The User Input GUI for the Recursive Filtering Workbench in Java

Learn to write and to understand the user input GUI for the Recursive Filtering Workbench that was presented in an earlier lesson. Also learn how the interactive behavior of the program was accomplished using cross-linked listener objects.

Published: November 14, 2006
By Richard G. Baldwin

Java Programming Notes # 1512

Preface

This is the third installment of a multi-part tutorial on digital recursive filtering. The first installment was published in the earlier tutorial lesson entitled A Recursive Filtering Workbench in Java. The second installment was published in the previous lesson entitled The Driver Class for the Recursive Filtering Workbench in Java.

The primary purpose of this lesson is to present and explain the Java code required to implement the user input GUI shown in the upper right corner of Figure 1, and also shown in Figure 3. A second purpose is to teach you how the interactive behavior of the program was accomplished using cross-linked listener objects.

This lesson builds on the information provided in the earlier lessons. I won't repeat much of the information from those earlier lessons here. Unless you are already well-schooled in digital recursive filtering, it will probably be a good idea for you to study the earlier lessons before embarking on this lesson.

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.

By the end of the last installment, you should have learned how to write a Java program to create such an interactive 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.

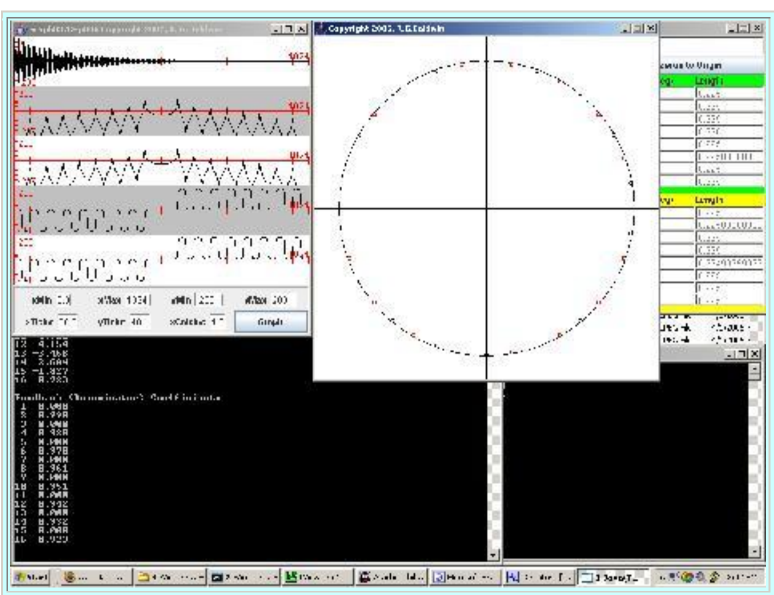

Figure 1

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 Figure 1 is 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 from Figure 1 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.)*

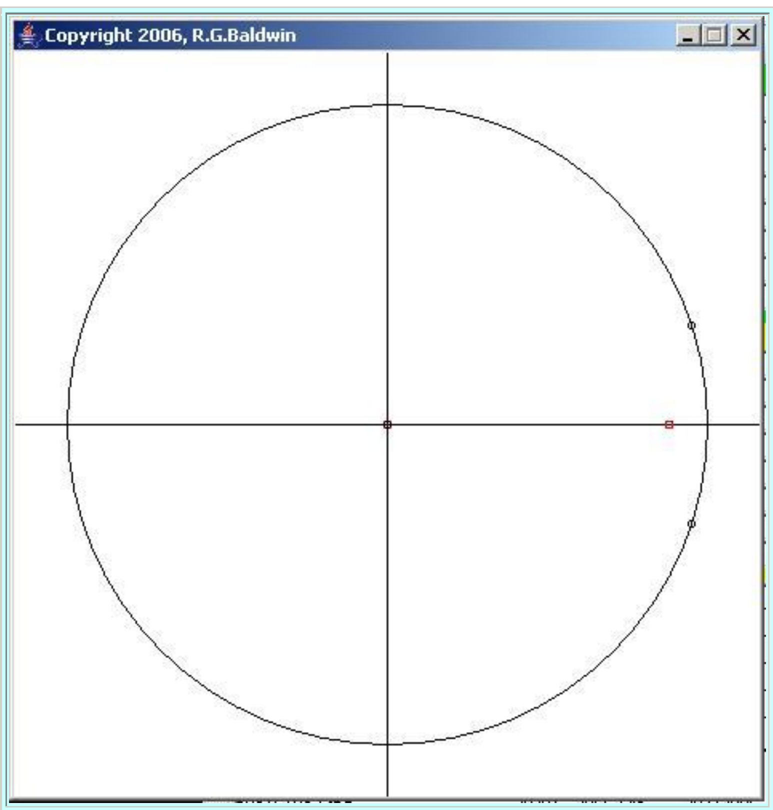

Figure 2

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 corner 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 3.)*

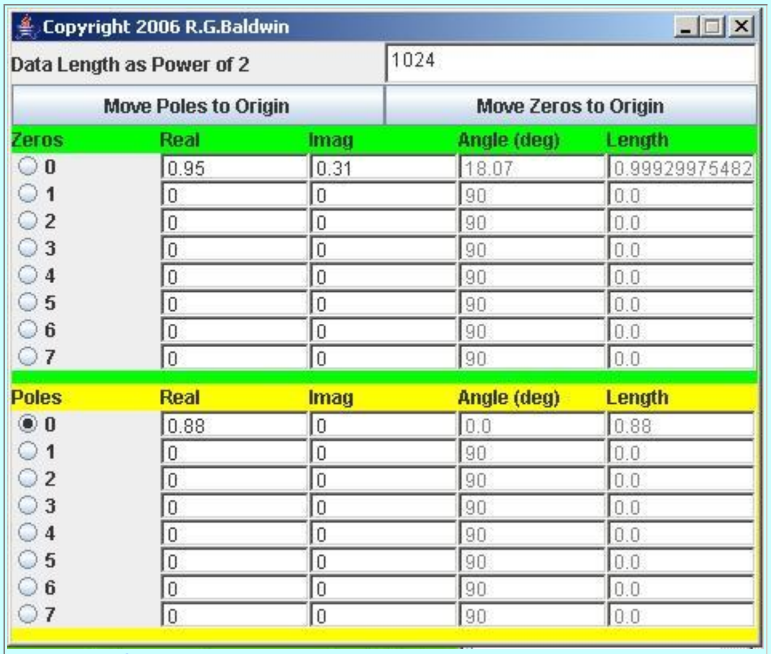

Figure 3

Purpose of the GUI

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.

As mentioned earlier, the primary purpose of this lesson is to present and explain the Java code required to implement the GUI shown in Figure 3.

The two GUIs are connected

When this GUI is used to relocate a complex 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 3 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 in the z-plane 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 4 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.

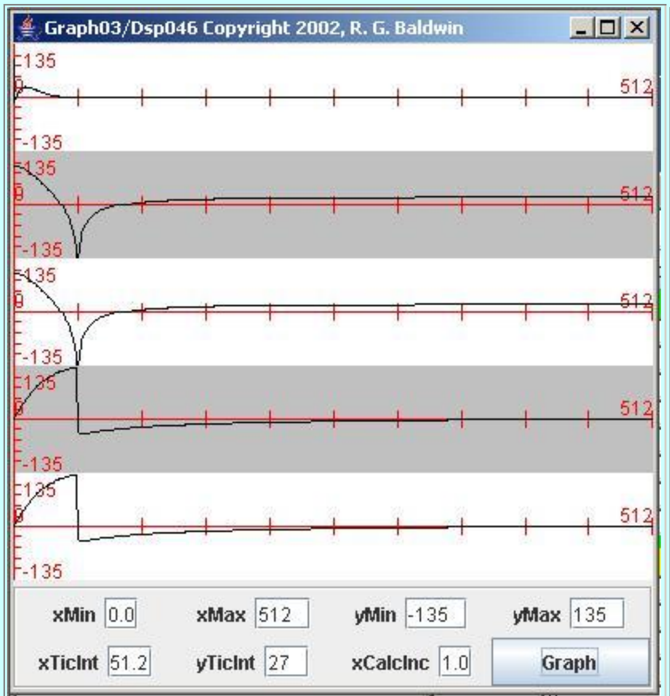

Figure 4

Two computational approaches

Two versions of the amplitude response and two versions of the phase response are plotted in the GUI shown in Figure 4. 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 4 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 in this multi-part tutorial. 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 .)*

See the earlier lesson entitled A Recursive Filtering Workbench in Java for general background information on recursive filtering in general and this recursive filtering workbench in particular.

Discussion and Sample Code

This program is composed of several rather long classes, resulting in a large program. The size of the program is the primary reason that I am publishing it in several installments. In the earlier lessons, I promised that this installment would present and explain the class named **InputGUI**. This installment will fulfill that promise.

Sixteen complex poles and sixteen complex zeros

An object of the class named **InputGUI** stores and displays the real and imaginary parts of sixteen complex poles and sixteen complex zeros as shown in Figure 3. The sixteen poles and sixteen zeros form eight complex conjugate pairs. Thus, there are eight complex conjugate pairs of poles and eight complex conjugate pairs of zeros.

> *(Only one half of each complex conjugate pair is actually stored. The other half is created on the fly when it is needed.)*

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 values

The real and imaginary values for each pole and each zero are 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 in Figure 3 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 *(see Figure 2)*.

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.

The angle and the vector length

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.

Will discuss in fragments

I will discuss the class named **InputGUI** as a series of code fragments. A complete listing of the entire program, *(including the class named **InputGUI**)* is provided in Listing 29 near the end of the lesson.

Beginning of the class definition

The class definition for the class named **InputGUI** begins in Listing 1.

```
class InputGUI{
  static InputGUI refToObj = null;
```

Listing 1

The code in Listing 1 declares a single static variable named **refToObj**. A reference to an object of this class is stored in that variable. That 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.

> *(This process was explained from the viewpoint of the driver class in the previous lesson entitled The Driver Class for the Recursive Filtering Workbench in Java.)*

Declare and initialize a ButtonGroup variable

Listing 2 declares and initializes a **ButtonGroup** variable with a reference to a new **ButtonGroup** object. This **ButtonGroup** object is used later to group the radio buttons shown on the left side of Figure 3 to cause them to behave in a mutually exclusive way.

The radio buttons are used to select a zero or a pole for processing, such as for changing its location by clicking the mouse in the z-plane shown in Figure 2. 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 for processing at any point in time.

```
  ButtonGroup buttonGroup = new ButtonGroup();
```

Listing 2

A reference to the z-plane GUI object

Listing 3 declares a reference variable that will be used later to store a reference to the z-plane GUI object shown in Figure 2.

```
  ZPlane refToZPlane;
```

Listing 3

Default number of poles and zeros

Listing 4 declares and initializes two variables that specify 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 that are used in the recursive filtering process 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 coefficients to go to zero. However, this does not reduce the computational cost of applying the filter. If computational cost is of concern, the program should be modified so as to actually eliminate the unwanted poles and/or zeros.)*

```
  int numberPoles = 16;
  int numberZeros = 16;
```

Listing 4

Moving poles and zeros to the origin

The poles and the zeros can be selectively moved to the origin by entering real and imaginary values of 0 in the corresponding text fields in Figure 3. Alternatively, all of the poles or all of the zeros can be moved to the origin in a wholesale fashion by clicking one or the other of the two buttons shown near the top of the GUI in Figure 3.

The code in Listing 5 instantiates and saves references to the two buttons shown near the top of Figure 3 that are used to move the poles and/or the zeros to the origin in a wholesale fashion. 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");
```

Listing 5

Listener objects will be registered on these buttons later to accomplish the intended purpose of the buttons.

The default data length

Listing 6 declares a variable that is used to specify the length of the impulse response that is used to estimate the frequency response of the recursive filter as shown by the second and fourth graphs in Figure 4. The code in Listing 6 initializes this variable to its default length at startup.

```
  TextField dataLengthField = new
TextField("1024");
```

Listing 6

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.

As you can see from Listing 6, the default value for the data length is an even power of two. As explained in the earlier lesson entitled A Recursive Filtering Workbench in Java, this value must be an even power of two for the FFT program that is used to estimate the frequency response to work properly. If the user enters a new value that is not an even power of two, that value is automatically modified to cause it to become an even power of two. The code to accomplish this was explained in the previous lesson.

Numeric pole and zero data values

The array objects that are created in Listing 7 are populated later with numeric values from the text fields by the method named **captureTextFieldData** *(to be discussed later)*. That method is called to populate the array objects with the most current text field data whenever 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];
```

Listing 7

References to the radio buttons

The array objects shown in Listing 8 are populated later with references to the radio buttons shown on the left side of Figure 3. Each radio button is associated with a specific pair of complex conjugate poles or zeros.

```
  JRadioButton[] poleRadioButtons =
                         new
JRadioButton[numberPoles/2];
  JRadioButton[] zeroRadioButtons =
                         new
```

```
JRadioButton[numberZeros/2];
```

Listing 8

References to the text fields

The array objects shown in Listing 9 are populated later with references to text fields, each of which is associated with a specific pair of complex conjugate poles or zeros.

```
  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];
```

Listing 9

The individual text fields contain the real values, the imaginary values, the values of the angles, and the values of the lengths of the vectors specified by the real and imaginary values.

The size of the array objects

The size of each array object created in Listing 9 is only half the number of poles and zeros because the conjugate is generated on the fly when it is needed.

> *(I originally planned to use **JTextField** objects for this purpose, but realized during the writing of this program that the **JTextField** class 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 objects of the AWT **TextField** class instead.)*

The constructor for the InputGUI class

The constructor for the **InputGUI** class begins in Listing 10.

```
  InputGUI(){//constructor
    JFrame guiFrame =
                  new JFrame("Copyright 2006
```

```
R.G.Baldwin");
    guiFrame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
```

Listing 10

The code in Listing 10 instantiates a new **JFrame** object and conditions its close button so that the program will terminate when the user clicks the X-button in the upper right corner of Figure 3.

The northControlPanel object

Listing 11 instantiates a new **JPanel** object that contains a **JLabel**, a **TextField**, and two **JButton** objects. This panel appears at the very top of Figure 3.

```
    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);
```

Listing 11

Purposes of the components

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/or the zeros to be moved to the origin in the wholesale fashion described earlier.

> *(Moving all the poles and all the zeros to the origin effectively converts the recursive filter into an all-pass filter.)*

This panel is placed in the NORTH location of the **JFrame** object resulting in the name **northControlPanel** for the variable that refers to the object.

An anonymous listener to move all the poles to the origin

Listing 12 registers an anonymous action listener on the **clearPolesButton** shown in the upper left of Figure 3 to set the contents of the text fields that represent the locations of the poles to 0.

```
    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
```

Listing 12

As explained earlier, this action also causes the poles to move to the origin in the display of the z-plane shown in Figure 2. You will see how that is accomplished in the presentation and explanation of the inner class named **MyTextListener** in the next installment of this multi-part tutorial.

An anonymous listener to move all the zeros to the origin

Listing 13 registers an anonymous action listener on the **clearZerosButton** shown in the upper right of Figure 3 to set the contents of the text fields that represent the locations of the zeros to 0.

```
    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
```

Listing 13

As explained earlier, this action also causes the zeros to move to the origin in the display of the z-plane shown in Figure 2, and you will also see how that is accomplished in the next installment of this multi-part tutorial.

A JPanel in the center of the JFrame

The **JPanel** object that is instantiated in Listing 14 appears below the two buttons in Figure 3. It contains two other **JPanel** objects, one for poles and the other for zeros.

```
    JPanel centerControlPanel = new JPanel();
```

```
    centerControlPanel.setLayout(new
GridLayout(2,1));
```

Listing 14

The two **JPanel** objects contained by this object are placed in the two cells of a **GridLayout** having two rows and one column. Therefore, 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.

The panel that is instantiated in Listing 14 is placed in the CENTER of the **JFrame** object. This led to the name **centerControlPanel** for the reference variable that refers to this object.

Instantiate the green and yellow panels

The code in Listing 15 instantiates and sets the background colors on the green and yellow **JPanel** objects discussed above.

```
    //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
radio
    // buttons to the larger
centerControlPanel.
    centerControlPanel.add(zeroPanel);
    centerControlPanel.add(polePanel);
```

Listing 15

Listing 15 also adds the green and yellow panels to the **centerControlPanel** that was instantiated in Listing 14. The green panel containing the zero data is added to the top cell in the grid layout. The yellow panel containing the pole data is added to the bottom cell in the grid layout.

Add the control panel to the center of the JFrame object

Listing 16 adds the **centerControlPanel** that was instantiated in Listing 14 to the center of the main **JFrame** object that was instantiated in Listing 10.

```
    //Add the centerControlPanel to the CENTER
of the
    // JFrame object.
    guiFrame.getContentPane().add(

centerControlPanel,BorderLayout.CENTER);
```

Listing 16

At this point, the GUI is coming along pretty well, but it still doesn't contain the large number of required radio buttons and text fields. Also, some very important listener objects haven't been instantiated and registered yet.

A TextListener object

Listing 17 instantiates a text listener object of the **MyTextListener** class that will be registered on each of the text fields containing real and imaginary values. Each pair of such text fields specifies the location of one pole or one zero. I will present and explain the listener class named **MyTextListener** from which this object is instantiated in the next installment of this multi-part tutorial.

```
    MyTextListener textListener = new
MyTextListener();
```

Listing 17

Prepare the yellow panel to be populated

The code in Listing 18 prepares the yellow **polePanel** *(see Figure 3)* that was instantiated in Listing 15 to be populated with labels, radio buttons, and text fields. Listing 18 also begins the population process.

```
    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"));
```

Listing 18

As you can see, the layout manager for the yellow **polePanel** is set to a **GridLayout** with an unspecified number of rows and five columns. *(Setting the number of rows to 0 causes the number of rows to be unspecified.)*

Then Listing 18 adds five labels to the first row in the grid. These labels correspond to the text showing from left to right in the yellow row in Figure 3. This row appears to be yellow because the **JLabel** objects are transparent by default allowing the background color of the yellow **polePanel** to show through.

Some very potent *for* loops

The next several code fragments will present two very important **for** loops. A great deal is accomplished in each of the loops. The first of the loops is presented in Listing 19.

Instantiate the radio buttons

The code in the **for** loops begins by instantiating radio buttons that will be needed later and saving the references to those radio buttons in array objects that were instantiated in Listing 8. The code places all of the radio buttons associated with both the poles and the zeros in the same **ButtonGroup** object *(instantiated in Listing 2)* to cause them to behave in a mutually exclusive way.

Instantiate text fields

The code in the **for** loops instantiates **TextField** objects that will be needed later and saves their references in array objects that were instantiated in Listing 9.

Place radio buttons and text fields in the GUI

The code in the **for** loops places the radio buttons and the text fields in the cells on the green and yellow panels shown in Figure 3. These panels were instantiated in Listing 15, and the layout manager on the yellow **polePanel** was set to **GridLayout** in Listing 18. *(The layout manager will be set on the green panel later.)*

Disable *Angle* and *Length* text fields

The code in the **for** loops disables the text fields in the *Angle* and *Length* columns in Figure 3 to prevent the user from being able to enter data into the text fields in those columns. As a result, the text fields in those columns are used for display purposes only.

> *(Note that disabling the text fields does not prevent the program from writing text into those text fields. The text fields are disabled only insofar as manual input by the user is concerned.)*

Set *name* property values

The code in the **for** loops sets the **name** property for each of the **TextField** objects that contain real and imaginary data to a unique property value.

For example, the property values for the text fields containing the real values for the poles are set to the *string "poleReal"* with a numeric index concatenated onto the end of the string *(poleReal0 and poleReal5 for example).*

Similarly, the property values for the text fields containing the imaginary values for the poles are set to the *string "poleImag"* with a numeric index concatenated onto the end of the string.

These property values will be used later by a text listener object to determine which text field fired a text event. The numeric indices for the property values correspond to the numeric labels on the radio buttons in Figure 3.

Register a TextListener on the text fields

Finally, the code in the **for** loops registers the single **TextListener** object that was instantiated in Listing 17 on every text field containing real or imaginary values for poles or zeros. This listener object causes the angle and the length values to be computed and displayed when a corresponding text value changes for any reason. The listener object also causes the new locations of the poles and zeros to be reflected in the graphic display of the z-plane shown in Figure 2.

Enough talk, let's see a *for* loop

Listing 19 contains a **for** loop, which in turn contains the code that takes 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
```

Listing 19

Deal with the zeros

The code in Listing 20 begins by setting the **GridLayout** manager on the green **zeroPanel** that contains the radio buttons and the text fields for the zeros.

Then it adds the labels that produce the column headers shown in the green row in Figure 3.

Finally, Listing 20 contains a **for** loop, which in turn contains the code that takes the actions described above with respect to 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
```

Listing 20

Instantiate the z-plane object

Listing 21 instantiates the z-plane GUI object containing the unit circle shown in Figure 2. As you will recall from earlier discussions, the user locates poles and zeros on the z-plane display by first selecting the radio button in Figure 3 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();
```

Listing 21

Register an anonymous MouseListener on the z-plane object

Listing 22 shows the beginning of the code that registers an anonymous **MouseListener** object on the z-plane shown in Figure 2. This listener object is notified whenever the z-plane fires a **MouseEvent** of the **mousePressed** variety, causing the **mousePressed** method to be executed.

```
    refToZPlane.addMouseListener(
      new MouseAdapter(){
        public void mousePressed(MouseEvent e){
          int realCoor = e.getX() - 
refToZPlane.

translateOffsetHoriz;
          int imagCoor = -(e.getY() - 
refToZPlane.

translateOffsetVert);
```

Listing 22

Get and save the coordinates of the mouse click

The **mousePressed** method in Listing 22 begins by getting and saving the coordinates of the mouse click relative to an origin 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 values

The code in Listing 22 changes the sign on the vertical coordinate values to cause the results to match our expectation of positive vertical values going up the screen instead of going down the screen, which is the default.

Deposit the coordinate values in the text fields

The new coordinate values will be deposited in the real and imaginary text fields shown in Figure 3 that are associated with the selected radio button.

The code to accomplish that examines the radio buttons to identify the pair of real and imaginary text fields into which the new coordinate values should be deposited.

> *(Caution: 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.)*

The integer coordinate values are converted to fractional coordinate values by dividing the integer coordinate values *(in pixels)* by the radius *(in pixels)* of the unit circle as displayed on the z-plane.

Examine the pole buttons first

The code in Listing 23 cycles through the pole buttons in the yellow **polePanel** in Figure 3 to determine if any of those radio buttons have been selected.

```
          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
```

Listing 23

If a pole button has been selected...

If one of the pole buttons is determined to have been selected when the mouse event was fired by the z-plane object, the code in Listing 23:

- Sets the real and imaginary coordinate values into the corresponding text fields in Figure 3.
- Sets the **boolean** variable named **selectedFlag** to true to prevent the zero buttons from being examined.
- Breaks out of the **for** loop.

If none of the pole buttons is determined to have been selected when the event was fired, control exits the **for** loop with the **selectedFlag** variable set to false.

Possibly examine the zero buttons

If the **selectedFlag** variable is false at this point, the code in the body of the **for** loop in Listing 24 is executed. Otherwise, the code in the body of that **for** loop is skipped.

```
          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
```

Listing 24

If the code in the body of the **for** loop in Listing 24 is executed, it behaves essentially as described above for the **for** loop in Listing 23. However, in this case the code is dealing with the radio buttons and the text fields that are associated with the *zeros* in the green **zeroPanel** at the top of Figure 3 instead of the radio buttons and the text fields that are associated with the *poles* in the yellow **polePanel** at the bottom in Figure 3.

Repaint the z-plane object

Regardless of whether the mouse click was associated with a pole or with a zero, the code in Listing 25 invokes the **repaint** method on the z-plane object causing the entire display shown in Figure 2 to be repainted. This causes the pole or the zero that was affected by the mouse click to be drawn in the new location with all other poles and zeros being drawn in their previous locations. Exactly how that happens will become clearer when I explain the inner class named **ZPlane** in the next installment of this multi-part tutorial.

```
          refToZPlane.repaint();
        }//end mousePressed
      }//end new class
    );//end addMouseListener
```

Listing 25

Listing 25 also signals the end of the **mousePressed** method as well as the end of the definition of the anonymous listener class and object that is registered on the z-plane object.

Miscellaneous housekeeping operations

The code in Listing 26 performs some miscellaneous housekeeping operations to complete the constructor for the **InputGUI** class.

```
    guiFrame.setBounds(1024-472,0,472,400);

    guiFrame.setResizable(false);
    guiFrame.setVisible(true);
    refToZPlane.setResizable(false);
    refToZPlane.setVisible(true);

    InputGUI.refToObj = this;
  }//end constructor
```

Listing 26

The housekeeping operations performed by the code in Listing 26 are:

- Set the size and location of the **InputGUI** *(JFrame)* object on the screen. Position it in the upper right corner of a 1024x768 screen.
- Cause the input GUI and z-plane displays to become visible. Prevent the user from resizing them.
- Save the reference to this input GUI object in the static variable that was declared in Listing 1 so that it can be recovered later when a new instance of the **Dsp046** class is instantiated.

That completes the discussion and explanation of the constructor for the **InputGUI** class.

The captureTextFieldData method

This is a utility method that is used to capture the latest text field data, convert it into type **double**, and store the numeric values into array objects.

The try-catch handlers in this method are designed to deal with the possibility that a text field contains a text value that cannot be converted to a numeric **double** value when the method is invoked. In that case, the text value is replaced by the string "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 into the text field without throwing an exception 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 while still causing the angle and the length to be automatically computed and displayed each time a new value is entered into the text field.

The code in the captureTextFieldData method

The **captureTextFieldData** method begins in Listing 27.

```
  void captureTextFieldData(){
    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
```

Listing 27

A loop to process all of the text values

Listing 27 contains a **for** loop that:

- Cycles through the text representations of the real and imaginary values for each of the poles.
- Attempts to convert that data from type **String** into the numeric type **double**.
- Encapsulates the numeric values into the elements of an array object.

The **parseDouble** method will throw a **NumberFormatException** if the text that is extracted from a text field cannot be successfully converted to a numeric type **double**. In that case, the catch block will be executed, storing the string "0" in the text field and displaying an error message on the command-line screen.

Process the zero data

Listing 28 contains a **for** loop that processes the zero data in a manner that is identical to that described for the poles in Listing 27.

```
    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
```

Listing 28

Listing 28 also signals the end of the method named **captureTextFieldData**.

Conclusion

That concludes the explanation of the code in the third installment of this multi-part tutorial. There are two major inner classes contained in the program that haven't been explained in the first three installments.

- MyTextListener
- ZPlane

Those two classes will be explained in the fourth and final installment.

Run the Program

I encourage you to copy the code from Listing 29 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 29, 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 for links to those lessons)*:

- 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

The earlier tutorial lesson entitled A Recursive Filtering Workbench in Java was the first installment of a multi-part tutorial on recursive filtering. That lesson 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.

The second installment, entitled The Driver Class for the Recursive Filtering Workbench in Java, presented and explained the driver class for the workbench. *(The driver class is named **Dsp046.**)*

This third installment has presented and explained the class named **InputGUI**, from which one of three major GUIs is constructed. This GUI, which is shown in the upper right corner of Figure 1 deals mainly with text input and text output.

A second GUI, shown in the upper left corner of Figure 1 deals mainly with the presentation of graphic data to the user. It was explained in the second installment.

The third GUI, shown in the top center of Figure 1 deals mainly with mouse input and program output in a graphic context. The code for that GUI will be explained in the fourth and final installment.

What's Next?

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

1. **Dsp046 -** Driver class for the workbench *(explained in the second installment)*.
2. **InputGUI** - Provides user input capability *(mainly text)* for the workbench as shown in Figure 3 *(explained in this installment)*.
3. **MyTextListener -** An inner class that processes text input to the workbench, updating the angle output and updating the z-plane display.
4. **ZPlane** - An inner class that provides the z-plane display along with graphic mouse input capability for the workbench as shown in Figure 2.

In addition, an understanding of this program requires an understanding of the following Java classes, which were presented and explained in earlier lessons *(links to these lessons can be found in the References section of this lesson):*

- **ForwardRealToComplexFFT01** - Used to perform an FFT on the impulse response.
- **Graph03** - Plotting program as shown in Figure 4.
- **GraphIntfc01** - Required to use **Graph03** for plotting.
- **GUI** - Required to use **Graph03** for plotting.
- **GUI$MyCanvas** - Required to use **Graph03** for plotting.

The second installment of this multi-part tutorial explained the driver class named **Dsp046**.

This third installment has presented and explained the class named **InputGUI**.

The fourth and final installment will present and explain the inner classes named **MyTextListener** and **ZPlane**.

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:

- 1510 A Recursive Filtering Workbench in Java
- 1511 The Driver Class for the Recursive Filtering Workbench in Java
- 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

Complete listings of the classes discussed in this lesson are contained in Listing 29 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 multiplied in pairs to produce two eighth-order
    // polynomials.  The two eighth-order polynomials are
    // multiplied to produce one sixteenth-order
    // polynomial.
    //The algebraic 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 multiplication,
    // 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 multiplication,
    // 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
    // horizontal 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 eighthToSixteenthOrder
//-----------------------------------------------------//

//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 mutually
  // 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 origin 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 29

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" graph plot pole zero z-plane GUI "unit circle" conjugate "radio button" "text field" real imaginary

-end-