JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 5, July-August 2009

The Discrete Fourier Transform, Part 2: Radix 2 FFT

By Douglas Lyon

Abstract

This paper is part 2 in a series of papers about the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT). The focus of this paper is on a fast implementation of the DFT, called the FFT (Fast Fourier Transform) and the IFFT (Inverse Fast Fourier Transform). The implementation is based on a well-known algorithm, called the Radix 2 FFT, and requires that its' input data be an integral power of two in length.

Part 3 of this series of papers, demonstrates the computation of the PSD (Power Spectral Density) and applications of the DFT and IDFT. The applications include filtering, windowing, pitch shifting and the spectral analysis of re-sampling.

1 THE FFT

Given a sampled waveform

$$v_j, j \in [0 \ldots N-1] \qquad (1)$$

The Continuous Time Fourier Transform (CTFT) is defined by:

$$V(f) = F[v(t)] = \int_{-\infty}^{\infty} v(t)e^{-2\pi ift}\,dt \qquad (2).$$

The DFT is given by:

$$V_k = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \qquad (3).$$

Direct computation of the DFT takes O(N^2) complex multiplications while the FFT takes $O(N \log N)$ complex multiplications. The primary goal of the FFT is to speed computation of (3).

This paper describes an FFT algorithm known as the decimation-in-time radix-two FFT algorithm (also known as the Cooley-Tukey algorithm). The Cooley-Tukey algorithm is probably one of the most widely used of the FFT algorithms. Radix 2 means that the number of samples must be an integral power of two. The decimation in time means that the algorithm performs a subdivision of the input sequence into its odd and even members. We are able to perform this subdivision as a result of the Danielson-Lanczos Lemma:

$$V_k = \frac{1}{N}\left[V_k^e + W^k V_k^o\right] \qquad \forall_{k\in[0\ldots N-1]} \tag{4}$$

Proof of the Danielson-Lanczos Lemma:

Let

$$W = e^{-2\pi i/N} \text{ and } W^{jk} = e^{-2\pi ijk/N} \tag{5}$$

so that

$$W^{jk} = W^j W^{j(k-1)} \tag{6}$$

Substitute (5) into (3) to obtain

$$V_k = \frac{1}{N}\sum_{j=0}^{N-1} W^{jk} v_j \tag{7}.$$

We separate (7) into its odd and even components by altering how the samples are indexed:

$$V_k = \frac{1}{N}\left[\sum_{j=0}^{N/2-1} W^{2jk} v_{2j} + \sum_{j=0}^{N/2-1} W^{(2j+1)k} v_{2j+1}\right] \tag{8}$$

Where (8) shows summations operating over the odd and even indices. For example, if

$$j = 0,1,2,3\ldots, \tag{9}$$

then

$$2j = 0,2,4,6\ldots \text{ and } 2j+1 = 1,3,5\ldots. \tag{10}$$

Factoring the exponents in (8) yields

$$V_k = \frac{1}{N}\left[\sum_{j=0}^{N/2-1} W^{2jk} v_{2j} + \sum_{j=0}^{N/2-1} W^{2jk} W^k v_{2j+1}\right] \tag{11}$$

The W^k term in the right most summation is not a function of the index, so that:

$$V_k = \frac{1}{N}\left[\sum_{j=0}^{N/2-1} W^{2jk} v_{2j} + W^k \sum_{j=0}^{N/2-1} W^{2jk} v_{2j+1}\right] \tag{12}.$$

To reflect the odd and even summations, (12) is rewritten as

$$V_k = \frac{1}{N}\left[V_k^e + W^k V_k^o\right] \qquad \forall_{k\in[0\ldots N-1]} \tag{13}.$$

Q.E.D.

The implications of (13) are that we can divide the sequence into odd and even numbered samples. Thus the Danielson-Lancoz lemma enables a divide and conquer algorithm to recursively split the sample sequence in half. The computational result of the Danielson-Lancoz lemma is that the $O(N^2)$ DFT may be computed in $O(N \log N)$ time.

The Danielson-Lancoz lemma shows that a sequence must be divided up into its odd and even subsets. That these subsets must in-turn be divided into their subsets. This continues until we have only two members per subset. An illustration of this subdivision, for *N=8*, is shown in Figure 1.

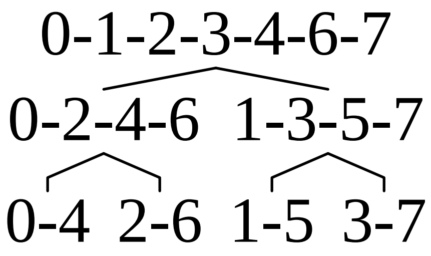

Figure 1 Decimation in time.

It is natural to implement the decimation in time using recursive calls with odd and even sets. It has been shown, however, that a recursive implementation is six times slower than a non-recursive implementation [Gonzalez et al.]. Figure 2 shows the Cooley-Tukey algorithm using bit-reversal in order to decimate in time without recursion.

N	A	B	C	C	B	A	bitr(N)
0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	4
2	0	1	0	0	1	0	2
3	0	1	1	1	1	0	6
4	1	0	0	0	0	1	1
5	1	0	1	1	0	1	5
6	1	1	0	0	1	1	3
7	1	1	1	1	1	1	7

Figure 2. An Example of how to decimate by bit reversal

To arrive at the bit reversal, we implement a Java method in the FFT class:

```
int bitr(int j) {
     int ans = 0;
     for (int i = 0; i< nu; i++) {
             ans = (ans <<1) + (j&1);
             j = j>>1;
     }
     return ans;
}
```

The *bitr* method works by linking together two software shift-registers, as shown in Figure 3.

$$j_n \quad j_{n-1}\text{K} \quad j_1 \quad j_0$$

$$a_n \quad a_{n-1}\text{K} \quad a_1 \quad a_0$$

Figure 3. The *j* and *a* registers are linked with the + operator.

After the decimation in time is performed, the balance of the computation is optimization hacks and housekeeping. For example, a simplification results from V_k being periodic in N so that $V_{k+N} = V_k$.

Proof:

Recall that the DFT is given by:

$$V_k = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \tag{14}$$

so that

$$V_{k+N} = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ij(k+N)/N} v_j \tag{15}$$

Expanding the exponents and simplifying using

$$V_{k+N} = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N} e^{-2\pi ijN/N} v_j \tag{16}$$

with $e^{-2\pi ij} = \cos(-2\pi j) + i\sin(-2\pi j) = 1$ yields:

$$V_{k+N} = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \tag{17}$$

with

$$V_{k+N} = V_k \tag{18}$$

Q.E.D.

In addition, it can be shown that

$$W^{k+N/2} = -W^k \qquad 0 \le k \le N/2 \tag{19}$$

Proof:

Using

$$W = e^{-2\pi i/N}$$

So that

$$e^{-2\pi i(k+N/2)/N} = \cos(-2\pi(k+N/2)/N) + i\sin(-2\pi(k+N/2)/N)$$

with

$$\begin{aligned}\cos(-2\pi(k+N/2)/N) &= \cos(2\pi k/N+\pi) = -\cos(2\pi k/N)\\ \sin(-2\pi(k+N/2)/N) &= \sin(2\pi k/N+\pi) = -\sin(2\pi k/N)\end{aligned}. \tag{20}$$

this leads to:

$$W^{k+N/2} = -W^{k} \qquad 0 \le k \le N/2$$

Q.E.D.

A further efficiency may be had by the use of the recurrence relation

$$W^{j}W^{j(k-1)} = W^{j}W^{jk-j} = W^{j}W^{jk}W^{-j} = W^{jk} \tag{21}$$

Proof:

$$\begin{aligned}
W^{jk} &= e^{-2\pi ijk/N} = \cos\left(-2\pi jk/N\right) + i\sin(-2\pi jk/N) \\
W^{jk} &= \cos\left(-2\pi jk/N\right) + i\sin(-2\pi jk/N) \\
W^{jk} &= \left[\cos\left(-2\pi j/N\right) + i\sin(-2\pi j/N)\right] \\
&\quad * \left[\cos\left(-2\pi j(k-1)/N\right) + i\sin(-2\pi j(k-1)/N)\right] \\
W^{jk} &= W^{j}W^{j(k-1)}
\end{aligned} \tag{22}$$

Alternative Proof:

$$W^{j}W^{j(k-1)} = W^{j}W^{jk-j} = W^{j}W^{jk}W^{-j} = W^{jk}$$

Q.E.D.

The real and imaginary parts of (22) are given by

$$real(z_1z_2) = x_1x_2 - y_1y_2$$

so that

$$W_r^{jk} = W_r^{j}W_r^{j(k-1)} - W_i^{j}W_i^{j(k-1)} \tag{23}$$

and the imaginary part of (22) is given by:

$$imaginary(z_1z_2) = x_1y_2 + y_1x_2$$

so that

$$W_i^{jk} = W_r^{j}W_i^{j(k-1)} + W_i^{j}W_r^{j(k-1)} \tag{24}$$

Equations (23) and (24) form the basis of the recurrence relationships that enables the quick computation of the next W^{jk} based on the previous W^{jk}. An implementation of (24) follows:

```
1.          // (eq 23) and (eq 24)
2.          wtemp = Wjk_r;
3.          Wjk_r = Wj_r * Wjk_r  - Wj_i * Wjk_i;
4.          Wjk_i = Wj_r * Wjk_i  + Wj_i * wtemp;
```

Line 2 shows the introduction of *wtemp*, a temporary variable that facilitates the computation of the multiplication of the two complex numbers.

2 THE FFT CLASS

The grapher package provides a simple interface to make an automatically scaled graph. Generally only a single method is invoked. This is best shown by the following example:

```
public void makeHanning () {
 double window[];
     window = makeHanning(256);
     Graph.graph(window,
                            "The Hanning window","f");
 }
```

Where the “The Hanning window” string appears along the x-axis and “f” appears on the y-axis. The Graph.graph may be invoked directly because the graph method is static. Also, it only graphs an array of type double.

2.1. Class Summary

```
package lyon.audio;
import java.io.*;
import java.awt.*;
import grapher.Graph;
import futils.bench.Timer;
public class FFT extends Frame {
public FFT(int N)
public FFT()
public void graphs()
public void graphs(String t)
public void setTitle(String t)
public static double getMaxValue(double in[])
public static int log2(int n)
public static double[] arrayCopy( double [] in)
public double [] computePSD ()
public double[] dft(double v[])
public double[] idft()
public double [] getReal()
public double [] getImaginary()
public void forwardFFT(double in_r[], double in_i[])
public void reverseFFT(double in_r[], double in_i[])
public void printArray(double[] v,String title)
public void printArrays(String title)
public void printReal(String title)
public static void main(String args[])
public static void timeFFT()
public static void testFFT()
public static void testDFT()
}
```

2.2. Class Usage

The FFT class maintains internal data arrays that are stored as doubles. These arrays are private and are used to assist computations. Further, the in-place Cooley-Tukey algorithm employed for the fast transform is destructive for the original data. The FFT class in the lyon.audio package uses doubles for all computations. This class is for 1-D (audio) transforms.

Suppose the following variables are predefined:

```
FFT f;
int N = 8;
double inputArray[];
String title = "My data title";
double aDoubleArray[];
double in_r[];
double in_i[];
```

To make a new instance of the FFT class, and allocate two internal arrays of double, each of length N:

```
f = new FFT(N);
```

To make a new instance of the FFT class, with no memory allocation:

```
f = new FFT();
```

To graph the real and imaginary data arrays:

```
f.graphs();
```

To graph the real and imaginary data arrays with a title:

```
f.graphs(title);
```

To set the title for the graphs:

```
f.setTitle(title);
```

To get the maximum value of an inputArray:

```
FFT.getMaxValue(inputArray);
```

To compute the floor of the log of an int to base 2:

```
int numberOfBits = FFT.log2(N);
```

To copy an array of double:

```
aDoubleArray = FFT.arrayCopy(inputArray);
```

To compute the psd (power spectral density) of the last dft or fft:

```
aDoubleArray = f.computePSD();
```

To non-destructively compute the dft of an input array and return the psd:

```
aDoubleArray = f.dft(inputArray);
```

DFT, IDFT, FFT and IFFT alter the internal data structures in an instance of the FFT class. To get the real part of the last transform:

```
aDoubleArray = f.getReal();
```

To get the imaginary part of the last transform:

```
aDoubleArray = f.getImaginary();
```

To take the idft of the internal data and return the real part:

```
aDoubleArray = f.idft();
```

To take the forward fft on two input arrays, destructively:

```
f.forwardFFT(in_r, in_i);
```

To take the inverse FFT on two input arrays, destructively

```
f.reverseFFT(in_r, in_i);
```

To print an array of double, with a title:

```
    f.printArray(aDoubleArray, title);
```

To print the internal real and imaginary arrays, with a title:

```
    f.printArrays(title);
```

To print the internal real array, with a title:

```
    f.printReal(title);
```

To test the DFT, IDFT, FFT and IFFT:

```
    FFT.main();
```

To time the FFT:

```
    FFT.timeFFT();
```

To test the FFT:

```
    FFT.testFFT();
```

To test the DFT:

```
    FFT.testDFT();
```

2.3. Testing the FFT and IFFT

The FFT class has a static method that permits the testing of the DFT, IDFT, FFT and IFFT. It also performs timing for a transform of 2048 doubles. To run this test, you must invoke

```
    FFT.main();
```

The code for the FFT.main method follows:

```
     public static void main(String args[]) {
          testDFT();
          timeFFT();
          testFFT();
     }
```

The test methods are run on an 8 point input array consisting of a linear ramp. This is to provide a short sequence of input data that can be verified by printing. The timing is performed on 2048 samples stored in two arrays of 2048 doubles each (real and imaginary). The output of the main method follows:

```
Executing DFT on 8 points...
Executing IDFT on 8 points...
j      x1[j]   re[j]          im[j]  v[j]
0      0 3.5   0 -3.10862e-15
1      1 -0.5  1.20711        1
2      2 -0.5  0.5            2.00000
3      3 -0.5  0.207107       3
4      4 -0.5  0 4
5      5 -0.500000            -0.207107      5
6      6 -0.500000            -0.5  6
7      7 -0.5  -1.20711       7
fft: bit reversal
Time for 2048point fftTime 0.178000 sec
fft: bit reversal
Time for 2048point ifftTime 0.164000 sec
Starting 1D FFT test...
fft: bit reversal
```

```
fft: bit reversal
j      x1[j]   re[j]             im[j]v[j]
0      0 3.5   0 0
1      1 -0.5  1.20711           1.00000
2      2 -0.5  0.5               2.00000
3      3 -0.500000               0.207107       3.00000
4      4 -0.5  0 4
5      5 -0.5  -0.207107         5
6      6 -0.5  -0.5              6
7      7 -0.500000               -1.20711       7
```

The reader will see that the input and output are highly correlated for both the DFT and FFT. The surprising thing is how accurate these two radically different algorithms and implementations are. Also, recall that the execution times for the DFT was benchmarked at 55 seconds. The FFT implementation is run in 0.178 seconds, a 308 times speed up. Keep in mind, at 8000 samples per second, the 2048 samples represent 0.256 seconds of data. Also, on a limited data rate connection (such as a 28.8 kbps modem) the time to transmit the data is 2048*8 bits /28800 bits/sec = 0.56 seconds. We suggest that many dial-up users experience a slower connection than the maximum their modem permits. Thus, there is a window of opportunity for devising a real-time codec (IN JAVA!!) able to perform FFT based compression algorithms. An algorithm based on transform compress typically takes the original data, performs the forward transform, selects coefficients, quantizes and then transmits. Data is recovered by taking the coefficients and performing an inverse transform. Very Low Bit Rate Voice Compression (VLBRVC) is a rich and growing field that lies beyond the scope of this paper. See http://www.bdti.com/faq/dsp_faq.htm for an FAQ that relates to this and other DSP topics.

2.4. Implementing the FFT.testFFT

The following code shows how to use the FFT class to perform a forward and inverse FFT. The static nature of the testFFT method indicates that invocation may be performed without making an instance of the FFT class.

Line 3 makes an instance of the FFT class, without performing any allocation for the internal data structures. Thus the allocation and copying of arrays is performed outside of the *forwardFFT* methods. This is due, in part, to the destructive nature of the in-place Cooley-Tukey FFT algorithm. The trade-off is that the programmer must keep track of the data that is being processed by the forwardFFT. The alternative is to automatically copy arrays, perform the in-place forwardFFT, then return the copies. Our findings indicate that the dynamic allocation of memory (particularly during the image processing, seen later in this book) can slow performance by up to 100 times! Thus, the house keeping chores performed by the programmer are warranted by a leap in performance.

```
1.   public static void testFFT()   {
2.   System.out.println("Starting 1D FFT test...");
3.   FFT f = new FFT();
```

Line 4 may be altered to any number of samples, N, but a large N will result in a large printout.

```
4.    int N = 8;
5.    int numBits = f.log2(N);
```

Lines 6-8 set up the input data to be a ramp that varies from 0 to N.

```
6.     double x1[] = new double[N];
7.    for (int j=0; j<N; j++)
8.           x1[j] = j;
```

Now the housekeeping. The programmer, interested in keeping copies of the original data, the result of the forward FFT and the result of the inverse FFT, must allocate four arrays! This is an unusual case, as it requires that all intermediate results be kept for checking purposes. Normally, production code would not have to keep all intermediate results.

```
9.     double[] in_r = new double[N];
10.     double[] in_i = new double[N];
```

The in_r and in_i arrays are copies of the input data, with the imaginary component equal to zero. Real data (like audio data) often has a zero imaginary component. There are algorithms that can save significant time by taking advantage of the zero imaginary part of the input data. This requires a different FFT implementation.

```
11.     double[] fftResult_r = new double[N];
12.     double[] fftResult_i = new double[N];

13.     // copy test signal.
14.     in_r = arrayCopy(x1);
```

Line 14 copies the input data into in_r.

```
15.     f.forwardFFT(in_r, in_i);
```

Line 15 replaces in_r and in_i with the forward FFT results.

```
16.     // Copy to new array because IFFT will
17.     // destroy the FFT results.
18.     fftResult_r = arrayCopy(in_r);
19.     fftResult_i = arrayCopy(in_i);
20.     f.reverseFFT(in_r, in_i);
21.     System.out.println("j\tx1[j]\tre[j]\tim[j]\tv[j]");
22.     for(int i=0; i<N; i++) {
23.          System.out.println(
24.          i + "\t" +
25.          x1[i] + "\t" +
26.          fftResult_r[i] + "\t" +
27.             fftResult_i[i] + "\t" +
28.             in_r[i]);
29.     }

30.   }
```

N	dft rate	fft rate
2	65	25
4	250	65
8	500	64
16	1000	103
32	2000	344
64	2000	821
128	4129	1174
256	1641	2753
512	1561	10894
1024	992	5044
2048	512	32508
4096	260	65016
8192	127	256000
16384	61	207392
32768	30	300624
65536	15	595782

Figure 1. Run Rate of the DFT vs the FFT

Figure 1 shows the rate of the DFT and FFT as a function of array length. The rate is given in floating point sample, per second, on a T2300 Intel CPU running at 1.66 Ghz with 504 MB of RAM under JDK 1.5. We ran the benchmarks again in Figure 2

N	dft	fft
2	286	83
4	444	129
8	889	104
16	1778	485
32	3200	561
64	3556	1085
128	3459	1208
256	3413	2226
512	2498	2860
1024	1513	4146
2048	803	6282
4096	409	4506
8192	204	7628
16384	100	7907
32768	49	5221
65536	25	6369

Figure 2. Run Rate of the DFT vs the FFT

Figure 2. shows Intel Core 2 CPU T7200 @2.00GHz with 1 GB RAM running JVM 1.6.0_07". We also see a great deal of variation in the benchmarking between the different JVMs and machines.

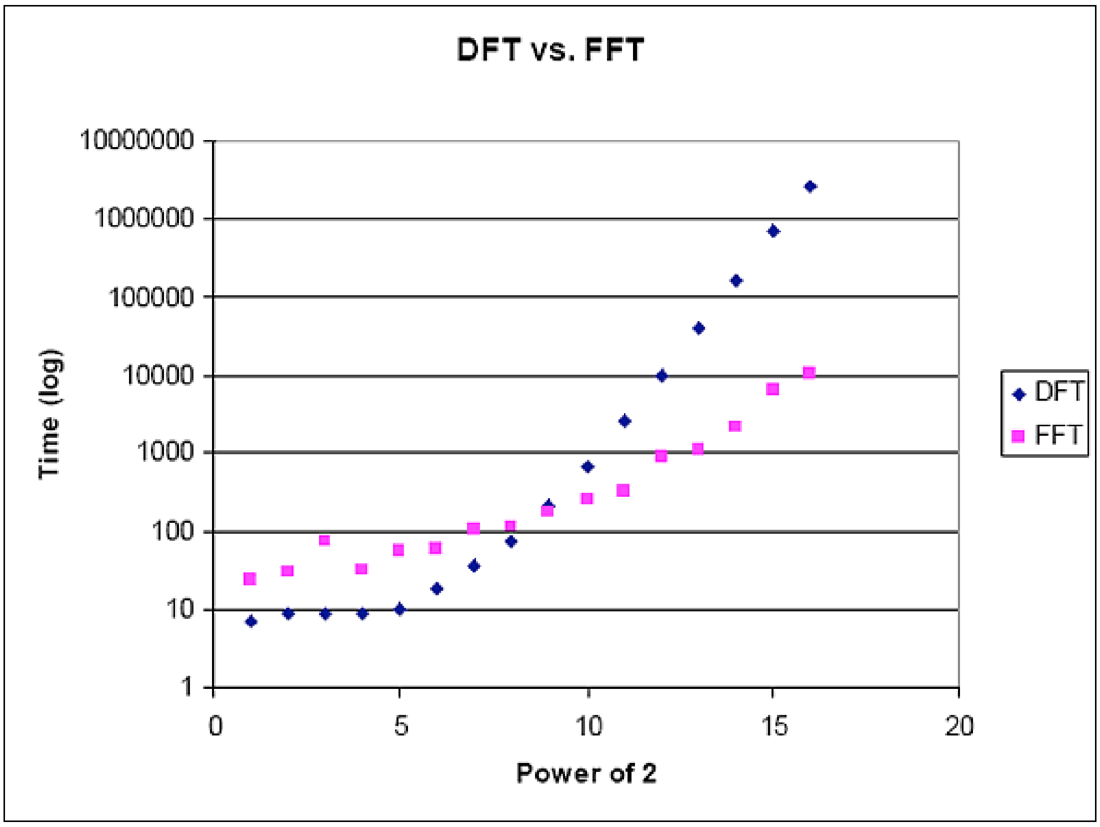

Figure 3. Comparing FFT vs DFT, Log scale

Figure 3 shows a crossover that exists between the DFT and the FFT. Plotted on a log scale, as a function of N, for values below 512 samples, the DFT is faster than the FFT, and should be the preferred means of performing the Fourier transform.

3 SUMMARY

This paper demonstrates that, for small numbers of samples (less than 512) the DFT is preferred over the FFT. We have also seen a great deal of variation in the performance of the benchmark, as we change from one JVM to another. Finally, we have created a new means of measuring the rate of the transform, the number of samples per second processed. This is of direct concern to those who are interested in real-time processing of signals as well as those who are interested in faster algorithms.

About the author

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 40 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.