Processing Image Pixels, Applying Image Convolution in Java

Learn 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: March 7, 2006
By Richard G. Baldwin

Java Programming, Notes # 412

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. This is also the first part of a two-part 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 several Java programs that can be used to experiment with, and to understand the effects of a wide variety of image-convolution operations.

The first lesson in the series was entitled Processing Image Pixels using Java, Getting Started. The previous lesson was entitled Processing Image Pixels, Understanding Image Convolution in Java. This lesson builds upon those earlier lessons.

Not a lesson on JAI

The lessons in this series do not provide instructions on how to use the Java Advanced Imaging *(JAI)* API. *(That will be the primary topic for a future series of lessons.)* The purpose of this series is to teach you how to implement common *(and some not so common)* image-processing algorithms by working directly with the pixels.

You will need a driver program

The lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness provided and explained a program 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. ***(ImgMod02a** displays the original and processed images in the standard format shown in Figure 57.)*

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**.

Class files required

I will be discussing several different Java programs in this lesson. One of those programs is based on a class named **ImgMod33**. To compile and execute that program, you will need access to the following class files:

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

A second Java program that I will discuss in this lesson is based on the class named **ImgMod033a**. To compile and execute that program you will need access to the following class files:

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

A third Java program that I will be discussing is based on a class named **Dsp041**. To compile and execute that program, you will need access to the following class files:

- Dsp041.class
- Graph03.class
- GraphIntfc01.class
- GUI.class

The source code for all of the above classes is provided either in this lesson or in lessons referred to in the References section of this lesson.

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 while you are reading about them.

Design rationale

In this lesson, I will walk you through the design rationale for several different types of convolution filters and show you the output produced by applying the filters to images.

Because of the limited dynamic range of the standard format for image color values, data normalization following convolution is an extremely important but seldom discussed issue. In this lesson, I will explain the design rationale for and provide examples of two different data normalization schemes.

Background Information

The earlier lesson entitled Processing Image Pixels using Java, Getting Started provided a great deal of background information as to how images are constructed, stored, transported, and rendered. I won't repeat that material here, but will simply refer you to the earlier lesson.

The earlier lesson introduced and explained the concept of a pixel. In addition, the lesson provided a brief discussion of image files, and indicated that the program named **ImgMod02a** is compatible with ***gif*** files, ***jpg*** files, and possibly some other file formats as well.

The lessons in this series are not particularly concerned with file formats. Rather, the lessons are concerned with what to do with the pixels after they have been extracted from an image file. Therefore, there is very little discussion about file formats.

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 57.
- 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 the earlier lesson entitled Processing Image Pixels using Java, Getting Started.

Concentrate on the three-dimensional array of type int

This lesson concentrates on showing you how to write image-processing programs that implement general purpose 2D image convolution. The convolution filter is read into the program from a text file, making it very easy to experimentally apply a variety of different convolution filters to the same image. The convolution programs receive raw pixel data in the form of a three-dimensional array of type **int**, and return processed pixel data in the form of a three-dimensional array of type **int**.

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.

Fundamentals

Once again, I will refer you to the earlier lesson entitled Processing Image Pixels using Java, Getting Started to learn:

- How the primary colors of red, green, and blue and the transparency of a pixel are represented by four ***unsigned*** 8-bit bytes of data.
- How specific colors are created by mixing different amounts of red, green, and blue.
- How the range of each primary color and the range of transparency extends from 0 to 255.
- How black, white, and the colors in between are created.
- How the overall color of each individual pixel is determined by the values stored in the three color bytes for that pixel, as modified by the transparency byte.

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.
- 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 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 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 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.
- 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.

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.

Preview

In this lesson, I will present, explain, and provide experimental results obtained from several major Digital Signal /Image Processing classes:

- Dsp041/Graph08
- ImgMod33/ImgMod32
- ImgMod33a/ImgMod32a

Dsp041 is a new Java class. **ImgMod33** is also a new Java class, which uses the earlier class named **ImgMod32** to perform the convolution operation and the normalization of the convolution output. **ImgMod32** was explained in the earlier lesson entitled Processing Image Pixels, Understanding Image Convolution in Java.

Graph08 is a major update to an existing program that is used for plotting.

ImgMod32a is a newly modified version of the class named **ImgMod32**, which provides an alternative approach to normalization. **ImgMod33a** is a copy of **ImgMod33** except that it uses **ImgMod32a** instead of **ImgMod32** to perform the convolution and the normalization of the convolution output.

Normalization

If color values in images were represented by values of type **double**, normalization would not be required, and therefore would not be an issue. However, color values in images are represented by eight-bit unsigned integers. As a result, normalization is required, and normalization is a very important issue. I will illustrate the importance of proper normalization in this lesson.

Convolution output values go out of their allowed range

When you convolve a 2D convolution filter with the color values on a color plane, there is a high probability that the results will include both negative values and values greater than 255, which is the maximum allowable value in an eight-bit unsigned integer. The big issue is deciding how to convert the convolution results back into values ranging from 0 to 255 inclusive *(normalization)*.

There is no one right solution to the problem. Some normalization schemes work best in some applications and other normalization schemes work best in other applications.

Two different normalization schemes

In this lesson, I will demonstrate the use of two different normalization schemes. The program combination **ImgMod33/ImgMod32** uses one scheme. The program combination **ImgMod33a/ImgMod32a** uses a different scheme. I will explain the two normalization schemes in detail in conjunction with demonstrations that illustrate their use.

The class named Dsp041

The purpose of the class named **Dsp041** is to make it easy to experiment with different time series and different convolution filters in order to understand the concepts involved in convolution filtering. In this lesson, I will use this class to explain complex image processing concepts in one dimension before proceeding to the more difficult case of two dimensions.

The class named **Dsp041** must be run under control of the class named **Graph08**. Thus, the class named **Dsp041** requires access to the class named **Graph08** and an interface named **GraphIntfc08**. **Graph08** and **GraphIntfc08** are updates to an earlier class named **Graph03** and an interface named **GraphIntfc01**. The updates allow the user to plot a maximum of eight graphs in a single display instead of a maximum of five graphs as is the case with **Graph03**.

> *(Graph03 and GraphIntfc01 were explained in an earlier lesson entitled Convolution and Matched Filtering in Java.)*

Executing Dsp041 as a program

To execute the class named **Dsp041** as a program, enter the following command at the command line:

```
java Graph08 Dsp041
```

Access to the following classes is required to compile and run this class under control of the class named **Graph08**:

- Dsp041.class
- Graph08.class
- GUI.class
- GraphIntfc08.class

The source code these classes and interfaces is provided in the section entitled Complete Program Listings.

Filtering a known waveform

The class named **Dsp041** 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 individual filters using comments and then to recompile the class in order to switch from one convolution filter to the other.)*

Five different convolution filters

The five convolution filters that are built into the class named **Dsp041** 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. 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.

Behavior of the program

These convolution filters are applied to signal waveforms having varying shapes, and in particular varying slopes. Several interesting graphic 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 decibels *(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.

*(See Figure 1 for an example of the graphic output produced by the class named **Dsp041**.)*

Normalization

The convolution algorithm used in this class emulates a one-dimensional version of the 2D image convolution algorithm used in the class named **ImgMod032**.

*(**ImgMod032** provides the convolution capability for the class named **ImgMod033**, which will be discussed later in this lesson.)*

There are two major differences between this algorithm and the 2D algorithm provided by the class named **ImgMod32**:

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 filter be flipped before it is passed to the method.

Second, whereas the 2D convolution 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 convolution algorithms assume that the incoming data consists of all positive values *(as is the case for image color values)* with regard to the normalization rationale. However, this is not a technical requirement.

The normalization scheme

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 mean value of zero. 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 algorithm computes the minimum value of the output and checks to see if it is negative. If it is negative, 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 algorithm 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.

Spectral graphs

In addition to computing and plotting the output from the convolution process, the class named **Dsp041** computes and displays the following spectral graphs in the frequency domain:

- The amplitude spectrum of the signal expressed in decibels *(db)*.
- The amplitude frequency response of the convolution filter expressed in db.
- The amplitude spectrum of the output produced by applying the convolution filter to the signal, also expressed in db.

This makes it possible for the user to relate the convolution results in the time domain with the spectral results in the frequency domain.

The class named Graph08

This is an updated version of the earlier class named **Graph03.** The update makes it possible for the user to plot up to eight functions in a single display instead of only 5 as is the case with **Graph03**.

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

This is a plotting program. It is designed to access an object instantiated from a class file that implements **GraphIntfc08**, and to plot the output from up to eight functions defined in that class file.

Required methods

The plotting surface is divided into the required number of equal sized plotting areas, and one function is plotted in Cartesian coordinates in each plotting area. The methods corresponding to the functions are named **f1**, **f2**, **f3**, **f4**, **f5**, **f6**, **f7**, and **f8**.

The class that defines the functions listed above 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 to be plotted is less than 8, then the absent method names must begin with f8 and work downward toward f1. For example, if the number of functions to be plotted is 3, then the program will expect to call methods named **f1**, **f2**, and **f3**.

The appearance of the graphic output

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually. *(See Figure 1 for an example.)*

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.

A self-test main method

The **main** method also compiles a sample class named **junk**, which implements **GraphIntfc08**, and which defines the eight methods listed above plus the method named **getNmbr**. This class is used to test the plotting capability on a stand-alone basis.

Running the program

At runtime, the name of the class that implements the interface named **GraphIntfc08** must be provided as a command-line parameter. If this parameter is not provided, 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.

User input

This class named **Graph08** provides the following text fields for user input, along with a button labeled **Graph**. This allows the user to adjust the plotting parameters and to replot the graph as many times with as many sets of plotting parameters as may be needed

- **xMin**: minimum x-axis value

- **xMax**: maximum x-axis value
- **yMin**: minimum y-axis value
- **yMax**: maximum y-axis value
- **xTicInt**: tic mark interval on the x-axis
- **yTicInt**: tic mark interval on the 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.

Behavior of the Graph button

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. 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)*.

The classes named ImgMod33 and ImgMod33a

The classes named **ImgMod33** and **ImgMod33a** are the primary classes for which this lesson was written. Each of these classes provides a general purpose 2D image convolution and color filtering capability in Java. 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. I will explain **ImgMod33** and **ImgMod33a**, and the differences between the two in this lesson.)*

The image file to be processed through convolution is specified on the command line.

Convolution filters are provided as text files

The name of a file containing the 2D convolution filter is provided via a **TextField** on an interactive control panel after the program starts running.

> *(See Figure 4 for an example of the interactive control panel containing four **TextField**s.)*

Different convolution filter files can be specified and applied to the image without a requirement to restart the program for each new filter.

Color filtering

Multiplicative factors, which are applied to the individual color planes following convolution and normalization, are also provided through three **TextField**s on the interactive control panel after the program starts running.

Running the programs

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

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

Then enter the name of a file containing a 2D convolution filter in the **TextField** that appears in the interactive control panel. Click the **Replot** button on the **Frame** that displays the image to cause the convolution filter to be applied to the image.

> *(See Figure 3 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.)*

Color filtering

You can modify the multiplicative factors in the three **TextFields** labeled **Red**, **Green**, and **Blue** in the interactive control panel before clicking the **Replot** button to cause the corresponding 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 contained in the specified file.
- The color values in the color planes will be scaled by the corresponding multiplicative factors after the convolution has been completed and the image has been normalized.
- The filtered image will appear in the bottom of the **Frame**.

> *(Figure 6 shows the result of reducing the Green and Blue multiplicative factors each to 0.5 and clicking the Replot button.)*

Wave number data

Each time you click the **Replot** button, two additional graphs are produced that show the following information in a color contour map format:

- The 2D convolution filter.
- The wave number response of the 2D convolution filter.

(Note that the maps appear on top of one another. You must move the one on the top to see the one on the bottom.)

Testing

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

Experimental Results

Before getting into the programming details for the programs presented in this lesson, I will show you some experimental results produced using those programs. My objective is to teach you what happens when you convolve a 2D convolution filter with an image. I will present and discuss the results for several different types of convolution filters:

- A simple copy filter *(See Figure 3.)*
- Smoothing or softening filters *(See Figure 55.)*
- Bipolar filters
    - Embossing filters that produce a 3D-like effect *(See Figure 57.)*
    - Edge detection filters *(See Figure 59.)*
    - Sharpening filters *(See Figure 54.)*

    (The last three types of filters in the above list could all be considered to be special cases of edge detection filters. More generally, they are filters having both positive and negative coefficient values.)

I will also teach you about two alternative normalization schemes.

Two normalization schemes

As mentioned earlier, normalization is a very important issue in image convolution. I will begin by using the classes **ImgMod33** and **ImgMod32** and the normalization scheme embodied in the class named **ImgMod32**. I will continue using those two classes until I notify you that I am switching to the use of the classes **ImgMod33a** and **ImgMod32a**.

By making the switch, I will be switching to the normalization scheme embodied in the class named **ImgMod32a**. At about that point, I will explain both normalization schemes in detail and I will explain the reasons for switching from one to the other.

Start simple in one dimension

In many cases, I will begin my explanation of a 2D convolution experiment with an explanation of a simplified version of the convolution process based on a one-dimensional convolution filter. Following that explanation, I will proceed to the more complex case based on a 2D convolution filter. My hope is that by first understanding the simplified one-dimensional case, you will be better prepared to understand the more complex 2D case.

In the one-dimensional case, I will relate convolution in the time domain to multiplicative filtering in the frequency domain, and explain the ramifications of that relationship.

In the two-dimensional case, I will relate convolution in the image domain to multiplicative filtering in the wave number domain.

A simple copy filter

I will begin with the simplest convolution filter that I know how to devise. This is a convolution filter consisting of a single impulse. I will explain this filter in the time domain, the frequency domain, the image domain, and the wave number domain. Along the way, I will prepare you to understand the different kinds of output displays that are produced by the programs being used.

The one-dimensional case

I will begin with a one-dimensional explanation in the time and frequency domains as shown in Figure 1.

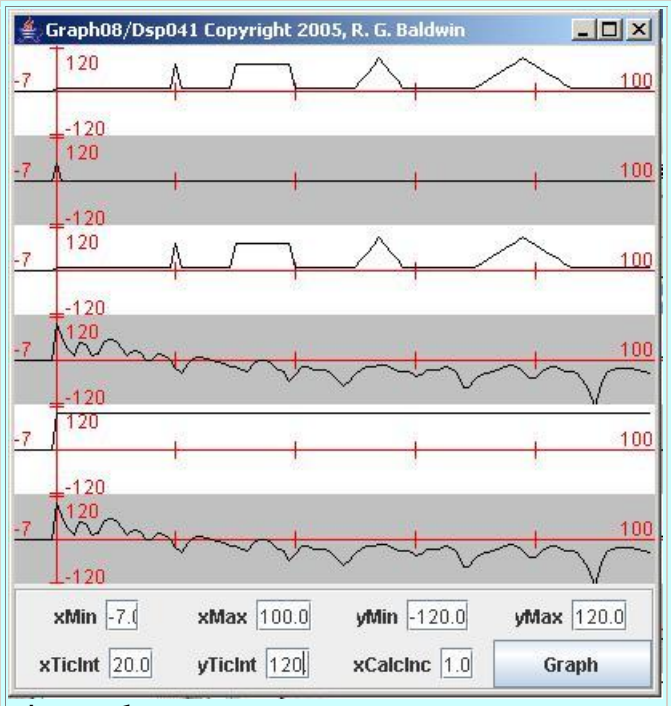

Figure 1

Six different graphs in one display

Figure 1 shows six different graphs, each plotted in Cartesian coordinates within its own plotting area.

> *(This is the type of graphic output produced by the class named **Dsp041** working in conjunction with the class named **Graph08**.)*

The individual plotting areas are alternately colored white and grey to make it easier to separate them visually. Going from top to bottom, the graphs show:

1. The signal waveform as a time series.
2. The convolution filter waveform as a time series.
3. The output waveform. This is the result of applying the convolution filter to the signal. The output waveform includes the impulse response of the filter.
4. The amplitude spectrum of the signal expressed in decibels *(db)*.
5. The amplitude frequency response of the convolution filter expressed in db.
6. The amplitude spectrum of the output expressed in db.

The signal waveform

The signal waveform shown in the top graph in Figure 1 is the waveform that will be used for all of the one-dimensional examples in this lesson. Going from left to right, the signal contains the following wavelets:

- An impulse consisting of a single value.
- A rectangular pulse.
- A triangular pulse with a high slope.
- A triangular pulse with a smaller slope.

All of these wavelets are made up of positive values and are riding on a positive, non-zero baseline. The intent is to start with an idealized version of the kinds of values that might be found in a single row of pixels in a single color plane in an image.

> *(Note that although the rectangular pulse appears to have sloping sides, that is an artifact of the plotting process. A straight line is drawn from the last value in the baseline before the pulse and the first value in the pulse. Because there is some distance between the two, the line has a slope. Also note that the sides of the triangles are straight lines. The small deviations from a straight line are also artifacts of the plotting process.)*

The convolution filter

For this case, the convolution filter consists of an impulse, or a single value shown at the origin in the second graph from the top.

The output from the convolution process

The output from the convolution process is shown in the third graph from the top. The convolution of an impulse with a time series simply reproduces the time series. Thus, the output produced by convolving the impulse with the signal in this case looks just like the signal.

> *(Convolving a convolution filter with an impulse produces the impulse response of the filter. Thus, the first feature in the output graph is the so-called impulse*

response of the filter, which in this case, is just another impulse. For subsequent experiments, however, the impulse response will have a different waveform.)

The spectral data

Convolution in the time domain is equivalent to multiplication in the frequency domain. In other words, when you convolve a convolution filter with a time series, the frequency spectrum of the result is the same as the product of the frequency spectrum of the time series and the frequency response of the filter.

A peak at zero frequency

The frequency spectrum of the signal is shown in the fourth graph in Figure 1. All of the spectral displays using this format in this lesson will extend from a frequency of zero to the Nyquist folding frequency, which is one-half the sampling frequency.

(As you can see, the spectrum has a peak at a frequency of zero. This is because all of the signal values are positive and therefore, the mean value of the signal is positive. The spectrum analysis process measures the mean value of the signal and uses the result of that measurement to represent the spectral value at a frequency of zero.)

Peaks and valleys

The spectrum of the signal contains various peaks and valleys. It also has a general downward slope off to the right *(the direction of increasing frequency)*. The absolute shape of the signal spectrum won't mean much to us except to the extent that the spectrum of the output is a modified version of the spectrum of the signal.

Frequency response of the convolution filter

The frequency response of the convolution filter is shown in the fifth graph in Figure 1.

(Note that all three spectral graphs are plotted in decibels or db. This was done to preserve the plotting dynamic range. To make a long story short, converting to decibels means multiplying a value by a constant, and replacing the resultant product by the log to the base 10 of that product. I discussed this to some extent in the earlier lessons entitled Plotting 3D Surfaces using Java, and Adaptive Filtering in Java, Getting Started. You will also find a good discussion of decibels on the Wikipedia web site.)

Frequency spectrum of the output

The frequency spectrum of the output time series produced by the convolution process is shown in the sixth graph.

The frequency response of the convolution filter in the fifth graph in Figure 1 is perfectly flat *(the frequency spectrum of an impulse is always flat)*. Thus, the product of the frequency response of the filter in the fifth graph and the spectrum of the signal in the fourth graph simply reproduces the spectrum of the signal as shown in the sixth graph in Figure 1.

The corresponding 2D example

The left panel in Figure 2 shows a 2D convolution filter consisting of a single impulse.

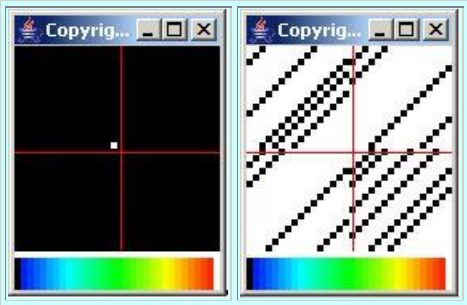

Figure 2

(*The plotting format used in Figure 2 was discussed in detail in the earlier lesson entitled Plotting 3D Surfaces using Java. Briefly, Figure 2 contains a pair of 3D plots with the elevation represented by color. The elevation color scale is shown at the bottom of each plot. Black and dark blue represent the lowest elevations. Red and white represent the highest elevations. Yellow, green, and light blue represent the elevations in between.*)

Thus, the single impulse in the 2D convolution filter shown in Figure 2 is represented by a single value at the maximum elevation *(white)* protruding from a surface that is otherwise at the lowest elevation *(black)*.

The wave number response

The right panel in Figure 2 shows the wave number response of the convolution filter in the left panel. *(This is analogous to the frequency response of the one-dimensional convolution filter in Figure 1.)* Although it isn't very pretty, this is a flat wave number response. The ratio of the highest to the lowest points in the right panel of Figure 2 is 1.0000000000000002.

The output image

The bottom panel in Figure 3 shows the result of applying this convolution filter to the image in the top panel of Figure 3.

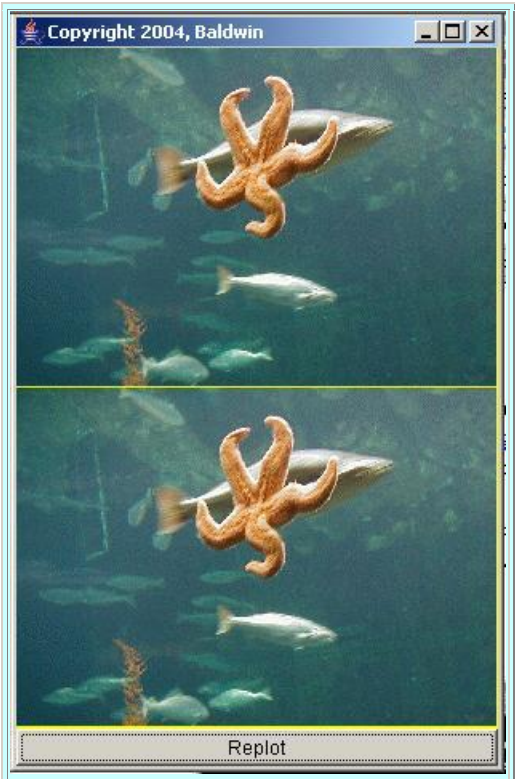

Figure 3

As you can see, this convolution filter consisting of a single impulse simply copies the input image into the output. The only differences between the two are differences resulting from computational inaccuracies.

Possible display problems

The display screen on my HP laptop computer causes colors to become progressively lighter going down the screen from top to bottom. Therefore, the image in the bottom panel of Figure 3 looks lighter than the image in the top panel of Figure 3 on my computer. However, if I copy the entire frame containing both images out to a graphics program and turn the frame upside down, the bottom image, which looked darker when it was at the top looks lighter when it is at the bottom.

If the bottom image in Figure 3 doesn't match the top image on your computer, your display may suffer from some similar problem.

An interactive control panel

The interactive control panel produced by the classes named **ImgMod33** and **ImgMod33a** is shown in Figure 4.

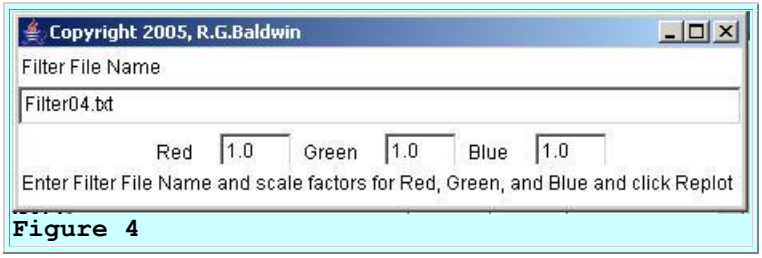

Figure 4

As you can see from the top **TextField** in Figure 4, this 2D convolution filter was stored in and retrieved from a file named **Filter04.txt**.

> *(I will come back and discuss the fields named Red, Green, and Blue later in this lesson.)*

Contents of the filter file

The actual contents of the filter file named Filter04.txt are shown in Figure 5.

```
//File Filter04.txt
//A single impulse copy filter
1
1

1
```
Figure 5

Briefly, the first two lines in Figure 5 are comments that are ignored when reading the filter values from the file. The blank line is a separator, which is also ignored when reading the filter values from the file. The third and fourth lines specify that the filter has one row and one column in that order. The last line with a value of 1 specifies the value of the single filter coefficient to be 1. The coefficient value is converted to type **double** by the program and therefore, is interpreted to have a value of 1.0.

Color filtering

As you can see in Figure 4, the multiplicative values for Red, Green, and Blue were at their default values of 1.0 when the image of the interactive control panel was captured. Therefore, no color filtering was applied to the image in the bottom panel of Figure 3.

Figure 6 shows the result of reducing the Green and Blue multiplicative factors each to 0.5 and clicking the **Replot** button.

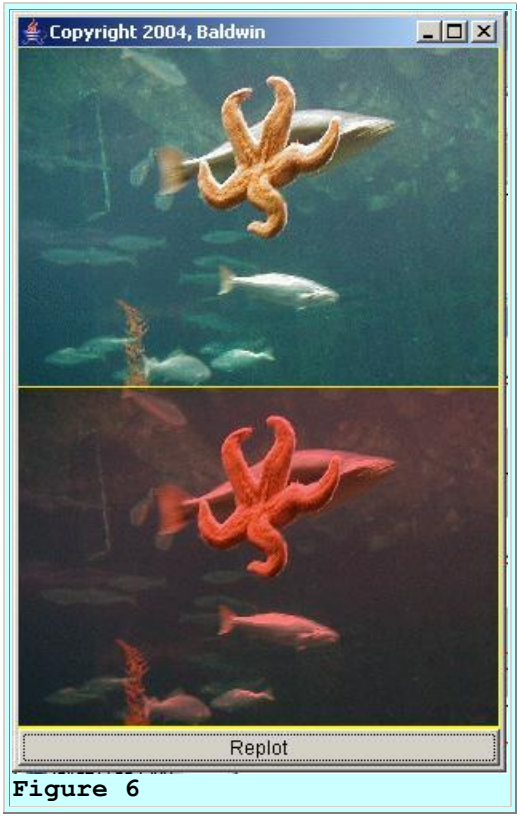

Figure 6

This caused the green and blue components of every pixel to be reduced to one-half of their original value, leaving the red component to dominate the image. As you will see later, image convolution can sometime produce undesirable color changes. In those cases, it may sometimes be possible to apply color filtering following convolution as described above to correct the problem.

A smoothing or softening filter

The next convolution filter that we will examine is a smoothing or softening filter. We will examine a four-point convolution filter in the one-dimensional time domain. Then we will examine a 1x4 convolution filter in the 2D image domain, followed by a 4x4 convolution filter in the 2D image domain. In all three cases, all of the filter coefficients that make up the convolution filter have the same positive value.

> *(Having the same value is not a requirement of smoothing filters. Although the coefficients in smoothing filters almost always have positive values, smoothing filters can be designed having a wide variety of combinations of coefficient values. Each combination of coefficient values provides somewhat different results.)*

The one-dimensional case

The third graph in Figure 7 shows the result of applying a four-point convolution filter to the signal in the first graph. The convolution filter is shown by the small flat-topped pulse at the left end of the second graph.

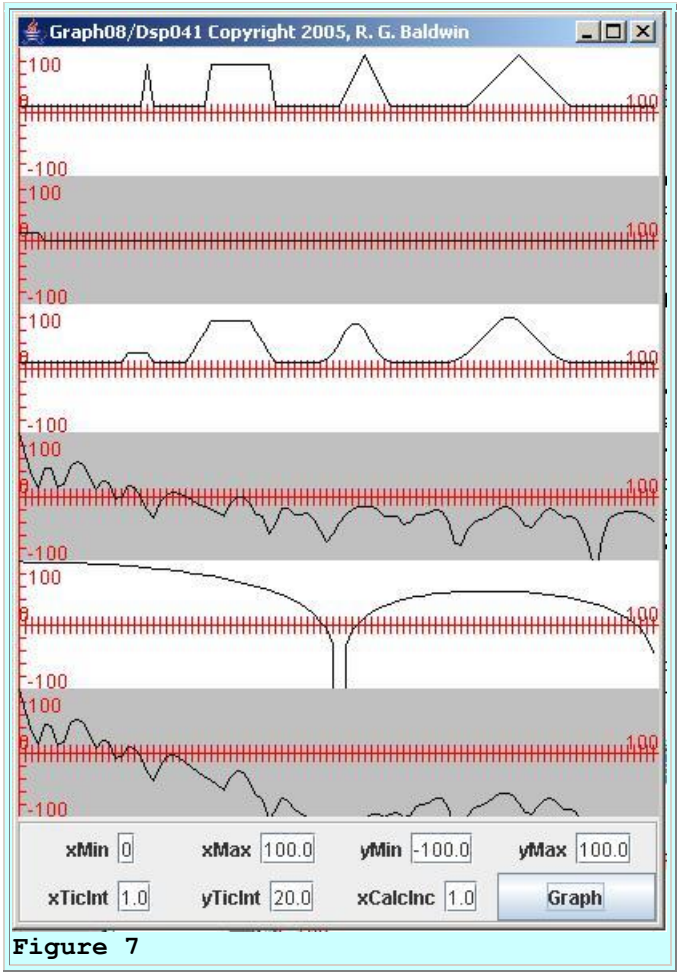

Figure 7

The convolution filter in Figure 7 consists of four coefficients, each having a value of 0.25. As the filter moves across the signal, the effect is to produce each output sample as the average of four consecutive input samples. Thus, the process tends to average out or smooth out the bumps in the signal.

The impulse response

For example, the single impulse at the left end of the input signal results in a replica of the convolution filter in the output. As mentioned earlier, this is often referred to as the *impulse response* of the convolution filter.

In effect, the impulse in the signal is turned into a plateau having a lower amplitude in the output. If you consider this to be image color data, the impulse would represent a single very bright pixel in the input. The output would consist of a line of four pixels having much less brightness.

A filtered rectangular pulse

The application of the convolution filter to the rectangular pulse in the input produced an output pulse of the same height. However, the base of the output pulse is broader than the input and the top of the output pulse is narrower than the input.

If you consider this to be color data, the input would represent a bright line ten pixels in length against a dark background. The total length of this feature in the output would be longer than ten pixels *(13 pixels)*, but the bright portion would be shorter than ten pixels *(7 pixels)*. Each end of the line would progress from dark to bright, or from bright to dark in a gradual, but linear fashion.

A filtered triangle

The application of the smoothing filter to the two triangular pulses caused the base of each output waveform to be broader than the base of the input. It also caused the sharp corners to be rounded or softened. *(Hence the commonly used name of a softening or smoothing filter.)* When viewed as color data, for both triangles, the intensity of the color of the input would transition from dark to bright and back to dark in a linear fashion with abrupt transitions at the beginning, the middle, and the end. The transitions from dark to bright and from bright to dark would be much smoother in the output. The transitions would also be longer.

The spectral results

Since the input signal in Figure 7 is the same as the input signal in Figure 1, the frequency spectrum of the input signal shown in the fourth graph in Figure 7 hasn't changed.

> *(However, I did expand the vertical scale of Figure 7 relative to Figure 1, so it may look a little different.)*

Multiplication in the frequency domain

I told you earlier that the output spectrum shown in the sixth graph in Figure 7 should be the product of the filter response shown in the fifth graph and the input spectrum shown in the fourth graph. I now need to qualify that statement.

Decibel addition is equivalent to multiplication

It is true that convolution in the time domain is equivalent to multiplication in the frequency domain. However, the spectral values in the graph shown in Figure 7 underwent a logarithmic transformation prior to plotting in order to preserve the plotting dynamic range. *(The raw spectral values were converted to decibels prior to plotting.)* If you are familiar with logarithms, you may recall that the addition of values that have undergone a logarithmic transformation is equivalent to the multiplication of the raw data.

(For example, one way to multiply two numbers is to compute the logarithm of each number, add the logarithmic values, and then compute the so-called antilogarithm of the sum. The result will be the product of the two original numbers.)

Need to add results expressed in decibels

Thus, when the individual spectra are viewed in decibel form, the spectral output of a convolution process is obtained by adding the spectrum of the input and the spectral response of the convolution filter.

(When plotting the result, it is often necessary to slide the result up or down on the page to make it fit in the plotting window. Sliding a decibel plot up or down on the page is equivalent to multiplying every raw value in the plot by the same constant value.)

As you can see, the spectrum of the output shown in the sixth graph in Figure 7 is the sum of the fourth and fifth graphs in Figure 7.

The decibel scale

Now let's discuss the significance of the vertical scale on the spectral plots in Figure 7. The values of +100 and -100 occur at the transitions between white and grey in Figure 7. Thus, the plotting area for each graph extends from 100 units below the axis *(-100)* to 100 units above the axis *(+100)*. Each tic mark on the vertical axis in Figure 7 represents 20 units.

In preparation for plotting, the data was scaled so that each plotting unit would represent one-fourth of a decibel. Thus, 100 plotting units represents 25 decibels and each tic mark represents 5 decibels.

Relationship of decibels to the real world

Each three-decibel change in the spectral response represents a doubling or halving of the power at that frequency. Thus, when the frequency response in the fifth graph drops from 100 units at the origin to about 50 units at the top of the first lobe to the right, that corresponds to a reduction of the response by about 12.5 db. This, in turn, corresponds to a reduction in the power in the output relative to the power in the input at that frequency by a factor of about sixteen. Thus, small changes in a decibel plot represent large changes in power in the real world. That is why the logarithmic decibel scale is chosen to preserve plotting dynamic range.

The filter response

The filter response in the fifth graph in Figure 7 is down by at least 11 or 12 db at all frequencies greater than about twenty-percent of the sampling frequency. This means that the power in the output at those frequencies will be significantly reduced relative to the power in the input.

It is a well-known fact that in order for the values in a time series to make rapid transitions from low values to high values and back to low values, the time series must contain significant high-frequency components. Thus, the elimination of high-frequency components by the convolution filter eliminates the possibility of such rapid transitions. This is evidenced by comparing the first and third graphs in Figure 7. *(The transitions take longer to occur in the output than is the case in the input.)*

Extension to image data

Extending this concept to images, the elimination of high wave number components by filtering the image using a convolution filter eliminates the ability of the color values in the image to make rapid transitions. This, in turn causes the transitions to become smoother or softer. In fact, as you will see later, extreme elimination of high wave number components can cause the image to appear to be significantly out of focus with no sharp edges anywhere in the image.

Applying a smoothing filter to an image

Now, let's examine the result of applying a one-dimensional smoothing filter to a 2D image. The following experiment was performed using the class named **ImgMod33**.

The filter and the wave number response of the filter are shown in Figure 8. The filter is shown in the left panel and the wave number response is shown in the right panel.

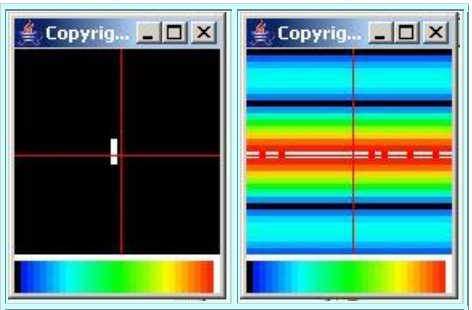

Figure 8

The filter coefficient values

The contents of the text file *(Filter07.txt)* containing this filter are shown in Figure 9.

```
//File Filter07.txt
//One-dimensional soft smoothing filter

4
1

1
1
1
1
```

```
Figure 9
```

As you can see from the left panel of Figure 8 and from the first two non-comment lines in Figure 9, this filter is defined in four rows with one column. There are four filter coefficients, each having a value of 1.0.

> *(Note that the 2D convolution algorithm used by **ImgMod33** divides each sum of products by the number of filter coefficients. Therefore, it isn't necessary to scale the coefficient values down to 0.25 as was the case for the one-dimensional filter in Figure 7.)*

The wave number response

The wave number response of the convolution filter shown in the right panel of Figure 8 is a 2D version of the frequency response of the filter shown in the fifth graph of Figure 7. The elevation values for a vertical slice taken through the center of the wave number response in Figure 8 would be very similar to the frequency response shown in Figure 7.

> *(However, the wave number response was not converted to decibels prior to display in Figure 8 as was the case for the frequency response in Figure 7. This would cause the two to have a different appearance.)*

Wave number range covered

The response at a wave number of zero is shown at the center of the right panel of Figure 8. The wave number response extends to the Nyquist folding wave number at the North, South, East, and West edges of the panel.

Peaks and troughs in the wave number response

The red and white horizontal band in the center of the wave number response in Figure 8 is analogous to the peak at zero frequency in the frequency response at the left end of the fifth graph in Figure 7.

> *(The highest value in the wave number response in Figure 8 is represented by white with red coming in as a close second. The lowest value is represented by black. See the color scale at the bottom of the image.)*

What about the left end of the fifth graph in Figure 7?

The fifth graph in Figure 7 is analogous to only one-half of a vertical slice through the center of Figure 8. For the two to be completely analogous, we would need to construct a mirror image of the fifth graph in Figure 7 and attach it to the left end of the graph in Figure 7. This would produce a graph that is symmetric about its center in the same way that a vertical slice through the wave number response in Figure 8 is symmetric about its center.

Deep troughs

The two black bands closest to and on either side of the red band in the wave number response in Figure 8 are analogous to the deep trough shown to the right of the peak in the frequency response in the fifth graph in Figure 7.

The black band at the top and the dark blue band at the bottom of the wave number response in Figure 8 are analogous to the developing trough at the right edge of the fifth graph in Figure 7.

Secondary peaks

The light blue bands near the top and bottom edges of the wave number response in Figure 8 are analogous to the peak in the frequency response about three-fourths of the way across the fifth graph in Figure 7.

Infer some conclusions

From the wave number response of the filter shown in Figure 8, we can infer the following:

1. Transitions along edges in the image that are parallel to the red band will experience the maximum amount of smoothing.
2. Transitions along edges that are perpendicular to the red band will not experience any smoothing at all.
3. Transitions along edges that are at some other angle relative to the red band will experience some amount of smoothing, with the amount of smoothing increasing as the edge becomes more nearly parallel to the red band.

A filtered Stick Man

Let's see if these conclusions are borne out in reality. Figure 10 shows the application of this convolution filter *(Filter07.txt)* to an image *(stickman2.gif)* of a black Stick Man on a white background. *(As you will see later, transitions from white to black behave differently from transitions from black to white in some cases.)* The image has lots of sharp edges at different angles.

> *(In the previous paragraph, I made reference to an image file named stickman2.gif. For my own records, I will frequently refer to such files so that I will be able to identify the file in the event that I need to repeat the experiment sometime in the future.)*

The top panel in Figure 10 shows the original image. The bottom panel shows the image that resulted from performing the convolution.

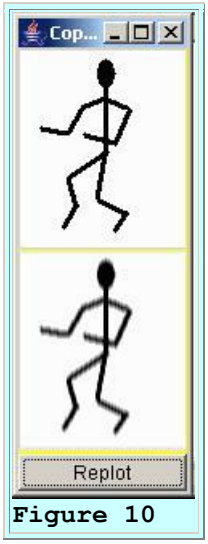

Figure 10

Lots of sharp edges

The fact that this image contains sharp edges with transitions from white to black, and from black to white indicates that the color values exhibit large transitions at different points in the image. Generally, the white background indicates high color values and the black areas indicate low color values.

Fuzzy forearms and shoulders

Note first the fuzziness of the forearms and the shoulders in the output image. The edges of the forearms and shoulders are almost parallel to the red band in the wave number response of Figure 8. The fuzziness at these edges indicates that the high wave number components required to support these transitions have been significantly reduced. This agrees with the first conclusion listed above.

No fuzz on the torso

Now note the torso, which is perpendicular to the red band in the wave number response. The edges of the torso are still crisp and free of fuzz. Therefore, those edges must still have their high wave number components. This agrees with the second conclusion listed above.

Now note the arms, legs, and feet, for which the edges are at various angles relative to the red band in the wave number response in Figure 8. These edges exhibit differing amounts of fuzz, depending on the angle between the respective edge and the red band in the wave number response in Figure 8. This agrees with the third conclusion listed above.

Input versus output wave number spectra

I would like very much to be able to show you the wave number spectrum of the input and the output so that you can see that the output wave number spectrum is the product of the input wave number spectrum and the wave number response of the convolution filter.

However, the time required to compute and display such a wave number spectrum is prohibitive on my computer, even for a small image like the Stick Man. To begin with, it is necessary to compute the wave number spectrum for each of three color planes in order to make any sense out of the results. Beyond that, the computation of the wave number spectrum for only one color plane requires more time than my limited patience will allow.

A true 2D convolution

Figure 10 shows the result of performing a convolution between a 1x4 convolution filter and a 2D image. Thus, although the 2D convolution algorithm was used, the experiment wasn't fully a 2D experiment because the filter wasn't a 2D filter.

Figure 11 shows the filter and its wave number response for a true 2D version of the four-point smoothing filter.

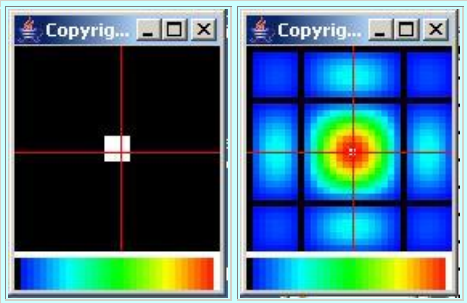

Figure 11

A 4x4 convolution filter

The filter *(Filter05.txt)* shown in Figure 11 consists of a 4x4 block of filter coefficients, each having a value of 1.0.

Wave number response comparison

If you compare the wave number response in Figure 11 with the wave number response in Figure 8, you should see a strong correlation between the two. A horizontal or vertical slice through the center of the wave number response in Figure 11 generally matches a vertical slice through the center of the wave number response in Figure 8.

> *(However, a horizontal slice through the center of the wave number response in Figure 8 is perfectly flat and doesn't have any resemblance to a horizontal slice through the center of the wave number response in Figure 11.)*

Suppress high wave number components at all angles

The wave number response in Figure 11 indicates that this filter will suppress the high wave number components necessary to support any transition edge in an image, regardless of the angle of that edge relative to the horizontal.

Also, because there are no light blue areas on the diagonals, high wave number suppression at those angles will probably be more severe than for edges that are horizontal or vertical.

(It isn't likely that we will be able to see the difference between suppression on the diagonals and suppression on the horizontal and vertical axes.)

A 2D smoothing convolution on the Stick Man

The bottom panel of Figure 12 shows the result of applying this 2D convolution filter *(Filter05.txt)* to the 2D Stick Man image *(stickman2.gif)* in the top panel.

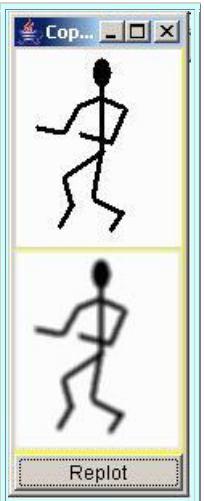

Figure 12

The result of the smoothing operation is obvious. As indicated above, the amount of smoothing is generally independent of the angle of the edge relative to the horizontal.

As mentioned earlier, the amount of smoothing that will be experienced is dependent on the design of the 2D convolution filter. Before leaving the Stick Man, I want to show you the result of applying a more severe smoothing filter to the Stick Man.

A pyramid-shaped convolution filter

The convolution filter and its wave number response for this case are shown in Figure 13. As before, the filter *(Filter03.txt)* is shown in the left panel and the wave number response is shown in the right panel.

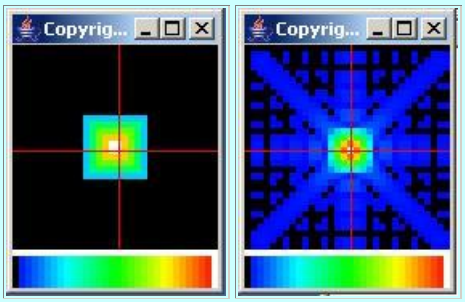

Figure 13

A 10x10 filter

This is a 10x10 convolution filter, and as mentioned above, it has the shape of a pyramid *(as opposed to a block, which was the case for the filter shown in Figure 11).* The value for each of the coefficients is shown in Figure 14.

```
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0
1.0 2.0 3.0 3.0 3.0 3.0 3.0 3.0 2.0 1.0
1.0 2.0 3.0 4.0 4.0 4.0 4.0 3.0 2.0 1.0
1.0 2.0 3.0 4.0 5.0 5.0 4.0 3.0 2.0 1.0
1.0 2.0 3.0 4.0 5.0 5.0 4.0 3.0 2.0 1.0
1.0 2.0 3.0 4.0 4.0 4.0 4.0 3.0 2.0 1.0
1.0 2.0 3.0 3.0 3.0 3.0 3.0 3.0 2.0 1.0
1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
```

Figure 14

The wave number response of the 10x10 pyramid filter

If you compare the wave number response in Figure 13 with the wave number response in Figure 11, you will see that this is a much more severe filter with respect to the suppression of high wave number components. The red and yellow area indicating the peak at a wave number of zero in Figure 13 is very small, indicating that the response falls off very quickly with increasing wave number.

> *(Even the green and light blue area surrounding the peak in Figure 13 isn't much larger than the red area in Figure 11.)*

Outside of the small light blue area, everything is either dark blue or black. Energy at all wave numbers outside the small red, yellow, and green area will be suppressed, and all energy outside the small light blue area will be suppressed in a very significant way.

The poor Stick Man

This assessment is borne out by the filtered Stick Man in the bottom panel of Figure 15.

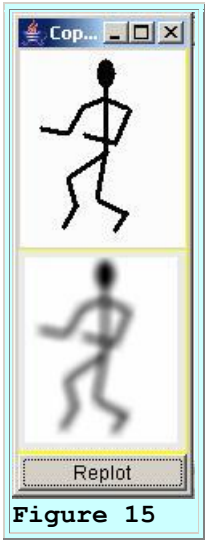

Figure 15

Applying this filter *(Filter03.txt)* to the image in the top panel of Figure 15 caused the poor Stick Man to be reduced to a mere shadow of himself. There isn't a sharp edge anywhere in the image. He looks completely soft and furry as a result of applying this severe softening filter. This is what happens when you suppress the high wave number components in an image in a very significant way.

The Stick Man becomes a ghost

Just for fun, I'm going to show you one more image *(stickman2a.gif)* involving the Stick Man and the 10x10 smoothing filter. However, in this case, the subject will be a white Stick Man on a black background instead of a black Stick Man on a white background.

The bottom panel in Figure 16 shows the result of applying the same 10x10 smoothing filter to the Stick Man shown in the top panel of Figure 16.

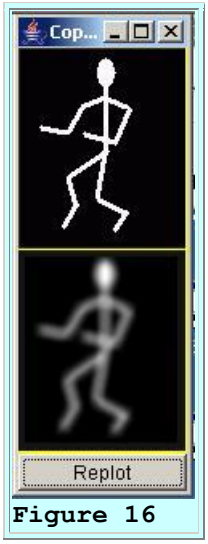

Figure 16

In this case, the Stick Man was turned into a ghost.

Enough already!

Although the Stick Man is very useful for illustrating image convolution concepts, we rarely have real images that are as clean and uncluttered as the Stick Man. Before leaving the topic of smoothing or softening filters, I want to show you the results of applying the same smoothing filters to a real photographic image *(background02.gif)*.

Applying the 4x1 smoothing filter

Figure 17 shows the result of applying the 4x1 smoothing filter contained in the file named Filter07 to the image.

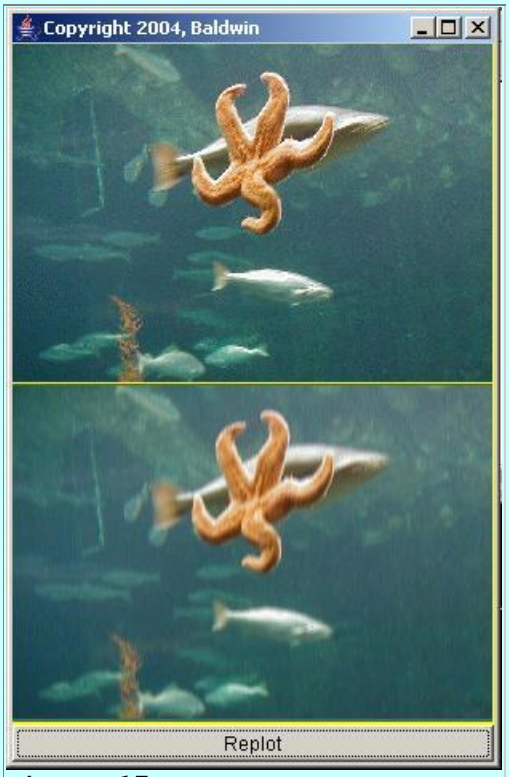

Figure 17

This is the same filter that was applied to the Stick Man in Figure 10. As we would expect from what we learned earlier, the output image in the bottom panel of Figure 17 exhibits some fuzz on the horizontal edges *(although clearly not as obvious as with the Stick Man in Figure 10)*. Figure 17 exhibits little or no new fuzz on the edges that are near to the vertical.

Applying the 4x4 smoothing filter

Figure 18 shows the result of applying the 4x4 smoothing filter *(Filter05.txt)* to the same image of the starfish. This is the same filter that was applied to the Stick Man in Figure 12.

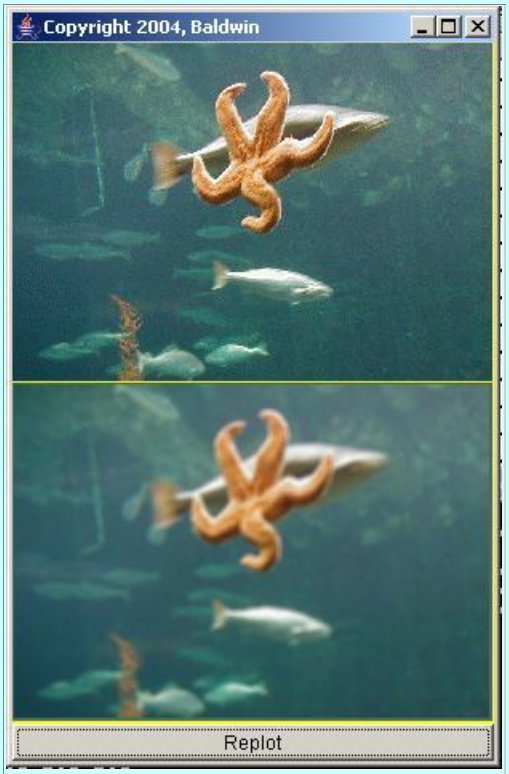

Figure 18

As with the Stick Man in Figure 12, all of the edges have been softened in Figure 18, regardless of their angle relative to the horizontal. If we were to reduce the filter to a 3x3 filter, or perhaps a 2x2 filter, the degree of softening would be less. If we were to increase the filter to a 5x5 or a 6x6, the degree of softening would be greater.

Applying the 10x10 smoothing filter

Figure 19 shows the result of applying the 10x10 filter *(Filter03)* to the image of the starfish. This is the same filter that was applied to the Stick Man in Figure 15.

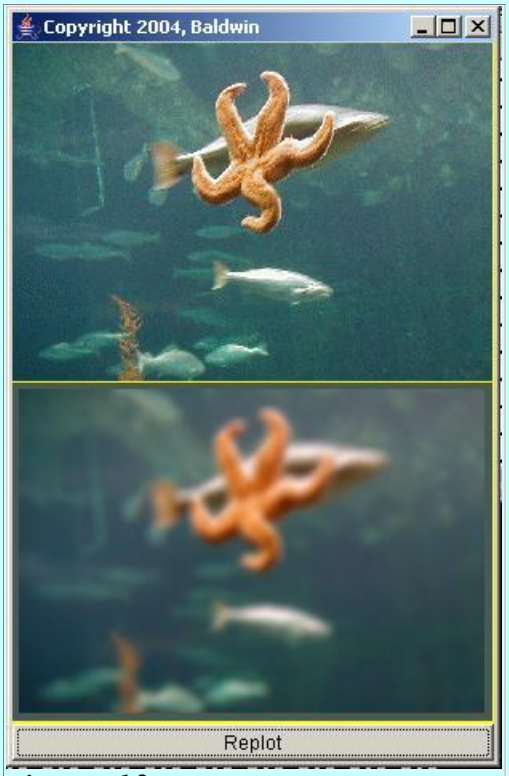

Figure 19

Oops! It looks like we went a little overboard with the smoothing and softening in this case. Now you know how to process an image if you want to make it look like you are viewing it through a foggy window.

A color shift

There also seems to be some color shifting in the output of this filter relative to the input in Figure 19. *(The starfish looks too red to me.)* In an earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness, I explained and gave examples of what happens when you change the distribution of the color values in an image. That may be what is happening here. It may, or may not be possible to correct for the color shift by changing the multiplicative factors that are applied to the Red, Green, and Blue planes *(see Figure 4 and the corresponding discussion of color filtering).*

> *(I will have a great deal more to say about color shifting later when we switch from the use of the normalization scheme in **ImgMod32** to the normalization scheme in **ImgMod32a**.)*

Now that you have learned all about 2D smoothing or softening filters, I encourage you to compile the class named **ImgMod033** and experiment with smoothing filters of your own design using your own images. As for this lesson, we are going to move along at this point to another type of filter.

Bipolar filters

This kind of filter opens up the possibility of having lots of fun with image convolution using some very simple filters. Generally, this kind of filter has a combination of positive and negative coefficient values.

Very similar results

With filters having only positive filter coefficients, changing the number of coefficients, or the values of the coefficients generally results in more or less the same general behavior. High wave number components may be suppressed to some degree but that is not always the case. Filters with all positive coefficients usually tend to smooth out the discontinuities causing the edges in the image to become less sharp.

Wildly different results

However, once we include both positive and negative coefficient values in a filter, we can get wildly different results by making small changes in the coefficient values, the signs of the coefficients, or the number of coefficients.

We will investigate the use of bipolar filters to produce filters that behave in the following ways:

- Embossing filters that produce a 3D-like effect *(See Figure 57.)*
- Edge detection filters *(See Figure 59.)*
- Sharpening filters *(See Figure 54.)*

We will begin our investigation with embossing filters.

Embossing filters that produce a 3D-like effect

Before we continue, I need to explain how the word embossing comes into play here. Embossed writing stationary is run through a machine that causes some portions of a picture to protrude from the surface of the paper and other portions to be recessed relative to the surface of the paper. The result is to create a very shallow 3D image on the paper. Although we can't create a true 3D image on the computer screen, we can create an optical illusion that results in a 3D-like effect. Figure 34 and Figure 57 show examples of a processed image that looks very much like embossed writing stationary.

How do you create a 3D optical illusion?

I will begin by showing you a couple of examples that illustrate what I mean by an embossing filter that produces a 3D-like effect. I will also discuss the methodology for creating the 3D optical illusion. After all, if we are going to design a convolution filter that creates a 3D optical illusion, we need to understand what it is that causes the optical illusion. Once we understand that, we will be better prepared to design the filter.

An example of a 3D optical illusion

The bottom panel in Figure 20 shows the result of applying a specific 2D convolution filter *(Filter02.txt)* to the image *(box01.jpg)* in the top panel.

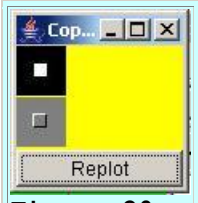

Figure 20

I believe that most people who work with computers would consider the image in the bottom panel of Figure 20 to have a 3D effect. Further, I believe that most of those people would agree that it appears that the box in the bottom panel in Figure 20 is protruding from the screen.

Another example of a 3D optical illusion

Similarly, I believe that those same people would consider the bottom panel in Figure 21 to have a 3D effect, in which the square in the bottom panel is recessed into the screen.

Figure 21

> *(Disclaimer: Optical illusions have different effects on different people, so you may be one of those people who see the above images in ways different from what I have described. For example, there is one famous optical illusion where some people see a young woman and others see an old hag. Sometimes I see one, and sometimes I see the other when I look at that picture. Which do you see?)*

Same convolution filter, different results

The same convolution filter was applied to the images in Figure 20 and Figure 21, but the results were very different. *(The image in Figure 21 is stored in the file named box02.jpg.)* Can you figure out why the results were so different?

What produces the 3D optical illusion?

The first thing that we need to establish is just what it is that produces the optical illusion of a 3D effect for a picture that is rendered on a flat computer screen *(or painted on a flat piece of canvas for that matter).*

Compare with GUI buttons

Compare the bottom image in Figure 20 with the three small buttons in the top right corner of the Frame and the large button labeled **Replot** in the bottom of the Frame in Figure 20. The GUI designers at Microsoft and Sun certainly want you to see those buttons in 3D and they want them to appear to be protruding from the screen. What do you see that is common between the GUI buttons and the bottom image in Figure 20?

Now compare the bottom image in Figure 21 with the same four buttons. What do you see that is different between those GUI buttons and the bottom image in Figure 21?

The secret is...

In case you haven't figured out the feature that produces the 3D effect in both cases, I will let you in on the secret of this particular optical illusion. That feature is a secret that most successful artists *(and all successful 2D GUI designers)* understand. It all has to do with light and shadows.

Illumination from above

Assume that a 3D button is illuminated by a light source shining down from above and to the left. The top edge and the left edge of the protruding button would be illuminated more brightly than the face of the button. As you can see, the top and left edges of the four GUI rectangles that represent the GUI buttons in Figure 20 are brighter than the face of the rectangle.

The illumination of a protruding button from above and to the left would cause the right edge and the bottom of the button to be in a shadow zone. Those edges would be darker than the face of the button. The right and bottom edges of the GUI rectangles that represent the buttons in Figure 20 are darker than the face of the rectangles.

A combination of light and shadow

This is the combination of light and shadow that causes the four GUI rectangles in Figure 20 to look like buttons that protrude from the screen. The left and top edges of the rectangles are brighter than the face of the rectangle. The right and bottom edges of the rectangle are darker than the face of the rectangle. The result is that to most people, the rectangles look like buttons that protrude from the screen.

That same technique involving light and shadow was applied to the bottom image in Figure 20 causing it to look like it is protruding from the screen.

A recessed square

Now consider the bottom image in Figure 21. If a square were indeed recessed into the face of a wall and illuminated from above and to the left, the right and bottom edges would be illuminated more brightly than the face of the square. The top and left edges would be in a shadow zone and would be darker than the face of the square.

Check out a 3D Windows button

If you happen to be running Windows *(with the classic look and feel)*, start the Notepad program. Then press and hold the *minimize* button. When you do that, the top and left edges of the GUI rectangle that represents the button become darker than the face of the rectangle. The bottom edge of the rectangle becomes brighter than the face of the rectangle. It's hard to tell what happens to the right edge of the rectangle due to its proximity to the rectangle that represents the *maximize* button. Most computer users would agree that this optical illusion causes the rectangle to look like a button that has been pushed into the computer screen.

The same effect of light and shadow was applied to the bottom image in Figure 21. That is what causes the 3D optical illusion to look like a recessed square to most computer users.

How do we do that using image convolution?

So, the big question at this point has to do with how one convolution filter can produce these different 3D effects. That is what I will explain in the sections that follow.

More generally, the question is how do we design a convolution filter that will produce the 3D optical illusion when applied to an image? That is the problem that we will tackle next.

Let's take inventory of what we know

Let's begin by taking inventory of what we already know to see if that will help us to design the convolution filter.

First, we know that in order to produce the 3D optical illusion, the filter will need to operate on edges that appear in the image. After all, the optical illusion is produced by highlighting some edges and making other edges darker.

We know that the process will probably need to emphasize the edges. That immediately eliminates the entire class of smoothing filters, which tend to de-emphasize the edges. That strongly suggests that the filter probably needs to be a bipolar filter because most filters containing only positive coefficient values tend to be smoothing filters.

Frequency domain considerations

We know that when viewed in the frequency or wave number domain, most smoothing filters are low-pass filters that tend to suppress high frequency or high wave number components. Since we know that we don't want a smoothing filter, we know that we probably don't want a low-pass filter. Assuming that we aren't going to create filters with lots of peaks and valleys in the frequency domain, that leaves only two possibilities:

1. Filters with a flat response
2. High-pass filters

We saw the result of applying a filter with a flat frequency response in Figure 3 and it certainly didn't produce a 3D effect. This suggests that we should concentrate on the use of a high-pass filter. Unfortunately, there are an infinite number of different high-pass filters that we could try, so we need to zero in a little closer.

Symmetry or the lack thereof

We know that in order to produce the 3D effect, the convolution process should not treat symmetrical features in the image in a symmetrical way. Rather, such features should be treated in a non-symmetrical way so as to highlight the edges on two sides of the feature and to darken the edges on the opposing sides. We know that the application of a symmetrical convolution filter to a symmetrical feature will produce a symmetrical result. That tells us that our filter needs to be non-symmetrical.

Summary of what we already know

In summary, we know that the filter should probably include both positive and negative coefficients, should be non-symmetrical, and should be a high-pass filter.

A very simple filter

Let's give it a try using the simplest non-symmetrical high-pass filter that I know of. If that filter seems to be going in the right direction, we can work to improve it in order to come up with a better 3D embossing filter.

This simple filter has only two coefficients. Those coefficients have values of -1 and +1. This is a non-symmetrical high-pass filter with an output that is proportional to the slope of the signal. For those of you familiar with differential calculus, the output of this filter approximates the first derivative of the signal.

Experimental results

The third graph in Figure 22 shows the result of applying this filter to the signal in the first graph in Figure 22. The convolution filter is shown at the left end of the second graph in Figure 22.

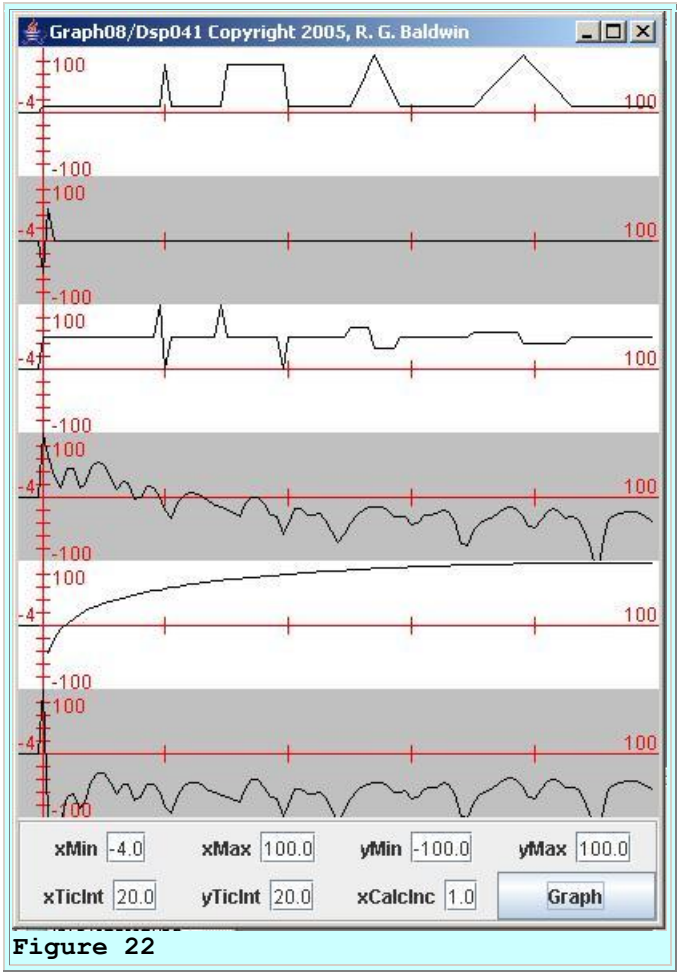

Figure 22

Think color

Once again, think of the input in the first graph in Figure 22 and the output in the third graph as a single row of color values in an image. From that viewpoint, this filter has a lot of promise.

The impulse response

For example, look what this filter did to the impulse in the signal. The impulse started out as a single bright spot in the image. It was converted to a bright spot followed by a black spot. Hence, it caused the left side of the impulse to be highlighted and the right side of the impulse to be darkened. That is just what we said earlier that we need.

The rectangular pulse

Now look at what it did to the rectangular pulse. It eliminated the pulse entirely replacing the left side of the pulse with a bright spot and replacing the right side of the pulse with a black spot. That also matches what we think we need. Unfortunately, it caused the body of the pulse to have the same brightness as the general background. That may not be what we need. We'll see later.

The triangular pulses

Now look what it did to the two triangular pulses. The left half of each triangle was replaced by a rectangular pulse somewhat brighter than the general background. The right half of each triangle was replaced by a rectangular pulse somewhat darker than the general background.

The distribution of color values

The general background level in the input signal was very low but not zero. The general background in the output was at approximately the mid point between the lowest and highest possible values of 0 and 100. Thus, except for the black spots and the very bright spots, the overall contrast between the darkest and brightest part of the image was reduced. If you were to compute a distribution of the color value in the output, it would probably be narrower than a distribution of the color values in the input.

> *(I explained what happens when you change the distribution of the color values in an image in an earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness. You might find it useful to go back and review that lesson. I will also have quite a lot more to say about the distribution of color values later in this lesson.)*

Spectral data

As indicated earlier, this is a high-pass filter as evidenced by the frequency response of the filter shown in the fifth graph in Figure 22. Consequently, except for zero frequency, the energy at low frequencies in the output is much lower than the energy at low frequencies in the input. High-frequency energy is preserved from input to output.

> *(The peak at zero frequency in the output is an artifact resulting from the fact that the convolution algorithm strives to cause the output to have the same mean value as the input. Otherwise, the energy at zero frequency in the output would be zero because this particular filter has a zero response at zero frequency.)*

Application of the filter to a 2D image

Let's apply this filter to a real 2D image and see what we get.

We will begin by creating a 2D filter having one row and two columns and applying it to two different 2D images. The filter coefficient values will be +1 and -1. *(The filter is stored in a file named Filter06.txt.)*

The left panel in Figure 23 shows the filter. The right panel in Figure 23 shows the wave number response of the filter in the same format as before.

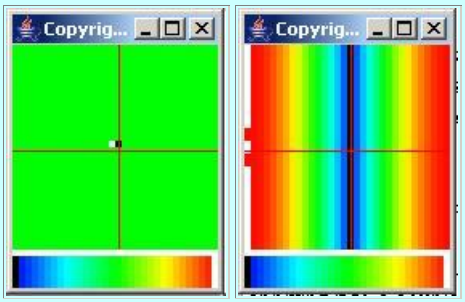

Figure 23

The wave number response

As you can see from the black/blue band in the center, and the red/white band at the edges, this filter will suppress components with a low wave number and enhance components with high wave numbers *for vertical edges.*

However, the filter will have no effect on the wave number components belonging to *horizontal edges*. For edges that are somewhere between horizontal and vertical, the effect will be roughly proportional to the angle of the edge relative to the vertical.

The output image

The result of applying this filter to the white square in the top image *(box01.jpg)* of Figure 24 is shown in the bottom image of Figure 24.

Figure 24

This is beginning to look like what we are after. Each row in the white square in the input image is analogous to the rectangular pulse in the input signal in the first graph in Figure 22. As we predicted, the left edge of the square was made very bright and the right edge of the square was made very dark.

Also as we predicted from an examination of the wave number response in Figure 23, the filter had no effect whatsoever on the horizontal edges on the top and the bottom of the square.

In addition, the general background was made brighter, and the face of the square has the same brightness as the general background.

A 3D effect

To my eyes, the square has taken on something of a 3D appearance and appears to protrude from the screen. In fact, as an optical illusion, my eyes tend to construct a very faint horizontal top and bottom on the output square even though the top and bottom edges are exactly the same color as the background and the face of the square. Optical illusions are very interesting.

Regarding the angle of the edge

Now, to get an idea how this filter behaves relative to the angle of the edge, let's call Stick Man back out of retirement. *(The image for this experiment is stored in a file named stickman2a.gif.)*

The bottom panel of Figure 25 shows the result of applying the same filter *(Filter06.txt)* to the Stick Man figure shown in the top panel of Figure 25.

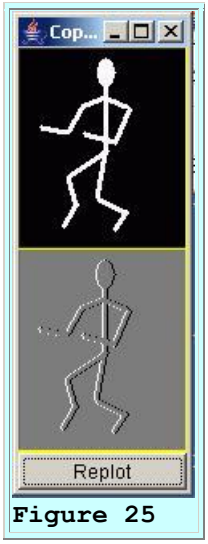

Figure 25

Compare the torso with the forearms

The thing that is the most interesting here is to compare the impact of the filter on the torso with the impact of the filter on the forearms. There is a solid white edge on the left side of the torso and a solid black edge on the right side of the torso. This is because the torso has vertical edges. The wave number response in Figure 23 tells us to expect the maximum effect of the filter on vertical edges.

On the other hand, the forearms have only a few black and white dots. This is because the forearms are almost horizontal and the wave number response in Figure 23 tells us that the filter will have no effect on horizontal edges. The different edges on the Stick Man fall at different angles relative to the vertical, and the effect of the filter can be seen to vary with respect to those angles.

Rotate the filter by ninety degrees

Now we will rotate the filter by ninety degrees and place it in two rows of the same column. We will use the same values as before, namely +1 and -1.

The convolution filter *(Filter10.txt)* and the wave number response of the convolution filter are shown in Figure 26 in the same format as before.

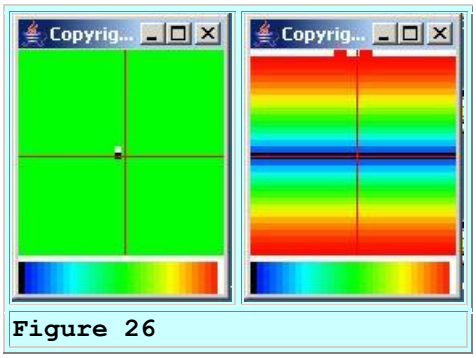

Figure 26

It will probably come as no surprise to anyone that the wave number response looks just like Figure 23 except that it has been rotated by ninety degrees. This wave number response tells us that the filter will have maximum effect on horizontal edges and no effect whatsoever on vertical edges.

Apply the filter to the Stick Man

This time, I will skip the image of the box and go straight to the Stick Man. Figure 27 shows the result of applying this filter to the same Stick Man image *(stickman2a.gif)* as shown in Figure 25.

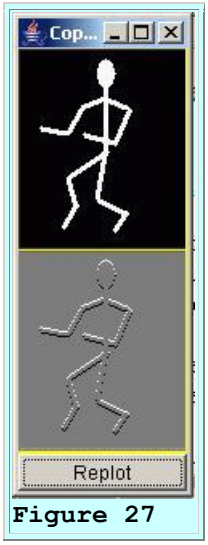

Figure 27

The Stick Man lost his torso

Compare this result with the result shown in Figure 25. In Figure 25, the vertical edges on the torso evidenced the maximum impact from the filter, showing solid white and black edges. In Figure 27, as predicted above, the vertical edges on the torso were not impacted at all by the filter. In fact, the torso has simply faded into the background.

Also compare the forearms, which are nearly horizontal. In Figure 25, the forearm was only barely impacted by the filter. In Figure 27, as predicted above, the forearms were impacted in a significant way.

What we need is a compromise

We have seen that a one-dimensional horizontal filter does a good job on vertical edges and doesn't impact horizontal edges. We have also seen that a one-dimensional vertical filter does a good job on horizontal edges and doesn't impact vertical edges.

Where is the light source?

From a 3D viewpoint, the horizontal filter makes it appear that the object is being illuminated by a light source that is to the left of and at the same level as the object. Since that is a fairly rare occurrence in the real world, that doesn't necessarily create the 3D optical illusion that we are looking for.

Also, from a 3D viewpoint, the vertical filter makes it appear that the object is being illuminated by a light source that is directly overhead. While that is more common than a light source that is at the same level as the object, in most cases, the object being illuminated is not directly below the light source.

We need a filter that makes it appear that the light source is above the object and off to one side or the other.

> *(To agree with what some of us have become conditioned to expect by working on a daily basis with Windows GUI objects, we probably need a filter that makes it appear that the light source is above and to the left of the object.)*

No need for a rocket scientist here

Since we know the effect of horizontal and vertical filters, it doesn't take a rocket scientist to surmise that the same filter arranged at an angle of forty-five degrees may do what we want. We will try that.

For this case, we will use a 2x2 convolution filter having the coefficient values shown in Figure 28. *(This filter is stored in a file named Filter02.txt.)*

```
1.0  0.0
0.0 -1.0
```

Figure 28

Will probably need to scale the output

Note that two of the filter coefficients have a value of 0 and don't contribute anything to the output. Also recall me telling you earlier that the 2D convolution algorithm divides each sum of

products by the number of filter coefficients. To make a long story short, this causes the entire output from this filter to be rather dark. To compensate for that, the color values in some of the output images that I will show have been scaled up to make them brighter. This was accomplished by entering a multiplicative factor in the Red, Green, and Blue fields in Figure 4.

The 2x2 filter and the wave number response

Figure 29 shows the filter and the wave number response of the filter in the same format as before.

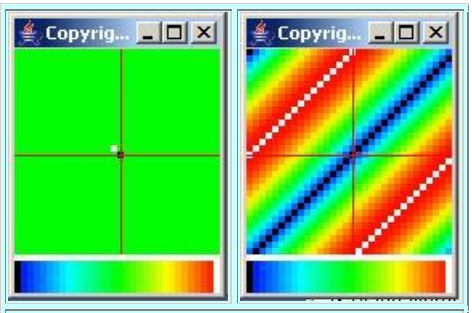

Figure 29

The wave number response of this filter is the same as you saw in Figure 23, except that it is rotated by forty-five degrees.

Apply the filter to the white square

The bottom panel in Figure 20, *(which you saw earlier)* shows the result of applying this filter to the white square in the top panel of Figure 20. *(This image is stored in a file named box01.jpg.)* As you will recall, this is the image that got us started talking about 3D optical illusions in the first place. As you saw earlier, simply applying the filter to the image makes it appear that the image is 3D, illuminated from above and off to the left.

Apply the filter to the Stick Man

Figure 30 shows the result of applying this filter to the same Stick Man that you saw in Figure 25 and Figure 27.

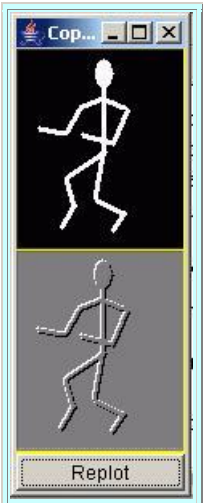

Figure 30

You may or may not agree that the Stick Man looks more 3D in Figure 30 than was the case in either of those two earlier figures. In Figure 30, at least, he has all of his body parts. He didn't lose his torso or his forearms.

Turn and face the other way

Figure 31 shows what happens when the Stick Man turns and faces the other way, presenting body parts with different angles to the light source. *(This image is stored in the file named stickman3a.gif.)*

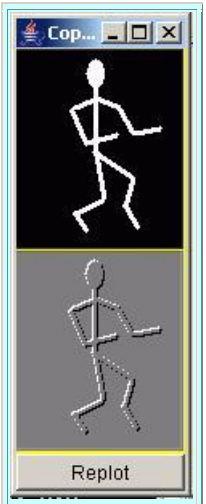

Figure 31

In my opinion, he still looks pretty good. One of his thighs is fading a little, but the edge effect was still sufficient to keep it from fading completely into the background.

(From the wave number response in Figure 29, we can see that this filter will ignore edges that are at forty-five degrees to the horizontal in a direction that is perpendicular to the red bands. That is what happened to the thigh in Figure 31.)

Finally some embossed stationary

The bottom panel in Figure 32 shows the result of applying the same embossing filter *(Filter02.txt)* to the photographic image *(penny05.jpg)* shown in the top panel of Figure 32 and then scaling all of the color values in the output by a factor of 2.0.

Figure 32

(Actually, this is the negative of a photographic image. It is interesting that the color of copper doesn't change very much between the positive and the negative.)

Figure 32 shows what the filter can do to a photographic image of a coin containing a fairly limited number of different colors. This almost looks like someone put a piece of pink paper on top of the coin and then rubbed it causing the paper to be pushed into the depressions in the coin.

An example with many different edges

Figure 33 shows an example of applying the embossing filter *(Filter02.txt)* to an image *(imgmod03a.gif)* that contains a wide variety of edges.

Figure 33

When the input is already 3D

This is a particularly interesting example due to the different kinds of edges that appear in the original image. Some of the features in the original image were already intended to have a 3D effect. The output from the convolution process for those features still looks 3D but looks entirely different.

For example, take a look at the rectangular feature near the top containing the label **BevelBorder LOWERED,2,2**. This feature appears to be a raised rectangular block in the original image due to the highlighting on the left and top edges and the darkening on the right and bottom edges. It no longer looks like a raised rectangular block in the output. Rather, it appears to have a protruding lip on the left and top edges and an etched groove on the right and bottom edges.

When the input is not already 3D

Now look at the feature labeled **Compound,Empty + LineBorder,10,10** near the bottom with the light grey or silver border. The border had no 3D effect in the original image, but has a significant raised 3D appearance in the output image.

The text

Now consider the text. None of the text had a 3D appearance in the original image. However, all of the text has a significant 3D appearance in the output image. What's more, most of the text in the output image appears to protrude from the screen, but some of the text in the output, such as the Copyright statement at the top, appears to be etched into the surface. Can you identify the conditions that cause some text to protrude and other text to appear etched?

A photograph with lots of colors

Figure 34 shows the result of applying the embossing filter *(Filter02.txt)* to a photographic image *(background02.gif)* containing lots of colors and then multiplying all of the color values by 1.5.

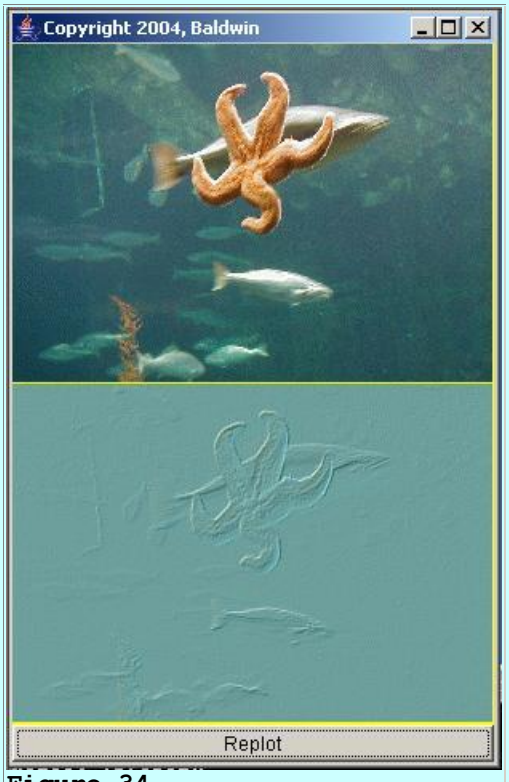

Figure 34

To me, the output looks very much like the photograph was embossed onto a piece of light blue paper. And this was all accomplished using a 2D convolution filter having only four coefficients, two of which had a value of zero. It is amazing how much power resides in a convolution filter having both positive and negative coefficient values.

(Later, in Figure 57, I will show you another result that was produced by applying the same filter to the same image but using a different normalization scheme to convert the convolution output values back into the required eight-bit unsigned format.)

A depressed Stick Man

Before leaving the topic of 3D embossing filters, I want to show you one more image of the Stick Man and leave you with a question.

Figure 35 shows the result of applying the same filter *(Filter02.txt)* to the image *(stickman2.gif)* shown in the top panel of Figure 35.

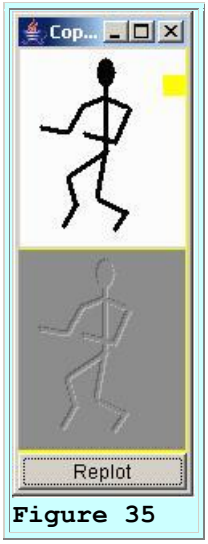

Figure 35

To me, this Stick Man appears to be depressed into the surface of the screen rather than protruding from the screen as is the case in Figure 30. Can you figure out why he appears to be depressed? Think about what we have discussed so far in this lesson.

Now it is time to move on and discuss edge detection filters.

Edge detection filters

Assume that for reasons of your own, you would like to convert an image to a sort of line drawing where the lines in the output occur at the edges in the original drawing similar to that shown in Figure 36. This process is commonly referred to as *edge detection*.

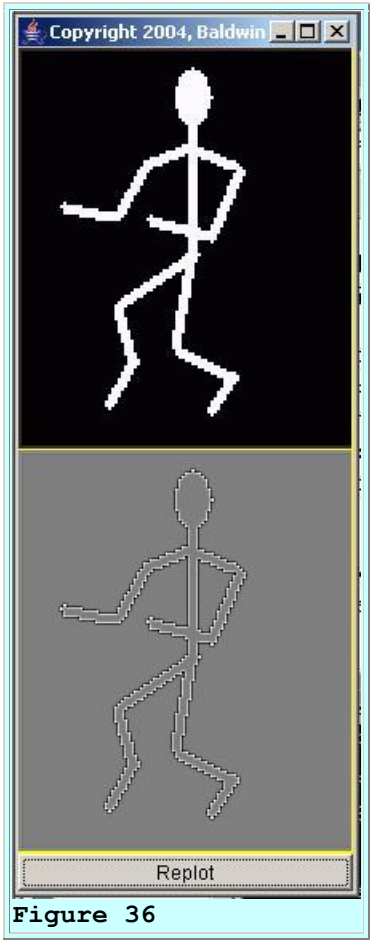

Figure 36

Why would I want to do that?

The purpose of this section of this lesson is to show you how to do edge detection, not why to do it. *(Go to Google and search for image edge detection and you will probably find more material on the topic than you have the time to read.)*

Embossing filter is an edge detector

The embossing filter in the previous section is a form of edge detector. However, it does some things to those edges that we might not like to see if we are simply trying to identify and highlight the edges.

Watch out for symmetry or the lack thereof

For example, the embossing filter treats symmetrical features in a non-symmetrical way, causing the edge on one side to be bright and the edge on the other side to be dark. Since that is probably not what we want in a general purpose edge detector filter, we probably want to treat symmetrical features in a symmetrical way. This suggests that we should probably use a symmetrical convolution filter.

Probably need a high-pass filter

From what we have learned in the earlier sections of this lesson, we know that low-pass filters tend to de-emphasize the edges in an image while high-pass filters tend to emphasize the edges. This suggests that we probably want to use a symmetrical high-pass filter.

Zero response at zero wave number

Because we want the filter to be a high-pass filter, we probably want the filter response at zero frequency or zero wave number to be very low, probably zero. Therefore, we probably want the algebraic sum of the filter coefficients to be zero. This suggests that we probably want to use a symmetrical high-pass filter for which the sum of the coefficients is zero.

How do I design such a filter?

How does one go about designing such a filter? Well, if you have a lot of digital signal processing experience, you can probably come up with something close just by thinking about it. If not, you can use a trial-and-error approach.

Trial and error

For a trial-and-error approach, I suggest that you go to the FFT Laboratory page, check the check box labeled *Origin Centered*, and then adjust the weights in the top-left *(Real)* box until you get what you want in the bottom-left box, making certain that the curve in the bottom-right box is flat at zero.

> *(To adjust a weight, just grab one of the circles with the mouse and move it up and down.)*

If you make the weights in the top-left box symmetrical about the center, the spectrum showing in the bottom-right box should be flat at zero.

> *(Once you check the Origin Centered check box, the center of each of the curves is identified by an open circle. The center of the bottom-left box will represent zero frequency and the right end will represent the Nyquist folding frequency.)*

A very simple convolution filter

The simplest one-dimensional convolution filter that I can think of that meets all of the criteria stated above is a filter having the following three coefficients:

-0.5, 1.0, -0.5

This is a high-pass filter with an output that is proportional to the rate of change of the slope of the signal. The convolution output approximates the second derivative of the signal.

Let's give it a try and see how it performs.

Apply the simple convolution filter

The third graph in Figure 37 shows the result of applying this convolution filter to the signal shown in the first graph. The convolution filter is shown in the second graph in Figure 37.

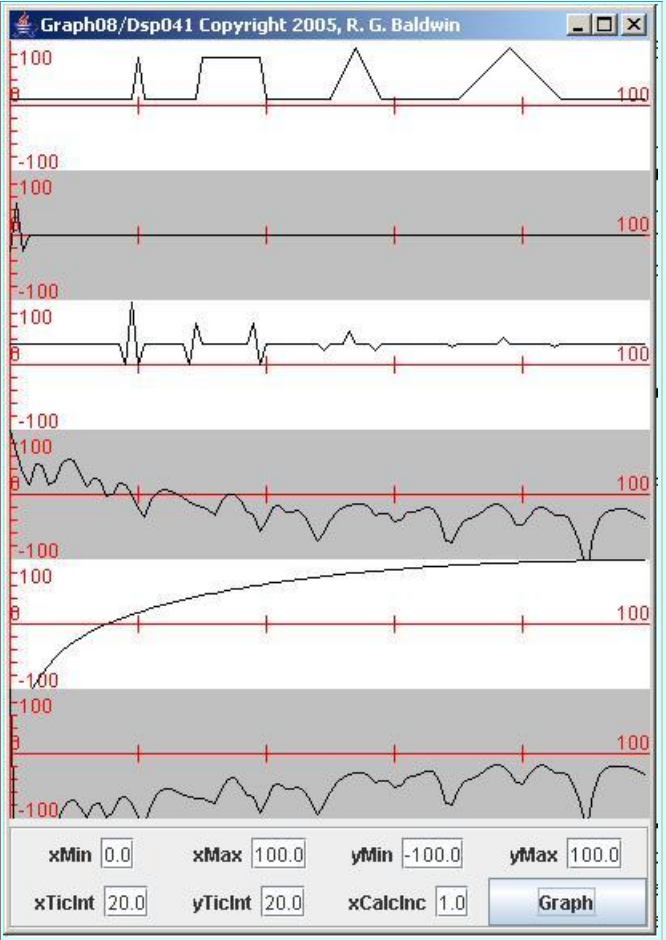

Figure 37

Think color again

If we think of the signal in the first graph in Figure 37 as representing the color values on a color plane in a single row of pixels in an image, this filter seems to have some promise. As we have come to expect, the impulse in the signal produces the impulse response of the filter in the output. The impulse in the input would represent a single bright spot. The impulse response in the output would represent a single bright spot with a black spot on each side. This confirms the symmetry that we are looking for.

The rectangular pulse

Now consider what the filter does to the rectangular pulse in the input. The main body of the output for the rectangular pulse is at about the same level as the general background level. Going from left to right into the pulse, we would see a black spot followed by a bright spot. While the bright spot in this case is not at the maximum level of brightness, it is still brighter than the general background and the main body of the rectangular pulse.

Going from left to right when leaving the pulse, we would see a bright spot followed by a black spot. These two spots are a mirror image of the two spots on the left side of the rectangular pulse. This combination of black and white spots should serve to identify the edges of the rectangular pulse feature in the input.

The triangular pulses

Now consider what the filter does to the triangular pulses. Here we can see that the filter produces a positive or negative spike in the output when the slope of the input changes.

> *(These spikes represent spots that are brighter or darker than the general background.)*

The size of the spike is proportional to the amount of change in slope. The direction of the spike depends on the direction of the change in the slope. When the slope change rotates in a counter-clockwise direction, the direction of the spike is toward the negative. When the slope change rotates in a clockwise direction, the direction of the spike is toward the positive. This is pretty much what we expect for the second derivative of the signal.

Relating that information back to the rectangular pulse and the two spikes in the output at the beginning of the rectangular pulse, we see that the slope changes from zero to infinite in a counter-clockwise direction, and then changes from infinite to zero again in a clockwise direction. This all occurs within two samples. Therefore, the spikes are adjacent to one another, they are large, and they have consistent directions.

Edges are determined by a change in slope

The edges of the features in an image are defined by changes in the slope of the surface that defines the image. Therefore, if we can identify the changes in the slope of the surface, we can identify the edges. On that basis, it looks like this filter should do the job.

A high-pass filter

Before leaving the discussion of Figure 37, we should point out that this is a high-pass filter as evidenced by the frequency response shown in the fifth graph in Figure 37.

Apply the filter to a real image

Figure 38 shows this three-point filter *(Filter08.txt)* along with the wave number response of the filter in the same format as before.

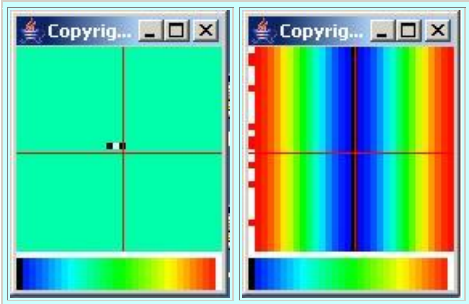

Figure 38

By now, you will recognize that this filter will have maximum impact on edges that are parallel to the red bands, and no impact whatsoever on edges that are perpendicular to the red bands.

Compare with the embossing filter

If you compare the wave number response of this filter with the wave number response of the embossing filter shown in Figure 23, you will see that the red/white pass bands for this filter are somewhat narrower and the black/blue reject band is somewhat wider than the pass bands and the reject band in Figure 23. You can also see this in the fifth graph of Figure 37 as compared to the fifth graph of Figure 22.

Results with a real but simple image

Figure 39 shows the result of applying this filter to an enlarged image *(stickman2b.gif)* of the Stick Man and multiplying all of the color values in the output by a factor of 2.0.

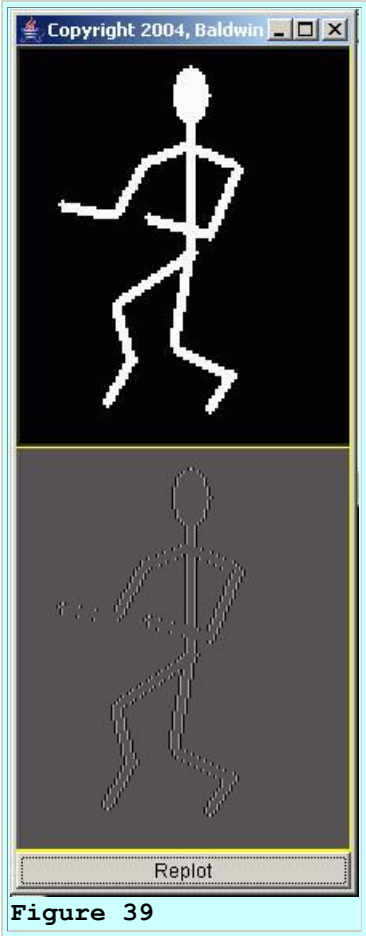

Figure 39

(This image of the Stick Man was enlarged by a factor of two relative to the original image (stickman2a.gif) without redrawing the image. As a result, this Stick Man suffers from a bad case if the jaggies on the sloping lines.)

Comparison with Figure 37

Let's compare Figure 39 with the signal in Figure 37. For every row of this image, all of the color values are either 0 or some very large value. For example, the intersection of the row with the Stick Man's neck is very similar to the rectangular pulse in the first graph in Figure 37. *(However, the baseline for the signal in Figure 37 is a little above zero.)* Therefore, we would expect the output for that row in this image to be very similar to the output for the rectangular pulse shown in Figure 37.

If that is our expectation, we won't be disappointed. If you examine the neck portion of the output very carefully, you will see that there are black and white vertical lines on the left side of the neck. Similarly, there are white and black vertical lines on the right side of the neck as a mirror image of the left side. The inner portion of the neck is essentially the same color as the general background as is the case in the output for the rectangular pulse in Figure 37.

All of the features in the Stick Man image consist of rectangular pulses when viewed as a single row of pixels. There are no triangular pulses or impulses.

Vertical is good, horizontal is not so good

As you probably predicted, this filter does a reasonably good job of identifying the vertical edges in the Stick Man image, and does a rather poor job of identifying the edges that are nearly horizontal, such as the forearms.

> *(For example, there are horizontal edges on the stair steps that make up all of the sloping lines, and those horizontal edges aren't identified at all.)*

Upgrade the filter

Now let's try upgrading the filter to get better results for horizontal edges. We will begin by expanding the filter from a 1x3 filter to a 3x3 filter using the filter coefficient values shown in Figure 40.

```
 0.0 -0.25 0.0
-0.25 1.0 -0.25
 0.0 -0.25 0.0
Figure 40
```

Note that the four corner values in Figure 40 are 0.0. Therefore, as a practical matter, this filter has only five coefficients.

Note also that the sum of all of the coefficients in this filter *(Filter11.txt)* is zero, ensuring that the wave number response will be zero at a wave number of zero.

The filter and the wave number response

Figure 41 shows the filter and the wave number response of the filter in the same format as before.

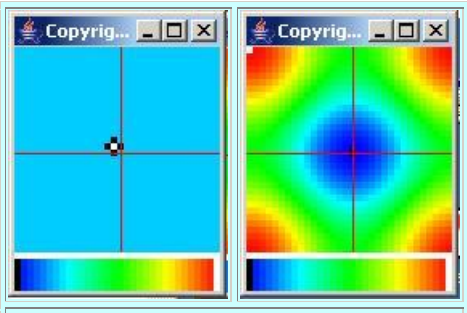

Figure 41

The wave number response tells us that the filter should provide wider angular coverage for the edges than was the case with the filter shown in Figure 38. In other words, we should expect equal treatment of both horizontal and vertical edges in the image.

Apply the filter to the Stick Man

Figure 42 shows the result of applying this filter *(Filter11.txt)* to the same Stick Man image *(stickman2b.gif)* and multiplying all of the color values in the output by a factor of 5.

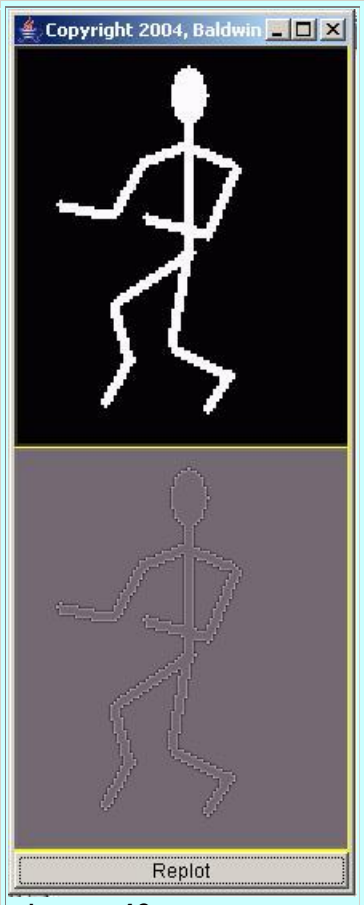

Figure 42

Better in some ways, worse in others

When compared with Figure 39, this filter provides much better performance on the nearly horizontal surfaces such as the forearms. However, the overall contrast between the background and the white lines that identify the edges doesn't seem to be as good in Figure 42 as in Figure 39.

Gaps in the wave number response

An examination of the wave number response in Figure 41 indicates that we might be able to improve on this filter. One problem with the filter, as evidenced by the wave number response in

Figure 41, is the lack of a red pass band in the North, South, East, and West directions. It might be good if we can get more red coverage in the high wave number areas around the entire perimeter of the plot.

Use a true nine-point filter

We might be able accomplish this by filling in the four filter coefficients that have values of zero in Figure 40. We will see what we can accomplish with the filter *(Filter12.txt)* shown in Figure 43.

```
-0.125 -0.125 -0.125
-0.125  1.0   -0.125
-0.125 -0.125 -0.125
```

Figure 43

Once again, note that the sum of the filter coefficients is zero for the reasons given earlier.

The filter and the wave number response

Figure 44 shows this filter *(Filter12.txt)* and the wave number response of the filter in the same format as before.

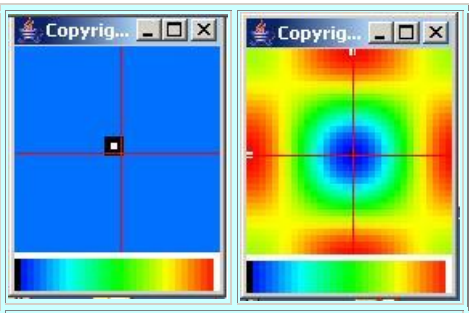

Figure 44

It appears from Figure 44 that we met our objective of getting better red pass band coverage around the perimeter of the wave number response than was the case for the filter shown in Figure 41. It remains to be seen if this will result in an improvement in edge detection performance.

Apply the filter to the Stick Man

Figure 45 shows the result of applying this filter *(Filter12.txt)* to the same enlarged Stick Man image *(stickman2b.gif)* and multiplying all of the color values in the output by a factor of 5.

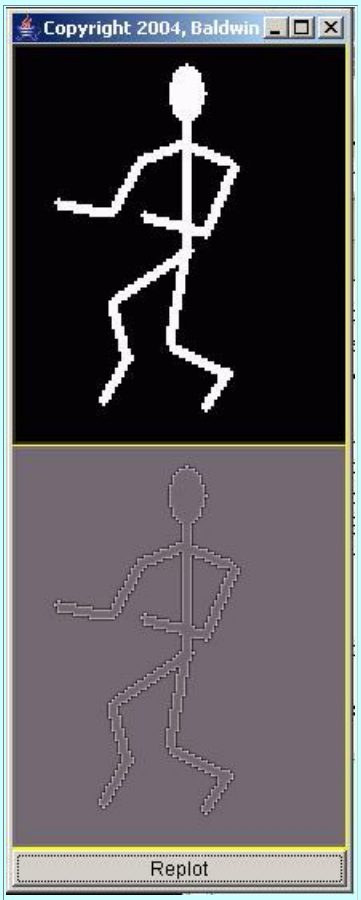

Figure 45

I will let you be the judge as to whether or not Figure 45 is an improvement over Figure 42, but it looks to me like the contrast is a little better in Figure 45.

Apply the filter to a photographic image

Real photographic images aren't normally made up of surfaces that contain rectangular towers or even pyramids for that matter. Rather, the changes in slope in the surface that represents the image are usually less pronounced than is the case with the Stick Man image. By comparing the output for the triangular pulses to the output for the rectangular pulse in Figure 37, we can predict that the results for real photographic images won't be as good as for the Stick Man image.

A little bit of black art

Also, because of the limited dynamic range of the values used to represent color values in images, and the requirement to transform all convolution output values back into the range from 0 through 255 inclusive, scaling in the image convolution process is something of a black art. If I simply apply the filter whose coefficient values are shown in Figure 43 to the starfish image shown in the top panel of Figure 34, the results are not usable. There are no scale factors that I can apply to the output from the convolution process using the interactive control panel shown in Figure 4 that will bring out the edges.

A new filter with larger coefficient values

However, if I create a new filter *(Filter13.txt)* in which I simply multiply all of the coefficient values from Figure 40 by a factor of 80, producing the filter coefficient values shown in Figure 46, the results of the convolution are close to what we are looking for.

```
-10.0 -10.0 -10.0
-10.0  80.0 -10.0
-10.0 -10.0 -10.0
```

Figure 46

> *(Note that multiplying every coefficient in a convolution filter does not change the basic shape of the wave number response. It simply scales the values in the wave number domain by the same multiplicative factor. Therefore, there is no point in showing you another picture of the wave number response for the new filter. It looks just like the wave number response in Figure 44.)*

Apply the filter to the starfish image

Figure 47 shows the result of applying this filter *(Filter13.txt)* to the starfish image *(background02.gif)* and multiplying all of the color values by a factor of 1.3 following the convolution process.

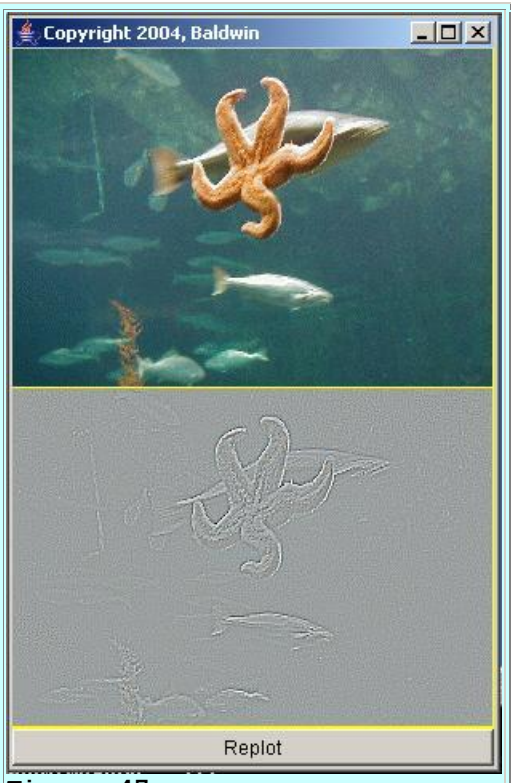

Figure 47

Many, but not all of the edges in the original photograph show up in the processed image. Note however that because we used a symmetrical convolution filter, the image in Figure 47 doesn't exhibit the 3D-like quality that is apparent in Figure 34.

> *(Later, in Figure 59, I will show you another result that was produced by applying the same filter to the same image but using a different normalization scheme to convert the convolution output values back into the required eight-bit unsigned format.)*

Two different kinds of filters

The embossing filter used for Figure 34 and the edge detection filter used for Figure 47 are fundamentally two different kinds of filters.

The embossing filter used to produce Figure 34 generates an output that is roughly proportional the slope of the surface that describes the image. Positive slopes produce positive values and negative slopes produce negative values. On the other hand, the edge detection filter used to produce Figure 47 generates an output that is roughly proportional to the magnitude of *changes* in the slope of the surface rather than being proportional to the slope itself.

The embossing filter estimates the first derivative of the surface.

The edge detection filter estimates the second derivative of the surface.

> *(The smoothing filter from an earlier section, by the way, estimates the integral of the surface.)*

Improvements to the edge detection filter

There are probably other things that can be done to improve the edge detection results, such as applying some sort of a threshold, for example. I encourage you to experiment and see if you can find a way to do a better job of edge detection.

Sharpening filters

Sharpening filters are used to process a photographic image to make it more crisp. Let me begin by quoting Ken Bennett, Wake Forest University Photographer.

> *"All digital images need to be sharpened. This is not related to the 'sharpness' of film images -- I'm assuming that you used a sharp lens, focused properly, and avoided camera and subject movement. Rather, we're talking about the apparent softness of raw digital images, either from scans or from digital images. We fix this by increasing edge contrast, making the image appear sharper."*

I will assume that Mr. Bennett is correct in his assessment. My objective here is to help you to understand how to sharpen, and not why to sharpen.

You will find a very good example of using convolution to sharpen a photographic image of an automobile at gamedev.net.

Different sharpening techniques are available

Although there are different techniques used for sharpening digital photographs, from what I read, most of them involve enhancing the higher wave number components relative to the lower wave number components in the image. To do that with a convolution filter, we need a high-pass filter. Many of the sharpening filters that I have read about seem to use bipolar 2D convolution filters where all of the coefficients add up to a value of 1.0.

> *(This means that the wave number response of the filter is 1.0 at a wave number of zero.)*

Important constraints

Unlike with the embossing filter and the edge detection filter, we are faced with some important constraints. Perhaps the most important constraint is that we usually don't want to change the color of the image in any significant way.

> *(This means that we must control the mean and the standard deviation of the color values in the output image relative to the input image. See the earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness for a discussion of the impact of changing the distribution of color values in an image.)*

If we are going to use a high-pass filter, it should probably be almost flat with only a hint of emphasis at the high end of the wave number spectrum. Otherwise, it will act like an edge detection filter or a 3D embossing filter.

How do I design such as filter?

Once again, let's take a trial-and-error approach using the FFT Laboratory page. Go to that page and check the box labeled *Origin Centered*. Then go to the upper-left box and drag the two black dots on each side of the center down about to about one-fifth of their maximum. Make sure that you drag both dots down the same distance. *(The curve in the bottom right box should be flat at zero after you do that.)*

Note the shape of the response

Note the shape of the curve in the bottom left box, keeping in mind that the empty circle represents a frequency of zero. At this point, the magnitude of the response at a frequency of zero is large. The sign of the response at zero frequency is negative. The response at the Nyquist folding frequency is high also, and is positive. The curve is nowhere near being flat.

Adjust the response at a frequency of zero

Now grab the empty circle in the upper-left box and start pulling it up toward the top of the screen. Move it a little at a time and turn it loose between moves. Observe what happens to the curve in the bottom-left box as you do that. You should see the value at zero frequency in the bottom-left box moving upward.

Continue this process until the curve in the bottom-left box is almost flat, but with the value at zero frequency being a little lower than the value at the folding frequency. You can use this trial-and-error technique to design a symmetrical three-point high-pass filter having the degree of flatness that you desire. Then estimate the relative sizes of the three weights in the top-left box. Those three weights are the coefficient values for your sharpening filter.

A one-dimensional example

Let's see what we get using a three-point filter having the following values:

-0.333333 1.666666 -0.333333

> *(Note that the sum of the coefficients for this filter is 1.0. That means that the filter has a response of 1.0 at a frequency of zero.)*

The frequency response of this filter is shown in the fifth graph in Figure 48. The third graph in Figure 48 shows the result of applying this filter to our standard signal, which is shown in the first graph in Figure 48. The filter itself is shown in the second graph in Figure 48.

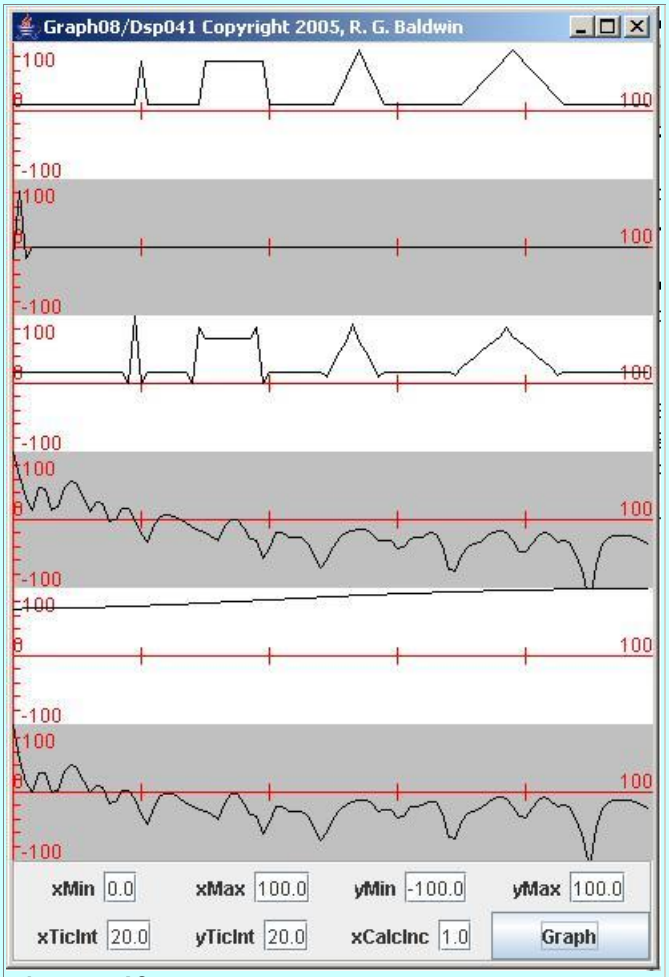

Figure 48

Approximate the filter interactively

If you approximate this filter on the FFT Laboratory page, you will see what the frequency response of this filter looks like on a linear scale, *(as opposed to the decibel scale shown in the fifth graph in Figure 48).*

A high-pass filter

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.

If we think of the signal values in Figure 48 as representing the color values in a single row of pixels in a color plane in an image, we see that this filter should do a reasonably good job of preserving the color values.

> *(However, we do see that the distance between the baseline and the flat portion of the rectangular pulse in the output is less than it is in the input. We also see that*

the general background is higher in the output than it is in the input. These characteristics suggest that there is some compression of the color distribution, and that the general background in the output will be brighter than the input.)

As mentioned above, a little blip will be produced in the output color values each time the slope of the surface that describes the color plane changes. We would think that these little blips might cause the changes in slope to be highlighted and to cause the image to become a little more crisp.

A 2D sharpening filter

Unfortunately, none of the 2D sharpening filters that I have found on various web sites and in various books seem to work properly when applied using the normalization algorithm in the class named **ImgMod32**. For example, I am going to show you the results of applying the 2D convolution filter shown in Figure 49. This is one of the sharpening filters recommended at gamedev.net.

```
-1.0 -1.0 -1.0
-1.0  9.0 -1.0
-1.0 -1.0 -1.0
```

Figure 49

Apply the filter to a photograph

Figure 50 shows the result of applying the filter *(Filter14.txt)* shown in Figure 49 to the starfish image *(background02.gif)*.

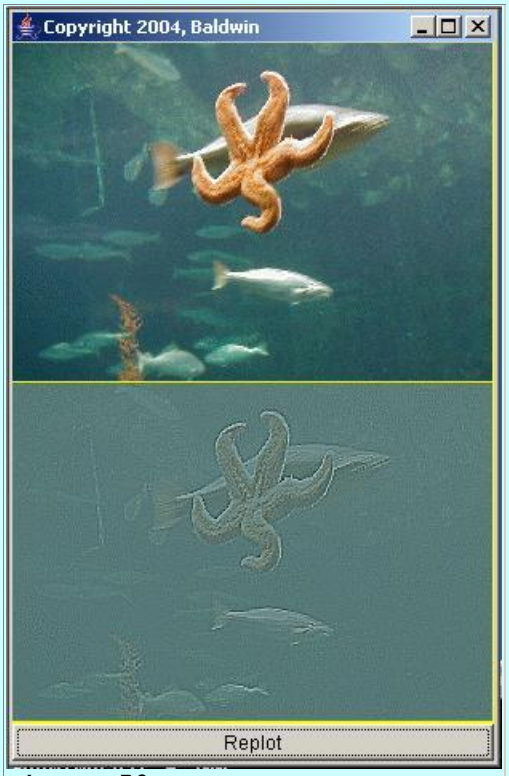

Figure 50

What happened to the color?

The results shown in Figure 50 certainly don't look anything like the results shown for the image of the automobile at gamedev.net. The big question is, why not?

I strongly suspect that the problem has something to do with the way the class named **ImgMod32** treats the bipolar convolution results at the point where it is necessary to convert those results back into eight-bit unsigned color values in the range from 0 to 255 inclusive.

The normalization scheme for ImgMod32

The normalization scheme in the class named **ImgMod32** causes the mean value of the output to match the mean value of the input. So far, so good. Then if there are any negative values remaining, all of the values are biased upward so as to cause the minimum value to be zero. In effect, this causes all of the colors to become brighter when it occurs.

After that, if there are any values greater than 255, all of the values are scaled down to force the maximum value to be 255. While this approach seems logical, it may not be the best approach. This last step may compress the color distribution. For example, a single large positive or negative value would cause all of the color values to be compressed into a narrower distribution. This, in turn, would cause the colors to appear to be somewhat washed out.

(By the way, I haven't found any hints in any books or on any websites indicating how others perform this normalization, so I'm flying blind in this area.)

The color distribution

The results shown in Figure 50 strongly suggest that the normalization process is reducing the width of the distribution of the color values. Recall that you learned in the earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness:

> *"The contrast of an image is determined by the width of the distribution of the color values belonging to the image. If all color values are grouped together in a narrow distribution ... the details in the image will tend to be washed out. In addition, the overall appearance of the image may tend toward some shade of gray. The shade of gray will depend on the location of the grouping between the extremes of 0 and 255."*

That appears to be what is happening in Figure 50. The output image has lost contrast relative to the input image. Also, the output image is tending toward a shade of gray.

An alternative normalization scheme

At this point, I am going to switch from the use of the classes named **ImgMod33** and **ImgMod32** to the classes named **ImgMod33a** and **ImgMod32a**.

The normalization scheme used in **ImgMod32a** is significantly different from the normalization scheme used in **ImgMod32**.

The class named **ImgMod33a** is the same as the class named **ImgMod33** except that it uses **ImgMod32a** for convolution instead of using **ImgMod32**.

Normalization in ImgMod32a

As before, this normalization process guarantees that the final color values on all three color planes have values between 0 and 255 inclusive. As before, this scheme causes the mean value of the convolution output to match the mean value of the input on each color plane. However, this scheme also causes the root mean square *(RMS)* value of the color values in the output to match the RMS values of the input on each color plane.

(The RMS value is a measure of the width of the color distribution.)

Thus, the scheme attempts to cause the width of the output color distribution to match the width of the input color distribution on each color plane. As you learned in the earlier lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness this should go a long way toward preventing the colors from becoming *washed out* as in Figure 50.

Back to the sharpening filter

Figure 51 shows the result of applying the sharpening filter *(Filter14.txt)* shown in Figure 48 to the starfish image *(background02.gif)*. This is the same process that was used to produce Figure 50, except that the normalization scheme used in **ImgMod32a** was used to produce Figure 51.

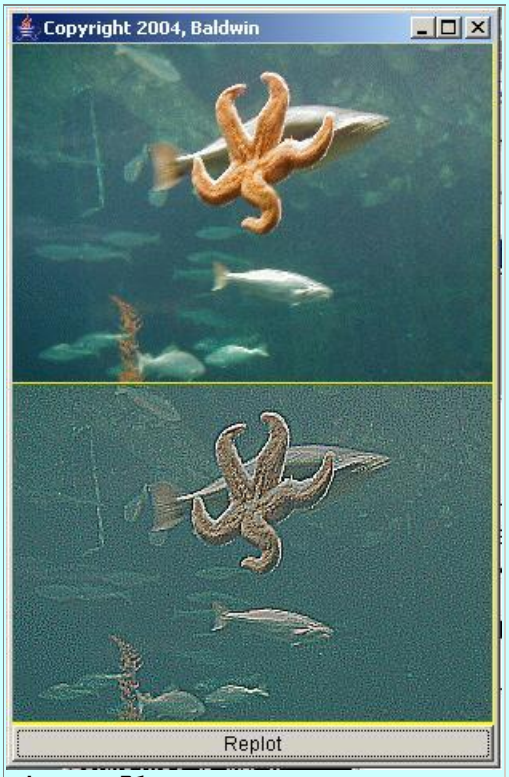

Figure 51

For the record, Figure 51 was produced by executing the following command at the command prompt and specifying the filter file named Filter14.txt at the interactive control panel *(see Figure 4)*:

java ImgMod02a ImgMod33a background02.gif

No scaling was applied to the output at the interactive control panel.

Now, that's more like it

If you compare Figure 51 with Figure 50, you will probably agree that Figure 51 is closer to what we would like to see in a sharpening filter. Thus, the normalization scheme used in **ImgMod32a** may be more appropriate than the normalization scheme used in **ImgMod32** for sharpening photographic images.

The sharpening filter definitely brought out the detail in the bottom image of Figure 51 as compared to the top image in Figure 51. For example, note the nearly horizontal lines in the large fish's tail and the detail showing in the seaweed at the bottom of the image in the bottom

panel. The lines in the fish's tail can barely be seen in the original image at the top. The seaweed is much more blurred in the original image at the top.

From an aesthetic viewpoint, this particular sharpening filter may have brought out a little too much detail. It seems also to have brought out some noise in the background and the image seems to be a little harsh. We'll see the results produced by a somewhat softer sharpening filter shortly.

The mean and RMS values

Just in case you are interested, the input and output mean and RMS values for Figure 51 are shown in Figure 52.

```
Input red mean: 73.35973143759874
Input green mean: 106.97551342812007
Input blue mean: 104.57263823064771
Input red RMS: 45.37941846420423
Input green RMS: 29.99973250165733
Input blue RMS: 27.389329517109754
Output red RMS: 45.37941846420752
Output green RMS: 29.999732501659715
Output blue RMS: 27.38932951711031
Output red mean: 73.35973143760023
Output green mean: 106.97551342812181
Output blue mean: 104.57263823065891
Figure 52
```

A softer sharpening filter

Now let's take a look at the results produced by a 2D convolution filter *(Filter15.txt)* having the coefficient values shown in Figure 53.

```
-0.25 -0.25 -0.25
-0.25   3.0 -0.25
-0.25 -0.25 -0.25
Figure 53
```

Note that as before, the sum of the filter coefficients adds up to 1.0. However, each of the eight negative values in this filter is only 8.3-percent of the central value of 3.0 whereas each of the negative values in Figure 48 is 11.1-percent of the central value of 9.0. As a result, the 2D filter surface represented by Figure 53 is flatter than the filter surface represented by Figure 48.

Apply to the starfish image

Figure 54 shows the result of applying the sharpening filter *(Filter15.txt)* to the starfish image *(background02.gif)* without performing any scaling on the output of the convolution process.

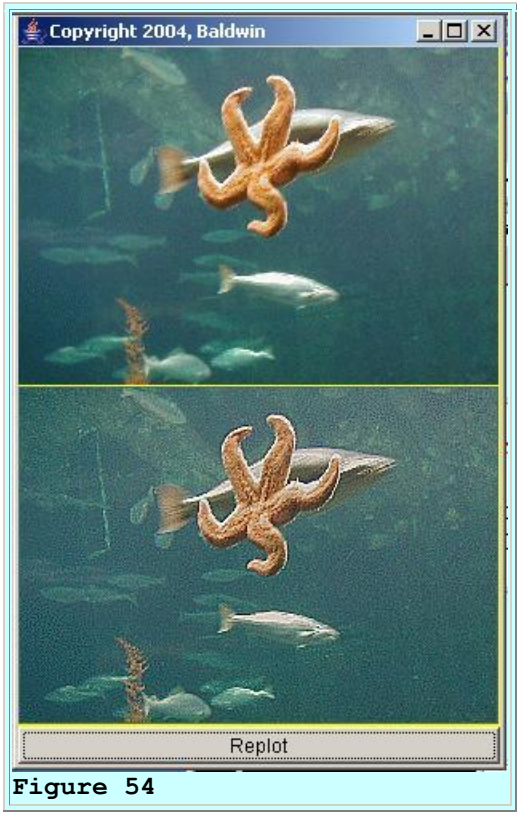

Figure 54

As you can see, the bottom image in Figure 54 is sharper and crisper than the top image, with the bottom image showing more detail. Once again, compare the nearly horizontal lines in the large fish's tail and the seaweed between the two images. However, the bottom image in Figure 54 is softer than the bottom image in Figure 51.

Continue the softening process

We could continue this softening process by reducing the magnitude of the ratio of the negative values to the central positive value in Figure 54 *(being careful to ensure that the sum of the coefficient values is always 1.0)* until we reached the point where the negative values are reduced to zero. At that point, the convolution filter will have degenerated into a simple copy filter having only one non-zero coefficient.

I will leave further experimentation with sharpening filters as an exercise for you to carry out on your own.

Go back and reprocess earlier images

Now that we have a normalization scheme that seems to do a pretty good job when used with sharpening filters on photographic images, let's go back and apply the same scheme to the other kinds of filters:

- Smoothing or softening filters

- Bipolar filters
    - Embossing filters that produce a 3D-like effect
    - Edge detection filters

I will apply some of the same filters to the same images as before using the new normalization scheme so that we can compare the results of the two normalization schemes.

Smoothing or softening filters

Figure 55 shows the result of applying the 4x4 smoothing filter from the file named Filter05.txt to the same image of the starfish. This is the same filter that was applied to the starfish image in Figure 18 using the earlier normalization scheme.

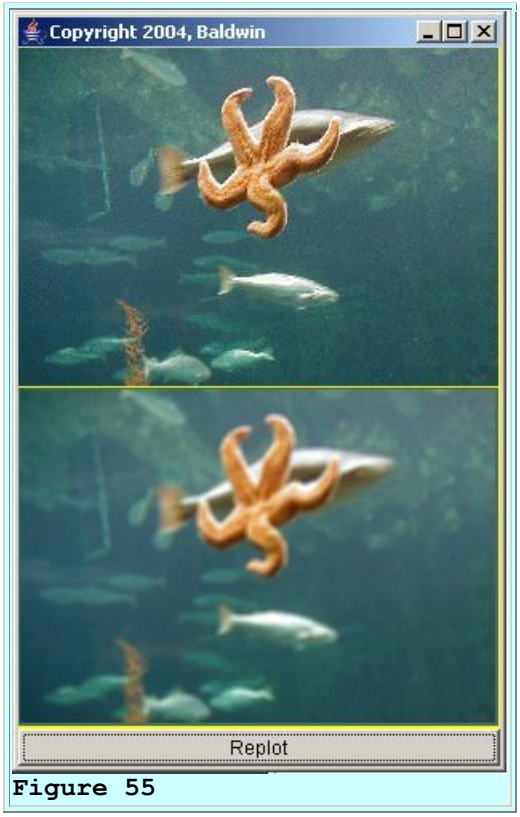

Figure 55

There is very little difference between Figure 55 and Figure 18, so the normalization scheme didn't seem to matter much for the smoothing operation. However, although it may be my imagination, it does appear to me that the color fidelity between input and output may be a little better in Figure 55. That should probably be expected given that the output in Figure 55 has the same mean and the same RMS value as the input.

A rather severe smoothing filter

Figure 56 should be compared with Figure 19. Both of these figures show the result of applying the 10x10 smoothing filter from the file named Filter03 to the image of the starfish. Figure 19

was produced with the earlier normalization scheme. Figure 56 was produced with the later normalization scheme.

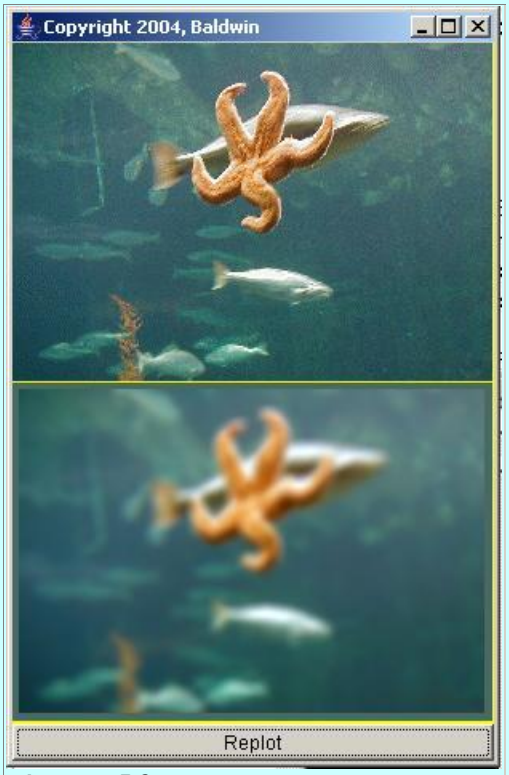

Figure 56

This time, I'm certain that it is not my imagination. The color fidelity from input to output is definitely better in Figure 56 than in Figure 19. *(The starfish is too red in Figure 19.)* Recall that I mentioned the possibility of an undesirable color shift when discussing Figure 19 earlier.

3D embossing filters

Figure 57 should be compared with Figure 34. The two figures show the result of applying the 3D embossing filter *(Filter02.txt)* to the starfish image using the two different normalization schemes.

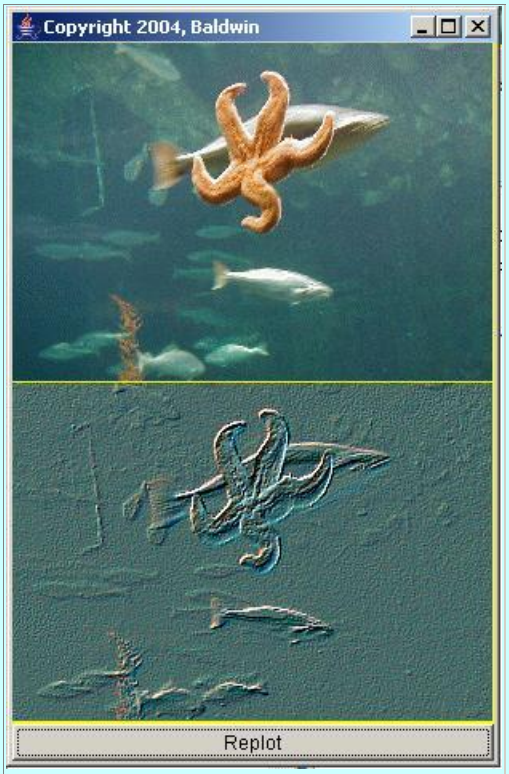

Figure 57

The results in Figure 57 are dramatically different from the results in Figure 34. In this case, the normalization scheme that maintained the width of the color distribution seems to have produced a more pronounced 3D effect. Many features, such as the small fish and the seaweed, that were faded out in Figure 34 are clearly visible in Figure 57.

Moving the light source

While we are at this point, I want to show you something that I haven't shown you before.

Figure 58 is the same as Figure 57 with one major exception.

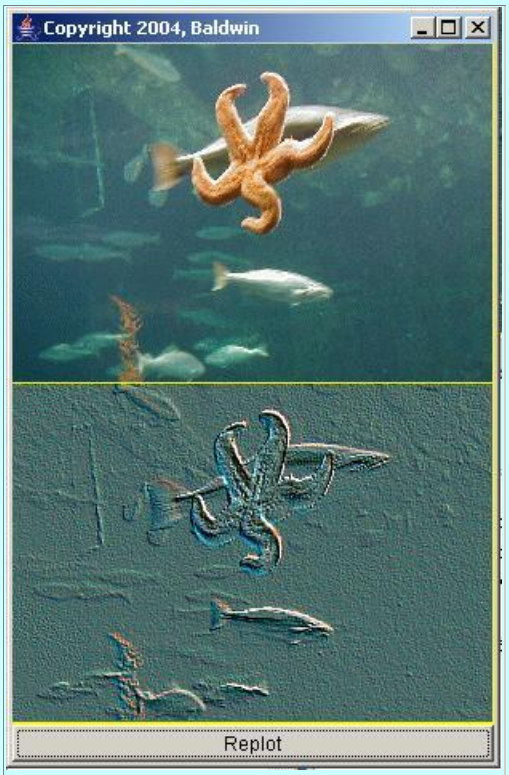

Figure 58

In Figure 57, the filter *(Filter02.txt)* was designed to produce a 3D optical illusion making it appear that the light source is above and to the left of the starfish. The filter *(Filter09.txt)* in Figure 58 was designed to make it appear that the light source is above and to the right of the starfish. Both designs produce a very realistic embossed 3D optical illusion when used with the normalization scheme that maintains the width of the color distribution from input to output.

Edge detection filter

Figure 59 should be compared with Figure 47. Both figures show the result of applying the edge detection filter *(Filter13.txt)* to the starfish image.

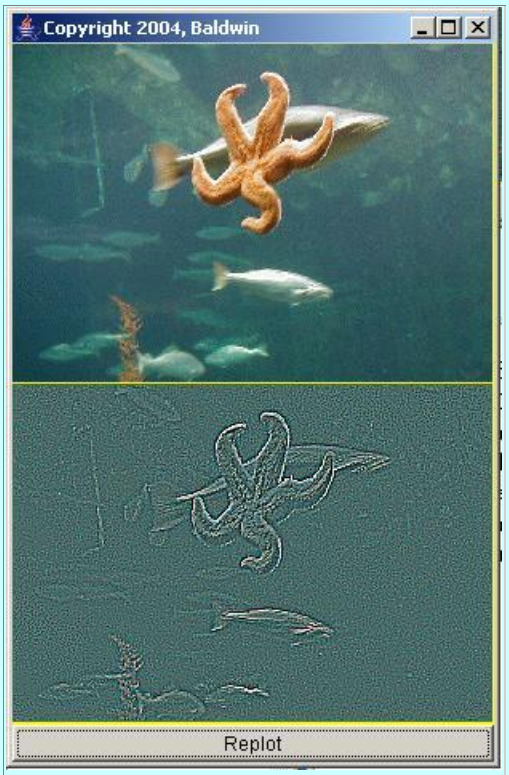

Figure 59

The edge detection output shown in Figure 59 is clearly superior to the output shown in Figure 47. Many edges, such as some of the small fish that are lost in Figure 47 are identifiable in Figure 59.

And the verdict is...

For every case *(from Figure 51 through Figure 59)* but one, where the filter was applied to the photographic image of the starfish, the normalization scheme implemented in the class named **ImgMod32a** seems to be superior to the normalization scheme implemented in the class named **ImgMod32**. The one exception was the case of a smoothing filter where the comparison was something of a toss up with the **ImgMod32a** scheme beating out the other scheme by a whisker.

On the other hand

However, had we evaluated the two schemes against a different image, we may have reached a different conclusion. For example, if you compare Figure 60 with Figure 20, you may conclude that the normalization scheme used for Figure 20 was more successful in producing the 3D optical illusion in this case.

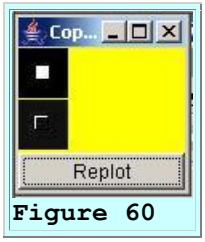

Figure 60

*(Figure 20 was produced using the normalization scheme in the class named **ImgMod32** and Figure 60 was produced using the normalization scheme in the class named **ImgMod32a.**)*

Why bother with two normalization schemes?

By now you may be wondering why I took the time and the effort to walk you through two different normalization schemes if I already knew that one was probably superior to the other. Mainly I wanted to impress on you that digital signal processing *(DSP)* involves much more than simply doing a lot of arithmetic. If all signals of interest were represented as 64-bit floating-point values, DSP may be reduced to that, but that is not how things usually turn out in the real world. DSP usually requires the user to make decisions based on knowledge of the context of the problem.

Historical footnote

During my DSP career *(which admittedly ended several years ago when I retired from the real world and became a college professor)*, I never had the luxury of working with 64-bit floating-point signal data. Virtually all of the real-world data that I worked with was quantized to many fewer than 64 bits. Typically the signals were quantized as integer values, usually in twelve to sixteen bits. For example, here is a quotation describing the sampled data format on an audio CD:

> *"The original musical signal is a waveform in time. A sample of this waveform in time is taken and "digitized" into two 16-bit words, one for the left channel and one for the right channel."*

Integer arithmetic

In addition, many of the DSP arithmetic units that I was privileged to use were integer arithmetic units, making the possibility of arithmetic overflow a real possibility. Even when the arithmetic unit was a floating-point unit, it was almost always necessary to normalize the final results back into an integer format, as is the case with the color data values in this lesson.

A *safe* normalization scheme

I have described two normalization schemes in this lesson. The first scheme, as implemented in the class named **ImgMod32**, is a *safe* scheme in that all of the information in the convolution

output is re-quantized into the required eight-bit unsigned format. None of the data is discarded. The results are clearly more granular, but they are all there.

Unfortunately, this *safe* scheme doesn't always produce pleasing images when applied to photographs, because it can compress the width of the color distribution of the image, causing the image to have a "washed out" appearance.

An unsafe but aesthetically pleasing normalization scheme

The second scheme, as implemented in the class named **ImgMod32a**, often produces more aesthetically pleasing results for the photographic data, but it is not a *safe* scheme. In particular, the process of clipping the final values at 0 and 255 is an *unsafe* nonlinear process. It is entirely possible that valuable information may be discarded in the clipping process.

No single "right" answer

As I indicated earlier, there is no single *right* answer to the questions regarding the normalization of the results. Normalization and re-quantization of data always involves tradeoffs among different alternatives within the context of the overall problem.

In the final analysis, the person responsible for the work must understand the technical ramifications of those alternatives and must make an informed decision as to which scheme or schemes among different alternative schemes will be used.

Conversion to a production program

If I were converting these classes to production software, I would probably give the user three additional options at the interactive control panel:

- Accept the safe normalization discussed above as the default.
- Select the unsafe normalization discussed above.
- Perform normalization similar to the unsafe normalization discussed above, but allow the user to specify the mean value and the RMS value of the final output.

Program Code

The code in the classes used to produce the experimental results shown above will be explained in Part 2 of this lesson.

For the benefit of those of you who might want to start working with that code now, you will find the source code for the classes in the section entitled Complete Program Listings.

Summary

This is Part 1 of a two-part lesson on image convolution. In this lesson, I have walked you through several experiments intended to help you understand why and how 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
- A smoothing or softening filter
- Bipolar filters
    - Embossing filters that produce a 3D-like effect
    - Edge detection filters
    - Sharpening filters

What's Next?

In Part 2 of this lesson, I will explain the code in the classes used to perform the convolution experiments that were explained in this first part of the lesson.

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

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 of lessons 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.

```
/* 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 1

```
/* 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 2

```
/* 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 3

```
/*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 4

```
/*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 5

```
/*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 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 6

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-