A Recursive Filtering Workbench in Java

Use this interactive workbench to gain a better understanding of the behavior of digital recursive filters. Then learn how to write the code to do recursive filtering as well as to evaluate recursive filters using a workbench-style program.

Published: August 8, 2006
By Richard G. Baldwin

Java Programming Notes # 1510

Preface

This is the first installment of a multi-part lesson on digital recursive filtering.

A Recursive Filtering Workbench

The complete collection of installments presents and explains the code for an interactive *Recursive Filtering Workbench (See Figure 1)* that can be used to design, experiment with, and evaluate the behavior of digital recursive filters.

> *(The digital Recursive Filtering Workbench will be referred to hereafter simply as the workbench.)*

By the end of the last installment, you should have learned how to write a Java program to create such a workbench. Hopefully, you will also have gained some understanding of what recursive filters are, how they behave, and how they fit into the larger overall technology of Digital Signal Processing *(DSP)*.

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.

I also recommend that you pay particular attention to the lessons listed in the References section of this document.

Preview

Figure 1 shows a full screen shot of the workbench in operation.

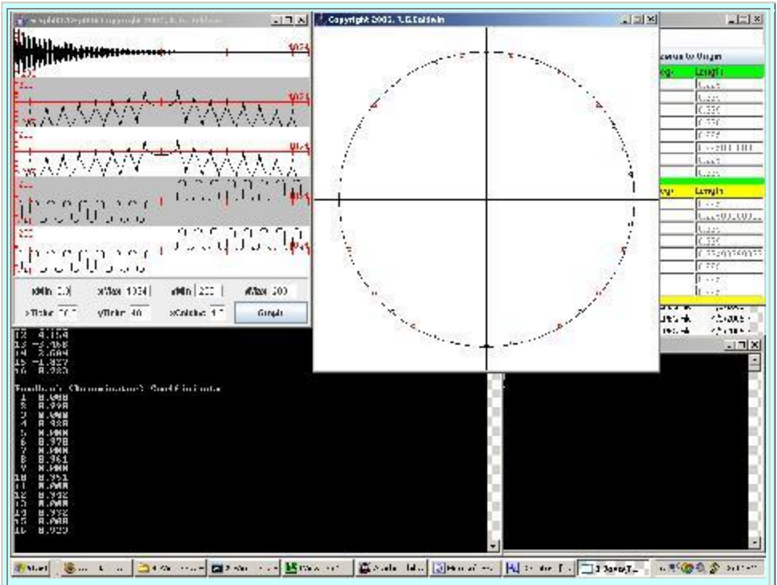


Figure 1

In order to accommodate this narrow publication format, the screen shot in Figure 1 was reduced to the point that the individual images are no longer legible. The purpose of providing the images in Figure 1 was to provide an overview of the workbench. That overview will be used in conjunctions with references in the paragraphs that follow. *(Full-size versions of the individual images will be presented later.)*

Complex poles and zeros

The workbench makes it possible for the user to design a recursive filter by specifying the locations of sixteen poles and sixteen zeros in the complex z-plane and then to evaluate the behavior of the recursive filter for that set of poles and zeros. The user can relocate the poles and zeros and re-evaluate the behavior of the corresponding recursive filter as many times as may be beneficial without a requirement to restart the program.

An image of the z-plane

The GUI in the top center portion of the screen in Figure 1 is an image of the complex z-plane showing the unit circle along with the locations of all the poles and zeros.

> *(Because of the reduction in the size of the image, the poles and zeros are only barely visible in Figure 1. A full-size version is shown in Figure 2.)*

The user can interactively change the location of any pair of complex poles or zeros by selecting a specific pair of poles or zeros and clicking the new location with the mouse in the z-plane GUI. This provides a quick and easy way to position the poles and zeros in the z-plane.

Numeric text input

The GUI that is partially showing in the upper right of Figure 1 contains four text fields and a radio button for each pair of complex poles and each pair of complex zeros. *(A full-size version of this GUI is shown in Figure 5.)*

This GUI has several purposes. For example, the user can specify the locations of pairs of poles or zeros very accurately by entering the real and imaginary coordinate values corresponding to the locations into the text fields in this GUI. The user can also view the length and the angle *(relative to the real axis)* of an imaginary vector that connects each pole and each zero to the origin.

This GUI also provides some other features that are used to control the behavior of the workbench program. These features will be explained later.

The two GUIs are connected

When this GUI is used to relocate a pair of poles or zeros, the graphical image of the z-plane in Figure 2 is automatically updated to reflect the new location. Similarly, when the mouse is used with the graphical image of the z-plane to relocate a pair of poles or zeros, the numeric information in the GUI in Figure 5 is automatically updated to reflect the new location.

Two ways to specify the location of poles and zeros

Thus, these two GUIs provide two different ways that the user can specify the locations of poles and zeros. Clicking the location with the mouse is very quick and easy but not particularly accurate. Entering the numeric coordinate values into the text fields takes a little more effort, but is very accurate.

The two approaches can be used in combination to create a rough design with the mouse and then to polish the design by entering accurate pole and zero locations into the text fields.

The behavior of the recursive filter

Once the locations of the poles and zeros that define a recursive filter have been established using either or both of the two GUIs discussed above, the leftmost GUI in Figure 1 can be used to display the following information about the recursive filter. *(See Figure 6 for a full-size version of this GUI.)*

- The impulse response in the time domain.
- The amplitude response in the frequency domain.
- The phase response in the frequency domain.

Two computational approaches

Two versions of the amplitude response and two versions of the phase response are plotted in the GUI shown in Figure 6. One version is based on the Fourier Transform of the impulse response. The other version is based on a vector analysis of the locations of the poles and zeros in the z-plane.

Adjusting the plotting parameters

The GUI shown in Figure 6 provides for the input of seven different plotting parameters *(vertical scale, horizontal scale, tic mark locations, etc.)*. The user can modify the plotting parameters and replot the graphs as many times as may be needed to get a good visual representation of the behavior of the recursive filter.

General Background Information

I will not attempt to teach you the theory behind recursive filtering. Rather, I will assume that you already understand that theory, or that you are studying a good theoretical resource on recursive filtering concurrently with the study of this lesson.

> *(A very good resource on the theory of digital filtering in general, and recursive digital filtering in particular, is the Introduction to Digital Filters with Audio Applications by Julius O. Smith III .)*

As I explain the material in this lesson, I will make technical statements of fact without providing a theoretical justification for those statements. The theoretical justification for those statements can be found in Smith's material as well as other resources on the web.

What is a recursive filter?

Here is some of what Wikipedia has to say about recursive filters:

Recursive filters *are signal processing filters which re-use one or more output(s) of the filter as inputs. This feedback results in an unending impulse response characterized by either exponentially growing, decaying, or sinusoidal signal output components.*

A simple recursive filter

Although it is possible to design very complex recursive filter structures, this lesson will deal with simple recursive filters. As I define it, a simple recursive filter contains a single feed-forward section, which is essentially equivalent to the convolution filters described in earlier lessons. It also contains a single feedback loop, which performs a convolution operation on the previous output samples before combining that result with the new input data.

The recursive filter used in this workbench implements a difference equation very similar to that provided by Smith.

Why use recursive filters?

Most digital filters produce one output sample for every input sample. The computational cost of a digital filtering operation can usually be measured in terms of the number of multiply-add operations *(MADs)* required for each output sample.

It is not possible to design a recursive filter to achieve every possible filtering behavior that may be needed. Given all the possible needs for digital filters, it is more likely that you can design a non-recursive convolution filter to satisfy a given need than that you can design a recursive filter to satisfy that same need. However, if a recursive filter can be designed to satisfy a given need, the implementation of the recursive filter is likely to have a lower computational cost than would be the case with a non-recursive convolution filter designed to satisfy the same need.

Greater arithmetic accuracy is required

However, in the computing world as well as the world outside the computer, there is no such thing as a free lunch. While a recursive filter designed to satisfy a given filtering need may have a lower computational cost than a non-recursive convolution filter design to satisfy the same need, the recursive filter will probably require greater arithmetic accuracy than the convolution filter. This is because a recursive filter feeds part of its output back into the input. As a result, small arithmetic errors can build up to produce large errors.

Phase distortion can be a problem

All filters produce some amount of phase distortion. This means that some frequency components in the output are delayed more or less than other frequency components relative to the relationship between those components in the input. This results in a modification of the output waveform relative to the input waveform. This may, or may not be a problem, depending on the application.

A time delay

A non-recursive convolution filter can be designed in such a way that the phase distortion manifests itself simply as a time delay between the input and the output with the waveform being otherwise unmodified by the phase response. This is the result of designing the filter in such a way that the phase response is a linear function of frequency. Sometimes this is a very acceptable form of phase distortion, particularly if all the signals being processed are subjected to the same amount of time delay.

To my knowledge, it is not possible to design a general recursive filter where the phase response is always a linear function of frequency. *(It is possible, or almost so for some unique cases of recursive filters.)* This may be another disadvantage of recursive filters relative to non-recursive convolution filters if non-linear phase distortion is a problem.

Advantages and disadvantages of recursive filters

Thus, the primary advantage of a recursive filter relative to a non-recursive convolution filter is that it is likely to have a lower computational cost. The potential disadvantages are the requirement for greater arithmetic accuracy and a nonlinear phase response.

Design of a non-recursive convolution filter

If you are not constrained as to the number of filter coefficients required, non-recursive convolution filters are relatively easy to design. Although there are several design approaches available, one of the most straightforward approaches involves developing the time-domain impulse response of the filter by performing an inverse Fourier Transform on the desired complex frequency response.

However, if you are constrained as to the number of filter coefficients required, this often involves making a compromise between convolution filter length and expectations in the frequency domain. The first attempt may not produce an acceptable frequency response and an acceptable convolution filter length. If not, the process can be repeated with modifications to the frequency-domain expectations until an impulse response having an acceptable length that matches an acceptable frequency response is achieved. This process is discussed on one of the pages of London's Global University.

Perform an inverse Fourier Transform on complex frequency response

With this approach, you specify the desired amplitude and phase response of the filter as a set of complex samples in the frequency domain. Then you perform an inverse Fourier Transform on that set of complex samples.

> *(In this case, it is probably better to use an ordinary inverse Fourier Transform algorithm instead of an inverse FFT algorithm due to the improved flexibility of the non-FFT algorithm relative to the frequency sampling interval, the time-domain sampling interval, and the length of the time-domain impulse response.)*

If the complex samples are correctly specified, the inverse Fourier Transform will produce a purely-real impulse response in the time domain.

Truncate or window the impulse response

The next step is to truncate *(or possibly window)* the resulting impulse response to the desired number of convolution filter coefficients. Then perform a forward Fourier Transform on the truncated *(or windowed)* impulse response to see how well the frequency response of the truncated *(or windowed)* convolution filter matches the desired frequency response.

If the match is good, your design task is finished. Use the truncated *(or windowed)* impulse response as your convolution filter.

If the match is not good ...

If the match is not good, you must modify your expectations and desires in order to make them more realistic. At this point, you can either

- Extend the length of the truncated *(or windowed)* impulse response until you get a good match, or
- Modify the desired frequency response and start the design process over in an attempt to get a truncated *(or windowed)* impulse response of a reasonable length that matches an acceptable frequency response, or
- A combination of the two.

Typically, for the second case, the required modifications will involve reducing the slope of changes in the amplitude response. Abrupt changes in the required amplitude response typically require long impulse responses to achieve.

Designing recursive filters

The design of recursive filters is considerably more complex than the design of non-recursive convolution filters, even for the common cases of low-pass, band-pass, or high-pass filters. *(The design of recursive filters for other more exotic frequency-response characteristics is even more complex.)*

There are mathematical techniques available for designing recursive filters *(many of which derive largely from the design of analog electronic filters)*. However, this workbench doesn't use mathematical design techniques. Rather, it provides a visual interactive design approach instead of a mathematical approach.

Not intended for recursive filter design

Although this workbench provides the capability for designing recursive filters *(even those having exotic frequency response characteristics)*, the workbench is intended to serve as a learning tool rather than a design tool. My hope is that through experimentation with the workbench and examination of the code, you will gain a better understanding of the behavior of recursive filters, and an understanding of how to program them. However, if you also want to use the workbench as a design tool, be my guest.

> *(If you do decide to use the workbench for serious recursive filter design, you might want to expand the size of the z-plane GUI shown in Figure 2. The radius of the unit circle is currently 200 pixels. Therefore, you cannot possibly use the mouse to specify the location of a pole or a zero to an accuracy greater than about one part in 200. You might want to consider changing the radius to 500 pixels (or more) and displaying only the top half of the z-plane. This will improve the accuracy with which you can use the mouse to specify the location of a pole or a zero to about one part in 500.)*

A ratio of polynomials in the z-plane

The transfer function **H(z)** for a digital recursive filter can be thought of as a ratio of two polynomials:

```
H(z) = B(z)/A(z)
```

Smith develops the theory behind the concept and presents such a transfer function in his equation 7.5.

The essence of recursive filter design

The transfer function consists of a numerator polynomial **B(z)** and a denominator polynomial **A(z)**. Essentially the design of a recursive filter amounts to determining the values for the coefficients of the two polynomials *(shown as subscripted representations of **b** and **a** in Smith's equation 7.5)* that produce the desired response in the frequency domain.

Design on the basis of the roots of the polynomials

One of the common approaches to the design of a recursive filter *(determination of the polynomial coefficient values)* is to define the two polynomials such that the roots of the polynomials occur at specific locations in the complex z-plane shown in Figure 2.

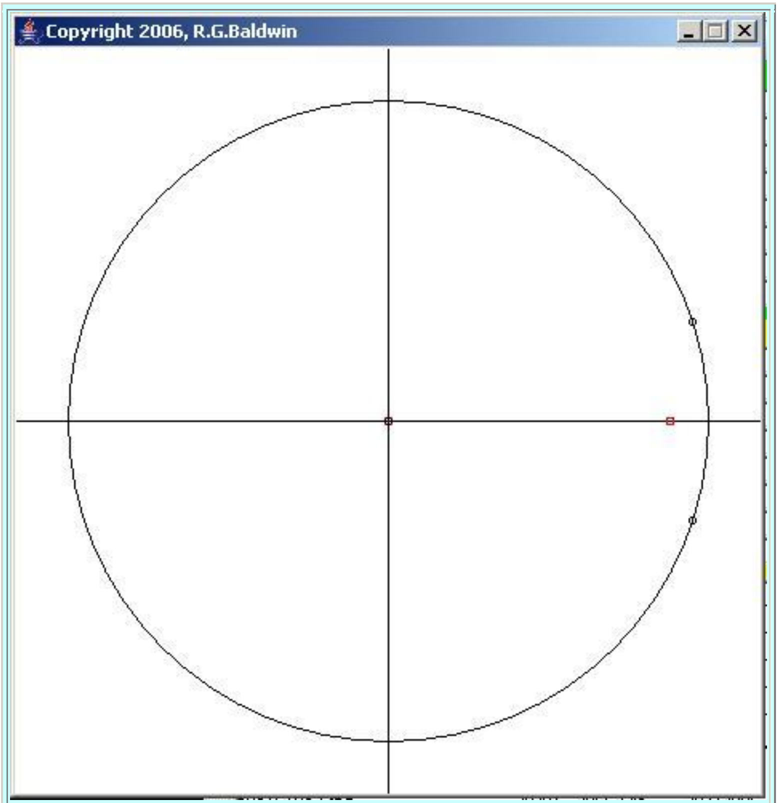


Figure 2

The z-plane

Figure 2 shows the complex z-plane with the horizontal axis being the *real* axis and the vertical axis being the *imaginary* axis.

> *(Figure 2 is a full-size representation of the top-center image shown in reduced size on the screen shot in Figure 1.)*

The unit circle

The large circle shown in Figure 2 is a circle with a radius of 1.0, commonly referred to as the *unit circle*. I will have more to say about the importance of the unit circle later.

Poles and *zeros*

The roots of the denominator polynomial are commonly referred to as *poles*, while the roots of the numerator polynomial are commonly referred to as *zeros*.

The zeros in Figure 2 are represented by small black circles. The poles are represented by small red squares.

A pair of complex conjugate zeros

Figure 2 shows a pair of complex conjugate zeros on *(or very close to)* the right side of the unit circle. The real coordinate values of the zeros are 0.95 and the imaginary coordinate values are plus and minus 0.31.

An imaginary vector drawn from the origin to each of the zeros forms an angle of plus or minus 18.07 degrees relative to the real *(horizontal)* axis. The length of each of those vectors is 0.99929975482. *(A length of 1.0 would be required to place the zero exactly on the unit circle.)*

A pair of poles

Figure 2 also shows a pair of poles stacked on top of one another at a real coordinate value of 0.88 and an imaginary coordinate value of 0.0.

A vector drawn from the origin to the poles forms an angle of 0.0 degrees relative to the real axis, and has a length of 0.88.

Poles and zeros at the origin

In addition to the two poles and the two zeros described above, fourteen additional poles and fourteen additional zeros are stacked on top of one another at the origin where they are of no consequence to the behavior of the recursive filter.

The polynomial coefficients

If we ignore the poles and zeros at the origin, the numerator and denominator coefficients for Smith's equation 7.5 that are required to produce the pole and zero locations shown in Figure 2 are shown in Figure 3.

```
Feed-Forward (Numerator) Coefficients
0  1.000
1 -1.900
2  0.999

Feedback (Denominator) Coefficients
1 -1.760
2  0.774
```

Figure 3

The *Feed-Forward (Numerator) Coefficients* in Figure 3 correspond to the coefficients identified by a **b** in Smith's equation 7.5, with the value in the first column in Figure 3 corresponding to the coefficient subscript in the equation.

Similarly, the *Feedback (Denominator) Coefficients* in Figure 3 correspond to the coefficients identified by **a** in Smith's equation. *(The denominator coefficient with a subscript of 0 has an implicit value of 1.0.)*

The full set of numerator and denominator coefficients

The workbench is designed to allow you to specify the locations of and to analyze the behavior of up to eight pairs of complex conjugate zeros and eight pairs of complex conjugate poles.

The coefficient values that correspond to the set of sixteen zero locations and the set of sixteen pole locations for the poles and zeros shown in Figure 2 are displayed *(on the command-line screen)* by the workbench in the format shown in Figure 4.

```
Feed-Forward (Numerator) Coefficients
 0  1.000
 1 -1.900
 2  0.999
 3 -0.000
 4  0.000
 5 -0.000
 6  0.000
 7 -0.000
 8  0.000
 9 -0.000
10  0.000
11 -0.000
12  0.000
13 -0.000
14  0.000
15 -0.000
16  0.000

Feedback (Denominator) Coefficients
 1 -1.760
 2  0.774
 3 -0.000
 4  0.000
 5 -0.000
 6  0.000
 7 -0.000
 8  0.000
 9 -0.000
10  0.000
11 -0.000
12  0.000
13 -0.000
14  0.000
15 -0.000
16  0.000
```

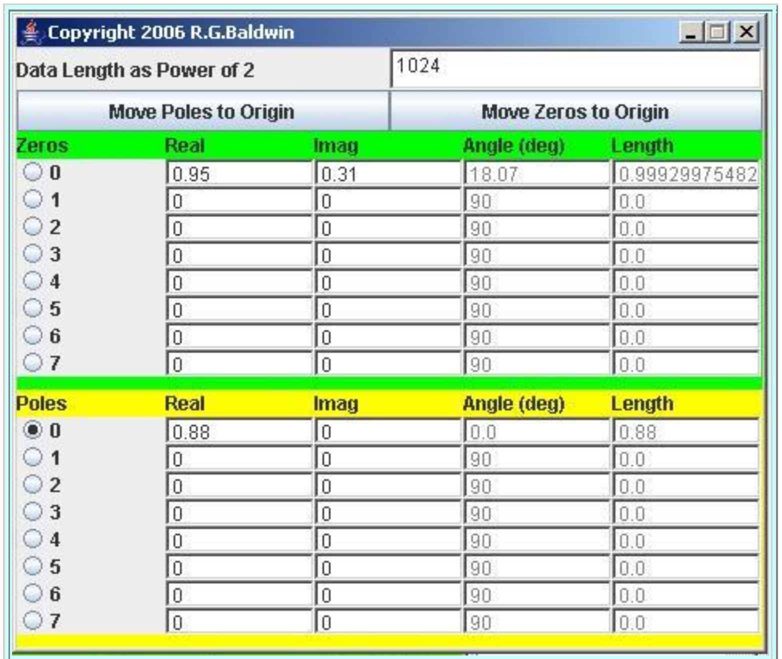
Figure 4

The coefficients shown in Figure 4 include the poles and zeros shown at the origin in Figure 2.

> *(Note that when fourteen of the poles and fourteen of the zeros are located at the origin as is the case in Figure 2, most of the polynomial coefficients have a value of zero.)*

The numeric input/output panel

Figure 5 shows a full-size representation of the reduced image shown in the top-right of the screen shot of Figure 1. The information in Figure 5 corresponds to the pole and zero locations shown in the z-plane in Figure 2.


Copyright 2006 R.G.Baldwin

Data Length as Power of 2	1024

Move Poles to Origin	Move Zeros to Origin

Zeros	Real	Imag	Angle (deg)	Length
○ 0	0.95	0.31	18.07	0.99929975482
○ 1	0	0	90	0.0
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Poles	Real	Imag	Angle (deg)	Length
● 0	0.88	0	0.0	0.88
○ 1	0	0	90	0.0
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Figure 5

The data length

The text field in the top right in Figure 5 shows that this pole-zero configuration will be analyzed using an impulse response having a length of 1024 samples, and that the Fourier Transform of the impulse response will be computed at 1024 points between a frequency of zero and the sampling frequency. We will see the results of that analysis later.

Numeric pole and zero coordinates

The grid of radio buttons and text fields bordered by green in Figure 5 contains information about eight pairs of complex-conjugate zeros that are created and analyzed by the workbench. The grid of radio button and text fields bordered by yellow contain information about eight pairs of complex conjugate poles that are created and analyzed by the workbench.

> *(A recursive filter can also be described by individual poles and zeros on the real axis. However, to reduce the complexity of the program, this workbench always deals with pole and zeros in complex conjugate pairs. If a pair is placed on the real axis, as in Figure 2, the two items in the pair are stacked on top of one another. You may want to modify the program to relax this requirement.)*

A complex conjugate pair

Each row of four text fields and a radio button in Figure 5 corresponds to a complex conjugate pair of poles or zeros. As you can see, one text field contains the real coordinate value for the pole or zero. A second text field contains the imaginary coordinate value. The third text field contains the angle in degrees relative to the real axis described by a vector drawn from the origin to one of the poles or zeros in the pair. The fourth text field contains the length of that vector.

Disabled text fields

The text fields containing angle and length information are disabled insofar as user input is concerned. Thus, they serve to display the angle and length values computed on the basis of the contents of the real and imaginary fields.

The radio buttons

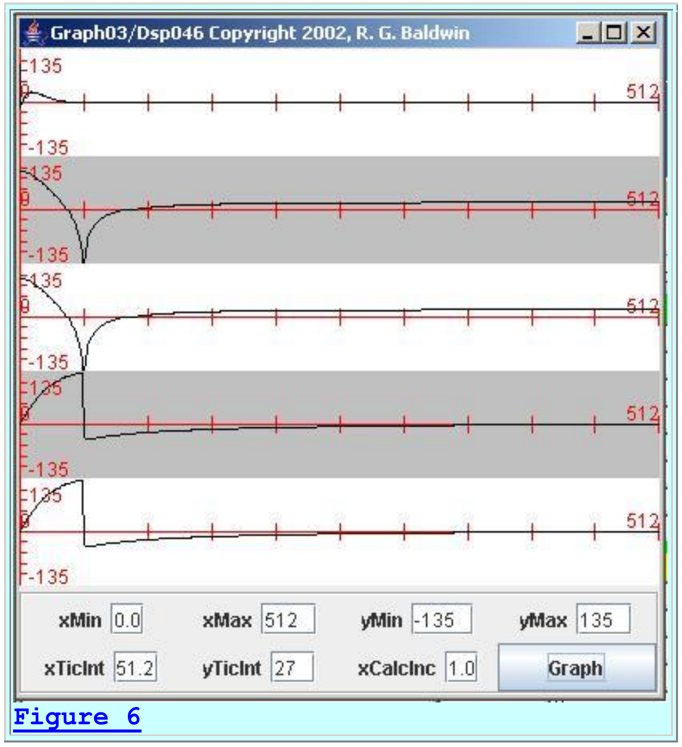
Only one of the radio buttons shown in Figure 5 can be selected at any point in time. When one of the radio buttons is selected and the mouse is clicked in the z-plane shown in Figure 2, the selected pole or zero is moved to the location of the mouse click in the z-plane.

The coordinate, angle, and length information in the corresponding text fields in Figure 5 are automatically updated to reflect the new location for the pole or zero.

Conversely, if a new numeric value is entered into one of the text fields in Figure 5 to cause the location of the pole or zero to change, the image of the corresponding pole or zero is automatically updated to show the new location in the z-plane in Figure 2.

The behavior of the recursive filter

There are five graphs in Figure 6. The top graph shows the first 512 samples of the impulse response in the time domain of a recursive filter having the pole-zero configuration shown in Figure 2.



xMin 0.0 xMax 512 yMin -135 yMax 135
xTicInt 51.2 yTicInt 27 xCalcInc 1.0

Figure 6

The amplitude response

The second and third graphs in Figure 6 show the amplitude response of that recursive filter *(computed two different ways)* between a frequency of zero and the Nyquist folding frequency. The amplitude response is plotted in decibels.

The decibel scale

Quite a bit of scaling and normalization takes place in the workbench with regard to the amplitude response graphs. For example, the amplitude response graphs are normalized so that the peak in the amplitude response always plots as a value of 100 units.

When the amplitude response graphs are calibrated in such a way as to take the scaling and normalization into account, the vertical scale comes out to be approximately 0.2 decibels per plotting unit. Thus, the amplitude response graphs shown in Figure 6 cover a range of 270 units or 54 decibels.

Each factor of two change in the amplitude response represents a change of approximately six decibels. Therefore, a range of 54 decibels represents a dynamic range in the amplitude response of approximately a factor of 512.

The phase response

The fourth and fifth graphs in Figure 6 show the phase response of the recursive filter *(also computed two different ways)* over the same frequency range as the amplitude-response graphs. The phase response is plotted in degrees

The effect of the pair of poles

The peak in the amplitude at a frequency of zero in Figure 6 is caused by the pair of poles that reside on the real axis near the intersection of the real axis and the unit circle in Figure 2.

The effect of the pair of zeros

The notch in the amplitude response and the corresponding rapid change in the phase response in Figure 6 are caused by the pair of complex conjugate zeros that are very close to the right side of the unit circle in Figure 2.

The unit circle

The top half of the unit circle in Figure 2 maps into the frequency axes in the four bottom graphs in Figure 6.

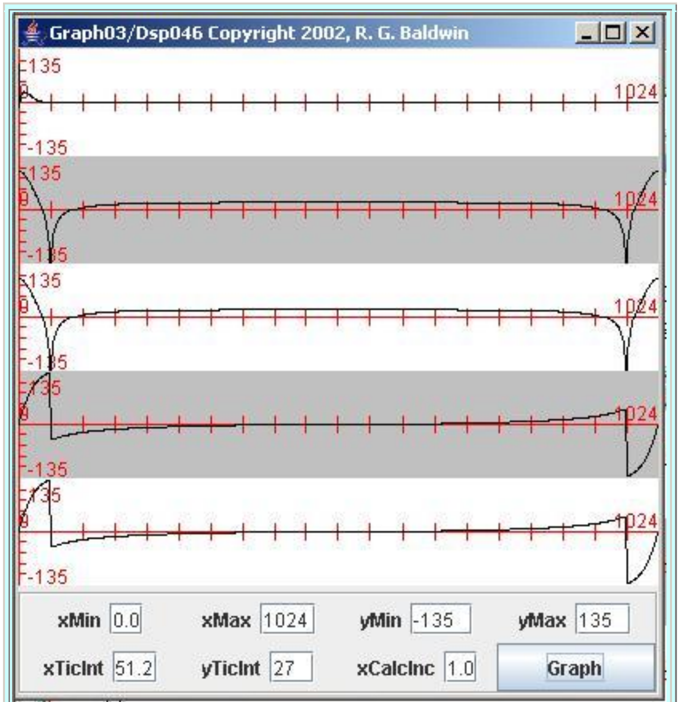
The right-most intersection of the unit circle and the real axis corresponds to a frequency of zero in Figure 6.

The left-most intersection of the unit circle and the real axis corresponds to the Nyquist folding frequency *(one-half the sampling frequency)* at the extreme right end of Figure 6.

The top-most intersection of the unit circle and the imaginary axis in Figure 2 corresponds to one-fourth the sampling frequency *(the center of the horizontal frequency axes in Figure 6)*.

The bottom half

The bottom half of the unit circle in Figure 2 maps into the frequency range from the Nyquist folding frequency to the sampling frequency.   This corresponds to the frequency range from the center of the horizontal frequency axes to the right-most end of the frequency axes in Figure 7.



xMin 0.0 xMax 1024 yMin -135 yMax 135

xTicInt 51.2 yTicInt 27 xCalcInc 1.0 Graph

Figure 7

(The horizontal scale for Figure 7 was modified relative to Figure 6 so as to show the entire amplitude and phase response from zero to the sampling frequency. The Nyquist folding frequency is at the right end of the frequency axes in Figure 6, and is at the center of the frequency axes in Figure 7.)

Represents the complete unit circle

Thus, the frequency axes in Figure 7 represent all of the points on the entire unit circle in Figure 2.

Note that the amplitude response is symmetric about the Nyquist folding frequency at the center of Figure 7, and the phase response is asymmetric about the Nyquist folding frequency. Therefore, no new information is imparted by observing the plotted results to the right of the folding frequency relative to the results to the left of the folding frequency. As a result, I will often ignore the results to the right of the folding frequency in this lesson.

A point represents a frequency

Now let's turn our attention back to Figure 6. A specific point somewhere on the top half of the unit circle in Figure 2 represents a specific frequency somewhere along a frequency axis in Figure 6. The value of the amplitude response at that frequency depends entirely on the distances from that point on the unit circle to the locations of all the poles and all the zeros in the z-plane.

A nearby zero

For example, a zero on or near the point on the unit circle will cause a low value in the amplitude response at that frequency unless offset by poles located nearby. Thus, the zero near the unit circle in Figure 2 causes a notch in the amplitude response shown at that frequency in Figure 6.

A nearby pole

A pole near the point on the unit circle will cause a high value in the amplitude response at that frequency unless offset by zeros located nearby. Thus, the pair of poles located on the real axis near the intersection of the real axis and the unit circle in Figure 2 causes the peak at zero frequency in the amplitude response shown in Figure 6.

An unstable recursive filter

Before I forget it, let me mention that the existence of any pole outside the unit circle in the z-plane will cause the recursive filter to be unstable. This will cause the output values produced by the recursive filter to increase without bounds until the arithmetic processor overflows in some manner.

Computing the amplitude response

The amplitude response of the recursive filter at a given frequency *(a point on the unit circle)* can be determined by computing the product of the distances from that point on the unit circle to all of the zeros and dividing that value by the product of the distances from the same point on the unit circle to all of the poles.

The amplitude response at all frequencies between zero and the Nyquist folding frequency can be determined by performing this computation at all points on the top half of the unit circle.

> *(Because there are an infinite number of points on the top half of the unit circle, we must choose the set of points at which to perform the computation.)*

The amplitude response shown in the third graph in Figure 6 was computed in this manner. *(A little later, I will explain the method that was used to compute the amplitude response shown in the second graph in Figure 6.)*

Poles and zeros at the origin

When poles and zeros are located at the origin, the distance to those poles and zeros from any point on the unit circle is 1.0. Therefore, multiplying or dividing by that distance is of no consequence, and poles and zeros located at the origin have no effect on the amplitude response of the recursive filter.

Computing the phase angle

The phase angle of the recursive filter at a particular frequency *(represented by a point on the unit circle)* can be determined by subtracting the sum of the angles from the point on the unit circle to all of the poles from the sum of the angles from the same point on the unit circle to all of the zeros.

The phase response in the bottom graph of Figure 6 was computed using this method.

Getting the recursive filter coefficients

The primary objective of a recursive filter design effort is not to identify the pole and zero locations necessary to produce the desired recursive filtering behavior. Rather, it is to obtain the feed-forward and feedback coefficients to use in the application of the recursive filtering algorithm to produce the desired behavior.

So far, we have only discussed the locations of the poles and zeros for the denominator and numerator polynomials. The next thing that we need to do is to use those pole and zero locations to determine the values of the coefficients for the numerator and denominator polynomials.

This is the easy part

The zeros represent the roots of the numerator polynomial and the poles represent the roots of the denominator polynomial. All that is necessary to obtain the polynomial coefficients is to multiply the roots. For example, the real and imaginary values for the first zero in Figure 5 would require us to multiply the following roots:

```
(z-0.95+0.31j)*(z-0.95-0.31j)
```

The multiplication of these roots produces a second-order polynomial with the *Feed-Forward (Numerator) Coefficients* shown in Figure 3.

Similarly, the denominator coefficients are obtained by using the pole values to construct the roots of the denominator polynomial and to multiply those roots to produce the coefficients for the denominator polynomial.

Must implement the recursive filtering algorithm

Having obtained the feed-forward and feedback coefficients, the next step is to install those coefficients in a recursive filtering algorithm and to use it to filter the sampled data of interest. Once we do that, we need a way to confirm that the coefficients are being used correctly and that the recursive filtering algorithm is behaving as it should. One way to obtain this confirmation is to:

- Apply the recursive filter to a time series consisting of a single non-zero value followed by sample values of zero thereafter. *(The output of such a filtering operation is commonly referred to as the impulse response of the filter.)*
- Perform a Fourier Transform on the impulse response.

- Compare the amplitude response and the phase response produced by the Fourier Transform of the impulse response to that produced using the poles and zeros as described above.

Do they match?

If the amplitude and phase responses produced using the two different approaches match, this is a very good indication that the filter coefficients are being correctly applied in the recursive filtering process. If they don't match, the recursive filtering process should be subject to question.

The results for this example

The impulse response is shown in the first graph in Figure 6. The amplitude response shown in the second graph in Figure 6 was produced by performing a Fourier Transform on the impulse response.

The phase response shown in the fourth graph in Figure 6 was produced by performing the same Fourier Transform on the impulse response.

The amplitude and phases responses produced by applying the Fourier Transform to the impulse response match those produced using the pole and zero locations in the third and fifth graphs in Figure 6. This is a strong indication that the recursive filtering process in the workbench is implemented correctly.

Practical Examples

Now that you understand how to control the locations of the poles and zeros, and understand how those locations impact the behavior of the recursive filter, it's time to take a look at a few practical examples.

The complete source code for the workbench is provided for your benefit in Listing 1 near the end of the lesson. If you wish to do so, you can copy, compile, and execute that source code and follow along for the practical examples that follow.

Summary of the behavior of the workbench program

Before getting into the details of the examples, I will summarize the behavior of the workbench program.

This program provides a visual, interactive recursive filtering workbench. The purpose of the program is to make it easy to experiment with the behavior of recursive filters and to visualize the results of those experiments.

Complex poles and zeros

The program implements a recursive filter having eight pairs of complex conjugate poles and eight pairs of complex conjugate zeros. The locations of the pairs of poles and zeros in the z-plane are controlled by the user.

Although the pairs of poles and zeros can be co-located on the real axis, the program does not support the placement of individual poles and zeros on the real axis.

The user can reduce the number of poles and zeros that impact the behavior of the recursive filter by moving excess poles and zeros to the origin in the z-plane. This renders them ineffective in the behavior of the recursive filter.

The impulse response

The program provides three interactive displays on the screen. The first display *(see Figure 6)* contains five graphs in a vertical stack. The first graph in this display shows the impulse response of the recursive filter in the time domain.

The amplitude response

The second and third graphs show the amplitude response of the recursive filter in the frequency domain computed using two different computational approaches. The two different computational approaches are provided for comparison purposes.

The first computational approach for computing the amplitude response is to perform a Fourier Transform on the impulse response using an FFT algorithm. The quality of the estimate of the amplitude response using this approach is dependent on the extent to which the entire impulse response is captured in the set of samples used to perform the FFT. If the impulse response is truncated, the estimate will be degraded.

The second approach for computing the amplitude response involves computing the product of the vector lengths from each point on the unit circle to each of the poles and each of the zeros in the z-plane. This approach provides an idealized estimate of the amplitude response of the recursive filter, unaffected by impulse-response length considerations. This approach produces the same results that should be produced by performing the FFT on a set of impulse-response samples of sufficient length to guarantee that the values in the impulse response have damped down to zero *(the impulse response is totally captured in the set of samples on which the FFT is performed).*

The phase response

The fourth and fifth graphs in Figure 6 show the phase response of the recursive filter computed using similar approaches to those described above for the amplitude response. *(The second approach uses the sum of vector angles instead of the product of vector lengths.)* Once again, the two approaches are provided for comparison purposes.

Computational data length

By default, the program computes and captures the impulse response for a length of 1024 samples and performs the Fourier Transform on that length. However, the length of the captured impulse response and the corresponding FFT can be changed by the user to any length between 2 samples and 16384 samples, provided that the length is an even power of two. *(If the length specified by the user is not an even power of two, it is automatically changed to an even power of two by the program.)*

Interactive plotting parameters

The display shown in Figure 6 is interactive in the sense that there are seven different plotting parameters *(such as vertical scale, horizontal scale, location of tic marks, etc.)* that can be adjusted by the user to produce plots that are visually useful. The user can modify any of the parameters and then click the **Graph** button to cause the graphs to be re-plotted using the new parameters.

Poles and zeros in the z-plane

The second display *(see Figure 2)* shows the locations of all of the poles and all the zeros in the z-plane. The user can use the mouse to change the location of any pair of complex conjugate poles or zeros by first selecting a specific pair of poles or zeros and then clicking the new location in the z-plane.

This interactive capability makes it possible for the user to modify the design of the recursive filter in a completely graphic manner by positioning the poles and zeros in the z-plane with the mouse.

Having relocated one or more pairs of poles or zeros in the z-plane, the user can then click the **Graph** button in Figure 6 to cause the impulse response, the amplitude response, and the phase response of the new recursive filter with the modified pole and zero locations to be computed and displayed.

A general user control panel

The third display *(see Figure 5)* is a control panel that uses text fields, ordinary buttons, and radio buttons to allow the user to perform the following tasks:

- Specify a new length for the impulse response as an even power of two. *(Once again, if the user fails to specify an even power of two, the value provided by the user is automatically converted to an even power of two by the program.)*
- Cause all of the poles to be moved to the origin in the z-plane.
- Cause all of the zeros to be moved to the origin in the z-plane.
- Select a pair of complex conjugate poles or zeros to be relocated using the mouse in the display of the z-plane.
- View the angle *(relative to the horizontal axis in the z-plane)* described by an imaginary vector that connects each pole and each zero to the origin.

- View the length of an imaginary vector that connects each pole and each zero to the origin in the z-plane.
- Enter *(into a text field)* a new real or imaginary value specifying the location of a pair of complex conjugate poles or zeros in the z-plane.

Entering a new location value

When a new real or imaginary value is entered into a text field, the angle and the length are automatically updated to reflect the new location of the pole or zero and the display of the pole or zero in the z-plane is also updated to show the new location.

Clicking a new location in the z-plane

Conversely, when the mouse is used to relocate a pole or zero in the z-plane, the corresponding real and imaginary values in the text fields and the corresponding angle and length in Figure 5 are automatically updated to reflect the new location for the pole or zero.

Entry of invalid data

Note that mainly for convenience *(but for technical reasons as well)*, the angle and length in the text fields and the location of the pole or zero in the Z-plane display are updated as the user enters each character into the text field.

If the user enters text into the text field that cannot be converted into a numeric value of type **double**, *(such as the pair of characters "-." for example)* the contents of the text field are automatically converted to a single "0" character. A warning message is displayed on the command-line screen when this happens.

There is no long-term harm when this occurs, but the user may need to start over to enter the new value. Thus, the user should exercise some care regarding the order in which the characters in the text field are modified when entering new real and imaginary values.

Coefficients displayed on the command-line screen

Each time the impulse response and the spectral data are plotted, the seventeen feed-forward filter coefficients and the sixteen feedback coefficients used by the recursive filter to produce the output being plotted are displayed on the command-line screen.

Usage

This program requires access to the following source code or class files, which were published in earlier tutorials *(see the References section of this document)*:

- GraphIntfc01
- ForwardRealToComplexFFT01
- Graph03

Having compiled all the source code, enter the following command at the command prompt to run this program:

```
java Graph03 Dsp046
```

The program was tested using J2SE 5.0 under WinXP. J2SE 5.0 or later is required due to the use of static imports and the **printf** method.

The behavior at startup

The workbench establishes a set of default pole and zero locations on startup. Among other reasons, this is done to confirm that the workbench program is operating properly.

When you first start the program and make a few adjustments to the plotting parameters, the three GUIs that appear on your screen should match those shown in Figure 8, Figure 9, and Figure 10.

> *(To adjust the plotting parameters, modify the values in the text fields at the bottom and click the **Graph** button Figure 10.)*

Sixteen poles and sixteen zeros

Figure 8 shows the locations of eight pairs of complex conjugate poles and eight pairs of complex conjugate zeros near the unit circle in the z-plane. These are the default pole and zero locations at startup.

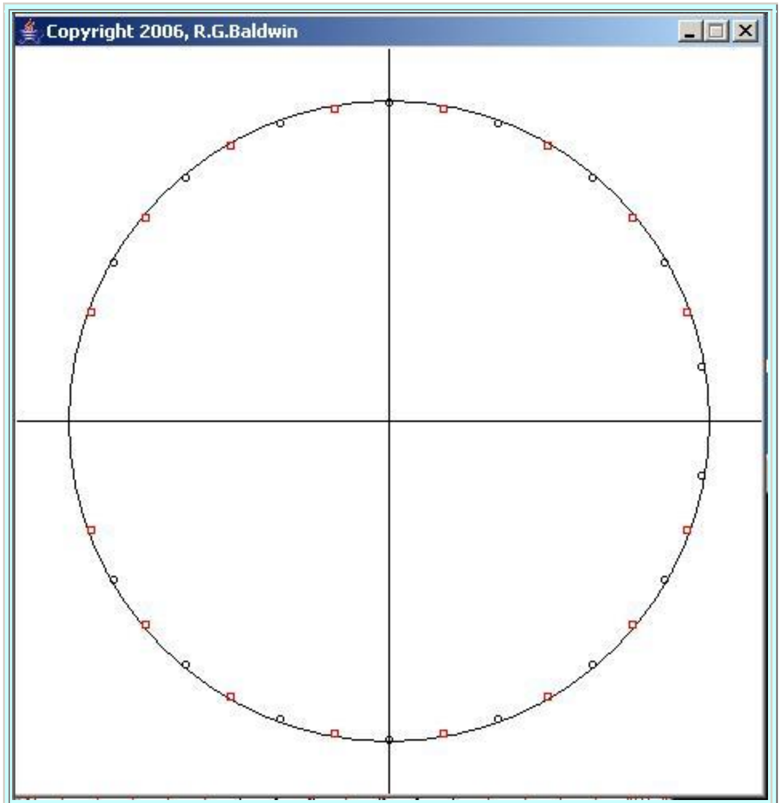
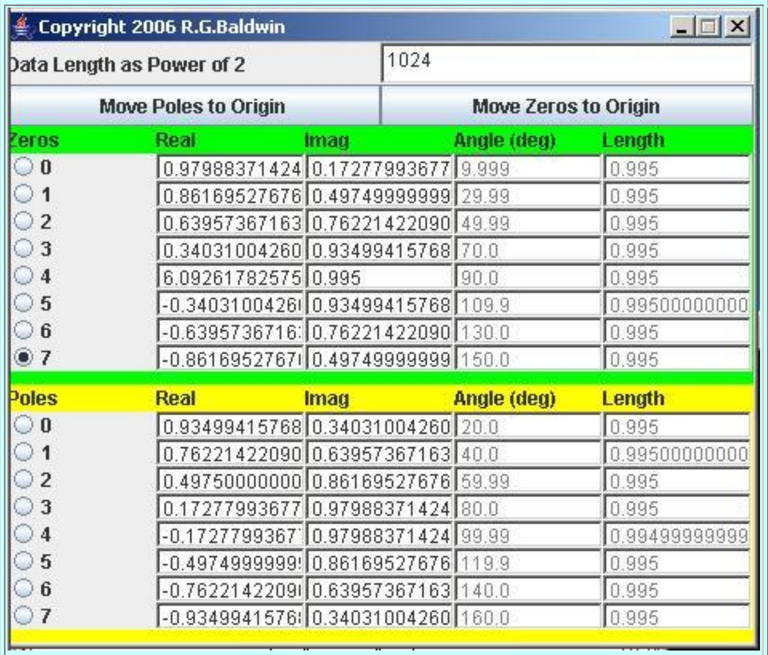


Figure 8

The poles and zeros in Figure 8 are located at a distance of 0.995 from the origin. This is very close to, but not on the unit circle.

Located at incremental angles

Figure 9 shows the poles to be located at angles that are multiples of 20 degrees relative to the real axis. Thus, the poles cover the angular range from 20 degrees through 160 degrees inclusive.


Copyright 2006 R.G.Baldwin

Data Length as Power of 2: 1024

Zeros	Real	Imag	Angle (deg)	Length
0	0.97988371424	0.17277993677	9.999	0.995
1	0.86169527676	0.49749999999	29.99	0.995
2	0.63957367163	0.76221422090	49.99	0.995
3	0.34031004260	0.93499415768	70.0	0.995
4	6.09261782575	0.995	90.0	0.995
5	-0.3403100426	0.93499415768	109.9	0.99500000000
6	-0.6395736716	0.76221422090	130.0	0.995
● 7	-0.8616952767	0.49749999999	150.0	0.995

Poles	Real	Imag	Angle (deg)	Length
0	0.93499415768	0.34031004260	20.0	0.995
1	0.76221422090	0.63957367163	40.0	0.99500000000
2	0.49750000000	0.86169527676	59.99	0.995
3	0.17277993677	0.97988371424	80.0	0.995
4	-0.1727799367	0.97988371424	99.99	0.99499999999
5	-0.4974999999	0.86169527676	119.9	0.995
6	-0.7622142209	0.63957367163	140.0	0.995
7	-0.9349941576	0.34031004260	160.0	0.995

Figure 9

Figure 9 shows that the zeros are also located at angular intervals of 20 degrees with the first zero being located at 10 degrees relative to the real axis. Thus, the zeros are half way between the poles.

A data length of 1024 samples

In addition, Figure 9 shows that the workbench will capture and perform a Fourier analysis on 1024 samples of impulse response data.

(The workbench is limited to a maximum data length of 16384 samples.)

The impulse response and the Fourier analysis

The 1024 samples of impulse response data are shown in the top graph in Figure 10. It is important to note that the values in the impulse response don't damp down to near zero until about 700 samples. I will have more to say about this later.

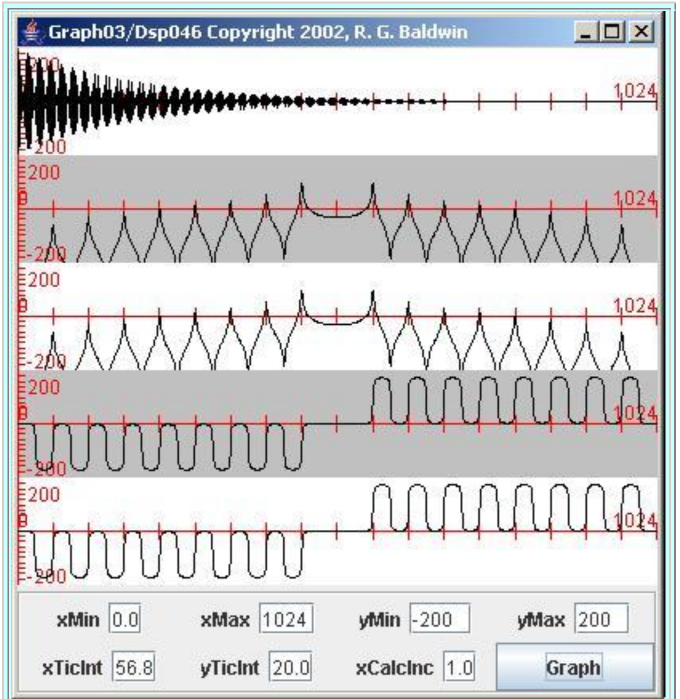


Figure 10

The amplitude and phase response

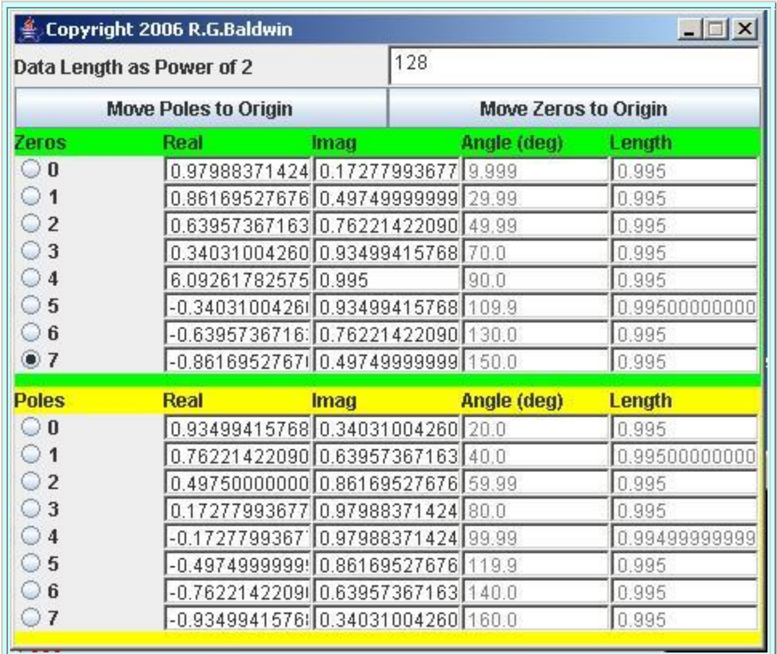
The remaining four graphs in Figure 10 show the amplitude response and the phase response computed using both methods discussed earlier and displayed from zero frequency to the sampling frequency.

If you examine the amplitude response carefully, you will see sixteen peaks corresponding to the locations of the sixteen poles shown in Figure 8. You will also see sixteen notches corresponding to the sixteen zeros shown in Figure 8.

Similarly, the big changes in the phase response occur at the frequency locations of the poles and the zeros.

Insufficient data length

Figure 11 and Figure 12 show the results of leaving the poles and zeros in exactly the same locations but reducing the length of the data that contains the impulse response from 1024 samples to 128 samples. This is accomplished by entering the new data length value of 128 in the upper-right text field in Figure 11 and clicking the **Graph** button in Figure 12.



Copyright 2006 R.G.Baldwin

Data Length as Power of 2: 128

Move Poles to Origin | Move Zeros to Origin

Zeros	Real	Imag	Angle (deg)	Length
○ 0	0.97988371424	0.17277993677	9.999	0.995
○ 1	0.86169527676	0.49749999999	29.99	0.995
○ 2	0.63957367163	0.76221422090	49.99	0.995
○ 3	0.34031004260	0.93499415768	70.0	0.995
○ 4	6.09261782575	0.995	90.0	0.995
○ 5	-0.3403100426	0.93499415768	109.9	0.99500000000
○ 6	-0.6395736716	0.76221422090	130.0	0.995
● 7	-0.8616952767	0.49749999999	150.0	0.995

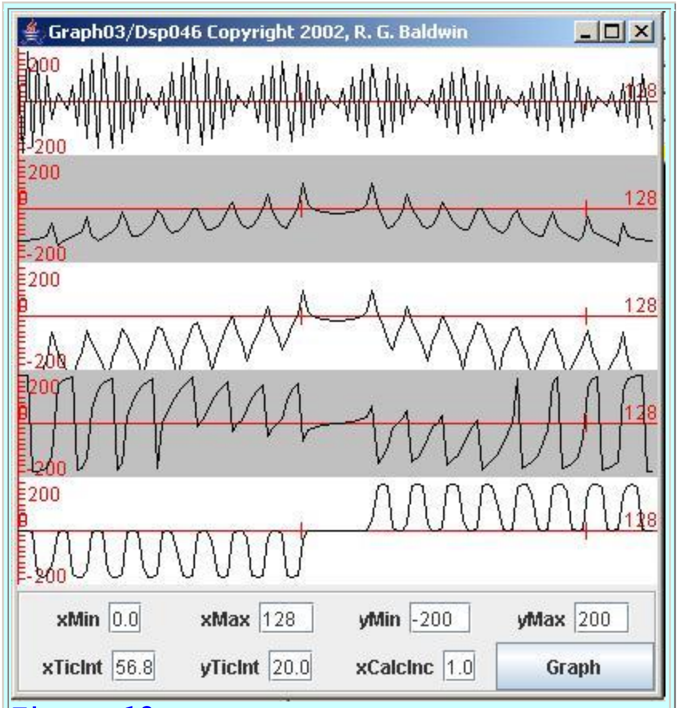
Poles	Real	Imag	Angle (deg)	Length
○ 0	0.93499415768	0.34031004260	20.0	0.995
○ 1	0.76221422090	0.63957367163	40.0	0.99500000000
○ 2	0.49750000000	0.86169527676	59.99	0.995
○ 3	0.17277993677	0.97988371424	80.0	0.995
○ 4	-0.1727799367	0.97988371424	99.99	0.99499999999
○ 5	-0.4974999999	0.86169527676	119.9	0.995
○ 6	-0.7622142209	0.63957367163	140.0	0.995
○ 7	-0.9349941576	0.34031004260	160.0	0.995

Figure 11

Note also that the **xMax** plotting parameter in Figure 12 was reduced from 1024 to 128 so that only 128 samples are displayed horizontally in the graphs.

A truncated impulse response

As I mentioned earlier, about 700 samples are required for the impulse response to damp down to near zero. As you can see in the top graph of Figure 12, that portion of the impulse response captured and analyzed for this example was only a small portion of the total impulse response.



xMin 0.0 xMax 128 yMin -200 yMax 200
xTicInt 56.8 yTicInt 20.0 xCalcInc 1.0

Figure 12

A poor match to the actual amplitude and phase response

The Fourier Transform of the truncated impulse response provided a poor estimate of the behavior of the recursive filter.

The estimate of the amplitude response produced by the Fourier Transform shown in the second graph in Figure 12 is a poor match for the amplitude response shown in the third graph.

Similarly, the estimate of the phase response produced by the Fourier Transform shown in the fourth graph is a poor match for the phase response shown in the fifth graph.

Not a problem with the recursive filter

This doesn't mean that there was something wrong with the recursive filter. It didn't change. Rather, the change was made in one of the computational approaches used to estimate the amplitude and phase response of the recursive filter. The amount of data captured and subjected to Fourier analysis was simply insufficient to provide good results.

Don't truncate the impulse response

The bottom line is that when using Fourier analysis to confirm proper operation of a recursive filter, you must be careful to capture the entire impulse response instead of truncating it and performing Fourier analysis on a truncated version of the impulse response.

Moving poles and zeros to the origin

Figure 11 shows two buttons having the following labels:

- Move Poles to Origin
- Move Zeros to Origin

Clicking these buttons resets the real and imaginary values to values of 0.0 for all of the poles in one case and all of the zeros in the other case. This causes all of the poles and all of the zeros to be moved to the origin of the z-plane in a wholesale fashion. These buttons can be used to reset the locations of the poles and zeros whenever appropriate.

An extremely narrow band filter

Now I am going to show you an example that illustrates the issue of computational cost in an extreme way. Suppose that for some reason you wanted to build an extremely narrow band filter that would pass a very narrow range of frequencies centered at 33.32222 percent of the folding frequency. You could accomplish this by placing a single pair of complex conjugate poles very near to the unit circle as shown in Figure 13.

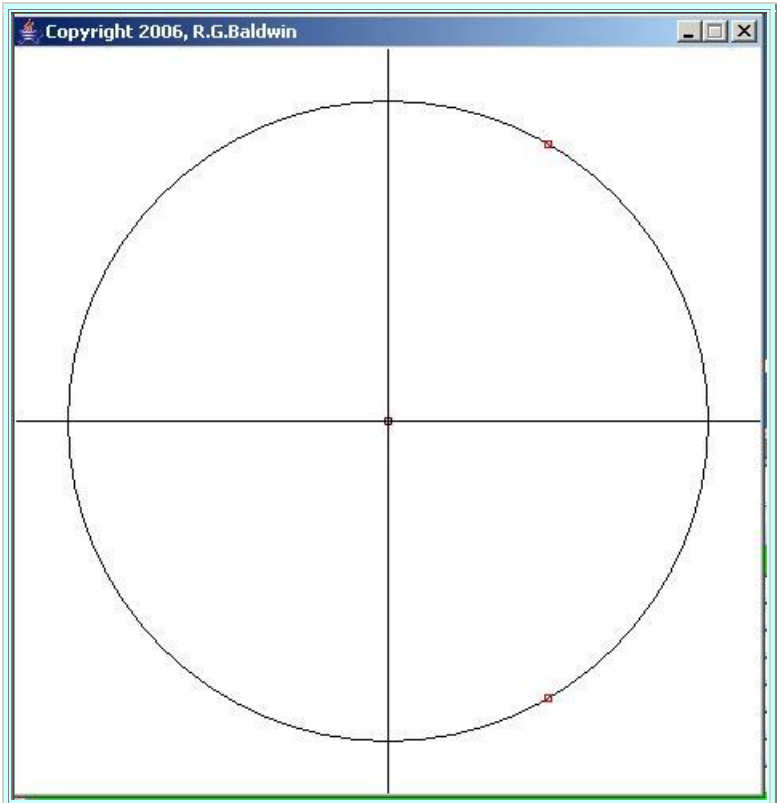
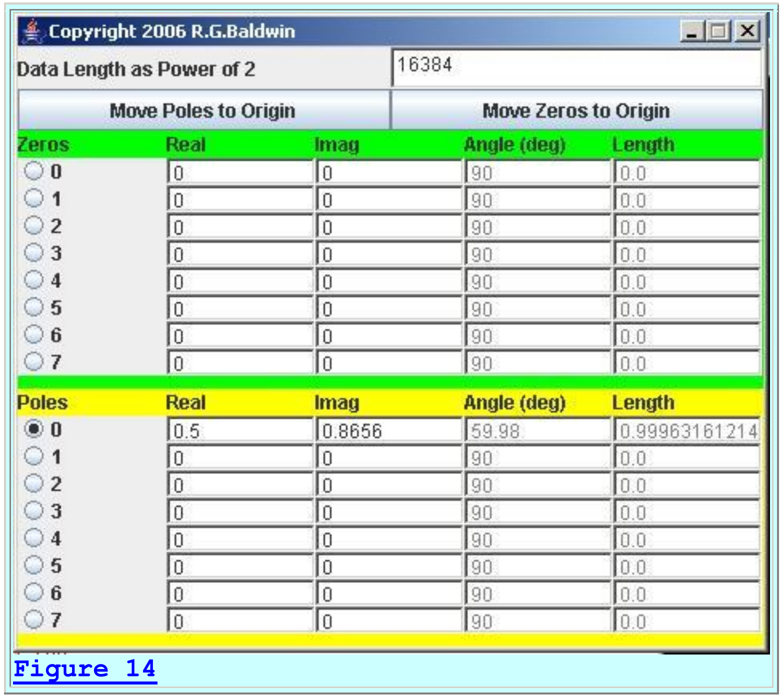


Figure 13

The coordinate values

The real and imaginary coordinate values for this pole are shown in Figure 14.



Data Length as Power of 2 | 16384

Zeros	Real	Imag	Angle (deg)	Length
○ 0	0	0	90	0.0
○ 1	0	0	90	0.0
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Poles	Real	Imag	Angle (deg)	Length
● 0	0.5	0.8656	59.98	0.99963161214
○ 1	0	0	90	0.0
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Figure 14

A very long impulse response

Figure 14 also shows that the Fourier analysis of the impulse response will be performed on a filter output time series containing 16384 samples.

The first 8192 samples of that impulse response are shown in Figure 15. As you can see, almost 8192 output samples from the recursive filter are required for the values in the impulse response to damp down to near zero.

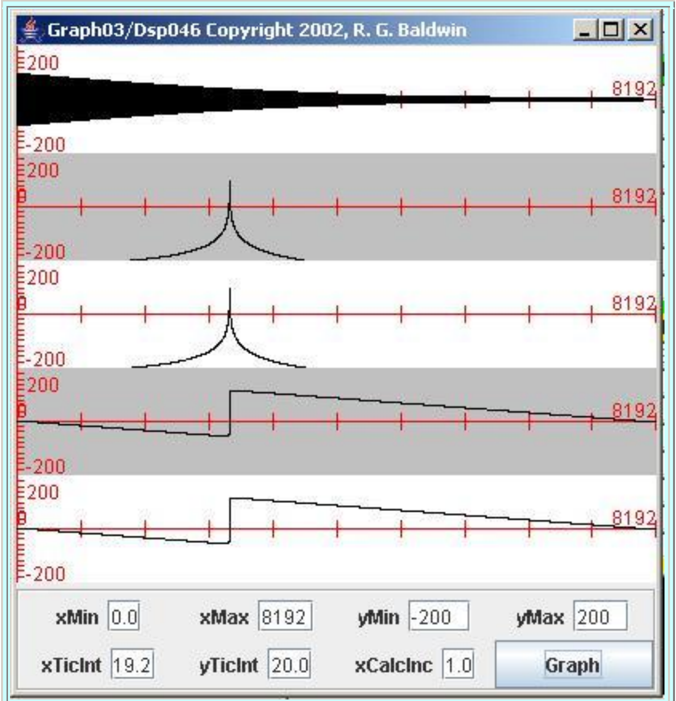


Figure 15

Figure 15 also shows the amplitude and phase response for the recursive filter with a very narrow pass band at the desired frequency.

The filter coefficients

Perhaps the most important information for this demonstration is shown in Figure 16. This figure shows that this filter can be achieved with one non-zero coefficient in the feed-forward loop *(with a value of 1.0)* and only two non-zero coefficients in the feedback loop.

```
Feed-Forward (Numerator) Coefficients
0  1.000

Feedback (Denominator) Coefficients
1 -1.000
2  0.999
```
Figure 16

The computational cost

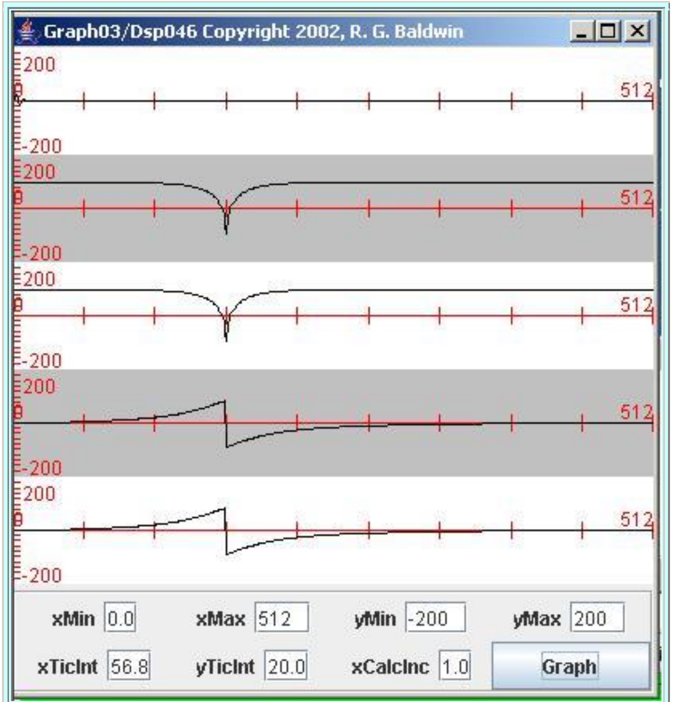
Thus, a recursive filtering algorithm designed to apply no zeros and only one pair of complex conjugate poles could accomplish this narrow-band filtering task with only about three MADs required per output sample.

To do exactly the same job using a non-recursive convolution filter would require a convolution filter having an impulse response identical to that shown in Figure 15. The impulse response of a non-recursive convolution filter simply traces out of the filter coefficients. Stated differently, the filter coefficients for a non-recursive convolution filter are the same as the values in the sampled impulse response.

In general, the application of a convolution filter requires one MAD per coefficient per output sample, so the use of a non-recursive convolution filter in this case would require thousands of MADs per output sample. The reduction in the computational cost of the recursive filter relative to a non-recursive convolution filter for this task would be very significant.

A narrow notch filter

Now suppose that for some reason you need to build a narrow notch filter at a frequency of 33.3333 percent of the folding frequency, as shown by the amplitude response graphs in Figure 17. In addition to a narrow notch, you need to keep the amplitude response as flat as practical at frequencies other than the frequency of the notch.



xMin 0.0 xMax 512 yMin -200 yMax 200
xTicInt 56.8 yTicInt 20.0 xCalcInc 1.0

Figure 17

Figure 17 shows the amplitude and phase response from zero to the Nyquist folding frequency for such a filter.

Pole and zero locations

I was able to design the notch filter in Figure 17 using the mouse to position two pairs of poles and two pairs of zeros in the z-plane as shown in Figure 18.

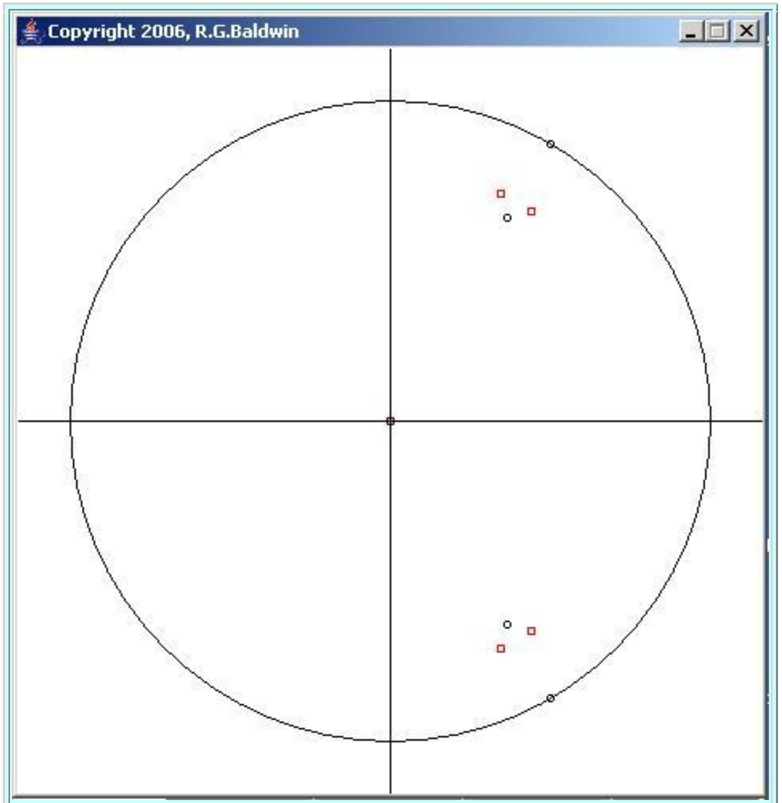


Figure 18

(After finding the approximate locations of the poles and zeros using the mouse, I used a calculator to polish the real and imaginary coordinate values that specify the locations.)

How does this work?

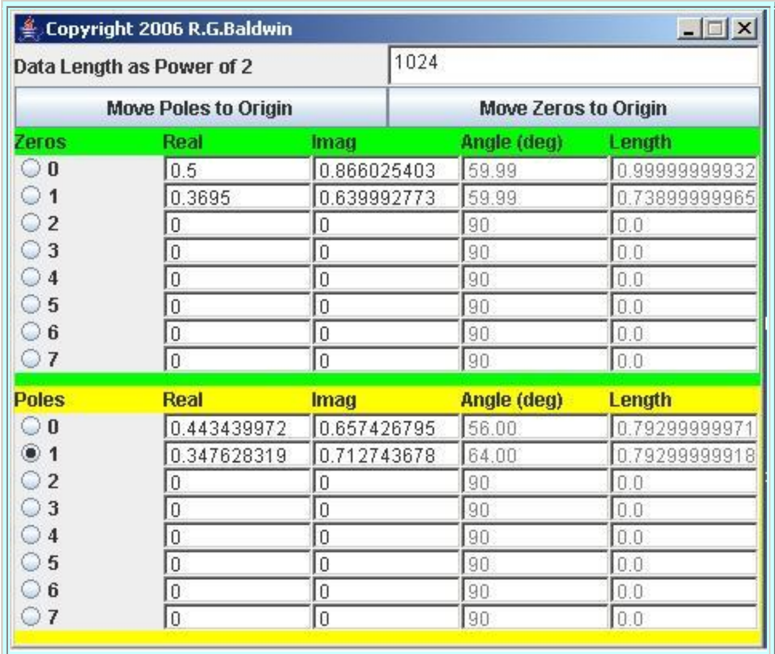
The zero on the unit circle in Figure 18 produces the notch at the desired frequency. Unfortunately, this zero also tends to drag down the response on both sides of the notch.

The pair of poles in Figure 18 helps to bring the response on each side of the notch back up to achieve a narrower notch and a flatter overall response.

The zero between the poles and the origin also tends to cause the overall response to be flatter outside the vicinity of the notch. This is because at points on the unit circle some distance from the notch, there are the same number of poles and zeros all at approximately the same distance from the point on the unit circle. As a result, the product of the distances to the poles is approximately the same as the product of the distances to the zeros. When the product of zero distances is divided by the product of pole distances, the quotient is approximately unity.

Locations of the poles and zeros

The actual real and imaginary coordinate values for the poles and the zeros in this filter are shown in Figure 19. I used the angle and length information shown in Figure 19 in conjunction with the mouse to locate the poles and the zeros. The angle and length information helped me to create a pattern of poles and zeros that were nearly symmetrical with respect to a line at an angle of 60 degrees relative to the real axis. As mentioned earlier, I then went back with a calculator and polished the coordinate values producing the values shown in Figure 19.



Data Length as Power of 2	1024

Zeros	Real	Imag	Angle (deg)	Length
○ 0	0.5	0.866025403	59.99	0.99999999932
○ 1	0.3695	0.639992773	59.99	0.73899999965
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Poles	Real	Imag	Angle (deg)	Length
○ 0	0.443439972	0.657426795	56.00	0.79299999971
● 1	0.347628319	0.712743678	64.00	0.79299999918
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Figure 19

The filter coefficients

The feed-forward and feedback filter coefficients that resulted from this placement of poles and zeros are shown in Figure 20. This recursive filter has a total of nine non-zero coefficients. Therefore, this filter could be realized at a computational cost of nine MADs per output sample.

```
Feed-Forward (Numerator) Coefficients
0  1.000
1 -1.739
2  2.285
3 -1.285
4  0.546

Feedback (Denominator) Coefficients
1 -1.582
2  1.874
3 -0.995
4  0.395
```

Figure 20

A narrower notch filter

Now assume that you need the notch to be even narrower than that shown in Figure 17 and that you are willing to sacrifice some flatness in the overall amplitude response to achieve the narrower notch as shown in Figure 21.

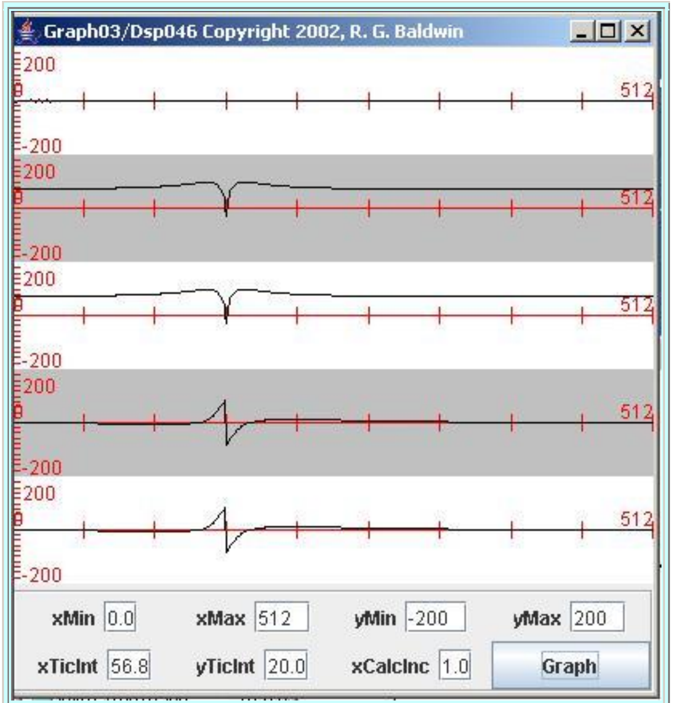


Figure 21

New locations for the poles

This can be accomplished by leaving the zeros alone and pushing the pair of complex conjugate poles in Figure 18 closer to the unit circle and closer to an imaginary line that connects the zero on the unit circle to the origin. This is shown in the z-plane in Figure 22.

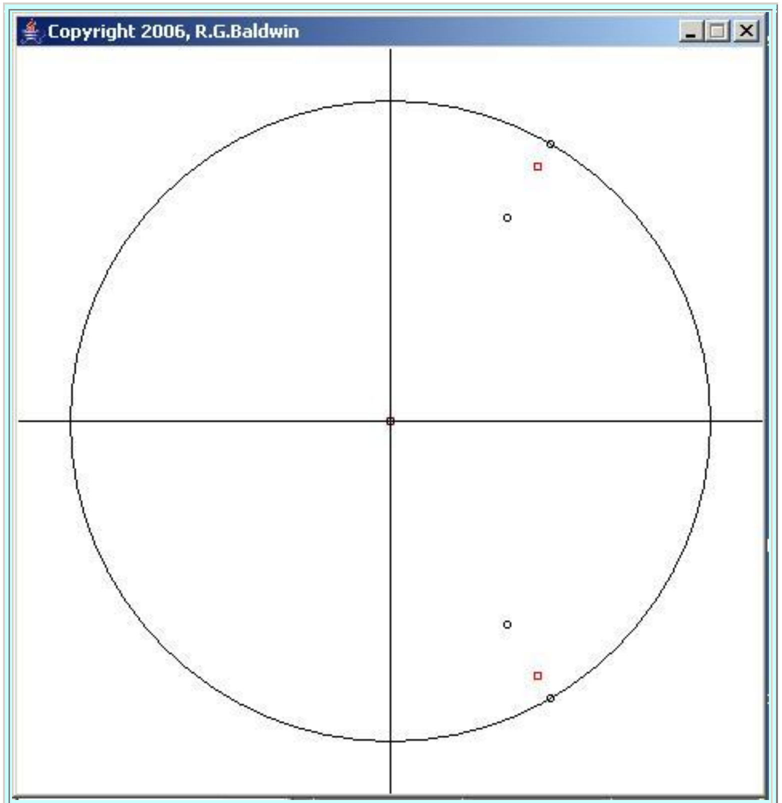
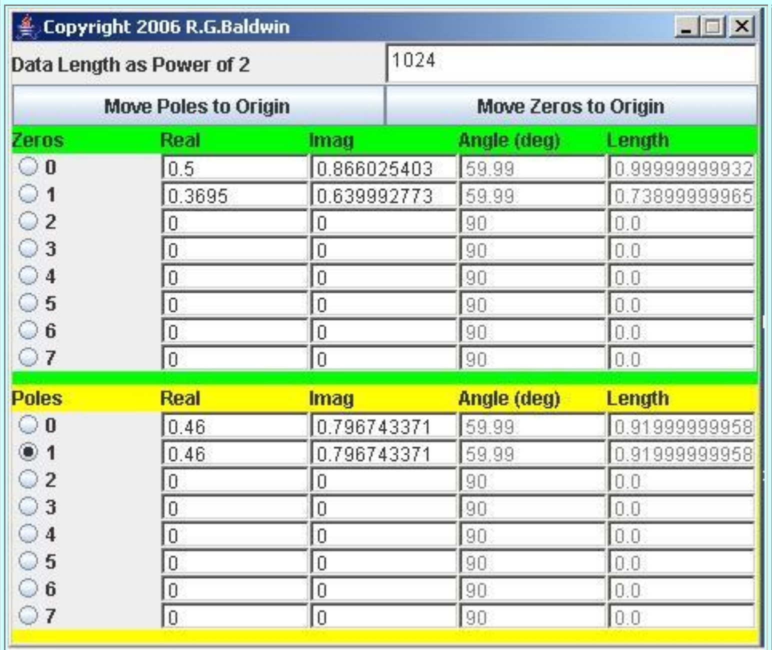


Figure 22

The poles in Figure 22 were pushed so close to the imaginary line mentioned above that they are actually stacked on top of one another on that imaginary line.

Coordinates of the poles and zeros

The numeric coordinates for the poles and zeros in Figure 22 are shown in Figure 23.



Data Length as Power of 2	1024			
Zeros	**Real**	**Imag**	**Angle (deg)**	**Length**
○ 0	0.5	0.866025403	59.99	0.99999999932
○ 1	0.3695	0.639992773	59.99	0.73899999965
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0
Poles	**Real**	**Imag**	**Angle (deg)**	**Length**
○ 0	0.46	0.796743371	59.99	0.91999999958
● 1	0.46	0.796743371	59.99	0.91999999958
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Figure 23

The filter coefficients

Finally, the feed-forward and feedback coefficient values for this filter are shown in Figure 24.

```
Feed-Forward (Numerator) Coefficients
0  1.000
1 -1.739
2  2.285
3 -1.285
4  0.546

Feedback (Denominator) Coefficients
1 -1.840
2  2.539
3 -1.557
4  0.716
```
Figure 24

As before, this filter has a total of nine non-zero coefficients. Therefore, the filter could be realized at a computational cost of nine MADs per output sample.

If you compare Figure 21 with Figure 17, you will see that the notch for this filter is considerably narrower than the notch for the filter shown in Figure 17.

When the poles move to the origin ...

The next thing that I will show you is what happens when all of the poles are moved to the origin. To do this demonstration, I will restart the workbench program causing the poles and zeros to be placed in their default locations as shown in Figure 8. Then I will click the button labeled **Move Poles to Origin** to cause all of the poles to be moved to the origin. This leaves the zeros positioned as shown graphically in Figure 25.

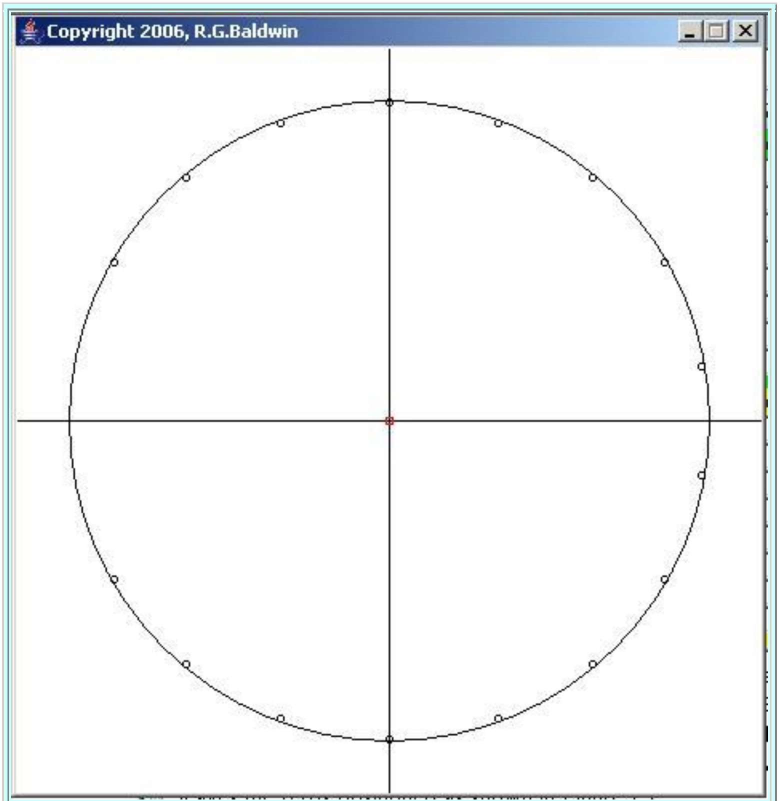
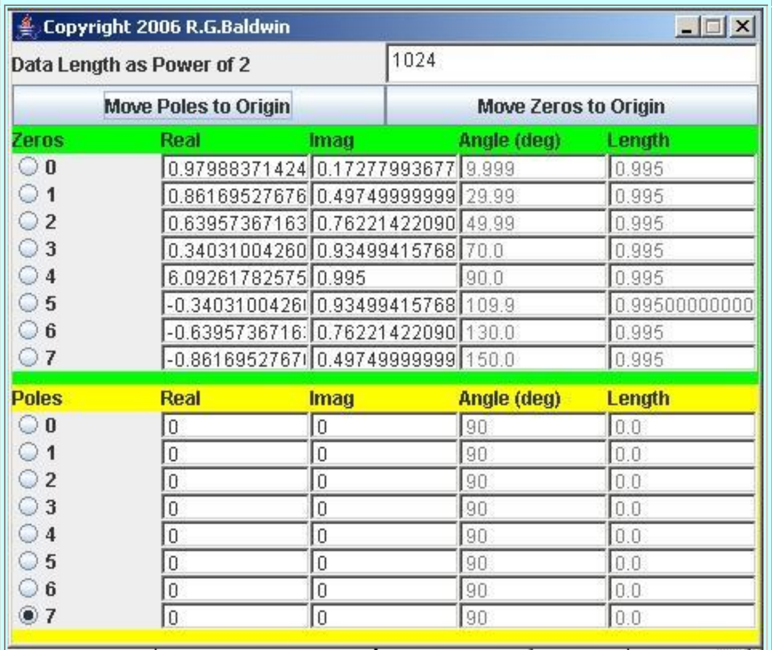


Figure 25

The numeric pole and zero locations

The numeric values for the pole and zero locations are shown in Figure 26.



Data Length as Power of 2: 1024

Zeros	Real	Imag	Angle (deg)	Length
0	0.97988371424	0.17277993677	9.999	0.995
1	0.86169527676	0.49749999999	29.99	0.995
2	0.63957367163	0.76221422090	49.99	0.995
3	0.34031004260	0.93499415768	70.0	0.995
4	6.09261782575	0.995	90.0	0.995
5	-0.3403100426	0.93499415768	109.9	0.99500000000
6	-0.6395736716	0.76221422090	130.0	0.995
7	-0.8616952767	0.49749999999	150.0	0.995

Poles	Real	Imag	Angle (deg)	Length
0	0	0	90	0.0
1	0	0	90	0.0
2	0	0	90	0.0
3	0	0	90	0.0
4	0	0	90	0.0
5	0	0	90	0.0
6	0	0	90	0.0
● 7	0	0	90	0.0

Figure 26

The filter coefficients

As you learned earlier and can confirm from Figure 27, when all of the poles are moved to the origin, all of the feedback coefficients go to zero. This means that none of the output is fed back into the input.

```
Feed-Forward (Numerator) Coefficients
 0  1.000
 1 -1.960
 2  2.851
 3 -3.646
 4  4.324
 5 -4.864
 6  5.251
 7 -5.476
 8  5.532
 9 -5.421
10  5.147
11 -4.720
12  4.154
13 -3.468
14  2.684
15 -1.827
16  0.923

Feedback (Denominator) Coefficients
 1 -0.000
 2  0.000
 3 -0.000
 4  0.000
 5 -0.000
 6  0.000
 7 -0.000
 8  0.000
 9 -0.000
10  0.000
11 -0.000
12  0.000
13 -0.000
14  0.000
15 -0.000
16  0.000
```
Figure 27

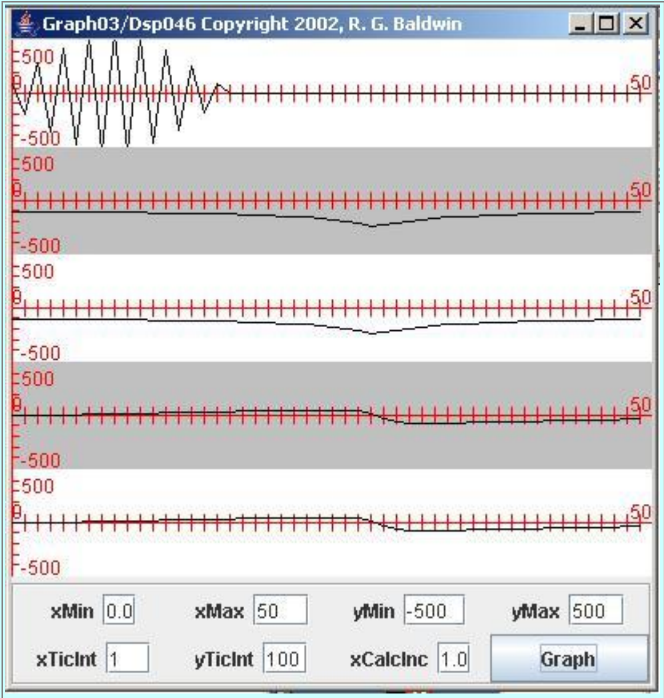
A non-recursive convolution filter

The result of eliminating the feedback in the recursive filter algorithm is to cause that algorithm to behave like a non-recursive convolution filter.

The impulse response of a non-recursive convolution is the same length as the number of coefficients in the convolution filter. Once the impulse moves through all of the filter coefficients, the output goes to zero where it remains from that time forward.

The impulse response of the filter

The top graph in Figure 28 shows that is what happens in this case. *(Simply ignore the other graphs in Figure 28 at this point.)*



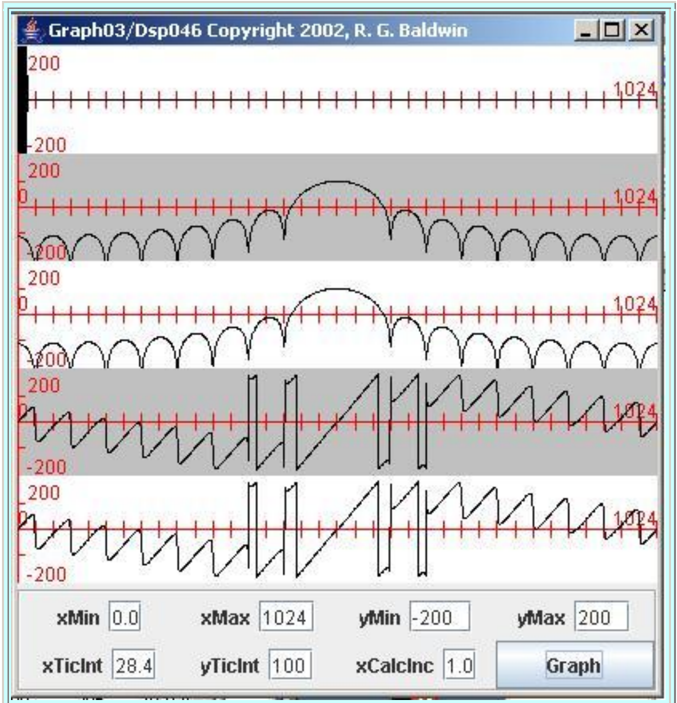
xMin 0.0 xMax 50 yMin -500 yMax 500
xTicInt 1 yTicInt 100 xCalcInc 1.0

Figure 28

As you can see from Figure 27, there are 17 coefficients in the feed-forward loop. The horizontal scale in Figure 28 was expanded to show the details of the filter output. If you count the number of non-zero values in the output shown in the top graph of Figure 28, you will see that it matches the number of non-zero values in Figure 27. If you estimate the amplitudes of the non-zero values in Figure 28, you will see that they match the values of the feed-forward coefficients in Figure 27.

The amplitude and phase response

Figure 29 shows the same data as Figure 28 except that the horizontal and vertical scale factors were adjusted to emphasize the amplitude and phase response of the filter.



xMin 0.0 xMax 1024 yMin -200 yMax 200
xTicInt 28.4 yTicInt 100 xCalcInc 1.0

Figure 29

If you count the number of notches in the amplitude response in Figure 29, you will see that the number matches the number of zeros shown in the z-plane in Figure 25. In addition, the locations of the notches in Figure 29 match the locations of the zeros near the unit circle in the z-plane shown in Figure 25.

Add a pole

Now I will show you what happens to the impulse response when a single pair of complex conjugate poles is moved away from the origin, causing some of the output to be fed back into the input. To do this, I will leave the zeros in the same locations as in Figure 25 and revert back to the same horizontal plotting scale factor that I used in Figure 28.

I moved one pair of complex conjugate poles from the origin to a location near the unit circle at about twenty degrees off the real axis as shown in Figure 30.

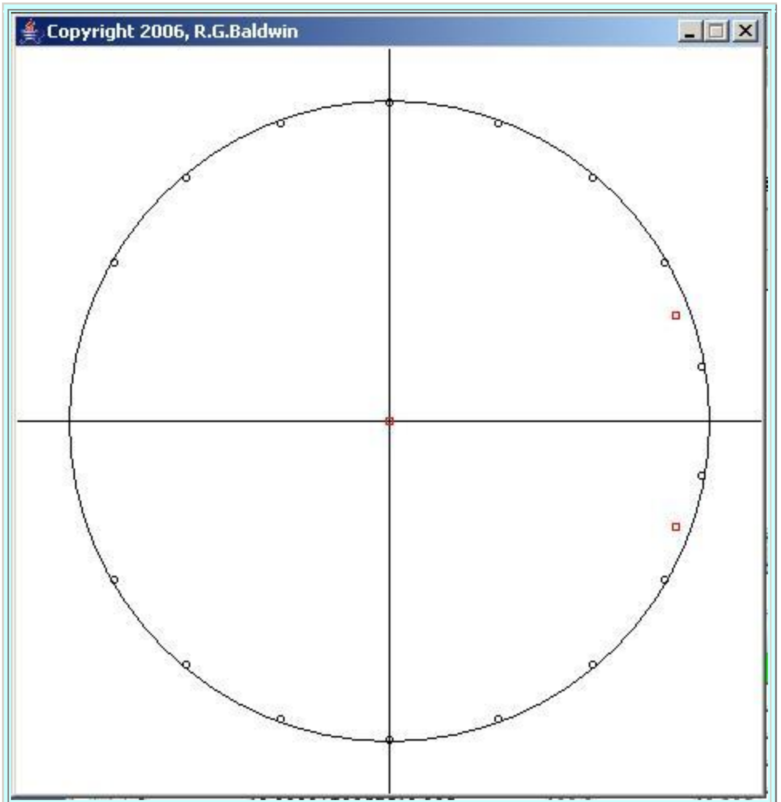
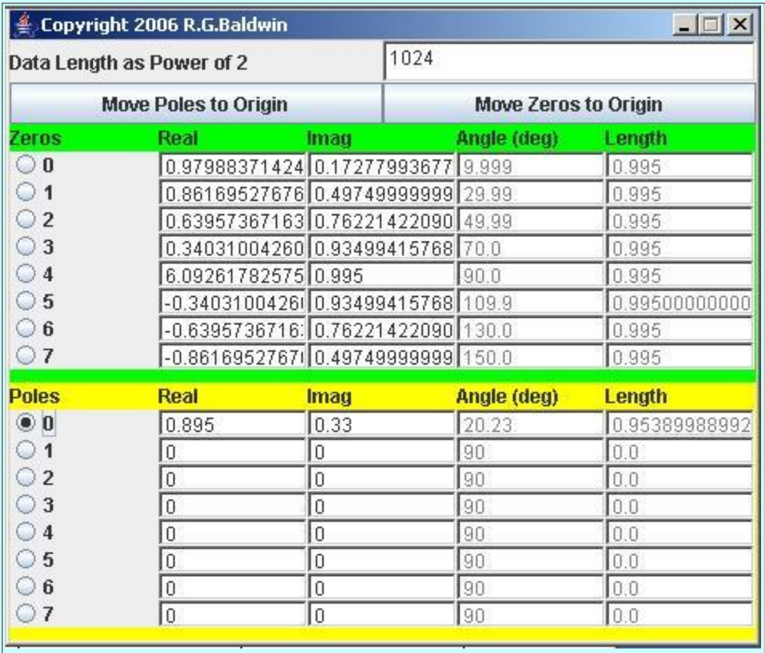


Figure 30

The numeric location of the pole

The actual location of the pole in numeric terms is shown for the first pole in the bottom chart in Figure 31.



Data Length as Power of 2: 1024

Zeros	Real	Imag	Angle (deg)	Length
○ 0	0.97988371424	0.17277993677	9.999	0.995
○ 1	0.86169527676	0.49749999999	29.99	0.995
○ 2	0.63957367163	0.76221422090	49.99	0.995
○ 3	0.34031004260	0.93499415768	70.0	0.995
○ 4	6.09261782575	0.995	90.0	0.995
○ 5	-0.3403100426	0.93499415768	109.9	0.99500000000
○ 6	-0.6395736716	0.76221422090	130.0	0.995
○ 7	-0.8616952767	0.49749999999	150.0	0.995

Poles	Real	Imag	Angle (deg)	Length
● 0	0.895	0.33	20.23	0.95389988992
○ 1	0	0	90	0.0
○ 2	0	0	90	0.0
○ 3	0	0	90	0.0
○ 4	0	0	90	0.0
○ 5	0	0	90	0.0
○ 6	0	0	90	0.0
○ 7	0	0	90	0.0

Figure 31

The feedback coefficients

Because the locations of the zeros were not modified, the values of the feed-forward coefficients did not change. However, there are now two non-zero feedback coefficients as shown in Figure 32.

```
Feedback (Denominator) Coefficients
1 -1.790
2  0.910
```
Figure 32

The new impulse response

One of the results of moving this pole away from the origin in the z-plane is shown by the top graph in Figure 33.

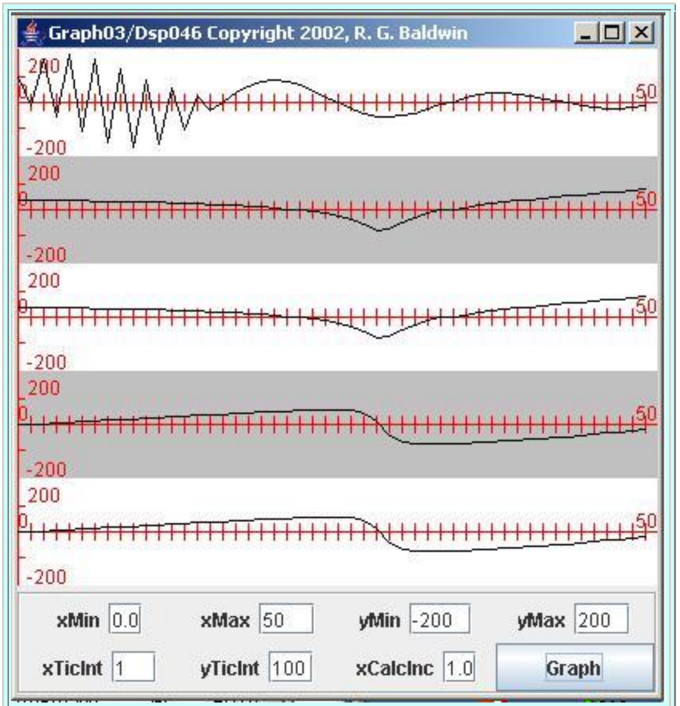


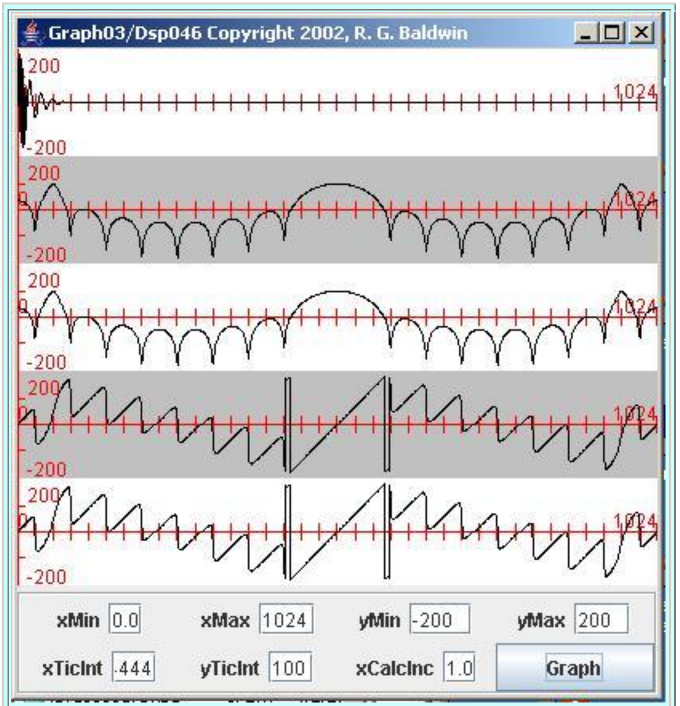
Figure 33

As you can see, the filter output no longer goes to zero after seventeen samples. Rather, the value that is fed back into the input as a result of the non-zero feedback coefficients continues to circulate in the filter causing the output to resemble a damped low-frequency sinusoid.

> *(Although I didn't show it here, by compressing the horizontal scale and making the vertical scale more sensitive for the first graph shown in Figure 33, it can be shown that at least 750 output samples are required to cause the impulse response for this filter to go to near zero.)*

The amplitude and phase response

Figure 34 shows amplitude and phase response graphs for this filter that can be compared with those shown in Figure 29.



xMin 0.0 xMax 1024 yMin -200 yMax 200

xTicInt 444 yTicInt 100 xCalcInc 1.0

Figure 34

As you can see by comparing the amplitude response graphs, moving the pair of complex conjugate poles from the origin to near the unit circle produces a peak at a low frequency in the amplitude response that isn't there in Figure 29.

A pole very close to the unit circle

I told you earlier that the recursive filter is unstable if any of the poles are outside the unit circle. What happens if a pole is on the unit circle? The answer is that the recursive filter turns into an oscillator. I will illustrate this with several graphs.

Let's begin with the pole-zero configuration shown in Figure 35.

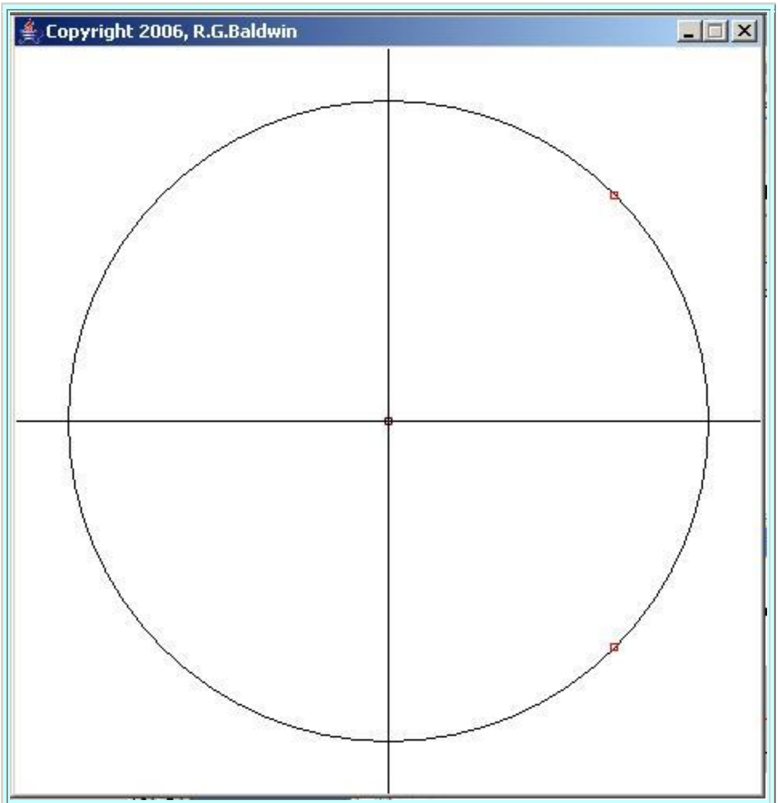


Figure 35

As you can see, all of the zeros in Figure 35 are located at the origin. A single pair of complex conjugate poles is placed very near the unit circle at an angle of 45 degrees relative to the real axis. The distance from the pole to the origin is 0.9998489885977782, so you can see that the pole is very close to the unit circle.

(The real and imaginary coordinate values for this case were 0.707.)

The beginning of the impulse response

The top graph in Figure 36 shows the first 100 samples of the impulse response of the filter. This will give you some idea of the general waveform of the impulse response.

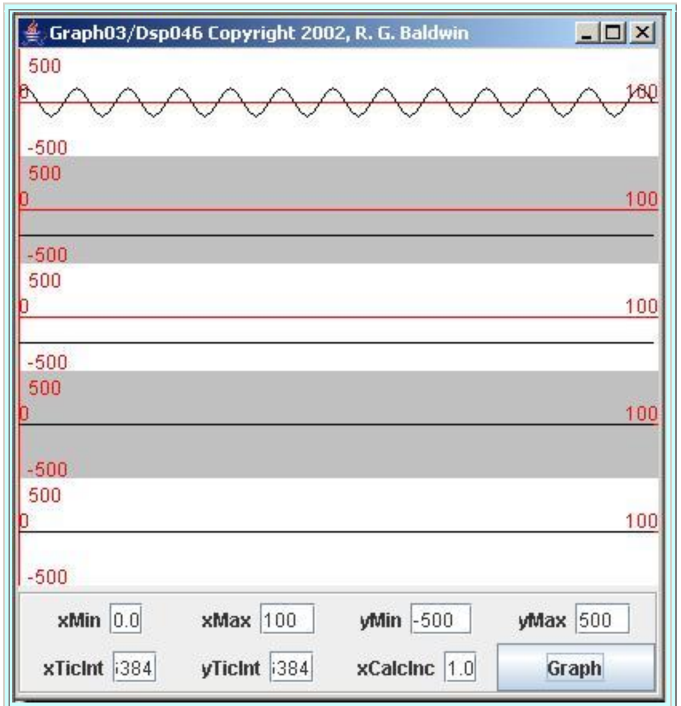


Figure 36

As you can see, the top graph is Figure 36 looks a lot like a sinusoid.

The complete impulse response

The top graph in Figure 37 shows the first 16384 samples of the impulse response.

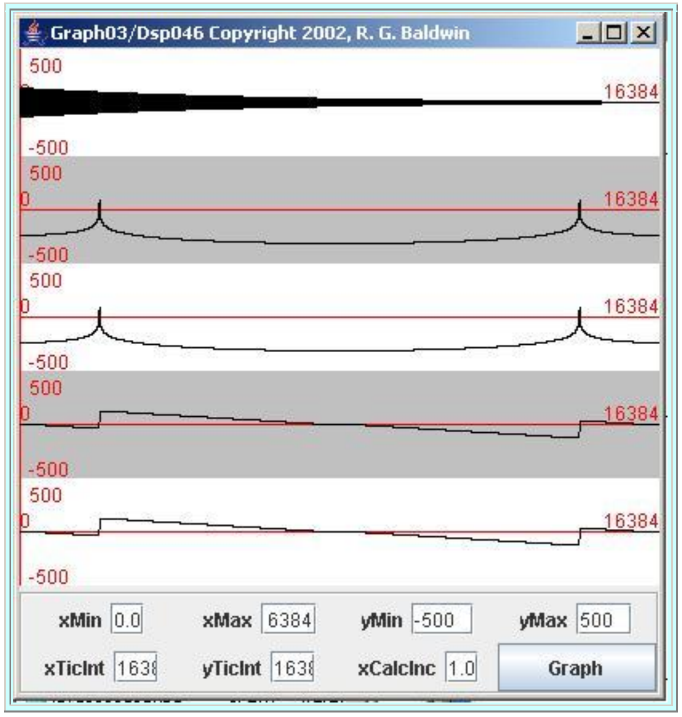


Figure 37

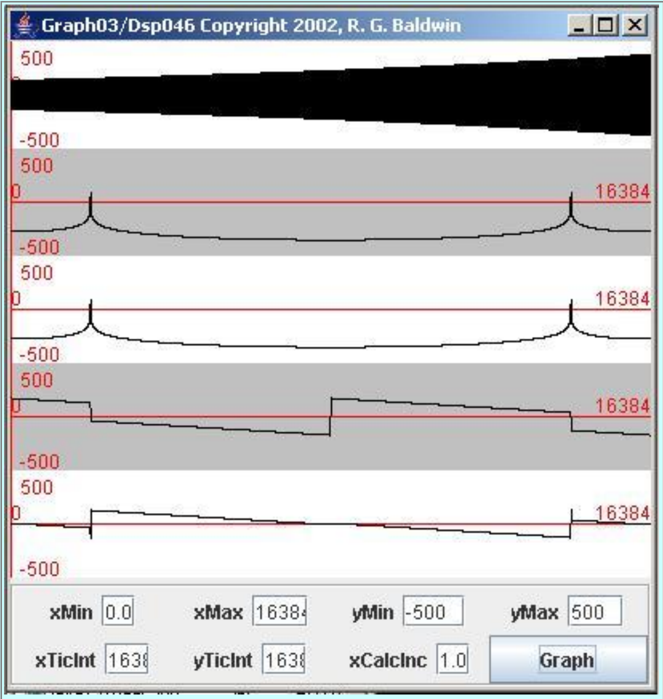
This graph shows that although it takes a long time to do so, the impulse response does finally damp down to zero, at least that appears to be the case for the vertical scale used for the graph.

Move the pole outside the unit circle

Now let's move the pole to a location that is only slightly outside the unit circle. We will leave it at an angle of 45 degrees relative to the real axis and cause its distance from the origin to be 1.0000611206321341.

(The real and imaginary coordinate values for this case were 0.70715.)

The top graph in Figure 38 shows the first 16384 samples of the impulse response.



xMin 0.0 xMax 1638 yMin -500 yMax 500
xTicInt 1638 yTicInt 1638 xCalcInc 1.0

Figure 38

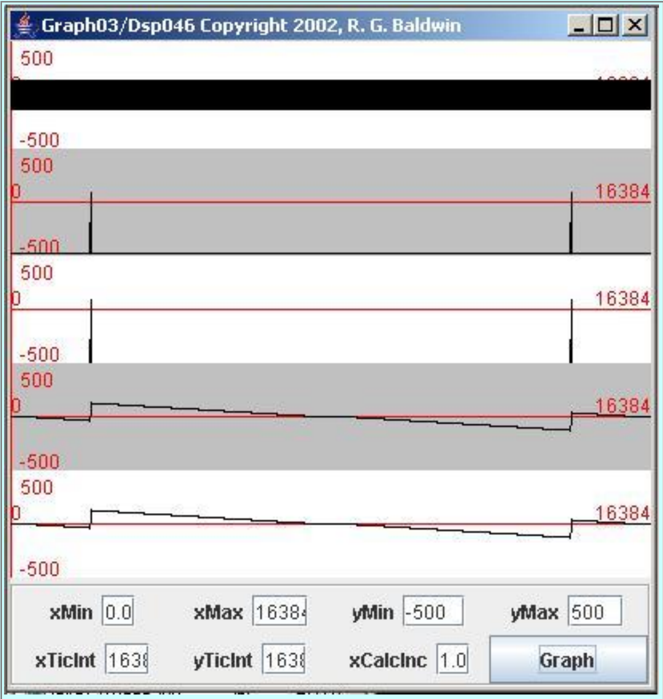
An unstable recursive filter

As you can see, this impulse response continues to grow over time. Eventually the values in the impulse response will grow so large the double precision numeric format in my computer won't be able to handle them and the output will become unpredictable. This is what I mean when I refer to an unstable recursive filter.

Put the pole on the unit circle

Now I am going to set the real and imaginary coordinate values for the pole to 0.70710678118. This will keep the pole at an angle of 45 degrees relative to the real axis and cause the distance from the origin to the pole to be 0.9999999999907404 *(displayed as a value of type* ***double****)*. This pole is as close to being on the unit circle as I am able to position it.

The top graph in Figure 39 shows the first 16384 samples of the impulse response.



xMin 0.0 xMax 1638 yMin -500 yMax 500
xTicInt 1638 yTicInt 1638 xCalcInc 1.0

Figure 39

Is the filter output growing or shrinking?

I can't say with absolute certainty that the amplitude of the filter output is neither growing nor shrinking. However, if the amplitude is changing over time, the magnitude of the change is very small.

A single frequency oscillator

This is what I meant when I said that placing a pole on the unit circle will cause the recursive filter to become an oscillator. Ideally, the output for this filter will continue producing the sinusoidal waveform shown in Figure 36 indefinitely without growing or shrinking.

> *(If you wanted an oscillator with a more complex waveform, you could put more than one pole on the unit circle.)*

The amplitude response

The amplitude response for this filter as shown in Figure 39 is also quite interesting. As you can see, it consists of a very narrow spike. No matter how much I increase the vertical sensitivity of the plot, I can't see any grass growing around the bottom of the spike.

> *(However, this probably means that the estimate of the amplitude response is going through zero at the computation points, and if I were to compute at half the frequency interval without changing the length of the sample, I would probably see some grass.)*

In any event, this strongly suggests that if this filter receives any input at all, it will *"ring"* at a constant frequency for a very long time, which is a characteristic of a very narrow band filter. If it rings without growing or shrinking forever, it is an oscillator.

A real-world analogy

The previous three examples are analogous to a child playing on a swing at the playground. As long as the child continues pumping energy into the process, the magnitude of the excursions of the swing will continue to grow. This is analogous to having a pole outside the unit circle.

> *(Of course there are physical factors at work on the playground that will limit the size of the excursions at some point, no matter how much energy the child attempts to pump into the process.)*

As every child knows, if they stop pumping, the excursions will eventually damp down to zero, although that may take quite a while. This is analogous to having a pole inside the unit circle and hitting the filter with an impulse.

If the child pumps just the right amount of energy into the process during each cycle, the excursions will become constant and neither grow nor shrink. This is analogous to having a pole exactly on the unit circle.

The pendulum of a clock

This is also analogous to the pendulum in a clock where the spring mechanism pumps a little bit of energy into the process during each cycle of the pendulum, keeping it swinging at a constant rate until the energy in the spring is dissipated. Conscientious clock owners rarely let that happen so that the pendulums in many clocks oscillate at the same rate for years on end.

A band-pass filter

Now I will demonstrate how to use the workbench to design a band-pass filter.

> *(Again, however, I want to emphasize that the workbench is intended primarily as a learning tool and not as a design tool. The purpose of the workbench is to help you develop an understanding of recursive filtering so that you will understand what you are doing when you use other design techniques.)*

Will develop in two stages

I will develop this filter in two stages. In the first stage, I will position a set of sixteen poles in the pass band to illustrate how the poles contribute to the behavior of the band-pass filter. In the second stage, I will add sixteen zeros to show how zeros can be used to improve the out-of-band reject characteristics of the filter as well as to improve the flatness in the pass band.

The filter specifications

Assume that the specifications call for a filter that will pass energy at all frequencies between 18.3-percent and 26-percent of the folding frequency and reject energy at all other frequencies. When viewed in the z-plane, this represents the angular region of the unit circle extending from 33 degrees to 47 degrees. Therefore, we know at the outset that we will primarily be dealing with that angular range of the unit circle.

The poles and the zeros

Given the above specification, when I started to develop this filter, I knew generally where the poles and zeros should be to produce the desired behavior. However, I didn't know exactly where they should be or how they should be arranged.

An interactive graphic design process

I developed the initial pole-zero configuration for the filter by clicking in the z-plane to locate the poles and zeros and then observing the results of that placement in the frequency domain. Once I had determined approximately where the poles and zeros should be using the graphic approach, I resorted to the use of a calculator to polish those locations producing the kind of symmetry that I wanted.

> *(As mentioned earlier, the ability to design a filter by clicking in the z-plane could be improved by enlarging the image of the z-plane. This would make it possible to use the mouse to locate the poles and zeros more accurately.)*

The locations of the poles

Figure 40 shows the pole configuration that I came up with to satisfy this need.

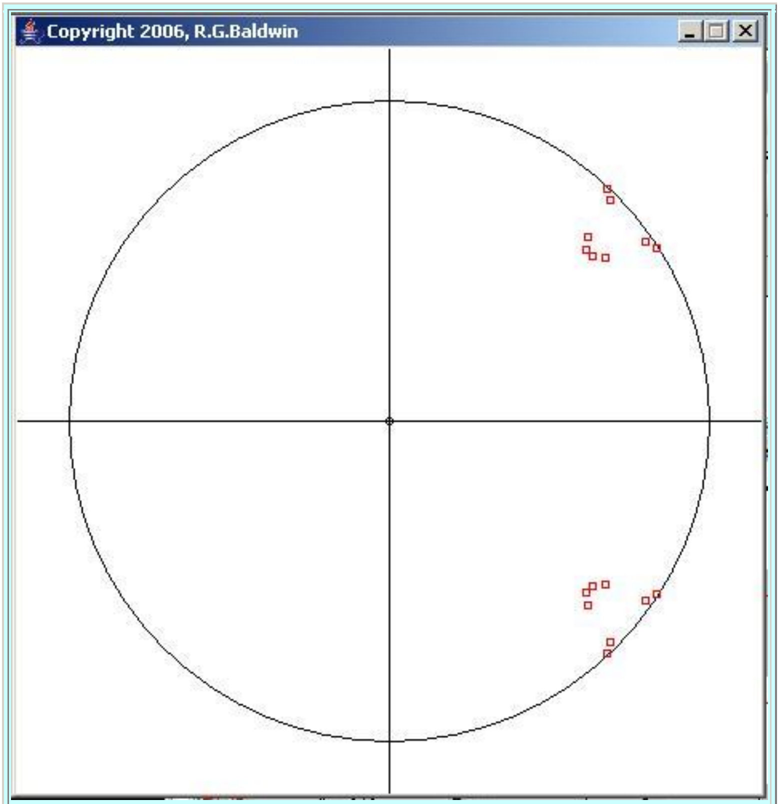


Figure 40

The design rationale for poles

As mentioned above, this was an iterative click and evaluate process, but it definitely wasn't a process of randomly positioning the poles until something good was produced. Generally the rationale for this pole configuration included two important considerations:

1. Each edge of the pass band should be well defined. Therefore, I placed one pole at each edge of the pass band close to the unit circle *(four poles total when you count those in the bottom and the top halves of the unit circle.)*.
2. As you traverse the pass band along the unit circle, the product of the distance to all of the poles should remain relatively constant. This caused me to come up with the pole configuration shown in Figure 40.

The amplitude response for poles only

With all the zeros located at the origin, this pole configuration resulted in the amplitude response shown in Figure 41.

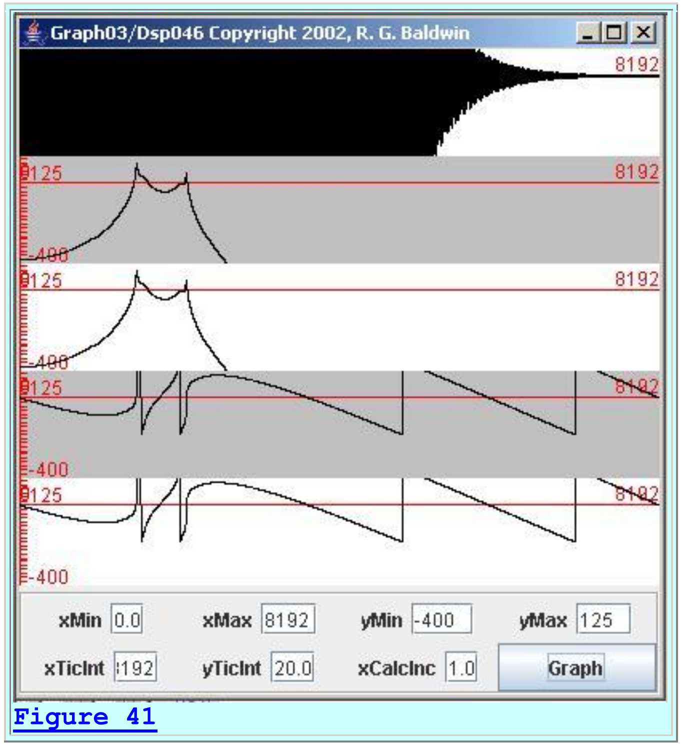


Figure 41

Now add the zeros

Figure 41 meets the two criteria listed above reasonably well, but the horns at the edges of the pass band are a little high. Given this contribution from the poles, I then added the zeros shown in Figure 42.

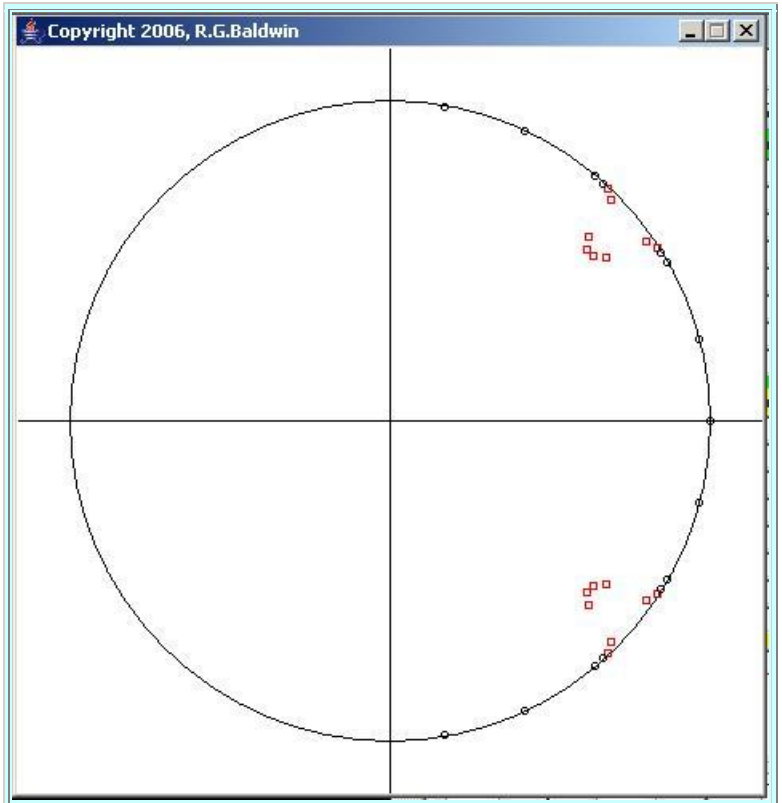


Figure 42

Zeros near the edge of the pass band

As you can see, I added two zeros on the unit circle very close to each of the four poles that define the edges of the pass band. These eight zeros are close to, but outside the pass band. There were two purposes for adding them:

1. To pull down the horns at the edges of the pass band in Figure 41.
2. To cause the sides of the pass band to be more vertical than is the case in Figure 41.

The remaining eight zeros

Beyond that, I added eight zeros at 15-degree intervals on each side of the pass band. This was done to hold the response down in regions far removed from the pass band.

Was this a successful design?

You can judge for yourself how successful this was. The amplitude response of the resulting recursive filter is shown in the second and third graphs of Figure 43.

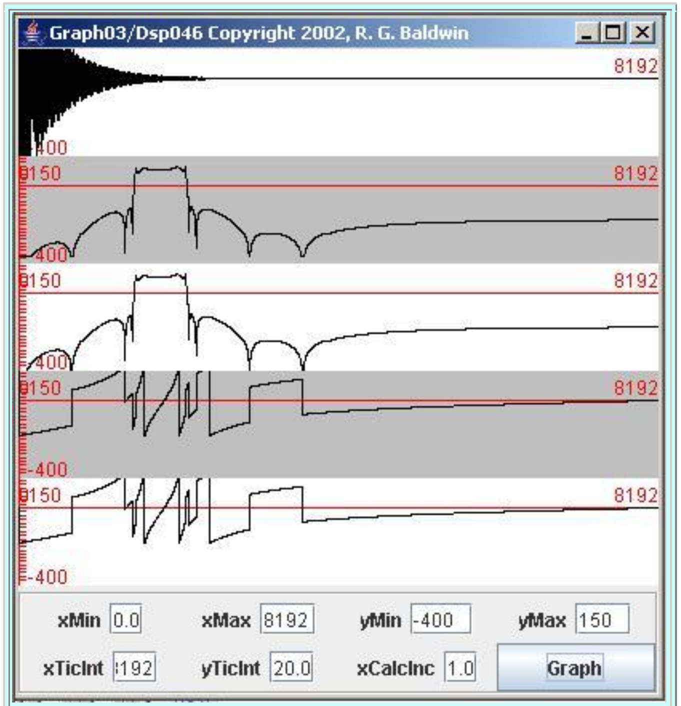


Figure 43

The amplitude response

The pass band is reasonably flat. By working with the plotting parameters in Figure 43, I determined that the pass band is flat to within plus or minus about 1.8 decibels.

As you can see in Figure 43, the pass band has nice vertical sides.

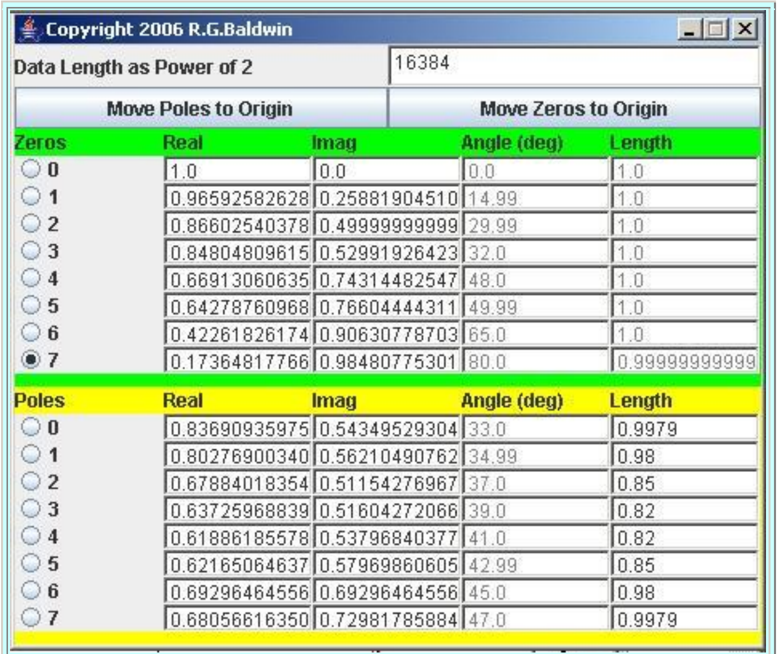
Again, by working with the plotting parameters, I determined that the first lobe outside the pass band is down by about 24 decibels. Other than the first lobe, the maximum response outside the pass band is down by about 36 decibels.

The impulse response

As you can see from the first graph in Figure 43, the impulse response for this filter has a very long tail. Approximately 2000 output samples are required for the impulse response to damp down to near zero. This suggests that a non-recursive convolution filter capable of providing this same band-pass filter would be quite long, requiring thousands of MADs per output sample. On the other hand, this recursive filter requires only 33 MADs per output sample.

The numeric coordinates

The numeric coordinates of the poles and zeros shown in Figure 42 are listed in Figure 44.



Data Length as Power of 2: 16384

Zeros	Real	Imag	Angle (deg)	Length
○ 0	1.0	0.0	0.0	1.0
○ 1	0.96592582628	0.25881904510	14.99	1.0
○ 2	0.86602540378	0.49999999999	29.99	1.0
○ 3	0.84804809615	0.52991926423	32.0	1.0
○ 4	0.66913060635	0.74314482547	48.0	1.0
○ 5	0.64278760968	0.76604444311	49.99	1.0
○ 6	0.42261826174	0.90630778703	65.0	1.0
● 7	0.17364817766	0.98480775301	80.0	0.99999999999

Poles	Real	Imag	Angle (deg)	Length
○ 0	0.83690935975	0.54349529304	33.0	0.9979
○ 1	0.80276900340	0.56210490762	34.99	0.98
○ 2	0.67884018354	0.51154276967	37.0	0.85
○ 3	0.63725968839	0.51604272066	39.0	0.82
○ 4	0.61886185578	0.53796840377	41.0	0.82
○ 5	0.62165064637	0.57969860605	42.99	0.85
○ 6	0.69296464556	0.69296464556	45.0	0.98
○ 7	0.68056616350	0.72981785884	47.0	0.9979

Figure 44

The filter coefficients

The recursive filter coefficients are shown in Figure 45.

```
Feed-Forward (Numerator) Coefficients
 0      1.000
 1    -11.176
 2     61.512
 3   -221.102
 4    580.629
 5  -1180.833
 6   1923.497
 7  -2559.773
 8   2812.494
 9  -2559.773
10   1923.497
11  -1180.833
12    580.629
13   -221.102
14     61.512
15    -11.176
16      1.000

Feedback (Denominator) Coefficients
 1    -11.140
 2     60.899
 3   -215.617
 4    551.413
 5  -1077.473
 6   1661.374
 7  -2059.823
 8  2074.043
 9 -1701.302
10  1133.309
11  -606.977
12   256.488
13   -82.798
14    19.303
15    -2.914
16     0.216
```
Figure 45

Run the Program

I encourage you to copy the code from Listing 1 into your text editor, compile it, and execute it. Experiment with it, making changes, and observing the results of your changes.

Remember that in addition to the code from Listing 1, this workbench program requires access to the following source code or class files, which were published in earlier tutorials *(see the References section of this document)*:

- GraphIntfc01
- ForwardRealToComplexFFT01
- Graph03

  *(You can also find the source code for these classes by searching for the class names along with the keywords **baldwin java** on Google.)*

Having compiled all the source code, enter the following command at the command prompt to run this program:

```
java Graph03 Dsp046
```

Summary

This document, which is the first part of a multi-part lesson on recursive filtering, has provided an overview of an interactive *Recursive Filtering Workbench* that can be used to design, experiment with, and evaluate the behavior of digital recursive filters.

Numerous practical examples of recursive filtering were presented and explained. Hopefully the study of this lesson has helped you to gain a better understanding of the behavior of digital recursive filters.

What's Next?

An understanding of this workbench program requires an understanding of the following Java classes, which are new to this program *(plus some anonymous classes not listed here)*:

1. **Dsp046** - Driver class for the workbench.
2. **InputGUI** - Provides user input capability *(mainly text)* for the workbench as shown in Figure 9.
3. **InputGUI$MyTextListener -** Processes text input to the workbench, updating the angle output and updating the z-plane display.
4. **InputGUI$ZPlane** - Provides the z-plane display along with graphic mouse input capability for the workbench as shown in Figure 8.

In addition, an understanding of this program requires an understanding of the following Java classes, which were presented and explained in earlier lessons:

- **ForwardRealToComplexFFT01** - Used to perform an FFT on the impulse response.
- **Graph03** - Plotting program as shown in Figure 10.
- **GraphIntfc01** - Required to use **Graph03** for plotting.
- **GUI** - Required to use **Graph03** for plotting.
- **GUI$MyCanvas** - Required to use **Graph03** for plotting.

This installment of this multi-part lesson has provided an overview of the workbench.

Although it is possible that I may change the schedule as I write and publish the future installments of this lesson, here are my plans at this point in time.

The second installment will present and explain the class named **Dsp046**, and will also explain its relationship to the classes in the second list above.

The third installment will present and explain the class named **InputGUI** along with the inner class named **InputGUI$MyTextListener**.

The fourth installment will present and explain the class named **InputGUI$MyTextListener**.

References

An understanding of the material in the following previously published lessons will be very helpful to you in understanding the material in this lesson:

- 1468 Plotting Engineering and Scientific Data using Java
- 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

- 1486 Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm
- 1487 Convolution and Frequency Filtering in Java
- 1488 Convolution and Matched Filtering in Java
- 1492 Plotting Large Quantities of Data using Java
- Other previously-published lessons on DSP including adaptive processing and image processing

In addition, there are numerous good references on DSP available on the web. For example, good references can be found at the following URLs:

- http://ccrma.stanford.edu/~jos/filters/filters.html
- http://www.dspguide.com/pdfbook.htm
- Wikipedia

Complete Program Listing

A complete listing of the program discussed in this lesson is shown in Listing 1 below.

```
/* File Dsp046.java
Copyright 2006, R.G.Baldwin

This program provides a visual, interactive recursive
filtering workbench.

The purpose of the program is to make it easy to experiment
with the behavior of recursive filters and to visualize the
results of those experiments.

The program implements a recursive filter having eight
pairs of complex conjugate poles and eight pairs of complex
conjugate zeros.  The locations of the pairs of poles and
zeros in the z-plane are controlled by the user.

Although the pairs of poles and zeros can be co-located on
the real axis, the program does not support the placement
of individual poles and zeros on the real axis.

The user can reduce the number of poles and zeros used by
the recursive filter by moving excess poles and zeros to
the origin in the z-plane, rendering them ineffective in
the behavior of the recursive filter.

The program provides three interactive displays on the
screen.  The first display (in the leftmost position on the
screen) contains five graphs.  The first graph in this
display shows the impulse response of the recursive filter
in the time domain.

The second and third graphs in the first display show the
amplitude response of the recursive filter in the frequency
domain computed using two different approaches.  The two
```

different computational approaches are provided for comparison purposes.

The first computational approach for computing the amplitude response is to perform a Fourier transform on the impulse response using an FFT algorithm. The quality of the estimate of the amplitude response using this approach is dependent on the extent to which the entire impulse response is captured in the set of samples used to perform the FFT. If the impulse response is truncated, the estimate will be degraded.

The second approach for computing the amplitude response involves computing the product of the vector lengths from each point on the unit circle to each of the poles and each of the zeros in the z-plane. This approach provides an idealized estimate of the amplitude response of the recursive filter, unaffected by impulse-response considerations. This approach provides the same results that should be produced by performing the FFT on a set of impulse-response samples of sufficient length to guarantee that the values in the impulse response have been damped down to zero (the impulse response is totally captured in the set of samples on which the FFT is performed).

The fourth and fifth graphs in the first display show the phase response of the recursive filter computed using the same (or similar) approaches described above for the amplitude response. (The second approach uses the sum of vector angles instead of the product of vector lengths.) Once again, the two approaches are provided for comparison purposes.

By default, the program computes and captures the impulse response for a length of 1024 samples and performs the Fourier transform on that length. However, the length of the captured impulse response and the corresponding FFT can be changed by the user to any length between 2 samples and 16384 samples, provided that the length is an even power of two. (If the length specified by the user is not an even power of two, it is automatically changed to an even power of two by the program.)

The first display is interactive in the sense that there are seven different plotting parameters that can be adjusted by the user in order to produce plots that are visually useful in terms of the vertical scale, the horizontal scale, the location of tic marks, etc. The user can modify any of the parameters and then click a Graph button to have the graphs re-plotted using the new parameters.

The second display (which appears in the upper center of the screen) shows the locations of all of the poles and zeros in the z-plane. The user can use the mouse to change the location of any pair of complex conjgate poles or zeros by first selecting a specific pair of poles or zeros and then clicking the new location in the z-plane. This interactive capability makes it possible for the user to modify the design of the recursive filter in a completely graphic manner by positioning the poles and zeros in the z-plane with the mouse.

Having relocated one or more pairs of poles or zeros in the z-plane, the user can then click the Graph button in the first display described earlier to cause the new impulse response, the new amplitude response, and the new phase response of the new recursive filter with the modified pole and zero locations to be computed and displayed.

The third display (that appears in the upper-right of the screen) is a control panel that uses text fields, ordinary buttons, and radio buttons to allow the user to perform the following tasks.

1. Specify a new length for the impulse response as an even power of two. (Once again, if the user fails to specify an even power of two, the value provided by the user is converted to an even power of two by the program.)

2. Cause all of the poles to be moved to the origin in the z-plane.

3. Cause all of the zeros to be moved to the origin in the z-plane.

4. Select a particular pair of complex conjugate poles or zeros to be relocated using the mouse in the display of the z-plane.

5. View the angle described by each pole and zero relative to the origin and the horizontal axis in the z-plane.

6. View the length of an imaginary vector that connects each pole and zero to the origin.

7. Enter (into a text field) a new real or imaginary value specifying the location of a pair of complex conjugate poles or zeros in the z-plane.

When a new real or imaginary value is entered, the angle and the length are automatically updated to reflect the new location of the pole or zero and the display of the pole or zero in the z-plane is also updated to show the new location.

Conversely, when the mouse is used to relocate a pole or zero in the z-plane display, the corresponding real and imaginary values in the text fields and the corresponding angle and length are automatically updated to reflect the new location for the pole or zero.

```
Note that mainly for convenience (but for technical reasons
as well), the angle and length in the text fields and the
location of the pole or zero in the Z-plane display are
updated as the user enters each character into the text
field.  If the user enters text into the text field that
cannot be converted into a numeric value of type double,
(such as the pair of characters "-." for example) the
contents of the text field are automatically converted to a
single "0" character.  A warning message is displayed on
the command-line screen when this happens.  There is no
long-term harm when this occurs, but the user may need to
start over to enter the new value.  Thus, the user should
exercise some care regarding the order in which the
characters in the text field are modified when entering new
real and imaginary values.

Each time the impulse response and the spectral data are
plotted, the seventeen feed-forward filter coefficients and
the sixteen feedback coefficients used by the recursive
filter to produce the output being plotted are displayed on
the command-line screen.

Usage: This program requires access to the following source
code or class files, which were published in earlier
tutorials:

GraphIntfc01
ForwardRealToComplexFFT01
Graph03

Enter the following command at the command prompt to run
the program:

java Graph03 Dsp046.

Tested using J2SE 5.0 under WinXP.  J2SE 5.0 or later is
required due to the use of static imports and printf.
**********************************************************/
import static java.lang.Math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Dsp046 implements GraphIntfc01{

  //The value stored in the following variable specifies
  // the number of samples of the impulse response that are
  // captured.  The impulse response serves as the input to
  // an FFT for the purpose of estimating the amplitude and
  // phase response of the recursive filter.  This data
  // length must be a power of 2 for the FFT program to
  // work correctly.  If the user enters a value for the
  // data length that is not a power of two, the value is
  // automatically converted to a power of two by the
  // program.
  int dataLength;
  double[] impulseResponse;
  double[] fourierAmplitudeRespnse;
  double[] fourierPhaseAngle;
  double[] vectorAmplitudeResponse;
  double[] vectorPhaseAngle;
  //This variable contains a reference to a user input GUI
  // containing buttons, radio buttons, and text fields.
  InputGUI inputGUI = null;
  //-----------------------------------------------------//

  Dsp046(){//constructor
    //If the InputGUI object doesn't already exist,
    // create it.  However, if it already exists, retrieve
    // the reference to the object from a static variable
    // belonging to the InputGUI class.  This is
    // necessary because a new object of the Dsp046 class
    // is instantiated each time the user clicks the Graph
    // button on the main (Graph03) GUI.  However, the
    // InputGUI object needs to persist across many
    // clicks of that button because it stores the state of
    // the poles and zeros designed by the user. When the
    // InputGUI object is first created, its pole and
    // zero text fields are initialized with a set of
    // default pole and zero data values.  Its data length
    // text field is initialized to 1024 samples.
    if(InputGUI.refToObj == null){
      //Instantiate a new InputGUI object.
      inputGUI = new InputGUI();

      //Initialize the length of the array objects that
      // will contain pole and zero data to a value that is
      // maintained in the InputGUI object.
      double[] defaultPoleReal =
                        new double[inputGUI.numberPoles/2];
      double[] defaultPoleImag =
                        new double[inputGUI.numberPoles/2];
      double[] defaultZeroReal =
                        new double[inputGUI.numberZeros/2];
      double[] defaultZeroImag =
                        new double[inputGUI.numberZeros/2];

      //Create the default data for eight pairs of complex
      // conjugate poles spaced at 20-degree intervals
      // around the unit circle.  These are the locations
      // of the poles in the complex z-plane.  The poles
      // are barely (0.995) inside the unit circle.
      for(int cnt = 0;cnt < inputGUI.numberPoles/2;cnt++){
        defaultPoleReal[cnt] =
                           0.995*cos((20+20*cnt)*PI/180.0);
        defaultPoleImag[cnt] =
                           0.995*sin((20+20*cnt)*PI/180.0);
      }//end for loop

      //Create the default data for eight pairs of complex
      // conjugate zeros spaced at 20-degree intervals
      // around the unit circle.  These are the locations
      // of the zeros in the complex z-plane.  The zero
      // positions are half way between the pole positions.
      for(int cnt = 0;cnt < inputGUI.numberZeros/2;cnt++){
        defaultZeroReal[cnt] =
                            0.995*cos((10+20*cnt)*PI/180.0);
        defaultZeroImag[cnt] =
                            0.995*sin((10+20*cnt)*PI/180.0);
      }//end for loop

      //At various points in the program, you may notice
      // that I have performed separate iterations on
      // poles and zeros even though the number of poles
      // is the same as the number of zeros, and I could
      // have combined them.  I did this to make it
      // possible to modify the number of poles or the
      // number of zeros later without the requirement for
      // a major overhaul of the program source code.

      //Initialize the real and imaginary text fields in
      // the InputGUI object with the default real and
      // imaginary pole and zero values.
      //Start by setting the pole values.
      for(int cnt = 0;cnt < inputGUI.numberPoles/2;cnt++){
        inputGUI.poleReal[cnt].setText(
                     String.valueOf(defaultPoleReal[cnt]));
        inputGUI.poleImag[cnt].setText(
                     String.valueOf(defaultPoleImag[cnt]));
      }//end for loop

      //Now set the zero values.
      for(int cnt = 0;cnt < inputGUI.numberZeros/2;cnt++){
        inputGUI.zeroReal[cnt].
             setText(String.valueOf(defaultZeroReal[cnt]));
        inputGUI.zeroImag[cnt].setText(
                     String.valueOf(defaultZeroImag[cnt]));
      }//end for loop

      //Get the default data length from the new object.
      // There is no requirement to convert it to a power
      // of two because a power of two is hard-coded into
      // the program as the default value.
      dataLength = Integer.parseInt(
                         inputGUI.dataLengthField.getText());

    }else{//An InputGUI object already exists.
      //Retrieve the reference to the existing object that
      // was saved earlier.
      inputGUI = InputGUI.refToObj;
      //Get the current data length from the object.  This
      // value may have been modified by the user and may
      // not be an even power of two.  Convert it to an
      // even power of two and store the converted value
      // back into the text field in the existing object.
      dataLength = Integer.parseInt(
                         inputGUI.dataLengthField.getText());
      dataLength = convertToPowerOfTwo(dataLength);
      inputGUI.dataLengthField.setText("" + dataLength);
    }//end if

    //Establish the length of some arrays based on the
    // current data length
    impulseResponse = new double[dataLength];
    fourierAmplitudeRespnse = new double[dataLength];
    fourierPhaseAngle = new double[dataLength];

    //Compute and save the impulse response of the filter.
    // The following code implements a recursive filtering
    // operation based on the poles and zeros previously
    // established.  However, the conversion from poles
    // and zeros to feedForward and feedback coefficients
    // are not routinely involved in the application of a
    // recursive filter to input data. Therefore, there is
    // more code here than would be needed for a routine
    // recursive filtering operation.

    //Invoke the captureTextFieldData method on the
    // InputGUI object to cause the current values in the
    // text fields describing the poles and zeros to be
    // converted into double values and stored in arrays.
    inputGUI.captureTextFieldData();

    //Create the feedback coefficient array based on the
    // values stored in the text fields of the InputGUI
    // object.  The process is to first multiply the roots
    // corresponding to each of eight pairs of complex
    // conjugate poles.  This produces eight second-order
    // polynomials.  These second-order polynomials are
    // multiplied in pairs to produce four fourth-order
    // polynomials.  These four fourth-order polynomials
    // are multipled in pairs to produce two eighth-order
    // polynomials.  The two eighth-order polynomials are
    // multiplied to produce one sixteenth-order
    // polynomial.
    //The algbraic sign of the real and imag values were
    // changed to make them match the format of a root
    // located at a+jb.  The format of the root is
    // (x-a-jb)
    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp1 = conjToSecondOrder(
      -inputGUI.poleRealData[0],-inputGUI.poleImagData[0]);
    double[] temp2 = conjToSecondOrder(
      -inputGUI.poleRealData[1],-inputGUI.poleImagData[1]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp3 = secondToFourthOrder(
                      temp1[1],temp1[2],temp2[1],temp2[2]);

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp4 = conjToSecondOrder(
      -inputGUI.poleRealData[2],-inputGUI.poleImagData[2]);
    double[] temp5 = conjToSecondOrder(
      -inputGUI.poleRealData[3],-inputGUI.poleImagData[3]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp6 = secondToFourthOrder(
                      temp4[1],temp4[2],temp5[1],temp5[2]);
    //Multiply a pair of fourth-order polynomials to
    // produce an eighth-order polynomial.
    double[] temp7 = fourthToEighthOrder(
                      temp3[1],temp3[2],temp3[3],temp3[4],
                      temp6[1],temp6[2],temp6[3],temp6[4]);

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp11 = conjToSecondOrder(
      -inputGUI.poleRealData[4],-inputGUI.poleImagData[4]);
    double[] temp12 = conjToSecondOrder(
      -inputGUI.poleRealData[5],-inputGUI.poleImagData[5]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp13 = secondToFourthOrder(
                  temp11[1],temp11[2],temp12[1],temp12[2]);

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp14 = conjToSecondOrder(
      -inputGUI.poleRealData[6],-inputGUI.poleImagData[6]);
    double[] temp15 = conjToSecondOrder(
      -inputGUI.poleRealData[7],-inputGUI.poleImagData[7]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp16 = secondToFourthOrder(
                  temp14[1],temp14[2],temp15[1],temp15[2]);
    //Multiply a pair of fourth-order polynomials to
    // produce an eighth-order polynomial.
    double[] temp17 = fourthToEighthOrder(
                  temp13[1],temp13[2],temp13[3],temp13[4],
                  temp16[1],temp16[2],temp16[3],temp16[4]);

    //Perform the final polynomial multiplation,
    // multiplying a pair of eighth-order polynomials to
    // produce a sixteenth-order polynomial.  Place the
    // coefficients of the sixteenth-order polynomial in
    // the feedback coefficient array.
    double[] feedbackCoefficientArray =
                eighthToSixteenthOrder(
                  temp7[1],temp7[2],temp7[3],temp7[4],
                  temp7[5],temp7[6],temp7[7],temp7[8],
                  temp17[1],temp17[2],temp17[3],temp17[4],
                  temp17[5],temp17[6],temp17[7],temp17[8]);

    //Determine the length of the delay line required to
    // perform the feedback arithmetic.
    int feedbackDelayLineLength =
                           feedbackCoefficientArray.length;
    //Create the feedForward coefficient array based on
    // the values stored in the text fields of the
    // InputGUI object.  The process is  the same as
    // described above for the poles.

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp21 = conjToSecondOrder(
      -inputGUI.zeroRealData[0],-inputGUI.zeroImagData[0]);
    double[] temp22 = conjToSecondOrder(
      -inputGUI.zeroRealData[1],-inputGUI.zeroImagData[1]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp23 = secondToFourthOrder(
                  temp21[1],temp21[2],temp22[1],temp22[2]);

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp24 = conjToSecondOrder(
      -inputGUI.zeroRealData[2],-inputGUI.zeroImagData[2]);
    double[] temp25 = conjToSecondOrder(
      -inputGUI.zeroRealData[3],-inputGUI.zeroImagData[3]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp26 = secondToFourthOrder(
                  temp24[1],temp24[2],temp25[1],temp25[2]);
    //Multiply a pair of fourth-order polynomials to
    // produce an eighth-order polynomial.
    double[] temp27 = fourthToEighthOrder(
                  temp23[1],temp23[2],temp23[3],temp23[4],
                  temp26[1],temp26[2],temp26[3],temp26[4]);

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp31 = conjToSecondOrder(
      -inputGUI.zeroRealData[4],-inputGUI.zeroImagData[4]);
    double[] temp32 = conjToSecondOrder(
      -inputGUI.zeroRealData[5],-inputGUI.zeroImagData[5]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp33 = secondToFourthOrder(
                  temp31[1],temp31[2],temp32[1],temp32[2]);

    //Multiply two pairs of complex conjugate roots to
    // produce two second-order polynomials.
    double[] temp34 = conjToSecondOrder(
      -inputGUI.zeroRealData[6],-inputGUI.zeroImagData[6]);
    double[] temp35 = conjToSecondOrder(
      -inputGUI.zeroRealData[7],-inputGUI.zeroImagData[7]);
    //Multiply a pair of second-order polynomials to
    // produce a fourth-order polynomial.
    double[] temp36 = secondToFourthOrder(
                  temp34[1],temp34[2],temp35[1],temp35[2]);
    //Multiply a pair of fourth-order polynomials to
    // produce an eighth-order polynomial.
    double[] temp37 = fourthToEighthOrder(
                  temp33[1],temp33[2],temp33[3],temp33[4],
                  temp36[1],temp36[2],temp36[3],temp36[4]);

    //Perform the final polynomial multiplation,
    // multiplying a pair of eighth-order polynomials to
    // produce a sixteenth-order polynomial.  Place the
    // coefficients of the sixteenth-order polynomial in
    // the feedForward coefficient array.
    double[] feedForwardCoefficientArray =
                eighthToSixteenthOrder(
                  temp27[1],temp27[2],temp27[3],temp27[4],
                  temp27[5],temp27[6],temp27[7],temp27[8],
                  temp37[1],temp37[2],temp37[3],temp37[4],
                  temp37[5],temp37[6],temp37[7],temp37[8]);

    //Determine the length of the delay line required to
    // perform the feedForward arithmetic.
    int feedForwardDelayLineLength =
                        feedForwardCoefficientArray.length;

    //Display the feedForward and feedback coefficients
    System.out.println(
                  "Feed-Forward (Numerator) Coefficients");
    for(int cnt = 0;
            cnt < feedForwardCoefficientArray.length;
            cnt++){
      System.out.printf ("%2d % 5.3f%n",cnt,
                         feedForwardCoefficientArray[cnt]);
    }//end for loop

    System.out.println(
                  "\nFeedback (Denominator) Coefficients");
    for(int cnt = 1;
            cnt < feedbackCoefficientArray.length;
            cnt++){
      System.out.printf ("%2d % 5.3f%n",cnt,
                            feedbackCoefficientArray[cnt]);
    }//end for loop
    System.out.println();

    //Create the data delay lines used for feedForward
    // and feedback arithmetic.
    double[] feedForwardDelayLine =
                    new double[feedForwardDelayLineLength];
    double[] feedbackDelayLine =
                       new double[feedbackDelayLineLength];

    //Initial input and output data values
    double filterInputSample = 0;//input data
    double filterOutputSample = 0;//output data

    //Compute the output values and populate the output
    // array for further analysis such as FFT analysis.
    // This is the code that actually applies the
    // recursive filter to the input data given the
    // feedForward and feedback coefficients.
    for(int dataLenCnt = 0;dataLenCnt < dataLength;
                                         dataLenCnt++){
      //Create the input samples consisting of a single
      // impulse at time zero and sample values of 0
      // thereafter.
      if(dataLenCnt == 0){
        filterInputSample = 100.0;
      }else{
        filterInputSample = 0.0;
      }//end else

      //**********************************************//
      //This is the beginning of one cycle of the actual
      // recursive filtering process.

      //Shift the data in the delay lines.  Oldest value
      // has the highest index value.
      for(int cnt = feedForwardDelayLineLength-1;
                                       cnt > 0;cnt--){
        feedForwardDelayLine[cnt] =
                             feedForwardDelayLine[cnt-1];
      }//end for loop

      for(int cnt = feedbackDelayLineLength-1;
                                       cnt > 0;cnt--){
        feedbackDelayLine[cnt] = feedbackDelayLine[cnt-1];
      }//end for loop

      //Insert the input signal into the delay line at
      // zero index.
      feedForwardDelayLine[0] = filterInputSample;

      //Compute sum of products for input signal and
      // feedForward coefficients from 0 to
      // feedForwardDelayLineLength-1.
      double xTemp = 0;
      for(int cnt = 0;cnt < feedForwardDelayLineLength;
                                               cnt++){
        xTemp += feedForwardCoefficientArray[cnt]*
                               feedForwardDelayLine[cnt];
      }//end for loop

      //Compute sum of products for previous output values
      // and feedback coefficients from 1 to
      // feedbackDelayLineLength-1.
      double yTemp = 0;
      for(int cnt = 1;cnt < feedbackDelayLineLength;cnt++){
        yTemp += feedbackCoefficientArray[cnt]*
                                  feedbackDelayLine[cnt];
      }//end for loop

      //Compute new output value as the difference.
      filterOutputSample = xTemp - yTemp;

      //Save the output value in the array containing the
      // impulse response.
      impulseResponse[dataLenCnt] = filterOutputSample;

      //Insert the output signal into the delay line at
      // zero index.
      feedbackDelayLine[0] = filterOutputSample;

      //This is the end of one cycle of the recursive
      // filtering process.
      //**********************************************//
    }//end for loop

    //Now compute the Fourier Transform of the impulse
    // response, placing the magnitude result from the FFT
    // program into the fourierAmplitudeRespnse array  and
    // the phase angle result in the fourierPhaseAngle
    // array, each to be plotted later.
    ForwardRealToComplexFFT01.transform(
                              impulseResponse,
                              new double[dataLength],
                              new double[dataLength],
                              fourierPhaseAngle,
                              fourierAmplitudeRespnse);

    /*DO NOT DELETE THIS CODE
    //Note that this normalization code is redundant
    // because of the normalization that takes place in
    // the method named convertToDB later.  However, if
    // the conversion to decibels is disabled, the
    // following code should be enabled.
    //Scale the fourierAmplitudeRespnse to compensate for
    // the differences in data length.
    for(int cnt = 0;cnt < fourierAmplitudeRespnse.length;
                                                  cnt++){
      fourierAmplitudeRespnse[cnt] =
           fourierAmplitudeRespnse[cnt]*dataLength/16384.0;
    }//end for loop
    */

    //Compute the amplitude response based on the ratio of
    // the products of the pole and zero vectors.
    vectorAmplitudeResponse = getVectorAmplitudeResponse(
                                inputGUI.poleRealData,
                                inputGUI.poleImagData,
                                inputGUI.zeroRealData,
                                inputGUI.zeroImagData);

    //Compute the phase angle based on sum and difference
    // of the angles of the pole and zero vectors.
    vectorPhaseAngle = getVectorPhaseAngle(
                                inputGUI.poleRealData,
                                inputGUI.poleImagData,
                                inputGUI.zeroRealData,
                                inputGUI.zeroImagData);

    //Convert the fourierAmplitudeRespnse data to decibels.
    // Disable the following statement to disable the
    // conversion to decibels.
    convertToDB(fourierAmplitudeRespnse);

    //Convert the vectorAmplitudeResponse to decibels.
    // Disable the following statement to disable the
    // conversion to decibels.
    convertToDB(vectorAmplitudeResponse);

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

  //The purpose of this method is to convert an incoming
  // array containing amplitude response data to decibels.
  void convertToDB(double[] magnitude){
    //Eliminate or modify all values that are incompatible
    // with conversion to log base 10 and the log10 method.
    //  Also limit small values to be no less than 0.0001.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      if((magnitude[cnt] == Double.NaN) ||
                             (magnitude[cnt] <= 0.0001)){
        magnitude[cnt] = 0.0001;
      }else if(magnitude[cnt] == Double.POSITIVE_INFINITY){
        magnitude[cnt] = 9999999999.0;
      }//end else if
    }//end for loop

    //Find the peak value for use in normalization.
    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 the peak value to make the values easier
    // to plot with regard to scaling.
    for(int cnt = 0;cnt < magnitude.length;cnt++){
      magnitude[cnt] = magnitude[cnt]/peak;
    }//end for loop

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

  }//end convertToDB
  //-------------------------------------------------------//

  //This method makes certain that the incoming value is a
  // non-zero positive power of two that is less than or
  // equal to 16384. If the input is not equal to either a
  // power of two or one less than a power of two, it is
  // truncated to the next lower power of two.  If it is
  // either a power of two or one less than a power of two,
  // the returned value is that power of two.  Negative
  // input values are converted to positive values before
  // making the conversion.
  int convertToPowerOfTwo(int dataLength){
    //Eliminate negative values
    dataLength = abs(dataLength);
    //Make sure the data length is not zero by adding a
    // value of 1.
    dataLength++;
    //Make sure the data length is not greater than 16384.
    if(dataLength > 16384){
      dataLength = 16384;
    }//end if

    int cnt = 0;
    int mask = 0x4000;
    //Loop and left shift left until the msb of the data
    // length value matches 0x4000.  Count the number of
    // shifts required to make the match.  No shifts are
    // required for a data length of 16384.
    while((dataLength & mask) == 0){
      cnt++;
      dataLength = dataLength << 1;
    }//end while loop

    //Now shift the mask to the right the same number of
    // places as were required to make the above match.
    // Because the mask consists of a single bit, this
    // guarantees that the resulting value is an even
    // power of two.
    dataLength = mask >> cnt;

    return dataLength;
  }//end convertToPowerOfTwo
  //-----------------------------------------------------//

  //The amplitude response of the recursive filter at a
  // given frequency (a point on the unit circle) can be
  // estimated by dividing the product of the lengths of
  // the vectors connecting that point on the unit circle
  // to all of the zeros by the product of the lengths of
  // the vectors connecting the same point on the unit
  // circle to all of the poles.  The amplitude response
  // at all frequencies between zero and the Nyquist
  // folding frequency can be estimated by performing this
  // calculation at all points in the top half of the unit
  // circle.
  //The bottom half of the unit circle provides the
  // amplitude response for frequencies between the Nyquist
  // folding frequency and the sampling frequency. These
  // values are redundant because they are the same as the
  // values below the folding frequency.
  //Despite the redundancy, this method computes the
  // amplitude response at a set of frequencies between
  // zero and the sampling frequency where the number of
  // frequencies in the set is equal to the data length.
  // This causes the amplitude response to be computed at
  // a set of frequencies that matches the set of
```

```
// frequencies for which the amplitude response is
// computed using the FFT algorithm, making it easier
// to plot the two for comparison purposes.
double[] getVectorAmplitudeResponse(
                                  double[] poleRealData,
                                  double[] poleImagData,
                                  double[] zeroRealData,
                                  double[] zeroImagData){
  double[] amplitudeResponse = new double[dataLength];
  double freqAngle = 0;
  double freqHorizComponent = 0;
  double freqVertComponent = 0;

  //Divide the unit circle into dataLength frequencies
  // and compute the amplitude response at each
  // frequency.
  for(int freqCnt = 0;freqCnt < dataLength;freqCnt++){
    //Get the angle from the origin to the point on the
    // unit circle relative to the horizontal axis.
    freqAngle = freqCnt*2*PI/dataLength;

    //Get the horizontal and vertical components of the
    // distance from the origin to the point on the unit
    // circle.
    freqHorizComponent = cos(freqAngle);
    freqVertComponent = sin(freqAngle);

    //Compute the product of the lengths from the point
    // on the unit circle to each of the zeros.
    //Get the distance from the point on the unit circle
    // to each zero as the square root of the sum of the
    // squares of the sides of a right triangle formed
    // by the zero and the point on the unit circle with
    // the base of the triangle being parallel to the
    // horiontal axis.
    //Declare some working variables.
    double base;//Base of the right triangle
    double height;//Height of the right triangle
    double hypo;//Hypotenuse of the right triangle

    double zeroProduct = 1.0;//Initialize the product.
    //Loop and process all complex conjugate zeros
    for(int cnt = 0;cnt < zeroRealData.length;cnt++){
      //First compute the product for a zero in the
      // upper half of the z-plane
      //Get base of triangle
      base = freqHorizComponent - zeroRealData[cnt];
      //Get height of triangle
      height = freqVertComponent - zeroImagData[cnt];
      //Get hypotenuse of triangle
      hypo = sqrt(base*base + height*height);

      //Compute the running product.
      zeroProduct *= hypo;

      //Continue computing the running product using the
```

```
        // conjugate zero in the lower half of the z-plane.
        // Note the sign change on the imaginary
        // part.
        base = freqHorizComponent - zeroRealData[cnt];
        height = freqVertComponent + zeroImagData[cnt];
        hypo = sqrt(base*base + height*height);
        zeroProduct *= hypo;
      }//end for loop - all zeros have been processed

      //Now compute the product of the lengths to the
      // poles.
      double poleProduct = 1.0;//Initialize the product.
      for(int cnt = 0;cnt < poleRealData.length;cnt++){
        //Begin with the pole in the upper half of the
        // z-plane.
        base = freqHorizComponent - poleRealData[cnt];
        height = freqVertComponent - poleImagData[cnt];
        hypo = sqrt(base*base + height*height);

        //Compute the running product.
        poleProduct *= hypo;

        //Continue computing the running product using the
        // conjugate pole in the lower half of the z-plane.
        // Note the sign change on the imaginary part.
        base = freqHorizComponent - poleRealData[cnt];
        height = freqVertComponent + poleImagData[cnt];
        hypo = sqrt(base*base + height*height);
        poleProduct *= hypo;//product of lengths
      }//end for loop

      //Divide the zeroProduct by the poleProduct.
      //Compute and save the amplitudeResponse for this
      // frequency and then go back to the top of the loop
      // and compute the amplitudeResponse for the next
      // frequency.
      amplitudeResponse[freqCnt] = zeroProduct/poleProduct;

    }//end for loop on data length - all frequencies done

    return amplitudeResponse;
  }//end getVectorAmplitudeResponse
  //-----------------------------------------------------//

  //The phase angle of the recursive filter at a
  // particular frequency (represented by a point on the
  // unit circle) can be determined by subtracting the sum
  // of the angles from the point on the unit circle to all
  // of the poles from the sum of the angles from the same
  // point on the unit circle to all of the zeros.
  //The phase angle for an equally-spaced set of
  // frequencies between zero and the sampling frequency is
  // computed in radians and then converted to degrees in
  // the range from -180 degrees to +180 degrees for return
  // to the calling program..
  double[] getVectorPhaseAngle(double[] poleRealData,
                              double[] poleImagData,
                              double[] zeroRealData,
                              double[] zeroImagData){
    double[] phaseResponse = new double[dataLength];
    double freqAngle = 0;
    double freqHorizComponent = 0;
    double freqVertComponent = 0;

    //Divide the unit circle into dataLength frequencies
    // and compute the phase angle at each frequency.
    for(int freqCnt = 0;freqCnt < dataLength;freqCnt++){
      //Note that the following reference to an angle is
      // not a reference to the phase angle.  Rather, it
      // is a reference to the angle of a point on the unit
      // circle relative to the horizontal axis and the
      // origin of the z-plane.
      freqAngle = freqCnt * 2 * PI/dataLength;

      //Get the horizontal and vertical components of the
      // distance from the origin to the point on the unit
      // circle.
      freqHorizComponent = cos(freqAngle);
      freqVertComponent = sin(freqAngle);

      //Begin by processing all of the complex conjugate
      // zeros.
      //Compute the angle from the point on the unit circle
      // to each of the zeros.  Retain as an angle between
      // zero and 2*PI (360 degrees).
      //Declare some working variables.
      double base;//Base of a right triangle
      double height;//Height of a right triangle
      double zeroAngle;

      double zeroAngleSum = 0;
      //Loop and process all complex conjugate zeros
      for(int cnt = 0;cnt < zeroRealData.length;cnt++){
        //Compute using the zero in upper the half of the
        // z-plane.
        //Get base of triangle
        base = -(freqHorizComponent - zeroRealData[cnt]);
        //Get height of triangle
        height = -(freqVertComponent - zeroImagData[cnt]);

        if(base == 0){//Avoid division by zero.
          zeroAngle = PI/2.0;//90 degees
        }else{//Compute the angle.
          zeroAngle = atan(height/base);
        }//end else

        //Adjust for negative coordinates
        if((base < 0) && (height > 0)){
          zeroAngle = PI + zeroAngle;
        }else if((base < 0) && (height < 0)){
          zeroAngle = PI + zeroAngle;
        }else if((base > 0) && (height < 0)){
```

```
        zeroAngle = 2*PI + zeroAngle;
      }//end else

      //Compute the running sum of the angles.
      zeroAngleSum += zeroAngle;

      //Continue computing the running sum of angles
      // using the conjugate zero in the lower half of
      // the z-plane.  Note the sign change on the
      // imaginary part.
      base = -(freqHorizComponent - zeroRealData[cnt]);
      height = -(freqVertComponent + zeroImagData[cnt]);

      if(base == 0){
        zeroAngle = 3*PI/2.0;//270 degees
      }else{
        zeroAngle = atan(height/base);
      }//end else

      //Adjust for negative coordinates
      if((base < 0) && (height > 0)){
        zeroAngle = PI + zeroAngle;
      }else if((base < 0) && (height < 0)){
        zeroAngle = PI + zeroAngle;
      }else if((base > 0) && (height < 0)){
        zeroAngle = 2*PI + zeroAngle;
      }//end else

      //Add the angle into the running sum of zero
      // angles.
      zeroAngleSum += zeroAngle;

    }//end for loop - all zeros have been processed


    //Now compute the sum of the angles from the point
    // on the unit circle to each of the poles.
    double poleAngle;
    double poleAngleSum = 0;
    //Loop and process all complex conjugate poles
    for(int cnt = 0;cnt < poleRealData.length;cnt++){
      //Compute using pole in the upper half of the
      // z-plane
      base = -(freqHorizComponent - poleRealData[cnt]);
      height = -(freqVertComponent - poleImagData[cnt]);
      if(base == 0){//Avoid division by zero
        poleAngle = PI/2.0;//90 degees
      }else{
        poleAngle = atan(height/base);
      }//end else

      //Adjust for negative coordinates
      if((base < 0) && (height > 0)){
        poleAngle = PI + poleAngle;
      }else if((base < 0) && (height < 0)){
        poleAngle = PI + poleAngle;
      }else if((base > 0) && (height < 0)){
        poleAngle = 2*PI + poleAngle;
      }//end else

      //Compute the running sum of the angles.
      poleAngleSum += poleAngle;

      //Continue computing the sum of angles using the
      // conjugate pole in the lower half of the Z-plane.
      // Note the sign change on the imaginary part.
      base = -(freqHorizComponent - poleRealData[cnt]);
      height = -(freqVertComponent + poleImagData[cnt]);

      if(base == 0){//Avoid division by 0.
        poleAngle = 3*PI/2.0;//270 degees
      }else{
        poleAngle = atan(height/base);
      }//end else

      //Adjust for negative coordinates
      if((base < 0) && (height > 0)){
        poleAngle = PI + poleAngle;
      }else if((base < 0) && (height < 0)){
        poleAngle = PI + poleAngle;
      }else if((base > 0) && (height < 0)){
        poleAngle = 2*PI + poleAngle;
      }//end else

      poleAngleSum += poleAngle;
    }//end for loop - all poles have been processed

    //Subtract the sum of the pole angles from the sum of
    // the zero angles.  Convert the angle from radians
    // to degrees in the process.
    //Note that the minus sign in the following
    // expression is required to cause the sign of the
    // angle computed using this approach to match the
    // sign of the angle computed by the FFT algorithm.
    // This indicates that either this computation or the
    // FFT computation is producing a phase angle having
    // the wrong sign.
    double netAngle =
                   -(zeroAngleSum - poleAngleSum)*180/PI;

    //Normalize the angle to the range from -180 degrees
    // to +180 degrees to make it easier to plot.
    if(netAngle > 180){
      while(netAngle > 180){
        netAngle -= 360;
      }//end while
    }else if(netAngle < -180){
      while(netAngle < -180){
        netAngle += 360;
      }//end while
    }//end else if
    //Save the phase angle for this frequency and then go
    // back to the top of the loop and compute the phase
    // angle for the next frequency.
    phaseResponse[freqCnt] = netAngle;

  }//end for loop on data length - all frequencies done

  return phaseResponse;
}//end getVectorPhaseAngle
//-----------------------------------------------------//

//Receives the complex conjugate roots of a second-order
// polynomial in the form (a+jb)(a-jb).  Multiplies the
// roots and returns the coefficients of the second-order
// polynomial as x*x + 2*a*x + (a*a + b*b) in a three
// element array of type double.
double[] conjToSecondOrder(double a,double b){
  double[] result = new double[]{1,2*a,(a*a + b*b)};
  return result;
}//end conjToSecondOrder
//-----------------------------------------------------//

//Receives the coefficients of a pair of second-order
// polynomials in the form:
// x*x + a*x + b
// x*x + c*x + d
//Multiplies the polynomials and returns the coefficients
// of a fourth-order polynomial in a five-element array
// of type double.
double[] secondToFourthOrder(double a,double b,
                             double c,double d){
  double[] result = new double[]{1,
                                 a + c,
                                 b + c*a + d,
                                     c*b + d*a,
                                           d*b};
  return result;
}//end secondToFourthOrder
//-----------------------------------------------------//

//Receives the coeficients of a pair of fourth order
// polynomials in the form:
// x*x*x*x + a*x*x*x + b*x*x + c*x + d
// x*x*x*x + e*x*x*x + f*x*x + g*x + h
//Multiplies the polynomials and returns the coefficients
// of an eighth-order polynomial in a nine-element
// array of type double.
double[] fourthToEighthOrder(double a,double b,
                             double c,double d,
                             double e,double f,
                             double g,double h){
  double[] result = new double[]{
                             1,
                             a + e,
                             b + e*a + f,
                             c + e*b + f*a + g,
                             d + e*c + f*b + g*a + h,
                                 e*d + f*c + g*b + h*a,
                                       f*d + g*c + h*b,
                                             g*d + h*c,
                                                   h*d};
  return result;
}//end fourthToEighthOrder
//-----------------------------------------------------//

//Receives the coefficients of a pair of eighth order
// polynomials in the following form where xn indicates
// x to the nth power:
// x8 + ax7 + bx6 + cx5 + dx4 + ex3 + fx2 + gx + h
// x8 + ix7 + jx6 + kx5 + lx4 + mx3 + nx2 + ox + p
//Multiplies the polynomials and returns the coefficients
// of a sixteenth-order polynomial in a 17-element
// array of type double
double[] eighthToSixteenthOrder(double a,double b,
                                double c,double d,
                                double e,double f,
                                double g,double h,
                                double i,double j,
                                double k,double l,
                                double m,double n,
                                double o,double p){
  double[] result = new double[]{
                     1,
                     a+i,
                     b+i*a+j,
                     c+i*b+j*a+k,
                     d+i*c+j*b+k*a+l,
                     e+i*d+j*c+k*b+l*a+m,
                     f+i*e+j*d+k*c+l*b+m*a+n,
                     g+i*f+j*e+k*d+l*c+m*b+n*a+o,
                     h+i*g+j*f+k*e+l*d+m*c+n*b+o*a+p,
                       i*h+j*g+k*f+l*e+m*d+n*c+o*b+p*a,
                           j*h+k*g+l*f+m*e+n*d+o*c+p*b,
                               k*h+l*g+m*f+n*e+o*d+p*c,
                                   l*h+m*g+n*f+o*e+p*d,
                                       m*h+n*g+o*f+p*e,
                                           n*h+o*g+p*f,
                                               o*h+p*g,
                                                   p*h};
  return result;
}//end eiththToSixteenthOrder
//-----------------------------------------------------//

//The following six methods are declared in the interface
// named GraphIntfc01, and are required by the plotting
// program named Graph03.
//-----------------------------------------------------//

//This method specifies the number of functions that will
// be plotted by the program named Graph03.
public int getNmbr(){
  //Return number of functions to
  // process.  Must not exceed 5.
  return 5;
}//end getNmbr
//-----------------------------------------------------//

//This method returns the values that will be plotted in
// the first graph by the program named Graph03.
public double f1(double x){
  //Return the impulse response of the filter.
  if(((int)x >= 0) && ((int)x < impulseResponse.length)){
    return impulseResponse[(int)x];
  }else{
    return 0;
  }//end else
}//end f1
//-----------------------------------------------------//

//This method returns the values that will be plotted in
// the second graph by the program named Graph03.
public double f2(double x){
  //Return the amplitude response of the recursive filter
  // obtained by performing a Fourier Transform on the
  // impulse response and converting the result to
  // decibels.  Recall that adding a constant to a
  // decibel plot is equivalent to multiplying the
  // original data by the constant.
  if(((int)x >= 0) &&
              ((int)x < fourierAmplitudeRespnse.length)){
    return 100 +
               (100.0 * fourierAmplitudeRespnse[(int)x]);
  }else{
    return 0;
  }//end else
}//end f2
//-----------------------------------------------------//

//This method returns the values that will be plotted in
// the third graph by the program named Graph03.
public double f3(double x){
  //Return the amplitude response of the recursive filter
  // obtained by dividing the product of the zero vector
  // lengths by the product of the pole vector lengths
  // and converting the result to decibels.  Recall that
  // adding a constant to a decibel plot is equivalent to
  // multiplying the original data by the constant.
  if(((int)x >= 0)
           && ((int)x < vectorAmplitudeResponse.length)){
    return 100 +
               (100.0 * vectorAmplitudeResponse[(int)x]);
  }else{
    return 0;
  }//end else
}//end f3
//-----------------------------------------------------//

//This method returns the values that will be plotted in
// the fourth graph by the program named Graph03.
public double f4(double x){
  //Return the phase response of the recursive filter
  // obtained by performing a Fourier Transform on the
  // impulse response.
  if(((int)x >= 0) &&
                   ((int)x < fourierPhaseAngle.length)){
    return fourierPhaseAngle[(int)x];
  }else{
    return 0;
  }//end else
}//end f4
//-----------------------------------------------------//

//This method returns the values that will be plotted in
// the fifth graph by the program named Graph03.
public double f5(double x){
  //Return the phase response of the recursive filter
  // obtained by subtracting the sum of the pole vector
  // angles from the sum of the zero vector angles.
  if(((int)x >= 0) &&
                    ((int)x < vectorPhaseAngle.length)){
    return vectorPhaseAngle[(int)x];
  }else{
    return 0;
  }//end else
}//end f5

}//end class Dsp046
//=======================================================//



//An object of this class stores and displays the real and
// imaginary parts of sixteen complex poles and sixteen
// complex zeros.  The sixteen poles and sixteen zeros
// form eight conjugate pairs.  Thus, there are eight
// complex conjugate pairs of poles and eight complex
// conjugate pairs of zeros.
//The object also computes and displays the angle in
// degrees for each pole and each zero relative to the
// origin of the z-plane.  It also computes and displays
// the length of an imaginary vector connecting each
// pole and zero to the origin.
//The real and imaginary part for each pole or zero is
// displayed in a TextField object.  The user can modify
// the values by entering new values into the text fields.
// The user can also modify the real and imaginary values
// by selecting a radio button associated with a specific
// pole or zero and then clicking a new location for that
// pole or zero in an auxiliary display that shows the
// complex z-plane and the unit circle in that plane.  When
// the user clicks in the z-plane, the corresponding values
// in the text fields for the selected pole or zero are
// automatically updated.  When the user changes a real
// or imaginary value in a text field, the radio button
// for that pole or zero is automatically selected, and
// the image of that pole or zero in the z-plane is
// automatically changed to show the new position of the
// pole or zero.
//Regardless of which method causes the contents of a text
// field to be modified, a TextListener that is registered
// on the text field causes the displayed angle and length
// to be updated to match the new real and imaginary
// values.
class InputGUI{
  //A reference to an object of this class is stored in the
  // following static variable.  This makes it possible for
  // the original object that created this object to cease
  // to exist without this object becoming eligible for
  // garbage collection.  When that original object is
  // replaced by a new object, the new object can assume
  // ownership of this object by getting its reference from
  // the static variable. Thus, ownership of this object
  // can be passed along from one object to the next.
  static InputGUI refToObj = null;

  //The following ButtonGroup object is used to group
  // radio buttons to cause them to behave in a mututlly
  // exclusive way.  The zero buttons and the pole buttons
  // are all placed in the same group so that only one zero
  // or one pole can be selected at any point in time.
  ButtonGroup buttonGroup = new ButtonGroup();

  //A reference to an auxiliary display showing the
  // z-plane is stored in the following instance variable.
  ZPlane refToZPlane;

  //This is the default number of poles and zeros
  // including the conjugates.  These values cannot be
  // changed by the user.  However, the effective number of
  // poles or zeros can be reduced by moving poles and/or
  // zeros to the origin in the z-plane, rendering them
  // ineffective in the recursive filtering process.
  // Moving poles and zeros to the origin causes feedback
  // and feed-forward zeros to go to zero.
  int numberPoles = 16;
  int numberZeros = 16;

  //The following variables refer to a pair of buttons used
  // to make it possible for the user to place all the
  // poles and zeros at the origin. In effect, these are
  // reset buttons relative to the pole and zero locations.
  JButton clearPolesButton =
                      new JButton("Move Poles to Origin");
  JButton clearZerosButton =
                      new JButton("Move Zeros to Origin");

  //The following text field stores the default data length
  // at startup.  This value can be changed by the user to
  // investigate the impact of changes to the data length
  // for a given set of poles and zeros.
  TextField dataLengthField = new TextField("1024");

  //The following array objects get populated with numeric
  // values from the text fields by the method named
  // captureTextFieldData.  That method should be called
  // to populate them with the most current text field
  // data when the most current data is needed.
  double[] poleRealData = new double[numberPoles/2];
  double[] poleImagData = new double[numberPoles/2];
  double[] zeroRealData = new double[numberZeros/2];
  double[] zeroImagData = new double[numberZeros/2];

  //The following arrays are populated with references to
  // radio buttons, each of which is associated with a
  // specific pair of complex conjugate poles or zeros.
  JRadioButton[] poleRadioButtons =
                         new JRadioButton[numberPoles/2];
  JRadioButton[] zeroRadioButtons =
                         new JRadioButton[numberZeros/2];

  //The following arrays are populated with references to
  // text fields, each of which is associated with a
  // specific pair of complex conjugate poles or zeros.
  // The text fields contain the real values, imaginary
  // values, and the values of the angle and the length
  // specified by the real and imaginary values.
  //The size of the following arrays is only half the
  // number of poles and zeros because the conjugate is
  // generated on the fly when it is needed.
  //I wanted to use JTextField objects, but JTextField
  // doesn't have an addTextListener method.  I needed to
  // register a TextListener object on each real and
  // imaginary text field to compute the angle and length
  // each time the contents of a text field changes, so I
  // used the AWT TextField class instead.
  TextField[] poleReal = new TextField[numberPoles/2];
  TextField[] poleImag = new TextField[numberZeros/2];
  TextField[] poleAngle = new TextField[numberZeros/2];
  TextField[] poleLength = new TextField[numberZeros/2];
  TextField[] zeroReal = new TextField[numberZeros/2];
  TextField[] zeroImag = new TextField[numberZeros/2];
  TextField[] zeroAngle = new TextField[numberZeros/2];
  TextField[] zeroLength = new TextField[numberZeros/2];
  //-------------------------------------------------------//

  InputGUI(){//constructor

    //Instantiate a new JFrame object and condition its
    // close button.
    JFrame guiFrame =
                  new JFrame("Copyright 2006 R.G.Baldwin");
    guiFrame.setDefaultCloseOperation(
                                   JFrame.EXIT_ON_CLOSE);

    //The following JPanel contains a JLabel, a TextField,
    // and two JButton objects.  The label simply provides
    // instructions to the user regarding the entry of a
    // new data length.  The text field is used for entry
    // of a new data length value by the user at runtime.
    // The buttons are used to cause the poles and zeros
    // to be moved to the origin in two groups.  Moving
    // both the poles and the zeros to the origin
    // effectively converts the recursive filter to an
    // all-pass filter.
    //This panel is placed in the NORTH location of the
    // JFrame resulting in the name northControlPanel
    JPanel northControlPanel = new JPanel();
    northControlPanel.setLayout(new GridLayout(0,2));
    northControlPanel.
              add(new JLabel("Data Length as Power of 2"));
    northControlPanel.add(dataLengthField);
    northControlPanel.add(clearPolesButton);
    northControlPanel.add(clearZerosButton);
    guiFrame.add(northControlPanel,BorderLayout.NORTH);

    //Register an action listener on the clearPolesButton
    // to set the contents of the text fields that
    // represent the locations of the poles to 0.  This
    // also causes the poles to move to the origin in the
    // display of the z-plane.
    clearPolesButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          for(int cnt = 0;cnt < numberPoles/2;cnt++){
            poleReal[cnt].setText("0");
            poleImag[cnt].setText("0");
          }//end for loop
        }//end actionPerformed
      }//end new ActionListener
    );//end addActionListener(

    //Register action listener on the clearZerosButton to
    // set the contents of the text fields that
    // represent the locations of the zeros to 0  This
    // also causes the zeros to move to the origin in the
    // display of the z-plane.
    clearZerosButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          for(int cnt = 0;cnt < numberZeros/2;cnt++){
            zeroReal[cnt].setText("0");
            zeroImag[cnt].setText("0");
          }//end for loop
        }//end actionPerformed
      }//end new ActionListener
    );//end addActionListener(


    //The following JPanel object contains two other JPanel
    // objects, one for zeros and one for poles.  They
    // are the same size with one located above the other.
    // The panel containing zero data is green.  The panel
    // containing pole data is yellow.  This panel is
    // placed in the CENTER of the JFrame object.  Hence
    // the name centerControlPanel.
    JPanel centerControlPanel = new JPanel();
    centerControlPanel.setLayout(new GridLayout(2,1));

    //The following JPanel is populated with text fields
    // and radio buttons that represent the zeros.
    JPanel zeroPanel = new JPanel();
    zeroPanel.setBackground(Color.GREEN);

    //The following JPanel is populated with text fields
    // and radio buttons that represent the poles.
    JPanel polePanel = new JPanel();
    polePanel.setBackground(Color.YELLOW);

    //Add the panels containing textfields and readio
    // buttons to the larger centerControlPanel.
    centerControlPanel.add(zeroPanel);
    centerControlPanel.add(polePanel);

    //Add the centerControlPanel to the CENTER of the
    // JFrame object.
    guiFrame.getContentPane().add(
                   centerControlPanel,BorderLayout.CENTER);

    //Instantiate a text listener that will be registered
    // on each of the text fields containing real and
    // imaginary values, each pair of which specifies the
    // location of a pole or a zero.
    MyTextListener textListener = new MyTextListener();

    //A great deal is accomplished in each of the following
    // two for loops.
    //Begin by populating the arrays described earlier with
    // radio buttons and text fields.
    //Then place the radio buttons associated with the
    // poles and zeros in the same group to make them
    // behave in a mutually exclusive manner.
    //Next, place the components on the polePanel and the
    // zero panel.
    //Then disable the text fields containing the angle
    // and the length to prevent the user from entering
    // data into them.  Note that this does not prevent
    // the program from writing text into the text fields
    // containing the angle and the length.  The text
    // fields are disabled only insofar as manual input by
    // the user is concerned.
    //After that, set the name property for each of the
    // real and imaginary text fields.  These name property
    // values will be used later by a common TextListener
    // object to determine which text field fired a
    // TextEvent.
    //Finally, register a common TextListener object on the
    // real and imaginary text fields to cause the angle
    // and the length to be computed and displayed when the
    // text value changes for any reason.

    //Deal with the poles.
    polePanel.setLayout(new GridLayout(0,5));
    //Place a row of column headers
    polePanel.add(new JLabel("Poles"));
    polePanel.add(new JLabel("Real"));
    polePanel.add(new JLabel("Imag"));
    polePanel.add(new JLabel("Angle (deg)"));
    polePanel.add(new JLabel("Length"));
    //Take the actions described above with respect to the
    // poles.
    for(int cnt = 0;cnt < numberPoles/2;cnt++){
      poleRadioButtons[cnt] = new JRadioButton("" + cnt);
      poleReal[cnt] = new TextField("0");
      poleImag[cnt] = new TextField("0");
      poleAngle[cnt] = new TextField("0");
      poleLength[cnt] = new TextField("0");
      buttonGroup.add(poleRadioButtons[cnt]);
      polePanel.add(poleRadioButtons[cnt]);
      polePanel.add(poleReal[cnt]);
      polePanel.add(poleImag[cnt]);
      polePanel.add(poleAngle[cnt]);
      poleAngle[cnt].setEnabled(false);
      polePanel.add(poleLength[cnt]);
      poleLength[cnt].setEnabled(false);
      poleReal[cnt].setName("poleReal" + cnt);
      poleImag[cnt].setName("poleImag" + cnt);
      poleReal[cnt].addTextListener(textListener);
      poleImag[cnt].addTextListener(textListener);
    }//end for loop

    //Deal with the zeros.
    zeroPanel.setLayout(new GridLayout(0,5));
    //Place a row of column headers
    zeroPanel.add(new JLabel("Zeros"));
    zeroPanel.add(new JLabel("Real"));
    zeroPanel.add(new JLabel("Imag"));
    zeroPanel.add(new JLabel("Angle (deg)"));
    zeroPanel.add(new JLabel("Length"));
    //Now take the actions described above with respect to
    // the zeros.
    for(int cnt = 0;cnt < numberZeros/2;cnt++){
      zeroRadioButtons[cnt] = new JRadioButton("" + cnt);
      zeroReal[cnt] = new TextField("0");
      zeroImag[cnt] = new TextField("0");
      zeroAngle[cnt] = new TextField("0");
      zeroLength[cnt] = new TextField("0");
      buttonGroup.add(zeroRadioButtons[cnt]);
      zeroPanel.add(zeroRadioButtons[cnt]);
      zeroPanel.add(zeroReal[cnt]);
      zeroPanel.add(zeroImag[cnt]);
      zeroPanel.add(zeroAngle[cnt]);
      zeroAngle[cnt].setEnabled(false);
      zeroPanel.add(zeroLength[cnt]);
      zeroLength[cnt].setEnabled(false);
      zeroReal[cnt].setName("zeroReal" + cnt);
      zeroImag[cnt].setName("zeroImag" + cnt);
      zeroReal[cnt].addTextListener(textListener);
      zeroImag[cnt].addTextListener(textListener);
    }//end for loop

    //Now create an auxiliary display of the z-plane
    // showing a unit circle.  The user locates poles and
    // zeros on it by first selecting the radio button that
    // specifies a particular pole or zero, and then
    // clicking in the z-plane with the mouse.  (Note that
    // locating a pole outside the unit circle should
    // result in an unstable recursive filter with an
    // output that continues to grow with time.)
    refToZPlane = new ZPlane();

    //Register an anonymous MouseListerer object on the
    // z-plane.
    refToZPlane.addMouseListener(
      new MouseAdapter(){
        public void mousePressed(MouseEvent e){
          //Get and save the coordinates of the mouse click
          // relative to an orgin that has been translated
          // from the upper-left corner to a point near the
          // center of the frame.
          //Change the sign on the vertical coordinate to
          // cause the result to match our expectation of
          // positive vertical values going up the screen
          // instead of going down the screen.
          int realCoor = e.getX() - refToZPlane.
                                    translateOffsetHoriz;
          int imagCoor = -(e.getY() - refToZPlane.
                                    translateOffsetVert);

          //The new coordinate values are deposited in the
          // real and imaginary text fields associated with
          // the selected radio button.
          //Examine the radio buttons to identify the pair
          // of real and imaginary text fields into which
          // the new coordinate values should be deposited.
          //Note that one radio button is always selected,
          // so don't click in the z-plane unless you
          // really do want to modify the coordinate values
          // in the text fields associated with the
          // selected radio button.
          //Note that the integer coordinate values are
          // converted to fractional coordinate values by
          // dividing the integer coordinate values by the
          // radius (in pixels) of the unit circle as
          // displayed on the z-plane.

          //Examine the pole buttons first.
          boolean selectedFlag = false;
          for(int cnt = 0;cnt < numberPoles/2;cnt++){
            if(poleRadioButtons[cnt].isSelected()){
              poleReal[cnt].setText("" + (realCoor/
                  (double)(refToZPlane.unitCircleRadius)));
              poleImag[cnt].setText("" + abs(imagCoor/
                  (double)(refToZPlane.unitCircleRadius)));
              //Set the selectedFlag to prevent the zero
              // radio buttons from being examined.
              selectedFlag = true;
              //No other button can be selected.  They are
              // mutually exclusive.
              break;
            }//end if
          }//end for loop

          if(!selectedFlag){//Skip if selectedFlag is true.
            //Examine the zero buttons
            for(int cnt = 0;cnt < numberZeros/2;cnt++){
              if(zeroRadioButtons[cnt].isSelected()){
                zeroReal[cnt].setText("" + (realCoor/
                  (double)(refToZPlane.unitCircleRadius)));
                zeroImag[cnt].setText("" + abs(imagCoor/
                  (double)(refToZPlane.unitCircleRadius)));
                break;//No other button can be selected
              }//end if
            }//end for loop
          }//end if on selectedFlag

          //Cause the display of the z-plane to be
          // repainted showing the new location for the
          // pole or zero.
          refToZPlane.repaint();
        }//end mousePressed
      }//end new class
    );//end addMouseListener

    //Set the size and location of the InputGUI (JFrame)
    // object on the screen.  Position it in the upper
    // right corner of a 1024x768 screen.
    guiFrame.setBounds(1024-472,0,472,400);

    //Cause two displays to become visible.  Prevent the
    // user from resizing them.
    guiFrame.setResizable(false);
    guiFrame.setVisible(true);
    refToZPlane.setResizable(false);
    refToZPlane.setVisible(true);

    //Save the reference to this GUI object so that it can
    // be recovered later after a new instance of the
    // Dsp046 class is instantiated.
    InputGUI.refToObj = this;
  }//end constructor
  //-----------------------------------------------------//


  //This is a utility method used to capture the latest
  // text field data, convert it into type double, and
  // store the numeric values into arrays.
```

```
//The try-catch handlers are designed to deal with the
// possibility that a text field contains a text value
// that cannot be converted to a double value when the
// method is invoked.  In that case, the value is
// replaced by 0 and an error message is displayed on
// the command-line screen. This is likely to happen,
// for example, if the user deletes the contents of a
// text field in preparation for entering a new value.
// In that case, the TextListener will invoke this
// method in an attempt to compute and display new
// values for the angle and the length.
//One way to enter a new value in the text field is to
// highlight the old value before starting to type the
// new value.  Although this isn't ideal, it is the best
// that I could come up with in order to cause the angle
// and the length to be automatically computed and
// displayed each time a new value is entered into the
// text field.
void captureTextFieldData(){
  //Encapsulate the pole data in an array object.
  for(int cnt = 0;cnt < numberPoles/2;cnt++){
    try{
      poleRealData[cnt] =
             Double.parseDouble(poleReal[cnt].getText());
    }catch(NumberFormatException e){
      //The text in the text field could not be converted
      // to type double.
      poleReal[cnt].setText("0");
      System.out.println("Warning: Illegal entry for " +
             "poleReal[" + cnt + "], " + e.getMessage());
    }//end catch

    try{
      poleImagData[cnt] =
             Double.parseDouble(poleImag[cnt].getText());
    }catch(NumberFormatException e){
      poleImag[cnt].setText("0");
      System.out.println("Warning: Illegal entry for " +
             "poleImag[" + cnt + "], " + e.getMessage());
    }//end catch
  }//end for loop

  //Encapsulate the zero data in an array object.
  for(int cnt = 0;cnt < numberZeros/2;cnt++){
    try{
      zeroRealData[cnt] =
             Double.parseDouble(zeroReal[cnt].getText());
    }catch(NumberFormatException e){
      zeroReal[cnt].setText("0");
      System.out.println("Warning: Illegal entry for " +
             "zeroReal[" + cnt + "], " + e.getMessage());
    }//end catch

    try{
      zeroImagData[cnt] =
             Double.parseDouble(zeroImag[cnt].getText());
      }catch(NumberFormatException e){
        zeroImag[cnt].setText("0");
        System.out.println("Warning: Illegal entry for " +
               "zeroImag[" + cnt + "], " + e.getMessage());
      }//end catch
    }//end for loop

  }//end captureTextFieldData method
  //-----------------------------------------------------//

//=======================================================//
//This is an inner text listener class.  A registered
// object of the class is notified whenever the text value
// for any of the real or imaginary values in the pole and
// zero text fields changes for any reason.  When the
// event handler is notified, it computes and displays the
// angle specified by the ratio of the imaginary part to
// the real part.  All angles are expressed in degrees
// between 0 and 359.9 inclusive.
//Also, when notified, the event handler computes the
// length of an imaginary vector connecting the pole or
// zero to the origin as the square root of the sum of the
// squares of the real and imaginary parts.
//Finally, the event handler also causes the z-plane to be
// repainted to display the new location for the pole or
// zero represented by the text field that fired the event.
class MyTextListener implements TextListener{

  public void textValueChanged(TextEvent e){
    //Invoke the captureTextFieldData method to cause the
    // latest values in the text fields to be converted to
    // numeric double values and stored in arrays.
    captureTextFieldData();

    //Identify the text field that fired the event and
    // respond appropriately.  Cause the radio button
    // associated with the modified text field to become
    // selected.
    boolean firingObjFound = false;
    String name = ((Component)e.getSource()).getName();
    for(int cnt = 0;cnt < numberPoles/2;cnt++){
      if((name.equals("poleReal" + cnt)) ||
                          (name.equals("poleImag" + cnt))){
        //Compute and set the angle to the pole.
        poleAngle[cnt].setText(getAngle(
                    poleRealData[cnt],poleImagData[cnt]));
        //Compute and set the length of an imaginary vector
        // connecting the pole to the origin.
        poleLength[cnt].setText("" + sqrt(
                    poleRealData[cnt]*poleRealData[cnt] +
                    poleImagData[cnt]*poleImagData[cnt]));
        //Select the radio button.
        poleRadioButtons[cnt].setSelected(true);
        firingObjFound = true;//Avoid testing zeros
        break;
      }//end if
    }//end for loop

    if(!firingObjFound){
      for(int cnt = 0;cnt < numberZeros/2;cnt++){
        if((name.equals("zeroReal" + cnt)) ||
                                  (name.equals("zeroImag" + cnt))){
          //Compute and set the angle to the zero.
          zeroAngle[cnt].setText(getAngle(
                         zeroRealData[cnt],zeroImagData[cnt]));
          //Compute and set the length of an imaginary
          // vector connecting the zero to the origin.
          zeroLength[cnt].setText("" + sqrt(
                         zeroRealData[cnt]*zeroRealData[cnt] +
                         zeroImagData[cnt]*zeroImagData[cnt]));
          //Select the radio button.
          zeroRadioButtons[cnt].setSelected(true);
          break;
        }//end if
      }//end for loop
    }//end if on firingObjFound

    //Repaint the z-plane to show the new pole or zero
    // location.
    refToZPlane.repaint();

  }//end textValueChanged
  //-------------------------------------------------------//

  //This method returns the angle in degrees indicated by
  // the incoming real and imaginary values in the range
  // from 0 to 359.9 degrees.
  String getAngle(double realVal,double imagVal){
    String result = "";
    //Avoid division by 0
    if((realVal == 0.0) && (imagVal >= 0.0)){
      result = "" + 90;
    }else if((realVal == 0.0) && (imagVal < 0.0)){
      result = "" + 270;
    }else{
      //Compute the angle in radians.
      double angle = atan(imagVal/realVal);

      //Adjust for negative coordinates
      if((realVal < 0) && (imagVal == 0.0)){
        angle = PI;
      }else if((realVal < 0) && (imagVal > 0)){
        angle = PI + angle;
      }else if((realVal < 0) && (imagVal < 0)){
        angle = PI + angle;
      }else if((realVal > 0) && (imagVal < 0)){
        angle = 2*PI + angle;
      }//end else

      //Convert from radians to degrees
      angle = angle*180/PI;
      //Convert the angle from double to String.
      String temp1 = "" + angle;
      if(temp1.length() >= 5){
        result = temp1.substring(0,5);
      }else{
        result = temp1;
      }//end else
    }//end else
    return result;
  }//end getAngle
  //----------------------------------------------------//
}//end inner class MyTextListener
//====================================================//

//This is an inner class.  An object of this class is
// an auxiliary display that represents the z-plane.
class ZPlane extends Frame{
  Insets insets;
  int totalWidth;
  int totalHeight;
  //Set the size of the display area in pixels.
  int workingWidth = 464;
  int workingHeight = 464;
  int unitCircleRadius = 200;//Radius in pixels.
  int translateOffsetHoriz;
  int translateOffsetVert;

  ZPlane(){//constructor
    //Get the size of the borders and the banner.  Set the
    // overall size to accommodate them and still provide
    // a display area whose size is specified by
    // workingWidth and workingHeight.
    //Make the frame visible long enough to get the values
    // of the insets.
    setVisible(true);
    insets = getInsets();
    setVisible(false);
    totalWidth = workingWidth + insets.left + insets.right;
    totalHeight =
                workingHeight + insets.top + insets.bottom;
    setTitle("Copyright 2006, R.G.Baldwin");

    //Set the size of the new Frame object so that it will
    // have a working area that is specified by
    // workingWidth and workingHeight. Elsewhere in the
    // program, the resizable property of the Frame is set
    // to false so that the user cannot modify the size.

    //Locate the Frame object in the upper-center of the
    // screen
    setBounds(408,0,totalWidth,totalHeight);

    //Move the origin to the center of the working area.
    // Note, however, that the direction of positive-y is
    // down the screen.  This will be compensated for
    // elsewhere in the program.
    translateOffsetHoriz = workingWidth/2 + insets.left;
    translateOffsetVert = workingHeight/2 + insets.top;

    //Register a window listener that can be used to
    // terminate the program by clicking the X-button in
    // the upper right corner of the frame.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing
      }//end class def
    );//end addWindowListener
  }//end constructor
  //--------------------------------------------------------//

  //This overridden paint method is used to repaint the
  // ZPlane object when the program requests a repaint on
  // the object.
  public void paint(Graphics g){
    //Translate the origin to the center of the working
    // area in the frame.
    g.translate(translateOffsetHoriz,translateOffsetVert);
    //Draw a round oval to represent the unit circle in the
    // z-plane.
    g.drawOval(-unitCircleRadius,-unitCircleRadius,
               2*unitCircleRadius,2*unitCircleRadius);
    //Draw horizontal and vertical axes at the new origin.
    g.drawLine(-workingWidth/2,0,workingWidth/2,0);
    g.drawLine(0,-workingHeight/2,0,workingHeight/2);

    //Invoke the captureTextFieldData method to cause the
    // latest data in the text fields to be converted to
    // double numeric values and stored in arrays.
    captureTextFieldData();

    //Draw the poles in red using the data retrieved from
    // the text fields.  Note that the unitCircleRadius in
    // pixels is used to convert the locations of the
    // poles from double values to screen pixels.
    g.setColor(Color.RED);

    for(int cnt = 0;cnt < poleRealData.length;cnt++){
      //Draw the conjugate pair of poles as small red
      // squares centered on the poles.
      int realInt =
                 (int)(poleRealData[cnt]*unitCircleRadius);
      int imagInt =
                 (int)(poleImagData[cnt]*unitCircleRadius);
      //Draw the pair of conjugate poles.
      g.drawRect(realInt-2,imagInt-2,4,4);
      g.drawRect(realInt-2,-imagInt-2,4,4);
    }//end for loop

    //Draw the zeros in black using the data retrieved from
    // the text fields.  Note that the unitCircleRadius in
    // pixels is used to convert the locations of the
    // zeros from double values to screen pixels.

    g.setColor(Color.BLACK);//Restore color to black

    for(int cnt = 0;cnt < zeroRealData.length;cnt++){
      //Draw the conjugate pair of zeros as small black
      // circles centered on the zeros.
      int realInt =
                  (int)(zeroRealData[cnt]*unitCircleRadius);
      int imagInt =
                  (int)(zeroImagData[cnt]*unitCircleRadius);
      //Draw the pair of conjugate zeros.
      g.drawOval(realInt-2,imagInt-2,4,4);
      g.drawOval(realInt-2,-imagInt-2,4,4);
    }//end for loop

  }//end overridden paint method
}//end inner class ZPlane
//===================================================//

}//end class InputGUI
```

Listing 1

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 "recursive filter" convolution graph plot Fourier pole zero z-plane GUI amplitude phase polynomial root "unit circle" conjugate "radio button" "text field" decibel frequency "impulse response" low-pass band-pass high-pass narrow-band real imaginary oscillator

-end-