Using the Java 2D ConvolveOp Filter Class to Process Images

Learn how to write programs that use the ConvolveOp class of the Java 2D API to perform two-dimensional image convolution. Also learn about some of the weaknesses of the ConvolveOp class that result from a lack of options for dealing with convolution output values greater than 255 and less than 0.

Published: August 28, 2007
By Richard G. Baldwin

Java Programming Notes # 460

Preface

General

In an earlier lesson titled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see Resources)*, I taught you how to write a framework program that makes it easy to use the image-filtering classes of the Java 2D API to process the pixels in an image and to display the processed image.

At the close of that lesson, I told you that future lessons would teach you how to use the following image-filtering classes from the Java 2D API:

- **LookupOp**
- **AffineTransformOp**
- **BandCombineOp**
- **ConvolveOp**
- **RescaleOp**
- **ColorConvertOp**

In several of the previous lessons listed in the Resources section, I taught you how to use the **LookupOp**, the **AffineTransformOp**, and the **BandCombineOp** image-filtering classes.

In this lesson, I will teach you how to use the **ConvolveOp** image-filtering class to perform a variety of filtering operations on images.

I will teach you how to use the remaining classes from the above list in future lessons.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find a consolidated index at www.DickBaldwin.com.

General background information

Constructing images

Before getting into the programming details, it may be useful for you to review the concept of how images are constructed, stored, transported, and rendered in Java *(and in most modern computer environments for that matter).*

I provided a great deal of information on those topics in the earlier lesson titled "Processing Image Pixels using Java, Getting Started" *(see Resources)*. Therefore, I won't repeat that information here. Rather, I will simply refer you back to the earlier lesson.

The framework program named ImgMod05

It will also be useful for you to understand the behavior of the framework program named **ImgMod05**. Therefore, I strongly recommend that you study the earlier lesson titled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see Resources)*.

However, if you don't have the time to do that, you should take a look at the earlier lesson titled "Using the Java 2D LookupOp Filter Class to Process Images" *(see Resources)*, in which I summarized the behavior of the framework program named **ImgMod05**.

ConvolveOp Examples

In my opinion, the **ConvolveOp** class is the weakest of the image-filtering classes in the Java 2D API. I will explain my reasons for this opinion later in the section titled Assessment. First, however, I will show you some examples of image convolution using the capabilities of the **ConvolveOp** class.

Edge treatment

When performing image convolution, you must always decide how you are going to treat the edges of the image. The **ConvolveOp** class provides the following choices:

- Copy edge pixels in unmodified form
- Zero fill the edge pixels

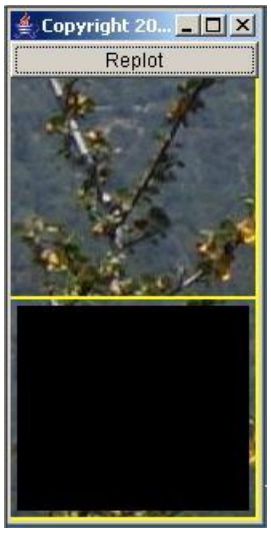
Figures 1 and 2 show the results of electing each of those two choices.

Emphasizing edge treatment
The processed portion of the images in Figures 1 and 2 was purposely blacked out or set to white to emphasize the treatment of the edges.

Copy edge pixels in unmodified form

In Figure 1, the pixels at the edges of the input image were simply copied to the edges of the output image without modification. This is evidenced by the fact that the edges of the output image look just like the edges of the input image.

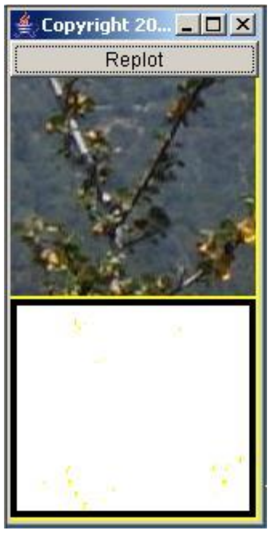
Figure 1. Illustration of an edge treatment option.



Zero fill the edge pixels

In Figure 2, the pixels at the edge of the output image were set to zero, producing the black border around the output image.

Figure 2. Illustration of another edge treatment option.

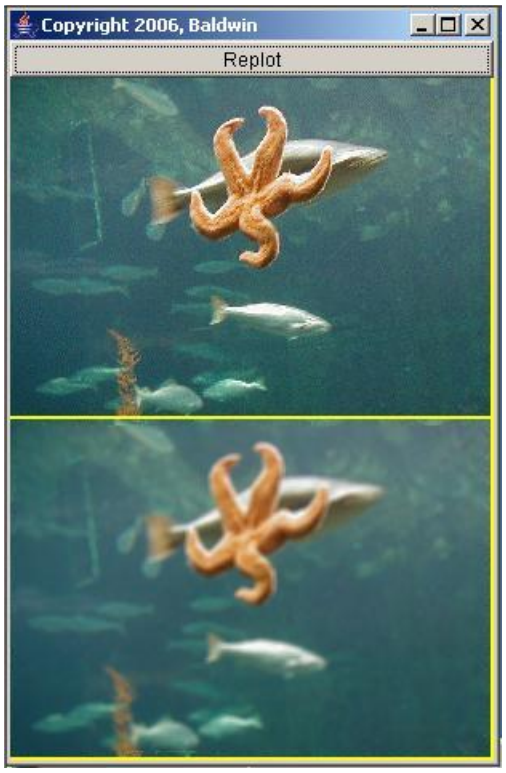


The remaining examples will use the first alternative and simply copy the pixels from the edges of the input image to the edges of the output image in unmodified form.

A flat 4x4 smoothing filter

Figure 3 shows the result of applying a flat 4x4 smoothing filter to the same image of a starfish that was used in the earlier lesson titled "Processing Image Pixels, Applying Image Convolution in Java" *(see Resources)*.

Figure 3. Application of a flat 4x4 smoothing filter.

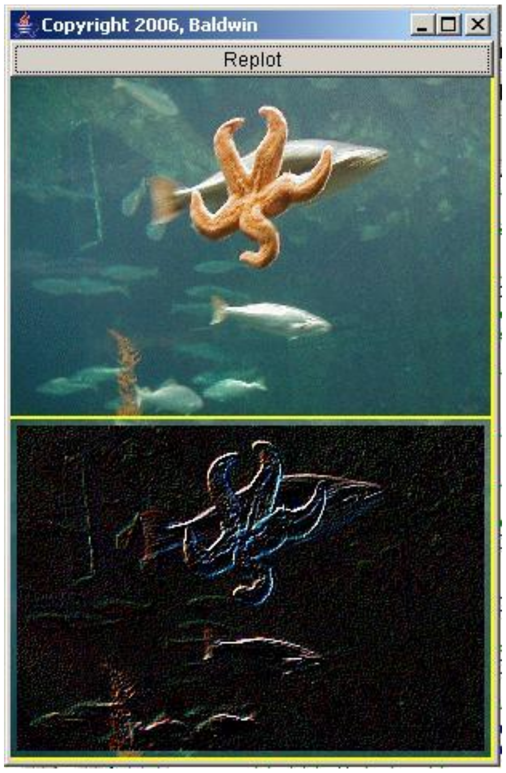


If you compare the output image in Figure 3 with Figure 18 in the earlier lesson, you will see good agreement. This is because none of the output color values fell outside the range from 0 to 255 inclusive, and it was not necessary to deal with the normalization issue that I will discuss later in the Assessment section.

An embossing filter

Figure 34 in the earlier lesson presented the results of a convolution filtering process that was intended to produce an output that looks like embossed stationary. Figure 4 shows the same process implemented using the **ConvolveOp** class.

Figure 4. Application of an embossing filter using ConvolveOp.



The differences are striking

The difference between the results shown in Figure 4 above and Figure 34 in the earlier lesson are striking. The difference lies solely in the *normalization* scheme used to deal with convolution output values that fall outside the allowable range from 0 through 255 inclusive.

One approach is to simply clip the values

Apparently the **ConvolveOp** class simply clips those values at 0 and 255. This is not a *safe* scheme because it throws away some of the output values replacing them by either 0 or 255.

A statistical scheme

The scheme used in the earlier lesson, on the other hand doesn't throw away any of the output values. Rather, the distribution of the output values is compressed, while maintaining its general shape, so as to cause all of the output values to fall within the allowable range. Thus, this is a *safe* scheme in that it doesn't throw away any information.

Don't clip the data
When we perform digital signal processing *(DSP)* in the world of signals, we usually try to avoid clipping the data. Clipping is a nonlinear, non-reversible process, which is usually considered to be a bad idea.

Another statistical scheme

Another image in the earlier lesson shows the results for the same input image and the same convolution filter with still another normalization scheme. This scheme causes the mean and the standard deviation of the output to match the mean and the standard deviation of the input. As a result, some of the output values may still fall outside the acceptable limits. In that case, those values are simply clipped at 0 and 255, making this scheme less *safe* than the one described above, but probably more safe than simply clipping the output values at 0 and 255.

Assessment

Earlier I indicated that in my opinion, the **ConvolveOp** class is the weakest of the image-filtering classes in the Java 2D API. Now I will explain my reasons for that opinion.

Output normalization is the real issue

In the earlier lesson titled "Processing Image Pixels, Applying Image Convolution in Java" *(see Resources)*, I explained that image convolution results can, and frequently do result in color values that fall outside the allowable color value range from 0 through 255 inclusive. Thus, the real issue in image convolution is not how to do the convolution arithmetic for a given convolution filter. The arithmetic algorithm for image convolution is almost trivial. The real issue for the serious image processor has to do with how you normalize the output values to:

- Force them into the range from 0 through 255,
- Discard those that are outside the range from 0 through 255, or
- Some combination of the two

Apparently the ConvolveOp class simply clips the output

Although the documentation for the **ConvolveOp** class doesn't indicate how the results are normalized, observation of the results suggests that values outside the acceptable range are simply clipped to values of 0 and 255.

Not necessarily the best approach

While that is the easiest approach to implement, it probably isn't the best approach from an image processing viewpoint. In fact, there probably isn't any one best approach. The normalization scheme that works best for one situation is likely to give way to a different normalization scheme for another situation. Unfortunately, simply clipping the output at 0 and 255 isn't likely to be the best approach for very many situations.

Two normalization schemes were described earlier

In the earlier lesson titled "Processing Image Pixels, Applying Image Convolution in Java" *(see Resources)*, I described two different normalization schemes that are based on the statistical distribution of the input and output color values. I showed that the choice of normalization scheme can have a dramatic effect on the visual results.

Would like optional normalization schemes in the ConvolveOp class

I would like to have seen a choice among several such normalization schemes provided by the **ConvolveOp** class instead of simply clipping the results to values of 0 and 255. In my opinion, simply clipping the results seriously limits the value of the **ConvolveOp** class for the serious image processor. Be that as it may, the remaining sections of this tutorial lesson will teach you how to use the **ConvolveOp** class for performing image convolution.

Preview

In this lesson, I will present and explain a Java class named **ImgMod42**. *(A complete listing of this class is shown in Listing 18 near the end of the lesson.)* The purpose of this class is to illustrate the use of the **ConvolveOp** filter class of the Java 2D API.

General comments
General comments regarding the uses of classes such as this one can be found in the class named **ImgMod038** in the lesson named "Using the Java 2D LookupOp Filter Class to Process Images" *(see Resources).*

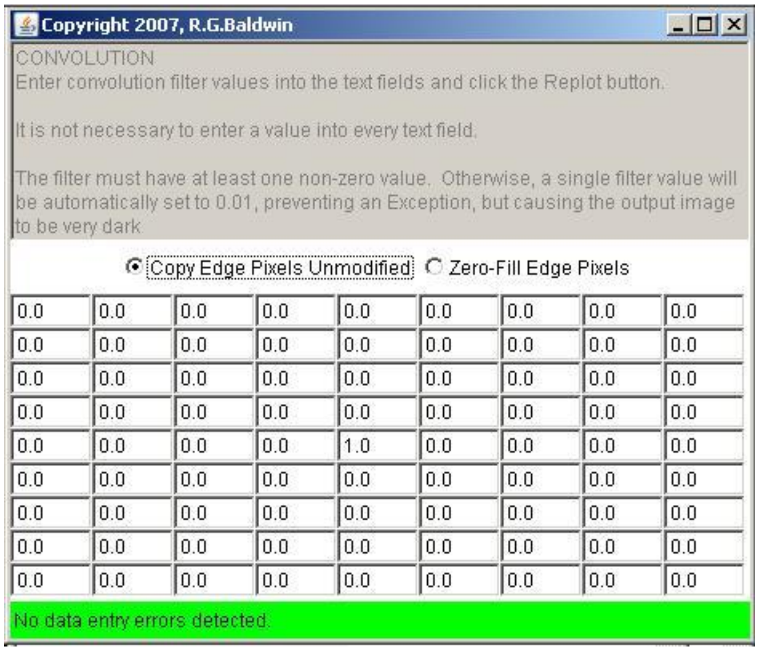
Compatible with ImgMod05

The class named **ImgMod42** is compatible with the use of the driver program named **ImgMod05**. The driver program named **ImgMod05** displays the original and the modified image as shown in Figure 6. It also writes the modified image into an output file in JPEG format. The name of the output file is **junk.jpg** and it is written into the current directory.

A user input GUI

Image processing programs such as this one may provide a GUI for data input making it possible for the user to modify the behavior of the image processing method each time the **Replot** button shown in Figure 6 is clicked. Such a GUI is provided for this program. Figure 5 shows a screen shot of the user input GUI.

Figure 5. Screen shot of the user input GUI.



CONVOLUTION
Enter convolution filter values into the text fields and click the Replot button.

It is not necessary to enter a value into every text field.

The filter must have at least one non-zero value. Otherwise, a single filter value will be automatically set to 0.01, preventing an Exception, but causing the output image to be very dark

[✓ Copy Edge Pixels Unmodified] Zero-Fill Edge Pixels

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

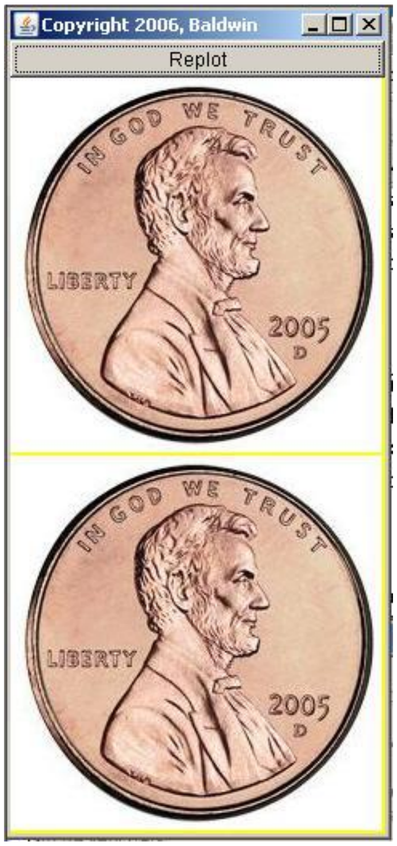
No data entry errors detected.

I will have more to say about the user input GUI later.

Typical program output

Figure 6 shows a screen shot of the program output produced by the input GUI values shown in Figure 5 in conjunction with an image file named **penny02.jpg**.

Figure 6. Screen shot of typical program output.



Usage information

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

```
java ImgMod05 ImgMod42 ImageFileName
```

If the program is unable to load the image file within ten seconds, it will abort with an error message.

The user input GUI

As can be seen in Figure 5, the program creates a GUI for user input containing:

- User instructions
- Two radio buttons, which are used to select the treatment given to the edge of the image during the convolution process.
- Eighty-one individual text fields, which allow the user to specify the values of a 9x9 2Dconvolution filter.

The 2D convolution filter

The program convolves a user-specified image with a user-specified 2D convolution filter having up to 81 filter coefficients, nine along each dimension.

Figure 5 shows the user input values for such a two-dimensional convolution filter, with all but one of the filter coefficient values being 0.0. The remaining filter coefficient is shown at the center of the matrix in Figure 5 with a value of 1.0.

The user-specified 9x9 convolution filter is used to filter the image each time the **Replot** button shown in Figure 6 is clicked.

Filter initialization

The convolution filter values are initialized at startup to a set of values that will simply pass the input image through to the output without modification. To accomplish this, the filter is initialized at startup with a value of 1.0 at the center and values of 0.0 in all other locations. *(This is the filter shown in Figure 5.)* This filter simply reproduces the input image in the output as shown in Figure 6. *(The image format shown in Figure 6 has the input image at the top and the output image at the bottom.)*

User input filter values

The user specifies the convolution filter coefficient values by changing the individual filter values in the 9x9 matrix shown in Figure 5. It is not necessary for the user to change all 81 filter values. Although they do consume some computing resources, filter coefficients having a value of 0.0 contribute nothing to the output image.

What happens when convolution values are out of the allowable range?

As mentioned earlier, it is unclear in the documentation what happens to the color values for an output image pixel if the value resulting from the convolution process falls outside the range from 0 to 255. However, observation of the results suggests that those values are simply clipped at 0 and 255.

This program was tested using J2SE 6 under WinXP.

Discussion and sample code

Will discuss in fragments

A complete listing of this class is presented in Listing 18. As is my custom, I will present and explain this class in fragments.

Beginning of the class definition

The class definition begins in the first program fragment shown in Listing 1. Note that it is necessary for this class to implement the interface named **ImgIntfc05** in order to be compatible with the driver program named **ImgMod05**.

Listing 1. Beginning of the class definition.

```
class ImgMod42 extends Frame implements
ImgIntfc05{

  Panel mainPanel = new Panel();//main control
panel

  CheckboxGroup radioButtonGroup = new
CheckboxGroup();
  Checkbox edgeNoOp = new Checkbox(
     "Copy Edge Pixels
Unmodified",radioButtonGroup,true);
  Checkbox edgeZeroFill = new Checkbox(
          "Zero-Fill Edge
Pixels",radioButtonGroup,false);
```

Listing 1 begins the instantiation of components that are used to construct the main control panel portion of the user input GUI. Components that require local access only are defined locally. Others are defined as instance variables.

In addition to instantiating the main control panel object, Listing 1 instantiates a pair of radio buttons that are used to specify the treatment of the pixels at the edge of the image. These two radio buttons are shown near the center of Figure 5.

An array of TextField objects

Listing 2 instantiates an array of **TextField** objects that are used for specifying matrix/convolution filter values.

Listing 2. The 9x9 convolution filter matrix.

```
  TextField[][] matrixField = new
TextField[9][9];
```

This 9x9 matrix of **TextField** objects is visible immediately below the radio buttons in Figure 5.

A Label object for error notification

Listing 3 instantiates and populates a **Label** object that is used to notify of data entry errors. This label is shown in the green area at the bottom of Figure 5. In the event of a data entry error, this area turns red and displays the message "Bad input data for the convolution filter."

Listing 3. Instantiation of Label object for display of data entry errors.

```
  String okMessage = "No data entry errors 
detected.";
  Label errorMsg = new Label(okMessage);
```

The primary constructor

Listing 4 shows the primary constructor. This constructor calls another method to construct the main control panel so as to separate the construction of the GUI into easily understandable units.

Listing 4. The primary constructor.

```
  ImgMod42(){//constructor

    constructMainPanel();
    add(mainPanel);

    setTitle("Copyright 2007, R.G.Baldwin");
    setBounds(555,0,470,400);
    setVisible(true);

    //Define a WindowListener to terminate the 
program.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent 
e){
          System.exit(1);
        }//end windowClosing
      }//end windowAdapter
    );//end addWindowListener
  }//end constructor
```

The code in Listing 4 is straightforward and shouldn't require further explanation.

Constructing the main control panel

Listing 5 begins the definition of the method named **constructMainPanel**. This method constructs the main control panel containing all of the controls in the user input GUI. This method is called from the primary constructor shown in Listing 4.

Listing 5. Beginning of the method named constructMainPanel.

```
  void constructMainPanel(){
    mainPanel.setLayout(new BorderLayout());

    String text ="CONVOLUTION\n"
      + "Enter convolution filter values into 
the text "
```

```
      + "fields and click the Replot 
button.\n\n"
      + "It is not necessary to enter a value 
into every "
      + "text field.\n\n"
      + "The filter must have at least one 
non-zero "
      + "value.  Otherwise, a single filter 
value will "
      + "be automatically set to 0.01, 
preventing an "
      + "Exception, but causing the output 
image to be "
      + "very dark";

    TextArea textArea = new TextArea(text,7,1,

TextArea.SCROLLBARS_NONE);

mainPanel.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);
```

After setting the layout manager, Listing 5 creates and adds the instructional text to a **TextArea** object in the panel. This text appears in a disabled text area at the top of the panel as shown in Figure 5.

Note that the number of columns specified for the **TextArea** in Listing 5 is immaterial because the **TextArea** object is placed in the **NORTH** location of a **BorderLayout**. Therefore, it will always fill the available horizontal space.

Create panel for radio buttons and text fields

Listing 6 creates a panel that contains the radio buttons and the text fields into which the user enters convolution coefficient values.

Listing 6. Creation of the panel for radio buttons and text fields.

```
    Panel controlPanel = new Panel();
    controlPanel.setLayout(new 
BorderLayout());
```

Create and populate a sub-panel for the radio buttons

Listing 7 creates and populates a sub-panel that contains the radio buttons. It is populated with check boxes that behave like radio buttons.

Listing 7. Creation and population of a sub-panel for the radio buttons.

```
    Panel radioButtonPanel = new Panel();
    radioButtonPanel.add(edgeNoOp);
    radioButtonPanel.add(edgeZeroFill);
```

Create the sub-panel that contains the text fields

Listing 8 creates a sub-panel that will contain the text fields into which the user enters convolution filter coefficient values.

Listing 8. Creation of a sub-panel that contains the text fields.

```
    Panel textFieldPanel = new Panel();
    textFieldPanel.setLayout(new
GridLayout(9,9));
```

Populate and initialize the array of TextField objects

Listing 9 populates and initializes the array of **TextField** objects for the convolution filter coefficients, with all zero values and adds them to their panel. **matrixField** is a reference to a 9x9 array object tree populated with **TextField** objects. **textFieldPanel** is a **Panel** object with a 9x9 grid layout.

Listing 9. Population and initialization of the array of TextField objects.

```
    for(int row = 0;row <
matrixField.length;row++){
      for(int col = 0;col <
matrixField[0].length;col++){
        matrixField[row][col] = new
TextField("0.0",6);

textFieldPanel.add(matrixField[row][col]);
      }//end inner loop
    }//end outer loop

    matrixField[4][4].setText("1.0");
```

Finally, Listing 9 initializes one **TextField** object with a value of 1.0 *(as shown in Figure 5)* so as to create a convolution filter that simply copies the input image to the output.

Complete construction of the user input GUI

Listing 10 completes the construction of the user input GUI.

Listing 10. Complete the construction of the user input GUI.

```
    //Populate the control panel.

controlPanel.add(radioButtonPanel,BorderLayout.NORTH);
controlPanel.add(textFieldPanel,BorderLayout.CENTER);

    //Finish populating the main panel.
    mainPanel.add(controlPanel,BorderLayout.CENTER);

    //Add the errorMsg label.
    mainPanel.add(errorMsg,BorderLayout.SOUTH);
    errorMsg.setBackground(Color.GREEN);
  }//end constructMainPanel
```

All of the code in Listing 10 is straightforward and shouldn't require further explanation.

The processMainPanel method

Listing 11 shows the beginning of the **processMainPanel** method. This method processes the image according to the filter values provided by the user. This method uses the **ConvolveOp** filter class to process the image. The **processMainPanel** method is called from within the method named **processImg**, which is the primary image processing method in this program. The method named **processImg** is called by the driver program named **ImgMod05**.

Listing 11. Beginning of the processMainPanel method.

```
  BufferedImage processMainPanel(BufferedImage
theImage){

    //Reset the error message to the default.
    errorMsg.setText(okMessage);
    errorMsg.setBackground(Color.GREEN);
```

Convert the 2D convolution filter into a 1D array

Listing 12 converts the matrix/convolution filter into a one-dimensional array. This is the format required for input to the constructor for the **Kernel** class, which is used to pass the convolution coefficients into the filtering process.

Listing 12 tests for all zero values and values that can't be converted to numeric format in the process of converting to a one-dimensional array. If either condition is detected, Listing 12 creates a set of convolution coefficients having a single very small value.

Listing 12. Conversion of the 2D convolution filter into a 1D array.

```
    boolean zeroTest = true;
    float[] matrix = new float[
              matrixField.length *
matrixField[0].length];
    int matrixCnt = 0;
    try{
      for(int row = 0;row <
matrixField.length;row++){
        for(int col = 0;col <
matrixField[0].length;

col++,matrixCnt++){
          matrix[matrixCnt] =
Float.parseFloat(

matrixField[row][col].getText());
          if(matrix[matrixCnt] != 0.0f){
            zeroTest = false;
          }//end if
        }//end col loop
      }//end row loop
    }catch(java.lang.NumberFormatException e){
      //Bad input data for the convolution
filter. Cause
      // the output image to be very dark so
that it will
      // be obvious to the user that there is
a problem.
      // Also cause the label containing the
error message
      // to turn from green to red.
      matrixCnt = 0;
      for(int row = 0;row <
matrixField.length;row++){
        for(int col = 0;col <
matrixField[0].length;

col++,matrixCnt++){
          matrix[matrixCnt] = 0.0f;
        }//end col loop
      }//end row loop
      //Set one filter coefficient to a non-
zero, very
      // small value.
      matrix[0] = 0.01f;
      errorMsg.setText(
            "Bad input data for the
convolution filter.");
      errorMsg.setBackground(Color.RED);
    }//end catch
```

Although the code in Listing 12 is somewhat complicated, there is nothing in Listing 12 that is new to this lesson, so a detailed explanation of Listing 12 shouldn't be necessary

Correct for all filter values equal to 0.0f

If all of the filter values are 0.0f, Listing 13 causes one value to be a very small non-zero value and displays an error message. This will prevent the program from throwing an exception when it tries to perform a convolution operation using a convolution filter having all zero values.

Listing 13. Correct for all filter values equal to 0.0f.

```
    if(zeroTest){
      matrix[0] = 0.1f;
      errorMsg.setText(
            "Bad input data for the
convolution filter.");
      errorMsg.setBackground(Color.RED);
    }//end if
```

Set the edge treatment parameters

Listing 14 uses the state of the radio buttons to establish the manner in which the edge of the image will be treated during the convolution process.

Listing 14. Setting the edge treatment parameters.

```
    int edgeTreatment;
    if(edgeZeroFill.getState() == true){
      edgeTreatment =
ConvolveOp.EDGE_ZERO_FILL;
    }else{//edgeNoOp must have been selected
      edgeTreatment = ConvolveOp.EDGE_NO_OP;
    }//end else
```

Create the ConvolveOp filter

Listing 15 begins by creating the required **ConvolveOp** filter object that will be used to convolve the image with the filter. It creates the required **Kernel** object as an anonymous object in the parameter list for the **ConvolveOp** constructor. The third parameter to the constructor with a value of null is for "rendering hints," which are not used by this program, but which have been briefly discussed in earlier lessons.

Listing 15. Creation of the ConvolveOp filter.

```
    ConvolveOp filterObj = new ConvolveOp(
                         new
Kernel(matrixField.length,

matrixField[0].length,
                                    matrix),
                         edgeTreatment,
                         null);
```

Apply the filter and return the filtered image

It all comes down to this. After the **ConvolveOp** filter object has been created, Listing 16 calls the **filter** method on the **ConvolveOp** filter object to apply the convolution filter to the image. All of the code discussed prior to this point was required just to get ready to perform the convolution.

Listing 16. Apply the ConvolveOp filter and return the filtered image.

```
    return filterObj.filter(theImage,null);

  }//end processMainPanel
```

The kernel was created using the filter coefficient values provided by the user. The **filter** method convolves the image with the kernel, returning the filtered image as type **BufferedImage**. This filtered image is immediately returned by the **processMainPanel** method.

The second parameter to the **filter** method with a value of null allows the filtered image to be deposited in a separate destination **BufferedImage** object. This feature was not used in this program.

The required processImg method

The **processImg** method shown in Listing 17 must be defined to implement the **ImgIntfc05** interface. This method is called by the driver program named **ImgMod05**. This method has been discussed in numerous earlier lessons in this series and shouldn't require further discussion in this lesson.

Listing 17. The required processImg method.

```
  public BufferedImage 
processImg(BufferedImage theImage){
    BufferedImage outputImage = 

processMainPanel(theImage);
    return outputImage;
  }//end processImg
}//end class ImgMod42
```

And that is the end of the class named **ImgMod42** and the end of the program.

Run the program

I encourage you to copy the code from Listing 18 into your text editor, compile it, and execute it. Experiment with it, making changes, and observing the results of your changes.

Keep in mind that you will also need to compile and use the program named **ImgMod05**. You will find the source code for **ImgMod05** in the earlier lesson titled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see Resources)*.

My recommendation is that you:

- Go back and review some of my earlier lessons on image convolution that did not use the **ConvolveOp** class to perform the convolution.

- Use the convolution filters that I explained in those lessons in conjunction with this program in an attempt to replicate the results produced by those programs.
- See if you can understand why the different approaches produce similar results when they do, and why they don't provide similar results when they don't.

Summary

In this lesson, I provided and explained an image-processing class named **ImgMod42** that is compatible with the framework program named **ImgMod05**.

The purpose of this class is to teach you how to write programs that use the **ConvolveOp** class of the Java 2D API to perform two-dimensional image convolution.

I also explained some of the weaknesses of the **ConvolveOp** class that result from a lack of options for dealing with convolution output values greater than 255 and less than 0.

What's next?

Future lessons in this series will teach you how to use the following image-filtering classes from the Java 2D API:

- **RescaleOp**
- **ColorConvertOp**

Complete program listing

A complete listing of the class discussed in this lesson is shown in Listing 18 below.

Listing 18. Complete listing for the class named ImgMod42.

```
/*File ImgMod42.java
Copyright 2007, R.G.Baldwin

The purpose of this class is to illustrate the use of the
ConvolveOp filter class of the Java 2D API.

See general comments in the class named ImgMod038 in the
lesson named "Using the Java 2D LookupOp Filter Class to
Process Images".

This class is compatible with the use of the driver
program named ImgMod05.

The driver program named ImgMod05 displays the original
and the modified image.  It also writes the modified image
into an output file in JPEG format.  The name of the
output file is junk.jpg and it is written into the current
```

```
directory.

Image processing programs such as this one may provide a
GUI for data input making it possible for the user to
modify the behavior of the image processing method each
time the Replot button is clicked.  Such a GUI is provided
for this program.

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

java ImgMod05 ImgMod42 ImageFileName

If the program is unable to load the image file within ten
seconds, it will abort with an error message.

This program convolves a user-specified image with a
user-specified two-dimensional convolution filter having
up to 81 filter coefficients, nine along each dimension.

A 9x9 convolution filter is used to filter the image each
time the Replot button is clicked.  The filter is
initialized with a value of 1.0 at the center and values
of 0.0 in all other locations.  The user specifies the
convolution filter coefficient values by changing the
individual filter values. It is not necessary for the user
to change all 81 filter values.  Filter coefficients
having a value of 0.0 contribute nothing to the output.

The program creates a GUI for user input containing:
User instructions
Two radio buttons used to select the type of treatment
 given to the edge of the image during the convolution
 process.
Text fields used to specify the values of a 9x9
 convolution filter.

The convolution filter values are initialized so as to
simply pass the input image through to the output
without modification.

It is unclear in the documentation what happens to the
color value if the value resulting from the convolution
process falls outside the range from 0 to 255.  However,
observation of the results suggests that those values are
clipped at 0 and 255.

Tested using J2SE 6 under WinXP.
*********************************************************/

import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ImgMod42 extends Frame implements ImgIntfc05{
```

```
//Components used to construct the main panel.
// Components that require local access only are defined
// locally.  Others are defined here as instance
// variables.
Panel mainPanel = new Panel();//main control panel

//A pair of radio buttons used to specify the treatment
// of the pixels at the edge of the image.
CheckboxGroup radioButtonGroup = new CheckboxGroup();
Checkbox edgeNoOp = new Checkbox(
   "Copy Edge Pixels Unmodified",radioButtonGroup,true);
Checkbox edgeZeroFill = new Checkbox(
        "Zero-Fill Edge Pixels",radioButtonGroup,false);

//An array of TextField objects for specifying
// matrix/convolution filter values.  Normally, I would
// refer to the data entered into these TextField
// objects simply as the filter.  However, in order to
// distinguish between the convolution filter, and the
// filter method of the ConvolveOp class, in most cases,
// I will refer to the data as a matrix.
TextField[][] matrixField = new TextField[9][9];

//The following Label is used to notify of data entry
// errors.
String okMessage = "No data entry errors detected.";
Label errorMsg = new Label(okMessage);
//-----------------------------------------------------//

//This is the primary constructor.  It calls another
// method to construct the main panel so as to separate
// the construction of the GUI into easily
// understandable units.
ImgMod42(){//constructor

  constructMainPanel();
  add(mainPanel);

  setTitle("Copyright 2007, R.G.Baldwin");
  setBounds(555,0,470,400);
  setVisible(true);

  //Define a WindowListener to terminate the program.
  addWindowListener(
    new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(1);
      }//end windowClosing
    }//end windowAdapter
  );//end addWindowListener
}//end constructor
//-----------------------------------------------------//

//This method constructs the main panel containing all
// of the controls.  This method is called from the
// primary constructor.
void constructMainPanel(){
  mainPanel.setLayout(new BorderLayout());

  //Create and add the instructional text to the panel.
  // This text appears in a disabled text area at the
  // top of the panel.
  String text ="CONVOLUTION\n"
    + "Enter convolution filter values into the text "
    + "fields and click the Replot button.\n\n"
    + "It is not necessary to enter a value into every "
    + "text field.\n\n"
    + "The filter must have at least one non-zero "
    + "value.  Otherwise, a single filter value will "
    + "be automatically set to 0.01, preventing an "
    + "Exception, but causing the output image to be "
    + "very dark";

  //Note:  The number of columns specified for the
  // following TextArea is immaterial because the
  // TextArea object is placed in the NORTH location of
  // a BorderLayout.  Therefore, it will always fill the
  // available horizontal space.
  TextArea textArea = new TextArea(text,7,1,
                            TextArea.SCROLLBARS_NONE);
  mainPanel.add(textArea,BorderLayout.NORTH);
  textArea.setEnabled(false);

  //Create the panel that contains the radio buttons
  // and the text fields into which the user enters
  // convolution coefficient values.
  Panel controlPanel = new Panel();
  controlPanel.setLayout(new BorderLayout());

  //Create and populate the sub-panel that contains
  // the radio buttons. Populate it with check boxes
  // that behave like radio buttons.
  Panel radioButtonPanel = new Panel();
  radioButtonPanel.add(edgeNoOp);
  radioButtonPanel.add(edgeZeroFill);

  //Create the sub-panel that contains the text fields.
  Panel textFieldPanel = new Panel();
  textFieldPanel.setLayout(new GridLayout(9,9));

  //Populate and initialize the array of TextField
  // objects for the convolution filter, with all zero
  // values and add them to their panel. matrixField
  // is a reference to a 9x9 array object tree
  // populated with TextField objects. textFieldPanel
  // is a Panel object with a 9x9 grid layout.
  for(int row = 0;row < matrixField.length;row++){
    for(int col = 0;col < matrixField[0].length;col++){
      matrixField[row][col] = new TextField("0.0",6);
      textFieldPanel.add(matrixField[row][col]);
    }//end inner loop
  }//end outer loop

  //Initialize one TextField object so as to create a
  // filter that simply copies the input image to the
  // output.
  matrixField[4][4].setText("1.0");

  //Populate the control panel.
  controlPanel.add(radioButtonPanel,BorderLayout.NORTH);
  controlPanel.add(textFieldPanel,BorderLayout.CENTER);

  //Finish populating the main panel.
  mainPanel.add(controlPanel,BorderLayout.CENTER);

  //Add the errorMsg label.
  mainPanel.add(errorMsg,BorderLayout.SOUTH);
  errorMsg.setBackground(Color.GREEN);
}//end constructMainPanel
//----------------------------------------------------//

//This method processes the image according to the
// filter values provided by the user.
//The method uses the ConvolveOp filter class to
// process the image.  The method is called from within
// the method named processImg, which is the primary
// image processing method in this program.
BufferedImage processMainPanel(BufferedImage theImage){

  //Reset the error message to the default.
  errorMsg.setText(okMessage);
  errorMsg.setBackground(Color.GREEN);

  //Now convert the matrix/convolution filter into a
  // one-dimensional array. This is the required format
  // for input to the constructor for the Kernel class,
  // which is  used to pass the convolution
  // coefficients into the filtering process.
  //Test for all zero values and values that can't be
  // converted to numeric format in the process of
  // converting to a one-dimensional array. If either
  // condition is detected, create a set of
  // convolution coefficients having a single very small
  // value.
  boolean zeroTest = true;
  float[] matrix = new float[
            matrixField.length * matrixField[0].length];
  int matrixCnt = 0;
  try{
    for(int row = 0;row < matrixField.length;row++){
      for(int col = 0;col < matrixField[0].length;
                                     col++,matrixCnt++){
        matrix[matrixCnt] = Float.parseFloat(
                       matrixField[row][col].getText());
        if(matrix[matrixCnt] != 0.0f){
          zeroTest = false;
        }//end if
      }//end col loop
    }//end row loop
  }catch(java.lang.NumberFormatException e){
    //Bad input data for the convolution filter. Cause
    // the output image to be very dark so that it will
    // be obvious to the user that there is a problem.
    // Also cause the label containing the error message
    // to turn from green to red.
    matrixCnt = 0;
    for(int row = 0;row < matrixField.length;row++){
      for(int col = 0;col < matrixField[0].length;
                                     col++,matrixCnt++){
        matrix[matrixCnt] = 0.0f;
      }//end col loop
    }//end row loop
    //Set one filter coefficient to a non-zero, very
    // small value.
    matrix[0] = 0.01f;
    errorMsg.setText(
          "Bad input data for the convolution filter.");
    errorMsg.setBackground(Color.RED);
  }//end catch

  //If all filter values are 0.0f, cause one value to
  // be a very small non-zero value and display an
  // error message. This will prevent the program from
  // throwing an exception when it tries to perform a
  // convolution operation using a convolution filter
  // having all zero values.
  if(zeroTest){
    matrix[0] = 0.1f;
    errorMsg.setText(
          "Bad input data for the convolution filter.");
    errorMsg.setBackground(Color.RED);
  }//end if

  //Use the state of the radio buttons to set the manner
  // in which the edge of the image will be treated
  // during the convolution process.
  int edgeTreatment;
  if(edgeZeroFill.getState() == true){
    edgeTreatment = ConvolveOp.EDGE_ZERO_FILL;
  }else{//edgeNoOp must have been selected
    edgeTreatment = ConvolveOp.EDGE_NO_OP;
  }//end else

  //Create the filter object. Create the Kernel object
  // as an anonymous object in the parameter list for
  // the ConvolveOp constructor.
  ConvolveOp filterObj = new ConvolveOp(
                       new Kernel(matrixField.length,
                                  matrixField[0].length,
                                  matrix),
                       edgeTreatment,
                       null);
  //Apply the filter and return the filtered image
  return filterObj.filter(theImage,null);

}//end processMainPanel
//-----------------------------------------------------//

//The following method must be defined to implement the
// ImgIntfc05 interface.  It is called by the driver
// program named ImgMod05.
public BufferedImage processImg(BufferedImage theImage){
  BufferedImage outputImage =
                         processMainPanel(theImage);
  return outputImage;
}//end processImg
}//end class ImgMod42
```

Copyright

Resources

- 400 Processing Image Pixels using Java, Getting Started
- 402 Processing Image Pixels using Java, Creating a Spotlight
- 404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
- 406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
- 408 Processing Image Pixels, Performing Convolution on Images
- 410 Processing Image Pixels, Understanding Image Convolution in Java
- 412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
- 414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
- 416 Processing Image Pixels, An Improved Image-Processing Framework in Java
- 450 A Framework for Experimenting with Java 2D Image-Processing Filters
- 452 Using the Java 2D LookupOp Filter Class to Process Images
- 454 Using the Java 2D AffineTransformOp Filter Class to Process Images
- 456 Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images
- 458 Using the Java 2D BandCombineOp Filter Class to Process Images

About the author

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

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

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

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

Baldwin@DickBaldwin.com

Keywords
java 2D image pixel framework filter ConvolveOp

-end-