An Adaptive Line Tracker in Java

Learn how to use a general-purpose LMS adaptive engine to write an adaptive spectral line tracker in Java.

Published: December 27, 2005
by Richard G. Baldwin

Java Programming Notes # 2356

Preface

DSP and adaptive filtering

With the decrease in cost and the increase in speed of digital devices, Digital Signal Processing *(DSP)* is showing up in everything from cell phones to hearing aids to rock concerts. Many applications of DSP are static. That is, the characteristics of the digital processor don't change with time or circumstances. However, a particularly interesting branch of DSP is *adaptive filtering*. This is a scenario where the characteristics of the digital processor change with time, circumstances, or both.

Fourth in a series

This is the fourth lesson in a series designed to teach you about adaptive filtering in Java. The first lesson, titled Adaptive Filtering in Java, Getting Started, introduced you to the topic by showing you how to write a Java program to adaptively design a time-delay convolution filter with a flat amplitude response and a linear phase response using an LMS adaptive algorithm.

An adaptive whitening filter

The second lesson in the series, titled An Adaptive Whitening Filter in Java showed you how to write an adaptive *whitening filter* program in Java.

That lesson showed you how to use the whitening filter to extract wide-band signal from a channel in which the signal was corrupted by one or more components of narrow-band noise. The material in this lesson extends what you learned in that lesson, using similar concepts for an entirely different purpose.

A general-purpose adaptive engine

The third lesson in the series, titled A General-Purpose LMS Adaptive Engine in Java, backtracked a bit. The first two lessons were primarily intended to gain your interest in the topic of adaptive filtering. They provided some working sample programs without getting too heavily involved in an explanation of the adaptive process. As a result of that approach, it was a little difficult to separate the adaptive behavior of those sample programs from the behavior designed solely to manage the data and to display the results.

In the third lesson, I presented and explained a general-purpose LMS adaptive engine written in Java that can be used to solve a wide variety of adaptive problems. I applied that engine to three different adaptive signal processing problems, emphasizing the separation of the code that provides the adaptive behavior from the code that is used simply to manage data and to display results.

An adaptive line tracker

In this lesson, I will use the general-purpose LMS adaptive engine to develop an adaptive spectral line tracker in Java. The line tracker will adapt in the time domain and track spectral lines in the frequency domain.

Viewing tip

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

In preparation for understanding the material in this lesson, I recommend that you also study the lessons identified in the References section of this document.

General Background Information

Why track spectral lines?

At this point, you may be asking why a person would want to track spectral lines. There are a variety of good reasons that a person involved in signal processing would want to do such a thing, most of them fairly boring. In an attempt to avoid putting you to sleep, I will justify the need to track spectral lines by giving an explanation in the style of Tom Clancy and *The Hunt for Red October*.

An underwater acoustics listening post

Assume that you are stationed at an underwater acoustics listening post with the task of listening for and identifying enemy submarines. At any point in time, there may be hundreds or even thousands of vessels within range of your listening post, each emitting acoustic energy into the water. Your task is to separate the acoustic energy emitted by the enemy submarine from the acoustic energy emitted by the other vessels in the area.

An acoustic signature

In order to separate the two, you must have some information about the *signature* of the acoustic energy emitted by the submarine. This information may have been gained from clandestine sources.

Suppose you know, for example, that the particular submarine of interest has a pump with a two-speed motor that is responsible for maintaining water pressure in the system that provides water for the sailors to take personal showers. When several showers are being used, the pump runs at a high speed to maintain the required water pressure. When the showers are not being used, the pump motor automatically switches to a lower speed to conserve energy.

If you were to listen to the sound emitted by that pump, it might sound like a high-frequency whine when running at high speed, and it might sound like a low-frequency whine when running at low speed. *(At least, that is what the two-speed pump on my swimming pool sounds like.)*

The spectral signature

If you were to record the sound emitted by the pump and perform a spectral analysis on the recorded data, there would be a peak in the spectrum at a high frequency when the pump is running at high speed, and there would be a peak in the spectrum at a low frequency when the pump is running at low speed. Assuming that those two speeds are the only speeds at which the pump is capable of running, knowledge of those two frequencies could be a small part of a useful acoustic signature for the submarine.

> *(Most of the other vessels in the area probably wouldn't have an onboard motor that runs at exactly those two speeds.)*

A graphic output

If you were to perform the spectral analysis at uniform time intervals and plot the spectral results one below the other, the plotted output might look something like Figure 1.

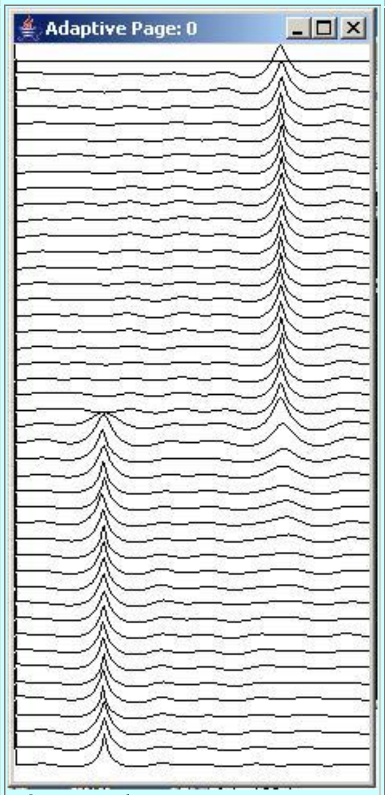

Figure 1

The graphic format

Figure 1 shows increasing time going down the page and increasing frequency going across the page from left to right. Zero frequency is shown at the extreme left. The frequency at the extreme right is one-half the sampling frequency.

Two spectral peaks

The high-frequency peak in the top-half of Figure 1 represents the acoustic energy emitted by the pump during the time interval when the pump is running at high speed. The low-frequency peak in the bottom-half of Figure 1 represents the acoustic energy emitted by the pump during the time interval when the pump is running at low speed.

Just for the record

Although it isn't important at this point in the discussion, the parameters shown in Figure 2 were used to produce the plot shown in Figure 1. *(You can simply ignore Figure 2 for now. It will have more meaning when you see the code that produced Figure 1.)*

```
Using following values:
feedbackGain: 1.0E-5
numberIterations: 3375
filterLength: 15
```

```
wideBandNoiseScale: 20.0
fmSignalScale: 20.0
fmSignalCase: 2
freqSlideConst: 4.0E-4
freqShiftFactorLow: 0.5
freqShiftFactorHigh: 1.5
spectralWidth: 222
lengthMultiplier: 2
Conventional data length: 32
```

Figure 2

Another scenario

Now consider another scenario. Assume that the submarine, *(or some other vessel in the vicinity)*, is running at a very low speed and that the screw is turning very slowly.

> *(In this case, the screw would emit relatively narrow-band acoustic energy at a low frequency.)*

Assume that the skipper of that vessel decides to accelerate in a very controlled manner by slowly increasing the rotational speed of the screw. If the acoustic energy emitted by the propulsion system were recorded and subjected to spectral analysis as before, the plotted results might look something like Figure 3.

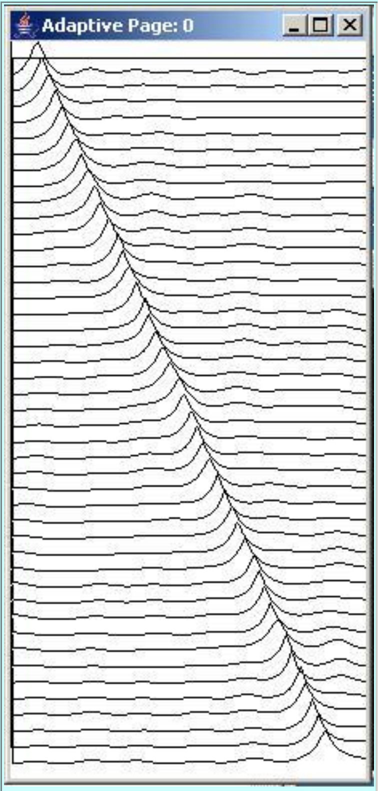

Figure 3

Narrow-band acoustic energy

Because the propulsion system that drives the screw is a rotating machine, it could be expected to emit acoustic energy in a very narrow band of frequencies. As the screw turns faster, the position of that narrow band of frequencies could be expected to move toward the right *(higher)* in the spectrum of the emitted acoustic energy.

Figure 3 represents the spectrum of the acoustic energy emitted by the propulsion system as the speed of the screw advances from a low speed to a high speed at a uniform rate. Of course, any vessel in the area could do that, so there is little or nothing in Figure 3 by itself that identifies the energy as being emitted by the submarine.

The program parameters

For the record, the parameters in Figure 4 were used to produce the output shown in Figure 3.

```
Using following values:
feedbackGain: 1.0E-5
numberIterations: 3375
filterLength: 15
wideBandNoiseScale: 20.0
fmSignalScale: 20.0
fmSignalCase: 1
freqSlideConst: 4.0E-4
freqShiftFactorLow: 0.5
freqShiftFactorHigh: 1.5
spectralWidth: 222
lengthMultiplier: 2
Conventional data length: 32
```
Figure 4

A third scenario

Assume that during the period that the submarine is moving very slowly, many of the sailors have been excused from their duty stations. Many of those sailors decided to take advantage of the break to get a much-needed shower. As a result, the shower pump is running at high speed working to keep the pressure up in the shower system.

Then the captain decides to perform a well-controlled acceleration of the submarine to deal with some situation that has him concerned. At the same time, he orders all of the sailors to report to their duty stations. Those sailors who are in the showers exit quickly and no other sailors enter the showers. As a result, the shower pump breathes a sigh of relief and switches to low speed.

A more complex acoustic signature

If you were to record the acoustic emissions from the submarine during that period and perform the same sort of spectral analysis on the recorded data as before, the output might look something like Figure 5.

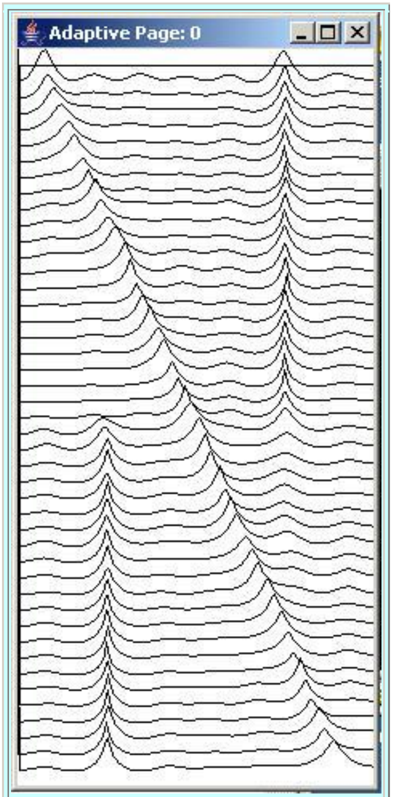

Figure 5

Aha! you say to yourself

You have seen that signature before. When the screw starts speeding up, the shower pump starts to loaf. That might be the important piece of information that lets you conclude with a high degree of confidence that you are looking at the signature of an enemy submarine and not the signature of a fishing trawler.

> *(And, of course, you turn out to be the hero of the novel who saves the world from destruction and marries the heroine.)*

The parameters

Once again, just for the record, the parameters shown in Figure 6 were used to produce the output shown in Figure 5.

```
Using following values:
feedbackGain: 1.0E-5
numberIterations: 3375
filterLength: 15
wideBandNoiseScale: 20.0
fmSignalScale: 20.0
fmSignalCase: 3
freqSlideConst: 4.0E-4
freqShiftFactorLow: 0.5
```

```
freqShiftFactorHigh: 1.5
spectralWidth: 222
lengthMultiplier: 2
Conventional data length: 32
```
Figure 6

An important capability

All joking aside, the capability to track moving spectral lines in wide-band background noise is an important capability for many applications as far flung as sonar and voice analysis. Furthermore, the ability to estimate the actual frequency of the spectral lines to a high degree of accuracy is usually very important.

A side-by-side comparison

Now I am going to show you a comparison between two different approaches to tracking spectral lines. The approach you have been seeing so far is based on adaptively developing and analyzing a whitening filter of the sort that you learned about in the earlier lesson titled An Adaptive Whitening Filter in Java. The other approach involves the use of conventional spectral analysis.

Figure 7 shows the results of applying both approaches to the same scenario that was shown in Figure 5.

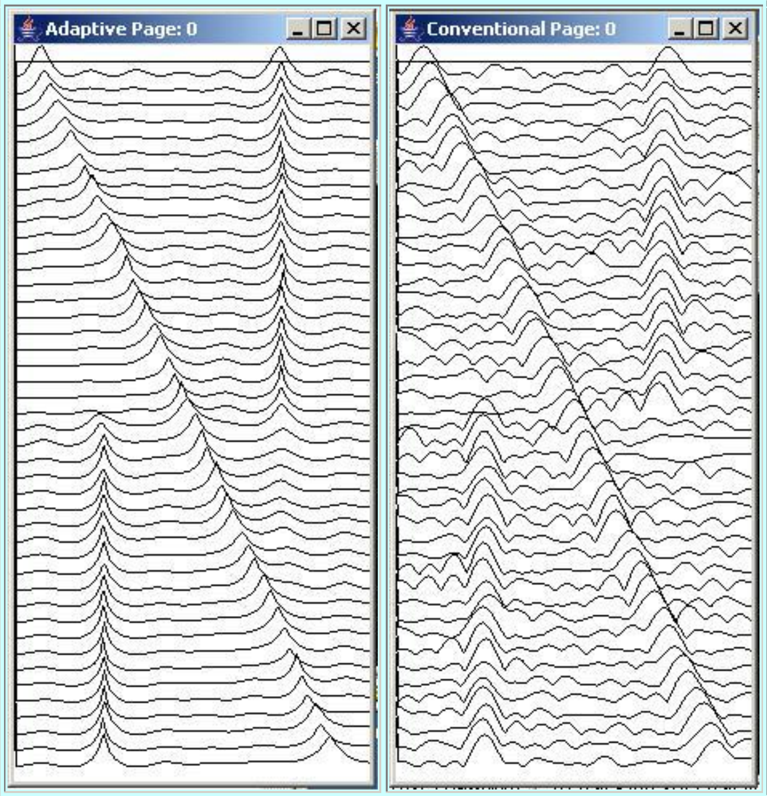

Figure 7

The left panel

The output in the left panel of Figure 7 was produced by:

- Updating the coefficients in a whitening filter having 16 coefficients each time a new sample of signal plus noise is received.
- Computing the frequency response of the whitening filter at the end of every 75th adaptive iteration.
- Displaying the frequency response formatted in such a way that the notches *(nulls)* in the whitening filter are converted to peaks that provide an indication of the location of the spectral lines in the spectrum of signal plus noise.

As you can see, the peaks in the left panel are relatively sharp and the background noise is fairly low.

The right panel

The output in the right panel of Figure 7 was produced by:

- Grabbing the 32 previous samples of signal plus noise at the end of every 75th adaptive iteration.
- Performing a Fourier transform on those 32 samples to provide an estimate of the spectral content of the signal plus noise.
- Displaying the amplitude spectrum in such a way that the peaks in the spectrum provide an indication of the location of the spectral lines in the spectrum of signal plus noise.

As you can see, the peaks in the right panel are relatively broad, and the background noise is relatively high.

Computational requirements

The panel on the left in Figure 7 requires a modest computation for every sample of signal plus noise. In addition, it performs a Fourier transform on the 16-coefficient whitening filter at the end of every 75th iteration.

The panel on the right doesn't require any computations until time comes to perform the spectral analysis. Then it is required to perform a Fourier transform on a time series having 32 samples *(as opposed to a whitening filter having on only 16 coefficients)*. I elected to use twice as many samples for the conventional approach as for the whitening approach in order to end up with roughly a comparable amount of arithmetic required for both approaches.

Resolution and background noise

As you can see, the resolution of the conventional approach is much poorer than the resolution of the adaptive whitening approach. In addition, there is much more noise between the peaks for the conventional approach.

Null processing

The reason that the adaptive whitening approach provides better resolution has to do with the fact that it depends on the nulls in the response of the whitening filter to indicate the locations of the spectral lines rather than depending on the peaks. In many signal processing areas, nulls are much sharper than peaks and better resolution can be obtained through a process often referred to as *"null steering"*. Of course, there are always tradeoffs, but in this case the tradeoff appears to be advantageous.

A radio direction finder

I'm going to illustrate this point by describing a completely different, but somewhat related technology.

At one point in my career, I was responsible for using radio direction finding *(RDF)* equipment to determine the location of radio transmitters. Many types of equipment were available for this task. The simplest was a van with a directional radio antenna mounted on the top. The person inside the van could turn a knob to rotate the radio antenna.

A directional response pattern

The radio antenna had a directional response pattern that looked something like a figure **8**. The lobes on the top and the bottom of the figure 8 were approximately the same size, and nearly round.

Operation

In operation, the user would listen to a radio transmitter with a pair of headphones while manually rotating the antenna. He would note the two positions of the antenna that resulted in the strongest or loudest signal, and the weakest signal. Having calibrated the antenna, could then estimate the bearing *(direction)* to the transmitter with an ambiguity of 180 degrees.

The response pattern

By directional response pattern, I mean the effectiveness with which the antenna can successfully receive signals from a transmitter. For a directional response pattern, this depends on the direction to the transmitter relative to the rotational position of the antenna.

Transmitters located on a line that intersects the figure 8 along its long dimension *(the vertical dimension relative to this figure* **8***)* would be received the strongest. Transmitters located on a line that intersects the figure 8 along its narrow dimension *(the horizontal dimension relative to this figure* **8***)* would be poorly received if they are received at all. This is because the directional response pattern has a null response for radio signals received along the direction of the narrow dimension.

How does the system work?

Now envision what happens when the radio transmitter is fixed in a particular location and the antenna *(and its response pattern)* is rotated. When the antenna is rotated to the point where the long dimension of the figure 8 is aligned with the direction of the transmitter, the signal is strong. When the antenna is rotated by ninety degrees in either direction, the signal becomes weaker and weaker until it may no longer be discernable. At that point, it can be concluded that the transmitter is somewhere along a line that intersects the figure 8 along its narrow dimension.

There is still an ambiguity of 180 degrees because it isn't possible to determine which side of the antenna the transmitter is on. However, that ambiguity can be resolved by moving the van containing the antenna to a different location and taking another directional reading on the same transmitter.

The null is narrow

As I mentioned above, the two lobes in the antenna response pattern are nearly circular *(probably more so than the typographical figure* **8** *that you see here)*. Thus, the angular width of the null is very narrow. As a result, rotating the antenna by only a few degrees produces a significant change in the *loudness* of the transmitter when the direction is near the null.

On the other hand, the main lobes of the figure 8 are very broad. Rotating the antenna by the same number of degrees produces very little change in the *loudness* of the signal when the direction is along the long dimension of the figure 8. Consequently, a much more accurate estimate of the bearing to the transmitter can be made by rotating the signal through the null in the pattern than can be made by rotating the signal through the peak in the pattern.

This is null steering

This well-known process is often referred to as null-steering. This is the general principle by which the adaptive spectral line tracking approach provides a more accurate estimate of the frequency of the line than does the conventional spectral analysis approach.

In effect, the null steering *(adaptive line tracking)* approach sacrifices accuracy in the estimate of the strength of the spectral line for better accuracy in the frequency of the line. *(As you will recall from the earlier lesson titled An Adaptive Whitening Filter in Java, the whitening filter used in the adaptive line tracker places nulls at the locations of the lines in the spectrum.)*

In the case of radio direction finding, the null steering approach sacrifices accuracy in the estimate of the strength of the transmitter for better accuracy in the estimate of the direction to the transmitter.

What about the signal-to-noise ratio?

Although the adaptive approach provides better frequency resolution than the conventional approach in Figure 7, the existence of the spectral lines is apparent in both displays. The peak-to-peak amplitude of each of the narrow-band signals in Figure 7 was equal to the peak-to-peak amplitude of the wide-band noise. It is important to ask what happens when the signal-to-noise ratio is reduced.

Figure 8 shows the result of decreasing the signal-to-noise ratio by a factor of three relative to that for Figure 7. In Figure 8, the peak-to-peak amplitude of the wide-band noise is three times greater than the peak-to-peak amplitude of each of the narrow-band signals.

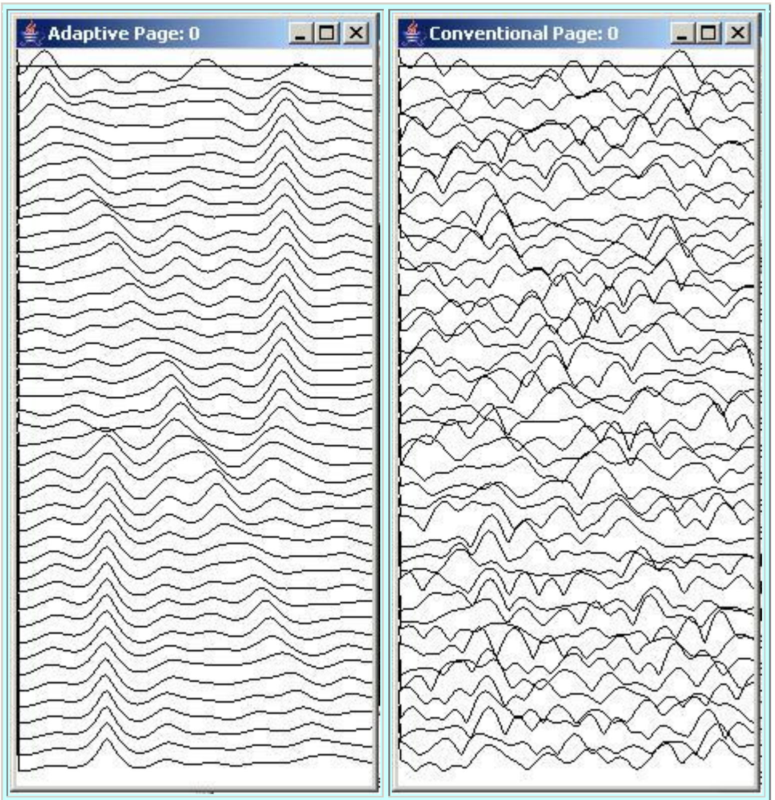

Figure 8

The program parameters

The parameters shown in Figure 9 were used to produce the output shown in Figure 8. The signal and noise level information is provided by the two boldface lines in Figure 9.

```
Using following values:
feedbackGain: 1.25E-6
numberIterations: 3375
filterLength: 15
wideBandNoiseScale: 60.0
fmSignalScale: 20.0
fmSignalCase: 3
freqSlideConst: 4.0E-4
freqShiftFactorLow: 0.5
freqShiftFactorHigh: 1.5
spectralWidth: 222
lengthMultiplier: 2
```

```
Conventional data length: 32
```
Figure 9

Which approach is best?

You can judge for yourself, but it looks to me like the existence and location of the spectral lines in the adaptive *(left)* panel in Figure 8 is much better than the conventional *(right)* panel.

Increase conventional spectrum data length

The right panel of Figure 10 shows the result of keeping the signal-to-noise ratio the same as in Figure 8 while increasing to 80 samples the amount of signal plus noise data included in each conventional Fourier transform. *(This significantly increases the computational requirement for the conventional approach.)* In this case, each chunk of data subjected to conventional spectrum analysis overlaps the previous chunk by five samples.

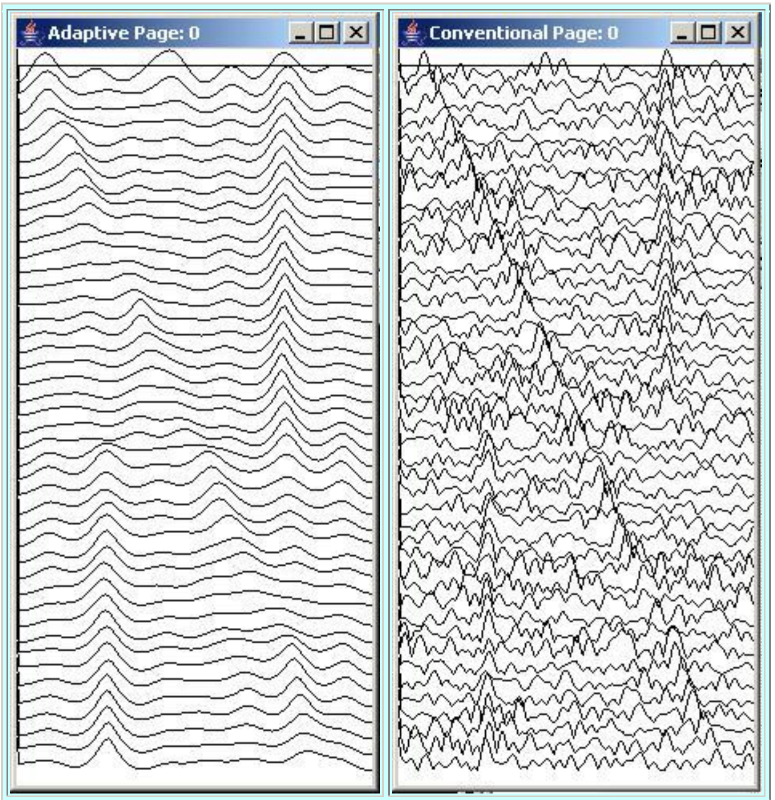

Figure 10

Which is best now?

Once again, you can be the judge. Although the existence and location of the spectral lines in the right panel of Figure 10 is more obvious than the existence and location of the spectral lines in the right panel of Figure 8, it still looks to me like the adaptive approach in the left panel in

Figure 10 is superior. Furthermore, the left panel in Figure 10 is much less computationally demanding than the right panel.

The program parameters

Figure 11 shows the parameters that were used to produce the output shown in Figure 10. The increased length of data used for conventional spectral analysis is shown by the boldface lines in Figure 11.

```
Input values were provided.
Using following values:
feedbackGain: 1.25E-6
numberIterations: 3375
filterLength: 15
wideBandNoiseScale: 60.0
fmSignalScale: 20.0
fmSignalCase: 3
freqSlideConst: 4.0E-4
freqShiftFactorLow: 0.5
freqShiftFactorHigh: 1.5
spectralWidth: 222
lengthMultiplier: 5
Conventional data length: 80
```
Figure 11

Enough talk, let's see some code!

Now let's see the code that produced these results.

Preview

The program named Adapt05

The adaptive line tracker program named **Adapt05** can be viewed in its entirety in Listing 20 near the end of the lesson.

The purpose of the program is to use the general purpose LMS adaptive engine provided by **AdaptEngine01** *(see an explanation of the* ***AdaptEngine01*** *class in the lesson titled A General-Purpose LMS Adaptive Engine in Java)* to implement an adaptive spectral line tracker. The line tracker is designed to track the frequency of frequency-modulated signals buried in wide-band noise. Adaptive processing takes place in the time domain. The signals are tracked in the frequency domain.

Comparison with conventional methods

Experimental results produced by the adaptive line tracker are compared with results produced by conventional spectrum analysis.

A whitening filter is used

The program develops a whitening filter *(see the lesson titled An Adaptive Whitening Filter in Java)* performing one adaptive iteration for each incoming sample. In addition, the program computes the amplitude frequency response of the whitening filter at the end of every 75th iteration. The notches in the whitening filter are converted to peaks and provide an indication of the frequencies of the FM signals.

The result of computing each amplitude response on a whitening filter is displayed as one of the plots in the left panel of Figure 10. The plots are stacked one below the other with increasing time going down the page. The peaks in the plots in Figure 10 show the frequencies of the FM signals in the time period immediately preceding each plot. Frequency increases from left to right with zero at the left and one-half the sampling frequency at the right.

Experimental results

A demonstration of the program capability is accomplished by processing time series consisting of wide-band noise plus FM signals. Three different experimental cases can be specified by the user:

1. A single FM sweep from a low frequency to a high frequency. The user specifies the rate at which the signal sweeps.
2. A single FM signal that switches back and forth between two frequencies. The user specifies each of the two frequencies.
3. An additive combination of the two cases described above.

User input

User input to control a variety of experimental parameters is provided by way of command-line parameters. The command-line parameters are:

- **double feedbackGain**: This is a multiplicative factor that is used in the feedback loop of the LMS adaptive algorithm.
- **int numberIterations**: This is the number of iterations that the adaptive algorithm is allowed to execute before terminating and displaying the results. In Figure 10, the number of iterations was chosen so as to fill one page with plotted results.
- **int filterLength**: This is the length of the prediction filter that is developed within the adaptive algorithm. Note that this length is one less than the length of the whitening filter mentioned above. As you learned in the lesson titled An Adaptive Whitening Filter in Java, the whitening filter consists of the prediction filter with a -1 concatenated onto its end.
- **double wideBandNoiseScale**: This is a scale factor that is applied to the wide band noise before it is added to the FM signal.
- **double fmSignalScale**: This is a scale factor that is applied to each FM signal before it is added to the wide-band noise.

- **int fmSignalCase**: This parameter specifies the test case described above. The value must be 1, 2, or 3.
- **double freqSlideConst**: This value specifies the rate at which the FM sweep signal changes frequency. The higher the value of this parameter, the faster will be the change in frequency.
- **double freqShiftFactorLow**: The base frequency for the frequency switching signal is one-fourth of the sampling frequency. This multiplicative factor is applied to the base frequency to establish the low frequency for the frequency-switching signal. For example, a value of 0.5 for this parameter results in a frequency that is one-eighth of the sampling frequency.
- **double freqShiftFactorHigh**: This parameter is applied as a multiplicative factor to the base frequency to establish the upper frequency for the frequency shifting signal.
- **int lengthMultiplier**: This parameter specifies the length of each chunk of data that is analyzed using conventional spectral analysis. This value is a multiple of the length of the whitening filter.

Default parameters

If the user doesn't provide the required ten command-line parameters, parameter values are used.

See the default-value comments in the code for an indication of the approximate values that might be appropriate for any particular parameter.

A marker at zero frequency

The program puts a marker at zero frequency in each spectral plot. This makes it possible to visually confirm that the spectral plots are properly aligned, with one spectral plot shown above the other.

If the plots are properly aligned, this results in the thin black vertical line shown at a frequency of zero in Figure 10. If the plots are not properly aligned, this marker will appear to walk across the page, probably in a fairly regular fashion.

Program testing

The program was tested using J2SE 5.0 and WinXP. J2SE 5.0 or later is required.

The class named PlotALot08

This program uses a new version of a plotting class from the **PlotALot** family of plotting classes. You can view a complete listing of the **PlotALot08** class in Listing 21. By now, you should be very familiar with the general construct of the classes in the **PlotALot** family, so I won bore you with an explanation of the class.

The adaptive engine named AdaptEngine01

The adaptive engine named **AdaptEngine01** provides all of the required adaptive behavior for this program. You learned about the adaptive engine in the lesson titled A General-Purpose LMS Adaptive Engine in Java, so I won't repeat the explanation of that class in this lesson.

Discussion and Sample Code

Adapt05

The class named **Adapt05** and the **main** method begin in Listing 1.

```
class Adapt05{

  public static void main(String[] args){
    //Default parameter values.  See a
description of each
    // of these parameters in the opening
comments above.
    // Note that this set of default values
represents a
    // high signal-to-noise ratio.
    double feedbackGain = 0.00001;
    int numberIterations = 3375;
    int filterLength = 15;
    double wideBandNoiseScale = 1.0;
    double fmSignalScale = 20.0;
    int fmSignalCase = 3;
    double freqSlideConst = 0.0004;
    double freqShiftFactorLow = 0.5;
    double freqShiftFactorHigh = 1.5;
    int lengthMultiplier = 2;

    int spectralWidth = 222;//Not a user input
value

    if(args.length != 10){
      System.out.println(
               "Usage with all parameters
following the " +
               "program name:\n" +
               "java Adapt05\n" +
               "feedbackGain\n" +
               "numberIterations\n" +
               "filterLength\n" +
               "wideBandNoiseScale\n" +
               "fmSignalScale\n" +
               "fmSignalCase\n" +
               "freqSlideConst\n" +
               "freqShiftFactorLow\n" +
               "freqShiftFactorHigh\n" +
               "lengthMultiplier\n");

      System.out.println(
          "Input values were not provided.\n"+
          "Using following values:\n" +
          "feedbackGain: " + feedbackGain +
          "\nnumberIterations: " +
numberIterations +
          "\nfilterLength: " + filterLength +
          "\nwideBandNoiseScale: " +
wideBandNoiseScale +
          "\nfmSignalScale: " + fmSignalScale +
          "\nfmSignalCase: " + fmSignalCase +
          "\nfreqSlideConst: " + freqSlideConst
+
          "\nfreqShiftFactorLow: " +
freqShiftFactorLow +
          "\nfreqShiftFactorHigh: " +
freqShiftFactorHigh +
          "\nspectralWidth: " + spectralWidth +
          "\nlengthMultiplier: " +
lengthMultiplier +
          "\nConventional data length: " +
                         (lengthMultiplier *
filterLength +

lengthMultiplier));
    }else{//Command line params were provided.
      feedbackGain =
Double.parseDouble(args[0]);
      numberIterations =
Integer.parseInt(args[1]);
      filterLength = Integer.parseInt(args[2]);
      wideBandNoiseScale =
Double.parseDouble(args[3]);
      fmSignalScale =
Double.parseDouble(args[4]);
      fmSignalCase = Integer.parseInt(args[5]);
      freqSlideConst =
Double.parseDouble(args[6]);
      freqShiftFactorLow =
Double.parseDouble(args[7]);
      freqShiftFactorHigh =
Double.parseDouble(args[8]);
      lengthMultiplier =
Integer.parseInt(args[9]);

      System.out.println(
          "Input values were provided.\n"+
          "Using following values:\n" +
          "feedbackGain: " + feedbackGain +
          "\nnumberIterations: " +
numberIterations +
          "\nfilterLength: " + filterLength +
          "\nwideBandNoiseScale: " +
wideBandNoiseScale +
          "\nfmSignalScale: " + fmSignalScale +
          "\nfmSignalCase: " + fmSignalCase +
          "\nfreqSlideConst: " +
freqSlideConst+
          "\nfreqShiftFactorLow: " +
freqShiftFactorLow +
          "\nfreqShiftFactorHigh: " +
freqShiftFactorHigh +
          "\nspectralWidth: " + spectralWidth +
          "\nlengthMultiplier: " +
lengthMultiplier +
          "\nConventional data length: " +
                         (lengthMultiplier *
filterLength +

lengthMultiplier));
    }//end else

Listing 1
```

The code in Listing 1 is straightforward, and shouldn't require any explanation beyond the comments in the code.

Instantiate an object of the class

Listing 2 instantiates a new object of the **Adapt05** class and invokes the method named **process** on that object.

```
    new Adapt05().process(feedbackGain,
                          numberIterations,
                          filterLength,
                          wideBandNoiseScale,
                          fmSignalScale,
                          fmSignalCase,
                          freqSlideConst,
                          freqShiftFactorLow,
                          freqShiftFactorHigh,
                          spectralWidth,
                          lengthMultiplier);
  }//end main

Listing 2
```

Listing 2 also signals the end of the **main** method.

The method named process

Listing 3 shows the beginning of the method named **process**.

```
  void process(double feedbackGain,
               int numberIterations,
               int filterLength,
               double wideBandNoiseScale,
               double fmSignalScale,
               int fmSignalCase,
               double freqSlideConst,
               double freqShiftFactorLow,
               double freqShiftFactorHigh,
               int spectralWidth,
               int lengthMultiplier){

    //Declare and initialize working variables.
    double err = 0;
    double wideBandNoise = 0;
    double fmSignal = 0;
    double freqSlideValue = 0;
    double freqShiftFactor =
freqShiftFactorLow;
```

Listing 3

This is the primary processing and plotting method for the program. This method uses an object of the **AdaptEngine01** class to provide the adaptive behavior.

The code in Listing 3 declares and initializes some working variables.

Array for the whitening filter

Listing 4 creates an array object to contain the whitening filter. The actual whitening of the data is accomplished within the object of type **AdaptEngine01**. That object returns the prediction portion of the whitening filter, but does not return an actual whitening filter.

This copy of the whitening filter is required solely for the purpose of computing and displaying the frequency response of the whitening filter. Note that the length of the whitening filter is one greater than the length of the filter that is returned by the adaptive object. The extra coefficient in the whitening filter is set to a value of -1 by the code in Listing 4.

```
    double[] whiteningFilter =
                              new
double[filterLength + 1];

    whiteningFilter[filterLength] = -1;
```

Listing 4

All other values in the whitening filter are initialized to zero by the code in Listing 4. Coefficient values returned by the adaptive process are copied into the lower elements of the whitening filter later.

A data delay line

Listing 5 creates an array to contain two samples of the data that will be adaptively processed.

```
    double[] data = new double[2];
```

```
Listing 5
```

This array is used as a tapped delay line. During each adaptive iteration, the data sample to be filtered is located at index 0. The value of the adaptive target is located at index 1. Each new data sample is inserted at index 1, causing the value at that index to be moved to index 0. This causes the value at index 0 to fall off the end of the delay line.

A delay line for raw data

Listing 6 creates an array object to serve as a delay line to contain a chunk of raw data. This raw data is used for conventional spectral analysis.

```
    double[] rawData = new double[

lengthMultiplier*whiteningFilter.length];

Listing 6
```

The length of this delay line is an integer multiple of the length of the whitening filter. The multiplier is provided as a user input parameter.

A general-purpose adaptive engine object

Listing 7 instantiates a general purpose adaptive processing object of the class **AdaptEngine01**. This object is used later to provide the adaptive behavior for the entire program.

```
    AdaptEngine01 theAdapter =
              new
AdaptEngine01(filterLength,feedbackGain);

Listing 7
```

Instantiate plotting objects

Listing 8 instantiates two plotting objects of the **PlotALot08** class. The first plotting object is used later to display the frequency response of the whitening filter. The second plotting object is used later to display the results of the conventional spectrum analysis.

```
    //Instantiate a plotting object to display
whitening
    // filter frequency response data.
    PlotALot08 freqPlotObj = new PlotALot08(

"Adaptive",

spectralWidth + 8,
                                          487,
                                          10,
                                          1,
                                          0,
                                          0);

    //Instantiate a plotting object to display
the results
    // of conventional spectrum analysis.
    PlotALot08 conventionalPlotObj = new
PlotALot08(

"Conventional",

spectralWidth + 8,
                                          487,
                                          10,
                                          1,
                                          0,
                                          0);
```

Listing 8

The class named **PlotALot08** can be viewed in Listing 21 near the end of the lesson. Although this class is new to this lesson, it is very similar to the previously-published classes in the family of **PlotALot** classes and should not require an explanation.

Start looping and get wide-band noise sample

Listing 9 shows the beginning of a **for** loop that will execute the specified number of adaptive iterations.

```
    for(int cnt = 0;cnt <
numberIterations;cnt++){
      wideBandNoise =

wideBandNoiseScale*(2.0*(random() - 0.5));
```

Listing 9

Listing 9 also gets the next sample of **wideBandNoise** with a uniform distribution from -1.0 to +1.0. The noise is scaled by the specified scale factor. *(Note the use of a static import directive for the Math class, which requires J2SE 5.0 or later.)*

Get the next sample of narrow-band signal

Listing 10 gets the next sample of frequency-modulated sinusoidal *(narrow-band)* signal.

```
      //Get the next sample of fmSignal data.
```

```
The contents
      // of the following variable sets the
frequency for
      // the FM sweep signal for this sample.
This value
      // increases for each successive
iteration.
      freqSlideValue += freqSlideConst;

      //Use the value of fmSignalCase to
determine which
      // configuration of FM signal to
generate.
      if(fmSignalCase == 1){
        //Single source, sweep frequency
        fmSignal = fmSignalScale*(sin(
                          cnt*(freqSlideValue +
2*PI/32)));
      }else if(fmSignalCase == 2){
        //Single source, shift frequency.
Frequency shifts
        // every 1650 iterations.
        if(cnt%1650 == 0){
          if(freqShiftFactor ==
freqShiftFactorLow){
            freqShiftFactor =
freqShiftFactorHigh;
          }else{
            freqShiftFactor =
freqShiftFactorLow;
          }//end else
        }//end if
        fmSignal = fmSignalScale*sin(

freqShiftFactor*cnt*2*PI/4);
      }else if(fmSignalCase == 3){
        //Two signal sources with one of each
of the above
        // configurations.
        if(cnt%1650 == 0){
          if(freqShiftFactor ==
freqShiftFactorLow){
            freqShiftFactor =
freqShiftFactorHigh;
          }else{
            freqShiftFactor =
freqShiftFactorLow;
          }//end else
        }//end if
        fmSignal = fmSignalScale*(sin(
                          cnt*(freqSlideValue +
2*PI/32)) +

sin(freqShiftFactor*cnt*2*PI/4));
      }else{
        System.out.println(
            "Incorrect signal case,
terminating");
        System.exit(0);
      }//end else
```

Listing 10

The narrow-band signal consists of:

- A frequency-modulated sweep as shown in Figure 3, or
- A frequency-modulated shift as shown in Figure 1, or
- A combination of the two as shown in Figure 5.

The choice among the three alternatives is provided by the user as an input parameter.

Although the code in Listing 10 is somewhat long, it is straightforward and shouldn't require a detailed explanation.

Insert the raw data into the delay lines

The first statement in Listing 11 inserts the raw signal plus noise data into the two-element delay line that is used to feed the adaptive process.

The second statement in Listing 11 inserts the raw signal plus noise data into the longer delay line that is used to feed the conventional spectrum analysis process.

```
      //Insert the wideBandNoise plus fmSignal
into the
      // two-element delay line.
      flowLine(data,wideBandNoise+fmSignal);

      //Insert signal plus noise into delay
line used for
      // conventional spectrum analysis.
      flowLine(rawData,wideBandNoise+fmSignal);
```

Listing 11

Obviously, I could have used the same delay line for both purposes. However, since each delay line is used to feed an entirely different process, I decided to separate the two for clarity of purpose.

Execute the adaptive process

The single statement in Listing 12 invokes the **adapt** method on the adaptive engine of type **AdaptEngine01**, to perform the adaptive process for the entire program. *(All of the other code in the **process** method is used to manage data and to display results.)*

```
      AdaptiveResult result =

theAdapter.adapt(data[0],data[1]);

Listing 12
```

Input to the adapt method

The code in Listing 12 passes the two samples from the two-element delay line to the **adapt** method. The first parameter is the next sample of raw data that is to be filtered by the adaptive prediction filter. The second parameter is the sample that is used as the predictive target.

In other words, the **adapt** method uses the first parameter in conjunction with previous samples that have been saved in an attempt to predict the value of the second parameter. The prediction error is then used to adjust the coefficients in the prediction filter that is maintained by the **adapt** method.

Several values are returned by the adapt method

Several important results are returned by the **adapt** method. These results are encapsulated in an object of the class **AdaptiveResult**. This class is a wrapper class that is designed to encapsulate the adaptive results in a single object.

Frequency response and conventional spectrum

Now that the adaptive process has been completed for this iteration, the next major task is to compute and to display the frequency response of the whitening filter and the conventional spectrum of the raw data at the end of every 75th adaptive iteration.

Construct the whitening filter

As mentioned earlier, the actual whitening process is performed inside the **adapt** method. A copy of the whitening filter is not returned by the **adapt** method. However, a copy of the prediction filter is returned by the **adapt** method. The whitening filter is simply the prediction filter with an extra coefficient having a value of -1 concatenated onto its end.

An array for storage of the whitening filter was constructed in Listing 4. The value of -1 was also deposited into the last element in that array in Listing 4. Listing 13 constructs and saves the whitening filter by copying the coefficients from the prediction filter that was returned by the **adapt** method into the lower elements of the whitening filter array, leaving the value of -1 in the last element.

```
      System.arraycopy(result.filterArray,
                       0,
                       whiteningFilter,
                       0,
                       filterLength);
```

Listing 13

*(While performing the final edit on this lesson, I realized that I should have moved the statement in Listing 13 inside of the **if** statement in Listing 14 to conserve computer resources. The new whitening filter needs to be constructed only when the frequency response of the filter is to be computed and displayed.)*

Compute and display frequency data

Listing 14 contains an **if** statement that causes frequency data to be computed and displayed at the end of every 75th adaptive iteration, as shown in Figure 7.

```
      if(cnt%75 == 0){
        //Compute and display the frequency
response of the
        // whitening filter.
        displayFreqResponse(whiteningFilter,
                            freqPlotObj,
                            spectralWidth,

whiteningFilter.length - 1);

        //Compute and display the conventional
spectrum of
        // a chunk of raw signal plus noise
data.
        displaySpectrum(rawData,
                        conventionalPlotObj,
                        spectralWidth,
                        rawData.length - 1);
      }//End display of frequency data
    }//End for loop, End adaptive process
```

Listing 14

The invocation of the **displayFreqResponse** method in Listing 14 computes and displays the amplitude response of the whitening filter as shown in the left panel of Figure 7.

The invocation of the **displaySpectrum** method in Listing 14 computes and displays the conventional amplitude spectrum for comparison purposes as shown in the right panel of Figure 7.

I will have more to say about these two methods later.

Display the results

Listing 14 also signals the end of the **for** loop that began in Listing 9. Once the code in Listing 14 has been executed, all that remains is to cause the graphic data that has been stored in the plotting objects to be displayed on the screen. This is accomplished by the code in Listing 15.

```
    freqPlotObj.plotData(0,0);
    conventionalPlotObj.plotData(232,0);

  }//end process method

Listing 15
```

Listing 15 also signals the end of the **process** method and the end of the program.

Computing and displaying frequency data

Listing 14 above invokes the **displayFreqResponse** method and the **displaySpectrum** method to cause the frequency data to be displayed.

The purpose of the **displayFreqResponse** method is to compute and display the amplitude frequency response of an incoming whitening filter.

Partially explained in an earlier lesson

A method very similar to and having the same name as the **displayFreqResponse** method was explained in the lesson titled An Adaptive Whitening Filter in Java. Therefore, I am going to begin by referring you back to that lesson for an explanation of the overall method.

You can view the new version of **displayFreqResponse** used in this lesson in Listing 20 near the end of the lesson. This explanation will pick up at the point in the method where the version of the method used in this lesson differs from the version of the method used in the lesson titled An Adaptive Whitening Filter in Java.

Primary differences

The primary difference between the two versions of the method has to do with the final formatting of the frequency response curve for display. If you refer back to Figure 3 in the lesson titled An Adaptive Whitening Filter in Java, you will see that the locations of spectral lines in the spectrum were indicated by deep notches in the amplitude response curve. *(That lesson also displayed phase response information, which is not of interest in this lesson.)*

If you refer back to Figure 1 in this lesson, you will see that the presence of narrow-band signal is indicated by a peak at the signal frequency in the display of the response. This is accomplished by turning the response curve upside down and then normalizing it.

Code in the displayFreqResponse method

The code in Listing 16 picks up at the point where the code differs from the code in the previously-explained version of the method.

Listing 16 changes the algebraic sign on all the amplitude response values to turn the response curve upside down. This causes the notches in the response curve to appear as peaks in the display as shown in Figure 1.

```
    //Note that from this point forward, all
references to
    // magnitude are referring to log base 10
data, which
    // can be thought of as scaled decibels.

    for(int cnt = 0;cnt <
magnitude.length;cnt++){
      magnitude[cnt] = -magnitude[cnt];
    }//end for loop

Listing 16
```

Was this necessary?

Obviously it wasn't necessary to turn the notches into peaks. However, this results in a display that is more in line with what we are accustomed to seeing. We tend to expect the presence of energy to be represented by a peak in the spectrum and that is the information that we are conveying here.

Normalization of the data

The effective display of a large amount of data is not an easy thing to accomplish. A typical display medium normally provides a fixed amount of space in which to display the data. If the data values are large, excursions in the display will typically be large, and may even exceed the allowable space.

On the other hand, if the data values are small, excursions in the display will often be very small and possibly not even visible. A lot of work is usually required to make it possible to display data having a wide variety of values in a limited space in a meaningful way.

The purpose of much of the following code is to normalize the amplitude response such that each response curve will occupy the same size rectangle in the display regardless of the magnitude of the input data. *(See Figure 1 as an example of this normalization.)*

Make the smallest value equal to zero

The code in Listing 17 is executed once for each frequency response curve that is to be displayed. We begin by biasing the values in the frequency response such that the smallest value becomes zero.

```
    //First find the smallest value.
    double min = 9999999999.0;
    for(int cnt = 0;cnt <
magnitude.length;cnt++){
      if(magnitude[cnt] < min){
        min = magnitude[cnt];
      }//end if
    }//end for loop
    //Now apply the bias.
    for(int cnt = 0;cnt <
magnitude.length;cnt++){
      magnitude[cnt] -= min;
    }//end for loop
```

Listing 17

Once you know the purpose of the code in Listing 17, the behavior of the code is straightforward and should not require further explanation.

Normalize the peak value

The code in Listing 18 performs the following steps:

- Find and save the absolute peak value for the response curve
- Set the value at zero frequency to the peak value for the response curve.
- Scale all the values in the response curve to cause the peak value to end up with a vale of 20. This causes each response curve to occupy overlapping rectangular strips in the final plot *(see Figure 1)*, each of which has a vertical dimension of 20 pixels.

```
    //Find the absolute peak value.  Begin with
a negative
    // trial value with a large magnitude and
replace it
    // with the largest magnitude value.
    double peak = -9999999999.0;
    for(int cnt = 0;cnt <
magnitude.length;cnt++){
      if(peak < abs(magnitude[cnt])){
        peak = abs(magnitude[cnt]);
      }//end if
    }//end for loop

    //Set the zero frequency value to the peak
so that it
    // can be used to visually confirm plot
synchronization
    // later.
    magnitude[0] = peak;

    //Normalize to 20 times the peak value.
Each response
    // curve will now occupy a horizontal strip
of the
    // plotting area that is 20 pixels from top
to bottom.
```

```
    for(int cnt = 0;cnt <
magnitude.length;cnt++){
      magnitude[cnt] = 20*magnitude[cnt]/peak;
    }//end for loop
```

Listing 18

The value at zero frequency

Typically the value of a frequency response curve at a frequency of zero is of little or no interest.

> *(Signals in the real world don't usually exist at a frequency of zero, unless you are measuring the output from a battery or direct-current generator. A non-zero value at zero frequency usually indicates an undesirable electronic bias in the acquisition of the sampled data.)*

Therefore, the values of the response curves at zero frequency were used to solve a potential plotting-alignment problem.

Alignment can be difficult

It can be a little difficult to get all of the parameters set correctly in the use of the class named **PlotALot08** to cause the plots to be properly aligned with each response curve correctly placed above the one below it. Therefore, the value of the response at zero frequency was artificially set to the peak value to make it visually obvious if the plot is not properly aligned. When the plot is properly aligned, those values all line up vertically on the left side of the plot as shown in Figure 1.

Even without the markers at zero frequency, it would be rather obvious if the plots were not properly aligned in Figure 1. However, that would not be the case for the right panel in Figure 8 or the right panel in Figure 10. Therefore, this marker at zero frequency can be very helpful in providing assurance that the spectral data is being properly displayed.

Feed the plotting object

The code in Listing 19 feeds the normalized frequency response data to the plotting object. Note that this data was converted to decibels in that portion of the method that was explained in the earlier lesson titled An Adaptive Whitening Filter in Java.

```
    for(int cnt = 0;cnt <
magnitude.length;cnt++){
      plot.feedData(magnitude[cnt]);
    }//end for loop

  }//end displayFreqResponse
```

Listing 19

Listing 19 also signals the end of the method named **displayFreqResponse**.

The method named displaySpectrum

The method named **displaySpectrum** is used to compute and to display the results of the conventional spectrum analysis that is performed on the raw signal plus noise data. You can view the method in its entirety in Listing 20 near the end of the lesson.

The code in this method is similar to, but is not identical to the code in the method named **displayFreqResponse** that was explained above.

The major differences between the two methods are:

- The conversion to decibels is disabled in the method named **displaySpectrum**. However, the required code to convert the spectral data to decibels is still there so that you can re-enable it to see the results of conversion of spectral data to decibels if you wish to do so.
- The spectral data values are not turned upside down in the method named **displaySpectrum**. Thus peaks in the spectral energy are displayed as peaks in the plot, as shown in the right panels of Figure 8 and Figure 10.

The method named **displaySpectrum** shown in Listing 20 is completely documented through the use of in-code comments. Therefore, further explanation of the method should not be necessary.

Run the Program

I encourage you to copy the code from the classes in the section titled Complete Program Listings. Compile and execute the programs. Experiment with the code. Make changes to the code, recompile, execute, and observe the results of your changes.

In addition to the classes named **Adapt05** and **PlotALot08** *(for which the source code is provided in this lesson),* you will need access to the following classes. The source code for these classes can be found in the lessons indicated.

- ForwardRealToComplex01: Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
- AdaptEngine01: A General-Purpose LMS Adaptive Engine in Java
- AdaptiveResult: A General-Purpose LMS Adaptive Engine in Java

Summary

In this lesson, I showed you how to use the general-purpose LMS adaptive engine from a previous lesson to write an adaptive line tracker in Java.

What's Next?

Future lessons in this series will become somewhat more general. I plan to publish lessons that explain and provide examples of four common scenarios in which adaptive filtering is used:

- System Identification
- Inverse System Identification
- Noise Cancellation
- Prediction

Somewhere along the way I may also publish a lesson that explains and illustrates the difference between *least mean square (LMS)* and *recursive least squares (RLS)* adaptive algorithms.

References

In preparation for understanding the material in this lesson, I recommend that you study the material in the following previously-published lessons:

- 100 Periodic Motion and Sinusoids
- 104 Sampled Time Series
- 108 Averaging Time Series
- 1478 Fun with Java, How and Why Spectral Analysis Works
- 1482 Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
- 1483 Spectrum Analysis using Java, Frequency Resolution versus Data Length
- 1484 Spectrum Analysis using Java, Complex Spectrum and Phase Angle
- 1485 Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain
- 1487 Convolution and Frequency Filtering in Java
- 1488 Convolution and Matched Filtering in Java
- 1492 Plotting Large Quantities of Data using Java
- 2350 Adaptive Filtering in Java, Getting Started
- 2352 An Adaptive Whitening Filter in Java
- 2354 A General-Purpose LMS Adaptive Engine in Java

Links to the lessons in the above list can be found at Digital Signal Processing-DSP.

Complete Program Listings

Complete listings of the classes discussed in this lesson are shown in the listings below.

```
/*File Adapt05.java
Copyright 2005, R.G.Baldwin

The purpose of the program is to use the general purpose
adaptive engine provided by AdaptEngine01 to implement an
adaptive spectral line tracker. The line tracker is
designed to track the frequency of frequency-modulated
```

```
signals buried in wide-band noise. Adaptive processing
takes place in the time domain.  The signals are tracked
in the frequency domain. Experimental results produced by
the adaptive line tracker are compared with results
produced by conventional spectrum analysis.

The program develops a whitening filter and uses the
notches in the whitening filter as an indication of the
frequency of the FM signals at equal intervals in time.

Demonstration of the program capability is accomplished by
processing time series consiting of wide-band noise plus FM
signals.

Three different experimental cases can be specified by the
user:

1. A single FM sweep from a low frequency to a high
frequency. The user specifies the rate at which the signal
sweeps.

2. A single FM signal that switches back and forth between
two frequencies.  The user specifies each of the
frequencies.

3. An additive combination of the two cases described
above.

User input is provided by way of command-line parameters.
The command-line parameters are:

double feedbackGain: This is the multiplicative factor that
is used in the feedback loop of the LMS adaptive algorithm.

int numberIterations: This is the number of iterations that
the adaptive algorithm is allowed to execute before
terminating and displaying the results.

int filterLength: This is the length of the filter that is
developed within the adaptive algorithm. Note that this
length is one less than the length of the whitening filter
mentioned above.  The whitening filter consists of this
filter with a -1 concatenated onto its end.

double wideBandNoiseScale: This is a scale factor that is
applied to the wide band noise before it is added to the
FM signal.

double fmSignalScale: This is a scale factor that is
applied to each FM signal before it is added to the
wide-band noise.

int fmSignalCase: This the test case described above.  Must
be 1, 2, or 3.

double freqSlideConst: This value specifies the rate at
```

```
which the FM sweep signal changes frequency. The higher the
value of this parameter, the faster will be the change in
frequency.

double freqShiftFactorLow: The base frequency for the
frequency switching signal is one-fourth of the sampling
frequency. This multiplicative factor is applied to the
base frequency to establish the low frequency for the
frequency-switching signal.  For example, a value of 0.5
for this parameter results in a frequency that is
one-eighth of the sampling frequency.

double freqShiftFactorHigh: This parameter is applied as a
multiplicative factor to the base frequency to establish
the upper frequency for the frequency shifting signal.

int lengthMultiplier:  This parameter specifies the length
of the chunks of data that will be analyzed using
conventional spectral analysis.  This value is a multiple
of the length of the whitening filter.

If the user doesn't provide ten command-line parameters,
a set of default values is used.  See the default-value
comments in the code for an indication of the approximate
values that might be appropriate for any particular
parameter.

The program puts a marker at zero frequency in each
spectral plot. This makes it possible to visually confirm
that the spectral plots are properly synchronized, with one
spectral plot above the other.

Tested using J2SE 5.0 and WinXP.  J2SE 5.0 or later is
required.
**********************************************************/
import static java.lang.Math.*;//J2SE 5.0 req

class Adapt05{

  public static void main(String[] args){
    //Default parameter values.  See a description of each
    // of these parameters in the opening comments above.
    // Note that this set of default values represents a
    // high signal-to-noise ratio.
    double feedbackGain = 0.00001;
    int numberIterations = 3375;
    int filterLength = 15;
    double wideBandNoiseScale = 1.0;
    double fmSignalScale = 20.0;
    int fmSignalCase = 3;
    double freqSlideConst = 0.0004;
    double freqShiftFactorLow = 0.5;
    double freqShiftFactorHigh = 1.5;
    int lengthMultiplier = 2;

    int spectralWidth = 222;//Not a user input value
    if(args.length != 10){
      System.out.println(
               "Usage with all parameters following the " +
               "program name:\n" +
               "java Adapt05\n" +
               "feedbackGain\n" +
               "numberIterations\n" +
               "filterLength\n" +
               "wideBandNoiseScale\n" +
               "fmSignalScale\n" +
               "fmSignalCase\n" +
               "freqSlideConst\n" +
               "freqShiftFactorLow\n" +
               "freqShiftFactorHigh\n" +
               "lengthMultiplier\n");

      System.out.println(
          "Input values were not provided.\n"+
          "Using following values:\n" +
          "feedbackGain: " + feedbackGain +
          "\nnumberIterations: " + numberIterations +
          "\nfilterLength: " + filterLength +
          "\nwideBandNoiseScale: " + wideBandNoiseScale +
          "\nfmSignalScale: " + fmSignalScale +
          "\nfmSignalCase: " + fmSignalCase +
          "\nfreqSlideConst: " + freqSlideConst +
          "\nfreqShiftFactorLow: " + freqShiftFactorLow +
          "\nfreqShiftFactorHigh: " + freqShiftFactorHigh +
          "\nspectralWidth: " + spectralWidth +
          "\nlengthMultiplier: " + lengthMultiplier +
          "\nConventional data length: " +
                         (lengthMultiplier * filterLength +
                                        lengthMultiplier));
    }else{//Command line params were provided.
      feedbackGain = Double.parseDouble(args[0]);
      numberIterations = Integer.parseInt(args[1]);
      filterLength = Integer.parseInt(args[2]);
      wideBandNoiseScale = Double.parseDouble(args[3]);
      fmSignalScale = Double.parseDouble(args[4]);
      fmSignalCase = Integer.parseInt(args[5]);
      freqSlideConst = Double.parseDouble(args[6]);
      freqShiftFactorLow = Double.parseDouble(args[7]);
      freqShiftFactorHigh = Double.parseDouble(args[8]);
      lengthMultiplier = Integer.parseInt(args[9]);

      System.out.println(
          "Input values were provided.\n"+
          "Using following values:\n" +
          "feedbackGain: " + feedbackGain +
          "\nnumberIterations: " + numberIterations +
          "\nfilterLength: " + filterLength +
          "\nwideBandNoiseScale: " + wideBandNoiseScale +
          "\nfmSignalScale: " + fmSignalScale +
          "\nfmSignalCase: " + fmSignalCase +
          "\nfreqSlideConst: " + freqSlideConst+
          "\nfreqShiftFactorLow: " + freqShiftFactorLow +
          "\nfreqShiftFactorHigh: " + freqShiftFactorHigh +
          "\nspectralWidth: " + spectralWidth +
          "\nlengthMultiplier: " + lengthMultiplier +
          "\nConventional data length: " +
                         (lengthMultiplier * filterLength +
                                        lengthMultiplier));
    }//end else

    //Instantiate a new object of the Adapt05 class and
    // invoke the method named process on that object.
    new Adapt05().process(feedbackGain,
                          numberIterations,
                          filterLength,
                          wideBandNoiseScale,
                          fmSignalScale,
                          fmSignalCase,
                          freqSlideConst,
                          freqShiftFactorLow,
                          freqShiftFactorHigh,
                          spectralWidth,
                          lengthMultiplier);
  }//end main
  //-------------------------------------------------------//

  //This is the primary processing and plotting method for
  // the program.  This method uses an object of the
  // AdaptEngine01 class to provide the adaptive behavior.
  void process(double feedbackGain,
               int numberIterations,
               int filterLength,
               double wideBandNoiseScale,
               double fmSignalScale,
               int fmSignalCase,
               double freqSlideConst,
               double freqShiftFactorLow,
               double freqShiftFactorHigh,
               int spectralWidth,
               int lengthMultiplier){

    //Declare and initialize working variables.
    double err = 0;
    double wideBandNoise = 0;
    double fmSignal = 0;
    double freqSlideValue = 0;
    double freqShiftFactor = freqShiftFactorLow;

    //Create an array to contain the whitening filter. The
    // actual whitening of the data is accomplished within
    // the object of type AdaptEngine01. That object
    // returns the prediction portion of the whitening
    // filter, but does not return an actual whitening
    // filter. This copy of the whitening filter is
    // required solely for the purpose of computing and
    // displaying the frequency response of the whitening
    // filter.  Note that the length of the whitening
```

```
    // filter is one greater than the length of the filter
    // that is returned by the adaptive object. The extra
    // coefficient in the whitening filter is set to a
    // value of -1.
    double[] whiteningFilter =
                          new double[filterLength + 1];
    //Set the last coefficient value in the whitening
    // filter to -1. All other values are initialized to
    // zero. Coefficient values returned by the adaptive
    // process will be copied into the lower elements of
    // thewhitening filter later.
    whiteningFilter[filterLength] = -1;

    //Create an array to contain two samples of the data to
    // be adaptively processed.  This array is used as a
    // tapped delay line. The data sample to be filtered is
    // located at index 0.  The value of the adaptive
    // target is located at index 1.
    double[] data = new double[2];

    //Create an array to serve as a delay line to contain a
    // chunk of raw data that will be used for conventional
    // spectral analysis. Make the length of the data an
    // integer multiple of the length of the whitening
    // filter.
    double[] rawData = new double[
              lengthMultiplier*whiteningFilter.length];

    //Instantiate a general purpose adaptive processing
    // object. This object provides the adaptive behavior
    // for the entire program.
    AdaptEngine01 theAdapter =
              new AdaptEngine01(filterLength,feedbackGain);

    //Instantiate a plotting object to display whitening
    // filter frequency response data.
    PlotALot08 freqPlotObj = new PlotALot08(
                                     "Adaptive",
                                     spectralWidth + 8,
                                     487,
                                     10,
                                     1,
                                     0,
                                     0);
    //Instantiate a plotting object to display the results
    // of conventional spectrum analysis.
    PlotALot08 conventionalPlotObj = new PlotALot08(
                                     "Conventional",
                                     spectralWidth + 8,
                                     487,
                                     10,
                                     1,
                                     0,
                                     0);

    //Perform the specified number of iterations
    for(int cnt = 0;cnt < numberIterations;cnt++){
      //Generate the synthetic wideBandNoise and fmSignal
      // data.

      //Get the next sample of wideBandNoise with a
      // uniform distribution from -1.0 to +1.0.  Scale the
      // noise by the specified scale factor. Note the use
      // of a static import directive for the Math class,
      // which requires J2SE 5.0 or later.
      wideBandNoise =
                 wideBandNoiseScale*(2.0*(random() - 0.5));

      //Get the next sample of fmSignal data. The contents
      // of the following variable set the frequency for
      // the FM sweep signal for this sample. This value
      // increases for each successive iteration.
      freqSlideValue += freqSlideConst;

      //Use the value of fmSignalCase to determine which
      // configuration of FM signal to generate.
      if(fmSignalCase == 1){
        //Single source, sweep frequency
        fmSignal = fmSignalScale*(sin(
                         cnt*(freqSlideValue + 2*PI/32)));
      }else if(fmSignalCase == 2){
        //Single source, shift frequency.  Frequency shifts
        // every 1650 iterations.
        if(cnt%1650 == 0){
          if(freqShiftFactor == freqShiftFactorLow){
            freqShiftFactor = freqShiftFactorHigh;
          }else{
            freqShiftFactor = freqShiftFactorLow;
          }//end else
        }//end if
        fmSignal = fmSignalScale*sin(
                              freqShiftFactor*cnt*2*PI/4);
      }else if(fmSignalCase == 3){
        //Two signal sources with one of each of the above
        // configurations.
        if(cnt%1650 == 0){
          if(freqShiftFactor == freqShiftFactorLow){
            freqShiftFactor = freqShiftFactorHigh;
          }else{
            freqShiftFactor = freqShiftFactorLow;
          }//end else
        }//end if
        fmSignal = fmSignalScale*(sin(
                         cnt*(freqSlideValue + 2*PI/32)) +
                         sin(freqShiftFactor*cnt*2*PI/4));
      }else{
        System.out.println(
            "Incorrect signal case, terminating");
        System.exit(0);
      }//end else

      //Insert the wideBandNoise plus fmSignal into the
      // two-element delay line.
      flowLine(data,wideBandNoise+fmSignal);

      //Insert signal plus noise into delay line used for
      // conventional spectrum analysis.
      flowLine(rawData,wideBandNoise+fmSignal);

      //Execute the adaptive whitening process. Pass the
      // two samples to the adapt method, one as the data
      // to be filtered and the other as the predictive
      // target. This one statement is responsible for all
      // of the adaptive behavior of the program.
      AdaptiveResult result =
                         theAdapter.adapt(data[0],data[1]);

      //Compute and plot the frequency response and the
      // conventional spectrum every 75 iterations.

      //Construct the whitening filter by copying the
      // prediction filter that was returned by the
      // adapter into the lower elements of the whitening
      // filter array, leaving the value of -1 in the last
      // element.
      System.arraycopy(result.filterArray,
                       0,
                       whiteningFilter,
                       0,
                       filterLength);

      if(cnt%75 == 0){
        //Compute and display the frequency response of the
        // whitening filter.
        displayFreqResponse(whiteningFilter,
                            freqPlotObj,
                            spectralWidth,
                            whiteningFilter.length - 1);

        //Compute and display the conventional spectrum of
        // a chunk of raw signal plus noise data.
        displaySpectrum(rawData,
                        conventionalPlotObj,
                        spectralWidth,
                        rawData.length - 1);
      }//End display of frequency data
    }//End for loop, End adaptive process

    //Cause all the data to be plotted.
    freqPlotObj.plotData(0,0);
    conventionalPlotObj.plotData(232,0);

  }//end process method
  //-----------------------------------------------------//

  //This method simulates a tapped delay line. It receives
  // a reference to an array and a value.  It discards the
  // value at index 0 of the array, moves all the other
  // values by one element toward 0, and inserts the new
  // value at the top of the array.
  static void flowLine(double[] line,double val){
    for(int cnt = 0;cnt < (line.length - 1);cnt++){
      line[cnt] = line[cnt+1];
    }//end for loop
    line[line.length - 1] = val;
  }//end flowLine
  //-----------------------------------------------------//

  //This method is used to compute and display the
  // amplitude frequency response of an incoming whitening
  // filter.
  void displayFreqResponse(
     double[] filter,PlotALot08 plot,int len,int zeroTime){

    //Create the arrays required by the Fourier Transform.
    double[] timeDataIn = new double[len];
    double[] realSpect = new double[len];
    double[] imagSpect = new double[len];
    double[] angle = new double[len];
    double[] magnitude = new double[len];

    //Copy the filter into the timeDataIn array
    System.arraycopy(filter,0,timeDataIn,0,filter.length);

    //Compute DFT of the filter from zero to the folding
    // frequency and save it in the output arrays.
    ForwardRealToComplex01.transform(timeDataIn,
                                     realSpect,
                                     imagSpect,
                                     angle,
                                     magnitude,
                                     zeroTime,
                                     0.0,
                                     0.5);

    //Display the magnitude data. Convert to normalized
    // decibels first.
    //Eliminate or change any values that are incompatible
    // with log10 method.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if((magnitude[cnt] == Double.NaN) ||
                                    (magnitude[cnt] <= 0)){
        //Replace the magnitude by a very small positive
        // value.
        magnitude[cnt] = 0.0000001;
      }else if(magnitude[cnt] == Double.POSITIVE_INFINITY){
        //Replace the magnitude by a very large positive
        // value.
        magnitude[cnt] = 9999999999.0;
      }//end else if
    }//end for loop

    //Now convert magnitude data to log base 10
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = log10(magnitude[cnt]);
    }//end for loop

    //Note that from this point forward, all references to
    // magnitude are referring to log base 10 data, which
    // can be thought of as scaled decibels.

    //Change the algebraic sign on all the values to turn
    // the response curve upside down.  This causes the
    // notches in the response curve to appear as peaks in
    // the display.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = -magnitude[cnt];
    }//end for loop

    //The purpose of much of the following code is to
    // normalize the amplitude response such that each
    // response curve will occupy the same size horizontal
    // strip in the plot regardless of the magnitude
    // of the input data.

    //Bias the values such that the smallest value becomes
    // zero.
    //First find the smallest value.
    double min = 9999999999.0;
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if(magnitude[cnt] < min){
        min = magnitude[cnt];
      }//end if
    }//end for loop
    //Now apply the bias.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] -= min;
    }//end for loop

    //Find the absolute peak value.  Begin with a negative
    // peak value with a large magnitude and replace it
    // with the largest magnitude value.
    double peak = -9999999999.0;
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if(peak < abs(magnitude[cnt])){
        peak = abs(magnitude[cnt]);
      }//end if
    }//end for loop

    //Set the zero frequency value to the peak so that it
    // can be used to visually confirm plot synchronization
    // later.
    magnitude[0] = peak;

    //Normalize to 20 times the peak value. Each response
    // curve will now occupy a horizontal strip of the
    // plotting area that is 20 pixels from top to bottom.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = 20*magnitude[cnt]/peak;
    }//end for loop
    //Now feed the normalized decibel data to the plotting
    // system.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      plot.feedData(magnitude[cnt]);
    }//end for loop

  }//end displayFreqResponse
  //-----------------------------------------------------//

  //This method is used to compute and display the
  // conventional amplitude spectrum of a chunk of incoming
  // data.
  //The code in this method is similar to, but not
  // identical to the code in the method named
  // displayFreqResponse.
  void displaySpectrum(
     double[] data,PlotALot08 plot,int len,int zeroTime){

    //Create the arrays required by the Fourier Transform.
    double[] timeDataIn = new double[len];
    double[] realSpect = new double[len];
    double[] imagSpect = new double[len];
    double[] angle = new double[len];
    double[] magnitude = new double[len];

    //Copy the data into the timeDataIn array
    System.arraycopy(data,0,timeDataIn,0,data.length);

    //Compute DFT of the data from zero to the folding
    // frequency and save it in the output arrays.
    ForwardRealToComplex01.transform(timeDataIn,
                                     realSpect,
                                     imagSpect,
                                     angle,
                                     magnitude,
                                     zeroTime,
                                     0.0,
                                     0.5);

    //Display the magnitude data. Note that conversion to
    // decibels has been disabled in this version of the
    // method. Enable the following code to re-enable
    // the conversion to decibels.
/*
    //Eliminate or change any values that are incompatible
    // with log10 method.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if((magnitude[cnt] == Double.NaN) ||
                                    (magnitude[cnt] <= 0)){
        //Replace the magnitude by a very small positive
        // value.
        magnitude[cnt] = 0.0000001;
      }else if(magnitude[cnt] == Double.POSITIVE_INFINITY){
        //Replace the magnitude by a very large positive
        // value.
        magnitude[cnt] = 9999999999.0;
      }//end else if
    }//end for loop

    //Now convert magnitude data to log base 10
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = log10(magnitude[cnt]);
    }//end for loop
*/

    //These spectral results are normalized the same way
    // the frequency response curves are normalized as
    // described in the method named displayFreqResponse.
    //Bias the values such that the smallest value becomes
    // zero.
    //First find the smallest value.
    double min = 9999999999.0;
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if(magnitude[cnt] < min){
        min = magnitude[cnt];
      }//end if
    }//end for loop
    //Now apply the bias.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] -= min;
    }//end for loop

    //Find the absolute peak value.  Begin with a negative
    // peak value with a large magnitude and replace it
    // with the largest magnitude value.
    double peak = -9999999999.0;
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if(peak < abs(magnitude[cnt])){
        peak = abs(magnitude[cnt]);
      }//end if
    }//end for loop

    //Set the zero frequency value to the peak so that it
    // can be used to confirm plot synchronization later.
    magnitude[0] = peak;

    //Normalize to 20 times the peak value.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = 20*magnitude[cnt]/peak;
    }//end for loop

    //Now feed the normalized data to the plotting
    // system.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      plot.feedData(magnitude[cnt]);
    }//end for loop

  }//end displaySpectrum
  //-----------------------------------------------------//
}//end class Adapt05
//=====================================================//
```

Listing 20

```
/*File PlotALot08.java
Copyright 2005, R.G.Baldwin
This is an update to the program named
PlotALot01.
The purpose of this update is to eliminate the
drawing of the horizontal axes on the plot.
Otherwise, it is identical to PlotALot01.

This program is designed to plot large amounts of
time-series data for a single channel.  See
PlotALot02.java for a two-channel program.

Note that by carefully adjusting the plotting
parameters, this program could also be used to
plot large quantities of spectral data in a
waterfall display.

The class provides a main method so that the
class can be run as an application to test
itself.

There are three steps involved in the use of this
class for plotting time series data:
1. Instantiate a plotting object of type
   PlotALot08 using one of two overloaded
   constructors.
2. Feed data that is to be plotted to the
   plotting object by invoking the feedData
   method once for each data value.
3. Invoke one of two overloaded plotData methods
   on the plotting object once all of the data
   has been fed to the object.  This causes all
   of the data to be plotted.

A using program can instantiate as many
plotting objects as are needed to plot all of the
different time series that need to be plotted.
Each plotting object can be used to plot as many
data values as need be plotted until the program
runs out of available memory.

The plotting object of type PlotALot08 owns one
or more Page objects that extend the Frame class.
The plotting object can own as many Page objects
as are necessary to plot all of the data that is
fed to that plotting object.

The program produces a graphic output consisting
of a stack of Page objects on the screen, with
the data plotted on a Canvas object contained by
the Page object.  The Page showing the earliest
data is on the top of the stack and the Page
showing the latest data is on the bottom of the
stack.  The Page objects on the top of the stack
must be physically moved in order to see the
Page objects on the bottom of the stack.

Each Page object contains one or more horizontal
axes on which the data is plotted.  The earliest
data is plotted on the axis nearest the top of
the Page moving from left to right across the
axis.  Positive data values are plotted above
the axis and negative values are plotted below
the axis.  When the right end of an axis is
reached, the next data value is plotted on the
left end of the axis immediately below it.  When
the right end of the last axis on the Page is
reached, a new Page object is created and the
next data value is plotted at the left end of the
top axis on that Page object.

A mentioned above, there are two overloaded
versions of the constructor for the PlotALot08
class. One overloaded version accepts several
incoming parameters allowing the user to control
various aspects of the plotting format. A second
overloaded version accepts a title string only
and sets all of the plotting parameters to
default values. You can easily modify these
default values and recompile the class if you
prefer different default values.

The parameters for the version of the constructor
that accepts plotting format information are:

String title: Title for the Frame object. This
 title is concatenated with the page number and
 the result appears in the banner at the top of
 the Frame.
int frameWidth:The Frame width in pixels.
int frameHeight: The Frame height in pixels.
int traceSpacing: Distance between trace axes in
 pixels.
int sampSpace: Number of pixels dedicated to each
 data sample in pixels per sample.  Must be 1 or
 greater.
int ovalWidth: Width of an oval that is used to
 mark the sample value on the plot.
int ovalHeight: Height of an oval that is used to
 mark the sample value on the plot.

For test purposes, the main method instantiates
and feeds two independent plotting objects.
Plotting parameters are specified for the first
plotting object. Default plotting parameters are
accepted for the second plotting object.
```

The data that is fed to each plotting object is white random noise. However, for the first plotting object, fifteen of the data values are not random. Rather, seven of the values are set to values of 0,0,25,-25,25,0,0 to confirm the proper transition from the end of one page to the beginning of the next page. In addition, eight of the values are set to 0,0,20,20,-20,-20,0,0 in order to confirm the proper transition from one trace to the next trace on the same page.

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly. Note, however that these are the correct locations for an AWT Frame object under WinXP. A Frame may have different inset values under other operating systems, which may cause these specific locations to be incorrect for that operating system. In that case, the values will be plotted but they won't confirm the proper transition.

The following information about the plotting parameters for each plotting object is displayed on the command line screen when the class is used for plotting. The values shown below result from the execution of the main method of the class for test purposes. One of the plotting objects instantiated by the main method is titled "A" and the other is titled "B".

```
Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
Traces per page: 5
Samples per page: 150

Title: B
Frame width: 400
Frame height: 410
Page width: 392
Page height: 383
Trace spacing: 50
Sample spacing: 2
Traces per page: 7
Samples per page: 1372
```

There are two overloaded versions of the plotData method. One version allows the user to specify the location on the screen where the stack of

```
plotted pages will appear. This version requires
two parameters, which are coordinate values in
pixels.  The first parameter specifies the
horizontal coordinate of the upper left corner of
the stack of pages relative to the upper left
corner of the screen.  The second parameter
specifies the vertical coordinate of the upper
left corner of the stack of pages relative to the
upper left corner of the screen. Specifying
coordinate values of 0,0 causes the stack to be
located in the upper left corner of the screen.

The other overloaded version of plotData places
the stack of pages in the upper left corner of
the screen by default.

Each page has a WindowListener that will
terminate the program if the user clicks the
close button on the Frame.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.
*************************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot08{
  //This main method is provided so that the
  // class can be run as an application to test
  // itself.
  public static void main(String[] args){
    //Instantiate two independent plotting
    // objects.  Control plotting parameters for
    // the first object.  Accept default plotting
    // parameters for the second object.
    PlotALot08 plotObjectA =
            new PlotALot08("A",158,237,36,5,4,4);
    PlotALot08 plotObjectB = new PlotALot08("B");

    //Feed the data to the first plotting object.
    for(int cnt = 0;cnt < 275;cnt++){
      //Plot some white random noise in the first
      // object using specified plotting
      // parameters. Note, that fifteen of the
      // following values are not random.  Seven
      // values are set to 0,0,25,-25,25,0,0
      // specifically to confirm the proper
      // transition from the end of one page to
      // the beginning of the next page.  Eight
      // values are set to 0,0,20,20,-20,-20,0,0
      // to confirm the proper transition from
      // one trace to the next trace on the same
      // page.  Note that these are the correct
      // values for an AWT Frame object under
      // WinXP.  However, a Frame may have
      // different inset values on other
      // operating systems, which may cause these
      // specific values to be incorrect.
      if(cnt == 147){
        plotObjectA.feedData(0);
      }else if(cnt == 148){
        plotObjectA.feedData(0);
      }else if(cnt == 149){
        plotObjectA.feedData(25);
      }else if(cnt == 150){
        plotObjectA.feedData(-25);
      }else if(cnt == 151){
        plotObjectA.feedData(25);
      }else if(cnt == 152){
        plotObjectA.feedData(0);
      }else if(cnt == 153){
        plotObjectA.feedData(0);
      }else if(cnt == 26){
        plotObjectA.feedData(0);
      }else if(cnt == 27){
        plotObjectA.feedData(0);
      }else if(cnt == 28){
        plotObjectA.feedData(20);
      }else if(cnt == 29){
        plotObjectA.feedData(20);
      }else if(cnt == 30){
        plotObjectA.feedData(-20);
      }else if(cnt == 31){
        plotObjectA.feedData(-20);
      }else if(cnt == 32){
        plotObjectA.feedData(0);
      }else if(cnt == 33){
        plotObjectA.feedData(0);
      }else{
        plotObjectA.feedData(
                         (Math.random() - 0.5)*25);
      }//end else
    }//end for loop
    //Cause the data to be plotted.
    plotObjectA.plotData(401,0);

    //Plot white random noise in the second
    // plotting object using default plotting
    // parameters.
    //Feed the data to the second plotting
    // object.
    for(int cnt = 0;cnt < 2600;cnt++){
      plotObjectB.feedData(
                         (Math.random() - 0.5)*25);
    }//end for loop
    //Cause the data to be plotted.
    plotObjectB.plotData();

  }//end main
  //--------------------------------------------//
```

```
String title;
int frameWidth;
int frameHeight;
int traceSpacing;//pixels between traces
int sampSpacing;//pixels between samples
int ovalWidth;//width of sample marking oval
int ovalHeight;//height of sample marking oval

int tracesPerPage;
int samplesPerPage;
int pageCounter = 0;
int sampleCounter = 0;
ArrayList <Page> pageLinks =
                         new ArrayList<Page>();

//There are two overloaded versions of the
// constructor for this class.  This
// overloaded version accepts several incoming
// parameters allowing the user to control
// various aspects of the plotting format. A
// different overloaded version accepts a title
// string only and sets all of the plotting
// parameters to default values.
PlotALot08(String title,//Frame title
           int frameWidth,//in pixels
           int frameHeight,//in pixels
           int traceSpacing,//in pixels
           int sampSpace,//in pixels per sample
           int ovalWidth,//sample marker width
           int ovalHeight)//sample marker hite
{//constructor
  //Specify sampSpace as pixels per sample.
  // Should never be less than 1.  Convert to
  // pixels between samples for purposes of
  // computation.
  this.title = title;
  this.frameWidth = frameWidth;
  this.frameHeight = frameHeight;
  this.traceSpacing = traceSpacing;
  //Convert to pixels between samples.
  this.sampSpacing = sampSpace - 1;
  this.ovalWidth = ovalWidth;
  this.ovalHeight = ovalHeight;

  //The following object is instantiated solely
  // to provide information about the width and
  // height of the canvas. This information is
  // used to compute a variety of other
  // important values.
  Page tempPage = new Page(title);
  int canvasWidth = tempPage.canvas.getWidth();
  int canvasHeight =
                 tempPage.canvas.getHeight();
  //Display information about this plotting
  // object.
```

```
    System.out.println("\nTitle: " + title);
    System.out.println(
          "Frame width: " + tempPage.getWidth());
    System.out.println(
        "Frame height: " + tempPage.getHeight());
    System.out.println(
                   "Page width: " + canvasWidth);
    System.out.println(
                 "Page height: " + canvasHeight);
    System.out.println(
               "Trace spacing: " + traceSpacing);
    System.out.println(
         "Sample spacing: " + (sampSpacing + 1));
    if(sampSpacing < 0){
      System.out.println("Terminating");
      System.exit(0);
    }//end if
    //Get rid of this temporary page.
    tempPage.dispose();
    //Now compute the remaining important values.
    tracesPerPage =
                 (canvasHeight - traceSpacing/2)/
                                   traceSpacing;
    System.out.println("Traces per page: "
                               + tracesPerPage);
    if(tracesPerPage == 0){
      System.out.println("Terminating program");
      System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                              (sampSpacing + 1);
    System.out.println("Samples per page: "
                              + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
  }//end constructor
  //-------------------------------------------//

  PlotALot08(String title){
    //Invoke the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
  }//end overloaded constructor
  //-------------------------------------------//

  //Invoke this method for each point to be
  // plotted.
  void feedData(double val){
    if((sampleCounter) == samplesPerPage){
      //if the page is full, increment the page
      // counter, create a new empty page, and
      // reset the sample counter.
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if
    //Store the sample value in the MyCanvas
    // object to be used later to paint the
    // screen.  Then increment the sample
    // counter.  The sample value passes through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
                              val,sampleCounter);
    sampleCounter++;
  }//end feedData
  //--------------------------------------------//

  //There are two overloaded versions of the
  // plotData method.  One version allows the
  // user to specify the location on the screen
  // where the stack of plotted pages will
  // appear.  The other version places the stack
  // in the upper left corner of the screen.

  //Invoke one of the overloaded versions of
  // this method once when all of the data has
  // been fed to the plotting object in order to
  // rearrange the order of the pages with
  // page 0 at the top of the stack on the
  // screen.

  //For this overloaded version, specify xCoor
  // and yCoor to control the location of the
  // stack on the screen.  Values of 0,0 will
  // place the stack at the upper left corner of
  // the screen.  Also see the other overloaded
  // version, which places the stack at the upper
  // left corner of the screen by default.
  void plotData(int xCoor,int yCoor){
    Page lastPage =
             pageLinks.get(pageLinks.size() - 1);
    //Delay until last page becomes visible.
    while(!lastPage.isVisible()){
      //Loop until last page becomes visible
    }//end while loop

    Page tempPage = null;
    //Make all pages invisible
    for(int cnt = 0;cnt < (pageLinks.size());
                                          cnt++){
      tempPage = pageLinks.get(cnt);
      tempPage.setVisible(false);
    }//end for loop

    //Now make all pages visible in reverse order
    // so that page 0 will be on top of the
    // stack on the screen.
    for(int cnt = pageLinks.size() - 1;cnt >= 0;
                                        cnt--){
      tempPage = pageLinks.get(cnt);
      tempPage.setLocation(xCoor,yCoor);
      tempPage.setVisible(true);
    }//end for loop

  }//end plotData(int xCoor,int yCoor)
  //-------------------------------------------//

  //This overloaded version of the method causes
  // the stack to be located in the upper left
  // corner of the screen by default
  void plotData(){
    plotData(0,0);//invoke overloaded version
  }//end plotData()
  //-------------------------------------------//

  //Inner class.  A PlotALot08 object may
  // have as many Page objects as are required
  // to plot all of the data values.  The
  // reference to each Page object is stored
  // in an ArrayList object belonging to the
  // PlotALot08 object.
  class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
      canvas = new MyCanvas();
      add(canvas);
      setSize(frameWidth,frameHeight);
      setTitle(title + " Page: " + pageCounter);
      setVisible(true);

      //---------------------------------------//
      //Anonymous inner class to terminate the
      // program when the user clicks the close
      // button on the Frame.
      addWindowListener(
        new WindowAdapter(){
          public void windowClosing(
                                WindowEvent e){
            System.exit(0);//terminate program
          }//end windowClosing()
        }//end WindowAdapter
      );//end addWindowListener
      //---------------------------------------//
    }//end constructor
    //=========================================//

    //This method receives a sample value of type
    // double and stores it in an array object
    // belonging to the MyCanvas object.
    void putData(double sampleValue,
                 int sampleCounter){
      canvas.data[sampleCounter] = sampleValue;
      //Save the sample counter in an instance
      // variable to make it available to the
      // overridden paint method. This value is
      // needed by the paint method so it will
      // know how many samples to plot on the
      // final page which probably won't be full.
      this.sampleCounter = sampleCounter;
    }//end putData

    //===========================================//
    //Inner class
    class MyCanvas extends Canvas{
      double [] data =
                      new double[samplesPerPage];

      //Override the paint method
      public void paint(Graphics g){
/*
Eliminate the drawing of horizontal axes for this
version of the program.
        //Draw horizontal axes, one for each
        // trace.
        for(int cnt = 0;cnt < tracesPerPage;
                                          cnt++){
          g.drawLine(0,
                     (cnt+1)*traceSpacing,
                     this.getWidth(),
                     (cnt+1)*traceSpacing);
        }//end for loop
*/
        //Plot the points if there are any to be
        // plotted.
        if(sampleCounter > 0){
          for(int cnt = 0;cnt <= sampleCounter;
                                          cnt++){
            //Compute a vertical offset to locate
            // the data on a particular trace.
            int yOffset =
                   (1 + cnt*(sampSpacing + 1)/
                   this.getWidth())*traceSpacing;
            //Draw an oval centered on the sample
            // value to mark the sample.  It is
            // best if the dimensions of the oval
            // are evenly divisable by 2 for
            // centering purposes.
            //Reverse the sign on sample value to
            // cause positive sample values to go
            // up on the screen
            g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() - ovalWidth/2,
              yOffset - (int)data[cnt]
                                  - ovalHeight/2,
              ovalWidth,
              ovalHeight);

            //Connect the sample values with
            // straight lines.  Do not draw a
            // line connecting the last sample in
            // one trace to the first sample in
            // the next trace.
            if(cnt*(sampSpacing + 1)%
                                   this.getWidth() >=
                                    sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)data[cnt-1],
                cnt*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)data[cnt]);
            }//end if
          }//end for loop
        }//end if for sampleCounter > 0
      }//end overridden paint method
    }//end inner class MyCanvas
  }//end inner class Page
}//end class PlotALot08
//============================================//
```

Listing 21

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 have 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

Keywords
Java adaptive filtering convolution filter frequency spectrum LMS amplitude phase time-delay linear DSP impulse decibel log10 DFT transform bandwidth signal noise real-time dot-product vector time-series prediction whitening

-end-