Understanding the 2D Discrete Cosine Transform in Java, Part 2

Learn how to sub-divide an image before applying a forward and inverse 2D-Discrete Cosine Transform similar to the way it is done in the JPEG image compression algorithm. Also learn some of the theory behind and some of the reasons for sub-dividing images, such as improved speed.

Published: October 3, 2006
By Richard G. Baldwin

Java Programming Notes # 2448

Preface

This lesson is part of a series designed to teach you about the inner workings of data and image compression. The first lesson in the series was lesson number 2440 entitled Understanding the Lempel-Ziv Data Compression Algorithm in Java.

An earlier lesson entitled Understanding the Discrete Cosine Transform in Java taught you about the one-dimensional Discrete Cosine Transform. The previous lesson, Part 1 of this two-part lesson, deals with the two-dimensional Discrete Cosine Transforms *(2D-DCT)*.

JPEG image compression

One of the objectives of this series is to teach you about the inner workings of JPEG image compression. According to Wikipedia,

*"... **JPEG** ... is a commonly used standard method of lossy compression for photographic images. ... JPEG/JFIF is the most common format used for storing and transmitting photographs on the World Wide Web."*

Central components

One of the central components of JPEG image compression is entropy encoding. Huffman encoding, which was the primary topic of the earlier lesson entitled Understanding the Huffman Data Compression Algorithm in Java, is a common form of entropy encoding.

Another central component of JPEG compression is the two-dimensional Discrete Cosine Transform *(2D-DCT)*. This is the primary topic of this lesson. In Part 1 of this lesson, I taught you how to use the *forward* 2D-DCT to compute and display the frequency spectrum of an image. I also taught you how to apply the *inverse* 2D-DCT to the spectral data to reconstruct a replica of the original image.

In this part of the lesson, I will teach you how to sub-divide the image before applying the 2D-DCT in a manner similar to the way it is done in the JPEG image compression algorithm. I will also discuss some of the theory behind and some of the reasons for such sub division

A third central component of the JPEG image compression algorithm is selective spectral re-quantization. This will be the primary topic of a future lesson.

In order to understand JPEG ...

In order to understand JPEG image compression, you must understand Huffman encoding, the 2D-DCT, image sub-division, selective spectral re-quantization, and perhaps some other topics as well. I plan to teach you about the different components of JPEG in separate lessons, and then to provide a lesson that teaches you how they work together to produce "*the most common format used for storing and transmitting photographs on the World Wide Web.*"

Viewing tip

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

Supplementary material

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

In preparation for understanding the material in this lesson, I recommend that you also study the lessons referred to in the References section.

General Background Information

In earlier lessons, I have explained that the JPEG image compression algorithm transforms the original image into the frequency domain using a 2D-DCT. The primary purpose of the transformation is to transform the information into a form that is more amenable to compression than is the form of the original image.

In Part 1 of this lesson, I explained that rather than to perform the 2D-DCT on the entire image, the JPEG image compression algorithm sub-divides the image into 8x8 blocks of pixels and performs the 2D-DCT on each block independently. However, I didn't tell you why that is done.

I can't tell you all of the reasons that the members of the Joint Photographic Experts Group decided to do that, but I can tell you that one of the primary reasons probably had to do with speed. Sub-dividing the image into a set of smaller contiguous images and processing them independently can improve the processing speed significantly in two different ways:

- Reducing the total number of multiply-add operations *(MADs)* required to perform the transform.
- Eliminating the requirement to evaluate the cosine of many different arguments.

Reducing the total number of MADs

If an image is sub-divided into a contiguous set of smaller images and the 2D-DCT is performed on each of the smaller images independently of the others, a significant reduction in the total number of MADs can be achieved relative to the number of MADs required to transform the original image.

Reduction by the square root of the ratio

For example, if the dimensions of the larger image are an even multiple of the dimensions of the smaller image, the total number of MADs required to sub-divide and then transform is reduced by the square root of the ratio of the number of pixels in the large image to the number of pixels in the small image.

> *(If the dimensions of the large image are not an even multiple of the dimensions of small image, it is necessary to expand the dimensions of the large image to force it to be an even multiple. This can reduce the gain slightly, but the gain can still be significant.)*

A numeric example

For example, if a square image having 320 pixels on each side is sub-divided into 1600 small images having 8 pixels on each side, the total number of MADs required to transform the 1600

small images will be a factor of 40 less than the total number of MADs required to transform the large image.

> *(According to my calculations, 196,608,000 MADs are required to transform all three color planes for a square image that is 320 pixels on each side. By contrast, if that image is sub-divided into square images that are 8 pixels on each side, only 4,915,200 MADs are required to transform all of the small images. This reduces the computational effort by a factor of 40, which is very significant. Furthermore, the gain increases as the ratio of the size of the large image to the small image becomes even larger.)*

Eliminating the requirement to evaluate the cosine function

If you write a program to perform a 2D-DCT on an image of any arbitrary size, you must repetitively evaluate the cosine function millions of times for many different arguments. The actual arguments that must be used in the evaluation of the cosine function depend on the dimensions of the image.

On the other hand, if you write a program to perform a 2D-DCT on an image of a fixed size, you can determine in advance the arguments that must be used in the evaluation of the cosine function. This makes it possible to evaluate the cosine function for that set of arguments in advance, and to code the cosine values for that set of arguments directly into a lookup table in the program. This makes it possible to eliminate the requirement to repeatedly evaluate the cosine function, substituting a table-lookup operation instead.

The 8x8-pixel image size used by the JPEG image compression algorithm requires a cosine table containing only 64 values. The use of a table-lookup operation as an alternative to actually evaluating a cosine function should also provide a significant speed improvement.

> *(The programs that you will see later use table lookup when the size of the image is 8x8 and evaluates the cosine function for other image sizes.)*

Sub-dividing an image

Consider the pair of images shown in Figure 1.

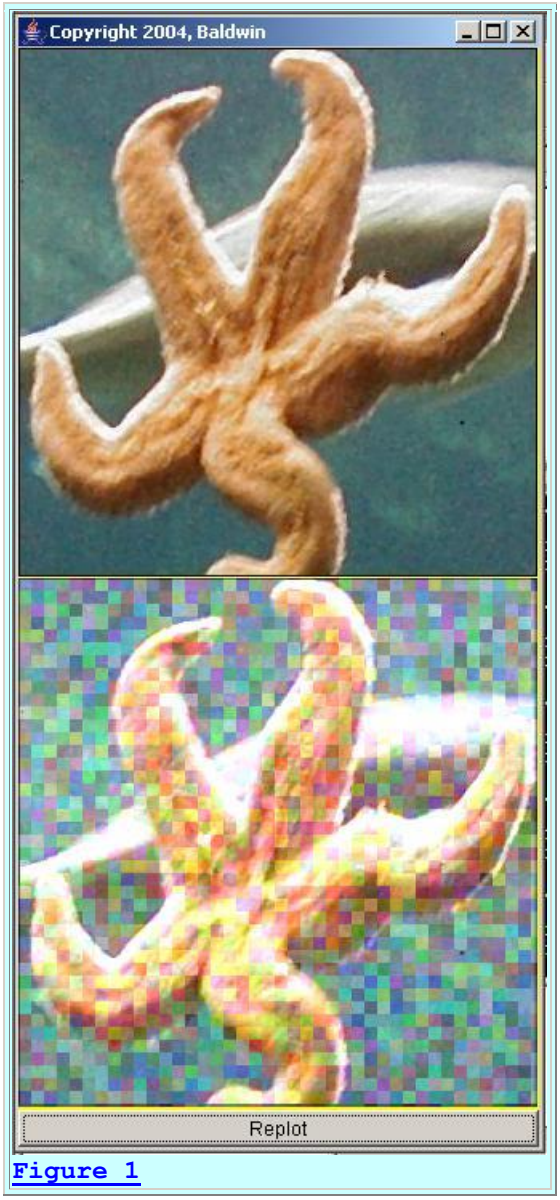

Figure 1

The image in the bottom panel of Figure 1 was originally the same as the image in the top panel. However, I sub-divided the image in the bottom panel into 8x8 blocks of pixels and purposely modified the color of every pixel in each block so as to cause the blocks to be visually distinguishable from one another. The idea was to enforce the concept that any image can be sub-divided into smaller images.

Disassembly and reassembly

For example, I could sub-divide a large image into blocks of pixels of a uniform size and send each block to you as an attachment to a separate email message. You could then reassemble the blocks into a replica of the original image provided you knew the correct location for each block.

The point is that it doesn't do any harm to sub-divide an image into blocks of pixels and then to reassemble them later into a replica of the original image. If it serves a useful purpose, that is a perfectly good operation.

Similar to JPEG compression algorithm

This is similar to part of what the JPEG image compression algorithm does. It sub-divides the original image into a set of 8x8-pixel blocks and transforms each block independently of all the other blocks. Then it modifies each block of spectral data using *selective re-quantization* to make it more suitable for compression. The transformed and modified spectral data is then compressed.

A *lossy* algorithm

Later, the data is uncompressed. An inverse transform is performed on each uncompressed spectral block, creating a *lossy* replica of the original image block. The replicas of each image block are then reassembled into a lossy replica of the original image. The lossy nature of the algorithm apparently occurs mainly at the stage where the spectral data blocks are modified through selective re-quantization to make them more suitable for compression.

Two different programs

I will present and explain two different programs in this lesson. The first program, named **ImgMod35A** makes it possible for a user to sub-divide an image into blocks of a user-specified size, and to perform a forward 2D-DCT on each block. The frequency spectra of the blocks are then normalized to make them suitable for being displayed as ordinary images, and the normalized spectra are displayed in an image format.

The bottom panel in Figure 2 shows the individual 8x8 normalized frequency spectra that represent the top image in Figure 2.

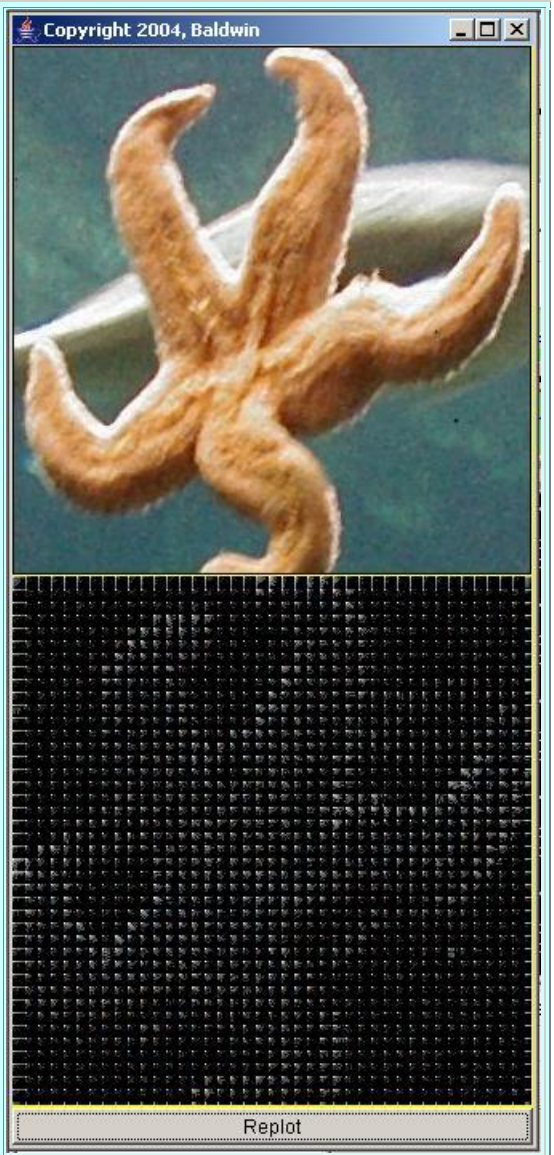

Figure 2

Figure 6 shows another example where the image was processed in square blocks where each block was twenty pixels on each side.

The second program

The second program, named **ImgMod35** subdivides an image into a set of 8x8-pixel blocks. It performs a forward 2D-DCT on each individual block producing a set of 8x8 spectral blocks. Then it performs an inverse 2D-DCT on each spectral block producing a replica of each of the original image blocks. The replicas are then reassembled into a replica of the original image, which is displayed in the format shown in Figure 3.

> *(An option is provided to mark the individual blocks as shown in Figure 1 so as to identify the individual blocks. This option requires the user to modify and*

recompile the source code for the program. This option obviously corrupts the image and should be used for illustration purposes only.)

Example output from ImgMod35

Figure 3 shows the result of running this program on the same image as shown in the top panels of Figure 1 and Figure 2 without exercising the option to mark the individual blocks.

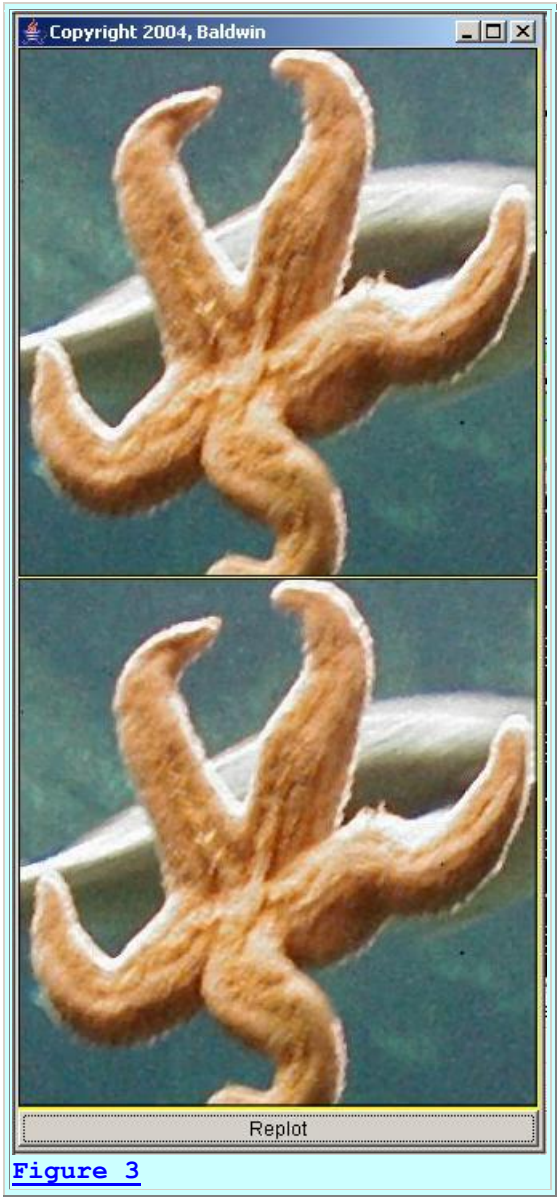

Figure 3

The image in the bottom panel of Figure 3 was produced by performing an inverse 2D-DCT on the 8x8 spectral blocks shown in the bottom panel of Figure 2 and reassembling the resulting 8x8-pixel image blocks into the full image. As you can see, the result is a very good replica of the original image shown in the top panel of Figure 3.

(Note, however, that the individual 8x8 spectral blocks were not modified through selective re-quantization as is the case with JPEG compression. Selective re-quantization of the spectral data will be the subject of a future lesson.)

Preview

ImgMod35a

The first program that I will present and explain in this lesson is named **ImgMod35a**. This program sub-divides an image into square blocks of pixels and performs a forward DCT on each block producing and displaying square blocks of spectral data. The size of the blocks is specified by entering the block size into the text field shown in Figure 4 and pressing the **Replot** button shown at the bottom of Figure 5.

Processing the color planes

Each color plane is individually processed. The program performs a forward 2D-DCT on each color plane converting each of the three color planes into spectral planes. The three spectral planes are normalized so as to make them suitable for being displayed as standard image data. The normalization includes transformation of the spectral data to log base 10 *(scaled decibels)* and scaling to cause the values to fall between 0 and 255 inclusive.

Optimized for 8x8 block size

This program can handle any block size from one pixel per block up to the size of the original image. However, it is optimized for block sizes of 8x8 pixels. When the block size is 8x8 pixels, the program uses an 8x8 cosine lookup table instead of computing the cosine value every time it is needed. This should cause the program to run faster than would otherwise be the case if it were necessary to compute the cosine value every time that it is needed.

Instructions for running the program

This program class is designed to be driven by the class named **ImgMod02a**. Enter the following at the command line to run this program:

```
java ImgMod02a ImgMod35a ImageFileName
```

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

When the user presses the **Replot** button, the process is repeated using the value in the text field of Figure 4 for the block size. The new results are displayed in the format shown in Figure 5.

Class files required

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

- ImgIntfc02.class
- ImgMod02a.class
- ImgMod35a.class
- ForwardDCT02.class

The source code for **ImgMod02a** and **ImgIntfc02** can be found in the earlier lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started.

The source code for **ImgMod35a** is provided in Listing 17. The source code for **ForwardDct02** is provided in Listing 18.

ImgMod35

The second program that I will present and explain in this lesson is named **ImgMod35**. This program performs a forward 2D-DCT on an image converting the three color planes into spectral planes. Then it performs an inverse 2D-DCT on the three spectral planes converting them back into image color planes. Then the original image and the reconstructed image are both displayed in the format shown in Figure 3.

No modification to the spectral data

Nothing is done to the spectral planes following the forward DCT and before the inverse DCT as is the case in JPEG compression. However, additional processing, such as re-quantization and compression, followed by decompression could be inserted at that point.

Update to an earlier program

This is an update to the program named **ImgMod34** that was explained in Part 1 of this lesson. This update sub-divides the image into blocks of 8x8-pixels and performs the forward 2D-DCT on each block producing 8x8 blocks of spectral coefficient data. Then it performs an inverse 2D-DCT on each of the 8x8 spectral blocks, producing replicas of the original the 8x8 pixel blocks.

A composite of all the reconstructed 8x8-pixel blocks is created to produce a replica of the original image.

Zero padding at the edges

Zero padding is applied to the right and bottom edges of the image to force each dimension of the image to be a multiple of 8 pixels. This padding is trimmed from the resulting composite image before the processed composite image is returned to the calling method. This process doesn't appear to have a detrimental impact on the quality of the image at the edges.

Block structure can be shown

For illustration purposes only, the method named **inverseXform8x8Plane** contains a statement that can be activated to cause the 8x8 block structure of the entire process to become visible in the final image as shown in Figure 1. This feature should be disabled by default because it badly corrupts the visual quality of the image when it is enabled.

Improved speed

This program is significantly faster than the program named **ImgMod034** that was presented and explained in Part 1 of this lesson. The program named **ImgMod034** does not sub-divide the image into 8x8-pixel blocks, but rather does the forward and inverse transforms on the entire image. This program named **ImgMod035** does sub-divide the image into 8x8-pixel blocks before performing the forward and inverse transforms. In addition, it uses an 8x8 cosine lookup table instead of computing the cosine value every time the cosine value is needed.

The use of a lookup table is probably one factor in the speed improvement. Another factor is the fact that less arithmetic is required to perform the transform when the image is sub-divided into blocks and the transforms are performed on the individual blocks instead of transforming the entire image as a whole.

Instructions for running the program

The class named **ImgMod035** is designed to be driven by the class named **ImgMod02a**. Enter the following at the command line to run this program:

```
java ImgMod02a ImgMod35 ImageFileName
```

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

When you click the **Replot** button shown in Figure 3, the process will be repeated and the results will be re-displayed. However, because there is no opportunity for user input after the program is started, the **Replot** button is of little value to this program.

Class files required

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

- ImgIntfc02.class
- ImgMod02a.class
- ImgMod35.class
- ForwardDCT02.class
- InverseDCT02.class

The source code for **ImgMod02a** and **ImgIntfc02** can be found in the earlier lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started.

The source code for **ImgMod35** is provided in Listing 19. The source code for **ForwardDct02** is provided in Listing 18, and the source code for **InverseDCT02** is provided in Listing 20.

Program testing

All of the programs discussed in this document were tested using J2SE 5.0 and WinXP.

Discussion

Experimental results for ImgMod35a

Before getting into the details of the code, I want to show you some experimental results. I will begin with experimental results for the program named **ImgMod35a**.

The display format

The program named **ImgMod35a** displays results in the format shown in Figure 2. The top panel in Figure 2 shows the original image. The bottom panel in Figure 2 shows the set of contiguous two-dimensional spectra produced by applying the 2D-DCT individually to contiguous 8x8-pixel blocks from the original image, *(as identified by the blocks in the bottom panel in Figure 1).*

The spectral origin

The value of the spectrum at a frequency of zero appears at the upper-left corner of each two-dimensional spectral block with higher-frequency components appearing to the right and down from that corner. The energy at a frequency of zero is often the highest point in the spectrum *(represented by a bright color in the display)*. Therefore, most of the bright dots in Figure 2 correspond to the upper-left corners of individual 2D-DCT spectral blocks.

The user interface

Figure 4 shows the user interface panel that is produced by the program named **ImgMod35a**.

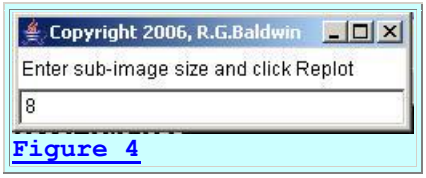

Figure 4

The initial block size

When the program first starts running, the text field shown in Figure 4 is initialized with a value of 8. This causes the program to sub-divide the image into 8x8-pixel blocks *(as indicated by the bottom panel in Figure 1)* and to compute and display the 2D-DCT for each of the 8x8-pixel blocks as shown in Figure 2 and Figure 5.

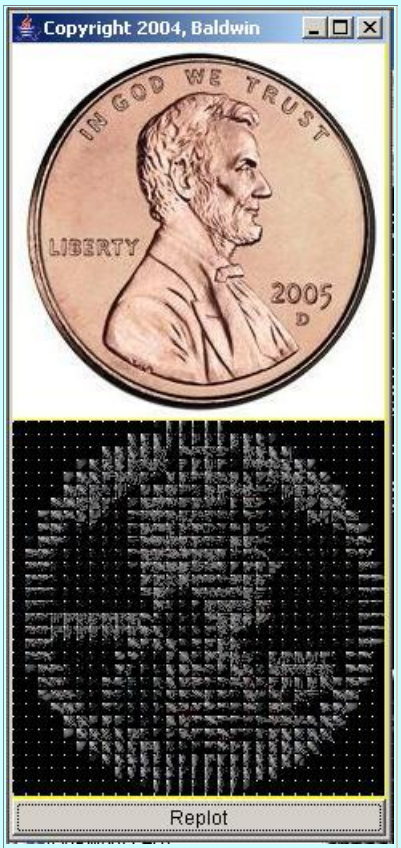

Figure 5

As mentioned earlier, most of the bright dots in the bottom panel of Figure 5 indicate the upper-left corners of individual 8x8 2D-DCT spectral blocks.

Re-running with a different block size

After startup, whenever the **Replot** button in Figure 5 is pressed, the process is rerun using the current value in the text field of Figure 4 to specify the size of the blocks. For example, changing the value in the text field to 20 and pressing the **Replot** button produces the output shown in Figure 6.

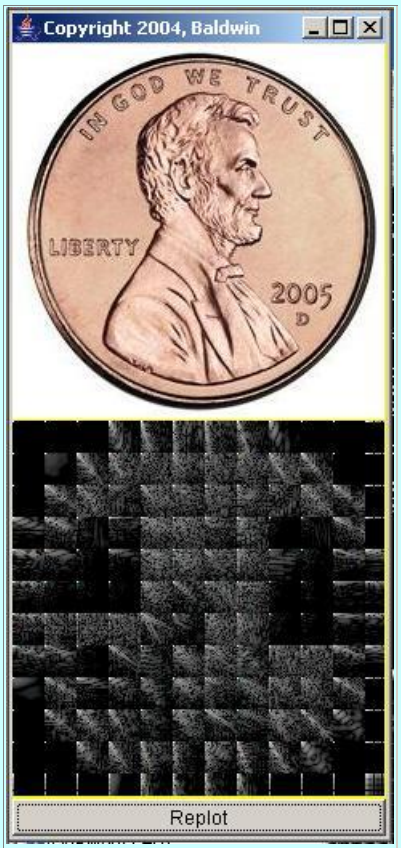

Figure 6

If you compare Figure 6 with Figure 5, you will see that each individual spectrum in the bottom panel of Figure 6 is 2.5 times larger than the size of the individual spectra in Figure 5.

Nothing magic about an 8x8 block size

As near as I have been able to determine, there was nothing magic about the selection of a block size of 8x8 pixels for the JPEG image compression algorithm. The process would probably work for just about any block size as long as it is large enough to satisfy the data compression objectives.

Smaller blocks mean more speed

However, as I explained earlier, the speed of the process can be improved by reducing the block size. Therefore, the members of the JPEG design committee were probably looking for the smallest block size that would do the job insofar as compression and quality of results is concerned, and they settled on a block size of 8x8 to meet that criterion.

> *(The compression algorithm that is used depends on being able to identify and to modify high-frequency components in the 2D-DCT spectrum. Therefore, the spectral block must be large enough to distinguish between low-frequency and*

high-frequency components. An 8x8 spectral block provides eight spectral samples between zero and the Nyquist folding frequency.)

What are we seeing in Figure 6?

It is good to remember what we are actually seeing in Figure 6. Each individual 20x20 spectral block in Figure 6 does not represent the true wave-number spectrum of the corresponding block of 20x20 pixels. Rather, each 20x20 spectral block represents the true wave-number spectrum of a modified version of the 20x20 block of pixels. As I explained in Part 1 of this lesson, each 20x20 block of pixels is implicitly extended to the left and above the origin *(at the upper-left corner of the block)* producing a new image that is symmetrical about the origin and four times as large.

(This symmetrical extension eliminates the requirement to perform complex arithmetic when computing the transform. It also modifies the results relative to the true wave-number spectrum of the block. In other words, the spectral blocks shown in Figure 6 are not true wave-number spectra of the original blocks of pixels.)

A true wave-number spectrum is not needed

However, the members of the JPEG committee weren't necessarily looking to compute the true wave-number spectrum of the block of image pixels. Rather, they were simply looking for a transformation that would accomplish their purpose and one that is economical to compute. They apparently concluded that the Discrete Cosine Transform meets that criterion.

(If they had wanted the true wave number spectrum of the block, they would probably have used the Fourier transform instead of the Discrete Cosine Transform.)

Why choose the Discrete Cosine Transform?

The 2D-DCT is simple and easy to compute. That was probably an important consideration in the original design of the JPEG image compression algorithm.

The 2D-DCT is reversible, meaning that computing the forward and inverse 2D-DCT on a block of pixels produces a very good replica of the original block of pixels. That is an absolute requirement of the algorithm.

And probably most important, the results of the 2D-DCT contribute greatly to the compressibility of the image data.

How does it contribute to compressibility?

Somewhere along the way, someone determined that you can significantly corrupt the values of the higher-frequency spectral components resulting from the 2D-DCT of an image block and the

image that results from performing an inverse 2D-DCT on the corrupted spectral data will still produce an acceptable, but not perfect replica of the original image block.

The JPEG committee determined that corrupting the higher-frequency spectral components through *selective re-quantization* would make the data more compressible while still producing acceptable results when an inverse 2D-DCT is performed on the corrupted spectral data.

> *(As mentioned earlier, selective re-quantization will be the topic of a future lesson.)*

The bottom line is that the 2D-DCT serves the purpose well

Although the 2D-DCT does not produce the true wave-number spectrum of the original image block, it does a very good job of satisfying the need to compress the image for transport and storage while still producing acceptable results when the image is reconstructed from the compressed spectral data.

An extreme case

I want to show you the results of two more experimental cases using **ImgMod35a** to help you better understand what's going on here. Figure 7 shows the result of setting the block size to 1 and re-computing and displaying the spectral results in the bottom panel.

Figure 7

Not a reconstructed image

The bottom panel in Figure 7 does not show a reconstructed version of the original image produced by performing an inverse 2D-DCT on the spectral data as was the case in Figure 3. Rather, the bottom panel of Figure 7 shows the spectral data just like Figure 6, Figure 5, and Figure 2.

Why does the spectral data look so much like the original image?

When the original image is sub-divided into 1x1-pixel blocks, each 2D-DCT is performed on a single image pixel producing a 1x1 spectral block. When only one spectral value is computed, it is the value at a frequency of zero.

Whenever the spectral value at a frequency of zero is computed, *(regardless of the size of the image block)*, the result is simply the *(possibly scaled)* average of all the pixel values. When there is only one pixel value involved, the average is simply the *(possibly scaled)* value of the pixel.

A scaled version of the pixel value

Thus, each of the 1x1 spectral values in Figure 7 is simply a scaled version of the corresponding pixel in the 1x1 pixel block. Displaying those spectral values reproduces the original image.

> *(However, the scaling that has been applied to the spectral values through the normalization process causes the colors to be washed out as shown in the bottom image of Figure 7. The normalization process includes a log base 10 transformation that tends to flatten the spectral surface.)*

Including the entire image in the block

In contrast, Figure 8 shows the result of setting the block size to the height of the image, causing the entire image to be included in one block of pixels.

Figure 8

As a result, the bottom panel in Figure 8 shows the 2D-DCT spectrum for the entire image taken as a whole.

Spectral energy near the origin

Most of the spectral energy is concentrated in the lighter portions of the spectrum near the origin in the upper-left corner. Although it isn't obvious, *(due to the adjacent white background)*, there is probably a very bright dot at the origin of the spectrum in Figure 8.

> *(Each of the individual spectra shown in Figure 7, Figure 6, Figure 5, and Figure 2 are simply smaller versions of Figure 8 where each of the individual spectra represent the information in a small block of corresponding image pixels.)*

Resemblance to the original image

The bottom panel in Figure 8 doesn't look anything like the original image because it represents the information for the entire image in a different form.

Figure 7, on the other hand, consisting of more than 50,000 individual single-point spectra looks a lot like the original image because each of the spectra in the bottom panel of Figure 7 corresponds to a single pixel in the original image.

The spectra in the bottom panels of Figure 2 and Figure 5 bear some resemblance to a grayscale version of the original image because each of the spectra represents the information in the corresponding small 8x8-pixel block of the image.

The spectra in the bottom panel of Figure 6 looks less like the image than Figure 5, but more like the image than Figure 8. Each of the individual spectra in Figure 6 represents the information in the corresponding 20x20-pixel block of the original image.

Individual spectra don't resemble corresponding image blocks

Except for the unique case of the single-point spectra shown in Figure 7, none of the individual spectra look much like the image blocks that they represent, and this is to be expected. In general, the frequency spectrum doesn't look like the image that it represents.

> *(Recall the section entitled **An analogy** in Part 1 of this lesson. Ice and liquid water both represent water, and it is relatively easy to transform the material back and forth between those two states. However, they don't look much alike.)*

The spatial organization

However, the spatial organization of the spectra listed above, particularly Figure 2 and Figure 5, provides a strong hint as to the organization of the information in the image.

Very little high-frequency energy

For example, the spectra that correspond to the large blank areas of the image in Figure 5 consist mainly of a peak at the origin with very little in the way of high-frequency energy. *(There are blank areas both inside and outside the coin.)* This causes those spectra to be mainly black.

Lots of high-frequency energy

On the other hand, the spectra for those areas of the image containing lots of detail, such as President Lincoln's hair and beard in Figure 5, exhibit considerably more high-frequency energy. This causes those spectra to have a much whiter appearance.

The spectral format

Once again, each of these spectra is a miniature version of the spectral format shown in the bottom panel of Figure 8. The energy at a frequency of zero is shown at the origin, which is at the upper-left corner of the spectral block. *(There is typically a bright dot at that point.)* The higher-frequency energy is shown to the right and below the origin.

> *(For example, the tip of President Lincoln's nose in Figure 5 and Figure 6 has a strong component at zero frequency and considerable energy in the low-frequency triangular area that makes up the upper-left portion of the spectral block. The*

high-frequency triangular area that makes up the bottom-right portion of the spectral block is largely black.)

Experimental results for ImgMod35

As explained earlier, the program, named **ImgMod35** subdivides an image into a set of 8x8-pixel blocks. *(The block size for this program is not an input parameter. It is fixed at 8x8 pixels.)* The program performs a forward 2D-DCT on each individual block producing a set of 8x8 spectral blocks. Then it performs an inverse 2D-DCT on each spectral block producing a replica of each of the original image blocks. The replicas of the small image blocks are then reassembled into a replica of the original image, which is displayed in the format shown in Figure 3.

Figure 3 shows the result of running **ImgMod35** on the same image shown in the top panels of Figure 1 and Figure 2 without exercising the option to mark the individual blocks as was done in Figure 1.

As another example, Figure 9 shows result of applying **ImgMod35** to the same image shown in the top panel of Figure 5.

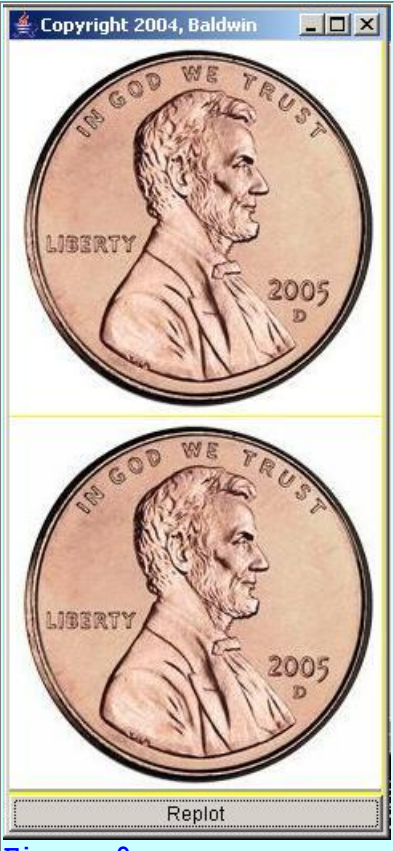

Figure 9

Whereas the bottom panel of Figure 5 shows the individual 8x8 spectral blocks, the bottom panel of Figure 9 shows the results of performing an inverse 2D-DCT on each of the 8x8 spectral blocks and using the resulting 8x8 image blocks to reconstruct the original image. As you can see, the reconstructed image is a very good replica of the original image.

Sample Code

The Program Named ImgMod35a

I will discuss this program in fragments. The first fragment, shown in Listing 1, shows the beginning of the class definition along with the declaration and initialization of a couple of instance variables.

```
class ImgMod35a extends Frame implements
ImgIntfc02{

  TextField subSizeField = new TextField("8");
  Label instructions = new Label(
                  "Enter sub-image size and
click Replot");
```

Listing 1

The two instance variables provide the label and the text field shown in Figure 4.

The TextField variable

This text field supports the interactive nature of the program. The user can enter a value into the text field and then click the **Replot** button shown at the bottom of Figure 5. This causes the image to be broken down into square blocks of pixels where the number of pixels on each side of the block matches the value entered into the text field by the user. Then the image is processed one block at a time producing spectral results as shown in Figure 6.

The interface named ImgIntfc02

Note that the class implements the interface named **ImgIntfc02**. This is necessary to support the graphic display aspects of the program as explained in the earlier lesson entitled Processing Image Pixels using Java, Getting Started.

The constructor

The constructor for the class is shown in its entirety in Listing 2.

```
  ImgMod35a(){//constructor
    add(instructions,BorderLayout.CENTER);
    add(subSizeField,BorderLayout.SOUTH);
    setTitle("Copyright 2006, R.G.Baldwin");
```

```
    setBounds(400,0,250,75);
    setVisible(true);
  }//end constructor

Listing 2
```

The code in Listing 2 is straightforward and shouldn't require further explanation.

The processImg method

Because this class implements the **ImgIntfc02** interface, and because that interface declares the **processImg** method, this class must define the method named **processImg**.

The signature for the method named **processImg** is shown in Listing 3.

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

Listing 3
```

As mentioned above, this method is required by **ImgIntfc02**. As explained in the earlier lesson entitled Processing Image Pixels using Java, Getting Started, the method is called at the beginning of the run and each time thereafter that the user clicks the **Replot** button shown at the bottom of Figure 6.

The block size

Also as explained earlier, the image is broken down into square blocks for processing. The dimensions of the blocks are specified by the value stored in the variable named **imgSubSize** shown in Listing 4.

```
    int imgSubSize = Integer.parseInt(

subSizeField.getText());

Listing 4
```

Straightforward code from an earlier lesson

The comments in Listing 5, in conjunction with information provided in the earlier lesson, should suffice to explain the code in Listing 5.

```
    //Create an empty output array of the same
size as the
    // incoming array.
    int[][][] output = new
```

```
int[imgRows][imgCols][4];

    //Make a working copy of the 3D pixel array
as type
    // double to avoid making permanent changes
to the
    // original image data.  Also, all
processing will be
    // performed as type double.
    double[][][] working3D =
copyToDouble(threeDPix);

    //The following code can be enabled to set
any of the
    // three colors to black, thus removing
them from the
    // output.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
//        working3D[row][col][1] = 0;//Red
//        working3D[row][col][2] = 0;//Green
//        working3D[row][col][3] = 0;//Blue
      }//end inner loop
    }//end outer loop

Listing 5
```

Performing a forward DCT

Listing 6 begins the process of extracting and performing a forward DCT on the red color plane of the image, one block at a time.

```
    double[][] redPlane =
extractPlane(working3D,1);

Listing 6
```

The extractPlane method

Listing 6 invokes the method named **extractPlane** to extract the red color plane from the image. The purpose of the **extractPlane** 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**. The method is straightforward and shouldn't require further explanation. It can be viewed in its entirety in Listing 17 near the end of the lesson.

The expandPlane method

Listing 7 invokes the **expandPlane** method to expand the plane such that both dimensions are a multiple of **imgSubSize**.

```
    redPlane =
expandPlane(redPlane,imgSubSize);
```

Listing 7

The extra rows and columns of pixel data *(if any)* that are required to cause the dimensions of the plane to be a multiple of **imgSubSize** are filled with pixel values of zero. Otherwise, the **expandPlane** method is straightforward and shouldn't require further explanation. The method can be viewed in its entirety in Listing 17 near the end of the lesson.

The forwardXformPlane method

Listing 8 invokes the **forwardXformPlane** method to perform the forward DCT on the red plane turning it into a spectral plane where each individual spectrum stored in the plane is a square block with dimensions of **imgSubSize**.

```
    forwardXformPlane(redPlane,imgSubSize);
```

Listing 8

Ultimately, the spectra for all three color planes are combined and displayed in an image format as shown in Figure 6. To see the spectra for the individual color planes, you can disable all but one color plane as instructed by the comments in Listing 5.

At this point, I will set the **processImg** method aside for awhile and discuss the **forwardXformPlane** method. Before doing that, however, I want to review some of the concepts that you learned about in Part 1 of this lesson.

A separable transform

As was explained in Part 1 of this lesson, one of the significant attributes of the two-dimensional Discrete Cosine Transform *(2D-DCT)* is that it is separable. What this means in practice is that to compute the DCT for a block in a color plane of a 2D image, you can begin by performing a *one-dimensional* DCT on each row of the block, creating a new 2D structure where each row of the new structure contains the DCT of the corresponding row of the block of pixel data. Then you can perform a one-dimensional DCT on each column of the new 2D structure creating a third 2D structure containing the 2D-DCT of the original block.

An in-place transform

Also important, at least from a memory utilization viewpoint, is the fact that you can perform the transforms *"in-place"* using the original color plane for intermediate and final data storage without a requirement to allocate memory for the new structures.

Don't need a new DCT program

What this means for me is that I don't need to develop a new DCT program to handle the 2D case. Rather, I could perform all the necessary DCT transforms that I need using the static one-dimensional forward DCT method named **transform** belonging to the class named **ForwardDCT01**. I developed and explained that class in the earlier lesson entitled Understanding the Discrete Cosine Transform in Java.

New version for optimization

Note however that I did develop an optimized version of the **transform** method that is optimized for speed for the case where the image is divided into 8x8 blocks of pixels prior to performing the transform.

The forwardXformPlane method

The **forwardXformPlane** method is shown in its entirety in Listing 9.

```
  void forwardXformPlane(double[][] plane,int
imgSubSize){
    int pixRows = plane.length;
    int pixCols = plane[0].length;
    //Loop on rows of blocks
    for(int segRow = 0;segRow <
pixRows/imgSubSize;

segRow++){
      //Loop on cols of blocks
      for(int segCol = 0;segCol <
pixCols/imgSubSize;

segCol++){
        double[][] theSubPlane =

getSubPlane(plane,segRow,segCol,imgSubSize);

forwardXformSubPlane(theSubPlane,imgSubSize);

        insertSubPlane(plane,
                       theSubPlane,
                       segRow,
                       segCol,
                       imgSubSize);
      }//end inner loop
    }//end outer loop
  }//end forwardXformPlane

Listing 9
```

The **forwardXformPlane** method breaks a **double** color plane down into square blocks of size **imgSubSize** and performs a forward DCT on each block. Then it assembles the resulting spectral data blocks into a spectral plane.

*(The method assumes that the dimensions of the color plane are multiples of **imgSubSize**.)*

Invokes three methods in succession

The code in Listing 9 does its job by invoking the following three methods in succession inside a nested **for** loop:

- **getSubPlane** - Extracts and returns a square block of size **imgSubSize** from a **double** 2D plane
- **forwardXformSubPlane** - performs a forward DCT on a square block of pixels of size **imgSubSize** received as an incoming parameter.
- **insertSubPlane** - Inserts a square block of size **imgSubSize** into a double 2D plane

The first and third methods in the above list are completely straightforward and shouldn't require any explanation beyond the comments contained in the code. You can view the code for those methods in Listing 17 near the end of the lesson.

The method named forwardXformSubPlane

The method named **forwardXformSubPlane**, which performs a forward DCT on a square block of pixels of size imgSubSize, is shown in its entirety in Listing 10.

```
  void forwardXformSubPlane(double[][]
theSubPlane,
                            int imgSubSize){

    int imgRows = imgSubSize;
    int imgCols = imgSubSize;

    //Extract each row from the block and
perform
    // a forward DCT on the row. Then insert
the
    // transformed row back into the block.  At
that point,
    // the row no longer contains color pixel
data, but
    // has been transformed into a row of
spectral data.
    for(int row = 0;row < imgRows;row++){
      double[] theRow =
extractRow(theSubPlane,row);

      double[] theXform = new
double[theRow.length];
      //Perform the forward transform.
      ForwardDCT02.transform(theRow,theXform);

      //Insert the transformed row back into
the block.
```

```
      insertRow(theSubPlane,theXform,row);
    }//end for loop

    //The block now contains the results of
doing the
    // horizontal DCT one row at a time.  The
block no
    // longer contains color pixel data.
Rather, it
    // contains spectral data for the
horizontal dimension
    // only.

    //Extract each column from the block and
perform a
    // forward DCT on the column. Then insert
the
    // transformed column back into the block.
    for(int col = 0;col < imgCols;col++){
      double[] theCol =
extractCol(theSubPlane,col);

      double[] theXform = new
double[theCol.length];
      ForwardDCT02.transform(theCol,theXform);

      insertCol(theSubPlane,theXform,col);
    }//end for loop

    //The square block of size imgSubSize has
now been
    // converted into a square block of
spectral
    // coefficient data of the same size.

  }//end forwardXformSubPlane
```

Listing 10

Very similar to a method from Part 1

The method named **forwardXformSubPlane** is very similar to the method named **processPlane** that I explained in Part 1 of this lesson. Therefore, an explanation should not be required in this lesson.

The main difference

The main difference between the method named **forwardXformSubPlane** in Listing 10 and the method named **processPlane** in Part 1 of this lesson has to do with the method that is called to perform the actual DCT.

Optimized for speed

The method in the earlier lesson calls the **transform** method in the class named **ForwardDCT01**, whereas the method in Listing 10 calls the **transform** method in the class named **ForwardDCT02**. I will explain the difference between the two later. For now, suffice it to say that the latter method was optimized for speed for the case where the block size is 8x8 pixels.

Back to the processImg method

Returning once again to the explanation of the **processImg** method, and picking up where Listing 8 left off, Listing 11 invokes the **normalize** method to normalize the data in the 2D double plane to make it compatible with being displayed as an image plane.

```
    normalize(redPlane);
```

Listing 11

The normalize method

Normalizing data for display purposes is almost always problematic. The display will often look very different depending on how you perform the normalization. The approach to normalization used in the **normalize** method performs the following steps in order:

1. All negative values are converted to positive values. This is equivalent to the computation of the magnitude of the spectrum for purely real data.
2. All values are converted to log base 10 *(scaled decibels)* to preserve the dynamic range of the plotting system. All negative results *(if any)* are set to 0.
3. All values that are below X-percent of the maximum value are set to X-percent of the maximum value producing a floor for the values where the value of X is determined by a hard-coded scale factor.
4. All values are biased *(slid toward 0)* so that the minimum value *(the floor)* becomes 0.
5. All values are scaled so that the maximum value becomes 255. As a result, all values range from 0 as a minimum to 255 as a maximum.

Once the normalizing rationale has been established, the code to establish the normalization is relatively straightforward. You can view the **normalize** method in its entirety in Listing 17.

The insertPlane method

Listing 12 invokes the **insertPlane** method to insert the spectral plane back into the 3D array.

```
    insertPlane(working3D,redPlane,1);
```

Listing 12

The **insertPlane** method also trims off any extra rows and columns created by expanding the dimensions to a multiple of **imgSubSize** earlier in Listing 7. The code in the **insertPlane** method is straightforward. The method can be viewed in its entirety in Listing 17.

Process green and blue color planes

Listing 13 executes essentially the same code as that explained above to extract and perform forward Discrete Cosine Transforms on the green and blue color planes.

```
    double[][] greenPlane =
extractPlane(working3D,2);
    greenPlane =
expandPlane(greenPlane,imgSubSize);
    forwardXformPlane(greenPlane,imgSubSize);
    normalize(greenPlane);
    insertPlane(working3D,greenPlane,2);

    double[][] bluePlane =
extractPlane(working3D,3);
    bluePlane =
expandPlane(bluePlane,imgSubSize);
    forwardXformPlane(bluePlane,imgSubSize);
    normalize(bluePlane);
    insertPlane(working3D,bluePlane,3);

Listing 13
```

All three color planes have been transformed

At this point, all three color planes have now been transformed into spectral planes where each spectral plane is composed of a potentially large number of contiguous square spectral data blocks of size **imgSubSize**.

Convert to type int and return

Listing 14 invokes the **copyToInt** method to convert the image color planes to type **int**. Then Listing 14 returns the array of pixel data to the calling method where it is displayed as shown in the bottom panel of Figure 5.

```
    output = copyToInt(working3D);
    return output;
  }//end processImg method

Listing 14
```

You can view the **copyToInt** method, along with a few other utility methods not discussed above in Listing 17.

The Class Named ForwardDCT02

As mentioned earlier, whereas the program in Part 1 of this lesson used the **transform** method in the class named **ForwardDCT01** to perform the DCT, this program uses the **transform** method of the class named **ForwardDCT02** to perform the DCT.

The static method named **transform** performs a forward DCT on an incoming series and returns the DCT spectrum.

Optimized for 8x8 blocks

The **transform** method of the **ForwardDCT01** class is a general purpose version that works for images of different sizes. The **transform** method of the **ForwardDCT02** class will also work with images of any size, but it is optimized for use with images with a size of 8x8 pixels.

Uses a cosine lookup table

When transforming an 8x8-pixel image, the **transform** method of the **ForwardDCT02** class uses an 8x8 cosine lookup table instead of calling the cos function. If the image size is anything other than 8x8 pixels, the method simply calls the cosine function during each iteration.

It is probably faster to use the lookup table than it is to call the cosine function every time a cosine value is needed.

The class named **ForwardDCT02** is shown in its entirety in Listing 18 near the end of this lesson. The manner in which the cosine table is created and then used is well documented. If you understood the **transform** method of the **ForwardDCT01** class in Part 1 of this lesson, you should have no difficulty understanding the **transform** method of the class named **ForwardDCT02**.

The Program Named ImgMod35

This program performs a forward DCT on an image converting the three color planes into spectral planes. Then it performs an inverse DCT on the three spectral planes converting them back into image color planes.

A rather detailed description of the program was provided earlier in this lesson. I will refer you back to that description and will not repeat that description here.

Note that this version of the program is significantly faster than the version named **ImgMod034** from Part 1 of this lesson. The reasons for the improvement in speed were also explained earlier.

Experimental results

Figure 3 and Figure 9 show examples of the results produced by this program.

Similar code

Much of the code in this program is the same as, or very similar to the code in the program named **ImgMod35a**, which I explained earlier. Therefore, I won't repeat the explanation for that code. I will only explain the code that is significantly different. The code that I will explain will be explained in fragments. You can view a listing of the entire program in Listing 19 near the end of the lesson.

The first interesting code fragment begins in Listing 15, which extracts the red color plane and performs a forward DCT on that plane.

```
    double[][] redPlane =
extractPlane(working3D,1);
    redPlane = expandPlane(redPlane);
    forwardXformPlane(redPlane);
    insertPlane(working3D,redPlane,1);

Listing 15
```

The expandPlane method

Having extracted the red color plane, Listing 15 invokes the **expandPlane** method to expand the plane such that both dimensions are an even multiple of 8 pixels.

> *(Unlike the **expandPlane** method in the previous program named **ImgMod35a**, the **expandPlane** method in this program does not accept an incoming parameter that specifies the block size. Rather, the block size is fixed at 8x8 pixels in this program.)*

The forwardXformPlane method

Then Figure 15 invokes the **forwardXformPlane** method to perform a forward DCT on the red plane turning it into a spectral plane where each individual spectrum stored in the plane is an 8x8 square.

The method named **forwardXformPlane** breaks a double color plane down into 8x8-pixel blocks and performs a forward DCT on each block. Then it assembles the resulting spectral data blocks into a spectral plane. The method is hard-coded to assume that the dimensions of the color plane are multiples of 8.

The insertPlane method

Finally, Listing 15 invokes the **insertPlane** method to insert the spectral plane back into the 3D array. This method also trims off any extra rows and columns that may have been created by expanding the dimensions to a multiple of 8.

Process the green and blue color planes

Then the program performs essentially the same operations on the green and blue color planes. Once that is completed, all three color planes have been transformed into spectral planes where each spectral plane is composed of a potentially large number of contiguous 8x8 spectral data blocks.

> *(Note that because there is no requirement to display the spectral planes, they are not normalized to values between 0 and 255 as is the case with the program named **ImgMod035a**.)*

Transform back into image color planes

Following this, the program transforms the spectral planes back into image color planes. The inverse transform on the red spectral plane consists of the four method calls shown in Listing 16.

```
    redPlane = extractPlane(working3D,1);
    redPlane = expandPlane(redPlane);
    inverseXformPlane(redPlane);
    insertPlane(working3D,redPlane,1);

Listing 16
```

The process begins by invoking the **extractPlane** method to extract the red spectral plane from the 3D array. This method has been used in numerous previous lessons and can be found in Listing 19.

Then Listing 16 invokes the **expandPlane** method to expand the plane such that both dimensions are an even multiple of 8. There is nothing new here.

Then Listing 16 invokes the **inverseXformPlane** method to perform the inverse DCT on the red plane. The only thing new here is that the **inverseXformPlane** method calls a method named **inverseXform8x8Plane**, which uses the **transform** method of the **InverseDCT02** class to perform the inverse transform.

Optimized for 8x8 spectral blocks

This **transform** method is essentially the same as the **transform** method of the class named **InverseDCT01**, which was explained in Part 1 of this lesson, except that the **transform** method of the **InverseDCT02** class is optimized for speed when applied to spectra having a size of 8x8 spectral coefficient values. Basically, it uses a cosine lookup table rather than invoking the **cos** method each time a cosine value is needed.

A complete listing of the class named **InverseDCT02** is provided in Listing 20 near the end of the lesson.

Finally, Listing 16 invokes the **insertPlane** method to insert the new red image plane back into the working 3D array.

Process the green and blue spectral planes

Following this, the program performs essentially the same inverse transform operations on the green and blue spectral planes to produce the green and blue image planes.

Then the new image color planes are converted to type **int** and returned to the calling method for display as shown in the bottom panel of Figure 3 and Figure 9.

Run the Program

I encourage you to copy the code from the listings in the section entitled Complete Program Listings. Compile the code and run it. Experiment with the code, making changes, and observing the results of your changes.

For example, you might want to make modifications to the image data prior to performing the forward transforms and observe the impact of those modifications on the spectral data.

Similarly, you might want to make modifications to the spectral data prior to performing the inverse transform, and observe the impact of those modifications on the resulting images.

See instructions here and here on how to run the programs.

Whatever you do, have fun and learn as much as you can about the use of the 2D-DCT with image data.

Summary

In Part 1 of this lesson I taught you how to use the forward two-dimensional Discrete Cosine Transform *(2D-DCT)* to compute and display the wave-number spectrum of an image. I also taught you how to apply the inverse 2D-DCT to the spectral data to reconstruct and display a replica of the original image.

In this document *(Part 2)*, I taught you how to sub-divide the image before applying the forward and inverse 2D-DCTs in a manner similar to the way it is done in the JPEG image compression algorithm.

I showed that sub-dividing the image can result in a significant improvement in transform speed while still preserving the visual quality of the original image.

I also explained some of the theory behind and some of the reasons for sub-dividing images in that manner.

What's Next?

Future lessons in this series will explain the inner workings behind several data and image compression schemes, including the following:

- Run-length data encoding
- GIF image compression
- JPEG image compression

Along the way, I will probably also branch off and show you how to use the 2D-DCT to create hidden watermarks on images.

References

General

2440 Understanding the Lempel-Ziv Data Compression Algorithm in Java
2442 Understanding the Huffman Data Compression Algorithm in Java
2444 Understanding the Discrete Cosine Transform in Java
2446 Understanding the 2D Discrete Cosine Transform in Java, Part 1
1468 Plotting Engineering and Scientific Data using Java
1478 Fun with Java, How and Why Spectral Analysis Works
1482 Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
1483 Spectrum Analysis using Java, Frequency Resolution versus Data Length
1484 Spectrum Analysis using Java, Complex Spectrum and Phase Angle
1485 Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain
1486 Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm
1489 Plotting 3D Surfaces using Java
1490 2D Fourier Transforms using Java
1491 2D Fourier Transforms using Java, Part 2

Discrete Cosine Transform equations

Discrete cosine transform - Wikipedia, the free encyclopedia
The Data Analysis Briefbook - Discrete Cosine Transform
National Taiwan University - DSP Group - Discrete Cosine Transform

Complete Program Listings

Complete listings of the programs discussed in this lesson are shown in Listing 17 through Listing 20 below.

```
/*File ImgMod35a.java
Copyright 2005, R.G.Baldwin

This is a modification of ImgMod35 to normalize and display
```

wave-number spectra instead of constructing a replica of the original image. In addition, whereas ImgMod35 uses a fixed block size of 8x8 pixels, this program allows the user to specify the block size.

This program performs a forward DCT on an image converting the three color planes into spectral planes. The three spectral planes are normalized so as to make them suitable for being displayed as standard image data. The normalization includes transformation of the spectral data to log base 10 and scaling to cause the values to fall between 0 and 255 inclusive.

This program breaks the image down into square blocks of pixels and performs a forward DCT on each block producing and displaying square blocks of spectral data. The size of the blocks is specified by the user by entering the size into a text field and pressing the Replot button.

Each color plane is individually processed.

Note that for small block sizes, this version of the program is significantly faster than the version named ImgMod034a, which does not break the image down into blocks, but rather does the forward DCT on the entire image.

This program can handle any block size up to the size of the original image. However, it is optimized for block sizes of 8x8 pixels. When the block size is 8x8 pixels, the program uses an 8x8 cosine lookup table instead of computing the cosine value every time it is needed. This should cause the program to run faster.

The class is designed to be driven by the class named ImgMod02a.

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

```
java ImgMod02a ImgMod35a ImageFileName
```

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

When the user clicks the Replot button, the process is repeated using the potentially new value for block size and the new results are displayed.

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

```
ImgIntfc02.class
ImgMod02a.class
ImgMod35a.class
```

```
ForwardDCT02.class

Tested using J2SE 5.0 and WinXP.  J2SE 5.0 or later is
required due to the use of static imports.
**********************************************************/
import java.awt.*;
import java.io.*;
import static java.lang.Math.*;

class ImgMod35a extends Frame implements ImgIntfc02{

  //The user can enter a value into the following field and
  // then click the Replot button.  The image will be
  // broken down into square blocks of pixels where the
  // number of pixels on each side matches the value
  // entered by the user.  Then the image will be
  // processed one block at a time.
  TextField subSizeField = new TextField("8");
  Label instructions = new Label(
                  "Enter sub-image size and click Replot");
  //-------------------------------------------------------//
  ImgMod35a(){//constructor
    add(instructions,BorderLayout.CENTER);
    add(subSizeField,BorderLayout.SOUTH);
    setTitle("Copyright 2006, R.G.Baldwin");
    setBounds(400,0,250,75);
    setVisible(true);
  }//end constructor
  //-------------------------------------------------------//

  //This method is required by ImgIntfc02.  It is called at
  // the beginning of the run and each time thereafter that
  // the user clicks the Replot button on the Frame
  // containing the images.
  public int[][][] processImg(int[][][] threeDPix,
                              int imgRows,
                              int imgCols){

    //The image is broken down into square blocks where the
    // following value specifies the dimensions of the
    // blocks.
    int imgSubSize = Integer.parseInt(
                                  subSizeField.getText());

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

    //Make a working copy of the 3D pixel array as type
    // double to avoid making permanent changes to the
    // original image data.  Also, all processing will be
    // performed as type double.
    double[][][] working3D = copyToDouble(threeDPix);

    //The following code can be enabled to set any of the
    // three colors to black, thus removing them from the
```

```
    // output.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
//        working3D[row][col][1] = 0;//Red
//        working3D[row][col][2] = 0;//Green
//        working3D[row][col][3] = 0;//Blue
      }//end inner loop
    }//end outer loop

    //Extract and do a forward DCT on the red color plane,
    // one block at a time.
    double[][] redPlane = extractPlane(working3D,1);
    //Expand the plane such that both dimensions are a
    // multiple of imgSubSize.
    redPlane = expandPlane(redPlane,imgSubSize);
    //Do the forward DCT on the redPlane turning it into a
    // spectral plane where each individual spectrum
    // stored in the plane is a square block with
    // dimensions of imgSubSize.
    //To see the individual spectra, disable all but one
    // color plane above.
    forwardXformPlane(redPlane,imgSubSize);

    //Normalize the data in a 2D double plane to make it
    // compatible with being displayed as an image plane.
    // See the comments in the normalize method for an
    // explanation as to how the normalization is
    // accomplished.
    normalize(redPlane);

    //Insert the spectral plane back into the 3D array.
    // This method also trims off any extra rows and
    // columns created by expanding the dimensions to a
    // multiple of imgSubSize.
    insertPlane(working3D,redPlane,1);

    //Extract and do a forward DCT on the green color
    // plane using the same procedure used for the red
    // plane above.
    double[][] greenPlane = extractPlane(working3D,2);
    greenPlane = expandPlane(greenPlane,imgSubSize);
    forwardXformPlane(greenPlane,imgSubSize);
    normalize(greenPlane);
    insertPlane(working3D,greenPlane,2);

    //Extract and do a forward DCT on the blue color plane
    //  using the same procedure used for the red plane
    // above..
    double[][] bluePlane = extractPlane(working3D,3);
    bluePlane = expandPlane(bluePlane,imgSubSize);
    forwardXformPlane(bluePlane,imgSubSize);
    normalize(bluePlane);
    insertPlane(working3D,bluePlane,3);

    //All three color planes have now been transformed into
    // spectral planes where each spectral plane is
```

```
    // composed of a potentially large number of contiguous
    // square spectral data blocks of size imgSubSize.

    //Convert the image color planes to type int and return
    // the array of pixel data to the calling method.
    output = copyToInt(working3D);
    //Return a reference to the output array.
    return output;

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

  //The purpose of this method is to extract a specified
  // row from a double 2D plane and to return it as a one-
  // dimensional array of type double.
  double[] extractRow(double[][] colorPlane,int row){

    int numCols = colorPlane[0].length;
    double[] output = new double[numCols];
    for(int col = 0;col < numCols;col++){
      output[col] = colorPlane[row][col];
    }//end outer loop
    return output;
  }//end extractRow
  //-----------------------------------------------------//

  //The purpose of this method is to insert a specified
  // row of double data into a double 2D plane.
  void insertRow(double[][] colorPlane,
                 double[] theRow,
                 int row){
    int numCols = colorPlane[0].length;
    double[] output = new double[numCols];
    for(int col = 0;col < numCols;col++){
      colorPlane[row][col] = theRow[col];
    }//end outer loop
  }//end insertRow
  //-----------------------------------------------------//

  //The purpose of this method is to extract a specified
  // col from a double 2D plane and to return it as a one-
  // dimensional array of type double.
  double[] extractCol(double[][] colorPlane,int col){
    int numRows = colorPlane.length;
    double[] output = new double[numRows];
    for(int row = 0;row < numRows;row++){
      output[row] = colorPlane[row][col];
    }//end outer loop
    return output;
  }//end extractCol
  //-----------------------------------------------------//

  //The purpose of this method is to insert a specified
  // col of double data into a double 2D color plane.
  void insertCol(double[][] colorPlane,
                 double[] theCol,
```

```
                int col){
  int numRows = colorPlane.length;
  double[] output = new double[numRows];
  for(int row = 0;row < numRows;row++){
    colorPlane[row][col] = theCol[row];
  }//end outer loop
}//end insertCol
//-----------------------------------------------------//

//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.
public double[][] extractPlane(
                            double[][][] threeDPixDouble,
                            int plane){

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

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

  //Copy the values from the specified plane to the
  // double array.
  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 extractPlane
//-----------------------------------------------------//

//The purpose of this method is to insert a double 2D
// plane into the double 3D array that represents an
// image.  This method also trims off any extra rows and
// columns in the double 2D plane.
public void insertPlane(double[][][] threeDPixDouble,
                        double[][] colorPlane,
                        int plane){

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

  //Copy the values from the incoming color plane to the
  // specified plane in the 3D array.
  for(int row = 0;row < numImgRows;row++){
    for(int col = 0;col < numImgCols;col++){
      threeDPixDouble[row][col][plane] =
                                      colorPlane[row][col];
    }//end loop on col
  }//end loop on row
}//end insertPlane
```

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

  //This method copies an int version of a 3D pixel array
  // to an new pixel array of type double.
  double[][][] copyToDouble(int[][][] threeDPix){
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    double[][][] new3D = new double[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] = threeDPix[row][col][0];
        new3D[row][col][1] = threeDPix[row][col][1];
        new3D[row][col][2] = threeDPix[row][col][2];
        new3D[row][col][3] = threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
  }//end copyToDouble
  //-------------------------------------------------------//

  //This method copies a double version of a 3D pixel array
  // into a new pixel array of type int.
  int[][][] copyToInt(double[][][] threeDPixDouble){
    int imgRows = threeDPixDouble.length;
    int imgCols = threeDPixDouble[0].length;

    int[][][] new3D = new int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] =
                          (int)threeDPixDouble[row][col][0];
        new3D[row][col][1] =
                          (int)threeDPixDouble[row][col][1];
        new3D[row][col][2] =
                          (int)threeDPixDouble[row][col][2];
        new3D[row][col][3] =
                          (int)threeDPixDouble[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
  }//end copyToInt
  //-------------------------------------------------------//

  //Expand a double 2D plane such that both dimensions are
  // a multiple of imgSubSize.
  double[][] expandPlane(double[][] plane,int imgSubSize){
    int rows = plane.length;
    int cols = plane[0].length;
    double[][] output;

    int rowsRem = rows%imgSubSize;
    int colsRem = cols%imgSubSize;

    if((rowsRem == 0) && (colsRem == 0)){
      //No expansion is required.
```

```
      return plane;
    }else{
      //An expansion is required.  Copy the data from the
      // incoming array to a new expanded array, leaving
      // pad values of 0.0 at the right and bottom edges.
      output = new double[rows + (imgSubSize - rowsRem)]
                         [cols + (imgSubSize - colsRem)];
      for(int row = 0;row < rows;row++){
        for(int col = 0;col < cols;col++){
          output[row][col] = plane[row][col];
        }//end inner loop
      }//end outer loop
      return output;
    }//end else
  }//end expandPlane
  //-----------------------------------------------------//

  //Extracts and returns a square block of size imgSubSize
  // from a double 2D plane
  double[][] getSubPlane(double[][] colorPlane,
                         int segRow,
                         int segCol,
                         int imgSubSize){
    double[][] theSubPlane =
                        new double[imgSubSize][imgSubSize];
    for(int bigRow = segRow * imgSubSize,smallRow = 0;
              bigRow < (segRow * imgSubSize + imgSubSize);
                                       bigRow++,smallRow++){
      for(int bigCol = segCol * imgSubSize,smallCol = 0;
                bigCol < segCol * imgSubSize + imgSubSize;
                                       bigCol++,smallCol++){
        theSubPlane[smallRow][smallCol] =
                                  colorPlane[bigRow][bigCol];
      }//end inner loop
    }//end outer loop
    return theSubPlane;
  }//end getSubPlane
  //-----------------------------------------------------//

  //Inserts square block of size imgSubSize into a double
  // 2D plane
  void insertSubPlane(double[][] colorPlane,
                      double[][] theSubPlane,
                      int segRow,
                      int segCol,
                      int imgSubSize){
    for(int bigRow = segRow * imgSubSize,smallRow = 0;
              bigRow < (segRow * imgSubSize + imgSubSize);
                                       bigRow++,smallRow++){
      for(int bigCol = segCol * imgSubSize,smallCol = 0;
                bigCol < segCol * imgSubSize + imgSubSize;
                                       bigCol++,smallCol++){
        colorPlane[bigRow][bigCol] =
                             theSubPlane[smallRow][smallCol];
      }//end inner loop
    }//end outer loop
```

```
  }//end insertSubPlane
  //-------------------------------------------------//

  //Breaks a double color plane down into square blocks
  // of size imgSubSize and does a forward DCT xform on
  // each block.
  //Assembles the resulting spectral data blocks into a
  // spectral plane.
  //Assumes that the dimensions of the color plane are
  // multiples of imgSubSize.
  void forwardXformPlane(double[][] plane,int imgSubSize){
    int pixRows = plane.length;
    int pixCols = plane[0].length;
    //Loop on rows of blocks
    for(int segRow = 0;segRow < pixRows/imgSubSize;
                                                segRow++){
      //Loop on cols of blocks
      for(int segCol = 0;segCol < pixCols/imgSubSize;
                                                segCol++){
        double[][] theSubPlane =
               getSubPlane(plane,segRow,segCol,imgSubSize);
        forwardXformSubPlane(theSubPlane,imgSubSize);

        insertSubPlane(plane,
                       theSubPlane,
                       segRow,
                       segCol,
                       imgSubSize);
      }//end inner loop
    }//end outer loop
  }//end forwardXformPlane
  //-------------------------------------------------//

  //This method does a forward DCT on a square block of
  // pixels of size imgSubSize received as an incoming
  // parameter.
  void forwardXformSubPlane(double[][] theSubPlane,
                            int imgSubSize){

    int imgRows = imgSubSize;
    int imgCols = imgSubSize;

    //Extract each row from the block and perform
    // a forward DCT on the row. Then insert the
    // transformed row back into the block.  At that point,
    // the row no longer contains color pixel data, but
    // has been transformed into a row of spectral data.
    for(int row = 0;row < imgRows;row++){
      double[] theRow = extractRow(theSubPlane,row);

      double[] theXform = new double[theRow.length];
      //Perform the forward transform.
      ForwardDCT02.transform(theRow,theXform);

      //Insert the transformed row back into the block.
      insertRow(theSubPlane,theXform,row);
```

```
    }//end for loop

    //The block now contains the results of doing the
    // horizontal DCT one row at a time.  The block no
    // longer contains color pixel data.  Rather, it
    // contains spectral data for the horizontal dimension
    // only.

    //Extract each column from the block and perform a
    // forward DCT on the column. Then insert the
    // transformed column back into the block.
    for(int col = 0;col < imgCols;col++){
      double[] theCol = extractCol(theSubPlane,col);

      double[] theXform = new double[theCol.length];
      ForwardDCT02.transform(theCol,theXform);

      insertCol(theSubPlane,theXform,col);
    }//end for loop

    //The square block of size imgSubSize has now been
    // converted into a square block of spectral
    // coefficient data of the same size.

  }//end forwardXformSubPlane
  //-------------------------------------------------//

  //Normalizes the data in a 2D double plane to make it
  // compatible with being displayed as an image plane.
  //First all negative values are converted to positive
  // values.
  //Then all values are converted to log base 10 to
  // preserve the dynamic range of the plotting system.
  // All negative values are set to 0 at this point.
  //Then all values that are below X-percent of the maximum
  // value are set to X-percent of the maximum value
  // producing a floor for the values.
  //Then all values are biased so that the minimum value
  // (the floor) becomes 0.
  //Then all values are scaled so that the maximum value
  // becomes 255.
  void normalize(double[][] plane){
    int rows = plane.length;
    int cols = plane[0].length;

    //Begin by converting all negative values to positive
    // values.  This is equivalent to the computation of
    // the magnitude for purely real data.
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        if(plane[row][col] < 0){
          plane[row][col] = - plane[row][col];
        }//end if
      }//end inner loop
    }//end outer loop
```

```
    //Convert the values to log base 10 to preserve the
    // dynamic range of the plotting system.  Set negative
    // values to 0.

    //First eliminate or change any values that are
    // incompatible with log10 method.
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        if(plane[row][col] == 0.0){
          plane[row][col] = 0.0000001;
        }else if(plane[row][col] == Double.NaN){
          plane[row][col] = 0.0000001;
        }else if(plane[row][col] ==
                                  Double.POSITIVE_INFINITY){
          plane[row][col] = 9999999999.0;
        }//end else
      }//end inner loop
    }//end outer loop

    //Now convert the data to log base 10 setting all
    // negative results to 0.
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        plane[row][col] = log10(plane[row][col]);
        if(plane[row][col] < 0){
          plane[row][col] = 0;
        }//end if
      }//end inner loop
    }//end outer loop

    //Now set everything below X-percent of the maximum
    // value to X-percent of the maximum value where X is
    // determined by the value of scale.
    double scale = 1.0/7.0;
    //First find the maximum value.
    double max = Double.MIN_VALUE;
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        if(plane[row][col] > max){
          max = plane[row][col];
        }//end if
      }//end inner loop
    }//end outer loop

    //Now set everything below X-percent of the maximum to
    // X-percent of the maximum value and slide
    // everything down to cause the new minimum to be
    // at 0.0
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        if(plane[row][col] < scale * max){
          plane[row][col] = scale * max;
        }//end if
        plane[row][col] -= scale * max;
      }//end inner loop
    }//end outer loop
```

```
    //Now scale the data so that the maximum value is 255.

    //First find the maximum value
    max = Double.MIN_VALUE;
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        if(plane[row][col] > max){
          max = plane[row][col];
        }//end if
      }//end inner loop
    }//end outer loop
    //Now scale the data.
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        plane[row][col] = plane[row][col] * 255.0/max;
      }//end inner loop
    }//end outer loop

  }//end normalize
  //-----------------------------------------------------//
}//end class ImgMod35a
```

Listing 17

Listing 18

```
/*File ForwardDCT02.java
Copyright 2006, R.G.Baldwin
Rev 01/19/06

THIS VERSION IS OPTIMIZED FOR USE WITH 8x8 IMAGES.

See ForwardDCT01 for a general purpose version that works
for images of different sizes.  This version will also
work with images of any size, but it is optimized for use
with 8x8-pixel images.

When transforming an 8x8-pixel image, this version uses an
8x8 cosine lookup table instead of calling the cos
function.  Otherwise, it calls the cosine function during
each iteration.  It is probably faster to use the lookup
table than it is to call the cosine function.

The static method named transform performs a forward
Discrete Cosine Transform (DCT) on an incoming series
and returns the DCT spectrum.

See http://en.wikipedia.org/wiki/Discrete_cosine_transform
#DCT-II and http://rkb.home.cern.ch/rkb/AN16pp/node61.html
for background on the DCT.

This formulation is from
http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/
coding/transform/dct.html
```

```
Incoming parameters are:
  double[] x - incoming real data
  double[] y - outgoing real data

Tested using J2SE 5.0 under WinXP.  Requires J2SE 5.0 or
later due to the use of static import of Math class.
**********************************************************/
import static java.lang.Math.*;

public class ForwardDCT02{

  //Enable the following statement to count and display
  // the number of computations.
  //static long callCount = 0;
  public static void transform(double[] x,double[] y){

  //The following values for the cosine table were obtained
  // by running the program with the call to the cos
  // function intact, printing the cosine of the argument,
  // capturing it, and then inserting the values here.
  // These are the cosine values used for transforming a
  // series containing 8 values.

  double[][] cosineTable = {
                            {1.0,
                            1.0,
                            1.0,
                            1.0,
                            1.0,
                            1.0,
                            1.0,
                            1.0},

                            {0.9807852804032304,
                            0.8314696123025452,
                            0.5555702330196023,
                            0.19509032201612833,
                            -0.1950903220161282,
                            -0.555570233019602,
                            -0.8314696123025453,
                            -0.9807852804032304},

                            {0.9238795325112867,
                            0.38268343236508984,
                            -0.3826834323650897,
                            -0.9238795325112867,
                            -0.9238795325112868,
                            -0.38268343236509034,
                            0.38268343236509,
                            0.9238795325112865},

                            {0.8314696123025452,
                            -0.1950903220161282,
                            -0.9807852804032304,
```

```
                            -0.5555702330196022,
                            0.5555702330196018,
                            0.9807852804032304,
                            0.19509032201612878,
                            -0.8314696123025451},

                            {0.7071067811865476,
                            -0.7071067811865475,
                            -0.7071067811865477,
                            0.7071067811865474,
                            0.7071067811865477,
                            -0.7071067811865467,
                            -0.7071067811865471,
                            0.7071067811865466},

                            {0.5555702330196023,
                            -0.9807852804032304,
                            0.1950903220161283,
                            0.8314696123025456,
                            -0.8314696123025451,
                            -0.19509032201612803,
                            0.9807852804032307,
                            -0.5555702330196015},

                            {0.38268343236508984,
                            -0.9238795325112868,
                            0.9238795325112865,
                            -0.3826834323650899,
                            -0.38268343236509056,
                            0.9238795325112867,
                            -0.9238795325112864,
                            0.38268343236508956},

                            {0.19509032201612833,
                            -0.5555702330196022,
                            0.8314696123025456,
                            -0.9807852804032307,
                            0.9807852804032304,
                            -0.831469612302545,
                            0.5555702330196015,
                            -0.19509032201612858}
                          };

    //Enable the following statement to count and display
    // the number of computations.
    //long count = 0;
    int N = x.length;
    if(N == 8){
      //Run optimized version by using cosine lookup table
      // instead of computing cosine values for each
      // iteration.
      //Outer loop interates on frequency values.
      for(int k=0; k < N;k++){
        double sum = 0.0;
        //Inner loop iterates on time-series values.
        for(int n=0; n < N; n++){
```

```
          double cosine = cosineTable[k][n];
          double product = x[n]*cosine;
          sum += product;
          //Enable the following statement to count and
          //display the number of computations.
          //count++;
        }//end inner loop

        double alpha;
        if(k == 0){
          alpha = 1.0/sqrt(2);
        }else{
          alpha = 1;
        }//end else
        y[k] = sum*alpha*sqrt(2.0/N);
      }//end outer loop
    }else{
      //Run regular version by computing cosine values for
      // each iteration.
      //Outer loop interates on frequency values.
      for(int k=0; k < N;k++){
        double sum = 0.0;
        //Inner loop iterates on time-series values.
        for(int n=0; n < N; n++){
          double arg = PI*k*(2.0*n+1)/(2*N);
          double cosine = cos(arg);
          double product = x[n]*cosine;
          sum += product;
          //Enable the following statement to count and
          //display the number of computations.
          //count++;
        }//end inner loop

        double alpha;
        if(k == 0){
          alpha = 1.0/sqrt(2);
        }else{
          alpha = 1;
        }//end else
        y[k] = sum*alpha*sqrt(2.0/N);
      }//end outer loop
    }//end else
    //Enable the following two statements to count and
    //display the number of computations.
    //callCount++;
    //System.out.println(callCount + " " + count + " "
    //                   + callCount*count);
  }//end transform method
  //-------------------------------------------------//
}//end class ForwardDCT02
```

Listing 18

Listing 19

```
/*File ImgMod35.java
Copyright 2005, R.G.Baldwin

This program performs a forward DCT on an image converting
the three color planes into spectral planes.  Then it
performs an inverse DCT on the three spectral planes
converting them back into image color planes.

Nothing is done to the spectral planes following the
forward DCT and before the inverse DCT.  However,
additional processing, such as requantization and
compression, followed by decompression could be inserted at
that point.

This is an update to ImgMod34.  This update breaks the
image down into blocks of 8x8-pixels and performs the
forward DCT on each block producing 8x8 blocks of spectral
coefficient data.  Then it performs an inverse DCT on each
of the 8x8 spectral blocks, reproducing the 8x8 pixel
blocks.  The composite of all the 8x8-pixel blocks
constitutes the total image.  Zero padding is applied to
the right and bottom of the image to force each
dimension of the image to be a multiple of 8 pixels.  This
padding is trimmed from the resulting spectral planes and
the resulting composite image before the processed
composite image is returned to the calling method.  This
process doesn't appear to have a detrimental impact on the
quality of the image at the edges.

For illustration purposes only, the method named
inverseXform8x8Plane contains a statement that can be
activated to cause the 8x8 block structure of the entire
process to become visible in the final image. This feature
should be disabled by default because it badly corrupts the
visual quality of the image when it is enabled.

Note that this version of the program is significantly
faster than the version named ImgMod034, which does not
break the image down into 8x8-pixel blocks, but rather does
the forward and inverse DCT on the entire image.

Among other things, this program uses an 8x8 cosine lookup
table instead of computing the cosine value every time it
is needed.  This is probably one factor in the speed
improvement.  Another factor is the simple fact that less
arithmetic is required to perform the transform when the
image is sub-divided into blocks and the transforms are
performed on the individual blocks instead of transforming
the entire image as a whole.

The class is designed to be driven by the class named
ImgMod02a.

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

```
java ImgMod02a ImgMod35 ImageFileName

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

When you click the Replot button, the process will be
repeated and the results will be re-displayed.  Because
there is no opportunity for user input after the program is
started, the Replot button is of little value to this
program.

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

ImgIntfc02.class
ImgMod02a.class
ImgMod35.class
InverseDCT02.class
ForwardDCT02.class

Tested using J2SE 5.0 and WinXP.  J2SE 5.0 or later is
required due to the use of static imports.
**********************************************************/
import java.awt.*;
import java.io.*;
import static java.lang.Math.*;

class ImgMod35 implements ImgIntfc02{

  //This method is required by ImgIntfc02.  It is called at
  // the beginning of the run and each time thereafter that
  // the user clicks the Replot button on the Frame
  // containing the images.
  public int[][][] processImg(int[][][] threeDPix,
                              int imgRows,
                              int imgCols){

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

    //Make a working copy of the 3D pixel array as type
    // double to avoid making permanent changes to the
    // original image data.  Also, all processing will be
    // performed as type double.
    double[][][] working3D = copyToDouble(threeDPix);

    //The following code can be enabled to set any of the
    // three colors to black, thus removing them from the
    // output.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
//        working3D[row][col][1] = 0;
//        working3D[row][col][2] = 0;
//        working3D[row][col][3] = 0;
```

```
      }//end inner loop
    }//end outer loop

    //Extract and do a forward DCT on the red color plane
    double[][] redPlane = extractPlane(working3D,1);
    //Expand the plane such that both dimensions are a
    // multiple of 8.
    redPlane = expandPlane(redPlane);
    //Do the forward DCT on the redPlane turning it into a
    // spectral plane where each individual spectrum stored
    // in the plane is an 8x8 square.  To see the
    // individual spectra, disable all but one color plane
    // and disable the inverse transforms that are
    // performed later.
    forwardXformPlane(redPlane);
    //Insert the spectral plane back into the 3D array.
    // This method also trims off any extra rows and
    // columns created by expanding the dimensions to a
    // multiple of 8.  Since those rows and columns
    // contain spectral data, the trimming could
    // conceivably cause problems with the quality of the
    // image at the right and bottom edges later.  If so, I
    // will need to save the spectral plane somewhere else
    // without trimming it.
    insertPlane(working3D,redPlane,1);

    //Extract and do a forward DCT on the green color
    // plane.
    double[][] greenPlane = extractPlane(working3D,2);
    greenPlane = expandPlane(greenPlane);
    forwardXformPlane(greenPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,greenPlane,2);

    //Extract and do a forward DCT on the blue color plane.
    double[][] bluePlane = extractPlane(working3D,3);
    bluePlane = expandPlane(bluePlane);
    forwardXformPlane(bluePlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,bluePlane,3);

    //All three color planes have now been transformed into
    // spectral planes where each spectral plane is
    // composed of a large number of contiguous 8x8
    // spectral data blocks.

    //Now transform the spectral planes back into image
    // color planes.
    //You can disable this section of code and disable two
    // of the three colors to get some idea as to what the
    // spectral planes look like.  If you don't disable two
    // of the colors, you will see a composite of all three
    // spectral planes, which may or may not be
    // informative.  At least, it shows the 8x8 block
    // structure of the spectral planes even if it is a
    // composite of three spectral planes.
```

```
    //Extract and do an inverse DCT on the red spectral
    // plane to produce a color image plane.
    redPlane = extractPlane(working3D,1);
    //Expand the plane such that both dimensions are a
    // multiple of 8.
    redPlane = expandPlane(redPlane);
    //Do the inverse DCT on the spectral plane producing a
    // color image plane.
    inverseXformPlane(redPlane);
    //Insert the color plane back into the 3D array. This
    // also trims off any extra rows and columns created by
    // expanding to a multiple of 8.
    insertPlane(working3D,redPlane,1);

    //Extract and do an inverse DCT on the green spectral
    // plane to produce a color image plane.
    greenPlane = extractPlane(working3D,2);
    greenPlane = expandPlane(greenPlane);
    inverseXformPlane(greenPlane);
    insertPlane(working3D,greenPlane,2);

    //Extract and do an inverse DCT on the blue spectral
    // plane to produce a color image plane.
    bluePlane = extractPlane(working3D,3);
    bluePlane = expandPlane(bluePlane);
    inverseXformPlane(bluePlane);
    insertPlane(working3D,bluePlane,3);

    //End of section that does the inverse transforms.

    //Convert the image color planes to type int and return
    // the array of pixel data to the calling method.
    output = copyToInt(working3D);
    //Return a reference to the output array.
    return output;

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

  //The purpose of this method is to extract a specified
  // row from a double 2D plane and to return it as a one-
  // dimensional array of type double.
  double[] extractRow(double[][] colorPlane,int row){

    int numCols = colorPlane[0].length;
    double[] output = new double[numCols];
    for(int col = 0;col < numCols;col++){
      output[col] = colorPlane[row][col];
    }//end outer loop
    return output;
  }//end extractRow
  //-------------------------------------------------//

  //The purpose of this method is to insert a specified
  // row of double data into a double 2D plane.
```

```
  void insertRow(double[][] colorPlane,
                 double[] theRow,
                 int row){
    int numCols = colorPlane[0].length;
    double[] output = new double[numCols];
    for(int col = 0;col < numCols;col++){
      colorPlane[row][col] = theRow[col];
    }//end outer loop
  }//end insertRow
  //-----------------------------------------------------//

  //The purpose of this method is to extract a specified
  // col from a double 2D plane and to return it as a one-
  // dimensional array of type double.
  double[] extractCol(double[][] colorPlane,int col){
    int numRows = colorPlane.length;
    double[] output = new double[numRows];
    for(int row = 0;row < numRows;row++){
      output[row] = colorPlane[row][col];
    }//end outer loop
    return output;
  }//end extractCol
  //-----------------------------------------------------//

  //The purpose of this method is to insert a specified
  // col of double data into a double 2D color plane.
  void insertCol(double[][] colorPlane,
                 double[] theCol,
                 int col){
    int numRows = colorPlane.length;
    double[] output = new double[numRows];
    for(int row = 0;row < numRows;row++){
      colorPlane[row][col] = theCol[row];
    }//end outer loop
  }//end insertCol
  //-----------------------------------------------------//

  //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.
  public double[][] extractPlane(
                              double[][][] threeDPixDouble,
                              int plane){

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

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

    //Copy the values from the specified plane to the
    // double array.
    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 extractPlane
  //-------------------------------------------------//

  //The purpose of this method is to insert a double 2D
  // plane into the double 3D array that represents an
  // image.  This method also trims off any extra rows and
  // columns in the double 2D plane.
  public void insertPlane(double[][][] threeDPixDouble,
                          double[][] colorPlane,
                          int plane){

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

    //Copy the values from the incoming color plane to the
    // specified plane in the 3D array.
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        threeDPixDouble[row][col][plane] =
                                      colorPlane[row][col];
      }//end loop on col
    }//end loop on row
  }//end insertPlane
  //-------------------------------------------------//

  //This method copies an int version of a 3D pixel array
  // to an new pixel array of type double.
  double[][][] copyToDouble(int[][][] threeDPix){
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    double[][][] new3D = new double[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] = threeDPix[row][col][0];
        new3D[row][col][1] = threeDPix[row][col][1];
        new3D[row][col][2] = threeDPix[row][col][2];
        new3D[row][col][3] = threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
  }//end copyToDouble
  //-------------------------------------------------//

  //This method copies a double version of a 3D pixel array
  // into a new pixel array of type int.
  int[][][] copyToInt(double[][][] threeDPixDouble){
    int imgRows = threeDPixDouble.length;
    int imgCols = threeDPixDouble[0].length;

    int[][][] new3D = new int[imgRows][imgCols][4];
```

```
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] =
                          (int)threeDPixDouble[row][col][0];
        new3D[row][col][1] =
                          (int)threeDPixDouble[row][col][1];
        new3D[row][col][2] =
                          (int)threeDPixDouble[row][col][2];
        new3D[row][col][3] =
                          (int)threeDPixDouble[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
  }//end copyToInt
  //-----------------------------------------------------//

  //The purpose of this method is to clip all negative
  // color values in a double color plane to a value of 0.
  void clipToZero(double[][] colorPlane){
    int numImgRows = colorPlane.length;
    int numImgCols = colorPlane[0].length;
    //Do the clip
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        if(colorPlane[row][col] < 0){
          colorPlane[row][col] = 0;
        }//end if
      }//end inner loop
    }//end outer loop
  }//end clipToZero
  //-----------------------------------------------------//

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

  //Expand a double 2D plane such that both dimensions are
  // a multiple of 8.
  double[][] expandPlane(double[][] plane){
    int rows = plane.length;
    int cols = plane[0].length;
    double[][] output;
```

```
    int rowsRem = rows%8;
    int colsRem = cols%8;

    if((rowsRem == 0) && (colsRem == 0)){
      //No expansion is required.
      return plane;
    }else{
      //An expansion is required.  Copy the data from the
      // incoming array to a new expanded array, leaving
      // pad values of 0.0 at the right and bottom edges.
      output = new double
              [rows + (8 - rowsRem)][cols + (8 - colsRem)];
      for(int row = 0;row < rows;row++){
        for(int col = 0;col < cols;col++){
          output[row][col] = plane[row][col];
        }//end inner loop
      }//end outer loop
      return output;
    }//end else
  }//end expandPlane
  //-------------------------------------------------//

  //Extracts and returns an 8x8 block from a double 2D
  // plane.
  double[][] get8x8Block(double[][] colorPlane,
                         int segRow,
                         int segCol){
    double[][] the8x8Block = new double[8][8];
    for(int bigRow = segRow * 8,smallRow = 0;
                                 bigRow < (segRow * 8 + 8);
                                      bigRow++,smallRow++){
      for(int bigCol = segCol * 8,smallCol = 0;
                                   bigCol < segCol * 8 + 8;
                                      bigCol++,smallCol++){
        the8x8Block[smallRow][smallCol] =
                                  colorPlane[bigRow][bigCol];
      }//end inner loop
    }//end outer loop
    return the8x8Block;
  }//end get8x8Block
  //-------------------------------------------------//

  //Inserts an 8x8 block into a double 2D plane
  void insert8x8Block(double[][] colorPlane,
                      double[][] the8x8Plane,
                      int segRow,
                      int segCol){
    for(int bigRow = segRow * 8,smallRow = 0;
                                 bigRow < (segRow * 8 + 8);
                                      bigRow++,smallRow++){
      for(int bigCol = segCol * 8,smallCol = 0;
                                   bigCol < segCol * 8 + 8;
                                      bigCol++,smallCol++){
        colorPlane[bigRow][bigCol] =
                             the8x8Plane[smallRow][smallCol];
      }//end inner loop
```

```
    }//end outer loop
  }//end insert8x8Block
  //-------------------------------------------------//

  //Breaks a double color plane down into 8x8-pixel blocks
  // and does a forward DCT xform on each block.  Assembles
  // the resulting spectral data blocks into a spectral
  // plane. Assumes that the dimensions of the color plane
  // are multiples of 8
  void forwardXformPlane(double[][] plane){
    int pixRows = plane.length;
    int pixCols = plane[0].length;
    //Loop on rows of 8x8 blocks
    for(int segRow = 0;segRow < pixRows/8;segRow++){
      //Loop on cols of 8x8 blocks
      for(int segCol = 0;segCol < pixCols/8;segCol++){
        double[][] the8x8Plane =
                          get8x8Block(plane,segRow,segCol);
        forwardXform8x8Block(the8x8Plane);
        insert8x8Block(plane,the8x8Plane,segRow,segCol);
      }//end inner loop
    }//end outer loop
  }//end forwardXformPlane
  //-------------------------------------------------//

  //This method does a forward DCT on an 8x8 block of
  // pixels from a color plane received as an incoming
  // parameter.
  void forwardXform8x8Block(double[][] the8x8Block){

    int imgRows = 8;
    int imgCols = 8;

    //Extract each row from the 8x8-pixel block and perform
    // a forward DCT on the row. Then insert the
    // transformed row back into the block.  At that point,
    // the row no longer contains color pixel data, but
    // has been transformed into a row of spectral data.
    for(int row = 0;row < imgRows;row++){
      double[] theRow = extractRow(the8x8Block,row);

      double[] theXform = new double[theRow.length];
      //Perform the forward transform.
      ForwardDCT02.transform(theRow,theXform);

      //Insert the transformed row back into the block.
      insertRow(the8x8Block,theXform,row);
    }//end for loop

    //The block now contains the results of doing the
    // horizontal DCT one row at a time.  The block no
    // longer contains color pixel data.  Rather, it
    // contains spectral data for the horizontal dimension
    // only.

    //Extract each column from the block and perform a
```

```
    // forward DCT on the column. Then insert the
    // transformed column back into the block.
    for(int col = 0;col < imgCols;col++){
      double[] theCol = extractCol(the8x8Block,col);

      double[] theXform = new double[theCol.length];
      ForwardDCT02.transform(theCol,theXform);

      insertCol(the8x8Block,theXform,col);
    }//end for loop

    //The 8x8 block has now been converted into an 8x8
    // block of spectral coefficient data.

  }//end forwardXform8x8Block
  //-------------------------------------------------//

  //Breaks a spectral plane down into 8x8 blocks, performs
  // an inverse DCT on each block, and inserts the
  // resulting 8x8 image blocks into a color plane. Assumes
  // that the dimensions of the plane are multiples of 8
  void inverseXformPlane(double[][] plane){
    int pixRows = plane.length;
    int pixCols = plane[0].length;
    //Loop on rows of 8x8 blocks
    for(int segRow = 0;segRow < pixRows/8;segRow++){
      //Loop on cols of 8x8 blocks
      for(int segCol = 0;segCol < pixCols/8;segCol++){
        double[][] the8x8Block =
                          get8x8Block(plane,segRow,segCol);
        inverseXform8x8Plane(the8x8Block);
        insert8x8Block(plane,the8x8Block,segRow,segCol);
      }//end inner loop
    }//end outer loop
  }//end inverseXformPlane
  //-------------------------------------------------//

  //This method performs an inverse DCT on an 8x8 spectral
  // data block.
  void inverseXform8x8Plane(double[][] the8x8Block){

    int imgRows = 8;
    int imgCols = 8;

    //Extract each col from the spectral data block and
    // perform an inverse DCT on the column. Then insert it
    // back into the block, which is being transformed into
    // a block of color pixel data.
    for(int col = 0;col < imgCols;col++){
      double[] theXform = extractCol(the8x8Block,col);

      double[] theCol = new double[theXform.length];
      //Now perform the inverse transform.
      InverseDCT02.transform(theXform,theCol);

      //Insert it back into the block.
```

```
      insertCol(the8x8Block,theCol,col);
    }//end for loop

    //At this point, an inverse DCT has been performed on
    // each column in the spectral data block, one column
    // at a time.

    //Extract each row from the block and perform an
    // inverse DCT on the row. Then insert it back into the
    // block.  The row now contains color pixel data.
    for(int row = 0;row < imgRows;row++){
      double[] theXform = extractRow(the8x8Block,row);

      double[] theRow = new double[theXform.length];
      //Now perform the inverse transform.
      InverseDCT02.transform(theXform,theRow);

      //Insert it back into the block.
      insertRow(the8x8Block,theRow,row);
    }//end for loop

    //For illustration purposes only, activate the
    // following statement to cause the 8x8 block
    // structure of the entire process to become visible
    // in the displayed image.  This causes each individual
    // block of pixel data to be a slightly different
    // color.
    //addRandom(the8x8Block);

    //At this point, the spectral data block has been
    // converted into an 8x8 block of pixel color data.
    // Ultimately it will be necessary to convert it to
    // 8-bit unsigned pixel color format in order to
    // display it.  Clip to zero and 255.
    clipToZero(the8x8Block);
    clipTo255(the8x8Block);
  }//end inverseXform8x8Plane
  //-------------------------------------------------//

  //The purpose of this method is to add the same random
  // color value to all of the values in an 8x8 block of
  // image color data to cause the block structure of the
  // process to be visible.  This capability is intended
  // to be used for illustration purposes only as it
  // badly corrupts the quality of the image.
  void addRandom(double[][] colorPlane){
    double bias = 100.0 * random();
    int numImgRows = colorPlane.length;
    int numImgCols = colorPlane[0].length;
    //Do the clip
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
          colorPlane[row][col] += bias;
      }//end inner loop
    }//end outer loop
  }//end addRandom
```

```
  //-----------------------------------------------------//
}//end class ImgMod35

Listing 19
```

Listing 20

```
/*File InverseDCT02.java
Copyright 2006, R.G.Baldwin
Rev 01/19/06

THIS VERSION IS OPTIMIZED FOR USE WITH 8x8 IMAGES.

See InverseDCT01 for a general purpose version that works
for images of different sizes.  This version will also
work with images of any size, but it is optimized for use
with 8x8-pixel images.

When transforming an 8x8-pixel image, this version uses an
8x8 cosine lookup table instead of calling the cos
function.  Otherwise, it calls the cosine function during
each iteration.  It is probably faster to use the lookup
table than it is to call the cosine function.

The static method named transform performs an inverse
Discreet Cosine Transform (DCT) on in incoming DCT
spectrum and returns the DCT time or image series.

See http://en.wikipedia.org/wiki/Discrete_cosine_transform
#DCT-II and http://rkb.home.cern.ch/rkb/AN16pp/node61.html
for background on the DCT.

This formulation is from
http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/
coding/transform/dct.html

Incoming parameters are:
  double[] y - incoming real data
  double[] x - outgoing real data

Tested using J2SE 5.0 under WinXP.  Requires J2SE 5.0 or
later due to the use of static import of Math class.
**********************************************************/
import static java.lang.Math.*;

public class InverseDCT02{

  public static void transform(double[] y,double[] x){

  //The following values for the cosine table were obtained
  // by running the program with the call to the cos
  // function intact, printing the cosine of the argument,
  // capturing it, and then inserting the values here.
  // These are the cosine values used for transforming a
  // series containing 8 values.
```

```
double[][] cosineTable = {
                          {1.0,
                          0.9807852804032304,
                          0.9238795325112867,
                          0.8314696123025452,
                          0.7071067811865476,
                          0.5555702330196023,
                          0.38268343236508984,
                          0.19509032201612833},

                          {1.0,
                          0.8314696123025452,
                          0.38268343236508984,
                          -0.1950903220161282,
                          -0.7071067811865475,
                          -0.9807852804032304,
                          -0.9238795325112868,
                          -0.5555702330196022},

                          {1.0,
                          0.5555702330196023,
                          -0.3826834323650897,
                          -0.9807852804032304,
                          -0.7071067811865477,
                          0.1950903220161283,
                          0.9238795325112865,
                          0.8314696123025456},

                          {1.0,
                          0.19509032201612833,
                          -0.9238795325112867,
                          -0.5555702330196022,
                          0.7071067811865474,
                          0.8314696123025456,
                          -0.3826834323650899,
                          -0.9807852804032307},

                          {1.0,
                          -0.1950903220161282,
                          -0.9238795325112868,
                          0.5555702330196018,
                          0.7071067811865477,
                          -0.8314696123025451,
                          -0.38268343236509056,
                          0.9807852804032304},

                          {1.0,
                          -0.555570233019602,
                          -0.38268343236509034,
                          0.9807852804032304,
                          -0.7071067811865467,
                          -0.1950903220161283,
                          0.9238795325112867,
                          -0.831469612302545},
```

```
                              {1.0,
                              -0.8314696123025453,
                              0.38268343236509,
                              0.19509032201612878,
                              -0.7071067811865471,
                              0.9807852804032307,
                              -0.9238795325112864,
                              0.5555702330196015},

                              {1.0,
                              -0.9807852804032304,
                              0.9238795325112865,
                              -0.8314696123025451,
                              0.7071067811865466,
                              -0.5555702330196015,
                              0.38268343236508956,
                              -0.19509032201612858}
                             };

  int N = y.length;
  if(N == 8){
    //Run optimized version by using cosine lookup table
    // instead of computing cosine values for each
    // iteration.
    //Outer loop interates on time values.
    for(int n=0; n < N;n++){
      double sum = 0.0;
      //Inner loop iterates on frequency values
      for(int k=0; k < N; k++){
        double cosine = cosineTable[n][k];
        double product = y[k]*cosine;
        double alpha;
        if(k == 0){
          alpha = 1.0/sqrt(2);
        }else{
          alpha = 1;
        }//end else

        sum += alpha * product;
      }//end inner loop

      x[n] = sum * sqrt(2.0/N);
    }//end outer loop
  }else{
    //Run regular version by computing cosine values for
    // each iteration.
    //Outer loop interates on time values.
    for(int n=0; n < N;n++){
      double sum = 0.0;
      //Inner loop iterates on frequency values
      for(int k=0; k < N; k++){
        double arg = PI*k*(2.0*n+1)/(2*N);
        double cosine = cos(arg);
        double product = y[k]*cosine;
        double alpha;
        if(k == 0){
```

```
            alpha = 1.0/sqrt(2);
          }else{
            alpha = 1;
          }//end else

          sum += alpha * product;

        }//end inner loop

        x[n] = sum * sqrt(2.0/N);

      }//end outer loop
    }//end else
  }//end transform method
  //-------------------------------------------------//
}//end class InverseDCT02
```

Listing 20

Copyright 2006, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

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 data image compression two-dimensional Discrete Cosine Transform, DCT Huffman Lempel Ziv

-end-