Understanding the 2D Discrete Cosine Transform in Java

Learn how to use the forward two-dimensional Discrete Cosine Transform (2D-DCT) to compute and display the wave-number spectrum of an image. Also learn how to apply the inverse 2D-DCT to the spectral data to reconstruct and display a replica of the original image.

Published: September 5, 2006
By Richard G. Baldwin

Java Programming Notes # 2446

Preface

This lesson is one in a series designed to teach you about the inner workings of data and image compression. The first lesson in the series was Understanding the Lempel-Ziv Data Compression Algorithm in Java. The previous lesson was Understanding the Discrete Cosine Transform in Java.

The previous lesson dealt with one-dimensional Discrete Cosine Transforms. This lesson is the first part of a two-part lesson on 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 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, which is the primary topic of this lesson. In this lesson, I will teach you how to use the *forward* 2D-DCT to compute and display the wave-number spectrum of an image. I will also teach you how to apply the *inverse* 2D-DCT to the spectral data to reconstruct a replica of the original image.

A third central component of JPEG 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 Discrete Cosine Transform, 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 also recommend that you also study the lessons referred to in the References section.

General Background Information

One of the main reasons that we are studying 2D Discrete Cosine Transforms *(2D-DCT)* is to further our understanding of JPEG image compression. The 2D-DCT is a central component in JPEG.

An analogy

I will begin my discussion with an analogy to a portion of the JPEG image compression algorithm.

Assume that you have been charged with the task of transporting all the water in a twenty-gallon tank from one building to a twenty-gallon tank in another building approximately three miles away. Unfortunately, you don't have any watertight containers in which to transport the water. All that is available for use in transporting the water is a sack made out of cloth. If you pour water into the sack, it simply runs through and out onto the floor.

How can you accomplish your assigned task?

Fortunately, the tank is mounted on small wheels that allow you to roll it around inside the building *(but not outside the building)*. You search the building, and are happy to find a large walk-in freezer with a doorway large enough to accommodate the tank of water.

Transform the water into ice

So, you roll the tank into the freezer. When all of the water has frozen into ice, you use an ice pick that you found nearby and you chop the ice up into pieces. You fill your sack with ice and run as fast as you can to the other building where you empty the sack into the other tank. You keep making trips from one building to the next until you have transported all of the ice from the original tank to the tank in the other building. You let the ice melt in the tank in the other building, and you have accomplished your task.

Is this a lossless process?

Unfortunately, a small quantity of the ice melts during the trips to transport it from one building to the other, so you end up with a little less than twenty-gallons of water in the second tank. *(This is not a lossless process.)*

Transformation is the key to success

What you have done is to transform the state of the water from one form to another to make it possible to transport it in a leaky cloth sack.

How does this apply to JPEG?

JPEG image compression does something similar, but not for exactly the same reasons. When an image is compressed using JPEG, the form of the image that is stored, and possibly transported from one machine to another, is not the form that you are probably accustomed to seeing. Rather, the information that constitutes the image is transformed from the image or space domain into the frequency or wave-number domain. The information is compressed, stored, and transported in the frequency domain. Later on, the information is transformed back into the image domain for presentation to a human consumer.

(Image information in the wave-number domain is typically not very useful to most human consumers of that information. See the bottom panel of Figure 9 for an example of image information in the frequency domain.)

The transformation to and from the frequency domain is accomplished using a 2D-DCT.

Image information looks totally different

As you will see later, the information that constitutes the image looks totally different when in the frequency domain than when it is in the image domain. *(Once again, see Figure 9 for an example.)*

An *almost lossless* process

If the image were simply transformed from the image domain into the frequency domain and back into the image domain, the process would be almost lossless.

(There would probably be a small amount of image distortion as a result of tiny errors resulting from computational inaccuracy.)

Lossless compression is not the primary objective

Generally speaking, however, the primary objective in JPEG image compression is usually not to implement a lossless process. Rather, the primary objective is to compress the image to make it smaller to store and to transport. A certain amount of distortion in the reconstructed image is usually considered tolerable.

(On the other hand, the earlier lessons entitled Understanding the Lempel-Ziv Data Compression Algorithm in Java and Understanding the Huffman Data Compression Algorithm in Java introduced you to two lossless data compression algorithms.)

The Lempel-Ziv algorithm

LZ77 is the name commonly given to a lossless data compression algorithm published in papers by Abraham Lempel and Jacob Ziv in 1977. LZ77 is a *lossless* compression data algorithm.

What is a lossless data compression algorithm?

If you compress a document using the LZ77 algorithm, and then decompress the compressed version, the result will be an exact copy of the original document.

(Not all data compression algorithms are lossless. The JPEG image compression algorithm, for example, does not produce an exact copy of an image that has been compressed using the algorithm.)

Dictionary and/or entropy encoding algorithms

LZ77 is known as a *dictionary* encoding algorithm, as opposed for example to the Huffman encoding algorithm, which is a statistical or *entropy* encoding algorithm.

Compression in the LZ77 algorithm is based on the notion that strings of characters *(words, phrases, etc.)* occur repeatedly in the message being compressed. Compression with the Huffman encoding algorithm is based on the probability of occurrence of individual characters in the message.

Not very effective for image data

Apparently neither of these compression algorithms is particularly effective when applied directly to the pixel color data that makes up an image. However, it has been determined that if an image is first transformed into the frequency domain, a form of frequency filtering can be applied to the frequency-domain data without having a serious adverse impact on the quality of the image when it is reconstructed.

What kind of frequency filtering?

The frequency filtering involves re-quantizing the frequency-domain data values at the higher wave-number frequencies. Re-quantization makes the frequency-domain data much more susceptible to the benefits of entropy encoding.

> *(Apparently it has been determined that substantial corruption of the higher wave-number spectral data can be tolerated without seriously damaging the visual quality of the reconstructed image.)*

This is where the major compression benefits of JPEG image compression occur. This is also what prevents the reconstructed image from being a nearly exact match for the original image, thus causing JPEG to be a *lossy* image compression algorithm.

A brief look at JPEG image compression

That has been a brief look into the overall process of JPEG image compression so that you will know where the 2D-DCT fits into the process. I will get into the details in a future lesson. For now, the objective is to isolate and to understand the 2D-DCT portion of the process.

An *(almost lossless)* example

Before getting into a detailed discussion of the 2D-DCT, I want to show you an *(almost)* lossless example along with some backup information. The bottom image in Figure 1 shows the results of performing a forward 2D-DCT on the top image and then turning around and performing an inverse 2D-DCT on the spectrum that was produced by the forward transform.

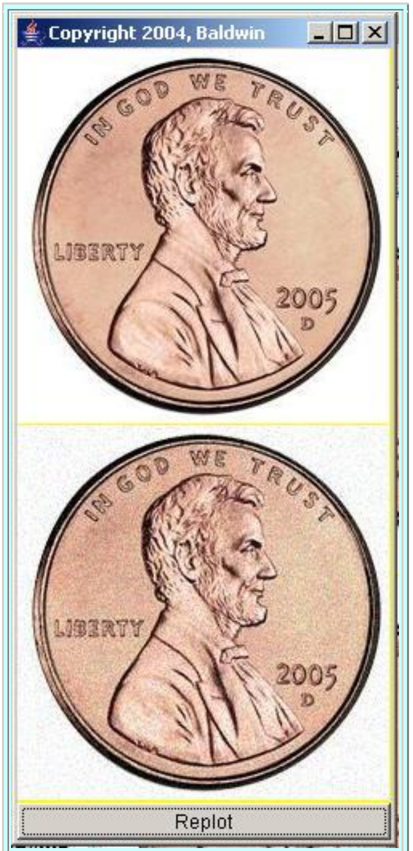

Figure 1

Re-quantization was applied

However, to partially simulate the behavior of the JPEG image compression algorithm, the spectral data was uniformly re-quantized so that it could be stored in an eleven-bit twos complement integer format *(-1024 to +1023).*

> *(Note that this is not the type of frequency-dependent re-quantization that is performed in the JPEG algorithm for the purpose of data compression.)*

Otherwise, nothing was done to compress, or otherwise corrupt the spectral data prior to transforming that data back into the image domain. Therefore, the output image should be almost as good as the input image, with the only deterioration being the result of the re-quantization noise and arithmetic errors in the forward and inverse transforms.

Quantization noise is visible

It looks to me like the background areas, such as the large white areas in the image are grainier on the bottom image than the top image. This graininess did not exist in my original version which treated the spectral data strictly in **double** format. The graininess became evident at the point in time that I inserted the re-quantization of the spectral data to make it fit into eleven bits.

More difficult than Discrete Fourier Transform

Others may disagree, but in my opinion, it is more difficult to comprehend the Discrete ***Cosine*** Transform process than it is to comprehend the Discrete ***Fourier*** Transform process.

> *(See the earlier lesson entitled Fun with Java, How and Why Spectral Analysis Works, for an introduction to the Discrete Fourier Transform.)*

To paraphrase a popular comedian named Flip Wilson from my younger days, when you work with the DCT, *"What you see is not what you get."*

DFT is linear with superposition

With the DFT, which is a linear transform for which superposition applies, many of us with Digital Signal Processing *(DSP)* experience can look at a simple 2D space-domain function and have a pretty good idea what the wave-number spectrum for that function will look like. We do that by breaking the space-domain function down into familiar components, transforming the familiar components in our heads, and then reconstructing the individual transforms in our heads to produce an idea of the wave-number spectrum.

More difficult with the DCT

This is much more difficult to do with the 2D-DCT. The reason is that the DCT is not applied directly to the given space-domain function. Rather, the given space-domain function is implicitly expanded through mirror-imaging to produce a new space-domain function. Then, to a very close approximation, a Discrete Fourier Transform is applied to the new expanded space-domain function. Thus, there is an added degree of complexity involving the implicit modification of the original space-domain function. I will illustrate this by providing some comparisons between the Discrete Fourier Transform and the Discrete Cosine Transform.

A simple diagonal line

Consider first the case of a simple diagonal line in the space domain as shown in the left panel of Figure 2.

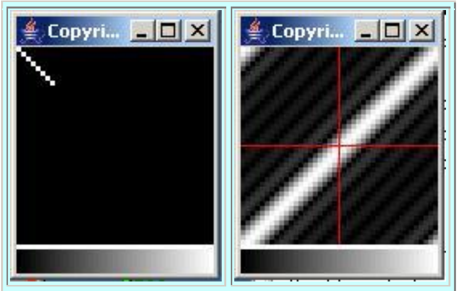

Figure 2

(This example was resurrected from Figure 9 in the earlier lesson entitled 2D Fourier Transforms using Java, Part 2.)

I explained this example in the earlier lesson, so I won't repeat that explanation here. Suffice it at this point to say that the left panel in Figure 2 shows a diagonal line in the space domain. The right panel shows the wave-number spectrum for that line produced by performing a 2D Discrete Fourier Transform *(2D-DFT)* on the image in the left panel.

The important attribute

Without getting into the details, the important thing to note is that the lines in the Fourier wave-number spectrum are perpendicular to the line in the space domain.

A similar 2D-DCT example

On the other hand, the top panel in Figure 3 shows an image with a similar diagonal line. The bottom panel in Figure 3 shows the result of performing a 2D-DCT on the image. *(We might refer to this as the DCT spectrum.)*

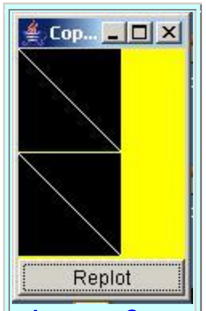

Figure 3

(Note that the entire bottom panel in Figure 3 corresponds only to the upper-left quadrant in the right panel in Figure 2. In other words, the spectrum shown in Figure 2 extends from a wave-number of zero at the upper left corner to a wave-number corresponding to the sampling frequency in space on the right and the bottom. On the other hand, the bottom panel in Figure 3 extends from a wave-number of zero at the upper left corner to the Nyquist folding frequency at the right and bottom.)

Something looks very wrong here!

Something looks wrong when we compare the DCT spectrum in Figure 3 with the Fourier spectrum in Figure 2. The line in the DCT spectrum is not perpendicular to the line in the space domain. Rather, it is parallel to the line in the space domain.

(It took me quite a lot of head scratching to conclude that this is probably correct. I believe that I finally understand the reason for the difference and I will share that knowledge with you.)

And the problem is ...

The problem is that there is no such thing as a single diagonal line in a DCT. True, we started with the single diagonal line shown in the top panel of Figure 3. However, the DCT process implicitly quadrupled the size of our sample. *(The size was doubled in both dimensions.)*

For example, the DCT process created a mirror image of the original area across its upper boundary producing an area twice as tall as what we started with.

Then it produced a mirror image of the entire double-high area across the left boundary. This resulted in a new area four times as large as the original area with the original line being reflected into the new areas in a mirror-image fashion.

Then it effectively performed a DFT on the new surface containing the original line plus the mirror images of the original line.

What does the expanded area look like?

If you draw that out on paper, you will see that the image that was actually transformed using the DFT contained two lines, each twice as long as the original and arranged in a cross or X with the center of the cross at the original origin. Each of the legs on the cross was on a diagonal.

A similar 2D-DFT

Figure 4 shows the results of applying a 2D-DFT to two lines similarly arranged to form a cross.

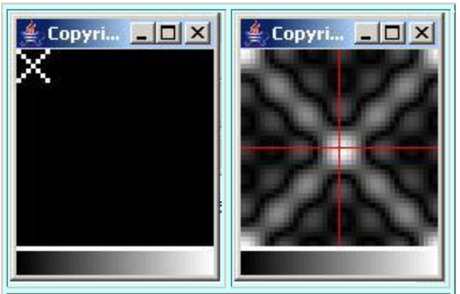

Figure 4

As before, the image in the space domain is shown in the left panel of Figure 4, and the Fourier spectrum of that image is shown in the right panel.

Taking into account that the bottom panel in Figure 3 corresponds to only the upper-left quadrant in the Fourier spectrum in Figure 4, the two look remarkably similar.

> *(Note that the plotting approach used in Figure 4 shows a lot more detail than the plotting approach used in Figure 3.)*

A diagonal line in both cases

Both spectra show a diagonal line extending from the origin at a wave-number of zero down to the Nyquist folding frequency at the opposite corner.

I believe that the reason for the spectral line in Figure 3 being parallel to the time-domain line instead of being perpendicular to that line is that the 2D-DCT shown in Figure 3 actually performed a 2D-DFT on two intersecting time-domain lines that form a cross.

A vertical line

The bottom image in Figure 5 shows the results of performing a 2D-DCT on the top image.

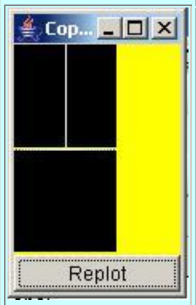

Figure 5

At first glance, it appears that there is no output in the bottom image. However, if you look very carefully, you will see a broken line at the top of the bottom image next to the area that separates the two images.

Does this make sense?

Once again, the top image has been implicitly expanded to a size that is four times as large as that shown. The expansion is accomplished by creating mirror images as described earlier. As a result, the input image no longer contains a single vertical line. Rather, it contains two parallel vertical lines and a DFT is performed on those parallel vertical lines.

A similar DFT example

The right panel in Figure 6 shows the results of performing an ordinary 2D-DFT on the pair of parallel vertical lines in the left panel.

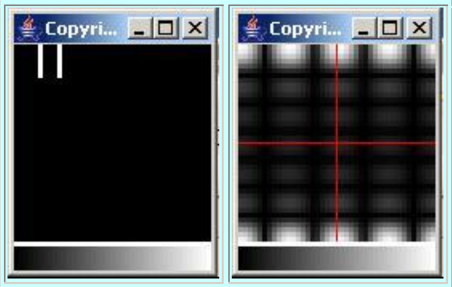

Figure 6

Considering the Fourier spectrum in the upper-left quadrant of the right panel, we see that the output consists mainly of a broken line along the top of the spectrum, beginning at a wave-number of zero and extending to the right all the way to the sampling frequency *(twice the distance shown in Figure 5)*.

So, it does make sense to have the DCT spectrum shown in Figure 5 consist mostly of a broken line along the top edge of the DCT spectrum from a wave number of zero to the Nyquist folding frequency at the right edge.

A horizontal line

The bottom image in Figure 7 shows the results of performing a 2D-DCT on the top image containing the single horizontal line.

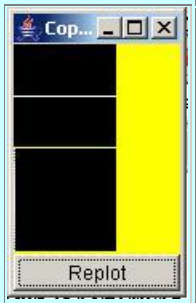

Figure 7

As before, because of the implicit expansion of the input image through the creation of mirror images, the mage that actually got transformed consisted of two parallel horizontal lines.

Also as before, the output is difficult to see. If you look very closely, you will see a vertical broken line along the left edge of the bottom image. I could go through the same steps as before, performing a 2D-DCT on a pair of parallel horizontal lines to show that we should expect the output to be a broken vertical line along the left edge, beginning at a wave number of zero and extending downward to the Nyquist folding frequency at the bottom edge.

So what?

By now you are probably wondering why I am going to all of this trouble to convince you of what you should expect to see. For that, I am going to give you a preview of the next lesson.

A preview

In the next lesson, you will learn that rather than to perform the 2D-DCT on the entire image, the JPEG image compression algorithm sub-divides the image into 8x8 blocks and performs the 2D-DCT on each block independently. At this point, I won't go into what the algorithm does with the spectra produced through that process, but I will show you a preview of what the spectra can look like.

Spectrum of a sub-divided image

The bottom panel in Figure 8 shows the results of performing a separate 2D-DCT on several hundred contiguous 8x8-pixel blocks that make up the image shown in the top panel.

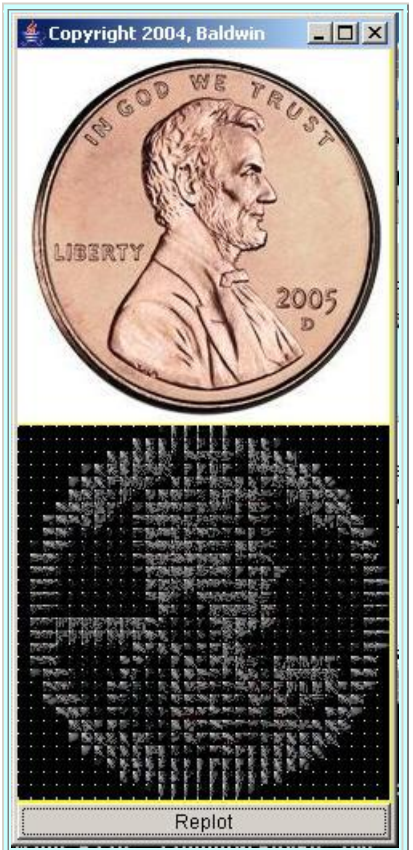

Figure 8

Each spectrum represents an 8x8-pixel image

When performed in this fashion, each spectrum represents only one 8x8-pixel portion of the original image. In most cases, the upper-left corner of each spectrum, *(representing a wave-number of zero)*, is identified by a bright dot. The spectral values for higher-frequency wave numbers appear to the right and down from that dot.

The constant-color areas

In those areas of the image where there are little or no color details, *(such as in the white portion of the image or the portion behind President Lincoln's head)*, each spectrum consists of a single bright dot only. This is because the spectrum consists of a zero-frequency component only with a significant lack of energy at other frequencies.

Areas with lots of color details

In areas where there is a lot of detail, such as in President Lincoln's hair for example, the 8x8 spectral blocks tends to be gray, indicating lots of energy at the higher frequencies.

Some particularly interesting areas

There are several areas that I find particularly interesting. For example, on the left and right edges of the penny where the circular edge is tangent to a vertical line, you can see a tendency for the spectra to exhibit strong horizontal lines. This agrees with the results shown earlier in Figure 5.

Vertical spectral lines

At the top and the bottom of the penny where the circular edge is tangent to a horizontal line, there is a tendency for the spectra to exhibit strong vertical lines. This agrees with the results shown earlier in Figure 7.

Diagonal spectral lines

At the four edges of the penny where the slope is tangent to a line that is at an angle of forty-five degrees to the horizontal *(northeast, southeast, southwest, and northwest)* there is a tendency for the spectra to exhibit strong diagonal lines sloping down and to the right.

> *(Note that the slope of the spectral lines is always down and to the right regardless of the slope of the tangential lines.)*

We also see this behavior at the lower-left edge of President Lincoln's beard near his ear, and at the edge of his collar where the lines in the image tend to be diagonal. In those areas also, there is a tendency for the spectra to exhibit strong diagonal lines sloping down and to the right.

This agrees with the results shown earlier in Figure 3.

I will discuss this in much more detail in the next lesson where I address the procedure of sub-dividing the image into 8x8-pixel blocks.

Now back to the main topic of this lesson

For the remainder of this lesson I will concentrate on applying the 2D-DCT to the entire image rather than applying the 2D-DCT to a sub-divided image. I will get back to the sub-divided image in Part 2 of this lesson, which I will publish in a few weeks.

The bottom panel in Figure 9 shows the 2D-DCT spectrum computed for the entire image in the top panel of Figure 9.

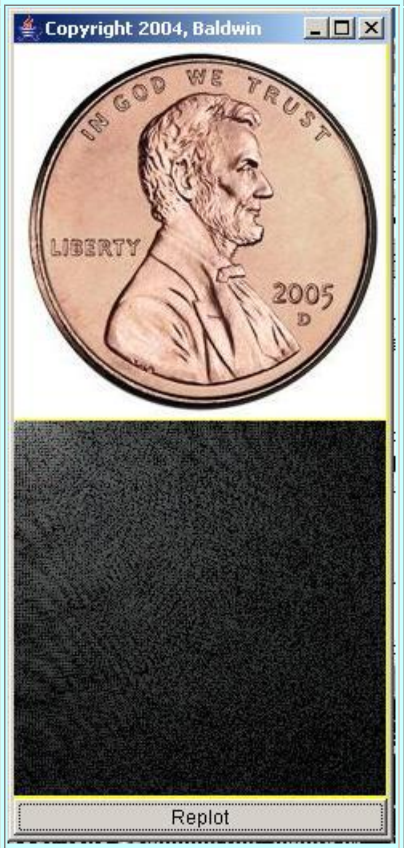

Figure 9

Not much to look at

Unlike the case in Figure 8, there is nothing in the spectrum in Figure 9 that would lead an ordinary human to suspect that this spectrum represents an image of a United States penny. Rest assured, however, that the spectrum shown in Figure 9 does contain all of the information necessary to reconstruct the original image. The bottom image in Figure 1 was produced by performing an inverse 2D-DCT on this spectral data.

Two representations of the same information

Thus, the top and bottom panels in Figure 9 contain the same information. They simply contain that information in different forms. The form in the top panel of Figure 9 is the one that we humans are most accustomed to seeing. We might think of that as the water in my earlier analogy. The form in the bottom panel of Figure 9 might be thought of as the ice in my earlier analogy. As it turns out, it is very easy to transform that information back and forth between the two forms.

I will explain two programs in the remainder of this lesson that illustrate how to transform the data back and forth between the two forms.

Preview

I am going to present and explain two different, but similar programs in this part of this lesson. *(I will present a couple more programs in Part 2 later when I publish it.)* The first program, named **ImgMod34a** performs a forward 2D-DCT on an image, and then displays the original image along with the DCT spectrum of that image *(as shown in Figure 9)*.

The second program named **ImgMod34** also performs a forward 2D-DCT on an image. However, rather than stopping at that point and displaying the DCT spectrum, this program re-quantizes the spectral data to eleven bits, and then performs an inverse 2D-DCT on the re-quantized spectral data to produce and display a replica of the original image *(as shown in Figure 1)*.

Discussion and Sample Code

The program named ImgMod34a

I will discuss this program in fragments. A complete listing of the program is shown in Listing 23.

The purpose of this program is to compute and display the wave-number spectrum of an image using a 2D-DCT.

A forward transform

This program performs a forward 2D-DCT on each color plane belonging to an image producing a wave-number spectrum that describes each color plane in the image. Then it converts the wave-number spectrum to log base 10 to preserve the dynamic range of the display and normalizes the results to cover the range from 0 to 255. This makes the results suitable for being displayed as an image.

Display the wave-number spectrum

Then the program returns the wave-number spectrum for each color plane in an image format.

> *(When I refer to an image format here, I am speaking of the classical image format consisting of an array of pixels wherein each pixel contains different contributions the colors red, green, and blue. This is accomplished through three color planes where each plane is a rectangular surface consisting of elevation values between 0 and 255 inclusive.)*

When displayed as an image, the visual result is the composite of the normalized wave number spectra of all three color planes being displayed.

The capability to isolate specific colors

The program provides the capability to enable statements that will effectively eliminate one, two, or all three of the color planes from the computation and the resulting display. *(This requires modification of the source code and recompilation of the program.)*

Runs under control of ImgMod02a

The program is designed to run under the control of the class named **ImgMod02a**. Enter the following at the command line to run this program:

```
java ImgMod02a ImgMod34a ImageFileName
```

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

Class files required

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

- ImgMod34a.class
- ImgIntfc02.class
- ImgMod02a.class
- ForwardDCT01.class

The source code for **ImgMod34a** is presented in Listing 23. The source code for **ForwardDCT01** was developed and explained in the previous lesson entitled Understanding the Discrete Cosine Transform in Java.

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.

Program testing

The program was tested using J2SE 5.0 and WinXP. J2SE 5.0 or later is required due to the use of static imports.

Source code for the ImgMod34a class

The source code for the **ImgMod34a** class and the method named **processImg** begins in Listing 1.

```
class ImgMod34a 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);
```

Listing 1

Need to understand ImgMod02a

In order to understand this program, you will need to understand how it interacts with the class named **ImgMod02a**, as described in the earlier lessons entitled Processing Image Pixels using Java, Getting Started and Processing Image Pixels Using Java: Controlling Contrast and Brightness. Once you understand that interaction, the code in Listing 1 should be straightforward.

One color plane at a time

The code in Listing 2 is provided to make it easy for you to experiment with only one color plane at time.

```

    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 2

To experiment with only one color plane, enable two of the statements in Listing 2, causing the color values for those two planes to be set to zero. Following that, the program output will contain a contribution only from the color plane corresponding to the statement that you did not enable.

Process the red plane

Listing 3 extracts the red color plane from the 3D pixel image array, processes it, and inserts it back into the 3D pixel image array in the form of a wave-number spectrum.

```
    //Extract and process the red plane
    double[][] redPlane =
extractPlane(working3D,1);
    processPlane(redPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,redPlane,1);
```

Listing 3

The code in the methods named **extractPlane** and **insertPlane** is straightforward and shouldn't require an explanation. You can view the source code for those methods in Listing 23.

The processPlane method

The **processPlane** method, on the other hand, is the workhorse of this entire program and does deserve a thorough explanation. Therefore, I will set the **processImg** method aside for awhile and explain the **processPlane** method.

Perform a forward 2D-DCT

This method processes a color plane received as an incoming parameter. First it performs a forward 2D-DCT on the color plane producing the DCT spectrum for the plane.

Convert to log base 10 and normalize

Then it normalizes the spectral values in the plane to make them compatible with being displayed as an image. First it converts the spectral data to log base 10 in order to preserve the dynamic range of the display system. Then it causes the logarithmic spectral data to fall in the range 0 to 255 which is a requirement for being displayed as an image.

A separable transform

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 single color plane of a 2D image, you can begin by performing a *one-dimensional* DCT on each row of the color plane, creating a new 2D structure where each row of the new structure contains the DCT of the corresponding row of the color plane. 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 image.

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 can 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.

Code for the processPlane method

The **processPlane** method begins in Listing 4.

```
  void processPlane(double[][] colorPlane){

    int imgRows = colorPlane.length;
    int imgCols = colorPlane[0].length;
```

Listing 4

The code in Listing 4 determines the number of rows and the number of columns in the 2D plane to be processed.

Transform one row at a time

Listing 5 shows the beginning of a **for** loop that:

- Extracts each row of image data from the color plane.
- Performs a forward one-dimensional DCT on the row.
- Inserts the spectral data for that row back into the corresponding row of the color plane, thereby using the color plane array to store the spectral data for the row.

```
    for(int row = 0;row < imgRows;row++){
      double[] theRow =
extractRow(colorPlane,row);
```

Listing 5

The code in the **extractRow** method is straightforward and shouldn't require a further explanation. You can view the **extractRow** method in its entirety in Listing 23.

Perform the one-dimensional Discrete Cosine Transform on the row

Listing 6 invokes the static **transform** method of the **ForwardDCT01** class to perform the one-dimensional DCT on the row of image data.

```
      double[] theXform = new
double[theRow.length];
      ForwardDCT01.transform(theRow,theXform);
```

Listing 6

Assuming that you have studied the earlier lesson entitled Understanding the Discrete Cosine Transform in Java, there is nothing in Listing 6 that should require a further explanation.

The results of the transform are temporarily stored in the one-dimensional array referred to by **theXform** before being inserted back into the corresponding row of the color plane.

Insert the spectral data into the color plane

Listing 7 invokes the method named **insertRow** to insert the spectral data back into the corresponding row in the color plane. From this point forward, that row contains spectral data and not image color data.

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

Listing 7

You can view the method named **insertRow** in its entirety in Listing 23.

Listing 7 also signals the end of the **for** loop, operating on rows that began in Listing 5.

Transform the column data

Listing 8 shows the complete **for** loop that:

- Extracts each column now containing spectral data from the color plane.
- Performs a forward one-dimensional DCT on the column.
- Inserts the spectral data for that column back into the corresponding column of the color plane, thereby using the color plane array to store the spectral data for the column.

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

```
      double[] theXform = new
double[theCol.length];
      ForwardDCT01.transform(theCol,theXform);

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

Listing 8

The color plane now contains spectral data

When the **for** loop in Listing 8 terminates, the 2D array referred to as **colorPlane** no longer contains image color data. Instead, it now contains the results of the 2D-DCT of the original image data for that color plane. In other words the data in the 2D array has now been transformed from image or space-domain data to frequency or wave-number spectral data.

Problems with displaying the results

The objective of this program is to display the 2D spectral data in the form of a standard image. This presents some problems that are not new to this program. In particular, the results of the 2D-DCT contain both positive and negative values. Furthermore, the magnitude of some of those values *(particularly at or near a wave-number of zero)* can be quite large.

Standard image data, on the other hand must be unsigned data in the range from 0 through 255 inclusive. This almost always presents problems involving the methodology for converting the bipolar data to that value range. The methodology that I developed for this program isn't perfect, but it seems to work pretty well.

Normalize the spectral data

Listing 9 invokes the **normalize** method to prepare the 2D spectral data for being displayed as an image.

```
    normalize(colorPlane);
  }//end processPlane
```

Listing 9

Listing 9 also signals the end of the **processPlane** method.

The normalize method

Before returning to the discussion of the **processImg** method *(last seen in Listing 3)* I will explain the **normalize** method.

This method is fairly complex and implements the results of several decisions that I had to make regarding the best way to display the spectral data.

This method normalizes the data in a 2D array containing data of type **double** to make it compatible with being displayed as an image plane.

Eliminate negative values

Normally, when viewing standard Fourier spectral data, unless we are specifically interested in phase angles, we usually compute and view the *amplitude spectrum*. The amplitude spectrum is computed by computing the square root of the sum of the squares of the real and imaginary parts of each complex spectral value *(the length of the hypotenuse of a right triangle formed by the real and imaginary vectors)*.

For the case of the DCT, there are no imaginary values. Therefore, the same results can be achieved simply by changing the sign on all negative spectral values making them positive instead of negative. The **normalize** method begins by converting all negative spectral values to positive values.

Coping with a limited dynamic range for the display

The dynamic range of the 2D-DCT spectral data is very large. The spectral values at and near the zero wave-number origin are much larger than values elsewhere in the spectrum.

Seven shades of gray

I learned at one point in my digital signal processing *(DSP)* career that a typical human can only distinguish between seven levels of gray with white and black representing two of those levels. Thus, a gray scale 2D display of the type shown in the bottom panel of Figure 9 has a very limited dynamic range.

A log base 10 transform

One common way to take better advantage of the available dynamic range of a display system when viewing spectral data is to convert the spectral data to decibels. This involves transforming the spectral values through a log base 10 transform and applying specific a specific scale factor to the resulting values. Although this program doesn't apply the specific scale factor required to qualify as a decibel scale, this program does apply a log base 10 transform to all of the spectral values. This results in a set of spectral values requiring less dynamic range in the display.

Can produce negative values

However, this transformation can result in negative values for very low spectral values; so once again, I am forced to deal with negative values. In this program the negative logarithmic values are simply set to zero.

Making the data fit between 0 and 255

This still leaves us with the problem of how to squeeze the logarithmic data into the value range from 0 to 255 inclusive to satisfy the fundamental image pixel value requirement.

One approach would be to scale all of the values such that the largest value becomes 255 and the smallest value becomes 0. I tried this, but even with the log transform, the values at and near the origin still overwhelmed the remaining portions of the spectrum, so this wasn't very satisfactory.

Simply discard the lower-level values

What I ended up doing was to simply discard all values below X-percent of the maximum by artificially setting all of those values to X-percent of the maximum. This created a floor at X-percent of the maximum. Then I shifted the floor down to a value of zero *(black)*, and scaled the resulting data so that the maximum value ended up at 255 *(bright red, bright green, or bright blue)*.

What should the level of the floor be?

Harking back to my *"seven levels of gray"* rule, I decided to put the floor at one-seventh of the maximum value. That is the value for the floor that produced the spectral display in the bottom panel of Figure 9.

Raising the floor so as to make it closer to the maximum would cause the display in Figure 9 to become more sparse with more black area *(a smaller percentage of the actual spectral data is actually being displayed)*. For example, if the floor is adjusted upward to fifty-percent of the maximum value, the spectrum for the image in Figure 9 shows only the bright spot at the origin plus a few light gray specs near the origin.

If the floor is adjusted downward to one-percent of the maximum value, the spectrum for the image in Figure 9 shows quite a bit more light gray and quite a bit less black.

Change the sign of the negative spectral values

The code for the **normalize** method begins in Listing 10.

```
  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
```

Listing 10

Listing 10 gets the size of the incoming 2D array, and then changes the sign of all negative values stored in the array.

Convert to log base 10

Listing 11 converts all of the values to log base 10 to preserve the dynamic range of the plotting system. Negative log values are then set to zero.

```
    //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
```

Listing 11

Establish the floor

Listing 12 sets all values below X-percent of the maximum value to X-percent of the maximum value where X is determined by the value of scale. Listing 12 also slides all values down to cause the floor to be 0.0.

```
    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
```

Listing 12

Scale to accommodate the 0 to 255 rule

Listing 13 scales the data so that the maximum value becomes 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
```

Listing 13

Listing 13 also signals the end of the **normalize** method.

Process green and blue color planes

That brings us back to the **processImg** method, which we last saw in Listing 3. When the code in Listing 3 finishes execution, the red color plane has been transformed to spectral format and is ready to be displayed.

Listing 14 applies exactly the same process to the green and blue color planes.

```
//Back in the processImg method

    //Extract and process the green plane
    double[][] greenPlane =
extractPlane(working3D,2);
    processPlane(greenPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,greenPlane,2);

    //Extract and process the blue plane
    double[][] bluePlane =
extractPlane(working3D,3);
    processPlane(bluePlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,bluePlane,3);
```

Listing 14

When the code in Listing 14 finishes execution, all three color planes have been transformed to spectral format, and are ready to be displayed.

Return to the plotting program

Listing 15 converts the results to a 3D array containing data of type **int** and returns that array to the calling method in the class named **ImgMod02a**, where it will be plotted in the format shown in Figure 9.

```
    output = copyToInt(working3D);

    return output;

  }//end processImg method
```

Listing 15

Listing 15 also signals the end of the **processImg** method and the end of the class named **ImgMod34a**.

Now let's turn our attention from the program named **ImgMod34a** to the program named **ImgMod34**.

The program named ImgMod34

This program performs a forward 2D-DCT on an image followed by an inverse 2D-DCT on the spectral data produced by the forward DCT. The result is to use the spectral data to reconstruct a replica of the original image as shown in Figure 1.

Re-quantize to eleven bits

To partially simulate the behavior of the JPEG image compression algorithm, all of the spectral results produced by the forward transform are re-quantized so that the data could be stored in an eleven-bit twos complement format *(-1024 to +1023)*. Although the re-quantized data is never actually stored in an eleven-bit integer format, this process should create the same re-quantization noise that would be experienced if the data were actually stored in eleven bits.

Otherwise, nothing is done to the spectral data

Other than the re-quantization to eleven bits mentioned above, nothing is done to the spectral data following the forward DCT and before the inverse DCT. However, additional processing, such as selective high-frequency re-quantization and entropy compression, followed by decompression could be inserted at that point in the program for demonstration purposes.

Also runs under control of ImgMod02a

As with the earlier program, this program is designed to run under control of the class named **ImgMod02a**. Enter the following at the command line to run this program:

```
java ImgMod02a ImgMod34 ImageFileName
```

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

Other class files required

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

- ImgMod34.class
- ImgIntfc02.class
- ImgMod02a.class
- InverseDCT01.class
- ForwardDCT01.class

The source code for the first class in the list is presented in Listing 24. The source code for the last two classes in the list was developed and explained in the previous lesson entitled Understanding the Discrete Cosine Transform in Java. The source code for the other two classes can be found in the earlier lessons entitled Processing Image Pixels using Java, Getting Started and Processing Image Pixels Using Java: Controlling Contrast and Brightness.

Program testing

This program was tested using J2SE 5.0 and WinXP. J2SE 5.0 or later is required due to the use of static imports.

Code is very similar to earlier program

Much of the code in this program is the same as, or very similar to the code in the program named **ImgMod34a**, which I explained earlier in this lesson. I won't repeat that explanation, but rather will simply refer you to Listing 24 where you can find a complete listing of **ImgMod34**. I will concentrate on the code that is different between the two programs.

The beginning of the ImgMod34 class

Listing 16 shows the beginning of the class definition for **ImgMod34**.

```
class ImgMod34 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){

//Code deleted for brevity

    //Extract and process the red plane
    double[][] redPlane =
extractPlane(working3D,1);
    processPlane(redPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,redPlane,1);
```

Listing 16

Note that because of the similarity of the code in this portion of the program to the code that I explained earlier, I deleted most of the code from Listing 16, leaving just enough code to keep us in synch with the execution of the program.

At this point, I will set the discussion of the method named **processImg** aside and explain the method named **processPlane**, which is invoked near the bottom of Listing 16.

The processPlane method

The code for the **processPlane** method begins in Listing 17.

```
  void processPlane(double[][] colorPlane){

//Code deleted for brevity.

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

Listing 17
```

This method processes a color plane received as an incoming parameter.

- First it performs a forward 2D-DCT on the color plane producing spectral results.
- Then it re-quantizes the spectral data such that it could be stored in an eleven-bit twos complement integer format.
- Following that, the method performs an inverse 2D-DCT on the spectral plane producing an image color plane.

Once again, for the reasons given earlier, I deleted most of the code from Listing 17 for brevity. See Listing 24 for the missing code.

Re-quantization to eleven bits

Although I'm not absolutely certain at this point as to the exact format that JPEG uses to represent numeric spectral values, I'm reasonably confident that it is not a Java **double** format, but instead is an integer format. One source on the web, entitled The JPEG Tutorial, implies that the spectral values can range from -1024 to +1023. This strongly suggests an eleven-bit twos complement integer format.

To approximate this apparent characteristic of JPEG, Listing 18 invokes the method named **requanToElevenBits** to re-quantize the data such that it would fit into eleven bits *(-1024 to +1023)* as a twos complement integer type.

```
    //Get, save, and display the max value.
    double max = getMax(colorPlane);
    System.out.println(max);
    requanToElevenBits(colorPlane,max/1023);
    //Display requantized max value. (Should be 
1023.)
    System.out.println(getMax(colorPlane));

Listing 18
```

This is probably not exactly how it is done in JPEG, but hopefully it is a good approximation.

> *(Note that I am assuming that the maximum spectral value for the plane can be saved along with the spectral data until the time comes to perform the inverse transform. Exactly how that all is accomplished is still to be determined by my ongoing research into the details of the JPEG algorithm.)*

Not stored in an integer format

In this program, even though the spectral data is re-quantized so that it will fit into an eleven-bit integer format, the data is never actually stored in an eleven-bit integer format. Rather, immediately after being re-quantized, each value is converted back to type **double** for storage in the array of type **double**.

The method named requanToElevenBits

You can view the method named **requanToElevenBits** in Listing 24. I am not going to explain that method in this lesson, but will explain re-quantization in a general sense in a future lesson.

> *(Re-quantization plays a much more significant role in JPEG than the role that it plays in this program. In fact, re-quantization is one of the central components of JPEG compression, and for that reason, it will be the primary topic of a future lesson in its own right.)*

Image is currently in eleven-bit spectral format

At this point, the image has been transformed from the image or space domain into the frequency or wave-number domain. In addition, the spectral data has been re-quantized so that it could be converted to an eleven-bit integer format and stored in that format if there were a need to do so.

> *(According to my earlier analogy, it has been transformed from water to ice.)*

Restore the spectral magnitude

Now I will convert the spectral data back into image data. First I will restore the magnitude of the spectral data that has been re-quantized to the range -1024 to +1023. This is necessary so that the relative magnitudes among the spectra for the three color planes will be correct.

> *(Note, however, that the spectral data may have been corrupted by the introduction of quantization noise as a result of having been re-quantized, and the following operation will not eliminate such quantization noise. Once the re-quantization noise is there, it is there to stay.)*

Listing 19 invokes the method named **restoreSpectralMagnitude** in order to restore the magnitude of the spectral data.

```
restoreSpectralMagnitude(colorPlane,max/1023);
    //Display restored max value.
    System.out.println(getMax(colorPlane));
```

Listing 19

The method named **restoreSpectralMagnitude** is straightforward and shouldn't require an explanation. The method can be viewed in its entirety in Listing 24.

Perform the inverse 2D Discrete Cosine Transform

Listing 20 uses the static method named **transform** belonging to the class named **InverseDCT01** to perform the reverse of the operation explained earlier that began in Listing 5.

```
    //Extract each col from the spectral plane
and perform
    // an inverse DCT on the column.  Then
insert it back
    // into the color plane.
    for(int col = 0;col < imgCols;col++){
      double[] theXform =
extractCol(colorPlane,col);

      double[] theCol = new
double[theXform.length];
      //Now transform it back
      InverseDCT01.transform(theXform,theCol);

      //Insert it back into the color plane.
      insertCol(colorPlane,theCol,col);
    }//end for loop

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

      double[] theRow = new
double[theXform.length];
      //Now transform it back
      InverseDCT01.transform(theXform,theRow);

      //Insert it back in
      insertRow(colorPlane,theRow,row);
    }//end for loop
    //End inverse transform code
```

Listing 20

The transform method

The static method named **transform** belonging to the class named **InverseDCT01** was explained in the previous lesson in this series. Assuming that you have studied the earlier lesson entitled Understanding the Discrete Cosine Transform in Java, there is nothing in Listing 20 that should require a further explanation.

Clip at 0 and 255

At this point, the spectral data has been converted back into image color data and we are faced with the familiar problem of guaranteeing that the values are compatible with representation as unsigned eight-bit values *(0 to 255).* In this case, I elected not to do anything fancy. Listing 21 invokes two methods that clip the values at 0 and 255 respectively, simply discarding any values that fall outside those bounds.

```
    clipToZero(colorPlane);
    clipTo255(colorPlane);

  }//end processPlane
```

Listing 21

Both of the methods invoked in Listing 21 are straightforward. You can view them in their entirety in Listing 24.

Return control to the processImg method

Listing 21 also signals the end of the **processPlane** method, returning control to the code in the **processImg** method shown in Listing 22.

Listing 22 shows the remaining code in the **processImg** method.

```
//Back in the processImg method

    //Extract and process the green plane
    double[][] greenPlane =
extractPlane(working3D,2);
    processPlane(greenPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,greenPlane,2);

    //Extract and process the blue plane
    double[][] bluePlane =
extractPlane(working3D,3);
    processPlane(bluePlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,bluePlane,3);

    //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
```

Listing 22

Listing 22 processes the green and blue color planes in a manner identical to the previous processing of the red color plane.

Then Listing 22 converts the resulting image data to the required **int** format and returns the array reference to the calling method in the class named **ImgMod029a**.

That's a wrap!

So there you have it,

- Transformation of an image into the frequency domain using a 2D Discrete Cosine Transform.
- Re-quantization of the spectral data to fit in eleven bits.
- Transformation of the re-quantized spectral data back into an image.

The results for one image are shown in Figure 1. As mentioned earlier, there appears to be some noise in the large empty areas of the image in Figure 1, *(such as the empty white areas)*, which I believe is the result of the introduction of quantization noise into the spectral data when re-quantizing the spectral data to force it to fit in eleven bits.

Run the Program

I encourage you to copy and compile the code from Listing 23 and Listing 24. Experiment with the code, making changes, and observing the results of your changes.

For example, try eliminating the code in **ImgMod034** that re-quantizes the spectral data to see if it makes any difference in the quality of the resulting image.

Try running **ImgMod34a** on a variety of different images to see if you can reach any conclusions regarding the appearance of the wave-number spectrum and the appearance of the image, as shown in Figure 3, Figure 5, Figure 7, and Figure 9 for example.

Summary

In this lesson, I taught you how to use the *forward* 2D-DCT to compute and to 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.

What's Next?

The next publication in this series will be the second part of this two-part lesson on two-dimensional Discrete Cosine Transforms *(2D-DCT)*.

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

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
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 23 and Listing 24 below.

```
/*File ImgMod34a.java
Copyright 2006, R.G.Baldwin

This program is a modification of ImgMod34.  The purpose of
this program is to compute and to display the wave-number
spectrum of an image using a Discrete Cosine Transform.

This program performs a forward DCT on each color plane
```

```
belonging to an image producing a wave-number spectrum that
describes each color plane in the image.

Then it converts the wave-number spectrum to decibels and
normalizes the result to cover the range from 0 to 255.
This makes it suitable for being displayed as an image.

Then it returns the wave-number spectrum for each color
plane in an image format.

When displayed as an image, the result is the composite of
the normalized wave number spectra of all three color
planes.

The capability is provided to enable statements that will
effectively eliminate one, two, or all three of the color
planes from the computation.  This requires modification
of the source code and recompilation of the program.

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 ImgMod34a 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
ImgMod34a.class
ForwardDCT01.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 ImgMod34a implements ImgIntfc02{

  //Note that many of the comments in this source code are
  // left over from the class named ImgMod34, which was the
  // class from which this class was created.

  //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
  // containg 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 process the red plane
    double[][] redPlane = extractPlane(working3D,1);
    processPlane(redPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,redPlane,1);

    //Extract and process the green plane
    double[][] greenPlane = extractPlane(working3D,2);
    processPlane(greenPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,greenPlane,2);

    //Extract and process the blue plane
    double[][] bluePlane = extractPlane(working3D,3);
    processPlane(bluePlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,bluePlane,3);

    //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 double version of a 3D pixel array
// to 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
//-----------------------------------------------------//

//This method processes a color plane received as an
// incoming parameter.  First it performs a 2D-DCT on
// the color plane producing spectral results.  Then it
// normalizes the spectral values in the plane to make
// them compatible with being displayed as an image.  In
// so doing, it converts the spectral data to decibels in
// order to preserve the plotting dynamic range.

void processPlane(double[][] colorPlane){

  int imgRows = colorPlane.length;
  int imgCols = colorPlane[0].length;

  //Extract each row from the color plane and perform a
  // forward DCT on the row.  Then insert it back into
  // the color plane.
  for(int row = 0;row < imgRows;row++){
    double[] theRow = extractRow(colorPlane,row);

    double[] theXform = new double[theRow.length];
    ForwardDCT01.transform(theRow,theXform);
    //Insert the transformed row into the color plane.
    // The row now contains spectral data.
    insertRow(colorPlane,theXform,row);
  }//end for loop

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

    double[] theXform = new double[theCol.length];
    ForwardDCT01.transform(theCol,theXform);

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

  //At this point, the image has been transformed from
  // image or space data to spectral data in both
  // dimensions.

  //Normalize the spectral values to the range 0 - 255.
  normalize(colorPlane);
}//end processPlane
//-------------------------------------------------------//
//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 ImgMod34a
```

Listing 23

Listing 24

```
/*File ImgMod34.java
Copyright 2006, R.G.Baldwin

This program performs a forward DCT on an image followed by
an inverse DCT on the spectral planes produced by the
forward DCT.

To partially simulate the behavior of JPEG, the spectral
results produced by the forward transform are requantized
so that the data could be stored in an eleven-bit twos
complement format (-1024 to +1023).  However, the data is
never actually stored in an integer format.  However, this
process should create the same requantization noise that
would be experienced if the data were actually stored in
eleven bits.

Other than the requantization to eleven bits mentioned
above, nothing is done to the spectral planes following the
forward DCT and before the inverse DCT.  However,
additional processing, such as high-frequency
requantization and entropy compression, followed by
decompression could be inserted at that point in the
program for demonstration purposes.

This program runs significantly slower than ImgMod35,
which sub-divides the image into 8x8-pixel subplanes and
processes the subplanes separately.
```

```
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 ImgMod34 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
ImgMod34.class
InverseDCT01.class
ForwardDCT01.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 ImgMod34 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
  // contaning 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 process the red plane
    double[][] redPlane = extractPlane(working3D,1);
    processPlane(redPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,redPlane,1);

    //Extract and process the green plane
    double[][] greenPlane = extractPlane(working3D,2);
    processPlane(greenPlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,greenPlane,2);

    //Extract and process the blue plane
    double[][] bluePlane = extractPlane(working3D,3);
    processPlane(bluePlane);
    //Insert the plane back into the 3D array
    insertPlane(working3D,bluePlane,3);

    //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 double version of a 3D pixel array
  // to 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
  //----------------------------------------------------------//

  //This method processes a color plane received as an
  // incoming parameter.  First it performs a 2D-DCT on
  // the color plane producing spectral results.  Then it
  // performs an inverse DCT on the spectral plane
  // producing an image color plane.
  void processPlane(double[][] colorPlane){

    int imgRows = colorPlane.length;
    int imgCols = colorPlane[0].length;

    //Extract each row from the color plane and perform a
```

```
    // forward DCT on the row.  Then insert it back into
    // the color plane.
    for(int row = 0;row < imgRows;row++){
      double[] theRow = extractRow(colorPlane,row);

      double[] theXform = new double[theRow.length];
      ForwardDCT01.transform(theRow,theXform);

      //Insert the transformed row into the color plane.
      // The row now contains spectral data.
      insertRow(colorPlane,theXform,row);
    }//end for loop

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

      double[] theXform = new double[theCol.length];
      ForwardDCT01.transform(theCol,theXform);

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

    //To approximate the behavior of JPEG, I need to
    // re-quantize the data such that it would fit into
    // eleven bits (-1024 to +1023) as an integer type.
    // This is probably not exactly how it is done in
    // JPEG, but hopefully it is a good approximation.
    // I am assuming that the maximum value for this
    // plane can be saved along with the spectral data
    // until time comes to perform the inverse transform.
    //Note that in this program, even though the spectral
    // data is requantized so that it will fit into
    // an eleven-bit integer format, the data is never
    // actually stored in an eleven-bit integer format.
    // Rather, immediately after being requantized, each
    // value is converted back to type double for storage
    // in the array of type double[][].

    //Get, save, and display the max value.
    double max = getMax(colorPlane);
    System.out.println(max);
    requanToElevenBits(colorPlane,max/1023);
    //Display requantized max value. (Should be 1023.)
    System.out.println(getMax(colorPlane));

    //At this point, the image has been transformed from
    // image or space data to spectral data in both
    // dimensions. In addition, the spectral data has been
    // requantized so that could be converted to an
    // eleven-bit integer format and stored in that format
    // if there were a need to do so.

    //Now convert the spectral data back into image data.
    //First restore the magnitude of the spectral data
    // that has been requantized to the range -1024 to
    // +1023.  This is necessary so that the relative
    // magnitudes among the spectra for the three color
    // planes will be correct.
    //Note that the spectral data may have been corrupted
    // by quantization noise as a result of having
    // been requantized.
    restoreSpectralMagnitude(colorPlane,max/1023);
    //Display restored max value.
    System.out.println(getMax(colorPlane));

    //Extract each col from the spectral plane and perform
    // an inverse DCT on the column.  Then insert it back
    // into the color plane.
    for(int col = 0;col < imgCols;col++){
      double[] theXform = extractCol(colorPlane,col);

      double[] theCol = new double[theXform.length];
      //Now transform it back
      InverseDCT01.transform(theXform,theCol);

      //Insert it back into the color plane.
      insertCol(colorPlane,theCol,col);
    }//end for loop

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

      double[] theRow = new double[theXform.length];
      //Now transform it back
      InverseDCT01.transform(theXform,theRow);

      //Insert it back in
      insertRow(colorPlane,theRow,row);
    }//end for loop
    //End inverse transform code

    //At this point, the spectral data has been converted
    // back into image color data.  Ultimately it will be
    // necessary to convert it to 8-bit unsigned pixel
    // color format in order to display it as an image.
    //  Clip to zero and 255.
    clipToZero(colorPlane);
    clipTo255(colorPlane);

  }//end processPlane
  //-----------------------------------------------------//

  //Purpose: to find and return the maximum value
  double getMax(double[][] plane){
    int rows = plane.length;
    int cols = plane[0].length;
    double max = Double.MIN_VALUE;
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        double value = plane[row][col];
        if(value < 0){
          value = -value;
        }//end if
        if(value > max){
          max = value;
        }//end if
      }//end inner loop
    }//end outer loop
    return max;
  }//end getMax
  //-----------------------------------------------------//

  //Purpose:  To requantize the spectral data such that it
  // would fit into eleven bits (-1024 to 1023).  Note
  // that even though the data is rounded to type int in
  // this method, it is immediately converted back to type
  // double when it is stored in the array referred to by
  // plane.  Thus, it is never actually stored in an
  // integer format.
  void requanToElevenBits(double[][] plane,double divisor){
    int rows = plane.length;
    int cols = plane[0].length;
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        plane[row][col] = round(plane[row][col]/divisor);
      }//end inner loop
    }//end outer loop
  }//end requanToElevenBits
  //-----------------------------------------------------//

  //Purpose:  To restore the magnitude of spectral data
  // that has been requantized to the range from -1024 to
  // +1023.  This is necessary so that the relative
  // magnitude among the spectra for the three color planes
  // will be correct.
  void restoreSpectralMagnitude(
                         double[][] plane,double factor){
    int rows = plane.length;
    int cols = plane[0].length;
    for(int row = 0;row < rows;row++){
      for(int col = 0;col < cols;col++){
        plane[row][col] = factor * plane[row][col];
      }//end inner loop
    }//end outer loop
  }//end restoreSpectralMagnitude
  //-----------------------------------------------------//
}//end class ImgMod34
```

Listing 24

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-