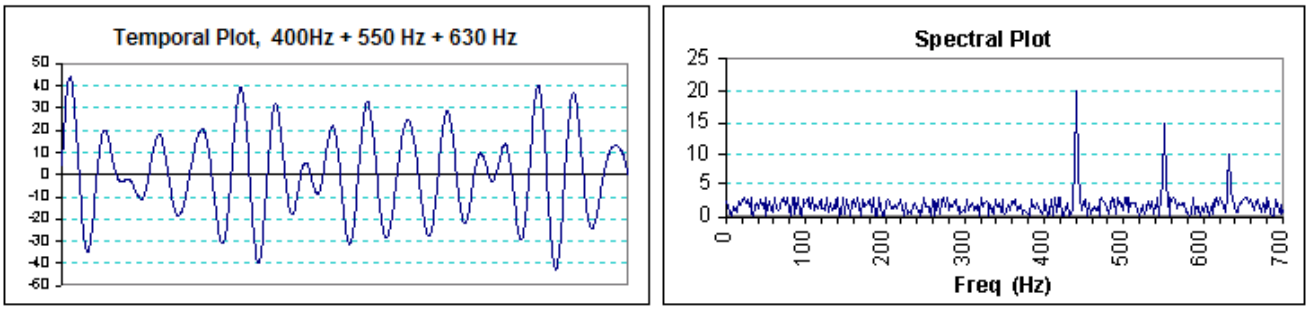
The Fast Fourier Transform (FFT):

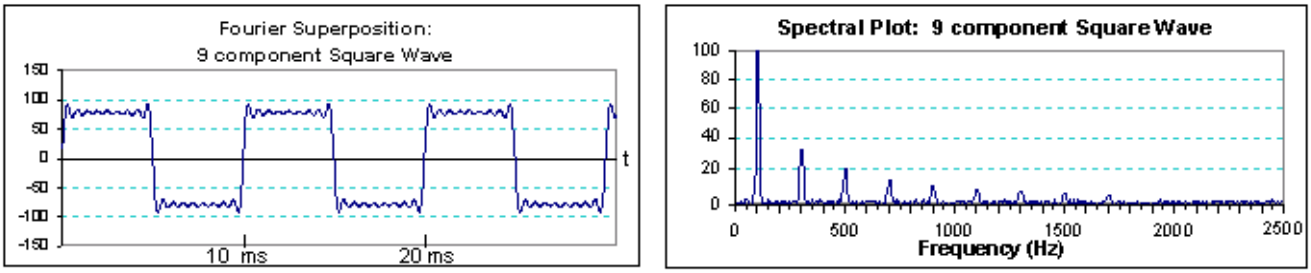
Steve Lee
EpiphanyBySteveLee.com

Introduction:

Unlike most of the other projects described in this section of our website (EpiphanyBySteveLee.com, misc), this document offers something of a basic tutorial, with the primary focus being on the now ubiquitous Fast Fourier Transform (FFT). In this tutorial we include both a general description of the FFT, how it is implemented, critical concerns related to its use, and a brief snippet of software code written in Java (and adaptable for C/C++ applications) which can be used in analyzing a wide range of data (e.g. audio recordings, forensic lab data, stock market historical data, etc.).

Though analyzing data accumulated over time tells us something about the object being studied, transforming that time-related data to see its "spectral" nature brings an additional level of clarity and insight into the analysis. In other words, we often learn a great deal more about a phenomenon under study by looking at it not only from how it behaves over time, but also by then transforming that time-related data and looking at it from a frequency-related view as well, in order to identify any *cyclic* (repetitive) aspects of that data (i.e. "frequency peaks").





Examples of such an FFT application include analyzing audio recordings of someone's voice to define their unique "tone quality" spectral "fingerprint"; analyzing the electrical signals emitted by a test circuit exposed to an injected pulse of energy to help describe the spectral "band pass" of that circuit; analyzing video images of a valley covered in vegetation to strip

away all random elements (i.e. foliage) to highlight all man-made (periodic) structures in the area; the studying of the light radiated off an unknown object pulsed by a laser to help identify the composition of that test sample, (etc.).

The FFT is such an extremely useful tool in fact, that it has become an absolutely vital component in a wide range of things we use everyday, including our Digital TV receivers and cell phones. The latest generation cell phones in particular now use the FFT on an almost *continuous* basis as an integral part of how they send and receive radio signals, along with the information embedded in those signals. Being such a indispensable thing in our modern world, we thought a brief "user-friendly" introduction describing how the FFT works (along with a brief sample of software code for those who would like to take it for a test run) might be of interest to anyone even remotely interested in how the modern "electronic universe" in which we now live operates.

How it works:

The *general* Fourier Transform was originally devised way back in the late 1800's by a rather clever guy named Jean Baptist Fourier. Since then it has itself become an absolutely essential component of a wide variety of innovations in the hard sciences, including quantum physics, semiconductor physics, chemistry, nano-scale microscopy, wireless telecommunications, as well as a whole host of other engineering applications too numerous to list here.

The specialized Fast Fourier Transform (FFT) is itself an adapted version of the *general* Fourier Transform (FT). By exploiting certain mathematical symmetries and relationships that naturally exist or can be built into a digitally implemented Fourier Transform, the FFT effectively streamlines the FT processing down into a much faster "computer friendly" hybrid version of the FT.

In either the general FT or the digital FFT version, this transformation process takes a set of time-related data samples and mathematically transforms those time samples, "switching" them into their related frequency version of that same data. This is possible due to the fact there is a direct relationship between the frequency of a periodic waveform and the time between each cycle in that same waveform. In the case of a sine or cosine wave for example, the frequency of the wave is the inverse of the time between each wave pulse, i.e. $F = 1/t$ (e.g. if $F = 1000$ Hz, then the time between each wave pulse is $1/1000 = 1$mS). This inherent relationship is exploited in the Fourier Transform to allow us to switch back and forth between a temporal view of an event, and the related spectral view of that same event.

Since the general FT operates on a continuous function, while digital computers operate in very discrete "steps" (i.e. processing one single command or chunk of data at a time), the first step towards developing a digital computer version of the FT is to define the "Discrete Fourier Transform" (DFT).

When we analyze something in a lab, we typically take data samples of that object at regular intervals, recording each sample in a sample array. If we define "X(t)" to be that array of data samples taken over time, the Discrete Fourier Transform process effectively takes each sample in that array and multiplies it by a Cosine and a Sine "wavelet", as if each Cosine and Sine factor were something similar to a sinusoidal "scaling" factor (sometimes referred to as the "twiddle" factor). We can write this "scaling"/"twiddling" of each data sample X(t) as:

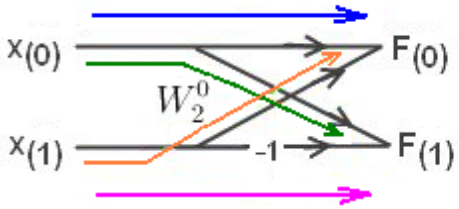
$$\text{DFT:} \qquad F_{(w)} = \sum_{K=0}^{K=N} X_{(t)} * \left(\text{Cos}\left(\frac{2\pi k}{N}\right) + j\,\text{Sin}\left(\frac{2\pi k}{N}\right) \right)$$

Where: "F(w)" is the spectral transformed "twin" of the time samples X(t) created through this transformation process. In this notation, "∑k" tells us to sum up all the "scaled"/"twiddled" time samples (I.e.. each of the X(t) samples multiplied by the Cosine and Sine factors), with "k" being a simple index that ranges from k=0 (i.e. the first time sample) to k=N (where "N" is the total number of time samples recorded in our data array X(t)).

If the total number of samples "N" is small, this discrete version of the general FT would work fine. The problem however is that in most cases, for the data to be useful we typically need to take *many* samples, on the order of 1000 to 10,000 samples. Unfortunately performing all the calculations involved for a sample set containing *thousands* or *tens of thousands* of samples using the DFT, very quickly becomes *extremely* time consuming, even for the typical high speed digital computer available today.

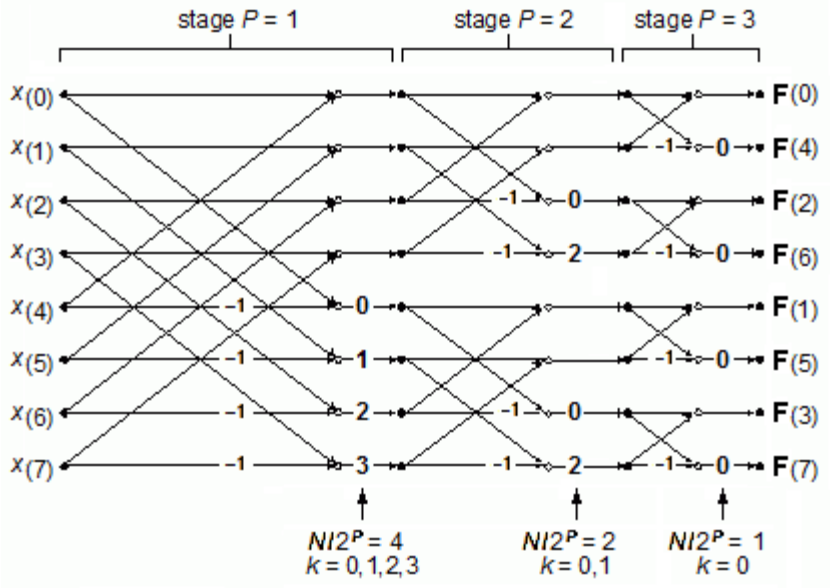
To address this significant problem, the inventors of the FFT (Cooley and Tukey, 1968) realized they could "trim" some of the calculations (and thus processing time) if they exploited the inherent periodicity of the Sine and Cosine wave (i.e. Sin(θ) = Sin(θ + 2π)). For the FFT algorithm to work correctly, we are required to restrict our total number of samples "N" to a number equal to a power of 2 (i.e. N must equal 2^k; e.g. 512, 1024, 4096, etc.). This is due to the fact that all the calculations performed in the FFT are performed in *pairs*, as shown in the following diagrams:

This diagram represents the basic "Butterfly" flow of the FFT calculations, 2 inputs yield 2 outputs.



$$F_{(0)} = x(0) + W_2^0 x(1)$$
$$F_{(1)} = x(0) - W_2^0 x(1)$$

$$\text{with:} \qquad W_N^k = \text{Cos}\left(\frac{2\pi k}{N}\right) + j\,\text{Sin}\left(\frac{2\pi k}{N}\right)$$



8-point FFT signal flow diagram.

From the diagram above, we see that the time-related data samples are effectively "boiled down" through a series of calculation stages, with the output of each stage feeding forward as the input to the next stage (for this reason a great deal of care needs to be taken to insure our process is as error-free as possible; see "caveat" section below). Consequently as our data size gets larger, more calculating stages are required to "boil down" all that time-related data into its spectral counterpart represented by the F(k)'s at the right edge of the diagram.

Caveats:

1) Since the FFT algorithm works the data calculations in "pairs", the number of data samples "N" MUST be 2^k. If the data samples are not 2^k, then a DFT must be used or "phantom" frequency "artifacts" will be created in the FFT.

2) If the sample rate is *less* than the highest frequency component in the data, the inadequate sampling of these higher freq's will create "*Aliasing*". For example, if the sample rate is once every millisecond (i.e. a 1000 Hz sampling rate) and an event occurs *twice* in each millisecond (i.e. a 2000 Hz event), then half of those events will be missed and the flawed sampling will see that event as a 1000 Hz event. A common example of Aliasing occurs in motion pictures as wagon wheels or tires begin to spin faster than the camera shutter speed can capture, making the wheel appear to spin backwards. To correctly capture the full cycle of an event (the positive and negative halves of the cycle), we need to sample slightly greater than 2*Fmax (this is known as the '*Nyquist Theorem*').

3) If some event in our sampling window occurs sporadically or is otherwise discontinuous throughout our sampling process, these "fragment" events can be incorrectly processed as the FFT calculations "bin out" their energy into higher frequency "bins" (e.g. if a transient 1000 Hz waveform is cut short or otherwise "fragmented" during the sampling process, this short fragment sample will be interpreted as a higher frequency). This error effectively creates "phantom" spectral components in the transformed data. In that sense, the resulting energy from these artifacts effectively "leaks" into other frequency "bins" during the processing of those samples, hence this kind of error is known as "Spectral leakage".

4) If the sampling process itself has some "jitter" involved (e.g. poorly regulated sampling clock), the sample end/start points will have discontinuities. This can also contribute to Spectral leakage. Minimizing clock jitter and sync'ing the sampling process to the fundamental frequency in the data helps minimize this problem.

5) Similar artifacts can be minimized (but *not* eliminated) by "windowing" the data and thus attenuating higher frequency transients/noise from the sample set (i.e. windowing essentially applies a mathematical equivalent of a band-pass filter to the data). The "Hamming window" is reasonably effective window function, and consequently is now used heavily in many FFT applications.

6) Since FFT calculations cascade forward from one stage's output to the next stage's input, errors introduced early on in the process *compound* with each subsequent stage. Making matters worse is the fact that ever-larger sample sizes ("N") correspond to a larger number of processing stages, compounding these cascaded errors all the more. It is therefore *extremely* important to minimize all processing errors, including the initial sampling

errors described previously, as well as any rounding errors (etc.) during the calculations performed in each stage.

Sample Code:

The following code sample is from a program originally written in java, however its construction makes it easily adaptable for use in a C/C++ program as well. Note the use of double precision sample variables and data arrays with the specific intent of minimizing the previously mentioned cascading of rounding errors.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void  ff_xform(double timeArr[], double freqArr[], int nn, int isign)
		{
		int n, mmax, m, jj, istep, ii;
		double wtemp, wr, wpr, wpi, wi, theta;
		double tempr, tempi;
		double[] tempArr = new double[2*nn+1];


		if(isign>0)
			{
			for (ii=0;  ii<2*nn;  ii++)
				tempArr[ii] = timeArr[ii];
			}
		else
			{
			for (ii=0;  ii<2*nn;  ii++)
				tempArr[ii] = freqArr[ii];
			}

		n=nn << 1;
		jj=1;
		for (ii=1;  ii<n;  ii+=2)
			{
			if (jj>ii)
				{
				SWAP(tempArr[jj], tempArr[ii]);
				SWAP(tempArr[jj+1], tempArr[ii+1]);
				}
			m=n >> 1;
			while (m>=2  &&  jj>m)
				{
				jj -= m;
				m >>= 1;
				}
			jj += m;
			}
```

```
                mmax=2;
                while (n > mmax)
                        {
                        istep=2*mmax;
                        theta=6.28318530717959/(isign*mmax);
                        wtemp=Math.sin(0.5*theta);
                        wpr = -2.0*wtemp*wtemp;
                        wpi=Math.sin(theta);
                        wr=1.0;
                        wi=0.0;
                        for (m=1;  m<mmax;  m+=2)
                                {
                                for (ii=m;  ii<=n;  ii+=istep)
                                        {
                                        jj = ii+mmax;
                                        tempr = wr*tempArr[jj]-wi*tempArr[jj+1];
                                        tempi = wr*tempArr[jj+1]+wi*tempArr[jj];
                                        tempArr[jj] = tempArr[ii]-tempr;
                                        tempArr[jj+1] = tempArr[ii+1]-tempi;
                                        tempArr[ii] += tempr;
                                        tempArr[ii+1] += tempi;
                                        }
                                wr = (wtemp=wr)*wpr-wi*wpi+wr;
                                wi = wi*wpr+wtemp*wpi+wi;
                                }

                        mmax=istep;
                        }

                ////////////////////////////////////////////////
                if(isign>0)
                        {
                        for (ii=0;  ii<2*nn;  ii++)
                                freqArr[ii] = tempArr[ii];
                        }
                else
                        {
                        for (ii=0;  ii<2*nn;  ii++)
                                timeArr[ii] = tempArr[ii];
                        }
                }

/*****************************************************************************/
 public void  SWAP(double dd1, double dd2)
                {
                double tempf = 0;

                tempf = dd1;
                dd1 = dd2;
```

```
                dd2 = tempf;
                }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```