Spectrum Analysis using Java, Frequency Resolution versus Data Length

Baldwin provides the code and explains the requirements for using spectral analysis to resolve spectral peaks for pulses containing closely spaced truncated sinusoids.

Published: August 10, 2004

By Richard G. Baldwin

Java Programming, Notes # 1483

Preface

The how and the why of spectral analysis

A previous lesson titled Fun with Java, How and Why Spectral Analysis Works explained some of the fundamentals regarding spectral analysis. An understanding of that lesson is a prerequisite to an understanding of this lesson.

Another previous lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm presented and explained several Java programs for doing spectral analysis. In that lesson, I used a DFT program to illustrate several aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

I also used and briefly explained two different plotting programs that were originally explained in the earlier lesson titled Plotting Engineering and Scientific Data using Java.

An understanding of the lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm is also a prerequisite to an understanding of this lesson.

Frequency resolution versus data length

In this lesson I will use similar programs to explain and illustrate the manner in which spectral frequency resolution behaves with respect to data length.

A hypothetical situation

Consider a hypothetical situation in which you are performing spectral analysis on underwater acoustic signals in an attempt to identify enemy submarines.

You are aware that the enemy submarine contains a device that operates occasionally in short bursts. You are also aware that this device contains two rotating machines that rotate at almost but not quite the same speed.

During an operating burst of the device, each of the two machines contained in the device will emit acoustic energy that may appear as a peak in your spectral analysis output. *(Note that I said, "may appear" and did not say, "will appear.")* If you can identify the two peaks, you can conclusively identify the acoustic source as an enemy submarine.

The big question

How long must the operating bursts of this device be in order for you to resolve the peaks and identify the enemy submarine under ideal conditions? That is the question that I will attempt to answer in this lesson by teaching you about the relationship between frequency resolution and data length.

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

Before I get into the technical details, here is a preview of the programs and their purposes that I will present and explain in this lesson:

- **Dsp031** - Illustrates frequency resolution versus pulse length for pulses consisting of a truncated single sinusoid.
- **Dsp031a** - Displays the pulses analyzed by Dsp031.

- **Dsp032** - Illustrates frequency resolution versus pulse length for pulses consisting of the sum of two truncated sinusoids with closely spaced frequencies.
- **Dsp032a** - Displays the pulses analyzed by Dsp032.
- **Dsp033** - Illustrates frequency resolution versus pulse length for pulses consisting of the sum of two truncated sinusoids whose frequencies are barely resolvable.
- **Dsp033a** - Displays the pulses analyzed by Dsp033.

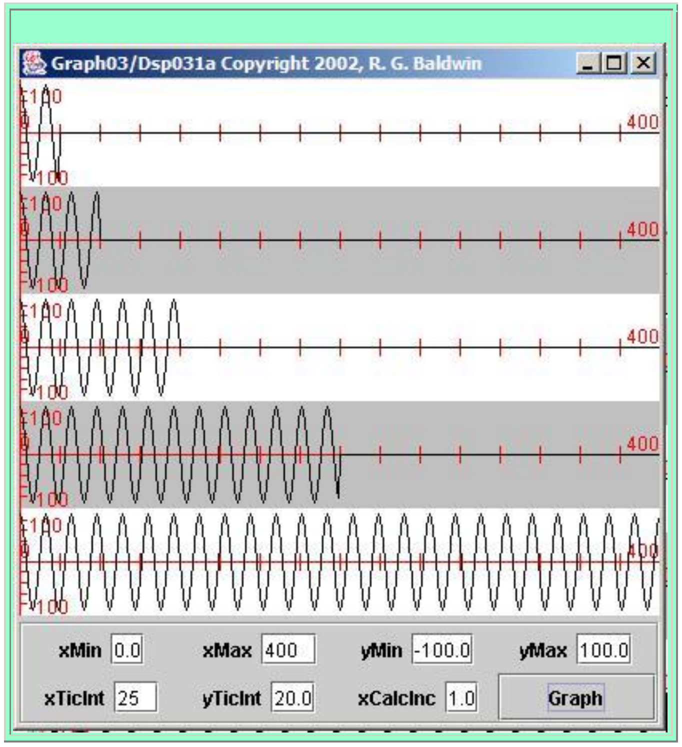
In addition, I will use the following programs that I explained in the lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.

- **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

Let's begin by looking at the time series data that will be used as input to the first spectral analysis experiment. Figure 1 shows five pulses in the time domain. Figure 2 and Figure 3 show the result of performing a spectral analysis on each of these pulses.

> *(The display in Figure 1 was produced by the program named **Dsp031a**, which I will explain later.)*



xMin 0.0 xMax 400 yMin -100.0 yMax 100.0
xTicInt 25 yTicInt 20.0 xCalcInc 1.0

```
Figure 1 Five pulses in the time domain.
```

The length of the pulses

If you examine Figure 1 carefully, you will see that each pulse is twice as long as the pulse above it. *(There is a tick mark on the horizontal axes every twenty-five samples.)* The bottom pulse is 400 samples long while the top pulse is 25 samples long.

Truncated sinusoids

Each pulse consists of a cosine wave that has been truncated at a different length. The frequency of the cosine wave is the same for every pulse. As you will see when we examine the code, the frequency of the cosine wave is 0.0625 times the sampling frequency. If you do the arithmetic, you will conclude that this results in 16 samples per cycle of the cosine wave.

In all five cases, the length of the time series upon which spectral analysis will be performed is 400 samples. For those four cases where the length of the pulse is less than 400 samples, the remaining samples in the time series have a value of zero.

Will compute at 400 frequencies

When the spectral analysis is performed later, the number of individual frequencies at which the amplitude of the spectral energy will be computed will be equal to the total data length. Therefore, the amplitude of the spectral energy will be computed at the same 400 frequencies for each of the five time series. That makes it convenient for us to stack the spectral plots up vertically and compare them *(as in Figure 2)*. This makes it easy for us to compare the distribution of energy across the frequency spectrum for pulses of different lengths.

Graph03 and Graph06

The plots in Figure 1 were produced using the program named **Graph03**. Other plots in this lesson will be produced using the program named **Graph06**. I explained those programs in earlier lessons, and I provided the source code for both programs in the previous lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm. Therefore, I won't repeat those explanations or provide the source code for those programs in this lesson.

The program named Dsp031a

A complete listing of the program named **Dsp031a** is provided in Listing 9 near the end of the lesson.

This program displays sinusoidal pulses identical to those processed by the program named **Dsp031**, which will be discussed later.

Time series containing sinusoidal pulses

The program named **Dsp031a** creates and displays five separate time series, each 400 samples in length. Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of a truncated sinusoid. The frequency of the sinusoid for each of the pulses is the same.

Frequency values are specified as type **double** as a fraction of the sampling frequency. The frequency of each sinusoid is 0.0625 times the sampling frequency.

The pulse lengths

The lengths of the five pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

Will discuss in fragments

This program is very similar to programs that I explained in previous lessons in this series, so my explanation will be very brief. As usual, I will discuss the program in fragments.

The beginning of the class, along with the declaration and initialization of several variables is shown in Listing 1. The names of the variables along with the embedded comments should make the code self explanatory.

```
class Dsp031a implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  int numberPulses = 5;
  //Frequency of the sinusoids
  double freq = 0.0625;
  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain sinusoidal data
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];
```

Listing 1

The constructor

Listing 2 shows the constructor, which creates the raw sinusoidal data and stores that data in the array objects created in Listing 1.

> *(Recall that all element values in the array objects are initialized with a value of zero. Therefore, the code in Listing 2 only needs to store the non-zero values in the array objects.)*

```
public Dsp031a(){//constructor

   //Create the raw data
   for(int x = 0;x < len/16;x++){
     data1[x] = amp*Math.cos(2*pi*x*freq);
   }//end for loop

   for(int x = 0;x < len/8;x++){
     data2[x] = amp*Math.cos(2*pi*x*freq);
   }//end for loop

   for(int x = 0;x < len/4;x++){
     data3[x] = amp*Math.cos(2*pi*x*freq);
   }//end for loop

   for(int x = 0;x < len/2;x++){
     data4[x] = amp*Math.cos(2*pi*x*freq);
   }//end for loop

   for(int x = 0;x < len;x++){
     data5[x] = amp*Math.cos(2*pi*x*freq);
   }//end for loop

 }//end constructor
```

Listing 2

The code in the conditional clause of each of the **for** loops in Listing 2 controls the length of each of the sinusoidal pulses.

The interface methods

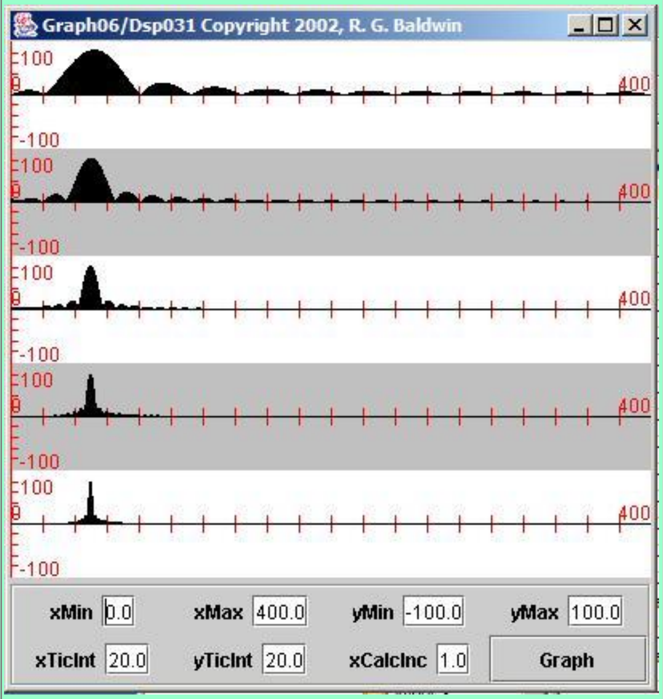
As you can see in Listing 1, the class implements the interface named **GraphIntfc01**. I introduced this interface in the earlier lesson titled Plotting Engineering and Scientific Data using Java and also discussed it in the previous lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.

The remaining code for the class named **Dsp031a** consists of the methods necessary to satisfy the interface. These methods are invoked by the plotting programs named **Graph03** and **Graph06** to obtain and plot the data returned by the methods. As implemented in **Dsp031a**, these interface methods return the values stored in the array objects referred to by **data1** through **data5**. Thus, the values stored in those array objects are plotted in Figure 1.

Spectral analysis results

Figure 2 shows the result of using the program named **Dsp031** to perform a spectral analysis on each of the five pulses shown in Figure 1. These results were plotted using the program named **Graph06**. With this plotting program, each data value is plotted as a vertical bar. However, in this case, the sides of each of the bars are so close together that the area under the spectral curve appears to be solid black.

> *(When you run this program, you can expand the display to full screen and see the individual vertical bars. However, I can't to that and maintain the narrow publication format required for this lesson.)*



xMin 0.0 xMax 400.0 yMin -100.0 yMax 100.0
xTicInt 20.0 yTicInt 20.0 xCalcInc 1.0

Figure 2 Spectral analyses of five pulses.

Interpretation of the results

Before I get into the interpretation, I need to point out that I normalized the data plotted in Figure 2 to cause each spectral peak to have approximately the same value. Otherwise, the spectral analysis result values for the short pulses would have been too small to be visible in this plotting format.

Therefore, the fact that the area under the curve in the top plot is greater than the area under the curve in the bottom plot doesn't indicate that the first pulse contains more energy than the last pulse. It simply means that I normalized the data for best results in plotting.

Spectrum of an ideal sinusoid

That having been said, different people will probably interpret these results in different ways. Let's begin by stating that the theoretical spectrum for a sinusoid of infinite length in the absence of noise is a single vertical line having zero width and infinite height.

In the real world of measurements, however, there is no such thing as a sinusoid of infinite length. Rather, every measurement that we make must truncate the sinusoid at some point in time. For a theoretical signal of infinite length, every spectral analysis that we can perform is an imperfect estimate of the spectrum.

Two viewpoints

There are at least two ways to think of the pulses shown in Figure 1.

1. Each pulse is a truncated section of an ideal sinusoid of infinite length.
2. Each pulse is a signal having a definite ***planned*** start and stop time.

The way that you interpret the results shown in Figure 2 depends on your viewpoint regarding the pulses.

The first viewpoint

If your viewpoint is that each pulse is a truncated section of an ideal sinusoid of infinite length, then the width of each of the peaks *(beyond zero width)* is the result of measurement error introduced by the truncation process.

The second viewpoint

If your viewpoint is that each pulse is a signal having a definite planned start and stop time, then the widths and the shape of each of the peaks describes the full range of frequency components required to physically generate such a pulse. This is the viewpoint that is consistent with the hypothetical situation involving a device on a submarine that I described earlier in this lesson.

A simplified hypothetical situation

Assume for the moment that the hypothetical device on the submarine contains only one rotating machine and that this device is turned on and off occasionally in short bursts. Because of the rotating machine, when the device is turned on, it will emit acoustic energy whose frequency matches the rotating speed of the machine.

(In reality, it will probably also emit acoustic energy at other frequencies as well, but we will consider it to be a very ideal machine. We will also assume the complete absence of any other acoustic noise in the environment.)

Assume that you have a recording window of 400 samples, and that you are able to record five such bursts within each of five separate recording windows. Further assume that the lengths of the individual bursts match the time periods indicated by the pulses in Figure 1.

The spectrum of the bursts

If you perform spectral analysis on each of the five individual 400-sample windows containing the bursts, and if you normalize the peak values for plotting purposes, you should get results similar to those shown in Figure 2.

The spectral bandwidth of the signal

The frequency range over which energy is distributed is referred to as the bandwidth of the signal. As you can see in Figure 2, shorter pulses require wider bandwidth.

For example, considerably more bandwidth is required of a communication system that is required to reliably transmit a series of short truncated sinusoids than one that is only required to reliably transmit a continuous tone at a single frequency.

At the same time, it is very difficult to convey very much information with a signal consisting of a continuous tone at a single frequency *(other than the fact that the tone exists).* Communication systems designed to convey information usually encode that information by either turning the tone on and off or by causing it to shift among a set of previously defined frequencies. The tone is often referred to as the carrier and the encoding of the information is often referred to as modulating the carrier.

Thus, you need greater bandwidth to reliably convey more information.

The relationship between pulse length and bandwidth

So far, we can draw one important conclusion from our experiment.

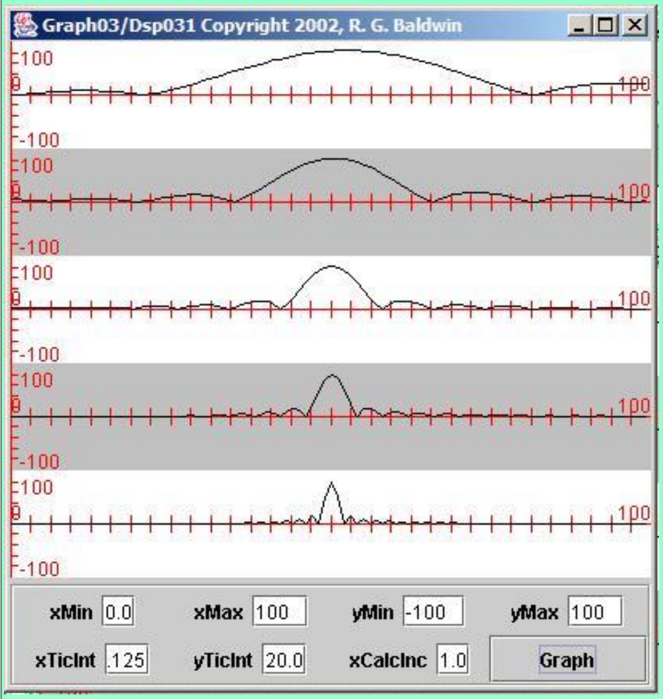
Shorter pulses require greater bandwidth.

This leads to an important question. What is the numerical relationship between pulse length and bandwidth? Although we can draw the above general conclusion from Figure 2, it is hard to draw any quantitative conclusions from Figure 2. That brings us to the expanded plot of the spectral data shown in Figure 3.

An expanded plot of the spectral results

Figure 3 shows the left one-fourth of the spectral results from Figure 2 plotted in the same horizontal space. In other words, Figure 3 discards the upper three-fourths of the spectral results from Figure 2 and shows only the lower one-fourth of the spectral results on an expanded scale. Figure 3 also provides tick marks that make it convenient to perform measurements on the plots.

Also, whereas Figure 2 was plotted using the program named **Graph06**, Figure 3 was plotted using the program named **Graph03**. Thus, Figure 3 uses a different plotting format than Figure 2



xMin 0.0 xMax 100 yMin -100 yMax 100
xTicInt .125 yTicInt 20.0 xCalcInc 1.0

Figure 3 Expanded spectral analyses of five pulses.

Picking numeric values

The curves in Figure 3 are spread out to the point that we can pick some approximate numeric values off the plot, and from this, we can draw a very significant conclusion.

For purposes of our approximation, consider the bandwidth to be the distance along the frequency axis between the points where the curves hit zero on either side of the peak. Using this approximation, the bandwidth indicated by the spectral analyses in Figure 3 shows the bandwidth of each spectrum to be twice as wide as the one below it.

Referring back to Figure 1, recall that the length of each pulse was half that of the one below it. The conclusion is:

The bandwidth of a truncated sinusoidal pulse is inversely proportional to the length of the pulse.

If you reduce the length of the pulse by a factor of two, you must double the bandwidth of a transmission system designed to reliably transmit a pulse of that length.

This will also be an important conclusion regarding our ability to separate and identify the two spectral peaks in the burst of acoustic energy described in our original hypothetical situation.

Let's see some code

The generation of the signals and the spectral analysis for the results presented in Figure 2 and Figure 3 were performed using the program named **Dsp031**. A complete listing of the program is shown in Listing 10 near the end of the lesson.

Description of the program named Dsp031

This program performs spectral analyses on five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths. *(The lengths of the individual pulses match that shown in Figure 1.)* Each pulse consists of a truncated sinusoid. The frequency of the sinusoid for each pulse is the same.

All frequency values are specified as type **double** as a fraction of the sampling frequency. The frequency of all five sinusoids is 0.0625 times the sampling frequency.

The lengths of the pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

If this sounds familiar, it is because the pulses are identical to those displayed in Figure 1 and discussed under **Dsp031a** above.

Uses a DFT algorithm

The spectral analysis process uses a DFT algorithm and computes the amplitude of the spectral energy at 400 equally spaced frequ4encies between zero and the folding frequency.

(Recall from the previous lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm that the folding frequency is one-half the sampling frequency.)

This program computes and displays the amplitude spectrum at frequency intervals that are one-half of the frequency intervals for a typical FFT algorithm.

Normalize the results

The results of the spectral analysis are multiplied by the reciprocal of the lengths of the individual pulses to normalize all five plots to the same peak value. Otherwise, the results for the short pulses would be too small to see on the plots.

Will discuss in fragments

Once again, this program is very similar to programs explained in the previous lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm. Therefore, this discussion will be very brief.

The code in Listing 3 declares and initializes some variables and creates the array objects that will contain the sinusoidal pulses.

In addition, the code in Listing 3 declares reference variables that will be used to refer to array objects containing results of the spectral analysis process that are not used in this program.

Finally, Listing 3 declares reference variables that will be used to refer to array objects containing the results plotted in Figure 2 and Figure 3.

Given the names of the variables, the comments, and what you learned in the previous lesson, the code in Listing 3 should be self explanatory.

```
class Dsp031 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  //Sample that represents zero time.
  int zeroTime = 0;
  //Low and high frequency limits for the
  // spectral analysis.
  double lowF = 0.0;
  double highF = 0.5;
  int numberSpectra = 5;
  //Frequency of the sinusoids
  double freq = 0.0625;
  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain data that is
  // input to the spectral analysis process.
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];

  //Following arrays receive information back
  // from the spectral analysis that is not used
  // in this program.
  double[] real;
  double[] imag;
  double[] angle;

  //Following arrays receive the magnitude
  // spectral information back from the spectral
  // analysis process.
  double[] mag1;
  double[] mag2;
  double[] mag3;
  double[] mag4;
  double[] mag5;
```

Listing 3

The constructor

The constructor begins in Listing 4. The code in Listing 4 is identical to that shown earlier in Listing 2. This code generates the five sinusoidal pulses and stores the data representing those pulses in the arrays referred to by **data1** through **data5**. So far, except for the declaration of some extra variables, this program isn't much different from the program named **Dsp031a** discussed earlier in this lesson.

```
  public Dsp031(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/8;x++){
      data2[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/4;x++){
      data3[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/2;x++){
      data4[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len;x++){
      data5[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop
```

Listing 4

Perform the spectral analysis

The remainder of the constructor is shown in Listing 5. This code invokes the **transform** method of the **ForwardRealToComplex01** class five times in succession to perform the spectral analysis on each of the five pulses shown in Figure 1.

> *(I explained the **transform** method in detail in the previous lesson titled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.)*

```
    mag1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1,real,
      imag,angle,mag1,zeroTime,lowF,highF);

    mag2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2,real,
      imag,angle,mag2,zeroTime,lowF,highF);

    mag3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3,real,
      imag,angle,mag3,zeroTime,lowF,highF);

    mag4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data4,real,
      imag,angle,mag4,zeroTime,lowF,highF);

    mag5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5,real,
      imag,angle,mag5,zeroTime,lowF,highF);

  }//end constructor
```

Listing 5

Each time the **transform** method is invoked, it computes the magnitude spectra for the incoming data and saves it in the output array.

*(Note that the **real**, **imag**, and **angle** arrays are not used later, so they are discarded each time a new spectral analysis is performed.)*

The interface methods

The **Dsp031** class also implements the interface named **GraphIntfc01**. The remaining code in the program consists of the methods required to satisfy that interface. Except for the identification of the arrays from which the methods extract data to be returned for plotting, these methods are identical to those defined in the earlier class named **Dsp031a**. Therefore, I won't discuss them further.

What we have learned so far

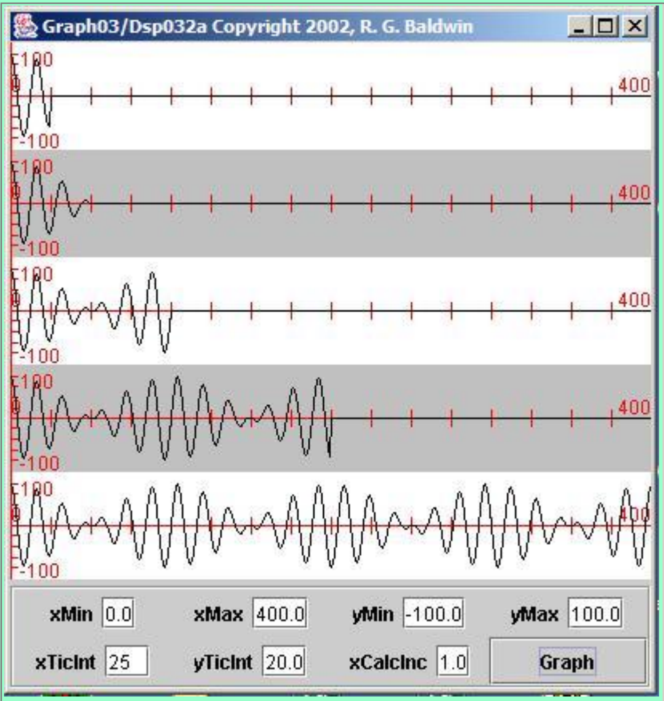
So far, the main things that we have learned is that shorter pulses require greater bandwidth, and the bandwidth required to faithfully represent a truncated sinusoidal pulse is the reciprocal of the length of the pulse.

Where do we go from here?

Now we will look at the issues involved in using spectral analysis to separate and identify the frequencies of two closely-spaced spectral peaks for a pulse composed of the sum of two sinusoids. Once again, we will begin by looking at some results and then discuss the code that produced those results.

The five pulses

The five pulses that we will be working with in this example are shown in Figure 4. As you can see, these pulses are a little more ugly than the pulses shown in Figure 1. As you can also see, as was the case in Figure 1, each pulse appears to be a shorter or longer version of the other pulses in terms of its waveform.



xMin 0.0 xMax 400.0 yMin -100.0 yMax 100.0
xTicInt 25 yTicInt 20.0 xCalcInc 1.0

Figure 4 Five pulses with two sinusoids each.

Produced by Dsp032a

The plots in Figure 4 were produced by the program named **Dsp032a**, which I will briefly discuss later. *(A complete listing of the program is shown in Listing 11 near the end of the lesson.)* This program creates and displays pulses identical to those processed by the program named **Dsp032**, which I will also briefly discuss later. *(A complete listing of the program named **Dsp32** is presented in Listing 12.)*

Five time series containing pulses

The program creates and displays five separate time series, each 400 samples in length. Each time series contains a pulse and the pulses are different lengths. As before, each of the pulses shown in Figure 4 is half the length below the pulse below it.

The sum of two sinusoids

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids for all pulses are the same.

All frequency values are specified as type **double** as a fraction of the sampling frequency. The frequencies of the two sinusoids are equidistant from a center frequency of 0.0625 times the sampling frequency.

(Recall that 0.0625 was the frequency of the only sinusoid contained in the pulses shown in Figure 1 and processed by the program named ***Dsp031****.)*

The frequencies and pulse lengths

The frequency of one sinusoid is (0.0625 - 2.0/len) times the sampling frequency, where **len** is the length of the time series containing the pulse. *(The value for* ***len*** *is 400 samples in this program.)* The frequency of the other sinusoid is (0.0625 + 2.0/len) times the sampling frequency.

The lengths of the five pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

(Note that Figure 4 has tick marks every 25 samples.)

The program named Dsp032a

The only new code in this program is the code in the constructor that creates the pulses and stores them in the data arrays. This code consists of five separate **for** loops, one for each pulse. The code for the first **for** loop, which is typical of the five, is shown in Listing 6.

```
  public Dsp032a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq1)
                     + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

//... code removed for brevity

  }//end constructor
```

Listing 6

As you can see from Listing 6, the values that make up the pulse are produced by adding together the values of two different cosine functions having different frequencies. The values for **freq1** and **freq2** are as described above.

You can view the remainder of this program in Listing 11.

Spectral analysis output

The results of running the program named **Dsp032** and displaying the results with the program named **Graph03** are shown in Figure 5.

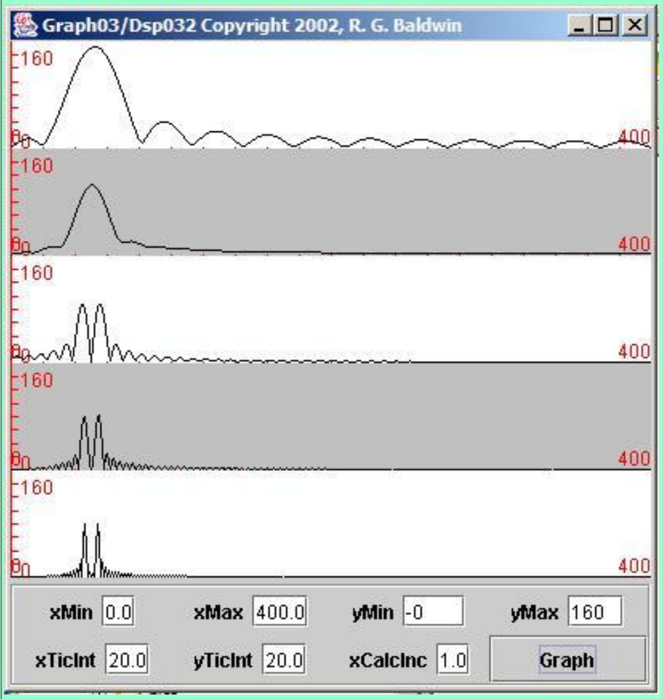


Figure 5 Spectral analyses of five pulses.

Each of the peaks in the third, fourth, and fifth plots in Figure 5 corresponds to the frequency of one of the two sinusoids that were added together to produce the pulses shown in Figure 4.

Can we answer the original question now?

The question posed in the original hypothetical situation was *"how long must the operating bursts of this device be in order for you to resolve the peaks and identify the enemy submarine under ideal conditions?"*

We are looking at very ideal conditions in Figure 4 and Figure 5. In particular, the pulses exist completely in the absence of noise.

> *(The existence of wide-band noise added to the pulses in Figure 4 would cause a change in the spectral results in Figure 5. That change might be described as having the appearance of grass and weeds growing on the baseline across the entire spectrum. The stronger the wide-band noise, the taller would be the weeds.)*

Cannot resolve two peaks for first two pulses

Clearly for the ideal condition of recording the bursts in the total absence of noise, you cannot resolve the peaks from the top two plots in Figure 5. For those two pulses, the spectral peaks simply merge together to form a single broad peak. Therefore, for this amount of separation between the frequencies of the two sinusoids, the lengths of the first two pulses in Figure 4 are insufficient to allow for separation and identification of the separate peaks.

We must be in luck

We seem to have the problem bracketed. *(Were we really lucky, or did I plan it this way?)* Under the ideal conditions of this experiment, the peaks are separable in the middle plot of Figure 5. Thus, for the amount of separation between the frequencies of the two sinusoids, the length of the third pulse is Figure 4 is sufficient to allow for separation and identification of the separate peaks.

A qualified answer to the question

The peaks are even better separated in the bottom two plots in Figure 5. For the five pulses used in this experiment and the amount of separation between the frequencies of the two sinusoids, any pulse as long or longer than the length of the third pulse is Figure 4 is sufficient to allow for separation and identification of the separate peaks.

What about the effects of noise?

If you were to add a nominal amount of wide-band noise to the mix, it would become more difficult to resolve the peaks for the bottom three plots in Figure 5 because the peaks would be growing out of a bed of weeds.

> *(If you add enough wide-band noise, you couldn't resolve the peaks using any of the plots, because the peaks would be completely "lost in the noise.")*

What can we learn from this?

Since we have concluded that the middle pulse in Figure 4 is sufficiently long to allow us to resolve the two peaks, let's see what we can learn from the parameters that describe that pulse.

Pulse length and frequency separation

To begin with, the length of the pulse is 100 samples.

What about the frequency separation of the two sinusoids? Recall that the frequency of one sinusoid is (0.0625 - 2.0/len) times the sampling frequency, where **len** is the length of the time series containing the pulse. The frequency of the other sinusoid is (0.0625 + 2.0/len) times the sampling frequency.

Thus, total separation between the two frequencies is 4/len, or 4/400. Dividing through by 4 we see that the separation between the two frequencies is 1/100.

Eureka, we have found it

For the third pulse, the frequency separation is the reciprocal of the length of the pulse. Also, the length of the third pulse is barely sufficient to allow for separation and identification of the two peaks in the spectrum. Thus, the two spectral peaks are separable in the absence of noise if the frequency separation is the reciprocal of the pulse length.

> *(That is too good to be a coincidence. I must have planned that way.)*

Thus, we have reached another conclusion.

> **Under ideal conditions, the two peaks in the spectrum can be resolved when the separation between the frequencies of the two sinusoids is equal to the reciprocal of the pulse length.**

The general answer

There is no single answer to the question *"how long must the operating bursts of this device be in order for you to resolve the peaks and identify the enemy submarine under ideal conditions?"*

The answer depends on the frequency separation. The general answer is that the length of the bursts must be at least as long as the reciprocal of the frequency separation for the two sinusoids. If the separation is large, the pulse length may be short. If the separation is small, the pulse length must be long.

The program named Dsp032

As I indicated earlier, the plots shown in Figure 5 were the result of running the program named **Dsp032** and displaying the data with the program named **Graph03**.

The only thing that is new in this program is the code that generates the five pulses and saves them in their respective data arrays. Even that code is not really new, because it is identical to the code shown in Listing 6. Therefore, I won't discuss this program further in this lesson.

One more experiment

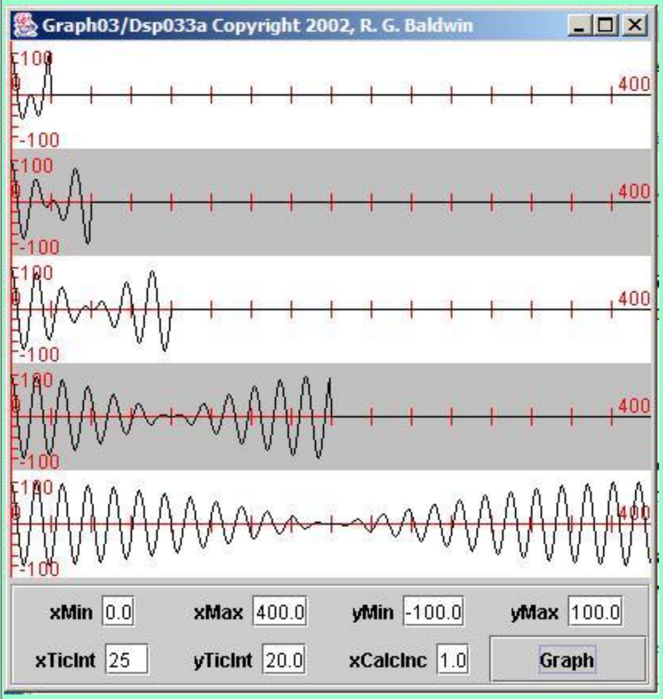
As you can surmise from the conclusions reached above, in order to be able to resolve the two peaks in the spectrum, you can either keep the pulse length the same and increase the frequency separation, or you can keep the frequency separation the same and increase the pulse length.

Let's examine an example where we keep the pulse lengths the same as before and adjust the frequency separation between the two sinusoids to make it barely possible to resolve the peaks for each of the five pulses.

We will need to increase the frequency separation for the first two pulses, and we can decrease the frequency separation for the fourth and fifth pulses. We will leave the frequency separation the same as before for the third pulse since it already seems to have the optimum relationship between pulse length and frequency separation.

The five pulses

The five pulses used in this experiment are shown in Figure 6.



xMin 0.0 xMax 400.0 yMin -100.0 yMax 100.0

xTicInt 25 yTicInt 20.0 xCalcInc 1.0

Figure 6 Five pulses with additive sinusoids.

Unlike in the previous two cases shown in Figure 1 and Figure 4, each of these pulses has a different shape from the others. In other words, in the previous two cases, each pulse simply looked like a longer or shorter version of the other pulses. That is not the case in this example.

> *(Note however that the third pulse in Figure 6 looks just like the third pulse in Figure 4. They were created using the same parameters. However, none of the other pulses in Figure 6 look like the corresponding pulses in Figure 4, and none of the pulses in Figure 6 look like the pulses in Figure 1.)*

Spectral analysis results

Figure 7 shows the result of performing spectral analysis on the five time series containing the pulses shown in Figure 6.

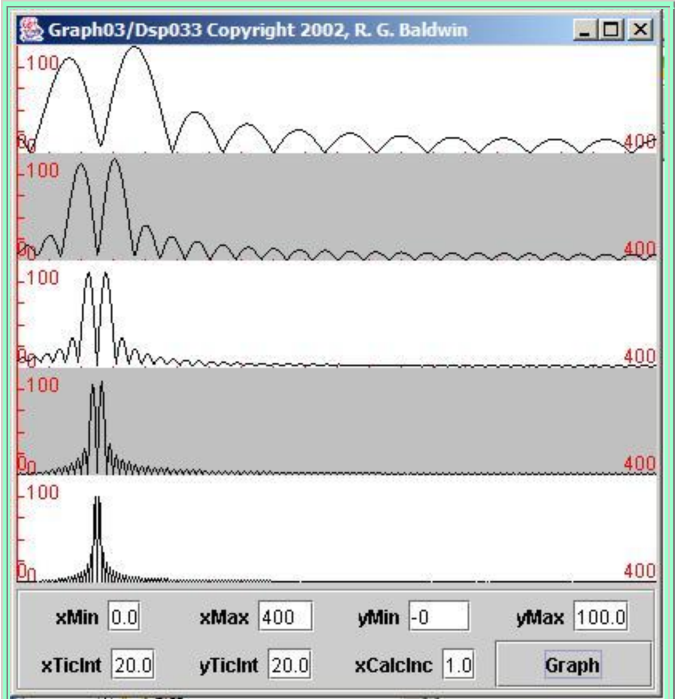


Figure 7 Spectral analyses of five pulses.

Peaks for first two pulses are now resolvable

When we examine the code, you will see that the frequency separation for the first two pulses has been increased to the reciprocal of the pulse length in each case. This results in the two peaks in the spectrum for each of the first two pulses being resolvable in Figure 7.

Third pulse hasn't changed

The spectrum for the third pulse shown in Figure 7 is almost identical to the spectrum for the third pulse shown in Figure 5. The only difference is that I had to decrease the vertical scaling on all of the plots in Figure 5 to keep the peak in the top plot within the bounds of the plot.

Spectral peaks for last two pulses are closer

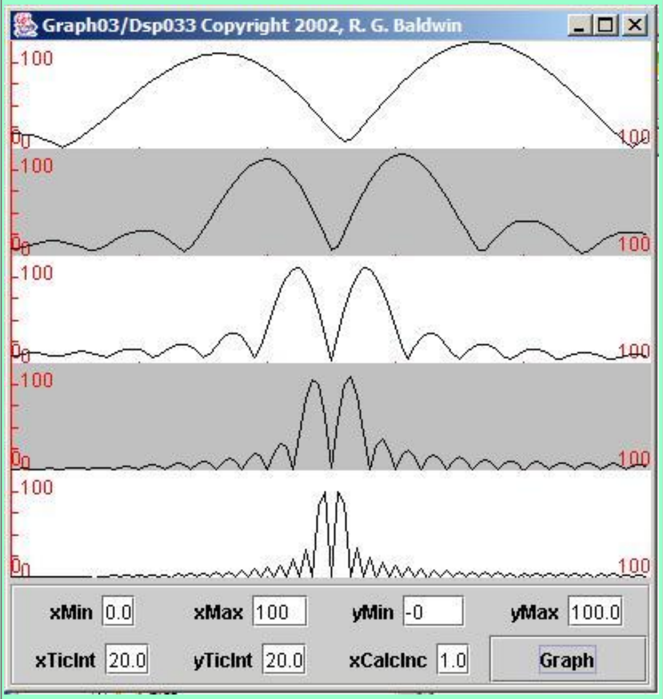
When we examine the code, you will also see that the frequency separation for the last two pulses has been decreased to the reciprocal of the pulse length in each case. This results in the two peaks in the spectrum for each of the last two pulses being closer than before in Figure 7.

The peaks in the bottom two plots in Figure 7 appear to be resolvable, but we can't be absolutely certain because they are so close together, particularly for the last plot.

> *(If you expand the Frame to full screen when you run this program, you will see that the two peaks are resolvable, but I can't do that and stay within this narrow publication format.)*

Expand the horizontal plotting scale

Figure 8 adjusts the plotting parameters to cause the left-most one-fourth of the data in Figure 7 to be plotted in the full width of the **Frame** in Figure 8.



xMin 0.0 xMax 100 yMin -0 yMax 100.0

xTicInt 20.0 yTicInt 20.0 xCalcInc 1.0

```
Figure 8 Expanded spectral analyses of
five pulses.
```

The peaks are barely resolvable

Figure 8 shows that the two peaks are barely resolvable for all five of the pulses shown in Figure 6.

> *(There is no space between the peaks at the baseline in Figure 8, but the plots do go almost down to the baseline half way between the two peaks.)*

The program named Dsp033

The plots in Figure 7 and Figure 8 were produced by running the program named **Dsp033** and plotting the results with the program named **Graph03**.

A complete listing of the program named **Dsp033** is shown in Listing 14 near the end of the lesson.

This program is the same as **Dsp032** except that the separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse in each case.

The program performs spectral analysis on five separate time series, each 400 samples in length. Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids are equidistant from a center frequency of 0.0625 times the sampling frequency. The total separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse.

All frequency values are specified as type **double** as a fraction of the sampling frequency.

The lengths of the pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

The spectral analysis

The spectral analysis computes the spectra at 400 equally spaced frequencies between zero and the folding frequency *(one-half the sampling frequency).*

The results of the spectral analysis are multiplied by the reciprocal of the lengths of the individual pulses to normalize the five plots. Otherwise, the results for the short pulses would be too small to see on the plots.

Because of the similarity of this program to the previous programs, my discussion of the code will be very brief.

Computation of the frequencies

The code in Listing 7 shows the computation of the frequencies of the sinusoids that will be added together to form each of the five pulses.

```
//Frequencies of the sinusoids
double freq1a = 0.0625 - 8.0/len;
double freq2a = 0.0625 + 8.0/len;

double freq1b = 0.0625 - 4.0/len;
double freq2b = 0.0625 + 4.0/len;

double freq1c = 0.0625 - 2.0/len;
double freq2c = 0.0625 + 2.0/len;

double freq1d = 0.0625 - 1.0/len;
double freq2d = 0.0625 + 1.0/len;
```

```
  double freq1e = 0.0625 - 0.5/len;
  double freq2e = 0.0625 + 0.5/len;
```

Listing 7

Create the pulses

The code in Listing 8 uses those frequency values to create the data for the pulses and to store that data in the arrays used to hold the pulses.

```
    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq1a)
                   + amp*Math.cos(2*pi*x*freq2a);
    }//end for loop

    for(int x = 0;x < len/8;x++){
      data2[x] = amp*Math.cos(2*pi*x*freq1b)
                   + amp*Math.cos(2*pi*x*freq2b);
    }//end for loop

    for(int x = 0;x < len/4;x++){
      data3[x] = amp*Math.cos(2*pi*x*freq1c)
                   + amp*Math.cos(2*pi*x*freq2c);
    }//end for loop

    for(int x = 0;x < len/2;x++){
      data4[x] = amp*Math.cos(2*pi*x*freq1d)
                   + amp*Math.cos(2*pi*x*freq2d);
    }//end for loop

    for(int x = 0;x < len;x++){
      data5[x] = amp*Math.cos(2*pi*x*freq1e)
                  + amp*Math.cos(2*pi*x*freq2e);
    }//end for loop
```

Listing 8

Other than the code shown in Listing 7 and Listing 8, the program named **Dsp033** is the same as the programs that were previously explained, and I won't discuss it further.

Run the Programs

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

Create more complex experiments. For example, you could create pulses containing three or more sinusoids at closely spaced frequencies, and you could cause the amplitudes of the sinusoids to be different. See what it takes to cause the peaks in the spectra of those pulses to be separable and identifiable.

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.

Try using the random number generator of the **Math** class to add some random noise to every value in the 400-sample time series. See what this does to your spectral analysis results.

Move the center frequency up and down the frequency axis. See if you can explain what happens as the center frequency approaches zero and as the center frequency approaches the folding frequency.

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

Summary

This program provides the code for three spectral analysis experiments of increasing complexity.

Bandwidth versus pulse length

The first experiment performs spectral analyses on five simple pulses consisting of truncated sinusoids. This experiment shows:

- Shorter pulses require greater bandwidth.
- The bandwidth of a truncated sinusoidal pulse is inversely proportional to the length of the pulse.

Peak resolution versus pulse length and frequency separation

The second experiment performs spectral analyses on five more complex pulses consisting of the sum of two truncated sinusoids having closely spaced frequencies. The purpose is to determine the required length of the pulse in order to use spectral analysis to resolve spectral peaks attributable to the two sinusoids. The experiment shows that the peaks are barely resolvable when the length of the pulse is the reciprocal of the frequency separation between the two sinusoids.

Five pulses with barely resolvable spectral peaks

The third experiment also performs spectral analyses on five pulses consisting of the sum of two truncated sinusoids having closely spaced frequencies. In this case, the frequency separation for each pulse is the reciprocal of the length of the pulse. The results of the spectral analysis reinforce the conclusions drawn in the second experiment.

What's Next?

So far, the lessons in this series have ignored the complex nature of the results of spectral analysis. The complex results have been converted into real results by computing the square root of the sum of the squares of the real and imaginary parts.

The next lesson in the series will meet the issue of complex spectral results head on and will explain the concept of phase angle. In addition, the lesson will explain the behavior of the phase angle with respect to time shifts in the input time series.

Complete Program Listings

Complete listings of all the programs discussed in this lesson are provided in this section.

```
/* File Dsp031a.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004

Displays sinusoidal pulses identical to those
processed by Dsp031.

Creates and displays five separate time series,
each 400 samples in length.

Each time series contains a pulse and the pulses
are different lengths.

Each pulse consists of a truncated sinusoid.  The
frequency of the sinusoid for all pulses is the
same.

All frequency values are specified as type
double as a fraction of the sampling frequency.

The frequency of all sinusoids is 0.0625 times
the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

Tested using J2SEE 1.4.2 under WinXP.
************************************************/
import java.util.*;

class Dsp031a implements GraphIntfc01{
  final double pi = Math.PI;
```

```
int len = 400;//data length
int numberPulses = 5;
//Frequency of the sinusoids
double freq = 0.0625;
//Amplitude of the sinusoids
double amp = 160;

//Following arrays will contain sinusoidal data
double[] data1 = new double[len];
double[] data2 = new double[len];
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];

public Dsp031a(){//constructor

  //Create the raw data
  for(int x = 0;x < len/16;x++){
    data1[x] = amp*Math.cos(2*pi*x*freq);
  }//end for loop

  for(int x = 0;x < len/8;x++){
    data2[x] = amp*Math.cos(2*pi*x*freq);
  }//end for loop

  for(int x = 0;x < len/4;x++){
    data3[x] = amp*Math.cos(2*pi*x*freq);
  }//end for loop

  for(int x = 0;x < len/2;x++){
    data4[x] = amp*Math.cos(2*pi*x*freq);
  }//end for loop

  for(int x = 0;x < len;x++){
    data5[x] = amp*Math.cos(2*pi*x*freq);
  }//end for loop

}//end constructor

//--------------------------------------------//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
  //Return number of functions to process.
  // Must not exceed 5.
  return 5;
}//end getNmbr
//--------------------------------------------//
public double f1(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > data1.length-1){
    return 0;
  }else{
    //Scale the amplitude of the pulses to make
    // them compatible with the default
```

```
      // plotting amplitude of 100.0.
      return data1[index]*90.0/amp;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data2.length-1){
      return 0;
    }else{
      return data2[index]*90.0/amp;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data3.length-1){
      return 0;
    }else{
      return data3[index]*90.0/amp;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data4.length-1){
      return 0;
    }else{
      return data4[index]*90.0/amp;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data5.length-1){
      return 0;
    }else{
      return data5[index]*90.0/amp;
    }//end else
  }//end function

}//end sample class Dsp031a
```

Listing 9

```
/* File Dsp031.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004

Performs spectral analysis on five separate time
series, each 400 samples in length.
```

```
Each time series contains a pulse and the pulses
are different lengths.

Each pulse consists of a truncated sinusoid.  The
frequency of the sinusoid for all pulses is the
same.

All frequency values are specified as type
double as a fraction of the sampling frequency.

The frequency of all sinusoids is 0.0625 times
the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

The spectral analyis computes the spectra at
400 equally spaced points between zero and the
folding frequency (one-half the sampling
frequency).

The results of the spectral analysis are
multiplied by the reciprocal of the lengths of
the individual pulses to normalize all five
plots to the same peak value.  Otherwise, the
results for the short pulses would be too
small to see on the plots.

Tested using J2SEE 1.4.2 under WinXP.
************************************************/
import java.util.*;

class Dsp031 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  //Sample that represents zero time.
  int zeroTime = 0;
  //Low and high frequency limits for the
  // spectral analysis.
  double lowF = 0.0;
  double highF = 0.5;
  int numberSpectra = 5;
  //Frequency of the sinusoids
  double freq = 0.0625;
  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain data that is
  // input to the spectral analysis process.
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];

  //Following arrays receive information back
  // from the spectral analysis that is not used
  // in this program.
  double[] real;
  double[] imag;
  double[] angle;

  //Following arrays receive the magnitude
  // spectral information back from the spectral
  // analysis process.
  double[] mag1;
  double[] mag2;
  double[] mag3;
  double[] mag4;
  double[] mag5;

  public Dsp031(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/8;x++){
      data2[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/4;x++){
      data3[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/2;x++){
      data4[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len;x++){
      data5[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop


    //Compute magnitude spectra of the raw data
    // and save it in output arrays.  Note that
    // the real, imag, and angle arrays are not
    // used later, so they are discarded each
    // time a new spectral analysis is performed.
    mag1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1,real,
      imag,angle,mag1,zeroTime,lowF,highF);
    mag2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2,real,
      imag,angle,mag2,zeroTime,lowF,highF);

    mag3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3,real,
      imag,angle,mag3,zeroTime,lowF,highF);

    mag4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data4,real,
      imag,angle,mag4,zeroTime,lowF,highF);

    mag5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5,real,
      imag,angle,mag5,zeroTime,lowF,highF);

  }//end constructor

  //-------------------------------------------//
  //The following six methods are required by the
  // interface named GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
  }//end getNmbr
  //-------------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag1.length-1){
      return 0;
    }else{
      //Scale the magnitude data by the
      // reciprocal of the length of the sinusoid
      // to normalize the five plots to the same
      // peak value.
      return mag1[index]*16.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag2.length-1){
      return 0;
    }else{
      return mag2[index]*8.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag3.length-1){
      return 0;
    }else{
      return mag3[index]*4.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag4.length-1){
      return 0;
    }else{
      return mag4[index]*2.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag5.length-1){
      return 0;
    }else{
      return mag5[index]*1.0;
    }//end else
  }//end function

}//end sample class Dsp031
```

Listing 10

```
/* File Dsp032a.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004

Displays sinusoidal pulses identical to those
processed by Dsp032.

Creates and displays five separate time series,
each 400 samples in length.

Each time series contains a pulse and the pulses
are different lengths.

Each pulse consists of the sum of two sinusoids
at closely spaced frequencies.  The frequencies
```

```
of the two sinusoids for all pulses are the same.

All frequency values are specified as type
double as a fraction of the sampling frequency.

The frequencies of the two sinusoids are
equidistant from 0.0625 times the sampling
frequency.

The frequency of one sinusoid is
(0.0625 - 2.0/len) times the sampling frequency.

The frequency of the other sinusoid is
(0.0625 + 2.0/len) times the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

Tested using J2SEE 1.4.2 under WinXP.
************************************************/
import java.util.*;

class Dsp032a implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  int numberPulses = 5;
  //Frequencies of the sinusoids
  double freq1 = 0.0625 - 2.0/len;
  double freq2 = 0.0625 + 2.0/len;

  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain sinusoidal data
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];

  public Dsp032a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq1)
                   + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/8;x++){
      data2[x] = amp*Math.cos(2*pi*x*freq1)
```

```
                    + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len/4;x++){
    data3[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len/2;x++){
    data4[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len;x++){
    data5[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

}//end constructor

//-----------------------------------------------//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
  //Return number of functions to process.
  // Must not exceed 5.
  return 5;
}//end getNmbr
//-----------------------------------------------//
public double f1(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > data1.length-1){
    return 0;
  }else{
    //Scale the amplitude of the pulses to make
    // them compatible with the default
    // plotting amplitude of 100.0.
    return data1[index]*40.0/amp;
  }//end else
}//end function
//-----------------------------------------------//
public double f2(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > data2.length-1){
    return 0;
  }else{
    return data2[index]*40.0/amp;
  }//end else
}//end function
//-----------------------------------------------//
public double f3(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > data3.length-1){
    return 0;
  }else{
    return data3[index]*40.0/amp;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data4.length-1){
      return 0;
    }else{
      return data4[index]*40.0/amp;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data5.length-1){
      return 0;
    }else{
      return data5[index]*40.0/amp;
    }//end else
  }//end function

}//end sample class Dsp032a
```

Listing 11

```
/* File Dsp032.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004

Performs spectral analysis on five separate time
series, each 400 samples in length.

Each time series contains a pulse and the pulses
are different lengths.

Each pulse consists of the sum of two sinusoids
at closely spaced frequencies.  The frequencies
of the two sinusoids for all pulses are the same.

All frequency values are specified as type
double as a fraction of the sampling frequency.

The frequencies of the two sinusoids are
equidistant from 0.0625 times the sampling
frequency.

The frequency of one sinusoid is
(0.0625 - 2.0/len) times the sampling frequency.

The frequency of the other sinusoid is
(0.0625 + 2.0/len) times the sampling frequency.
```

```
The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

The spectral analyis computes the spectra at
400 equally spaced points between zero and the
folding frequency (one-half the sampling
frequency).

The results of the spectral analysis are
multiplied by the reciprocal of the lengths of
the individual pulses to normalize the five
plots.  Otherwise, the results for the short
pulses would be too small to see on the plots.

Tested using J2SEE 1.4.2 under WinXP.
************************************************/
import java.util.*;

class Dsp032 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  //Sample that represents zero time.
  int zeroTime = 0;
  //Low and high frequency limits for the
  // spectral analysis.
  double lowF = 0.0;
  double highF = 0.5;
  int numberSpectra = 5;
  //Frequencies of the sinusoids
  double freq1 = 0.0625 - 2.0/len;
  double freq2 = 0.0625 + 2.0/len;

  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain data that is
  // input to the spectral analysis process.
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];

  //Following arrays receive information back
  // from the spectral analysis that is not used
  // in this program.
  double[] real;
  double[] imag;
  double[] angle;
```

```
//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] mag1;
double[] mag2;
double[] mag3;
double[] mag4;
double[] mag5;

public Dsp032(){//constructor

  //Create the raw data
  for(int x = 0;x < len/16;x++){
    data1[x] = amp*Math.cos(2*pi*x*freq1)
                  + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len/8;x++){
    data2[x] = amp*Math.cos(2*pi*x*freq1)
                  + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len/4;x++){
    data3[x] = amp*Math.cos(2*pi*x*freq1)
                  + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len/2;x++){
    data4[x] = amp*Math.cos(2*pi*x*freq1)
                  + amp*Math.cos(2*pi*x*freq2);
  }//end for loop

  for(int x = 0;x < len;x++){
    data5[x] = amp*Math.cos(2*pi*x*freq1)
                  + amp*Math.cos(2*pi*x*freq2);
  }//end for loop


  //Compute magnitude spectra of the raw data
  // and save it in output arrays.  Note that
  // the real, imag, and angle arrays are not
  // used later, so they are discarded each
  // time a new spectral analysis is performed.
  mag1 = new double[len];
  real = new double[len];
  imag = new double[len];
  angle = new double[len];
  ForwardRealToComplex01.transform(data1,real,
    imag,angle,mag1,zeroTime,lowF,highF);

  mag2 = new double[len];
  real = new double[len];
  imag = new double[len];
  angle = new double[len];
  ForwardRealToComplex01.transform(data2,real,
    imag,angle,mag2,zeroTime,lowF,highF);
  mag3 = new double[len];
  real = new double[len];
  imag = new double[len];
  angle = new double[len];
  ForwardRealToComplex01.transform(data3,real,
    imag,angle,mag3,zeroTime,lowF,highF);

  mag4 = new double[len];
  real = new double[len];
  imag = new double[len];
  angle = new double[len];
  ForwardRealToComplex01.transform(data4,real,
    imag,angle,mag4,zeroTime,lowF,highF);

  mag5 = new double[len];
  real = new double[len];
  imag = new double[len];
  angle = new double[len];
  ForwardRealToComplex01.transform(data5,real,
    imag,angle,mag5,zeroTime,lowF,highF);

}//end constructor

//-------------------------------------------//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
  //Return number of functions to process.
  // Must not exceed 5.
  return 5;
}//end getNmbr
//-------------------------------------------//
public double f1(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > mag1.length-1){
    return 0;
  }else{
    //Scale the magnitude data by the
    // reciprocal of the length of the sinusoid
    // to normalize the five plots to the same
    // peak value.
    return mag1[index]*16.0;
  }//end else
}//end function
//-------------------------------------------//
public double f2(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > mag2.length-1){
    return 0;
  }else{
    return mag2[index]*8.0;
  }//end else
}//end function
//-------------------------------------------//
public double f3(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > mag3.length-1){
    return 0;
  }else{
    return mag3[index]*4.0;
  }//end else
}//end function
//-------------------------------------------//
public double f4(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > mag4.length-1){
    return 0;
  }else{
    return mag4[index]*2.0;
  }//end else
}//end function
//-------------------------------------------//
public double f5(double x){
  int index = (int)Math.round(x);
  if(index < 0 || index > mag5.length-1){
    return 0;
  }else{
    return mag5[index]*1.0;
  }//end else
}//end function

}//end sample class Dsp032
```

Listing 12

```
/* File Dsp033a.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004

Displays sinusoidal pulses identical to those
processed by Dsp033.

Creates and displays five separate time series,
each 400 samples in length.

Each time series contains a pulse and the pulses
are different lengths.

Each pulse consists of the sum of two sinusoids
at closely spaced frequencies.  The frequencies
of the two sinusoids are equidistant from 0.0625
times the sampling frequency.  The total
separation between the frequencies of the two
sinusoids is the reciprocal of the length of the
pulse.

All frequency values are specified as type
```

```
double as a fraction of the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

Tested using J2SEE 1.4.2 under WinXP.
************************************************/
import java.util.*;

class Dsp033a implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  int numberPulses = 5;
  //Frequencies of the sinusoids
  double freq1a = 0.0625 - 8.0/len;
  double freq2a = 0.0625 + 8.0/len;

  double freq1b = 0.0625 - 4.0/len;
  double freq2b = 0.0625 + 4.0/len;

  double freq1c = 0.0625 - 2.0/len;
  double freq2c = 0.0625 + 2.0/len;

  double freq1d = 0.0625 - 1.0/len;
  double freq2d = 0.0625 + 1.0/len;

  double freq1e = 0.0625 - 0.5/len;
  double freq2e = 0.0625 + 0.5/len;

  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain sinusoidal data
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];

  public Dsp033a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq1a)
                    + amp*Math.cos(2*pi*x*freq2a);
    }//end for loop

    for(int x = 0;x < len/8;x++){
      data2[x] = amp*Math.cos(2*pi*x*freq1b)
                    + amp*Math.cos(2*pi*x*freq2b);
    }//end for loop

    for(int x = 0;x < len/4;x++){
      data3[x] = amp*Math.cos(2*pi*x*freq1c)
                   + amp*Math.cos(2*pi*x*freq2c);
    }//end for loop

    for(int x = 0;x < len/2;x++){
      data4[x] = amp*Math.cos(2*pi*x*freq1d)
                   + amp*Math.cos(2*pi*x*freq2d);
    }//end for loop

    for(int x = 0;x < len;x++){
      data5[x] = amp*Math.cos(2*pi*x*freq1e)
                  + amp*Math.cos(2*pi*x*freq2e);
    }//end for loop

  }//end constructor

  //-----------------------------------------//
  //The following six methods are required by the
  // interface named GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
  }//end getNmbr
  //-----------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data1.length-1){
      return 0;
    }else{
      //Scale the amplitude of the pulses to make
      // them compatible with the default
      // plotting amplitude of 100.0.
      return data1[index]*40.0/amp;
    }//end else
  }//end function
  //-----------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data2.length-1){
      return 0;
    }else{
      return data2[index]*40.0/amp;
    }//end else
  }//end function
  //-----------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data3.length-1){
      return 0;
    }else{
      return data3[index]*40.0/amp;
    }//end else
  }//end function
  //-----------------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data4.length-1){
      return 0;
    }else{
      return data4[index]*40.0/amp;
    }//end else
  }//end function
  //-----------------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data5.length-1){
      return 0;
    }else{
      return data5[index]*40.0/amp;
    }//end else
  }//end function

}//end sample class Dsp033a
```

Listing 13

```
/* File Dsp033.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004

Same as Dsp032 except that the separation between
the frequencies of the two sinusoids is the
reciprocal of the length of the pulse.

Performs spectral analysis on five separate time
series, each 400 samples in length.

Each time series contains a pulse and the pulses
are different lengths.

Each pulse consists of the sum of two sinusoids
at closely spaced frequencies.  The frequencies
of the two sinusoids are equidistant from 0.0625
times the sampling frequency.  The total
separation between the frequencies of the two
sinusoids is the reciprocal of the length of the
pulse.

All frequency values are specified as type
double as a fraction of the sampling frequency.

The lengths of the pulses are:

25 samples
```

```
50 samples
100 samples
200 samples
400 samples

The spectral analyis computes the spectra at
400 equally spaced points between zero and the
folding frequency (one-half the sampling
frequency).

The results of the spectral analysis are
multiplied by the reciprocal of the lengths of
the individual pulses to normalize the five
plots.  Otherwise, the results for the short
pulses would be too small to see on the plots.

Tested using J2SEE 1.4.2 under WinXP.
************************************************/
import java.util.*;

class Dsp033 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 400;//data length
  //Sample that represents zero time.
  int zeroTime = 0;
  //Low and high frequency limits for the
  // spectral analysis.
  double lowF = 0.0;
  double highF = 0.5;
  int numberSpectra = 5;
  //Frequencies of the sinusoids
  double freq1a = 0.0625 - 8.0/len;
  double freq2a = 0.0625 + 8.0/len;

  double freq1b = 0.0625 - 4.0/len;
  double freq2b = 0.0625 + 4.0/len;

  double freq1c = 0.0625 - 2.0/len;
  double freq2c = 0.0625 + 2.0/len;

  double freq1d = 0.0625 - 1.0/len;
  double freq2d = 0.0625 + 1.0/len;

  double freq1e = 0.0625 - 0.5/len;
  double freq2e = 0.0625 + 0.5/len;


  //Amplitude of the sinusoids
  double amp = 160;

  //Following arrays will contain data that is
  // input to the spectral analysis process.
  double[] data1 = new double[len];
  double[] data2 = new double[len];
  double[] data3 = new double[len];
  double[] data4 = new double[len];
  double[] data5 = new double[len];

  //Following arrays receive information back
  // from the spectral analysis that is not used
  // in this program.
  double[] real;
  double[] imag;
  double[] angle;

  //Following arrays receive the magnitude
  // spectral information back from the spectral
  // analysis process.
  double[] mag1;
  double[] mag2;
  double[] mag3;
  double[] mag4;
  double[] mag5;

  public Dsp033(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
      data1[x] = amp*Math.cos(2*pi*x*freq1a)
                   + amp*Math.cos(2*pi*x*freq2a);
    }//end for loop

    for(int x = 0;x < len/8;x++){
      data2[x] = amp*Math.cos(2*pi*x*freq1b)
                   + amp*Math.cos(2*pi*x*freq2b);
    }//end for loop

    for(int x = 0;x < len/4;x++){
      data3[x] = amp*Math.cos(2*pi*x*freq1c)
                   + amp*Math.cos(2*pi*x*freq2c);
    }//end for loop

    for(int x = 0;x < len/2;x++){
      data4[x] = amp*Math.cos(2*pi*x*freq1d)
                   + amp*Math.cos(2*pi*x*freq2d);
    }//end for loop

    for(int x = 0;x < len;x++){
      data5[x] = amp*Math.cos(2*pi*x*freq1e)
                  + amp*Math.cos(2*pi*x*freq2e);
    }//end for loop


    //Compute magnitude spectra of the raw data
    // and save it in output arrays.  Note that
    // the real, imag, and angle arrays are not
    // used later, so they are discarded each
    // time a new spectral analysis is performed.
    mag1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1,real,
      imag,angle,mag1,zeroTime,lowF,highF);

    mag2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2,real,
      imag,angle,mag2,zeroTime,lowF,highF);

    mag3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3,real,
      imag,angle,mag3,zeroTime,lowF,highF);

    mag4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data4,real,
      imag,angle,mag4,zeroTime,lowF,highF);

    mag5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5,real,
      imag,angle,mag5,zeroTime,lowF,highF);

  }//end constructor

  //-------------------------------------------//
  //The following six methods are required by the
  // interface named GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
  }//end getNmbr
  //-------------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag1.length-1){
      return 0;
    }else{
      //Scale the magnitude data by the
      // reciprocal of the length of the sinusoid
      // to normalize the five plots to the same
      // peak value.
      return mag1[index]*16.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag2.length-1){
      return 0;
    }else{
      return mag2[index]*8.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag3.length-1){
      return 0;
    }else{
      return mag3[index]*4.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag4.length-1){
      return 0;
    }else{
      return mag4[index]*2.0;
    }//end else
  }//end function
  //-------------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag5.length-1){
      return 0;
    }else{
      return mag5[index]*1.0;
    }//end else
  }//end function

}//end sample class Dsp033
```

Listing 14

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-