Processing Image Pixels, Applying Image Convolution in Java, Part 2

Part 2 of this lesson teaches you how to design copying filters, smoothing filters, sharpening filters, 3D embossing filters, and edge detection filters, and how to apply those filters to images.

Published: April 4, 2006
By Richard G. Baldwin

Java Programming, Notes # 414

Preface

Part of a series

This lesson is one in a series designed to teach you how to use Java to create special effects with images by directly manipulating the pixels in the images. The first lesson in the series was entitled Processing Image Pixels using Java, Getting Started. The previous lesson was Part 1 of this two-part lesson.

This is Part 2 of the two-part lesson. You are strongly encouraged to review the first part of this lesson entitled Processing Image Pixels, Applying Image Convolution in Java, Part 1 before continuing with this lesson.

The primary objective of this lesson is to teach you how to integrate much of what you have already learned about Digital Signal Processing *(DSP)* and Image Convolution into Java programs that can be used to experiment with, and to understand the effects of a wide variety of image-convolution operations.

Part 1 of this lesson showed you how to design copying filters, smoothing filters, sharpening filters, 3D embossing filters, and edge detection filters, and how to apply those filters to images. The results of numerous experiments using filters of the types listed above were presented. This part of the lesson explains the code required to perform the experiments that were presented in Part 1.

You will need a driver program

The lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness provided and explained a class named **ImgMod02a** that makes it easy to:

- Manipulate and modify the pixels that belong to an image.
- Display the processed image along with the original image.

ImgMod02a serves as a driver that controls the execution of a second program that actually processes the pixels. It displays the original and processed images in the standard format shown in Figure 1.

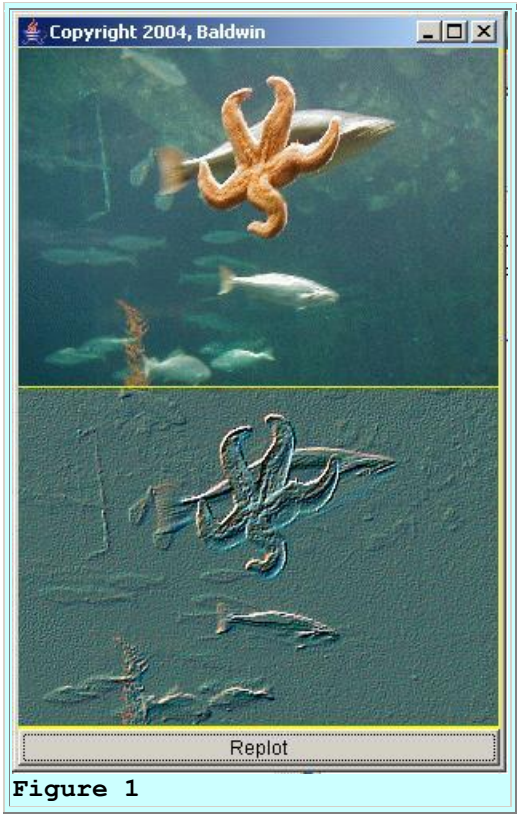

Figure 1

Get the class and the interface

The image-processing programs that I will explain in this lesson run under the control of **ImgMod02a**. In order to compile and run the programs that I will provide in this lesson, you will need to go to the lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started to get copies of the class named **ImgMod02a** and the interface named **ImgIntfc02**

Viewing tip

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

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

I particularly recommend that you study the lessons referred to in the References section of this lesson.

Background Information

A three-dimensional array of pixel data as type int

The driver program named **ImgMod02a**:

- Extracts the pixels from an image file.
- Converts the pixel data to type **int.**
- Stores the pixel data in a three-dimensional array of type **int** that is well suited for processing.
- Passes the three-dimensional array object's reference to a method in an object instantiated from an image-processing class.
- Receives a reference to a three-dimensional array object containing processed pixel data from the image-processing method.
- Displays the original image and the processed image in a stacked display as shown in Figure 1.
- Makes it possible for the user to provide new input data to the image-processing method, invoking the image-processing method repeatedly in order to create new displays showing the newly-processed image along with the original image.

The manner in which that is accomplished was explained in earlier lessons.

A grid of colored pixels

Each three-dimensional array object represents one image consisting of a grid of colored pixels. The pixels in the grid are arranged in rows and columns when they are rendered. One of the dimensions of the array represents rows. A second dimension represents columns. The third dimension represents the color *(and transparency)* of the pixels.

Convolution in one dimension

The earlier lesson entitled Convolution and Frequency Filtering in Java taught you about performing convolution in one dimension. In that lesson, I showed you how to apply a convolution filter to a sampled time series in one dimension. As you may recall, the mathematical process in one dimension involves the following steps:

- Register the n-point convolution filter with the first **n** samples in the time series.
- Compute an output value, which is the sum of the products of the convolution filter coefficient values and the corresponding time series values.
- Optionally divide the sum of products output value by the number of filter coefficients.
- Move the convolution filter one step forward, registering it with the next **n** samples in the time series and compute the next output value as a sum of products.
- Repeat this process until all samples in the time series have been processed.

Convolution in two dimensions

Convolution in two dimensions involves essentially the same steps except that in this case we are dealing with three different 3D sampled surfaces and a 3D convolution filter surface instead of a simple sampled time series.

> *(There is a red surface, a green surface, and a blue surface, each of which must be processed. Each surface has width and height corresponding to the first two dimensions of the 3D surface. In addition, each sampled value that represents the surface can be different. This constitutes the third dimension of the surface. There is also an alpha or transparency surface that could be processed, but the programs in this lesson don't process the alpha surface. Similarly, the convolution filter surface has three dimensions corresponding to width, height, and the values of the coefficients in the operator. Don't be confused by the dimensions of the array object containing the surface or the convolution filter and the dimensions of the surface or the convolution filter.)*

Steps in the processing

Basically, the steps involved in processing one of the three surfaces to produce one output surface consist of:

- Register the 2D aspect *(width and height)* of the convolution filter with the first 2D area centered on the first row of samples on the input surface.
- Compute a point for the output surface, by computing the sum of the products of the convolution filter values and the corresponding input surface values.
- Optionally divide the sum of products output value by the number of filter coefficients.
- Move the convolution filter one step forward along the row, registering it with the next 2D area on the surface and compute the next point on the output surface as a sum of products. When that row has been completely processed, move the convolution filter to the beginning of the next row, registering with the corresponding 2D area on the input surface and compute the next point for the output surface.
- Repeat this process until all samples in the surface have been processed.

Repeat once for each color surface

Repeat the above set of steps three times, once for each of the three color surfaces.

Watch out for the edges

Special care must be taken to avoid having the edges of the convolution filter extend outside the boundaries of the input surface.

Testing

All of the code in this lesson was tested using J2SE 5.0 and WinXP

Discussion and Sample Code

There are a rather large number of classes involved in producing the experimental results described in the first part of this lesson, which was entitled Processing Image Pixels, Applying Image Convolution in Java, Part 1.

Some of those classes have already been discussed in detail in earlier lessons. For retrieval of the source code for those classes, I will simply refer you to the earlier lessons referred to in the References section of this lesson.

> *(As an alternative, you can probably just go to Google and enter the name of the class along with the keywords Baldwin and java and find the lessons online.)*

Some of the classes are updated versions of classes discussed in earlier lessons. In those cases, I will provide some, but not very much discussion of the new version of the class. For the most part, I will simply refer you to the lesson containing the earlier version for the discussion.

Some of the classes are new to this lesson. I will discuss and explain those classes in detail.

The class named Graph08

I will begin with the class named **Graph08**. A complete listing of this class is provided in Listing 28 near the end of this lesson.

This is an updated version of the earlier plotting class named **Graph03**. The update allows the user to plot up to eight functions instead of only 5 as is the case with **Graph03**.

> *(Figure 2 shows a sample of the plotting format produced by the class named **Graph08**.)*

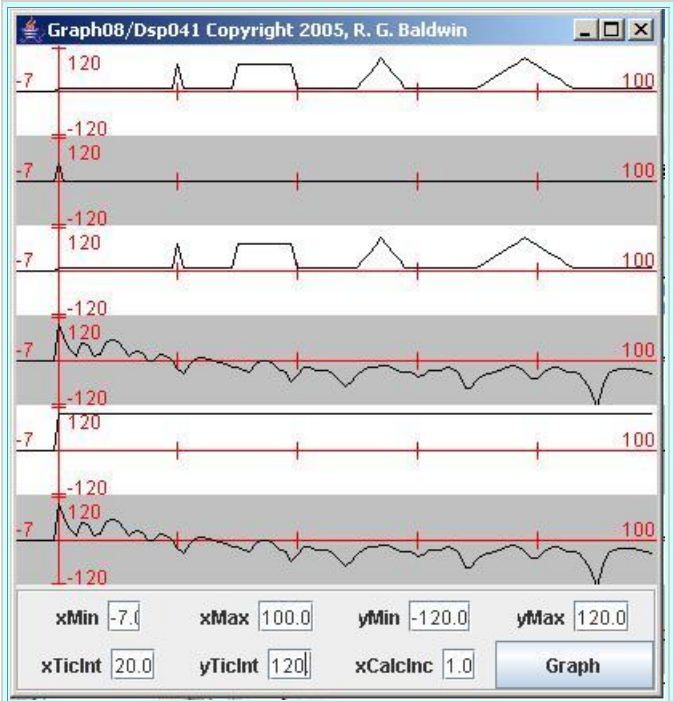

Figure 2

The use of this class requires the use of a corresponding interface named. **GraphIntfc08**. This interface, which is provided in Listing 29, is an update to the earlier interface named **GraphIntfc01**.

Graph03 and **GraphIntfc01** were explained in the earlier lesson entitled Convolution and Matched Filtering in Java. Because of the similarity between **Graph08** and **Graph03**, I will simply refer you to that explanation, and won't repeat the explanation here.

The Dsp041 class

This class is new to this lesson. A detailed description of the class was provided in Processing Image Pixels, Applying Image Convolution in Java, Part 1 of this lesson in the section entitled Preview.

This class must be run under control of the class named **Graph08**. To run this class as a program, enter the following command at the command line prompt:

java Graph08 Dsp041

I will explain this class in fragments. The source code for the class is provided in its entirety in Listing 30 near the end of the lesson.

This class illustrates the application of a convolution filter to signals having a known waveform displaying the results in the format shown in Figure 2.

The class definition

The class definition for the class named **Dsp041** begins in Listing 1.

```
class Dsp041 implements GraphIntfc08{
  //Establish length for various arrays
  int filterLen = 200;
  int signalLen = 400;
  int outputLen = signalLen - filterLen;
  //Ignore right half of signal, which is all
zeros, when
  // computing the spectrum.
  int signalSpectrumPts = signalLen/2;
  int filterSpectrumPts = outputLen;
  int outputSpectrumPts = outputLen;


  //Create arrays to store different types of
data.
  double[] signal = new double[signalLen];
  double[] filter = new double[filterLen];
  double[] output = new double[outputLen];
  double[] spectrumA = new
double[signalSpectrumPts];
  double[] spectrumB = new
double[filterSpectrumPts];
  double[] spectrumC = new
double[outputSpectrumPts];

Listing 1
```

Note that the class implements the interface named **GraphIntfc08**. This is a requirement for any class that is to be run under control of the class named **Graph08**.

The code in Listing 1 declares variables and creates array objects. The code is straightforward and shouldn't require further explanation.

The constructor for the class named Dsp041

The constructor begins in Listing 2.

```
  public Dsp041(){//constructor

    //This is a single impulse filter that
simply copies
    // the input to the output.
    filter[0] = 1;

/*
    //This is a high-pass filter with an output
that is
    // proportional to the slope of the
```

```
signal.  In
    // essence,the output approximates the
first derivative
    // of the signal.
    filter[0] = -1.0;
    filter[1] = 1.0;

    //This is a high-pass filter with an output
that is
    // proportional to the rate of change of
the slope of
    // the signal. In essence, the output
approximates the
    // second derivative of the signal.
    filter[0] = -0.5;
    filter[1] = 1.0;
    filter[2] = -0.5;

    //This is a relatively soft high-pass
filter, which
    // produces a little blip in the output
each time the
    // slope of the signal changes.  The size
of the blip
    // is roughly proportional to the rate of
change of the
    // slope of the signal.
    filter[0] = -0.2;
    filter[1] = 1.0;
    filter[2] = -0.2;

    //This is a low-pass smoothing filter.  It
approximates
    // a four-point running average or
integration of the
    // signal.
    filter[0] = 0.250;
    filter[1] = 0.250;
    filter[2] = 0.250;
    filter[3] = 0.250;
*/
```

Listing 2

Enable and disable code

An object of the class named **Dsp041** applies a one-dimensional convolution filter to a signal with a known waveform and displays the results in the format shown in Figure 2. By enabling and disabling code using comments, you can create and save a convolution filter having given set of coefficient values. Then you can recompile the class and rerun the program to see the effect of the filter on the signal.

Several predefined filter waveforms are provided in Listing 2. Obviously, you can modify the code in Listing 2 to create a filter of your own design.

Create a signal time series

Listing 3 creates a signal time series containing four distinct waveforms. These waveforms consist of all positive values riding on a positive non-zero baseline:

- An impulse.
- A rectangular pulse.
- A triangular pulse with a large slope.
- A triangular pulse with a smaller slope.

```
    //First create a baseline in the signal
time series.
    //Modify the following value and recompile
the class
    // to change the baseline.
    double baseline = 10.0;
    for(int cnt = 0;cnt < signalLen;cnt++){
      signal[cnt] = baseline;
    }//end for loop

    //Now add the pulses to the signal time
series.

    //First add an impulse.
    signal[20] = 75;

    //Add a rectangular pulse.
    signal[30] = 75;
    signal[31] = 75;
    signal[32] = 75;
    signal[33] = 75;
    signal[34] = 75;
    signal[35] = 75;
    signal[36] = 75;
    signal[37] = 75;
    signal[38] = 75;
    signal[39] = 75;

    //Add a triangular pulse with a large
slope.
    signal[50] = 10;
    signal[51] = 30;
    signal[52] = 50;
    signal[53] = 70;
    signal[54] = 90;
    signal[55] = 70;
    signal[56] = 50;
    signal[57] = 30;
    signal[58] = 10;

    //Add a triangular pulse with a smaller
slope.
    signal[70] = 10;
    signal[71] = 20;
```

```
    signal[72] = 30;
    signal[73] = 40;
    signal[74] = 50;
    signal[75] = 60;
    signal[76] = 70;
    signal[77] = 80;
    signal[78] = 90;
    signal[79] = 80;
    signal[80] = 70;
    signal[81] = 60;
    signal[82] = 50;
    signal[83] = 40;
    signal[84] = 30;
    signal[85] = 20;
    signal[86] = 10;

Listing 3
```

The signal created by the code in Listing 3 is shown in the first graph at the top of Figure 2.

Obviously, you could replace the waveforms created by Listing 3 with signal waveforms of your own design if you elect to do so. Just remember that if you are using this program to investigate the application of convolution to image color data, all color values in an image are positive.

Apply the convolution filter to the signal

Listing 4 invokes the method named **convolve** to apply the convolution filter to the signal.

```
    convolve(signal,filter,output);

Listing 4
```

You will find a complete listing of the method named **convolve** in Listing 30.

The code in the method named **convolve** emulates a one-dimensional version of the 2D image convolution scheme used in **ImgMod032** with respect to normalization and scaling. That normalization scheme was explained in an earlier lesson entitled Processing Image Pixels, Understanding Image Convolution in Java, and is also explained in detail in the comments in Listing 30. While the normalization process is rather long and tedious, it is also straightforward and shouldn't require an explanation beyond the comments in Listing 30.

Aside from the normalization and scaling code, the actual convolution process implemented in the method named **convolve** has been explained in earlier lessons referred to in the References section. Therefore, it shouldn't be necessary to provide a detailed explanation of the method named **convolve**.

Compute Discrete Fourier Transform of signal

Listing 5 invokes the method named **dft** to compute and save the Discrete Fourier Transform *(DFT)* of the signal expressed in decibels.

```
    //Ignore right half of signal which is all
zeros.
    dft(signal,signalSpectrumPts,spectrumA);

Listing 5
```

The computation of the DFT has been explained in several earlier lessons explained in the References section. Also, there are extensive comments provided with the **dft** method in Listing 30. Therefore, I won't repeat that explanation here.

The results of the DFT computation on the signal are shown in the fourth graph in Figure 2.

Compute the amplitude frequency response of the filter

Listing 6 computes and saves the DFT of the convolution filter expressed in db.

```
    dft(filter,filterSpectrumPts,spectrumB);

Listing 6
```

Note that the convolution filter is embedded in a long time series having zero values. This causes the output of the DFT to be finely sampled and produces a smooth curve for the frequency response of the convolution filter.

The results of the DFT computation on the convolution filter are shown in the fifth graph in Figure 2.

Compute the spectrum of the filtered output

Listing 7 computes and saves the DFT of the filtered output expressed in decibels.

```
    dft(output,outputSpectrumPts,spectrumC);
  }//end constructor

Listing 7
```

The results of the DFT computation on the filtered output are shown in the sixth graph in Figure 2.

Listing 7 also signals the end of the constructor.

Plot the results

All of the time series and frequency domain functions have now been produced and saved. They may be retrieved and plotted by invoking the method named **getNmbr** and the methods named **f1** through **f6** shown in Listing 30. These methods are invoked by the object of the class named **Graph08** for the purpose of producing the graphic output in the format shown in Figure 2.

The purpose of these methods is simply to scale and return the data to be plotted. They are straightforward and shouldn't require an explanation beyond the comments provided in Listing 30.

The ImgMod33 class and the ImgMod33a class

These two classes are new to this lesson. A detailed description of the two classes was provided in the Preview section of Part 1 of this lesson.

It is recommended that you read the material in that section before attempting to understand the program code in this section.

Program listings

A complete listing of the class named **ImgMod33** is provided in Listing 31. A complete listing of the class named **ImgMod33a** is provided in Listing 32.

The two classes are very similar

The two classes are the same except that **ImgMod33a** uses the class named **ImgMod32a** to perform the 2D convolution whereas **ImgMod33** uses the class named **ImgMod32** to perform the 2D convolution. As a result, I will explain **ImgMod33**, but will not explain **ImgMod33a**. However, I will explain the differences between **ImgMod32** and **ImgMod32a** later in the lesson.

A general purpose 2D image convolution capability

Each of these classes provides a general purpose 2D image convolution and color filtering capability in Java, wherein the convolution filter is provided to the program by way of a text file.

Running the program

Both classes are designed to be driven by the class named **ImgMod02a**.

> *(The class named **ImgMod02a** was explained in the earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness.)*

The image file to be processed by convolution is specified on the command line. Convolution filters are provided as text files.

Enter one of the following at the command line to run one of these programs *(where **ImageFileName** is the name of a .gif or .jpg file, including the extension):*

```
java ImgMod02a ImgMod33 ImageFileName
java ImgMod02a ImgMod33a ImageFileName
```

Specify the filter file

Then enter the name of a file containing a 2D convolution filter in the **TextField** that appears in the interactive control panel shown in Figure 3.

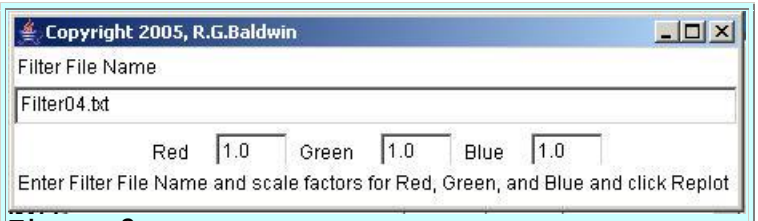

Figure 3

Click the **Replot** button on the **Frame** shown in Figure 1 to cause the convolution filter to be applied to the image and the filtered result to be displayed.

> *(See Figure 1 for an example of the Frame containing the original image, the processed image, and the Replot button. See comments at the beginning of the method named **getFilter** in the class named **ImgMod33** for a description and an example of the required format for the file containing the 2D convolution filter.)*

Wave number data

Each time you click the **Replot** button, the following additional information is displayed in two separate color contour maps in the format shown in Figure 4:

- The convolution filter.
- The wave number response of the convolution filter.

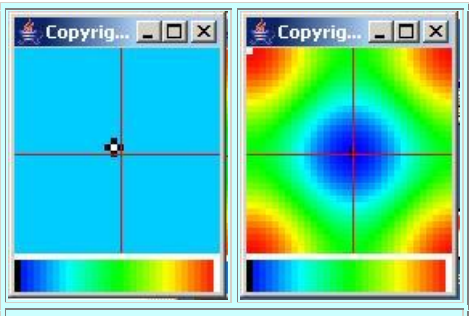

Figure 4

(Note that the two contour maps do not appear side-by-side as shown in Figure 4. Rather, they appear on top of one another. You must move the one on the top to see the one on the bottom.)

The class definition

The class definition for the class named **ImgMod33** begins in Listing 8. Note that the class extends **Frame** and implements **ImgIntfc02**.

```
class ImgMod33 extends Frame implements
ImgIntfc02{

  TextField fileNameField = new TextField("");
  Panel rgbPanel = new Panel();
  TextField redField = new TextField("1.0");
  TextField greenField = new TextField("1.0");
  TextField blueField = new TextField("1.0");
  Label instructions = new Label(
          "Enter Filter File Name and scale
factors for " +
          "Red, Green, and Blue and click
Replot");

Listing 8
```

The class extends **Frame** so that an object of the class serves as the interactive control panel shown in Figure 3. The class implements the interface named **ImgIntfc02** to make it possible for the class to run under the control of the class named **ImgMod02a** and to display the modified image in the format shown in Figure 1.

The code in Listing 8 simply declares and initializes some instance variables.

The constructor

The constructor for the class named **ImgMod33** is shown in its entirety in Listing 9.

```
  ImgMod33(){//constructor
    setLayout(new GridLayout(4,1));
    add(new Label("Filter File Name"));
    add(fileNameField);

    //Populate the rgbPanel
    rgbPanel.add(new Label("Red"));
    rgbPanel.add(redField);
    rgbPanel.add(new Label("Green"));
    rgbPanel.add(greenField);
    rgbPanel.add(new Label("Blue"));
    rgbPanel.add(blueField);

    add(rgbPanel);
    add(instructions);
```

```
    setTitle("Copyright 2005, R.G.Baldwin");
    setBounds(400,0,460,125);
    setVisible(true);
  }//end constructor

Listing 9
```

The code in the constructor constructs the interactive control panel shown in Figure 3.

The method named processImg

The **processImg** method begins in Listing 10.

```
  public int[][][] processImg(int[][][]
threeDPix,
                              int imgRows,
                              int imgCols){

    //Create an empty output array of the same
size as the
    // incoming array.
    int[][][] output = new
int[imgRows][imgCols][4];

Listing 10
```

The **processImg** method must be defined by all classes that implement **ImgIntfc02**. The method is called at the beginning of the run and each time thereafter that the user clicks the **Replot** button on the **Frame** shown in Figure 1.

The **processImg** method gets a 2D convolution filter from a text file, applies it to the incoming 3D array of pixel data and returns a filtered 3D array of pixel data.

The code in Listing 10 creates an output array object in which to return the filtered image data.

Make a working copy

Listing 11 makes a working copy of the 3D pixel array to avoid making permanent changes to the original image data.

```
    int[][][] working3D = new
int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        working3D[row][col][0] =
threeDPix[row][col][0];
        working3D[row][col][1] =
threeDPix[row][col][1];
        working3D[row][col][2] =
threeDPix[row][col][2];
```

```
        working3D[row][col][3] =
threeDPix[row][col][3];
        //Copy alpha values directly to the
output. They
        // are not processed when the image is
filtered
        // by the convolution filter.
        output[row][col][0] =
threeDPix[row][col][0];
      }//end inner loop
    }//end outer loop

Listing 11
```

Get the convolution filter from the file

Listing 12 gets the convolution filter from the specified text file containing the filter.

```
    //Get the file name containing the filter
from the
    // textfield.
    String fileName = fileNameField.getText();
    if(fileName.equals("")){
      //The file name is an empty string. Skip
the
      // convolution process and pass the input
image
      // directly to the output.
      output = working3D;
    }else{
      //Get a 2D array that is populated with
the contents
      // of the file containing the 2D filter.
      double[][] filter = getFilter(fileName);

Listing 12
```

Listing 12 gets the name of the file containing the convolution filter from the interactive control panel shown in Figure 3. If the **TextField** contains an empty string *(as is the case the first time the **processImg** method is called)* the convolution process is skipped and the input image is passed directly to the output.

If the user later enters a valid file name into the **TextField** and clicks the **Replot** button, causing the **processImg** method to be called again, Listing 12 invokes the method named **getFilter** to read the convolution filter from the file and to deposit it into the array object referred to by **filter**.

The getFilter method

The **getFilter** method is shown in its entirety in Listing 31. Although the method is rather long, it is straightforward and shouldn't require an explanation beyond the comments provided in the

listing. The comments also describe the required format for the text file containing the filter coefficients.

Plot impulse response and wave number response

Listing 13 plots the impulse response and wave number response of the 2D convolution filter as a pair of colored contour maps.

```
      //Plot the impulse response and the wave-number
      // response of the convolution filter.  These items
      // are not computed and plotted when the program
      // starts running.  Rather, they are computed and
      // plotted each time the user clicks the Replot
      // button after entering the name of a file
      // containing a convolution filter into the
      // TextField.

      //Begin by placing the impulse response in the
      // center of a large flat surface with an elevation
      // of zero.This is done to improve the resolution of
      // the Fourier Transform, which will be computed
      // later.
      int numFilterRows = filter.length;
      int numFilterCols = filter[0].length;
      int rows = 0;
      int cols = 0;
      //Make the size of the surface ten pixels larger than
      // the convolution filter with a minimum size of
      // 32x32 pixels.
      if(numFilterRows < 22){
        rows = 32;
      }else{
        rows = numFilterRows + 10;
      }//end else
      if(numFilterCols < 22){
        cols = 32;
      }else{
        cols = numFilterCols + 10;
      }//end else

      //Create the surface, which will be initialized to
      // all zero values.
      double[][] filterSurface = new double[rows][cols];
      //Place the convolution filter in the center of the
      // surface.
      for(int row = 0;row < numFilterRows;row++){
        for(int col = 0;col < numFilterCols;col++){
          filterSurface[row + (rows - numFilterRows)/2]
                       [col + (cols - numFilterCols)/2] =
                                          filter[row][col];
        }//end inner loop
      }//end outer loop

      //Display the filter and the surface on which it
```

```
      // resides as a 3D plot in a color contour format.
      new ImgMod29(filterSurface,4,true,1);

      //Get and display the 2D Fourier Transform of the
      // convolution filter.

      //Prepare arrays to receive the results of the
      // Fourier transform.
      double[][] real = new double[rows][cols];
      double[][] imag = new double[rows][cols];
      double[][] amp = new double[rows][cols];
      //Perform the 2D Fourier transform.
      ImgMod30.xform2D(filterSurface,real,imag,amp);
      //Ignore the real and imaginary results.  Prepare the
      // amplitude spectrum for more-effective plotting by
      // shifting the origin to the center in wave-number
      // space.
      double[][] shiftedAmplitudeSpect =
                                 ImgMod30.shiftOrigin(amp);

      //Get and display the minimum and maximum wave number
      // values.  This is useful because the way that the
      // wave number plots are normalized. it is not
      // possible to infer the flatness or lack thereof of
      // the wave number surface simply by viewing the
      // plot.  The colors that describe the elevations
      // always range from black at the minimum to white at
      // the maximum, with different colors in between
      // regardless of the difference between the minimum
      // and the maximum.
      double maxValue = -Double.MAX_VALUE;
      double minValue = Double.MAX_VALUE;
      for(int row = 0;row < rows;row++){
        for(int col = 0;col < cols;col++){
          if(amp[row][col] > maxValue){
            maxValue = amp[row][col];
          }//end if
          if(amp[row][col] < minValue){
            minValue = amp[row][col];
          }//end if
        }//end inner loop
      }//end outer loop

      System.out.println("minValue: " + minValue);
      System.out.println("maxValue: " + maxValue);
      System.out.println("ratio: " + maxValue/minValue);

      //Generate and display the wave-number response
      // graph by plotting the 3D surface on the computer
      // screen.
      new ImgMod29(shiftedAmplitudeSpect,4,true,1);
```

Listing 13

I'm not going to tell you that the code in Listing 13 is straightforward. It isn't. However, I am going to tell you that everything that results from the code in Listing 13 has been explained in earlier lessons referred to in the References section of this lesson and I'm not going to repeat those explanations here. If you don't understand the code in Listing 13, you simply need to go back and study those earlier lessons.

Perform the convolution

Finally, we have reached the main objective of the class named **ImgMod33**. Listing 14 invokes the static **convolve** method of the **ImgMod32** class to convolve the image with the 2D convolution filter.

```
      output =
ImgMod32.convolve(working3D,filter);
    }//end else

Listing 14
```

The **convolve** method of the **ImgMod32** class was explained in detail in the earlier lesson entitled Processing Image Pixels, Understanding Image Convolution in Java. Therefore, I won't repeat that explanation here.

The important point is that this class named **ImgMod33** serves as a driver, reading 2D convolution filters from text files, applying the filters to images, and causing the raw and processed images to be displayed under control of the class named **ImgMod02a** in the format shown in Figure 1.

Listing 14 also signals the end of the **else** clause that began in Listing 12.

Changes for ImgMod33a

Listing 14 also shows the location of the difference between the classes named **ImgMod33** and **ImgMod33a**. Whereas the class named **ImgMod33** invokes the static **convolve** method belonging to the class named **ImgMod32**, the class named **ImgMod33a** invokes the **convolve** method belonging to the class named **ImgMod32a**.

These two **convolve** method differ in the way that they normalize the convolution output to produce the required image color format consisting of unsigned eight-bit values. I explained the normalization scheme implemented by **ImgMod32** in the earlier lesson entitled Processing Image Pixels, Understanding Image Convolution in Java. I will explain the normalization scheme implemented by the class named **ImgMod32a** later in this lesson.

Apply color filtering

Following the completion of the convolution operation, Listing 15 scales the color values in each color plane if the multiplicative factor in the corresponding **TextField** in the interactive control panel has a value other than 1.0.

```
    if(!redField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                       redField.getText());
      scaleColorPlane(output,1,scale);
    }//end if on redField

    if(!greenField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                     greenField.getText());
      scaleColorPlane(output,2,scale);
    }//end if on greenField

    if(!blueField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                      blueField.getText());
      scaleColorPlane(output,3,scale);
    }//end if on blueField

Listing 15
```

The scaling for each color plane is actually performed by invoking the method named **scaleColorPlane**.

The scaleColorPlane method

The **scaleColorPlane** method is shown in its entirety in Listing 31. The code in the method is straightforward and shouldn't require an explanation beyond the comments in the listing.

Return the processed image

Listing 16 returns a reference to the array containing the image, which has undergone convolution filtering, color filtering, or a combination of the two.

```
    return output;

  }//end processImg method

Listing 16
```

Listing 16 also signals the end of the **processImg** method, and the end of the **ImgMod33** class insofar as this explanation is concerned. Once again, however, the class contains a couple of additional methods, which do not need to be explained in detail in this lesson.

The class named ImgMod32a

The class named **ImgMod32a** is shown in its entirety in Listing 33. This class is similar to **ImgMod32** except that it uses a different normalization scheme when converting convolution results back to eight-bit unsigned values. The normalization scheme causes the mean and the RMS of the output to match the mean and the RMS of the input. Then it sets negative values to 0 and sets values greater than 255 to 255.

The class named **ImgMod32** was explained in detail in the earlier lesson entitled Processing Image Pixels, Understanding Image Convolution in Java. Much of the code in **ImgMod32a** is identical to the code in **ImgMod32**. Therefore, I won't repeat the explanation for that code. Rather, I will explain the code that differs between the two classes in the following sections.

The convolve method

All of the important changes were made to the static method named **convolve**. I will confine my explanation to that method. The static **convolve** method applies an incoming 2D convolution filter to each color plane in an incoming 3D array of pixel data and returns a filtered 3D array of pixel data.

The convolution filter is applied separately to each color plane.

The alpha plane is not modified.

Normalization

The output is normalized so as to guarantee that the output color values fall within the range from 0 to 255. This is accomplished by causing the mean and the RMS of the color values in each output color plane to match the mean and the RMS of the color values in the corresponding input color plane. Then, all negative color values are set to a value of 0 and all color values greater than 255 are set to 255.

Computations in type double

The convolution filter is passed to the method as a 2D array of type **double**. All convolution and normalization arithmetic is performed as type **double**. The normalized results are converted to type **int** before returning them to the calling method.

This method does not modify the contents of the incoming array of pixel data.

A dead zone around the perimeter of the image

An unfiltered dead zone equal to half the filter length is left around the perimeter of the filtered image to avoid any attempt to perform convolution using data outside the bounds of the image.

The code for the convolve method

The **convolve** method begins in Listing 17.

```
  public static int[][][] convolve(
                      int[][][]
threeDPix,double[][] filter){
    //Get the dimensions of the image and
filter arrays.
```

```
    int numImgRows = threeDPix.length;
    int numImgCols = threeDPix[0].length;
    int numFilRows = filter.length;
    int numFilCols = filter[0].length;

    //Display the dimensions of the image and
filter
    // arrays.
    System.out.println("numImgRows = " +
numImgRows);
    System.out.println("numImgCols = " +
numImgCols);
    System.out.println("numFilRows = " +
numFilRows);
    System.out.println("numFilCols = " +
numFilCols);

    //Make a working copy of the incoming 3D
pixel array to
    // avoid making permanent changes to the
original image
    // data. Convert the pixel data to type
double in the
    // process.  Will convert back to type int
when
    // returning from this method.
    double[][][] work3D =
intToDouble(threeDPix);
```

Listing 17

The code in Listing 17 is straightforward and shouldn't require further explanation.

Display the mean values

Listing 18 invokes the **getMean** method to get and to display the mean value for each incoming color plane.

```
    //Display the mean value for each color
plane.
    System.out.println(
                    "Input red mean: " +
getMean(work3D,1));
    System.out.println(
                   "Input green mean: " +
getMean(work3D,2));
    System.out.println(
                    "Input blue mean: " +
getMean(work3D,3));
```

Listing 18

The **getMean** method is shown in its entirety in Listing 33. The code in the method is straightforward and shouldn't require further explanation.

Remove the mean value

Listing 19 invokes the method named **removeMean** to get and save the mean value for each color plane, and to remove the mean from every color value in each color plane.

```
    double redMean = removeMean(work3D,1);
    double greenMean = removeMean(work3D,2);
    double blueMean = removeMean(work3D,3);

Listing 19
```

You can view the **removeMean** method in Listing 33. The code in the method is straightforward.

Get, save, and display the RMS value

The root mean square *(RMS)* value of a set of color values is simply the square root of the sum of the squares of all the color values.

Listing 20 invokes the **getRms** method to get and save the RMS value for the color values belonging to each color plane. These values will be used later to cause the RMS value for each color plane in the filtered image to match the RMS values for each color plane in the original image.

```
    //Get and save the input RMS value for
later
    // restoration.
    double inputRedRms = getRms(work3D,1);
    double inputGreenRms = getRms(work3D,2);
    double inputBlueRms = getRms(work3D,3);

    //Display the input RMS value
    System.out.println("Input red RMS: " +
inputRedRms);
    System.out.println(
                       "Input green RMS: " +
inputGreenRms);
    System.out.println("Input blue RMS: " +
inputBlueRms);

Listing 20
```

Once you know the definition of the RMS value, the code in the **getRms** method is straightforward. You can view the method in its entirety in Listing 33.

Listing 20 also displays the RMS values for each color plane in the original image.

Create and condition the output array object

I will let the comments in Listing 21 speak for themselves.

```
    //Create an empty output array of the same
size as the
    // incoming array of pixels.
    double[][][] output =
                        new
double[numImgRows][numImgCols][4];

    //Copy the alpha values directly to the
output array.
    // They will not be processed during the
convolution
    // process.
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        output[row][col][0] =
work3D[row][col][0];
      }//end inner loop
    }//end outer loop
```

Listing 21

Perform the 2D convolution

The code in Listing 22 performs the actual 2D convolution.

```
//Because of the length of the following
statements, and
// the width of this publication format, this
format
// sacrifices indentation style for clarity.
Otherwise,it
// would be necessary to break the statements
into so many
// short lines that it would be very difficult
to read
// them.

//Use nested for loops to perform a 2D
convolution of each
// color plane with the 2D convolution filter.

for(int yReg = numFilRows-1;yReg <
numImgRows;yReg++){
  for(int xReg = numFilCols-1;xReg <
numImgCols;xReg++){
    for(int filRow = 0;filRow <
numFilRows;filRow++){
```

```
      for(int filCol = 0;filCol <
numFilCols;filCol++){

        output[yReg-numFilRows/2][xReg-
numFilCols/2][1] +=
                      work3D[yReg-filRow][xReg-
filCol][1] *

filter[filRow][filCol];

        output[yReg-numFilRows/2][xReg-
numFilCols/2][2] +=
                      work3D[yReg-filRow][xReg-
filCol][2] *

filter[filRow][filCol];

        output[yReg-numFilRows/2][xReg-
numFilCols/2][3] +=
                      work3D[yReg-filRow][xReg-
filCol][3] *

filter[filRow][filCol];

      }//End loop on filCol
    }//End loop on filRow

    //Divide the result at each point in the
output by the
    // number of filter coefficients.  Note
that in some
    // cases, this is not helpful.  For
example, it is not
    // helpful when a large number of the
filter
    // coefficients have a value of zero.
    output[yReg-numFilRows/2][xReg-
numFilCols/2][1] =
           output[yReg-numFilRows/2][xReg-
numFilCols/2][1]/

(numFilRows*numFilCols);
    output[yReg-numFilRows/2][xReg-
numFilCols/2][2] =
           output[yReg-numFilRows/2][xReg-
numFilCols/2][2]/

(numFilRows*numFilCols);
    output[yReg-numFilRows/2][xReg-
numFilCols/2][3] =
           output[yReg-numFilRows/2][xReg-
numFilCols/2][3]/

(numFilRows*numFilCols);

  }//End loop on xReg
```

```
}//End loop on yReg

Listing 22
```

This is the same code that I explained in the earlier lesson entitled Processing Image Pixels, Understanding Image Convolution in Java. Therefore, I won't repeat that explanation here.

Restoring the mean and RMS values

In an earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness I explained how to modify the width of the color distribution of each color plane in an image. The RMS value is a measure of the width of the color distribution if the RMS value is computed after the mean has been removed from the color values.

Given a set of color values with a zero mean, we can modify the width of the color distribution simply by multiplying every color value in the set by the same constant. If the constant is greater than 1.0, the width of the resulting distribution will be wider. If the constant is less than 1.0, the width of the resulting distribution will be narrower.

Our objective here is to adjust the RMS value, and hence the width of the color distribution on each color plane, to the RMS value for that color plane in the unfiltered image.

Make sure the mean is zero

Although the mean was removed prior to convolution, it is possible that arithmetic inaccuracies during the convolution process could result in a slightly non-zero mean in the filtered image. The code in Listing 23 makes certain that the mean value is zero before adjusting the RMS value.

```
    removeMean(output,1);
    removeMean(output,2);
    removeMean(output,3);

Listing 23
```

Adjust the RMS value

The code in Listing 24 makes the adjustment to the RMS value for each color plane as described above.

```
    //Get and save the RMS value of the output for each
    // color plane.
    double outputRedRms = getRms(output,1);
    double outputGreenRms = getRms(output,2);
    double outputBlueRms = getRms(output,3);

    //Scale the output to cause the RMS value of the
output
```

```
    // to match the RMS value of the input
    scaleColorPlane(output,1,inputRedRms/outputRedRms);

scaleColorPlane(output,2,inputGreenRms/outputGreenRms);

scaleColorPlane(output,3,inputBlueRms/outputBlueRms);

    //Display the adjusted RMS values.  Should match
the
    // input RMS values.
    System.out.println(
                      "Output red RMS: " +
getRms(output,1));
    System.out.println(
                    "Output green RMS: " +
getRms(output,2));
    System.out.println(
                     "Output blue RMS: " +
getRms(output,3));
```

Listing 24

Listing 24 also displays the final RMS value for each color plane in the filtered image.

Restore the mean value

The mean value of a set of color values in a color plane can be changed simply by adding the same constant to every color value belonging to the color plane. Listing 25 restores the mean value for each color plane of the filtered output to mean value of the corresponding color plane in the unfiltered image.

```
    addConstantToColor(output,1,redMean);
    addConstantToColor(output,2,greenMean);
    addConstantToColor(output,3,blueMean);

    System.out.println(
                     "Output red mean: " +
getMean(output,1));
    System.out.println(
                   "Output green mean: " +
getMean(output,2));
    System.out.println(
                    "Output blue mean: " +
getMean(output,3));
```

Listing 25

Listing 25 also displays the final mean value of each color plane in the filtered image.

Guarantee that all color values are within the proper range

Listing 26 guarantees that all color values fall within the required range of 0 to 255 inclusive by setting negative values to zero and setting all values greater then 255 to 255.

```
    //Clip all negative color values at zero
and all color
    // values that are greater than 255 at 255.
    clipToZero(output,1);
    clipToZero(output,2);
    clipToZero(output,3);

    clipTo255(output,1);
    clipTo255(output,2);
    clipTo255(output,3);

Listing 26
```

The methods named **clipToZero** and **clipTo255** are straightforward. They can be viewed in their entirety in Listing 33.

Return the filtered image

Listing 27 returns a reference to the array containing the filtered pixels, converting the color values from type **double** to type **int** in the process.

```
    return doubleToInt(output);

  }//end convolve method

Listing 27
```

Listing 27 also signals the end of the static **convolve** method belonging to the class named **ImgMod32a**.

Run the Programs

I encourage you to copy, compile, and run the programs that are provided in the section named Complete Program Listings.

Experiment with different filters and images

Experiment with the programs, creating new convolution filters and applying them to your own images. Try to make certain that you understand the results of your experiments. Modify the programs and observe the results of your modifications.

The required classes

The required classes and the instructions for running the programs are provided in the comments at the beginning of the programs. The source code for the classes that are not provided in this lesson can be found in the earlier lessons referred to in the References section.

Perhaps the easiest way to find the source code for those classes is to go to Google and search for the following keywords where *className* is the name of the class of interest:

java baldwin "class className"

Have fun and learn

Above all, have fun and use these programs to learn as much as you can about image convolution.

Summary

This is Part 2 of a two-part lesson. In the two parts of this lesson, I have provided you with a general purpose image convolution program. In addition, I have walked you through several experiments intended to help you understand why image convolution does what it does.

I also showed you how to design and implement the following types of convolution filters:

- A simple copy filter
- Smoothing or softening filters
- Sharpening filters
- Embossing filters that produce a 3D-like effect
- Edge detection filters

What's Next?

Future lessons will show you how to write image-processing programs that implement many common special effects as well as a few that aren't so common. This will include programs to do the following:

- Deal with the effects of noise in an image.
- Morph one image into another image.
- Rotate an image.
- Change the size of an image.
- Create a kaleidoscope of an image.
- Other special effects that I may dream up or discover while doing the background research for the lessons in this series.

References

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

- 100 Periodic Motion and Sinusoids
- 104 Sampled Time Series
- 108 Averaging Time Series
- 1478 Fun with Java, How and Why Spectral Analysis Works
- 1482 Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
- 1483 Spectrum Analysis using Java, Frequency Resolution versus Data Length
- 1484 Spectrum Analysis using Java, Complex Spectrum and Phase Angle
- 1485 Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain
- 1487 Convolution and Frequency Filtering in Java
- 1488 Convolution and Matched Filtering in Java
- 1489 Plotting 3D Surfaces using Java
- 1490 2D Fourier Transforms using Java
- 1491 2D Fourier Transforms using Java, Part 2
- 1492 Plotting Large Quantities of Data using Java
- 400 Processing Image Pixels using Java, Getting Started
- 402 Processing Image Pixels using Java, Creating a Spotlight
- 404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
- 406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
- 408 Processing Image Pixels, Performing Convolution on Images
- 410 Processing Image Pixels, Understanding Image Convolution in Java
- 412 Processing Image Pixels, Applying Image Convolution in Java, Part 1

Complete Program Listings

Complete listings of the programs discussed in this lesson are provided in this section.

A disclaimer

The programs that I am providing and explaining in this series of lessons are not intended to be used for high-volume production work. Numerous integrated image-processing programs are available for that purpose. In addition, the Java Advanced Imaging *(JAI)* API has a number of built-in special effects if you prefer to write your own production image-processing programs using Java.

The programs that I am providing in this series are intended to make it easier for you to develop and experiment with image-processing algorithms and to gain a better understanding of how they work, and why they do what they do.

Listing 28

```
/* File Graph08.java
Copyright 2005, R.G.Baldwin

This is an updated version of Graph03 to allow the user to
plot up to eight functions instead of only 5.

GraphIntfc08 is a corresponding update to the earlier
interface named GraphIntfc01.

Graph03 and GraphIntfc01 were explained in lesson 1488
entitled Convolution and Matched Filtering in Java.

This program is very similar to Graph01 except that it has
been modified to allow the user to manually resize and
replot the frame.

Note:  This program requires access to the interface named
GraphIntfc08.

This is a plotting program.  It is designed to access a
class file, which implements GraphIntfc08, and to plot up
to eight functions defined in that class file. The plotting
surface is divided into the required number of equally
sized plotting areas, and one function is plotted on
Cartesian coordinates in each area.

The methods corresponding to the functions are named f1,
f2, f3, f4, f5, f6, f7, and f8.

The class containing the functions must also define a
method named getNmbr(), which takes no parameters and
returns the number of functions to be plotted.  If this
method returns a value greater than 8, a
NoSuchMethodException will be thrown.

Note that the constructor for the class that implements
GraphIntfc08 must not require any parameters due to the
use of the newInstance method of the Class class to
instantiate an object of that class.

If the number of functions is less than 8, then the absent
method names must begin with f8 and work down toward f1.
For example, if the number of functions is 3, then the
program will expect to call methods named f1, f2, and f3.
It is OK for the absent methods to be defined in the class.
They simply won't be invoked.  In fact, because they are
declared in the interface, they must be defined as dummy
methods in the class that implements the interface.

The plotting areas have alternating white and gray
backgrounds to make them easy to separate visually.

All curves are plotted in black.  A Cartesian coordinate
system with axes, tic marks, and labels is drawn in red
in each plotting area.
```

```
The Cartesian coordinate system in each plotting area has
the same horizontal and vertical scale, as well as the
same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values
of the extreme edges of the plotting area.

The program also compiles a sample class named junk, which
contains eight methods and the method named getNmbr.  This
makes it easy to compile and test this program in a
stand-alone mode.

At runtime, the name of the class that implements the
GraphIntfc08 interface must be provided as a command-line
parameter.  If this parameter is missing, the program
instantiates an object from the internal class named junk
and plots the data provided by that class.  Thus, you can
test the program by running it with no command-line
parameter.

This program provides the following text fields for user
input, along with a button labeled Graph.  This allows the
user to adjust the parameters and replot the graph as many
times with as many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these parameters and then click
the Graph button to cause the eight functions to be
re-plotted according to the new parameters.

Whenever the Graph button is clicked, the event handler
instantiates a new object of the class that implements
the GraphIntfc08 interface.  Depending on the nature of
that class, this may be redundant in some cases.  However,
it is useful in those cases where it is necessary to
refresh the values of instance variables defined in the
class (such as a counter, for example).

This program uses constants that were first defined in the
Color class of v1.4.0.  Therefore, the program requires
v1.4.0 or later to compile and run correctly.

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

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
```

```
import javax.swing.border.*;

class Graph08{
  public static void main(String[] args)
                                throws NoSuchMethodException,
                                       ClassNotFoundException,
                                       InstantiationException,
                                       IllegalAccessException{
    if(args.length == 1){
      //pass command-line paramater
      new GUI(args[0]);
    }else{
      //no command-line parameter given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph08 definition
//=======================================================//

class GUI extends JFrame implements ActionListener{

  //Define plotting parameters and their default values.
  double xMin = 0.0;
  double xMax = 400.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 20.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small on x-axis, a default
  // value is used later.
  double xTicLen = (yMax-yMin)/50;
  double yTicLen = (xMax-xMin)/50;

  //Calculation interval along x-axis
  double xCalcInc = 1.0;

  //Text fields for plotting parameters
  JTextField xMinTxt = new JTextField("" + xMin);
  JTextField xMaxTxt = new JTextField("" + xMax);
  JTextField yMinTxt = new JTextField("" + yMin);
  JTextField yMaxTxt = new JTextField("" + yMax);
  JTextField xTicIntTxt = new JTextField("" + xTicInt);
  JTextField yTicIntTxt = new JTextField("" + yTicInt);
  JTextField xCalcIncTxt = new JTextField("" + xCalcInc);

  //Panels to contain a label and a text field
  JPanel pan0 = new JPanel();
  JPanel pan1 = new JPanel();
  JPanel pan2 = new JPanel();
  JPanel pan3 = new JPanel();
  JPanel pan4 = new JPanel();
  JPanel pan5 = new JPanel();
  JPanel pan6 = new JPanel();
```

```
//Misc instance variables
int frmWidth = 408;
int frmHeight = 430;
int width;
int height;
int number;
GraphIntfc08 data;
String args = null;

//Plots are drawn on the canvases in this array.
Canvas[] canvases;

//Constructor
GUI(String args)throws NoSuchMethodException,
                       ClassNotFoundException,
                       InstantiationException,
                       IllegalAccessException{

  if(args != null){
    //Save for use later in the ActionEvent handler
    this.args = args;
    //Instantiate an object of the target class using the
    // String name of the class.
    data = (GraphIntfc08)Class.forName(args).
                                            newInstance();
  }else{
    //Instantiate an object of the test class named junk.
    data = new junk();
  }//end else

  //Create array to hold correct number of Canvas
  // objects.
  canvases = new Canvas[data.getNmbr()];

  //Throw exception if number of functions is greater
  // than 8.
  number = data.getNmbr();
  if(number > 8){
    throw new NoSuchMethodException(
                                  "Too many functions.  "
                                    + "Only 8 allowed.");
  }//end if

  //Create the control panel and give it a border for
  // cosmetics.
  JPanel ctlPnl = new JPanel();
  ctlPnl.setLayout(new GridLayout(0,4));//?rows x 4 cols
  ctlPnl.setBorder(new EtchedBorder());

  //Button for replotting the graph
  JButton graphBtn =new JButton("Graph");
  graphBtn.addActionListener(this);

  //Populate each panel with a label and a text field.
  // Will place these panels in a grid on the control
```

```
    // panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);

    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);

    //Add the populated panels and the button to the
    // control panel with a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);

    //Create a panel to contain the Canvas objects.  They
    // will be displayed in a one-column grid.
    JPanel canvasPanel = new JPanel();
    canvasPanel.setLayout(new GridLayout(0,1));//?rows 1col

    //Create a custom Canvas object for each function to
    // be plotted and add them to the one-column grid.
    // Make background colors alternate between white and
    // gray.
    for(int cnt = 0;
                  cnt < number; cnt++){
      switch(cnt){
        case 0 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt].setBackground(Color.WHITE);
          break;
        case 1 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt].setBackground(Color.LIGHT_GRAY);
          break;
        case 2 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt].setBackground(Color.WHITE);
```

```
          break;
        case 3 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt].setBackground(Color.LIGHT_GRAY);
          break;
        case 4 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt]. setBackground(Color.WHITE);
          break;
        case 5 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt].setBackground(Color.LIGHT_GRAY);
          break;
        case 6 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt]. setBackground(Color.WHITE);
          break;
        case 7 :
          canvases[cnt] = new MyCanvas(cnt);
          canvases[cnt].setBackground(Color.LIGHT_GRAY);
      }//end switch
      //Add the object to the grid.
      canvasPanel.add(canvases[cnt]);
    }//end for loop

    //Add the sub-assemblies to the frame.  Set its
    // location, size, and title, and make it visible.
    getContentPane(). add(ctlPnl,"South");
    getContentPane(). add(canvasPanel,"Center");

    setBounds(0,0,frmWidth,frmHeight);

    if(args == null){
      setTitle("Graph08, Copyright 2005, " +
                                      "Richard G. Baldwin");
    }else{
      setTitle("Graph08/" + args + " Copyright 2005, " +
                                           "R. G. Baldwin");
    }//end else

    setVisible(true);

    //Set to exit on X-button click
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    //Guarantee a repaint on startup.
    for(int cnt = 0;cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

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

  //This event handler is registered on the JButton to
  // cause the functions to be replotted.
  public void actionPerformed(ActionEvent evt){
```

```
    //Re-instantiate the object that provides the data
    try{
      if(args != null){
        data = (GraphIntfc08)Class.
                              forName(args).newInstance();
      }else{
        data = new junk();
      }//end else
    }catch(Exception e){
      //Known to be safe at this point. Otherwise would
      // have aborted earlier.
    }//end catch

    //Set plotting parameters using data from the text
    // fields.
    xMin = Double.parseDouble(xMinTxt.getText());
    xMax = Double.parseDouble(xMaxTxt.getText());
    yMin = Double.parseDouble(yMinTxt.getText());
    yMax = Double.parseDouble(yMaxTxt.getText());
    xTicInt = Double.parseDouble(xTicIntTxt.getText());
    yTicInt = Double.parseDouble(yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(xCalcIncTxt.getText());

    //Calculate new values for the length of the tic marks
    // on the axes.  If too small on x-axis, a default
    // value is used later.
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting areas
    for(int cnt = 0;cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end actionPerformed
  //-----------------------------------------------------//


//This is an inner class, which is used to override the
// paint method on the plotting surface.
class MyCanvas extends Canvas{
  int cnt;//object number
  //Factors to convert from double values to integer pixel
  // locations.
  double xScale;
  double yScale;

  MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
  }//end constructor

  //Override the paint method
  public void paint(Graphics g){

    //Get and save the size of the plotting surface
    width = canvases[0].getWidth();
```

```
    height = canvases[0].getHeight();

    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                                    (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);

    //Get initial data values
    double xVal = xMin;
    int oldX = getTheX(xVal);
    int oldY = 0;
    //Use the Canvas obj number to determine which method
    // to invoke to get the value for y.
    switch(cnt){
      case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
      case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
      case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
      case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
      case 4 :
        oldY = getTheY(data.f5(xVal));
        break;
      case 5 :
        oldY = getTheY(data.f6(xVal));
        break;
      case 6 :
        oldY = getTheY(data.f7(xVal));
        break;
      case 7 :
        oldY = getTheY(data.f8(xVal));
    }//end switch

    //Now loop and plot the points
    while(xVal < xMax){
      int yVal = 0;
      //Get next data value.  Use the Canvas obj number to
      // determine which method to invoke to get the value
      // for y.
      switch(cnt){
        case 0 :
          yVal = getTheY(data.f1(xVal));
          break;
        case 1 :
          yVal = getTheY(data.f2(xVal));
```

```
          break;
        case 2 :
          yVal = getTheY(data.f3(xVal));
          break;
        case 3 :
          yVal = getTheY(data.f4(xVal));
          break;
        case 4 :
          yVal = getTheY(data.f5(xVal));
          break;
        case 5 :
          yVal = getTheY(data.f6(xVal));
          break;
        case 6 :
          yVal = getTheY(data.f7(xVal));
          break;
        case 7 :
          yVal =getTheY(data.f8(xVal));
      }//end switch1

      //Convert the x-value to an int and draw the next
      // line segment
      int x = getTheX(xVal);
      g.drawLine(oldX,oldY,x,yVal);

      //Increment along the x-axis
      xVal += xCalcInc;

      //Save end point to use as start point for next line
      // segment.
      oldX = x;
      oldY = yVal;
    }//end while loop

  }//end overridden paint method
  //-----------------------------------------------------//

  //Method to draw axes with tic marks and labels in the
  // color RED
  void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Lable left x-axis and bottom y-axis.  These are the
    // easy ones.  Separate the labels from the ends of the
    // tic marks by two pixels.
    g.drawString("" + (int)xMin,getTheX(xMin),
                                    getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,getTheX(yTicLen/2)+2,
                                          getTheY(yMin));

    //Label the right x-axis and the top y-axis.  These are
    // the hard ones because the position must be adjusted
    // by the font size and the number of characters.
    //Get the width of the string for right end of x-axis
    // and the height of the string for top of y-axis
    //Create a string that is an integer representation of
```

```
    // the label for the right end of the x-axis.  Then get
    // a character array that represents the string.
    int xMaxInt = (int)xMax;
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.toCharArray();

    //Get a FontMetrics object that can be used to get the
    // size of the string in pixels.
    FontMetrics fontMetrics = g.getFontMetrics();
    //Get a bounding rectangle for the string
    Rectangle2D r2d = fontMetrics.getStringBounds(
                                  array,0,array.length,g);

    //Get the width and the height of the bounding
    // rectangle.  The width is the width of the label at
    // the right end of the x-axis.  The height applies to
    // all the labels, but is needed specifically for the
    // label at the top end of the y-axis.
    int labWidth = (int)(r2d.getWidth());
    int labHeight = (int)(r2d.getHeight());

    //Label the positive x-axis and the positive y-axis
    // using the width and height from above to position
    // the labels.  These labels apply to the very ends of
    // the axes at the edge of the plotting surface.
    g.drawString("" + (int)xMax,getTheX(xMax)-labWidth,
                                    getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMax,getTheX(yTicLen/2)+2,
                                  getTheY(yMax)+labHeight);

    //Draw the axes
    g.drawLine(getTheX(xMin),getTheY(0.0),getTheX(xMax),
                                             getTheY(0.0));

    g.drawLine(getTheX(0.0),getTheY(yMin),getTheX(0.0),
                                            getTheY(yMax));

    //Draw the tic marks on axes
    xTics(g);
    yTics(g);
  }//end drawAxes

  //-------------------------------------------------------//

  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =getTheY(-xTicLen/2);

    //If the vertical size of the plotting area is small,
    // the calculated tic size may be too small.  In that
    // case, set it to 10 pixels.
```

```
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short lines to serve as
    // tic marks. Begin with the positive x-axis moving to
    // the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving to the left from
    // zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while

  }//end xTics
  //-----------------------------------------------------//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short lines to serve as tic
    // marks. Begin with the positive y-axis moving up from
    // zero.
    while(yDoub < yMax){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving down from zero.
    yDoub = 0;
    while(yDoub > yMin){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics
  //-----------------------------------------------------//

  //This method translates and scales a double y value to
  // plot properly in the integer coordinate system. In
  // addition to scaling, it causes the positive direction
```

```
  // of the y-axis to be from bottom to top.
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //-------------------------------------------------------//

  //This method scales a double x value to plot properly
  // in the integer coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //-------------------------------------------------------//

}//end inner class MyCanvas
//=========================================================//

}//end class GUI
//=========================================================//

//Sample test class.  Required for compilation and
// stand-alone testing.
class junk implements GraphIntfc08{
  public int getNmbr(){
    //Return number of functions to process.  Must not
    // exceed 8.
    return 8;
  }//end getNmbr

  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    return (x*x)/200.0;
  }//end f3

  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
    return 100*Math.sin(x/20.0);
  }//end f5

  public double f6(double x){
    return 100*Math.sin(x/30.0);
  }//end f6

  public double f7(double x){
    return 100*Math.sin(x/5.0);
```

```
  }//end f7

  public double f8(double x){
    return 100*Math.sin(x/2.5);
  }//end f8
}//end sample class junk

Listing 28
```

Listing 29

```
/* File GraphIntfc08.java
Copyright 2005, R.G.Baldwin

This interface must be implemented by classes whose objects
produce data to be plotted by the program named Graph08.

This is an upgrade from GraphIntfc01, designed to handle
eight graphs instead of only five.

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

public interface GraphIntfc08{
  public int getNmbr();
  public double f1(double x);
  public double f2(double x);
  public double f3(double x);
  public double f4(double x);
  public double f5(double x);
  public double f6(double x);
  public double f7(double x);
  public double f8(double x);
}//end GraphIntfc08

Listing 29
```

Listing 30

```
/* File Dsp041.java
Copyright 2005, R.G.Baldwin

This class is loosely based on the class named Dsp040a. The
class named Dsp040a was explained in detail in Lesson 1488
entitled Convolution and Matched Filtering in Java.

The purpose of this class is to make it easy to experiment
with different time series and different convolution
filters.

This class must be run under control of the class named
Graph08. Thus, it requires access to the class named
Graph08 and the interface named GraphIntfc08.
```

Graph08 and GraphIntfc08 are updates to Graph03 and GraphIntfc01. The updates allow the user to plot a maximum of eight graphs instead of a maximum of five graphs as is the case with Graph03.

Graph03 and GraphIntfc01 were explained in lesson 1488 entitled Convolution and Matched Filtering in Java.

To run this program, enter the following command at the command line:

```
java Graph08 Dsp041
```

Access to the following classes, plus some inner classes defined in these classes is required to compile and run this class under control of the class named Graph08:

```
Dsp041.class
Graph08.class
GraphIntfc08.class
GUI.class
```

The source code for all of the above classes is provided either in this file or in lesson 412 entitled Processing Image Pixels, Applying Image Convolution in Java.

This program illustrates the application of a convolution filter to signals having a known waveform. In its current state, five different convolution filters are coded into the class. Since the class can only apply one convolution filter at a time, it is necessary to enable and disable the filters using comments and then recompile the class to switch from one convolution filter to the other. The five convolution filters are:

1. A single impulse filter that simply copies the input to the output.
2. A high-pass filter with an output that is proportional to the slope of the signal. In essence, the output approximates the first derivative of the signal.
3. A high-pass filter with an output that is proportional to the rate of change of the slope of the signal. This output approximates the second derivative of the signal.
4. A relatively soft high-pass filter, which produces a little blip in its output each time the slope of the signal changes. The size of the blip is roughly proportional to the rate of change of the slope of the signal.
5. A low-pass smoothing filter. The output approximates a four-point running average or integration of the signal.

These convolution filters are applied to signal waveforms having varying slopes. Several interesting results are displayed. (The filters and the signal waveforms can be easily modified by modifying that part of the program and

```
recompiling the program.)

The display contains six graphs and shows the following:
1. The signal waveform as a time series.
2. The convolution filter waveform as a time series.
3. The result of applying the convolution filter to the
   signal, including the impulse response of the filter.
4. The amplitude spectrum of the signal expressed in db.
5. The amplitude frequency response of the convolution
   filter expressed in db.
6. The amplitude spectrum of the output produced by
   applying the convolution filter to the signal.

The convolution algorithm emulates a one-dimensional
version of the 2D image convolution algorithm used in the
class named ImgMod032 with respect to output normalization
and scaling.  See an explanation of just what this means in
the comments at the beginning of the convolve method.

In addition to computing and plotting the output from the
convolution process, the class computes and plots several
spectral graphs.

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

class Dsp041 implements GraphIntfc08{
  //Establish length for various arrays
  int filterLen = 200;
  int signalLen = 400;
  int outputLen = signalLen - filterLen;
  //Ignore right half of signal, which is all zeros, when
  // computing the spectrum.
  int signalSpectrumPts = signalLen/2;
  int filterSpectrumPts = outputLen;
  int outputSpectrumPts = outputLen;


  //Create arrays to store different types of data.
  double[] signal = new double[signalLen];
  double[] filter = new double[filterLen];
  double[] output = new double[outputLen];
  double[] spectrumA = new double[signalSpectrumPts];
  double[] spectrumB = new double[filterSpectrumPts];
  double[] spectrumC = new double[outputSpectrumPts];

  public Dsp041(){//constructor

    //Create and save a filter.  Change the locations of
    // the following comment indicators to enable and/or
    // disable a particular filter.  Then recompile the
    // class and rerun the program to see the effect of
    // the newly enabled filter on the signal.

    //This is a single impulse filter that simply copies
    // the input to the output.
```

```
    filter[0] = 1;

/*
    //This is a high-pass filter with an output that is
    // proportional to the slope of the signal.  In
    // essence,the output approximates the first derivative
    // of the signal.
    filter[0] = -1.0;
    filter[1] = 1.0;

    //This is a high-pass filter with an output that is
    // proportional to the rate of change of the slope of
    // the signal. In essence, the output approximates the
    // second derivative of the signal.
    filter[0] = -0.5;
    filter[1] = 1.0;
    filter[2] = -0.5;

    //This is a relatively soft high-pass filter, which
    // produces a little blip in the output each time the
    // slope of the signal changes.  The size of the blip
    // is roughly proportional to the rate of change of the
    // slope of the signal.
    filter[0] = -0.2;
    filter[1] = 1.0;
    filter[2] = -0.2;

    //This is a low-pass smoothing filter.  It approximates
    // a four-point running average or integration of the
    // signal.
    filter[0] = 0.250;
    filter[1] = 0.250;
    filter[2] = 0.250;
    filter[3] = 0.250;
*/

    //Create a signal time series containing four distinct
    // waveforms:
    //  An impulse.
    //  A rectangular pulse.
    //  A triangular pulse with a large slope.
    //  A triangular pulse with a smaller slope.

    //First create a baseline in the signal time series.
    //Modify the following value and recompile the class
    // to change the baseline.
    double baseline = 10.0;
    for(int cnt = 0;cnt < signalLen;cnt++){
      signal[cnt] = baseline;
    }//end for loop

    //Now add the pulses to the signal time series.

    //First add an impulse.
    signal[20] = 75;
```

```
    //Add a rectangular pulse.
    signal[30] = 75;
    signal[31] = 75;
    signal[32] = 75;
    signal[33] = 75;
    signal[34] = 75;
    signal[35] = 75;
    signal[36] = 75;
    signal[37] = 75;
    signal[38] = 75;
    signal[39] = 75;

    //Add a triangular pulse with a large slope.
    signal[50] = 10;
    signal[51] = 30;
    signal[52] = 50;
    signal[53] = 70;
    signal[54] = 90;
    signal[55] = 70;
    signal[56] = 50;
    signal[57] = 30;
    signal[58] = 10;

    //Add a triangular pulse with a smaller slope.
    signal[70] = 10;
    signal[71] = 20;
    signal[72] = 30;
    signal[73] = 40;
    signal[74] = 50;
    signal[75] = 60;
    signal[76] = 70;
    signal[77] = 80;
    signal[78] = 90;
    signal[79] = 80;
    signal[80] = 70;
    signal[81] = 60;
    signal[82] = 50;
    signal[83] = 40;
    signal[84] = 30;
    signal[85] = 20;
    signal[86] = 10;

    //Convolve the signal with the convolution filter.
    // Note, this convolution algorithm emulates a
    // one-dimensional version of the 2D image
    // convolution algorithm used in ImgMod032 with respect
    // to normalization and scaling.
    convolve(signal,filter,output);

    //Compute and save the DFT of the signal, expressed in
    // db.
    //Ignore right half of signal which is all zeros.
    dft(signal,signalSpectrumPts,spectrumA);

    //Compute and save the DFT of the convolution filter
    // expressed in db.
```

```
    //Note that the convolution filter is added to a long
    // time series having zero values.  This causes the
    // output of the DFT to be finely sampled and produces
    // a smooth curve for the frequency response of the
    // convolution filter.
    dft(filter,filterSpectrumPts,spectrumB);

    //Compute and save the DFT of the output expressed in
    // decibels.
    dft(output,outputSpectrumPts,spectrumC);

    //All of the time series have now been produced and
    // saved.  They may be retrieved and plotted by
    // invoking the methods named f1 through f6 below.

  }//end constructor

  //-------------------------------------------------------//
  //The following nine methods are required by the
  // interface named GraphIntfc08.  They are invoked by the
  // plotting class named Graph08.
  public int getNmbr(){
    //Return number of functions to process. Must not
    // exceed 6.
    return 6;
  }//end getNmbr
  //-------------------------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    //This version of this method returns the signal.
    if(index < 0 || index > signal.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return signal[index] * 1.0;
    }//end else
  }//end f1
  //-------------------------------------------------------//
  public double f2(double x){
    //Return the convolution filter.
    int index = (int)Math.round(x);
    if(index < 0 || index > filter.length-1){
      return 0;
    }else{
      //Scale for display and return.
      return filter[index] * 50.0;
    }//end else
  }//end f2
  //-------------------------------------------------------//
  public double f3(double x){
    //Return convolution output.
    int index = (int)Math.round(x);
    if(index < 0 || index > output.length-1){
      return 0;
    }else{
      //Scale for display and return.
```

```
      return output[index] * 1.0;
    }//end else
  }//end f3
  //-------------------------------------------------//
  public double f4(double x){
    //Return frequency spectrum of the signal.
    int index = (int)Math.round(x);
    if(index < 0 || index > spectrumA.length-1){
      return 0;
    }else{
      //Adjust peak amplitude for display and return.  With
      // this scaling, 100 vertical units in the plot
      // produced by Graph08 represents 25 decibels.  In
      // addition, the db values are adjusted to cause the
      // maximum value to be plotted at 100 units above
      // the horizontal axis.
      return spectrumA[index] * 4.0 + 100;
    }//end else
  }//end f4
  //-------------------------------------------------//
  public double f5(double x){
    //Return frequency response of convolution filter.
    int index = (int)Math.round(x);
    if(index < 0 || index > spectrumB.length-1){
      return 0;
    }else{
      //Adjust peak amplitude for display and return. See
      // comments in f4.
      return spectrumB[index] * 4.0 + 100;
    }//end else
  }//end f5
  //-------------------------------------------------//
  public double f6(double x){
    //Return frequency spectrum of the output.
    int index = (int)Math.round(x);
    if(index < 0 || index > spectrumC.length-1){
      return 0;
    }else{
      //Adjust peak amplitude for display and return. See
      // comments in f4.
      return spectrumC[index] * 4.0 + 100;
    }//end else
  }//end f6
  //-------------------------------------------------//
  public double f7(double x){
    //This method is not used but must be defined.
    return 0.0;
  }//end f7
  //-------------------------------------------------//
  public double f8(double x){
    //This method is not used but must be defined.
    return 0.0;
  }//end f8
  //-------------------------------------------------//

  //This method computes and returns the amplitude spectrum
```

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

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

    //Convert the amplitude spectrum to decibels.
    for(int cnt = 0;cnt < dataLen;cnt++){
      //Set zero and negative values to -Double.MAX_VALUE
      // before converting to log values. Shouldn't be any
      // negative values. May be some zero values. An
      // amplitude value of 0 should result in negative
      // infinity decibels.
      if(spectrum[cnt] <= 0){
        spectrum[cnt] = -Double.MAX VALUE;
      }//end if
      if(spectrum[cnt] > 0){
        //Ignore zero and negative values. Convert positive
        // values to log base 10.
        spectrum[cnt] = 20*Math.log10(spectrum[cnt]);
      }//end if
    }//end for loop

    //The amplitude spectrum has now been converted to db.
    // Normalize the peak to zero db.

    //Get the max value.
    double max = -Double.MAX_VALUE;
    for(int cnt = 0;cnt < dataLen;cnt++){
```

```
      if(spectrum[cnt] > max)max = spectrum[cnt];
    }//end for loop

    //Subtract the max from every value
    for(int cnt = 0;cnt < dataLen;cnt++){
      spectrum[cnt] -= max;
    //System.out.print(spectrum[cnt] + " ");
    }//end for loop
  }//end dft
  //-------------------------------------------------//
  //This method applies an incoming convolution filter
  // to an incoming set of data and deposits the filtered
  // data in an output array whose reference is received as
  // an incoming parameter.
  //This convolution algorithm emulates a one-dimensional
  // version of the 2D image convolution algorithm used in
  // the class named ImgMod032 with respect to
  // normalization and scaling.
  //There are two major differences between this algorithm
  // and the 2D algorithm.  First, this algorithm flips the
  // convolution filter end-for-end whereas the 2D
  // algorithm does not flip the convolution filter.
  // Thus, the 2D algorithm requires that the convolution
  // operator be flipped before it is passed to the method.
  // Second, whereas the 2D algorithm normalizes the output
  // data so as to guarantee that the output values range
  // from 0 to 255 inclusive, this algorithm normalizes the
  // output data so as to guarantee that the output values
  // range from 0 to 100 inclusive.  This difference is of
  // no practical significance other than to cause the
  // output values to be plotted on a scale that is
  // somewhat easier to interpret.
  //Both algorithms assume that the incoming data consists
  // of all positive values (as is the case with color
  // values) with regard to the normalization rationale.
  // However, that is not a technical requirement.
  //The algorithm begins by computing and saving the mean
  // value of the incoming data.  Then it makes a copy of
  // the incoming data, removing the mean in the process.
  // (The copy is made simply to avoid modifying the
  // original data.)  Then the method applies the
  // convolution filter to the copy of the incoming data
  // producing an output time series with a zero mean
  // value.
  //Then the method adds the  original mean value to the
  // output values causing the mean value of the output
  // to be the same as the mean value of the input.
  //Following this, the method computes the minimum value
  // of the output and checks to see if it is negative.  If
  // so, the minimum value is subtracted from all output
  // values, causing the minimum value of the output to be
  // zero.  Otherwise, no adjustment is made on the basis
  // of the minimum value.
  //Then the method computes the maximum value and checks
  // to see if the maximum value is greater than 100.  If
  // so, all output values are scaled so as to cause the
```

```
// maximum output value to be 100.  Otherwise, no
// adjustment is made on the basis of the maximum value.
public  void convolve(double[] data,
                      double[] operator,
                      double[] output){
  int dataLen = data.length;
  double[] temp = new double[dataLen];


  //Get, save, and remove the mean value.
  //Copy the data into a temporary array, removing the
  // mean value in the process.
  double sum = 0;
  for(int cnt = 0;cnt < dataLen;cnt++){
    sum += data[cnt];
  }//end for loop
  double mean = sum/dataLen;
  for(int cnt = 0;cnt < dataLen;cnt++){
    temp[cnt] = data[cnt] - mean;
  }//end for loop

  //Apply the convolution filter to the copy of the
  // data, dealing with the index reversal required
  // to flip the operator end-for-end.
  int operatorLen = operator.length;
  for(int i = 0;i < dataLen-operatorLen;i++){
    output[i] = 0;
    for(int j = operatorLen-1;j >= 0;j--){
      output[i] += temp[i+j]*operator[j];
    }//end inner loop
  }//end outer loop

  //Restore the original mean value to the output.
  for(int cnt = 0;cnt < dataLen-operatorLen;cnt++){
    output[cnt] += mean;
  }//end for loop

  //Find the minimum value in the output.
  double min = Double.MAX_VALUE;
  for(int cnt = 0;cnt < dataLen-operatorLen;cnt++){
    if(output[cnt] < min){
      min = output[cnt];
    }//end if
  }//end for loop
  System.out.println("Output min: " + min);

  //If the minimum value is negative, subtract it from
  // all of the output values to ensure that the output
  // minimum is zero.
  if(min < 0.0){
    for(int cnt = 0;cnt < dataLen-operatorLen;cnt++){
      output[cnt] -= min;
    }//end for loop
  }//end if

  //Find the maximum value in the output.
```

```
    double max = -Double.MAX_VALUE;
    for(int cnt = 0;cnt < dataLen-operatorLen;cnt++){
      if(output[cnt] > max){
        max = output[cnt];
      }//end if
    }//end for loop
    System.out.println("Output max: " + max);

    //If the maximum value is greater than 100, scale all
    // of the output values to ensure that the output
    // maximum is 100.
    if(max > 100.0){
      for(int cnt = 0;cnt < dataLen-operatorLen;cnt++){
        output[cnt] *= 100.0/max;
      }//end for loop
    }//end if

  }//end convolve method
}//end class Dsp041
//=============================================================//


Listing 30
```

Listing 31

```
/*File ImgMod33.java
Copyright 2005, R.G.Baldwin

This class provides a general purpose 2D image convolution
and color filtering capability in Java. The class is
designed to be driven by the class named ImgMod02a.

The image file is specified on the command line. The name
of a file containing the 2D convolution filter is provided
via a TextField after the program starts running.
Multiplicative factors, which are applied to the individual
color planes are also provided through three TextFields
after the program starts running.

Enter the following at the command line to run this
program:

java ImgMod02a ImgMod33 ImageFileName

where ImageFileName is the name of a .gif or .jpg file,
including the extension.

Then enter the name of a file containing a 2D convolution
filter in the TextField that appears on the screen.  Click
the Replot button on the Frame that displays the image
to cause the convolution filter to be applied to the image.
You can modify the multiplicative factors in the three
TextFields labeled Red, Green, and Blue before clicking the
Replot button to cause the color values to be scaled by
```

```
the respective multiplicative factors.  The default
multiplicative factor for each color plane is 1.0.

When you click the Replot button, the image in the top of
the Frame will be convolved with the filter, the color
values in the color planes will be scaled by the
multiplicative factors, and the filtered image will appear
in the bottom of the Frame.

Each time you click the Replot button, two additional
graphs are produced that show the following information
in a color contour map fomat:
1.  The convolution filter.
2.  The wave number response of the convolution filter.

Because the GUI that contains the TextField for entry of
the convolution filter file name also contains three
additional TextFields that allow for the entry of
multiplicative factors that are applied to the three color
planes, it is possible to implement color filtering in
addition to convolution filtering.  To apply color
filtering, enter new multiplicative scale factors into
the TextFields for Red, Green, and Blue and click the
Replot button.

Once the program is running, different convolution filters
and different color filters can be successively applied to
the same image, either separately or in combination, by
entering the name of each new filter file into the
TextField and/or entering new color multiplicative factors
into the respective color TextFields and then clicking the
Replot button

See comments at the beginning of the method named getFilter
for a description and an example of the required format for
the file containing the 2D convolution filter.

This program requires access to the following class files
plus some inner classes that are defined inside the
following classes:

ImgIntfc02.class
ImgMod02a.class
ImgMod29.class
ImgMod30.class
ImgMod32.class
ImgMod33.class

Tested using J2SE 5.0 and WinXP
**********************************************************/
import java.awt.*;
import java.io.*;

class ImgMod33 extends Frame implements ImgIntfc02{

  TextField fileNameField = new TextField("");
```

```
  Panel rgbPanel = new Panel();
  TextField redField = new TextField("1.0");
  TextField greenField = new TextField("1.0");
  TextField blueField = new TextField("1.0");
  Label instructions = new Label(
          "Enter Filter File Name and scale factors for " +
          "Red, Green, and Blue and click Replot");
  //-------------------------------------------------//

  ImgMod33(){//constructor
    setLayout(new GridLayout(4,1));
    add(new Label("Filter File Name"));
    add(fileNameField);

    //Populate the rgbPanel
    rgbPanel.add(new Label("Red"));
    rgbPanel.add(redField);
    rgbPanel.add(new Label("Green"));
    rgbPanel.add(greenField);
    rgbPanel.add(new Label("Blue"));
    rgbPanel.add(blueField);

    add(rgbPanel);
    add(instructions);
    setTitle("Copyright 2005, R.G.Baldwin");
    setBounds(400,0,460,125);
    setVisible(true);
  }//end constructor
  //-------------------------------------------------//

  //This method is required by ImgIntfc02.  It is called at
  // the beginning of the run and each time thereafter that
  // the user clicks the Replot button on the Frame
  // containing the images.
  //The method gets a 2D convolution filter from a text
  // file, applies it to the incoming 3D array of pixel
  // data and returns a filtered 3D array of pixel data.
  public int[][][] processImg(int[][][] threeDPix,
                              int imgRows,
                              int imgCols){

    //Create an empty output array of the same size as the
    // incoming array.
    int[][][] output = new int[imgRows][imgCols][4];

    //Make a working copy of the 3D pixel array to avoid
    // making permanent changes to the original image data.
    int[][][] working3D = new int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        working3D[row][col][0] = threeDPix[row][col][0];
        working3D[row][col][1] = threeDPix[row][col][1];
        working3D[row][col][2] = threeDPix[row][col][2];
        working3D[row][col][3] = threeDPix[row][col][3];
        //Copy alpha values directly to the output. They
        // are not processed when the image is filtered
```

```
      // by the convolution filter.
      output[row][col][0] = threeDPix[row][col][0];
    }//end inner loop
  }//end outer loop

  //Get the file name containing the filter from the
  // textfield.
  String fileName = fileNameField.getText();
  if(fileName.equals("")){
    //The file name is an empty string. Skip the
    // convolution process and pass the input image
    // directly to the output.
    output = working3D;
  }else{
    //Get a 2D array that is populated with the contents
    // of the file containing the 2D filter.
    double[][] filter = getFilter(fileName);


    //Plot the impulse response and the wave-number
    // response of the convolution filter.  These items
    // are not computed and plotted when the program
    // starts running.  Rather, they are computed and
    // plotted each time the user clicks the Replot
    // button after entering the name of a file
    // containing a convolution filter into the
    // TextField.

    //Begin by placing the impulse response in the
    // center of a large flat surface with an elevation
    // of zero.This is done to improve the resolution of
    // the Fourier Transform, which will be computed
    // later.
    int numFilterRows = filter.length;
    int numFilterCols = filter[0].length;
    int rows = 0;
    int cols = 0;
    //Make the size of the surface ten pixels larger than
    // the convolution filter with a minimum size of
    // 32x32 pixels.
    if(numFilterRows < 22){
      rows = 32;
    }else{
      rows = numFilterRows + 10;
    }//end else
    if(numFilterCols < 22){
      cols = 32;
    }else{
      cols = numFilterCols + 10;
    }//end else

    //Create the surface, which will be initialized to
    // all zero values.
    double[][] filterSurface = new double[rows][cols];
    //Place the convolution filter in the center of the
    // surface.
```

```
    for(int row = 0;row < numFilterRows;row++){
      for(int col = 0;col < numFilterCols;col++){
        filterSurface[row + (rows - numFilterRows)/2]
                     [col + (cols - numFilterCols)/2] =
                                          filter[row][col];
      }//end inner loop
    }//end outer loop

    //Display the filter and the surface on which it
    // resides as a 3D plot in a color contour format.
    new ImgMod29(filterSurface,4,true,1);

    //Get and display the 2D Fourier Transform of the
    // convolution filter.

    //Prepare arrays to receive the results of the
    // Fourier transform.
    double[][] real = new double[rows][cols];
    double[][] imag = new double[rows][cols];
    double[][] amp = new double[rows][cols];
    //Perform the 2D Fourier transform.
    ImgMod30.xform2D(filterSurface,real,imag,amp);
    //Ignore the real and imaginary results.  Prepare the
    // amplitude spectrum for more-effective plotting by
    // shifting the origin to the center in wave-number
    // space.
    double[][] shiftedAmplitudeSpect =
                               ImgMod30.shiftOrigin(amp);

    //Get and display the minimum and maximum wave number
    // values.  This is useful because the way that the
    // wave number plots are normalized. it is not
    // possible to infer the flatness or lack thereof of
    // the wave number surface simply by viewing the
    // plot.  The colors that describe the elevations
    // always range from black at the minimum to white at
    // the maximum, with different colors in between
    // regardless of the difference between the minimum
    // and the maximum.
    double maxValue = -Double.MAX_VALUE;
    double minValue = Double.MAX_VALUE;
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        if(amp[row][col] > maxValue){
          maxValue = amp[row][col];
        }//end if
        if(amp[row][col] < minValue){
          minValue = amp[row][col];
        }//end if
      }//end inner loop
    }//end outer loop

    System.out.println("minValue: " + minValue);
    System.out.println("maxValue: " + maxValue);
    System.out.println("ratio: " + maxValue/minValue);
```

```
      //Generate and display the wave-number response
      // graph by plotting the 3D surface on the computer
      // screen.
      new ImgMod29(shiftedAmplitudeSpect,4,true,1);

      //Perform the convolution.
      output = ImgMod32.convolve(working3D,filter);
    }//end else

    //Scale output color planes.  Color planes will be
    // scaled only if the corresponding scale factor in the
    // TextField has a value other than 1.0.  Otherwise,
    // there is no point in consuming computer time to do
    // the scaling.
    if(!redField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                        redField.getText());
      scaleColorPlane(output,1,scale);
    }//end if on redField

    if(!greenField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                      greenField.getText());
      scaleColorPlane(output,2,scale);
    }//end if on greenField

    if(!blueField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                       blueField.getText());
      scaleColorPlane(output,3,scale);
    }//end if on blueField

    //Return a reference to the array containing the image,
    // which has undergone both convolution filtering and
    // color filtering.
    return output;

  }//end processImg method
  //-----------------------------------------------------//


  //The purpose of this method is to scale every color
  // value in a specified color plane in the int version
  // of an image pixel array by a specified scale factor.
  // The scaled values are clipped at 255 and 0.
  static void scaleColorPlane(int[][][] inputImageArray,
                              int plane,
                              double scale){
    int numImgRows = inputImageArray.length;
    int numImgCols = inputImageArray[0].length;
    //Scale each color value
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){

        double result =
                  inputImageArray[row][col][plane] * scale;
```

```
        if(result > 255){
          result = 255;//clip large numbers
        }//end if
        if(result < 0){
          result = 0;//clip negative numbers
        }//end if

        //Cast the result to int and put back into the
        // color plane.
        inputImageArray[row][col][plane] = (int)result;
      }//end inner loop
    }//end outer loop
  }//end scaleColorPlane
  //-----------------------------------------------------//
  /*
  The purpose of this method is to read the contents of a
   text file and to use those contents to create a 2D
    convolution filter by populating a 2D array with the
    contents of the text file.

  The text file consists of a series of lines with each
   line containing a single string of characters.

  Whitespace is allowed before and after the strings on a
   line.

  Lines containing empty strings are ignored.

  The file is allowed to contain comments, which must begin
  with //

  Comments are displayed on the standard output device.

  Comments in the text file are ignored and do not factor
   into the programming comments that follow.

  The first two strings must be convertible to type int and
   every other string must be convertible to type double.

  The first string specifies the number of rows in the 2D
  filter as type int.

  The second string specifies the number of columns in the
  2D filter as type int.

  The remaining strings specify the filter coefficients as
  type double in row-column order.

  The total number of strings must be (2 + rows*cols).
  Otherwise, the program will throw an exception and abort.

  Here are the results for a test file named Filter01.txt.
  The file contents are shown below. Note that the comment
  indicators are comment indicators in the file and are
  not comment indicators in this program.
```

```
//File Filter01.txt
//This is a test file, and this is a comment.
//This is a high-pass filter in the wave-number domain.
3
3

-1
-1
-1

-1
 8
-1

-1
//This is another comment put here for test purposes.
//There is whitespace following the next item.
-1
-1

The text output produced by the method for this input
file is shown below.

//File Filter01.txt
//This is a test file, and this is a comment.
//This is a high-pass filter in the wave-number domain.
//This is another comment put here for test purposes.
//There is whitespace following the next item.
-1.0 -1.0 -1.0
-1.0 8.0 -1.0
-1.0 -1.0 -1.0
*/
double[][] getFilter(String fileName){
  double[][] filter = new double[0][0];
  try{
    BufferedReader inData =
            new BufferedReader(new FileReader(fileName));

    String data;
    int count = 0;
    int rows = 0;
    int cols = 0;
    int row = 0;
    int col = 0;
    while((data = inData.readLine()) != null){
      if(data.startsWith("//")){
        //Display and ignore comments.
        System.out.println(data);
      }else{//Not a comment
        if(!data.equals("")){//ignore empty strings
          count++;
          if(count == 1){
            //Get row dimension value. Trim whitespace in
            // the process.
            rows = Integer.parseInt(data.trim());
          }else if(count == 2){
```

```
              //Get column dimension value. Trim whitespace
              // in the process.
              cols = Integer.parseInt(data.trim());
              //Create a new array object to be populated
              // with the remaining contents of the file.
              filter = new double[rows][cols];
            }else{
              //Populate the filter array with the contents
              // of the file. Trim whitespace in the
              // process.
              row = (count-3)/cols;
              col = (count-3)%cols;
              filter[row][col] =
                            Double.parseDouble(data.trim());
            }//end else
          }//end if on empty strings
        }//end else, not a comment
      }//end while data != null

      inData.close();//Close the input stream.

      //Display the filter coefficient values in a
      // rectangular array format.
      for(int outCnt = 0;outCnt < rows;outCnt++){
        for(int inCnt = 0;inCnt < cols;inCnt++){
          System.out.print(filter[outCnt][inCnt] + " ");
        }//end for loop
        System.out.println();//new line
      }//end for loop
    }catch(IOException e){}

    return filter;//Return the filter.
  }//end getFilter
  //----------------------------------------------------//
}//end class ImgMod33
```

Listing 31

Listing 32

```
/*File ImgMod33a.java
Copyright 2005, R.G.Baldwin

This class is identical to ImgMod33 except that it calls
ImgMod32a instead of ImgMod32.

This class provides a general purpose 2D image convolution
and color filtering capability in Java. The class is
designed to be driven by the class named ImgMod02a.

The image file is specified on the command line. The name
of a file containing the 2D convolution filter is provided
via a TextField after the program starts running.
Multiplicative factors, which are applied to the individual
color planes are also provided through three TextFields
```

```
after the program starts running.

Enter the following at the command line to run this
program:

java ImgMod02a ImgMod33a ImageFileName

where ImageFileName is the name of a .gif or .jpg file,
including the extension.

Then enter the name of a file containing a 2D convolution
filter in the TextField that appears on the screen.  Click
the Replot button on the Frame that displays the image
to cause the convolution filter to be applied to the image.
You can modify the multiplicative factors in the three
TextFields labeled Red, Green, and Blue before clicking the
Replot button to cause the color values to be scaled by
the respective multiplicative factors.  The default
multiplicative factor for each color plane is 1.0.

When you click the Replot button, the image in the top of
the Frame will be convolved with the filter, the color
values in the color planes will be scaled by the
multiplicative factors, and the filtered image will appear
in the bottom of the Frame.

Each time you click the Replot button, two additional
graphs are produced that show the following information
in a color contour map fomat:
1.  The convolution filter.
2.  The wave number response of the convolution filter.

Because the GUI that contains the TextField for entry of
the convolution filter file name also contains three
additional TextFields that allow for the entry of
multiplicative factors that are applied to the three color
planes, it is possible to implement color filtering in
addition to convolution filtering.  To apply color
filtering, enter new multiplicative scale factors into
the TextFields for Red, Green, and Blue and click the
Replot button.

Once the program is running, different convolution filters
and different color filters can be successively applied to
the same image, either separately or in combination, by
entering the name of each new filter file into the
TextField and/or entering new color multiplicative factors
into the respective color TextFields and then clicking the
Replot button

See comments at the beginning of the method named getFilter
for a description and an example of the required format for
the file containing the 2D convolution filter.

This program requires access to the following class files
plus some inner classes that are defined inside the
```

```
following classes:

ImgIntfc02.class
ImgMod02a.class
ImgMod29.class
ImgMod30.class
ImgMod32a.class
ImgMod33a.class

Tested using J2SE 5.0 and WinXP
**************************************************************/
import java.awt.*;
import java.io.*;

class ImgMod33a extends Frame implements ImgIntfc02{

  TextField fileNameField = new TextField("");
  Panel rgbPanel = new Panel();
  TextField redField = new TextField("1.0");
  TextField greenField = new TextField("1.0");
  TextField blueField = new TextField("1.0");
  Label instructions = new Label(
          "Enter Filter File Name and scale factors for " +
          "Red, Green, and Blue and click Replot");
  //--------------------------------------------------------//

  ImgMod33a(){//constructor
    setLayout(new GridLayout(4,1));
    add(new Label("Filter File Name"));
    add(fileNameField);

    //Populate the rgbPanel
    rgbPanel.add(new Label("Red"));
    rgbPanel.add(redField);
    rgbPanel.add(new Label("Green"));
    rgbPanel.add(greenField);
    rgbPanel.add(new Label("Blue"));
    rgbPanel.add(blueField);

    add(rgbPanel);
    add(instructions);
    setTitle("Copyright 2005, R.G.Baldwin");
    setBounds(400,0,460,125);
    setVisible(true);
  }//end constructor
  //--------------------------------------------------------//

  //This method is required by ImgIntfc02.  It is called at
  // the beginning of the run and each time thereafter that
  // the user clicks the Replot button on the Frame
  // containing the images.
  //The method gets a 2D convolution filter from a text
  // file, applies it to the incoming 3D array of pixel
  // data and returns a filtered 3D array of pixel data.
  public int[][][] processImg(int[][][] threeDPix,
                              int imgRows,
```

```
                                int imgCols){

    //Create an empty output array of the same size as the
    // incoming array.
    int[][][] output = new int[imgRows][imgCols][4];

    //Make a working copy of the 3D pixel array to avoid
    // making permanent changes to the original image data.
    int[][][] working3D = new int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        working3D[row][col][0] = threeDPix[row][col][0];
        working3D[row][col][1] = threeDPix[row][col][1];
        working3D[row][col][2] = threeDPix[row][col][2];
        working3D[row][col][3] = threeDPix[row][col][3];
        //Copy alpha values directly to the output. They
        // are not processed when the image is filtered
        // by the convolution filter.
        output[row][col][0] = threeDPix[row][col][0];
      }//end inner loop
    }//end outer loop

    //Get the file name containing the filter from the
    // textfield.
    String fileName = fileNameField.getText();
    if(fileName.equals("")){
      //The file name is an empty string. Skip the
      // convolution process and pass the input image
      // directly to the output.
      output = working3D;
    }else{
      //Get a 2D array that is populated with the contents
      // of the file containing the 2D filter.
      double[][] filter = getFilter(fileName);


      //Plot the impulse response and the wave-number
      // response of the convolution filter.  These items
      // are not computed and plotted when the program
      // starts running.  Rather, they are computed and
      // plotted each time the user clicks the Replot
      // button after entering the name of a file
      // containing a convolution filter into the
      // TextField.

      //Begin by placing the impulse response in the
      // center of a large flat surface with an elevation
      // of zero.This is done to improve the resolution of
      // the Fourier Transform, which will be computed
      // later.
      int numFilterRows = filter.length;
      int numFilterCols = filter[0].length;
      int rows = 0;
      int cols = 0;
      //Make the size of the surface ten pixels larger than
      // the convolution filter with a minimum size of
```

```
      // 32x32 pixels.
      if(numFilterRows < 22){
        rows = 32;
      }else{
        rows = numFilterRows + 10;
      }//end else
      if(numFilterCols < 22){
        cols = 32;
      }else{
        cols = numFilterCols + 10;
      }//end else

      //Create the surface, which will be initialized to
      // all zero values.
      double[][] filterSurface = new double[rows][cols];
      //Place the convolution filter in the center of the
      // surface.
      for(int row = 0;row < numFilterRows;row++){
        for(int col = 0;col < numFilterCols;col++){
          filterSurface[row + (rows - numFilterRows)/2]
                       [col + (cols - numFilterCols)/2] =
                                            filter[row][col];
        }//end inner loop
      }//end outer loop

      //Display the filter and the surface on which it
      // resides as a 3D plot in a color contour format.
      new ImgMod29(filterSurface,4,true,1);

      //Get and display the 2D Fourier Transform of the
      // convolution filter.

      //Prepare arrays to receive the results of the
      // Fourier transform.
      double[][] real = new double[rows][cols];
      double[][] imag = new double[rows][cols];
      double[][] amp = new double[rows][cols];
      //Perform the 2D Fourier transform.
      ImgMod30.xform2D(filterSurface,real,imag,amp);
      //Ignore the real and imaginary results.  Prepare the
      // amplitude spectrum for more-effective plotting by
      // shifting the origin to the center in wave-number
      // space.
      double[][] shiftedAmplitudeSpect =
                                  ImgMod30.shiftOrigin(amp);

      //Get and display the minimum and maximum wave number
      // values.  This is useful because the way that the
      // wave number plots are normalized. it is not
      // possible to infer the flatness or lack thereof of
      // the wave number surface simply by viewing the
      // plot.  The colors that describe the elevations
      // always range from black at the minimum to white at
      // the maximum, with different colors in between
      // regardless of the difference between the minimum
      // and the maximum.
```

```
      double maxValue = -Double.MAX_VALUE;
      double minValue = Double.MAX_VALUE;
      for(int row = 0;row < rows;row++){
        for(int col = 0;col < cols;col++){
          if(amp[row][col] > maxValue){
            maxValue = amp[row][col];
          }//end if
          if(amp[row][col] < minValue){
            minValue = amp[row][col];
          }//end if
        }//end inner loop
      }//end outer loop

      System.out.println("minValue: " + minValue);
      System.out.println("maxValue: " + maxValue);
      System.out.println("ratio: " + maxValue/minValue);

      //Generate and display the wave-number response
      // graph by plotting the 3D surface on the computer
      // screen.
      new ImgMod29(shiftedAmplitudeSpect,4,true,1);

      //Perform the convolution.
      output = ImgMod32a.convolve(working3D,filter);
    }//end else

    //Scale output color planes.  Color planes will be
    // scaled only if the corresponding scale factor in the
    // TextField has a value other than 1.0.  Otherwise,
    // there is no point in consuming computer time to do
    // the scaling.
    if(!redField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                       redField.getText());
      scaleColorPlane(output,1,scale);
    }//end if on redField

    if(!greenField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                     greenField.getText());
      scaleColorPlane(output,2,scale);
    }//end if on greenField

    if(!blueField.getText().equals(1.0)){
      double scale = Double.parseDouble(
                                      blueField.getText());
      scaleColorPlane(output,3,scale);
    }//end if on blueField

    //Return a reference to the array containing the image,
    // which has undergone both convolution filtering and
    // color filtering.
    return output;

  }//end processImg method
  //-----------------------------------------------------//
```

```
//The purpose of this method is to scale every color
// value in a specified color plane in the int version
// of an image pixel array by a specified scale factor.
// The scaled values are clipped at 255 and 0.
static void scaleColorPlane(int[][][] inputImageArray,
                            int plane,
                            double scale){
  int numImgRows = inputImageArray.length;
  int numImgCols = inputImageArray[0].length;
  //Scale each color value
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){

      double result =
                inputImageArray[row][col][plane] * scale;
      if(result > 255){
        result = 255;//clip large numbers
      }//end if
      if(result < 0){
        result = 0;//clip negative numbers
      }//end if

      //Cast the result to int and put back into the
      // color plane.
      inputImageArray[row][col][plane] = (int)result;
    }//end inner loop
  }//end outer loop
}//end scaleColorPlane
//-----------------------------------------------------//
/*
The purpose of this method is to read the contents of a
 text file and to use those contents to create a 2D
  convolution filter by populating a 2D array with the
  contents of the text file.

The text file consists of a series of lines with each
 line containing a single string of characters.

Whitespace is allowed before and after the strings on a
 line.

Lines containing empty strings are ignored.

The file is allowed to contain comments, which must begin
with //

Comments are displayed on the standard output device.

Comments in the text file are ignored and do not factor
 into the programming comments that follow.

The first two strings must be convertible to type int and
 every other string must be convertible to type double.
```

```
The first string specifies the number of rows in the 2D
filter as type int.

The second string specifies the number of columns in the
2D filter as type int.

The remaining strings specify the filter coefficients as
type double in row-column order.

The total number of strings must be (2 + rows*cols).
Otherwise, the program will throw an exception and abort.

Here are the results for a test file named Filter01.txt.
The file contents are shown below. Note that the comment
indicators are comment indicators in the file and are
not comment indicators in this program.

//File Filter01.txt
//This is a test file, and this is a comment.
//This is a high-pass filter in the wave-number domain.
3
3

-1
-1
-1

-1
 8
-1

-1
//This is another comment put here for test purposes.
//There is whitespace following the next item.
-1
-1

The text output produced by the method for this input
file is shown below.

//File Filter01.txt
//This is a test file, and this is a comment.
//This is a high-pass filter in the wave-number domain.
//This is another comment put here for test purposes.
//There is whitespace following the next item.
-1.0 -1.0 -1.0
-1.0 8.0 -1.0
-1.0 -1.0 -1.0
*/
double[][] getFilter(String fileName){
  double[][] filter = new double[0][0];
  try{
    BufferedReader inData =
            new BufferedReader(new FileReader(fileName));

    String data;
```

```
      int count = 0;
      int rows = 0;
      int cols = 0;
      int row = 0;
      int col = 0;
      while((data = inData.readLine()) != null){
        if(data.startsWith("//")){
          //Display and ignore comments.
          System.out.println(data);
        }else{//Not a comment
          if(!data.equals("")){//ignore empty strings
            count++;
            if(count == 1){
              //Get row dimension value. Trim whitespace in
              // the process.
              rows = Integer.parseInt(data.trim());
            }else if(count == 2){
              //Get column dimension value. Trim whitespace
              // in the process.
              cols = Integer.parseInt(data.trim());
              //Create a new array object to be populated
              // with the remaining contents of the file.
              filter = new double[rows][cols];
            }else{
              //Populate the filter array with the contents
              // of the file. Trim whitespace in the
              // process.
              row = (count-3)/cols;
              col = (count-3)%cols;
              filter[row][col] =
                            Double.parseDouble(data.trim());
            }//end else
          }//end if on empty strings
        }//end else, not a comment
      }//end while data != null

      inData.close();//Close the input stream.

      //Display the filter coefficient values in a
      // rectangular array format.
      for(int outCnt = 0;outCnt < rows;outCnt++){
        for(int inCnt = 0;inCnt < cols;inCnt++){
          System.out.print(filter[outCnt][inCnt] + " ");
        }//end for loop
        System.out.println();//new line
      }//end for loop
    }catch(IOException e){}

    return filter;//Return the filter.
  }//end getFilter
  //-----------------------------------------------------//
}//end class ImgMod33a
```

Listing 32

Listing 33

```
/*File ImgMod32a.java
Copyright 2005, R.G.Baldwin

This class is similar to ImgMod32 except that it uses a
different normalization scheme when converting convolution
results back to eight-bit unsigned values.  The
normalization scheme causes the mean and the RMS of the
output to match the mean and the RMS of the input.  Then it
sets negative values to 0 and sets values greater than 255
to 255.

This class provides a general purpose 2D image convolution
capability in the form of a static method named convolve.

The convolve method that is defined in this class receives
an incoming 3D array of image pixel data of type int
containing four planes. The format of this image data is
consistent with the format for image data used in the
program named ImgMod02a.

The planes are identified as follows:
0 - alpha or transparency data
1 - red color data
2 - green color data
3 - blue color data

The convolve method also receives an incoming 2D array of
type double containing the weights that make up a 2D
convolution filter.

The pixel values on each color plane are convolved
separately with the same convolution filter.

The results are normalized so as to cause the filtered
output to fall within the range from 0 to 255.

The values on the alpha plane are not modified.

The method returns a filtered 3D pixel array in the same
format as the incoming pixel array.  The returned array
contains filtered values for each of the three color
planes.

The method does not modify the contents of the incoming
array of pixel data.

An unfiltered dead zone equal to half the filter length is
left around the perimeter of the filtered image to avoid
any attempt to perform convolution using data outside the
bounds of the image.

Although this class is intended to be used to implement 2D
convolution in other programs, a main method is provided so
that the class can be tested in a stand-alone mode.  In
addition, the main method ilustrates the relationship
between convolution in the image domain and the
```

```
wave-number spectrum of the raw and filtered image.

When run as a stand-alone program, this class displays raw
surfaces, filtered surfaces, and the Fourier Transform of
both raw and filtered surfaces.  See the details in the
comments in the main method.  The program also displays
some text on the command-line screen.

Execution of the main method in this class requires access
to the following classes, plus some inner classed defined
within these classes:

ImgMod29.class - Displays 3D surfaces
ImgMod30.class - Provides 2D Fourier Transform
ImgMod32a.class - This class

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

class ImgMod32a{
  //The primary purpose of this main method is to test the
  // class in a stand-alone mode.  A secondary purpose is
  // to illustrate the relationship between convolution
  // filtering in the image domain and the spectrum of the
  // raw and filtered images in the wave-number domain.

  //The code in this method creates a nine-point
  // convolution filter and applies it to three  different
  // surfaces.  The convolution filter has a dc response of
  // zero with a high response at the folding wave numbers.
  // Hence, it tends to have the characteristic of a
  // sharpening or edge-detection filter.

  //The three surfaces consist of:
  // 1. A single impulse
  // 2. A 3x3 square
  // 3. A 5x5 square

  //The three surfaces are constructed on what ordinarily
  // is considered to be the color planes in an image.
  // However, in this case, the surfaces have nothing in
  // particular to do with color.  They simply  represent
  // three surfaces on which it is convenient to
  // synthetically construct 3D shapes that are useful for
  // testing and illustrating the image convolution
  // concepts.  But, in order to be consistent with the
  // concept of color planes, the comments in the main
  // method frequently refer to the values as color values.

  //In addition to the display of some text material on the
  // command-line screen, the program displays twelve
  // different graphs.  They are described as follows:

  //The following surfaces are displayed:
  // 1. The impulse
  // 2. The raw 3x3 square
```

```
  // 3. The raw 5x5 square
  // 4. The filtered impulse
  // 5. The filtered 3x3 square
  // 6. The filtered 5x5 square

  // In addition, a 2D Fourier Transform is computed and
  // the results are displayed for the following surfaces:
  // 1. The impulse
  // 2. The 3x3 square input
  // 3. The 5x5 square input
  // 4. The filtered impulse
  // 5. The filtered 3x3 square
  // 6. The filtered 5x5 square
  public static void main(String[] args){

    //Create a 2D convolution filter having nine weights in
    // a square.
    double[][] filter = {
                         {-1,-1,-1},
                         {-1, 8,-1},
                         {-1,-1,-1}
                        };

    //Create synthetic image pixel data.  Use a surface
    // that is sufficiently large to produce good
    // resolution in the 2D Fourier Transform.  Zero-fill
    // those portions of the surface that don't describe
    // the shapes of interest.
    int rowLim = 31;
    int colLim = 31;
    int[][][] threeDPix = new int[rowLim][colLim][4];

    //Place a single impulse in the red plane 1
    threeDPix[3][3][1] = 255;

    //Place a 3x3 square in the green plane 2
    threeDPix[2][2][2] = 255;
    threeDPix[2][3][2] = 255;
    threeDPix[2][4][2] = 255;

    threeDPix[3][2][2] = 255;
    threeDPix[3][3][2] = 255;
    threeDPix[3][4][2] = 255;

    threeDPix[4][2][2] = 255;
    threeDPix[4][3][2] = 255;
    threeDPix[4][4][2] = 255;

    //Place a 5x5 square in the blue plane 3
    threeDPix[2][2][3] = 255;
    threeDPix[2][3][3] = 255;
    threeDPix[2][4][3] = 255;
    threeDPix[2][5][3] = 255;
    threeDPix[2][6][3] = 255;

    threeDPix[3][2][3] = 255;
```

```
    threeDPix[3][3][3] = 255;
    threeDPix[3][4][3] = 255;
    threeDPix[3][5][3] = 255;
    threeDPix[3][6][3] = 255;

    threeDPix[4][2][3] = 255;
    threeDPix[4][3][3] = 255;
    threeDPix[4][4][3] = 255;
    threeDPix[4][5][3] = 255;
    threeDPix[4][6][3] = 255;

    threeDPix[5][2][3] = 255;
    threeDPix[5][3][3] = 255;
    threeDPix[5][4][3] = 255;
    threeDPix[5][5][3] = 255;
    threeDPix[5][6][3] = 255;

    threeDPix[6][2][3] = 255;
    threeDPix[6][3][3] = 255;
    threeDPix[6][4][3] = 255;
    threeDPix[6][5][3] = 255;
    threeDPix[6][6][3] = 255;

    //Perform the convolution.
    int[][][] output = convolve(threeDPix,filter);

    //All of the remaining code in the main method is used
    // to display material that is used to test and to
    // illustrate the convolution process.

    //Remove the mean values from the filtered color planes
    // before plotting and computing spectra.

    //First convert the color values from int to double.
    double[][][] outputDouble = intToDouble(output);
    //Now remove the mean color value from each plane.
    removeMean(outputDouble,1);
    removeMean(outputDouble,2);
    removeMean(outputDouble,3);

    //Convert the raw image data from int to double
    double[][][] rawDouble = intToDouble(threeDPix);

    //Get and plot the raw red plane 1.  This is an input
    // to the filter process.
    //Get the plane of interest.
    double[][] temp = getPlane(rawDouble,1);
    //Generate and display the graph by plotting the 3D
    // surface on the computer screen.
    new ImgMod29(temp,4,true,1);

    //Get and display the 2D Fourier Transform of plane 1.
    //Get the plane of interest.
    temp = getPlane(rawDouble,1);
    //Prepare arrays to receive the results of the Fourier
    // transform.
```

```
    double[][] real = new double[rowLim][colLim];
    double[][] imag = new double[rowLim][colLim];
    double[][] amp = new double[rowLim][colLim];
    //Perform the 2D Fourier transform.
    ImgMod30.xform2D(temp,real,imag,amp);
    //Ignore the real and imaginary results.  Prepare the
    // amplitude spectrum for more-effective plotting by
    // shifting the origin to the center in wave-number
    // space.
    double[][] shiftedAmplitudeSpect =
                                  ImgMod30.shiftOrigin(amp);
    //Generate and display the graph by plotting the 3D
    // surface on the computer screen.
    new ImgMod29(shiftedAmplitudeSpect,4,true,1);


    //Get and plot the filtered plane 1.  This is the
    // impulse response of the convolution filter.
    temp = getPlane(outputDouble,1);
    new ImgMod29(temp,4,true,1);

    //Get and display the transform of filtered plane 1.
    // This is the transform of the impulse response of
    // the convolution filter.
    temp = getPlane(outputDouble,1);
    real = new double[rowLim][colLim];
    imag = new double[rowLim][colLim];
    amp = new double[rowLim][colLim];
    ImgMod30.xform2D(temp,real,imag,amp);
    shiftedAmplitudeSpect = ImgMod30.shiftOrigin(amp);
    new ImgMod29(shiftedAmplitudeSpect,4,true,1);


    //Get and plot the raw green plane 2.  This is another
    // input to the filter process.
    temp = getPlane(rawDouble,2);
    new ImgMod29(temp,4,true,1);

    //Get and display the transform of plane 2.
    temp = getPlane(rawDouble,2);
    real = new double[rowLim][colLim];
    imag = new double[rowLim][colLim];
    amp = new double[rowLim][colLim];
    ImgMod30.xform2D(temp,real,imag,amp);
    shiftedAmplitudeSpect = ImgMod30.shiftOrigin(amp);
    new ImgMod29(shiftedAmplitudeSpect,4,true,1);


    //Get and plot the filtered plane 2.
    temp = getPlane(outputDouble,2);
    new ImgMod29(temp,4,true,1);

    //Get and display the transform of filtered plane 2.
    temp = getPlane(outputDouble,2);
    real = new double[rowLim][colLim];
    imag = new double[rowLim][colLim];
```

```
    amp = new double[rowLim][colLim];
    ImgMod30.xform2D(temp,real,imag,amp);
    shiftedAmplitudeSpect = ImgMod30.shiftOrigin(amp);
    new ImgMod29(shiftedAmplitudeSpect,4,true,1);


    //Get and plot the raw blue plane 3.  This is another
    // input to the filter process.
    temp = getPlane(rawDouble,3);
    new ImgMod29(temp,4,true,1);

    //Get and display the transform of plane 3.
    temp = getPlane(rawDouble,3);
    real = new double[rowLim][colLim];
    imag = new double[rowLim][colLim];
    amp = new double[rowLim][colLim];
    ImgMod30.xform2D(temp,real,imag,amp);
    shiftedAmplitudeSpect = ImgMod30.shiftOrigin(amp);
    new ImgMod29(shiftedAmplitudeSpect,4,true,1);


    //Get and plot the filtered plane 3.
    temp = getPlane(outputDouble,3);
    new ImgMod29(temp,4,true,1);

    //Get and display the transform of filtered plane 3
    temp = getPlane(outputDouble,3);
    real = new double[rowLim][colLim];
    imag = new double[rowLim][colLim];
    amp = new double[rowLim][colLim];
    ImgMod30.xform2D(temp,real,imag,amp);
    shiftedAmplitudeSpect = ImgMod30.shiftOrigin(amp);
    new ImgMod29(shiftedAmplitudeSpect,4,true,1);

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

  //The purpose of this method is to extract a color plane
  // from the double version of an image and to return it
  // as a 2D array of type double.  This is useful, for
  // example, for performing Fourier transforms on the data
  // in a color plane.
  //This method is used only in support of the operations
  // in the main method.  It is not required for performing
  // the convolution.

  public static double[][] getPlane(
                  double[][][] threeDPixDouble,int plane){

    int numImgRows = threeDPixDouble.length;
    int numImgCols = threeDPixDouble[0].length;

    //Create an empty output array of the same
    // size as a single plane in the the incoming array of
    // pixels.
    double[][] output =new double[numImgRows][numImgCols];
```

```
  //Copy the values from the specified plane to the
  // double array converting them to type double in the
  // process.
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      output[row][col] =
                        threeDPixDouble[row][col][plane];
    }//end loop on col
  }//end loop on row
  return output;
}//end getPlane
//-----------------------------------------------------//

//The purpose of this method is to get and remove the
// mean value from a specified color plane in the double
// version of an image pixel array.  The method returns
// the mean value that was removed so that it can be
// saved by the calling method and restored later.
static double removeMean(
                double[][][] inputImageArray,int plane){
  int numImgRows = inputImageArray.length;
  int numImgCols = inputImageArray[0].length;

  //Compute the mean color value
  double sum = 0;
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      sum += inputImageArray[row][col][plane];
    }//end inner loop
  }//end outer loop

  double mean = sum/(numImgRows*numImgCols);

  //Remove the mean value from each pixel value.
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      inputImageArray[row][col][plane] -= mean;
    }//end inner loop
  }//end outer loop
  return mean;
}//end removeMean
//-----------------------------------------------------//

//The purpose of this method is to get and return the
// mean value from a specified color plane in the double
// version of an image pixel array.
static double getMean(
                double[][][] inputImageArray,int plane){
  int numImgRows = inputImageArray.length;
  int numImgCols = inputImageArray[0].length;

  //Compute the mean color value
  double sum = 0;
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
```

```
        sum += inputImageArray[row][col][plane];
      }//end inner loop
    }//end outer loop

    double mean = sum/(numImgRows*numImgCols);
    return mean;
  }//end getMean
  //-----------------------------------------------------//

  //The purpose of this method is to add a constant to
  // every color value in a specified color plane in the
  // double version of an image pixel array.  For example,
  // this method can be used to restore the mean value to a
  // color plane that was removed earlier.
  static void addConstantToColor(
                              double[][][] inputImageArray,
                              int plane,
                              double constant){
    int numImgRows = inputImageArray.length;
    int numImgCols = inputImageArray[0].length;
    //Add the constant value to each color value
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        inputImageArray[row][col][plane] =
              inputImageArray[row][col][plane] + constant;
      }//end inner loop
    }//end outer loop
  }//end addConstantToColor
  //-----------------------------------------------------//

  //The purpose of this method is to scale every color
  // value in a specified color plane in the double version
  // of an image pixel array by a specified scale factor.
  static void scaleColorPlane(
      double[][][] inputImageArray,int plane,double scale){
    int numImgRows = inputImageArray.length;
    int numImgCols = inputImageArray[0].length;
    //Scale each color value
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        inputImageArray[row][col][plane] =
                  inputImageArray[row][col][plane] * scale;
      }//end inner loop
    }//end outer loop
  }//end scaleColorPlane
  //-----------------------------------------------------//

  //The purpose of this method is to convert an image pixel
  // array (where the pixel values are represented as type
  // int) to an image pixel array where the pixel values
  // are reprented as type double.
  static double[][][] intToDouble(
                                int[][][] inputImageArray){

    int numImgRows = inputImageArray.length;
    int numImgCols = inputImageArray[0].length;
```

```
  double[][][] outputImageArray =
                   new double[numImgRows][numImgCols][4];
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      outputImageArray[row][col][0] =
                            inputImageArray[row][col][0];
      outputImageArray[row][col][1] =
                            inputImageArray[row][col][1];
      outputImageArray[row][col][2] =
                            inputImageArray[row][col][2];
      outputImageArray[row][col][3] =
                            inputImageArray[row][col][3];
    }//end inner loop
  }//end outer loop
  return outputImageArray;
}//end intToDouble
//-----------------------------------------------------//

//The purpose of this method is to convert an image pixel
// array (where the pixel values are represented as type
// double) to an image pixel array where the pixel values
// are reprented as type int.
static int[][][] doubleToInt(
                           double[][][] inputImageArray){

  int numImgRows = inputImageArray.length;
  int numImgCols = inputImageArray[0].length;

  int[][][] outputImageArray =
                       new int[numImgRows][numImgCols][4];
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      outputImageArray[row][col][0] =
                        (int)inputImageArray[row][col][0];
      outputImageArray[row][col][1] =
                        (int)inputImageArray[row][col][1];
      outputImageArray[row][col][2] =
                        (int)inputImageArray[row][col][2];
      outputImageArray[row][col][3] =
                        (int)inputImageArray[row][col][3];
    }//end inner loop
  }//end outer loop
  return outputImageArray;
}//end doubleToInt
//-----------------------------------------------------//

//The purpose of this method is to clip all negative
// color values in a plane to a value of 0.
static void clipToZero(double[][][] inputImageArray,
                       int plane){
  int numImgRows = inputImageArray.length;
  int numImgCols = inputImageArray[0].length;
  //Do the clip
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
```

```
        if(inputImageArray[row][col][plane] < 0){
          inputImageArray[row][col][plane] = 0;
        }//end if
      }//end inner loop
    }//end outer loop
  }//end clipToZero
  //-----------------------------------------------------//
  //The purpose of this method is to clip all color values
  // in a plane that are greater than 255 to a value
  // of 255.
  static void clipTo255(double[][][] inputImageArray,
                        int plane){
    int numImgRows = inputImageArray.length;
    int numImgCols = inputImageArray[0].length;
    //Do the clip
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        if(inputImageArray[row][col][plane] > 255){
          inputImageArray[row][col][plane] = 255;
        }//end if
      }//end inner loop
    }//end outer loop
  }//end clipTo255
  //-----------------------------------------------------//

  //The purpose of this method is to get and return the
  // RMS value from a specified color plane in the double
  // version of an image pixel array.
  static double getRms(
                   double[][][] inputImageArray,int plane){
    int numImgRows = inputImageArray.length;
    int numImgCols = inputImageArray[0].length;

    //Compute the RMS color value
    double sumSq = 0;
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        sumSq += (inputImageArray[row][col][plane]*
                          inputImageArray[row][col][plane]);
      }//end inner loop
    }//end outer loop

    double mean = sumSq/(numImgRows*numImgCols);
    double rms = Math.sqrt(mean);
    return rms;
  }//end getRms
  //-----------------------------------------------------//

  //This method applies an incoming 2D convolution filter
  // to each color plane in an incoming 3D array of pixel
  // data and returns a filtered 3D array of pixel data.
  //The convolution filter is applied separately to each
  // color plane.
  //The alpha plane is not modified.
  //The output is normalized so as to guarantee that the
  // output color values fall within the range from 0
```

```
  // to 255.  This is accomplished by causing the mean and
  // the RMS of the color values in each output color plane
  // to match the mean and the RMS of the color values in
  // the corresponding input color plane.  Then, all
  // negative color values are set to a value of 0 and all
  // color values greater than 255 are set to 255.
  //The convolution filter is passed to the method as a 2D
  // array of type double.  All convolution and
  // normalization arithmetic is performed as type double.
  //The normalized results are converted to type int before
  // returning them to the calling method.
  //This method does not modify the contents of the
  // incoming array of pixel data.
  //An unfiltered dead zone equal to half the filter length
  // is left around the perimeter of the filtered image to
  // avoid any attempt to perform convolution using data
  // outside the bounds of the image.
  public static int[][][] convolve(
                   int[][][] threeDPix,double[][] filter){
    //Get the dimensions of the image and filter arrays.
    int numImgRows = threeDPix.length;
    int numImgCols = threeDPix[0].length;
    int numFilRows = filter.length;
    int numFilCols = filter[0].length;

    //Display the dimensions of the image and filter
    // arrays.
    System.out.println("numImgRows = " + numImgRows);
    System.out.println("numImgCols = " + numImgCols);
    System.out.println("numFilRows = " + numFilRows);
    System.out.println("numFilCols = " + numFilCols);

    //Make a working copy of the incoming 3D pixel array to
    // avoid making permanent changes to the original image
    // data. Convert the pixel data to type double in the
    // process.  Will convert back to type int when
    // returning from this method.
    double[][][] work3D = intToDouble(threeDPix);

    //Display the mean value for each color plane.
    System.out.println(
                    "Input red mean: " + getMean(work3D,1));
    System.out.println(
                  "Input green mean: " + getMean(work3D,2));
    System.out.println(
                   "Input blue mean: " + getMean(work3D,3));

    //Remove the mean value from each color plane.  Save
    // the mean values for later restoration.
    double redMean = removeMean(work3D,1);
    double greenMean = removeMean(work3D,2);
    double blueMean = removeMean(work3D,3);

    //Get and save the input RMS value for later
    // restoration.
    double inputRedRms = getRms(work3D,1);
```

```
    double inputGreenRms = getRms(work3D,2);
    double inputBlueRms = getRms(work3D,3);

    //Display the input RMS value
    System.out.println("Input red RMS: " + inputRedRms);
    System.out.println(
                      "Input green RMS: " + inputGreenRms);
    System.out.println("Input blue RMS: " + inputBlueRms);

    //Create an empty output array of the same size as the
    // incoming array of pixels.
    double[][][] output =
                     new double[numImgRows][numImgCols][4];

    //Copy the alpha values directly to the output array.
    // They will not be processed during the convolution
    // process.
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        output[row][col][0] = work3D[row][col][0];
      }//end inner loop
    }//end outer loop

//Because of the length of the following statements, and
// the width of this publication format, this format
// sacrifices indentation style for clarity. Otherwise,it
// would be necessary to break the statements into so many
// short lines that it would be very difficult to read
// them.

//Use nested for loops to perform a 2D convolution of each
// color plane with the 2D convolution filter.

for(int yReg = numFilRows-1;yReg < numImgRows;yReg++){
  for(int xReg = numFilCols-1;xReg < numImgCols;xReg++){
    for(int filRow = 0;filRow < numFilRows;filRow++){
      for(int filCol = 0;filCol < numFilCols;filCol++){

        output[yReg-numFilRows/2][xReg-numFilCols/2][1] +=
                      work3D[yReg-filRow][xReg-filCol][1] *
                                    filter[filRow][filCol];

        output[yReg-numFilRows/2][xReg-numFilCols/2][2] +=
                      work3D[yReg-filRow][xReg-filCol][2] *
                                    filter[filRow][filCol];

        output[yReg-numFilRows/2][xReg-numFilCols/2][3] +=
                      work3D[yReg-filRow][xReg-filCol][3] *
                                    filter[filRow][filCol];

      }//End loop on filCol
    }//End loop on filRow

    //Divide the result at each point in the output by the
    // number of filter coefficients.  Note that in some
    // cases, this is not helpful.  For example, it is not
```

```
    // helpful when a large number of the filter
    // coefficients have a value of zero.
    output[yReg-numFilRows/2][xReg-numFilCols/2][1] =
           output[yReg-numFilRows/2][xReg-numFilCols/2][1]/
                                     (numFilRows*numFilCols);
    output[yReg-numFilRows/2][xReg-numFilCols/2][2] =
           output[yReg-numFilRows/2][xReg-numFilCols/2][2]/
                                     (numFilRows*numFilCols);
    output[yReg-numFilRows/2][xReg-numFilCols/2][3] =
           output[yReg-numFilRows/2][xReg-numFilCols/2][3]/
                                     (numFilRows*numFilCols);

  }//End loop on xReg
}//End loop on yReg

    //Return to the normal indentation style.

    //Remove any mean value resulting from computational
    // inaccuracies.  Should be very small.
    removeMean(output,1);
    removeMean(output,2);
    removeMean(output,3);

    //Get and save the RMS value of the output for each
    // color plane.
    double outputRedRms = getRms(output,1);
    double outputGreenRms = getRms(output,2);
    double outputBlueRms = getRms(output,3);

    //Scale the output to cause the RMS value of the output
    // to match the RMS value of the input
    scaleColorPlane(output,1,inputRedRms/outputRedRms);
    scaleColorPlane(output,2,inputGreenRms/outputGreenRms);
    scaleColorPlane(output,3,inputBlueRms/outputBlueRms);

    //Display the adjusted RMS values.  Should match the
    // input RMS values.
    System.out.println(
                    "Output red RMS: " + getRms(output,1));
    System.out.println(
                  "Output green RMS: " + getRms(output,2));
    System.out.println(
                   "Output blue RMS: " + getRms(output,3));

    //Restore the original mean value to each color plane.
    addConstantToColor(output,1,redMean);
    addConstantToColor(output,2,greenMean);
    addConstantToColor(output,3,blueMean);

    System.out.println(
                   "Output red mean: " + getMean(output,1));
    System.out.println(
                 "Output green mean: " + getMean(output,2));
    System.out.println(
                  "Output blue mean: " + getMean(output,3));
```

```
    //Guarantee that all color values fall within the range
    // from 0 to 255.

    //Clip all negative color values at zero and all color
    // values that are greater than 255 at 255.
    clipToZero(output,1);
    clipToZero(output,2);
    clipToZero(output,3);

    clipTo255(output,1);
    clipTo255(output,2);
    clipTo255(output,3);

    //Return a reference to the array containing the
    // filtered pixels.  Convert the color
    // values to type int before returning.
    return doubleToInt(output);

  }//end convolve method
  //-------------------------------------------------------//
}//end class ImgMod32a
```

Listing 33

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

Keywords
Java pixel convolution filter smooth blur image jpg gif color linear DSP 3D 2D

-end-