Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain

Baldwin illustrates and explains forward and inverse Fourier transforms using both DFT and FFT algorithms. He also illustrates and explains the implementation of frequency filtering by modifying the complex spectrum in the frequency domain and transforming the modified complex spectrum back into the time domain.

Published: November 16, 2004

By Richard G. Baldwin

Java Programming, Notes # 1485

Preface

Spectral analysis

A previous lesson entitled Fun with Java, How and Why Spectral Analysis Works explained some of the fundamentals regarding spectral analysis.

The lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm presented and explained several Java programs for doing spectral analysis, including both DFT programs and FFT programs. That lesson illustrated the fundamental aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

The lesson entitled Spectrum Analysis using Java, Frequency Resolution versus Data Length used similar Java programs to explain spectral frequency resolution.

The lesson entitled Spectrum Analysis using Java, Complex Spectrum and Phase Angle explained issues involving the complex spectrum, the phase angle, and shifts in the time domain.

This lesson will illustrate and explain forward and inverse Fourier transforms using both DFT and FFT algorithms. I will also illustrate and explain the implementation of frequency filtering

by modifying the complex spectrum in the frequency domain and then transforming the modified complex spectra back into the time domain.

Viewing tip

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

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.

Preview

In this lesson, I will present and explain the following new programs:

- **Dsp035** - 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.
- **InverseComplexToReal01** - Class that implements an inverse DFT algorithm for transforming a complex spectrum into a real time series.
- **Dsp036** - Replicates the behavior of the program named Dsp035 but uses an FFT algorithm instead of a DFT algorithm.
- **InverseComplexToRealFFT01** - Class that implements an inverse FFT algorithm for transforming a complex spectrum into a real time series.
- **Dsp037** - Illustrates filtering in the frequency domain. Uses an FFT algorithm to transform a time-domain impulse into the frequency domain. Modifies the complex spectrum, eliminating energy within a specific band of frequencies. Uses an inverse FFT algorithm to produce the filtered version of the impulse in the time domain.

In addition, I will use the following programs that I explained in the lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.

- **ForwardRealToComplex01** - Class that implements a forward DFT algorithm for transforming a real time series into a complex spectrum.
- **ForwardRealToComplexFFT01** - Class that implements a forward FFT algorithm for transforming a real time series into a complex spectrum.
- **Graph03** - Used to display various types of data. *(The concepts were explained in an earlier lesson.)*

- **Graph06** - Also used to display various types of data in a somewhat different format. *(The concepts were also explained in an earlier lesson.)*

Discussion and Sample Code

Description of the program named Dsp035

The program named **Dsp035** illustrates forward and inverse Fourier transforms using DFT algorithms.

The program performs spectral analysis on a time series consisting of pulses and a sinusoid. Then it passes the resulting real and complex parts of the spectrum to an inverse Fourier transform program. This program performs an inverse Fourier transform on the complex spectral data to reconstruct the original time series.

This program can be run with either **Graph03** or **Graph06** in order to plot the results. Enter the following at the command-line prompt to run the program with Graph03:

java Graph03 Dsp035

The program was tested using J2SE 1.4.2 under WinXP.

The order of the plotted results

When the data is plotted *(see Figure 1)* using the programs **Graph03** or **Graph06**, the plots appear in the following order from top to bottom:

- The input time series
- The real spectrum of the input time series
- The imaginary spectrum of the input time series
- The amplitude spectrum of the input time series
- The output time series produced by the inverse Fourier transform

The format of the plots

There were 256 values plotted horizontally in each section. I plotted the values on a grid that is 270 units wide to make it easier to view the plots on the rightmost end. This leaves some blank space on the rightmost end to contain the numbers, preventing the numbers from being mixed in with the plotted values. The last actual data value coincides with the rightmost tick mark on each plot.

The forward Fourier transform

A static method named **transform** belonging to the class named **ForwardRealToComplex01** was used to perform the forward Fourier transform.

*(I explained this class and the **transform** method in the earlier lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm.)*

The method named **transform** does not implement an FFT algorithm. Rather, it implements a DFT algorithm, which is more general than, but much slower than an FFT algorithm.

(See the program named Dsp036 later in the lesson for the use of an FFT algorithm.)

The inverse Fourier transform

A static method named **inverseTransform** belonging to the class named **InvereComplexToReal01** was used to perform the inverse Fourier transform. I will explain this method later in this lesson.

Let's see some results

Before getting into the technical details of the program, let's take a look at the results shown in Figure 1.

The top plot in Figure 1 shows the input time series used in this experiment.

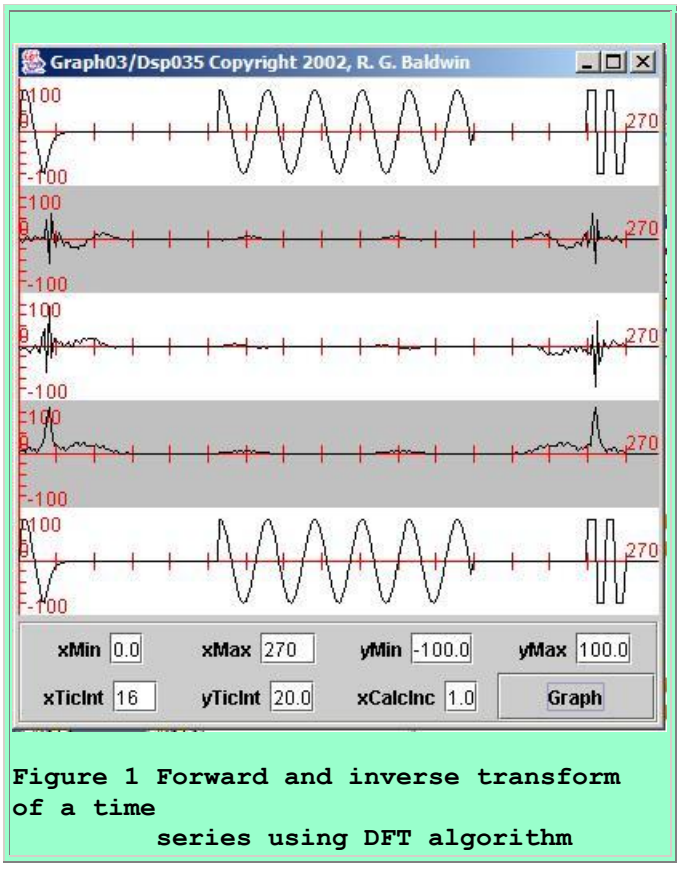

Figure 1 Forward and inverse transform of a time series using DFT algorithm

Length is a power of two

The time series is 256 samples long. Although the DFT algorithm can accommodate time series of arbitrary lengths, I set the length of this time series to a power of two so that I can compare the results with results produced by an FFT algorithm later in the lesson.

> *(Recall that most FFT algorithms are restricted to input data lengths that are a power of two.)*

The input time series

As you can see, the input time series consists of three concatenated pulses separated by blank spaces. The pulse on the leftmost end consists simply of some values that I entered into the time series to create a pulse with an interesting shape.

The middle pulse is a truncated sinusoid.

The rightmost pulse is a truncated square wave.

The objective

The objective of the experiment is to confirm that it is possible to transform this time series into the frequency domain using a forward Fourier transform, and then to recreate the time series by using an inverse Fourier transform to transform the complex spectrum back into the time domain.

The real part of the spectrum is symmetrical

The real part of the complex spectrum is shown in the second plot from the top in Figure 1. It will become important later to note that the real part of the spectrum is symmetrical about the folding frequency near the center of the plot *(at the eighth tick mark).*

Without attempting to explain why, I will simply tell you that the real part of the Fourier transform of a complex series whose imaginary part is all zeros is always symmetrical about the folding frequency.

The imaginary part of the spectrum is asymmetrical

The imaginary part of the complex spectrum is shown in the third plot from the top. Again, it will become important later to note that the imaginary part of the spectrum is asymmetrical about the folding frequency.

Once again, without attempting to explain why, the imaginary part of the Fourier transform of a complex series whose imaginary part is all zeros is always asymmetrical about the folding frequency.

The converse is also true

It is also true that the values of the imaginary part of the Fourier transform of a complex spectrum whose real part is symmetrical about the folding frequency and whose imaginary part is asymmetrical about the folding frequency will be all be zero. I will take advantage of these facts later to simplify the computing and plotting process.

The amplitude spectrum

The amplitude spectrum is shown in the fourth plot down from the top. Recall from previous lessons that the amplitude values are always positive, consisting of the square root of the sum of the squares of the real and imaginary parts.

The output time series

The output time series, produced by performing an inverse Fourier transform on the complex spectrum is shown in the bottom plot in Figure 1. Compare the bottom plot to the top plot. As you can see, they are the same, demonstrating the reversible nature of the Fourier transform.

Let's see some code

I will discuss this program in fragments. A complete listing of the program is provided in Listing 14 near the end of the lesson.

The beginning of the class for **Dsp035**, including the declaration of some variables and the creation of some array objects is shown in Listing 1. This code is straightforward.

```
class Dsp035 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 256;

  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;
```

Listing 1

The constructor

The constructor begins in Listing 2. The code in Listing 2 creates the input time series data shown in the top plot of Figure 1.

```
  public Dsp035(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
//...
//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 2

Note that I deleted much of the code from Listing 2 for brevity. You can view the missing code in Listing 14 near the end of the lesson.

Compute the complex spectrum

The code in Listing 3 invokes the static **transform** method of the **ForwardRealToComplex01** class to compute and save the complex spectrum of the time series shown in the top plot of Figure 1.

```
ForwardRealToComplex01.transform(timeDataIn,

realSpect,

imagSpect,
                                 angle,

magnitude,
                                 zero,
                                 0.0,
                                 1.0);
```

Listing 3

The method parameters

I explained the **transform** method in the earlier lesson entitled Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm. The three middle plots in Figure 1 are plots of the data returned in the arrays referred to by **realSpect**, **imagSpect**, and **magnitude** by the **transform** method.

The angle results returned by the **transform** program are not used in this lesson.

One of the parameters *(zero)* establishes that the first sample in the time series array referred to by **timeDataIn** represents the zero time origin.

The parameters also specify that the complex spectrum is to be computed at a set of equally spaced frequencies ranging from zero *(0.0)* to the sampling frequency *(1.0)*.

Perform the inverse Fourier transform

The code in Listing 4 invokes the static **inverseTransform** method of the **InverseComplexToReal01** class to perform an inverse Fourier transform on the complex spectral data, producing the output time series shown in the bottom plot in Figure 1.

```
InverseComplexToReal01.inverseTransform(
                    realSpect,
                    imagSpect,
                    timeDataOut);
  }//end constructor
```

Listing 4

I will explain the **inverseTransform** method later.

An object of the class Dsp035

Listing 4 also signals the end of the constructor. Once the constructor has completed executing, an object of the **Dsp035** class exists. The array objects belonging to the object have been populated with the original time series, the complex spectrum of the original time series, and the output time series produced by performing an inverse Fourier transform on that complex spectrum. This data is ready for plotting.

The interface methods

All of the remaining code in **Dsp035** consists of the six methods necessary to satisfy the interface named **GraphIntfc01**. Those methods are required to provide data to the plotting program, as explained in earlier lessons in this series.

If you have studied the previous lessons in this series, you probably don't want to hear any more about those methods, so I won't discuss them further. You can view the six interface methods in Listing 14 near the end of the lessons.

The inverseTransform method of the InverseComplexToReal01 class

The static method named **inverseTransform** performs a complex-to-real inverse discrete Fourier transform returning a real result only. In other words, the method transforms a complex input to a real output.

There are more efficient ways to write this method taking known symmetry and asymmetry conditions into account. However, I wrote the method the way that I did because I wanted it to mimic the behavior of an FFT algorithm. Therefore, the complex input must extend from zero to the sampling frequency.

The method does not implement an FFT algorithm. Rather, the **inverseTransform** method implements a straight forward sampled-data version of the continuous inverse Fourier transform that is defined using integral calculus.

The parameters

The parameters to the **inverseTransform** are:

- double[] realIn - incoming real data
- double[] imagIn - incoming imag data
- double[] realOut - outgoing real data

The method considers the data length to be **realIn.length**, and considers the computational time increment to be **1.0/realIn.length**.

Assumptions

The method returns a number of points equal to the data length. It assumes that the real input consists of positive frequency points for a symmetric real frequency function. That is, the real input is assumed to be symmetric about the folding frequency. The method does not test this assumption.

The method assumes that imaginary input consists of positive frequency points for an asymmetric imaginary frequency function. That is, the imaginary input is assumed to be asymmetric about the folding frequency. Once again, the method does not test this assumption.

A real output

A symmetric real part and an asymmetric imaginary part guarantee that the imaginary output will be all zero values. Having made that assumption, the program makes no attempt

to compute an imaginary output. If the assumptions described above are not valid, the results won't be valid.

The program was tested using J2SE v1.4.2 under WinXP.

The inverseTransform method

The beginning of the class and the beginning of the static **inverseTransform** method is shown in Listing 5.

```
public class InverseComplexToReal01{

  public static void inverseTransform(

double[] realIn,

double[] imagIn,

double[] realOut){

    int dataLen = realIn.length;
    double delT = 1.0/realIn.length;
    double startTime = 0.0;
```

Listing 5

Listing 5 declares and initializes some variables that will be used later.

The inverse transform computation

Listing 6 contains a pair of nested **for** loops that perform the actual inverse transform computation.

```
    //Outer loop interates on time
domain
    // values.
    for(int i=0; i < dataLen;i++){
      double time = startTime +
i*delT;
      double real = 0;
      //Inner loop iterates on
frequency
      // domain values.
      for(int j=0; j < dataLen; j++){
        real += realIn[j]*

Math.cos(2*Math.PI*time*j)
            + imagIn[j]*
```

```
Math.sin(2*Math.PI*time*j);
      }//end inner loop
      realOut[i] = real;
    }//end outer loop
  }//end inverseTransform

}//end class InverseComplexToReal01
```

Listing 6

If you have been studying the earlier lessons in this series, you should be able to understand the code in Listing 6 without further explanation. Pay particular attention to the comments that describe the two **for** loops.

The program named Dsp036

The program named **Dsp036** replicates the behavior of the program named **Dsp035**, except that it uses an FFT algorithm to perform the inverse Fourier transform instead of using a DFT algorithm as in **Dsp035**.

The output from Dsp036

The output produced by running the program named **Dsp036** and plotting the output using the program named **Graph03** is shown in Figure 2.

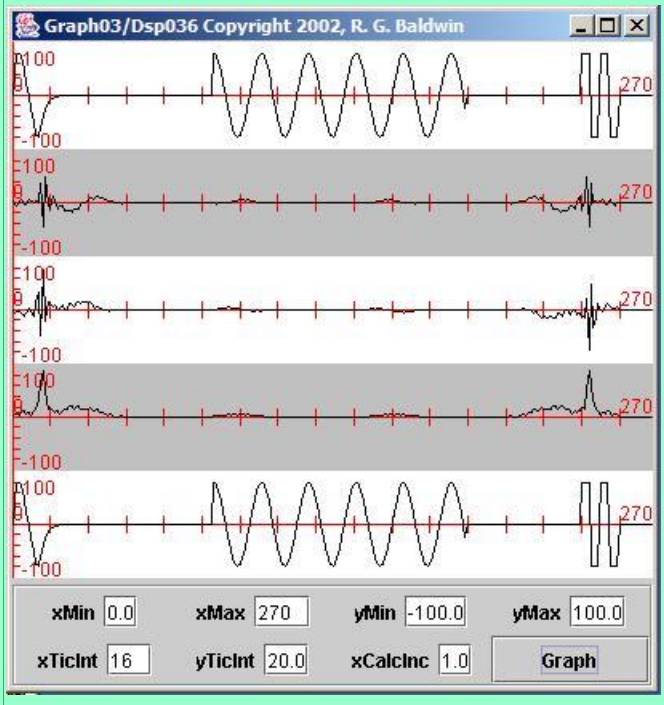

Figure 2 Forward and inverse transform of a time

```
series using FFT algorithm
```

Compare Figure 2 with Figure 1. The two should be identical. The program named **Dsp036** was designed to use an FFT algorithm for the inverse Fourier transform and to replicate the behavior of the program named **Dsp035**, which uses a DFT algorithm for the inverse Fourier transform. In addition, the same plotting parameters were used for both figures.

The code

I'm only going to show you one short code fragment from the program named **Dsp036**. Listing 7 shows the code that invokes the methods to perform the forward and inverse Fourier transforms using the FFT algorithm. A complete listing of the program named **Dsp036** is shown in Listing 16 near the end of the lesson.

```
    //Compute FFT of the time data and
save it in
    // the output arrays.

ForwardRealToComplexFFT01.transform(

timeDataIn,

realSpect,

imagSpect,

angle,

magnitude);

    //Compute inverse FFT of the
spectral data
    InverseComplexToRealFFT01.

inverseTransform(

realSpect,

imagSpect,

timeOut);
```

Listing 7

The forward Fourier transform

The **transform** method used to perform the forward Fourier transform in Listing 7 was discussed in an earlier lesson, so I won't discuss it further here.

The inverse Fourier transform

The static **inverseTransform** method of the **InverseComplexToRealFFT01** class was used to perform the inverse Fourier transform in Listing 7. You can view this method in Listing 17 near the end of the lesson.

I'm not going to discuss this method in detail either, because it is very similar to the method named **InverseComplexToReal01** discussed earlier in conjunction with Listing 4 and the listings following that one.

A couple of things to note

There are a couple of things, however, that I do want to point out.

The **transform** method and the **inverseTransform** method each invoke a method named **complexToComplex** to actually perform the Fourier transform. This method implements a classical FFT algorithm accepting complex input data and producing complex output data. The restriction of real-to-complex and complex-to-real is imposed by the methods named **transform** and **inverseTransform**.

> *(The method named **complexToComplex** is also suitable for use if you have a need to perform complex-to-complex Fourier transforms.)*

The signature of the complexToComplex method

The signature for the **complexToComplex** method is shown in Figure 3.

```
  public static void complexToComplex(
                                     int
sign,
                                     int
len,

double real[],

double imag[]){
```

Figure 3

The **complexToComplex** method can be used to perform either a forward or an inverse transform. The value of the first parameter determines whether the method performs a forward or an inverse Fourier transform.

The first parameter of the complexToComplex method

A value of +1 for the first parameter causes the **complexToComplex** method to perform a forward Fourier transform.

A value of -1 for the first parameter causes the **complexToComplex** method to perform an inverse Fourier transform.

The forward transform

Although I didn't include the code in this lesson, *(because it was shown in an earlier lesson),* the **transform** method in Figure 7 passes a value of +1 to the **complexToComplex** method to cause it to perform a forward Fourier transform.

The inverse transform

Similarly, the **inverseTransform** method shown in Listing 17 passes a value of -1 to the **complexToComplex** method to cause it to perform an inverse Fourier transform.

FFT and DFT produce equivalent results

As evidenced in Figure 1 and Figure 2, the program named **Dsp035**, which uses a DFT algorithm, produces the same results as the program named **Dsp036**, which uses an FFT algorithm. However, if you were to put a timer on each of the programs, you would find that **Dsp036** runs faster due to the improved speed of the FFT algorithm over the DFT algorithm.

Using a Fourier transform to perform frequency filtering

The program named **Dsp037** illustrates frequency filtering accomplished by modifying the complex spectrum in the frequency domain and then performing an inverse Fourier transform on the modified frequency-domain data. The results are shown in Figure 4.

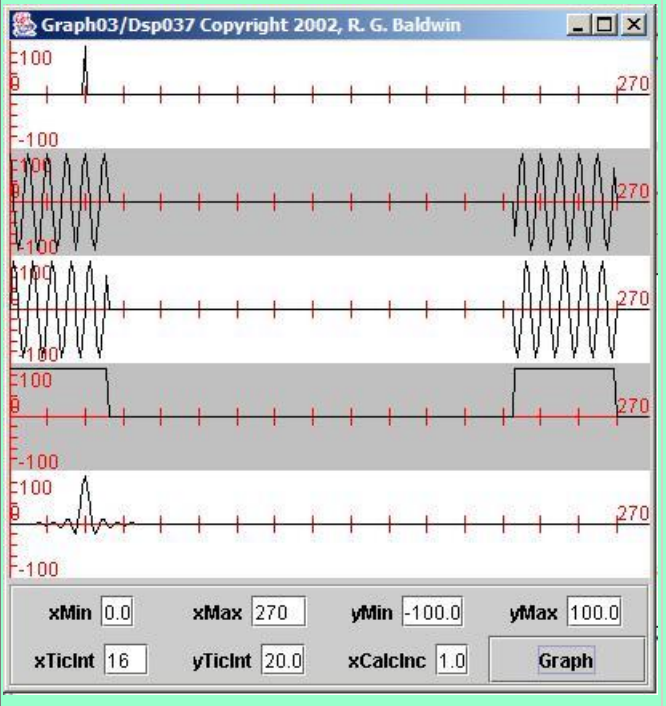

Figure 4 Illustrates filtering in the frequency

```
domain
```

Operation of the program

The program begins by using an FFT algorithm to perform a forward Fourier transform on a single impulse in the time domain.

> *(A DFT algorithm could have been used equally as well, but it would have been slower.)*

The impulse is shown as the input time series in the topmost plot in Figure 4.

> *(Although I didn't show the complex spectrum of the impulse, we know that the magnitude of the spectrum of an impulse is constant across all frequencies. In other words, the magnitude spectrum of an impulse is a flat line from zero to the sampling frequency and above.)*

Modify the complex spectrum

Then the program eliminates all energy between one-sixth and five-sixths of the sampling frequency by setting the real and imaginary parts of the FFT output to zero.

The second, third, and fourth plots in Figure 4 show the real part, imaginary part, and amplitude respectively of the modified complex spectrum.

> *(The two boxes in the fourth plot in Figure 4 show what's left of the spectral energy after the energy in the middle of the band has been eliminated.)*

The folding frequency in these three plots is near the center of the plot at the eighth tick mark.

The plotting format

The input data length was 256 samples. All but one of the input data values was set to zero resulting in a single impulse in the input time series near the second tick mark in Figure 4.

> *(The real and complex parts of the frequency spectrum were computed at 256 frequencies between zero and the sampling frequency.)*

There were 256 values plotted horizontally in each separate plot. Once again, to make it easier to view the plots on the rightmost end, I plotted the values on a grid that is 270 units wide. This leaves some blank space on the rightmost end to contain the numbers, thus preventing the numbers from being mixed in with the plotted values. The last actual data value coincides with the rightmost tick mark on each plot.

Perform an inverse Fourier transform

After modifying the complex spectrum as described above, the program performs an inverse Fourier transform on the modified complex spectrum to produce the filtered impulse.

The filtered impulse

The filtered impulse is shown as the bottom plot in Figure 4. As you can see, the pulse is smeared out in time relative to the input pulse in the top plot. This is the typical result of reducing the bandwidth of a pulse.

> *(This particular modification of the complex spectrum resulted in a filtered pulse that has the waveform of a SIN(X)/X function. A different modification of the complex spectrum would have resulted in a filtered pulse with a different waveform.*
>
> *This example also illustrates one of the miracles of digital signal processing. Energy appears in the output before it occurs in the input. Obviously that is not possible in the real world of analog systems, but many things are possible in the digital world that are not possible in the real world.)*

The code for Dsp037

Listing 8 shows the beginning of the class definition for the program named **Dsp037**.

```
class Dsp037 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 256;

  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[] timeOut = new double[len];
```

Listing 8

Listing 8 simply declares and initializes some variables that will be used later.

The constructor

The constructor begins in Listing 9.

```
  public Dsp037(){//constructor

    timeDataIn[32] = 90;

Listing 9
```

Listing 9 creates the raw pulse data shown in the topmost plot in Figure 4.

When the array object referred to by **timeDataIn** is created, the values of all array elements are set to zero by default. Listing 9 modifies one of the elements to have a value of 90. This results in a single impulse at an index of 32.

Compute the Fourier transform

Still in the constructor, the code in Listing 10 uses an FFT algorithm in the method named **transform** *(discussed earlier)* to compute the Fourier transform of the impulse.

```
    //Compute FFT of the time data and
save it in
    // the output arrays.

ForwardRealToComplexFFT01.transform(

timeDataIn,

realSpect,

imagSpect,

angle,

magnitude);

Listing 10
```

The results of the Fourier transform are stored in the array objects referred to by **realSpect**, **imagSpect**, and **magnitude**.

(The phase angle is also computed but is of no interest in this example.)

Apply the filter to the frequency data

Listing 11 applies the filter by setting sample values in a portion of the real and imaginary parts of the complex spectrum to zero.

```
    for(int cnt = len/6;cnt <
5*len/6;cnt++){
```

```
      realSpect[cnt] = 0.0;
      imagSpect[cnt] = 0.0;
    }//end for loop

Listing 11
```

This code eliminates all energy between one-sixth and five-sixths of the sampling frequency. The modified data for the real and imaginary parts of the complex spectrum are shown in the second and third plots in Figure 4.

Recompute the magnitude

Listing 12 recomputes the magnitude values for the modified real and imaginary values of the complex spectrum.

```
    //Recompute the magnitude based on
the
    // modified real and imaginary
spectra.
    for(int cnt = 0;cnt < len;cnt++){
      magnitude[cnt] =
        (Math.sqrt(

realSpect[cnt]*realSpect[cnt]
           +
imagSpect[cnt]*imagSpect[cnt])/len);
    }//end for loop

Listing 12
```

The modified data for the amplitude of the complex spectrum are shown in the fourth plot in Figure 4.

Compute the inverse Fourier transform

Listing 13 uses the **inverseTransform** method to compute the inverse Fourier transform of the modified complex spectrum stored in **realSpect** and **imagSpect**. The results of the inverse transform are stored in **timeOut**.

```


InverseComplexToRealFFT01.inverseTransform(

realSpect,

imagSpect,

timeOut);
  }//end constructor
```

```
Listing 13
```

The results of the inverse transform are shown in the bottom plot in Figure 4.

Listing 13 also signals the end of the constructor.

Display the results

Once the constructor returns, all of the data that is to be plotted has been stored in the various array objects. The remaining code in the program consists of the definition of the six methods required by the interface named **GraphIntfc01**. These methods are required to make it possible to use the program named **Graph03** to plot the results as shown in Figure 4.

I have discussed these methods on numerous previous occasions, and won't repeat that discussion here.

One more example, Dsp038

Figure 5 illustrates one more example of performing frequency filtering by modifying the complex spectrum and then performing an inverse transform on the modified spectrum.

While discussing the program named **Dsp037**, I told you that performing a different modification on the complex spectrum would result in a different waveform for the filtered impulse. The program named **Dsp038** applies a different modification to the complex spectrum, but is otherwise the same as **Dsp037**.

> *(Because of the similarity of the two programs, I won't discuss the code in **Dsp038**. You can view that code in Listing 19 near the end of the lesson.)*

Figure 5 shows the output produced by the program named **Dsp038**.

```

```

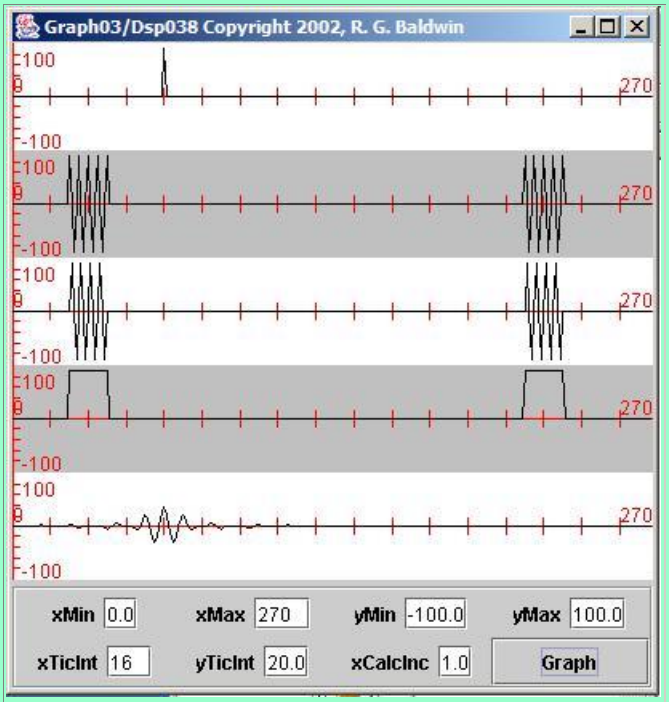

Figure 5 Illustrates filtering in the frequency domain

Compare Figure 5 with Figure 4

The basic plotting format of Figure 5 is the same as Figure 4.

The first difference to note between the two figures is that I moved the impulse in the input time series in the topmost plot sixteen samples further to the right in **Dsp038**.

> *(This has no impact on the final result, which you can verify by modifying the program to move the impulse to a different position and then compiling and running the modified program.)*

Compare the bandwidth of the pass band

The second difference to note is shown in the modified amplitude spectrum in the fourth plot in the two figures. The bandwidth of the pass band is significantly narrower in Figure 5 than in Figure 4. Also, the pass band in Figure 4 extends all the way down to zero frequency, while Figure 5 eliminates all energy below a frequency of three thirty-seconds of the sampling frequency.

Waveforms of filtered impulse

Finally, note the waveforms of the two filtered impulses. The overall amplitude of the filtered impulse in Figure 5 is less than in Figure 4, simply because it contains less total energy. In

addition, the filtered impulse in Figure 5 is broader than the filtered impulse in Figure 4. This is because it has a narrower bandwidth.

> *(Pulses that are narrow in terms of time duration require a wider bandwidth than pulses that have a longer time duration. The time duration of the pulse tends to be inversely related to the bandwidth of the pulse.)*

Run the Programs

I encourage you to copy, compile, and run the programs provided in this lesson. Experiment with them, making changes and observing the results of your changes.

Create more complex experiments. For example, use more complex input time series when experimenting with frequency filtering. Apply different modifications to the complex spectrum when experimenting with frequency filtering.

Most of all enjoy yourself and learn something in the process.

Summary

This lesson illustrates and explains forward and inverse Fourier transforms using both DFT and FFT algorithms.

The lesson also illustrates and explains the implementation of frequency filtering by modifying the complex spectrum in the frequency domain and then transforming the modified complex spectrum back into the time domain.

Complete Program Listings

Complete listings of the programs discussed in this lesson follow.

```
import java.util.*;

class Dsp035 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 256;

  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 Dsp035(){//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 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 spectral 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 Dsp035
```

Listing 14

```
/*File InverseComplexToReal01.java
Copyright 2004, R.G.Baldwin
Rev 5/24/04

Although there are more efficient ways to write
this program, it was written the way it was to
mimic the behavior of an FFT algorithm.
Therefore, the complex input must extend from
zero to the sampling frequency.

The static method named inverseTransform performs
a complex to real inverse discrete Fourier
transform returning a real result only.  In other
words, the method transforms a complex input to a
real output.

Does not implement the FFT algorithm. Implements
```

```
a straight-forward sampled-data version of the
continuous inverse Fourier transform defined
using integral calculus.

The parameters are:
double[] realIn - incoming real data
double[] imagIn - incoming imag data
double[] realOut - outgoing real data

Considers the data length to be
 realIn.length
Computational time increment is
 1.0/realIn.length

Returns a number of points equal to the data
length.

Assumes real input consists of positive
frequency points for a symmetric real frequency
function.  That is, the real input is assumed to
be symmetric about the folding frequency.  Does
not test this assumption.

Assumes imaginary input consists of positive
frequency points for an asymmetric imaginary
frequency function.  That is, the imaginary input
is assumed to be asymmetric about  the
folding frequency.  Does not test this
assumption.

The assumption of a symmetric real part and an
asymmetric imaginary part guarantees that the
imaginary output would be all zero if it were to
be computed.  Thus the program makes no attempt
to compute an imaginary output.

Tested using J2SE v1.4.2 under WinXP.
**********************************************/

public class InverseComplexToReal01{

  public static void inverseTransform(
                                double[] realIn,
                                double[] imagIn,
                                double[] realOut){
    int dataLen = realIn.length;
    double delT = 1.0/realIn.length;
    double startTime = 0.0;
    //Outer loop interates on time domain
    // values.
    for(int i=0; i < dataLen;i++){
      double time = startTime + i*delT;
      double real = 0;
      //Inner loop iterates on frequency
      // domain values.
      for(int j=0; j < dataLen; j++){
```

```
        real += realIn[j]*
                      Math.cos(2*Math.PI*time*j)
             + imagIn[j]*
                      Math.sin(2*Math.PI*time*j);
      }//end inner loop
      realOut[i] = real;
    }//end outer loop
  }//end inverseTransform

}//end class InverseComplexToReal01
```

Listing 15

```
/* File Dsp036.java
Copyright 2004, R.G.Baldwin
Revised 5/24/04

Illustrates forward and inverse Fourier
 transforms using FFT algorithms.

Performs spectral analysis on a time series
consisting of pulses and a sinusoid.

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

Run with Graph03.

Tested using J2SE 1.4.2 under WinXP.
**********************************************/
import java.util.*;

class Dsp036 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 256;

  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[] timeOut = new double[len];

  public Dsp036(){//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 FFT of the time data and save it in
    // the output arrays.
    ForwardRealToComplexFFT01.transform(
                                    timeDataIn,
                                    realSpect,
                                    imagSpect,
                                    angle,
                                    magnitude);

    //Compute inverse FFT of spectral data
    InverseComplexToRealFFT01.
                                inverseTransform(
                                    realSpect,
                                    imagSpect,
                                    timeOut);
```

```
  }//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]/len;
    }//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]/len;
    }//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 > timeOut.length-1){
      return 0;
```

```
    }else{
      //scale for convenient viewing
      return timeOut[index]/len;
    }//end else
  }//end function

}//end sample class Dsp036
```

Listing 16

```
/*File InverseComplexToRealFFT01.java
Copyright 2004, R.G.Baldwin
Rev 5/24/04

The static method named inverseTransform performs
a complex to real Fourier transform using a
complex-to-complex FFT algorithm.  A specific
parameter is passed to the FFT algorithm that
causes this to be an inverse Fourier transform.

See InverseComplexToReal01 for a version that
does not use an FFT algorithm but uses a DFT
algorithm instead.

Incoming parameters are:
  double[] realIn - incoming real data
  double[] imagIn - incoming imaginary data
  double[] realOut - outgoing real data

Requires spectral input data extending from zero
to the sampling frequency.

Assumes real input consists of positive
frequency points for a symmetric real frequency
function.  That is, the real input is assumed to
be symmetric about the folding frequency.  Does
not test this assumption.

Assumes imaginary input consists of positive
frequency points for an asymmetric imaginary
frequency function.  That is, the imaginary input
is assumed to be asymmetric about  the
folding frequency.  Does not test this
assumption.

The assumption of a symmetric real part and an
asymmetric imaginary part guarantees that the
imaginary output is all zeros.  Thus, the
program does not return an imaginary output.
Does not test the assumption that the imaginary
is all zeros.

CAUTION: THE INCOMING DATA LENGTH MUST BE A
POWER OF TWO. OTHERWISE, THIS PROGRAM WILL FAIL
TO RUN PROPERLY.
```

```
Returns a number of points equal to the incoming
data length.  Those points are uniformly
distributed beginning at zero.
********************************************/

public class InverseComplexToRealFFT01{

  public static void inverseTransform(
                            double[] realIn,
                            double[] imagIn,
                            double[] realOut){
    double pi = Math.PI;//for convenience
    int dataLen = realIn.length;
    double[] imagOut = new double[dataLen];
    //The complexToComplex FFT method does an
    // in-place transform causing the output
    // complex data to be stored in the arrays
    // containing the input complex data.
    // Therefore, it is necessary to copy the
    // input data into the output arrays before
    // passing them to the FFT algorithm.
    System.arraycopy(realIn,0,realOut,0,dataLen);
    System.arraycopy(imagIn,0,imagOut,0,dataLen);

    //Perform the spectral analysis.  The results
    // are stored in realOut and imagOut.  Note
    // that the -1 value for the first
    // parameter causes the transform to be an
    // inverse transform.  A +1 value would cause
    // it to be a forward transform.
    complexToComplex(-1,dataLen,realOut,imagOut);

  }//end inverseTransform method
  //-----------------------------------------//

  //This method computes a complex-to-complex
  // FFT.  The value of sign must be 1 for a
  // forward FFT and -1 for an inverse FFT.
  public static void complexToComplex(
                                int sign,
                                int len,
                                double real[],
                                double imag[]){
    double scale = 1.0;
    //Reorder the input data into reverse binary
    // order.
    int i,j;
    for (i=j=0; i < len; ++i) {
      if (j>=i) {
        double tempr = real[j]*scale;
        double tempi = imag[j]*scale;
        real[j] = real[i]*scale;
        imag[j] = imag[i]*scale;
        real[i] = tempr;
        imag[i] = tempi;
```

```
      }//end if
      int m = len/2;
      while (m>=1 && j>=m) {
        j -= m;
        m /= 2;
      }//end while loop
      j += m;
    }//end for loop

    //Input data has been reordered.
    int stage = 0;
    int maxSpectraForStage,stepSize;
    //Loop once for each stage in the spectral
    // recombination process.
    for(maxSpectraForStage = 1,
                stepSize = 2*maxSpectraForStage;
                maxSpectraForStage < len;
                maxSpectraForStage = stepSize,
                stepSize = 2*maxSpectraForStage){
      double deltaAngle =
                 sign*Math.PI/maxSpectraForStage;
      //Loop once for each individual spectra
      for (int spectraCnt = 0;
                 spectraCnt < maxSpectraForStage;
                 ++spectraCnt){
        double angle = spectraCnt*deltaAngle;
        double realCorrection = Math.cos(angle);
        double imagCorrection = Math.sin(angle);

        int right = 0;
        for (int left = spectraCnt;
                   left < len;left += stepSize){
          right = left + maxSpectraForStage;
          double tempReal =
                     realCorrection*real[right]
                     - imagCorrection*imag[right];
          double tempImag =
                     realCorrection*imag[right]
                     + imagCorrection*real[right];
          real[right] = real[left]-tempReal;
          imag[right] = imag[left]-tempImag;
          real[left] += tempReal;
          imag[left] += tempImag;
        }//end for loop
      }//end for loop for individual spectra
      maxSpectraForStage = stepSize;
    }//end for loop for stages
  }//end complexToComplex method

}//end class InverseComplexToRealFFT01
```

Listing 17

```
/* File Dsp037.java
Copyright 2004, R.G.Baldwin
```

```
Revised 5/24/04

Illustrates filtering in the frequency domain.
Performs FFT on an impulse.  Eliminates all
energy between one-sixth and five-sixths of the
sampling frequency by modifying the real and
imaginary parts of the FFT output.  Then performs
inverse FFT to produce the filtered impulse.

Run with Graph03.

Tested using J2SE 1.4.2 under WinXP.
*********************************************/
import java.util.*;

class Dsp037 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 256;

  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[] timeOut = new double[len];

  public Dsp037(){//constructor

    //Create the raw data pulse
    timeDataIn[32] = 90;

    //Compute FFT of the time data and save it in
    // the output arrays.
    ForwardRealToComplexFFT01.transform(
                                    timeDataIn,
                                    realSpect,
                                    imagSpect,
                                    angle,
                                    magnitude);

    //Apply the frequency filter eliminating all
    // energy between one-sixth and five-sixths
    // of the sampling frequency by modifying the
    // real and imaginary parts of the spectrum.
    for(int cnt = len/6;cnt < 5*len/6;cnt++){
      realSpect[cnt] = 0.0;
      imagSpect[cnt] = 0.0;
    }//end for loop

    //Recompute the magnitude based on the
    // modified real and imaginary spectra.
    for(int cnt = 0;cnt < len;cnt++){
      magnitude[cnt] =
        (Math.sqrt(
           realSpect[cnt]*realSpect[cnt]
```

```
          + imagSpect[cnt]*imagSpect[cnt])/len);
    }//end for loop

    //Compute inverse FFT of modified spectral
    // data.
    InverseComplexToRealFFT01.inverseTransform(
                                        realSpect,
                                        imagSpect,
                                        timeOut);
  }//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{
      return 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{
            return 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 len*magnitude[index];
    }//end else
```

```
  }//end function
  //-----------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                  index > timeOut.length-1){
      return 0;
    }else{
      //scale for convenient viewing
      return 3.0*timeOut[index]/len;
    }//end else
  }//end function

}//end sample class Dsp037
```

Listing 18

```
/* File Dsp038.java
Copyright 2004, R.G.Baldwin
Revised 5/24/04

Illustrates filtering in the frequency domain.
Performs FFT on an impulse.  Modifies the
complex spectrum.  Then performs inverse FFT
to produce the filtered impulse.

Run with Graph03.

Tested using J2SE 1.4.2 under WinXP.
**********************************************/
import java.util.*;

class Dsp038 implements GraphIntfc01{
  final double pi = Math.PI;

  int len = 256;

  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[] timeOut = new double[len];

  public Dsp038(){//constructor

    //Create the raw data pulse
    timeDataIn[64] = 90;

    //Compute FFT of the time data and save it in
    // the output arrays.
    ForwardRealToComplexFFT01.transform(
                                    timeDataIn,
                                    realSpect,
                                    imagSpect,
```

```
                                      angle,
                                      magnitude);

    //Apply the frequency filter.
    for(int cnt = 0;cnt <= len/2;cnt++){
      if(cnt < 3*len/32){
        realSpect[cnt] = 0;
        imagSpect[cnt] = 0;
      }//end if

      if(cnt > 5*len/32){
        realSpect[cnt] = 0;
        imagSpect[cnt] = 0;
      }//end if

      //Fold complex spectral data
      if(cnt > 0){
        realSpect[len - cnt] = realSpect[cnt];
      }//end if
      if(cnt > 0){
        imagSpect[len - cnt] = -imagSpect[cnt];
      }//end if
    }//end for loop

    //Recompute the magnitude based on the
    // modified real and imaginary spectra.
    for(int cnt = 0;cnt < len;cnt++){
      magnitude[cnt] =
        (Math.sqrt(
           realSpect[cnt]*realSpect[cnt]
           + imagSpect[cnt]*imagSpect[cnt])/len);
    }//end for loop

    //Compute inverse FFT of modified spectral
    // data.
    InverseComplexToRealFFT01.inverseTransform(
                                        realSpect,
                                        imagSpect,
                                        timeOut);
  }//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{
      return 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{
            return 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 len*magnitude[index];
    }//end else
  }//end function
  //-----------------------------------------//
  public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
                   index > timeOut.length-1){
      return 0;
    }else{
      //scale for convenient viewing
      return 3.0*timeOut[index]/len;
    }//end else
  }//end function

}//end sample class Dsp038
```

Listing 19

About the author

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

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

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

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

Baldwin@DickBaldwin.com

-end-