Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm

Baldwin explains the underlying signal processing concepts that make the Fast Fourier Transform (FFT) algorithm possible.

Published: January 4, 2005
By Richard G. Baldwin

Java Programming, Notes # 1486

Preface

Programming in Java doesn't have to be dull and boring. In fact, it's possible to have a lot of fun while programming in Java. This lesson is one in a series that concentrates on having fun while programming in Java.

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.

General Discussion

The purpose of this lesson is to help you to understand how the Fast Fourier Transform (FFT) algorithm works. In order to understand the FFT, you must first understand the Discrete Fourier

Transform (DFT). I explained how the DFT works in an earlier lesson titled Fun with Java, How and Why Spectral Analysis Works.

There are several different FFT algorithms in common use. In addition, there are many sites on the web where you can find explanations of the mechanics of FFT algorithm. I won't replicate those explanations. Rather, I will explain the underlying concepts that make the FFT possible and illustrate those concepts using a simple program. Hopefully, once you understand the underlying concepts, one or more of the explanations of the mechanics that you find on other sites will make sense to you.

A general-purpose transform

The Fourier transform is most commonly associated with its use in transforming time-domain data into frequency-domain data. However, it is important to understand that there is nothing inherent in the Fourier transform regarding either the time domain or the frequency domain. Rather, the Fourier transform is a general-purpose transform that is used to transform a set of complex data in one domain into a different set of complex data in another domain. It is purely happenstance that it happens to be so valuable in describing the relationship between the time domain and the frequency domain.

Transforming from space domain to wave number domain

For example, my first job after earning a BSEE degree in 1962 was in the Seismic Research Department of Texas Instruments. That is where I had my first encounter with Digital Signal Processing. In that job, I did a lot of work with Fourier transforms involving the time domain and the frequency domain. I also did a lot of work with Fourier transforms involving the space domain and the wave-number domain.

Wave number is the name given to the reciprocal of wavelength for compression and shear waves propagating through a medium such as an iron bar, earth, water, or air, and also for electromagnetic waves such as radio and radar propagating through space.

> *(Those familiar with the subject will know that while compression waves will propagate through water and air, those media won't support shear waves.)*

Two-dimensional Fourier transforms

For example, one of the things that we did was to compute two-dimensional Fourier transforms on diagrams representing weighted points in two-dimensional space. We would transform the weighted points in the space domain into points in the wave-number domain.

The weighted points in the space domain represented the locations and amplifications of seismometers in a two-dimensional array on the surface of the earth. Each seismometer was amplified by a different gain factor and polarity. The amplified outputs of the seismometers were added together in various and complex ways intended to enhance signals and suppress noise.

Wave-number response to seismic waves

In this case, the wave number was the reciprocal of the wave length of seismic waves propagating across the array. By plotting the results of the transformation in the wave-number domain, we could estimate which seismic waves would be enhanced and which seismic waves would be suppressed by the processing being applied to the seismometer outputs.

We could also perform experiments on the computer where we caused the weights to vary with frequency, thus, allowing us to design and place digital filters on the seismometers to optimize the response of the array to earthquake signals while suppressing seismic noise associated with nearby cities and other sources of seismic noise.

A general purpose mathematical transform

I mention all of this simply to illustrate the general nature of the Fourier transform. Once again, the Fourier transform is simply a mathematical process that can be used to transform a set of complex values in one domain into a set of complex values in a different domain.

Before getting into the details of this discussion, I want to refer you to a couple of excellent references on the FFT. Of course, you can find many more by performing a Google search for the keyword FFT.

Fourier transform images

Many of the images that you will see in this lesson were produced using an applet that I downloaded from http://sepwww.stanford.edu/oldsep/hale/FftLab.html. I changed the name of the applet class to cause it to fit into my file-naming scheme. I also made a couple of minor modifications to force its output to fit into this narrow publication format. Otherwise, I used the applet in its original form. This applet is extremely useful in performing FFT experiments very quickly and easily. I strongly recommend that you become familiar with it.

Information on the FFT algorithm

I also want to refer you to Chapter 12 of the excellent online book titled *The Scientist and Engineer's Guide to Digital Signal Processing* by Steven W. Smith, Ph.D. You will find the book at http://www.dspguide.com/pdfbook.htm. This book contains a wealth of information, including Smith's explanation of the mechanics of the FFT algorithm.

Will discuss underlying concepts

As mentioned earlier, the FFT algorithm is very complicated. I won't discuss the mechanics of the algorithm in this lesson. Rather, I will explain the underlying concepts that make the FFT algorithm possible.

Hopefully after reading my explanation of the basic concepts, you will be able to understand the explanation of the mechanics of the algorithm provided by Smith and others.

A linear transform

The FFT algorithm is an algorithm that takes advantage of several reasonably well-know facts along with some less well-known facts.

One of those facts is that the Fourier transform is a linear transform. By this, I mean that the transform of the sum of two or more input series is equal to the sum of the transforms of the individual input series. I will attempt to illustrate this in Figure 1, Figure 2, and Figure 3.

Images produced by the FFT applet

The images in these figures were produced using the FFT applet mentioned earlier.

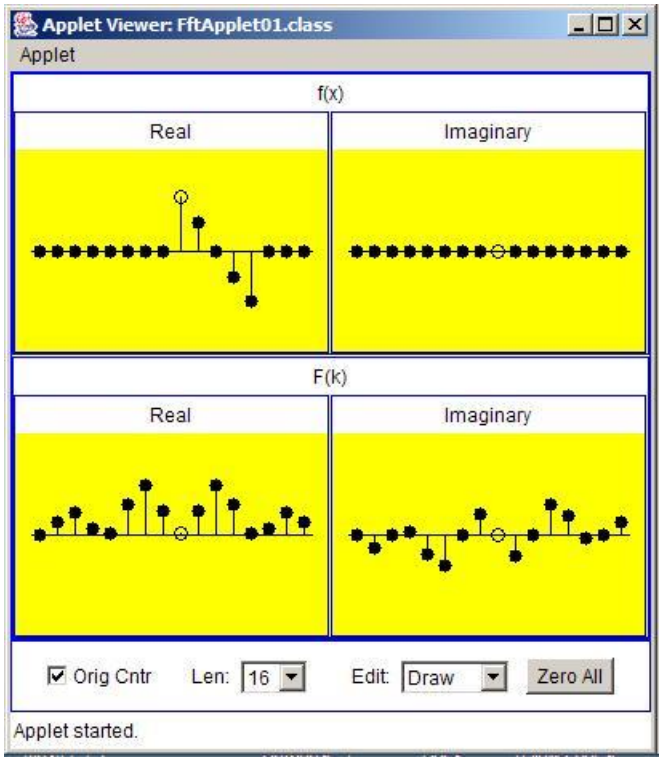

Figure 1 Transform of pulse with negative slope.

An examination of Figure 1 shows that the display produced by the applet contains two sections. One section is labeled f(x) and the other section is labeled F(k).

This is an interactive applet with the ability to transform the complex samples represented by f(x) into complex samples represented by F(k). Alternatively, the applet can be used to transform complex samples represented by F(k) into complex samples represented by f(x).

Real and imaginary sections

Each section contains two boxes, one labeled **Real** and the other labeled **Imaginary**. One box contains a visual representation of a set of real samples and the other box contains a visual representation of a set of imaginary samples.

With one exception, each sample is represented by a black circle. In each box, one of the samples is represented by an empty circle. The empty circle represents an index value of zero. Samples to the right of the sample with the empty circle are samples at positive indices, and samples to the left of the sample with the empty circle are samples at negative indices.

A complex sample

A pair of values, one taken from the Real box and one taken from the Imaginary box, represents a complex sample. Any of the circles can be interactively moved up or down with the mouse. The value of each sample is represented by the distance of the corresponding circle from the horizontal line.

When a change is made to the value of any sample belonging to either f(x) for F(k), the transformation is recomputed and the display of the other function is modified accordingly. If you modify the value of a sample in f(x), the values in F(k) are automatically modified to show the Fourier transform of f(x). If you modify the value of a sample in F(k), the values in f(x) are automatically modified to show the inverse Fourier transform of F(k).

This is an extremely powerful interactive tool.

Powers of two

Most FFT algorithms require the input series to contain a number of complex samples that is a power of two such as 2, 4, 8, 16, 32, etc. Most FFT algorithms also produce the same number of complex samples in the output as are provided in the input. The FFT algorithm used in this applet is no exception to those rules.

A pull-down list at the bottom of the applet lets the user specify 16, 32, or 64 complex samples for both the input and the output. All of the examples in this lesson use 16 complex samples for input and output.

Location of the origin

The applet also provides a check box that allows the user to cause the origin *(the empty circle at index value zero)* to either be centered or placed at the left end. The display in Figure 1 has the origin centered. Other displays that I will use later have the origin at the left end.

Other applet controls

The other pull-down list and the button at the bottom of the applet provide other control features that don't need to be discussed here. I strongly urge you to download this applet and experiment with it. The results can be very enlightening.

Back to the concept of the linear transform

Having discussed the features of the interactive FFT tool that I used to produce many of the images in this lesson, it is time to get back to the discussion of the Fourier transform as a linear transform. The fact that the Fourier transform is a linear transform is illustrated in Figure 1, Figure 2, and Figure 3.

In these three figures, the input series is shown in the real area in the upper left. For simplification, the values of the imaginary part of the input series shown in the upper right are all zero.

Also, for simplification, the zero origin is shown in the center by the value with the empty circle.

The real and imaginary parts of the transform output are shown in the bottom of each figure.

Figure 1 shows an input series consisting of a pulse that starts with a high value at the origin and extends down and to the right for five samples, ending in a large negative value.

This input series produces a rather complicated transform output series, as can be seen in the bottom two boxes in Figure 1. I will come back to a discussion of the transform output later.

A mirror-image pulse

Figure 2 shown an input series consisting of a pulse that begins with a large negative value four samples to the left of the origin and extends up and to the right ending with a large positive value at the origin. The input series in Figure 2 is the mirror image of the input series in Figure 1 relative to the origin.

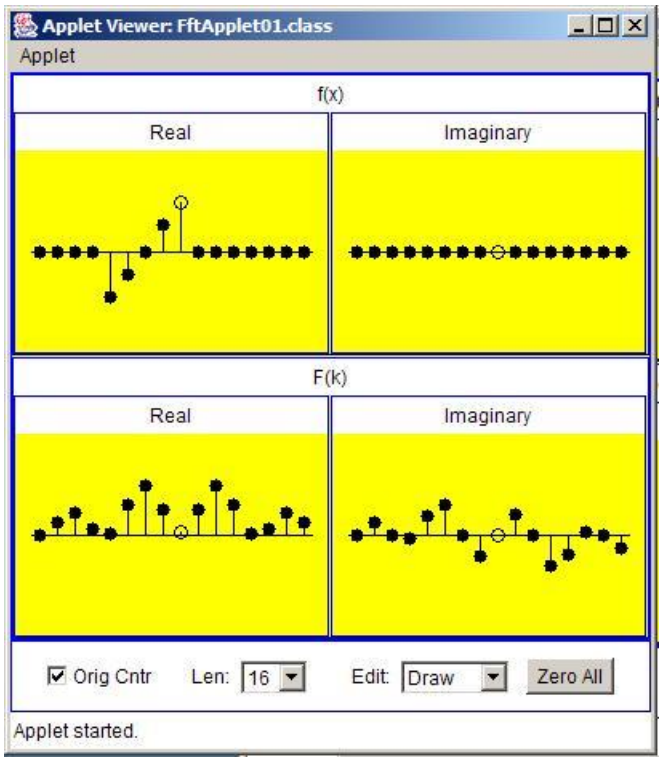

Figure 2 Transform of pulse with positive slope.

The transform output

Once again, the output from the transform of the input series is shown in the bottom two boxes of Figure 2.

A comparison of the real part of each of the transforms for Figure 1 and Figure 2 shows that the real parts are the same, at least insofar as I was able to control the input by interactively adjusting the locations of the circles using the mouse.

A comparison of the imaginary part of each of the transforms shows that the imaginary parts are the same except for the algebraic sign of each of the values in the imaginary part. The algebraic sign of each of the values in Figure 2 is the reverse of the algebraic sign of each of the values in Figure 1.

Now sum the two input series

To demonstrate that the Fourier transform is a linear transform, I will create a new input series that is the sum of the input series from Figure 1 and Figure 2. I will show that the transform of the sum is the sum of the transforms.

This is illustrated in Figure 3.

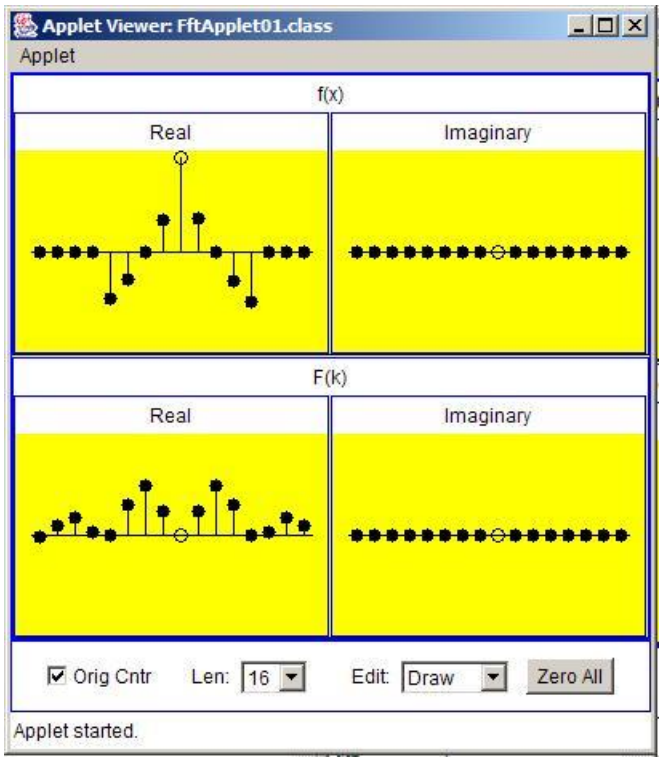

Figure 3 Transform of the sum of two pulses.

The transform of the sum equals the sum of the transforms

Figure 3 shows an input series that is the sum of the individual input series from Figure 1 and Figure 2. This produces a pulse that is symmetric around the origin indicated by the value with the empty circle.

Normalized output

Note that the display of the transform values produced by this applet is normalized so as to keep them in a reasonable range for plotting. As a result, absolute values don't have much meaning. Only relative values have meaning.

The real part of the transform of the input series in Figure 3 has the same shape as the real parts of the transforms of the input series in Figure 1 and Figure 2. This is what would be produced by adding the real parts of the transforms of the pulses in Figure 1 and Figure 2, and then normalizing the result.

The imaginary part sums to zero

The imaginary part of the transform of the input series in Figure 3 is zero at all sample values. This is what would be produced by adding the imaginary parts of the transforms of the input series in Figure 1 and Figure 2.

(Recall that the values in the imaginary parts of the two earlier transforms had the same magnitude but opposite signs).

Thus, Figure 1, Figure 2, and Figure 3 demonstrate that the transform of the sum of two or more input series is equal to the sum of the transforms of the individual input series. The Fourier transform is a linear transform.

Single sample real pulse with a delay

The real part of the transform of a single real sample with a shift relative to the origin has the shape of a cosine curve with a period that is proportional to the reciprocal of the shift. Negative sample values produce cosine curves with negative amplitudes.

The imaginary part of the transform of a single real sample with a shift relative to the origin has the shape of a sine curve with a period that is proportional to the reciprocal of the shift. Negative sample values produce sine curves with negative amplitudes.

The magnitude of the transform is the square root of the sum of the squares of the real and imaginary parts at each output sample point. For the case of a single input sample with a shift, that magnitude is constant for all output sample points and is proportional to the absolute value of the sample.

The above facts are illustrated in Figure 4, Figure 5, Figure 6, and Figure 7.

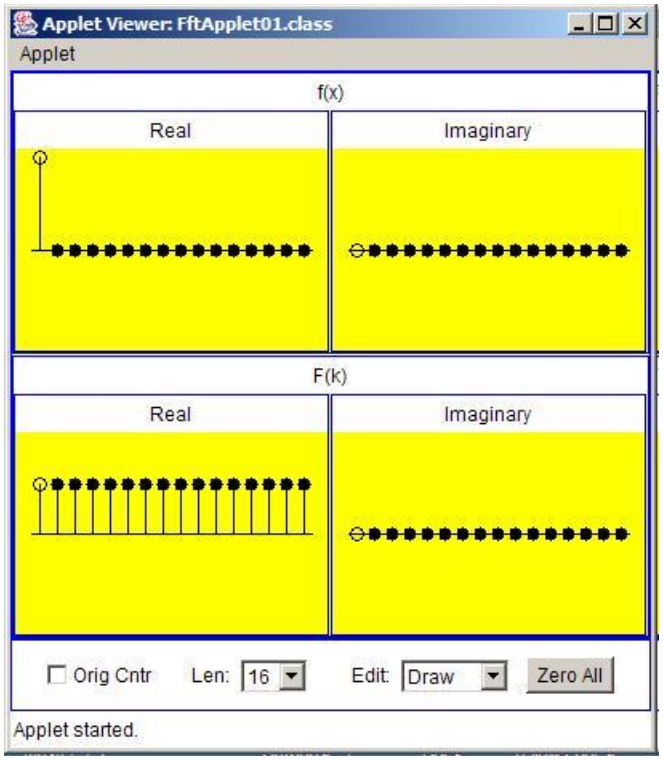

Figure 4 Transform of a real single sample with no shift.

A shift of zero

Figure 4 shows the transform of a single real pulse with a shift of zero relative to the origin.

> *(Note that in this series of figures, the origin was moved from the center to the left end. Once again, the sample with the empty circle represents the origin.)*

Although it isn't obvious, the real part of the transform in Figure 4 has the shape of a cosine curve with a period that is the reciprocal of the shift. Because the shift is zero, the period of the cosine curve is infinite, producing real values that are constant at all output sample values.

Similarly, the imaginary part of the transform in Figure 4 has a shape that is a sine curve with an infinite period. Thus, it is zero at all output sample values.

A shift of one sample interval

Figure 5 shows the transform of a single real sample with a negative value and a shift of one sample interval relative to the origin.

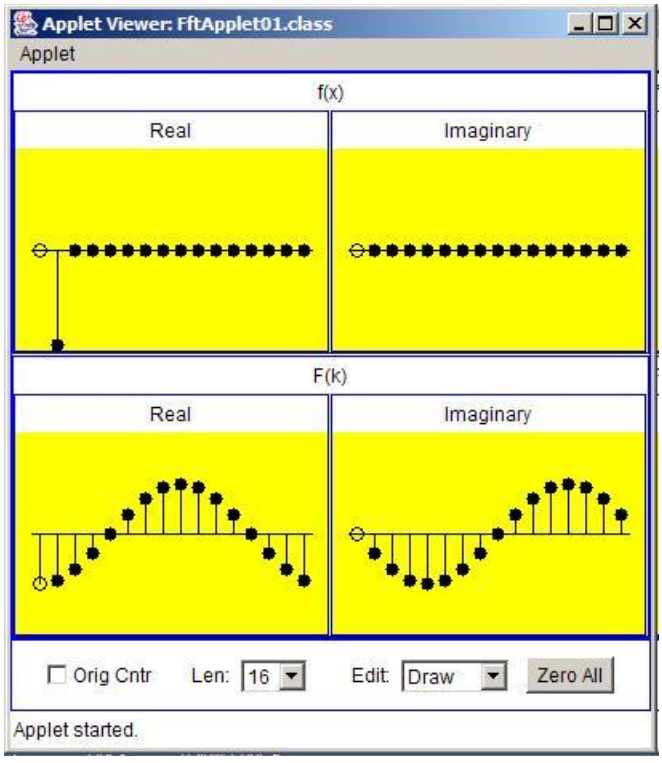

Figure 5 Transform of a real single sample with a shift equal to one sample interval and a negative value.

A cosine curve and a sine curve

The shape of the real part of the transform output is an upside down cosine curve. It is upside down because it has a negative amplitude. This is caused by the fact that the input sample has a negative value.

The shape of the imaginary part of the transform is an upside down sine curve.

Number of output samples equals number of input samples

This transform program computes real and imaginary values from zero to an output index that is one output sample interval less than the sampling frequency. The number of output values is equal to the number of samples in the input series. This is very typical of FFT algorithms.

In this case, I set the applet up to accept sixteen input samples and to produce sixteen output samples.

Representing time and frequency

For the moment, lets think in terms of time and frequency. Assume that the input series f(x) is a time series and the output series F(k) is a frequency spectrum.

To make the arithmetic easy, let's assume that the sampling interval for the input time series in the upper left box of Figure 5 is one second. This gives a sampling frequency of one sample per second, and a total elapsed time of sixteen seconds.

The sine and cosine curves in Figure 5 each go through one complete period between a frequency of zero and the sampling frequency, which one sample per second. Thus, the period of the sine and cosine curves along the frequency axis is one sample per second. This is the reciprocal of the time shift of one sample interval at a sampling frequency of one sample per second.

Stated differently, the number of periods of the sine and cosine curves in the real and imaginary parts of the transform between a frequency of zero and a frequency equal to the sampling frequency is equal to the shift in sample intervals. A shift of one sample interval produces sine and cosine curves having one period in the frequency range from zero to the sampling frequency. A shift of two sample intervals produces sine and cosine curves having two periods in the frequency range from zero to the sampling frequency, etc. This is illustrated by Figure 6.

A shift of two sample intervals

Figure 6 shows the transform of a real single sample with a shift equal to two sample intervals and a positive value.

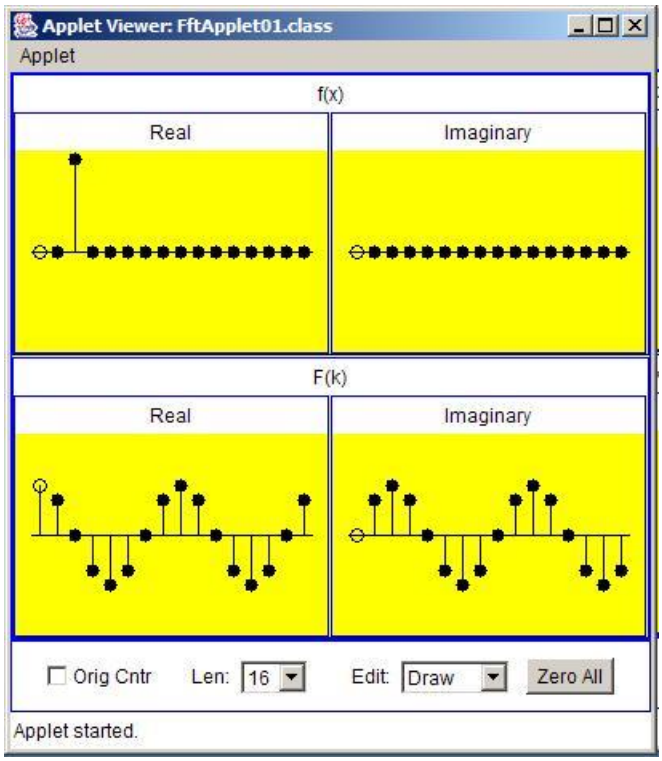

Figure 6 Transform of a real single sample with a shift equal to two sample intervals and a positive value.

The real part of the transform has the shape of a cosine curve with two complete periods between zero and an output index equal to the sampling frequency.

The imaginary part of the transform has the shape of a sine curve with two complete periods within the same output interval. This agrees with the conclusions stated in the previous section.

A shift of four sample intervals

Finally, Figure 7 shows the transform of a real single sample with a shift equal to four sample intervals.

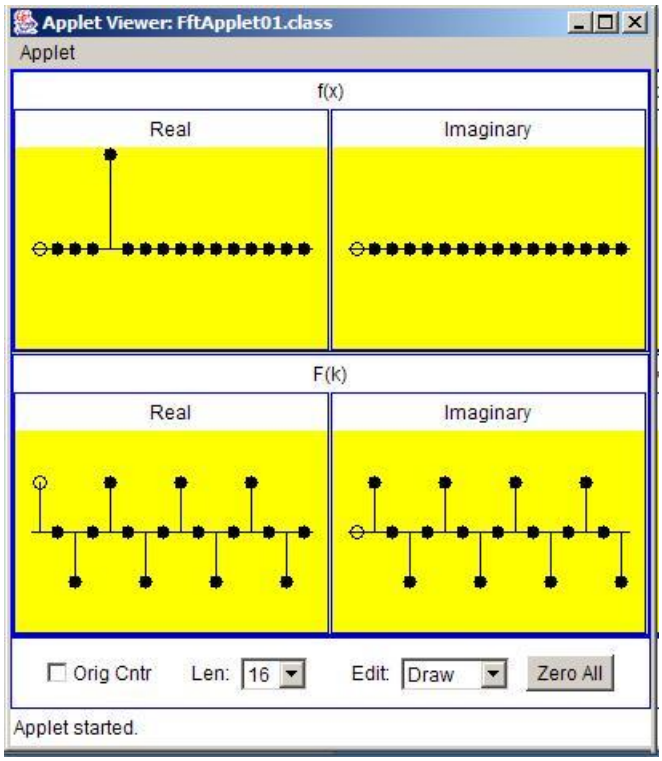

Figure 7 Transform of a real single sample with a shift equal to four sample intervals and a positive value.

The cosine and sine curves that represent the real and imaginary parts of the transform each have four complete periods between zero and an output index equal to the sampling frequency.

Equations to describe the real and imaginary parts of the transform

The main point is that if you know the value of a single real sample and you know its position in the series relative to the origin, you can write equations that describe the real and imaginary parts of the transform of that single sample without any requirement to actually perform a Fourier transform.

Those equations are simple sine and cosine equations as a function of the units of the output domain. This is an important concept that contributes greatly to the implementation of the FFT algorithm.

Transformation of a complex series

The FFT algorithm is an algorithm that transforms a series of complex values in one domain into a series of complex values in another domain. The images in the figures discussed so far indicate a transformation of a complex function given by f(x) into another complex function given by F(k). There is nothing in these images to indicate anything about time and frequency.

If the complex part of the input series f(x) is not zero, things get somewhat more complicated. For example, the real and imaginary parts of the transform of a single delayed sample having both real and imaginary parts are not necessarily cosine and sine curves. This is illustrated in Figure 8.

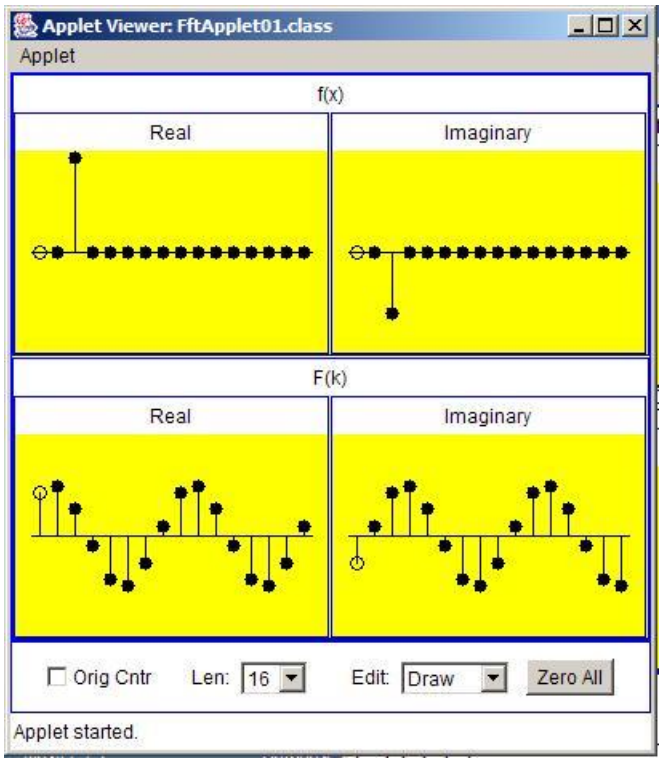

Figure 8 Transform of a complex single sample with a shift equal to two sample intervals.

Figure 8 shows the results of transforming a single sample having both real and imaginary parts and a shift of two sample intervals.

Although both the real and imaginary parts of the transformed result have the shape of a sinusoid, neither is a cosine curve and neither is a sine curve. Both of the curves are sinusoidal curves that have been shifted along the horizontal output axis moving their peaks and zero crossings away from the origin.

Linearity still applies

Because the Fourier transform is a linear transform, you can transform the real and imaginary parts of the input separately and add the two resulting transforms. The sum of the two transforms represents the transform of the entire input series including both real and imaginary parts. The program that I will discuss later takes advantage of this fact.

Even for a complex input series, if you know the values of the real and imaginary parts of a sample and you know the value of the shift associated with that sample, you can write equations that describe the real part and the imaginary part of the transform results.

Can produce the transform of a time series by the adding transforms of the individual samples

That brings us to the crux of the matter.

> ***Given an input series consisting of a set of sequential samples taken at uniform sampling intervals, we know how to write equations for the real and imaginary parts that would be produced by performing a Fourier transform on each of those samples individually.***

The input series is the sum of the individual samples

We know that we can consider the input series to consist of the sum of the individual samples, each having a specified value and a different shift. We know that the Fourier transform is a linear transform. Therefore, the Fourier transform of an input series is the sum of the transforms of the individual samples.

If we are clever enough, we can use these facts to develop a computational algorithm that can compute the Fourier transform of a time series much faster than can be obtained using a brute force DFT algorithm. Fortunately, some very clever people have already developed that algorithm. It goes by the name of the Fast Fourier Transform, or FFT algorithm.

Steps in the FFT algorithm

In truth, there are several different forms of the FFT algorithm, and the mechanics of each may be slightly different. At least one, and probably many of the algorithms operate by performing the following steps:

- Decompose an N-point complex series into N individual complex series, each consisting of a single complex sample. The order of the decomposition in an FFT algorithm is rather complicated. It is this order of decomposition, and the order of the subsequent recombination of transform results that causes the FFT algorithm to be so fast. It is also that order that makes the algorithm somewhat difficult to understand. Note that the program that I will discuss later ***does not*** implement that special order of decomposition and recombination.
- Calculate the transform of each of the N complex series, each consisting of a single complex sample. This treats each complex sample as if it is located at the origin of a complex series. This step is trivial. The real part of the transform of a single complex sample located at the origin of the series is a complex constant whose values are proportional to the real and imaginary values that make up the complex sample. Since the complex input series consists of only one complex sample, there is only one complex value in the complex transform.
- Correct each of the N transform results to reflect the original position of the complex sample in the input series. This involves the application of sine and cosine curves to the real and imaginary parts of the transform. This step is usually combined with the recombination step that follows.

- Recombine the N transform results into a single transform result that represents the transform of the original complex series. This is a very complicated operation in a real FFT algorithm. It must reverse the order of decomposition in the first step described earlier. As mentioned earlier, it is the order of the decomposition and subsequent recombination that minimizes the arithmetic operations required and gives the FFT its tremendous speed. The program that I will discuss later does not implement the special order of decomposition and recombination used in an actual FFT algorithm.

A Sample Program

I want to emphasize at the outset that this program DOES NOT implement an FFT algorithm. Rather, this program illustrates the underlying signal processing concepts that make the FFT possible in a form that is more easily understood than is normally the case with an actual FFT algorithm.

Separate processes in an FFT algorithm

In summary, a typical FFT algorithm performs the following processes:

- Decompose an N-point complex series into N individual complex series, each consisting of a single complex sample.
- Recognize that the complex transform of a single complex sample is equal to the value of the complex sample.
- Correct the transform for each complex sample to reflect the original position of the complex sample in the input series.
- Recombine the N transform results into a single transform result that represents the transform of the original complex series.

This program performs each of the processes listed above. However, it does not perform those processes in the special order used by an FFT algorithm that causes the FFT algorithm to be able to perform those processes at very high speed.

How the processes are implemented

The decomposition process in this program takes the complex samples in the order that they appear in the input complex series.

The transform of each complex sample is simply the sample itself. This is the result that would be obtained by actually computing the transform of the complex sample if the sample were the first sample in the series.

The transform result for each complex sample *(the sample itself)* is then corrected for position by applying sine and cosine curves to reflect the actual position of the complex sample within the original complex series.

In order to accomplish the recombination of the corrected transform results, the real and

imaginary parts of the corrected transform are added to accumulators. These accumulators are used to accumulate the corrected real and imaginary parts from the corrected transforms for all of the individual complex samples.

Once the real and imaginary parts have been accumulated for all of the complex samples, the real part of the accumulator represents the real part of the transform of the original complex series. The imaginary part of the accumulator represents the imaginary part of the transform of the original complex series. However, an actual transform was never performed on the original complex series.

Three cases are examined

This program creates three separate complex series, applies the processes listed above to each of those series, and displays the results on the screen.

No attempt is made to manage the decomposition and the subsequent recombination in the manner of a true FFT algorithm. Therefore, this program is designed to illustrate the processes involved, and is not designed to provide the speed of a true FFT algorithm.

This program was tested using SDK 1.4.2 under WinXP.

Will discuss in fragments

As is my usual approach, I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 9 near the end of the lesson.

The program begins in Listing 1, which shows the beginning of the controlling class named **Fft02** and the beginning of the **main** method.

```
class Fft02{

  public static void main(String[] args){
    Transform transform = new Transform();
```

Listing 1

Instantiate a Transform object

The first statement in the **main** method instantiates an object of the **Transform** class. This object implements the processes used in an FFT, but does not implement those processes in the special order required by an FFT algorithm.

The purpose of an object of the **Transform** class is to illustrate the processes commonly used in an FFT in a manner that is more easily understood than is often the case with an actual FFT algorithm.

I will put the **main** method on the back burner for the moment and explain the class named **Transform**.

The class named Transform

Listing 2 presents the beginning of the class named **Transform**. Listing 2 also presents the beginning of an instance method of that class named **doIt**. The **doIt** method computes and returns the complex transform *(via output parameters)* of an incoming complex series.

```
class Transform{

  void doIt(double[] realIn,
            double[] imagIn,
            double scale,
            double[] realOut,
            double[] imagOut){
```

Listing 2

The method parameters

The **doIt** method receives five incoming parameters. The first two parameters are references to two array objects of type **double** containing the real and imaginary parts of the input series.

The third parameter is a scale factor that is applied to the transform output in an attempt to keep the values in a range suitable for plotting if desired.

The last two parameters are references to array objects of type **double**. The results of performing the transform are used to populate these two arrays. This is the mechanism by which the object returns the transform results to the calling program. It is assumed that all of the elements in these two array objects contain values of zero upon entry to the **doIt** method.

Performing the transform

The body of the **doIt** method is presented in Listing 3. The code in Listing 3 iterates on the input arrays, passing each complex sample contained in those two arrays to a method named **correctAndRecombine**.

```
    for(int cnt = 0;cnt < realIn.length;cnt++){
      correctAndRecombine(realIn[cnt],
                          imagIn[cnt],
                          cnt,
                          realIn.length,
                          scale,
                          realOut,
                          imagOut);
    }//end for loop
```

```
  }//end doIt

Listing 3
```

The transforms of the complex input samples

Each complex value in the incoming arrays represents both a complex sample and the transform of that complex sample under the assumption that the complex sample appears at the origin of the input series.

Correct for actual position and recombine

The method named **correctAndRecombine** corrects the transform result for each of the complex samples in the series so as to reflect the actual position of the complex sample in the original input series.

Then the method named **correctAndRecombine** adds the corrected transform result into a pair of accumulators, one for the real part and one for the imaginary part. This accomplishes the recombination of the corrected transforms of the input samples in order to produce the transform of the entire original complex input series.

The correctAndRecombine method

The **correctAndRecombine** method is shown in Listing 4. Listing 4 also signals the end of the **Transform** class.

```
  void correctAndRecombine(double realSample,
                           double imagSample,
                           int position,
                           int length,
                           double scale,
                           double[] realOut,
                           double[] imagOut){

    //Calculate the complex transform values for
    // each sample in the complex output series.
    for(int cnt = 0; cnt < length; cnt++){
      double angle =
               (2.0*Math.PI*cnt/length)*position;

      //Calculate output based on real input
      realOut[cnt] +=
                realSample*Math.cos(angle)/scale;
      imagOut[cnt] +=
                realSample*Math.sin(angle)/scale;

      //Calculate output based on imag input
      realOut[cnt] -=
                imagSample*Math.sin(angle)/scale;
```

```
      imagOut[cnt] +=
                  imagSample*Math.cos(angle)/scale;
    }//end for loop
  }//end correctAndRecombine

}//end class transform
```

Listing 4

This method accepts an incoming complex sample value and the position in the series associated with that sample. The method corrects the real and imaginary transform values for that complex sample to reflect the specified position in the input series.

After correcting the transform values for the sample on the basis of position, the method updates the corresponding real and imaginary values contained in array objects that are used to accumulate the real and imaginary values for all of the samples.

References to the array objects are received as input parameters. Outgoing results are scaled by an incoming parameter in an attempt to cause the output values to fall within a reasonable range in case someone wants to plot them.

The incoming parameter named **length** specifies the number of output samples that are to be produced.

Hopefully this explanation will make it possible for you to understand the code in Listing 4.

Note in particular the use of the **Math.cos** and **Math.sin** methods to apply the cosine and sine curves in the correction of the transforms of the individual complex samples. This is used to produce results similar to those shown in Figure 5 through Figure 7.

Note the use of the **position** and **length** parameters in the computation of the angle that is passed as an argument to the **Math.cos** and **Math.sin** methods.

Also note how the correction is made separately on the real and imaginary parts of the input. This produces results similar to those shown in Figure 7 after those results are added in the accumulators.

Back to the main method

Returning now to the **main** method, the code in Listing 5 prepares the input data and the output arrays for the first case that we will look at. This case is labeled as Case A.

```
    System.out.println("Case A");
    double[] realInA =
               {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    double[] imagInA =
               {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

```
    double[] realOutA = new double[16];
    double[] imagOutA = new double[16];

    //Perform the transform and display the
    // transformed results for the original
    // complex series.
    transform.doIt(realInA,imagInA,2.0,realOutA,
                                       imagOutA);
    display(realOutA,imagOutA);

Listing 5
```

Note that for Case A, the input complex series contains non-zero values only in the real part. Also, most of the values in the real part are zero.

The graphic form

Case A is shown in graphic form in Figure 9. As you can see, the input series consists of two non-zero values in the real part. All the values in the imaginary part are zero.

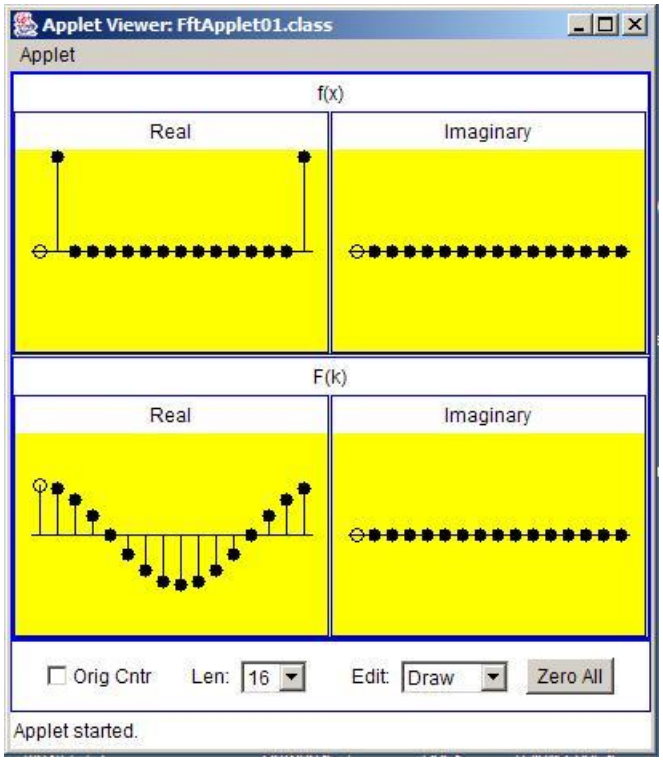

Figure 9 Case A. Transform of a real sample with two non-zero values.

The real part of the transform of the complex input series looks like one cycle of a cosine curve. All of the values in the imaginary part of the transform result are zero.

The numeric output

As you saw in Listing 5, the code in the **main** method invokes a method named **display** to display the complex transform output in numeric form on the screen. The output produced by Listing 5 is shown in Figure 10. *(Note that I manually inserted line breaks to force the material to fit in this narrow publication format.)*

```
Case A
Real:
1.0 0.923 0.707 0.382 0.0 -0.382 -0.707 -0.923
-1.0 -0.923 -0.707 -0.382 0.0 0.382 0.707 0.923
imag:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Figure 10

If you plot the real and imaginary values in Figure 10, you will see that they match the transform output shown in graphic form in Figure 9.

Case B code

The code from the **main** method for Case B is shown in Listing 6. Note that the input complex series contains non-zero values in both the real and imaginary parts.

```
    System.out.println("\nCase B");
    double[] realInB =
               {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    double[] imagInB =
             {0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,-1};

    double[] realOutB = new double[16];
    double[] imagOutB = new double[16];

    transform.doIt(realInB,imagInB,2.0,realOutB,
                                       imagOutB);
    display(realOutB,imagOutB);
```

Listing 6

Case B in graphical form

Case B is shown in graphical form in Figure 11.

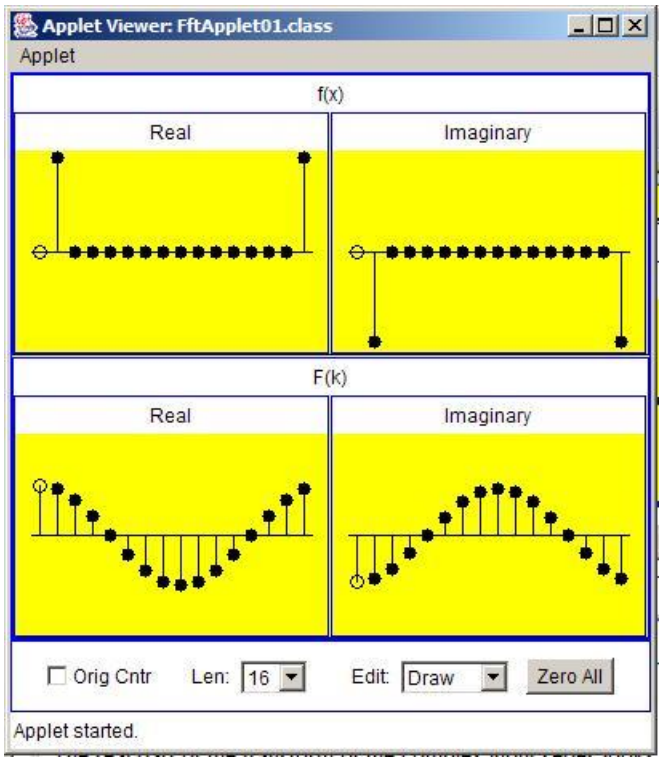

Figure 11 Case B. Transform of a simple complex series.

Case B in numeric form

The output from the code in Listing 6 is shown in Figure 12.

```
Case B
Real:
1.0 0.923 0.707 0.382 0.0 -0.382 -0.707 -0.923
-0.999 -0.923 -0.707 -0.382 0.0 0.382 0.707 0.923
imag:
-1.0 -0.923 -0.707 -0.382 0.0 0.382 0.707 0.923
1.0 0.923 0.707 0.382 0.0 -0.382 -0.707 -0.923

Figure 12
```

If you plot the values for the real and imaginary parts from Figure 12, you will see that they match the real and imaginary output shown in Figure 11.

Case C code

The code extracted from the **main** method for Case C is shown in Listing 7.

```
    System.out.println("\nCase C");
```

```
    double[] realInC =
      {1.0,0.923,0.707,0.382,0.0,-0.382,-0.707,
          -0.923,-1.0,-0.923,-0.707,-0.382,0.0,
           0.382,0.707,0.923};
    double[] imagInC =
      {0.0,-0.382,-0.707,-0.923,-1.0,-0.923,
           -0.707,-0.382,0.0,0.382,0.707,0.923,
            1.0,0.923,0.707,0.382};

    double[] realOutC = new double[16];
    double[] imagOutC = new double[16];

    transform.doIt(realInC,imagInC,16.0,realOutC,
                                        imagOutC);
    display(realOutC,imagOutC);

Listing 7
```

The complex input series for Case C is a little more complicated than that for either of the previous two cases. Note in particular that the input complex series contains non-zero values in both the real and imaginary parts. In addition, very few of the values in the complex series have a value of zero.

(The values of the complex samples actually describe a cosine curve and a negative sine curve as shown in Figure 13.)

The graphic form of Case C

Case C is shown in graphic form in Figure 13.

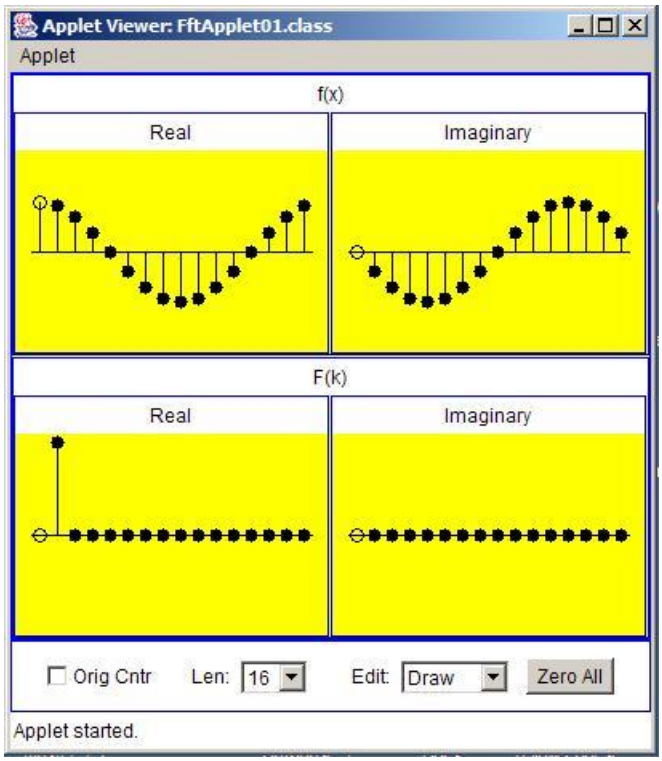

Figure 13 Case C. Transform of a more complicated complex series.

The Fourier transform is reversible

One of the interesting things to note about Figure 13 is the similarity of Figure 13 and Figure 5. These two figures illustrate the reversible nature of the Fourier transform.

If I had used a positive input real value instead of a negative input real value in Figure 5, the input of Figure 5 would look exactly like the output in Figure 13, and the output of Figure 5 would look exactly like the input of Figure 13.

With that as a hint, you should now be able to figure out how I used a mouse and drew the perfect sine and cosine curves in Figure 13. In fact, I didn't draw them at all. Rather, I used my mouse and drew the output, and the applet gave me the corresponding input automatically.

Case C in numeric form

The output produced by the code in Listing 7 is shown in Figure 14.

```
Case C
Real:
0.0 0.999 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
imag:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

```
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Figure 14

If you plot the real and imaginary input values from Listing 7, you will see that they match the input values in Figure 13. If you plot the real and imaginary output values in Figure 14, you will see that they match the output values shown in Figure 13.

Listing 7 signals the end of the **main** method.

The display method

Listing 8 shows the code for a simple method named **display**. The purpose of the **display** method is to display a real series and an imaginary series, each contained in an incoming array object of type **double**. The double values are truncated to no more than four digits before displaying them. Then they are displayed on a single line.

```
  static void display(double[] real,
                      double[] imag){
    System.out.println("Real: ");
    for(int cnt=0;cnt < real.length;cnt++){
      System.out.print(((int)(1000.0*real[cnt]))
                                  /1000.0 + " ");
    }//end for loop
    System.out.println();

    System.out.println("imag: ");
    for(int cnt=0;cnt < imag.length;cnt++){
      System.out.print(((int)(1000.0*imag[cnt]))
                                  /1000.0 + " ");
    }//end for loop
    System.out.println();
  }//end display

}//end class Fft02
```

Listing 8

Listing 8 also signals the end of the controlling class named **Fft02**.

Run the Program

I encourage you to copy and compile the program that you will find in Listing 9. Experiment with different complex input series.

I also encourage you to download the applet from http://sepwww.stanford.edu/oldsep/hale/FftLab.html and experiment with it as well. Compare the numeric output produced by this program with the graphic output produced by the applet.

I also encourage you to read what Steven Smith has to say about the FFT algorithm at http://www.dspguide.com/ch12.htm. Pay particular attention to his explanation of the order in which the input series is decomposed and the order in which the individual transform outputs are recombined.

Finally, I encourage you to examine the source code for the applet. You will find that source code at http://sepwww.stanford.edu/oldsep/hale/FftLab.java. Concentrate on that portion of the source code that performs the FFT. Hopefully, what you have learned in this lesson in addition to what you learn from Steven Smith's book will make it easier for you to understand the source code for the FFT.

Summary

In this lesson, I have explained some of the underlying signal processing concepts that make the FFT possible. I illustrated those concepts in a program designed specifically to be as simple as possible while still illustrating the concepts.

Now that you understand those concepts, you should be able to better understand explanations of the mechanics of the FFT algorithm that appear on various websites.

Complete Program Listing

A complete listing of the program is provided in Listing 9 below.

```
/*File Fft02.java Copyright 2004, R.G.Baldwin
Rev 4/30/04

This program DOES NOT implement an FFT algorithm.
Rather, this program illustrates the underlying
FFT concepts in a form that is much more easily
understood than is normally the case with an
actual FFT algorithm.  The steps in the
implementation of a typical FFT algorithm are as
follows:

1. Decompose an N-point complex series into N
individual complex values, each consisting of a
single complex sample.  The order of the
decomposition in an FFT algorithm is rather
complicated.  It is this order of decomposition,
and the order of the subsequent recombination of
transform results that causes the FFT to be so
fast.  It is also that order that makes the
algorithm somewhat difficult to understand. This
program does not implement that order of
decomposition and recombination.

2. Calculate the transform of each of the N
```

complex samples, treating each as if it were
located at the beginning of the complex series.
This step is trivial. The real part of the
transform of a single complex sample located at
the beginning of the series is a complex constant
whose values are proportional to the real and
imaginary values that make up the complex sample.

3. Correct each of the N transform results to
reflect the actual position of the complex sample
in the series. This involves the application of
sine and cosine curves to the real and imaginary
parts of the transform. This step is usually
combined with the recombination step that
follows.

4. Recombine the N transform results into a
single transform result that represents the
transform of the original complex series. This
is a very complicated operation in a real FFT
algorithm. It must reverse the order of
decomposition in the first step described
earlier. As mentioned earlier, it is the order
of the decomposition and subsequent recombination
that minimizes the arithmetic operations required
and gives the FFT its tremendous speed. This
program does not implement the special order of
decomposition and recombination used in an actual
FFT algorithm.

This program creates three separate complex
series, applies the processes listed above to
each of those series, and displays the results on
the screen. No attempt is made to manage the
decomposition and the subsequent recombination in
the manner of a true FFT algorithm. Therefore,
this program is designed to illustrate the
processes involved, and is not designed to
provide the speed of a true FFT algorithm.

The decomposition process in this program takes
the complex samples in the order that they appear
in the input complex series.

The transform of each complex sample is simply
the sample itself. This is the result that would
be obtained by actually computing the transform
of the complex sample if the sample were the
first sample in the series.

The transform result for each complex sample is
then corrected by applying sine and cosine curves
to reflect the actual position of the complex
sample within the complex series.

The real and imaginary parts of the corrected

```
transform results are then added to accumulators
that are used to accumulate the corrected real
and imaginary parts from the corrected
transforms for all of the individual complex
samples.

Once the real and imaginary parts have been
accumulated for all of the complex samples, the
real part of the accumulator represents the real
part of the transform of the original complex
series.  The imaginary part of the accumulator
represents the imaginary part of the transform of
the original complex series.

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


class Fft02{

  public static void main(String[] args){

    //Instantiate an object that will implement
    // the processes used in an FFT, but not in
    // the order required by an FFT algorithm.
    Transform transform = new Transform();

    //Prepare the input data and the output
    // arrays for Case A.  Note that for this
    // case, the input complex series contains
    // non-zero values only in the real part.
    // Also, most of the values in the real part
    // are zero.
    System.out.println("Case A");
    double[] realInA =
               {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    double[] imagInA =
               {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    double[] realOutA = new double[16];
    double[] imagOutA = new double[16];

    //Perform the transform and display the
    // transformed results for the original
    // complex series.
    transform.doIt(realInA,imagInA,2.0,realOutA,
                                       imagOutA);
    display(realOutA,imagOutA);


    //Process and display the results for Case B.
    // Note that the input complex series
    // contains non-zero values in both the real
    // and imaginary parts.  However, most of the
    // values in the real and imaginary parts are
    // zero.
```

```
    System.out.println("\nCase B");
    double[] realInB =
               {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    double[] imagInB =
             {0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,-1};

    double[] realOutB = new double[16];
    double[] imagOutB = new double[16];

    transform.doIt(realInB,imagInB,2.0,realOutB,
                                       imagOutB);
    display(realOutB,imagOutB);


    //Process and display the results for Case C.
    // Note that the input complex series
    // contains non-zero values in both the real
    // and imaginary parts.  In addition, very
    // few of the values in the complex series
    // have a value of zero.  (The values of the
    // complex samples actually describe a cosine
    // curve and a sine curve.)
    System.out.println("\nCase C");
    double[] realInC =
      {1.0,0.923,0.707,0.382,0.0,-0.382,-0.707,
          -0.923,-1.0,-0.923,-0.707,-0.382,0.0,
           0.382,0.707,0.923};
    double[] imagInC =
      {0.0,-0.382,-0.707,-0.923,-1.0,-0.923,
           -0.707,-0.382,0.0,0.382,0.707,0.923,
            1.0,0.923,0.707,0.382};

    double[] realOutC = new double[16];
    double[] imagOutC = new double[16];

    transform.doIt(realInC,imagInC,16.0,realOutC,
                                       imagOutC);
    display(realOutC,imagOutC);

  }//end main
  //=======================================//

  //The purpose of this method is to display
  // a real series and an imaginary series,
  // each contained in an incoming array object
  // of type double.  The double values are
  // truncated to no more than four digits
  // before displaying them.  Then they are
  // displayed on a single line.
  static void display(double[] real,
                      double[] imag){
    System.out.println("Real: ");
    for(int cnt=0;cnt < real.length;cnt++){
      System.out.print(((int)(1000.0*real[cnt]))
                                 /1000.0 + " ");
    }//end for loop
```

```
    System.out.println();

    System.out.println("imag: ");
    for(int cnt=0;cnt < imag.length;cnt++){
      System.out.print(((int)(1000.0*imag[cnt]))
                                  /1000.0 + " ");
    }//end for loop
    System.out.println();
  }//end display

}//end class Fft02
//=============================================//

//This class applies the processes normally used
// in an FFT algorithm.  However, this class does
// not apply those processes in the special order
// required of an FFT algorithm.  It is that
// special order that minimizes the arithmetic
// requirements of an FFT algorithm and causes it
// to be very fast.  The purpose of an object of
// this class is to illustrate the processes in a
// more easily understood fashion that is often
// the case with an actual FFT algorithm.
class Transform{

  void doIt(double[] realIn,double[] imagIn,
            double scale,double[] realOut,
            double[] imagOut){
    //Each complex value in the incoming arrays
    // represents both a complex sample and the
    // transform of that complex sample under the
    // assumption that the complex sample appears
    // at the beginning of the series.
    //Correct the transform result for each of
    // the complex samples in the series to
    // reflect the actual position of the complex
    // sample in the series.  Add the corrected
    // transform result into accumulators in
    // order to produce the transform of the
    // original complex series.
    for(int cnt = 0;cnt < realIn.length;cnt++){
      correctAndRecombine(realIn[cnt],
                          imagIn[cnt],
                          cnt,
                          realIn.length,
                          scale,
                          realOut,
                          imagOut);
    }//end for loop
  }//end doIt

  //=============================================//

  //This method accepts an incoming complex
  // sample value and the position in the series
  // associated with that sample. The method
```

```
  // calculates the real and imaginary transform
  // values associated with that complex sample
  // when it is located at the specified
  // position.  Then it updates the corresponding
  // real and imaginary values contained in array
  // objects used to accumulate the real and
  // imaginary values for all of the samples.
  // References to the array objects are received
  // as input parameters. Outgoing results are
  // scaled by an incoming parameter in an
  // attempt to cause the output values to fall
  // within a reasonable range in case someone
  // wants to plot them.
  void correctAndRecombine(
            double realSample,double imagSample,
            int position,int length,double scale,
            double[] realOut,double[] imagOut){
    //Calculate the complex transform values for
    // each sample in the complex output series.
    for(int cnt = 0; cnt < length; cnt++){
      double angle =
                (2.0*Math.PI*cnt/length)*position;
      //Calculate output based on real input
      realOut[cnt] +=
                realSample*Math.cos(angle)/scale;
      imagOut[cnt] +=
                realSample*Math.sin(angle)/scale;

      //Calculate output based on imag input
      realOut[cnt] -=
                imagSample*Math.sin(angle)/scale;
      imagOut[cnt] +=
                imagSample*Math.cos(angle)/scale;
    }//end for loop
  }//end correctAndRecombine

}//end class transform

//=============================================//
```

Listing 9

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-