# Understanding the Discrete Cosine Transform in Java

*Learn the basics of the Discrete Cosine Transform, which is used in many applications, including JPEG image compression.*

**Published:** July 11, 2006
**By Richard G. Baldwin**

Java Programming Notes # 2444



---

## 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 Huffman Data Compression Algorithm in Java.

### JPEG image compression

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

> *"... **JPEG** (pronounced jay-peg) is a commonly used standard method of lossy compression for photographic images. ... The name stands for **Joint Photographic Experts Group**. JPEG itself specifies only how an image is transformed into a stream of bytes, but not how those bytes are encapsulated in any particular storage medium. A further standard, created by the **Independent JPEG Group**, called **JFIF** (JPEG File Interchange Format) specifies how to produce a file suitable for computer storage and transmission (such as over the Internet) from a JPEG stream. ... JPEG/JFIF is the most common format used for storing and transmitting photographs on the World Wide Web."*

### Entropy encoding

Without getting into the technical details at this point, let me tell you that one of the central components of JPEG compression is entropy encoding. Huffman encoding, which was the primary topic of the previous lesson is a common form of entropy encoding.

### The Discrete Cosine Transform

Another central component of JPEG compression is the Discrete Cosine Transform, which is the primary topic of this lesson. Again, according to Wikipedia,

> *"The **discrete cosine transform** (DCT) is a Fourier-related transform similar to the discrete Fourier transform (DFT), but using only real numbers. It is equivalent to a DFT of roughly twice the length, operating on real data with even symmetry ..."*

### In order to understand JPEG ...

In order to understand JPEG image compression, you must understand Huffman encoding, the Discrete Cosine Transform, and some other topics as well, such as spectral re-quantization. I plan to teach you about the different components of JPEG in separate lessons, and then to teach 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

Although the Discrete Cosine Transform *(DCT)* may be *"similar to the discrete Fourier transform"*, it is not a Discrete Fourier Transform *(DFT)* as described in my earlier lesson entitled Fun with Java, How and Why Spectral Analysis Works. There are some major differences between DFT and DCT. Nonetheless, the DCT is very interesting from a technical viewpoint and is a central component of JPEG image compression.

## The DFT and symmetrical real data

To get started, I'm going to show you the results of applying the DFT to symmetrical real data. In the earlier lesson entitled Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain, I introduced you to a program named **Dsp035**, which:

> *"... illustrates the reversible nature of the Fourier transform. This program transforms a real time series into a complex spectrum, and then reproduces the real time series by performing an inverse Fourier transform on the complex spectrum. This is accomplished using a DFT algorithm."*

## The program named Dsp043

Listing 11 contains a complete listing of a program named **Dsp043**, which is designed to demonstrate that the imaginary part of the Fourier transform of a real symmetrical time series is all zeros if the origin is properly located.

> *(The code in this program is very similar to the code in the program named **Dsp035** that I explained earlier, so I won't repeat that explanation in this lesson. Rather, I will just explain the results of executing the program named **Dsp043**.)*

## The program output

Figure 1 shows the results of executing the program named **Dsp043** shown in Listing 11.

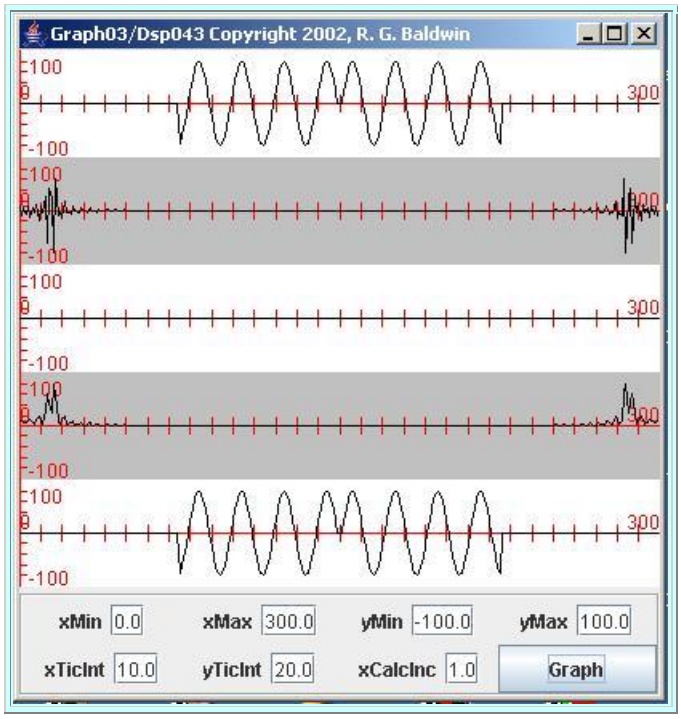

**Figure 1**

The first graph at the top of Figure 1 shows a 300-sample symmetrical time series. Note that this time series is symmetrical about its center point.

## The spectral results

The DFT was applied to this time series. The next three graphs going down the page in Figure 1 show the spectral results of applying the DFT to the time series. The three spectral graphs are plotted from a frequency of zero to the sampling frequency.

> *(Recall that the Nyquist folding frequency occurs half way between zero and the sampling frequency, and that the spectrum shown above the folding frequency is the mirror image of the spectrum below the folding frequency. Therefore, we are usually interested only in the spectral results below the folding frequency. Therefore, you can ignore the right half of the spectral graphs.)*

## What do the five graphs show?

The five graphs in Figure 1 show *(in order from top to bottom)*:

1. The 300-sample symmetrical real time series to which the DFT was applied.
2. The real part of the output from the DFT.
3. The imaginary part of the output from the DFT.
4. The magnitude of the output from the DFT.
5. The result of applying an inverse DFT to the spectral data in order to reconstruct the original time series from the spectral data.

## The important points

The most important points illustrated by Figure 1 are:

- The time series is purely real.
- The time series is symmetrical about its center point.
- The imaginary part of the spectrum has all zero values.

Therefore, Figure 1 demonstrates that *the imaginary part of the Fourier transform of a real symmetrical time series is all zeros if the origin is properly located*. This is important because I will use that fact to develop the rationale for the Discrete Cosine Transform, and why it works the way that it does.

## Equations for the DFT

Referring back to the equations in my earlier lesson, you can see that the real part of the output from the DFT results from a *sum of products* involving a cosine term, and that the imaginary part of the output results from a *sum of products* involving a sine term.

## Can sometimes avoid the sine computation

When computing the DFT, if we already know that the input time series is symmetrical and that the imaginary part of the output will be zero, we can simply forego the computation of the imaginary part that involves the sine term, thereby reducing the computational requirements.

## Can make the cosine computation less burdensome

In addition, if we know that the input time series is symmetrical, we can reformulate the computation of the real part of the transform involving the cosine term wherein we perform the computation on one-half of the time series only and then double the result. Thus for a symmetrical time series having a length of 2N+1 samples, we can reduce the number of real-part computations to N+1.

## How does the DCT work?

Basically the DCT works by implicitly doubling the length of the input time series by concatenating it to a mirror image of itself. The concatenation of the original time series to the mirror image results in a symmetrical time series.

> *(You don't actually see the doubling of the time series. Rather, the doubling is implicit in the formulation of the equations for the DCT.)*

Because the new time series is symmetrical, it is known in advance that if we were to perform a DFT on the new double-length time series, the imaginary part would be zero. Thus, it is also known in advance that we can forego the computation of the imaginary part that uses the sine term in the DFT.

## No need to double the number of cosine computations

In addition, because the new double-length time series is symmetrical, we don't need to double the number of real-part computations involving the cosine term *(but we will have to reformulate the real-part computation relative to a straight DFT computation)*.

## The definition of the DCT

The definition of the DCT is very similar to the definition of the DFT but the computation of the imaginary part using the sine term simply isn't part of the definition.

In addition to eliminating the computation of the imaginary part, the definition of the DCT also reformulates the DFT to take advantage of the symmetry of the time series relative to the computation of the real part using the cosine term.

## Let's see some equations

Rather than to deal with the somewhat difficult task of producing equations in this HTML document, I have provided three separate references that contain the equations for the DCT. The equations in these three references are essentially the same *(to within a scale factor)*.

I elected to formulate my DCT program for this lesson using the one-dimensional form of the DCT equations that you will find at National Taiwan University - DSP Group - Discrete Cosine Transform.

> *(The next lesson in this series will deal with the two-dimensional formulation of the DCT.)*

## No sine term

If you examine those equations, you will find that there is no sine term in either the forward or the inverse DCT. Only the cosine term is included, and it is included in such a way as to take advantage of the symmetry of the double-length time series.

> *(While you are at the site mentioned above, also note that the author provides an explanation of the doubling of the length of the time series in order to produce a new symmetrical time series for which the Fourier Transform is guaranteed to have a zero imaginary part.)*

## The forward Discrete Cosine Transform *(DCT)* code

Listing 12 provides a class named **ForwardDCT01**, which is an implementation of the forward DCT equation discussed above.

The code for the forward DCT is amazingly simple, considering what it is capable of accomplishing.

> *(Perhaps the simplicity of the code had something to do with why the DCT was selected to be a standard part of JPEG image compression.)*

## The *transform* method

The static method named **transform** belonging to the class named **ForwardDCT01** performs a forward Discreet Cosine Transform *(DCT)* on an incoming time series and returns the DCT spectrum.

## Input and output

The incoming parameters are:

- double[] x - incoming real data
- double[] y - outgoing real data

Thus, the input time series is provided by way of a **double[]** array object referred to by **x**. The **transform** method populates an output **double[]** array object referred to by **y**.

> *(Insofar as practical, the variable names and terms used in the **transform** method match the terms used in the equations discussed earlier.)*

Knowing the equation that the method is designed to implement, you should find the code in Listing 12 to be straightforward.

### The inverse Discrete Cosine Transform *(DCT)* code

Listing 13 provides the code for a class named **InverseDCT01**, which is an implementation of the inverse DCT equation discussed earlier.

The static method named **transform** of the **InverseDCT01** class performs an inverse Discreet Cosine Transform *(DCT)* on an incoming DCT spectrum and returns the DCT time series.

> *(As before, insofar as practical, the variable names and terms used in the method match the terms used in the equations discussed earlier.)*

### Input and output

Incoming parameters to the method are:

- double[] y - incoming real data
- double[] x - outgoing real data

Thus, the DCT spectrum to be transformed is provided by the user in an array object of type **double[]** referred to by **y**. The method named **transform** populates an array object of type **double[]** referred to by **x** with the time-series resulting from the transform.

As before, you should find the code in Listing 13 to be straightforward.

# Preview

### The program named Dsp042

The program named **Dsp042** illustrates the application of the forward and inverse Discrete Cosine Transform *(DCT)* to three different waveforms, all concatenated into a single time series. This program is very similar to **Dsp035**, which was explained in the earlier lesson entitled Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain.

### The program named Dsp035

The program named **Dsp035** applies the full forward and inverse Discrete Fourier Transform *(DFT)* to the same three waveforms producing the output shown in Figure 2.

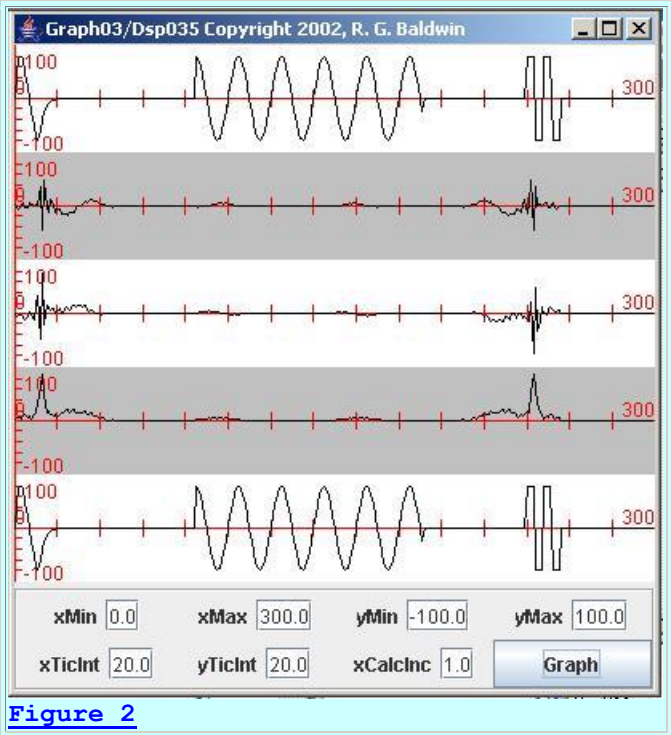

Figure 2

Figure 2 does not contain new material. You saw a figure similar to this one in the earlier lesson.

## What do the five graphs show?

The five graphs in Figure 2 show the following information, in order from top to bottom:

1. A 256-sample input time series consisting of three waveforms in sequence with values of zero between the waveforms. *(Note that even though the display is 300 points wide, the actual data being plotted on each of the five graphs ends at 256 samples.)*
2. The real part of the frequency spectrum produced by the forward DFT, plotted from a frequency of zero on the left to the sampling frequency *(twice the Nyquist folding frequency)* on the right.
3. The imaginary part of the frequency spectrum on the same scale as the real part of the transform.
4. The magnitude of the frequency spectrum, also on the same scale.
5. The result of applying the inverse DFT to the spectral data in order to reconstruct the original time series from the spectral data.

The most important thing to note in Figure 2 is that the imaginary part of the DFT output is clearly not zero for this time series input. The time series output in the bottom graph can only be produced by performing a complex transform on the real and imaginary parts of the frequency spectrum.

## Back to the program named Dsp042

The material shown in Figure 3, which is the output produced by the program named **Dsp042** is new to this lesson.

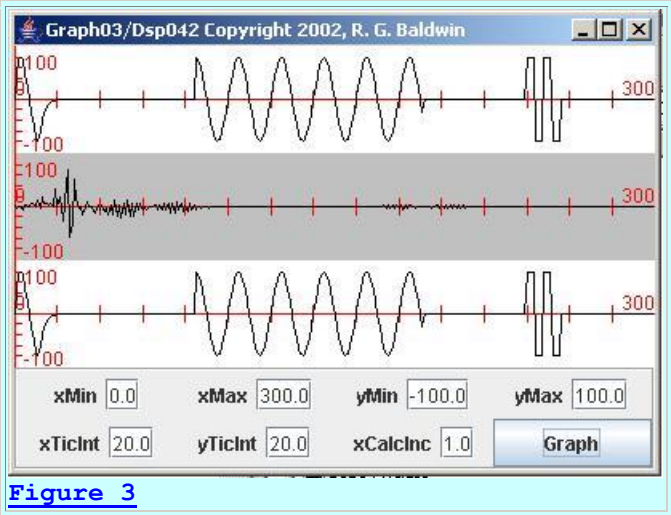

Figure 3

The three graphs in Figure 3 show:

1. A 256-sample input time series consisting of the same three waveforms shown in Figure 2. *(As with Figure 2, even though the display is 300 points wide, the actual data being plotted on each of the three graphs ends at 256 samples.)*
2. The real frequency spectrum produced by the forward DCT. In this case, the spectrum is plotted from a frequency of zero on the left to the Nyquist folding frequency on the right.
3. The result of applying the inverse DCT to the spectral data in order to reconstruct the original time series from the spectral data.

## Where is the imaginary part of the spectrum?

There is no imaginary part of the spectrum plotted in Figure 3, simply because the DCT doesn't produce an imaginary part. Since there is no imaginary part, there is also no need for a plot of the magnitude spectrum for the DCT. *(It would look exactly like the real part of the spectrum, with all negative values converted to positive values, if it were computed and plotted.)*

If you compare Figure 3 with Figure 2, you will see that the DFT and the DCT both appeared to do an equally good job of reproducing the original time series when the inverse transform was applied to the spectral data. The big difference between Figure 2 and Figure 3 is that this was accomplished with the somewhat more economical DCT in Figure 3.

## Water to ice and back to water

There is one point that I would like to make here, because it will become very important in a future lesson that deals with the inner workings of JPEG. The second and third graphs in Figure

2 represent the same information as the first graph in Figure 2. Similarly, the second graph in Figure 3 represents the same information as the first graph in Figure 3. In other words, the spectral data represents the same information as the time-series data. They simply represent that information in different forms.

As an analogy, if we lower the temperature of a container of water to less than 32 degrees Fahrenheit, the form of the water will change from a liquid to a solid. If we warm it back up, the form will change back to a liquid. This is roughly analogous to transforming a time series into the frequency domain and then transforming the frequency spectrum back into the time domain. The same information is represented in both cases. That information is simply represented in different forms.

## Sub-dividing the input

When performing spectral analysis, it is common practice to perform a DFT *(or perhaps a DCT)* on the entire time series as was the case in Figure 1, Figure 2, and Figure 3. However, in a future lesson we will learn that this is not the case for JPEG image compression. Instead, the JPEG procedure sub-divides the image into a set of small images where each small image consists of an 8x8 block of 64 pixels.

Then the DCT is performed on each individual block of 64 pixels. Several additional processing steps are performed on the spectra produced for the set of 8x8 blocks to produce the compressed image. Later on, when the image is reconstructed, the 8x8 blocks are individually reconstructed and are then assembled into a larger image that approximates the original image.

## The program named Dsp044

The program named **Dsp044** is designed to investigate the impact of sub-dividing the time series into eight-sample segments and processing those segments individually in order to be more consistent with the 8x8-pixel block concept in JPEG. As it turns out, there appears to be no noticeable impact. As you can see in Figure 4, the reconstructed output shown in the second graph is a very good replica of the input shown in the first graph.

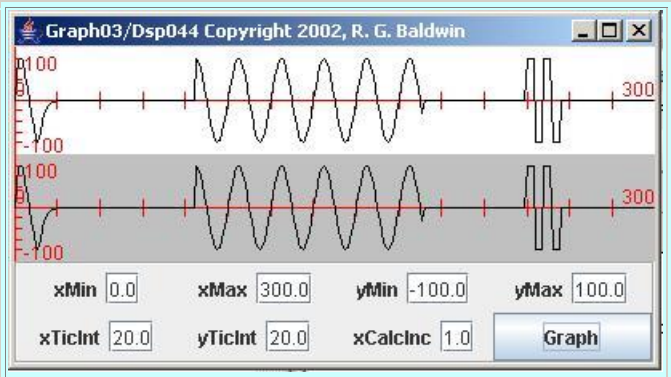


Figure 4

*(Figure 4 shows only the input and output time series. In order to display the spectral results, it would have been necessary to display 32 individual spectra, one computed for each 8-sample segment of the input time series. That would have been fairly impractical.)*

### Testing

Both programs were tested using J2SE 5.0 under WinXP. *(Both programs require J2SE 5.0 or later due to the use of static import of **Math** class.)*

# Discussion and Sample Code

### The program named Dsp042

The class definition for **Dsp042** begins in Listing 1 by declaring and initializing some instance variables.

```
class Dsp042 implements GraphIntfc01{

  int len = 256;

  double[] timeDataIn = new double[len];
  double[] realSpect = new double[len];
  double[] timeDataOut = new double[len];
  int zero = 0;
```

**Listing 1**

Note that the class named **Dsp042** implements the interface named **GraphIntfc01**.

### The constructor for Dsp042

The constructor begins in Listing 2.

```
  public Dsp042(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
//code deleted for brevity
    timeDataIn[20] = -2;
    timeDataIn[21] = -1;

    timeDataIn[240] = 80;
    timeDataIn[241] = 80;
//code deleted for brevity
    timeDataIn[254] = -80;
    timeDataIn[255] = -80;
```

**Listing 2**

The code in Listing 2 creates the first and last waveforms shown in the first graph in Figure 3. Note that I deleted quite a lot of code from Listing 2 for brevity. You can view the code that I deleted in Listing 14.

## Create the sinusoidal waveform

Listing 3 creates the truncated sinusoidal waveform shown near the center of the first graph in Figure 3.

```
    for(int x = len/3;x < 3*len/4;x++){
      timeDataIn[x] = 80.0 *
Math.sin(2*PI*(x)*1.0/20.0);
    }//end for loop
```

**Listing 3**

## Perform the forward Discrete Cosine Transform

Listing 4 invokes the static **transform** method of the **ForwardDCT01** class to compute the forward DCT of the time data and to save the results in the array referred to by **realSpect**, which was created in Listing 1.

```
ForwardDCT01.transform(timeDataIn,realSpect);
```

**Listing 4**

## Perform the inverse Discrete Cosine Transform

Listing 5 invokes the static **transform** method of the **InverseDCT01** class to compute the inverse DCT of the spectral data and to save the results in the array referred to by **timeDataOut**, which was created in Listing 1.

```
InverseDCT01.transform(realSpect,timeDataOut);

  }//end constructor
```

**Listing 5**

Listing 5 also signals the end of the constructor for the class named **Dsp042**.

## The remaining code

The remaining code in the class named **Dsp042** consists of six methods, which are required by the interface named **GraphIntfc01**. The purpose of these methods is simply to plot the data contained in three arrays, producing the three graphs shown in Figure 3. I explained those six methods in the earlier lesson entitled Plotting Engineering and Scientific Data using Java and won't repeat that explanation here. You can view the methods in Listing 14.

## The class named Dsp044

This class is a modification of the class named **Dsp042** designed to investigate the impact of sub-dividing the input time series into eight-sample segments in order to be consistent with the 8x8 blocks in JPEG.

Listing 6 shows the beginning of the class and the beginning of the constructor.

```
class Dsp044 implements GraphIntfc01{

  int len = 256;

  double[] timeDataIn = new double[len];
  double[] timeDataOut = new double[len];
  int zero = 0;

  public Dsp044(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
//code deleted for brevity
    timeDataIn[20] = -2;
    timeDataIn[21] = -1;

    timeDataIn[240] = 80;
    timeDataIn[241] = 80;
//code deleted for brevity
    timeDataIn[254] = -80;
    timeDataIn[255] = -80;

    //Create raw data sinusoid
    for(int x = len/3;x < 3*len/4;x++){
      timeDataIn[x] = 80.0 *
Math.sin(2*PI*(x)*1.0/20.0);
    }//end for loop

Listing 6
```

Once again, note that the class named **Dsp044** implements the interface named **GraphIntfc01**.

As before, I deleted some of the code from Listing 6 for brevity. The code in Listing 6 is very similar to the code that I explained earlier with respect to the class named **Dsp042**.

## Eight-element array objects

The real difference between **Dsp042** and **Dsp044** begins in Listing 7 where I create some eight-element array objects to handle the eight-sample segments.

```
    double[] workingArrayIn = new double[8];
    double[] workingArrayOut = new double[8];
    double[] realSpect = new double[8];


Listing 7
```

## Process eight samples at a time

Listing 8 shows the beginning of a **while** loop, which computes a forward and an inverse DCT on each successive eight-sample segment of the input time series. Code inside the **while** loop also concatenates the output segments from the inverse DCT to produce the output signal, which is shown by the bottom graph in Figure 4.

```
    int segmentCnt = 0;

    while((segmentCnt + 8) <= len){
      System.arraycopy(timeDataIn,
                       segmentCnt,
                       workingArrayIn,
                       0,
                       8);

Listing 8
```

During each iteration of the while loop, the code in Listing 8 copies the next eight samples from the input time series into an eight-element working array.

## Compute the forward and the inverse transforms

Listing 9 performs a forward DCT on the contents of the eight-element working array. Then it performs an inverse DCT on the eight-samples of spectral data produced by the forward transform.

```
ForwardDCT01.transform(workingArrayIn,realSpect);


InverseDCT01.transform(realSpect,workingArrayOut);

Listing 9
```

## Concatenate the output time-series segments

Listing 10 copies the eight samples of output time-series data produced by the inverse transform into the next eight samples of the array designated to hold the final output. This concatenates the eight-sample segments into the time series shown in the bottom graph in Figure 4.

```
      System.arraycopy(workingArrayOut,
                       0,
                       timeDataOut,
                       segmentCnt,
                       8);

      segmentCnt += 8;
    }//end while

  }//end constructor

Listing 10
```

Then Listing 10 increments the segment counter by 8 and control is transferred back to the top of the **while** loop shown in Listing 8.

Listing 10 also signals the end of the constructor.

### The remaining code

As before, the remaining code in the class named **Dsp044** consists of six methods, which are required by the interface named **GraphIntfc01**. The purpose of these methods is to plot the data contained in two arrays, producing the graphs shown in Figure 4. You can view the methods in Listing 15.

## Run the Programs

I encourage you to copy, compile, and execute the code from the listings in the section entitled Complete Program Listings. Experiment with the code, making changes and observing the results of your changes. For example, as one experiment you can see what happens if you make changes to the computed argument for the cosine function in either the forward or the inverse transform.

### Run under control of Graph03

**Dsp042**, **Dsp043**, and **Dsp044** must all be run under the control of the program named **Graph03**.

> *(The program named **Graph03** is a plotting program. See the earlier lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm, which was the first lesson in which the plotting program named **Graph03** was used. Also see **Graph01**, which was a predecessor of **Graph03**, in the lesson entitled Plotting Engineering and Scientific Data using Java.)*

The source code for **Graph03** is provided in Listing 16.

The source code for the interface named **GraphIntfc01**, which is required by these programs, is provided in Listing 17.

### Running the programs

To run these programs, first compile the programs and then enter one of the following statements at the command prompt.

```
java Graph03 Dsp042
java Graph03 Dsp042
java Graph03 Dsp044
```

### Support classes

You will need some support classes in order to run these programs. In those cases where the source code for a required support class is not included in this lesson, you should be able to find the source code in the lessons referred to in the References section.

You can also find the source code by searching for it on Google. For example, searching for the following keywords on Google will identify the earlier lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm containing the source code for the class named **ForwardRealToComplex01**.

```
java Baldwin "ForwardRealToComplex01.java"
```

## Summary

I introduced you to the basics of the Discrete Cosine Transform *(DCT)* by:

- Explaining some of the underlying theory behind the transform.
- Demonstrating the use of the transform in two one-dimensional cases.

## What's Next?

The next lesson will explain the use of the *two-dimensional* Discrete Cosine Transform and will illustrate its use to transform images into the wave-number domain and back into the space or image domain.

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
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 provided in the following listings:

**Listing 11**

```
/* File Dsp043.java
Copyright 2006, R.G.Baldwin
Revised 01/05/06

The purpose of this program is to demonstrate that the
imaginary part of the Fourier transform of a symmetrical
time series is all zeros if the origin is properly located.

Illustrates forward and inverse Fourier transforms on a
symmetrical time series using DFT algorithms.

Passes resulting real and complex parts to inverse Fourier
transform program to reconstruct the original time series.

Run with Graph03. Enter the following to run the program:
java Graph03 Dsp043
```

```
Exection of this program requires access to the following
class files:
Dsp043.class
ForwardRealToComplex01.class
Graph01.class
GraphIntfc01.class
GUI$MyCanvas.class
GUI.class
InverseComplexToReal01.class

Tested using J2SE 5.0 under WinXP.
**********************************************************/
import java.util.*;

class Dsp043 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 300;

  double[] timeDataIn = new double[len];
  double[] realSpect = new double[len];
  double[] imagSpect = new double[len];
  double[] angle = new double[len];//unused
  double[] magnitude = new double[len];
  double[] timeDataOut = new double[len];
  int zero = 0;

  public Dsp043(){//constructor

    //Create raw waveform consisting of mirror image
    // sinusoidal segments with a sample having a value
    // of 0 in the center.
    //Set shift to a nonzero value to cause the imaginary
    // part of the transform to be non zero.
    for(int x = 0;x < len/4;x++){
      timeDataIn[len/2 + x + 1] =
                        80.0 * Math.sin(2*pi*(x)*1.0/20.0);
      timeDataIn[len/2 - x - 1] =
                                  timeDataIn[len/2 + x + 1];
    }//end for loop

    //Compute DFT of the time data and save it in
    // the output arrays.
    ForwardRealToComplex01.transform(timeDataIn,
                                     realSpect,
                                     imagSpect,
                                     angle,
                                     magnitude,
                                     zero,
                                     0.0,
                                     1.0);

    //Compute inverse DFT of the spectal data and
    // save output time data in output array
    InverseComplexToReal01.inverseTransform(realSpect,
                                            imagSpect,
```

```
                                                    timeDataOut);
  }//end constructor
  //-----------------------------------------------------//

  //The following six methods are required by the interface
  // named GraphIntfc01.
  public int getNmbr(){
    //Return number of curves to plot.  Must not exceed 5.
    return 5;
  }//end getNmbr
  //-----------------------------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > timeDataIn.length-1){
      return 0;
    }else{
      return timeDataIn[index];
    }//end else
  }//end function
  //-----------------------------------------------------//
  public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > realSpect.length-1){
      return 0;
    }else{
      //scale for convenient viewing
      return 5*realSpect[index];
    }//end else
  }//end function
  //-----------------------------------------------------//
  public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imagSpect.length-1){
      return 0;
    }else{
      //scale for convenient viewing
      return 5*imagSpect[index];
    }//end else
  }//end function
  //-----------------------------------------------------//
  public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > magnitude.length-1){
      return 0;
    }else{
      //scale for convenient viewing
      return 5*magnitude[index];
    }//end else
  }//end function
  //-----------------------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                   index > timeDataOut.length-1){
      return 0;
    }else{
```

```
      return timeDataOut[index];
    }//end else
  }//end function

}//end sample class Dsp043
```

**Listing 11**

## Listing 12

```
/*File ForwardDCT01.java
Copyright 2006, R.G.Baldwin
Rev 01/03/06

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

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

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


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

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

public class ForwardDCT01{

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

    int N = x.length;

    //Outer loop interates on frequency values.
    for(int k=0; k < N;k++){
      double sum = 0.0;
      //Inner loop iterates on time-series points.
      for(int n=0; n < N; n++){
        double arg = PI*k*(2.0*n+1)/(2*N);
        double cosine = cos(arg);
        double product = x[n]*cosine;
        sum += product;
      }//end inner loop

      double alpha;
```

```
      if(k == 0){
        alpha = 1.0/sqrt(2);
      }else{
        alpha = 1;
      }//end else
      y[k] = sum*alpha*sqrt(2.0/N);
    }//end outer loop
  }//end transform method
  //-------------------------------------------------//
}//end class ForwardDCT01

Listing 12
```

## Listing 13

```
/*File InverseDCT01.java
Copyright 2006, R.G.Baldwin
Rev 01/03/06

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

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

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

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

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

public class InverseDCT01{

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

    int N = y.length;

    //Outer loop interates on time values.
    for(int n=0; n < N;n++){
      double sum = 0.0;
      //Inner loop iterates on frequency values
      for(int k=0; k < N; k++){
        double arg = PI*k*(2.0*n+1)/(2*N);
        double cosine = cos(arg);
        double product = y[k]*cosine;
```

```
      double alpha;
      if(k == 0){
        alpha = 1.0/sqrt(2);
      }else{
        alpha = 1;
      }//end else

      sum += alpha * product;

      }//end inner loop

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

    }//end outer loop
  }//end transform method
  //-------------------------------------------------//
}//end class InverseDCT01
```

**Listing 13**

## Listing 14

```
/* File Dsp042.java
Copyright 2006, R.G.Baldwin
Revised 01/05/06

Note:  Dsp044 will investigate the impact of breaking the
time series into eight-sample segments to be consistent
with the 8x8 blocks in JPEG.

Illustrates the application of forward and inverse Discrete
Cosine Transform (DCT) to three different waveforms.  Very
similar to Dsp035, which applies full forward and inverse
Fourier transforms to the same three waveforms

See http://en.wikipedia.org/wiki/Discrete_cosine_transform
#DCT-II, http://rkb.home.cern.ch/rkb/AN16pp/node61.html,
and http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/
coding/transform/dct.html fortechnical background
information on the Discrete Cosine Transform.

Shows a plot of the input time series, the DCT spectrum,
and the output time series resulting from the inverse DCT.

Run with Graph03.  Enter the following to run the program:
java Graph03 Dsp042

Execution of this program requires access to the following
class files:
Dsp042.class
ForwardDCT01.class
Graph01.class
GraphIntfc01.class
GUI$MyCanvas.class
```

```
GUI.class
InverseDCT01.class

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

class Dsp042 implements GraphIntfc01{

  int len = 256;

  double[] timeDataIn = new double[len];
  double[] realSpect = new double[len];
  double[] timeDataOut = new double[len];
  int zero = 0;

  public Dsp042(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
    timeDataIn[2] = 75;
    timeDataIn[3] = 80;
    timeDataIn[4] = 75;
    timeDataIn[5] = 50;
    timeDataIn[6] = 25;
    timeDataIn[7] = 0;
    timeDataIn[8] = -25;
    timeDataIn[9] = -50;
    timeDataIn[10] = -75;
    timeDataIn[11] = -80;
    timeDataIn[12] = -60;
    timeDataIn[13] = -40;
    timeDataIn[14] = -26;
    timeDataIn[15] = -17;
    timeDataIn[16] = -11;
    timeDataIn[17] = -8;
    timeDataIn[18] = -5;
    timeDataIn[19] = -3;
    timeDataIn[20] = -2;
    timeDataIn[21] = -1;

    timeDataIn[240] = 80;
    timeDataIn[241] = 80;
    timeDataIn[242] = 80;
    timeDataIn[243] = 80;
    timeDataIn[244] = -80;
    timeDataIn[245] = -80;
    timeDataIn[246] = -80;
    timeDataIn[247] = -80;
    timeDataIn[248] = 80;
    timeDataIn[249] = 80;
    timeDataIn[250] = 80;
    timeDataIn[251] = 80;
```

```
    timeDataIn[252] = -80;
    timeDataIn[253] = -80;
    timeDataIn[254] = -80;
    timeDataIn[255] = -80;

    //Create raw data sinusoid
    for(int x = len/3;x < 3*len/4;x++){
      timeDataIn[x] = 80.0 * Math.sin(
                              2*PI*(x)*1.0/20.0);
    }//end for loop

    //Compute forward DCT of the time data and save it in
    // the output array.
    ForwardDCT01.transform(timeDataIn,realSpect);

    //Compute inverse DCT of the time data and save it in
    // the output array.
    InverseDCT01.transform(realSpect,timeDataOut);

  }//end constructor

  //-------------------------------------------------------//
  //The following six methods are required by the interface
  // named GraphIntfc01.
  public int getNmbr(){
    //Return number of curves to plot.  Must not exceed 5.
    return 3;
  }//end getNmbr
  //-------------------------------------------------------//
  public double f1(double x){
    int index = (int)round(x);
    if(index < 0 || index > timeDataIn.length-1){
      return 0;
    }else{
      return timeDataIn[index];
    }//end else
  }//end function
  //-------------------------------------------------------//
  public double f2(double x){
    int index = (int)round(x);
    if(index < 0 || index > realSpect.length-1){
      return 0;
    }else{
      //Scale for convenient viewing
      return 0.22*realSpect[index];
    }//end else
  }//end function
  //-------------------------------------------------------//
  public double f3(double x){
    int index = (int)round(x);
    if(index < 0 || index > timeDataOut.length-1){
      return 0;
    }else{
      return timeDataOut[index];
    }//end else
  }//end function
```

```
  //-------------------------------------------------------//
  public double f4(double x){
    return 0;
  }//end function
  //-------------------------------------------------------//
  public double f5(double x){
    return 0;
  }//end function
  //-------------------------------------------------------//
}//end sample class Dsp042


Listing 14
```

## Listing 15

```
/* File Dsp044.java
Copyright 2006, R.G.Baldwin
Revised 01/03/06

Update of Dsp042 to investigate the impact of breaking the
time series into eight-sample segments in order to be
consistent with the 8x8 blocks in JPEG.  There appears to
be no impact. The output is a very good replica of the
input.

Illustrates the application of forward and inverse Discrete
Cosine Transform (DCT) to three different waveforms.  Very
similar to Dsp035, which applies full forward and inverse
Fourier transforms to the same three waveforms

See http://en.wikipedia.org/wiki/Discrete_cosine_transform
#DCT-II, http://rkb.home.cern.ch/rkb/AN16pp/node61.html,
and http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/
coding/transform/dct.html fortechnical background
information on the Discrete Cosine Transform.

Shows a plot of the input time series, and the output
time series resulting from the inverse DCT.  Can't show
the spectrum because a new spectrum is produced every
eight samples.

Run with Graph03. Enter the following to run the program:
java Graph03 Dsp044

Execution of this program requires access to the following
class files:
Dsp044.class
ForwardDCT01.class
Graph01.class
GraphIntfc01.class
GUI$MyCanvas.class
GUI.class
InverseDCT01.class
```

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

class Dsp044 implements GraphIntfc01{

  int len = 256;

  double[] timeDataIn = new double[len];
  double[] timeDataOut = new double[len];
  int zero = 0;

  public Dsp044(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
    timeDataIn[2] = 75;
    timeDataIn[3] = 80;
    timeDataIn[4] = 75;
    timeDataIn[5] = 50;
    timeDataIn[6] = 25;
    timeDataIn[7] = 0;
    timeDataIn[8] = -25;
    timeDataIn[9] = -50;
    timeDataIn[10] = -75;
    timeDataIn[11] = -80;
    timeDataIn[12] = -60;
    timeDataIn[13] = -40;
    timeDataIn[14] = -26;
    timeDataIn[15] = -17;
    timeDataIn[16] = -11;
    timeDataIn[17] = -8;
    timeDataIn[18] = -5;
    timeDataIn[19] = -3;
    timeDataIn[20] = -2;
    timeDataIn[21] = -1;

    timeDataIn[240] = 80;
    timeDataIn[241] = 80;
    timeDataIn[242] = 80;
    timeDataIn[243] = 80;
    timeDataIn[244] = -80;
    timeDataIn[245] = -80;
    timeDataIn[246] = -80;
    timeDataIn[247] = -80;
    timeDataIn[248] = 80;
    timeDataIn[249] = 80;
    timeDataIn[250] = 80;
    timeDataIn[251] = 80;
    timeDataIn[252] = -80;
    timeDataIn[253] = -80;
    timeDataIn[254] = -80;
    timeDataIn[255] = -80;
```

```
    //Create raw data sinusoid
    for(int x = len/3;x < 3*len/4;x++){
      timeDataIn[x] = 80.0 * Math.sin(
                              2*PI*(x)*1.0/20.0);
    }//end for loop

    double[] workingArrayIn = new double[8];
    double[] workingArrayOut = new double[8];
    double[] realSpect = new double[8];
    int segmentCnt = 0;

    //Compute forward and inverse DCTs on the input signal,
    // eight samples at a time.  Concatenate the output
    // segments from the DCT to represent the output
    // signal.
    while((segmentCnt + 8) <= len){
      System.arraycopy(timeDataIn,
                       segmentCnt,
                       workingArrayIn,
                       0,
                       8);

      //Compute forward DCT of the time data and save it in
      // the output array.
      ForwardDCT01.transform(workingArrayIn,realSpect);

      //Compute inverse DCT of the time data and save it in
      // the output array.
      InverseDCT01.transform(realSpect,workingArrayOut);

      //Concatenate the new output with the old output.
      System.arraycopy(workingArrayOut,
                       0,
                       timeDataOut,
                       segmentCnt,
                       8);

      segmentCnt += 8;
    }//end while

  }//end constructor

  //-----------------------------------------------------//
  //The following six methods are required by the interface
  // named GraphIntfc01.
  public int getNmbr(){
    //Return number of curves to plot.  Must not exceed 5.
    return 2;
  }//end getNmbr
  //-----------------------------------------------------//
  public double f1(double x){
    int index = (int)round(x);
    if(index < 0 || index > timeDataIn.length-1){
      return 0;
    }else{
```

```
      return timeDataIn[index];
    }//end else
  }//end function
  //-------------------------------------------------------//
  public double f2(double x){
    int index = (int)round(x);
    if(index < 0 || index > timeDataOut.length-1){
      return 0;
    }else{
      return timeDataOut[index];
    }//end else
  }//end function
  //-------------------------------------------------------//
  public double f3(double x){
    return 0;
  }//end function
  //-------------------------------------------------------//
  public double f4(double x){
    return 0;
  }//end function
  //-------------------------------------------------------//
  public double f5(double x){
    return 0;
  }//end function
  //-------------------------------------------------------//

}//end class Dsp044
```

**Listing 15**

## Listing 16

```
/* File Graph03.java
Copyright 2002, R.G.Baldwin

This program is very similar to Graph01
except that it has been modified to
allow the user to manually resize and
replot the frame.

Note:  This program requires access to
the interface named GraphIntfc01.

This is a plotting program.  It is
designed to access a class file, which
implements GraphIntfc01, and to plot up
to five functions defined in that class
file.  The plotting surface is divided
into the required number of equally
sized plotting areas, and one function
is plotted on cartesian coordinates in
each area.

The methods corresponding to the
```

```
functions are named f1, f2, f3, f4,
and f5.

The class containing the functions must
also define a method named
getNmbr(), which takes no parameters
and returns the number of functions to
be plotted.  If this method returns a
value greater than 5, a
NoSuchMethodException will be thrown.

Note that the constructor for the class
that implements GraphIntfc01 must not
require any parameters due to the
use of the newInstance method of the
Class class to instantiate an object
of that class.

If the number of functions is less
than 5, then the absent method names
must begin with f5 and work down toward
f1.  For example, if the number of
functions is 3, then the program will
expect to call methods named f1, f2,
and f3.  It is OK for the absent
methods to be defined in the class.
They simply won't be invoked.

The plotting areas have alternating
white and gray backgrounds to make them
easy to separate visually.

All curves are plotted in black.  A
cartesian coordinate system with axes,
tic marks, and labels is drawn in red
in each plotting area.

The cartesian coordinate system in each
plotting area has the same horizontal
and vertical scale, as well as the
same tic marks and labels on the axes.

The labels displayed on the axes,
correspond to the values of the extreme
edges of the plotting area.

The program also compiles a sample
class named junk, which contains five
methods and the method named getNmbr.
This makes it easy to compile and test
this program in a stand-alone mode.

At runtime, the name of the class that
implements the GraphIntfc01 interface
must be provided as a command-line
parameter.  If this parameter is
```

```
missing, the program instantiates an
object from the internal class named
junk and plots the data provided by
that class.  Thus, you can test the
program by running it with no
command-line parameter.

This program provides the following
text fields for user input, along with
a button labeled Graph.  This allows
the user to adjust the parameters and
replot the graph as many times with as
many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these
parameters and then click the Graph
button to cause the five functions
to be re-plotted according to the
new parameters.

Whenever the Graph button is clicked,
the event handler instantiates a new
object of the class that implements
the GraphIntfc01 interface.  Depending
on the nature of that class, this may
be redundant in some cases.  However,
it is useful in those cases where it
is necessary to refresh the values of
instance variables defined in the
class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were
first defined in the Color class of
v1.4.0.  Therefore, the program
requires v1.4.0 or later to compile and
run correctly.
***********************************/

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph03{
  public static void main(
```

```
          String[] args)
          throws NoSuchMethodException,
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{
    if(args.length == 1){
      //pass command-line paramater
      new GUI(args[0]);
    }else{
      //no command-line parameter given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph03 definition
//===================================//

class GUI extends JFrame
             implements ActionListener{

  //Define plotting parameters and
  // their default values.
  double xMin = 0.0;
  double xMax = 400.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 20.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small
  // on x-axis, a default value is
  // used later.
  double xTicLen = (yMax-yMin)/50;
  double yTicLen = (xMax-xMin)/50;

  //Calculation interval along x-axis
  double xCalcInc = 1.0;

  //Text fields for plotting parameters
  JTextField xMinTxt =
             new JTextField("" + xMin);
  JTextField xMaxTxt =
             new JTextField("" + xMax);
  JTextField yMinTxt =
             new JTextField("" + yMin);
  JTextField yMaxTxt =
             new JTextField("" + yMax);
  JTextField xTicIntTxt =
          new JTextField("" + xTicInt);
  JTextField yTicIntTxt =
          new JTextField("" + yTicInt);
  JTextField xCalcIncTxt =
         new JTextField("" + xCalcInc);

  //Panels to contain a label and a
```

```
  // text field
  JPanel pan0 = new JPanel();
  JPanel pan1 = new JPanel();
  JPanel pan2 = new JPanel();
  JPanel pan3 = new JPanel();
  JPanel pan4 = new JPanel();
  JPanel pan5 = new JPanel();
  JPanel pan6 = new JPanel();

  //Misc instance variables
  int frmWidth = 408;
  int frmHeight = 430;
  int width;
  int height;
  int number;
  GraphIntfc01 data;
  String args = null;

  //Plots are drawn on the canvases
  // in this array.
  Canvas[] canvases;

  //Constructor
  GUI(String args)throws
                NoSuchMethodException,
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{

    if(args != null){
      //Save for use later in the
      // ActionEvent handler
      this.args = args;
      //Instantiate an object of the
      // target class using the String
      // name of the class.
      data = (GraphIntfc01)
                    Class.forName(args).
                          newInstance();
    }else{
      //Instantiate an object of the
      // test class named junk.
      data = new junk();
    }//end else

    //Create array to hold correct
    // number of Canvas objects.
    canvases =
            new Canvas[data.getNmbr()];

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
      throw new NoSuchMethodException(
                "Too many functions.  "
```

```
                + "Only 5 allowed.");
    }//end if

    //Create the control panel and
    // give it a border for cosmetics.
    JPanel ctlPnl = new JPanel();
    ctlPnl.setLayout(//?rows x 4 cols
                  new GridLayout(0,4));
    ctlPnl.setBorder(
                   new EtchedBorder());

    //Button for replotting the graph
    JButton graphBtn =
                  new JButton("Graph");
    graphBtn.addActionListener(this);

    //Populate each panel with a label
    // and a text field.  Will place
    // these panels in a grid on the
    // control panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);

    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);

    //Add the populated panels and the
    // button to the control panel with
    // a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);

    //Create a panel to contain the
    // Canvas objects.  They will be
    // displayed in a one-column grid.
```

```
    JPanel canvasPanel = new JPanel();
    canvasPanel.setLayout(//?rows,1 col
                  new GridLayout(0,1));

    //Create a custom Canvas object for
    // each function to be plotted and
    // add them to the one-column grid.
    // Make background colors alternate
    // between white and gray.
    for(int cnt = 0;
                  cnt < number; cnt++){
      switch(cnt){
        case 0 :
          canvases[cnt] =
                     new MyCanvas(cnt);
          canvases[cnt].setBackground(
                          Color.WHITE);
          break;
        case 1 :
          canvases[cnt] =
                     new MyCanvas(cnt);
          canvases[cnt].setBackground(
                     Color.LIGHT_GRAY);
          break;
        case 2 :
          canvases[cnt] =
                     new MyCanvas(cnt);
          canvases[cnt].setBackground(
                          Color.WHITE);
          break;
        case 3 :
          canvases[cnt] =
                     new MyCanvas(cnt);
          canvases[cnt].setBackground(
                     Color.LIGHT_GRAY);
          break;
        case 4 :
          canvases[cnt] =
                     new MyCanvas(cnt);
          canvases[cnt].
            setBackground(Color.WHITE);
      }//end switch
      //Add the object to the grid.
      canvasPanel.add(canvases[cnt]);
    }//end for loop

    //Add the sub-assemblies to the
    // frame.  Set its location, size,
    // and title, and make it visible.
    getContentPane().
                   add(ctlPnl,"South");
    getContentPane().
             add(canvasPanel,"Center");

    setBounds(0,0,frmWidth,frmHeight);
```

```
    if(args == null){
      setTitle("Graph03, " +
                  "Copyright 2002, " +
                  "Richard G. Baldwin");
    }else{
      setTitle("Graph03/" + args +
                  " Copyright 2002, " +
                  "R. G. Baldwin");
    }//end else

    setVisible(true);

    //Set to exit on X-button click
    setDefaultCloseOperation(
                        EXIT_ON_CLOSE);

    //Guarantee a repaint on startup.
    for(int cnt = 0;
                   cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end constructor
  //--------------------------------//

  //This event handler is registered
  // on the JButton to cause the
  // functions to be replotted.
  public void actionPerformed(
                      ActionEvent evt){
    //Re-instantiate the object that
    // provides the data
    try{
      if(args != null){
        data = (GraphIntfc01)Class.
           forName(args).newInstance();
      }else{
        data = new junk();
      }//end else
    }catch(Exception e){
      //Known to be safe at this point.
      // Otherwise would have aborted
      // earlier.
    }//end catch

    //Set plotting parameters using
    // data from the text fields.
    xMin = Double.parseDouble(
                    xMinTxt.getText());
    xMax = Double.parseDouble(
                    xMaxTxt.getText());
    yMin = Double.parseDouble(
                    yMinTxt.getText());
    yMax = Double.parseDouble(
                    yMaxTxt.getText());
    xTicInt = Double.parseDouble(
```

```
                xTicIntTxt.getText());
    yTicInt = Double.parseDouble(
                yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(
               xCalcIncTxt.getText());

    //Calculate new values for the
    // length of the tic marks on the
    // axes.  If too small on x-axis,
    // a default value is used later.
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting areas
    for(int cnt = 0;
                cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end actionPerformed
  //--------------------------------//


//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
  int cnt;//object number
  //Factors to convert from double
  // values to integer pixel locations.
  double xScale;
  double yScale;

  MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
  }//end constructor

  //Override the paint method
  public void paint(Graphics g){

    //Get and save the size of the
    // plotting surface
    width = canvases[0].getWidth();
    height = canvases[0].getHeight();

    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);
```

```
    //Get initial data values
    double xVal = xMin;
    int oldX = getTheX(xVal);
    int oldY = 0;
    //Use the Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
      case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
      case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
      case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
      case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
      case 4 :
        oldY = getTheY(data.f5(xVal));
    }//end switch

    //Now loop and plot the points
    while(xVal < xMax){
      int yVal = 0;
      //Get next data value.  Use the
      // Canvas obj number to
      // determine which method to
      // invoke to get the value for y.
      switch(cnt){
        case 0 :
          yVal =
                getTheY(data.f1(xVal));
          break;
        case 1 :
          yVal =
                getTheY(data.f2(xVal));
          break;
        case 2 :
          yVal =
                getTheY(data.f3(xVal));
          break;
        case 3 :
          yVal =
                getTheY(data.f4(xVal));
          break;
        case 4 :
          yVal =
                getTheY(data.f5(xVal));
      }//end switch1

      //Convert the x-value to an int
      // and draw the next line segment
      int x = getTheX(xVal);
```

```
      g.drawLine(oldX,oldY,x,yVal);

      //Increment along the x-axis
      xVal += xCalcInc;

      //Save end point to use as start
      // point for next line segment.
      oldX = x;
      oldY = yVal;
    }//end while loop

  }//end overridden paint method
  //--------------------------------//

  //Method to draw axes with tic marks
  // and labels in the color RED
  void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Lable left x-axis and bottom
    // y-axis.  These are the easy
    // ones.  Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                 getTheX(xMin),
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                        getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis.  These are the hard
    // ones because the position must
    // be adjusted by the font size and
    // the number of characters.
    //Get the width of the string for
    // right end of x-axis and the
    // height of the string for top of
    // y-axis
    //Create a string that is an
    // integer representation of the
    // label for the right end of the
    // x-axis.  Then get a character
    // array that represents the
    // string.
    int xMaxInt = (int)xMax;
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.
                        toCharArray();

    //Get a FontMetrics object that can
    // be used to get the size of the
    // string in pixels.
    FontMetrics fontMetrics =
                    g.getFontMetrics();
```

```
    //Get a bounding rectangle for the
    // string
    Rectangle2D r2d =
           fontMetrics.getStringBounds(
               array,0,array.length,g);
    //Get the width and the height of
    // the bounding rectangle.  The
    // width is the width of the label
    // at the right end of the
    // x-axis.  The height applies to
    // all the labels, but is needed
    // specifically for the label at
    // the top end of the y-axis.
    int labWidth =
                 (int)(r2d.getWidth());
    int labHeight =
                (int)(r2d.getHeight());

    //Label the positive x-axis and the
    // positive y-axis using the width
    // and height from above to
    // position the labels.  These
    // labels apply to the very ends of
    // the axes at the edge of the
    // plotting surface.
    g.drawString("" + (int)xMax,
                getTheX(xMax)-labWidth,
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMax,
              getTheX(yTicLen/2)+2,
              getTheY(yMax)+labHeight);

    //Draw the axes
    g.drawLine(getTheX(xMin),
                         getTheY(0.0),
                         getTheX(xMax),
                         getTheY(0.0));

    g.drawLine(getTheX(0.0),
                        getTheY(yMin),
                        getTheX(0.0),
                        getTheY(yMax));

    //Draw the tic marks on axes
    xTics(g);
    yTics(g);
  }//end drawAxes

  //--------------------------------//

  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
```

```
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
                   getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small.  In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while

  }//end xTics
  //---------------------------------//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
```

```
    while(yDoub > yMin){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics

  //---------------------------------//

  //This method translates and scales
  // a double y value to plot properly
  // in the integer coordinate system.
  // In addition to scaling, it causes
  // the positive direction of the
  // y-axis to be from bottom to top.
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //---------------------------------//

  //This method scales a double x value
  // to plot properly in the integer
  // coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //---------------------------------//

}//end inner class MyCanvas
//===================================//

}//end class GUI
//===================================//

//Sample test class.  Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 4;
  }//end getNmbr

  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
```

```
    return (x*x)/200.0;
  }//end f3

  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
    return 100*Math.sin(x/20.0);
  }//end f5

}//end sample class junk


Listing 16
```

**Listing 17**

```
/* File GraphIntfc01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04

This interface must be implemented by classes
whose objects produce data to be plotted by
programs such as Graph03 and Graph06.

Tested using SDK 1.4.2 under WinXP.
**********************************************/

public interface GraphIntfc01{
  public int getNmbr();
  public double f1(double x);
  public double f2(double x);
  public double f3(double x);
  public double f4(double x);
  public double f5(double x);
}//end GraphIntfc01

Listing 17
```

---



**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 Discrete Cosine Transform, DCT Huffman Lempel Ziv

-end-