JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 8, No. 3, May-June 2009

# The Discrete Fourier Transform, Part 1

**By Douglas Lyon**

**Abstract**

This paper describes an implementation of the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT). We show how the computation of the DFT and IDFT may be performed in Java and show why such operations are typically considered slow.

This is a multi-part paper, in part 2, we discuss a speed up of the DFT and IDFT using a class of algorithms known as the FFT (Fast Fourier Transform) and the IFFT (Inverse Fast Fourier Transform).

Part 3 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 DISCRETE FOURIER TRANSFORM

Let

$$v_j, j \in [0...N-1] \quad (1)$$

be the sampled version of the waveform, *v(t)* where *N* is the number of samples.

Equation (1) numbers from zero, rather than one, to reflect the start point of arrays in Java. Fourier transform of *v(t)* is given by

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

*V(f)* can only exists if *v(t)* is absolutely integrable, i.e.

$$\int_{-\infty}^{\infty} |v(t)|\,dt < +\infty$$

In fact, *v(t)* is integrable if and only if |*v(t)*| is integrable, so the terms “absolutely integrable” and “integrable” amount to the same thing. The inverse Fourier transform of *V(f)* is given by

$$v(t) = F^{-1}[V(f)] = \int_{-\infty}^{\infty} V(f)e^{2\pi ift}\,dt \quad (3).$$

In order to compute (2) for the sampled waveform of (1), we use the DFT:

$$V_k = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \qquad (4).$$

Using Euler's relation

$$e^{i\theta} = \cos\theta + i\sin\theta \qquad (4a).$$

In the kernel of the transform in (4) to be re-written as:

$$e^{-2\pi ijk/N} = \cos(-2\pi jk/N) + i\sin(-2\pi jk/N) \qquad (4b).$$

An even function is one that has the property that $f(-x) = f(x)$. An odd function has the property that $f(-x) = -f(x)$. Using this definition, $\sin(-\theta) = -\sin\theta$, is an odd function and, since $\cos(-\theta) = \cos\theta$, cosine is an even function. Using the odd-even function properties of sine and cosine, we rewrite (4b) as:

$$e^{-2\pi ijk/N} = \cos(2\pi jk/N) - i\sin(2\pi jk/N) \qquad (4c).$$

Substituting (4c) into (4) we obtain:

$$V_k = \frac{1}{N}\sum_{j=0}^{N-1} (\cos(2\pi jk/N) - i\sin(2\pi jk/N)) v_j \qquad (5).$$

When implementing the Java program we compute the real and imaginary parts of (5) using:

```
for(int j = 0; j < N; j++) {
   twoPijkOnN = twoPikOnN * j;
   r_data[k] +=  v[j] * Math.cos( twoPijkOnN );
   i_data[k] -=  v[j] * Math.sin( twoPijkOnN );
}
r_data[k] /= N;
i_data[k] /= N;
```

The above loop is executed N times, dividing the spectrum into $N$ buckets. Each spectral bucket is accessed using the *frequency index*, $k$. The frequency of bucket $k$ is given by:

$$f_k = \frac{k}{N\Delta t} \qquad (6),$$

The sampling period, $\Delta t$, is computed from the sampling rate:

$$\Delta t = 1/f_s \qquad (7).$$

Equations (5) and (6) show that the spectrum is described by integral harmonics of $f_s/N$. For example, suppose that the sampling rate, $f_s$, is 8000 Hz and that the number of points is 2048; then the smallest change in frequency that can be detected

is given by $f_1 = \frac{1}{2048} 8000 \approx 4Hz$. Therefore, integral harmonics of 4 Hz will be used to approximate *v(t)*.

The psd (Power Spectral Density) gives the power at a specific frequency index, $psd_k$. We compute the power at any $f_k$ by summing the square of the real and imaginary components of the amplitude:

$$psd_k = real^2(V_k) + imaginary^2(V_k) \qquad (8).$$

The amplitude spectral density is given by the square root of the power spectral density. An implementation of the DFT follows:

```
public class FFT  {

 double r_data[] = null;
 double i_data[] = null;
  ...
 public double[] dft(double v[]) {
  int N = v.length;

  double t_img, t_real;

  double twoPikOnN;
  double twoPijkOnN;

  // how many bits do we need?
  N=log2(N);
  //Truncate input data to a power of two
  // length = 2**(number of bits).
   N = 1<<N;

  double twoPiOnN = 2 * Math.PI / N;
  // We truncate to a power of two so that
  // we can compare execution times with the FFT.
  // DFT generally does not need to truncate its input.

  r_data = new double [N];
  i_data = new double [N];
  double psd[] = new double[N];



  System.out.println("Executing DFT on "+N+" points...");

  for(int k=0; k<N; k++) {

     twoPikOnN = twoPiOnN *k;


     for(int j = 0; j < N; j++) {
        twoPijkOnN = twoPikOnN * j;
        r_data[k] +=  v[j] * Math.cos( twoPijkOnN );
```

```
            i_data[k] -=  v[j] * Math.sin( twoPijkOnN );
        }
        r_data[k] /= N;
        i_data[k] /= N;

        psd[k] =
          r_data[k] * r_data[k] +
          i_data[k] * i_data[k];

    }
    return(psd);
}
```

## 2 BIT COMPUTATIONS AND A LOG REVIEW

The log function is defined by:

$$\text{if } x = a^y \text{ then } y = \log_a x$$

For example, if $x = 2^{10}$ then $10 = \log_2 x$.

The following are known as the laws of logarithms:

$$\log_a(xy) = \log_a x + \log_a y$$
$$\log_a(x / y) = \log_a x - \log_a y$$
$$\log_a x^n = n \log_a x$$

For example, to find $\log_2 4096$ use:

$$4096 = 2^y$$
$$\ln 4096 = y \ln 2$$
$$\frac{\ln 4096}{\ln 2} = y = 12$$

so,

$$\log_2 x = \frac{\ln x}{\ln 2}$$

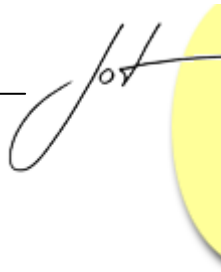
Also, to find $\log_B x$ use:

$$\log_B x = \frac{\ln x}{\ln B} = \frac{\log_{10} x}{\log_{10} B} \qquad (9)$$

To compute (9) to the base 2, we use:

```
public static int log2(int n) {
    return (int) (Math.log(n)/Math.log(2.0));
}
```

DFT does not need to the length of the data to be an integral power of two. On the other hand, the *radix 2* FFT (Fast Fourier Transform) is a common implementation of the FFT that requires data be an integral power of two in length. To fairly compare the



performance of the DFT implementation against the FFT implementation, we must perform the same truncation for both algorithms.

## 2.1. Benchmarking the DFT

Benchmarking is a craft that permits the measurement of hardware and software performance. One of the remarkable things about Java is that the compile-once-run-anywhere attribute enables the same bytes codes to be executed on different implementations of the Java machine. In addition, we have similar Java *virtual* machines that are implemented on different hardware platforms. This enables a base-line comparison of different hardware platforms when executing the byte codes. One use of benchmarking is to measure the effect of the implementation of an algorithm on the execution time. Also of interest is the measurement of two different algorithm's execution time for the same data.

There is a difference between the performance measurement of various algorithmic *implementations* and the performance measurement of various *algorithms*. Better algorithms are generally combined with better implementations to yield performance improvement.

## 2.2. BenchMarking the DFT method

In this section we show how to use the Timer class to measure the execution time of the DFT. This serves both as an example of how to use the DFT and how to benchmark a method. The following example of how to use the DFT:

```
public void dft() {
      FFT f = new FFT();
      double [] doubleData = ulc.getDoubleArray();
      double [] psd;
// Time the fft
      Timer t1 = new Timer();
      t1.mark();
      psd=f.dft(doubleData);
// Stop the timer and report.
      t1.record();
      System.out.println("Time to perform DFT:");
      t1.report();
      f.graphs();
      new DoubleGraph(psd,"psd");
}
```

## 3 THE INVERSE DFT

Recall also, that the inverse Fourier transform of *V(f)* was given by

$$v(t) = F^{-1}\left[V(f)\right] = \int_{-\infty}^{\infty} V(f)e^{2\pi ift}\,dt \qquad (3).$$

In order to compute (3) for the sampled waveform of (1), we must perform an inverse discrete time Fourier transform called the IDFT (Inverse Discrete Fourier Transform). The IDFT is given by

$$V_k = \sum_{j=0}^{N-1} e^{2\pi ijk/N} v_j \qquad (10).$$

Recall that in (4):

$$V_k = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \qquad (4)$$

the summation result is multiplied by 1/N. This is not the case in (10). In some expositions, both (10) and (4) are multiplied by $1/\sqrt{N}$, in order to keep the DFT and IDFT symmetric. We abandoned such an approach during development because it both complicates the presentation of the PSD and requires slightly more computation.

Substituting Euler's relation,

$$e^{i\theta} = \cos\theta + i\sin\theta \qquad (4a)$$

into (10) results in:

$$v_k = \sum_{j=0}^{N-1}\left[\cos\left(2\pi jk/N\right) + i\sin\left(2\pi jk/N\right)\right]V_j \qquad (11).$$

The multiplication of two complex numbers result may be expressed as:

$$z_1 z_2 = (x_1 + iy_1)(x_2 + iy_2) = x_1x_2 - y_1y_2 + i\left(x_1y_2 + y_1x_2\right) \qquad (11a)$$

Based on (11a) we conclude that the real part of $z_1 z_2$ is given by:

$$real(z_1 z_2) = x_1 x_2 - y_1 y_2 \qquad (11b)$$

and the imaginary part of $z_1 z_2$ is given by:

$$imaginary(z_1 z_2) = x_1 y_2 + y_1 x_2 \qquad (11c).$$

If we use only a real-valued signal on input (as opposed to complex-valued signal, as may be common in radar systems) we need not to compute the result in (11c). Substituting (11b) into (11) yields:

$$real(v_k) = \sum_{j=0}^{N-1} \left[ \cos(2\pi jk / N) real(V_j) - \sin(2\pi jk / N) imaginary(V_j) \right] \qquad (12).$$

Computing only the real part of the IDFT saves $2N$ multiplies and $N$ additions for each frequency index, $k$ computed in (12) than in (11). A comparison of (12) with (5),

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} \left( \cos(2\pi jk / N) - i \sin(2\pi jk / N) \right) v_j \qquad (5)$$

shows that both take about the same amount of time to compute (something that our experiments confirm).

The following code implements the IDFT shown in (12):

```
  // assume that r_data and i_data are
  // set. Also assume that the real
  // value is to be returned
public double[] idft() {
   int N = r_data.length;
   double twoPiOnN = 2 * Math.PI / N;

   double twoPikOnN;
   double twoPijkOnN;

   double v[] = new double[N];

   System.out.println("Executing IDFT on "+N+" points...");


   for(int k=0; k<N; k++) {
          twoPikOnN = twoPiOnN *k;
      for(int j = 0; j < N; j++) {
         twoPijkOnN = twoPikOnN * j;
         v[k] +=  r_data[j] * Math.cos(twoPijkOnN )
                  - i_data[j] * Math.sin(twoPijkOnN );

      }
   }
   return(v);
}
```

## 4 NUMERIC CHECK OF THE DFT AND IDFT

A numeric check should be an integral part of every class. The FFT class contains a method called *testDFT* whose role is to verify the correctness of the DFT and IDFT implementation. With the number of samples set to 8, the testing method is able to print the input and output points for human comparison.

```
public static void testDFT () {

 int N = 8;
 FFT f = new FFT(N);

 double v[];

 double x1[] = new double[N];
 for (int j=0; j<N; j++)
      x1[j] = j;

 // take dft
 f.dft(x1);


 v = f.idft();
 System.out.println("j\tx1[j]\tre[j]\tim[j]\t v[j]");
 for (int j=0; j < N; j++)
     System.out.println(
     j+"\t"+
     x1[j]+"\t"+
     f.r_data[j]+"\t"+
     f.i_data[j]+"\t"+
     v[j]);

}
```

We print the intermediate DFT results to permit a detailed check against variations in implementation. Full data disclosure allows a base-line comparison between different implementations of the DFT. As we shall see in the following section, this is a comforting data result, particularly when compared with the FFT data.

```
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
```

While the input is not quite the same as the output, it is quite close.

| Length | Time in MS |
|---|---|
| 4 | 0 |
| 8 | 0 |
| 16 | 0 |
| 32 | 0 |
| 64 | 0 |
| 128 | 16 |
| 256 | 16 |
| 512 | 47 |
| 1024 | 234 |
| 2048 | 859 |
| 4096 | 3219 |
| 8192 | 13281 |
| 16384 | 54282 |
| 32768 | 235046 |
| 65536 | 934375 |

Figure 1. Runtime of the DFT

Figure 1 shows the runtime of the DFT as a function of array length. The machine is a 1.5 GHz Celeron with 1 GB RAM and a 1.6.11 version of the JVM. The order N**2 nature of the DFT execution time is more clearly seen in Figure 2.

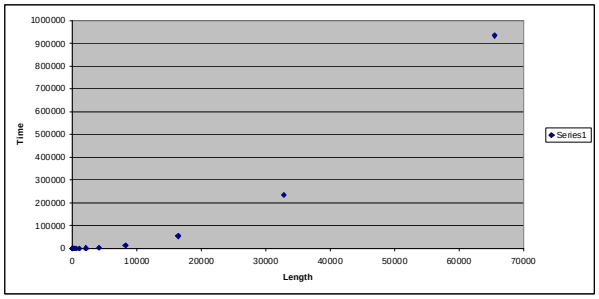



Figure 2. Graph of the DFT Runtime

For the u-law CODEC, audio is sampled at 8 KHz [Lyon 08G]. A sample window of 256 samples last 32 ms, but the DFT can be performed in just 16 ms. This makes the DFT fast enough for real-time operation, using a modest machine, u-law sample rates and 32 ms windowing.

## 5 SUMMARY

The DFT is typically held as too slow for direct computation, because of its' O(N**2) complexity. However, for small windows of time, on even modest machines and voice-grade single-channel 8-bit audio, we find that the computation can be fast enough for real-time processing. This was not always the case, as shown in [Lyon 97] and [Lyon 99]. In fact, it was always a given that Java was too slow for anything approaching real-time.

So, what has changed? We now see a mix of very fast hot-spot compiler technologies, integrated into the JVMs. We also see the cheapening of fast hardware, to the point where machines that run faster than 1.5 GHz are common.

On the other hand, if increase sample rates are involved, with better quality signals, we need to narrow the event window, get a faster machine, or go to a better algorithm. Considering how slow the DFT algorithm is, running on a virtual machine, it is good to know that Java is able to keep up, using only modest hardware. In our next paper we will address a class of algorithms known as the FFT.

## REFERENCES

[Lyon 08G] "The U-Law CODEC",by Douglas A. Lyon, Journal of Object Technology, vol. 7, no. 8, November-December 2008, pp. 17-31.

[Lyon 97] *Java Digital Signal Processing*, Co-Authored with H. Rao, M&T Press (an imprint of Henry Holt). November 1997.

[Lyon 99] *Image Processing in Java*, Douglas A. Lyon, Prentice Hall. April 1999

### 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.