Convolution and Matched Filtering in Java

Learn how to perform matched filtering in Java. Learn how matched filtering often makes it possible to detect properly designed signals in noisy environments where simple frequency filtering alone cannot do the job.

Published: March 22, 2005
By Richard G. Baldwin

Java Programming, Notes # 1488

Preface

The previous lesson entitled Convolution and Frequency Filtering in Java taught you how to use convolution to perform frequency filtering in Java.

Miscellaneous DSP topics

Other previous lessons in this series, going all the way back to the lesson entitled Plotting Engineering and Scientific Data using Java have taught you quite a lot about the use of Java for Digital Signal Processing *(DSP)*.

The following lessons have taught you about the topics indicated in the titles:

- Periodic Motion and Sinusoids
- Sampled Time Series
- Averaging Time Series

Spectral analysis

Numerous other lessons have taught you about spectral analysis. For example, several previous lessons, including the lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm, have discussed spectral analysis in detail.

Matched Filtering

This lesson will introduce you to *matched filtering.* The lesson will teach you how to use convolution in Java to perform matched filtering, and will provide examples of matched filtering using Java.

The lesson will demonstrate that the use of matched filtering often makes it possible to detect properly designed signals in noisy environments where simple frequency filtering alone cannot do the job.

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.

General Discussion

The purpose of this lesson is to help you to understand the use of convolution for *matched filtering.*

What is convolution?

I won't attempt to explain the theoretical basis for convolution. If you search the web using Google, you will undoubtedly find numerous links to theoretical discussions on convolution.

I explained the mechanics of convolution in the previous lesson entitled Convolution and Frequency Filtering in Java, so I won't repeat that discussion here.

Signals in noise

Many situations involve dealing with signals buried in noise. In some situations, such as tuning a radio or television receiver, the objective is to extract the signal from the noise so that something can be done with it, *(such as listening to the music).*

This is not a situation where matched filtering can be particularly helpful, particularly if the system is all analog.

Detection systems

In other situations, such as SONAR, RADAR, and possibly network cables, the objective is to determine the presence or absence of a signal. Determining the presence or absence usually involves *detecting* the signal.

SONAR

For example, an active SONAR system sends acoustic pulses into the water and then listens for echoes produced when the acoustic energy bounces off of something like a submarine. If the system detects an echo, that usually means that there is something in the area that is reflecting the acoustic pulses.

RADAR

Similarly, a RADAR system sends electromagnetic pulses into the air and listens for echoes from airplanes *(or speeding automobiles).* If echoes are detected, that usually means that there is something in the area that is reflecting the electromagnetic pulses.

Network cable receiver

A receiver on the end of a network cable may listen for electromagnetic or optical pulses sent by a transmitter on the other end of the cable. The fact that such pulses are detected is provided to other parts of the system capable of decoding the information that was encoded in those pulses.

Matched filtering for detection

Matched filtering is often used in situations where it is desired to detect the presence or absence of pulses of a known waveform buried in noise. I will present and explain a program in this lesson that simulates such a situation.

Frequency filtering

One of the early stages in most systems designed to extract or detect signals in noise is to apply a frequency filter to eliminate the energy at all frequencies outside the frequency band occupied by the signal. This is what you are doing when you tune your radio receiver to a particular radio station. Hopefully, you are eliminating the energy being transmitted by all other radio stations that operate on different frequencies.

If the data involved is digital data, digital convolution is one approach that can be used for such frequency filtering. I explained this approach in the previous lesson entitled Convolution and Frequency Filtering in Java.

Matched filtering goes further

Matched filtering goes further than simple frequency filtering. Although matched filtering does make use of information regarding the operating frequency of the source of the signal, it also takes advantage of the complex spectrum and phase information contained in the signal.

(I explained these concepts in the earlier lesson entitled Spectrum Analysis using Java, Complex Spectrum and Phase Angle.)

The complex spectrum and phase characteristics of a time-domain pulse manifest themselves in the waveform of the pulse. Matched filtering is a time-domain process that takes advantage of the waveform of the signal pulse. This in turn takes advantage of the complex spectral and phase information that describes the pulse.

Benefits of matched filtering

The use of matched filtering often makes it possible to detect the presence of properly designed signal pulses in noisy situations where simple frequency filtering alone cannot do the job. This will be illustrated by one of the sample programs in this lesson.

(This lesson does not go into design issues for creating a properly designed pulse that works well with matched filtering. Generally speaking, such pulses need to have a large time-bandwidth product.)

Preview

I will present and explain two programs in this lesson. The first program named **Dsp040a** illustrates an implementation of matched filtering in the absence of noise. This program is useful in showing you how matched filtering achieves the benefits that it provides.

The second program named **Dsp040b** illustrates an implementation of matched filtering for a signal pulse buried deep in noise. Simple frequency filtering is performed first to illustrate that simple frequency filtering in this case is not sufficient for detection of the signal. Then matched filtering is applied, followed by an automatic detection process. Even though the signal-to-noise ratio is fairly low, the process can usually detect the presence of the signal.

Sample Programs

The program named Graph03

The program named **Graph03** is a utility plotting program from a series of programs first discussed in the lesson entitled Plotting Engineering and Scientific Data using Java. I will refer you to that lesson for a detailed discussion of the concepts involved.

A listing of the **Graph03** program is provided in Listing 17. Similarly, a listing of a required interface named **GraphIntfc01** is provided in Listing 18.

The program named Dsp040a

This program illustrates the use of convolution for matched filtering. In particular, it is designed to show the behavior of a matched filter process in the absence of noise.

An infinite signal-to-noise ratio

The program begins by populating a 400-sample noise array with a zero for every noise value. This later results in an infinite signal-to-noise ratio when a signal pulse is added into the array.

Linear sweep FM signal pulse

Then the program creates a linear sweep FM signal pulse. The frequency sweep extends from zero to 3300 Hz at a sampling rate of 16000 samples per second. The pulse has a constant peak amplitude throughout the length of the pulse.

Signal plus noise

Then the signal pulse is added to the noise beginning at sample index 200. This results in an infinite signal-to-noise ratio because all noise values are zero.

Perform a matched filter operation

After that, the signal plus noise is convolved with a replica of the signal to implement the matched filter process.

Compute the spectral content

Then a spectral analysis is performed on the raw signal plus noise to show the frequency spectrum of the signal.

(The spectrum isn't impacted by the noise because all the noise values are zero.)

Because a replica of the signal is used as the convolution operator, the frequency spectrum of the signal is also the frequency response of the convolution operator.

Plot the results

Then the program plots the following curves:

- Signal plus noise
- Convolution operator
- Matched filter output
- Spectrum of raw signal plus noise

Miscellaneous

This program requires access to the interface named **GraphIntfc01**.

The program was tested using JDK 1. 4. 2 under WinXP.

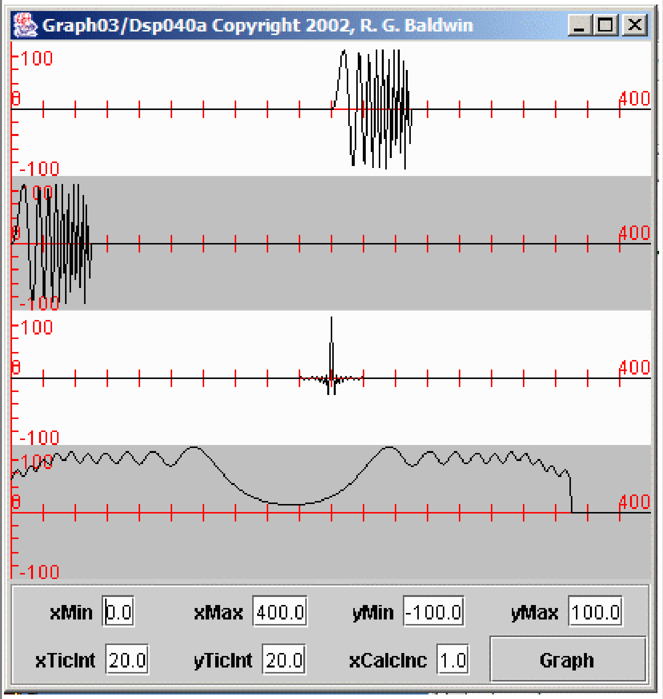
Running the program

You can run this program by entering the following command at the command-line prompt.

```
java Graph03 Dsp040a
```

The program output

The graphic output produced by this program is shown in Figure 1.



xMin 0.0 xMax 400.0 yMin -100.0 yMax 100.0

xTicInt 20.0 yTicInt 20.0 xCalcInc 1.0

Figure 1 Output from program Dsp040a

The signal pulse

The first plot down from the top in Figure 1 shows the signal pulse added to noise where all of the noise values are zero.

> *(In the next program, I will add the same signal to noise where the level of the noise is substantial. In this program, I wanted to be able to show you how matched filtering behaves without having to deal with the impact of noise on the situation.)*

The signal spectrum

The bottom curve in Figure 1 shows the spectral content of the signal from zero to the sampling frequency.

(Recall that the amplitude spectrum of a real time series is always symmetric about the folding frequency, which is half the sampling frequency. See the previous lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.*)*

The area of interest is the area on the left from zero to the folding frequency, midway between the two ends of the spectral plot. Under the assumption that the sampling frequency is 16000 samples per second, the folding frequency is at 8000 Hz.

The convolution operator

The second plot down from the top in Figure 1 shows a replica of the signal pulse. The pulse in the second plot will be used as a convolution operator. Because it is a replica of the signal for which the spectral plot was computed, the spectral plot at the bottom represents the frequency response of the convolution operator.

Pass and reject frequency bands

The pass band for this filter extends from zero to approximately 6000 Hz. The reject band extends from approximately 6000 Hz to 8000 Hz, which is the folding frequency.

(This frequency response information will be important in the next program. We will use the convolution filter to limit the spectral content of the noise to that exhibited by the signal before adding the signal to the noise. This will eliminate or greatly reduce any possibility of detecting the signal in the noise based on frequency filtering alone.)

Matched filtering through convolution

The convolution operator will be convolved with the signal plus noise in the first plot, producing the convolution output shown in the third plot. Because the convolution operator is a replica of the signal waveform, the convolution process is also a matched filter process. In other words, the convolution operator matches the complex spectral and phase *(waveform)* characteristics of the signal.

Time compression

Pay particular attention to the difference between the waveform of the signal in the first plot and the waveform of the matched filter output in the third plot.

The signal in the first plot is 50 samples in length. However, all of the energy in those 50 samples has been compressed into perhaps ten samples in the output of the matched filter in the third plot.

Furthermore, the waveform of the matched filter output in the third plot is symmetrical about its peak. Also, the peak in the matched filter output occurs at the beginning of the signal in the first plot. In other words, the matched filter output does a good job of locating the position of the signal in the first plot.

The bottom line

Convolving a properly designed signal with an operator that is a replica of the signal causes all of the energy in the signal to be time-compressed into a very narrow and very tall waveform.

> *(If plotted at the same scale, the waveform in the third plot would be much taller than the waveform in the first plot.)*

Some signal waveforms do a much better job of time compression than others. Generally, the wider the bandwidth of the signal, the narrower will be the matched filter output. The longer the signal extends in time, the taller will be the match filter output. Thus, a properly designed pulse for matched filtering usually has a large time-bandwidth product.

Benefits of time compression

Time compression can have a very beneficial effect when trying to detect pulses buried in noise. As we will demonstrate in the next program, converting the long low signal buried in the noise to a narrow tall signal buried in the same noise will often cause the signal to extend above the noise and make it detectable, when it would not be detectable otherwise.

Discussion of the code for Dsp040a

I will discuss the code in fragments. You can view the entire program in Listing 15.

The class definition for **Dsp040a** begins in Listing 1. Note that this class implements **GraphIntfc01**. This is necessary to satisfy the requirements of the utility plotting program named **Graph03**.

```
class Dsp040a implements GraphIntfc01{

  int signalLen = 50;
  int noiseLen = 400;
  int outputLen = noiseLen -
signalLen;
  int spectrumPts = outputLen;

  double[] noise = new
double[noiseLen];
  double[] signal = new
double[signalLen];
  double[] output = new
double[outputLen];
  double[] spectrumA = new
double[spectrumPts];
```

Listing 1

Listing 1 establishes the lengths for several arrays, and then creates the array objects. These array objects are used later to store different kinds of data, generally indicated by the names of the references to the array objects.

Note that the elements in all four of these arrays, and particularly the array referred to by **noise**, are initialized to their default values of 0.0.

The constructor

The constructor begins in Listing 2.

```
  public Dsp040a(){//constructor

    fmSweep(signalLen,signal);
```

Listing 2

The fmSweep method

The code in Listing 2 invokes the method named **fmSweep** to create a linear FM sweep signal where the frequency sweep extends from zero to 3300 Hz at a sampling rate of 16000 samples per second. The peak amplitude of the pulse is the same from the beginning to the end.

I explained this method in detail in the previous lesson entitled Convolution and Frequency Filtering in Java. Therefore, I won't discuss it further in this lesson. You can view the method in Listing 15.

A convolution operator

As described earlier, in addition to being used as the signal, this pulse will also be used as a convolution operator to perform a matched filter operation on the signal plus noise.

The convolution operator is shown in the second plot in Figure 1.

Add the signal to the noise

Listing 3 adds the signal into the empty noise array, beginning at sample index 200. The result is shown in the first plot in Figure 1.

```
    for(int cnt = 0; cnt < signalLen;
cnt++){
      noise[cnt + 200] += signal[cnt];
    }//end for loop
```

Listing 3

Because all of the noise values in the array were initialized to zero, this results in an infinite signal-to-noise ratio.

Perform the convolution

Listing 4 invokes the static **convolve** method of the **Convolve01** class to apply the convolution operator in the second plot in Figure 1 to the signal plus noise in the first plot in Figure 1.

```
Convolve01.convolve(noise,signal,output);
```

Listing 4

Because the noise values are all zero, this amounts to applying the convolution operator to the signal alone. The convolution operator is a replica of the signal. Thus, this operation applies a matched filter to the signal, producing the time-compressed output shown in the third plot in Figure 1.

I explained the **convolve** method in detail in the previous lesson entitled Convolution and Frequency Filtering in Java so I won't discuss it further here. You can view the class named **Convolve01** in Listing 15.

Compute the frequency response of the convolution filter

Listing 5 invokes the static **dft** method of the **Dft01** class to compute the spectrum of the signal plus noise shown in the first plot in Figure 1.

```
Dft01.dft(noise,spectrumPts,spectrumA);

  }//end constructor
```

Listing 5

Because the noise values are all zero, and because the convolution operator is a replica of the signal pulse, this also computes the frequency response of the convolution operator. The result is shown in the bottom plot in Figure 1.

I have explained the method named **dft** in several previous lessons, so I won't discuss it further here. You can view the **Dft01** class in Listing 15.

End of the constructor

Listing 5 also signals the end of the constructor.

At this point, the data for all four plots has been computed and saved in the arrays declared early in the program. That data can be accessed and plotted at this time.

Plotting the data

The program named **Graph03** invokes the methods named **f1**, **f2**, **f3**, and **f4** to access the data for plotting.

> *(You can view the program named **Graph03** in Listing 17.)*

I explained a program very similar to **Graph 03** and the methods named **f1** through **f5** in the lesson entitled Plotting Engineering and Scientific Data using Java. Therefore, I won't repeat that explanation here. You can view the methods in Listing 15. You can view the program named **Graph03** in Listing 17, and you can view the interface named **GraphIntfc01**, which declares the methods, in Listing 18.

That concludes the discussion of the program named **Dsp040a**.

The program named Dsp040b

Now for something a little more exciting. This program illustrates the use of convolution for matched filtering in the presence of noise. The program buries a signal in noise having the same bandwidth as the signal. Then it uses a matched filter to detect that signal.

Changing the signal-to-noise ratio

The program is designed to make it easy to experiment with signal detectability versus signal-to-noise ratio. You can determine the success or lack thereof in detecting a signal at a different signal-to-noise ratio by changing the value of one variable named **SNR** and then recompiling and rerunning the program.

Generate some white noise

The program begins by creating 400 samples of white noise using a random noise generator.

> *(I discussed the concept of white noise in detail in the previous* lesson entitled Convolution and Frequency Filtering in Java.*)*

Create a signal pulse

Then the program creates a linear sweep FM signal pulse with the frequency range extending from zero to 3300 Hz at a sampling rate of 16000 samples per second. The pulse has a constant peak value throughout the length of the pulse.

Shaping the noise spectrum

Following this, the program uses the signal pulse as a convolution operator to filter the white noise and to eliminate noise energy that is outside the frequency band of the signal. This is done so that most of the improvement in signal detectability that is achieved later through matched filtering will be due solely to matched filtering. Little or none of the improvement will be due to simple frequency filtering.

Adjusting the noise amplitude

After the noise has been filtered for bandwidth control, it is scaled to a peak value of 1.0. This is done so that it will be possible to establish a definitive peak signal to peak noise ratio later.

Combining the signal and the noise

Then the signal pulse is scaled to a peak value of 0.35 and added to the scaled noise at sample index 200. This results in a peak signal to peak noise ratio of 0.35.

Adjusting the signal-to-noise ratio

The scale factor named **SNR** that is applied to the signal at this point can be modified to increase or decrease the signal-to-noise ratio. This is useful for experimenting with signal detectability versus signal-to-noise ratio.

Perform the matched filter process

After the signal is added to the noise, the signal plus noise data is convolved with a replica of the signal to perform the matched filter process.

Automatic peak detection

A peak detector is then applied to the filtered signal plus noise to automatically find the peak value in the matched filter output. The location and size of the peak is displayed on the screen. The location of the peak is marked in an empty array, which will later be displayed below a plot of the filtered signal plus noise.

Plot the results

Then the program plots the following curves as shown in Figure 2:

- Raw signal plus noise
- Convolution operator
- Output from matched filter process
- Peak marker indicating the location of the highest positive peak in the output from the matched filter process

Miscellaneous

This program requires access to the interface named **GraphIntfc01**.

The program was tested using JDK 1.4.2 under WinXP.

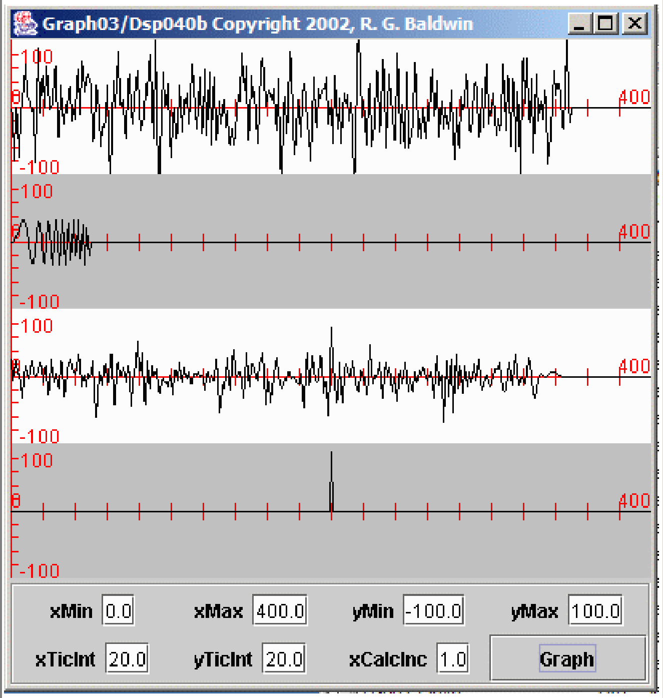
Running the program

You can run this program by entering the following command at the command-line prompt:

java Graph03 Dsp040b

The program output

The graphic output produced by this program is shown in Figure 2.



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

```
Figure 2 Output for program Dsp040b
```

The signal plus noise data

The first plot from the top in Figure 2 shows the signal plus noise data. The second plot in Figure 2 is a replica of the signal that was added to the noise to produce the data in the first plot.

> *(Note that the first two plots in Figure 2 were plotted at the same scale. Therefore, the amplitude of the signal replica in the second plot is correct relative to the amplitude of the noise in the first plot.)*

The signal was added to the noise beginning at sample index 200, which corresponds to the tenth tick mark counting from the left.

As you can see, the signal is completely lost in the noise.

Frequency filtering won't help much

Recall that the spectral content of the noise was preconditioned to cause it to match the spectral content of the signal. Therefore, there is little if any improvement in signal detectability that can be achieved at this point through additional simple frequency filtering. I will explain an easy way to demonstrate this later.

Looks like a job for a matched filter

The convolution operator shown in the second plot in Figure 2 was applied to the signal plus noise shown in the first plot, producing the output shown in the third plot. In other words, the third plot is the output from the matched filter.

> *(The plotting scale factor that was applied to the data in the matched filter output in the third plot was smaller by a factor of four than the scale factor that was applied to the first two plots. In other words, if the same scale factor had been used on the third plot, as was used for the first two plots, the peaks in the third plot would be four times larger. This would cause many of those peaks to extend completely out of the plotting area allocated to the third plot. Thus, the important thing to note is the signal-to-noise ratio in the third plot as compared to the signal-to-noise ratio in the first plot.)*

The tallest peak

What we are interested in is the tallest peak in the third plot. As you can see, this peak occurs at the tenth tick mark. This is the location of the beginning of the signal in the first plot. Therefore, this peak was produced by applying the matched filter to the signal that was buried in the noise in the first plot. This peak represents the signal in the third plot.

Can easily be detected

This peak can easily be detected by eye at this relatively low signal-to-noise ratio. Thus, the use of a matched filter has caused this signal to become detectable in the third plot, when it was not detectable in raw form in the first plot.

An automatic peak detector

For even lower signal-to-noise ratios, the matched filter is not always successful in causing the largest peak in the third plot to correspond to the location of the signal in the first plot.

Even when successful for lower signal-to-noise ratios, the detection peak may be so close to the size of the other peaks that it may be difficult to identify the detection peak by eye.

Therefore, the fourth plot in Figure 2 contains a marker identifying the largest positive peak in the third plot. This can be very helpful in visually identifying the largest peak in the third plot.

Multiple pings

In a real SONAR or RADAR system, it is likely that multiple pings would be fired at the target with each ping producing an output similar to the third plot in Figure 2. However, most of the peaks other than the detection peak would likely be in different locations in the output for each successive ping.

Typically, additional ping-to-ping processing would be applied to make it possible to identify *tracks* produced by detection peaks in successive pings, some of which might be detectable and others of which might not be detectable.

The bottom line

The application of a matched filter to a properly designed signal pulse can cause the energy in the pulse to be compressed into a shorter time interval with a larger amplitude. This can often be useful in detecting the presence of such signals in noise at low signal-to-noise ratios.

Discussion of the code for Dsp039b

Once again, I will discuss the program in fragments.

The class definition for **Dsp040b** begins in Listing 6.

```
class Dsp040b implements GraphIntfc01{

  int signalLen = 50;
  int noiseLen = 400;
  int outputLen = noiseLen - 
signalLen;
  int peakMarkerLen = outputLen;

  double[] noise = new 
double[noiseLen];
  double[] signal = new 
double[signalLen];
  double[] output = new 
double[outputLen];
  double[] peakMarker = 
                       new 
double[peakMarkerLen];
```

Listing 6

This program starts out much the same way as the previous program. Listing 6 establishes the sizes of and create several arrays that will be used to hold various kinds of data throughout the program.

The constructor

The constructor begins in Listing 7.

```
  public Dsp040b(){//constructor

    Random generator = new Random(
                           new
Date().getTime());
    for(int cnt=0;cnt <
noise.length;cnt++){
      //Get noise and remove the dc
offset.
      noise[cnt] =
generator.nextDouble()-0.5;
    }//end for loop
```

Listing 7

White random noise

Listing 7 uses a random number generator to get and save 400 samples of white random noise. The random number generator is fed a different seed each time the program is run, so the 400 random numbers will be different for each run of the program.

> *(Note that after you run the program for the first time from the command line, you can easily rerun it many times in succession simply by clicking the button labeled **Graph** in Figure 2. A new batch of noise data will be generated and processed each time you rerun the program.)*

The code in Listing 7 is essentially the same as code that I explained in the previous lesson entitled Convolution and Frequency Filtering in Java. Therefore, I won't repeat that explanation here.

Need to limit the noise bandwidth

Because it is white noise, the noise at this point contains strong contributions of energy at all frequencies between zero and the folding frequency. Thus, it contains a lot of energy that is outside the bandwidth of the signal shown in Figure 1.

The noise data stored in the **noise** array at this point is not the noise that you see in the first plot in Figure 2. We need to limit the bandwidth of the noise to match the bandwidth of the signal in order to produce the noise that you see in the first plot. We will get to that in a moment. Right now, we need to generate the signal pulse.

Generating the signal pulse

The code in Listing 8 invokes the **fmSweep** method to generate a signal pulse identical to the signal pulse analyzed in the previous program named **Dsp040a**.

```
    fmSweep(signalLen,signal);
```

Listing 8

The code in Listing 8 is identical to the code in the previous program, so I won't discuss it further.

Note that a replica of this signal pulse will be used both to shape the bandwidth of the white noise, and to function as a matched filter for a signal that will be buried in the resulting noise.

Shaping the noise spectrum

Listing 9 uses a replica of the signal as a convolution filter to modify the spectrum of the white noise to make it match the spectrum of the signal. This produces all of the improvement in signal detectability that is available through simple frequency filtering.

```
Convolve01.convolve(noise,signal,output);
```

Listing 9

Although this isn't necessary at this point, I wanted to take advantage of simple frequency filtering prior to setting up the matched filter experiment. That way, I can determine how much of the improvement in detectability is due to matched filtering.

Adjust the noise level

I also wanted to have definitive control over the peak signal to peak noise ratio. To accomplish this, I needed to control the peak level of the noise.

At this point, the filtered noise resides in the array object referred to by **output**. I need to copy it back into the array object referred to by **noise**. Along the way, I will adjust the amplitude of the noise so that the largest peak has an absolute value of 1.0. This is accomplished in Listing 10.

```
    //Clear the noise array
    noise = new double[noiseLen];
    //Get current max value.
    double max = 0.0;
    for(int cnt = 0;cnt <
output.length;cnt++){
      if(Math.abs(output[cnt]) > max){
        max = output[cnt];
      }//end if
    }//end for loop

    //Scale to a peak value of 1.0
    for(int cnt = 0;cnt <
output.length;cnt++){
      noise[cnt] = output[cnt]/max;
    }//end for loop

    //clear the output array for
future use
    output = new double[outputLen];
```

Listing 10

The code in Listing 10 is straightforward and is reasonably well commented. Now that you know what the code is intended to do, there should be no further need for an explanation.

Adjust the amplitude of the signal

Before adding the signal to the noise that now resides in the **noise** array, I need to adjust the amplitude of the signal so as to achieve a specific peak signal to peak noise ratio. This is accomplished in Listing 11.

```
    double SNR = 0.35;//Signal-to-
noise ratio

    for(int cnt = 0;cnt <
signal.length;cnt++){
      signal[cnt] = SNR*signal[cnt];
    }//end for loop
```

Listing 11

Why choose SNR equal to 0.35?

Through experimentation, I determined that for signal-to-noise ratios above 0.35, the process is almost always successful in detecting the signal in the noise.

At a signal-to-noise ratio of 0.35, the process is successful most of the time with an occasional failure to detect the signal.

As the signal-to-noise ratio goes below 0.35, the process fails to successfully detect the signal more frequently.

Therefore, I decided to set the peak signal to peak noise ratio to 0.35. This seemed to be a good balance between the two extremes of total success and total failure.

As originally generated, the signal pulse has a peak amplitude of 1.0. Therefore, since the noise also has a peak value of 1.0, I need to scale the signal values by 0.35 to achieve the desired peak signal to peak noise ratio. This is accomplished in Listing 11, and produces the replica of the signal shown in the second plot in Figure 2.

For future experiments

To experiment with signal detectability versus signal-to-noise ratio, all you need to do is to change the value of the variable named **SNR** and recompile the program.

Add the signal to the noise

Listing 12 adds the scaled signal to the noise beginning at a noise sample index of 200. This produces the data shown in the first plot in Figure 2. The signal begins at the tenth tick mark in Figure 1 and extends for 50 samples, or 2.5 tick marks.

```
    for(int cnt = 0; cnt < signalLen;
cnt++){
      noise[cnt + 200] += signal[cnt];
    }//end for loop
```

Listing 12

In my opinion, even knowing where it was added, at a signal-to-noise ratio of only 0.35, the signal in the first plot is not visually distinguishable from the noise.

Apply the matched filter

Listing 13 uses the replica of the signal shown in the second plot in Figure 2 as a matched convolution operator, and convolves that operator with the data shown in the first plot, producing the output shown in the third plot.

```
Convolve01.convolve(noise,signal,output);
```

Listing 13

Improved signal-to-noise ratio

As discussed earlier, the signal-to-noise ratio is dramatically improved in the third plot relative to the first plot. The signal is easily detectable in the third plot and is not detectable in the first plot.

Improvement is due to matched filtering

Since the bandwidth of the noise was previously shaped to the bandwidth of the signal, this improvement in peak signal to peak noise ratio and the attendant improvement in signal detectability was due almost exclusively to the use of a matched filter. An unmatched filter having the same frequency amplitude response would not provide an improvement of this magnitude.

An easy way to demonstrate this is to modify the program and flip the convolution operator end-for-end before passing it to the **convolve** method. It will still have the same amplitude frequency response shown in the last plot in Figure 1. However, the phase characteristics will no longer match the phase characteristics of the signal, and it will be totally ineffective in improving the signal-to-noise ratio and the detectability of the signal.

An automatic peak detector

The code in Listing 14 searches for the largest positive peak in the data shown in the third plot in Figure 2.

```
    //Search for largest peak in the 
output
    int peakIndex = 0;
    max = 0.0;
    for(int cnt = 0;cnt < 
output.length;cnt++){
      if(output[cnt] > max){
        peakIndex = cnt;
        max = output[cnt];
      }//end if
    }//end for loop

    //Display the location and value 
of the peak
    // on the screen
    System.out.println(peakIndex + " " 
+ max);

    //Insert a marker in the 
peakMarker array
    // showing the location of the 
peak in the
    // filtered output.
    peakMarker[peakIndex] = 90;

  }//end constructor
```

```
Listing 14
```

Once it finds the location of the peak, it displays the location and the value of the peak on the screen. Ideally for this setup, the location should always be 200. However, at this signal-to-noise ratio of 0.35 the process sometimes fails to detect the signal and the highest peak will occur at some location other than 200.

The code in Listing 14 also puts a marker at the location of the peak in the data plotted in the bottom plot in Figure 2.

The code in Listing 14 is straightforward, so I won't discuss it further.

The remaining code

Except for the use of different plotting scale factors in the interface methods, the remaining code in the program is code that I have previously discussed. Therefore, I won't discuss it further. You can view that code in Listing 16.

Run the Programs

I encourage you to copy, compile, and run the programs that you will find in the listings near the end of the lesson. Modify them and experiment with them in order to learn as much as you can about the use of convolution for matched filtering.

I have suggested several possible experiments in the body of this lesson. You might want to try some of them.

Summary

In this lesson, I have explained and illustrated the use of time-domain convolution for matched filtering.

I showed that the use of matched filtering often makes it possible to detect properly designed signals in noisy environments where simple frequency filtering alone cannot do the job. This was illustrated by one of the sample programs in this lesson.

Complete Program Listings

A complete listing of each of the programs is provided in below.

```

/* File Dsp040a.java
Copyright 2005, R.G.Baldwin

This program illustrates the use of convolution
```

```
for matched filtering.  In particular, it is
designed to show the behavior of the matched
filter process in the absence of noise.

The program begins by creating 400 samples of
noise with every noise value equal to zero. This
later results in an infinite signal-to-noise
ratio.

Then it creates a linear sweep FM signal pulse
ranging from zero to 3300 Hz at a sampling rate
of 16000 samples per second.  The pulse has a
constant peak value throughout the length of the
pulse.

Then the signal pulse is added to the noise at
sample index 200.  This results in an infinite
signal-to-noise ratio because all noise values
are zero.

After that, the signal plus noise is convolved
with a replica of the signal to implement the
matched filter process.

Then a spectral analysis is performed on the raw
signal plus noise to show the frequency spectrum
of the signal.  Because a replica of the signal
is used as the convolution operator, the
frequency spectrum of the signal is also the
frequency response of the convolution operator.

Then the program plots the following curves:
Signal plus noise
Convolution operator
Matched filter output
Spectrum of raw signal plus noise

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

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

class Dsp040a implements GraphIntfc01{
  //Establish length for various arrays
  int signalLen = 50;
  int noiseLen = 400;
  int outputLen = noiseLen - signalLen;
  int spectrumPts = outputLen;

  //Create arrays to store different types of
  // data.
  double[] noise = new double[noiseLen];
  double[] signal = new double[signalLen];
  double[] output = new double[outputLen];
  double[] spectrumA = new double[spectrumPts];

  public Dsp040a(){//constructor

    //Note that the noise array contains all zero
    // values.

    //Create and save a signal pulse.  This pulse
    // will also be used as the convolution
    // operator to operate as a matched filter.
    fmSweep(signalLen,signal);

    //Add the signal into the empty noise array
    // beginning at sample index 200.
    for(int cnt = 0; cnt < signalLen; cnt++){
      noise[cnt + 200] += signal[cnt];
    }//end for loop

    //Use the signal pulse as a convolution
    // operator and apply it to the data
    // consisting of signal plus noise.  Recall
    // that the noise values are all zero in this
    // program.
    Convolve01.convolve(noise,signal,output);

    //Compute and save the DFT of the signal plus
    // noise, where the noise values are all
    // zero.
    Dft01.dft(noise,spectrumPts,spectrumA);

    //All of the time series have now been
    // produced and saved.  They may be
    // retrieved and plotted by invoking the
    // methods named f1 through f5.

  }//end constructor

  //-------------------------------------------//
  //The following six methods are required by the
  // interface named GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to process. Must
    // not exceed 5.
    return 4;
  }//end getNmbr
  //-------------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    //This version of this method returns the
    // signal plus noise.
    if(index < 0 || index > noise.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return noise[index] * 90.0;
    }//end else
  }//end f1
  //-------------------------------------------//
  public double f2(double x){
    //Return the signal pulse
    int index = (int)Math.round(x);
    if(index < 0 || index > signal.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return signal[index] * 90.0;
    }//end else
  }//end f2
  //-------------------------------------------//
  public double f3(double x){
    //Return convolution output
    int index = (int)Math.round(x);
    if(index < 0 || index > output.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return output[index] * 4.0;
    }//end else
  }//end f3
  //-------------------------------------------//
  public double f4(double x){
    //Return spectrum of signal plus noise
    int index = (int)Math.round(x);
    if(index < 0 || index > spectrumA.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return spectrumA[index] * 15.0;
    }//end else
  }//end f4
  //-------------------------------------------//
  public double f5(double x){
    return 0.0;
  }//end f5
  //-------------------------------------------//

  //This method generates a pulse with a linear
  // frequency sweep from zero to 3300 Hz at
  // a sampling rate of 16000 samples per
  // second.
  void fmSweep(int pulseLen,double[] output){
    double twoPI = 2*Math.PI;
    double sampleRate = 16000.0;
    double lowFreq = 0.0;
    double highFreq = 3300.0;

    for(int cnt = 0; cnt < pulseLen; cnt++){
      double time = cnt/sampleRate;

      double freq = lowFreq +
                 cnt*(highFreq-lowFreq)/pulseLen;
      double sinValue =
                       Math.sin(twoPI*freq*time);
      output[cnt] = sinValue;
    }//end for loop
  }//end method fmSweep

}//end class Dsp040a
//=============================================//

//This class provides a static method named
// convolve, which applies an incoming
// convolution operator to an incoming set of
// data and deposits the filtered data in an
// output array whose reference is received as an
// incoming parameter.
class Convolve01{
  public static void convolve(double[] data,
                              double[] operator,
                              double[] output){
    //Apply the operator to the data, dealing
    // with the index reversal required by
    // convolution.
    int dataLen = data.length;
    int operatorLen = operator.length;
    for(int i = 0;i < dataLen-operatorLen;i++){
      output[i] = 0;
      for(int j = operatorLen-1;j >= 0;j--){
        output[i] += data[i+j]*operator[j];
      }//end inner loop
      //Divide by the length of the operator
    }//end outer loop
  }//end convolve method
}//end Class Convolve01
//=============================================//

//This class provides a static method named dft,
// which computes and returns the amplitude
// spectrum of an incoming time series.  The
// amplitude spectrum is computed as the square
// root of the sum of the squares of the real and
// imaginary parts.
//Returns a number of points in the frequency
// domain equal to the number of samples in the
// incoming time series.  This is for convenience
// only and is not a requirement of a DFT.
//Deposits the frequency data in an array whose
// reference is received as an incoming
// parameter.
class Dft01{
  public static void dft(double[] data,
                         int dataLen,
                         double[] spectrum){
    double twoPI = 2*Math.PI;

    //Set the frequency increment to the
    // reciprocal of the data length.  This is
    // convenience only, and is not a requirement
    // of the DFT algorithm.
    double delF = 1.0/dataLen;
    //Outer loop interates on frequency values.
    for(int i = 0; i < dataLen;i++){
      double freq = i*delF;
      double real = 0;
      double imag = 0;
      //Inner loop iterates on time- series
      // points.
      for(int j=0; j < dataLen; j++){
        real += data[j]*Math.cos(twoPI*freq*j);
        imag += data[j]*Math.sin(twoPI*freq*j);
        spectrum[i] = Math.sqrt(
                           real*real + imag*imag);
      }//end inner loop
    }//end outer loop
  }//end dft

}//end Dft01
```

Listing 15

```
/* File Dsp040b.java
Copyright 2005, R.G.Baldwin

This program illustrates the use of convolution
for matched filtering.  It buries a signal in
noise having the same bandwidth as the signal and
then detects the signal using a matched filter.

The program is designed so that experiments in
signal detectability versus signal-to-noise ratio
can be performed by changing the value of one
variable and then recompiling the program.

The program begins by creating 400 samples of
white noise using a random noise generator.

Then it creates a linear sweeped FM signal pulse
ranging from zero to 3300 Hz at a sampling rate
of 16000 samples per second.  The pulse has a
constant peak value throughout the length of the
pulse.

Then the program uses the signal pulse as a
convolution operator to filter the white noise
and to eliminate noise energy that is outside the
frequency band of the signal.  This is done so
that all improvement that is achieved later
through matched filtering will be due to matched
filtering and none of the improvement will be due
to simple frequency filtering.
```

```
After the noise is filtered for bandwidth
control, it is scaled to a peak value of 1.0.
This is done so that it will be possible to
establish a definitive peak signal to peak noise
ratio later.

Then the signal pulse is scaled to a peak value
of 0.35 and added to the scaled noise at sample
index 200.  This results in a peak signal to peak
noise ratio of 0.35.  The scale factor that is
applied to the signal can be modified to
experiment with signal detectability versus
signal-to-noise ratio.

After that, the signal plus noise is convolved
with a replica of the signal to implement the
matched filter process.

A peak detector is then applied to the filtered
signal plus noise to find the peak value in the
matched filter output.  The location and size of
the peak is displayed on the screen.  The
location of the peak is also marked in an empty
array which will later be displayed below a plot
of the filtered signal plus noise.

Then the program plots the following curves:
Raw signal plus noise
Convolution operator
Output from matched filter process
Peak marker indicating the location of the
 highest positive peak in the output from the
 matched filter process

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

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

class Dsp040b implements GraphIntfc01{
  //Establish length for signal, noise, and
  // peak marker arrays
  int signalLen = 50;
  int noiseLen = 400;
  int outputLen = noiseLen - signalLen;
  int peakMarkerLen = outputLen;

  //Create arrays to hold various types of data.
  double[] noise = new double[noiseLen];
  double[] signal = new double[signalLen];
  double[] output = new double[outputLen];
  double[] peakMarker =
                       new double[peakMarkerLen];
```

```
public Dsp040b(){//constructor

  //Generate and save some wide-band random
  // noise.  Seed with a different value each
  // time the object is constructed.
  Random generator = new Random(
                          new Date().getTime());
  for(int cnt=0;cnt < noise.length;cnt++){
    //Get noise and remove the dc offset.
    noise[cnt] = generator.nextDouble()-0.5;
  }//end for loop

  //Create and save a signal pulse. This pulse
  // will also be used as the covlution
  // operator to operate as a matched filter.
  fmSweep(signalLen,signal);

  //Filter the noise to eliminate all energy
  // outside the band of the signal.  This is
  // done so that all improvement later in the
  // matched-filter operation will be due to
  // matched filtering and none of the
  // improvement will be due to simple
  // frequency filtering.
  Convolve01.convolve(noise,signal,output);

  //Copy the filtered noise back into the noise
  // array, scaling it to a peak value of 1.0
  //Clear the noise array
  noise = new double[noiseLen];
  //Get current max value.
  double max = 0.0;
  for(int cnt = 0;cnt < output.length;cnt++){
    if(Math.abs(output[cnt]) > max){
      max = output[cnt];
    }//end if
  }//end for loop
  //Scale to a peak value of 1.0
  for(int cnt = 0;cnt < output.length;cnt++){
    noise[cnt] = output[cnt]/max;
  }//end for loop

  //clear the output array for future use
  output = new double[outputLen];

  //Scale the size of the signal.  Change this
  // scale factor to experiment with
  // detectability versus signal-to-noise
  // ratio.
  double SNR = 0.35;//Signal-to-noise ratio
  for(int cnt = 0;cnt < signal.length;cnt++){
    signal[cnt] = SNR*signal[cnt];
  }//end for loop

  //Add the scaled signal into the white noise
```

```
    // beginning at sample index 200.
    for(int cnt = 0; cnt < signalLen; cnt++){
      noise[cnt + 200] += signal[cnt];
    }//end for loop

    //Use the signal pulse as a matched
    // convolution operator and apply it to the
    // data consisting of signal plus noise.
    Convolve01.convolve(noise,signal,output);

    //Search for largest peak in the output
    int peakIndex = 0;
    max = 0.0;
    for(int cnt = 0;cnt < output.length;cnt++){
      if(output[cnt] > max){
        peakIndex = cnt;
        max = output[cnt];
      }//end if
    }//end for loop

    //Display the location and value of the peak
    // on the screen
    System.out.println(peakIndex + " " + max);

    //Insert a marker in the peakMarker array
    // showing the location of the peak in the
    // filtered output.
    peakMarker[peakIndex] = 90;

    //All of the time series have now been
    // produced and saved.  They may be
    // retrieved and plotted by invoking the
    // methods named f1 through f4.

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

  //The following six methods are required by the
  // interface named GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to process. Must
    // not exceed 5.
    return 4;
  }//end getNmbr
  //-------------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    //This version of this method returns the
    // signal plus noise.
    if(index < 0 ||
                index > noise.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return noise[index] * 100.0;
    }//end else
  }//end f1
  //-------------------------------------------//
  public double f2(double x){
    //Return the signal pluse
    int index = (int)Math.round(x);
    if(index < 0 ||
            index > signal.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return signal[index] * 100.0;
    }//end else
  }//end f2
  //-------------------------------------------//
  public double f3(double x){
    //Return convolution output
    int index = (int)Math.round(x);
    if(index < 0 ||
              index > output.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return output[index] * 25.0;
    }//end else
  }//end f3
  //-------------------------------------------//
  public double f4(double x){
    //Return peakMarker
    int index = (int)Math.round(x);
    if(index < 0 ||
           index > peakMarker.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return peakMarker[index] * 1.0;
    }//end else
  }//end f4
  //-------------------------------------------//
  public double f5(double x){
    return 0.0;
  }//end f5
  //-------------------------------------------//

  //This method generates a pulse with a linear
  // frequency sweep from zero to 3300 Hz at
  // a sampling rate of 16000 samples per
  // second.
  void fmSweep(int pulseLen,double[] output){
    double twoPI = 2*Math.PI;
    double sampleRate = 16000.0;
    double lowFreq = 0.0;
    double highFreq = 3300.0;

    for(int cnt = 0; cnt < pulseLen; cnt++){
      double time = cnt/sampleRate;
      double freq = lowFreq +
                 cnt*(highFreq-lowFreq)/pulseLen;
      double sinValue =
                       Math.sin(twoPI*freq*time);
      output[cnt] = sinValue;
    }//end for loop
  }//end method fmSweep

}//end class Dsp040b
//===============================================//

//This class provides a static method named
// convolve, which applies an incoming
// convolution operator to an incoming set of
// data and deposits the filtered data in an
// output array whose reference is received as an
// incoming parameter.
class Convolve01{
  public static void convolve(double[] data,
                              double[] operator,
                              double[] output){
    //Apply the operator to the data, dealing
    // with the index reversal required by
    // convolution.
    int dataLen = data.length;
    int operatorLen = operator.length;
    for(int i = 0;i < dataLen-operatorLen;i++){
      output[i] = 0;
      for(int j = operatorLen-1;j >= 0;j--){
        output[i] += data[i+j]*operator[j];
      }//end inner loop
      //Divide by the length of the operator
    }//end outer loop
  }//end convolve method
}//end Class Convolve01
//===============================================//
```

Listing 16

```
/* 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 17

```
/* File GraphIntfc01.java
Copyright 2005, 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 18

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-