9

Discrete and digital Fourier transforms

9.1 History

Fourier transformation is formally an analytic process which uses integral calculus. In experimental physics and engineering, however, the integrand may be a set of experimental data, and the integration is necessarily done artificially. Since a separate integration is needed to give each point of the transformed function, the process would become exceedingly tedious if it were to be attempted manually, and many ingenious devices have been invented for performing Fourier transforms mechanically, electrically, acoustically and optically. These are all now part of history since the arrival of the digital computer and more particularly since the discovery – or invention – of the ‘fast Fourier transform’ algorithm or FFT as it is generally called. Using this algorithm, the data are put (‘read’) into a file (or ‘array’, depending on the computer jargon in use), the transform is carried out, and the array then contains the points of the transformed function. It can be achieved by a software program, or by a purpose-built integrated circuit. It can be done very quickly so that vibration-sensitive instruments with Fourier transformers attached can be used for tuning pianos and motor engines, for aircraft and submarine detection and so on. It must not be forgotten that the ear is Nature’s own Fourier transformer,¹ and, as used by an expert piano-tuner, for example, is probably the equal of any electronic simulator in the 20–20 000-Hz range. The diffraction grating, too, is a passive Fourier transformer device, provided that it is used as a spectrograph taking full advantage of the simultaneity of outputs.

The history of the FFT is complicated and has been researched by Brigham² and, as with many discoveries and inventions, it arrived before the (computer) world was ready for it. Its digital apotheosis came with the publication

¹ It detects the power transform, and is not sensitive to phase.
of the ‘Cooley–Tukey’ algorithm\(^3\) in 1965. Since then other methods have been virtually abandoned except for certain specialized cases and this chapter is a description of the principles underlying the FFT and how to use it in practice.

### 9.2 The discrete Fourier transform

There is a pair of formulae by which sets of numbers \([a_n]\) and \([A_m]\), each set having \(N\) elements, can be mutually transformed:

\[
A(m) = \frac{1}{N} \sum_{n=0}^{N-1} a(n)e^{2\pi inm/N}; \quad a(n) = \sum_{m=0}^{N-1} A(m)e^{-2\pi inm/N}. \tag{9.1}
\]

In appearance and indeed in function, these are very similar to the formulae of the analytic Fourier transform and are generally known as a ‘discrete Fourier transform’ (DFT). They can be associated with the true Fourier transform by the following argument.

Suppose, as usual, that \(f(x)\) and \(\phi(p)\) are a Fourier pair. If \(f(x)\) is multiplied by a \(\delta\)-function of period \(a\) then the Fourier transform becomes

\[
\Phi(p) = \int_{-\infty}^{\infty} f(x)\delta(x/a)e^{2\pi ipx} \, dx = (1/a)[\phi(p) * \delta_{1/a}(p)].
\]

Now suppose that \(f(x)\) is negligibly small for all \(x\) outside the limits \(-a/2 \rightarrow (N - 1/2)a\), so that there are \(N\) teeth in the Dirac comb, and \(f(x)\) extends over a range \(\leq Na\). We rewrite the integral and use the properties of \(\delta\)-functions so that

\[
\Phi(p) = \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(x)e^{2\pi ipx} \delta(x - na) \, dx = \sum_{n=-\infty}^{\infty} \int_{-\infty}^{\infty} f(x)e^{2\pi ipx} \delta(x - na) \, dx.
\]

Because there are only \(N\) teeth in the comb, the sum is finite and the integral means substituting the argument of the \(\delta\)-function as usual.

\[
\Phi(p) = \sum_{n=0}^{N-1} f(na)e^{2\pi ipna} = (1/a)[\phi(p) * \delta_{1/a}(p)].
\]

---

This in turn is periodic in \( p \) with period \( 1/a \), and can be written

\[
\Phi(p) = (1/a)\phi(p) \ast \mathbb{W}_{1/a}(p) = (1/a)(\phi(p) + \phi(p + 1/a) + \phi(p - 1/a) + \phi(p + 2/a) + \phi(p - 2/a) + \cdots),
\]

and in its first period \( \Phi(p) \) is the same as the analytic function \( (1/a)\phi(p) \).

Now consider \( n \) small intervals of \( p \), each of width \( 1/(Na) \). At the \( m \)th such interval the equation becomes

\[
\Phi(m/(Na)) = \sum_{n=0}^{N-1} f(na)e^{2\pi ina(m/(Na))} = (1/a)\phi(m/(Na))
\]
or, more succinctly,

\[
\sum_{n=0}^{N-1} f(n)e^{2\pi inm/N} = (1/a)\phi(m),
\]

and this approximates to the analytic Fourier transform. The approximation is that in its first period the periodic \( \Phi(p) = \phi(p) \). Theoretically it is not – there is bound to be some overlap since \( \phi(p) \) is not zero – but practically the discrepancy can be ignored.\(^4\)

The choice of the interval \(-a/2 \rightarrow (N-1/2)a\) for \( f(x) \) is so as to have exactly \( N \) teeth in the Dirac comb without the embarrassment of having teeth at the very edge – where a top-hat function changes from 1 to 0, for example. In theory any interval of the same length would do.

### 9.3 The matrix form of the DFT

One way of looking at the formula for the discrete Fourier transform is to set it out as a matrix operation. The data set \([a(n)]\) can be written as a column matrix or ‘vector’ (in an \( N \)-dimensional space), to be multiplied by a square matrix containing all the exponentials and giving another column matrix with \( N \) components, \([A(m)]\), as its result:

\[
\begin{bmatrix}
  A(0) \\
  A(1) \\
  A(2) \\
  A(3) \\
  \vdots \\
  A(N-1)
\end{bmatrix}
= \begin{bmatrix}
  1 & 1 & 1 & \cdots & 1 \\
  e^{2\pi i/N} & e^{4\pi i/N} & \cdots & e^{2(N-1)\pi i/N} \\
  e^{4\pi i/N} & e^{8\pi i/N} & \cdots & e^{4(N-1)\pi i/N} \\
  e^{6\pi i/N} & e^{12\pi i/N} & \cdots & e^{6(N-1)\pi i/N} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  1 & \cdots & \cdots & \cdots & e^{(N-1)^2\pi i/N}
\end{bmatrix}
\begin{bmatrix}
  a(0) \\
  a(1) \\
  a(2) \\
  a(3) \\
  \vdots \\
  a(N-1)
\end{bmatrix}.
\]

\(^4\) It is not possible for a function and its Fourier pair both to be finite in extent – one at least must extend to \( \pm \infty \) – but the condition that both be small compared with the values in the region of interest is allowable.
The process of matrix multiplication requires \( n^2 \) multiplications for its completion. If large amounts of data are to be processed, this can become inordinate, even for a computer. Some people like to process columns of data with \( 10^6 \) numbers occasionally, but normally experimenters make do with 1024, although they often require the transform in a few microseconds.

The secret of the fast Fourier transform is that it reduces the number of multiplications to be done from \( N^2 \) to about \( 2N \log_2(N) \). A data ‘vector’ \( 10^6 \) numbers long then requires \( \frac{4 \cdot 2^{10}}{10^7} \) multiplications instead of \( 10^{12} \), a gain in speed by a factor of approximately 26200. In this year of grace 2010, the computation time on a desktop computer is reduced from about 2 minutes to a few microseconds.

The way it does this is, in essence, to factorize the matrix of exponentials, but there are easier ways of looking at the process. For example, suppose that the number \( N \) of components in the vector is the product of two numbers \( k \) and \( l \). Instead of writing the subscript of each number in the vector to denote its position (0...\( N-1 \)), it can be given two subscripts \( s \) and \( t \), and written \( a(s, t) \), with \( a(s, t) = a(sk + t) \), where \( s \) takes values from 0 to \( (l-1) \) and \( t \) runs from 0 to \( (k-1) \). In this way all the numbers in the vector are labelled, but now with two suffixes instead of one. There is absolutely no point in doing this except for computational purposes: it is purely a piece of computer-mathematical manipulation, and would have struck mathematicians of pre-computer days as ludicrous. However, we now write the digital transform as

\[
A(u, v) = \sum_{s=0}^{l-1} \sum_{t=0}^{k-1} a(s, t) e^{2\pi i (sk + t)(ul + v)/kl},
\]

where the suffix \( m \) in the transformed vector has similarly been dissected into \( u \) and \( v \), with \( m = ul + v \). The suffix \( u \) runs from 0 to \( (k-1) \) and \( v \) runs from 0 to \( (l-1) \).

The exponent is now multiplied out and gives

\[
A(u, v) = \sum_{s=0}^{l-1} \sum_{t=0}^{k-1} a(s, t) e^{2\pi i su} e^{2\pi i sv/l} e^{2\pi i tu/k} e^{2\pi i vt/(kl)}.
\]

The first exponential factor is unity and is discarded. The double sum can be rewritten now as

\[
A(u, v) = \sum_{t=0}^{l-1} e^{2\pi i tu/k} e^{2\pi i vt/(kl)} \sum_{s=0}^{k-1} a(s, t) e^{2\pi i sv/l},
\]

which is legitimate since only the last exponent contains a factor of \( s \).
This sum over \(k\) terms gives a new set of numbers \([g(v, t)]\) and we write

\[
A(u, v) = \sum_{t=0}^{l-1} [g(v, t)e^{2\pi ivt/(kl)}]e^{2\pi itu/k}.
\]

The array \([g(v, t)]\) is multiplied by \(e^{2\pi ivt/(kl)}\) to give an array \([g'(v, t)]\) and finally the sum

\[
g''(v, u) = \sum_{t=0}^{l-1} g'(v, t)e^{2\pi itu/k}
\]

and \(g''(v, u) = A(u, v)\). (The reversing of the order of \(v\) and \(u\) is important.)

The transform has been split into two stages. There are \(k\) transforms, each of length \(l\), followed by \(N\) multiplications by the exponential factors \(e^{2\pi ivt/(kl)}\) (the ‘twiddle-factors’), followed by \(l\) transforms, each of length \(k\): a total of \(kl^2 + lk^2 = N(k + l)\) multiplications, apart from the relatively small number, \(N\), of multiplications (by \(e^{2\pi ivt/(kl)}\)) in the middle.

The lesson is that, provided \(N\) can be factorized, the vector \([a(n)]\) can be turned into a rectangular \(k \times l\) matrix and treated column by column as a set of shorter transforms. For example, if there were a factor of 2, the even-numbered \(a\)’s could be put into one vector of length \(N/2\) and the odd-numbered \(a\)’s into another. Then each is subjected to a Fourier transform of half the length to give two more vectors, and these, after multiplying by the ‘twiddle-factors’ as above, can be recombined into a vector of length \(N\).

The same process can be repeated, provided that \(N/2\) can be factorized; and if the factors are always 2, it continues until only \(2 \times 2\) matrices are left, with trivially easy Fourier transforms (and a multiplicity of twiddle-factors!). The interesting thing is that each number in the transformed vector has its address in bit-reversed order. In the example given earlier the final outcome was \(g''(v, u)\), so that the two indices have to be reversed – the number \(g''(v, u)\) is in the wrong place in the array. This effect is multiplied until, in the \(2^N\) transform, the transformed data appear in the wrong addresses, the true address being the bit-reversed order of the apparent address.

The fast Fourier transform is thus usually done with \(N\) a power of 2. This is not only very efficient in terms of computing time, but also ideally suited to the binary arithmetic of digital computers. The details of the way programs are written are given by Brigham\(^5\) and a BASIC listing of an FFT routine is given at the end of this chapter. There are many such routines, the results of many hours of research, and sometimes they are very efficient. This one is not

Fig. 9.1. The implementation of the FFT using a sinc-function as an example. The two cylinders, unwrapped, represent the input and output data arrays. Do not expect zero to be in the middle as in the analytic case of a Fourier transform. If the input data are symmetrical about the centre, these two halves must be exchanged (en-bloc, not mirror-imaged) before and after doing the FFT.

particularly fast but will suffice for practice and is certainly suitable for student laboratory work.

The data file for this program must be 2048 words long (1024 complex numbers, alternately real and imaginary parts), and, if only real data are to be transformed, they should go in the even-numbered elements of the array, from 0 to 2046. Some caution is needed: zero frequency is at array element 0. If you want to Fourier transform a sinc-function, for example, the positive part of the function should go at the beginning of the array and the negative part at the end. Figure 9.1 illustrates the point: the output will similarly contain the zero-frequency value in element 0, so that the top-hat appears to be split between the beginning and the end.

Alternatively, you can arrange to have zero frequency at point 1024 in the array, in which case the input and output arrays must both be transposed, by having the first and second halves interchanged (but not flipped over) before and after the FFT is done.

Attention to these details saves a lot of confusion! It helps to think of the array as wrapped around a cylinder, with the beginning of the array at zero frequency and the end at point \((-1\) instead of \((+1023)\).

9.3.1 Two-dimensional FFTs

Two-dimensional transforms can be done using the same routines. The data are in a rectangular array of ‘pixels’ which form the picture which is to be transformed. Each row should first have its right and left halves transposed. Then each column must have the top and bottom halves transposed, so that what was perhaps a circle in the middle of the picture becomes four quadrants, one in
each corner. Then each row is given the FFT treatment. Then each column in
the resulting array gets the same. Then the rows and finally the columns are
transposed again to give the complete FFT. At this stage periodic features, such
as a TV raster, for example, will appear as Dirac nails (provided that the original
picture has been sampled often enough) and can be suppressed by altering the
contents of the pixels where they appear. Then the whole procedure is reversed
to give the whole ‘clean’ picture.

Apodizing functions can similarly be applied to remove false information,
to smooth edges and to improve the picture cosmetically.

Obviously far more elaborate techniques than this have been developed, but
this is the basis of the whole process.

The output can be used in a straightforward way to give the power, phase
or modular transforms, and the data can be presented graphically with simple
routines which need no description here.

### 9.4 A BASIC FFT routine

FFT routines can be routinely downloaded from the Web, so that observational
or experimental data can be loaded into them, the handle pulled and, like magic,
out comes the Fourier transform. However, there are many people who like to
enter the computational fray at a more fundamental level, to load their own
FFT routine into a BASIC, FORTRAN or C++ program and experiment with
it. Translation of the instructions between one and another is relatively simple
and so I have resisted the urging of colleagues to delete the BASIC routine
which was given in previous editions.

#### 9.4.1 A routine for 1024 complex numbers

The listing below is of a simple BASIC routine for the fast Fourier transform
of 1024 complex numbers.\(^6\) This is a routine which can be incorporated into a
program which you can write for yourself.

The data to be transformed are put in an array D(I) declared at the begin-
ning of the program as ‘DIM D(2047)’. The reals go in the even-numbered
places, beginning at 0, and the imaginaries in the odd-numbered places. The
transformed data are found similarly in the same array. The variable G on line
131 should be set to 1 for a direct transform and to \(-1\) for an inverse transform.

Numbers to be entered into the D(I) array should be in ASCII format. The

\(^6\) But \(N\) can be changed by changing the first line of the program.
program should fill the D(I) array with data; call the FFT as a routine with a ‘GOSUB 100’ statement (the ‘RETURN’ is the last statement, on line 10), and this can be followed by instructions for displaying the data.

It is well worth your while to incorporate a routine for transposing the two halves of the D(I) array before and after doing the transform, as an aid to understanding what is happening.

```
100 N=2048        REM for 1024 complex points
PRINT "BEGIN FFT"  transform.
J=1
G=1        REM for direct transform. G=-1
FOR I=1 TO N STEP 2 for inverse.
IF (I-J)<0 GOTO 1
IF I=J GOTO 2
IF (I-J)>0 GOTO 2
1 T=D(J-1)
S=D(J)
D(J-1)=D(I-1)
D(J)=D(I)
D(I-1)=T
D(I)=S
2 M=N/2
3 IF (J-M)<0 GOTO 5
IF J=M GOTO 5
IF (J-M)>0 GOTO 4
4 J=J-M
M=M/2
IF (M-2)<0 GOTO 5
IF M=2 GOTO 3
IF(M-2)>0 GOTO 3
5 J=J+M
NEXT I
X=2
IF (X-N)<0 GOTO 7
6 IF X=N GOTO 8
IF (X-N)>0 GOTO 8
7 F=2*X
H=6.28319/(G*X)
R=SIN(H/2)
W=−2*R*R
```
9.4 A BASIC FFT routine

V = SIN(H)
P = 1
Q = 0
FOR M = 1 TO X STEP 2
FOR I = M TO N STEP F
J = I + X
T = P*D(J-1) - Q*D(J)
S = P*D(J) + Q*D(J-1)
D(J-1) = D(I-1) - T
D(J) = D(I) - S
D(I-1) = D(I-1) + T
D(I) = D(I) + S
NEXT I
T = P
P = P*W - Q*V + P
Q = Q*W + T*V + Q
NEXT M
X = F
GOTO 6

8 CLS
FOR I = 0 TO N-1
D(I) = D(I)/SQR(N/2)
NEXT I
PRINT "FFT DONE"
10 RETURN

Next, here is a short program to generate a file with .DAT extension which will contain a top-hat function of any width you choose. The data are generated in ASCII and can be used directly with the FFT program above.

REM Program to generate a “Top-hat” function.
INPUT “input desired file name”, A$
INPUT ‘Top-hat Half-width ?’, N
PI = 3.141592654
DIM B(2047)
FOR I = 1024-N TO 1024+N STEP 2
B(I) = 1/(2*N)
NEXT I
C$ = “.DAT”
C$ = A$ + C$
PRINT
OPEN CS FOR OUTPUT AS #1
FOR I=0 TO 2047
PRINT #1,B(I)
NEXT I
CLOSE #1

The simple file-generating arithmetic in lines 6–8 can obviously be replaced by something else, and this sort of ‘experiment’ is of great help in understanding the FFT process.

The file thus generated can be read into the FFT program with the following:

REM Subroutine FILELOAD
REM To open a file and load contents into D(I)
GOSUB 24
(insert the next stage of the program, e.g. ‘GOSUB 100’, here)
CLS:LOCATE 10,26,0
PRINT “NAME OF DATA FILE ?”
LOCATE 14,26,0
INPUT A$
ON ERROR GOTO 35
OPEN “I”,#1,A$
FOR I=0 TO 2047
ON ERROR GOTO 35
INPUT #1,D(I)
NEXT I
CLOSE I
35 RETURN