Processing Image Pixels, Understanding Image Convolution in Java

Learn how and why image convolution works by examining the changes to the wave-number spectrum produced by image convolution. Also learn how to write the code for a general purpose image-convolution class in Java.

Published: January 10, 2006
By Richard G. Baldwin

Java Programming, Notes # 410

Preface

Next in 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 lesson has two primary objectives:

- To teach you how and why image convolution works.
- To provide you with a general purpose image convolution capability in Java.

The first lesson in the series was titled Processing Image Pixels using Java, Getting Started. The previous lesson was titled Processing Image Pixels, Performing Convolution on Images. 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 sometimes uncommon)* image-processing algorithms by working directly with the pixels.

Viewing tip

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

Background Information

The earlier lesson titled 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.

A three-dimensional array of pixel data as type int

The earlier lessons introduced and explained the concept of a pixel. They also introduced you to the use of three-dimensional arrays of type **int** for maintaining pixel data within the programs.

Will receive and return three-dimensional array of type int

This lesson will provide information to help you understand the use of two-dimensional convolution on images. This will be accomplished through the explanation of several examples. The program that I will present to illustrate the concepts will receive and return a three-dimensional array of type **int** containing data in the same layered format as pixel data in an image.

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 titled 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 titled 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 operator 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 operator with the first **n** samples in the time series.
- Compute an output point value, which is the sum of the products of the convolution operator values and the corresponding time series values. Optionally divide the sum of products by the number of coefficient values in the convolution operator.
- Move the convolution operator one step forward, registering it with the next **n** samples in the time series and compute the next output point 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 operator 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. Note that when the values stored in the array are taken into account, a populated 2D array represents a 3D surface. 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 operator has three dimensions corresponding to width, height, and the values of the coefficients in the operator.)*

Lots of arithmetic required

Because each surface has three dimensions and there are three surfaces to be processed by the convolution operator, the amount of arithmetic that must be performed can be quite large. Although it is possible in certain cases to take advantage of special circumstances to make the arithmetic process more efficient, that won't be emphasized in this lesson. Rather, this lesson will present and explain a general purpose image convolution program that is designed to allow you to apply any two-dimensional convolution operator to an image so long as the operator is smaller than the image.

Steps in the processing

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

- Register the width and height of the convolution operator with the first 2D area at the beginning of 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 operator values and the corresponding input surface values. Optionally divide the sum of products by the number of points in the convolution operator.
- Move the convolution operator 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 in the manner described above. When that row has been completely processed, move the convolution operator to the beginning of the next row, registering it 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 operator extend outside the boundaries of the input surface. This will be taken into account in the program.

Some tricky issues

The convolution of color pixel data exposes some tricky issues for which there are several possible solutions, none of which is necessarily the *right* solution.

Color data is not bipolar data. Rather, color values in an image file consist of eight-bit unsigned values ranging from 0 to 255. Hence, there is an inherent DC offset in the color data. One of the issues is deciding how to deal with this DC offset. The program that I will present in this lesson deals with the issue using only one several possible approaches.

A second issue has to do with the fact that the convolution process can easily produce negative output values as well as values that exceed 255. The issue here is to decide how to restore the results of the convolution operation to the range from 0 to 255 inclusive without loss of information. Once again, the program that I will present in this lesson deals with the issue using only one of several possible approaches.

Supplementary material

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

In preparation for understanding the material in this lesson, I recommend that you make certain that you understand the material in the lessons that are listed in the References section of this lesson.

Preview

The class named ImgMod32

The class named **ImgMod32** that I will present and discuss in this lesson provides a general purpose 2D image convolution capability in the form of a static method named **convolve**. The **convolve** method class receives an incoming 3D array of image pixel data of type **int** containing four color planes *(red, green, blue, and alpha)*. The format of this image data is consistent with the format for image data used in the class named **ImgMod02a**.

> *(See the earlier lesson titled Processing Image Pixels Using Java: Controlling Contrast and Brightness for an explanation of the class named* ***ImgMod02a****.)*

Identification of the color planes

The color planes are identified as follows in terms of their index position in the 3D array:

- 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.

Apply same filter to all three planes

The color 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. This causes the color values to be compatible with standard eight-bit unsigned color values.

The values on the alpha plane are not modified.

Output format matches input format

The **convolve** 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.

Original array not modified

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

Treatment at the edges

An unfiltered dead zone equal to half the filter length is preserved around the edge of the filtered image to avoid any attempt to perform convolution using data from outside the bounds of the image. There is no *right* way to handle the problem at the edge of the image. There simply isn't enough available information to produce correct output values at the edge. This is simply one approach to handling that problem.

Stand-alone testing

Although this class is intended to be used to integrate 2D convolution into other programs, a **main** method is provided so that the class can be tested in a stand-alone fashion.

> *(A future lesson will explain how to integrate this image convolution class into other programs, and will also present several interesting examples of image convolution including edge detection, embossing, and smoothing.)*

Relationship between convolution and wave-number spectrum

In addition, the **main** method illustrates the relationship between convolution in the image domain and the wave-number spectrum of the raw and filtered image.

When the **main** method is executed, 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.

Other classes required

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

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

Source code for the classes in the above list can be found in the lessons listed in the References section of this lesson.

Testing

This program was tested using J2SE 5.0 and WinXP.

Experimental Results

Before getting into the program details, I am going to show you some experimental results.

The **convolve** method of this class applies the same 2D convolution filter to the red, green, and blue color planes in a 3D array containing pixel data. However, the **main** method of this class doesn't actually treat the three color planes as if they contain the color values for an image. Rather, the **main** method simply treats the three color planes as opportune places to construct interesting test surfaces. Then when the convolution filter is applied to the 3D array, it is applied separately to each of those surfaces.

The three surfaces and the convolution filter were chosen to illustrate some important characteristics of the application of convolution filters to images.

A rough ride ahead

It's time to fasten your seatbelt. Unless you already have quite a lot of Digital Signal Processing *(DSP)* experience, you may find the remainder of this section to be technically complex.

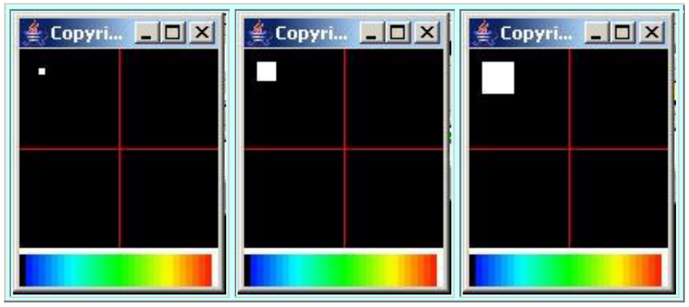
A separate browser window

This would also be an excellent time to open this lesson in a separate browser window. Locate this text in one window and Figure 1 in the other window. Place the two windows side-by-side on your screen so that you can easily refer to Figure 1 while reading the explanation of that material in the text.

A single impulse

A single impulse having dimensions of one pixel on each side and an elevation value of 255 was constructed on the first *(or red)* color surface. You might think of it as looking something like the Washington Monument without the pointy part on the top and being the same size from top to bottom.

When viewed from above, looking straight down on the surface, the impulse appears as shown in the upper-left image in Figure 1.



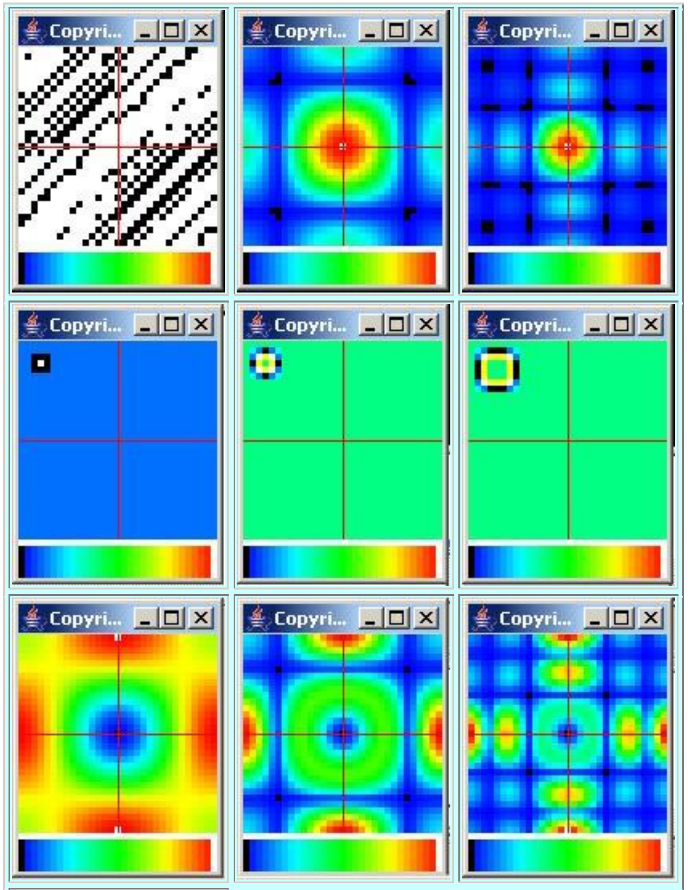


Figure 1

Shorthand notation

Figure 1 contains twelve individual images. Because I will be discussing the individual images at some length, I will need a shorthand way of referring to them. I will refer to the individual images in Figure 1 by row and column. For example, the upper-left image in Figure 1 will be Figure 1-1-1, for:

Figure 1, Row 1, Column 1

The bottom right image will be Figure 1-4-3 for:

Figure 1, Row 4, Column 3

The top of the impulse

The small white square in Figure 1-1-1 is the top surface of the impulse. The dimensions of the impulse are one pixel on each side.

An elevation scale

An elevation scale is shown at the bottom of each image in Figure 1 with black being the lowest elevation, white being the highest elevation, and the elevations in between being represented by the colors shown.

> *(I explained this surface plotting technique in the earlier lesson titled Plotting 3D Surfaces using Java.)*

The black area in Figure 1-1-1 shows that the remainder of the surface outside of the impulse has an elevation of 0. Because the surface elevations in Figure 1-1-1 only have values of 0 and 255, the only colors showing in this image are black and white. *(As you can see, many of the other images in Figure 1 show different colors as well.)*

A square 3x3 tower

A square tower having dimensions of three pixels on each side and an elevation value of 255 was constructed on the second or green color surface. When viewed from above, the 3x3 square tower appears as shown in Figure 1-1-2. You could think of this as nine Washington Monuments placed side-by-side so as to be arranged in a square. Once again, we would need to modify each monument to remove the pointy part on the top and to make the size the same from top to bottom.

A square 5x5 tower

A square tower having dimensions of five pixels on each side and an elevation value of 255 was constructed on the third or blue color surface. When viewed from above, the 5x5 square tower appears as shown in Figure 1-1-3.

Wave-number spectra

The **main** method performs two-dimensional Fourier Transforms on each of the three surfaces shown in the first row of Figure 1, producing the wave-number spectra shown in the second row of Figure 1.

> *(I explained the use of the 2D Fourier Transform to produce wave-number spectra in the lessons titled 2D Fourier Transforms using Java and 2D Fourier Transforms using Java, Part 2.)*

A flat wave-number spectrum

Figure 1-2-1 shows the wave-number spectrum of the impulse shown in Figure 1-1-1. Theoretically, the wave-number spectrum of an impulse is perfectly flat. However, due to small arithmetic inaccuracies, the Fourier Transform of an impulse is never perfectly flat. Rather, it will always have ripples, albeit very small ripples.

In order to preserve plotting dynamic range, the surface-plotting technique used in Figure 1 is normalized for each individual image such that the lowest elevation is plotted as black and the highest elevation is plotted as white. This is true even if the difference between the lowest and highest elevations is very small. Unless every value is exactly the same, there will still be a lowest value, which will be black and a highest value, which will be white.

Thus, the rather ugly result shown in Figure 1-2-1 is a representation of a nearly flat *(but not perfectly flat)* wave-number spectrum.

Non-flat wave-number spectra

The wave-number spectrum in Figure 1-2-2 corresponds to the square 3x3 tower above it in Figure 1-1-2. Similarly, the wave-number spectrum in Figure 1-2-3 corresponds to the square 5x5 tower above it in Figure 1-1-3.

Theoretically, the elevation values in these two spectra peak at a wave number of zero in the center of the image and fall off as a ***sin(x)/x*** function for larger values of wave number in all directions. The rate of the falloff in any particular direction depends on the projected width of the tower along that direction.

If you examine Figure 1-2-2 and Figure 1-2-3 carefully, you will see the white area at the 0,0 origin in the center with small black areas at larger wave numbers. You will also see many of the colors from the color scale showing up at the different wave numbers.

These wave-number spectra extend from a wave number of zero in the center to the Nyquist folding wave number at the edges. The Nyquist wave number is determined by the spacing of the pixels that make up the sampled surface.

The convolution operator

Each of the surfaces shown in the top row of Figure 1 was convolved with the square two-dimensional convolution operator shown in Figure 2, producing the three surfaces shown in the third row of Figure 1.

```
-1,-1,-1
-1, 8,-1
-1,-1,-1
```
Figure 2

The impulse response

When a convolution operator is convolved with an impulse, the result is commonly called the impulse response of the convolution filter. The impulse response is very important. The Fourier Transform of the impulse response shows the response of the convolution filter in the wave-number domain.

> *(Recall from earlier lessons that the image domain in two dimensions is analogous to the time domain in one dimension. Similarly the wave-number*

domain in two dimensions is analogous to the frequency domain in one dimension. Thus, a two-dimensional wave-number response is analogous to a one-dimensional frequency response.)

The impulse response of the convolution filter from Figure 2 is shown in Figure 1-3-1. The Fourier Transform of the impulse response is shown in Figure 1-4-1. Thus, Figure 1-4-1 shows the wave-number response of the convolution operator from Figure 2.

The shape of the impulse response

The impulse response shown in Figure 1-3-1 traces out a scaled version of the shape of the convolution filter shown in Figure 2. The black areas in Figure 1-3-1 represent the values of -1 in Figure 2. The white area in Figure 1-3-1 represents the value of +8 in Figure 2. The blue area in Figure 1-3-1 represents an elevation of zero.

Thus, the elevation values for the surface shown in Figure 1-3-1 extend from -255 *(-1 * 255)* to + 2040 *(+8 * 255).*

Zero response at the wave-number origin

By inspection, we can determine that the response of this convolution filter is zero at a wave number of zero because the sum of the positive values equals the sum of the negative values.

(If we were to apply this filter to a perfectly flat surface, the positive values in the sum of products computation would equal the negative values in the sum of products computation resulting in a net zero value for the sum of products at every location on the flat surface. A flat surface has a wave number of zero.)

Thus, we would expect the wave-number response of the convolution filter to go through 0 at the origin.

Brief analysis of the wave-number response

Probably the most important characteristic of the wave-number response of the convolution filter shown in Figure 1-4-1 is that it has a minimum response at the origin with the maximum response occurring at wave numbers near the edges of the wave-number plot. Further, the transition from the minimum at the center to the maxima at the edges seems to be a relatively smooth curve.

A high-pass filter

The conclusion is that when this convolution filter is applied to a surface, it will suppress the low wave numbers belonging to the surface and enhance the high wave numbers. Thus, it is a *high-pass* filter in the wave-number domain.

Applying the convolution operator to the 3x3 tower

This is exactly what happens when the convolution operator is applied to the 3x3 tower shown in Figure 1-1-2. The surface that results from that convolution is shown in Figure 1-3-2. The resulting surface does not consist of a 3x3 tower. Rather, it is more like a hollow 5x5 structure.

Because the surface has negative elevation values, an elevation value of zero in Figure 1-3-2 is represented by the green area instead of a black area. Black is reserved for the lowest elevation value which is a negative value in this case.

A cross section through the structure

If you trace out the elevation of the structure along a line running from West to East through the center of the structure, you find the following colors in order:

- green - this is the baseline at an elevation of 0
- black - go way down into negative territory
- yellow - go way up to about 80-percent of the highest possible value
- green - go back down to the baseline
- yellow - go way up again to about 80-percent of the highest possible value
- black - go way down again into negative territory
- green - go back to the baseline and stay there for the remainder of the trip

Needs high wave-number components

Those of us who are familiar with this sort of thing will immediately recognize that such a structure must be rich in high-valued wave-number components and that is true in this case. An important clue is the ability of the surface to make very rapid transitions from low values to high values and back again to low values. That is not possible in a surface made up mostly of low-valued wave-number components.

> *(This is analogous to the fact that in order for a time series to make rapid transitions from low to high values and back to low values, the time series must be rich in high-frequency components.)*

Wave-number spectrum of the convolution output

The wave-number spectrum of the output from the convolution process is shown in Figure 1-4-2. As you can see, the peak energy in this spectrum occurs in the red zones in the wave numbers at the edges of the wave-number plot. The energy in the origin at the center of the plot is very low because it is colored dark blue.

A very important concept

You should make it a point to remember this fact.

> **Convolution in the image domain is analogous to multiplication in the wave-number domain.**

The process of convolving the convolution filter shown in Figure 2 with the 3x3 tower shown in Figure 1-1-2 is equivalent to multiplying the wave-number response of the convolution operator *(Figure 1-4-1)* by the wave-number spectrum of the 3x3 tower *(Figure 1-2-2)* to produce the wave-number spectrum shown in Figure 1-4-2.

Wave-number spectrum must be low at the origin

Because the wave-number response of the convolution filter shown in Figure 1-4-1 is very low *(theoretically zero)* at the origin, the product of that response with the spectrum of the 3x3 tower shown in Figure 1-2-2 must also be very low at the origin regardless of the value of the spectrum of the 3x3 tower at the origin. *(Any value, no matter how large, multiplied by zero produces zero.)* This is borne out by Figure 1-4-2 which is dark blue *(possibly black)* at the origin.

Rich in high wave-number components

Because the wave-number response of the convolution filter is high in the red areas near the edges of the wave-number plot, and the wave-number spectrum of the 3x3 tower is not particularly low in those same wave-number regions, the product of the two can be expected to be high in those regions.

This is borne out by the wave-number spectrum in Figure 1-4-2 where the red peaks occur at the edges in the North, South, East, and West directions.

Because of the yellow and red areas at high wave-number values in Figure 1-4-2, the surface in Figure 1-3-2 is rich in high wave-number values as I predicted it must be to produce the shape of the surface shown in Figure 1-3-2.

Now on to the 5x5 tower surface

The result of convolving the convolution operator with the 5x5 tower shown in Figure 1-1-3 is similar to the result for the 3x3 tower. However, because the 5x5 tower has a larger footprint, its wave-number energy is more concentrated near the origin of its spectrum as shown in Figure 1-2-3.

The convolution output in the image domain

The convolution of the operator with the 5x5 tower produces a hollow output shape with a larger footprint but generally the same values in the walls of the structure.

The product of the wave-number response of the convolution filter in Figure 1-4-1 with the spectrum of the 5x5 tower shown in Figure 1-2-3 produces the wave-number spectrum shown in Figure 1-4-3. Once again, the resulting wave-number spectrum is rich in high wave-number values resulting in the hollow shape with rapid transitions from low to high elevations shown in Figure 1-3-3.

Summary of experimental results

In both cases, the original surfaces shown in Figure 1-1-2 and Figure 1-1-3 were rich in low wave-number components. This is evidenced by the spectra shown in Figures 1-2-2 and Figure 1-2-3 with large peaks at the origin in wave-number space.

However, the convolution filter has a very low response to low wave numbers and a high response to high wave numbers as shown by the wave-number response in Figure 1-4.1.

The application of the convolution filter to each of the two surfaces suppresses components with low wave numbers and preserves components with high wave numbers in the output as shown in Figures 1-4-2 and Figure 1-4-3.

More than you ever wanted to know

And that is probably more than you ever wanted to know about the relationships among:

- Image convolution
- The wave-number response of the convolution filter
- The wave-number spectrum of the surface being filtered, and
- The wave-number spectrum of the result of the convolution process

It's time to lighten up a bit and discuss the program code.

Discussion and Sample Code

The class named ImgMod32

I will discuss and explain the code in this class in fragments. The source code is presented in its entirety in Listing 20 near the end of the lesson.

Three categories of code

The code in this class falls into three categories:

- Ten utility methods.
- The method named **convolve**.
- The method named **main**, which is used to run and test the class on a stand-alone basis.

The utility methods

I will discuss the utility methods first. All of the utility methods are short and to the point. The code in these methods is straightforward and shouldn't require a detailed explanation. The code for each of the utility methods can be found in Listing 20 near the end of the lesson.

A brief description of each of the utility methods follows:

- **getPlane** - Extracts a color plane from a **double** version of an image and returns it as a 2D array of type **double**. Used only in support of the test operations in the **main** method. Not required for performing the convolution.
- **removeMean** - Removes the mean value from a specified color plane in the **double** version of an image pixel array. Returns the mean value that was removed so that it can be saved by the calling method and restored later.
- **addConstantToColor** - Adds 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.
- **scaleColorPlane** - Multiplies every color value in a specified color plane in the **double** version of an image pixel array by a specified scale factor.
- **getMax** - Returns the algebraic maximum color value for a specified color plane in the **double** version of an image pixel array.
- **getMin** - Returns the algebraic minimum color value for a specified color plane in the **double** version of an image pixel array.
- **intToDouble** - Converts an image pixel array *(where the pixel values are represented as type **int**)* to an image pixel array where the pixel values are represented as type **double**.
- **doubleToInt** - Converts an image pixel array *(where the pixel values are represented as type **double**)* to an image pixel array where the pixel values are represented as type **int**.
- **getMaxColor** - returns the maximum value among three color values where the color values are represented as type **double**.
- **getMinColor** - returns the minimum value among three color values where the color values are represented as type **double**.

Now that you are familiar with the purpose of each of the utility methods, you will be better prepared to understand the code that follows.

The convolve method

This method applies an incoming 2D convolution filter to each color plane in an incoming 3D array of pixel data of type **int** and returns a filtered 3D array of pixel data of type **int**.

The convolution operator 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.

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.

The **convolve** method begins in Listing 1.

```
  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);
```

Listing 1

The code in Listing 1 gets, saves, and displays the dimensions of the incoming arrays containing the pixel data and the convolution operator. These dimensions are frequently used in **for** loops in the code that follows.

Copy the incoming pixel array

Listing 2 makes a working copy of the incoming 3D pixel array to avoid making permanent changes to the original image data. The pixel data of type **int** is converted to type **double** in the process. The filtered pixel data will be converted back to type **int** when those results are returned by the method.

```
    double[][][] work3D = intToDouble(threeDPix);
```

Listing 2

Remove the mean value

One of the issues that we encounter when processing color pixels arises from the fact that the color pixel data is not bipolar data. Rather, every color value is described by a positive integer having a value between 0 and 255.

Many digital signal processing techniques work best when applied to bipolar data, preferably with a mean value of zero. However, the format of the color values results in an inherent non-zero mean bias across all the values on a color plane. Without getting into the details, the application of convolution to data with a significant non-zero mean bias can result in arithmetic complications and degrade the accuracy of the process.

There are several ways to deal with this issue, none of which is clearly the *right* way. I decided to remove the mean color bias from all the values on each color plane prior to convolution and then to restore the same mean color bias to the values on each color plane following convolution. Thus my approach doesn't change the bias when the color values on a plane are convolved with the convolution operator.

Listing 3 removes the mean value from each color plane and saves the values for later restoration.

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

Listing 3

Create an output array

Listing 4 creates an output array the same size as the incoming array of pixels. Then Listing 4 copies the alpha values directly to the output array. The alpha values are not modified during the convolution process.

```
    //Create an empty output array.
    double[][][] output =
                     new
double[numImgRows][numImgCols][4];

    //Copy the alpha values directly to the output array.
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        output[row][col][0] = work3D[row][col][0];
      }//end inner loop
    }//end outer loop
```

Listing 4

Perform the actual convolution

Listing 5 shows the beginning of four nested **for** loops that are used to convolve each of the three 2D color planes with the 2D convolution operator.

```
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
```

Listing 5

Rather than to try to explain the code in Listing 5, I am going to refer you back to the earlier section titled Steps in the processing for a general description as to what is taking place in Listing 5.

The code in Listing 5 is complicated by the fact that the surface being convolved is of a finite size and care must be taken to avoid trying to do arithmetic using values outside the dimensions of the surface. As mentioned earlier, the code in Listing 5 leaves an unfiltered dead zone equal to half the filter length around the perimeter of the filtered image to avoid this potential problem.

Divide by the number of convolution points

Note that the code in Listing 5 does not include the termination of the two outer **for** loops.

The typical definition of convolution involves computing the sum of products and then dividing the sum by the number of points in the convolution filter. The rationale for this usually assumes a very wide dynamic range for the bipolar values in the data being convolved. Ultimately, however, the dynamic range for color values is quite low, only one part in 127. For this situation, performing the division described above isn't always helpful. In particular, it is not helpful when a large percentage of the filter coefficients have a value of zero.

No clear *right* or *wrong* way

Once again, there is no *right* or *wrong* way to deal with this issue. I included the code to perform this division in Listing 6 but disabled that code when producing the experimental results shown in Figure 1.

> *(Although including or excluding the division by the number of convolution filter points would have a negligible impact on the results shown in Figure 1, it will have a significant impact on some of the convolution examples that I will show you in the next lesson in this series. Therefore, I show that code as being disabled in Listing 6.)*

```
/*
    output[yReg-numFilRows/2][xReg-numFilCols/2][1] =
           output[yReg-numFilRows/2][xReg-
numFilCols/2][1]/

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

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

(numFilRows*numFilCols);
*/
  }//End loop on xReg
}//End loop on yReg
```

Listing 6

Listing 6 also shows the termination of the two outer **for** loops that began in Listing 5. When the code in Listing 5 and Listing 6 has been executed, all of the convolution arithmetic has been completed.

Restore the mean values

As mentioned earlier, the approach that I use in this class to deal with the fact that the color data is not bipolar is to remove the mean value from each color plane prior to applying the convolution filter and then restoring the original mean value to each color plane following convolution. Listing 7 restores the original mean values to each of the color planes.

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

Listing 7

Normalize the results

Another major issue in convolving color data arises from the fact that convolution can easily produce new values that are less than zero or greater than 255. Such results must be converted back into the range from 0 to 255 before attempting to restore them into an actual color image. This is another case where there isn't a clear *right* or *wrong* way to do the job.

In this program, I begin by dealing with the minimum algebraic color value. If any color plane contains a negative value, I add a constant to all color values in all three color planes to guarantee that the minimum value in any color plane is 0.

I start by getting the minimum value of the filtered output across all color planes. Then I make the same adjustment to every color value in every plane if the minimum value is less than zero. Otherwise, I don't make any adjustment on the basis of the minimum value.

Having done that, I next deal with the maximum color value. If any color value in any color plane exceeds 255, I multiply all color values in all planes by the same scale factor to guarantee that the maximum color value in any color plane is 255.

First I get the peak value of the filtered output across all color planes. Then I make the adjustment if the maximum value is greater than 255. Otherwise, I don't make any adjustment on the basis of the maximum value.

Deal with the minimum color value

Listing 8 deals with the minimum color value as described above.

```
    //Get the minimum value of the filtered output across
    // all color planes.
    //Get the minimum value for each plane.
    double redOutMin = getMin(output,1);
    double greenOutMin = getMin(output,2);
    double blueOutMin = getMin(output,3);
    //Get and save the minimum color value among all three
    // color planes
    double minOut = getMinColor(

redOutMin,greenOutMin,blueOutMin);

    //Make the adjustment to every color value if the
    // minimum value is less than zero.
    if(minOut < 0){
      addConstantToColor(output,1,-minOut);
      addConstantToColor(output,2,-minOut);
      addConstantToColor(output,3,-minOut);
    }//end if

Listing 8
```

The code in Listing 8 is straightforward and shouldn't require any explanation beyond that already given.

Deal with the maximum color value

Listing 9 deals with the maximum color value in the manner described earlier.

```
    //Get the peak value of the filtered output across all
    // color planes.
    //Get the maximum value for each color plane.
    double redOutMax = getMax(output,1);
    double greenOutMax = getMax(output,2);
    double blueOutMax = getMax(output,3);
    //Get and save the maximum color value among all three
    // color planes
    double peakOut = getMaxColor(

redOutMax,greenOutMax,blueOutMax);

    //Make the adjustment if the maximum value is greater
    // than 255.
    if(peakOut > 255){
      scaleColorPlane(output,1,255/peakOut);
      scaleColorPlane(output,2,255/peakOut);
      scaleColorPlane(output,3,255/peakOut);
    }//end if
```

Listing 9

The code in Listing 9 is straightforward and shouldn't require further explanation.

Return the convolved results

Listing 10 converts the filtered **double** color values to a 3D array containing color value of type **int** and returns a reference to that array.

```
    return doubleToInt(output);

  }//end convolve method
```

Listing 10

Listing 10 also signals the end of the method named **convolve**. If the class didn't contain a **main** method used for self testing, we would be finished at this point. However, it does contain a **main** method, which is the next topic for discussion.

The main method

As mentioned earlier, this class is designed to be integrated into other programs in order to provide 2D convolution capability for those programs. However, the class also contains a **main** method that can be used to test the class in a stand-alone fashion.

> *(The next lesson in this series will teach you how to integrate this class into other image pixel processing programs.)*

The primary purpose

The primary purpose of the **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.

A nine-point convolution filter

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 an extreme sharpening or edge-detection filter.

The three surfaces

The three surfaces consist of:

- A single impulse.
- A 3x3 square tower.
- A 5x5 square tower.

The three surfaces are constructed on what is ordinarily 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.

However, to be consistent with the concept of color planes, the comments in the **main** method frequently refer to the values as color values.

Graphic display

In addition to the display of some text material on the command-line screen, the program displays the twelve different graphs shown in Figure 1. Six show image-domain data and the other six show spectral data.

Display of image-domain surfaces

The following image-domain surfaces are displayed:

- The impulse.
- The raw 3x3 square tower.
- The raw 5x5 square tower.
- The filtered impulse.
- The filtered 3x3 square tower.
- The filtered 5x5 square tower.

Display of wave-number surfaces

In addition, a 2D Fourier Transform is computed and the spectral results are displayed for the following surfaces:

- The impulse.

- The 3x3 square tower.
- The 5x5 square tower.
- The filtered impulse
- The filtered 3x3 square tower.
- The filtered 5x5 square tower.

The convolution filter

Listing 11 shows the beginning of the **main** method and the definition of the nine-point, 2D convolution filter array.

```
  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}
                        };

Listing 11
```

(Note that when the values are taken into account, a populated 2D array represents a 3D surface.)

Create the three surfaces

Listing 12 creates synthetic image pixel data for a 3D image array containing three color planes and an alpha plane. The color surfaces in the image array are of a size that is sufficiently large to produce good resolution in the 2D Fourier Transform that will be performed later. Those portions of the color surfaces that don't describe the shapes of interest are filled with pixel values of zero.

```
    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;
```

Listing 12

Perform the convolution

Listing 13 convolves each of the three color planes with the convolution filter described above.

```
    int[][][] output = convolve(threeDPix,filter);
```

Listing 13

All of the remaining code in the **main** method is used to construct and display the surfaces shown in Figure 1.

Convert to double and remove the mean values

The **convolve** method performs all of its computations as type **double**, but converts the filtered results to type **int** before they are returned. In order to preserve dynamic range during the computation of the 2D Fourier Transforms and the display of the 3D surfaces, it is useful to convert the color values from type **int** to type **double** and to remove the mean values from each of the filtered color planes. This is accomplished in Listing 14.

```
    //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);
```

Listing 14

For the same reasons, it is useful to convert the raw image data from type **int** to type **double** before performing the 2D Fourier Transforms. This is accomplished in Listing 15.

```
    double[][][] rawDouble = intToDouble(threeDPix);
```

Listing 15

Very repetitive code

The code required to produce the images in each of the three columns in Figure 1 is very similar. I will explain the code used to produce the four images in the left-most column and leave it up to you to study the code for the other two columns. You will find all of the code in Listing 20 near the end of the lesson.

Plot the surface for the red plane

Listing 16 produces the image in Figure 1-1-1.

```
    //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);
```

Listing 16

The code in Listing 16 gets and plots the surface that was earlier deposited on the red color plane, which is the plane at index 1 in the 3D array of pixel data.

The code begins by invoking the **getPlane** utility method to get the plane of interest. Then it instantiates a new object of the class named **ImgMod29** to construct and display the graph representing the surface. The class named **ImgMod29** was explained in detail the previous lesson titled Plotting 3D Surfaces using Java, so I won't repeat that explanation here.

Display 2D Fourier Transform of Plane 1

Listing 17 computes and plots the wave-number spectrum of Plane 1 as shown in Figure 1-2-1.

```
    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);
```

Listing 17

Get the plane

As before, Listing 17 begins by invoking the **getPlane** method to get access to the plane of interest.

Perform the 2D Fourier Transform

Then Listing 17 prepares three arrays to receive the results produced by the 2D Fourier Transform process. Of the three, only the array containing the amplitude data will be displayed. The other two arrays of data will simply be discarded.

Then Listing 17 invokes the static method named **xform2d** belonging to the class named **ImgMod30** to perform the 2D Fourier Transform. This class and method were explained in detail in the earlier lesson titled 2D Fourier Transforms using Java. Therefore, I won't repeat that explanation here.

Prepare wave-number spectrum for plotting

Then Listing 17 invokes the static method named **shiftOrigin** belonging to the class named **ImgMod30** to rearrange the wave-number data and make it more suitable for plotting. In a nutshell, this method rearranges the results of the Fourier Transform to place the wave-number origin at the center of the plot instead of at the upper-left corner. This class and method were also explained in the earlier lesson titled 2D Fourier Transforms using Java so I won't repeat that explanation here.

Plot the wave-number spectrum

Finally, Listing 17 instantiates a new object of the class named **ImgMod29** to plot the surface shown in Figure 1-2-1.

Plot the filtered plane

Listing 18 plots the filtered version of the plane at index 1. This is the impulse response of the convolution filter shown in Figure 1-3-1.

```
    temp = getPlane(outputDouble,1);
    new ImgMod29(temp,4,true,1);
```

Listing 18

By now, the code in Listing 18 should be familiar and shouldn't require further explanation.

Plot the wave-number response of the convolution filter

Listing 19 computes and plots the wave-number response of the convolution filter by performing a 2D Fourier Transform on the impulse response of the convolution filter. The result is shown in Figure 1-4-1.

```
    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);
```

Listing 19

Everything in Listing 19 should be very familiar to you by now.

The remainder of the main method

The remainder of the code in the **main** method performs very similar operations on the other two color planes to produce the images in the middle column and the right-most column in Figure 1. Because of the similarity of the remaining code to the code that I have already discussed, I won't discuss the remaining code further. As mentioned earlier, the remaining code is available in Listing 20 near the end of the lesson.

Run the Program

I encourage you to copy, compile, and run the program in Listing 20. Experiment with the code, making changes and observing the results of your changes. For example, you might want to try modifying the convolution filter shown in Figure 2 and see if you can explain the results produced using your new convolution filter.

You might also want to try creating some different surfaces on the three color planes. For example, replace the 5x5 column with a surface shaped like a pyramid and see if you can explain the results of convolving it with my convolution filter, or with a convolution filter of your own design. Then change the footprint from a square to a circle and change the pyramid to a cone and see if you can explain the results.

Try replacing the synthetic image with the data from a real image.

> *(You should be able to find the necessary code to read an image file in an earlier lesson in this series.)*

Then run the program to see the effects of image convolution in both the image domain and the wave-number domain. This should show you wave-number spectral information for each color plane of your chosen image both before and after convolution filtering. See if you can explain the wave-number spectral data for your image.

Have fun and learn

Above all, have fun and use this program to learn as much as you can about performing 2D convolution on images.

Other required classes

In order to compile and run the class named **ImgMod32**, you will need the following class files.

- ImgMod29.class - See the source code in the previous lesson titled Plotting 3D Surfaces using Java.
- ImgMod30.class - See the source code in the previous lesson titled 2D Fourier Transforms using Java.

Summary

In this lesson, I have attempted to teach you how and why image convolution works by examining the changes to the wave-number spectrum produced by image convolution.

I also provided you with a general purpose image convolution class and explained the code for the class in detail.

What's Next?

In the next lesson in the series, I will teach you how to integrate the general purpose image convolution capability provided in this lesson into other programs. I will also provide some interesting convolution examples including edge detection, embossing, sharpening, and softening of images.

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

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

References

In preparation for understanding the material in this lesson, I recommend that you study the material in the following previously-published lessons. You will find links to these lessons at Digital Signal Processing-DSP.

- 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

Complete Program Listing

A complete listing of the class discussed in this lesson is provided in Listing 20.

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 API *(JAI)* has a number of built-in special effects if you prefer to write your own production image-processing programs using Java.

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

```
/*File ImgMod32.java
Copyright 2005, R.G.Baldwin

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 illustrates 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
ImgMod32.class - This class

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

class ImgMod32{
  //The primary purpose of this main method is to test the
  // class in a stand-alone mode.  A secondary purpose is
  // to illustrate the relationship between convolution
  // filtering in the image domain and the spectrum of the
  // raw and filtered images in the wave-number domain.

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

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

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

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

  //The following surfaces are displayed:
  // 1. The impulse
  // 2. The raw 3x3 square
  // 3. The raw 5x5 square
  // 4. The filtered impulse
  // 5. The filtered 3x3 square
  // 6. The filtered 5x5 square

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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


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

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


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

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


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

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


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

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

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

  //The purpose of this method is to extract a color plane
  // from the double version of an image and to return it
  // as a 2D array of type double.  This is useful, for
  // example, for performing Fourier transforms on the data
  // in a color plane.
  //This method is used only in support of the operations
```

```
// in the main method.  It is not required for performing
// the convolution.

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

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

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

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

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

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

  double mean = sum/(numImgRows*numImgCols);

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

  return mean;
}//end removeMean
```

```
//-------------------------------------------------------//

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

  double maxValue = -Double.MAX_VALUE;
  //Find the maximum value
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      if(inputImageArray[row][col][plane] > maxValue){
        maxValue = inputImageArray[row][col][plane];
      }//end if
    }//end inner loop
  }//end outer loop
  return maxValue;
}//end getMax
//-------------------------------------------------------//

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

  double minValue = Double.MAX_VALUE;
  //Find the minimum value
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      if(inputImageArray[row][col][plane] < minValue){
        minValue = inputImageArray[row][col][plane];
      }//end if
    }//end inner loop
  }//end outer loop
  return minValue;
}//end getMin
//-------------------------------------------------------//

//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 represented 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 represented 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 get and return the
// maximum value among three color values where the color
// values are represented as type double.
static double getMaxColor(
                    double red,double green,double blue){
  double max = -Double.MAX_VALUE;
  if(red > max){
    max = red;
  }//end if

  if(green > max){
    max = green;
  }//end if

  if(blue > max){
    max = blue;
  }//end if

  return max;
}//end getMaxColor
//-------------------------------------------------------//

//The purpose of this method is to get and return the
// minimum value among three color values where the
// color values are represented as type double.
static double getMinColor(
                    double red,double green,double blue){
  double min = Double.MAX_VALUE;
  if(red < min){
    min = red;
  }//end if

  if(green < min){
    min = green;
  }//end if

  if(blue < min){
    min = blue;
  }//end if

  return min;
}//end getMinColor
//----------------------------------------------------------//

//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 operator 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.
//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);
  //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);

  //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.

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

  //Normalize the color values, if necessary, to
  // guarantee that they fall within the range
  // from 0 to 255.

  //Begin by dealing with the minimum algebraic color
  // value. If any color plane contains a negative value,
  // add a constant to all color values in all planes to
  // guarantee that the minimum value in any color plane
  // is 0.

  //Get the minimum value of the filtered output across
  // all color planes.
  //Get the minimum value for each plane.
  double redOutMin = getMin(output,1);
  double greenOutMin = getMin(output,2);
  double blueOutMin = getMin(output,3);
  //Get and save the minimum color value among all three
  // color planes
  double minOut = getMinColor(
                    redOutMin,greenOutMin,blueOutMin);

  //Make the adjustment to every color value if the
  // minimum value is less than zero.  Otherwise, don't
  // make any adjustment on the basis of the minimum
  // value.
  if(minOut < 0){
    addConstantToColor(output,1,-minOut);
    addConstantToColor(output,2,-minOut);
    addConstantToColor(output,3,-minOut);
  }//end if

  //If any color value exceeds 255, scale all color
  // values to guarantee that the maximum color value in
  // any  color plane is 255.

  //Get the peak value of the filtered output across all
  // color planes.
  //Get the maximum value for each color plane.
  double redOutMax = getMax(output,1);
  double greenOutMax = getMax(output,2);
  double blueOutMax = getMax(output,3);
  //Get and save the maximum color value among all three
  // color planes
  double peakOut = getMaxColor(
                       redOutMax,greenOutMax,blueOutMax);

  //Make the adjustment if the maximum value is greater
  // than 255.  Otherwise, don't make any adjustment on
  // the basis of the maximum value.
  if(peakOut > 255){
    scaleColorPlane(output,1,255/peakOut);
    scaleColorPlane(output,2,255/peakOut);
    scaleColorPlane(output,3,255/peakOut);
  }//end if

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

}//end convolve method
//-----------------------------------------------------//
}//end class ImgMod32
```

Listing 20

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 Gaussian smooth blur image jpg color linear DSP 3D 2D frequency spectrum wave number Fourier Transform amplitude impulse time-series

-end-