A General-Purpose LMS Adaptive Engine in Java

Learn how to write a general-purpose LMS adaptive engine in Java, and how to demonstrate the use of the engine for three different adaptive programs of increasing complexity.

Published: November 29, 2005
by Richard G. Baldwin

Java Programming Notes # 2354

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 situation where the characteristics of the digital processor change with time, circumstances, or both.

Third in a series

This is the third lesson in a series designed to teach you about adaptive filtering in Java.

Getting started

The first lesson, entitled 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. That was a relatively simple time-adaptive filtering problem for which the correct solution was well known in advance. That made it possible to check the adaptive solution against the known solution.

An adaptive whitening filter

The second lesson in the series, entitled An Adaptive Whitening Filter in Java showed you how to write an adaptive *whitening filter* program in Java. That project was conceptually more difficult than the filter that I explained in the first lesson.

That lesson also 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.

Will backtrack a bit

I will backtrack a bit in this lesson. 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.

A general-purpose LMS adaptive engine

In this lesson, I will present and explain a general-purpose LMS adaptive engine written in Java that can be used to solve a wide variety of adaptive problems.

To illustrate how this adaptive engine can be used to solve different problems, I will begin by applying the adaptive engine to the simplest adaptive problem that I can devise.

Following that, I will apply the adaptive engine to more complex problems similar to the problems that were addressed in the first two lessons in the series.

In all three cases, I will completely separate the part of the program that exhibits adaptive behavior from the part of the program that is used simply to manage data and to display results.

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

General Background Information

The LMS adaptive engine

The LMS adaptive engine is deceptively simple. In fact, once you have an LMS adaptive engine available, the use of that engine reminds me of an old story about an automobile mechanic and a screw.

The automobile mechanic and the screw

As the story goes, a customer watched the mechanic doing his job and then complained that the charges were two high for the small amount of effort that the mechanic had spent turning a screw to fix the problem. The mechanic's response was that he doesn't get paid for turning the screw, he gets paid for knowing which screw to turn, why that particular screw should be turned, and how far to turn it.

Deceptively simple

Working with the adaptive engine is a similar situation. As mentioned above, the LMS adaptive engine is deceptively simple. The same engine can be used to solve a wide variety of different adaptive problems. Once you know the mechanics of how the engine behaves, the real challenge is in knowing how and what to feed to the engine in order to solve a particular adaptive problem.

AdaptEngine01

The adaptive engine that I will provide in this lesson is an object of the class named **AdaptEngine01**. *(You can view a complete listing of that class in Listing 15 near the end of the lesson.)* An object of this class is a general purpose adaptive engine that implements a classical LMS adaptive algorithm.

The adapt method

The adaptive algorithm is implemented by the instance method named **adapt** belonging to an **AdaptEngine01** object. Each time the adapt method is called it receives one sample from each of two different time series. One time series is considered to be data that is to be filtered using an evolving convolution filter. The other time series is considered to be an adaptive target.

Transform the data into the target

The purpose of the **adapt** method is to adaptively create a convolution filter which, when applied to the data time series, will transform it into the target time series.

Adjusting the coefficients

Each time the **adapt** method is called it performs a dot product between the current version of the filter and the contents of a delay line in which historical data samples have been saved. The result of that dot product is compared with the target sample to produce an error value.

> *(The error value is produced by subtracting the value of the target sample from the result of the dot product.)*

The error value is then used in a classical LMS adaptive algorithm to adjust the coefficients of the convolution filter.

> *(The adjustment process is very straightforward. However, rather than to attempt to explain the mechanics of the adjustment process at this point, I will explain it later while discussing the actual code.)*

Drive the error to zero

The purpose of the adjustment is to produce a set of filter coefficients that will drive the error to zero over time. If the error goes to zero, the time series representing the data will have been transformed into the time series representing the target.

The constructor

The constructor for the **AdaptEngine01** class receives two parameters:

- filterLength
- feedbackGain

The **filterLength** value is used to construct two arrays whose size is equal to the value of **filterLength**. One array is used later to contain the filter coefficients. The other array is used later as a tapped delay line to contain the historical data samples and to shift them by one element each time the method is called.

The delay line

In other words, each time the **adapt** method is called, each value currently stored in the delay line is shifted by one element and the new data value is inserted at the end of the delay line.

The feedback gain

The **feedbackGain** value is used in the LMS adaptive algorithm to compute the new filter coefficients.

> *(You will see how the value of **feedbackGain** is used in the computation of the updated filter coefficients later.)*

The returned values

In order to make it possible for the **adapt** method to return more than one value each time it is called, the source code in Listing 15 defines a simple wrapper class named **AdaptiveResult**. Each time the **adapt** method is called, it instantiates, populates, and returns a reference to an object of type **AdaptiveResult**. This object is populated with the following values:

- **double[] filterArray**: A reference to the array object containing the current filter coefficients.
- **double output**: The result of performing the dot product between the filter and the historical data.
- **double err**: The result of comparing the output of the dot product with the target value.

Different adaptive applications will make use of none, one, or more of the returned values.

Program testing

All of the classes provided in this lesson were tested using J2SE 5.0 and WinXP. J2SE 5.0 or later is required.

Preview

AdaptEngine01

I will begin by presenting and explaining the code for the general-purpose adaptive engine class named **AdaptEngine01**, as explained above. You can view a complete listing of this class in Listing 15.

Adapt06

Then I will present and explain the code for the simplest sample program that I was able to devise to illustrate the use of the adaptive engine. This program is named **Adapt06**. You can view a complete listing of this program in Listing 16 near the end of the lesson. In this program, the code that controls the adaptive behavior is clearly separated from the code that manages the data and displays the results.

Transforming sinusoids

This program named **Adapt06** adaptively designs a convolution filter that transforms a cosine function into a sine function having the same amplitude and frequency.

There are no input parameters. You can just compile and run the program.

Graphic output

The following time series are plotted in color showing the convergence of the adaptive algorithm for the program named **Adapt06**:

- **black**: input to the filter - the cosine function
- **red**: output from the filter
- **blue**: adaptive target - a sine function
- **green**: error

Two coefficients

The filter has only two coefficients, each of which is initialized to a value of 0. The final values of the two filter coefficients are listed on the command-line screen after the specified number of adaptive iterations has been performed.

Purely deterministic

Note that because the time series used in this program do not contain random components, this is a purely deterministic problem that can easily be solved by writing and solving a pair of simultaneous equations.

Matrix equations

Thus, in this case, the adaptive solution is an adaptive approach to finding the roots of a pair of simultaneous equations. In fact, for many adaptive problems, an alternative non-adaptive solution can be written in terms of matrices involving autocorrelation and cross-correlation functions between the data time series and the target time series. In those cases, the adaptive solution provides a relatively simple adaptive approach to solving what may otherwise be a difficult matrix inversion problem.

Coefficient values

For the specific sinusoidal frequencies used by this program, when the program is allowed to run for 1000 iterations so as to fully converge, the values for the two coefficients converge to:

- 1.4142064344006575
- -0.9999927187203118

(You might recognize these two values as being very close to the square root of two and minus one.)

For other frequencies, however, the coefficients converge to different values.

Other classes required

This program requires the following classes in addition to **Adapt06**:

- AdaptEngine01
- AdaptiveResult
- PlotALot05

Adapt01a and Adapt02a

After I explain the program named **Adapt06**, I will present and briefly discuss the application of the adaptive engine to two other problems that are very similar to the two problems that I explained in the lessons entitled Adaptive Filtering in Java, Getting Started and An Adaptive Whitening Filter in Java.

These two programs are named **Adapt01a** and **Adapt02a**. The purpose of these two programs is to present solutions to the two problems using a structure where the adaptive code is clearly separated from the code required to manage the data and to display the results.

I will defer further discussion of these two programs until later in the lesson.

Discussion and Sample Code

The adaptive engine named AdaptEngine01

I will discuss the code for the adaptive engine in fragments. You can view the source code for the entire class in Listing 15 near the end of the lesson.

The beginning of the class and the constructor for the class are shown in Listing 1.

```
class AdaptEngine01{

  double[] filterArray;//filter coefficients
stored here
  double[] dataArray;//historical data is
stored here
  double feedbackGain;//used in LMS adaptive
algorithm

  //Constructor
  public AdaptEngine01(int filterLength,
                       double feedbackGain){
    //Construct the two arrays and save the
feedback gain.
    filterArray = new double[filterLength];
    dataArray = new double[filterLength];
    this.feedbackGain = feedbackGain;
  }//end constructor

Listing 1
```

Note that the constructor constructs two separate array objects. One array object, referred to by **filterArray**, will be used to contain the filter coefficients. The other array object, referred to by **dataArray** will be used as a tapped delay line to store historical values from the data time series that is to be filtered.

The constructor also receives and saves the value for **feedbackGain**.

The method named adapt

The method named **adapt** implements a classical LMS adaptive algorithm. This method creates and applies a convolution filter to an input data time series. The filter is adaptively adjusted to drive the difference between the filtered time series and a target time series to zero.

The filter output, the error, and a reference to the array object containing the filter coefficients are encapsulated in an object of type **AdaptiveResult** and returned to the calling method.

Two incoming parameters

The **adapt** method begins in Listing 2. The method receives two incoming parameters. One parameter is a single sample from a sampled time series that is to be filtered by the convolution filter. The other parameter is a single sample from the *target* time series mentioned above.

```
  AdaptiveResult adapt(double rawData,double
target){

    flowLine(dataArray,rawData);

Listing 2
```

Listing 2 invokes the **flowLine** method to insert the data sample into the tapped delay line.

> *(I have used and explained the **flowLine** method in several previous lessons, and won't repeat that explanation here. You can view the method in Listing 15 near the end of the lesson.)*

Apply the filter and compute the error

The code in Listing 3 invokes the **dotProduct** method to apply the current coefficient values to the data time series.

```
    double output =
dotProduct(filterArray,dataArray);

    double err = output - target;

Listing 3
```

Then Listing 3 subtracts the target value from the dot product output to compute the error value.

> *(I have used and explained the **dotProduct** method in several previous lessons and won't repeat that explanation here. You can view the method in Listing 15 near the end of the lesson.)*

Adjust the filter coefficient values

Listing 4 applies the classical LMS adaptive algorithm in a **for** loop to adjust the value of each filter coefficient using the following values:

- The current coefficient value.
- The current error value.
- The value of the data time series that was aligned with the filter coefficient when the dot product was computed *(the computation that resulted in the current error value)*.
- A feedback constant that controls the adaptation rate.

```
    for(int ctr = 0;ctr <
filterArray.length;ctr++){
      filterArray[ctr] -=
err*dataArray[ctr]*feedbackGain;
    }//end for loop.

Listing 4
```

All that remains ...

That's really all there is to it. All that remains is to return the appropriate values to the calling method.

I told you that the algorithm would be deceptively simple. It's somewhat amazing that such a simple algorithm can be used to solve so many different complex problems.

Return the results

Listing 5 constructs, populates, and returns a reference to an object of the class **AdaptiveResult**.

```
    return new
AdaptiveResult(filterArray,output,err);
  }//end adapt

Listing 5
```

As you can see, the object that is returned is populated with:

- A reference to the array containing the filter.
- The output from the filter.
- The value of the error.

These are the only values that change under the control of the adaptive algorithm.

You can view the class definition for the simple **AdaptiveResult** class in Listing 15.

End of the method

Listing 5 also signals the end of the **adapt** method, and will signal the end of my discussion of the LMS adaptive engine encapsulated in the class named **AdaptEngine01**. The remainder of the lesson will concentrate on how an object of this class can be used to solve three different adaptive problems.

The program named Adapt06

As mentioned earlier, the purpose of this program is to illustrate the use of the general-purpose LMS adaptive engine named **AdaptEngine01**. This is the simplest program that I could devise to illustrate the use of the adaptive engine. This program adaptively designs a convolution filter that transforms a cosine function into a sine function having the same amplitude and frequency.

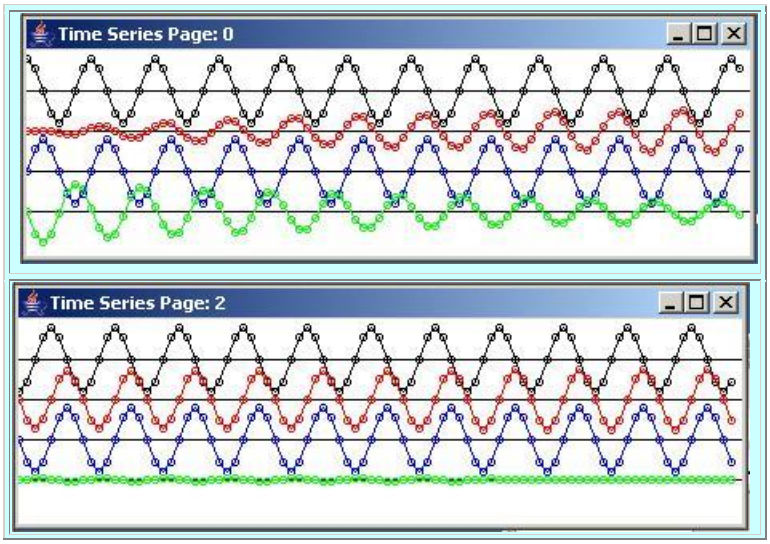
Before getting into the details of the code, I will show you some graphic output from the program. The following time series are plotted in color showing the convergence of the adaptive algorithm:

- **black**: input to the filter - the cosine function
- **red**: output from the filter
- **blue**: adaptive target - a sine function
- **green**: error

The first ninety iterations

The top panel in Figure 1 shows the four time series identified above for the first ninety adaptive iterations. The filter is applied to the black trace producing the red trace as the filter output. The filter coefficients are being adjusted in an attempt to cause the filter output *(red trace)* to look like the target *(blue trace)*. As the adaptive process converges toward a solution, the error *(green trace)* should approach zero.

As you can see, by the end of the first ninety iterations, the red trace is beginning to look like the blue *(target)* trace and the green *(error)* trace is getting smaller.



```
Figure 1
```

Iterations 181 through 270

The bottom panel in Figure 1 shows the results for iterations in the range from 181 to 270. As you can see, by this point, the adaptive process has nearly converged to a solution with the red trace nearly matching the blue trace and the error trace having been reduced to near zero.

Final coefficient values

As mentioned earlier, the filter consists of two coefficients, which are both initialized to a value of zero. When convergence is achieved, the coefficient values have converged to the two values shown in Figure 2.

```
Final filter coefficients:
1.4142064344006575
-0.9999927187203118
Figure 2
```

This solution is peculiar to the frequency that I selected for the sinusoids. Had I selected a different frequency, convergence would still have been achieved, but the final values of the two coefficients would have been different.

The Adapt06 class

Listing 6 shows the beginning of the **Adapt06** class and the **main** method.

```
class Adapt06{
  public static void main(String[] args){

    double feedbackGain = 0.0002;
    int numberIterations = 1000;
    int filterLength = 2;

    //Instantiate an object of the class and
execute the
    // adaptive algorithm using the specified
feedbackGain.
    new Adapt06().process(feedbackGain,
                          numberIterations,
                          filterLength);
  }//end main

Listing 6
```

The most important thing to note in Listing 6 is the invocation of the method named **process** on an object of the **Adapt06** class.

The process method

The **process** method begins in Listing 7.

```
  void process(double feedbackGain,
               int numberIterations,
               int filterLength){

    AdaptEngine01 adapter =
              new
AdaptEngine01(filterLength,feedbackGain);

Listing 7
```

The code in Listing 7 instantiates an object of the **AdaptEngine01** class, which will be used to provide the adaptive behavior for the program. As you can see, the filter length and the feedback gain values are passed to the constructor for the object.

Instantiate a plotting object and declare working variables

Listing 8 begins by instantiating a plotting object of the **PlotALot05** class to produce the graphic output shown in Figure 1.

> *(I have presented and explained objects from the **PlotALot** family of classes in several previous lessons and won't repeat that explanation here.)*

```
    //Instantiate a plotting object for four
data channels.
    // This object will be used to plot the
time series
    // data.
    PlotALot05 plotObj = new PlotALot05(
                            "Time
Series",460,155,25,5,4,4);

    //Declare and initialize working variables.
    double output = 0;
    double err = 0;
    double target = 0;
    double input = 0;
    double dataScale = 20;//Default data scale
    AdaptiveResult result = null;

Listing 8
```

Listing 8 also declares and initializes some working variables.

Execute the adaptive iterations

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

```
    for(int cnt = 0;cnt <
numberIterations;cnt++){
      //Generate the data to be filtered and
the adaptive
      // target on the fly.
      input = dataScale * cos(2*cnt*PI/8);
      target = dataScale * sin(2*cnt*PI/8);
```

Listing 9

Listing 9 also shows the code that creates the cosine and sine values used as the data to be filtered and the target respectively. The purpose of the adaptive process is to transform the time series produced by the **cos** method into a match for the time series produced by the **sin** method.

Execute the adaptive behavior

Now we come to the ***most important statement*** in the entire program. Listing 10 invokes the **adapt** method on the adaptive engine object to provide the adaptive behavior for the program. Note that the two values generated in Listing 9 are passed to the **adapt** method.

```
      result = adapter.adapt(input,target);
```

Listing 10

All of the adaptive behavior in the entire program is confined to this method call. All of the other code in the program simply manages objects and data and displays results.

The **adapt** method instantiates and populates an object of the **AdaptiveResult** class and returns a reference to that object. The code in Listing 10 stores that reference in the reference variable named **result** to make the adaptive results available for use later.

Feed the plotting object

The code in Listing 11 extracts the *error* and the filter *output* values from the **AdaptiveResult** object and feeds those values, along with the *input* and *target* values, to the plotting object. Eventually these values are plotted in the format shown in Figure 1.

> *(The values fed to the plotting object aren't actually plotted until the end of the program when the **plotData** method is invoked on the plotting object.)*

```
      //Get the results of the adaptive
behavior for
      // plotting.
      err = result.err;
      output = result.output;

      //Feed the time series data to the
plotting object.
```

```
plotObj.feedData(input,output,target,err);

    }//end for loop

Listing 11
```

Listing 11 also signals the end of the **for** loop that began in Listing 9.

Cleanup code

The code in Listing 12 causes all of the values that have been fed to the plotting object referred to by **plotObj** to actually be plotted on the screen.

```
    //Cause the data to be plotted.
    plotObj.plotData();

    //List the values of the filter
coefficients.
    System.out.println("\nFinal filter
coefficients:");
    double[] filter = result.filterArray;
    for(int cnt = 0;cnt < filter.length;cnt++){
      System.out.println(filter[cnt]);
    }//end for loop

  }//end process method

}//end class Adapt06

Listing 12
```

The code in Listing 12 also displays the final values for the filter coefficients on the command-line screen.

Listing 12 also signals the end of the **process** method and the end of the **Adapt06** class.

The program named Adapt01a

The program named **Adapt01a** is an update to the program that was presented and explained in the earlier lesson entitled Adaptive Filtering in Java, Getting Started. The purpose of this updated version of the program is to clearly separate the code that is responsible for the adaptive behavior from the code that is responsible for data management and the plotting of results.

This new version of the program is shown in its entirety in Listing 17 near the end of the lesson.

There are two major differences between this version and the earlier version of the program:

- This version was simplified somewhat by reducing the number of parameters that are provided by the user.
- All of the adaptive behavior was removed from the program and consolidated into the single statement shown in Listing 13.

```
      AdaptiveResult result =
adapter.adapt(input,target);

Listing 13
```

I explained the earlier version of the program in detail in the lesson entitled Adaptive Filtering in Java, Getting Started. If you understood that explanation, and you understand the adaptive engine presented in this lesson, you will surely understand the new version of the program named **Adapt01a.** Therefore, I won't repeat an explanation of the program in this lesson.

The program named Adapt02a

The program named **Adapt02a** is an update to the program that was presented and explained in the earlier lesson entitled An Adaptive Whitening Filter in Java. As was the case above, the purpose of this version of the program is to clearly separate the code that is responsible for the adaptive behavior from the code that is responsible for data management and the plotting of results.

This new version of the program is shown in its entirety in Listing 18 near the end of the lesson.

There are two major differences between this version and the earlier version of the program:

- This version was simplified by eliminating all user input.
- All of the adaptive behavior was removed from the program and consolidated into the single statement shown in Listing 14.

```
      AdaptiveResult result =
adapter.adapt(rawData[0],

rawData[1]);

Listing 14
```

As before, I explained the earlier version of this program in detail in the lesson entitled An Adaptive Whitening Filter in Java. If you understood that explanation, and you understand the adaptive engine presented in this lesson, you will also understand the new version of the program named **Adapt02a.** Therefore, I won't repeat an explanation of the program in this lesson.

Run the Programs

I encourage you to copy the code from the programs in the section entitled 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.

As I mentioned earlier, for the program named **Adapt06**, you will need the following class files in addition to the class file for the class named **Adapt06**:

- AdaptEngine01
- AdaptiveResult
- PlotALot05

For the program named **Adapt01a**, you will need the following class files in addition to the class file for the class named **Adapt01a**:

- AdaptEngine01
- AdaptiveResult
- ForwardRealToComplex01
- PlotALot01
- PlotALot03
- PlotALot05

For the program named **Adapt02a**, you will need the following class files in addition to the class file for the class named **Adapt02a**:

- AdaptEngine01
- AdaptiveResult
- ForwardRealToComplex01
- PlotALot01
- PlotALot03
- PlotALot07

The source code for the following classes can be found in the lessons indicated:

- ForwardRealToComplex01: Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
- PlotALot01: Plotting Large Quantities of Data using Java
- PlotALot03: Plotting Large Quantities of Data using Java
- PlotALot05: Adaptive Filtering in Java, Getting Started
- PlotALot07: An Adaptive Whitening Filter in Java

Summary

In this lesson, I showed you how to write a general-purpose LMS adaptive engine in Java, and how to demonstrate the use of the engine for three different adaptive programs of increasing complexity.

What's Next?

The next lesson in this series will teach you how to write an adaptive line tracking program in Java.

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

Complete Program Listings

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

```
/*File AdaptEngine01.java
Copyright 2005, R.G.Baldwin

General purpose LMS adaptive algorithm.

An object of this class is a general purpose adaptive
engine that implements the classical LMS adaptive
algorithm.

The adaptive algorithm is implemented by the instance
method belonging to the object named adapt.

Each time the adapt method is called, it receives one
sample from each of two different time series.  One time
series is considered to be the data that is to be filtered.
The other time series is considered to be a target.

The purpose of the adapt method is to adaptively create a
```

```
convolution filter which, when applied to the data time
series, will transform it into the target time series.

Each time the method is called, it performs a dot product
between the current version of the filter and the contents
of a delay line in which historical data samples have been
saved. The result of that dot product is compared with the
target sample to produce an error value. The error value is
produced by subtracting the value of the target sample from
the result of the dot product. The error value is then used
in a classical LMS adaptive algorithm to adjust the filter
coefficients.

The objective is to produce a set of filter coefficients
that will drive the error to zero over time.

This adaptive engine can be used as the solution to a
variety of different signal processing problems, depending
on the selection of time series that are provided as data
and target.

The constructor for the class receives two parameters:
filterLength
feedbackGain

The filter length is used to construct two arrays.  One
array is used later to contain the filter coefficients.

The other array is used later as a tapped delay line to
contain the historical data samples and to precess them by
one element each time the method is called.

The feedback gain is used in the LMS adaptive algorithm to
compute the new filter coefficients.

Tested using J2SE 5.0 and WinXP.
**********************************************************/

class AdaptEngine01{

  double[] filterArray;//filter coefficients stored here
  double[] dataArray;//historical data is stored here
  double feedbackGain;//used in LMS adaptive algorithm

  //Constructor
  public AdaptEngine01(int filterLength,
                       double feedbackGain){
    //Construct the two arrays and save the feedback gain.
    filterArray = new double[filterLength];
    dataArray = new double[filterLength];
    this.feedbackGain = feedbackGain;
  }//end constructor
  //-----------------------------------------------------//

  //This method implements a classical LMS adaptive
  // algorithm to create and to apply a convolution filter.
```

```
  // The filter output, the error, and a reference to the
  // array containing the filter coefficients are
  // encapsulated in an object of type AdaptiveResult and
  // returned to the calling method.
  AdaptiveResult adapt(double rawData,double target){

    //Insert the incoming data value into the data delay
    // line.
    flowLine(dataArray,rawData);

    //Apply the current filter coefficients to the data.
    double output = dotProduct(filterArray,dataArray);
    //Compute the error.
    double err = output - target;

    //Use the error to update the filter coefficients.
    for(int ctr = 0;ctr < filterArray.length;ctr++){
      filterArray[ctr] -= err*dataArray[ctr]*feedbackGain;
    }//end for loop.

    //Construct and return an object containing the filter
    // output, the error, and a reference to the array
    // object containing the current filter coefficients.
    return new AdaptiveResult(filterArray,output,err);
  }//end adapt
  //------------------------------------------------------//

  //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.
  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 receives two arrays and treats the first N
  // elements in each of the two arrays as a pair of
  // vectors.  It computes and returns the vector dot
  // product of the two vectors.  If the length of one
  // array is greater than the length of the other array,
  // it considers the number of dimensions of the vectors
  // to be equal to the length of the smaller array.
  double dotProduct(double[] v1,double[] v2){
    double result = 0;
    if((v1.length) <= (v2.length)){
      for(int cnt = 0;cnt < v1.length;cnt++){
        result += v1[cnt]*v2[cnt];
      }//emd for loop
      return result;
    }else{
      for(int cnt = 0;cnt < v2.length;cnt++){
```

```
        result += v1[cnt]*v2[cnt];
      }//emd for loop
      return result;
    }//end else
  }//end dotProduct
  //-----------------------------------------------------//
}//end class AdaptEngine01
//=====================================================//

//This class is used to encapsulate the adaptive results
// into an object for return to the calling method.
class AdaptiveResult{
  public double[] filterArray;
  public double output;
  public double err;

  //Constructor
  public AdaptiveResult(double[] filterArray,
                        double output,
                        double err){
    this.filterArray = filterArray;
    this.output = output;
    this.err = err;
  }//end constructor
}//end class AdaptiveResult
//=====================================================//
```

Listing 15

```
/*File Adapt06.java
Copyright 2005, R.G.Baldwin

The purpose of this program is to illustrate the use of the
general-purpose LMS adaptive engine named AdaptEngine01.

This is the simplest program that I was able to devise to
illustrate the use of the adaptive engine. This program
adaptively designs a convolution filter transforms a cosine
function into a sine function having the same amplitude and
frequency.

There are no input parameters.  Just compile and run.

The following time series are plotted in color showing the
convergence of the adaptive algorithm:

black: input to the filter - the cosine function
red: output from the filter
blue: adaptive target - a sine function
green: error

The filter has two coefficients, each of which is
initialized to a value of 0.
```

```
The final values of the two filter coefficients are
listed on the command-line screen. For the sinusoidal
frequencies used by the program, when the program is
allowed to run for 1000 iterations, the values for the two
coefficients converge to:

1.4142064344006575
-0.9999927187203118

You might recognize that is the square root of two and
minus 1.  For other frequencies, the coefficients
converge to different values.

This program requires the following classes in addition
to Adapt06:

Adapt06.class
AdaptEngine01.class
AdaptiveResult.class
PlotALot05.class

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

class Adapt06{
  public static void main(String[] args){

    double feedbackGain = 0.0002;
    int numberIterations = 1000;
    int filterLength = 2;

    //Instantiate an object of the class and execute the
    // adaptive algorithm using the specified feedbackGain.
    new Adapt06().process(feedbackGain,
                          numberIterations,
                          filterLength);
  }//end main
  //-------------------------------------------------------//

  void process(double feedbackGain,
               int numberIterations,
               int filterLength){

    //Instantiate object of the adaptive engine to provide
    //adaptive behavior for the program.
    AdaptEngine01 adapter =
              new AdaptEngine01(filterLength,feedbackGain);

    //Instantiate a plotting object for four data channels.
    // This object will be used to plot the time series
    // data.
    PlotALot05 plotObj = new PlotALot05(
                           "Time Series",460,155,25,5,4,4);
```

```
    //Declare and initialize working variables.
    double output = 0;
    double err = 0;
    double target = 0;
    double input = 0;
    double dataScale = 20;//Default data scale

    //Do the iterative adaptive process
    AdaptiveResult result = null;
    for(int cnt = 0;cnt < numberIterations;cnt++){
      //Generate the data to be filtered and the adaptive
      // target on the fly.
      input = dataScale * cos(2*cnt*PI/8);
      target = dataScale * sin(2*cnt*PI/8);

      //Execute the adaptive behavior.
      result = adapter.adapt(input,target);

      //Get the results of the adaptive behavior for
      // plotting.
      err = result.err;
      output = result.output;

      //Feed the time series data to the plotting object.
      plotObj.feedData(input,output,target,err);

    }//end for loop

    //Cause the data to be plotted.
    plotObj.plotData();

    //List the values of the filter coefficients.
    System.out.println("\nFinal filter coefficients:");
    double[] filter = result.filterArray;
    for(int cnt = 0;cnt < filter.length;cnt++){
      System.out.println(filter[cnt]);
    }//end for loop

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

}//end class Adapt06
```

Listing 16

```
/*File Adapt01a.java.java
Copyright 2005, R.G.Baldwin

This is an upgrade to the program named Adapt01.  The
purpose of this upgrade is to use the adaptive engine
named AdaptEngine01 to provide the adaptive behavior of the
program and to clearly deliniate the code that controls the
```

```
adaptive behavior from the code that simply handles the
data and displays the results.

As was the case with the earlier program named Adapt01,
this program illustrates one aspect of time-adaptive signal
processing.  However, this version of the program has been
greatly simplified by eliminating all but one of the
command-line parameters.

Two versions of the same sampled time series, one delayed
relative to the other, are presented to the an adaptive
algorithm. The objective is to adaptively create a
convolution filter that will delay the first time
series to cause it to match up in time with the second
time series.  This results in a convolution filter having
a flat amplitude response and a linear phase response.

The user provides the following information as a command
line parameter:

timeDelay - This value causes the second time series to
be delayed relative to the first time series by from 0 to
8 samples.  Negative values and values greater than 8 cause
the program to terminate.

The following time series are plotted in color showing the
convergence of the adaptive algorithm:
black: input to the filter
red: output from the filter
blue: adaptive target
green: error

In addition, the frequency response of the filter at the
end of every tenth iteration is computed and displayed when
the adaptive process terminates.  Both the amplitude and
the phase response of the filter are computed and plotted.
Also, the convolution filter is plotted as a time series on
the same  iterations that the frequency response is
computed.  Thus, the shape of the convolution filter can be
compared with the frequency response of the filter.

The filter is initialized with values of 0 for all
coefficients.  The ideal solution is a single coefficient
value of 1 at a location in the filter that matches the
time delay between the two time series along with a linear
phase response that matches the location of the single
filter coefficient.

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

class Adapt01a{
  public static void main(String[] args){
    //Default value for time delay input parameter.
```

```
    int timeDelay = 7;

    //Fixed values.
    double feedbackGain = 0.001;
    int numberIterations = 100;

    //Deal with command-line parameter.
    if(args.length != 1){
      System.out.println("Usage: java Adapt01a " +
                                                 "timeDelay");
    }else{//Command line param was provided.
      //Convert String to int
      timeDelay = Integer.parseInt(args[0]);
      System.out.println("timeDelay: " + timeDelay);
    }//end else

    if((timeDelay > 8) || (timeDelay < 0)){
      System.out.println(
                       "Time delay must be >= 0 and <= 8");
      System.out.println("Terminating");
      System.exit(0);
    }//end if

    //Instantiate an object of the class and execute the
    // adaptive algorithm using the  specified feedbackGain
    // and other parameters.
    new Adapt01a().process(timeDelay,
                           feedbackGain,
                           numberIterations);
  }//end main
  //-------------------------------------------------------//

  void process(int timeDelay,
               double feedbackGain,
               int numberIterations){

    int filterLength = 9;

    //Instantiate object of the adaptive engine to provide
    //adaptive behavior for the program.
    AdaptEngine01 adapter =
              new AdaptEngine01(filterLength,feedbackGain);

    //The filter array is created and maintained within the
    // adaptive engine object. A reference to the array is
    // returned by the adapt method.
    double[] filter;

    //Create array object that will be used as delay line.
    double[] rawData = new double[9];

    //Instantiate a plotting object for four data channels.
    // This object will be used to plot the time series
    // data.
    PlotALot05 plotObj = new PlotALot05(
                           "Time Series",398,250,25,5,4,4);
```

```
    //Instantiate a plotting object for two channels of
    // filter frequency response data.  One channel is used
    // to plot the amplitude response in db and the other
    // channel is used to plot the phase on a scale that
    // extends from -180 degrees to +180 degrees.
    PlotALot03 freqPlotObj =
                   new PlotALot03("Freq",264,487,20,2,0,0);

    //Instantiate a plotting object to display the filter
    // as a short time series at intervals during the
    // adaptive  process.
    //Note that the minimum allowable width for a Frame is
    // 112 pixels under WinXP. Therefore, the following
    // display doesn't synchronize properly for filter
    // lengths less than 25 coefficients.  However, the
    // code that feeds the filter data to the plotting
    // object later in the program extends the length of
    // the filter to cause it to synchronize and to plot
    // one set of filter coefficients on each axis.
    PlotALot01 filterPlotObj = new PlotALot01(
            "Filter",(filterLength * 4) + 8,487,40,4,0,0);

    //Declare and initialize working variables.
    double output = 0;
    double err = 0;
    double target = 0;
    double input = 0;
    double dataScale = 20;//Default data scale

    //Do the iterative adaptive process
    for(int cnt = 0;cnt < numberIterations;cnt++){

      //Add new input data to the delay line containing the
      // raw input data.  Raw data values are uniformly
      // distributed from -10.0 to +10.0.
      flowLine(rawData,dataScale*(random() - 0.5));

      //Get the raw input data from the end of the delay
      // line.
      input = rawData[8];

      //Get the target data from one of nine available taps
      // on the delay line.
      target = rawData[8 - timeDelay];

      //Execute the adaptive behavior.
      AdaptiveResult result = adapter.adapt(input,target);

      //Get the results of the adaptive behavior for
      // plotting and frequency response analysis.
      filter = result.filterArray;
      err = result.err;
      output = result.output;

      //Feed the time series data to the plotting object.
```

```
      plotObj.feedData(input,output,target,err);

      //Compute and plot the frequency response and plot
      // the filter as a time series every 10 iterations.
      if(cnt%10 == 0){
        displayFreqResponse(filter,
                            freqPlotObj,
                            128,
                            filter.length - 1);

        //Plot the filter coefficient values. Scale the
        // coefficient values by 30 to make them compatible
        // with the  plotting software.
        for(int ctr = 0;ctr < filter.length;ctr++){
          filterPlotObj.feedData(30*filter[ctr]);
        }//end for loop

        //Extend the filter with a value of 2.5 for
        // plotting purposes only to cause it to
        // synchronize with one filter being plotted on
        // each axis.
        if(filter.length <= 26){
          for(int count = 0;count < (26 - filter.length);
                                                  count++){
            filterPlotObj.feedData(2.5);
          }//end for loop
        }//end if
      }//end if on cnt%10

    }//end for loop

    //Cause all the data to be plotted in the
    // screen locations specified.
    plotObj.plotData();
    filterPlotObj.plotData(0,201);
    freqPlotObj.plotData(112,201);

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

  //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.
  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 receives a reference to a double
  // array containing a convolution filter
  // along with a reference to a plotting object
  // capable of plotting two channels of data.
  // It also receives a value specifying the
  // number of frequencies at which a DFT is
  // to be performed on the filter, along with
  // the sample number that represents the zero
  // time location in the filter.  The method
  // uses this information to perform a DFT on
  // the filter from zero to the folding
  // frequency.  It feeds the amplitude spectrum
  // and the phase spectrum to the plotting
  // object for plotting.
  void displayFreqResponse(double[] filter,
                           PlotALot03 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);


    //Plot the magnitude data.  Convert to
    // normalized decibels before plotting.

    //Eliminate or change all values that are
    // incompatible with log10 method.
    for(int cnt = 0;cnt < magnitude.length;
                                       cnt++){
      if((magnitude[cnt] == Double.NaN) ||
                          (magnitude[cnt] <= 0)){
        magnitude[cnt] = 0.0000001;
      }else if(magnitude[cnt] ==
                       Double.POSITIVE_INFINITY){
```

```
        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.

    //Find the absolute peak 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

    //Normalize to 50 times the peak value and
    // shift up the screen by 50 units to make
    // the values compatible with the plotting
    // program.  Recall that adding a constant to
    // log values is equivalent to scaling the
    // original data.
    for(int cnt = 0;cnt < magnitude.length;
                                          cnt++){
      magnitude[cnt] =
                    50*magnitude[cnt]/peak + 50;
    }//end for loop

    //Now feed the normalized decibel data to the
    // plotting object.  The angle data rangers
    // from -180 to +180.  Scale it down by a
    // factor of 20 to make it compatible with
    // the plotting format being used.
    for(int cnt = 0;cnt < magnitude.length;
                                          cnt++){
      plot.feedData(
                  magnitude[cnt],angle[cnt]/20);
    }//end for loop

  }//end displayFreqResponse
  //----------------------------------------//
}//end class Adapt01a
```

Listing 17

```
/*File Adapt02a.java.java
Copyright 2005, R.G.Baldwin

This program is an update of the program named Adapt01. The
purpose of the update is to delegate the adaptive behavior
of the program to an object of the class named
AdaptEngine02, clearly deliniating between that portion of
the program that provides adaptive behavior and that
portion that simply does data handling and displays
results.

This program has been simplified relative to the program
named Adapt02 by eliminating all command-line parameters.

This program illustrates one aspect of time-adaptive signal
processing.

This program implements a time-adaptive whitening filter
using a predictive approach. The program input is a time
series consisting of a wide band signal plus two sinusoidal
noise functions. The program adaptively creates a filter
that attempts to eliminate the sinusoidal noise while
preserving the wide band signal.

The following time series are displayed when the program
runs:

-err: This is the negative of the output from the whitening
filter.  Ideally this time series contains the wide band
signal with the sinusoidal noise having been removed.

signal: The raw wideband signal consisting of samples taken
from a random noise generator.

sineNoise: The raw noise consisting of the sum of two
sinusoidal functions.

input: The sum of the signal plus the sinusoidal noise.

output: The output produced by applying the prediction
filter to the input signal plus noise.

target: The target signal that is used to control the
adaptive process. This is the next sample beyond the
samples that are processed by the prediction filter. In
other words, the prediction filter attempts to predict this
value. Thus, the adaptive process attempts to cause  the
output from the prediction filter to match the next sample
in the incoming signal plus noise.  This is an attempt to
predict a future value based solely on the current and past
values.

Although not required by the adaptive process, the
frequency response of the whitening filter is computed and
displayed once every 100 iterations.  Ideally the amplitude
response is flat with very narrow notches at the
```

```
frequencies of the interfering sinusoidal noise components.
Both the amplitude and phase response are displayed. This
makes it possible to see the notches develop in the
frequency response of the whitening filter as it converges.
It also makes it possible to see how the phase behaves at
the notches.

The individual whitening filters on which the frequency
response is computed are also displayed as time series.

USAGE: Just compile and run the program.  There are no
user inputs.  The program uses fixed values for the
following:

feedbackGain: The gain factor that is used in the feedback
loop to adjust the coefficient values in the
prediction/whitening filter. (A whitening filter is a
prediction filter with a -1 appended to its end.)

numberIterations: The is the number of iterations that the
program executes before stopping and displaying all of the
graphic results.

pFilterLength: This is the number of coefficients
in the prediction filter.

signalScale: A scale factor that is applied to the wide
band signel provided by the random noise generator. The
random noise generator produces uniformaly distributed
values ranging from -0.5 to +0.5.

noiseScale: A scale factor that is applied to each of the
sinusoidal noise functions before they are added to the
signal. The raw sinusoids vary from -1.0 to +1.0.

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

class Adapt02a{
  public static void main(String[] args){
    //Default parameter values
    double feedbackGain = 0.00001;
    int numberIterations = 600;
    int pFilterLength = 26;
    double signalScale = 20;
    double noiseScale = 20;

    //Instantiate a new object of the Adapt02a class
    // and invoke the method named process on that object.
    new Adapt02a().process(feedbackGain,
                           numberIterations,
                           pFilterLength,
                           signalScale,
                           noiseScale);
```

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

//This is the primary adaptive processing and plotting
// method for the program.
void process(double feedbackGain,
             int numberIterations,
             int pFilterLength,
             double signalScale,
             double noiseScale){

  double[] pFilter;

  //Instantiate an object to handle the adaptive behavior
  // of the program.
  AdaptEngine01 adapter = new AdaptEngine01(
                            pFilterLength,feedbackGain);

  //Create the initial whiteningFilter and initialize it.
  double[] whiteningFilter =
                new double[pFilterLength + 1];

  //Set the final value in the whitening filter to -1.
  whiteningFilter[whiteningFilter.length - 1] = -1;

  //Create an array to serve as a two-sample delay line
  // for the raw data.  The data value at zero index will
  // be the data is to be filtered.  The data value at
  // index value 1 will be the predictive target.
  double[] rawData = new double[2];

  //Instantiate a plotting object for six channels of
  // time-serie data.
  PlotALot07 timePlotObj =
               new PlotALot07("Time",468,200,25,10,4,4);

  //Instantiate a plotting object for two channels of
  // filter frequency response data.  One channel is for
  // the amplitude and the other channel is the phase.
  PlotALot03 freqPlotObj =
                new PlotALot03("Freq",264,487,35,2,0,0);

  //Instantiate a plotting object to display the
  // whitening filter at specific time intervals during
  // the adaptive process.
  PlotALot01 filterPlotObj = new PlotALot01("Filter",
          (whiteningFilter.length * 4) + 8,487,70,4,0,0);

  //Declare and initialize working variables.
  double output = 0;
  double err = 0;
  double target = 0;
  double input = 0;
  double signal = 0;
  double sineNoise = 0;
```

```
    //Perform the specified number of iterations.
    for(int cnt = 0;cnt < numberIterations;cnt++){
      //Get the next sample of wideband signal.
      signal = signalScale*(Math.random() - 0.5);

      //Get the next sample of sinusoidal noise consisting
      // of two sinusoids at different frequencies.
      sineNoise = noiseScale*(Math.sin(2*cnt*PI/8) +
                              Math.sin(2*cnt*PI/3));

      //Insert the signal plus noise into the raw data
      // delay line.
      flowLine(rawData,signal + sineNoise);

      //Execute the adaptive behavior.
      AdaptiveResult result = adapter.adapt(rawData[0],
                                            rawData[1]);

      //Get and save adaptive results for plotting and
      // spectral analysis
      output = result.output;
      err = result.err;
      pFilter = result.filterArray;

      //Feed the time series data to the plotting object.
      timePlotObj.feedData(-err,signal,sineNoise,
                           rawData[0],output,rawData[1]);

      //Compute and plot the frequency response and plot
      // the whitening filter every 100 iterations.
      if(cnt%100 == 0){
        //Create a whitening filter from the data in the
        // prediction filter by copying the prediction
        // filter into the bottom elements of the whitening
        // filter. Recall that the last element in the
        // whitening filter already has a value of -1.
        System.arraycopy(pFilter,
                         0,
                         whiteningFilter,
                         0,
                         pFilter.length);

        displayFreqResponse(whiteningFilter,freqPlotObj,
                            128,whiteningFilter.length - 1);

        //Display the whitening filter coefficient values.
        for(int ctr = 0;ctr < whiteningFilter.length;
                                                ctr++){
          filterPlotObj.feedData(40*whiteningFilter[ctr]);
        }//end for loop
      }//End display of frequency response and whitening
       // filter
    }//End for loop,

    //Cause all the data to be plotted.
    timePlotObj.plotData();
```

```
    freqPlotObj.plotData(0,201);
    filterPlotObj.plotData(265,201);

  }//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.
  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
  //-------------------------------------------------//

  void displayFreqResponse(
     double[] filter,PlotALot03 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.

    //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

    //Normalize to 50 times the peak value and shift up the
    // page by 50 units to make the values compatible with
    // the plotting program.  Recall that adding a
    // constant to log values is equivalent to scaling the
    // original data.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = 50*magnitude[cnt]/peak + 50;
    }//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],angle[cnt]/20);
    }//end for loop

  }//end displayFreqResponse
  //-------------------------------------------------//
}//end class Adapt02a
```

Listing 18

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he*

believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which 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-