STOP
      END
```

The formulas for ILIM and PLIM are the same, so I have replaced both of those variables by IPLIM. I have also replaced some occurrences of NN(1) by the leading dimension LD, so that F can have 5 rows as shown above (the fact that it also has an extra column does not matter). This program reports the same exchanges and butterflies that we found separately before, but now the indices in DATA come out right for the picture on page 92.

To see how LD enters into the indexing, consider first what happens when IDIM=1. Then we are thinking of the first column of F as the data vector to be transformed, and the presence of an unused memory block at the end of that column has no effect on either the rearrangement or the butterfly calculation. Only when those calculations are replicated across the other columns does the actual length of the columns in F matter, so only the parameters of the K loops are affected. The largest K value of interest as the first element of an exchange or the top of a butterfly, KLIM, is still NN(2)-1 columns away, but now each column is LD long so KLIM=2*LD*NN(2)-2*LD. The stride KINC between columns is now the actual column length of 2*LD doublewords.

When IDIM=2, so that we are thinking of the first row of F as the data vector to be transformed, it is the indexing across the columns that is affected. Now the top of the second column is at index 1+2*LD in DATA, so that is IST. The top of the second-to-last column is at 2*LD*NN(2)-4*LD+1, so that is IPLIM (it comes out to 2*5*8-4*5+1=61 for our example). The increment between the tops of the columns, IINC, is now 2*LD. The propagation of the calculations down the rows of F does not depend on the length of the columns, so the K loop parameters are unaffected, but the distance PSTEP between the tops of butterflies gets multiplied by the actual column stride 2*LD as we progress from stage to stage in the signal-flow graph, so MLT=LD. In finding the columns to exchange it is still necessary to start with MSZ set to the number of columns to be rearranged times the stride in COMPLEX*16 data elements between them, so MSZST=LD*NN(2).

7.3.4 Library Subroutine

The indexing kernel on the previous page is embedded in the library subroutine FFT2ST that is listed on the next 6 pages (the DFT2 and FFT2TR routines are each less than 60% as long, but of course FFT2TR uses FFT to do most of its work). The calling sequence $\boxed{4}$ is the same as that for DFT2 and FFT2TR, so this routine can be used in place of either of those, and it can be tested using the same driver if only the routine name is changed. Doing that produces results identical to those found using FFT2TR for the pulse of §7.2, because they are ultimately obtained by identical arithmetic operations.

The parameter checking $\boxed{81\text{-}109}$ resembles that in FFT, but now both NX and NY must be powers of 2, and an added test $\boxed{118}$ ensures that F in the calling routine is big enough for the number of rows and columns in use. The scaling $\boxed{207\text{-}247}$ is like that in DFT2, and so is the frequency-shifting $\boxed{249\text{-}292}$ except that it must index into DATA to manipulate the COMPLEX*16 memory blocks of F one doubleword at a time.

Now we have three routines for computing the DFT of two-dimensional data, DFT2, FFT2TR, and FFT2ST. All are used in the same way and produce the same results, but we expect that FFT2TR and FFT2ST will both be faster than DFT2.

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE FFT2ST(LX,LY,NX,NY,INVDIR,SCALE,SHIFT,
     ;                  DATA,BX,BY, RC)
C     This routine computes in-place the direct or inverse fast
C     Fourier transform of the numbers in DATA, by using strides
C     to address the appropriate elements.  It assumes that DATA
C     points to an LX x LY COMPLEX*16 array F whose upper-left
C     corner contains the NX x NY matrix to be transformed.
C
C     variable  meaning
C     --------  -------
C     BX        limit of integration in X
C     BY        limit of integration in Y
C     DATA      input array, then its transform or inverse
C     DCOS      Fortran function returns cosine of REAL*8
C     DCMPLX    Fortran function returns COMPLEX*16 for two REAL*8s
C     DFLOAT    Fortran function returns REAL*8 for INTEGER*4
C     DSIN      Fortran function returns sine of a REAL*8
C     DX        sampling interval in X
C     DY        sampling interval in Y
C     I         index on DATA elements
C     IDIM      index on dimensions
C     IINC      increment on I in data rearrangement
C     INVDIR    +1 => direct transform, -1 => inverse transform
C     IPLIM     limit on I in data rearrangement, P in butterflies
C     IST       starting value of I in data rearrangement
C     J         second index on DATA elements for rearrangement
C     K         index on rows or columns for repeating swaps
C     KINC      increment on K in data rearrangement
C     KLIM      limit on K in data rearrangement
C     L         index on powers of W
C     LGN       log_2(NS(IDIM)) = stages in signal flow graph
C     LMAX      size of the transforms at this stage
C     LX        first dimension of F passed for DATA
C     LY        first dimension of F passed for DATA
C     M         index on stages in the signal flow graph
C     MLT       multiplier in formula for Q, P start and increment
C     MSZ       bit position of most significant 0 in an index value
C     MSZST     starting value of MSZ for this dimension
C     MZRO      bit pattern 10000000000000000000000000000000
C     NBITS     NN(IDIM) bit pattern shifted left
C     NN        [NX,NY]
C     NNPREV    values of NN on the previous call
C     NS        shifts to put NN(IDIM) 1 bit in sign position
C     NX        number of samples in X
C     NY        number of samples in Y
C     P         odd index of data pair at top of a butterfly
C     PSTEP     nodes between butterflies using the same power of W
C     Q         odd index of data pair at bottom of a butterfly
C     RC        return code; 0 => parameters make sense
C     SCALE     T => scale output or assume that input is scaled
C     SFX       scale factor in X
C     SFY       scale factor in Y
C     SHIFT     T => frequency-shift output or assume input shifted
C     T         location in DATA of butterfly element Q
C     TEMPI     temporary storage for real part of a DATA element
C     TEMPR     temporary storage for imag part of a DATA element
C     THETA     angle in W
C     TWOPI     twice circle constant
C     WI        imaginary part of W
C     WMI       imaginary part of W for this stage
C     WMR       real part of W for this stage
C     WR        real part of W
C     WTEMP     temporary in finding next power of W
```

```
C
C     formal parameters
      LOGICAL*4 SCALE,SHIFT
      REAL*8 DATA(2*LX*LY),BX,BY
      INTEGER*4 RC
C
C     local variables
      REAL*8 DX,DY,SFX,SFY,TEMPI,TEMPR,THETA,WI,WMI,WMR,WR,WTEMP
      REAL*8 TWOPI/6.2831853071795865D0/
      INTEGER*4 NN(2),NNPREV(2)/2*Z'80000000'/,LGN(2)
      INTEGER*4 MZRO/Z'80000000'/,P,PSTEP,Q,T
C
C ----------------------------------------------------------------
C
C     sanity-check the input parameters
      RC=0
C       each way the number of points must be a positive power of 2
        NN(1)=NX
        NN(2)=NY
        DO 1 IDIM=1,2
             IF(NN(IDIM).NE.NNPREV(IDIM)) THEN
C               this is a new NN; check it and find its log_2
                NBITS=NN(IDIM)
                DO 2 NS=1,32
                     IF(NBITS.LT.0) THEN
                        IF(NBITS.NE.MZRO) GO TO 3
                        IF(NS.EQ.1 .OR. NS.EQ.32) GO TO 3
                        LGN(IDIM)=32-NS
                        GO TO 1
                     ENDIF
                     NBITS=2*NBITS
    2           CONTINUE
    3           RC=1
             ENDIF
    1   CONTINUE
        IF(RC.EQ.0) THEN
C          they were both positive powers of 2; remember them
           NNPREV(1)=NX
           NNPREV(2)=NY
        ELSE
           NNPREV(1)=MZRO
           NNPREV(2)=MZRO
        ENDIF
C
C       the transform flag must denote either direct or inverse
        IF(INVDIR.NE.+1 .AND. INVDIR.NE.-1) RC=RC+2
C
C       the upper limit of integration was assumed positive
        IF(BX.LE.0.D0 .OR. BY.LE.0.D0) RC=RC+4
C
C       the adjustable dimensions must be big enough
        IF(LX.LT.NX .OR. LY.LT.NY) RC=RC+8
      IF(RC.NE.0) RETURN
C
```

```
C     step over the two dimensions
      DO 5 IDIM=1,2
C          set the parameters for this dimension
           IF(IDIM.EQ.1) THEN
C             going down the first column
              IST=3
              IPLIM=2*NX-3
              IINC=2
              KLIM=2*LX*NY-2*LX
              KINC=2*LX
              MSZST=NX
              MLT=1
           ELSE
C             going across the first row
              IST=1+2*LX
              IPLIM=2*LX*NY-4*LX+1
              IINC=2*LX
              KLIM=2*NX-2
              KINC=2
              MSZST=LX*NY
              MLT=LX
           ENDIF
C
C          arrange input sequence by ascending bit-reversed index
           J=1
           DO 6 I=IST,IPLIM,IINC
                MSZ=MSZST
    8           IF(J.LT.MSZ) GO TO 7
                   J=J-MSZ
                   MSZ=MSZ/2
                GO TO 8
    7           J=J+MSZ
                IF(J.GT.I) THEN
C                  repeat row swaps for all columns
                   DO 9 K=I,I+KLIM,KINC
C                       repeat the pair of row swaps in each column
                        T=J+(K-I)
                        TEMPR=DATA(K)
                        TEMPI=DATA(K+1)
                        DATA(K)=DATA(T)
                        DATA(K+1)=DATA(T+1)
                        DATA(T)=TEMPR
                        DATA(T+1)=TEMPI
    9              CONTINUE
                ENDIF
    6      CONTINUE
C
```

```
C          use the butterfly algorithm to evaluate signal flow graph
           LMAX=2
           DO 10 M=1,LGN(IDIM)
C                set W^0=1
                 WR=1.D0
                 WI=0.D0
C
C                compute the W for this stage
                 THETA=TWOPI/DFLOAT(INVDIR*LMAX)
                 WMR=DCOS(THETA)
                 WMI=DSIN(THETA)
C
C                consider each power of W used in this stage
                 PSTEP=2*LMAX*MLT
                 DO 11 L=1,LMAX,2
C                      butterflies for this power of W in all groups
                       DO 12 P=(L-1)*MLT+1,IPLIM,PSTEP
                             Q=P+LMAX*MLT
C                            repeat for all rows or columns
                             DO 13 K=P,P+KLIM,KINC
                                   T=Q+(K-P)
                                   TEMPR=WR*DATA(T)-WI*DATA(T+1)
                                   TEMPI=WR*DATA(T+1)+WI*DATA(T)
                                   DATA(T)=DATA(K)-TEMPR
                                   DATA(T+1)=DATA(K+1)-TEMPI
                                   DATA(K)=DATA(K)+TEMPR
                                   DATA(K+1)=DATA(K+1)+TEMPI
   13                        CONTINUE
   12                  CONTINUE
C
C                      find the next power of W as W*WP
                       WTEMP=WR
                       WR=WR*WMR-WI*WMI
                       WI=WI*WMR+WTEMP*WMI
   11            CONTINUE
                 LMAX=LMAX*2
   10      CONTINUE
    5 CONTINUE
C
```

```
C     find sampling intervals and new upper limits of integration
      IF(SCALE) THEN
         IF(INVDIR.EQ.+1) THEN
            DX=BX/DFLOAT(NX-1)
            DY=BY/DFLOAT(NY-1)
            IF(SHIFT) THEN
               BX=DFLOAT(NX/2)/(DX*DFLOAT(NX))
               BY=DFLOAT(NY/2)/(DY*DFLOAT(NY))
            ELSE
               BX=DFLOAT(NX-1)/(DX*DFLOAT(NX))
               BY=DFLOAT(NY-1)/(DY*DFLOAT(NY))
            ENDIF
         ELSE
            IF(SHIFT) THEN
               DX=2.D0*BX/DFLOAT(NX)
               DY=2.D0*BY/DFLOAT(NY)
            ELSE
               DX=BX/DFLOAT(NX-1)
               DY=BY/DFLOAT(NY-1)
            ENDIF
            BX=DFLOAT(NX-1)/(DX*DFLOAT(NX))
            BY=DFLOAT(NY-1)/(DY*DFLOAT(NY))
         ENDIF
         SFX=DX
         SFY=DY
      ELSE
         IF(INVDIR.EQ.+1) THEN
            SFX=1.D0
            SFY=1.D0
         ELSE
            SFX=1.D0/DFLOAT(NX)
            SFY=1.D0/DFLOAT(NY)
         ENDIF
      ENDIF
C
C     scale the output values
      IF(SFX.NE.1.D0 .OR. SFY.NE.1.D0) THEN
         DO 14 I=1,2*LX*LY
               DATA(I)=DCMPLX(SFX*SFY,0.D0)*DATA(I)
   14    CONTINUE
      ENDIF
C
```

```
C     shift, or compensate for having shifted before
      IF(SHIFT) THEN
         IF(INVDIR.EQ.1) THEN
C           frequency-shift forward transforms of columns
            DO 15 I=1,2*LX*NY-2*LX+1,2*LX
                  DATA(I+2*NX  )=DATA(I+NX  )
                  DATA(I+2*NX+1)=DATA(I+NX+1)
                  DO 16 J=I,I+NX-1,2
                        TEMPR=DATA(J)
                        TEMPI=DATA(J+1)
                        DATA(J)=DATA(J+NX)
                        DATA(J+1)=DATA(J+1+NX)
                        DATA(J+NX)=TEMPR
                        DATA(J+1+NX)=TEMPI
   16             CONTINUE
   15       CONTINUE
C           frequency-shift forward transforms of rows
            DO 17 I=1,2*(NX+1)-1,2
                  DATA(2*LX*NY+I  )=DATA(I+LX*NY  )
                  DATA(2*LX*NY+I+1)=DATA(I+LX*NY+1)
                  DO 18 J=I,I+LX*(NY-1),2*LX
                        TEMPR=DATA(J)
                        TEMPI=DATA(J+1)
                        DATA(J)=DATA(J+LX*NY)
                        DATA(J+1)=DATA(J+LX*NY+1)
                        DATA(J+LX*NY)=TEMPR
                        DATA(J+LX*NY+1)=TEMPI
   18             CONTINUE
   17       CONTINUE
         ELSE
C           fix up the inverse of the frequency-shifted transform
            DO 19 I=2*LX+1,2*LX*NY-NY+1,4*LX
                  DO 20 J=I,I+2*NX-2,2
                        DATA(J)=-DATA(J)
                        DATA(J+1)=-DATA(J+1)
   20             CONTINUE
   19       CONTINUE
            DO 21 I=3,2*NX-1,4
                  DO 22 J=I,I+2*LX*(NY-1),2*LX
                        DATA(J)=-DATA(J)
                        DATA(J+1)=-DATA(J+1)
   22             CONTINUE
   21       CONTINUE
         ENDIF
      ENDIF
      RETURN
      END
```

References

[1] **Burden, Richard L.**, **Faires, J. Douglas**, and **Reynolds, Albert C.**, *Numerical Analysis,* Prindle, Weber & Schmidt, 1978.

[2] **Cheney, Margaret** and **Borden, Brett**, *Fundamentals of Radar Imaging,* Society for Industrial and Applied Mathematics, 2009.

[3] **Close, Charles M.**, *The Analysis of Linear Circuits,* Harcourt, Brace & World, Inc., 1966.

[4] **Conte, S. D.** and **de Boor, Carl**, *Elementary Numerical Analysis: An Algorithmic Approach,* McGraw-Hill, 1972.

[5] **Heath, Michael T.**, *Scientific Computing: An Introductory Survey,* McGraw-Hill, 1997.

[6] **Kincaid, David** and **Cheney, Ward**, *Numerical Analysis: Mathematics of Scientific Computing*, Third Edition, American Mathematical Society, 2009.

[7] **Kupferschmid, Michael**, *Classical FORTRAN: Programming for Engineering and Scientific Applications,* Second Edition, CRC Press, 2009.

[8] **Lin, C. C.** and **Segel, L. A.**, *Mathematics Applied to Deterministic Problems in the Natural Sciences*, Macmillan Publishing Co., 1974.

[9] **Oppenheim, Alan V.** and **Schafer, Ronald W.**, *Digital Signal Processing,* Prentice-Hall, 1975.

[10] **Press, William H.**, **Teukolosky, Saul A.**, **Vetterling, William T.**, and **Flannery, Brian P.**, *Numerical Recipes in FORTRAN: The Art of Scientific Computing,* Second Edition, Cambridge University Press, 1992.