Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm

Baldwin explains several different programs used for spectral analysis. He also explains the impact of the sampling frequency and the Nyquist folding frequency on spectral analysis.

Published: July 6, 2004
By Richard G. Baldwin

Java Programming, Notes # 1482

Preface

The how and the why of spectral analysis

A previous lesson entitled Fun with Java, How and Why Spectral Analysis Works explained how and why spectral analysis works. An understanding of that lesson is a prerequisite to understanding this lesson.

Programs to perform spectral analysis

In this lesson I will provide and explain different programs used for performing spectral analysis. The first program is a very general program that implements a Discrete Fourier Transform *(DFT)* algorithm. I will explain this program in detail.

The second program is a less general, but much faster program that implements a Fast Fourier Transform *(FFT)* algorithm. I will defer an explanation of this program until a future lesson. I am providing it here so that you can use it and compare it with the DFT program in terms of speed and flexibility.

Fundamental aspects of spectral analysis

I will use the DFT program to illustrate several fundamental aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

Visual illustration of sampling

I will also provide and explain a program that produces a visual illustration of the impact of the sampling frequency and the Nyquist folding frequency.

Plotting programs

Finally, I will provide, but will not explain two different programs used for display purposes. These are newer versions of graphics display programs that I explained in the earlier lesson entitled Plotting Engineering and Scientific Data using Java.

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:

- **Dsp029** - Provides a visual illustration of the impact of the sampling frequency and the Nyquist folding frequency.
- **Dsp028** - Driver program for doing spectral analysis using a DFT algorithm.
- **ForwardRealToComplex01** - Class that implements the DFT algorithm.
- **Dsp030** - Driver program for doing spectral analysis using an FFT algorithm.
- **ForwardRealToComplexFFT01** - Class that implements the FFT algorithm *(will defer explanation until a future lesson).*
- **Graph03** - Used to display results of spectral analysis. *(The concepts were explained in an earlier lesson.)*
- **Graph06** - Used to display the impact of sampling frequency and the Nyquist folding frequency. Also used to display the results of spectral analysis. *(The concepts were explained in an earlier lesson).*

Discussion and Sample Code

This will be a long lesson involving lots of code and lots of explanations, so fill your cup with java and let's get started.

Sampling Frequency and the Nyquist Folding Frequency

I will begin the discussion with the program named **Dsp029**, which provides a visual illustration of the impact of the sampling frequency and the Nyquist folding frequency. A complete listing of this program is shown in Listing 16 near the end of the lesson.

Display sinusoids

This program generates and displays up to five sinusoids having the same sampling frequency but having different sinusoidal frequencies and amplitudes. The program provides a visual illustration of the way in which frequencies above one-half the sampling frequency fold back into the area bounded by zero and one-half the sampling frequency.

> *(The frequency at one-half the sampling frequency is known as the Nyquist folding frequency.)*

Input parameters

The program gets its input parameters from a file named **Dsp029.txt**. If that file doesn't exist in the current directory, the program uses a set of default parameters.

Each parameter value must be stored as characters on a separate line in the file named **Dsp029.txt**. The required parameters are shown in Figure 1.

```
Data length as type int
Number of sinusoids as type int.  Max
value is 5.
List of sinusoid frequency values as
type double.
List of sinusoid amplitude values as
type double.
```

Figure 1

The length of the two lists

The number of values in each of the lists must match the value for the number of sinusoids. Also, you must not allow blank lines at the end of the data in the file.

Frequency value specifications

Each frequency value is specified as a type **double** value representing a fractional part of the sampling frequency.

> *(For example, a **double** value of 0.5 specifies one-half the sampling frequency, or the Nyquist folding frequency. A **double** value of 2.0 specifies a frequency that is twice the sampling frequency.)*

Figure 2 shows the contents of the file named **Dsp029.txt** that I used to produce the output shown in Figure 3. I will discuss that output later.

```
50.0
5
0.03125
0.0625
0.125
0.25
0.5
90
90
90
90
90
```

Figure 2

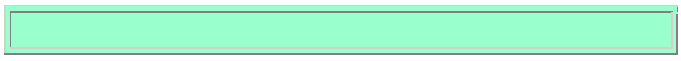
The plotting program named Graph06

The plotting program that is used to plot the output data from this program requires that the program implement **GraphIntfc01**. I discussed that interface in the earlier lesson entitled Plotting Engineering and Scientific Data using Java. For example, the plotting program named **Graph06** can be used to plot the data produced by this program. When it is used, the program is executed and its output is plotted by entering the following at the command line prompt:

```
java Graph06 Dsp029
```

Program output

Figure 3 shows the output produced by running the program named **Dsp029** with the parameters shown in Figure 2.



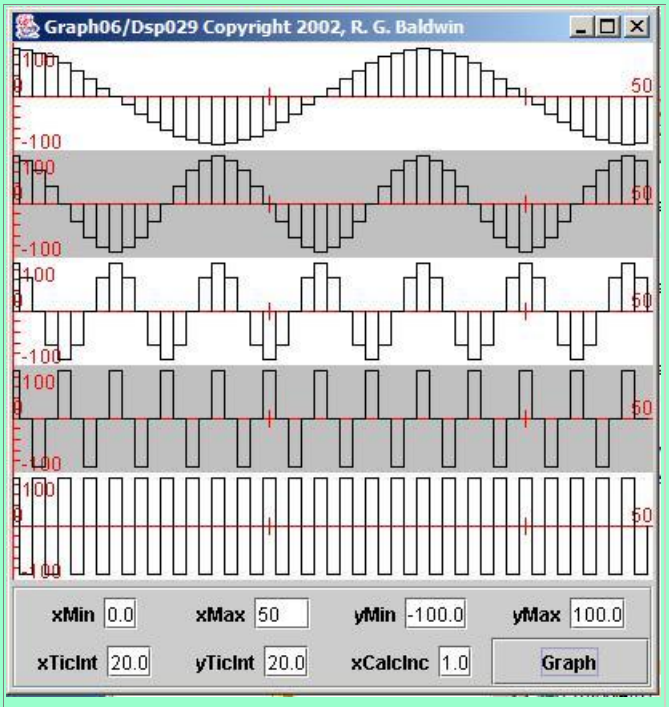


Figure 3 Program output for five sinusoids

Five horizontal plots

Each of the five horizontal plots in Figure 3 shows a sampled sinusoid. Each of the vertical bars represents one sample value for a given sinusoid.

If you examine the frequency values in Figure 2 carefully, you will see that they represent the sampling frequency divided by the factors 32, 16, 8, 4, and 2. Thus, the last frequency value is the Nyquist folding frequency and the first four frequency values are related to that frequency by multiples of two.

The horizontal plot at the top of Figure 3 is a reasonably well defined cosine wave. The horizontal plot at the bottom of Figure 3 shows the result of having exactly two samples per cycle of the sinusoid. Using this plotting scheme, the sampled sinusoid is represented as a square wave. Using another plotting scheme *(such as that used in the program named **Graph03**)* the sampled sinusoid at the bottom would be represented as a triangular wave.

An upper frequency limit

Regardless of the plotting scheme used, it should be obvious that a set of uniform samples cannot possibly represent frequencies higher than one-half the sampling frequency because then there would be less than one sample per cycle of the sinusoid.

The Nyquist folding frequency

Now I will show you why the frequency at one-half the sampling frequency is referred to as the *folding* frequency using the new set of frequency values shown in Figure 4.

> *(Figure 4 shows the values read from the file named **Dsp029.txt** and displayed in an improved format.)*

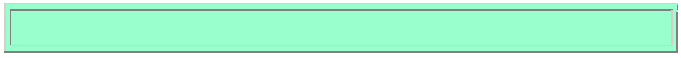
In this case, the third frequency in the list is one-half the sampling frequency, which is the folding frequency. The two frequencies on either side of that one have values that are symmetrical about the folding frequency.

```
Data length: 50
Number sinusoids: 5
Frequencies
0.125
0.25
0.5
0.75
0.875
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

Figure 4

The program output

The five horizontal plots in Figure 5 show the result of running the program named **Dsp029** with the frequencies shown in Figure 4.



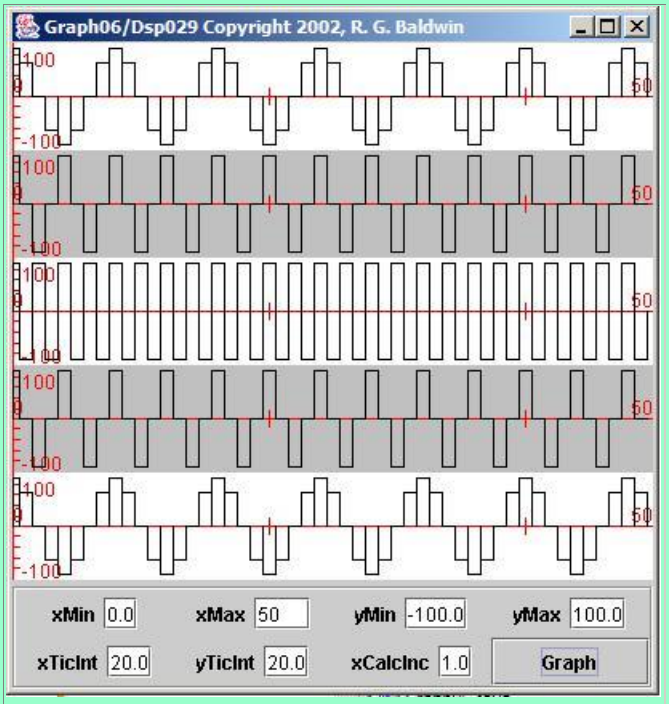


Figure 5 Program output for five sinusoids

The center plot in Figure 5 is the sampled representation of the folding frequency.

The top two plots in Figure 5 are obviously the sampled representations of the two lower frequencies specified by the first two frequencies in Figure 4.

Not so obvious ...

However, it is not so obvious that the bottom two plots in Figure 5 are the sampled representations of the two higher frequencies specified by the last two frequencies in Figure 4. They look exactly like the top two plots but in reverse order.

Unable to distinguish ...

Frequencies above one-half the sampling frequency are not distinguishable by viewing the sampled data. In fact, they are converted to lower frequencies by sampling process. The new lower frequencies fold around a point in the frequency spectrum given by one-half the sampling frequency. That is why it is called the folding frequency. *(In 1933, this frequency was named after scientist Harry Nyquist.)*

One more example

Let's look at one more example of plotted sinusoids. Consider the frequency values shown in Figure 6. The second and third frequencies are symmetrical about the sampling frequency. The fourth and fifth frequencies are symmetrical about twice the sampling frequency. The first

frequency value is the same distance from zero as the other four frequencies are from the sampling frequency and twice the sampling frequency.

```
Data length: 50
Number sinusoids: 5
Frequencies
0.1
0.9
1.1
1.9
2.1
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

Figure 6

The program output

Figure 7 shows the output produced by running the program named **Dsp029** with the frequency parameters specified by Figure 6.

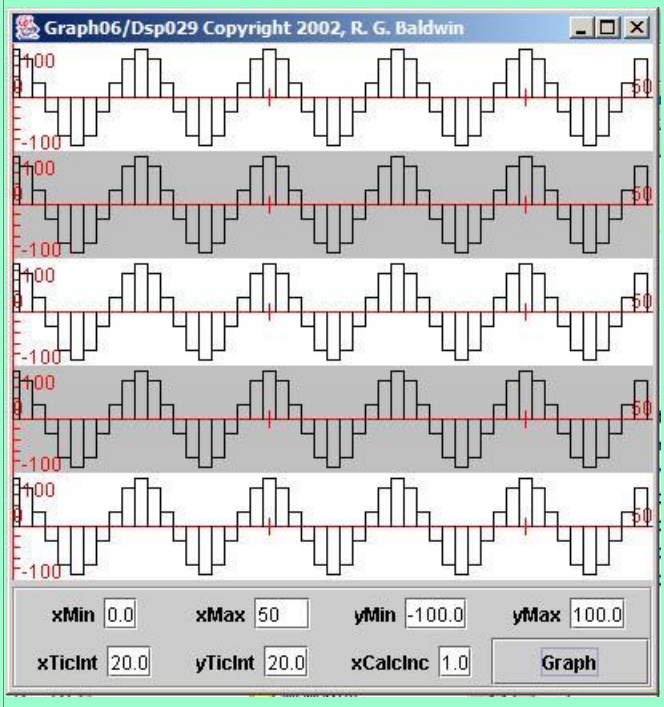


Figure 7 Program output for five sinusoids

The sinusoids are indistinguishable

Although the actual frequencies of the five cosine functions are significantly different, once they are sampled, they are indistinguishable. The sampling process converts the actual frequencies to new frequencies that not only fold around one-half the sampling frequency, they also fold around all multiples of one-half the sampling frequency.

Enough talk, let's see some code

The program named **Dsp029** is provided in Listing 16 near the end of the lesson. I will discuss the program in fragments, beginning with the fragment shown in Listing 1.

```
class Dsp029 implements GraphIntfc01{
  final double pi = Math.PI;//for
simplification

  //Begin default parameters
  int len = 400;//data length
  int numberSinusoids = 5;
  //Frequencies of the sinusoids
  double[] freq =
{0.1,0.25,0.5,0.75,0.9};
  //Amplitudes of the sinusoids
  double[] amp = {75,75,75,75,75};
  //End default parameters

Listing 1
```

The code in Listing 1 defines a convenience constant representing the value of pi and then defines the set of default parameters that will be used by the program in the event that the file named **Dsp029.txt** does not exist in the current directory.

Create array objects to hold sinusoidal data

The code in Listing 2 creates five array objects that will be populated with 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 2
```

Get the parameters

The constructor begins in Listing 3. The code in this fragment invokes the method named **getParameters** to read the parameters from the file named **Dsp029.txt**.

```
  public Dsp029(){//constructor

    if(new
File("Dsp029.txt").exists()){
      getParameters();
    }//end if

Listing 3
```

Before invoking the **getParameters** method, however, the program invokes the **exists** method of the **File** class to confirm that the file actually exists. If the file doesn't exist, the call to **getParameters** is skipped, causing the default parameters defined in Listing 1 to be used instead.

The getParameters method

The **getParameters** method is straightforward, so I won't discuss it in detail. You can view it in Listing 16. Suffice it to say that the method reads the input parameters from the disk file and writes their values into the variables declared in Listing 1, overwriting the default values stored in those variables.

In addition, the **getParameters** method displays the values read from the disk file in the format shown in Figure 4 and Figure 6.

Create the sinusoidal data

For simplicity, this program always generates five sinusoids, even if fewer than five were requested as the input parameter value for **numberSinusoids**. In that case, the extra sinusoids are generated using default values and are simply ignored when the sinusoids are plotted.

The code fragment in Listing 4 creates the sinusoidal data for each of the five specified frequencies and saves that data in the array objects that were created in Listing 2.

```
    for(int n = 0;n < len;n++){
      data1[n] =
amp[0]*Math.cos(2*pi*n*freq[0]);
      data2[n] =
amp[1]*Math.cos(2*pi*n*freq[1]);
      data3[n] =
amp[2]*Math.cos(2*pi*n*freq[2]);
      data4[n] =
amp[3]*Math.cos(2*pi*n*freq[3]);
      data5[n] =
amp[4]*Math.cos(2*pi*n*freq[4]);
    }//end for loop

  }//end constructor

Listing 4
```

The end of the constructor

Listing 4 also signals the end of the constructor. When the constructor terminates, an object of the **Dsp029** class has been instantiated. The five arrays shown in Listing 4 have been populated with sinusoidal data according to the parameters read from the file named **Dsp029.txt** or according to the default values of the parameters shown in Listing 1.

Plotting the sinusoidal data

In order to better understand what is going on in the plotting process, it would be helpful for you to read the previous lesson entitled Plotting Engineering and Scientific Data using Java. However, assuming that you don't have the time to do that, I will provide a very brief explanation as to how the plotting programs work.

Using Graph06 to plot the sinusoidal data

To use the plotting program named **Graph06** to plot the sinusoidal data, I needed to do the following:

- Define and compile the program named **Dsp029**, implementing the interface named **GraphIntfc01**. The definition of that interface is shown in Listing 5
- Start the plotting program named **Graph06** running by entering ***java Graph06 Dsp029*** at the command line prompt.

```
public interface GraphIntfc01{
  public int getNmbr();
  public double f1(double x);
  public double f2(double x);
  public double f3(double x);
  public double f4(double x);
  public double f5(double x);
}//end GraphIntfc01

Listing 5
```

What does this do?

When executed in this manner, the program named **Graph06** instantiates an object of the class named **Dsp029** and then invokes the interface methods on that object to obtain the data to be plotted.

In this case, the constructor for the **Dsp029** class populates the five array objects with sinusoidal data. The subsequent invocation of the interface methods by the program named **Graph06** causes that sinusoidal data to be retrieved and plotted by **Graph06**.

The GraphIntfc01 interface methods

A brief description of each of the interface methods is provided in the following sections.

The getNmbr method

Plotting programs based on **GraphIntfc01** can be used to plot any number of functions from one to five.

The method named **getNmbr** must return an integer value between 1 and 5 that specifies the number of functions to be plotted. The plotting program uses that value to divide the total plotting surface into the specified number of plotting areas, and plots each of the functions named **f1** through **fn** in one of those plotting areas.

The methods named f1, f2, f3, f4, and f5

As you can see in Listing 5, each of these methods receives a **double** value as an incoming parameter and returns a **double** value. In essence, each of these methods receives a value for the horizontal coordinate **x** and returns the corresponding value for the vertical coordinate **y**.

One plotting area per method

Each of these methods provides the data to be plotted in one plotting area. The method named **f1** provides the data for the top plotting area; the method named **f2** provides the data for the first plotting area down from the top, and so forth.

> *(For example, if the **getNmbr** method returns a value of 4, the method named **f5** will never be invoked. If **getNmbr** returns 5, the method named **f5** will be invoked to provide the data for the bottom plotting area.)*

How does it work?

Each plotting area contains a horizontal axis. The plotting program moves across the horizontal axis in each plotting area one step at a time *(moving in incremental steps equal to the plotting parameter named xCalcInc, which you will find if you examine the code for **Graph06**).*

At each step along the way, the plotting program invokes the method associated with that plotting area, *(f1, f2, etc.)*, passing the horizontal position as a parameter to the method.

The value returned by the method is assumed to be the vertical value associated with that horizontal position, and that is the vertical value that is plotted for that horizontal position.

Doesn't know and doesn't care

The plotting program doesn't know, and doesn't care how the interface method decides on the value to return for each value that it receives as an incoming parameter. The plotting program simply invokes the methods to get the data, and then plots the returned values.

Computed *"on the fly"*

For example, the returned values could be computed and returned *"on the fly,"* as was the case in the example program named **Graph01Demo**, which I explained in the earlier lesson entitled Plotting Engineering and Scientific Data using Java.

Returned from an array

On the other hand, the values could have been computed earlier and saved in an array. That is the case with all the programs that I will explain in this lesson.

From a disk file, a database, the Internet, etc.

The returned values could be read from a disk file, obtained from a database on another computer, or obtained from any other source such as another computer on the Internet.

All that matters is that when the plotting program invokes one of the five methods named **f1** through **f5**, passing a **double** value as a parameter, it expects to receive a **double** value as a return value, and it will plot the value that it receives.

The getNmbr method

The **getNmbr** method for the class named **Dsp029** is shown in Listing 6.

```
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return numberSinusoids;
  }//end getNmbr
```

Listing 6

This is a very simple method. It returns the value stored in the variable named **numberSinusoids**. This variable may contain the default value established by Listing 1, or may contain the value read from the file named **Dsp029.txt** by the method named **getParameters**.

The method named f1

The code for the method named **f1** is shown in Listing 7.

```
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index >
data1.length-1){
      return 0;
    }else{
      return data1[index];
    }//end else
```

```
  }//end function
```

Listing 7

Note that there is not a one-to-one correspondence between horizontal coordinate values and pixels on the screen. For example, it may sometimes be necessary to plot 90 values across an area of the screen containing 110 pixels. The plotting program must interpolate properly to deal with that issue. Therefore, the plotting program deals with horizontal coordinates at type **double** and then converts those coordinate values to integer pixel values when the time comes to actually draw the material on the screen.

Round from double to int

The method named **f1** receives an incoming horizontal coordinate value as type **double**. It rounds that value to the nearest value of type **long** and casts it to type **int** to determine the index value to use when retrieving the corresponding vertical value from the array object.

If the index value is outside the bounds of the array, the method simply returns a value of zero. Otherwise, it uses the index value to return the value stored in the array object at that index.

The remaining interface methods

The remaining four interface methods are identical to the method named **f1**, except that each method returns data values stored in a different array object. Therefore, I won't discuss those methods.

That's it for Dsp029

That's about it for the program named **Dsp029**. If you understand this program, you are well ahead of the game. The overall structure for the programs named **Dsp028** and **Dsp030** are very similar to the structure for **Dsp029**. The big difference is the manner in which they populate the array objects with the data that is to be plotted. Instead of simply plotting sinusoids, they perform spectral analysis on sinusoids and provide the results of the spectral analysis to be plotted.

Using the interface named GraphIntfc01

As you learned earlier, this is a very simple interface. However, because the class named **Dsp029** implements the interface, the interface definition file must be in the same directory as the source file for **Dsp029** in order to successfully compile **Dsp029**. Therefore, I have provided a complete listing of **GraphIntfc01** in Listing 17 near the end of the lesson.

The program named Graph06

A complete listing of the program named **Graph06** is provided in Listing 18 near the end of the lesson. This is simply a newer version of graphics display programs that I explained in the earlier lesson entitled Plotting Engineering and Scientific Data using Java. Therefore, I won't repeat that explanation here. The comments at the beginning and spread throughout the program provide considerable information about it.

Operational aspects of Graph06

However, an explanation of the operational aspects of the program will be useful here. You can use this program to display the output produced by **Dsp029** by entering the following at the command line prompt:

java Graph06 Dsp029

As you saw in Figure 3 and other previous figures, this program provides the following text fields for user input, along with a button labeled **Graph**:

- **xMin** = minimum x-axis value
- **xMax** = maximum x-axis value
- **yMin** = minimum y-axis value
- **yMax** = maximum y-axis value
- **xTicInt** = tic interval on x-axis
- **yTicInt** = tic interval on y-axis
- **xCalcInc** = calculation interval

These text fields make it possible for you to adjust the plotting parameters and to re-plot the graphs as many times as needed.

You can modify any of these parameters and then click the **Graph** button to cause the five functions to be re-plotted according to the new plotting parameters.

Spectral Analysis using a DFT Algorithm

Now that you have a good idea where we are heading, it's time to start doing some spectral analysis.

Let's begin by looking at some output obtained by performing a spectral analysis on the same five sinusoids shown in Figure 3. The parameters used to perform this spectral analysis are shown in Figure 8. *(I will explain each of these parameters as we go along.)*

```
Data length: 400
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 1.0
Number spectra: 5
Frequencies
```

```
0.03125
0.0625
0.125
0.25
0.5
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

Figure 8

Although more parameters are required to perform spectral analysis than are required to simply generate and plot the sinusoids, the number of sinusoids, the frequencies of the sinusoids, and the amplitudes of the sinusoids in Figure 8 are the same as in Figure 2.

The spectral analysis output

The output produced by performing spectral analysis on these five sinusoids is shown in Figure 9.

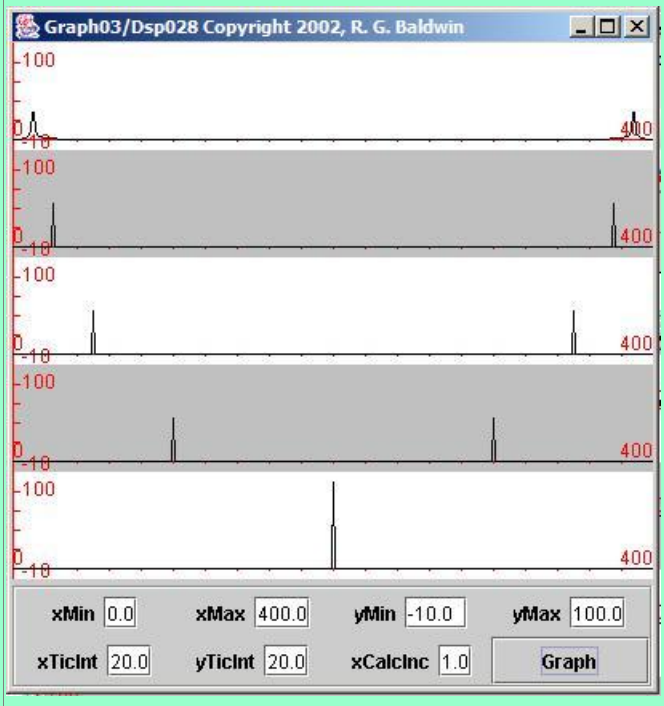


Figure 9 Spectral analysis of five sinusoids.

The format explained

Five separate spectral analyses were performed and the results of those five spectral analyses are shown in Figure 9. Each of the horizontal lines in Figure 9 is the horizontal axis used to display the result of performing a spectral analysis on a different sinusoid. In other words, Figure 9 contains five separate graphs moving from the top to the bottom of the display. The individual graphs have alternating white and gray backgrounds to make them easier to separate visually.

The top graph in Figure 9 shows the result of performing a spectral analysis on the top sinusoid in Figure 3. Moving down the page, each graph in Figure 9 shows the result of performing a spectral analysis on the corresponding sinusoid in Figure 3.

The frequency axes

The horizontal axes in Figure 9 represent the frequency range from zero to the sampling frequency.

> *(The frequency range covered is specified by the **Lower frequency bound** and the **Upper frequency bound** parameters in Figure 8.)*

The horizontal units

The horizontal units in Figure 9 don't represent frequency in an absolute sense of cycles per second or Hertz. Rather, the horizontal units in Figure 9 represent the frequency bins for which spectral energy was computed. In this case, the spectral energy for each sinusoid was computed in 400 equally spaced bins distributed between zero and the sampling frequency.

> *(The number of frequency bins for each individual spectrum computed by this program is equal to the **Data length** parameter in Figure 8. Those frequency bins are distributed uniformly between the **Lower frequency bound** and the **Upper frequency bound** parameters in Figure 8.)*

Location of the folding frequency

Because the right-most end of each horizontal axis in Figure 9 represents the sampling frequency, the center of each horizontal axis represents one-half the sampling frequency, or the Nyquist folding frequency. Thus, the frequency represented by the center of each horizontal axis represents the frequency specified by a value of 0.5 in Figure 8.

A peak at the folding frequency

You can see a large peak in energy at the folding frequency of the bottom graph in Figure 9. That peak corresponds to the frequency of the fifth sinusoid specified in the parameters shown in Figure 8. *(This also corresponds to the spectrum of the bottom graph in Figure 3.)*

Knowing that, you should be able to correlate each of the peaks to the left of center in Figure 9 with the frequencies of the sinusoids specified in Figure 8 and with the individual sinusoids plotted in Figure 3.

The frequency folding effect

Figure 9 clearly shows the frequency folding effect of the sampling process illustrated earlier. As you can see, the peaks in the various graphs to the right of the folding frequency are mirror images of the peaks to the left of the folding frequency. In other words, given a set of samples of a sinusoid, the spectral analysis process is unable to determine whether the peak is above or below the folding frequency, so the energy is equally distributed between two peaks on opposite sides of the folding frequency.

Size of the peak at the folding frequency

Note that the peak in the bottom graph is approximately twice the height of the peaks in the other graphs. This is because the peak at the folding frequency has no mirror-image partner, and all the energy is concentrated in that single peak.

> *(Another interpretation is that two mirror-image peaks converge at the folding frequency causing the resulting peak to be twice as large as either mirror-image peak. I will illustrate this effect with another example later.)*

A short fat peak at the top

You may also have noticed that the peaks in the top graph are shorter and wider than the peaks in the other graphs. This may be because the actual frequency of the sinusoid for the top graph is about half way between the values of the twelfth and thirteenth bins for which spectral energy was computed. Thus, the energy in the sinusoid was spread between the bins on either side of the actual frequency.

This frequency spreading effect can be minimized by increasing the data length to 800 samples. This causes the frequency bins to be only half as wide and the peak in the top graph becomes tall and narrow just like the peaks in the other graphs. You should try this and observe the result when you run the program later.

It is also instructive to plot these spectra with a data length of 400 using the program named **Graph06**. This will show you how the energy is distributed between the frequency bins. This is most effective when the graph is expanded as described in the next section.

Mapping the peaks to pixels

The broadening of the peak in the top graph may also have to do with the requirement to map the peaks in the spectrum to the locations of the actual pixels on the screen. If the location of the peak falls between the positions of two pixels, the plotting program must interpolate the energy in the peak so as to display that energy in actual pixel locations.

This effect can be minimized by plotting the same number of spectral values across a wider area of the screen. When you run this program later, click the maximize button on the **Frame** to

cause the display to occupy the entire screen. That will give you a much better look at the actual shape of each of the peaks. Do this using both **Graph03** and **Graph06** to plot the results.

> *(Note: When switching between the plotting programs, you may need to delete the class files from the old program and compile the new program to avoid having class files with the same names from the two programs becoming intermingled in the same directory.)*

Another DFT example

This next example is designed to illustrate the following features of the DFT algorithm which don't generally apply to an FFT algorithm:

- Ability to do spectral analysis on data of arbitrary lengths. *(With most FFT algorithms, the data length must be a power of two.)*
- Ability to zero in on an arbitrary range of frequencies and to ignore all other frequencies. *(Most FFT algorithms always compute the spectrum at uniform frequency increments from zero to one unit less than the sampling frequency.)*

As mentioned earlier, the DFT algorithm is much more flexible while the FFT algorithm is much faster, particularly for large data lengths.

Peaks merge at the folding frequency

In addition to illustrating these fundamental aspects of the DFT algorithm, this example also illustrates how the mirror image peaks on either side of the folding frequency merge into a single larger peak as the data frequency approaches the folding frequency.

The input parameters

The input parameters are shown in Figure 10. Note in particular the values for the following parameters:

- Data length: 200
- Sample for zero time: 0
- Lower frequency bound: 0.4
- Upper frequency bound: 0.6

```
Data length: 200
Sample for zero time: 0
Lower frequency bound: 0.4
Upper frequency bound: 0.6
Number spectra: 5
Frequencies
0.492
0.494
0.496
```

```
0.498
0.5
Amplitudes
90.0
90.0
90.0
90.0
90.0

Figure 10
```

A different data length

As you can see, the data length for this experiment is different from the data length of 400 used earlier. In addition, neither data length is a power of two.

Computational frequency bounds

As you can also see, the lower and upper frequency bounds are not 0.0 and 1.0 as in the earlier cases. In this case, the frequency bounds describe a much narrower range centered on the folding frequency.

Frequencies are close to the folding frequency

Finally, the frequencies of each of the five sinusoids specified in Figure 10 is progressively closer to the folding frequency with the frequency of the fifth sinusoid being equal to the folding frequency.

The spectral analysis output

The output from the spectral analysis for each of the five sinusoids is shown in Figure 11. *(Another interesting view of the same results is shown later in Figure 14.)*

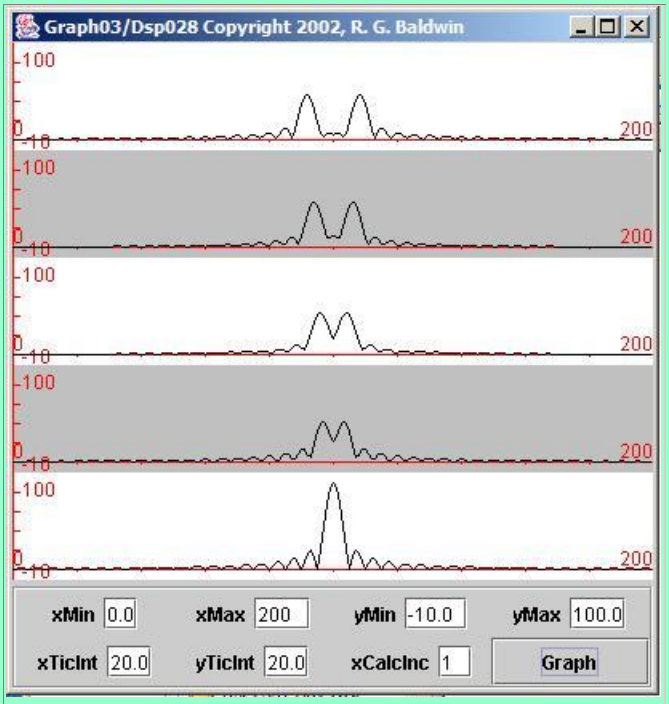


Figure 11 Spectral analysis of five sinusoids.

Peaks are symmetrical to the folding frequency

The spectral peaks shown in Figure 11 are symmetrical to the folding frequency, which in turn is centered horizontally in each of the graphs. As you already know, the peaks are always symmetrical to the folding frequency due to the frequency folding at that frequency.

The folding frequency is centered horizontally due to the way that I defined the lower and upper frequency bounds, and the way that I adjusted the plotting parameters.

Peaks are well defined and wider than before

The peaks are well defined because I computed the spectral energy at 200 points across the specified frequency range from 0.4 to 0.6. Thus, the frequency bins at which I computer spectral energy were much narrower than before.

The peaks are wider because I displayed a much smaller slice of the entire frequency spectrum in the same physical screen space.

Peaks merge at the folding frequency

As the frequency of each sinusoid approaches the folding frequency, the two mirror-image peaks corresponding to that sinusoid merge into a single peak with twice the height at the folding frequency. This agrees with what you saw in Figure 9, but on a much more detailed basis.

It would be difficult to perform this experiment using an FFT algorithm because of the inherent limitations built into the algorithm. The FFT algorithm sacrifices flexibility for speed.

Implementing the DFT algorithm

At this point, I will present and explain two different programs:

- **Dsp028** - Driver program for doing spectral analysis using a DFT algorithm.
- **ForwardRealToComplex01** - Class that implements the DFT algorithm.

In addition, I will present, but will not explain the plotting program named **Graph03**.

The Dsp028 program

This driver program is similar in many respects to the program named **Dsp029** that I explained earlier. It differs mainly in how it populates the array objects containing the data that is plotted by the plotting program.

The program named **Dsp029** simply populates those array objects with five sinusoidal functions. The program named **Dsp028** also creates five sinusoidal functions. However, it passes those functions to a static method named **transform** belonging to the **ForwardRealToComplex01** class to perform spectral analysis on those functions. The results of the spectral analysis are used to populate the five array objects whose contents are plotted by the plotting program.

Because of the similarity of the two programs, my discussion of **Dsp028** will be much more brief than was my discussion of **Dsp029**.

Computes and displays the magnitude spectrum

The program named **Dsp028** computes and displays the magnitude of the spectral content for up to five sinusoids having different frequencies and amplitudes.

> *(Future lessons will discuss other aspects of spectral analysis such as the complex spectrum, zero time, and the phase angle.)*

Input parameters

The program gets input parameters from a file named **Dsp028.txt**. If that file doesn't exist in the current directory, the program uses a set of default parameters. As with the program named **Dsp029**, each parameter value must be stored as characters on a separate line in the file named **Dsp028.txt**. The required parameters are shown in Figure mm.

```
Data length as type int
Sample number representing zero time
```

```
as type int
Lower frequency bound as type double
Upper frequency bound as type double
Number of spectra as type int.  Max
value is 5.
List of sinusoid frequency values as
type double.
List of sinusoid amplitude values as
type double.
```

Figure mm Required input parameters for Dsp028.

Don't allow blank lines at the end of the data in the file.

The number of values in each of the lists must match the value for the number of spectra.

Specification of sinusoidal frequencies

As before, each frequency value is specified as a **double** value representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is one-half the sampling frequency.

Example contents for Dsp028.txt

Figure 12 shows the contents of the file named **Dsp028.txt** that represent the parameters shown in Figure 10.

```
200
0
0.4
0.6
5
0.492
0.494
0.496
0.498
0.5
90
90
90
90
90
```

Figure 12 Contents of Dsp028.txt file

Performing the spectral analysis

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.

Will discuss the code in fragments

As usual, I will discuss the code in fragments. A complete listing of the program is presented in Listing 19 near the end of the lesson. Because of the similarity of **Dsp028** with **Dsp029** discussed earlier, the fragments for **Dsp028** will be much larger and will be explained in much less detail.

The class definition begins in Listing 8. For reasons that you already understand, this class implements the interface named **GraphIntfc01**.

```
class Dsp028 implements GraphIntfc01{
  final double pi = Math.PI;//for
simplification

  //Begin default parameters
  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 = 1.0;
  int numberSpectra = 5;
  //Frequencies of the sinusoids
  double[] freq =
{0.1,0.2,0.3,0.4,0.5};
  //Amplitudes of the sinusoids
  double[] amp = {60,70,80,90,100};
  //End default parameters

Listing 8
```

The code in Listing 8 defines a set of default parameter values that are used in the event that a file named **Dsp028.txt** does not exist in the current directory.

Declare array variables

The code in Listing 9 declares several array variables that will be used to point to array objects whose purposes are explained in the comments.

```
  //Following arrays will contain data
that is
  // input to the spectral analysis
```

```
process.
  double[] data1;
  double[] data2;
  double[] data3;
  double[] data4;
  double[] data5;

  //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[] magnitude1;
  double[] magnitude2;
  double[] magnitude3;
  double[] magnitude4;
  double[] magnitude5;
```

Listing 9

The constructor

The constructor for the class begins in Listing 10. The constructor begins by getting the parameters from a file named **Dsp028.txt**. If that file doesn't exist in the current directory, default parameters are used.

```
  public Dsp028(){//constructor

    if(new
File("Dsp028.txt").exists()){
      getParameters();
    }//end if
```

Listing 10

Always processes five sinusoids

For simplicity, this program always processes five sinusoids, even if fewer than five were requested as the input parameter for **numberSpectra**. In that case, the extra sinusoids are processed using default values and simply ignored when the results are plotted.

Create the raw sinusoidal data

The code in Listing 11 instantiates array objects and creates the sinusoidal data upon which spectral analysis will be performed.

```
    //First create empty array
objects.
    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];
    //Now populate the array objects
    for(int n = 0;n < len;n++){
      data1[n] =
amp[0]*Math.cos(2*pi*n*freq[0]);
      data2[n] =
amp[1]*Math.cos(2*pi*n*freq[1]);
      data3[n] =
amp[2]*Math.cos(2*pi*n*freq[2]);
      data4[n] =
amp[3]*Math.cos(2*pi*n*freq[3]);
      data5[n] =
amp[4]*Math.cos(2*pi*n*freq[4]);
    }//end for loop

Listing 11
```

Perform the spectral analysis

The code in Listing 12 creates array objects to receive the results and invokes the static **transform** method of the **forwardRealToComplex01** class five times in succession to perform the spectral analysis on each of the five sinusoids.

*(I will explain the **transform** method that performs the spectral analysis shortly.)*

Only the magnitude data is displayed by this program. Therefore, the arrays that receive the other spectral analysis results from the **transform** method are discarded each time a new spectral analysis is performed.

```
    magnitude1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];

ForwardRealToComplex01.transform(data1,real,

imag,angle,magnitude1,zeroTime,lowF,highF);

    magnitude2 = new double[len];
    real = new double[len];
    imag = new double[len];
```

```
    angle = new double[len];

ForwardRealToComplex01.transform(data2,real,

imag,angle,magnitude2,zeroTime,lowF,highF);

    magnitude3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];

ForwardRealToComplex01.transform(data3,real,

imag,angle,magnitude3,zeroTime,lowF,highF);

    magnitude4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];

ForwardRealToComplex01.transform(data4,real,

imag,angle,magnitude4,zeroTime,lowF,highF);

    magnitude5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];

ForwardRealToComplex01.transform(data5,real,

imag,angle,magnitude5,zeroTime,lowF,highF);
  }//end constructor
```

Listing 12

The spectral magnitude results

Note that the magnitude results are saved in the array objects referred to by **magnitude1**, **magnitude2**, etc. This will be important later when I discuss the interface methods defined by **Dsp028**.

The end of the constructor

Listing 12 also signals the end of the constructor. When the constructor terminates, the object has been instantiated and populated with spectral analysis results for five sinusoids using the parameters specified by the file named **Dsp028.txt**.

The getParameters method

The getParameters method used in this program is the same as that used in **Dsp029**, so I won't discuss it further.

The interface methods

The **Dsp028** class must define the same six interface methods as the **Dsp029** class discussed earlier. The only difference in the interface methods is the identification of the array objects from which the methods return data when the methods are invoked.

The code in Listing 13 is typical of the code for methods **f1** through **f5**. As you can see, these methods return the data stored in the magnitude arrays. These are the spectral analysis results that are plotted in Figure 9, Figure 11, and later in Figure 14.

```
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                 index >
magnitude1.length-1){
      return 0;
    }else{
      return magnitude1[index];
    }//end else
  }//end function

Listing 13
```

The program named Graph03

The plots in Figure 9 and Figure 11 were produced by entering the following at the command line prompt:

java Graph03 Dsp028

The program named **Graph03** is very similar to the program named **Graph06** discussed earlier. In fact, the program named **Graph06** can be used to produce very similar plots where the sample values are represented by vertical bars instead of being represented by connected dots. This results in the very interesting display shown in Figure 14. Each of the vertical bars in Figure 14 represents a computational frequency bin. *(Compare Figure 14 with Figure 11.)*

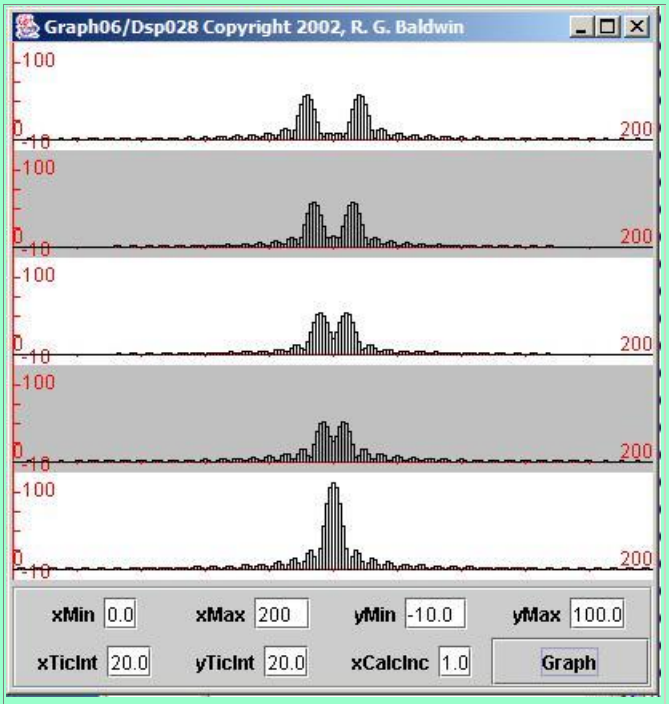


Figure 14 Spectral analysis of five sinusoids.

In any event, **Graph03** is so similar to **Graph06** that I'm not going to discuss it further. A complete listing of the program named **Graph03** is provided in Listing 20 near the end of the lesson.

The transform method of the ForwardRealToComplex01 class

That brings us to the heart of this lesson, which is the method that actually implements the DFT algorithm and performs the spectral analysis. This is a method named **transform**, which is a static method of the class named **ForwardRealToComplex01**. You saw this method being invoked five times in the code in Listing 12.

Will discuss in fragments

As usual, I will discuss this method in fragments. A complete listing of the class is presented in Listing 21 near the end of the lesson.

The **transform** method is a rather straightforward implementation of the concepts that I explained in the earlier lesson entitled Fun with Java, How and Why Spectral Analysis Works. If you have not done so already, I strongly urge you go to back and study that lesson at this time. You need to understand those concepts in order for the code in the **transform** method to make sense.

A brief description

For those of you who don't have the time to go back and study that lesson in detail, a brief description of the DFT algorithm follows.

Using a notation that I described in the earlier lesson, the expressions that you must evaluate to determine the frequency spectral content of a target time series at a frequency F are shown in Figure 15.

```
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)
Power(F) = Real(F)*Real(F) +
Imag(F)*Imag(F)
Amplitude(F) = SqRt(Power(F))
```

Figure 15

What does this really mean?

Before you panic, let me explain what this means in layman's terms. Given a time series, **x(n)**, you can determine if that time series contains a cosine component or a sine component at a given frequency, **F**, by doing the following:

- Create one new time series, **cos(n)**, which is a cosine function with the frequency **F**.
- Create another new time series, **sin(n)**, which is a sine function with the frequency **F**.
- Multiply **x(n)** by **cos(n)** and compute the sum of the products. Save this value, calling it **Real(F)**. This is an estimate of the amplitude, if any, of the cosine component with the matching frequency contained in the time series **x(n)**.
- Multiply **x(n)** by **sin(n)** and compute the sum of the products. Save this value, calling it **Imag(f)**. This is an estimate of the amplitude, if any, of the sine component with the matching frequency contained in the time series **x(n)**.
- Consider the values for **Real(F)** and **Imag(F)** to be the real and imaginary parts of a complex number.
- Consider the sum of the squares of the real and imaginary parts to represent the power at that frequency in the time series.
- Consider the square root of the power to be the amplitude at that frequency in the time series. *(This is the value that is plotted in Figure 9, Figure 11, and Figure 14.)*

Compute the complex energy at each frequency

That is all there is to it. For each frequency of interest, you can use this process to compute a complex number, **Real(F)-jImag(F)**, which represents the complex energy corresponding to that frequency in the target time series.

Similarly, you can compute the sum of the squares of the real and imaginary parts and consider that to be a measure of the power at that frequency in the time series. The square root of the power is the amplitude of the energy at that frequency.

Nested for loops

Normally we are interested in more than one frequency, so we would repeat the above procedure once for each frequency of interest. This suggests the use of nested **for** loops in the algorithm. The outer loop specifies the frequency of interest. The inner loop computes the sum of the products at a particular frequency.

Description of the transform method

The static method named **transform** performs a real to complex Fourier transform. The method does not implement the FFT algorithm. Rather, it implements a straightforward sampled data version of the continuous Fourier transform defined using integral calculus. *(See ForwardRealToComplexFFT01 for an FFT algorithm.)*

The return values

The method returns the following:

- Real part of the spectral analysis result
- Imaginary part of the spectral analysis result
- Magnitude of the spectral analysis result
- Phase angle of the spectral analysis result in degrees

The transform method parameters

The method parameters are:

- double[] data - incoming real data
- double[] realOut - outgoing real data
- double[] imagOut - outgoing imaginary data
- double[] angleOut - outgoing phase angle in degrees
- double[] magnitude - outgoing amplitude spectrum
- int zero - the index of the incoming data sample that represents zero time
- double lowF - low frequency limit for computation as a fraction of sampling frequency
- double highF - high frequency limit for computation as a fraction of sampling frequency

Frequency increment, magnitude spectrum, and number of returned values

The computational frequency increment is the difference between the high and low limits divided by the length of the magnitude array.

The magnitude or amplitude is computed as the square root of the sum of the squares of the real and imaginary parts. This value is divided by the incoming data length, which is given by **data.length**.

The method returns a number of points in the frequency domain equal to the incoming data length regardless of the high and low frequency limits.

The beginning of the transform method

The class and the **transform** method begin in Listing 14. The code in Listing 14 is described above.

```
public class ForwardRealToComplex01{

  public static void transform(
                                double[]
data,
                                double[]
realOut,
                                double[]
imagOut,
                                double[]
angleOut,
                                double[]
magnitude,
                                int
zero,
                                double
lowF,
                                double
highF){
    double pi = Math.PI;//for
convenience
    int dataLen = data.length;
    double delF = (highF-
lowF)/data.length;
```

Listing 14

The remainder of the method and the class

The nested **for** loops discussed above are included in the code shown in Listing 15. As suggested above, the outer loop iterates on frequency while the inner loop iterates on the values that make up the incoming samples. The code in the inner loop computes the sum of the product of the time series and the reference cosine and sine functions.

```
    //Outer loop iterates on frequency
    // values.
    for(int i=0; i < dataLen;i++){
```

```
      double freq = lowF + i*delF;
      double real = 0.0;
      double imag = 0.0;
      double ang = 0.0;
      //Inner loop iterates on time-
      // series points.
      for(int j=0; j < dataLen; j++){
        real += data[j]*Math.cos(

2*pi*freq*(j-zero));
        imag += data[j]*Math.sin(

2*pi*freq*(j-zero));
      }//end inner loop

      realOut[i] = real/dataLen;
      imagOut[i] = imag/dataLen;
      magnitude[i] = (Math.sqrt(
                   real*real +
imag*imag))/dataLen;

      //Calculate and return the phase
      // angle in degrees.
      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;
    }//end outer loop
  }//end transform method

}//end class ForwardRealToComplex01
```

Listing 15

Store results in output array objects

At the end of each iteration of the inner loop, code in the outer loop deposits the real, imaginary, magnitude, and phase angle results in the output array objects. To accomplish this, the code:

- Computes the magnitude or amplitude as the square root of the sum of the squares of the real and imaginary parts.
- Performs some trigonometry operations to determine the phase angle in degrees based on the values of the real and imaginary parts.

Now you know about the DFT algorithm

Now you know about the DFT algorithm. You also know about some of the fundamental aspects of spectral analysis involving the sampling frequency and the folding frequency.

Future lessons will discuss other aspects of spectral analysis including:

- Frequency resolution versus data length.
- The relationship between the phase angle and delays in the time domain.
- The reversible nature of the Fourier transform involving both forward and inverse Fourier transforms.

Spectral Analysis using an FFT Algorithm

At this point, I will present a similar spectral analysis program that uses an FFT algorithm. I will present this program with very little discussion. I am providing it in this lesson for two primary purposes:

- To allow you to experiment and appreciate the flexibility of the DFT as compared to the FFT.
- To allow you to experiment and appreciate the speed of the FFT as compared to the DFT.

The program named Dsp030

The program named **Dsp030** is very similar to **Dsp028**. The major differences are:

- Because it uses an FFT algorithm, **Dsp030** is much less flexible than **Dsp028**, particularly with respect to data length and selection of the frequencies of interest.
- Because it uses an FFT algorithm, **Dsp030** is much faster than **Dsp028**, particularly when used to perform spectral analysis on long data lengths.

A complete listing of **Dsp030** is provided in Listing 22.

Description of the program named Dsp030

This program uses an FFT algorithm to compute and display the magnitude of the spectral content for up to five sinusoids having different frequencies and amplitudes. *(See the program named **Dsp028** for a program that does not use an FFT algorithm.)*

The input parameters

The program gets input parameters from a file named **Dsp030.txt**. If that file doesn't exist in the current directory, the program uses a set of default parameters.

Each parameter value must be stored as characters on a separate line in the file named **Dsp030.txt**. The required input parameters are shown in Figure 16. *(Contrast this with the required input parameters for **Dsp028** shown in Figure mm.)*

```
Data length as type int
Number of spectra as type int. Max
value is 5.
List of sinusoid frequency values as
type double.
List of sinusoid amplitude values as
type double.

Figure 16 Required input parameters
for Dsp030.
```

Note in contrast with Figure mm, the required input parameters for **Dsp030** do not include the sample number representing zero time, the lower frequency bound for computation of the spectra, and the upper frequency bound for computation of the spectra.

> *(The computational frequency range cannot be specified for the FFT algorithm. It always computes the spectra from zero to one unit less than the sampling frequency.)*

Restrictions on the data length

Note also that the data length must always be a power of two. Otherwise, the FFT algorithm will fail to run properly.

> *(This restriction is an important contributor to the speed achieved by the FFT algorithm.)*

The sinusoidal frequency values

As with **Dsp028**, the number of values in each of the lists must match the value for the number of spectra.

All frequency values are specified as a **double** representing a fractional part of the sampling frequency.

Figure 17 shows the parameters used to produce the spectral analysis plots shown later in Figure 18.

> *(Note that the data length is a power of two as required by the FFT.)*

```
256
5
0.1
0.2
0.3
0.5
0.005
90
90
90
90
90
```

Figure 17

The plotting program

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. This requires that you enter the following at the command line prompt:

```
java Graph03 Dsp030
```

The plotting program named Graph06 can also be used to plot the data produced by this program, requiring that you enter the following at the command line prompt:

```
java Graph06 Dsp030
```

The transform method

A static method named **transform** belonging to the class named **ForwardRealToComplexFFT01** is used to perform the actual spectral analysis. The method named **transform** implements an FFT algorithm. The FFT algorithm requires that the data length be a power of two. This method will be discussed very briefly later.

A sample FFT spectral analysis

The output produced by running **Dsp030** using the input parameters shown in Figure 17 is shown in Figure 18.

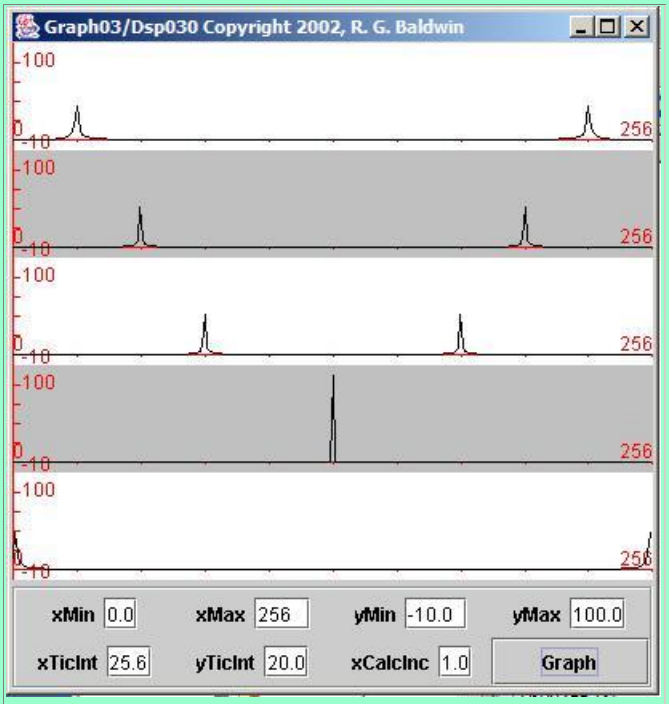


Figure 18 FFT of five sinusoids.

Nothing special here

There is nothing special about this particular spectral analysis. I presented it here to illustrate the use of the FFT algorithm for spectral analysis. You should be able to produce the same results using the same program and the same parameters.

A matching DFT spectral analysis

Figure 19 shows the parameters required for the program named **Dsp028** to perform a DFT spectral analysis producing the same results as those produced by the FFT analysis shown in Figure 18. Note that the data length has been set to 256 and the computational frequency range extends from zero to the sampling frequency in Figure 19.

```
Data length: 256
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 1.0
Number spectra: 5
Frequencies
0.1
0.2
0.3
0.5
0.0050
Amplitudes
```

```
90.0
90.0
90.0
90.0
90.0

Figure 19
```

The matching DFT output

The DFT output produced by running **Dsp028** with the parameters shown in Figure 19 is shown in Figure 20.

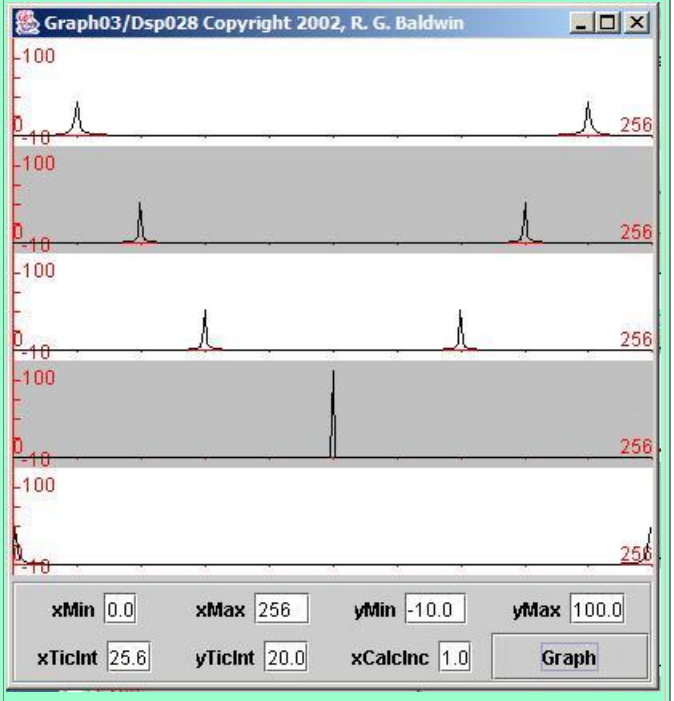


Figure 20 DFT of five sinusoids.

Hopefully you noticed that Figure 20 looks almost exactly like Figure 18. This is how it should be. The DFT algorithm and the FFT algorithm are simply two different algorithms for computing the same results. However, the DFT algorithm is much more flexible than the FFT algorithm while the FFT algorithm is much faster than the DFT algorithm.

Repeat these two experiments

I recommend that you repeat these two experiments several times increasing the data length to a higher power of two each time you run the experiments.

On my machine, the DFT algorithm used by **Dsp028** becomes noticeably slow by the time the data length reaches 2048 samples. However, the FFT algorithm used by **Dsp030** is still reasonably responsive at a data length of 131,072 samples.

> *(Performing the DFT on five input samples each having a data length of 131,072 samples would require an intolerably long time on my machine.)*

If what you need is speed for long data lengths, the FFT is your best approach. On the other hand, if you need more flexibility than the FFT provides and the data length is not too long, then the DFT may be your best approach.

The ForwardRealToComplexFFT01 class

The **ForwardRealToComplexFFT01** class containing the method that implements the FFT algorithm is provided in Listing 23 near the end of this lesson.

The FFT algorithm is based on some very complicated signal processing concepts. I'm not going to explain how this algorithm works in this lesson because I haven't given you the proper background for understanding it. I plan to explain additional signal processing concepts in future lessons that will prepare you to understand how the FFT algorithm works.

Fortunately, you don't have to understand how the FFT algorithm works to be able to use it.

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.

I suggest that you begin by compiling and running the following files to confirm that everything is working correctly on your machine before attempting to compile and run the spectral analysis programs:

- **Dsp029.java**
- **GraphIntfc01.java**
- **Graph06.java**

Make sure that you create an appropriate file named **Dsp029.txt**, as described in Figure 2. You should be able to reproduce my results if everything is working correctly.

Once you confirm that things are working correctly, copy, compile, and run the spectral analysis programs. Experiment with the parameters and try to understand the result of making changes to the parameters. Confirm the flexibility of the DFT algorithm and the speed of the FFT algorithm.

Summary

In this lesson I have provided and explained programs that illustrate the impact of sampling and the Nyquist folding frequency.

I have also provided and explained several different programs used for performing spectral analysis. The first program was a very general program that implements a Discrete Fourier Transform *(DFT)* algorithm. I explained this program in detail.

The second program was a less general, but much faster program that implements a Fast Fourier Transform *(FFT)* algorithm. I will defer an explanation of this program until a future lesson. I provided it in this lesson so that you can use it and compare it with the DFT program in terms of speed and flexibility.

What's Next?

Future lessons will discuss other aspects of spectral analysis including:

- Frequency resolution versus data length.
- The relationship between the phase angle and delays in the time domain.
- The reversible nature of the Fourier transform involving both forward and inverse Fourier transforms.
- Additional material aimed towards an understanding of the signal processing concepts behind the FFT algorithm.

Complete Program Listings

Complete listings of all the programs discussed in this lesson follow.

```
/* File Dsp029.java
Copyright 2004, R.G.Baldwin
Rev 5/6/04

Generates and displays up to five sinusoids
having different frequencies and amplitudes. Very
useful for providing a visual illustration of the
way in which frequencies above half the sampling
frequency fold back down into the area bounded
by zero and half the sampling frequency (the
Nyquist folding frequency).

Gets input parameters from a file named
Dsp029.txt.  If that file doesn't exist in the
current directory, the program uses a set of
default parameters.

Each parameter value must be stored as characters
on a separate line in the file named Dsp029.txt.
The required parameters are as follows:
```

```
Data length as type int
Number of sinusoids as type int.  Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

The number of values in each of the lists must
match the value for the number of spectra.

Note:  All frequency values are specified as a
double representing a fractional part of the
sampling frequency.

Here is a set of sample parameter values.  Don't
allow blank lines at the end of the data in the
file.

400.0
5
0.1
0.9
1.1
1.9
2.1
90
90
90
90
90

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 Graph06 can be used
to plot the data produced by this program.  When
it is used, the usage information is:

java Graph06 Dsp029

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

class Dsp029 implements GraphIntfc01{
  final double pi = Math.PI;//for simplification

  //Begin default parameters
  int len = 400;//data length
  int numberSinusoids = 5;
  //Frequencies of the sinusoids
  double[] freq = {0.1,0.25,0.5,0.75,0.9};
  //Amplitudes of the sinusoids
  double[] amp = {75,75,75,75,75};
  //End default parameters

  //Following arrays will be populated with
```

```
  // sinusoidal data to be plotted
  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 Dsp029(){//constructor

    //Get the parameters from a file named
    // Dsp029.txt.  Use the default parameters
    // if the file doesn't exist in the current
    // directory.
    if(new File("Dsp029.txt").exists()){
      getParameters();
    }//end if

    //Note that this program always generates
    // five sinusoids, even if fewer than five
    // were requested as the input parameter
    // for numberSinusoids.  In that case, the
    // extras are generated using default values
    // and simply ignored when the results are
    // plotted.

    //Create the raw data.  Note that the
    // argument for a sinusoid at half the
    // sampling frequency would be (2*pi*x*0.5).
    // This would represent one half cycle or pi
    // radians per sample.
    for(int n = 0;n < len;n++){
      data1[n] = amp[0]*Math.cos(2*pi*n*freq[0]);
      data2[n] = amp[1]*Math.cos(2*pi*n*freq[1]);
      data3[n] = amp[2]*Math.cos(2*pi*n*freq[2]);
      data4[n] = amp[3]*Math.cos(2*pi*n*freq[3]);
      data5[n] = amp[4]*Math.cos(2*pi*n*freq[4]);
    }//end for loop

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

  //This method gets processing parameters from
  // a file named Dsp029.txt and stores those
  // parameters in instance variables belonging
  // to the object of type Dsp029.
  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(
                                   "Dsp029.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;
    len = (int)Double.parseDouble(data[cnt++]);
    numberSinusoids = (int)Double.parseDouble(
                                      data[cnt++]);
    for(int fCnt = 0;fCnt < numberSinusoids;
                                          fCnt++){
      freq[fCnt] = Double.parseDouble(
                                      data[cnt++]);
    }//end for loop

    for(int aCnt = 0;aCnt < numberSinusoids;
                                          aCnt++){
      amp[aCnt] = Double.parseDouble(
                                      data[cnt++]);
    }//end for loop

    //Print parameter values.
    System.out.println();
    System.out.println("Data length: " + len);
    System.out.println(
         "Number sinusoids: " + numberSinusoids);
    System.out.println("Frequencies");
    for(cnt = 0;cnt < numberSinusoids;cnt++){
      System.out.println(freq[cnt]);
    }//end for loop
    System.out.println("Amplitudes");
    for(cnt = 0;cnt < numberSinusoids;cnt++){
      System.out.println(amp[cnt]);
    }//end for loop

  }//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 numberSinusoids;
```

```
  }//end getNmbr
  //-----------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > data1.length-1){
      return 0;
    }else{
      return data1[index];
    }//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];
    }//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];
    }//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];
    }//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];
    }//end else
  }//end function
  //-----------------------------------------//

}//end class Dsp029
```

Listing 16

```
/* File GraphIntfc01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04

This interface must be implemented by classes
whose objects produce data to be plotted by
programs such as Graph03 and Graph06.

Tested using SDK 1.4.2 under WinXP.
**********************************************/

public interface GraphIntfc01{
  public int getNmbr();
  public double f1(double x);
  public double f2(double x);
  public double f3(double x);
  public double f4(double x);
  public double f5(double x);
}//end GraphIntfc01
```

Listing 17

```
/* File Graph06.java
Copyright 2002, R.G.Baldwin
Revised 5/15/04

Very similar to Graph03, except that
each point is displayed as a
rectangle, centered on the sample.
Can be used to explain integration
through summation of the sample values.

Note:  This program requires access to
the interface named GraphIntfc01.

This is a plotting program.  It is
designed to access a class file, which
implements GraphIntfc01, and to plot up
to five functions defined in that class
file.  The plotting surface is divided
into the required number of equally
sized plotting areas, and one function
is plotted on cartesian coordinates in
each area.

The methods corresponding to the
functions are named f1, f2, f3, f4,
and f5.
```

The class containing the functions must also define a method named getNmbr(), which takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a NoSuchMethodException will be thrown.

Note that the constructor for the class that implements GraphIntfc01 must not require any parameters due to the use of the newInstance method of the Class class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with f5 and work down toward f1. For example, if the number of functions is 3, then the program will expect to call methods named f1, f2, and f3. It is OK for the absent methods to be defined in the class. They simply won't be invoked.

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

The cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named

```
junk and plots the data provided by
that class.  Thus, you can test the
program by running it with no
command-line parameter.

This program provides the following
text fields for user input, along with
a button labeled Graph.  This allows
the user to adjust the parameters and
replot the graph as many times with as
many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these
parameters and then click the Graph
button to cause the five functions
to be re-plotted according to the
new parameters.

Whenever the Graph button is clicked,
the event handler instantiates a new
object of the class that implements
the GraphIntfc01 interface.  Depending
on the nature of that class, this may
be redundant in some cases.  However,
it is useful in those cases where it
is necessary to refresh the values of
instance variables defined in the
class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were
first defined in the Color class of
v1.4.0.  Therefore, the program
requires v1.4.0 or later to compile and
run correctly.
**************************************/

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph06{
  public static void main(
          String[] args)
          throws NoSuchMethodException,
```

```
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{
    if(args.length == 1){
      //pass command-line paramater
      new GUI(args[0]);
    }else{
      //no command-line parameter given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph06 definition
//===================================//

class GUI extends JFrame
             implements ActionListener{

  //Define plotting parameters and
  // their default values.
  double xMin = 0.0;
  double xMax = 400.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 20.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small
  // on x-axis, a default value is
  // used later.
  double xTicLen = (yMax-yMin)/50;
  double yTicLen = (xMax-xMin)/50;

  //Calculation interval along x-axis
  double xCalcInc = 1.0;

  //Text fields for plotting parameters
  JTextField xMinTxt =
             new JTextField("" + xMin);
  JTextField xMaxTxt =
             new JTextField("" + xMax);
  JTextField yMinTxt =
             new JTextField("" + yMin);
  JTextField yMaxTxt =
             new JTextField("" + yMax);
  JTextField xTicIntTxt =
           new JTextField("" + xTicInt);
  JTextField yTicIntTxt =
           new JTextField("" + yTicInt);
  JTextField xCalcIncTxt =
          new JTextField("" + xCalcInc);

  //Panels to contain a label and a
  // text field
  JPanel pan0 = new JPanel();
```

```
  JPanel pan1 = new JPanel();
  JPanel pan2 = new JPanel();
  JPanel pan3 = new JPanel();
  JPanel pan4 = new JPanel();
  JPanel pan5 = new JPanel();
  JPanel pan6 = new JPanel();

  //Misc instance variables
  int frmWidth = 408;
  int frmHeight = 430;
  int width;
  int height;
  int number;
  GraphIntfc01 data;
  String args = null;

  //Plots are drawn on the canvases
  // in this array.
  Canvas[] canvases;

  //Constructor
  GUI(String args)throws
                NoSuchMethodException,
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{

    if(args != null){
      //Save for use later in the
      // ActionEvent handler
      this.args = args;
      //Instantiate an object of the
      // target class using the String
      // name of the class.
      data = (GraphIntfc01)
                   Class.forName(args).
                         newInstance();
    }else{
      //Instantiate an object of the
      // test class named junk.
      data = new junk();
    }//end else

    //Create array to hold correct
    // number of Canvas objects.
    canvases =
            new Canvas[data.getNmbr()];

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
      throw new NoSuchMethodException(
                "Too many functions.  "
                  + "Only 5 allowed.");
    }//end if
```

```
    //Create the control panel and
    // give it a border for cosmetics.
    JPanel ctlPnl = new JPanel();
    ctlPnl.setLayout(//?rows x 4 cols
                  new GridLayout(0,4));
    ctlPnl.setBorder(
                   new EtchedBorder());

    //Button for replotting the graph
    JButton graphBtn =
                  new JButton("Graph");
    graphBtn.addActionListener(this);

    //Populate each panel with a label
    // and a text field.  Will place
    // these panels in a grid on the
    // control panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);

    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);

    //Add the populated panels and the
    // button to the control panel with
    // a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);

    //Create a panel to contain the
    // Canvas objects.  They will be
    // displayed in a one-column grid.
    JPanel canvasPanel = new JPanel();
    canvasPanel.setLayout(//?rows,1 col
```

```
                new GridLayout(0,1));

    //Create a custom Canvas object for
    // each function to be plotted and
    // add them to the one-column grid.
    // Make background colors alternate
    // between white and gray.
    for(int cnt = 0;
                cnt < number; cnt++){
      switch(cnt){
        case 0 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                        Color.WHITE);
          break;
        case 1 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                    Color.LIGHT_GRAY);
          break;
        case 2 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                        Color.WHITE);
          break;
        case 3 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                    Color.LIGHT_GRAY);
          break;
        case 4 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].
            setBackground(Color.WHITE);
      }//end switch
      //Add the object to the grid.
      canvasPanel.add(canvases[cnt]);
    }//end for loop

    //Add the sub-assemblies to the
    // frame.  Set its location, size,
    // and title, and make it visible.
    getContentPane().
                  add(ctlPnl,"South");
    getContentPane().
             add(canvasPanel,"Center");

    setBounds(0,0,frmWidth,frmHeight);

    if(args == null){
      setTitle("Graph06, " +
```

```
                "Copyright 2002, " +
                "Richard G. Baldwin");
  }else{
    setTitle("Graph06/" + args +
                " Copyright 2002, " +
                "R. G. Baldwin");
  }//end else

  setVisible(true);

  //Set to exit on X-button click
  setDefaultCloseOperation(
                    EXIT_ON_CLOSE);

  //Guarantee a repaint on startup.
  for(int cnt = 0;
                cnt < number; cnt++){
    canvases[cnt].repaint();
  }//end for loop

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

//This event handler is registered
// on the JButton to cause the
// functions to be replotted.
public void actionPerformed(
                    ActionEvent evt){
  //Re-instantiate the object that
  // provides the data
  try{
    if(args != null){
      data = (GraphIntfc01)Class.
         forName(args).newInstance();
    }else{
      data = new junk();
    }//end else
  }catch(Exception e){
    //Known to be safe at this point.
    // Otherwise would have aborted
    // earlier.
  }//end catch

  //Set plotting parameters using
  // data from the text fields.
  xMin = Double.parseDouble(
                  xMinTxt.getText());
  xMax = Double.parseDouble(
                  xMaxTxt.getText());
  yMin = Double.parseDouble(
                  yMinTxt.getText());
  yMax = Double.parseDouble(
                  yMaxTxt.getText());
  xTicInt = Double.parseDouble(
                xTicIntTxt.getText());
  yTicInt = Double.parseDouble(
```

```
                yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(
                xCalcIncTxt.getText());

    //Calculate new values for the
    // length of the tic marks on the
    // axes.  If too small on x-axis,
    // a default value is used later.
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting areas
    for(int cnt = 0;
                  cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end actionPerformed
  //--------------------------------//


//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
  int cnt;//object number
  //Factors to convert from double
  // values to integer pixel locations.
  double xScale;
  double yScale;

  MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
  }//end constructor

  //Override the paint method
  public void paint(Graphics g){
    //Get and save the size of the
    // plotting surface
    width = canvases[0].getWidth();
    height = canvases[0].getHeight();

    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);

    //Get initial data values
    double xVal = xMin;
    int oldX = getTheX(xVal);
```

```
    int oldY = 0;
    //Use the Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
      case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
      case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
      case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
      case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
      case 4 :
        oldY = getTheY(data.f5(xVal));
    }//end switch

    //Now loop and plot the points
    while(xVal < xMax){
      int yVal = 0;
      //Get next data value.  Use the
      // Canvas obj number to
      // determine which method to
      // invoke to get the value for y.
      switch(cnt){
        case 0 :
          yVal =
                getTheY(data.f1(xVal));
          break;
        case 1 :
          yVal =
                getTheY(data.f2(xVal));
          break;
        case 2 :
          yVal =
                getTheY(data.f3(xVal));
          break;
        case 3 :
          yVal =
                getTheY(data.f4(xVal));
          break;
        case 4 :
          yVal =
                getTheY(data.f5(xVal));
      }//end switch1


      //Convert the x-value to an int
      // and draw the next horizontal
      // line segment

      int x = getTheX(xVal+xCalcInc/2);
```

```
      g.drawLine(oldX,yVal,x,yVal);

      //Draw a vertical line at the
      // old x-value
      int yZero = getTheY(0);
      g.drawLine(oldX,yZero,oldX,yVal);

      //Draw a vertical line at the
      // new y-value
      g.drawLine(x,yZero,x,yVal);

      //Increment along the x-axis
      xVal += xCalcInc;

      //Save end point to use as start
      // point for next line segment.
      oldX = x;
      oldY = yVal;
    }//end while loop

  }//end overridden paint method
  //--------------------------------//

  //Method to draw axes with tic marks
  // and labels in the color RED
  void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Lable left x-axis and bottom
    // y-axis.  These are the easy
    // ones.  Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                 getTheX(xMin),
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                        getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis.  These are the hard
    // ones because the position must
    // be adjusted by the font size and
    // the number of characters.
    //Get the width of the string for
    // right end of x-axis and the
    // height of the string for top of
    // y-axis
    //Create a string that is an
    // integer representation of the
    // label for the right end of the
    // x-axis.  Then get a character
    // array that represents the
    // string.
    int xMaxInt = (int)xMax;
```

```
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.
                        toCharArray();

    //Get a FontMetrics object that can
    // be used to get the size of the
    // string in pixels.
    FontMetrics fontMetrics =
                    g.getFontMetrics();
    //Get a bounding rectangle for the
    // string
    Rectangle2D r2d =
           fontMetrics.getStringBounds(
               array,0,array.length,g);
    //Get the width and the height of
    // the bounding rectangle.  The
    // width is the width of the label
    // at the right end of the
    // x-axis.  The height applies to
    // all the labels, but is needed
    // specifically for the label at
    // the top end of the y-axis.
    int labWidth =
                  (int)(r2d.getWidth());
    int labHeight =
                (int)(r2d.getHeight());

    //Label the positive x-axis and the
    // positive y-axis using the width
    // and height from above to
    // position the labels.  These
    // labels apply to the very ends of
    // the axes at the edge of the
    // plotting surface.
    g.drawString("" + (int)xMax,
                getTheX(xMax)-labWidth,
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMax,
              getTheX(yTicLen/2)+2,
              getTheY(yMax)+labHeight);

    //Draw the axes
    g.drawLine(getTheX(xMin),
                         getTheY(0.0),
                         getTheX(xMax),
                         getTheY(0.0));

    g.drawLine(getTheX(0.0),
                        getTheY(yMin),
                        getTheX(0.0),
                        getTheY(yMax));

    //Draw the tic marks on axes
    xTics(g);
    yTics(g);
  }//end drawAxes
```

```
  //---------------------------------//

  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
                   getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small.  In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while

  }//end xTics
  //---------------------------------//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
```

```
    while(yDoub < yMax){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics

  //--------------------------------//

  //This method translates and scales
  // a double y value to plot properly
  // in the integer coordinate system.
  // In addition to scaling, it causes
  // the positive direction of the
  // y-axis to be from bottom to top.
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //--------------------------------//

  //This method scales a double x value
  // to plot properly in the integer
  // coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //--------------------------------//

}//end inner class MyCanvas
//===================================//

}//end class GUI
//===================================//

//Sample test class.  Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 4;
  }//end getNmbr
```

```
  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    return (x*x)/200.0;
  }//end f3

  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
    return 100*Math.sin(x/20.0);
  }//end f5

}//end sample class junk
```

Listing 18

```
/* File Dsp028.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04

Computes and displays the magnitude of the
spectral content for up to five sinusoids having
different frequencies and amplitudes.

Gets input parameters from a file named
Dsp028.txt.  If that file doesn't exist in the
current directory, the program uses a set of
default parameters.

Each parameter value must be stored as characters
on a separate line in the file named Dsp028.txt.
The required parameters are as follows:

Data length as type int
Sample number representing zero time as type int
Lower frequency bound as type double (See note)
Upper frequency bound as type double
Number of spectra as type int.  Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

The number of values in each of the lists must
match the value for the number of spectra.
```

```
Note:  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.  Don't
allow blank lines at the end of the data in the
file.

400
0
0.0
1.0
5
0.1
0.2
0.3
0.4
0.45
60
70
80
90
100

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 Dsp028

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 Dsp028 implements GraphIntfc01{
  final double pi = Math.PI;//for simplification

  //Begin default parameters
  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 = 1.0;
  int numberSpectra = 5;
  //Frequencies of the sinusoids
  double[] freq = {0.1,0.2,0.3,0.4,0.5};
  //Amplitudes of the sinusoids
  double[] amp = {60,70,80,90,100};
  //End default parameters

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

  //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[] magnitude1;
  double[] magnitude2;
  double[] magnitude3;
  double[] magnitude4;
  double[] magnitude5;

  public Dsp028(){//constructor

    //Get the parameters from a file named
    // Dsp028.txt.  Use the default parameters
    // if the file doesn't exist in the current
    // directory.
    if(new File("Dsp028.txt").exists()){
      getParameters();
    }//end if

    //Note that this program always processes
    // five sinusoids, even if fewer than five
    // were requested as the input parameter
    // for numberSpectra.  In that case, the
    // extras are processed using default values
    // and simply ignored when the results are
    // plotted.

    //Create the raw data.  Note that the
    // argument for a sinusoid at half the
    // sampling frequency would be (2*pi*x*0.5).
```

```
    // This would represent one half cycle or pi
    // radians per sample.
    //First create empty array objects.
    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];
    //Now populate the array objects
    for(int n = 0;n < len;n++){
      data1[n] = amp[0]*Math.cos(2*pi*n*freq[0]);
      data2[n] = amp[1]*Math.cos(2*pi*n*freq[1]);
      data3[n] = amp[2]*Math.cos(2*pi*n*freq[2]);
      data4[n] = amp[3]*Math.cos(2*pi*n*freq[3]);
      data5[n] = amp[4]*Math.cos(2*pi*n*freq[4]);
    }//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.
    magnitude1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1,real,
      imag,angle,magnitude1,zeroTime,lowF,highF);

    magnitude2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2,real,
      imag,angle,magnitude2,zeroTime,lowF,highF);

    magnitude3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3,real,
      imag,angle,magnitude3,zeroTime,lowF,highF);

    magnitude4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data4,real,
      imag,angle,magnitude4,zeroTime,lowF,highF);

    magnitude5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5,real,
      imag,angle,magnitude5,zeroTime,lowF,highF);
```

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

  //This method gets processing parameters from
  // a file named Dsp028.txt and stores those
  // parameters in instance variables belonging
  // to the object of type Dsp028.
  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(
                                    "Dsp028.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;
    len = (int)Double.parseDouble(data[cnt++]);
    zeroTime = (int)Double.parseDouble(
                                      data[cnt++]);
    lowF = Double.parseDouble(data[cnt++]);
    highF = Double.parseDouble(data[cnt++]);
    numberSpectra = (int)Double.parseDouble(
                                      data[cnt++]);
    for(int fCnt = 0;fCnt < numberSpectra;
                                          fCnt++){
      freq[fCnt] = Double.parseDouble(
                                      data[cnt++]);
    }//end for loop
    for(int aCnt = 0;aCnt < numberSpectra;
                                          aCnt++){
      amp[aCnt] = Double.parseDouble(
                                      data[cnt++]);
    }//end for loop

    //Print parameter values.
    System.out.println();
    System.out.println("Data length: " + len);
```

```
    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(
            "Number spectra: " + numberSpectra);
    System.out.println("Frequencies");
    for(cnt = 0;cnt < numberSpectra;cnt++){
      System.out.println(freq[cnt]);
    }//end for loop
    System.out.println("Amplitudes");
    for(cnt = 0;cnt < numberSpectra;cnt++){
      System.out.println(amp[cnt]);
    }//end for loop

  }//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 numberSpectra;
  }//end getNmbr
  //----------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude1.length-1){
      return 0;
    }else{
      return magnitude1[index];
    }//end else
  }//end function
  //----------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude2.length-1){
      return 0;
    }else{
      return magnitude2[index];
    }//end else
  }//end function
  //----------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude3.length-1){
      return 0;
    }else{
      return magnitude3[index];
```

```
    }//end else
  }//end function
  //----------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude4.length-1){
      return 0;
    }else{
      return magnitude4[index];
    }//end else
  }//end function
  //----------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude5.length-1){
      return 0;
    }else{
      return magnitude5[index];
    }//end else
  }//end function
  //----------------------------------------//

}//end class Dsp028
```

Listing 19

```
/* File Graph03.java
Copyright 2002, R.G.Baldwin

This program is very similar to Graph01
except that it has been modified to
allow the user to manually resize and
replot the frame.

Note:  This program requires access to
the interface named GraphIntfc01.

This is a plotting program.  It is
designed to access a class file, which
implements GraphIntfc01, and to plot up
to five functions defined in that class
file.  The plotting surface is divided
into the required number of equally
sized plotting areas, and one function
is plotted on cartesian coordinates in
each area.

The methods corresponding to the
functions are named f1, f2, f3, f4,
and f5.
```

The class containing the functions must also define a method named getNmbr(), which takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a NoSuchMethodException will be thrown.

Note that the constructor for the class that implements GraphIntfc01 must not require any parameters due to the use of the newInstance method of the Class class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with f5 and work down toward f1. For example, if the number of functions is 3, then the program will expect to call methods named f1, f2, and f3. It is OK for the absent methods to be defined in the class. They simply won't be invoked.

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

The cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named

```
junk and plots the data provided by
that class.  Thus, you can test the
program by running it with no
command-line parameter.

This program provides the following
text fields for user input, along with
a button labeled Graph.  This allows
the user to adjust the parameters and
replot the graph as many times with as
many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these
parameters and then click the Graph
button to cause the five functions
to be re-plotted according to the
new parameters.

Whenever the Graph button is clicked,
the event handler instantiates a new
object of the class that implements
the GraphIntfc01 interface.  Depending
on the nature of that class, this may
be redundant in some cases.  However,
it is useful in those cases where it
is necessary to refresh the values of
instance variables defined in the
class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were
first defined in the Color class of
v1.4.0.  Therefore, the program
requires v1.4.0 or later to compile and
run correctly.
**************************************/

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph03{
  public static void main(
          String[] args)
          throws NoSuchMethodException,
```

```
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{
    if(args.length == 1){
      //pass command-line paramater
      new GUI(args[0]);
    }else{
      //no command-line parameter given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph03 definition
//===================================//

class GUI extends JFrame
             implements ActionListener{

  //Define plotting parameters and
  // their default values.
  double xMin = 0.0;
  double xMax = 400.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 20.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small
  // on x-axis, a default value is
  // used later.
  double xTicLen = (yMax-yMin)/50;
  double yTicLen = (xMax-xMin)/50;

  //Calculation interval along x-axis
  double xCalcInc = 1.0;

  //Text fields for plotting parameters
  JTextField xMinTxt =
             new JTextField("" + xMin);
  JTextField xMaxTxt =
             new JTextField("" + xMax);
  JTextField yMinTxt =
             new JTextField("" + yMin);
  JTextField yMaxTxt =
             new JTextField("" + yMax);
  JTextField xTicIntTxt =
           new JTextField("" + xTicInt);
  JTextField yTicIntTxt =
           new JTextField("" + yTicInt);
  JTextField xCalcIncTxt =
          new JTextField("" + xCalcInc);

  //Panels to contain a label and a
  // text field
  JPanel pan0 = new JPanel();
```

```
  JPanel pan1 = new JPanel();
  JPanel pan2 = new JPanel();
  JPanel pan3 = new JPanel();
  JPanel pan4 = new JPanel();
  JPanel pan5 = new JPanel();
  JPanel pan6 = new JPanel();

  //Misc instance variables
  int frmWidth = 408;
  int frmHeight = 430;
  int width;
  int height;
  int number;
  GraphIntfc01 data;
  String args = null;

  //Plots are drawn on the canvases
  // in this array.
  Canvas[] canvases;

  //Constructor
  GUI(String args)throws
                NoSuchMethodException,
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{

    if(args != null){
      //Save for use later in the
      // ActionEvent handler
      this.args = args;
      //Instantiate an object of the
      // target class using the String
      // name of the class.
      data = (GraphIntfc01)
                   Class.forName(args).
                         newInstance();
    }else{
      //Instantiate an object of the
      // test class named junk.
      data = new junk();
    }//end else

    //Create array to hold correct
    // number of Canvas objects.
    canvases =
            new Canvas[data.getNmbr()];

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
      throw new NoSuchMethodException(
                "Too many functions.  "
                  + "Only 5 allowed.");
    }//end if
```

```
    //Create the control panel and
    // give it a border for cosmetics.
    JPanel ctlPnl = new JPanel();
    ctlPnl.setLayout(//?rows x 4 cols
                  new GridLayout(0,4));
    ctlPnl.setBorder(
                   new EtchedBorder());

    //Button for replotting the graph
    JButton graphBtn =
                  new JButton("Graph");
    graphBtn.addActionListener(this);

    //Populate each panel with a label
    // and a text field.  Will place
    // these panels in a grid on the
    // control panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);

    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);

    //Add the populated panels and the
    // button to the control panel with
    // a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);

    //Create a panel to contain the
    // Canvas objects.  They will be
    // displayed in a one-column grid.
    JPanel canvasPanel = new JPanel();
    canvasPanel.setLayout(//?rows,1 col
```

```
                new GridLayout(0,1));

    //Create a custom Canvas object for
    // each function to be plotted and
    // add them to the one-column grid.
    // Make background colors alternate
    // between white and gray.
    for(int cnt = 0;
                cnt < number; cnt++){
      switch(cnt){
        case 0 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                        Color.WHITE);
          break;
        case 1 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                    Color.LIGHT_GRAY);
          break;
        case 2 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                        Color.WHITE);
          break;
        case 3 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                    Color.LIGHT_GRAY);
          break;
        case 4 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].
            setBackground(Color.WHITE);
      }//end switch
      //Add the object to the grid.
      canvasPanel.add(canvases[cnt]);
    }//end for loop

    //Add the sub-assemblies to the
    // frame.  Set its location, size,
    // and title, and make it visible.
    getContentPane().
                  add(ctlPnl,"South");
    getContentPane().
             add(canvasPanel,"Center");

    setBounds(0,0,frmWidth,frmHeight);

    if(args == null){
      setTitle("Graph03, " +
```

```
                "Copyright 2002, " +
                "Richard G. Baldwin");
  }else{
    setTitle("Graph03/" + args +
                " Copyright 2002, " +
                "R. G. Baldwin");
  }//end else

  setVisible(true);

  //Set to exit on X-button click
  setDefaultCloseOperation(
                    EXIT_ON_CLOSE);

  //Guarantee a repaint on startup.
  for(int cnt = 0;
                cnt < number; cnt++){
    canvases[cnt].repaint();
  }//end for loop

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

//This event handler is registered
// on the JButton to cause the
// functions to be replotted.
public void actionPerformed(
                    ActionEvent evt){
  //Re-instantiate the object that
  // provides the data
  try{
    if(args != null){
      data = (GraphIntfc01)Class.
         forName(args).newInstance();
    }else{
      data = new junk();
    }//end else
  }catch(Exception e){
    //Known to be safe at this point.
    // Otherwise would have aborted
    // earlier.
  }//end catch

  //Set plotting parameters using
  // data from the text fields.
  xMin = Double.parseDouble(
                  xMinTxt.getText());
  xMax = Double.parseDouble(
                  xMaxTxt.getText());
  yMin = Double.parseDouble(
                  yMinTxt.getText());
  yMax = Double.parseDouble(
                  yMaxTxt.getText());
  xTicInt = Double.parseDouble(
                xTicIntTxt.getText());
  yTicInt = Double.parseDouble(
```

```
                yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(
                xCalcIncTxt.getText());

    //Calculate new values for the
    // length of the tic marks on the
    // axes.  If too small on x-axis,
    // a default value is used later.
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting areas
    for(int cnt = 0;
                  cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end actionPerformed
  //--------------------------------//


//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
  int cnt;//object number
  //Factors to convert from double
  // values to integer pixel locations.
  double xScale;
  double yScale;

  MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
  }//end constructor

  //Override the paint method
  public void paint(Graphics g){

    //Get and save the size of the
    // plotting surface
    width = canvases[0].getWidth();
    height = canvases[0].getHeight();

    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);

    //Get initial data values
    double xVal = xMin;
```

```
    int oldX = getTheX(xVal);
    int oldY = 0;
    //Use the Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
      case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
      case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
      case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
      case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
      case 4 :
        oldY = getTheY(data.f5(xVal));
    }//end switch

    //Now loop and plot the points
    while(xVal < xMax){
      int yVal = 0;
      //Get next data value.  Use the
      // Canvas obj number to
      // determine which method to
      // invoke to get the value for y.
      switch(cnt){
        case 0 :
          yVal =
                getTheY(data.f1(xVal));
          break;
        case 1 :
          yVal =
                getTheY(data.f2(xVal));
          break;
        case 2 :
          yVal =
                getTheY(data.f3(xVal));
          break;
        case 3 :
          yVal =
                getTheY(data.f4(xVal));
          break;
        case 4 :
          yVal =
                getTheY(data.f5(xVal));
      }//end switch1

      //Convert the x-value to an int
      // and draw the next line segment
      int x = getTheX(xVal);
      g.drawLine(oldX,oldY,x,yVal);
```

```
      //Increment along the x-axis
      xVal += xCalcInc;

      //Save end point to use as start
      // point for next line segment.
      oldX = x;
      oldY = yVal;
    }//end while loop

  }//end overridden paint method
  //--------------------------------//

  //Method to draw axes with tic marks
  // and labels in the color RED
  void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Lable left x-axis and bottom
    // y-axis.  These are the easy
    // ones.  Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                 getTheX(xMin),
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                        getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis.  These are the hard
    // ones because the position must
    // be adjusted by the font size and
    // the number of characters.
    //Get the width of the string for
    // right end of x-axis and the
    // height of the string for top of
    // y-axis
    //Create a string that is an
    // integer representation of the
    // label for the right end of the
    // x-axis.  Then get a character
    // array that represents the
    // string.
    int xMaxInt = (int)xMax;
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.
                        toCharArray();

    //Get a FontMetrics object that can
    // be used to get the size of the
    // string in pixels.
    FontMetrics fontMetrics =
                    g.getFontMetrics();
    //Get a bounding rectangle for the
    // string
```

```
    Rectangle2D r2d =
           fontMetrics.getStringBounds(
               array,0,array.length,g);
    //Get the width and the height of
    // the bounding rectangle.  The
    // width is the width of the label
    // at the right end of the
    // x-axis.  The height applies to
    // all the labels, but is needed
    // specifically for the label at
    // the top end of the y-axis.
    int labWidth =
                 (int)(r2d.getWidth());
    int labHeight =
                (int)(r2d.getHeight());

    //Label the positive x-axis and the
    // positive y-axis using the width
    // and height from above to
    // position the labels.  These
    // labels apply to the very ends of
    // the axes at the edge of the
    // plotting surface.
    g.drawString("" + (int)xMax,
                getTheX(xMax)-labWidth,
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMax,
              getTheX(yTicLen/2)+2,
              getTheY(yMax)+labHeight);

    //Draw the axes
    g.drawLine(getTheX(xMin),
                         getTheY(0.0),
                         getTheX(xMax),
                         getTheY(0.0));

    g.drawLine(getTheX(0.0),
                        getTheY(yMin),
                        getTheX(0.0),
                        getTheY(yMax));

    //Draw the tic marks on axes
    xTics(g);
    yTics(g);
  }//end drawAxes

  //--------------------------------//

  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
```

```
                    getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small.  In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while

  }//end xTics
  //---------------------------------//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
      y = getTheY(yDoub);
```

```
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics

  //---------------------------------//

  //This method translates and scales
  // a double y value to plot properly
  // in the integer coordinate system.
  // In addition to scaling, it causes
  // the positive direction of the
  // y-axis to be from bottom to top.
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //---------------------------------//

  //This method scales a double x value
  // to plot properly in the integer
  // coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //---------------------------------//

}//end inner class MyCanvas
//===================================//

}//end class GUI
//===================================//

//Sample test class.  Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 4;
  }//end getNmbr

  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    return (x*x)/200.0;
  }//end f3
```

```
  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
    return 100*Math.sin(x/20.0);
  }//end f5

}//end sample class junk
```

Listing 20

```
/*File ForwardRealToComplex01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04

The static method named transform performs a real
to complex Fourier transform.

Does not implement the FFT algorithm. Implements
a straight-forward sampled-data version of the
continuous Fourier transform defined using
integral calculus.  See ForwardRealToComplexFFT01
for an FFT algorithm.

Returns real, imag, magnitude, and phase angle in
degrees.

Incoming parameters are:
  double[] data - incoming real data
  double[] realOut - outgoing real data
  double[] imagOut - outgoing imaginary data
  double[] angleOut - outgoing phase angle in
    degrees
  double[] magnitude - outgoing amplitude
    spectrum
  int zero - the index of the incoming data
    sample that represents zero time
  double lowF - Low freq limit as fraction of
    sampling frequency
  double highF - High freq limit as fraction of
    sampling frequency

The frequency increment is the difference between
high and low limits divided by the length of
the magnitude array

The magnitude is computed as the square root of
the sum of the squares of the real and imaginary
parts.  This value is divided by the incoming
data length, which is given by data.length.
```

```
Returns a number of points in the frequency
domain equal to the incoming data length
regardless of the high and low frequency
limits.
*********************************************/

public class ForwardRealToComplex01{

  public static void transform(
                              double[] data,
                              double[] realOut,
                              double[] imagOut,
                              double[] angleOut,
                              double[] magnitude,
                              int zero,
                              double lowF,
                              double highF){
    double pi = Math.PI;//for convenience
    int dataLen = data.length;
    double delF = (highF-lowF)/data.length;
    //Outer loop iterates on frequency
    // values.
    for(int i=0; i < dataLen;i++){
      double freq = lowF + i*delF;
      double real = 0.0;
      double imag = 0.0;
      double ang = 0.0;
      //Inner loop iterates on time-
      // series points.
      for(int j=0; j < dataLen; j++){
        real += data[j]*Math.cos(
                             2*pi*freq*(j-zero));
        imag += data[j]*Math.sin(
                             2*pi*freq*(j-zero));
      }//end inner loop
      realOut[i] = real/dataLen;
      imagOut[i] = imag/dataLen;
      magnitude[i] = (Math.sqrt(
                 real*real + imag*imag))/dataLen;

      //Calculate and return the phase
      // angle in degrees.
      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;
    }//end outer loop
  }//end transform method
```

```
}//end class ForwardRealToComplex01
```

Listing 21

```
/* File Dsp030.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04

Uses an FFT algorithm to compute and display the
magnitude of the spectral content for up to five
sinusoids having different frequencies and
amplitudes.

See the program named Dsp028 for a program that
does not use an FFT algorithm.

Gets input parameters from a file named
Dsp030.txt.  If that file doesn't exist in the
current directory, the program uses a set of
default parameters.

Each parameter value must be stored as characters
on a separate line in the file named Dsp030.txt.
The required parameters are as follows:

Data length as type int
Number of spectra as type int.  Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

CAUTION: THE DATA LENGTH MUST BE A POWER OF TWO.
OTHERWISE, THIS PROGRAM WILL FAIL TO RUN
PROPERLY.

The number of values in each of the lists must
match the value for the number of spectra.

Note:  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.  Don't
allow blank lines at the end of the data in the
file.

128.0
5
0.1
0.2
```

```
0.3
0.4
0.45
60
70
80
90
100

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 Dsp030

A static method named transform belonging to the
class named ForwardRealToComplexFFT01 is used to
perform the actual spectral analysis.  The method
named transform implements an FFT algorithm.  The
FFT algorithm requires that the data length be
a power of two.

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

class Dsp030 implements GraphIntfc01{
  final double pi = Math.PI;//for simplification

  //Begin default parameters
  int len = 128;//data length
  int numberSpectra = 5;
  //Frequencies of the sinusoids
  double[] freq = {0.1,0.2,0.3,0.4,0.5};
  //Amplitudes of the sinusoids
  double[] amp = {60,70,80,90,100};
  //End default parameters

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

  //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[] magnitude1;
double[] magnitude2;
double[] magnitude3;
double[] magnitude4;
double[] magnitude5;

public Dsp030(){//constructor

  //Get the parameters from a file named
  // Dsp030.txt.  Use the default parameters
  // if the file doesn't exist in the current
  // directory.
  if(new File("Dsp030.txt").exists()){
    getParameters();
  }//end if

  //Note that this program always processes
  // five sinusoids, even if fewer than five
  // were requested as the input parameter
  // for numberSpectra.  In that case, the
  // extras are processed using default values
  // and simply ignored when the results are
  // plotted.

  //Create the raw data.  Note that the
  // argument for a sinusoid at half the
  // sampling frequency would be (2*pi*x*0.5).
  // This would represent one half cycle or pi
  // radians per sample.
  //First create empty array objects.
  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];
  //Now populate the array objects
  for(int n = 0;n < len;n++){
    data1[n] = amp[0]*Math.cos(2*pi*n*freq[0]);
    data2[n] = amp[1]*Math.cos(2*pi*n*freq[1]);
    data3[n] = amp[2]*Math.cos(2*pi*n*freq[2]);
    data4[n] = amp[3]*Math.cos(2*pi*n*freq[3]);
    data5[n] = amp[4]*Math.cos(2*pi*n*freq[4]);
  }//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.
  magnitude1 = new double[len];
  real = new double[len];
  imag = new double[len];
  angle = new double[len];
```

```
    ForwardRealToComplexFFT01.transform(data1,
                      real,imag,angle,magnitude1);

    magnitude2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplexFFT01.transform(data2,
                      real,imag,angle,magnitude2);

    magnitude3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplexFFT01.transform(data3,
                      real,imag,angle,magnitude3);

    magnitude4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplexFFT01.transform(data4,
                      real,imag,angle,magnitude4);

    magnitude5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplexFFT01.transform(data5,
                      real,imag,angle,magnitude5);
  }//end constructor
  //----------------------------------------//

  //This method gets processing parameters from
  // a file named Dsp030.txt and stores those
  // parameters in instance variables belonging
  // to the object of type Dsp030.
  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(
                                    "Dsp030.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;
    len = (int)Double.parseDouble(data[cnt++]);
    numberSpectra = (int)Double.parseDouble(
                                      data[cnt++]);
    for(int fCnt = 0;fCnt < numberSpectra;
                                          fCnt++){
      freq[fCnt] = Double.parseDouble(
                                      data[cnt++]);
    }//end for loop
    for(int aCnt = 0;aCnt < numberSpectra;
                                          aCnt++){
      amp[aCnt] = Double.parseDouble(
                                      data[cnt++]);
    }//end for loop

    //Print parameter values.
    System.out.println();
    System.out.println("Data length: " + len);
    System.out.println(
             "Number spectra: " + numberSpectra);
    System.out.println("Frequencies");
    for(cnt = 0;cnt < numberSpectra;cnt++){
      System.out.println(freq[cnt]);
    }//end for loop
    System.out.println("Amplitudes");
    for(cnt = 0;cnt < numberSpectra;cnt++){
      System.out.println(amp[cnt]);
    }//end for loop

  }//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 numberSpectra;
  }//end getNmbr
  //----------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude1.length-1){
      return 0;
    }else{
      return magnitude1[index];
```

```
    }//end else
  }//end function
  //-----------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude2.length-1){
      return 0;
    }else{
      return magnitude2[index];
    }//end else
  }//end function
  //-----------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude3.length-1){
      return 0;
    }else{
      return magnitude3[index];
    }//end else
  }//end function
  //-----------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude4.length-1){
      return 0;
    }else{
      return magnitude4[index];
    }//end else
  }//end function
  //-----------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                index > magnitude5.length-1){
      return 0;
    }else{
      return magnitude5[index];
    }//end else
  }//end function
  //-----------------------------------------//

}//end class Dsp030
```

Listing 22

```
/*File ForwardRealToComplexFFT01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04
```

```
The static method named transform performs a real
to complex Fourier transform using a crippled
complex-to-complex FFT algorithm.  It is crippled
in the sense that it is not being used to its
full potential as a complex-to-complex forward
or inverse FFT algorithm.

See ForwardRealToComplex01 for a slower but more
general approach that does not use an FFT
algorithm.

Returns real, imag, magnitude, and phase angle in
degrees.

Incoming parameters are:
  double[] data - incoming real data
  double[] realOut - outgoing real data
  double[] imagOut - outgoing imaginary data
  double[] angleOut - outgoing phase angle in
    degrees
  double[] magnitude - outgoing amplitude
    spectrum

The magnitude is computed as the square root of
 the sum of the squares of the real and imaginary
 parts.  This value is divided by the incoming
 data length, which is given by data.length.

CAUTION: THE INCOMING DATA LENGTH MUST BE A
POWER OF TWO. OTHERWISE, THIS PROGRAM WILL FAIL
TO RUN PROPERLY.

Returns a number of points in the frequency
domain equal to the incoming data length.  Those
points are uniformly distributed between zero and
one less than the sampling frequency.
*********************************************/

public class ForwardRealToComplexFFT01{

  public static void transform(
                              double[] data,
                              double[] realOut,
                              double[] imagOut,
                              double[] angleOut,
                              double[] magnitude){
    double pi = Math.PI;//for convenience
    int dataLen = data.length;
    //The complexToComplex FFT method does an
    // in-place transform causing the output
    // complex data to be stored in the arrays
    // containing the input complex data.
    // Therefore, it is necessary to copy the
    // input data to this method into the real
    // part of the complex data passed to the
    // complexToComplex method.
```

```
    System.arraycopy(data,0,realOut,0,dataLen);

    //Perform the spectral analysis.  The results
    // are stored in realOut and imagOut. The +1
    // causes it to be a forward transform. A -1
    // would cause it to be an inverse transform.
    complexToComplex(1,dataLen,realOut,imagOut);

    //Compute the magnitude and the phase angle
    // in degrees.
    for(int cnt = 0;cnt < dataLen;cnt++){
      magnitude[cnt] =
           (Math.sqrt(realOut[cnt]*realOut[cnt]
           + imagOut[cnt]*imagOut[cnt]))/dataLen;

      if(imagOut[cnt] == 0.0
                        && realOut[cnt] == 0.0){
          angleOut[cnt] = 0.0;
      }//end if
      else{
        angleOut[cnt] = Math.atan(
             imagOut[cnt]/realOut[cnt])*180.0/pi;
      }//end else

      if(realOut[cnt] < 0.0
                         && imagOut[cnt] == 0.0){
        angleOut[cnt] = 180.0;
      }else if(realOut[cnt] < 0.0
                        && imagOut[cnt] == -0.0){
        angleOut[cnt] = -180.0;
      }else if(realOut[cnt] < 0.0
                          && imagOut[cnt] > 0.0){
        angleOut[cnt] += 180.0;
      }else if(realOut[cnt] < 0.0
                          && imagOut[cnt] < 0.0){
        angleOut[cnt] += -180.0;
      }//end else

    }//end for loop

  }//end transform method
  //-----------------------------------------//

  //This method computes a complex-to-complex
  // FFT.  The value of sign must be 1 for a
  // forward FFT.
  public static void complexToComplex(
                                  int sign,
                                  int len,
                                  double real[],
                                  double imag[]){
    double scale = 1.0;
    //Reorder the input data into reverse binary
    // order.
    int i,j;
    for (i=j=0; i < len; ++i) {
```

```
      if (j>=i) {
        double tempr = real[j]*scale;
        double tempi = imag[j]*scale;
        real[j] = real[i]*scale;
        imag[j] = imag[i]*scale;
        real[i] = tempr;
        imag[i] = tempi;
      }//end if
      int m = len/2;
      while (m>=1 && j>=m) {
        j -= m;
        m /= 2;
      }//end while loop
      j += m;
    }//end for loop

    //Input data has been reordered.
    int stage = 0;
    int maxSpectraForStage,stepSize;
    //Loop once for each stage in the spectral
    // recombination process.
    for(maxSpectraForStage = 1,
                stepSize = 2*maxSpectraForStage;
                maxSpectraForStage < len;
                maxSpectraForStage = stepSize,
                stepSize = 2*maxSpectraForStage){
      double deltaAngle =
                 sign*Math.PI/maxSpectraForStage;
      //Loop once for each individual spectra
      for (int spectraCnt = 0;
                 spectraCnt < maxSpectraForStage;
                 ++spectraCnt){
        double angle = spectraCnt*deltaAngle;
        double realCorrection = Math.cos(angle);
        double imagCorrection = Math.sin(angle);

        int right = 0;
        for (int left = spectraCnt;
                   left < len;left += stepSize){
          right = left + maxSpectraForStage;
          double tempReal =
                     realCorrection*real[right]
                     - imagCorrection*imag[right];
          double tempImag =
                     realCorrection*imag[right]
                     + imagCorrection*real[right];
          real[right] = real[left]-tempReal;
          imag[right] = imag[left]-tempImag;
          real[left] += tempReal;
          imag[left] += tempImag;
        }//end for loop
      }//end for loop for individual spectra
      maxSpectraForStage = stepSize;
    }//end for loop for stages
  }//end complexToComplex method
```

```
}//end class ForwardRealToComplexFFT01
```

Listing 23

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-