September 7, 2000

Fourier Transforms

1 Finite Fourier Transform

Any discussion of finite Fourier transforms and MATLAB immediately encounters a notational issue – we have to be careful about whether the subscripts start at zero or one. The usual notation for finite Fourier transforms uses subscripts j and k that run from 0 to $n-1$. MATLAB uses notation derived from matrix theory where the subscripts run from 1 to n, so we will use y_{j+1} for mathematical quantities that will also occur in MATLAB code. We will reserve i for the complex unit, $\sqrt{-1}$.

The finite, or discrete, Fourier transform of a complex vector y with n elements $y_{j+1}, j = 0, \ldots n-1$ is another complex vector Y with n elements

$$Y_{k+1} = \sum_{j=0}^{n-1} y_{j+1} e^{-2ijk\pi/n},\ k = 0, \ldots, n-1$$

The exponentials are all complex n-th roots of unity, i.e. they are all powers of

$$\omega = e^{-2\pi i/n} = \cos\delta - i\sin\delta$$

where $\delta = 2\pi/n$. The transform can also be expressed with matrix-vector notation

$$Y = Fy$$

where the finite Fourier transform matrix F has elements

$$f_{k+1,j+1} = \omega^{jk}$$

It turns out that F is nearly its own inverse. More precisely, the complex conjugate transpose of F satisfies

$$F^H F = nI$$

so

$$F^{-1} = \frac{1}{n} F^H$$

This allows us to invert the Fourier transform.

$$y = \frac{1}{n} F^H Y$$

Hence

$$y_{j+1} = \frac{1}{n} \sum_{k=0}^{n-1} Y_{k+1} e^{2ijk\pi/n},\ j = 0, \ldots, n-1$$

We should point out that this is not the only notation for finite Fourier transform in common use. The minus sign in the complex exponentials in the first equation, and in the definiton of ω, sometimes occurs in the inverse transpose instead. And the $1/n$ scaling factor in the inverse transform is sometimes replaced by $1/\sqrt{n}$ scaling factors in both transforms.

In MATLAB, the Fourier matrix F could be generated for any given `n` by

```
omega = exp(-2*pi*i/n);
j = 0:n-1;
k = j'
F = omega.^(k*j)
```

The quantity `k*j` is an *outer product*, an n-by-n matrix whose elements are the products of the elements of two vectors. However, the built-in function `fft` takes the finite Fourier transform of each column of a matrix argument, so an easier, and quicker, way to generate F is

```
F = fft(eye(n))
```

The function `fft` uses a fast algorithm to compute the finite Fourier transform. The first "`f`" stands for both "fast" and "finite". A more accurate name might be `ffft`, but nobody wants to use that. We will discuss the fast aspect of the algorithm in a later section.

2 fftshow

The GUI `fftshow` allows you to investigate properties of the finite Fourier transform. If `y` is a vector containing a few dozen elements.

```
fftshow(y)
```

produces four plots

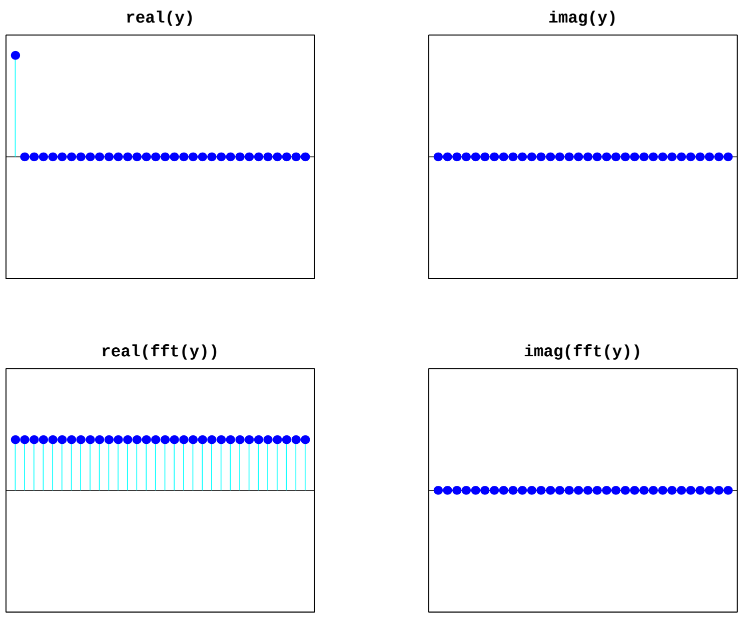
```
real(y)          imag(y)
real(fft(y))     imag(fft(y))
```

You can use the mouse to move any of the points in any of the plots and the points in the other plots will respond.

Please run `fftshow` and try the following examples. Each illustrates some property of the Fourier transform. When you start with no arguments

```
fftshow
```

all four plots are initialized to `zeros(1,32)`. Click your mouse in the upper left hand corner of the upper left hand plot. You will be taking the `fft` of the first unit vector, with one in the first component and zeros elsewhere. This should produce



The real part of the result is constant and the imaginary part is zero. You can also see this from the definition

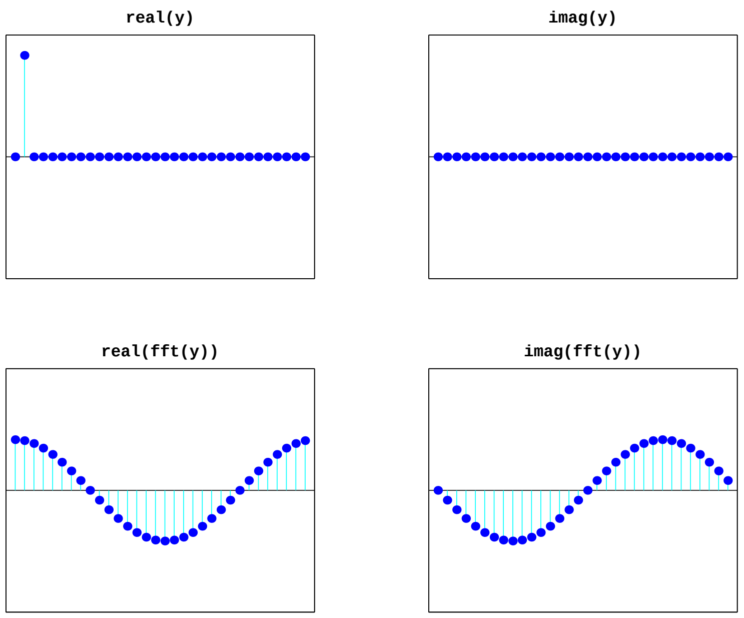
$$Y_{k+1} = \sum_{j=0}^{n-1} y_{j+1} e^{-2ijk\pi/n}, \ k = 0, \ldots, n-1$$

when $y_1 = 1$ and $y_2 = \cdots = y_n = 0$. The result is

$$Y_{k+1} = 1 \cdot e^0 + 0 + \cdots + 0 = 1, \ \text{for all } k$$

Click on y_1 again, hold the mouse down, and move the mouse vertically. The amplitude of the constant result varies accordingly.

Next, try the second unit vector. Use the mouse to set $y_1 = 0$ and $y_2 = 1$. This should produce



You are seeing the graph of

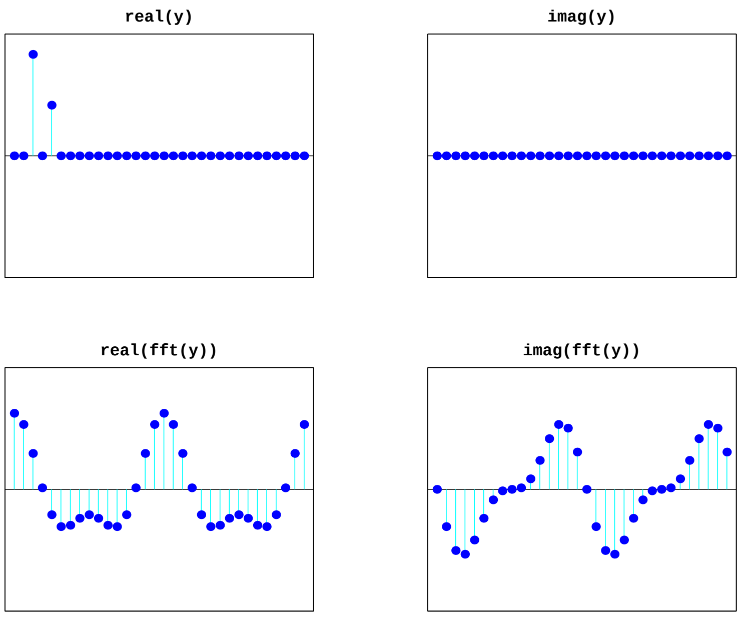
$$Y_{k+1} = 0 + 1 \cdot e^{-2ik\pi/n} + 0 + \cdots + 0$$

Using $\delta = 2\pi/n$,

$$\text{real}(Y_{k+1}) = \cos k\delta, \ \ \text{imag}(Y_{k+1}) = -\sin k\delta$$

for $k = 0, \cdots, n-1$. We have sampled two trig functions at n equally spaced points in the interval $0 \le x < 2\pi$. The first sample point is $x = 0$ and the last sample point is $x = 2\pi - \delta$.

Now set $y_3 = 1$ and vary y_5 with the mouse. One snapshot is



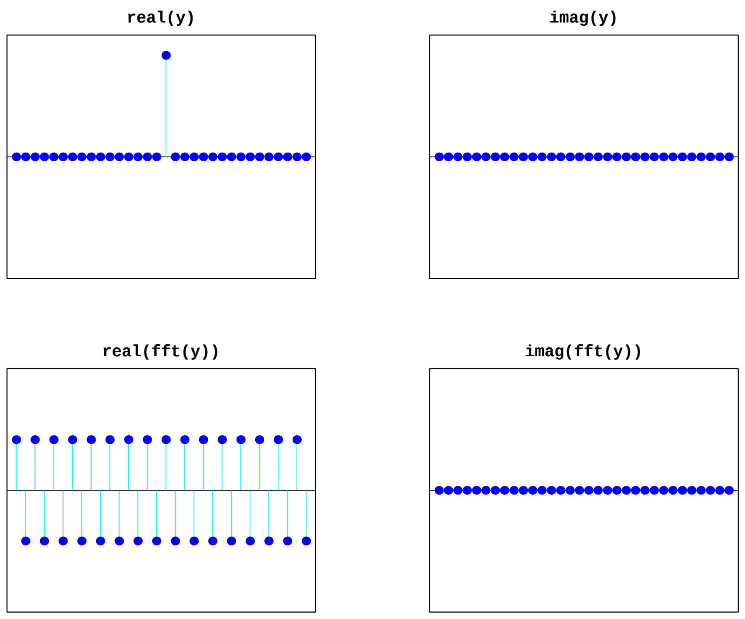
We have graphs of

$$\cos 2k\delta + \eta \cos 4k\delta, \text{ and } -\sin 2k\delta - \eta \sin 4k\delta$$

for various values of $\eta = y_5$

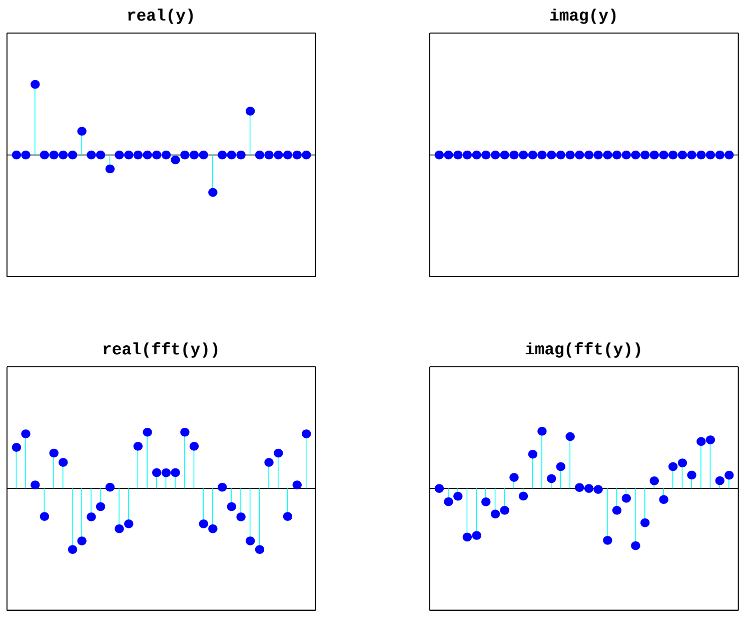
The point just to the right of the center of these graphs is particularly important. We will call it the *Nyquist point*. With the points numbered from 1 to n for even n, it's the point with index $\frac{n}{2}+1$. When $n = 32$, it's point number 17, just under the left parenthesis in the title.

Here is `fftshow` with a unit vector at the Nyquist point.



The `fft` is a sequence of alternating +1's and -1's.

Now let's look at some symmetries in the FFT. Make several random clicks on the `real(y)` plot. Leave the `imag(y)` plot flat zero. Here is an example.



Look carefully at the two `fft` plots. Ignoring the first point in each plot, the real part is symmetric about the Nyquist point and the imaginary part is antisymmetric about the Nyquist point. More precisely, if y is any real vector of length n and $Y = \text{fft}(y)$, then

$$\begin{aligned}
\text{real}(Y_1) &= \sum y_j \\
\text{imag}(Y_1) &= 0 \\
\text{real}(Y_{2+j}) &= \text{real}(Y_{n-j}),\ j = 0, \cdots, n/2-1 \\
\text{imag}(Y_{2+j}) &= -\text{imag}(Y_{n-j}),\ j = 0, \cdots, n/2-1
\end{aligned}$$

3 Sunspots

This section is an expansion of MATLAB's `sunspots` demo.

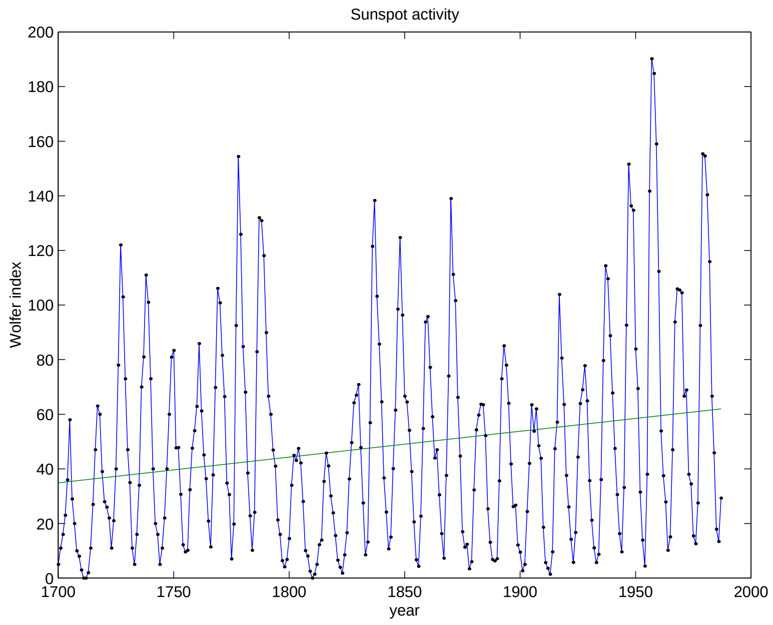
For centuries people have noted that the face of the sun is not constant or uniform in appearance, but that darker regions appear at random locations on a cyclical basis. This activity is correlated with weather and other economically significant terrestrial phenomena. In 1848, Rudolf Wolfer proposed a rule that combined the number and size of these sunspots into a single index. Using archival records, astronomers have applied Wolfer's rule to determine sunspot activity back to the year 1700. Today the sunspot index is measured by many astronomers and the worldwide distribution of the data is coordinated by the Sunspot Index Data Center at the Royal Observatory of Belgium. (See `http://www.astro.oma.be/SIDC/index.html`)

The text file `sunspot.dat` in MATLAB's `demos` directory has two columns of numbers. The first column is the years from 1700 to 1987 and the second column is the average Wolfer sunspot number for each year.

```
load sunspot.dat
t = sunspot(:,1)';
wolfer = sunspot(:,2)';
n = length(wolfer)
```

There is a slight upward trend to the data. A least squares fit gives the trend line.

```
c = polyfit(t,wolfer,1);
trend = polyval(c,t);
plot(t,[wolfer; trend],'-',t,wolfer,'k.')
xlabel('year')
ylabel('Wolfer index')
title('Sunspot index with linear trend')
```



You can definitely see the cyclic nature of the phenomenon. The peaks and valleys are a little more than 10 years apart.

Now, subtract off the linear trend and take the finite Fourier transform.

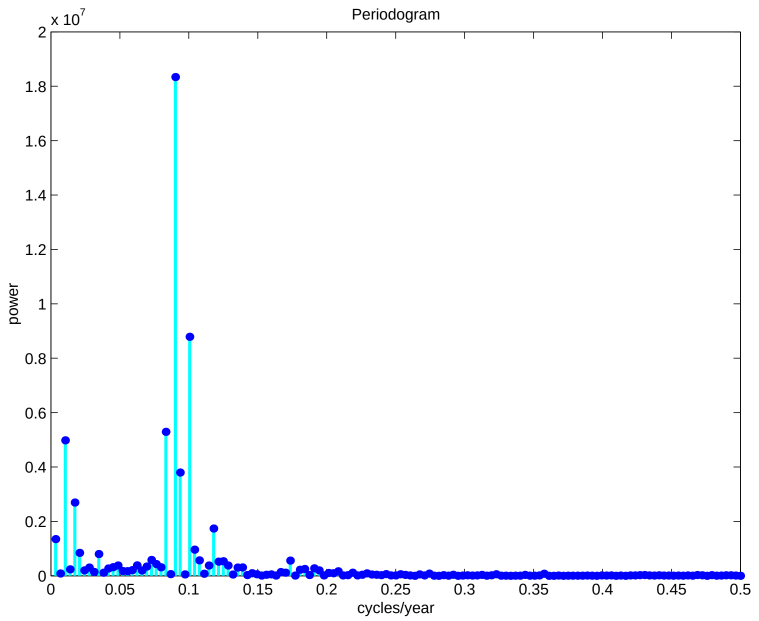
```
y = wolfer - trend;
Y = fft(y);
```

The first Fourier coefficient, `Y(1)`, can be deleted because subtracting the linear trend insures that `Y(1) = sum(y)` is zero.

```
Y(1) = [];
```

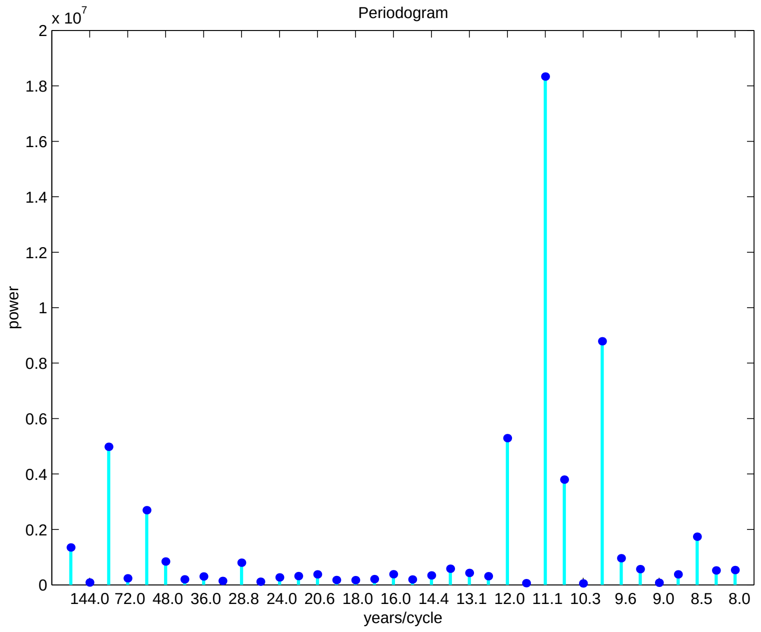
The complex magnitude squared of Y is called the power and a plot of power versus frequency is a "periodogram". The frequency is the array index scaled by n, the number of data points. Since the time increment is one year, the frequency units are cycles per year.

```
pow = abs(Y(1:n/2)).^2;
pmax = 20e6;
f = (1:n/2)/n;
plot([f; f],[0*pow; pow],'c-', f,pow,'b.', ...
   'linewidth',2,'markersize',16)
axis([0 .5 0 pmax])
xlabel('cycles/year')
ylabel('power')
title('Periodogram')
```



The maximum power occurs near frequency = 0.09 cycles/year. We would like to know the corresponding period in years/cycle. Let's zoom in on the plot and use the reciprocal of frequency to label the x-axis.

```
k = 1:36;
pow = pow(k);
ypk = n./k(2:2:end);  % Years per cycle
plot([k; k],[0*pow; pow],'c-',k,pow,'b.', ...
   'linewidth',2,'markersize',16)
axis([0 max(k)+1 0 pmax])
set(gca,'xtick',k(2:2:end))
xticklabels = sprintf('%5.1f|',ypk);
set(gca,'xticklabel',xticklabels)
xlabel('years/cycle')
ylabel('power')
title('Periodogram')
```



As expected, there is a very prominent cycle with a length of about 11 years. This shows that over the last 300 years, the period of the sunspot cycle has been slightly over 11 years.

4 Fast Finite Fourier Transform

We all use FFTs everyday without even knowing it. Cell phones, disc drives, DVD's and JPEG's all involve finite Fourier transforms. One-dimensional tranforms with a million points and two-dimensional 1000-by-1000 transforms are common. The key to modern signal and image processing is the ability to do these computations rapidly.

Direct application of the definition

$$Y_{k+1} = \sum_{j=0}^{n-1} \omega^{jk} y_{j+1}, \;\; k = 0, \ldots, n-1$$

requires n multiplications and n additions for each of the n components of Y for a total of $2n^2$ floating point operations. And that does not count the generation of the powers of ω. A computer capable of doing one multiplication and addition every microsecond would require a million seconds, or about 11.5 days, to do a million point FFT.

Several people discovered fast FFT algorithms independently and many people have since contributed to their development, but it was a 1965 paper by John Tukey of Princeton University and John Cooley of IBM Research that is generally credited as the starting point for the modern usage of the FFT.

Modern fast FFT algorithms have computational complexity $O(n \log_2 n)$ instead of $O(n^2)$. When n is a power of 2, a one-dimensional FFT of length n requires less than $3n \log_2 n$ floating point operations. For $n = 2^{20}$, that's a factor of almost 35,000 faster than $2n^2$. Even when $n = 1024 = 2^{10}$, the factor is about 70.

With MATLAB 5.3 and a 266 MHz Pentium laptop, the time required for `fft(x)` when `length(x)` is $2^{20} = 1048576$ is about 5 seconds. MATLAB 6.0 is still in development in February, 2000, but its new FFT code is available for testing. With the new code, a length 2^{20} `fft` takes less than half a second. The new code is based on FFTW, "The Fastest Fourier Transform in the West", developed at MIT and available from `http://www.fftw.org`.

The key to the fast FFT algorithms is the double angle formula for trig functions. Using complex notation and

$$\omega = \omega_n = e^{-2\pi i/n} = \cos\delta - i\sin\delta$$

we have

$$\omega_{2n}^2 = \omega_n$$

Written out in terms of separate real and imaginary parts, this is

$$\begin{aligned} \cos 2\delta &= \cos^2\delta - \sin^2\delta \\ \sin 2\delta &= 2\cos\delta\sin\delta \end{aligned}$$

Start with the basic definition.

$$Y_{k+1} = \sum_{j=0}^{n-1} \omega^{jk} y_{j+1},\ k = 0, \ldots, n-1$$

Assume that n is even and that $k \le n/2 - 1$ Divide the sum into terms with even subscripts and terms with odd subscripts.

$$\begin{aligned} Y_{k+1} &= \sum_{\text{even}j} \omega^{jk} y_{j+1} + \sum_{\text{odd}j} \omega^{jk} y_{j+1} \\ &= \sum_{j=0}^{n/2-1} \omega^{2jk} y_{2j+1} + \omega^k \sum_{j=0}^{n/2-1} \omega^{2jk} y_{2j+2} \end{aligned}$$

The two sums on the right are components of the FFTs of length $n/2$ of the portions of y with even and odd subscripts. In order to get the entire FFT of length n, we have to do two FFTs of length $n/2$, multiply one of these by powers of ω, and concatenate the results.

The relationship between an FFT of length n and two FFTs of length $n/2$ can be expressed compactly in MATLAB. If `n = length(y)` is even,

```
omega = exp(-2*pi*i/n);
k = (0:n/2-1)';
```

```
      w = omega .^ k;
      u = fft(y(1:2:n-1));
      v = w.*fft(y(2:2:n));
```

then

```
      fft(y) = [u+v; u-v];
```

Now, if n is not only even, but actually a power of 2, the process can be repeated. The FFT of length n is expressed in terms of two FFTs of length $n/2$, then four FFTs of length $n/4$, then eight FFTs of length $n/8$ and so on until we reach n FFTs of length one. An FFT of length one is just the number itself. If $n = 2^p$, the number of steps in the recursion is p. There is $O(n)$ work at each step, so the total amount of work is

$$O(np) = O(n \log_2 n)$$

If n is not a power of two, it is still possible to express the FFT of length n in terms of several shorter FFTs. An FFT of length 100 is two FFTs of length 50, or four FFTs of length 25. An FFT of length 25 can be expressed in terms five FFTs of length five. If n is not a prime number, an FFT of length n can be expressed in terms of FFTs whose lengths divide n. Even if n is prime, it is possible to embed the FFT in another whose length can be factored. We will not go into the details of these algorithms here.

The `fft` function in MATLAB 5 uses fast algorithms whenever the length is a product of small primes. The `fft` function in MATLAB 6 will use fast algorithms even when the length is prime.

5 ffttx

Our textbook function `ffttx` combines the two basic ideas of this chapter. If n is a power of 2, it uses the $O(n \log_2 n)$ fast algorithm. If n has an odd factor, it uses the fast recursion until it reaches an odd length, then sets up the discrete Fourier matrix and uses matrix-vector multiplication.

```
    function y = ffttx(x)
    %FFTTX Textbook Fast Finite Fourier Transform.
    % FFTTX(X) computes the same finite Fourier transform
    % as FFT(X).  The code uses a recursive divide and conquer
    % algorithm for even order and matrix-vector multiplication
    % for odd order.  If length(X) is m*p where m is odd and
    % p is a power of 2, the computational complexity of this
    % approach is O(m^2)*O(p*log2(p)).

    x = x(:);
    n = length(x);
    omega = exp(-2*pi*i/n);
```

```
if rem(n,2) == 0
   % Recursive divide and conquer
   k = (0:n/2-1)';
   w = omega .^ k;
   u = ffttx(x(1:2:n-1));
   v = w.*ffttx(x(2:2:n));
   y = [u+v; u-v];
else
   % The Fourier matrix.
   j = 0:n-1;
   k = j';
   F = omega .^ (k*j);
   y = F*x;
end
```

6 Fourier Integral Transform

The Fourier integral transform converts one complex function into another.

The transform is

$$F(\mu) = \int_{-\infty}^{+\infty} f(t) e^{-2\pi i \mu t} dt$$

The inverse transform is

$$f(t) = \int_{-\infty}^{+\infty} F(\mu) e^{+2\pi i \mu t} d\mu$$

The variables t and μ run over the entire real line. If t has units of seconds, then μ has units of radians per second. Both functions $f(t)$ and $F(\mu)$ are complex valued, but in most applications the imaginary part of $f(t)$ is zero.

Alternative units use $\nu = 2\pi\mu$, which has units of cycles or revolutions per second. With this change of variable, there are no factors of 2π in the exponentials, but there are factors of $1/\sqrt{2\pi}$ in front of the integrals, or a single factor of $1/(2\pi)$ in the inverse transform. Maple and MATLAB's Symbolic Toolbox use this alternative notation with the single factor in the inverse transform.

7 Fourier Series

A Fourier series converts a periodic function into an infinite sequence of Fourier coefficients. Let $f(t)$ be the periodic function and let L be its period, so

$$f(t + L) = f(t) \text{ for all } t$$

The Fourier coefficients are given by integrals over the period

$$c_j = \frac{1}{L} \int_{-L/2}^{L/2} f(t) e^{-2\pi i j t} dt, \; j = \ldots, -1, 0, 1, \ldots$$

With these coefficients, the complex form of the Fourier series is

$$f(t) = \sum_{j=-\infty}^{\infty} c_j e^{2\pi ijt/L}$$

8 Discrete time Fourier tranform

A discrete time Fourier transform converts an infinite sequence of data values into a periodic function. Let x_k be the sequence, with the index k taking on all integer values, positive and negative.

The discrete time Fourier transform is the complex valued periodic function

$$X(e^{i\omega}) = \sum_{k=-\infty}^{\infty} x_k e^{ik\omega}$$

The sequence can then be represented

$$x_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{i\omega})e^{-ik\omega}d\omega,\ k = \ldots, -1, 0, 1, \ldots$$

9 Finite Fourier Transform

The finite Fourier transform converts one finite sequence of coefficients into another sequence of the same length, n.

The transform is

$$Y_{k+1} = \sum_{j=0}^{n-1} y_{j+1}e^{-2ijk\pi/n},\ k = 0, \ldots, n-1$$

The inverse transform is

$$y_{j+1} = \frac{1}{n}\sum_{k=0}^{n-1} Y_{k+1}e^{2ijk\pi/n},\ j = 0, \ldots, n-1$$

10 Connections

The Fourier integral transform involves only integrals. The finite Fourier transform involves only finite sums of coefficients. Fourier series and the discrete time Fourier transform involve both integrals and sequences. It is possible to "morph" any of the systems into any of the others by taking limits or restricting domains.

Start with Fourier series. Let L, the length of the period, become infinite and let j/L, the coefficient index scaled by the period length, become a continuous variable, μ. Then the Fourier coefficients, c_j, become the Fourier transform, $F(\mu)$.

Again, start with Fourier series. Interchanging the roles of the periodic function and the infinite sequence of coefficients leads to the discrete time Fourier transform.

Start with Fourier series a third time. Now restrict t to a finite number of integral values, k and restrict j to the same finite number of values. Then the Fourier coefficients become the finite Fourier transform.

In the Fourier Integral context, Parseval's theorem says

$$\int_{-\infty}^{+\infty} |f(t)|^2 dt = \int_{-\infty}^{+\infty} |F(\mu)|^2 d\mu$$

This quantity is known as the *total power* in a signal.

Exercises

1. (*el Niño*) The climatological phenomenon *el Niño* results from changes in atmospheric pressure in the southern Pacific ocean. The "Southern Oscillation Index" is the difference in atmospheric pressure between Easter Island and Darwin Australia, measured at sea level at the same moment. The text file `elnino.dat` contains values of this index measured on a monthly basis over the 14 year period 1962 through 1975.

Your assignment is to carry out an analysis similar to the sunspot example on the *el Niño* data. The unit of time is one month instead of one year. You should find there is a prominent cycle with a period of 12 months and a second, less prominent, cycle with a longer period. This second cycle shows up in about three of the Fourier coefficients, so it is hard to measure its length, but see if you can make an estimate.

2. (Train signal) The MATLAB `demos` directory contains several sound samples. One of them is a train whistle. The statement

```
load train
```

will give you a long vector `y` and a scalar `Fs` whose value is the number of samples per second. The time increment is `1/Fs` seconds.

If your computer has sound capabilities, the statement

```
sound(y,Fs)
```

will play the signal, but you don't need that for this problem.

The data does not have a significant linear trend. There are two pulses of the whistle, but the harmonic content of both pulses is the same.

(a) Plot the data with time in seconds as the independent variable.

(b) Produce a periodogram with frequency in cycles/second as the independent variable.

(c) Identify the frequencies of the six peaks in the periodogram. You should find that ratios between these six frequences are close to ratios between small integers. For example, one of the frequencies of 5/3 times another. The frequences that are integer multiples of other frequencies are ”overtones”. How many of the peaks are fundamental frequencies and how many are overtones?