AC 2011-451: A TASTE OF JAVA - DISCRETE AND FAST FOURIER TRANSFORMS

Mohammad Rafiq Muqri, DeVry University, Pomona

Dr. Mohammad R. Muqri is a professor in the Department of Computer and Biomedical engineering technology at DeVry University. He received his M.S.E.E. degree from University of Tennessee, Knoxville. His research interests include simulations, algorithmic computing and digital signal processing.

Dr. Javad Shakib, DeVry University, Pomona

A Taste of Java - Discrete and Fast Fourier Transforms

This paper attempts to present the development and application of a practical teaching module introducing Java programming techniques to electronics, computer and bioengineering students before they encounter digital signal processing and its applications in junior and/or senior level courses.

The Fourier transform takes a signal in the time domain, switches it into the frequency domain, and vice versa. Fourier Transforms are extensively used in engineering and science in a wide variety of fields including acoustics, digital signal processing, image processing, geophysical processing, wavelet theory, optics and astronomy. The Discrete Fourier Transform (DFT) is an essential digital signal processing tool, and is highly desirable if the integral form of the Fourier Transform cannot be expressed as a mathematical equation. The key to spectral analysis is to choose a window length that suits the signal to be analyzed, since the length of window used for DFT calculations has a strong impact on the information the DFT can provide. The operation count of the DFT algorithm is time intensive, and as such a number of Fast Fourier Transform methods have been developed to perform DFT efficiently.

This paper will explain how this learning and teaching module was instrumental in progressive learning for students by presenting Java programming and the general theory of the Fourier Transform in order to demonstrate how the DFT and FFT algorithms are derived and computed through leverage of the Java data structures. This paper thereby serves as an innovative way to expose technology students to this difficult topic and gives them a fresh taste of Java programming while having fun learning the Discrete and Fast Fourier Transforms.

DSP algorithms may be implemented on any processor, but specialized digital signal processing hardware enables the greatest speed and efficiency.[8] Digital signal processors designed specifically for operations common in DSP have special features that permit them to accomplish in real time what other processors cannot. Real time means that outputs keep pace with collection of input samples during actual operations. For some operations such as filtering, this means that a new output sample can be produced as each new input sample is received. For others such as FFTs, output information can be produced only when a block of input samples has been recorded. A DFT decomposes a sequence of values into components of different frequencies. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical.
.
An FFT is a way to compute the same result more quickly; computing a DFT of N points in the naive way, using the definition, takes O (N^2) arithmetical operations, while an FFT can compute the same result in only O(N log N) operations. The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions—in practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to N / log(N). This huge improvement made many DFT-based algorithms practical; FFTs are of great importance to a wide variety of applications, from audio and speech signal processing, sonar and radar signal processing, sensor array processing, spectral estimation, statistical signal processing, signal processing for communications, control

of systems, biomedical signal processing, seismic data processing, and solving partial differential equations.

We will begin with a brief review of some key concepts and terminology of object orientation which were found to be useful by engineering technology students at our university. Object orientation uses classes to encapsulate (i.e., wrap together) data (attributes) and methods (behaviors). In procedural programming languages (such as C), the programming tends to be action oriented. Java programming, however, is object oriented. In procedural programming languages, the unit of programming is the function (functions are called methods in Java). In Java the unit of programming is the class. Objects are created from classes by instantiation, and attributes and behaviors are encapsulated within the "boundaries" of classes as methods and fields. Keywords public and private are access modifiers. Variables or methods declared public are accessible wherever the program has a reference to an object of the class. Variables or methods declared with access modifier private are accessible only to methods of the class in which they are declared.

Arrays in Java are data structures consisting of related data items of the same type. Arrays can be considered as fixed-length entities, although at times an array reference with proper syntax may be reassigned to a new array of a different length. On the other hand we have dynamic data structures, such as stacks, queues, trees and lists that can shrink or grow as programs execute. The students were introduced to simple examples using static arrays. Array objects occupy space in memory as such all objects in Java (including arrays) are created with keyword **new**.
The following declaration and array-creation expression creates 10 elements for the double array fdata.

```
double  fdata[ ]  =  new double[ 10 ] ;
```

The task can also be performed in two steps as follows:

```
double fdata[  ];
fdata  = new  double [ 10 ] ;
```

Sometimes, we may write programs which may employ a series of counter variables to summarize the discrete Fourier transform data. We also have introduced the students to the mechanisms used to pass arguments to methods. The two ways to pass arguments to methods in many programming languages (like C and C++) are pass-by-value and pass-by-reference, which are also sometimes called call-by-value and call-by-reference, respectively. Since arrays are objects in Java, arrays are passed to methods in Java by reference. Passing arrays by reference makes sense for performance reasons as well. If arrays were passed by value, a copy of each element would be passed. Imagine now for large, frequently passed arrays; this not only means time wasted in processing but a considerable memory overhead due to space utilization by the copies of arrays.

To pass an array argument to a method, you just specify the name of the array without any brackets. For example, if array fdata is declared as

```
double fdata[ ] = new double[2*N];
```

then the method call

```
fastFT ( fdata )  ;
```

passes a reference to array fdata to method fastFT. In Java, every array object knows its own length (via the length field). Thus, when you pass an array object into a method, you are not required to pass the length of the array as an additional argument.
For a method to receive an array through a method call, the method's parameters list must specify an array parameter (or several if more than one array is to be received).[6] For example, the method header for method fastFT might be written as
void fastFT (double gb[ ]) ;
indicating that fastFT expects to receive a double array in parameter gb. Since arrays are passed by reference, when the called method uses the array name gb, it refers to the actual array (fdata in the preceding call) in the calling method.

With some diligent care and guidance from faculty, any engineering technology student can quickly learn how to create and use classes and objects, a subject known as object-based programming (OBP). It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. The four Java programs or code snippets which will be given below were used by engineering technology students before even they have been exposed to control theory or DSP courses. This was accomplished by a class lecture in OBP which was followed by examples of simple substitutions in the Java, program code; however most motivated students will want to do more than that. The Faculty here at our university attempted to indulge students in such activities.

Given below is an example of an instructor lead program which the student edited, compiled and displayed the output. The purpose of this program is to show how general Java workhorse discrete Fourier Transform and other control theory methods [7] can be introduced at an earliest stage to engineering technology students with the tools and concepts they will further reinforce in future DSP courses.

```
public class Fourier {
        public static double[] discreteFT(double[]fdata, int N, boolean fwd){
                double X[] = new double[2*N];
                double omega;
                int k, ki, kr, n;
                if (fwd){
                        omega = 2.0*Math.PI/N;
                } else {
                        omega = -2.0*Math.PI/N;
                }
        for(k=0; k<N; k++) {
                kr = 2*k;
                ki = 2*k + 1;
                X[kr] = 0.0;
                X[ki] = 0.0;
                for(n=0; n<N; ++n) {
                        X[kr] += fdata[2*n]*Math.cos(omega*n*k) +
fdata[2*n+1]*Math.sin(omega*n*k);
```

```
                              X[ki] += -fdata[2*n]*Math.sin(omega*n*k) +
fdata[2*n+1]*Math.cos(omega*n*k);
                                }
               }
          if ( fwd ) {
                    for(k=0; k<N; ++k) {
                              X[2*k] /= N;
                              X[2*k + 1] /= N;
                    }
          }
          return X;
          }
          }
```

The TestDFT application class given below uses class Fourier and invokes its methods.
//TestDFT Program

```
public class TestDFT {
 public static void main(String args[]) {
           int N = 64;
           double T = 2.0;
           double tn, fk;
           double fdata[] = new double[2*N];
           for(int i=0; i<N; ++i) {
                     fdata[2*i] = Math.cos(4.0*Math.PI*i*T/N);
                     fdata[2*i+1] = 0.0;
           }
 double X[] = Fourier.discreteFT(fdata, N, true);
 for (int k=0; k<N; ++k) {
   fk = k/T;
   System.out.println("f["+k+"] = "+fk+"Xr["+k+"] = "+X[2*k]+ "  Xi["+k+"] = "+X[2*k + 1]) ;
}
 for (int i=0; i<N; ++i) {
           fdata[2*i] = 0.0;
           fdata[2*i+1] = 0.0;
           if (i == 4 || i == N-4 ) {
                     fdata[2*i] = 0.5;
           }
 }
 double x[] = Fourier.discreteFT(fdata, N, false);
 System.out.println();
 for (int n=0; n<N; ++n) {
   tn = n*T/N;
    System.out.println("t["+n+"] = "+tn+"xr["+n+"] = "+x[2*n]+"  xi["+n+"]  = "+x[2*n + 1]);
    }
  }
}
```

Here is a sample of TestDFT results:

```
f[0] = 0.0Xr[0] = -1.1275702593849246E-16  Xi[0] = 0.0
f[1] = 0.5Xr[1] = -4.5102810375396984E-17  Xi[1] = -6.505213034913027E-17
f[2] = 1.0Xr[2] = -5.898059818321144E-17  Xi[2] = -3.426078865054194E-17
f[3] = 1.5Xr[3] = 3.469446951953614E-17  Xi[3] = -1.5872719805187785E-16
```

Highly efficient algorithms for computing the DFT were first developed in the 1960s. Collectively known as Fast Fourier Transforms (FFTs), they all rely upon the fact that the standard DFT involves redundant calculation. Strictly speaking, there is no such thing as 'the FFT' [3]. Rather, there is a collection of algorithms with different features, advantages, and limitations. An algorithm which is suitable for programming in a high level-language on a general purpose computer may not be the best for special purpose DSP hardware. What the different algorithms have in common is their general approach – the decomposition of the DFT into a number of successively shorter, and simpler, DFTs.

There are various ways of explaining FFT decomposition. We can show that a DFT can be expressed in terms of shorter, simpler, DFT's by dividing the signal x[n] into subsequences. The method which is widely used in DSP literature is also referred to as conventional decomposition. Then there is also an alternative approach known as index-mapping. It should be clear in our mind that conventional decomposition and index mapping are just two ways of looking at the same problem and there is no essential difference between them.

Suppose we have a signal with N sample values, where N is an integer power of 2. We first separate x[n] into two subsequences, each with N/2 samples. The first subsequence consists of even number points in x[n], and the second consists of odd number points- Writing n = 2k, when n is even, and n = 2k + 1 when n is odd. We can thus express the original N-point DSP in terms of two N/2 point DFTs. Now we can take the decomposition further, by breaking each N/2- point subsequence down into two shorter, N/4-point subsequences. The process can continue until, in the limit, we are left with a series of 2-point subsequences, each of which requires a very simple 2-pointDFT. A complete decomposition of this type gives rise to one of the commonly used radix-2, decimation in time, FFT algorithms.

Now the students are ready to implement an FFT as a Java method. It is called the fastFFT( ) method and also defined in the Fourier class.

```
public class Fourier {
public static void fastFFT(double[ ] fdata, int N, boolean fwd) {
            double omega, tempr, tempi, fscale;
            double xtemp, cosine, sine, xr, xi;
            int i, j, k, n, m, M;

            j=0;
            for(i=0; i<N-1; i++) {
                  if (i<j) {
                        tempr = fdata[2*i];
                        tempi = fdata[2*i + 1];
                        fdata[2*i] = fdata[2*j];
                        fdata[2*i + 1] = fdata[2*j + 1];
                        fdata[2*j] = tempr;
                        fdata[2*j + 1] = tempi;          }
                k = N/2;
                while (k <= j) {
                        j  -=  k;
                        k >>= 1; }
                j  += k;
                }
        if (fwd)
                fscale = 1.0;
        else
                fscale = -1.0;
        M = 2;
        while( M < 2*N ) {
                omega = fscale*2.0*Math.PI/M;
                sin = Math.sin(omega);
                cos = Math.cos(omega) - 1.0;
                xr = 1.0;
                xi = 0.0;
                for (m=0; m<M-1; m+=2) {
                        for (i=m; i<2*N; i+=M*2) {
                j = i + m ;
                                tempr = xr*fdata[j] - xi*fdata[j+1];
                                tempi = xr*fdata[j+1] + xi*fdata[j];
                                fdata[j] = fdata[i] - tempr;
                                fdata[j+1] = fdata[i+1] - tempi;
                                fdata[i] += tempr;
                                fdata[i+1] += tempi; }
                xtemp = xr;
                xr = xr + xr*cos - xi*sin;
                xi = xi + xtemp*sin +xi*cos;
                }
        M *=2;
        }
        if( fwd ) {
                for (k=0; k<N; k++) {
                        fdata[2*k] /= N;
                        fdata[2*k + 1] /= N;
                }
        }
}
```

In the next step, the students apply this method to compute the FFT on a 2Hz cosine wave. They were instructed to take 64 data samples over a 2-second sample period. The program first computes the FFT to obtain the frequency spectrum for a 2Hz cosine wave. Then the program below was used by students to perform an inverse Fourier transform that reconstructs the 2Hz cosine wave from its frequency spectrum. Next, we implement the FFT as a Java method. It is called the fastFFT( ). The next thing to do is to test the fastFFT( ) by applying it to the composite cosine signal that were processed earlier. The amplitude time history for a signal containing three different frequency components is generated and sent to fastFFT () method. The TestFFT class source code is shown below:

```
public class TestFFT {
public static void main(String args[ ]) {
        int N = 64;
        double T = 1.0;
        double tn, fk;
        double fdata[ ] = new double[2*N];

        for(int i=0; i<N; ++i) {
                fdata[2*i] = Math.cos(8.0*Math.PI*i*T/N) +
Math.cos(14.0*Math.PI*i*T/N) +
   Math.cos(32.0*Math.PI*i*T/N);
                fdata[2*i+1] = 0.0;
                }
Fourier.fastFT(fdata, N, true);
System.out.println();
for(int k=0; k<N; ++k) {
        fk = k/T;
System.out.println(  "f["+k+"] = " + fk + " Xr["+k+"] = " + fdata[2*k] + " Xi["+k+"]
                                                                  = " + fdata[2*k+1]) ;
         }
      }
}
```

Here are the sample partial TestFFT results:

```
f[0] = 0.0 Xr[0] = 14.118483415068333 Xi[0] = 8.299076200364345
f[1] = 1.0 Xr[1] = -4.320681943456142 Xi[1] = -11.70913283509337
f[2] = 11.5931285252644 Xi[2] = 11.382484516157067
f[3] = 3.0 Xr[3] = 0.20410293867180673 Xi[3] = -9.83647671361791
…

f[60] = 60.0 Xr[60] = 0.012868248913439052 Xi[60] = 0.058867631665321246
f[61] = 61.0 Xr[61] = -0.20682636511072414 Xi[61] = 0.025288953333891355
f[62] = 62.0 Xr[62] = -0.03895429084677146 Xi[62] = -0.007474788441969488
f[63] = 63.0 Xr[63] = 0.02651390602928162 Xi[63] = 0.0
```

On most computers, only some of the total computation time of an FFT is spent performing the FFT butterfly computations- determining indices, loading and storing data, computing loop parameters and other operations consume the majority of cycles. Careful programming that allows the compiler to generate efficient code can make a several-fold improvement in the run-time of an FFT. The best choice of radix in terms of program speed may depend more on characteristics of the hardware (such as the number of CPU registers) or compiler than on the exact number of computations.

These few examples demonstrate how students can be introduced not only to object based programming with Java, but also to basic concepts of discrete Fourier, and fast Fourier Transforms in signal processing. Having being exposed to this Java programming, DFT, and FFT learning module has far reaching implications. Given time, it can be proved that this group of students who are now actually taking the junior level-DSP course, their understanding is far better than the group of students who were never exposed to this teaching module. Time did not permit me to perform a complete assessment and evaluation of result of experimental versus control group. However, I was fortunate to monitor and discuss the experiences of two students who took the senior project capstone class with me last session. They were very positive about the outcome and enhanced understanding of the DSP material which they attributed to their early exposure of Java Object Oriented Programming and the reinforcement of DFT and FFT concepts in signal processing to which they had been exposed later on during their junior and senior years.

In conclusion, it can be stated that with proper guidance, monitoring and diligent care the engineering technology students in a similar way can be exposed earlier to Java data structures and the basics of DSP. This will go a long way in motivating them, eliminating their fear, improving their understanding and enhancing their quality of education. With future conditioning and judicious course selection, they will become more motivated and this will help reinforce the best practices in implementing digital filtering by fast convolution, spectral analysis, seismic data processing, wavelet video compression, fingerprint image compression, and other advanced topics in DFT and FFT real time applications [9].

Bibliography

1. Shakib, J., Muqri, M., *Leveraging the Power of Java in the Enterprise*, American Society for Engineering Education, AC 2010-1701.
2. Dibble, P., *Real-Time Java Platform Programming,* Sun Microsystems Press, Prentice-Hall, June 2008.
3. Lynn, Paul A., Fuerst, Wolfgang, *Introductory Digital Signal Processing with Computer Applications*, John Wiley & Sons, 1994.
4. R. Meyer, H.W. Schuessler, and K. Schwarz. (1990). FFT Implmentation on DSP chips – Theory and Practice. *IEEE International Conference on Acoustics, Speech, and Signal Processing*.
5. H.V. Sorensen and C.S. Burrus. (1993, March). Efficient computation of the DFT with only a subset of input or output points. *IEEE Transactions on Signal Processing*, *41*(3), 1184-1200.
6. Deitel, H.M., Deitel, P.J., *Java How to program*, Prentice Hall, 2003.
7. Palmer G., *Technical Java - Developing Scientific and Engineering Applications*, Prentice Hall, 2003.
8. Joyce Van de Vegte, *Fundamentals of Digital Signal Processing*, Prentice Hall, 2002.
9. Java RTS Descriptive Documentation, *Java RTSReadme.html*, July 2008, http://download.oracle.com/Javase/realtime/doc_2.1/release/JavaRTSReadme.html