Ray Mack, W5IFS

17060 Conway Springs Ct, Austin, TX 78717; **w5ifs@arrl.net**

SDR: Simplified

Danger — Math Ahead

The Fourier Transform and Variations

Every DSP based radio of which I am aware has some sort of spectrum display. These come in panadapter and waterfall flavors. These are all "real time" displays of energy versus frequency. The heart of the spectrum display in all of these cases is the Discrete Fourier Transform (DFT). In this column we will look at the DFT and a special case called the Fast Fourier Transform (FFT).

The general form of the continuous Fourier Transform is:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} dt \qquad \text{[Eq 1]}$$

This yields a complex frequency function (contains both sine and cosine terms from the $e^{-j2\pi ft}$ term) that is continuous in frequency from a frequency of $-\infty$ to ∞. The transform is derived from a continuous time function that is not periodic (in general) and extends from time $-\infty$ to ∞. The Fourier Series we use to describe a periodic signal such as a square wave or triangle wave is a special case of the general Fourier Transform. If you plot a periodic function in time, it is continuous from $-\infty$ to ∞, but the spectrum is composed of discrete frequency elements. There is also an inverse function that will transform a continuous frequency function into a corresponding continuous time function (Inverse Fourier Transform):

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft} df \qquad \text{[Eq 2]}$$

Notice that the only difference is the sign of the complex exponent. Likewise, there is an inverse of the Fourier Series that takes all of the component sine and cosine waves and adds the frequencies from $-\infty$ to ∞. The integral is replaced by a summation of the discrete frequencies from $-\infty$ to ∞. These are not terribly useful in the real world since we cannot really go forward or backward in time to plus or minus infinity.

The math so far has been ugly, with imaginary exponents, infinities, and integrals, but the math gets really ugly when you do the transformation from the continuous Fourier and Inverse Fourier forms to the Discrete Fourier and inverse Discrete Fourier forms. To see just how ugly, go to the Wikipedia page for Discrete Fourier Transform — **http://en.wikipedia.org/wiki/Discrete_Fourier_transform**. Quite a few folks did the really hard math over the past 250 years. Oddly, the first Fourier work was actually for the discrete forms rather than the continuous forms! Nyquist detailed part of the theory of practical DFT in 1928, and his work was expanded by Shannon and others later in the 20th century (all before practical DSP computers). When all the math is done, they showed that you can sample a continuous function at equally spaced intervals for a finite time, convert that sequence of samples into a limited discrete frequency set (the DFT), and then convert that limited frequency set back into a continuous time function. All of those conversions come with some specific constraints in order to make the conversions work in both directions to produce the output the same as the input.

The first constraint is the Nyquist limit, which requires that we sample at a rate *greater than* twice the highest frequency component. A discrete transform always acts on a limited number of samples. The discrete transform is much like a Fourier Series because the transform only contains a sequence of discrete frequencies. The Fourier Series is a consequence of the input signal being periodic. The limited number of discrete frequencies of the DFT forces the math to look like we have applied a rectangular window to a periodic signal. The second constraint is that a periodic function must be an exact sub-multiple of the sample frequency in order to do an exact transform or inverse transform.

The definition of the DFT is an operation that transforms a finite sequence of time samples into a sequence of the same number of complex frequency samples. In math form:

$$X[f] = \sum_{n=0}^{N-1} x[t] W_N^{nf} \qquad \text{[Eq 3]}$$

where:

$X[f]$ is the set of frequency samples, $x[t]$ is the set of time samples, N is the number of samples in each set, and W is the transform operator. Each of the elements (X, x, and W) can be represented as an array of values in a computer, so this is perfect for implementation in software.

Consequences of Signals in the Real World

Let's look at some examples to see what happens with respect to the constraints. We will look at a spectrum for a system sampled at 8 kHz and look at the signal every 12.5 ms. This means we update the spectrum 80 times per second. The 12.5 ms will result in 100 samples of the signal. The first example is for a 240 Hz signal. The update rate of 80 corresponds to each frequency sample mapping to exactly 80 Hz. The 80 Hz comes about because we can fit exactly one cycle of 80 Hz in 100 samples at 8000 samples per second. This is a discrete transform, so mathematically there is no energy at 5 Hz or 60 Hz for example, just energy at dc, 80 Hz, 160 Hz, … 4 kHz. Each of those exact frequencies is called a "bin." Figure 1 shows the continuous and sampled 240 Hz signal. Notice that it holds exactly three cycles of the 240 Hz waveform but it has 45° negative offset from a sine wave. Figure 2 shows the DFT of the signal. It is important to note that the DFT is a complex function that has both a cosine term (real) and a sine term (imaginary). In our example, the cosine term at 240 Hz is positive and the sine term is negative because of that –45° offset. If the samples had started at 1 for a cosine wave, the real part would have been one and the imaginary would have been zero. If it had started at zero for a sine wave, the real part would have been zero and the imaginary part would have been one.

Now, let's look at a cosine wave at 280 Hz. Figure 3A shows the 100 samples of our waveform that we use in our computer. Figure 3B shows what the math "sees." The math assumes that the 3½ cycles of the 280 Hz repeats with that discontinuity occurring every 12.5 ms from $-\infty$ to ∞. Figure 4 shows the DFT of this new signal. Now, we have energy in multiple bins because the real energy falls between adjacent bins for the fundamental sine wave. There is also additional energy in other bins because of the discontinuity at the end of the set of samples.

More Math

A DFT is really a conversion from a complex (real and imaginary) set of samples in time to a set of complex samples in frequency. Our first two examples implement what is called a "real" transform. These examples placed all of the time samples in the real part of the input set and loaded all of the imaginary samples with zero. Many SDR systems do all of their work with I and Q channels, so we have both a real set of samples (I) and an imaginary set of samples (Q). For spectral analysis, there is no real advantage for complex or real transforms. The Nyquist rate is required for a real time sequence in order

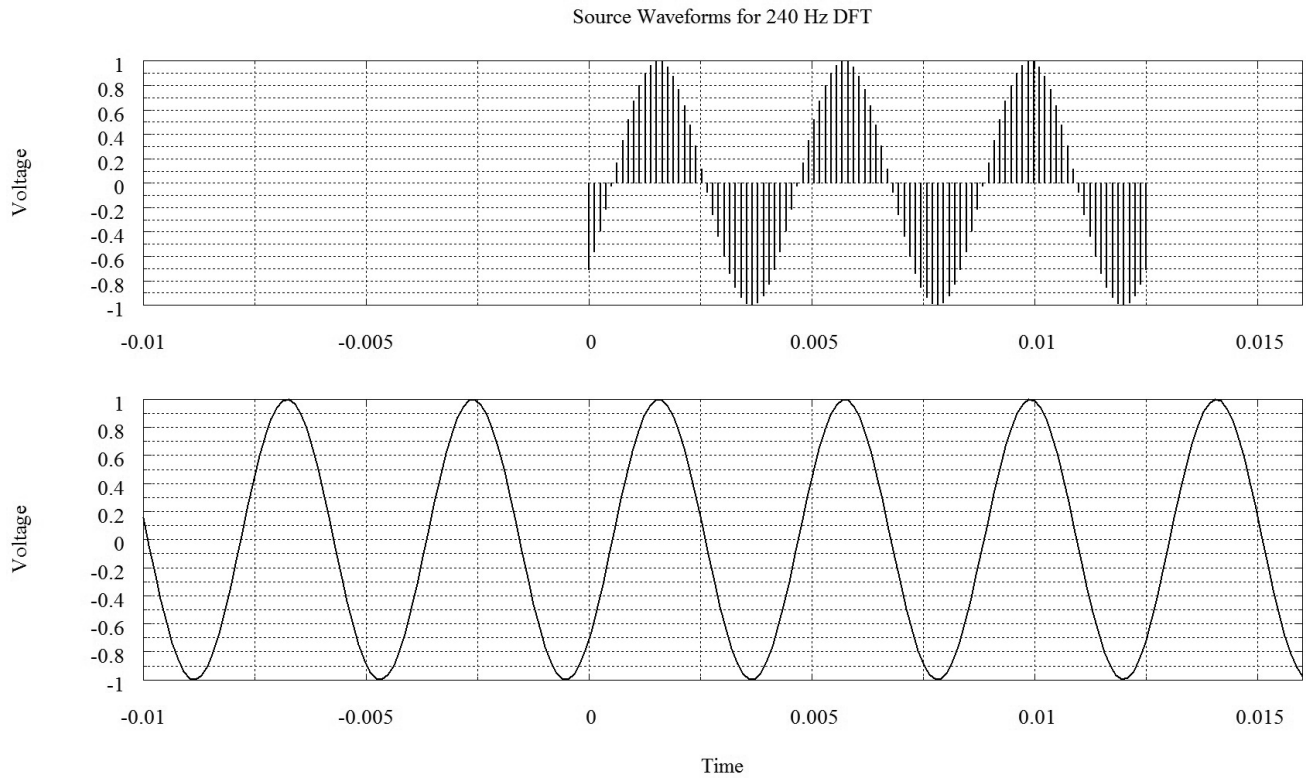

Figure 1 — Sampled and continuous versions of a 240 Hz sinusoid offset by 45°.

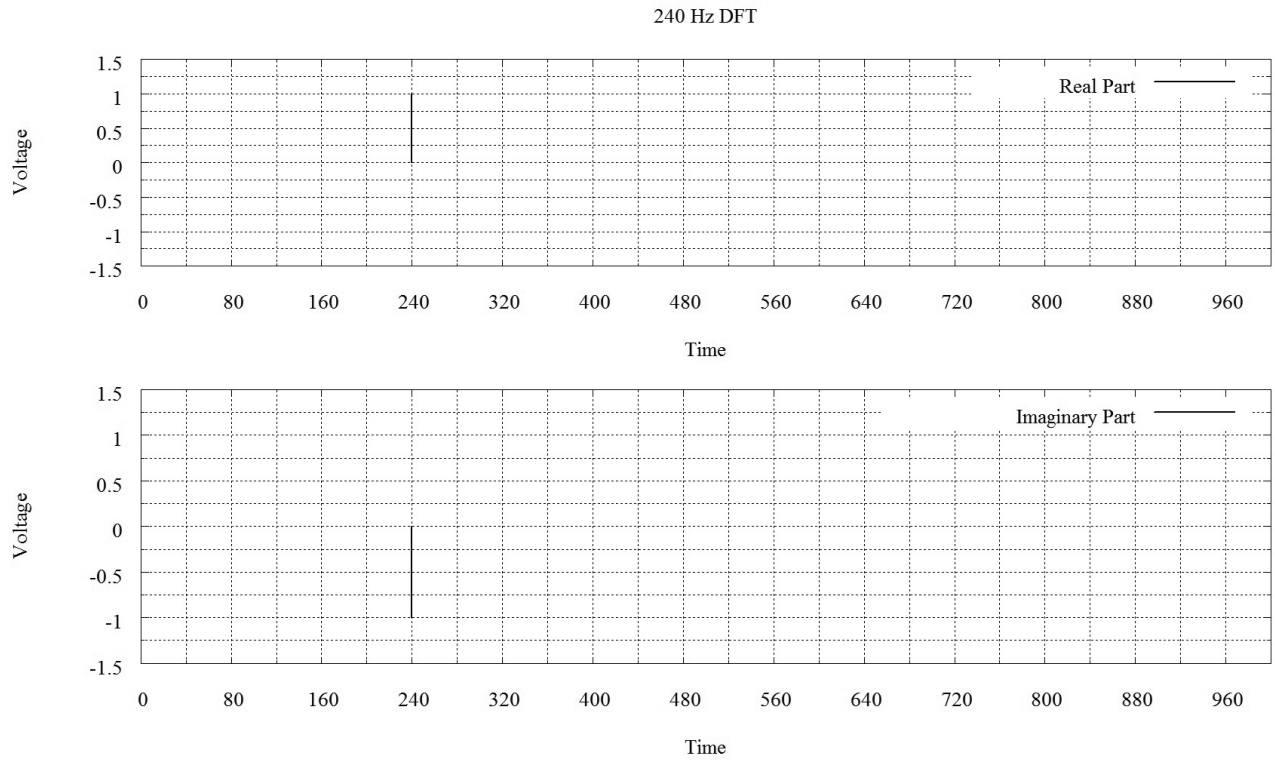

Figure 2 — The Discrete Fourier Transform (DFT) of the sinusoid in Figure 1.

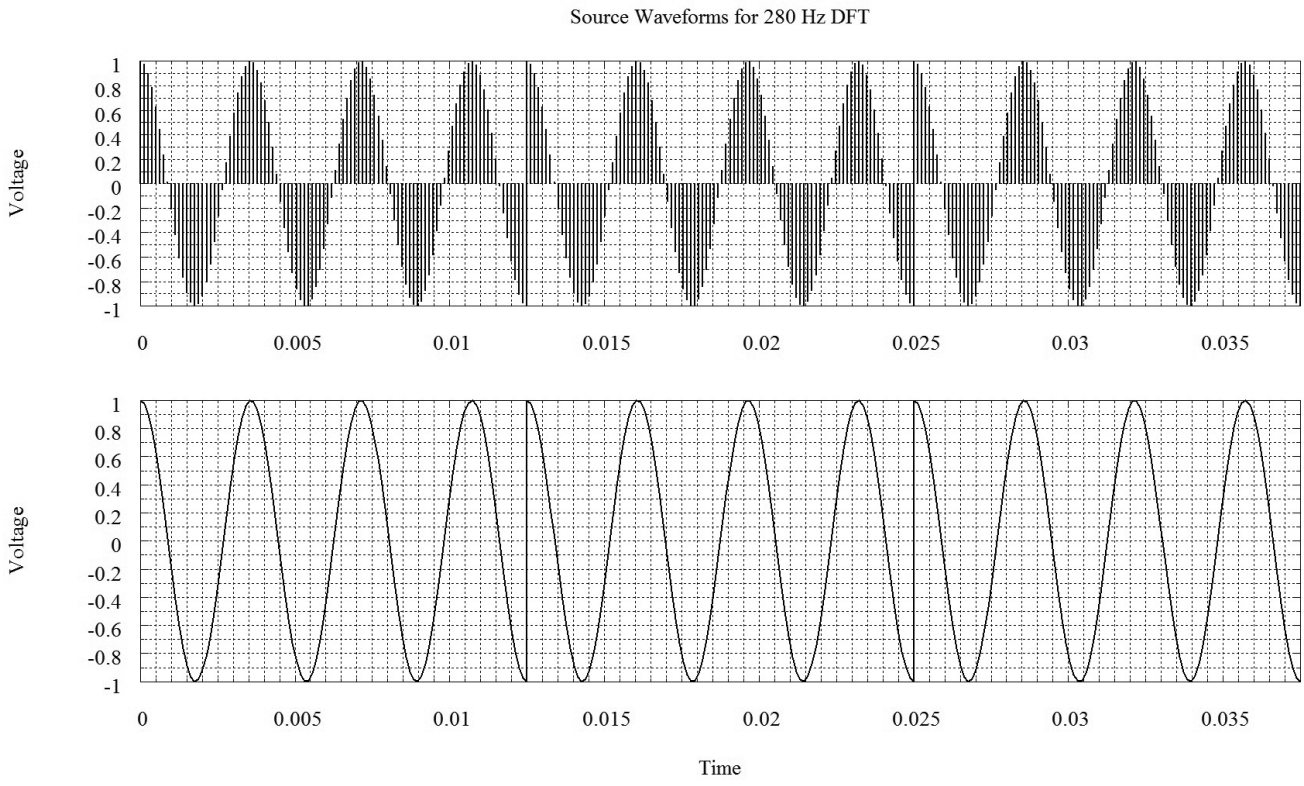

Figure 3 — Part A shows the sampled waveform for a 280 Hz sinusoid. Part B shows the effective signal that is transformed by the DFT.

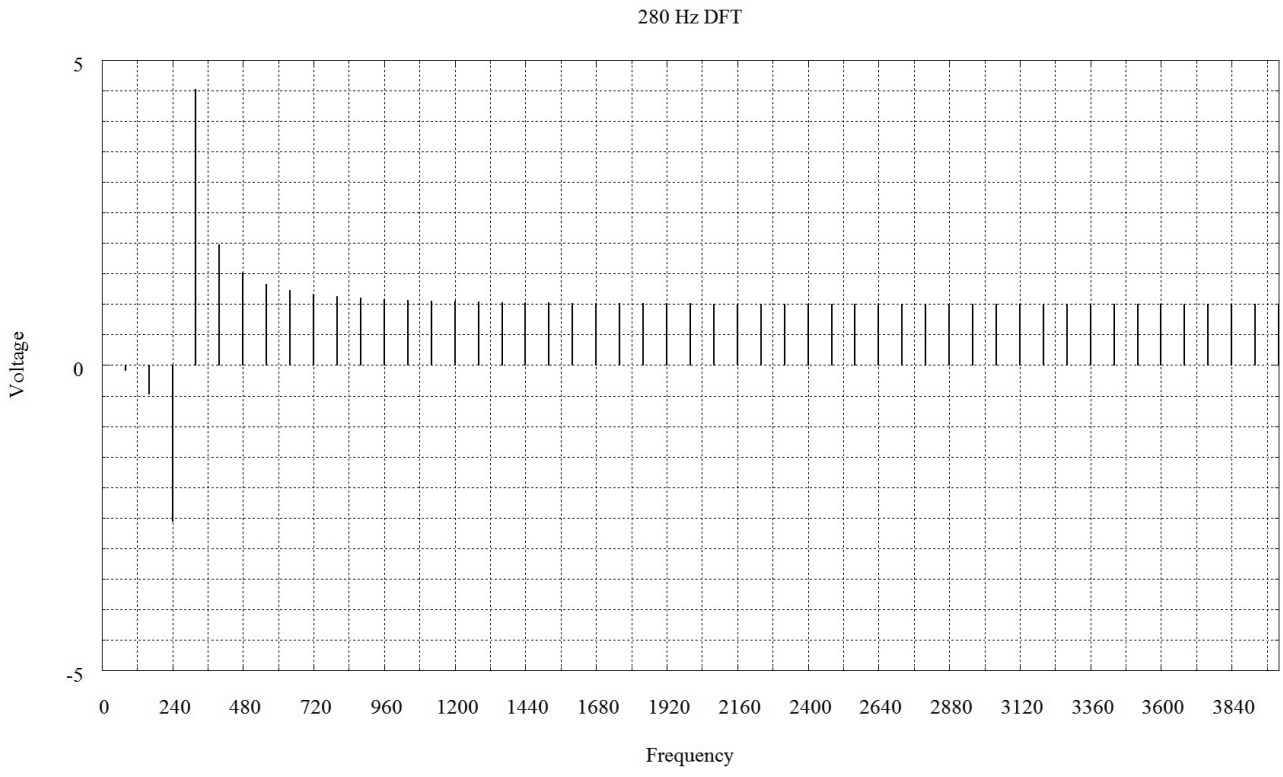

Figure 4 — The DFT of the signal of Figure 3.

to be able to determine both phase and frequency of the component signals, but only for positive frequencies. If we use an I-Q system we essentially double the sample rate and obtain exact phase and frequency for both positive and negative frequencies. I-Q systems double the sample rate, the computational load, and the covered output spectrum. The cost is twice the number of pieces of hardware for conversion.

The DFT is a general case function. J. W. Cooley and J.W. Tukey developed a computer algorithm in 1965 that takes advantage of the symmetry of the DFT if the number of samples is a power of two. This algorithm is called a Fast Fourier Transform (FFT) because it has significantly fewer arithmetic operations than a general DFT. The Cooley-Tukey FFT is just one of many possible fast transforms, but it is the one usually implemented. A DFT requires approximately N^2 operations, where an FFT requires approximately $N \log_2 N$ operations (where N is the number of samples). The software for this article contains an implementation of the FFT from *Numerical Recipes in C*.[1, 2]

The examples so far have looked at the real or complex input and complex output of the DFT. The complex output tells us both phase and frequency relative to a cosine wave that would begin at time sample zero. Phase is arbitrary since we almost never have a real reference between the signal we see and the place it was generated. For that reason, it is normal practice to convert the rectangular I-Q presentation to polar form with magnitude and phase (which we discard or simply choose not to calculate). This is called the power spectrum, since it is formed from squaring the two rectangular components. If we want the results in decibels, we save computation by doing just a log calculation rather than doing both a square root and log function. The log function allows us to multiply the result by 10 instead of 20 to get the magnitude in dB.

MATLAB and *Octave*

MATLAB and *Octave* contain functions for calculating the FFT and inverse FFT of data sets. Of course, *Octave* is a more useful tool for experimenters since it is free compared to hundreds of dollars for *MATLAB*. I find that the book *Computer Based Exercises for Signal Processing Using MATLAB 5* is a useful reference for understanding DSP concepts and using the computer tools for experiments.[3] It has a significant amount of engineering level math, though.

The FFT Algorithm

The heart of most FFT algorithms is a data flow pattern called a butterfly. Figure 5 shows a two element butterfly. The input contains two elements $x(0)$ and $x(1)$, and the output has two elements $y(0)$ and $y(1)$. The equations for the butterfly are:

[1]Notes appear on page 00.

$$y(0) = x(0) + x(1)W^k \quad \text{[Eq 4]}$$

$$y(1) = x(0) - x(1)W^k \quad \text{[Eq 5]}$$

where W^k is the appropriate element

$[\cos(2 \times k) + j\sin(2 \times k)]$.

Figure 6 shows the data flow diagram of the scrambled in/natural out (the output is ordered from lowest to highest element). The number in the circle at each stage indicates the value of k for W^k. It is interesting that the values of k are even for all stages except the final stage. A dashed line indicates an element that is added and a solid line indicates the element is multiplied by W^k. Notice that the first stage of calculation performs four 2×2 butterfly operations, the second is two 4×4 butterfly operations, and the final is one 8×8 butterfly. Each stage has fewer butterfly operations so it is called a decimation in time algorithm. The other thing to notice is that the input is grouped with the even time elements in the top group and the odd time elements in the bottom group.

This example requires just 24 complex multiplication operations. One other large advantage of the FFT is that the algorithm is a "multiply accumulate" type operation, where each horizontal position in Figure 6 has a multiply in place at each stage.

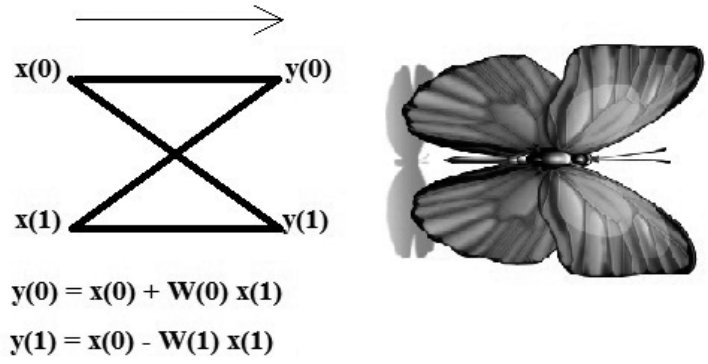

Figure 5 — A two element "butterfly" signal flow diagram. The crossing of the solid and dashed lines make the diagram look like stylized butterfly wings.

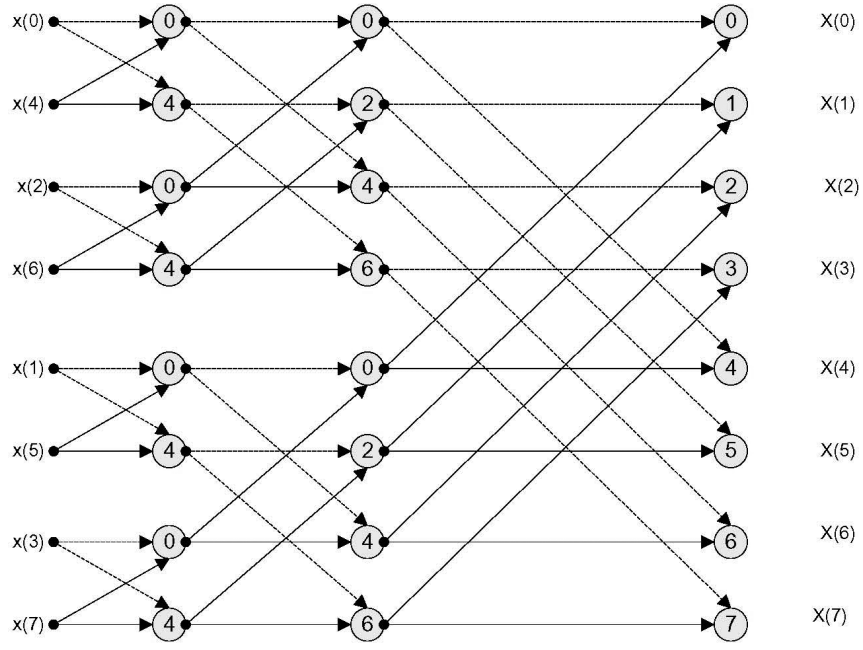

Figure 6 — A complete flow diagram for an 8 element Fast Fourier Transform (FFT) calculation. Each solid line represents a direct contribution of the element to the left. Each dashed line represents a contribution that is multiplied by the complex operator. Each number in a circle represents the k value for the W^k complex operator that is used for the multiplication.

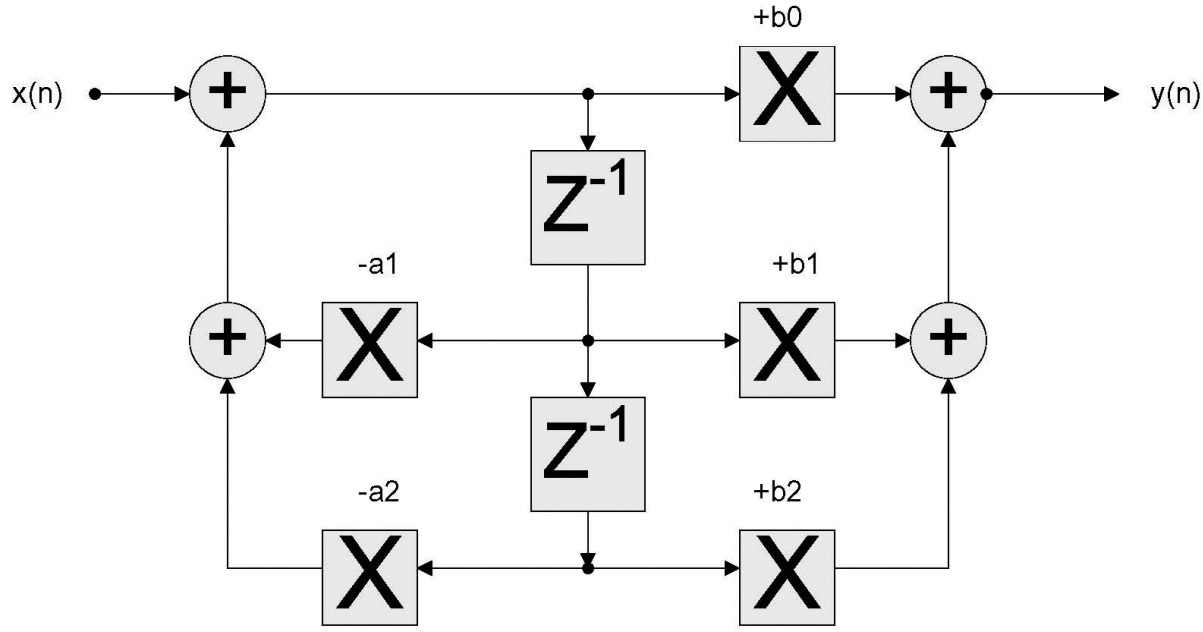

Figure 7 — A *z* space representation of an Infinite Impulse Response (IIR) filter. Each "b" value is a forward time contribution to the output, as in a Finite Impulse Response (FIR) filter, and each "a" value is a feedback element.

The result is that an *N* element FFT only requires an array of *N* registers to hold the entire data. The fixed sizes allow hardware implementation of an FFT with a very modest cost in resources. Our eight element FFT requires eight complex registers, logic to scramble the input as the data arrives, just eight *W* values (because the sine and cosine values will all use the same array from $-\pi$ to π), and eight multipliers. For this reason it is very easy for a processor such as the TMS320C5535 to include a fixed length (1024 in this case) hardware FFT block.

IIR Filters

We looked recently at Finite Impulse Response filters (FIR), which are non-recursive. They have the advantage that one can easily (more or less) calculate the filter tap values from a basic filter shape and apply a window function to minimize the Gibbs phenomenon. They also have the advantage of linear phase response and constant group delay. The biggest disadvantage to FIR filters is the large number of taps necessary for filters with maximum flatness and best transition band performance.

Infinite Impulse Response filters are a special case of a recursive digital filter. A recursive filter is built so that some of the output is fed back to the input. In an IIR filter, the output will continue to change forever even after the input signal is removed. It is possible to build recursive filters that do not have infinite impulse response but they are not very interesting.

I have been asked if there are programs to design IIR filters. Fortunately, the answer is yes. A group called *Octave*-forge has a large array of signal processing functions that work with *Octave* to perform an array of DSP design and analysis functions. We are interested in four functions: butter, cheby1, cheby2, and ellip. These let us design Butterworth, Chebyshev with pass band ripple, Chebyshev with stop band ripple, and elliptical filters. Each of these will return a filter design in a number of different formats. We will be interested in the z coefficient form. *Octave* tries to be as compatible with *MATLAB* as possible. I have found that it is frequently better to look at the *MATLAB* documentation on line rather than the documents from *Octave*-forge. The *MATLAB* documentation is professionally written and is easily understood. Frequently, the *Octave* documentation does not even exist.

The continuous time analysis uses the LaPlace transform *s* space to define a frequency response. In general, the response looks like:

$$H(s) = \frac{1 + a_0 s + a_1 s^2 + a_2 s^3 + \ldots}{1 + b_0 s + b_1 s^2 + b_2 s^3 + \ldots}$$

[Eq 6]

We factor the numerator polynomial to get the zeros of the response and factor the denominator polynomial to get the poles of the response. The response for a Butterworth filter is:

$$H(s) = \frac{1}{1 + s^{2N}}$$

[Eq 7]

where *N* is the order of the filter. The math allows us to convert from the polynomial form for an analog filter to an equivalent IIR digital filter. The equation follows the form:

$$H(z) = \frac{a_0 z^{-1} + a_1 z^{-2} + a_2 z^{-3} + \ldots}{b_0 z^{-1} + b_1 z^{-2} + b_2 z^{-3} + \ldots}$$

[Eq 8]

The filter design functions in *Octave* return two arrays containing the coefficients of "a" and "b" for the filter being designed. The filter design follows the method of Figure 7. The "a" coefficients appear on the left side of the system and apply to the feedback operation of the filter. The "b" coefficients appear on the right side of the system and apply to the forward operating part of the filter. Most of the filter design methods for IIR filters attempt to use polynomial representations from analog filters and adapt them to the *z* transform polynomials. Designing an analog filter using polynomial synthesis is a non-trivial process that involves a lot of math to achieve the desired features of a filter. The same is true of designing an IIR filter using those methods. That is one of the main reasons you won't see a lot of IIR filters in the literature; it is just a whole lot easier to do an impulse response and a DFT to generate an FIR filter from a desired frequency response.

Notes

[1]William Press, Saul Teukolsky, William Vetterling and Brian Flannery, *Numerical Recipes in C; The Art of Scientific Computing*, Cambridge University Press, 1992. This publication is available free on line at: **apps.nrbook.com/c/index.html**.

[2]The software for this column is available for download from the ARRL *QEX* files website. Go to **www.arrl.org/qexfiles** and look for the file **11x12_Mack_SDR.zip**.

[3]McClellan, Burrus, Oppenheim, Parks, Schafer, Schuessler , Computer Based Exercises for Signal Processing Using MATLAB 5, Prentice-Hall, 1997

QEX