Signals, Samples, and Stuff: A DSP Tutorial (Part 3)

Our tour continues with a foray into advanced DSP techniques.

By Doug Smith, KF6DX/7

Having learned about basic IF-DSP methods and their application in an actual transceiver, it's time to plunge into the truly magical stuff! In this third article in the series, we'll be looking at certain esoteric but extremely effective DSP techniques. By now, many of these concepts have found their way into production equipment, but they still are not generally well understood.

In this article, I will begin illustrating the underlying principles of current DSP noise reduction technology. Unfortunately, *QEX* space constraints require that Part 3 be broken across two issues. Therefore, Part 4 will complete the description of noise-reduction technology and the entire DSP series in the September/October issue.

[1]Notes appear on page 27.

PO Box 4074
Sedona, AZ 86340
e-mail: **dsmith@sedona.net**

DSP Noise Reduction Methods

Two noise reduction (NR) methods are prevalent in radio equipment today: the adaptive filtering method, and the Fourier transform method. We'll look at the theory behind each of these approaches, and discuss their implementation and performance. Then, a way of combining the two methods is considered. Along the way, I'll introduce a very fast way of calculating Fourier transforms—faster than the well-known "fast Fourier transform" algorithms.

Adaptive Filtering

In Part 1, we touched on the concept of an *adaptive interference canceler* and identified a design for an *adaptive notch filter* using the *least-mean-squares (LMS) algorithm*. These principles are explored in more detail here, as they apply to noise reduction systems. We'll find that it's possible to build an adaptive filter that accentuates the repetitive components of an input signal, and rejects the non-repetitive parts (noise). Further, we'll discover that the effectiveness of this technique depends on the characteristics of the input signals.

The nature of information-bearing signals is that they are in some way coherent; ie, they have some feature that distinguishes them from noise.

For example, voice signals have attributes relating to the pitch, syllabic content, and impulse response of a person's voice. CW signals are perhaps the simplest example because they constitute only the presence or absence of a single frequency.

Much research has been done about detection of a sinusoidal signal buried in noise.[1,2] Adaptive filtering methods are based on the exploitation of the statistical properties of the input signal, specifically the *autocorrelation*. Simply put, autocorrelation refers to how recent samples of a waveform resemble past samples. We'll build an *adaptive predictor*, which actually makes a reasonable guess at what the next sample will be based on past input samples. This leads directly to an adaptive noise-reduction system. Later, we'll discover how this technology is applied to compression of voice and other signals for digital transmission over the telephone network.

The Adaptive Interference Canceler[1]

Imagine that we have some input signal $x(k)$, and we want to filter it to enhance its sinusoidal content. The quantity $x(k)$ is just the discrete sample of continuous input signal x taken at time k. In the case of a CW signal, all that's required is a band-pass filter (BPF) centered at the desired frequency. We know the output will take the form of a sine wave, and that only its amplitude will change.

So we set up an FIR filter structure, and set the initial filter coefficients $h(k)$ to zero. Then we set up an error-measurement system to compare a sine wave $d(k)$ with the output of the filter, $y(k)$. See Fig 1. The reference input $d(k)$ is the same frequency we expect the CW input signal to be. The difference output $e(k)$ is known as the error signal. Then imagine we have some algorithm to adjust the filter coefficients so that the error $e(k)$ is reduced at each sample time. Think of the algorithm as some person who is "eye-balling" the error signal on an oscilloscope and has their hands on the filter controls. If they can minimize the error, then the filter will have converged to a BPF centered at the frequency of d(k).

We can already deduce that the speed and accuracy of convergence is going to depend on how well the person analyzes the error data. If it's difficult to tell that a sine wave is present, then adjusting the filter will be difficult, as well. Further, if the sampling rate is high enough, the person can't keep up; they can check the error only so often, or they can take long-term averages of the error.

Using the typical processes of the human mind, the person will soon discover that if they turn the controls the wrong way, the filter will diverge from the desired response; ie, the error increases. This information is used to reverse the direction of adjustment; the person will then turn the controls the other way. They will soon discover they are on a *performance surface* that has an "uphill" and a "downhill," and they know they want to go only downhill.

So they thrash about with the controls, sometimes making mistakes and heading the wrong way, but ultimately making headway overall down the hill. At some point, the error gets very small, and they know they're near the "bottom of the bowl." Once at the bottom, it's uphill no matter which way they go! So, they continue flailing about, but always staying near the bottom. They have successfully achieved the goal: Minimization of the total error $e(k)$. This story is analogous to aligning an analog BPF with an adjustment tool.

After doing this several times, the person finds that certain rules help them speed up the process. First, there is a relationship between the total error and the amount they need to tweak the controls. If the total error is large, then a large amount of tweaking must be done; if small, then it's better to make small adjustments to stay near the bottom. Second, there is a correlation between the error signal $e(k)$, the input samples $x(k)$ and the filter coefficient set $h(k)$ they need to adjust.

Derivation of algorithms that provide for the quickest descent down the hill is a very long and tedious exercise in linear algebra. Let's just say the person goes to school, becomes an expert in matrix mathematics and discovers that one of the fastest ways down the hill is to make adjustments at sample time k according to:

$$h_{k+1} = h_k + 2\mu e_k x_k \qquad \text{(Eq 1)}$$

This is the LMS algorithm. It was developed by Widrow and Hoff[3] in the late 1950s.

Properties of the Adaptive Interference Canceler

Now we have our adaptive interference canceler. See Fig 2. Note that both the desired output $y(k)$ and the undesired $e(k)$ are available. This is nice in case we want to exchange roles, to accept only the incoherent input signals and reject coherent ones. Such is the case for an adaptive notch filter, treated further below. This doesn't change the algorithm, however. Quantities of interest in this system are the adjustment error near the bottom of the performance surface, and the speed of adaptation.

One of the first things we discover about the LMS algorithm is that the speed of adaptation and the total misadjustment are both directly proportional to μ. We select its value, which ranges from 0 to 1, to set the desired properties. Note that there is a trade-off between speed and misadjustment. Large values of μ result in fast convergence, but large adjustment errors.

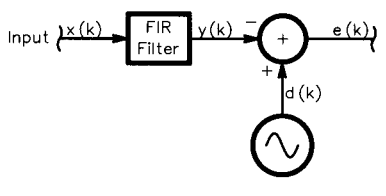


Fig 1—An adaptive modeling system.

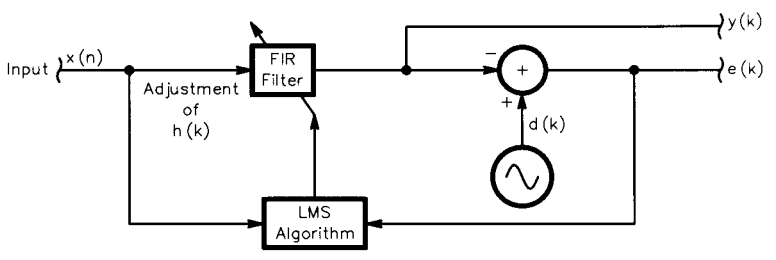


Fig 2—An adaptive interference canceler.

Also note that the number of filter coefficients $h(k)$ has a bearing on both of these performance parameters. It turns out that the total amount of misadjustment is directly proportional to the number of filter coefficients, and this places a limitation on the complexity of the filter. In addition, as the filter grows in length, the total delay through the filter grows proportionately. The delay through an FIR filter of length L is equal to:

$$T_{FIR} = \frac{LT_s}{2} \qquad \text{(Eq 2)}$$

where T_s is the sample time, and this may become unacceptable under certain conditions.

Attempts may be made to adjust the factor μ on an adaptive basis by using a value which changes in proportion to the total error $e(k)$. A large value is selected initially to obtain rapid convergence, then it's decreased to minimize the total misadjustment as we approach the steady-state solution. This works fine as long as the characteristics of the input signal don't change rapidly.

The Adaptive Interference Canceler Without an External Reference—The Adaptive Predictor

In the above example of a CW signal, we knew what to expect at the output: A sine wave of known frequency. What happens when we don't know much about the nature of the desired signal, except that it's coherent in the time domain? A number of circumstances arise wherein the only fact known about the desired signal is that it is distinguishable from noise in some way; ie, that it's periodic. It might seem at first that adaptive processing can't be applied. But if a delay, z^{-n} is inserted in the *primary input* $x(k)$ to create the reference input $d(k)$, periodic signals may be detected and, therefore, enhanced. See Fig 3. This delay is akin to an autocorrelation offset, and it represents the time differential used to compare past input samples with the present ones. The amount of delay must be chosen so that the desired components in the input signal autocorrelate, and the undesired components do not.

This system is an adaptive predictor. The predictable components are enhanced, while the unpredictable parts are removed. Fig 4 shows the result of an actual experiment using a sine wave buried in noise as the input. The input BW is 3 kHz, and the input SNR = 0 dB. For any given value of μ,

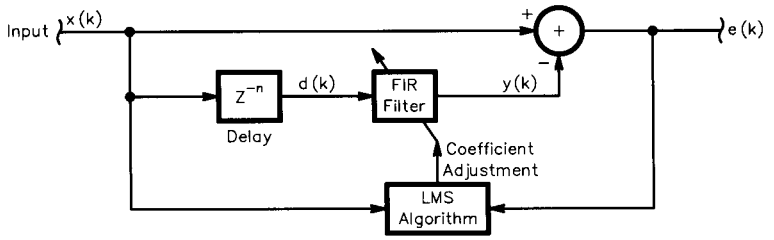


Fig 3—An adaptive predictor.

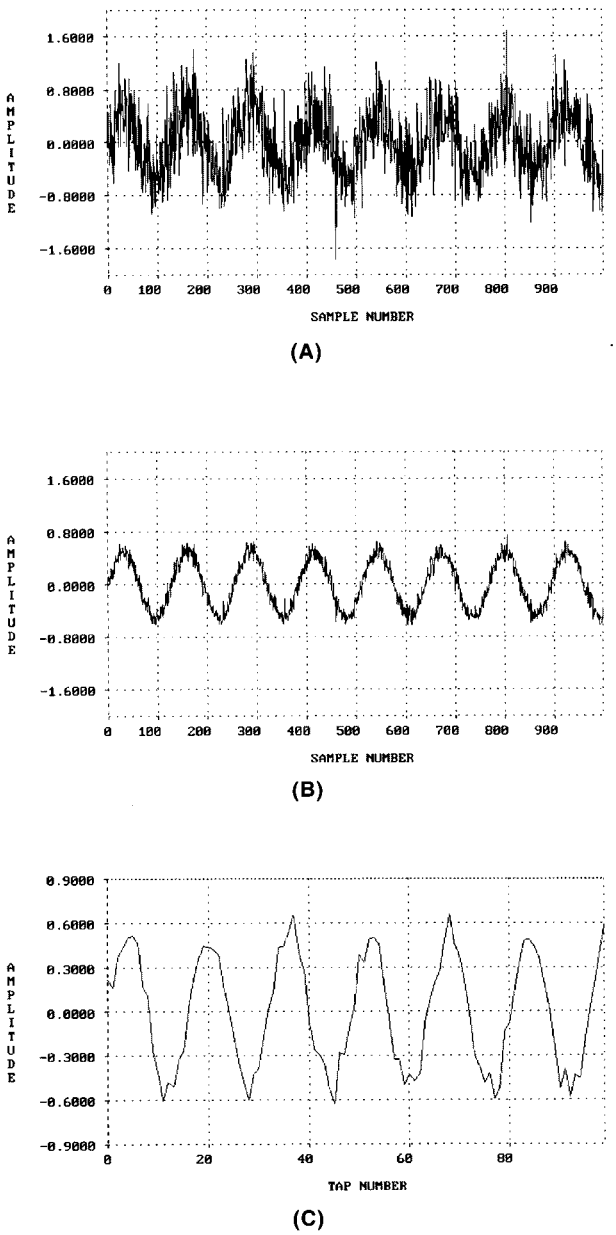


Fig 4—(A) A noisy sine wave. (B) Filtered output. (C) The adaptive filter's impulse response.

the filter converged on the optimal solution fastest when the delay was set roughly equal to the filter delay as defined in Eq 2. Note that the filter's impulse response is also a sinusoid. We find that the filter's BW[1] is:

$$BW = \frac{2\mu A^2}{T_s} \qquad \text{(Eq 3)}$$

where A is a long-term average of the amplitude of the input $x(k)$. So the speed of adaptation and the NR effectiveness are proportional to μ *and* to the amplitude of the input signal. In the example, $\mu = 0.005$, $A = 1$, and $T_s = (15 \times 10^3)^{-1}$. The SNR improvement is therefore:

$$\begin{aligned}\Delta SNR &= 10\log\left(\frac{3\text{ kHz}}{BW}\right)\\ &= 10\log\left(\frac{(3\times10^3)T_s}{2\mu A^2}\right)\\ &\approx 13\ dB\end{aligned} \qquad \text{(Eq 4)}$$

Alternatively, the unpredictable components $e(k)$ may be taken as the output. This forms an adaptive notch filter. Say we have a desired voice signal corrupted by the presence of a single interfering tone or carrier. This is a very common situation on today's HF ham bands! We can set the autocorrelation delay and variable μ so that the steady tone is predictable, and the rapidly changing voice characteristics are not. The filter will converge to the solution that removes the tone and leaves the voice signal virtually unscathed.

The BW of the notch is the same as in Eq 3, but its depth is determined only by numerical-accuracy effects in the DSP system. When adaptive filters with many taps are used, multiple tones may be notched. See Fig 5. In this experiment, several nonharmonically related tones, plus noise, are used as the input. The filter's response converges to notch them all, leaving only noise at the output. In this case, the undesired components are large compared to the desired components. When the undesired signal level is low, there might not be enough thrashing about on the performance surface for us to find our way down the hill. Adding artificial noise to satisfy this condition is tempting, but it turns out that we can alter the algorithm to improve the situation without actually adding noise. Such additional terms in the algorithm are referred to as *leakage* terms.

"Leaky" LMS Algorithms

The unique feature of *leaky* LMS algorithms is a continual "nudging" of the filter coefficients toward zero. The effect of the leakage term is striking, especially when applied to NR of voice signals. The SNR improvement increases because the filter coefficients tend toward a lower throughput gain in the absence of desired input components. More significantly, the leakage helps the filter adapt under low SNR conditions—the very conditions when NR is needed most.

One way to implement leakage is to add a small constant of the appropriate sign to each coefficient at every sample time. This constant is positive for negative coefficients, and negative for positive coefficients:

$$h_{k+1} = h_k + 2\mu e_k x_k - \lambda\left[sign(h_k)\right] \qquad \text{(Eq 5)}$$

The value of λ can be altered to vary the amount of leakage. Large values prevent the filter from converging on *any* input components, and things get very quiet indeed! Small values are useful in extending the noise floor of the system. In the absence of coherent input signals, the coefficients linearly move toward zero; during convergent conditions, the total misadjustment is increased to at least λ, but this isn't usually serious enough to affect received signal quality.

An alternate way to implement leakage is to scale the coefficients at each sample time by some factor, γ, thus also nudging them toward zero:

$$h_{k+1} = \gamma h_k + 2\mu e_k x_k \qquad \text{(Eq 6)}$$

For values of γ just less than one, leakage is small; values near zero represent large leakage and again prevent the filter from converging. This realization of the leaky LMS exhibits a logarithmic decay of coefficients toward zero, which may be advantageous under certain circumstances. It can be shown[1] that the leaky LMS is equivalent to adding normalized noise power to the input $x(k)$ equal to:

$$\sigma^2 = \frac{1-\gamma}{2\mu} \qquad \text{(Eq 7)}$$

Note that the leaky LMS algorithm must adapt to "survive." Were the factor μ suddenly set to zero, the coefficients would die away toward zero and never recover. Therefore, it's unwise to use these algorithms with adaptive

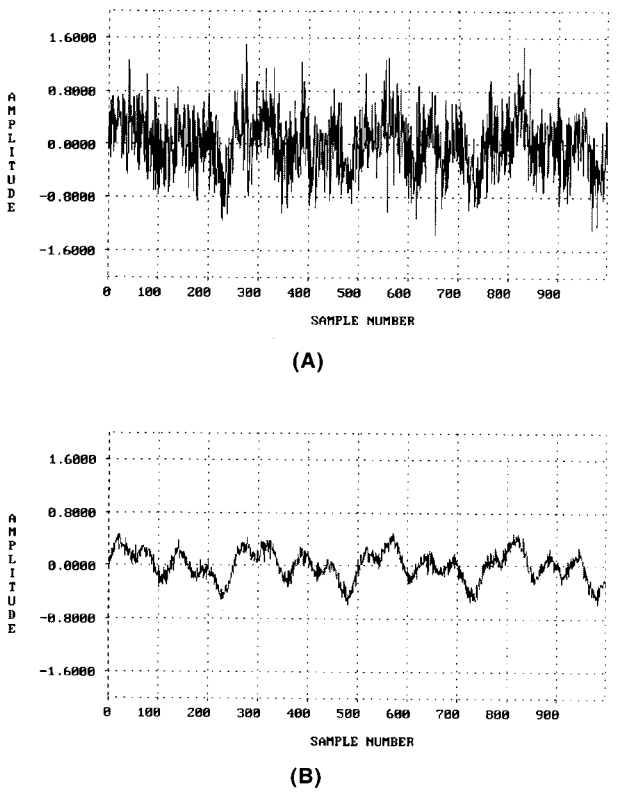


Fig 5— (A) Multiple noisy sine waves. (B) Filtered output.

values of μ. Although values for γ and μ greater than one have been tried, the inventors refer to these procedures as "the dangerous LMS algorithm." Enough said.

Next, I'll describe an even more powerful NR tool, the Fourier transform. Although it's generally more processing-intensive than methods described thus far, the results can be impressive. First, however, let's look at the great man whose work is so inexorably bound to modern electronic communications.

Joseph Fourier—A Legacy of Genius

Joseph Fourier (1768 to 1830) was born the son of a tailor.[4] Educated by monks in a military school, Fourier apparently thought only the army or the church could provide him a career. Despite a recommendation from the famous mathematician Legendre, the army rejected his application to the artillery; he opted instead for a religious life. Mathematics had been his primary scientific study since an early age, and at the onset of the French Revolution, he was appointed by the monks to the principal chair of mathematics at Auxerre. There, Fourier met Napoleon, who often attended lectures at the major universities.

When Napoleon organized an expedition to Egypt in 1798, Fourier was asked to join. After three years there in the capacity of an engineer, he collaborated with Napoleon to produce the *Description of Egypt*, which established his literary prowess and eventually won him election to the French Academy.

On his return to France, he was granted a comfortable governmental position, which gave him free time to pursue his mathematical interests. In 1807, he submitted to the Academy his first paper describing the motion of heat in solid bodies. In 1812, he was awarded the prize for scientific accomplishment for his complete explanation of the effect—the judges were Laplace, Legendre and Lagrange! His place in history was thereby confirmed.

Fourier was elected a member of the Academy of Sciences in 1817, and to the French Academy in 1822. He died while still in government service in 1830. He could scarcely have imagined what impact his work has had in the field of electronics, especially DSP.

The Fourier Transform and Its Inverse

The relationship Fourier discovered between the application of heat to a solid body and its propagation has direct analogy to the behavior of electrical signals as they pass through filters or other networks. The laws he found represent the connection between the time- and frequency-domain descriptions of signals. They form the basis for DSP spectral analysis, and therefore they are useful in digging signals out of noise, as we'll see below.

The *Fourier Transform*[5] of some continuous signal x_t is expressed as:

$$X_\omega = \int_{-\infty}^{\infty} e^{-j\omega t} x_t dt \qquad \text{(Eq 8)}$$

It's obtained by making the following assumptions: x_t is a continuous, periodic function of time; and any continuous, periodic function of time can be expressed as the superposition (integral) of sines and cosines. Recall the Euler identity:

$$e^{j\omega t} = \cos\omega t + j\sin\omega t \qquad \text{(Eq 9)}$$

and observe that when the real and imaginary parts are separated, Eq 8 produces coefficients a_ω and b_ω:

$$a_\omega = \int_{-\infty}^{\infty} x_t \cos(\omega t)\, dt \qquad \text{(Eq 10)}$$

$$b_\omega = -\int_{-\infty}^{\infty} x_t \sin(\omega t)\, dt \qquad \text{(Eq 11)}$$

X_ω is just the sum of these terms as a complex pair:

$$X_\omega = a_\omega + jb_\omega \qquad \text{(Eq 12)}$$

The coefficients yield the amplitude and phase of the signal x_t at the frequency ω:

$$A_\omega = \left(a_\omega^2 + b_\omega^2\right)^{\frac{1}{2}} \qquad \text{(Eq 13)}$$

$$\phi_\omega = \tan^{-1}\left(\frac{b_\omega}{a_\omega}\right) \qquad \text{(Eq 14)}$$

Working in reverse, we can reconstruct x_t by integrating X_ω for all values of ω:

$$x_t = \frac{1}{2\pi}\int_{-\pi}^{\pi} X_\omega e^{j\omega t} d\omega \qquad \text{(Eq 15)}$$

This is known as the *Inverse Fourier Transform* of X_ω. The limits of integration in this case are finite because $e^{j\omega t}$ and X_ω are themselves continuous, periodic functions of ω, repeating with period 2π This follows from Eq 8, since:

$$e^{-j(\omega+2\pi)t} = e^{-j\omega t} \qquad \text{(Eq 16)}$$

We had to use infinite limits in Eq 8, because we made no assumptions about the length of period of x_t. Ie, we didn't know how far in time to look until x_t began repeating itself. If we had some knowledge about the periodicity of x_t, then we could restrict the integration limits without major deleterious effects.

Using infinite integration limits, components in x_t not at frequency ω do not affect the result. In the DSP world, however, we deal with discrete samples of the amplitude of x_t, which we'll refer to as $x(n)$. The discrete Fourier Transform (DFT) gives us an expression equivalent to Eq 8 for sampled signals. We'll see that in this form, issues of frequency resolution arise because infinite evaluation intervals aren't practical.

The Discrete Fourier Transform (DFT)

The discrete-sample equivalent of Eq 8 is expressed as:

$$X(\omega) = \sum_{n=-\infty}^{\infty} e^{-j\omega n} x(n) \qquad \text{(Eq 17)}$$

where n is the sample number. As alluded to above, we can't compute this sum over an infinite number of samples. In discrete spectral analysis, many sums must be obtained; only a short time is available for these computations before the next iteration must be performed.

So let's say that $x(n)$ has a period less than some number of samples N. We limit our summation to that number of samples, and as a consequence, the results are available only in integer multiples of the fundamental frequency $2\pi / N$:

$$X(k) = \sum_{n=0}^{N-1} e^{\frac{-2\pi jkn}{N}} x(n) \qquad \text{(Eq 18)}$$

This is the DFT for a frequency proportional to k. It turns out there are just N frequencies available that are integer multiples of $2\pi / N$. This is because $e^{-2\pi jkn/N}$ is periodic with period N. It's a fact of DSP life that samples taken at discrete times transform to samples at discrete frequencies using the DFT.

DFTs are normally computed for N evenly spaced values of k. To relate the normalized analysis frequency k / N to the sampling frequency $1 / T_s$, we can write:

$$f_k = \frac{k}{NT_s}, \text{ for } k < \frac{N}{2} \qquad \text{(Eq 19)}$$

This is the actual frequency, in hertz, of the DFT represented by $X(k)$.

The idea is that if we can analyze our input signal at many frequencies, and exclude those results or *bins* not meeting certain criteria, we can eliminate undesired signals. Filters can be implemented by rejecting signals outside the frequencies of interest. And noise reduction can be accomplished by eliminating bins for which a preset amplitude threshold is not met.

The DFT In Noise-Reduction Systems

The efficacy of the DFT is that it evaluates the amplitude and phase of some particular frequency component to the exclusion of others. As far as we can reduce the *resolution bandwidth* (BW) of our frequency-specific measurements using the DFT, we can eliminate noise. Ie, the finer the frequency resolution, the less noise we are including in each bin, so any coherent signal in the measurement BW has an improved signal-to-noise ratio (SNR). Finer resolution is obtained by increasing the number of bins, N.

Shown in Fig 6 is the result of a DFT analysis of a low-level 1 kHz sine wave buried in noise. In the 3 kHz BW of interest, the noise power is just equal to the signal power at the input. The SNR is therefore 0 dB. The sampling frequency is 15 kHz, and $N = 1500$ for this DFT. Because the resolution BW of our DFT is:

$$f_{res\ BW} = \frac{1}{NT_s} = 10\ \text{Hz} \qquad \text{(Eq 20)}$$

the SNR improves at the bin centered on 1 kHz by the factor:

$$SNR_{10\ Hz} = 10\log\left(\frac{3\ \text{kHz}}{10\ \text{Hz}}\right) \approx 24.8\ \text{dB} \qquad \text{(Eq 21)}$$

The sine wave stands out clearly above the noise. We can modify the results of our spectral analysis by setting a threshold, below which we set the DFT results to zero. See Fig 7. We then convert this modified frequency-domain picture back to time-domain samples using the *inverse DFT* (DFT^{-1}):

$$x'(n) = \frac{1}{N}\sum_{k=0}^{N-1} e^{\frac{2\pi j kn}{N}} X'(k) \qquad \text{(Eq 22)}$$

Since the bins containing only noise have been zeroed in the modified transform samples X'(k), the output signal x'(n) now has an improved SNR of 24.8 dB! The remaining noise is centered in a 10 Hz BW around the 1 kHz tone. See Fig 8. Note that we have to compute 1500 DFT bins at each sample time to get this result. Any bins that are zeroed obviously make the conversion using the DFT^{-1} faster, since they need not be computed.

Setting the Threshold

Setting the cutoff threshold is critical to this noise-reduction method, because we may inadvertently exclude low-energy components, which are actually part of the desired signal. The simplest solution has the operator set it manually. One just "mows the grass" to whatever depth produces a pleasing result.

An automatic system seems possible, but we would need to make some assumptions about the nature of the desired signal or signals. The requirements for a voice signal, for example, might be quite different from those for a RTTY or CW signal. Selective fading and multipath effects ultimately limit the usefulness of any automatic system.

Since we control which bins get excluded, based not only on their amplitudes, but also their frequencies, the DFT gives us a way to include custom band-pass filter banks, similar to a graphic equalizer.

Computation of the DFT and Limitations on Accuracy

Components in $x(n)$ at frequencies other than k / N skew the result and significantly limit resolution BW. The limited summation range broadens the spectral line width, even for a single

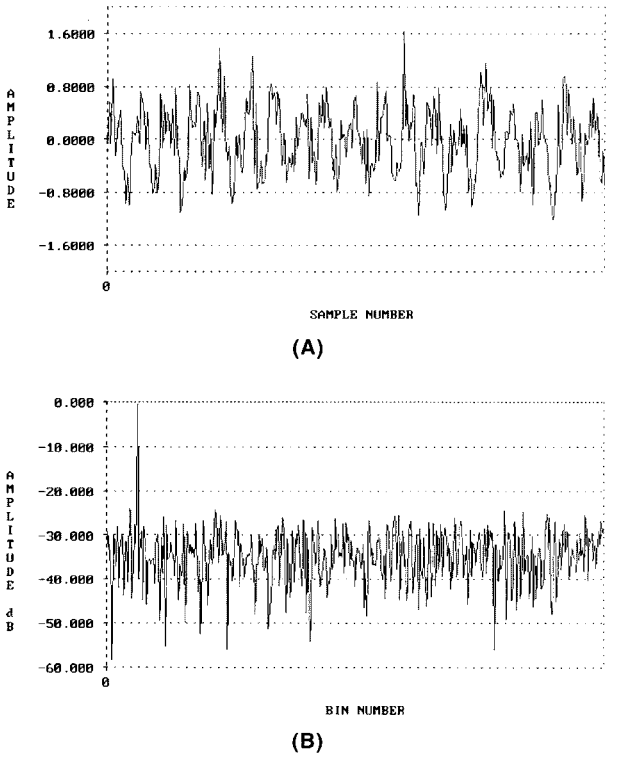


Fig 6—(A) A noisy sine wave. (B) Its DFT amplitude-*versus*-frequency.

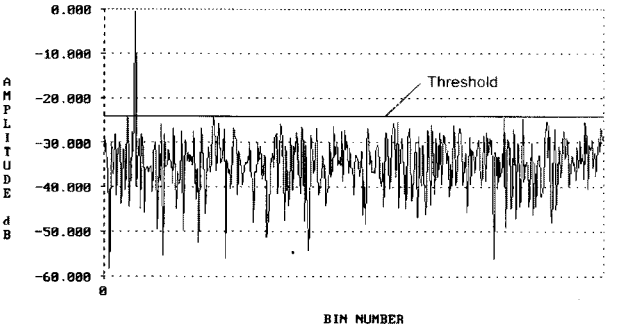


Fig 7—Applying a noise-reduction threshold.

input frequency as shown in Fig 9. The *side lobes* of the spectral broadening are evident in this diagram. Many advanced methods have been introduced to deal with this situation. They are based on either reduction of the computation time to obtain more bins, or modification of the data, or both.

Of help in reducing the broadening effect is the technique of *windowing* the input data. The data block is multiplied by a *window function*, then used as input to the DFT algorithm. Examples of window functions and their DFTs are shown in Fig 10. The rectangular window is equivalent to not using a window at all, since each input sample is multiplied by unity. The other window functions achieve various amounts of side-lobe reduction.

Also impacting the accuracy of our results are the familiar truncation and rounding effects previously discussed.[6] Their influence on the DFT is treated further below.

Other advanced spectral-estimation techniques have been developed over the years,[7, 8] some of which produce excellent results. Most use the DFT in some form. Because computation time is critical in embedded systems and Fourier analysis is so important to so many fields, much time and effort has been expended to find efficient DFT-computing algorithms.

In the years before computers, reduction of computational burden was extremely desirable, because computation was done by hand! Many excellent mathematicians, including Runge,[9] applied their wits to the problem of calculating DFTs more rapidly than the direct form of Eq 18. They recognized that the direct form required N complex multiplications and additions per bin, and N bins were to be calculated, for a total computational burden proportional to N^2. The first breakthrough was achieved when they realized that the complex exponential $e^{-2\pi jkn/N}$ is periodic with period N, so a reduction in computations was possible through the symmetry property:

$$e^{\frac{-2\pi jk(N-n)}{N}} = e^{\frac{2\pi jkn}{N}} \qquad \text{(Eq 23)}$$

This led to the construction of algorithms that effectively broke any N DFT computations of length, N, down into N computations of length $log_2(N)$. Thus, the computational burden was reduced to $N \log_2(N)$. Because even this much work wasn't practical by hand, the usefulness of the discovery was largely overlooked until Cooley and Tukey[10] picked up the gauntlet in the 1960s.

The Fast Fourier Transform (FFT)

Let's look at how FFT algorithms are derived from the repetitive nature of the complex exponential (Eq 23) above, and how they're implemented using *in-place* calculations. Since we're going to be dragging around a lot of complex exponentials, we'll adopt the simplified notation of Oppenheim and Schafer[5] where:

$$e^{\frac{-2\pi jkn}{N}} = W_N^{kn} \qquad \text{(Eq 24)}$$

To exploit the symmetry referred to, we have to break the DFT computation of length, N, down into successively smaller DFT computations. This is done by *decomposing* either the input or the output sequence. Algorithms wherein the input sequence $x(n)$ is decomposed into successively smaller subsequences are called *decimation-in-time* algorithms.

Let's begin by assuming that N is an integral power of two. That is, for a whole number p:

$$N = 2^p \qquad \text{(Eq 25)}$$

Next, we break the input sequence into two subsequences, one consisting of the even-numbered samples, and the other of the odd-numbered samples. For some index r, $n = 2r$ for n even, and $n = 2r + 1$ for n odd. Now, with Eq 18 in mind, we can write:

$$\begin{aligned} X(k) &= \sum_{r=0}^{\frac{N}{2}-1} W_N^{2rk} x(2r) + \sum_{r=0}^{\frac{N}{2}-1} W_N^{(2r+1)k} x(2r+1) \\ &= \sum_{r=0}^{\frac{N}{2}-1} \left(W_N^2\right)^{rk} x(2r) + W_N^k \sum_{r=0}^{\frac{N}{2}-1} \left(W_N^2\right)^{rk} x(2r+1) \end{aligned} \qquad \text{(Eq 26)}$$

To further simplify, we can use:

$$\begin{aligned} W_N^2 &= e^{\frac{(2)(-2\pi j)}{N}} \\ &= e^{\frac{-2\pi j}{\frac{N}{2}}} \\ &= W_{\frac{N}{2}} \end{aligned} \qquad \text{(Eq 27)}$$

and so now:

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} W_{\frac{N}{2}}^{rk} x(2r) + W_N^k \sum_{r=0}^{\frac{N}{2}-1} W_{\frac{N}{2}}^{rk} x(2r+1) \qquad \text{(Eq 28)}$$

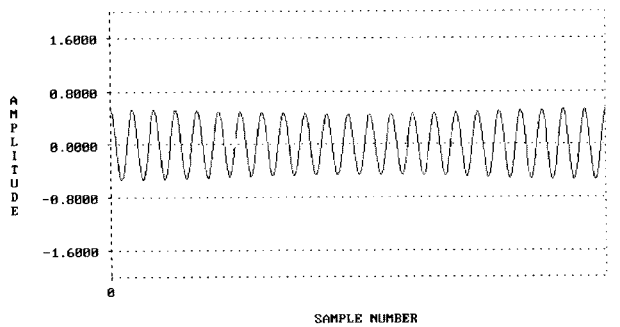


Fig 8—The reconstructed sine wave.

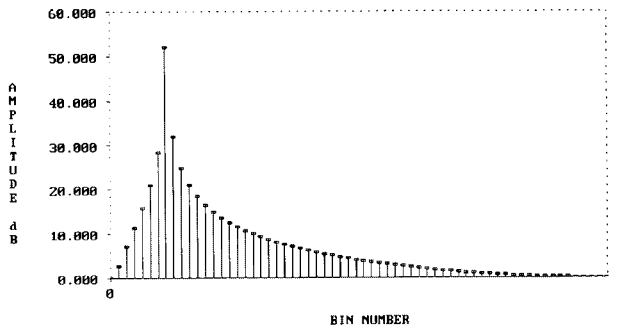
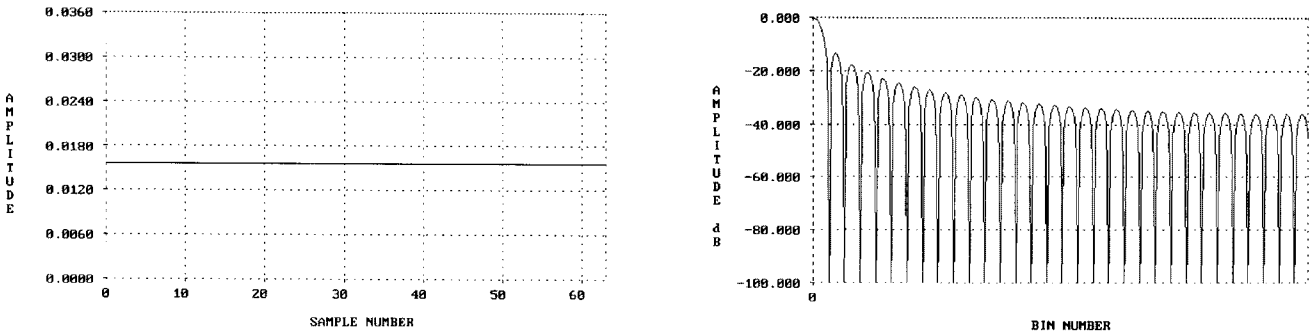
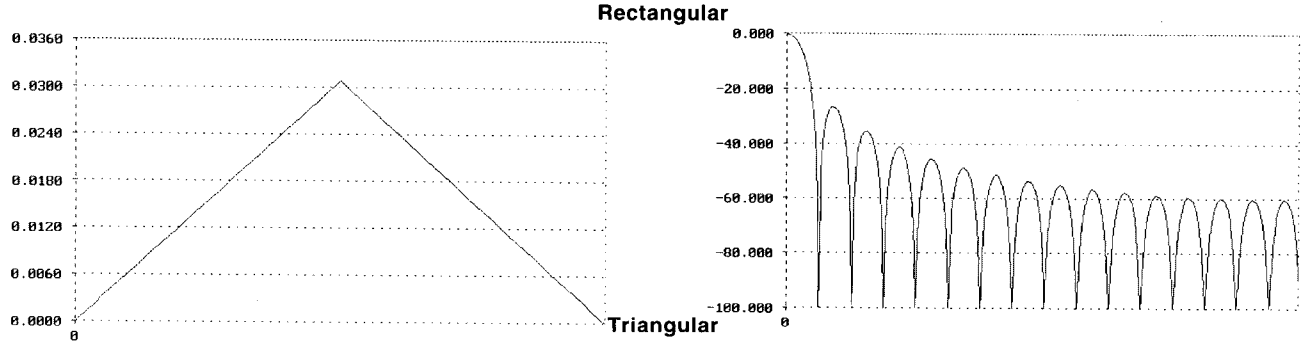


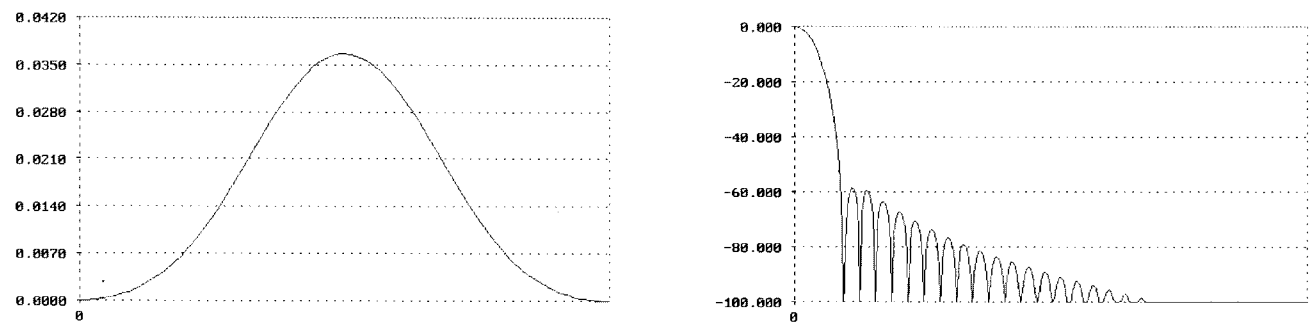
Fig 9—DFT of a noise-free sine wave, showing side lobes.



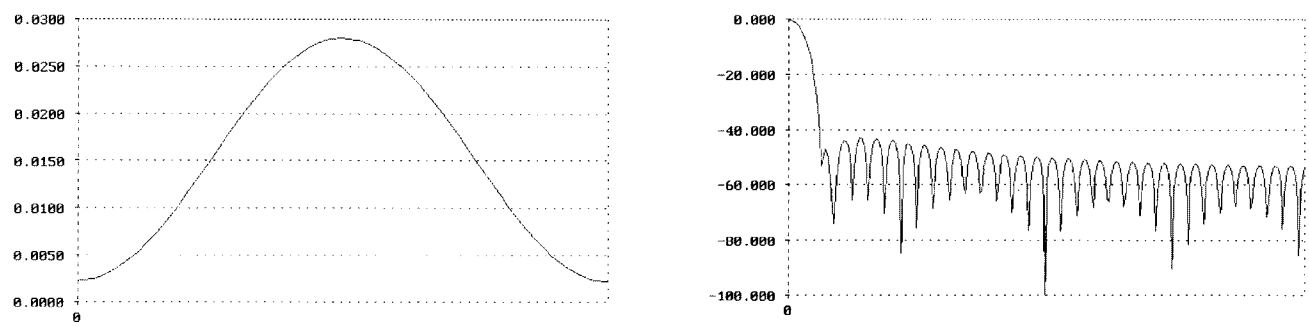
Rectangular



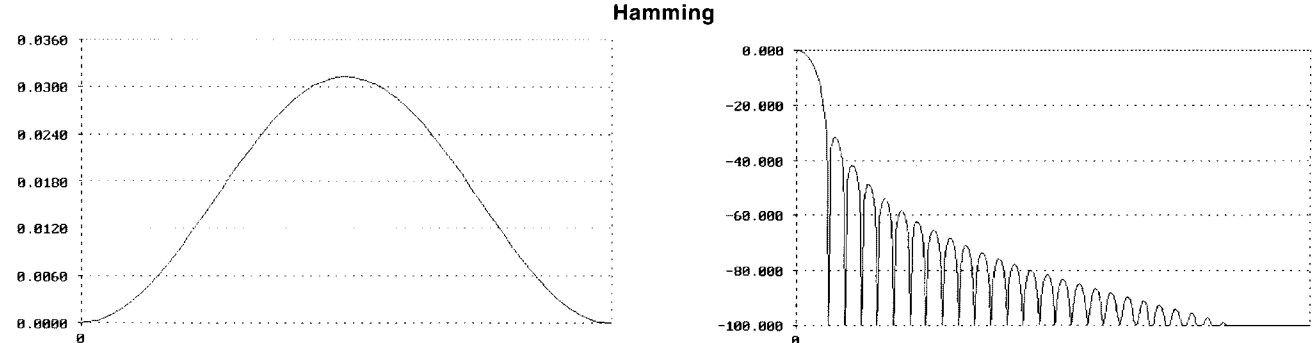
Triangular



Blackman



Hamming



Hanning

Fig 10—Various window functions and their Fourier transforms.

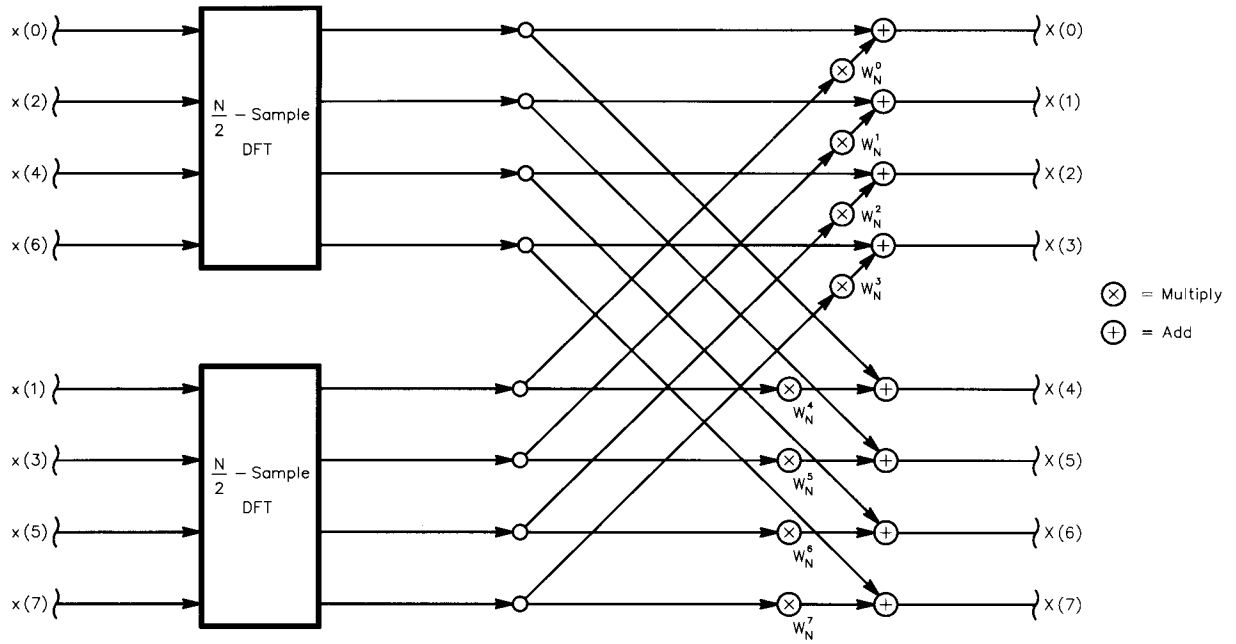


Fig 11—An eight-sample DFT as two four-sample DFTs.

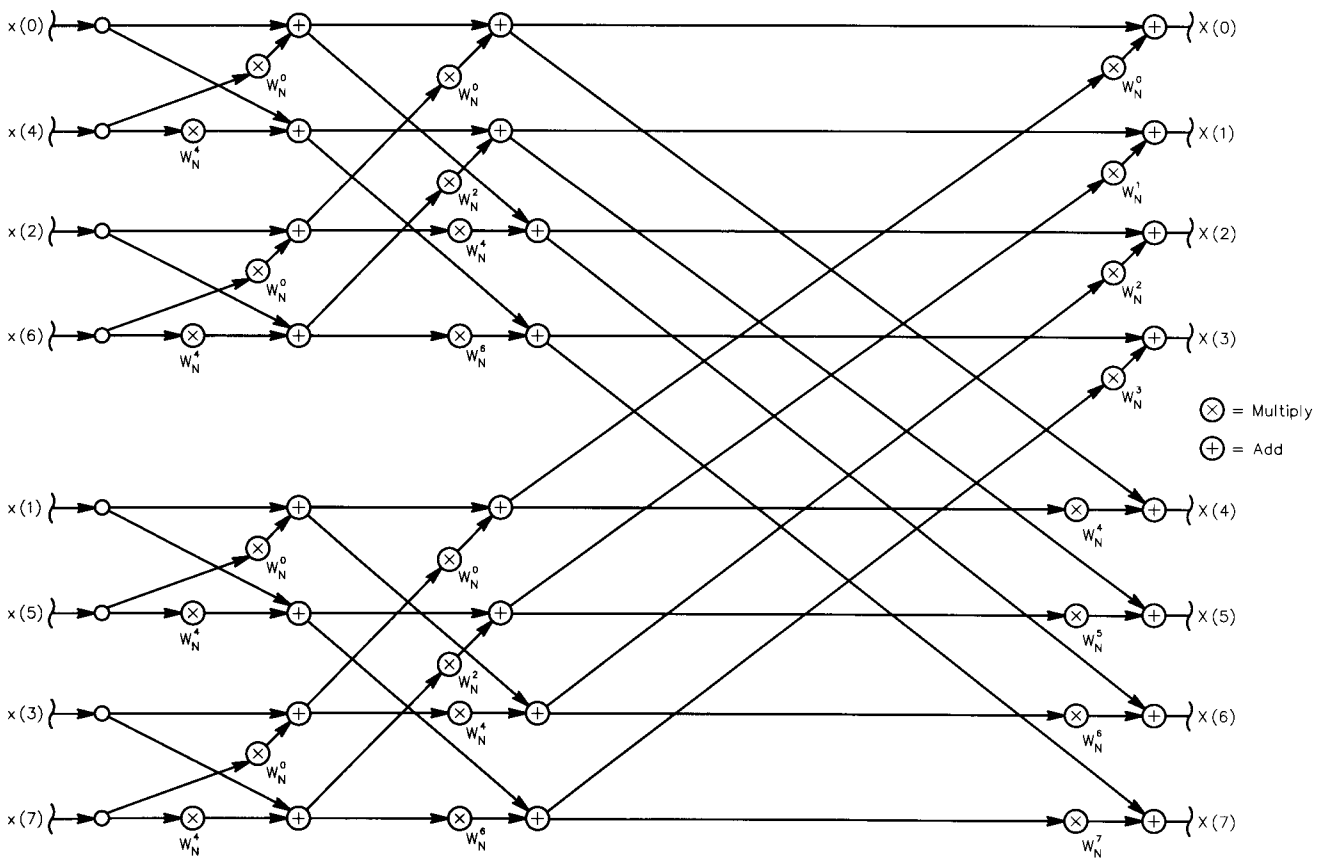


Fig 12—An eight-sample FFT.

It's evident that Eq 28 represents two N / 2-sample DFT calculations. It has in fact eased the computational load, since it requires:

$$N+2\left(\frac{N}{2}\right)^2 = N+\frac{N^2}{2} \qquad \text{(Eq 29)}$$

complex multiplications and additions. Note that $W_N{}^k$ is a function only of k, and is therefore a set of constants. We have reduced calculations by the factor:

$$\frac{N+\frac{N^2}{2}}{N^2} = \frac{1}{N}+\frac{1}{2} \qquad \text{(Eq 30)}$$

which for large N is nearly a two-fold reduction.

This is where flow charts become useful, so Eq 28 is used to produce Fig 11, an example of an eight-sample DFT calculation as broken into two four-sample calculations.

Carrying this idea further, since N is an integral power of two, we can break each of these N / 2-sample DFT calculations down into separate N / 4-sample calculations. Then we break each of those into separate N / 8-sample calculations, and so on, until

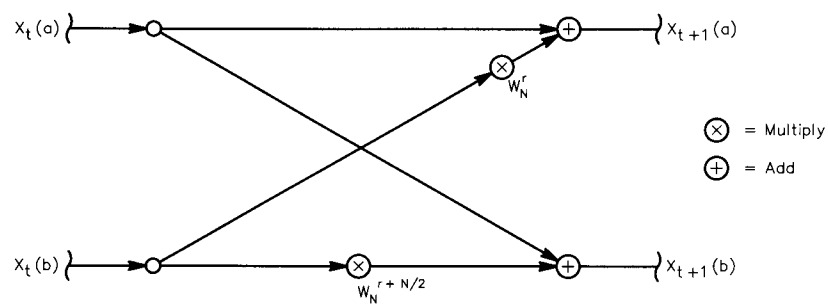


Fig 13—Butterfly calculation for a decimation-in-time FFT.

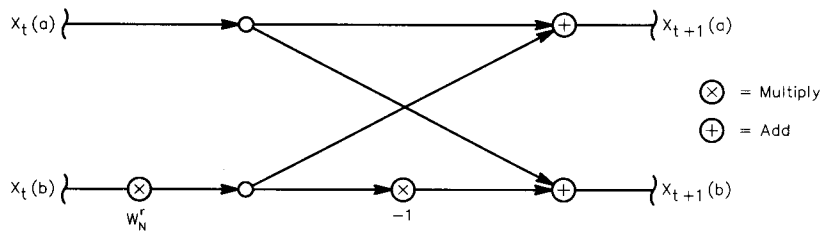


Fig 14—Modified butterfly.

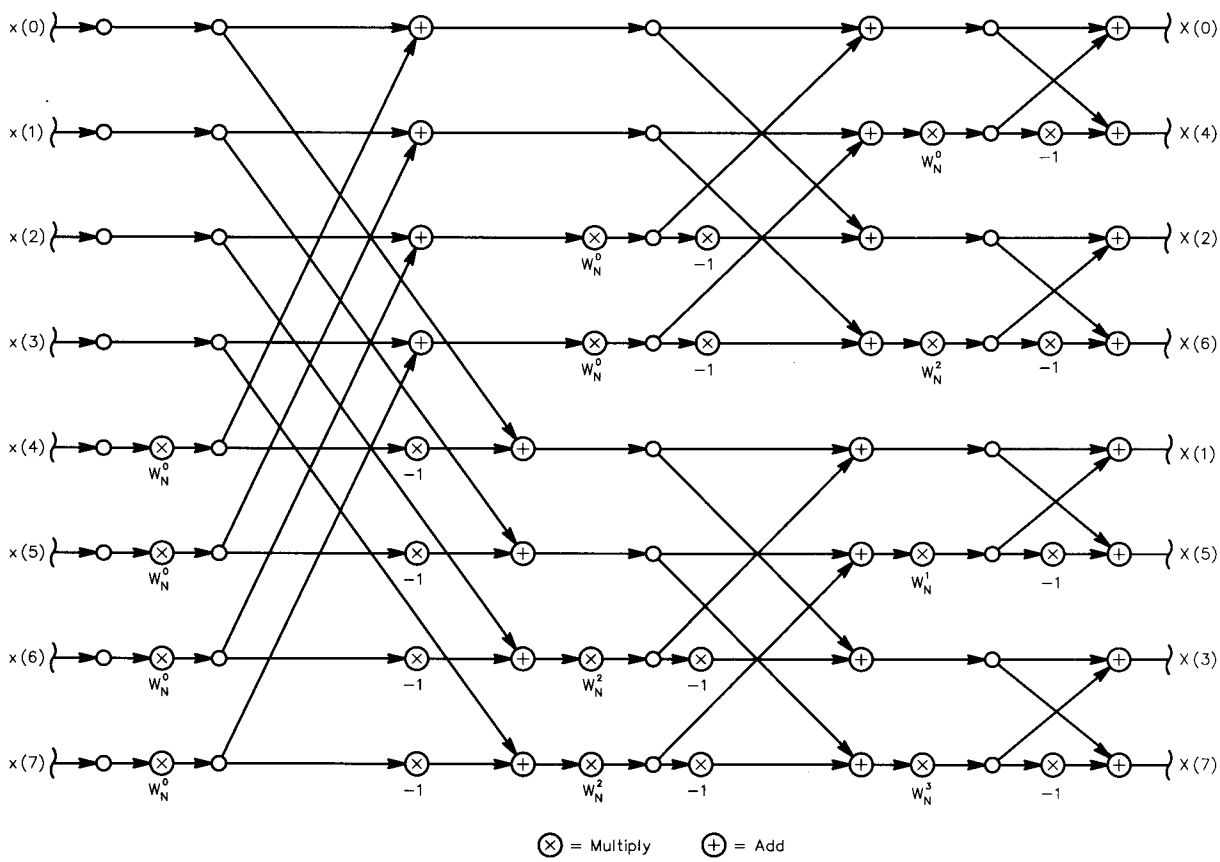


Fig 15—A decimation-in-time FFT with different input-output order and using modified butterflies.

we're left with only two-sample DFT calculations. From this discussion, it's obvious that we can break things down only $\log_2 N - 1$ times until we get to the two-sample level. When this has been done for our eight-sample DFT, the resulting flow chart is a complete FFT, as shown in Fig 12.

In-Place Calculation

The figure shows that starting with eight input samples, eight outputs are generated. Each stage requires N complex multiplications and additions, and there are $\log_2 N$ stages; hence, the total burden is $N \log_2 N$. Further, each stage transforms N complex numbers into another set of N complex numbers. This suggests we should use a complex array of dimension N to store the inputs and outputs of each stage as we go along. Finally, an examination of the branching of terms in the diagram reveals that pairs of intermediate results are linked by pairs of calculations like the one shown in Fig 13. Because of the appearance of this diagram, it is known as a *butterfly computation.*

This computational arrangement requires two complex multiplications and additions. For each butterfly, the intermediate results are in the form:

$$\begin{aligned} X_{t+1}(a) &= X_t(a) + W_N^{\,r} X_t(b) \\ X_{t+1}(b) &= X_t(a) + W_N^{\,r+\frac{N}{2}} X_t(b) \end{aligned} \qquad \text{(Eq 31)}$$

where t represents the stage number of the calculations, and a and b are the branch numbers. Note that:

$$W_N^{\,r+\frac{N}{2}} = W_N^{\,r} W_N^{\,\frac{N}{2}} \qquad \text{(Eq 32)}$$

and:

$$\begin{aligned} W_N^{\,\frac{N}{2}} &= e^{\frac{-2\pi j \frac{N}{2}}{N}} \\ &= e^{-j\pi} \\ &= -1 \end{aligned} \qquad \text{(Eq 33)}$$

So now Eq 31 can be written:

$$\begin{aligned} X_{t+1}(a) &= X_t(a) + W_N^{\,r} X_t(b) \\ X_{t+1}(b) &= X_t(a) - W_N^{\,r} X_t(b) \end{aligned} \qquad \text{(Eq 34)}$$

The equivalent flow diagram is shown as Fig 14. Now that's slick, since we just reduced the total multiplications by an additional factor of two! The total burden is now proportional to $(N / 2) \log_2 N$.

These calculations can be performed in place because of the one-to-one correspondence between the inputs and outputs of each butterfly. The nodes are connected horizontally on the flow diagrams. The data from locations a and b are required to compute the new data to be stored in those same locations, hence only one array is required during calculation.

An interesting result of our decomposition of the input sequence $x(n)$ is that in the FFT calculation of Fig 12, the input samples are no longer in ascending order. In fact, they are indexed in *bit-reversed* order. It turns out that this is a necessity for doing the computations in place. To see why this is so, let's review what we did in the derivation above.

First, we separated the input samples into evens and odds. So naturally, all the even samples appear in the top half, and the odds in the bottom half. The index n of an odd sample has its least-significant bit (LSB) set; an odd sample's LSB is cleared.

Next, we separated each of these sets into their even and odd parts, which can be done by examining the second LSB in the index n. This process was repeated until we had N subsequences of unity length. It resulted in the sorting of the input data in a bit-reversed way. This isn't very convenient for us in setting up the computation, but at least the output arrives in the correct order!

Alternative Forms

It's possible to rearrange things such that the input is in normal order, and the output is in bit-reversed order. See Fig 15. In-place computation is still possible. While it's even possible to arrange things such that the input *and* the output are in normal order, that makes in-place computation impossible.

We obtained a decimation-in-time algorithm by decomposing the input sequence $x(n)$ above. It's also possible to decompose the output sequence in the same way, with the same computational savings. Algorithms obtained in this way are called *decimation-in-frequency.*

To begin this derivation, we again separate the input sequence $x(n)$ into two parts, but this time, they are the first and second halves:

$$\begin{aligned} X(k) &= \sum_{n=0}^{\frac{N}{2}-1} W_N^{nk} x(n) + \sum_{n=\frac{N}{2}}^{N-1} W_N^{nk} x(n) \\ &= \sum_{n=0}^{\frac{N}{2}-1} W_N^{nk} x(n) + \left(W_N^{\frac{kN}{2}}\right) \sum_{n=0}^{\frac{N}{2}-1} W_N^{nk} x\left(n+\frac{N}{2}\right) \end{aligned} \qquad \text{(Eq 35)}$$

Using the relation:

$$W_N^{\frac{kN}{2}} = (-1)^k \qquad \text{(Eq 36)}$$

and combining the summations, we get:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} W_N^{nk} \left[x(n) + (-1)^k x\left(n+\frac{n}{2}\right) \right] \qquad \text{(Eq 37)}$$

Now let's break the output sequence into even and odd parts, again using index $r < N / 2$ as above:

$$\begin{aligned} X(2r) &= \sum_{n=0}^{\frac{N}{2}-1} W_N^{2rn} \left[x(n) + x\left(n+\frac{N}{2}\right) \right] \\ X(2r+1) &= W_N^{\,n} \sum_{n=0}^{\frac{N}{2}-1} W_N^{2rn} \left[x(n) - x\left(n+\frac{N}{2}\right) \right] \end{aligned} \qquad \text{(Eq 38)}$$

Since:

$$W_N^{2rn} = W_{\frac{N}{2}}^{rn} \qquad \text{(Eq 39)}$$

Eq 38 can be written:

$$\begin{aligned} X(2r) &= \sum_{n=0}^{\frac{N}{2}-1} W_{\frac{N}{2}}^{rn} \left[x(n) + x\left(n+\frac{N}{2}\right) \right] \\ X(2r+1) &= W_N^{\,n} \sum_{n=0}^{\frac{N}{2}-1} W_{\frac{N}{2}}^{rn} \left[x(n) - x\left(n+\frac{N}{2}\right) \right] \end{aligned} \qquad \text{(Eq 40)}$$

and the result is again two $N / 2$-sample DFT calculations.

Proceeding in direct analogy to the decimation-in-time algorithm above, decomposition continues until we have only two-sample DFT calculations left, and for the case $N = 8$, the flow diagram appears as shown in Fig 16.

Note that this algorithm can be performed as butterflies, and that calculation in place is possible just as before. The butterflies are a bit different now, however, as depicted in Fig 17. The corresponding equations are:

$$X_{t+1}(a) = X_t(a) + X_t(b)$$
$$X_{t+1}(b) = \left(X_t(a) - X_t(b)\right)W_N^r \qquad \text{(Eq 41)}$$

While it's a bit difficult to see at first, we can state the following: For every decimation-in-time algorithm there exists a decimation-in-frequency algorithm that is equivalent to swapping the input and the output, and reversing all the arrows in the flow diagram. This duality is useful as we consider computing the *inverse FFT* (FFT^{-1}).

Returning to the Time Domain

NR systems are typical in that after obtaining the FFT, we perform some modification of the frequency-domain data; we then transform the modified data back to the time domain. Since Eq 18 and 22 are so similar, the type of algorithms described above can be used to compute the FFT^{-1}. One way is to simply substitute $\frac{1}{2}W_N^{-kn}$ for W_N^{kn} at each stage in Fig 12, and of course use *X(k)* as the input to obtain *x(n)* as the output. This results in the diagram of Fig 18.

Alternatively, we can compute the FFT^{-1} by using either form of FFT flow diagram, swapping the inputs and outputs and reversing direction of signal flow, as mentioned before. It's important to note that this is a consequence of the fact that we can rearrange the nodes of the flow diagrams however we want without altering the result. So, they work just as well in reverse as they do in the forward direction!

It's convenient to have the output order of the FFT the same as the input order of the FFT^{-1}, so we're wise to use decimation-in-time for one conversion, and decimation-in-frequency for the other.

General Computational Considerations

This business of bit-reversed indexing is what usually ties one's brain in knots during coding of these algorithms, and it's certainly one of the first things to be tackled—so let's have at it! Several approaches are feasible: a look-up table, the bit-polling method, reverse bit-shifting and the reverse-counter approach.

The look-up table is perhaps the most straightforward method. The table is calculated ahead of time, and the index used as the address to the table. See Fig 19. Most systems don't require extremely large values of N, so the space taken by the table isn't

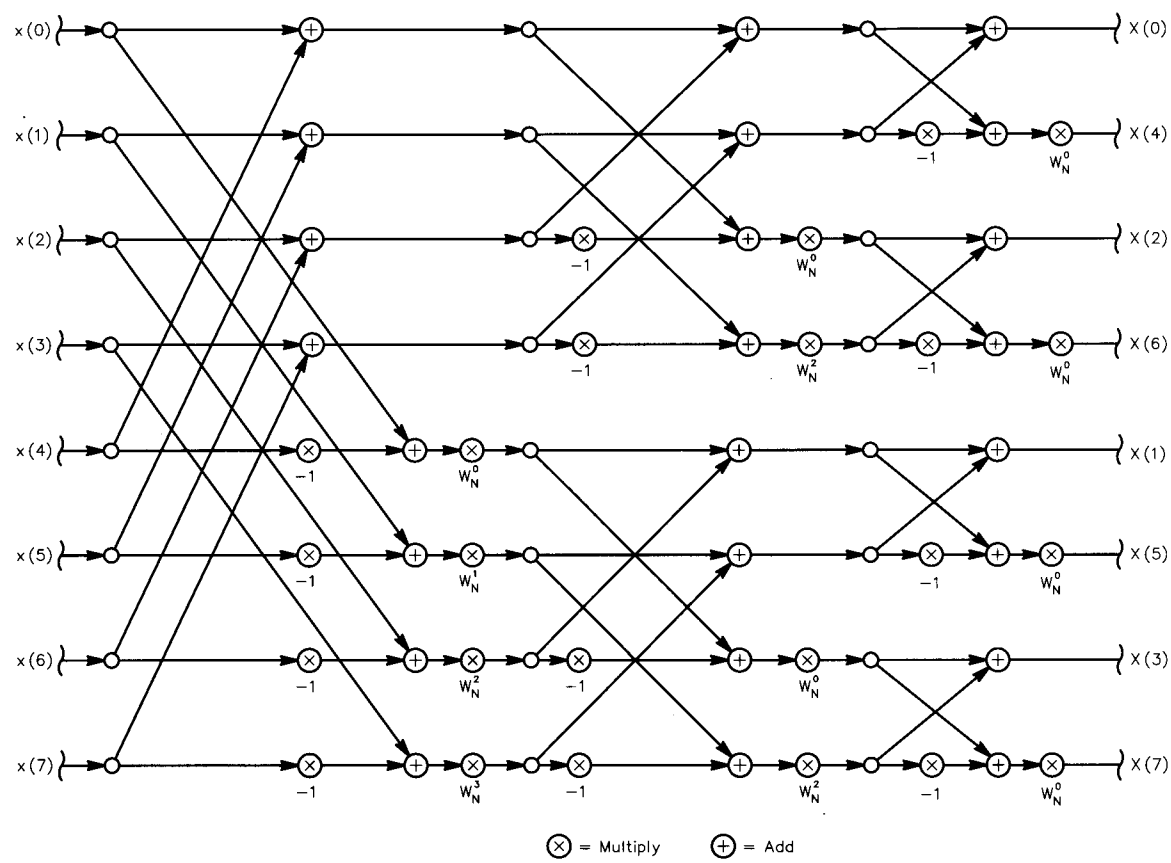


Fig 16—An eight-sample decimation-in-frequency FFT.

objectionable. For more space-sensitive applications, the bit-polling method may be attractive.

Since the bit-reversed indices were generated through successive divisions by two and determination of odd or even, a tree structure can be devised that leads us to the correct translation, based on bit-polling. See Fig 20. The algorithm examines the LSB, then branches either upward or downward in the tree based on the state of the bit. Then the second LSB is examined, a branch taken, and the procedure is repeated until all bits have been examined.

The bit-shifting method requires the same computation time. Two registers are used, one for the input index shifting right through the carry bit, the other shifting left through the carry. After all bits have been shifted, the left-shifting register contains the result. See Fig 21.

Finally, Gold and Rader[11] have described a flow diagram for a bit-reversal counter that can be "decremented" each time the index is to change. If the data are actually to be moved during sorting, the exchange is made between data at input index n and bit-reversed index m, but only once! In other words, only $N / 2$ exchanges need to be performed.

During the actual calculations, the indexing of data and coefficients requires attention to many details. In particular, several symmetries about offsets of the index can be exploited. In the case of a decimation-in-time FFT, at the first stage, all the multipliers are equal to $W_N^0 = 1$, so no actual multiplications need take place; all the butterfly inputs are adjacent elements of the input array $x(n)$. At the second stage, the multipliers are all either W_N^0 or integral powers of $W_N^{N/4}$, and the butterfly inputs are two samples apart. At the tth stage, the multipliers are all integral powers of $W_N^{N/2t}$, and the butterfly inputs are separated by

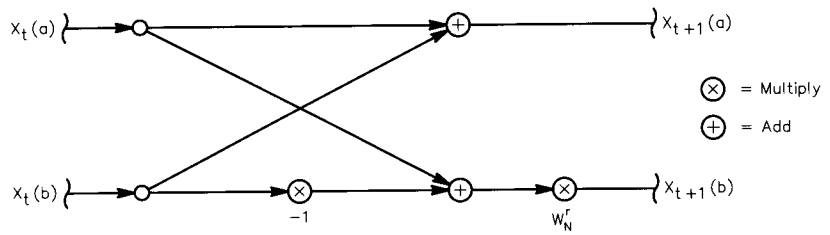


Fig 17—Butterfly calculation for decimation-in-frequency.

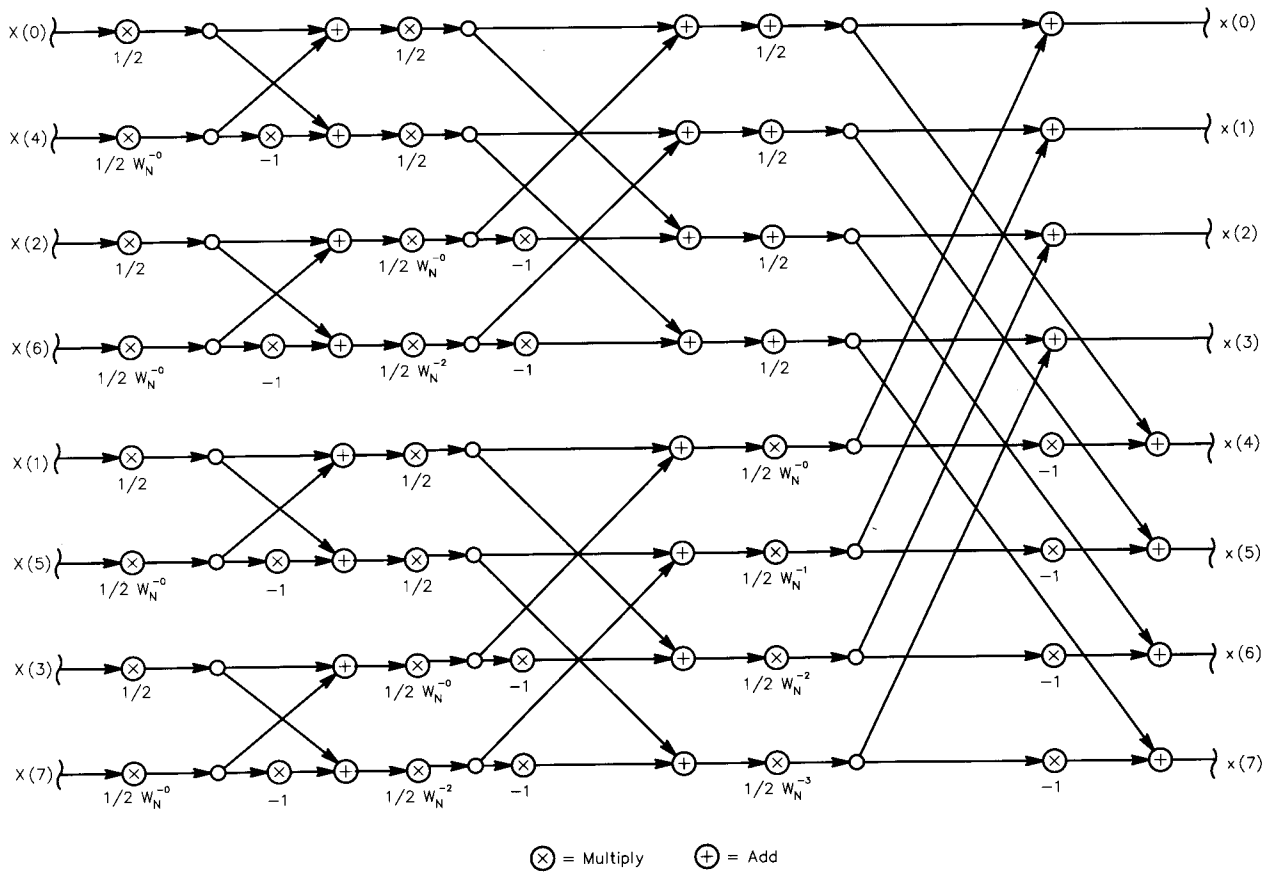


Fig 18—FFT^{-1} implemented by interchange of input, output and coefficients.

2^{t-1} samples. Note that in most of the algorithms we've described, the coefficients are indexed in normal order.

Each FFT algorithm has its own unique indexing requirements. For our NR system, we don't care that the FFT result is stored in bit-reversed order, since we're not performing any frequency-specific operations. We'll use a decimation-in-time FFT to have the input sequence $x(n)$ in normal order, and a decimation-in-frequency FFT^{-1} to produce an output sequence also in normal order.

The coefficients of the complex exponential W_N can be obtained in various ways. The most common way is to generate them ahead of time and store them in a table. Another way is to use a *recursion formula* to generate them as needed. Since in general, the coefficients are all integral powers of W_N, we can use:

$$W_N^{kh} = W_N^k W_N^{k(h-1)} \quad \text{(Eq 42)}$$

to get the hth coefficient from the $h-1$th. However, errors will build up over time with this method because of the finite precision of our mathematics; each multiplication generates a rounding or truncation error that adds to the total. It's necessary to reset the value at periodic intervals to prevent divergence.

We'll see below that we always have to accept *some* error in our results because of accumulated rounding or truncation no matter how the DFT is calculated. We will analyze these *quantization* effects for a direct DFT calculation, and for a decimation-in-time FFT calculation.

Numerical-Accuracy Effects in DFT and FFT Calculations

In each complex multiplication, we must perform four real multiplications. Each of these contributes a round-off or truncation error to the output. We need to make some assumptions about the errors in order to do any analysis of them. Fixed-point, two's-complement arithmetic is assumed as well, as is typical in an embedded implementation.[12]

First, if b is the number of bits used to represent numbers, we'll assume the errors are uniformly distributed over the range:

$$-2^{(-b-1)} \le \varepsilon \le 2^{(-b-1)} \quad \text{(Eq 43)}$$

Each one of the errors therefore has variance:[5]

$$\sigma^2 = \frac{2^{(-2b)}}{12} \quad \text{(Eq 44)}$$

Address	Data
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

Fig 19—Bit-reversal look-up table.

Also, we assume the errors are uncorrelated with each other, and also uncorrelated with the input and output.

Since noise powers add, the average value of the noise power is the *expected value* of four times the variance:

$$E\left[\varepsilon^2\right] = 4\left(\frac{2^{-2b}}{12}\right) = \frac{2^{-2b}}{3} \quad \text{(Eq 45)}$$

At the output, the noise power is N times worse for the direct DFT calculation:

$$\sum_{n=0}^{N-1} E\left[\varepsilon_n^2\right] = \frac{2^{-2b}N}{3} \quad \text{(Eq 46)}$$

Just as in the case of FIR filter calculations,[6] the noise at the output is directly proportional to N.

Also like the FIR analysis, the DFT calculations are subject to a dynamic-range limitation on the large-signal end of things. To prevent overflow, we require:

$$\left|X(k)\right| < 1 \quad \text{(Eq 47)}$$

and this can be ensured if:

$$\sum_{n=0}^{N-1} \left|x(n)\right| < 1 \quad \text{(Eq 48)}$$

We may need to scale the input by $1/N$ to prevent overflow. This scaling requirement has the effect of making the output noise worse, as will be discussed below.

For the decimation-in-time FFT calculation, the same assumptions about the nature of the noise are used. Referring to Fig 12, note that no more than one noise source is inserted at each node, because of the single complex multiplication there. The total noise at any node is the cumulative effect of all sources that propagate through to that node. Since we assume all the noise sources are uncorrelated, no more than $N-1$ noise sources propagate through to each output. So, the total output noise is, again, roughly proportional to N. When we take into account the requirement for data scaling, however, we'll see that the noise power must increase because the signal power at each node must decrease.

In fact, it can be shown[5] that the SNR at the output—using optimum stage-by-stage scaling—cannot be better than:

$$SNR_{OUTPUT} = \frac{2^{(2b-2)}}{N} \quad \text{(Eq 49)}$$

or a factor of 12 worse than the result given in Eq 46.

Until now, we've assumed absolute accuracy for the values of the coefficients W_N^{kn}. Whether these are held in a table or calculated "on the fly," they

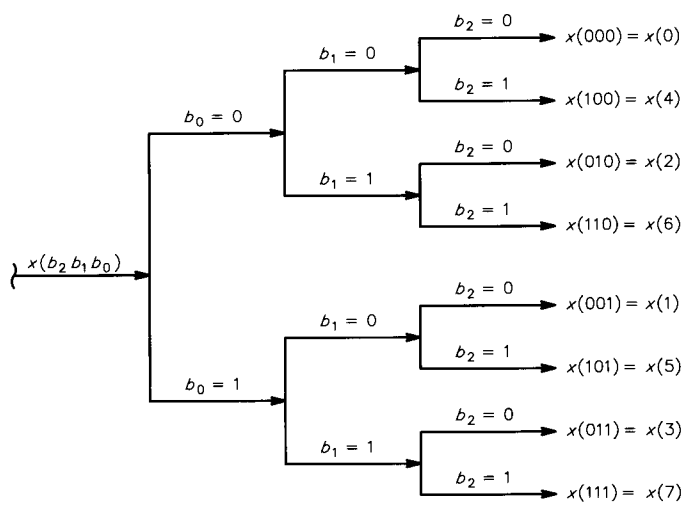


Fig 20—Bit-reversal tree.

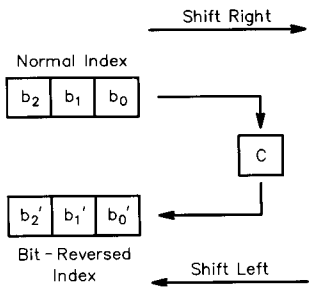


Fig 21—Bit-shifting register arrangement.

must be quantified to some number of bits, b; this results in further cumulative error in the output. The analysis of the effects is much more difficult than that for data quantization above, since the nature of coefficient quantization is inherently nonstatistical.

Useful results have been obtained[13] by introducing artificial noise or *jitter* into the coefficients, and analyzing the results for output error. The result obtained was that output SNR cannot exceed:

$$SNR_{OUTPUT} = \frac{3\left(2^{2b+1}\right)}{p} \qquad \text{(Eq 50)}$$

where

$$p = \log_2 N \qquad \text{(Eq 51)}$$

The experimental results confirmed that noise from this effect increases in proportion to p, which means the increase with respect to N is slow. In other words, doubling N results in only a slight degradation in the SNR.

Engineering statistics is about the *grungiest* thing in the world, isn't it? Nevertheless, it sure is nice to know what to expect from these wonderful theories before committing to an implementation, because otherwise, *unexpected things can occur*!

There is a different calculation method for the DFT that ultimately dispenses with complex exponentials and improves speed and simplicity by a significant factor. Frerking touches on the idea, but provides no method for the control of its inherent divergence problem (see Note 12). I call it the "Damn-Fast Fourier Transform." We'll begin with that method in Part 4 of this series.

Doug Smith, KF6DX/7, is an electrical engineer with 18 years experience designing HF transceivers, control systems and DSP hardware and software. He joined the amateur ranks in 1982 and has been involved in pioneering work for transceiver remote-control and automatic link-establishment (ALE) systems. At Kachina Communications in central Arizona, he is currently exploring the state of the art in digital transceiver design.

Notes

[1]Widrow, B. and Stearns, S. D., *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
[2]Skolnik, M. I., *Introduction to Radar Systems*, McGraw-Hill, New York, NY, 1962.
[3]Widrow, B. and Hoff, M. E., Jr, "Adaptive Switching Circuits," *IRE WESCON Convention Records*, Pt 4, IRE, 1960.
[4]Hutchins, R. M., Ed-in-chief, *Great Books of the Western World*, 31st printing, Encyclopaedia Britannica Inc, Chicago, Illinois, 1989.
[5]Oppenheim, A. V. and Schafer, R. W., *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
[6]Smith, D., "Signals, Samples and Stuff: A DSP Tutorial" (Pt 1), *QEX*, Mar/Apr 1998, ARRL, Newington, Connecticut.
[7]Gardner, W. A., *Statistical Spectral Analysis: A Non-probabilistic Theory*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
[8]Proakis, J. G., Rader, C. M., *et al*, *Advanced Digital Signal Processing*, Macmillan Publishing Company, New York, New York, 1992.
[9]Runge, C., *Z. Math. Physik*, Vol 48, 1903; also Vol. 53, 1905.
[10]Cooley, J. W. and Tukey, J. W., "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Computation*, Vol 19, 1965.
[11]Gold, B. and Rader, C. M., *Digital Processing of Signals*, McGraw-Hill, New York, New York, 1969.
[12]Frerking, M. E., *Digital Signal Processing in Communications Systems*, Van Nostrand-Reinhold, New York, New York, 1993.
[13]Weinstein, C. J., "Roundoff Noise in Floating Point Fast Fourier Transform Computation," *IEEE Transactions on Audio and Electroacoustics*, Vol AU-17, 1969.