Spectrum Analysis using Java, Complex Spectrum and Phase Angle

Baldwin discusses the complex spectrum and explains the relationship between the phase angle and shifts in the time domain.

Published: September 21, 2004

By Richard G. Baldwin

Java Programming, Notes # 1484

- Preface
- Preview
- Discussion and Sample Code
- Run the Program
- Summary
- What's Next?
- Complete Program Listing

Preface

Spectral analysis

A previous lesson titled Fun with Java, How and Why Spectral Analysis Works explained some of the fundamentals regarding spectral analysis.

The lesson lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm presented and explained several Java programs for doing spectral analysis, including both DFT programs and FFT programs. That lesson illustrated the fundamental aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

The lesson titled Spectrum Analysis using Java, Frequency Resolution versus Data Length used similar Java programs to explain spectral frequency resolution.

In this lesson, I will deal with issues involving the complex spectrum, the phase angle, and shifts in the time domain.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

In this lesson, I will present and explain a program named **Dsp034**. This program will be used to illustrate the behavior of the complex spectrum and the phase angle for several different scenarios.

InIn addition, I will use the following programs that I explained in the lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.

- **ForwardRealToComplex01** - Class that implements the DFT algorithm for spectral analysis.
- **Graph03** - Used to display various types of data. *(The concepts were explained in an earlier lesson.)*
- **Graph06** - Also used to display various types of data in a somewhat different format. *(The concepts were also explained in an earlier lesson.)*

Discussion and Sample Code

The program named **Dsp034**, when run in conjunction with either **Graph03** or **Graph04** computes and plots the amplitude, real, imaginary, and phase angle spectrum for a pulse that is read from a file named **Dsp034.txt**. If that file doesn't exist in the current directory, the program uses a set of default parameters that describe a damped sinusoidal pulse. The program also plots the pulse itself in addition to the spectral analysis results listed above.

The order of the plotted results

When the data is plotted *(see Figure 2)* using the programs **Graph03** or **Graph06**, the order of the plots from top to bottom in the display is:

- The pulse
- The amplitude spectrum
- The real spectrum
- The imaginary spectrum

- The phase angle in degrees

Parameter file format

Each parameter value must be stored as characters on a separate line in the file named **Dsp034.txt**. The required parameters and their order and type are as follows:

- Data length as type **int**
- Pulse length as type **int**
- Sample number representing zero time as type **int**
- Low frequency bound as type **double**
- High frequency bound as type **double**
- Sample values for the pulse as type **double** or **int** *(they are automatically converted to type **double** if they are provided as type **int**)*

The number of sample values for the pulse must match the value for the pulse length.

Format for frequency specifications

All frequency values are specified as a **double** representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is half the sampling frequency.

Sample parameters for testing

Figure 1 provides a set of sample parameter values that can be used to test the program. This sample data describes a triangular pulse. Be careful when you create the file containing these values. Don't allow blank lines at the end of the data in the file.

```
400
11
0
0.0
0.5
0
0
0
45
90
135
90
45
0
0
0
```

Figure 1

The plotting programs

The plotting program that is used to plot the output data from this program requires that the program implement **GraphIntfc01**.

(I explained the plotting programs and this interface in earlier lessons.)

For example, the plotting program named **Graph03** can be used to plot the data produced by this program. When it is used, the following should be entered at the command-line prompt:

java Graph03 Dsp034

Spectral analysis

A static method named **transform** belonging to the class named **ForwardRealToComplex01** is used to perform the actual spectral analysis.

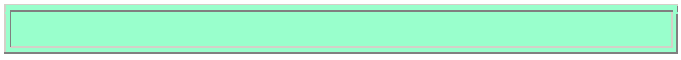
*(I explained this class and the **transform** m method in the earlier lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm. However, I skipped over that portion of the method that computes the phase angle. I will explain that portion of the **transform** method in this lesson.)*

The method named **transform** does not implement an FFT algorithm. Rather, it implements a DFT algorithm, which is more general than, but much slower than an FFT algorithm. *(See the program named Dsp030 in the lesson titled* Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm *for the use of an FFT algorithm.).*

Let's see some results

Before getting into the technical details of the program, let's take a look at some results. Figure 2 shows the results produced by using the program named **Graph06** to plot the output for the program named **Dsp034** using default parameters.

*(The file named **Dsp034.txt** did not exist in the current directory when the results shown in Figure 2 were generated.)*



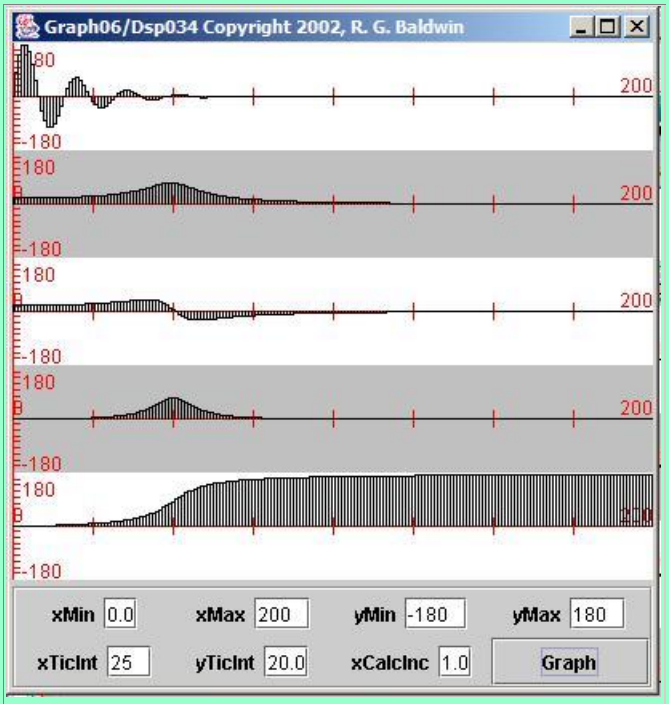


Figure 2 Spectral analysis of a damped pulse.

The input pulse

First consider the input pulse shown in the top plot of Figure 2. This is the sort of pulse that you would get if you attached a mass to a spring, gave it a swift kick, and then allowed the mass to come to rest in an unimpeded fashion.

The horizontal axis represents time moving from left to right. The values above and below the axis represent the position of the mass over time relative to its position at rest.

Potential energy in the spring

When the mass is at the most extreme positions and getting ready to reverse directions, the spring is extended. Thus, potential energy is stored in the spring. At these points in time, there is no kinetic energy stored in the mass.

Kinetic energy in the mass

As the mass goes through the rest position heading towards the other side, the spring is no longer extended, and there is no potential energy stored in the spring. However, at that point in time, the mass is moving causing it to have kinetic energy.

An exchange of energy

The behavior of the spring/mass system is to exchange that energy between potential energy and kinetic energy until such time as energy losses due to friction dissipate the energy. At that point in time, the entire system comes to rest, which is the case about one-third of the way across the top plot in Figure 2.

Now that our physics lesson for the day is over, let's get back to programming and digital signal processing.

Capturing the position of the mass as a sampled time series

Assume that you use an analog to digital converter to capture the position of the mass at a set of uniformly spaced intervals in time and then you plot those samples along a time axis. You should see something resembling the top plot in Figure 2.

Having captured that positional information as a set of uniformly spaced samples, you are now in a position to perform a Fourier transform on the sampled time series.

Computing the Fourier transform

IIn the lesson titled Fun with Java, How and Why Spectral Analysis Works, I explained that you could compute the Fourier transform of an input time series at any given frequency F by evaluating the first two expressions in Figure 3.

```
Real(F) = S(n=0,N-
1)[x(n)*cos(2Pi*F*n)]
Imag(F) = S(n=0,N-
1)[x(n)*sin(2Pi*F*n)]

ComplexAmplitude(F) = Real(F) -
j*Imag(F)

Figure 3 Fourier transform equations
```

I also explained that the Fourier transform of the input time series at that frequency can be viewed as a complex number having a real part and an imaginary part as in the third expression in Figure 3.

The amplitude of the spectrum at that frequency can be determined by computing the square root of the sum of the squares of the real and imaginary parts at that frequency.

The Fourier transform of an input time series can be computed by performing these calculations across a range of frequencies.

The results of performing a spectral analysis on the pulse

The bottom four plots in Figure 2 show the results of performing a Fourier transform on the pulse in the top plot.

*(In this display format, which was produced by the program named **Graph06**, each sample value is represented by a vertical bar whose height is proportional to the value of the sample.)*

These four plots show the values of the Fourier transform output at a set of uniformly spaced frequencies ranging from zero to 0.25 times the sampling frequency.

The amplitude spectrum

The second plot from the top in Figure 2 shows the value of the amplitude spectrum. This is the Fourier transform output that we have been using in the previous lessons in this series.

(Those lessons ignored the complex spectrum and the phase angle.)

As you can see, the amplitude spectrum peaks at a frequency equal to 0.0625 times the sampling frequency. The reason for this will become clear when we examine the code that produced the pulse shown in the first plot.

The real and imaginary parts of the transform

The real part of the transform is shown in the third plot and the imaginary part of the transform is shown in the fourth plot. This is the first time that I have presented the real and imaginary parts of the spectrum in this series of lessons.

The phase angle in degrees

The phase angle in degrees is shown in the bottom plot. There are a variety of different ways to display phase angles. This program displays the phase angle as values that range from -180 degrees to +180 degrees.

(It is also possible to display the phase angle as ranging from 0 to 360 degrees, or any combination that equates to 360 degrees or one full rotation. It is also possible to display the phase angle in radians instead of degrees.)

How is the phase angle computed?

Basically, the phase angle is the angle that you get when you compute the arc tangent of the ratio of the imaginary part to the real part of the complex spectrum at a particular frequency. However, beyond computing the arc tangent, you must do some additional work to take the quadrant into account.

How should we interpret the phase angle?

To begin with, you should ignore the result of phase angle computations at those frequencies for which there is insignificant energy. It is always possible to form a ratio of the values of the real and imaginary parts of the complex Fourier transform at any frequency. However, if the real and

imaginary values produced by the Fourier transform at that frequency are both very small, the phase angle resulting from that ratio is of no practical significance. In fact, the angle can be corrupted by arithmetic errors resulting from performing arithmetic on very small values.

Therefore, noting the amplitude spectrum in the second plot of Figure 2, phase angle results to the right of the fourth tick mark are probably useless.

Many combinations

The phase angle produced by performing a Fourier transform on a pulse of a given waveform is not unique. There are an infinite number of combinations of real and imaginary parts that can result from performing a Fourier transform on a given waveform, depending on how you define the origin of time. This means that there are also an infinite number of phase angle curves that can be produced from the ratio of those real and imaginary parts. I will explain this in more detail later using simpler pulses.

The frequency band of primary interest

In the case shown in Figure 2, the frequency band of primary interest lies approximately between the first and the third tick marks. Most of the energy can be seen to lie between those limits on the basis of the amplitude plot.

The phase angle curve goes from a little more than zero degrees to a little less than 180 degrees across this frequency interval. However, it is significant to note that the phase angle is not linear across this frequency interval. Rather the shape of the curve is more like an elongated **S** sloping to the right.

A nonlinear phase angle, so what?

What is the significance of the nonlinear phase angle? If this plot represented the frequency response of your audio system, the existence of the nonlinear phase angle would be bad news. In particular, it would mean that the system would introduce phase distortion into your favorite music.

Computation of the phase angle

In an earlier lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm, I explained most of the code in the method named **transform** belonging to the class named **ForwardRealToComplex01**. However, I skipped over that portion of the code that computes the phase angle on the basis of the values of the real and imaginary parts. At this time, I am going to explain that code. For an explanation of the rest of the code in the **transform** m method, go back and review the lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.

The code in the **transform** method that computes the phase angle for a particular frequency is shown in Listing 1. At this point in the execution of the method, the values of the real part ***(real)***

and the imaginary part ***(imag)*** of the Fourier transform at a particular frequency have been computed. Those values are used to compute the phase angle at that frequency.

What if both values are zero?

The code begins by testing to see if both the real and imaginary parts are equal to zero. If so, attempting to form the ratio of the imaginary part to the real part would be meaningless. In this case, the code in Listing 1 simply sets the phase angle to a value of zero.

```
      if(imag == 0.0 && real == 0.0){ang = 0.0;}
      else{ang = Math.atan(imag/real)*180.0/pi;}

      if(real < 0.0 && imag == 0.0){ang = 180.0;}
      else if(real < 0.0 && imag == -0.0){
                                  ang = -180.0;}
      else if(real < 0.0 && imag > 0.0){
                                  ang += 180.0;}
      else if(real < 0.0 && imag < 0.0){
                                 ang += -180.0;}
      angleOut[i] = ang;
```

Listing 1

The atan method of the Math class

If both the real and imaginary parts are not zero, then the ratio of the imaginary value to the real value is formed and passed as a parameter to the **atan** method of the **Math** class.

The **atan** method returns the angle in radians whose tangent matches the value received as a parameter. The angle that is returned is in the range from -pi/2 to +pi/2 *(-90 degrees to +90 degrees).* The code in Listing 1 multiplies that angle in radians by 180.0/pi to convert the angle from radians to degrees.

Correction required for the quadrant

Although we have an intermediate answer at this point, we're still not finished. There is more work to do. The **atan** method simply uses the sign of its incoming parameter to decide whether to report the angle as positive or negative, and it only covers angles in two quadrants *(-90 degrees to + 90 degrees).* We know that the angle can actually be in any one of four quadrants *(-180 degrees to +180 degrees).*

The first and third quadrants

For example, a positive ratio can result from a positive imaginary value and a positive real value, or from a negative imaginary value and a negative real value. Both of these would be reported by the **atan** method as being between 0 and 90 degrees when in fact, the negative imaginary

value and the negative real value means that the angle is actually between -90 degrees and -180 degrees.

The second and fourth quadrants

Similarly, a negative ratio can result from a negative imaginary value and a positive real value or from a positive imaginary value and a negative real value. Both of these would be reported by the **atan** method as being between 0 and -90 degrees when in fact, the positive imaginary value and the negative real value means that the angle is actually between 90 degrees and 180 degrees.

An exercise for the reader

I will leave it as an exercise for the reader to work through the remaining code in Listing 1 to see how this code determines the proper quadrant and adjusts the angle appropriately, all the while maintaining the angle between -180 degrees and +180 degrees.

Will discuss Dsp034 in fragments

I will discuss the program named **Dsp034** in fragments. A complete listing of the program is provided in Listing 6 near the end of the lesson.

Due to the similarity of this program to programs explained in previous lessons in this series, this discussion will be rather brief. Following a discussion of the code, I will provide and explain some more spectral analysis results obtained by running the program with parameters read from the file named **Dsp034.txt**.

The beginning of the Dsp034 class

The beginning of the class, along with the declaration of several variables is shown in Listing 2.

*(Note that the class implements the interface named **GraphIntfc01**.)*

The variable names and the embedded comments should make the purpose of these variables self explanatory. If not, you will see how they are used later in the program.

```
class Dsp034 implements GraphIntfc01{
  final double pi = Math.PI;//for simplification

  int dataLen;;//data length
  int pulseLen;//length of the pulse
  int zeroTime;//sample that represents zero time
  //Low and high frequency limits for the
  // spectral analysis.
  double lowF;
  double highF;
  double[] pulse;//data describing the pulse
```

```
  //Following array stores input data for the
  // spectral analysis process.
  double[] data;

  //Following arrays receive information back
  // from the spectral analysis process
  double[] real;
  double[] imag;
  double[] angle;
  double[] mag;
```

Listing 2

The constructor

The constructor begins in Listing 3. The code in the constructor either invokes the **getParameters** method to get the operating parameters from the file named **Dsp034.txt**, or creates a set of operating parameters on the fly if that file does not exist in the current directory. In either case, many of the variables declared in Listing 2 are populated as a result of that action.

```
  public Dsp034(){//constructor

    //Get the parameters from a file named
    // Dsp034.txt.  Create and use default
    // parameters describing a damped sinusoidal
    // pulse if the file doesn't exist in the
    // current directory.
    if(new File("Dsp034.txt").exists()){
      getParameters();
    }else{
      //Create default parameters
      dataLen = 400;//data length
      pulseLen = 100;//pulse length
      //Sample that represents zero time.
      zeroTime = 0;
      //Low and high frequency limits for the
      // spectral analysis.
      lowF = 0.0;
      highF = 0.5;//half the sampling frequency
      pulse = new double[pulseLen];
      double scale = 240.0;
      for(int cnt = 0;cnt < pulseLen;cnt++){
        scale = 0.94*scale;//damping scale factor
        pulse[cnt] =
                  scale*Math.sin(2*pi*cnt*0.0625);
      }//end for loop
      //End default parameters
    }//end else
```

Listing 3

The damped sinusoid pulse

For the case where the file named **Dsp034.txt** doesn't exist in the current directory, the code in the **else** clause in Listing 3 establishes default operating parameters, and creates the damped sinusoid pulse shown in the top plot of Figure 2.

Store the pulse in the time series

At this point in the process, the array referred to by the reference variable named **pulse** contains a set of samples that constitutes a pulse. The code in Listing 4 creates a data array containing the data upon which spectral analysis will be performed and stores the pulse in that array.

> *(All elements in the data array other than those elements occupied by values of the pulse have a value of zero.)*

```
    data = new double[dataLen];
    for(int cnt = 0;cnt < pulse.length;cnt++){
      data[cnt] = pulse[cnt];
    }//end for loop
```

Listing 4

Print the parameters and the pulse

Following this, code in the constructor prints the parameters and the values of the samples that constitute the pulse. You can view that code in Listing 6 near the end of the lesson.

Perform the spectral analysis

Finally, the code in Listing 5 creates the array objects that will receive the results of the spectral analysis, and invokes the **transform** method of the **ForwardRealToComplex01** class to perform the spectral analysis.

```
    mag = new double[dataLen];
    real = new double[dataLen];
    imag = new double[dataLen];
    angle = new double[dataLen];
    ForwardRealToComplex01.transform(data,real,
      imag,angle,mag,zeroTime,lowF,highF);

  }//end constructor
```

Listing 5

Listing 5 also signals the end of the constructor. At this point, the object has been instantiated and it's array objects have been populated with the input and output data from the spectral

analysis process. This data is ready to be handed over to the plotting program to be plotted, as shown in Figure 2.

The getParameters method

This method, invoked in Listing 3, reads parameter and pulse data from the file named **Dsp034.txt**, and deposits that data into the variables and the **pulse** array declared in Listing 2.

The code in the **getParameters** method is straightforward, so I won't bore you with an explanation. You can view the method in Listing 6 near the end of the lesson.

The interface methods

As pointed out earlier, the **Dsp034** class implements the **GraphIntfc01** interface. As such, the class must define the six methods declared in that interface. These methods are invoked by the plotting program to obtain the data that is to be plotted.

You have seen implementations of these methods in several earlier lessons, so there is nothing new here. Consequently, I won't discuss the interface methods. You can view the methods in Listing 6 near the end of the lesson.

The one thing that you might want to pay attention to in these methods is the scaling that is applied to the data before it is returned. This is an attempt to cause all of the curves to plot reasonably well within a value range of -180 to +180. This range is dictated by the fact that this is the range of values for the phase angle data.

Now that you understand the inner workings of the program, let's look at some more examples, this time getting the input data from the file named **Dsp034.txt**.

The simplest pulse of all, an impulse

The simplest pulse that you can create is a single non-zero valued sample among a bunch of zero-valued samples. This simple pulse, commonly called an impulse in digital signal processing (DSP), is an extremely important type of signal. It is used for a variety of purposes in testing both digital and analog signal processing systems.

Let's examine the result of performing spectral analysis on an impulse.

The parameters used for this experiment are shown in Figure 4.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
```

```
Pulse Values
180.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```

Figure 4

The format of Figure 4

The data that you see in Figure 4 is the screen output from the method named **getParameters**. To orient you with this format, the second item in Figure 4 indicates that the entire length of the data upon which spectral analysis was performed was 400 samples. All 400 samples had a value of zero except for the values of the sample in the pulse that occurred at the beginning. In this program, setting the total data length to 400 causes the spectral analysis to be performed at 400 individual frequencies.

> *(By now, you may be suspecting that I have a particular affinity for a data length of 400 samples. If so, you are correct. This is not a technical affinity. Rather, it is a visual one. These figures are formatted such that the plotted data occupies an area of the screen containing approximately 400 pixels. By matching the plotted points to the positions of the pixels, it is possible to avoid, or at least minimize, the distortion that can occur when attempting to map from sample points to pixel locations when there is a mismatch between the two.*
>
> *To see the result of such mapping problems, repeat the experiment shown in Figure 2 and use your mouse to stretch the Frame horizontally by a very small amount. Depending on how much you stretch the Frame, you should see vertical lines disappear, vertical lines that are too close together, or a combination of the two. This is another manifestation of the impact of sampling that I don't have the time to get into at this point.)*

The length of the pulse

As you can see in Figure 4, the length of the pulse for this experiment was 11 samples, all but one of which had a value of zero.

> *(In this case, I could have made the pulse length 1 but for simplicity, I will keep it at 11 for several different experiments.)*

Defining the origin of time

As you may have discovered by playing video games, we can do things with a computer that we can't do in the real world. For example, the Fourier transform program allows me to specify which sample I regard as representing zero time. Samples to the left of that sample represent negative time *(history)* and samples to the right of that one represent positive time *(the future).*

In this case, I specified that the first sample *(sample number 0)* represents zero time. As you will see later, this has a significant impact on the distribution of energy between the real and imaginary parts of the transform results, and as such, has a significant impact on the phase angle.

Computational frequency range

Because I am using a DFT algorithm *(instead of an FFT algorithm)* I can compute the Fourier transform across a range of frequencies of my choosing. As shown in Figure 4, I chose to compute the transform across the range of frequencies from zero to one-half the sampling frequency, known as the Nyquist folding frequency. Thus, the spectral analysis was performed at 400 individual frequencies between these two limits.

The values that make up the pulse

This is followed in Figure 4 by the values of the samples that make up the pulse. Note that the first sample has a value of 180 while the other ten samples all have a value of 0.

The plotted results of spectral analysis

I used the program named **Graph03** to plot the output from this program, and the results are shown in Figure 5.

> *(Note that rather than plotting each sample value as a vertical bar, Graph03 plots each sample as a dot and connects the dots with straight line segments.)*

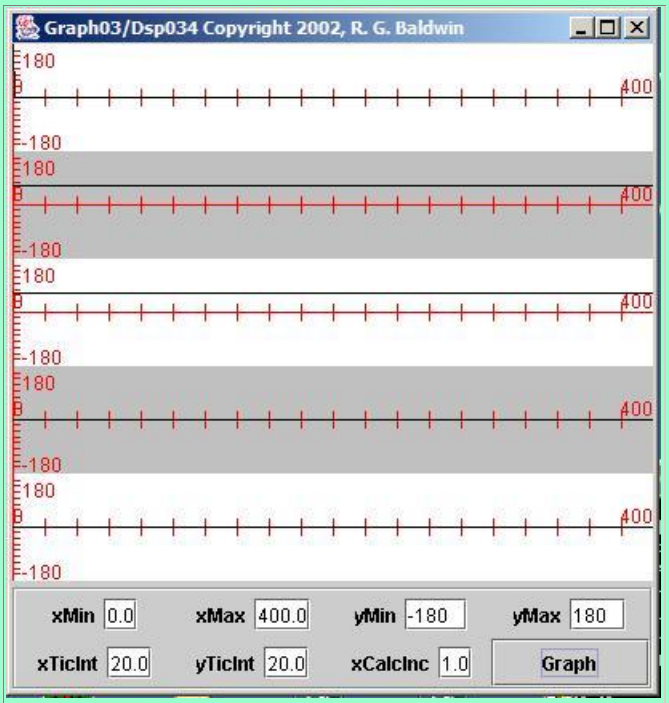


Figure 5 Spectral analysis of an impulse at zero time

Where is that impulse?

If you strain your eyes and you know exactly where to look, you can barely see a single black spike with an amplitude of 180 at the far left of the top plot. If you can't see it, you will simply have to trust me when I tell you that it is there.

The amplitude spectrum

Now take a look at the amplitude spectrum in the second plot from the top. What you should see is a straight black line extending from zero to the folding frequency on the right. This is because such an impulse *(theoretically)* contains an equal distribution of energy at every frequency from zero to infinity.

> *(In reality, there is no such thing as a perfect impulse, so there is no such thing as infinite bandwidth. However, the bandwidth of a practical impulse is very wide and the amplitude spectrum is very flat.)*

That is one of the things that make the impulse so useful as an input signal that is used for various testing purposes in both the analog world and the digital world.

The real spectrum

Now look at the real spectrum in the second plot from the top. As you can see, it looks exactly like the amplitude spectrum. This is because the impulse appears at zero time. We will change this in the next experiment so that you can see the impact of a time delay on the complex spectrum.

The imaginary spectrum

Moving on down the page, the imaginary part of the spectrum is a flat line with a value of zero across the entire frequency range. Once again, this is because the impulse appears at zero time.

The phase angle

Because the imaginary value is zero everywhere, the ratio of the imaginary value to the real value is also zero everywhere. Thus, the phase angle is also zero at all frequencies within the range.

Introduce a time delay

Now we are going to introduce a one-sample time delay in the location of the impulse relative to the time origin. We will keep the zero time reference at the first sample and cause the impulse to appear as the second sample in the eleven-sample sequence.

The new parameters are shown in Figure 6. The only change is the move of the impulse from the first sample in the eleven-sample pulse to the second sample in the eleven-sample pulse.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
0.0
180.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```

Figure 6

The spectral analysis output

The result of performing the spectral analysis on this new time series is shown in Figure 7.

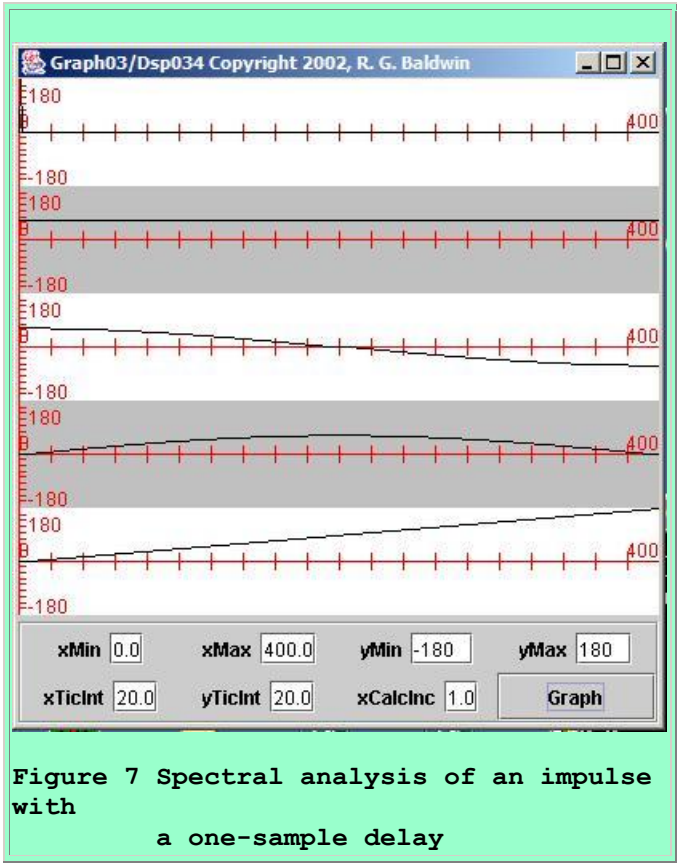


Figure 7 Spectral analysis of an impulse with a one-sample delay

Once again, if you strain your eyes, you can see the impulse on the left end of the top plot. It has been shifted one sample to the right relative to that shown in Figure 5.

The amplitude spectrum

The amplitude spectrum in the second plot looks exactly like it looked in Figure 5. That is as it should be. The spectral content of a pulse is determined by its waveform, not by its location in time. Simply moving the impulse by one sample into the future doesn't change its spectral content.

The real and imaginary parts of the spectrum

However, moving the impulse one sample into the future *(a time delay)* did change the values of the real and imaginary parts of the complex spectrum. As you can see from the third plot in Figure 7, the real part of the spectrum is no longer a replica of the amplitude spectrum. Also the imaginary part of the spectrum in the fourth plot is no longer zero across the frequency range from zero to the folding frequency.

Real and imaginary values are intrinsically linked

However, the real and imaginary parts cannot change in arbitrary ways relative to one another. Recall that the amplitude spectrum at each individual frequency is the square root of the

sum of the squares of the real and imaginary parts. In order for the amplitude spectrum to stay the same, changes to the real part of the spectrum must be accompanied by changes to the imaginary part that will maintain that relationship.

The phase angle spectrum

Finally, the phase angle shown in the bottom plot is no longer zero. Rather it is a straight line with a value of zero at zero frequency and a value of 180 degrees at the folding frequency.

An important conclusion

Shifting an impulse forward or backward in time introduces a phase shift that is linear with frequency. Shifting the pulse forward in time introduces a linear phase shift with a positive slope. Shifting the pulse backwards in time introduces a linear phase shift with a negative slope. In both cases, the amount of slope depends on the amount of time shift.

The converse is also true

A shift in time introduces a linear phase shift. Conversely, introducing a linear phase shift causes a shift in time.

A more acceptable form of phase distortion

Once again, consider your audio system. If your audio system introduces a phase shift across the frequency band of interest, you would probably like for that phase shift to be linear with frequency. That will simply cause the music to be delayed in time. In other words, all frequency components in the music will be delayed an equal amount of time.

On the other hand, if the phase shift is not linear with frequency, some frequencies will delayed more than other frequencies. This sometimes results in noticeable phase distortion in your music.

Introduce a large time delay

Now let's move the impulse to the center of the eleven-sample pulse and observe the result. The new parameters are shown in Figure 8.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
0.0
0.0
0.0
```

```
0.0
0.0
180.0
0.0
0.0
0.0
0.0
0.0

Figure 8
```

The output from the spectral analysis

Figure 9 shows the result of performing a spectral analysis on the time series containing this new time-delayed impulse.

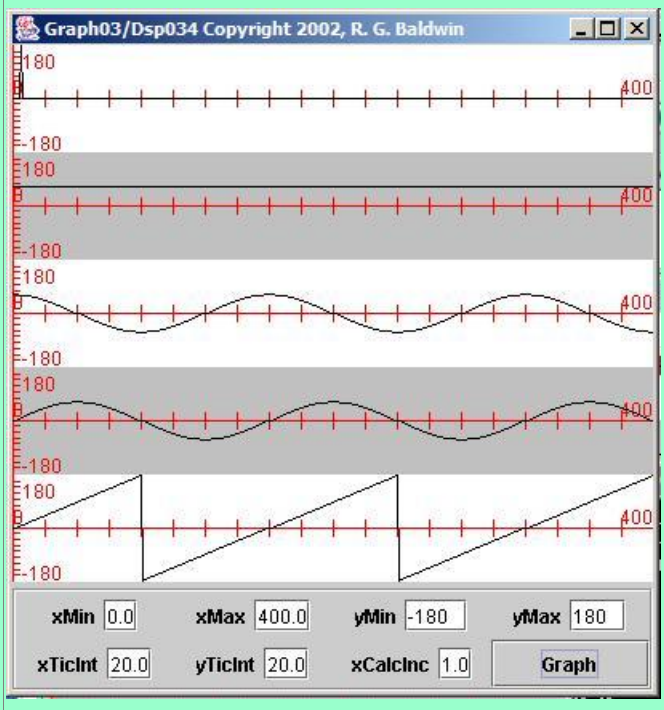


Figure 9 Spectral analysis of impulse with five-sample delay

Finally, you can see the impulse on the left side of the top plot in Figure 9 without straining your eyes too much.

As we would expect, the amplitude spectrum hasn't changed.

Although it wasn't apparent in Figure 7, Figure 9 shows that the real part of the spectrum takes on the shape of a cosine wave, while the imaginary part of the spectrum takes on the shape of a sine wave as a result of the time delay of the impulse.

The phase shift is still linear across frequency as would be expected, but the slope is now five times greater than the slope of the phase shift in Figure 7.

> *(Note that the time delay is five times greater in Figure 9. Note also that the plot of the phase angle wraps around from +180 degrees to -180 degrees each time the phase angle reaches +180 degrees. This produces the sawtooth effect shown in the bottom plot in Figure 9.)*

A boxcar pulse

If an impulse is the simplest kind of pulse to generate digitally, a *boxcar* pulse is probably the next simplest. A boxcar pulse is one where several adjacent samples have the same non-zero value. Let's examine the case for an eleven-sample boxcar pulse. The new parameters for this case are shown in Figure 10.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
```

Figure 10

The spectral analysis output for the boxcar pulse

The spectral analysis output for the eleven-sample boxcar pulse is shown in Figure 11.

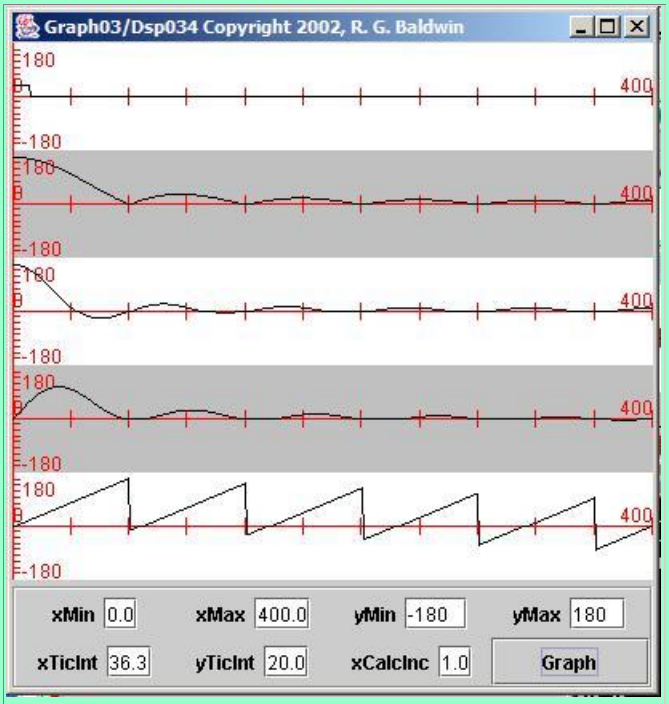


Figure 11 Spectral analysis of 11-sample boxcar pulse

The pulse itself is relatively easy to see on the leftmost end of the top plot.

The amplitude spectrum

The amplitude spectrum is no longer flat. Rather it has a peak at zero frequency and goes to zero between frequency sample 72 and frequency sample 73.

WiWithout getting into the technical details, I will simply tell you that the location of the point where it goes to zero is related to the reciprocal of the pulse width. If that sounds familiar, it is because we encountered similar situations involving bandwidth in the lesson titled Spectrum Analysis using Java, Frequency Resolution versus Data Length.

In fact, the shape of the amplitude spectrum is a familiar (sin x)/x curve with the negative lobes flipped up and turned into positive lobes instead.

The phase angle is still linear with frequency although it now shows some discontinuities at those frequencies where the amplitude spectrum touches zero.

The magic of digital

When working with digital time series, it is not only possible to physically shift pulses forward or backward in time, it is also possible to leave the pulses where they are and redefine the underlying time base. For the next experiment, I will leave everything else the same and

redefine the location of the origin of time. I will place the time origin at the middle of the boxcar pulse.

The new parameters for this experiment are shown in Figure 12. Note that the only significant difference between Figure 12 and Figure 10 is the redefinition of the sample that represents zero time. I redefined the time origin from sample 0 to sample 5. This causes the boxcar pulse to be centered on zero time. Five of the samples in the boxcar pulse occur in negative time. One sample occurs exactly at zero time. The other five samples occur in positive time.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 5
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
40.0
40.0
40.0
40.0
40.001
40.0
40.0
40.0
40.0
40.0
40.0
```

Figure 12

(I did make one other change. This change was to add a tiny spike to one of the samples near the center of the pulse. This creates a tiny amount of wide-band energy and tends to stabilize the computation of the phase angle. It prevents the imaginary part of the spectrum from switching back and forth between very small positive and negative values due to arithmetic errors.)

The spectral analysis output

The output from the spectral analysis is shown in Figure 13. The magnitude spectrum hasn't changed. The real part of the spectrum has changed significantly. It is now a true (sin x)/x curve with both positive and negative lobes.

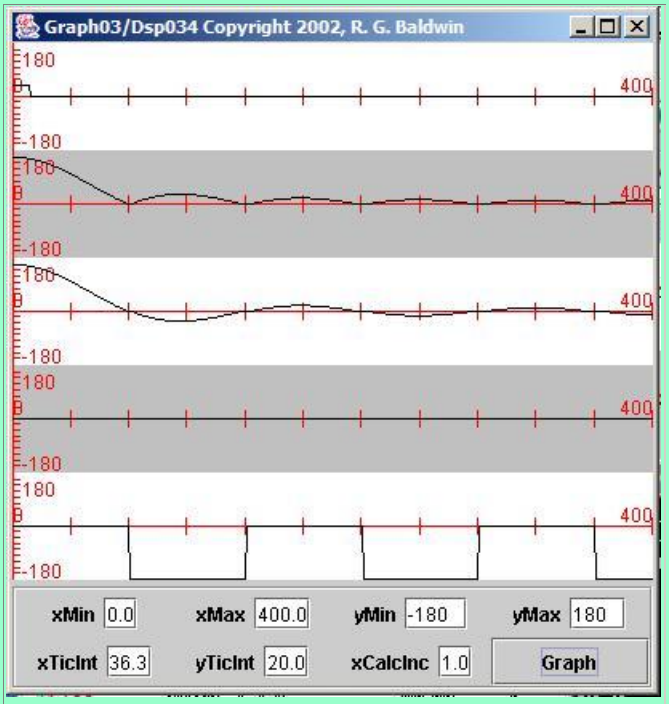


Figure 13 Spectral analysis of 11-sample boxcar centered in time

The imaginary part of the spectrum is zero or nearly zero at every frequency. *(It would be zero in the absence of arithmetic errors.)*

The phase angle is zero across the entire main energy lobe of the spectrum. It is -180 degrees in those frequency areas where the real part of the spectrum is negative, and is zero in those frequency areas where the real part of the spectrum is positive. There is no linear phase shift because the boxcar pulse is centered on the time origin.

Probably more than you ever wanted to know

And that is probably more than you ever wanted to know about the complex spectrum, phase angles, and time shifts. I will stop writing and leave it at that.

Run the Program

I encourage you to copy, compile, and run the program provided in this lesson. Experiment with it, making changes and observing the results of your changes.

Create more complex experiments. For example, you could create pulses of different lengths with complex shapes and examine the complex spectra and phase angles for those pulses.

If you really want to get fancy, you could create a pulse consisting of a sinusoid whose frequency changes with time from the beginning to the end of the pulse. *(A pulse of this type is often referred to as a frequency modulated sweep signal.)* See what you can conclude from doing spectral analysis on a pulse of this type. Pay particular attention to the phase angle across the frequency band containing most of the energy.

Most of all enjoy yourself and learn something in the process.

Summary

The default pulse for the **Dsp034** program is a damped sinusoid. This is a pulse whose shape is commonly found in mechanical and electronic systems in the real world. The phase angle in the complex spectrum for a pulse of this shape is nonlinear. Among other things, nonlinear phase angles introduce phase distortion into audio systems.

The simplest pulse of all is a single impulse. A pulse of this type has an infinite bandwidth *(theoretically)* and a linear phase angle. The slope of the phase angle depends on the location of the pulse relative to the time origin.

Shifting a pulse in time introduces a linear phase angle to the complex spectrum. Conversely, introducing a linear phase angle to the complex spectrum causes a pulse to be shifted in time.

What's Next?

The next lesson in this series will introduce the inverse Fourier transform and will explain the reversible nature of the Fourier transform.

Complete Program Listing

A complete listing of the program discussed in this lesson follows.

```
/* File Dsp034.java
Copyright 2004, R.G.Baldwin
Rev 5/21/04

Computes and plots the amplitude, real, imag,
and phase angle spectrum of a pulse that is read
from a file named Dsp034.txt.  If that file
doesn't exist in the current directory, the
program uses a set of default parameters
describing a damped sinusoidal pulse.

The program also plots the pulse.  When the data
is plotted using the programs Graph03 or Graph06,
the order of the plots from top to bottom in the
display are:
```

The pulse
The amplitude spectrum
The real spectrum
The imaginary spectrum
The phase angle in degrees

Each parameter value must be stored as characters
on a separate line in the file named Dsp034.txt.
The required parameters are as follows:

Data length as type int
Pulse length as type int
Sample number representing zero time as type int
Low frequency bound as type double
High frequency bound as type double
Sample values for the pulse as type double

The number of sample values must match the value
for the pulse length.

All frequency values are specified as a
double representing a fractional part of the
sampling frequency. For example, a value of 0.5
specifies a frequency that is half the sampling
frequency.

Here is a set of sample parameter values that can
be used to test the program. This sample data
describes a triangular pulse. Don't allow blank
lines at the end of the data in the file.

```
400
11
0
0.0
0.5
0
0
0
45
90
135
90
45
0
0
0
```

The plotting program that is used to plot the
output data from this program requires that the
program implement GraphIntfc01. For example,
the plotting program named Graph03 can be used
to plot the data produced by this program. When
it is used, the usage information is:

```
java Graph03 Dsp034

A static method named transform belonging to the
class named ForwardRealToComplex01 is used to
perform the actual spectral analysis.  The
method named transform does not implement an FFT
algorithm.  Rather, it is more general than, but
much slower than an FFT algorithm.  (See the
program named Dsp030 for the use of an FFT
algorithm.)

Tested using SDK 1.4.2 under WinXP.
**********************************************/
import java.util.*;
import java.io.*;

class Dsp034 implements GraphIntfc01{
  final double pi = Math.PI;//for simplification

  int dataLen;;//data length
  int pulseLen;//length of the pulse
  int zeroTime;//sample that represents zero time
  //Low and high frequency limits for the
  // spectral analysis.
  double lowF;
  double highF;
  double[] pulse;//data describing the pulse

  //Following array stores input data for the
  // spectral analysis process.
  double[] data;

  //Following arrays receive information back
  // from the spectral analysis process
  double[] real;
  double[] imag;
  double[] angle;
  double[] mag;

  public Dsp034(){//constructor

    //Get the parameters from a file named
    // Dsp034.txt.  Create and use default
    // parameters describing a damped sinusoidal
    // pulse if the file doesn't exist in the
    // current directory.
    if(new File("Dsp034.txt").exists()){
      getParameters();
    }else{
      //Create default parameters
      dataLen = 400;//data length
      pulseLen = 100;//pulse length
      //Sample that represents zero time.
      zeroTime = 0;
      //Low and high frequency limits for the
      // spectral analysis.
```

```
      lowF = 0.0;
      highF = 0.5;//half the sampling frequency
      pulse = new double[pulseLen];
      double scale = 240.0;
      for(int cnt = 0;cnt < pulseLen;cnt++){
        scale = 0.94*scale;//damping scale factor
        pulse[cnt] =
                  scale*Math.sin(2*pi*cnt*0.0625);
      }//end for loop
      //End default parameters
    }//end else

    //Create the data array and deposit the pulse
    // in it.
    data = new double[dataLen];
    for(int cnt = 0;cnt < pulse.length;cnt++){
      data[cnt] = pulse[cnt];
    }//end for loop

    //Print parameter values.
    System.out.println(
                      "Data length: " + dataLen);
    System.out.println(
                    "Pulse length: " + pulseLen);
    System.out.println(
            "Sample for zero time: " + zeroTime);
    System.out.println(
              "Lower frequency bound: " + lowF);
    System.out.println(
             "Upper frequency bound: " + highF);
    System.out.println("Pulse Values");
    for(int cnt = 0;cnt < pulseLen;cnt++){
      System.out.println(pulse[cnt]);
    }//end for loop

    //Create array objects to receive the results
    // and perform the spectral analysis.
    mag = new double[dataLen];
    real = new double[dataLen];
    imag = new double[dataLen];
    angle = new double[dataLen];
    ForwardRealToComplex01.transform(data,real,
      imag,angle,mag,zeroTime,lowF,highF);

  }//end constructor
  //-----------------------------------------//

  //This method gets processing parameters from
  // a file named Dsp034.txt and stores those
  // parameters in instance variables belonging
  // to the object of type Dsp034.
  void getParameters(){

    int cnt = 0;
    //Temporary holding area for strings.  Allow
    // space for a few blank lines at the end
```

```
    // of the data in the file.
    String[] data = new String[20];
    try{
      //Open an input stream.
      BufferedReader inData =
               new BufferedReader(new FileReader(
                                  "Dsp034.txt"));
      //Read and save the strings from each of
      // the lines in the file.  Be careful to
      // avoid having blank lines at the end,
      // which may cause an ArrayIndexOutOfBounds
      // exception to be thrown.
      while((data[cnt] =
                     inData.readLine()) != null){
        cnt++;
      }//end while
      inData.close();
    }catch(IOException e){}

    //Move the parameter values from the
    // temporary holding array into the instance
    // variables, converting from characters to
    // numeric values in the process.
    cnt = 0;
    dataLen =
            (int)Double.parseDouble(data[cnt++]);
    pulseLen =
            (int)Double.parseDouble(data[cnt++]);
    zeroTime =
            (int)Double.parseDouble(data[cnt++]);
    lowF = Double.parseDouble(data[cnt++]);
    highF = Double.parseDouble(data[cnt++]);

    //Create a new array object for the pulse
    // and populate it from the file file data.
    pulse = new double[pulseLen];
    for(int pCnt = 0;pCnt < pulseLen;pCnt++){
      pulse[pCnt] = Double.parseDouble(
                                    data[cnt++]);
    }//end for loop
    System.out.println(
                    "Parameters read from file");

  }//end getParameters
  //----------------------------------------//
  //The following six methods are required by the
  // interface named GraphIntfc01.  The plotting
  // program pulls the data values to be plotted
  // by invoking these methods.
  public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
  }//end getNmbr
  //----------------------------------------//
  //Provide the input data for plotting.
```

```
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data.length-1){
      return 0;
    }else{
      return data[index];
    }//end else
  }//end function
  //-----------------------------------------//
  //Provide the amplitude spectral data for
  // plotting.  Attempt to scale it so that it
  // will plot well in the range 0 to 180.
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag.length-1){
      return 0;
    }else{
      return (4*dataLen/pulseLen)*mag[index];
    }//end else
  }//end function
  //-----------------------------------------//
  //Provide the real spectral data for
  // plotting.  Attempt to scale it so that it
  // will plot well in the range -180 to 180.
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > real.length-1){
      return 0;
    }else{
      //Scale for convenient display
      return (4*dataLen/pulseLen)*real[index];
    }//end else
  }//end function
  //-----------------------------------------//
  //Provide the imaginary spectral data for
  // plotting.  Attempt to scale it so that it
  // will plot well in the range -180 to 180.
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imag.length-1){
      return 0;
    }else{
      //Scale for convenient display
      return (4*dataLen/pulseLen)*imag[index];
    }//end else
  }//end function
  //-----------------------------------------//
  //Provide the phase angle data for plotting.
  // The angle ranges from -180 degrees to +180
  // degrees.  This is thing that drives the
  // attempt to cause the other curves to plot
  // well in the range -180 to +180.
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > angle.length-1){
      return 0;
```

```
    }else{
      return angle[index];
    }//end else
  }//end function
  //-----------------------------------------//

}//end class Dsp034
```

Listing 6

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-