Chapter 6

The Fourier Transform

Many different fields, including medicine, optics, physics, and electrical engineering, use the Fourier Transform (FT) as a common analysis tool. In practice, the standards by compression groups JPEG (Joint Photographic Experts Group) and MPEG (Motion Picture Experts Group) use a modified form of the Fourier transform. Essentially, it allows us to look at frequency information instead of time information, which people find more natural for some data. For example, many stereo systems have rows of little lights that glow according to the strength of frequency bands. The stronger the treble, for example, the more lights along the row are lit, creating a light bar that rises and falls according to the music. This is the type of information produced by the Fourier transform.

The Discrete Fourier Transform (DFT) is the version of this transform that we will concentrate on, since it works on discrete data. Our data are discrete in time, and we can assume that they are periodic. That is, if we took another N samples they would just be a repeat of the data we already have, in terms of the frequencies present. In this case, we use the DFT, which produces discrete frequency information that we also assume is periodic. Below, in Figure 6.1, we see the frequency magnitude response graph for the "ee" sound. We got this by applying the discrete Fourier transform to the sound file. The figure shows the whole range along the top, and a close-up view on the bottom.

The following graph, Figure 6.2, shows the frequency magnitude response for another sound file: a brief recording of *Adagio from Toccata and Fuge in C*, written by J.S. Bach. The top of this figure shows the entire frequency range, from 0 to 22,050 Hz, while the bottom part shows a close-up view of the first 4500 frequencies. An interesting thing to notice is that the spikes in magnitude are regularly spaced. This happens often with real signals, especially music. For example, right before 1000 Hz, we see three spikes increasing in magnitude, corresponding to three different

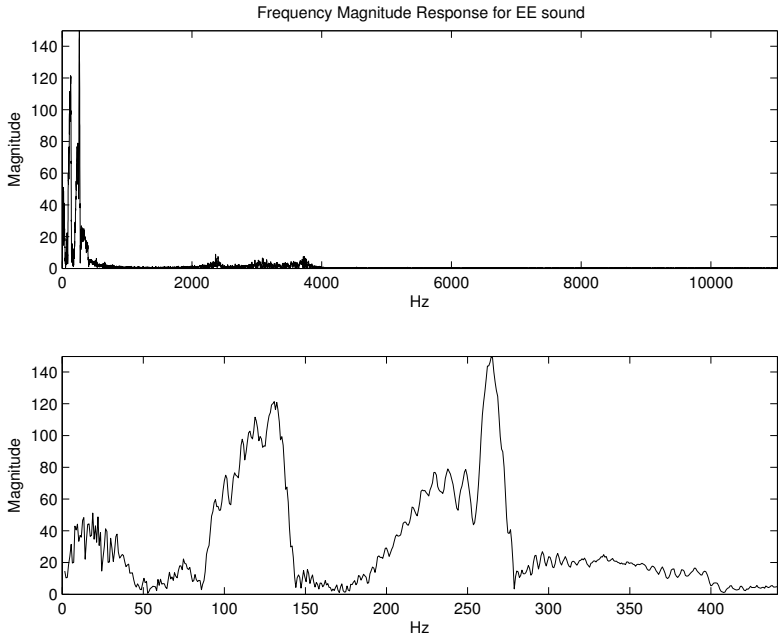

Figure 6.1: A person vocalizing the "ee" sound.

(but related) frequencies. We call this harmonics. This can be seen even more clearly in Figure 6.3, where a sustained note from a flute is played. Four frequencies are very pronounced, while most of the other frequencies are 0.

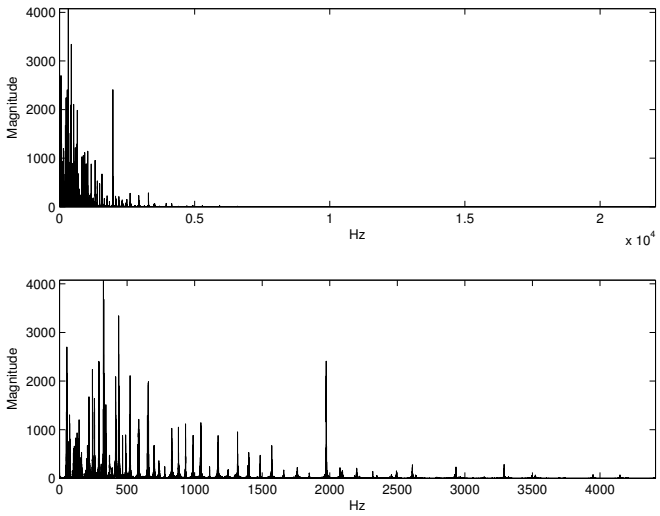

Figure 6.2: J.S. Bach's *Adagio from Toccata and Fuge in C*—frequency magnitude response.

The frequency range appearing above (0 to 22,050 Hz) was not arbitrarily chosen. Compact Disks (CDs) store music recorded at 44,100 samples per second, allowing for sounds in the range of 0 to 22,050 Hz, which is slightly greater than the maximum frequency that we can hear. It should not be surprising that almost all of the frequency content in Bach's music shown in Figure 6.2 is below 4000 Hz. The organ, for which this music was written, can produce a very wide range of sound. Some organs can produce infrasound notes (below 20 Hz, which most humans cannot hear), and also go well beyond 10 kHz. But these high notes are not necessary for pleasing music. To put this in perspective, consider that most instruments (guitar, violin, harp, drums, horns, etc.) cannot produce notes with fundamental frequencies above 4000 Hz, though instruments do produce harmonics that appear above this frequency [22]. A piano has a range of 27.5 Hz to just over 4186 Hz.

The Fourier transform is a way to map continuous time, nonperiodic data to continuous frequency, nonperiodic data in the frequency-domain. A variation called Fourier Series works with periodic data that is continuous in time, and turns it into a nonperiodic discrete frequency representation. When the data are discrete in time, and nonperiodic, we can use the discrete time Fourier transform (DTFT)

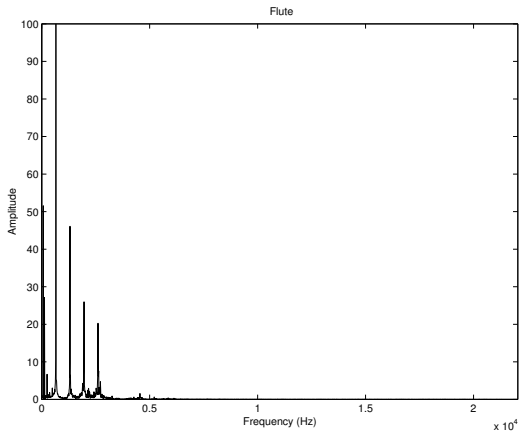

Figure 6.3: A sustained note from a flute.

to get a periodic, continuous frequency representation. Finally, when the data are discrete in time (and we can assume that they are periodic), we use the DFT to get a discrete frequency, assumed periodic representation.

6.1 Fast Fourier Transform Versus the Discrete Fourier Transform

The Fast Fourier Transform (FFT) is a clever algorithm to implement the DFT. As expected, it gives the same results as the DFT, but it arrives at them much faster due to the efficiency of the algorithm. The FFT was a major breakthrough, since it allowed researchers to calculate the Fourier transform in a reasonable amount of time. Figure 6.4 demonstrates this difference. The bottom curve is a plot of $Nlog_2(N)$, while the other is N^2. For example, a late-model Pentium-4 processor at 2 GHz does on the order of 2 billion calculations per second. An algorithm (such as the DFT) that performs N^2 operations would take about 5 seconds to complete for 100,000 data samples. For 100,000,000 data samples, this would take *about 2 months* to compute! In contrast, an algorithm (such as the FFT) that performs $Nlog_2(N)$ operations for 100,000,000 data samples would need only 1.33 seconds.

MATLAB provides both `fft` and `ifft` functions. For optimum speed, the FFT needs the data size to be a power of 2, though software (such as `fft` command in MATLAB) does not require this. For most applications, zeros can be appended to the data without negatively affecting the results. In fact, zeros are often appended to get better looking results. It does not add any information, but it changes the

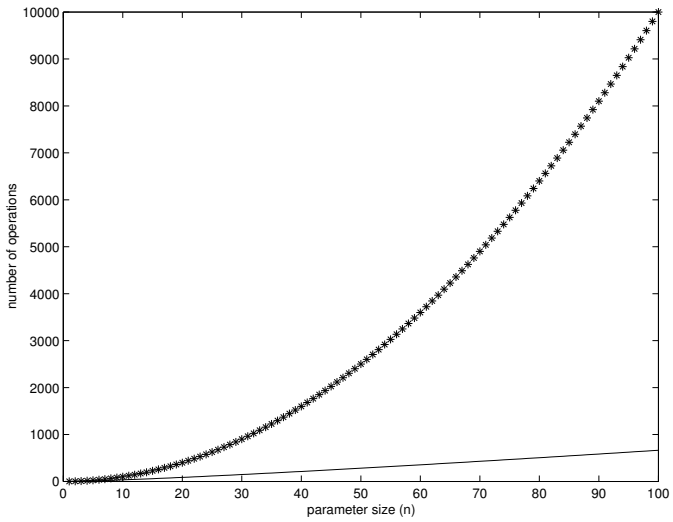

Figure 6.4: Comparing $Nlog_2(N)$ (line) versus N^2 (asterisks).

analysis frequencies used by the FFT/DFT. This technique is called *zero-padding*.

We saw in Chapter 3, "Filters," that filters perform convolution. One can use the FFT to compute convolution efficiently. Other uses include analyzing the effect that a filter has on a signal, and designing FIR filters (with the Inverse FFT). Any time that a problem requires the DFT, the FFT can be used instead.

6.2 The Discrete Fourier Transform

This is the Discrete Fourier Transform (DFT)

$$X[m] = \sum_{n=0}^{N-1} x[n](\cos(2\pi nm/N) - j\sin(2\pi nm/N))$$

where $m = 0..N-1$. We call this the "rectangular form" of the DFT, and there are many variations of this transform. For example, one common way to express it uses the $e^{-j2\pi nm/N}$ term instead of the sinusoids, which can make analysis by a human easier.

$$X[m] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi nm/N}$$

Here is a MATLAB function that calculates the DFT. Its purpose is to demonstrate how the DFT can be computed, but it does not have the efficiency of the `fft` function built in to MATLAB. It returns a complex array, which stores two pieces of information per complex number, corresponding to an x and y coordinate. Normally, we would need the magnitudes and phase angles. Just as we can convert from Cartesian coordinates to polar coordinates, we can convert from the complex array information to the magnitude and phase information.

```
function [X] = dft(x)
%
% Demonstrate DFT
%

Xsize = length(x);

% do DFT (the hard way)
for m=0:Xsize-1
   mysumm = 0;
   for n=0:Xsize-1
      mysumm = mysumm + x(n+1) * (cos(2*pi*n*m/Xsize) ...
         - j*sin(2*pi*n*m/Xsize));
   end
   X(m+1) = mysumm;
end
```

All that needs to be returned is the vector of complex numbers, or variable X. The magnitudes and phase angles can be calculated, though this information makes X redundant. To convert, we can use the `abs` and `angle` functions, such as:

```
Xmag = abs(X);
Xphase = angle(X);
```

Also notice that the comments are honest. This is not a very efficient way to obtain the Fourier transform! But it is meant to explain how the DFT can be calculated.

For the DFT, notice how the arguments of the cosine and sine functions are the same, yet the arguments are functions of both n and m. In effect, this transform spreads the original one-dimensional data over a two-dimensional matrix. Let's illustrate this with an example. The following code finds the DFT of an example

signal (it uses `fft`, but `dft` would also work assuming that the above program is present).

```
>> fft([6, 4, 9, 0, 1, 5, 2, 7])

ans =

  Columns 1 through 4

  34.0000  9.2426 - 1.3431i  -4.0000 - 2.0000i  0.7574 +12.6569i


  Columns 5 through 8

  2.0000   0.7574 -12.6569i  -4.0000 + 2.0000i  9.2426 + 1.3431i
```

Table 6.1 shows a matrix where we calculate the Fourier transform of the example signal. The rows correspond to n and we see the values from the example signal running down each of the columns (m). Observe how each row has a corresponding signal sample (time-domain data), while the sum of each column results in the DFT of the signal (frequency domain data). If we were to find the sum of magnitudes along each row (i.e., `sum(abs(Matrix(r,:)))`), we would get the time-domain sample multiplied by the number of points (N). Every $e^{-j\theta}$ value has a magnitude of 1; these values are complex vectors of unit magnitude. This 2D matrix comes from the expression for the DFT, $e^{-j\theta}$, where $\theta = 2\pi nm/N$, and $N = 8$ (our sample size). The values of $2\pi nm/N$ for a single column run from 0 to $2\pi m(N-1)/N$. Variable m also runs from 0 to $(N-1)$. The inverse DFT, which we will cover soon, essentially multiplies table entries by an $e^{+j\theta}$ vector, cancelling out the original complex vectors and giving us back the time-domain data.

How to add the values of complex exponentials together may not be obvious. We can always convert these to Cartesian coordinates (of the form $a + jb$) first. Table 6.2 shows another way of looking at this data, with the rectangular form of the DFT equation. We can easily verify that the sum of a column equals the DFT results given by MATLAB, since we add the real parts and imaginary parts separately.

Each column results in a single frequency-domain point. That is, for each output $X[m]$, the frequencies used to find $X[m]$ are $2\pi m(0)/N$, $2\pi m(1)/N$, ..., $2\pi m(N-1)/N$. The number of frequencies used depends entirely upon how many points we have, which is determined by our sampling frequency. Therefore, the analysis frequencies are given by the following relation:

Table 6.1: Example DFT calculations.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$	$6e^{-j2\pi 0}$
1	$4e^{-j2\pi 0}$	$4e^{-j2\pi 0.125}$	$4e^{-j2\pi 0.25}$	$4e^{-j2\pi 0.375}$	$4e^{-j2\pi 0.5}$	$4e^{-j2\pi 0.625}$	$4e^{-j2\pi 0.75}$	$4e^{-j2\pi 0.875}$
2	$9e^{-j2\pi 0}$	$9e^{-j2\pi 0.25}$	$9e^{-j2\pi 0.5}$	$9e^{-j2\pi 0.75}$	$9e^{-j2\pi}$	$9e^{-j2\pi 1.25}$	$9e^{-j2\pi 1.5}$	$9e^{-j2\pi 1.75}$
3	$0e^{-j2\pi 0}$	$0e^{-j2\pi 0.375}$	$0e^{-j2\pi 0.75}$	$0e^{-j2\pi 1.125}$	$0e^{-j2\pi 1.5}$	$0e^{-j2\pi 1.875}$	$0e^{-j2\pi 2.25}$	$0e^{-j2\pi 2.625}$
4	$1e^{-j2\pi 0}$	$1e^{-j2\pi 0.5}$	$1e^{-j2\pi 1}$	$1e^{-j2\pi 1.5}$	$1e^{-j2\pi 2}$	$1e^{-j2\pi 2.5}$	$1e^{-j2\pi 3}$	$1e^{-j2\pi 3.5}$
5	$5e^{-j2\pi 0}$	$5e^{-j2\pi 0.625}$	$5e^{-j2\pi 1.25}$	$5e^{-j2\pi 1.875}$	$5e^{-j2\pi 2.5}$	$5e^{-j2\pi 3.125}$	$5e^{-j2\pi 3.75}$	$5e^{-j2\pi 4.375}$
6	$2e^{-j2\pi 0}$	$2e^{-j2\pi 0.75}$	$2e^{-j2\pi 1.5}$	$2e^{-j2\pi 2.25}$	$2e^{-j2\pi 3}$	$2e^{-j2\pi 3.75}$	$2e^{-j2\pi 4.5}$	$2e^{-j2\pi 5.25}$
7	$7e^{-j2\pi 0}$	$7e^{-j2\pi 0.875}$	$7e^{-j2\pi 1.75}$	$7e^{-j2\pi 2.625}$	$7e^{-j2\pi 3.5}$	$7e^{-j2\pi 4.375}$	$7e^{-j2\pi 5.25}$	$7e^{-j2\pi 6.125}$
Σ	$34e^{j2\pi 0}$	$9.3e^{-j2\pi 0.023}$	$4.5e^{-j2\pi 0.426}$	$12.7e^{j2\pi 0.24}$	$2e^{-j2\pi 0}$	$12.7e^{-j2\pi 0.24}$	$4.5e^{j2\pi 0.4}$	$9.3e^{j2\pi 0.023}$

Table 6.2: Example DFT calculations (rectangular form).

	0	1	2	3	4	5	6	7
0	6.0 + 0.0j	6.0 + 0.0j	6.0 + 0.0j	6.0 + 0.0j	6.0 + 0.0j	6.0 + 0.0j	6.0 + 0.0j	6.0 + 0.0j
1	4.0 + 0.0j	2.8 - 2.8j	0.0 - 4.0j	-2.8 - 2.8j	-4.0 - 0.0j	-2.8 + 2.8j	-0.0 + 4.0j	2.8 + 2.8j
2	9.0 + 0.0j	0.0 - 9.0j	-9.0 - 0.0j	-0.0 + 9.0j	9.0 + 0.0j	0.0 - 9.0j	-9.0 - 0.0j	-0.0 + 9.0j
3	0.0 + 0.0j	0.0 + 0.0j	0.0 + 0.0j	0.0 + 0.0j	0.0 + 0.0j	0.0 + 0.0j	0.0 + 0.0j	0.0 + 0.0j
4	1.0 + 0.0j	-1.0 - 0.0j	1.0 + 0.0j	-1.0 - 0.0j	1.0 + 0.0j	-1.0 - 0.0j	1.0 + 0.0j	-1.0 - 0.0j
5	5.0 + 0.0j	-3.5 + 3.5j	0.0 - 5.0j	3.5 + 3.5j	-5.0 - 0.0j	3.5 - 3.5j	-0.0 + 5.0j	-3.5 - 3.5j
6	2.0 + 0.0j	-0.0 + 2.0j	-2.0 - 0.0j	0.0 - 2.0j	2.0 + 0.0j	-0.0 + 2.0j	-2.0 - 0.0j	-0.0 - 2.0j
7	7.0 + 0.0j	4.9 + 4.9j	-0.0 + 7.0j	-4.9 + 4.9j	-7.0 - 0.0j	-4.9 - 4.9j	-0.0 - 7.0j	4.9 - 4.9j
Σ	34.0 + 0.0j	9.2 - 1.3j	-4.0 - 2.0j	0.8 + 12.7j	2.0 - 0.0j	0.8 - 12.7j	-4.0 + 2.0j	9.2 + 1.3j

$$f_{analysis}[m] = \frac{m f_{sampling}}{N}.$$

6.3 Plotting the Spectrum

To get a spectral plot, we will start out with an example signal. The following code sets up a signal, x.

```
>> % Set up an example signal
>> n = 0:99; % number of points
>> fs = 200;  % sampling frequency
>> Ts = 1/fs; % sampling period
>> % x is our example signal
>> x = cos(2*pi*20*n*Ts + pi/4) + ...
    3*cos(2*pi*40*n*Ts - 2*pi/5) + ...
    2*cos(2*pi*60*n*Ts + pi/8);
>>
```

Now we need the frequency information from x, which we can get from the Fourier transform. Also, we will establish the index m.

```
>> X = fft(x);
>> m = 0:length(X)-1;
```

It may be helpful to know the frequency resolution; the following code displays it.

```
>> disp(sprintf('Freq resolution is every %5.2f Hz',...
    fs/length(X)));
Freq resolution is every  2.00 Hz
>>
```

To see the spectrum, we will show both the frequency magnitude response and a plot of the phase angles.

```
>> % Plot magnitudes
>> subplot(2,1,1);
>> stem(m*fs/length(X),abs(X), 'b');
>> ylabel('magnitude');
```

```
>> xlabel('frequency (Hz)');
>> title('Frequency magnitude response');
>> % Plot phase angles
>> subplot(2,1,2);
>> stem(m*fs/length(X),angle(X), 'b');
>> ylabel('phase angle');
>> xlabel('frequency (Hz)');
>> title('Phase angle plot');
```

When we run the previous code, we see the graphs as shown in Figure 6.5, and clearly there is frequency content at 20 Hz, 40 Hz, and 60 Hz, with magnitudes of 50, 150, and 100, corresponding to relative magnitudes of 1, 3, and 2 as in our original signal. We notice a few annoying things about this figure, though. First, since the original signal x is real, the frequency magnitude response has a mirror image, so we only really need the first half of it. For the phases, the pattern is also reflected around the x-axis, so again we only need half of the plot. The phase plot contains a lot of information, actually too much. At frequency 80 Hz, for example, we see that the amplitude approximates zero, while the phase angle is relatively large. (We expect the phase angles to be between $-\pi$ and $+\pi$.)

To solve the first problem, we will simply plot the first half of both the magnitudes and phases. We can set up a variable called *half_m* to give us half the range of m, and use it in place of m. But when we use it, we must be careful to add 1 to it before using it as an array offset, since it starts at 0.

```
half_m = 0:ceil(length(X)/2);
stem(half_m*fs/length(X),abs(X(half_m+1)), 'b');
```

For the second problem, we need a little more information.

```
>> X(2)

ans =

 -1.6871e-13 - 2.2465e-15i

>> angle(X(2))

ans =

   -3.1283
```

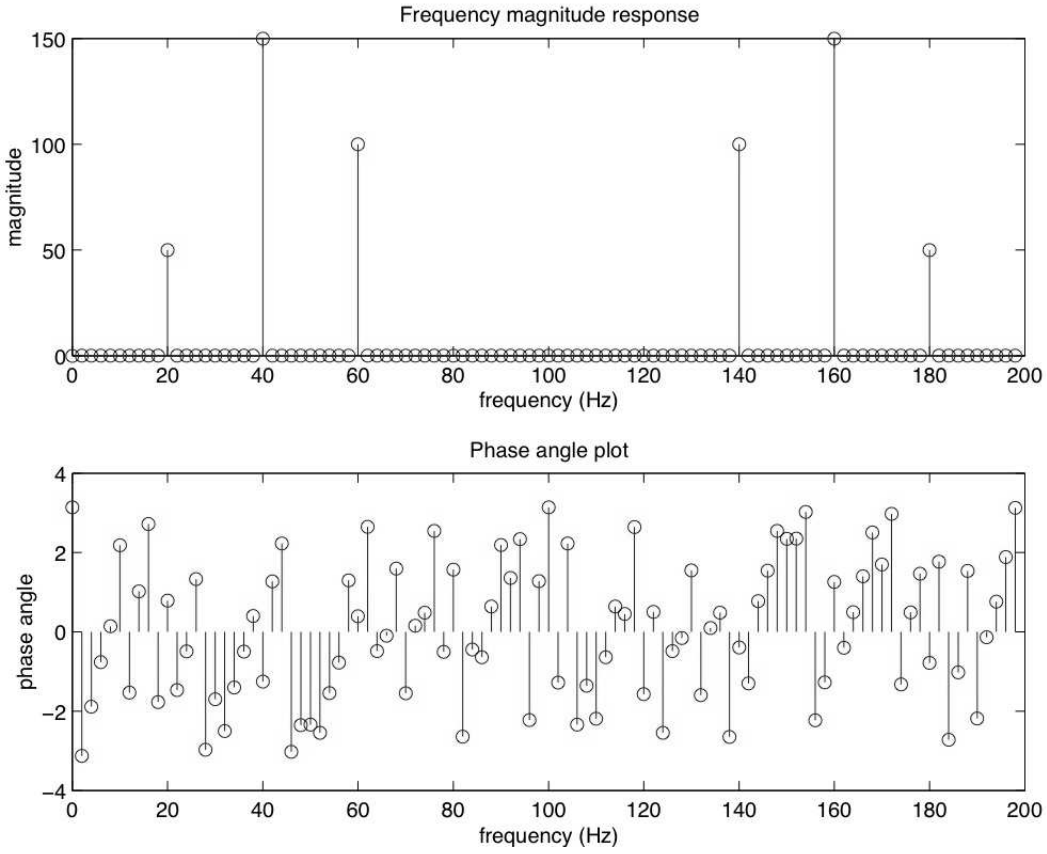

Figure 6.5: Spectrum for an example signal.

```
>> abs(X(2))

ans =

   1.6873e-13
```

The above commands confirm our suspicion that the phase angles are calculated and shown for every X value even when they are close to zero. Fixing this requires the use of a tolerance value. If the magnitude is smaller than the tolerance, then we should assume a zero phase. After all, the magnitudes correspond to how much a sinusoid contributes to our signal. If a sinusoid contributes about zero, then it makes no sense to keep a nonzero phase angle for it.

```
>> % The next 3 lines allow us to ignore phases
>> % that go with very small magnitudes.
>> tolerance = 0.00001;
>> X2 = ceil(abs(X) - tolerance);
>> X3 = round(X2 ./ (X2+1));
>> % X3 above is a vector of 0s and 1s
```

The above commands may be a bit confusing, but these lines set up a binary vector. The subtraction operation separates the magnitudes for us; since all magnitudes are positive, any negative values after the subtraction correspond to phases that we should ignore, and the `ceil` function will make them zero. Now the problem is to map the nonzero values in $X2$ to 1. Dividing each value of $X2$ by itself (plus 1) will return values that are either 0 or almost 1, while avoiding a divide-by-zero error. The `round` function simply takes care of making the "almost 1" values equal to 1. The code results in vector $X3$, consisting of zeros for the phases to ignore, and ones for the phases we want to see. We can then use it in the stem plot below, to zero-out any phases with magnitudes close to zero. The following plot does not use *half_m* as above, but we can put all of this together.

```
subplot(2,1,2);
stem(m*fs/length(X),angle(X).*X3, 'b');
```

Now we can put all of this together into a program, to plot the spectrum. When we run this program, we get the graphs shown in Figure 6.6, with the magnitudes and phase angles plotted versus the frequencies.

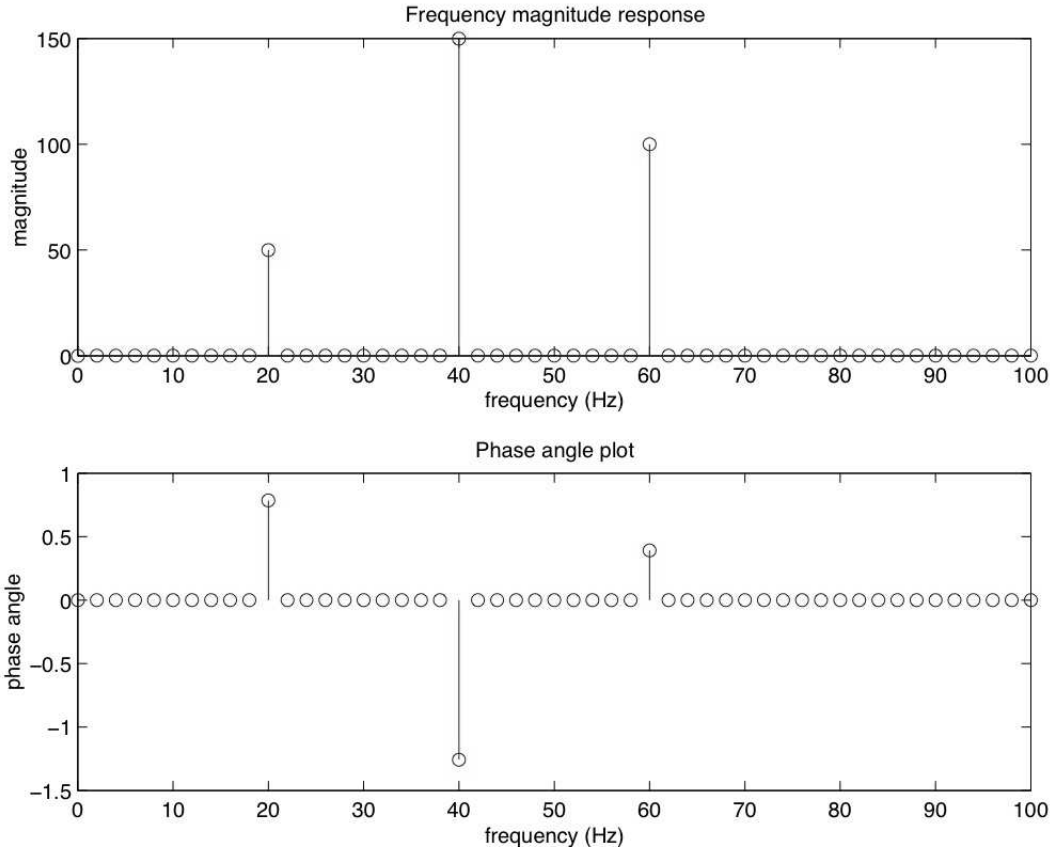

Figure 6.6: Improved spectrum for an example signal.

This program is shown in its entirety, since its pieces are shown above. To summarize what it does, first it makes an example signal by adding several sinusoids. Next, it finds the Fourier transform, then prints the frequency resolution. After this, it plots the magnitudes. It also plots the phases, but first it zeros-out the phases that have a tiny corresponding magnitude.

```
% spectrum.m
%
% Show the spectrum of an example signal.
%

% Set up an example signal
n = 0:99; % number of points
fs = 200;  % sampling frequency
Ts = 1/fs; % sampling period
% x is our example signal
x = cos(2*pi*20*n*Ts + pi/4) + ...
    3*cos(2*pi*40*n*Ts - 2*pi/5) + ...
    2*cos(2*pi*60*n*Ts + pi/8);

% Find the spectrum
X = fft(x);
%m = 0:length(X)-1;
half_m = 0:ceil(length(X)/2);

disp(sprintf('Freq resolution is every %5.2f Hz',...
    fs/length(X)));

% Plot magnitudes
subplot(2,1,1);
%stem(m*fs/length(X),abs(X), 'b');
stem(half_m*fs/length(X),abs(X(half_m+1)), 'b');
ylabel('magnitude');
xlabel('frequency (Hz)');
title('Frequency magnitude response');
% Plot phase angles
subplot(2,1,2);
% The next 3 lines allow us to ignore phases
% that go with very small magnitudes.
tolerance = 0.00001;
```

```
X2 = ceil(abs(X) - tolerance);
X3 = round(X2 ./ (X2+1));
% X3 above is a vector of 0s and 1s
%stem(m*fs/length(X),angle(X).*X3, 'b');
stem(half_m*fs/length(X), ...
    angle(X(half_m+1)).*X3(half_m+1), 'b');
ylabel('phase angle');
xlabel('frequency (Hz)');
title('Phase angle plot');
```

6.4 Zero Padding

When we zero-pad a signal in the time-domain, we get a smoother-looking frequency resolution. Why does this happen? Actually, the length of the sampled signal determines the frequency resolution, and zero-padding does not add any new information. Though the zero-padded signal results in a more visually appealing frequency-domain representation, it does not add to the resolution of the transform.

Why can we zero-pad our time-domain signal? Part of the explanation has to do with the continuous Fourier transform. For convenience, we use the radian frequency, $\omega = 2\pi f$. Using ω also allows us to use $f(t)$ for our function's name, without confusing it with frequency.

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

Suppose $f(t)$ is a signal that has zero value outside the interval $[a, b]$, and assume $a < b < c$. We can, therefore, replace the previous equation with the following:

$$F(\omega) = \int_{a}^{c} f(t)e^{-j\omega t}dt$$

$$= \int_{a}^{b} f(t)e^{-j\omega t}dt + \int_{b}^{c} f(t)e^{-j\omega t}dt.$$

Since $f(t) = 0$ outside the interval $[a, b]$ by definition, the second integral evaluates to 0. We can add 0 to the righthand side of the equation,

$$F(\omega) = 0 + \int_{a}^{b} f(t)e^{-j\omega t}dt.$$

Therefore, for continuous signals, $F(\omega)$ is the same whether or not we consider

$f(t)$ outside of its support. For a discrete signal, the number of points for the frequency-domain signal equals the number of points in the time-domain signal, so adding zeros means that there will be more frequency-domain points. The analysis frequencies, as a result, become finer since they are closer together.

6.5 DFT Shifting Theory

There is a DFT shifting theory, stating that sampling a signal will give the same results (in terms of frequency magnitude response), even when the samples are shifted, such as removing the first k samples and appending them to the end of the sample sequence. This is an important theory, since it tells us that there is no critical time to start recording samples. The program below demonstrates this idea. Notice how the `plot` command puts two signals on the graph at the same time, one in blue ('b') and the other in green ('g'). Try other colors like red ('r') and black ('k').

```
% Demonstrate DFT Shifting theory
%

% parameters to modify
number_of_samples = 16;
shift = 3;      % Must be less than number_of_samples
fs = 2000;      % sampling frequency

% Make our example signal
Ts = 1/fs;
range = 1:number_of_samples;
x(range) = 2*cos(2*pi*400*(range-1)*Ts) ...
         + cos(2*pi*700*(range-1)*Ts);

% Make y a shifted version of x
y = [x(shift:number_of_samples), x(1:shift-1)];

% Get fft of each signal
X = fft(x);
Y = fft(y);

% Find magnitudes
```

```
Xmag = abs(X);
Ymag = abs(Y);
% Find phase angles
Xphase = angle(X);
Yphase = angle(Y);

% Show results in a graph
subplot(3,1,1), plot(range, x,'bd', range, y,'g*');
mystr = strcat('original (blue diamond) and shifted', ...
    '(green *) versions of the sampled signal');
title(mystr);
subplot(3,1,2), plot(range, Xmag,'bd', range, Ymag,'k*');
title('Xmagnitude = diamond, Ymagnitude = *');
subplot(3,1,3), plot(range, Xphase,'bd', range, Yphase,'k*');
title('Xphase = diamond, Yphase = *');
```

Figure 6.7 shows what the output for this program looks like. Though the phase angles are different, the frequency magnitudes are the same for the original as well as the shifted signal. We should expect the phase angles to be different, since these values determine how a composite signal "lines up" with with the samples. Otherwise, we would not be able to get the original signals back via the inverse transform.

6.6 The Inverse Discrete Fourier Transform

This is the Inverse Discrete Fourier Transform (IDFT):

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X[m](\cos(2\pi nm/N) + j \sin(2\pi nm/N))$$

where $n = 0..N-1$. Notice how similar it is to the DFT, the main differences being the $1/N$ term, and the plus sign in front of the complex part. The plus sign is no coincidence, the inverse transform works by using what we call the *complex conjugate* of the forward transform. A complex conjugate, stated simply, occurs when the complex part has a negated sign. For example, $4 - j2$ and $4 + j2$ are complex conjugates. A superscripted asterisk denotes complex conjugation. If we say $a = 3 + j7$ and $b = 3 - j7$, then we could also say $a = b^*$ or $b = a^*$.

An alternate form of the IDFT follows, using Euler's formula to replace the sinusoids with exponentials.

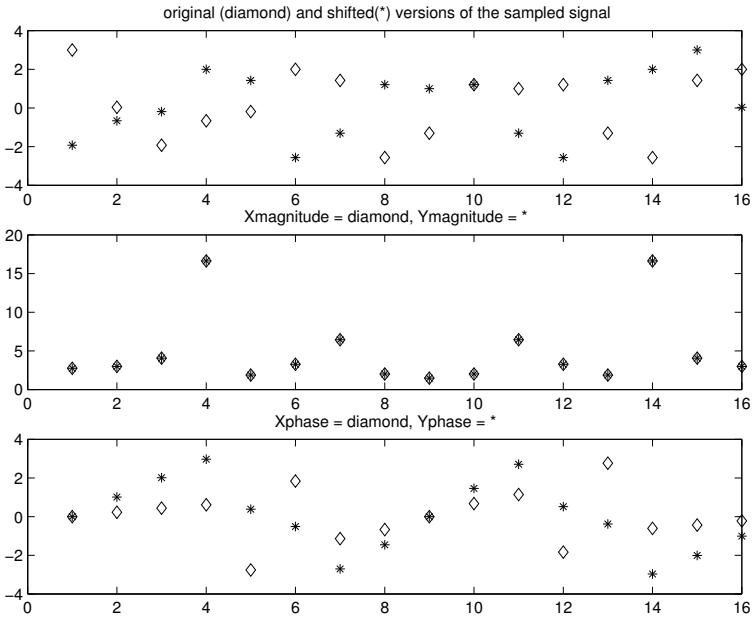

Figure 6.7: Example output of DFT-shift program.

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X[m] e^{j2\pi nm/N}$$

Below is a MATLAB function that calculates the Inverse Discrete Fourier Transform (IDFT). Like the DFT program above, this program intends only to demonstrate how the IDFT can be calculated. MATLAB has an `ifft` function that gives the same results, only faster.

```
function [x] = idft(X)
% Find the 1D Inverse DFT of an input signal.
% It returns the IDFT of the signal as a
% vector of complex numbers.
%
% Usage:
%     [x] = idft(X);
%
% This function is NOT very efficient,
```

```
% but it demonstrates how the IDFT is done.

Xsize = length(X);
% Do reconstruction
for n=0:Xsize-1
   for m=0:Xsize-1
      arra(n+1,m+1) = X(m+1) * (cos(2*pi*m*n/Xsize) + ...
         j*sin(2*pi*m*n/Xsize));
   end
end

for n=0:Xsize-1
   mysumm = 0;
   for m=0:Xsize-1
      mysumm = mysumm + arra(n+1,m+1);
   end
   % Keep only the real part
   %x(n+1) = real(mysumm) / Xsize;
   x(n+1) = mysumm / Xsize;
end
```

Below is shown a MATLAB session that demonstrates the DFT and IDFT functions. First, the signal *mysignal* is given some example values. Next, the "dft" function is called. Notice that the "dft" function will return three values, but here we only use the first.

After the DFT values are stored in variable *M*, the inverse discrete Fourier transform is computed, and stored in *mysignal_2*. This signal should have the same information as *mysignal*, and this is shown to be true. The `round` function gets rid of the part beyond the decimal point, and the `real` function ignores the imaginary part (which is zero).

```
  To get started, select "MATLAB Help" from the Help menu.

>> mysignal = [ 7 4 3 9 0 1 5 2 ];
>> M = dft(mysignal)

M =

  Columns 1 through 6
```

```
  31.0000             4.1716 - 5.0711i  -1.0000 + 6.0000i
9.8284 - 9.0711i  -1.0000 - 0.0000i   9.8284 + 9.0711i

  Columns 7 through 8

  -1.0000 - 6.0000i   4.1716 + 5.0711i

>> mysignal_2 = idft(M)

mysignal_2 =

  Columns 1 through 6

   7.0000 + 0.0000i   4.0000 + 0.0000i   3.0000 - 0.0000i
9.0000 - 0.0000i   0.0000 - 0.0000i   1.0000 + 0.0000i

  Columns 7 through 8

   5.0000 + 0.0000i   2.0000 + 0.0000i

>> real(round(mysignal_2))

ans =

     7     4     3     9     0     1     5     2

>> mysignal

mysignal =

     7     4     3     9     0     1     5     2
```

6.7 Forward and Inverse DFT

This section shows that when we take a signal, perform the DFT on it, and then perform the IDFT on the result, we will end up with the same values that we started with. We will begin by remembering the formulas.

Discrete Fourier Transform (DFT):

$$X[m] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nm/N}$$

where $m = 0..N-1$.

Inverse Discrete Fourier Transform (IDFT):

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X[m] e^{j2\pi nm/N}$$

where $n = 0..N-1$.

Let's start with an example signal, $x = \{x_0, x_1, x_2, x_3\}$. We can find the DFT of x using the general formulas above, and since we know that $N = 4$, we can replace it in the formula.

$$X[m] = \sum_{n=0}^{3} x[n] e^{-j2\pi nm/4}$$

$$X[m] = x_0 e^{-j2\pi 0m/4} + x_1 e^{-j2\pi 1m/4} + x_2 e^{-j2\pi 2m/4} + x_3 e^{-j2\pi 3m/4}$$

We can then find the inverse transform...

$$x[n] = \frac{1}{4} \sum_{m=0}^{3} X[m] e^{j2\pi nm/4}$$

$$x[n] = \frac{1}{4}(X[0]e^{j2\pi n0/4} + X[1]e^{j2\pi n1/4} + X[2]e^{j2\pi n2/4} + X[3]e^{j2\pi n3/4})$$

...and replace the $X[m]$ terms with their equivalents.

$$\begin{aligned} x[n] = \frac{1}{4}((&x_0 e^{-j2\pi 0\times 0/4} + x_1 e^{-j2\pi 1\times 0/4} + x_2 e^{-j2\pi 2\times 0/4} + x_3 e^{-j2\pi 3\times 0/4}) e^{j2\pi n0/4} + \\ &(x_0 e^{-j2\pi 0\times 1/4} + x_1 e^{-j2\pi 1\times 1/4} + x_2 e^{-j2\pi 2\times 1/4} + x_3 e^{-j2\pi 3\times 1/4}) e^{j2\pi n1/4} + \\ &(x_0 e^{-j2\pi 0\times 2/4} + x_1 e^{-j2\pi 1\times 2/4} + x_2 e^{-j2\pi 2\times 2/4} + x_3 e^{-j2\pi 3\times 2/4}) e^{j2\pi n2/4} + \\ &(x_0 e^{-j2\pi 0\times 3/4} + x_1 e^{-j2\pi 1\times 3/4} + x_2 e^{-j2\pi 2\times 3/4} + x_3 e^{-j2\pi 3\times 3/4}) e^{j2\pi n3/4}) \end{aligned}$$

Combining the exponents:

$$x[n] = \frac{1}{4}((x_0 e^{-j2\pi 0\times 0/4 + j2\pi n0/4} + x_1 e^{-j2\pi 1\times 0/4 + j2\pi n0/4} +$$
$$x_2 e^{-j2\pi 2\times 0/4 + j2\pi n0/4} + x_3 e^{-j2\pi 3\times 0/4 + j2\pi n0/4}) +$$
$$(x_0 e^{-j2\pi 0\times 1/4 + j2\pi n1/4} + x_1 e^{-j2\pi 1\times 1/4 + j2\pi n1/4} +$$
$$x_2 e^{-j2\pi 2\times 1/4 + j2\pi n1/4} + x_3 e^{-j2\pi 3\times 1/4 + j2\pi n1/4}) +$$
$$(x_0 e^{-j2\pi 0\times 2/4 + j2\pi n2/4} + x_1 e^{-j2\pi 1\times 2/4 + j2\pi n2/4} +$$
$$x_2 e^{-j2\pi 2\times 2/4 + j2\pi n2/4} + x_3 e^{-j2\pi 3\times 2/4 + j2\pi n2/4}) +$$
$$(x_0 e^{-j2\pi 0\times 3/4 + j2\pi n3/4} + x_1 e^{-j2\pi 1\times 3/4 + j2\pi n3/4} +$$
$$x_2 e^{-j2\pi 2\times 3/4 + j2\pi n3/4} + x_3 e^{-j2\pi 3\times 3/4 + j2\pi n3/4})).$$

Multiplying out the terms:

$$x[n] = \frac{1}{4}((x_0 e^{j2\pi(-0\times 0 + n0)/4} + x_1 e^{j2\pi(-1\times 0 + n0)/4} +$$
$$x_2 e^{j2\pi(-2\times 0 + n0)/4} + x_3 e^{j2\pi(-3\times 0 + n0)/4}) +$$
$$(x_0 e^{j2\pi(-0\times 1 + n1)/4} + x_1 e^{j2\pi(-1\times 1 + n1)/4} +$$
$$x_2 e^{j2\pi(-2\times 1 + n1)/4} + x_3 e^{j2\pi(-3\times 1 + n1)/4}) +$$
$$(x_0 e^{j2\pi(-0\times 2 + n2)/4} + x_1 e^{j2\pi(-1\times 2 + n2)/4} +$$
$$x_2 e^{j2\pi(-2\times 2 + n2)/4} + x_3 e^{j2\pi(-3\times 2 + n2)/4}) +$$
$$(x_0 e^{j2\pi(-0\times 3 + n3)/4} + x_1 e^{j2\pi(-1\times 3 + n3)/4} +$$
$$x_2 e^{j2\pi(-2\times 3 + n3)/4} + x_3 e^{j2\pi(-3\times 3 + n3)/4})).$$

Simplifying:

$$x[n] = \frac{1}{4}((x_0 e^{j2\pi 0/4} + x_1 e^{j2\pi 0/4} + x_2 e^{j2\pi 0/4} + x_3 e^{j2\pi 0/4}) +$$
$$(x_0 e^{j2\pi(n)/4} + x_1 e^{j2\pi(n-1)/4} + x_2 e^{j2\pi(n-2)/4} + x_3 e^{j2\pi(n-3)/4}) +$$
$$(x_0 e^{j2\pi(2n)/4} + x_1 e^{j2\pi(2n-2)/4} + x_2 e^{j2\pi(2n-4)/4} + x_3 e^{j2\pi(2n-6)/4}) +$$
$$(x_0 e^{j2\pi(3n)/4} + x_1 e^{j2\pi(3n-3)/4} + x_2 e^{j2\pi(3n-6)/4} + x_3 e^{j2\pi(3n-9)/4})).$$

Simplifying even more ($e^{j0} = 1$):

$$x[n] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0 e^{j2\pi(n)/4} + x_1 e^{j2\pi(n-1)/4} + x_2 e^{j2\pi(n-2)/4} + x_3 e^{j2\pi(n-3)/4})+$$
$$(x_0 e^{j2\pi(2n)/4} + x_1 e^{j2\pi(2n-2)/4} + x_2 e^{j2\pi(2n-4)/4} + x_3 e^{j2\pi(2n-6)/4})+$$
$$(x_0 e^{j2\pi(3n)/4} + x_1 e^{j2\pi(3n-3)/4} + x_2 e^{j2\pi(3n-6)/4} + x_3 e^{j2\pi(3n-9)/4})).$$

Let's see what we get when $n = 1$:

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0 e^{j2\pi(1)/4} + x_1 e^{j2\pi(0)/4} + x_2 e^{j2\pi(-1)/4} + x_3 e^{j2\pi(-2)/4})+$$
$$(x_0 e^{j2\pi(2)/4} + x_1 e^{j2\pi(0)/4} + x_2 e^{j2\pi(-2)/4} + x_3 e^{j2\pi(-4)/4})+$$
$$(x_0 e^{j2\pi(3)/4} + x_1 e^{j2\pi(0)/4} + x_2 e^{j2\pi(-3)/4} + x_3 e^{j2\pi(-6)/4})).$$

Remember that these exponential values, such as $2\pi(-6)/4$, correspond to angles, and any angle greater than 2π can have an integer multiple of 2π removed from it. For example, $e^{j2\pi(-6)/4} = e^{-j12\pi/4} = e^{-j3\pi} = e^{-j(2\pi+\pi)} = e^{-j\pi}$.

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0 e^{j\pi/2} + x_1 + x_2 e^{-j\pi/2} + x_3 e^{-j\pi})+$$
$$(x_0 e^{j\pi} + x_1 + x_2 e^{-j\pi} + x_3 e^{-j2\pi})+$$
$$(x_0 e^{j\pi 3/2} + x_1 + x_2 e^{-j\pi 3/2} + x_3 e^{-j\pi}))$$

Next, we will replace the exponentials with sinusoids, using Euler's formula.

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0(\cos(\pi/2) + j\sin(\pi/2)) + x_1 + x_2(\cos(\pi/2) - j\sin(\pi/2)) + x_3(\cos(\pi) - j\sin(\pi)))+$$
$$(x_0(\cos(\pi) + j\sin(\pi)) + x_1 + x_2(\cos(\pi) - j\sin(\pi)) + x_3(\cos(2\pi) - j\sin(2\pi)))+$$
$$(x_0(\cos(3\pi/2)+j\sin(3\pi/2))+x_1+x_2(\cos(3\pi/2)-j\sin(3\pi/2))+x_3(\cos(\pi)-j\sin(\pi))))$$

Now, we can calculate the sinusoids (Table 6.3), and plug in the values.

Table 6.3: Sinusoids that simplify things for us.

$\cos(\pi/2) = 0,$	$\sin(\pi/2) = 1$
$\cos(\pi) = -1,$	$\sin(\pi) = 0$
$\cos(2\pi) = 1,$	$\sin(2\pi) = 0$
$\cos(3\pi/2) = 0,$	$\sin(3\pi/2) = -1$

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$
$$(x_0(j) + x_1 + x_2(-j) + x_3(-1 - j0))+$$
$$(x_0(-1 + j0) + x_1 + x_2(-1 - j0) + x_3(1 - j0))+$$
$$(x_0(j(-1)) + x_1 + x_2(-j(-1)) + x_3(-1 - j0)))$$

Simplifying:

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$
$$(x_0(j) + x_1 + x_2(-j) + x_3(-1))+$$
$$(x_0(-1) + x_1 + x_2(-1) + x_3(1))+$$
$$(x_0(-j) + x_1 + x_2(j) + x_3(-1))).$$

Grouping terms:

$$x[1] = \frac{1}{4}(x_0(1 + j - 1 - j) + x_1(1 + 1 + 1 + 1) + x_2(1 - j - 1 + j) + x_3(1 - 1 + 1 - 1))$$

$$x[1] = \frac{1}{4}(x_0(0) + x_1(4) + x_2(0) + x_3(0))$$
$$x[1] = x_1.$$

Thus, the inverse transform gives us back the data that we had before the forward transform. The $\frac{1}{N}$ term appears in the inverse transform to scale the outputs back to the original magnitude of the inputs. We only show this for one of the original values, but the interested reader should be able to verify that this works for all four of the original values.

6.8 Leakage

The DFT does not have infinite resolution. A consequence of this is that sometimes frequencies that are present in a signal are not sharply defined in the DFT, and the frequency's magnitude response appears to be spread out over several analysis frequencies. This is called *DFT leakage*. The following code demonstrates this. In the code below, we simulate two sampled signals: $x1$ and $x2$. If you look closely, you will see that the equations defining these two signals are the same. What we change are the parameters; the sampling frequencies ($fs1$ and $fs2$), which alter the sampling periods ($Ts1$ and $Ts2$), and the number of samples to read (given by the lengths of $n1$ and $n2$).

```
fs1 = 1000;          fs2 = 1013;
Ts1 = 1/fs1;         Ts2 = 1/fs2;
n1 = 0:99;           n2 = 0:98;

x1 = 3*cos(2*pi*200*n1*Ts1 - 7*pi/8) + 2*cos(2*pi*300*n1*Ts1) ...
     + cos(2*pi*400*n1*Ts1 + pi/4);

x2 = 3*cos(2*pi*200*n2*Ts2 - 7*pi/8) + 2*cos(2*pi*300*n2*Ts2) ...
     + cos(2*pi*400*n2*Ts2 + pi/4);

mag1 = abs(fft(x1));
mag2 = abs(fft(x2));
```

The results are shown in the following graphs. When we take 100 samples of signal x at 1000 samples per second, we get the spectral plot shown in Figure 6.8.

$$x(t) = 3\cos(2\pi 200t - 7\pi/8) + \cos(2\pi 300t) + 2\cos(2\pi 400t + \pi/4)$$

It shows what happens when the analysis frequencies happen to match up nicely with the frequencies present in the signal. The second figure represents the same signal, only with the number of samples N, and the sampling frequency f_s changed to 99 samples at 1013 samples per second, Figure 6.9. Since

$$f_{analysis}[m] = mf_s/N,$$

changing either (or both) of these parameters affects the analysis frequencies used in the DFT. For the second figure, the analysis frequencies do not match up with the frequencies of the signal, so we see the spectral content spread across ALL frequencies. The signal has not changed, but our view of the frequency information

has. A trained individual would interpret the second figure as if it were the first figure, however.

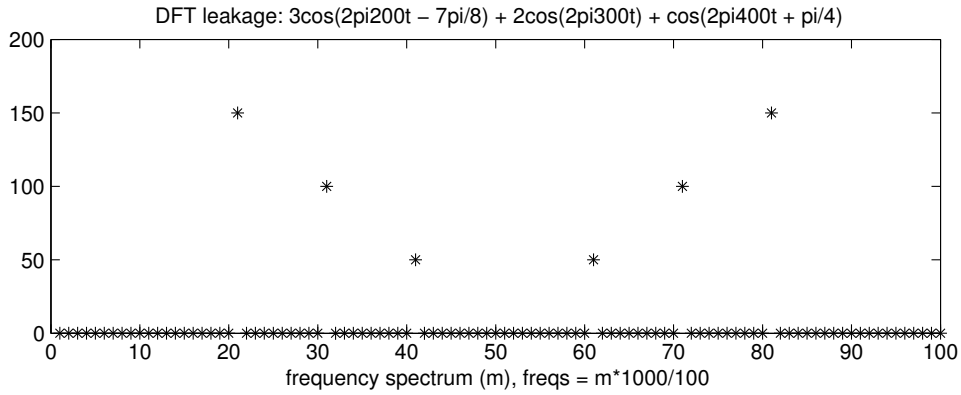

Figure 6.8: Frequency content appears at exact analysis frequencies.

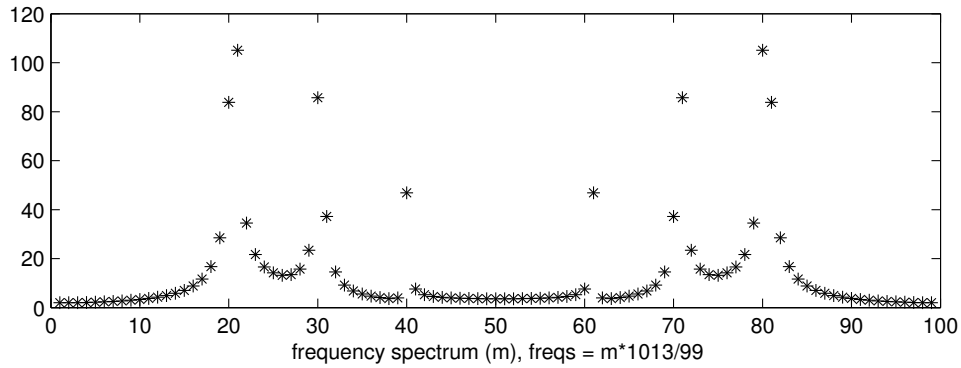

Figure 6.9: Frequency content appears spread out over analysis frequencies.

A *window* is a way of modifying the input signal so that there are no sudden jumps (discontinuities) in it. Even if you do not use a window function, when you sample a signal, you impose a rectangular window function on it [11]. In other words, the input is considered 0 for all values before you start sampling and all values after you stop. The sudden jumps (discontinuities) show up in the frequency response as the sinc function. Leakage occurs when the analysis frequencies do not

land on the actual frequencies present. This means that the actual information "leaks" into other DFT output bins; it shows up as other frequencies. Windowing reduces the sidelobes of the sinc function (of the Continuous Fourier Transform, or CFT for short), which in turn decreases the effect of leakage, since the DFT is a sampled version of the continuous Fourier transform [11].

In other words, looking at the CFT of a signal component (i.e., a single sinusoid), we see the sinc function. Looking at the CFT of a windowed version of the same signal, we see a sinc function with lower sidelobe levels (though a wider main lobe). If we sample the CFT, we might be lucky and sample it right at the point between sidelobes, so we have many zeros and a single spike. Realistically, our samples will land where we have one or two spikes, and several nonzero values, due to the sidelobes. The lower these sidelobes, the better our DFT indicates the frequencies that are actually present.

6.9 Harmonics and Fourier Transform

Harmonics and the Fourier transform are closely linked. Harmonics refers to the use of sinusoids related by a fundamental frequency f_0, that is, adding sinusoids of frequencies f_0, $2f_0$, $3f_0$, etc.

The following program demonstrates harmonics. Its objective is to approximate a triangle wave with a sum of sinusoids. When we run it, we see that the approximation gets better and better as more sinusoids are added. We use a fundamental frequency of 1 Hz. After creating signal x as a triangle wave, the program finds the DFT of it using the `fft` command in MATLAB. Next, it finds the magnitudes and phases for all of the sinusoids corresponding to the DFT results. Finally, the program shows the sum of the sinusoids, pausing briefly between iterations of the loop to show the progress. Essentially, this performs the inverse DFT. Figure 6.10 shows what the progress looks like about one-sixth of the way through (the solid line is the original signal, and the sum of sinusoids is shown as a dash-dot line). Running the program to completion shows that the final approximation appears directly over top of the original signal.

```
%
%  Show how the DFT function can represent a triangle wave
%

% first, make a triangle wave
for i=1:20
    x(i) = i;
```

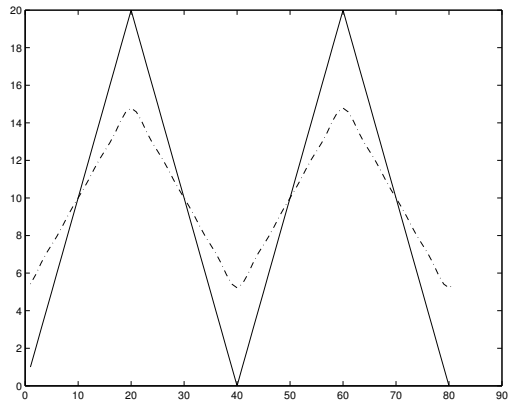

Figure 6.10: Approximating a triangle wave with sinusoids.

```
end
k=20;
for i=21:40
    k=k-1;
    x(i) = k;
end
for i=41:60
    x(i) = i-40;
end
k=20;
for i=61:80
    k=k-1;
    x(i) = k;
end
% OK, now we have the triangle wave as signal x

% Find the DFT of it, and scale the result.
% In other words, represent signal x as a sum of sinusoids
[y] = fft(x);
% Scale the result, so that it is the same size as original
y = y / length(x);
% Convert to polar coordinates (magnitudes and phases)
mag = abs(y);
phi = angle(y);
```

```
% Now, reconstruct it.
% This shows what the "sum of sinusoids" version looks like
t=0:(1/length(mag)):1;
f = 1;  % Our fundamental frequency is 1, since time t=n*m/N
a = 0;
% Show it, adding another sinusoid each time
for k=0:length(mag)-1
   a=a+mag(k+1)*(cos(2*pi*f*k*t+phi(k+1)) ...
     + j*sin(2*pi*f*k*t+phi(k+1)));
   plot(1:length(x), x, 'r', 1:length(a), real(a), 'b-.')
   pause(0.1);
end
```

The first thing this program does is create a signal to work with; a triangle wave. Below, we show how the program works with a square wave. Note the final `for` loop, enclosing `plot` and `pause` commands. We could use the "plotharmonic" function instead, but this program shows the reconstruction of the original signal as a step-by-step process.

Try the above program again, replacing the triangle wave above (program lines 5–22) with the following signals. It is a good idea to type `clear all` between them.

```
% Make a saw-tooth wave
for i=1:40
    x(i) = i;
end
for i=41:80
    x(i) = i-40;
end
% Now we have the saw-tooth wave as signal x
```

The progress of the approximation (one-sixth of the way) appears in Figure 6.11. This figure also shows the result when all terms are used. Notice that the biggest difference between the original and the approximation is the final point. The sinusoid approximation anticipates that the pattern will repeat.

```
% Make a square wave
for i=1:20
    x(i) = 0;
end
for i=21:40
```

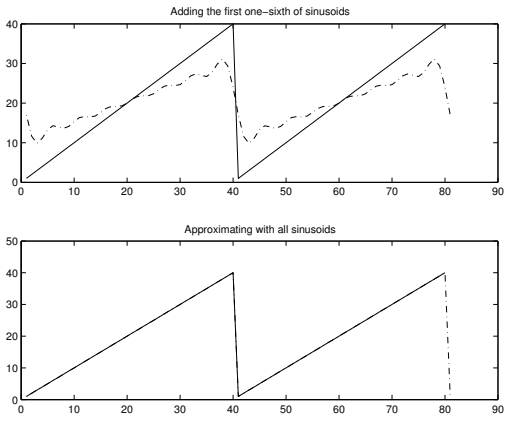

Figure 6.11: Approximating a saw-tooth wave with sinusoids.

```
    x(i) = 1;
end
for i=41:60
    x(i) = 0;
end
for i=61:80
    x(i) = 1;
end
% Now we have the square wave as signal x
```

Figure 6.12 shows the approximation of the previous square wave for one-sixth of sinusoids as well as all sinusoids.

```
% Make a combination saw-tooth square wave
for i=1:40
    x(i) = i;
end
for i=41:80
    x(i) = 40; %i-40;
end
% repeat
for i=81:120
    x(i) = i-80;
end
for i=121:160
```

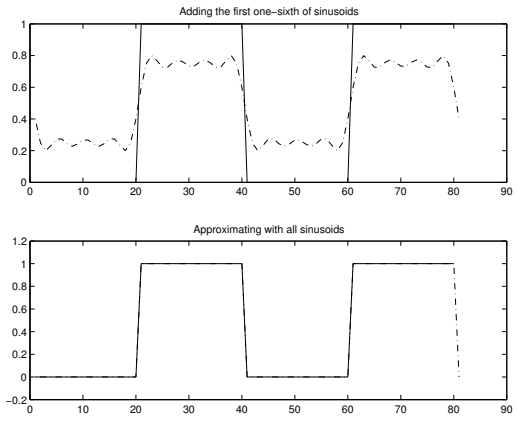

Figure 6.12: Approximating a square wave with sinusoids.

```
    x(i) = 40; %40;
end
```

Figure 6.13 shows the approximation of a combined sawtooth-square wave for use of one-sixth of the terms as well as all terms.

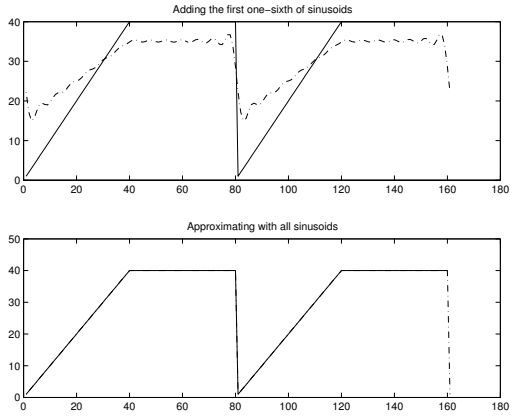

Figure 6.13: Approximating a saw-tooth square wave with sinusoids.

The DFT versions of the signals are good, but they are not perfect. Discontinuities in signals, such as the quick rise and quick fall of the square wave, are difficult to represent in the frequency-domain. The simple impulse function demonstrates this. It is very easy to represent in the time-domain:

$$x(t) = 1,\ t = 0$$

$$x(t) = 0,\ otherwise.$$

But to represent this in the frequency-domain, we need an infinite number of sinusoids.

Looking at this problem the other way, consider a simple sinusoid. In the frequency-domain, it is easily represented as a spike of half-amplitude at the positive and negative frequencies (recall the inverse Euler's formula). For all other frequencies, it is zero. If we want to represent this same signal in time, we have $amplitude \times \cos(2\pi ft + \phi)$, which gives us a function value for *any* value of time. Recording this signal's value for every value of time would require writing down an infinite number of terms.

This leads us to the observation that what is well-defined in the time-domain is poorly represented in the frequency-domain. Also, what is easy to represent in the frequency domain is difficult to represent in the time-domain. In effect, this is *Werner Heisenberg's uncertainty principle* [23].

6.10 Sampling Frequency and the Spectrum

As we saw with the DFT, when we sample a real signal at a rate of f_s samples/second, the frequency magnitude plot from $f_s/2$ to f_s looks like a mirror image of 0 to $f_s/2$.

A real signal is made up of components such as $\cos(\theta)$, or can be put in this form. If it were a complex signal, there would be a $j\sin(\theta)$ component, or something that can be put in that form (such as a $j\cos(\theta)$ component). Since $\cos(\theta) = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2}$, a real signal always has a positive and negative frequency component on the spectrum plot.

Given $\cos(\theta) = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2}$, and given Euler's law:

$$e^{j\theta} = \cos(\theta) + j\sin(\theta)$$

$$e^{j\theta} = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2} + j\sin(\theta)$$

$$\frac{e^{j\theta}}{2} = \frac{e^{-j\theta}}{2} + j\sin(\theta)$$

$$j\sin(\theta) = \frac{e^{j\theta}}{2} - \frac{e^{-j\theta}}{2}.$$

This means that a complex signal has two frequency components as well. Of course, if we had a signal such as $\cos(\theta) + j\sin(\phi)$, we would have:

$$\cos(\theta) + j\sin(\phi) = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2} + \frac{e^{j\phi}}{2} - \frac{e^{-j\phi}}{2}.$$

If θ equals ϕ, then

$$\cos(\theta) + j\sin(\theta) = e^{j\theta}.$$

For this reason, we concern ourselves with only the first half of the frequency response. When it looks like Figure 6.14, we say it is a lowpass filter. This means that any low-frequency (slowly changing) components will remain after the signal is operated on by the system. High-frequency components will be attenuated, or "filtered out." When it looks like Figure 6.15, we say it is a highpass filter. This means that any high-frequency (quickly changing) components will remain after the signal is operated on by the system.

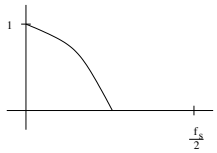

Figure 6.14: Frequency response of a lowpass filter.

The frequency response can be found by performing the DFT on a system's output when the impulse function is given as the input. This is also called the "impulse response" of the system. To get a smoother view of the frequency response, the impulse function can be padded with zeros.

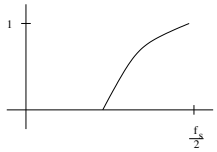

Figure 6.15: Frequency response of a highpass filter.

The more samples that are in the input function to the DFT, the better the resolution of the output will be.

Exercise:

$$w[n] = x[n] + x[n-1]$$
$$y[n] = x[n] - x[n-1]$$

Find and plot the frequency response for w and y.

First, assume x is a unit impulse function, and find $w[n]$ and $y[n]$. Pad these signals, say up to 128 values, so that the result will appear smooth. Next, find the DFT (or FFT) of w and y, and plot the magnitudes of the result. The plots should go to only half of the magnitudes, since the other half will be a mirror image.

6.11 Summary

This chapter covers the Fourier transform, inverse Fourier transform, and related topics. The Fourier transform gives frequency (spectral) content of a signal. Given any series of samples, we can create a sum of sinusoids that approximate it, if not represent it exactly.

6.12 Review Questions

1. Suppose we were to sample a signal at $f_s = 5000$ samples per second, and we were to take 250 samples. After performing the DFT, we find that the first 10 results are as follows. What does this say about the frequencies that are present in our input signal? (Assume that the other values for $X[m]$ up to $m = 125$ are 0.)

$$X[m] = 10, 0, 0, 2 + j4, 0, 1, 0, 0, 0, 2 - j4$$

2. An input sequence, $x[n]$, has the Fourier transform performed on it. The result is:
$X[m]$ = {3, 2+j4, 1, 5-j3, 0, 0, 0, 5+j3, 1, 2-j4}.

   a. Find (and plot) the magnitudes and phase angles.

   b. You should notice some symmetry in your answer for the first part. What

kind of symmetry do you expect (for the magnitudes), and why?

3. An input sequence, $x[n]$, has the Fourier transform performed on it. The result is:
   $X[m] = \{3,\ 2+j4,\ 1,\ 5-j3,\ 0,\ 0,\ 0,\ 5+j3,\ 1,\ 2-j4\}$.
   Given that $x[n]$ was sampled at $f_s = 100$ samples per second,

   a. What is the DC component for this signal?
   b. What frequencies are present in the input? Rank them in order according to amplitude.
   c. Using MATLAB, find $x[n]$.

4. Given $x_2 = [0.4786, -1.0821, -0.5214, -0.5821, -0.2286, 1.3321, 0.7714, 0.8321]$; find (using MATLAB) the FFT values $X_{magnitude}[m]$ and $X_{phase}[m]$ for $m =$ 0..7. Show all of your work. Also, graph your results.

5. Try a 5-tap FIR filter using the following random data. Note that this command will return different values every time.

   ```
   x = round(rand(1, 20)*100)
   ```

   Use $h_1[k] = \{0.5, 0.5, 0.5, 0.5, 0.5\}$, and compare it to $h_2[k] = \{0.1,\ 0.3,\ 0.5,\ 0.3,\ 0.1\}$, and $h_3[k] = \{0.9, 0.7, 0.5, 0.7, 0.9\}$. Graph the filter's output, and its frequency magnitude response. Be sure to use the same x values. Make one graph with the filters' outputs, and another graph with the frequency responses. Judging each set of filter coefficients as a lowpass filter, which is best? Which is worst? Why?

6. Try an 8-tap FIR filter using the following random data. Note that this command will return different values every time.

   ```
   x = round(rand(1, 20)*100)
   ```

   Use $h_1[k] = \{0.5,\ 0.5,\ 0.5,\ 0.5,\ 0.5,\ 0.5,\ 0.5,\ 0.5\}$, and compare it to $h_2[k] = \{0.1,\ 0.2,\ 0.3,\ 0.5,\ 0.5,\ 0.3,\ 0.2,\ 0.1\}$, and $h_3[k] = \{0.9,\ 0.7,\ 0.6,\ 0.5,\ 0.5,\ 0.6,\ 0.7,\ 0.1\}$. Graph the filter's output and its frequency magnitude response. Be sure to use the same x values. Make one graph with the filters' outputs, and another graph with the frequency responses. Judging each set of filter coefficients as a lowpass filter, which is best? Which is worst? Why?

7. What is $3e^{-j2\pi 0.2}$ in complex Cartesian coordinates? (That is, in the form $a + jb$.) Hint: use Euler's formula.

8. What is $1.7 - j3.2$ in complex polar coordinates? (That is, in the form $re^{j\phi}$.) Hint: use the magnitude and angle calculations from Chapter 1, "Introduction."