CHAPTER 14

Processing Sound

Chapter Objectives

This chapter discusses the following:

- How sound is physically recorded and played back and our internal storage of sound
- Operations that can be performed with the original time trace
- The ability to transform the data into the frequency domain and the physical significance of the transformed data
- Operations that can be performed in the frequency domain

14.1 The Physics of Sound

Any sound source produces sound in the form of pressure fluctuations in the air. While the air molecules move infinitesimal distances in order to propagate the sound, the important part of sound propagation is that pressure waves move rapidly through the air by causing air molecules to "jostle" each other. These pressure fluctuations can be viewed as analog signals—data that have a continuous range of values. These signals have two attributes: their amplitude and their frequency characteristics.

In absolute terms, sound is measured as the **amplitude** of pressure fluctuations on a surface like an eardrum or a microphone. However, the challenging characteristic of these data is their dynamic range. Our ears are able to detect small sounds with amplitudes around 10^{10} (10 billion) times smaller than the loudest comfortable sound. Sound intensity is therefore usually reported logarithmically, measured in decibels where the intensity of a sound in decibels is calculated as follows:

$$I_{DB} = 10 \log_{10}(I / I_0)$$

where I is the measured pressure fluctuation and I_0 is a reference pressure usually established as the lowest pressure fluctuation a really good ear can detect, 2×10^{-4} dynes/cm^2.

Also, sounds are pressure fluctuations at certain **frequencies**. The human ear can hear sounds as low as 50 Hz and as high as 20 kHz. Voices on the telephone sound odd because the upper frequency is limited by the telephone equipment to 4 kHz. Typically, hearing damage due to aging or exposure to excessive sound levels causes an ear to lose sensitivity to high and/or low frequencies.

14.2 Recording and Playback

Early attempts at sound recording concentrated first on mechanical, and later magnetic, methods for storing and reproducing sound. The phonograph/record player depended on the motion of a needle in a groove as a cylinder or disk rotated at constant speed under the playback head. Not surprisingly, when you see the incredible dynamic range required, even the best stereos could not reproduce high-quality sound. Later, analog magnetic tape in various forms replaced the phonograph, offering less wear on the recording and better, but still limited, dynamic range. Digital recording has almost completely supplanted analog recording and will be the subject of this chapter.

Of course, sound amplitude in analog form is unintelligible to a computer—it must be turned into an electrical signal by a microphone, amplified to suitable voltage levels, digitized, and stored, as shown in

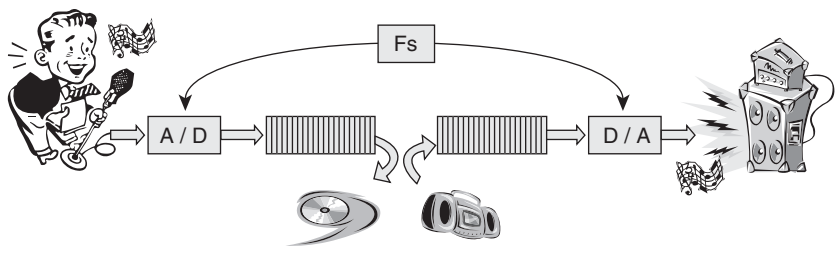

Figure 14.1 *Mechanics of sound recording and playback*

Figure 14.1. The key to successful digital recording and playback—whether by digital tape machines, compact disks, or computer files—is the design of the analog-to-digital (A/D) and digital-to-analog (D/A) devices. The reader should remember that this is still low-level data. Each word coming out of the A/D or going into the D/A merely represents the pressure on the microphone at a point in time.

The primary parameter governing the sound quality is the recording rate—how quickly the mechanism records samples of the sound (the sampling rate). Basic sampling theory suggests that we should use a sampling rate twice the highest frequency you are interested in reproducing, usually around 20,000 samples per second for good music, 5,000 samples per second for speech.

The other parameter, the resolution of the recorded data, has remarkably little effect on the quality of the recording to an untrained ear. The resolution is usually either 8 bits (−128 to 127) or 16 bits (−32,768 to 32767). While 8-bit resolution ought to offer very limited dynamic range, and theoretically should be used only for recording speech, in practice it results in a quality of reproduction for music that is, to an untrained ear, indistinguishable from that provided by 16-bit resolution.

Technical Insight 14.1

The background theory of sampling is beyond the scope of this text. Interested readers should research Nyquist on a good search engine.

These parameters must be stored with any digital sound recording medium and retrieved by the tools that play those sounds. To be able to play such a file, we must receive not only the data stream, but also information indicating the sample frequency, `Fs`, and the word size.

14.3 Implementation

MATLAB offers a number of tools for reading sound files: `wavread(...)` for `wav` files and `auread(...)` for `.au` files, for example. Both return three variables: a vector of sound values, the sampling frequency in Hz (samples per second), and the number of bits used to record the data (8 or 16).

To play a sound file, MATLAB provides the function `sound(data, rate)` where `data` is the vector of sound values, and `rate` is the playback frequency, usually the frequency at which the sound values were recorded. We will see that the function `sound(...)` passes the data directly to the computer's sound card, but different implementations will manage the behavior of the software that plays the sound in one of two ways.

Blocking vs. Non-blocking: "Blocking" refers to the behavior of your system after you have called the `sound(...)` function to play a sound. Blocking players will not return control to the code playing the sound until the sound has completed. This will allow only one sound to be played from an application at a time. Non-blocking players will not wait for the sound card to finish playing the sound, so multiple calls to the `sound(...)` function will overlay different sounds. You will need to experiment with your particular system to determine whether it blocks or not.

A number of `.wav` files are included on the book's Companion Web site to demonstrate many aspects of sound files.

14.4 Time Domain Operations

First, we consider three kinds of operations on sound files in the time domain: slicing, playback frequency changes, and sound file frequency changes.

14.4.1 Slicing and Concatenating Sound

Consider the problem of constructing comedic sayings by choosing and assembling words from published speeches. The Companion Web site contains a sampling of speech clips selected from various Web sites. In particular, it has the *Apollo 13* speech, "Houston, we have a problem"; "Frankly, my dear . . ." from *Gone with the Wind*; and "You can't handle the truth" from *A Few Good Men*. Exercise 14.1 describes the process of assembling parts of these speeches into a semi-coherent conversation.

The first part of Exercise 14.1 reads the *Apollo 13* speech, plays the speech, and plots the data (with the data index as x-axis). The resulting plot is shown in the left half of Figure 14.2. Since the sound actually includes more than we need, the next step is to crop this file to keep only the words we

Exercise 14.1 Locating the first part of the speech

```
>> [houston, Fsh] = wavread('a13prob.wav');
>> subplot(1, 2, 1)
>> plot(houston);
>> sound(houston, Fsh);
```

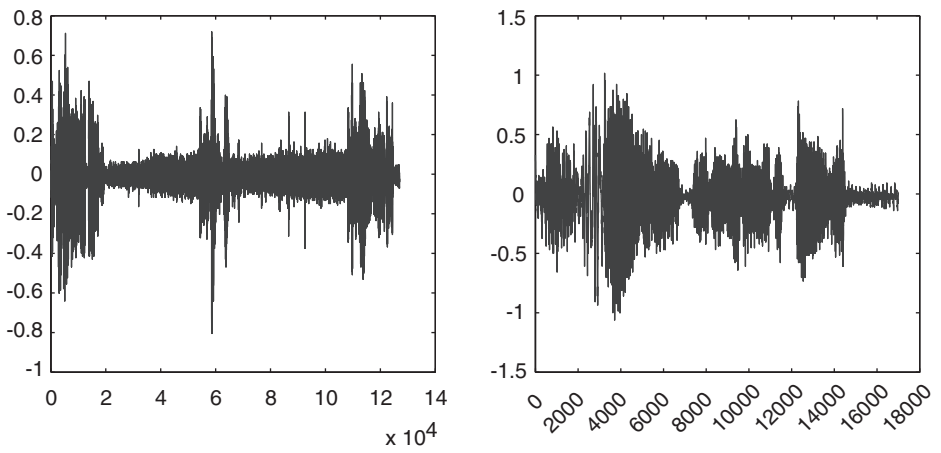

Figure 14.2 *Apollo 13 speech*

Exercise 14.2 Extracting the first part of the speech

```
>> clip = 110000;
>> prob = houston(clip:end)*2;
>> subplot(1, 2, 2)
>> plot(prob)
```

need. By listening to the speech using the function `sound(...)`, and judiciously zooming and panning the plot, it is possible to narrow down the location in the file where the problem speech starts, at about 111000. In Exercise 14.2 you will extract the first part of the speech.

In Exercise 14.2 we truncate the speech file to the words we need and also, realizing that the amplitude of these words is a little low, raise its amplitude by a factor of 2.

In Exercise 14.3, by a similar process, we remove "my dear" from the "frankly, my dear . . ." speech, reducing its amplitude by one-half, which results in Figure 14.3.

Exercise 14.3 Extracting "my dear"

```
>> figure
>> [damn, Fsd] = wavread('givdamn2.wav');
>> subplot(1, 2, 1)
>> plot(damn);
>> lo = 4500;
>> hi = 8700;
>> sdamn = [damn(1:lo); damn(hi:end) ] * .5;
>> subplot(1, 2, 2)
>> plot(sdamn);
```

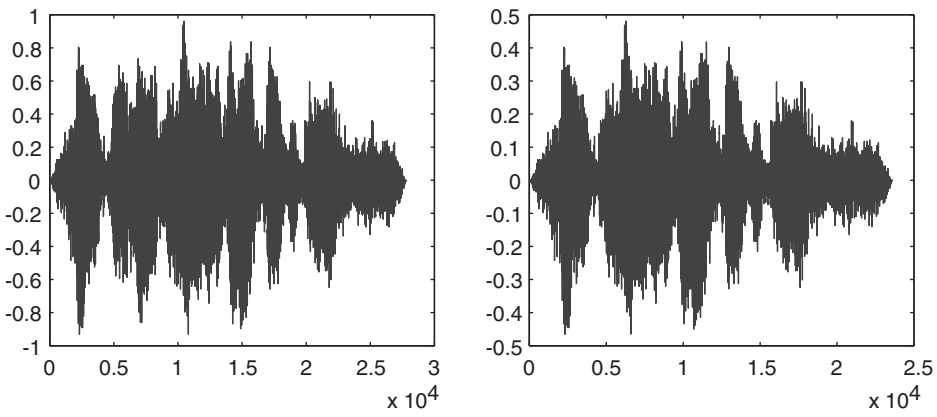

Figure 14.3 *Gone with the Wind speech*

Finally, in Exercise 14.4, we assemble the complete speech by concatenating these two fragments with the speech from *A Few Good Men*. The resulting picture is shown in Figure 14.4.

Exercise 14.4 Assemble the speech

```
>> [truth, Fst] = wavread('truth1.wav');
>> speech = [prob; sdamn; truth * .7];
>> figure
>> plot(speech);
>> sound(speech, Fst);
```

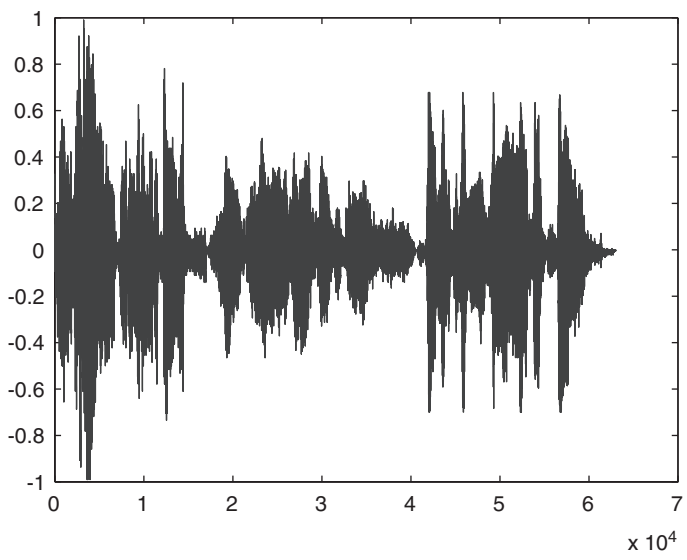

Figure 14.4 *The complete speech*

14.4.2 Musical Background

For good historical reasons, music is usually described graphically on a music score. The graphics describe for each note to be played its pitch and its duration, together with other notations indicating how to introduce expression and quality into the music. However, this graphical notation is not amenable to the simple representation of music we need for these experiments. Rather, we will use the representation illustrated in Figure 14.5. The right side of this figure shows a standard piano keyboard, the index of each white note, and the number of half steps necessary to achieve the pitch of each note. On the left side of the figure, we see the method to be used in this text to describe simple tunes. It will consist of an array with two columns and n rows, where n is the number of notes to be played for each tune. The first column is the key number to play, and the second column is the number of beats each note should be played.

The examples to follow will manipulate the file `piano.wav` to produce a snippet of music. This file is a recording of a single note played on a piano. Other files provided in the Companion Web site are the same note played on a variety of instruments. There are two ways to accomplish this, as follows:

1. Playing each note at a different playback frequency
2. Stretching or shrinking each note to match the required note pitch and playing them all at the same playback frequency

The first way is easier to understand and code, but very inflexible; the second method is a little more difficult to implement, but completely extensible. Musically speaking, if a sound is played at twice its natural frequency, it is heard as one musical octave higher. When you play a scale by playing each white key in turn from one note to the next octave, there are 8 keys to play with 7 frequency changes: 5 whole note steps (those separated by a black note) and 2 half note steps, for a total of 12 half note steps. These 12 half steps are logarithmically divided where the frequency multiplier between half note steps is $2^{1/12}$.

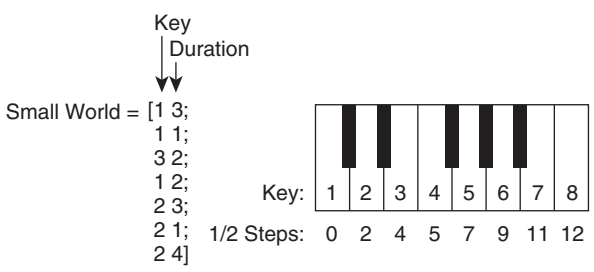

Figure 14.5 *Musical notes*

14.4.3 Changing Sound Frequency

We will leave as an exercise for the reader the question of playing a tune by changing the playback frequency of each note, which is really never a practical thing to do, and concentrate on playing all the notes of a tune with the same playback frequency. This allows the different notes to be copied into a single sound file and saved to be played back on any digital sound system.

In order to change the perceived note frequency without changing the playback frequency, we have to change the number of data samples in the original data file much as we stretched or shrunk an image in Section 13.4.1. Use Exercise 14.5 to experiment with this technique for playing notes at different pitches.

In Exercise 14.5 we first read and play the note at its natural frequency. Then we raise its pitch by removing about one-third of the samples and then lower the pitch by an octave by doubling the number of samples.

Play a Scale Listing 14.1 shows a script that uses this capability to play the C Major scale (all white notes) on the piano. It repeatedly shortens the vector `newNote` to increase the frequency of the note played.

In Listing 14.1:

Lines 1–3: Read the note and set the step multipliers.

Lines 4–12: Play eight notes of a major scale.

Exercise 14.5 Note pitch experiment

```
>> [note Fs] = wavread('instr_piano.wav');
>> sound(note, Fs);
>> sound(note(ceil(1:1.3:end)), Fs);
>> sound(note(ceil(1:0.5:end)), Fs);
```

Listing 14.1 Play a scale by shrinking the note

```
 1. [note, Fs] = wavread('instr_piano.wav');
 2. half = 2^(1/12);
 3. whole = half^2;
 4. for index = 1:8
 5.     sound(note, Fs);
 6.     if (index == 3) || (index == 7)
 7.         mult = half;
 8.     else
 9.         mult = whole;
10.     end
11.     note = note(ceil(1:mult:end));
12. end;
```

Line 5: Plays the note. This implementation uses a blocking `sound(...)` function. If your system does not block, you will need to insert `pause(0.3)` here to wait for most of the note to complete.
Lines 6–10: Choose the appropriate frequency multiplying factor.
Line 11: Shrinks the note file by the chosen factor.

Play a Simple Tune We now write a script to build a playable `.wav` file using the note shrinking technique. The script is shown in Listing 14.2. It uses the array `steps` to decide how many half-tone steps are necessary to reach the nth note on the scale and uses the array `doremi` to define the tune. The first column specifies the relative pitch (the note on the scale) and the second the duration in "beats." The script sets the beat time to be 0.2 seconds.

The goal of the script is to put the notes into a single sound array called `tune`, as illustrated in Figure 14.6, rather than playing the notes "on the fly." This is accomplished as follows:

- Create an empty array, `tune`, of the appropriate length (the length of the original note plus the total number of beats in the song)
- Initialize `storeAt` to store the first note at the start of the tune
- Iterate across the `tune` definition array `doremi` with the following steps:
  - Start with the original `note`
  - Get the key index to decide how many times to raise the `note` array by half a step
  - Raise the `note` to the right pitch and save it as `theNote`
  - Add that `theNote` vector to the `tune` vector, starting at `storeAt`
  - Move the `storeAt` variable down the `tune` vector a distance equivalent to the duration of that note
- When all the notes have been added to the tune file, play the tune and save it as a `.wav` file.

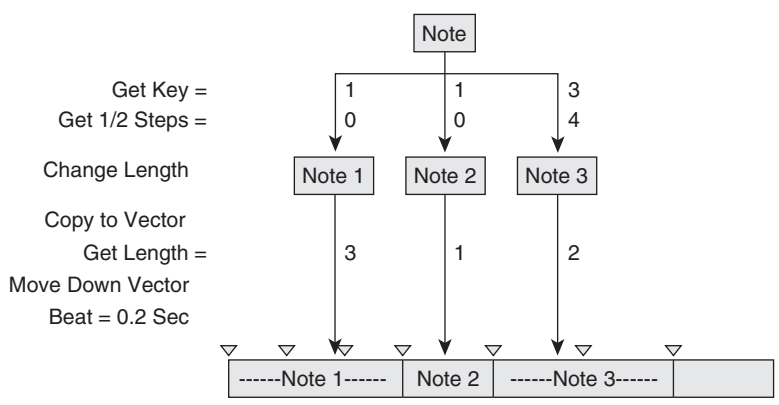

Figure 14.6 *Building a tune file*

Listing 14.2 Building a tune file

```
 1. [note, Fs] = wavread('instr_piano.wav');
 2. half = 2^(1/12);
 3. doremi = [1 3; 2 1; 3 3; 1 1; 3 2; 1 2; 3 4; 2 3;
 4.           3 1; 4 1; 4 1; 3 1; 2 1; 4 8; 3 3; 4 1;
 5.           5 3; 3 1; 5 2; 3 2; 5 4; 4 3; 5 1; 6 1;
 6.           6 1; 5 1; 4 1; 6 4 ];
 7. steps = [0 2 4 5 7 9 11 12];
 8. dt = .2;
 9. nCt = floor(dt*Fs);
10. storeAt = 1;
11. for index = 1:length(doremi)
12.     key = doremi(index,1);
13.     pow = steps(key);
14.     theNote = note(ceil(1:half^pow:end));
15.     noteLength = length(theNote);
16.     noteEnd = storeAt + noteLength - 1;
17.     tune(storeAt:noteEnd,1) = theNote;
18.     storeAt = storeAt + doremi(index,2) * nCt;
19. end
20. sound(tune, Fs)
21. wavwrite(tune, Fs, 'dohAdeer.wav')
```

In Listing 14.2:

Lines 1–6: Read the file and set up the parameters.

Line 7: A vector defining how many half steps it takes to set the frequency of notes 1–8.

Line 8: The time between notes of length 1—the beat of the tune.

Line 9: The number of samples to play for one beat of the tune.

Line 10: Begins storing notes at the beginning of the tune.

Lines 11–19: Insert each note in the song file into the tune file.

Line 12: Fetches the key number.

Line 13: Extracts the number of half steps required for this note.

Line 14: Stretches the original note by this multiplier.

Lines 15 and 16: Compute where the end of the note will be stored.

Line 17: Copies the note into the tune file.

Line 18: Advances the `storeAt` index down the tune file by the beat count multiplied by the beats required for this note.

Lines 20 and 21: Play the complete tune and save it as a `.wav` file.

14.5 The Fast Fourier Transform

Typically, the time history display of a sound shows you the amplitude of the sound as a function of time but makes no attempt at showing the frequency content. While this works for the exercises above, we are often

more interested in the frequency content of a sound file, for which we need a different presentation—a spectrum display.

14.5.1 Background

In general, a spectrum display shows the amount of sound energy in a given frequency band throughout the duration of the sound analyzed but ignores the time at which the sound at that frequency was generated. Many acoustic amplifiers (see Figure 14.7) include two features that allow you to customize the sound output:

- A spectral display that changes values as the sound is played, indicating the amount of sound energy (vertically) in different frequency bands (horizontally)
- Filter controls to change the relative amplification in different frequency bands

In the following paragraphs, we will consider only the analysis of the sound frequency content. The ability to reshape the sound frequency content as the sound plays is beyond the scope of this text.

To achieve the motion of the spectrum display, software to analyze a segment of the sound file runs periodically and updates the spectrum display. Typically, perhaps 20 times a second, 1/20th second of sound file is analyzed and transformed. The software used for this conversion is known as the Fourier transform.

While the mathematics of the Fourier transform is beyond the scope of this book, we can make use of the tools it offers without concerning ourselves with the details. There are a number of implementations of this transform; perhaps the most commonly used is the Fast Fourier Transform (FFT). The FFT uses clever matrix manipulations to optimize the

Figure 14.7 *A typical spectrum display*

mathematics needed to generate the forward (time to frequency) and reverse (frequency to time) transforms.

14.5.2 Implementation

Figure 14.8 illustrates the overall process of transforming between the time domain and frequency domain. It starts with a simple sound file, a vector of `N` sound values in the range (–1.0 to 1.0), which, if played back at a sample frequency `Fs` samples per second, reproduces the sound. The parameters of interest for characterizing the time trace are:

`N`	the number of samples
F_s	the sampling frequency
Δt	the time between samples, computed as $1/F_s$
`Tmax`	the maximum time is $N \times \Delta t$

The FFT consumes a file with these characteristics and produces a frequency spectrum with a corresponding set of characteristics. The frequency spectrum consists of the same number, `N`, of data points, each of which is a complex value with real and imaginary parts. (While many displays actually plot the magnitude of the spectrum values, to accomplish the inverse transform, the complex values must be retained.) The frequency

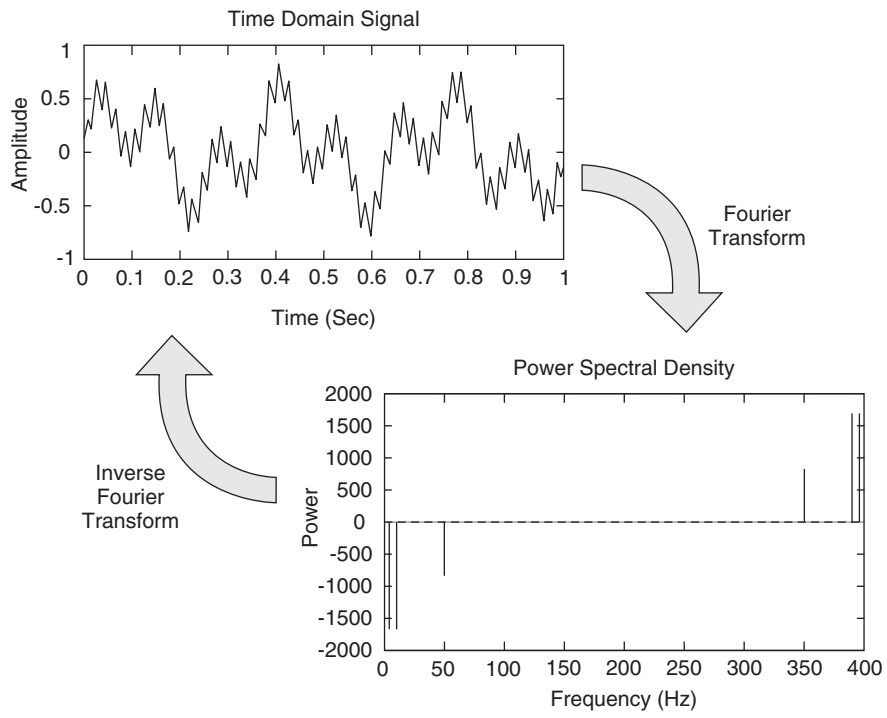

Figure 14.8 *Mechanics of the Fourier Transform*

values are "folded" on the plot so that zero frequency occurs at either end of the spectrum, and the maximum frequency occurs in the middle, at spectrum data point `N/2`.

The equivalent characteristics for the spectrum data are as follows:

`N` the number of samples

Δf the frequency difference between samples, computed as $1/T_{max}$

F_{max}, the frequency value at the end of the plot, is $N \times \Delta f$. However, since the mathematics force this frequency to actually replicate the beginning frequency, the maximum effective frequency actually occurs at the mid-point with value $F_{max}/2$.

Technical Insight 14.2

The fact that the actual maximum frequency is half of the sampling frequency is consistent with the Nyquist criterion that the maximum frequency you can discern with digital sampling is half the sampling frequency.

The FFT is mechanized using the function `fft(...)`, which consumes the time history and produces the complex spectrum file. The inverse FFT function, `ifft(...)`, takes a spectrum array and reconstructs the time history. This pair of functions provides a powerful set of tools for manipulating sound files.

14.5.3 Simple Spectral Analysis

Listing 14.3 illustrates a script that creates 10 seconds of an 8 Hz sine wave, plots the first second of it, performs the FFT, and plots the real and imaginary parts of the spectrum. Notice the following:

- A sine wave in the time domain transforms to a line in the frequency domain because all its energy is concentrated at that frequency—8 Hz in this example.
- Since the FFT is a linear process, multiple sine or cosine waves added together at different frequencies have additive effects in the spectrum.
- The resulting spectrum is complex (with real and imaginary parts) and symmetrical about its center, the point of maximum frequency. On the plot, of course, one cannot make the frequency axis labels reduce from the center to the end.
- The real part of the spectrum is mirrored about the center; the imaginary part is mirrored and inverted (the complex conjugate of the original data).
- The phase of the complex spectrum retains the position of the sine wave in the time domain—it would be totally real for a cosine wave symmetrically placed in time and totally imaginary for a sine wave in the same relationship.

Listing 14.3 FFT of a sine wave

```
 1. dt = 1/400              % sampling period (sec)
 2. pts = 10000             % number of points
 3. f = 8                   % frequency
 4. t = (1:pts) * dt;       % time array for plotting
 5. x = sin(2*pi*f*t);
 6. subplot(3, 1, 1)
 7. plot(t(1:end/25), x(1:end/25));
 8. title('Time Domain Sine Wave')
 9. ylabel('Amplitude')
10. xlabel('Time (Sec)')
11. Y = fft(x);             % perform the transform
12. df = 1 / t(end)         % the frequency interval
13. fmax = df * pts / 2
14. f = (1:pts) * 2 * fmax / pts;
                            % frequencies for plotting
15. subplot(3, 1, 2)
16. plot(f, real(Y))
17. title('Real Part')
18. xlabel('Frequency (Hz)')
19. ylabel('Energy')
20. subplot(3, 1, 3)
21. plot(f, imag(Y))
22. title('Imaginary Part')
23. xlabel('Frequency (Hz)')
24. ylabel('Energy')
```

The script in Listing 14.3 creates three sub-plots: the original sine wave and then the amplitude and phase of the spectrum.

In Listing 14.3:

Lines 1–5: Set up the time domain signal.

Lines 6–10: Plot the front part of the time trace.

Line 11: Performs the FFT.

Lines 12–14: Set up the frequency plots.

Lines 15–19: Plot the spectrum real part.

Lines 20–24: Plot the spectrum imaginary part.

Figure 14.9 shows the result from running this script. It confirms the earlier statement that the real part of the spectrum is mirrored about the center frequency, and the imaginary part is mirrored and inverted.

14.6 Frequency Domain Operations

As a typical example of operating in the frequency domain, we will consider analyzing the spectral quality of different musical instruments. The intent of this section is to develop a plot showing the spectra of a selection of different musical instruments. We will first build a function that plots the spectrum for a single instrument and then build the script to create all the

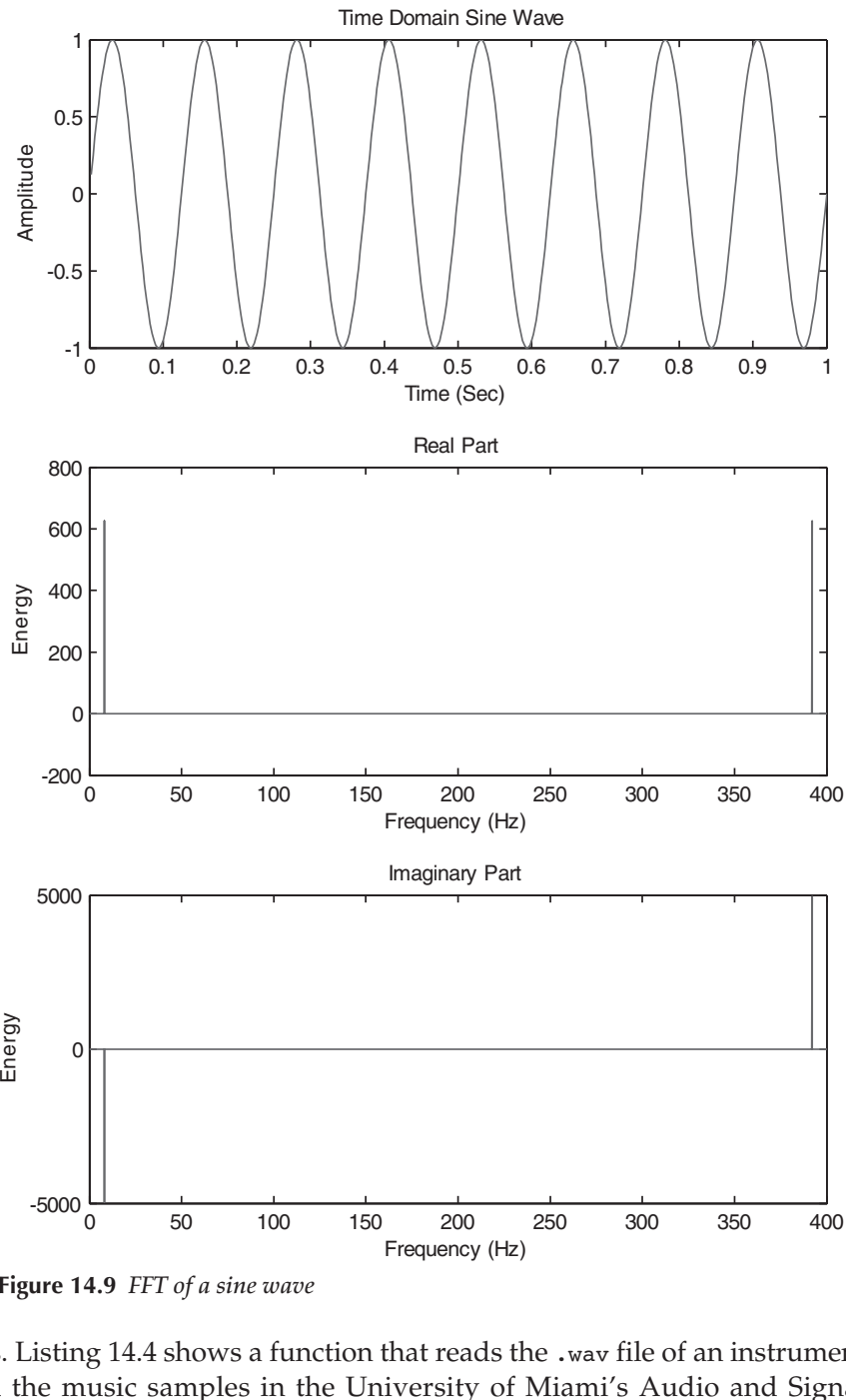

Figure 14.9 *FFT of a sine wave*

plots. Listing 14.4 shows a function that reads the `.wav` file of an instrument from the music samples in the University of Miami's Audio and Signal Processing Laboratory.[1] All the instruments are carefully playing a note at about 260 Hz.

[1]http://chronos.ece.miami.edu/~dasp/samples/samples.html

Listing 14.4 Plotting the spectrum of one instrument

```
1. function inst(name, ttl)
   % plot the spectrum of the instrument with
   % the given name, with the given plot title
2. [x, Fs] = wavread(['instr_' name '.wav']);
3. N = length(x);
4. dt = 1/Fs;       % sampling period (sec)
5. t = (1:N) * dt;  % time array for plotting
6. Y = abs(fft(x)); % perform the transform
7. mx = max(Y);
8. Y = Y * 100 / mx;
9. df = 1 / t(end) ; % the frequency interval
10. fmax = df * N / 2 ;
11. f = (1:N) * 2 * fmax / N;
12. up = floor(N/10);
13. plot(f(1:up), Y(1:up) );
14. title(ttl)
15. xlabel('Frequency (Hz)')
16. ylabel('Energy')
```

In Listing 14.4:

Line 1: Shows a function consuming two strings: the name of the instrument and the title of the plot.

Lines 2–5: Read the file and set up the plot parameters.

Line 6: Performs the FFT and computes the absolute value.

Lines 7 and 8: Scale the plot to be a percentage of the maximum energy at any frequency.

Lines 9–16: Set up and plot the first 10% of the spectrum.

The script that uses this function to plot the instrument data is shown in Listing 14.5.

In Listing 14.5:

Line 1: Sets up the sub-plots configuration.

Lines 2–17: Each pair of lines makes the sub-plots of one instrument.

Listing 14.5 Script to plot eight-instrument spectra

```
1. rows = 4; cols = 2
2. subplot(rows, cols, 1)
3. inst('sax', 'Saxophone');
4. subplot(rows, cols, 2)
5. inst('flute', 'Flute');
6. subplot(rows, cols, 3)
7. inst('tbone', 'Trombone');
8. subplot(rows, cols, 4)
9. inst('piano', 'Piano');
10. subplot(rows, cols, 5)
11. inst('tpt', 'Trumpet');
```

continued on next page

```
12. subplot(rows, cols, 6)
13. inst('mutetpt', 'Muted Trumpet');
14. subplot(rows, cols, 7)
15. inst('violin', 'Violin');
16. subplot(rows, cols, 8)
17. inst('cello', 'Cello');
```

The results are shown in Figure 14.10. It is interesting to notice the following:

- None of the instruments produce a pure tone. The lowest frequency at which there is energy is usually called the fundamental

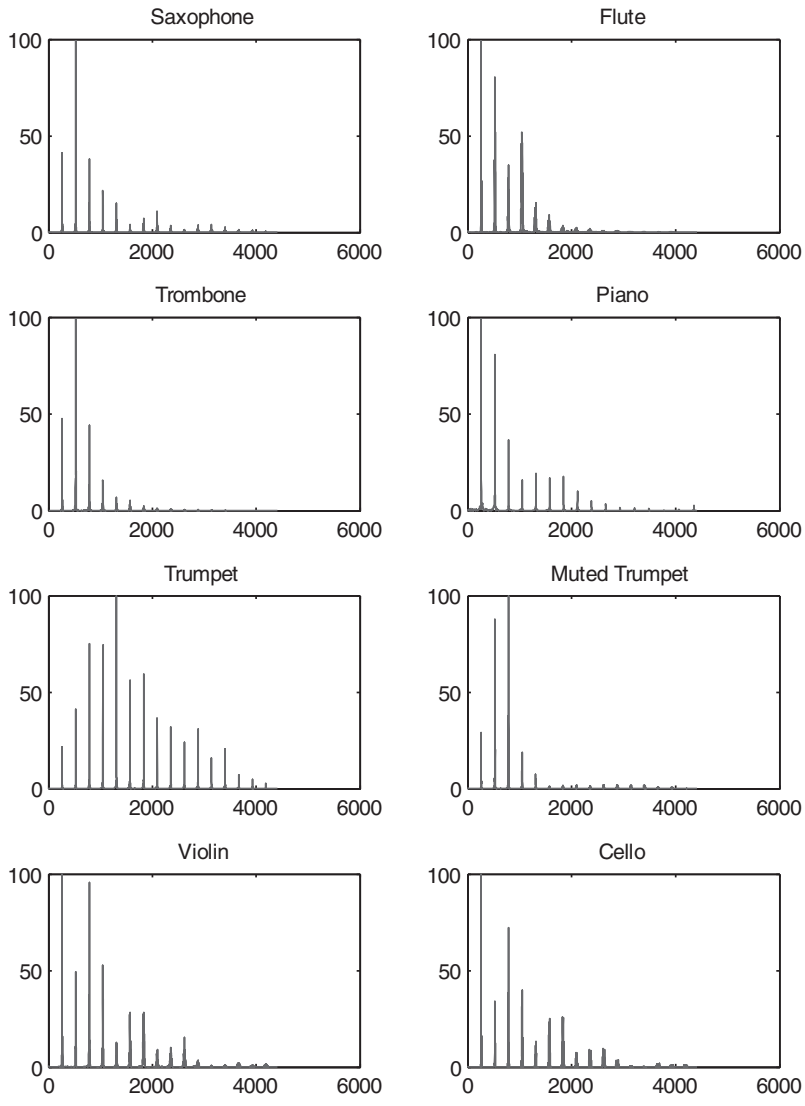

Figure 14.10 *Instrument spectra*

frequency, and successive peaks to the right at multiples of the fundamental frequency are referred to, for example, as the first, second, and third harmonics.

- Several instruments have much more energy in the harmonics than in the fundamental frequency.
- "Families" of instruments have similar spectral shapes—the strings, for example, have strong fundamental and second harmonic energy. In principle, these characteristic spectral "signatures" can be used to synthesize the sound of instruments, and even to identify individual instruments when played in groups.

14.7 Engineering Example—Music Synthesizer

A music synthesizer is an electronic instrument with a piano style keyboard that is able to simulate the sound of multiple instruments. Unlike the instrument sounds we have used so far, the instrument sounds are not stored as large time histories. Rather, they are stored as the Fourier coefficients similar to those illustrated in Figure 14.10. The sound is then reconstructed by multiplying sin or cosine waves of the right frequency by the stored coefficients. For some instruments, this is sufficient. Other instruments such as pianos need to have the amplitude of the resulting sound modified to match a typical profile. Listing 14.6 illustrates a possible technique for extracting the most important Fourier coefficients from the piano sound. The result will be a little disappointing because the sound does not fade with time. We will need some techniques from the next chapter to complete the story.

Listing 14.6 Synthesizing a piano

```
[snd Fs] = wavread('instr_piano.wav');
N = length(snd)
sound(snd, Fs)
tMax = N / Fs
dt = 1 / Fs
Y = fft(snd);
Ns = N/4;
fMax = Fs/4;
df = fMax / Ns;
f = ((1:Ns) - 1) * df;
rl = real(Y(1:Ns));
im = imag(Y(1:Ns));
plot(f, abs(Y(1:Ns)))
xlabel('frequency (Hz)')
ylabel('real amplitude')
zlabel('imag amplitude')
amps = abs(Y(1:end/2));
Nc = 25;
for ndx = 1:Nc
   [junk where] = max(amps);
```

continued on next page

```
21.    C(ndx).freq = where;
22.    C(ndx).coeff = Y(where);
23. amps(where-25:where+25) = 0;
24. end
25. frq = [C.freq];
26. [frq order] = sort(frq);
27. sortedStr = C(order);
28. Nt = 25;
29. t = (1:2*Fs) * dt;
30. f = zeros(1, length(t));
31. for ndx = 1:Nt
32. w = frq(ndx) * df * 2 * pi;
33. ct = cos(w*t);
34. st = sin(w*t);
35. Cf = sortedStr(ndx).coeff;
36. f = f + real(Cf) * ct + imag(Cf) * st;
37. end
    % amplitude shaping goes here
38. sf = f ./ max(f);
39. sound(sf, Fs)
```

In Listing 14.6:

Lines 1–5: Read the sound file and compute the representative parameters.

Lines 6–9: Perform the FFT and compute its representative parameters and Ns, the number of samples we are interested in.

Lines 10–12: Compute a vector of the frequencies and extract the real and imaginary coefficients.

Lines 13–16: Plot the coefficient absolute values (see Figure 14.11).

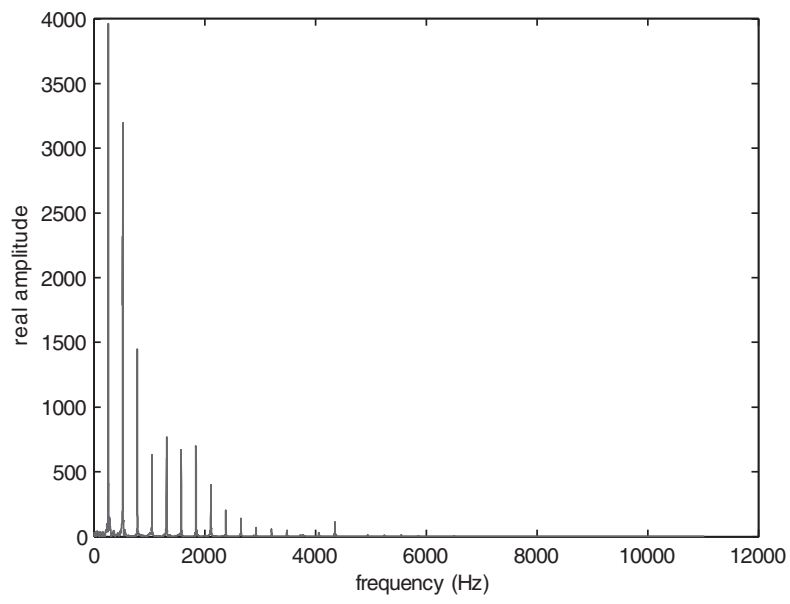

Figure 14.11 *Instrument spectrum*

Line 17: Stores the absolute values of the coefficients.

Lines 18–24: Extract the 25 largest coefficients by first finding the maximum absolute coefficient (Line 20), saving the frequency and complex amplitudes (Lines 21–22), and then removing that peak from the amplitude vector.

Lines 25–27: Sort the complex coefficients in frequency order.

Lines 29 and 30: Set up the time trace parameters and storage.

Lines 31–37: Build the sound file composed of the real coefficients times the cosine of the frequency and the imaginary coefficients times the sine of the frequency.

Lines 38–39: Scale and play the sound.

We will complete this synthesis for a piano sound in the next chapter.

Chapter Summary

This chapter presented the following:

- Sounds are read with specific readers that provide a time history and sampling frequency
- Sounds can be played through the computer's sound system and saved to disk as a sound file ready for playing on any digital player
- We can slice and concatenate sounds to edit speeches and change the frequency of the sound to change its pitch
- We can analyze the frequency content of sound using the Fast Fourier Transform (FFT)
- We can modify the spectra by adding, deleting, or changing the sound levels at chosen frequencies under certain controlled conditions
- We can reconstruct a sound from the FFT coefficients.

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`[data Fs nb] = auread(file)`	Reads an `.au` sound file in `.wav` format	14.3
`auwrite((data, Fs, nb, file)`	Writes a sound file in `.au` format	14.3
`fft(ftime)`	Performs the Fast Fourier Transform on a sound file	14.5.2
`ifft(ffreq)`	Performs the inverse Fourier Transform on a spectrum file	14.5.2

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`sound(data, Fs)`	Plays a sound file	14.3
`[data Fs nb] = wavread(file)`	Reads a `.wav` sound file in `.wav` format	14.3
`wavwrite(data, Fs, nb, file)`	Writes a sound file in `.wav` format	14.3

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. Playing a sound file at double the recorded sample frequency raises its pitch by an octave.
2. Removing every other sample from a sound file lowers the pitch by an octave.
3. The resolution of the recorded data has a significant effect on the quality of the recording.
4. After performing an FFT, the zero frequency occurs at either end of the spectrum and the maximum frequency occurs in the middle.
5. Since the mathematics of the FFT are linear, the spectrum of a sound added in the time domain is also added in the frequency domain.

Fill in the Blanks

1. Sound pressure fluctuations have two attributes: their _____ and their _____.
2. Each word coming out of the _____ or going into the _____ merely represents the _____ on the microphone at a point in time.
3. The steps from one note to the next higher octave are divided into _____ increments: _____ whole note steps and _____ half note steps, for a total of _____ half note steps.
4. A spectrum display shows the amount of _____ in a given _____ throughout the duration of the sound analyzed.

Programming Projects

1. These are fundamental exercises with sound files. You should not hard-code any of the answers for this problem, and you should not need iteration.
   a. Select and read a suitable `.wav` file, and save the sound values and sampling frequency.
   b. Create a new sound that has double the frequency of the original sound, and store your answer in the variable `sound_Double`.
   c. Create a new sound that is the same as the original except that the pitch is raised by five half tones. Store your answer in the variable `raised_pitch`.
   d. We need a figure showing two views each of these three sounds, created using `subplot`. In the left column, plot the original sound, `sound_Double`, and `raised_pitch`, labeling each plot accordingly.
   In the right column, plot the first quarter of the values of the power spectrum of each sound with the proper frequency values on the horizontal axis.
   e. Play each of the sounds in the following order: original sound, `sound_Double`, and `raised_pitch` each at the original sampling frequency.
2. Write a function that will accept a string specifying a sound file and do the following:
   a. Play back the sound.
   b. Plot the sound in the time domain, titling and labeling your plot appropriately.
   c. Compute the frequency with the most energy in this file. Validate your answer by plotting the lower quarter of the frequencies of the Fourier Transform of the sound. Don't forget that the Fourier Transform is complex; you will need to reason with and plot the absolute value of the spectrum.
   d. Test this function with suitable `.wav` files.
3. Write a function named `plotSound` that takes in the name of a sound file and produces a 1×2 figure with two plots. The first plot should be a plot of the sound in the time domain. The second plot should be a plot of the sound in the frequency domain. Your function should not return anything. Label the first plot `'Time Domain'` and label its axes appropriately. Label second plot `'Frequency Domain'` and label its axes appropriately.

   The Time Domain plot should be an amplitude vs. time plot. For simplicity make sure that your time vector starts at `dt` (delta time) and goes to `n*dt` (t_{max}) where n is the number of samples.

The Frequency Domain plot should be a power vs. frequency plot where power is the absolute value of the FFT of the amplitude values. For simplicity make sure that your frequency vector starts at `df` (delta frequency) and goes to `n*df` ($2*f_{max}$).

4. In this exercise, we will write a script to create an instrument sound from scratch.
   a. Create a vector, `t`, of time values from 0 to 2 seconds with length 40,000 samples.
   b. Convert the frequency of middle C (261.6 Hz) to ω radians per second.
   c. Compute a sound sample as $\cos(\omega t)$ over the range of `t` in part a.
   d. Play that sound at a sample frequency of 20,000, and verify that it sounds "about right."
   e. Perform the Fourier Transform on the sound vector, establish the correct axis values, and prove that the sound is exactly Middle C.

5. Write a function named `playNote` that takes in a string representing a note on the piano. Your function should return a vector representing the amplitude values of the note in addition to the correct sampling frequency to be used to play it back. You should do this by modifying the sound in the provided `instr_piano.wav` file which is Middle C played on the piano. Note that the returned sampling frequency should be the same as that in `instr_piano..wav`.

   Here is a list of all the possible note names representing notes that your function should work with and below that is the number of half steps above/below the middle C for that note:

```
 cn cn#  dn dn# en fn fn# gn gn# a(n+1) a(n+1)# b(n+1) c(n+1)
-12 -11 -10  -9 -8 -7  -6 -5  -4     -3      -2     -1      0
```

   where `c4` is the middle C, `c5` is 1 octave above it, and `c3` is 1 octave below it. Similarly, `f5` is 1 octave higher than `f4`, etc. For example, `[y1 fs] = playNote('c5');` should return a vector such that `sound(y1, fs)` should sound like middle C

6. Finally, you will use these tools to play your favorite song.
   a. Find the music for your favorite song, and translate it into the symbology of Problem 14.5.
   b. Write a script that uses the `playNote` function to play your song on the piano.
   c. Modify `playNote` to use your synthetic instrument from Problem 14.3, and save it as `playSynthetic`.
   d. Write a script that uses `playSynthetic` to play your song in futuristic style.