USING THE FFT FOR DSP SPECTRUM ANALYSIS: A TELEMETRY ENGINEERING APPROACH

Glenn Rosenthal and Thomas Salley

Metraplex Corporation
5305 Spectrum Drive
Frederick, Md. 21701
301-662-8000

Abstract

The Fast Fourier Transform (FFT) converts digitally sampled time domain data into the frequency domain. This paper will provide an advanced introduction for the telemetry engineer to basic FFT theory and then present and explain the different user pre-processing options that are available when using the FFT. These options include: using windowing functions, "zero filling" for frequency data interpolation, and setting the frequency resolution of the FFT resultant spectrum,

Key Words: Fast Fourier Transform, Windowing Functions, Zero-Filling, Frequency Resolution

Advanced Introduction to FFT Theory

The FFT algorithm (published by Cooley and Tukey in 1965) is a computation saving algorithm based on the Discrete Fourier Transform (DFT). The DFT translates discrete digital data from the time domain (amplitude vs. time) to the frequency domain (amplitude vs. frequency). DFT input data, $f(n)$, is a finite length sequence of N samples of data equally spaced in time., and it is assumed that this sequence repeats itself.

$$\text{DFT: } X(k) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi kn/N}$$

where $k = 0, 1, \ldots N\text{-}1$

DFT input data must follow the *Sampling Theorem*, which states that the data must be sampled in a periodic manner with a minimum of two data points per the fastest period of data. The frequency at one-half of the sampling rate is generally referred to as the *Nyquist rate*.

Sampling of analog input data is done using an Analog-to-Digital converter (A/D) with an anti-aliasing filter prior to the A/D. The anti-aliasing filter must filter frequencies higher than the Nyquist rate from the analog input data.

If frequencies higher than the Nyquist rate are passed to the A/D, aliasing of the spectrum data will be produced. Aliasing causes the frequency components of data that are greater than the Nyquist rate to be displayed as erroneous lower frequencies. The exact location of the aliased data is equal to the actual input data frequency minus a multiple of the Nyquist rate.

As previously stated, the FFT evolved from the DFT as a computational saving algorithm, because it reduced the number of multiplications required to evaluate the summation. The FFT recognizes certain symmetries and periodicities in the DFT process when the number of samples to be evaluated are a power of 2. If N is the number of data points in the summation, then the DFT requires N times N multiplications, whereas the FFT algorithm only requires only $N\log_2 N$ multiplications. This savings can be significant For example, if a 1024-point digital spectrum analysis is to be performed, the DFT will require 1 million multiplications compared to only ten thousand multiplications for the FFT. In this case, the FFT is 100 times faster to calculate than the DFT! There is no accuracy loss in performing the FFT instead of the DFT.

Since the FFT algorithm has complex terms (real and imaginary) in the definition, the output also has both a real (odd) portion of the spectrum and an imaginary (even) portion of the spectrum. If only real data is entered into the summation, the imaginary portion of the spectrum will be a mirror image across the origin (at DC or 0 Hz) of the real part of the spectrum. This means, if a 1024-point FFT is performed, 512 real and 512 imaginary data points are calculated.

A “C” language program for performing the FFT is included in the Appendix.

Pre-Processing Input Data

Before performing an FFT, the user has three different pre-processing options that can be performed to help enhance the output of the FFT algorithm: (1) the *number of data points* to use in the algorithm, (2) how much *zero-filling* of the input data to perform, and (3) which *windowing function* to perform on the input data. By making educated choices

for these three preprocessing options, the information in the output resultant FFT spectrum will be enhanced. These options involve tradeoffs between processing time, frequency resolution, and spectral component bandwidths. The following sections describe how to best use each pre-processing option.

FFT Frequency Resolution

When performing an FFT, the user is able to decide the number of data points to be used in the algorithm. The number of input data points sets the frequency resolution for the resultant FFT spectrum. The frequency resolution of each output frequency component is equal to the Nyquist rate divided by half the number of data points entered. In other words, the FFT output data spectral points are equally spaced from 0 Hz (DC) to the Nyquist rate in both positive and negative frequency.

To understand how the frequency resolution of the resultant FFT data is determined, one can consider the FFT algorithm as a group of bandpass filters in the spectrum from DC to the Nyquist rate. The number of bandpass filters is equal to half the number of the input data points because only one-half the output spectral data points are real and the other half are imaginary. The bandwidth of each bandpass filter is set by the number of data points entered. As more data points are entered into the FFT, the bandwidth of each bandpass filter is reduced, giving greater frequency resolution to each output term.

By increasing the number of input data points, the time to perform the FFT is also increased. The calculation time increases by $N1\text{og}_2N$ number of multiplications that are performed for a N-point FFT. Therefore, as the frequency resolution is increased, so is the calculation time for each FFT.

For example, consider a set of input data points that is sampled with a Nyquist rate of 512 kHz. If a 512-point FFT is calculated, the frequency resolution of the FFT will be 2 kHz and 4608 multiplications will be performed. If a 1024-point FFT is performed, the FFT frequency resolution would be 1 kHz and 10,240 multiplications will be performed. If the increase in computation time of 2.5 does not interfere with the overall system timing and performance, the size of the FFT should be set for the desired output frequency resolution desired.

Zero Filling

Zero filling is a method to interpolate spectral components and sidelobes of the FFT output data. Zero filling is accomplished by actually entering the value zero for some of the input data into the FFT.

Zero filling is used for two different reasons. The first reason is to interpolate new values between the original data samples. If a 512-point FFT is performed with 512 "non-zero filled" data points, potential ambiguities between non-interpolated sidelobes and spectral components can result. If only one-fourth of the data actually entered into the FFT is sampled data and three-fourths of the data is zero filled, the resultant FFT will still have the same frequency resolution but the sidelobe and secondary carriers will be interpolated to show the peaks and values of each term.

A second reason for zero-filling input data is when hardware or software timing restrictions in the total FFT generating system limit the number of data points that can be acquired. If the data acquisition system's A/D converter or input digital data transfer rate is much slower than the FFT generating system, the update rate of the output spectrum will be delayed while waiting for input data. By zero-filling some of the input data to the FFT, the update rate of the output FFT spectrum will be increased. The output FFT will have the appearance of good frequency resolution because of the interpolation of both the spectral components and spectral leakage terms or sidelobes.

Windowing Functions

Since the FFT processes data within a finite time duration, spectral leakage occurs within the sampled data. This spectral leakage corresponds to the abrupt start and stop of a sampled waveform which is non-periodic with the sampling period for the FFT. The spectral leakage appears as sidelobes in the FFT output. Windowing functions taper the beginning and ending data that forces periodicity within the sampling period and minimizes the spectral leakage.

Windows are applied to the sampled data as multiplication weighting functions. There are many different windowing functions which are used to minimize the side lobes generated by the FFT. Choosing the correct windowing function for a particular application isn't straight forward. Each windowing function affects the bandwidth of the main frequency components are well as the amplitude of the sidelobes. Ideally, the window to use would be one which produced the smallest frequency component bandwidth with the minimum sidelobe amplitude. Unfortunately, as a windowing function reduces the sidelobes, the bandwidth of the frequency components increase.

This paper will illustrate four different windowing functions: Rectangular, Hamming, Hanning, and Blackman. The series definition of each windowing function with a comparison of the maximum sidelobe amplitude verses frequency component bandwidth is illustrated in Table 1. A comparison of the shape of each of these windowing functions is shown in Figure 1.

Table 1. Windowing Function Definitions

Windowing Function	Series Definition	Max Sidelobe Amplitude	Normalized Bandwidth
Rectangular	$\sum_{n=0}^{N-1} 1$	-13 dB	1.00
Hanning	$\sum_{n=0}^{N-1} 0.5 - 0.5\cos(2\pi n/N)$	-32dB	1.62
Hamming	$\sum_{n=0}^{N-1} 0.54 - 0.46\cos(2\pi n/N)$	-43dB	1.46
Blackman	$\sum_{n=0}^{N-1} 0.42 - 0.5\cos(2\pi n/N + 0.8\cos(2\pi n/N)$	-58dB	1.88

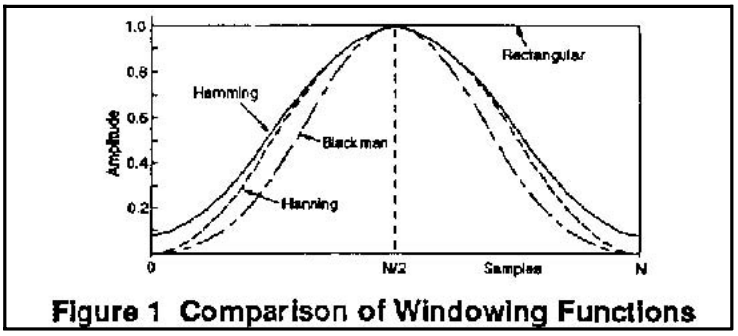

Figure 1 Comparison of Windowing Functions

The Rectangular window is sometimes called the “do nothing” window because the data is not changed by the windowing function. Therefore, if sampled data is directly put into the FFT, the resultant spectrum will have the smallest frequency bandwidth and the largest sidelobe amplitudes. The other windowing functions actually taper the data at the start and end of the sampled interval, which reduces the sidelobe amplitude but causes the bandwidth of the frequency components to spread. If a single frequency component is present in the data, then the greatest sidelobe suppression would be desirable. On the other hand, if multiple frequency components exist, spreading the bandwidth by using a sidelobe suppression window could merge the different frequency components into one term. Therefore, the choice of which window to use is dependent on the type of input data the user is expecting.

The last section of this paper shows the effects of the different windowing functions on a single input carrier.

FFT Resultant Spectrums

This section illustrates the relationship between the pre-processing variables previously described and the resultant FFT spectrums, The same input data will be used for each FFT with different pre-processing options. The input data is a computer generated waveform which is the sum of three spectral components. For clarity in describing the spectrums, the three spectral components of the combined waveform will be referred to as Components 1-3. The individual normalized FFTs of each of the three components of the summed waveform are shown in Figures 2a-2c.

Component 1 (Fig. 2a) is a sinewave having a period of 1/6 of the sample frequency or 1/3 of the Nyquist rate with a normalized amplitude of one. Component 2 (Fig. 2b) is also a sinewave but with its frequency of 1/9 of the Nyquist rate and a normalized amplitude of 0.0655 or -23.67 dB from Component 1. Component 3 (Fig. 2c) of the summed waveform is a squarewave having a frequency of 1/8 of the Nyquist rate with the fundamental frequency having a normalized amplitude of 0.0111 or -39.06 dB from the Component 1.

Figures 3a-3d show the sum of Components 1-3 in the time domain after each of the four windowing functions are performed on the input data. These four waveforms are provided for the user to be able to distinguish the effects of each of the previously defined windowing functions in Table 1 to the input summed data waveform. As a note, the Rectangular doesn't affect the data, therefore the shape of the input waveform is best seen in Figure 3a.

Figures 4 through 9 show the resultant FFT spectrums of the summed waveform after the various pre-processing options are performed. The figures are arranged by similar frequency resolution and zero-filling and different windowing functions. The frequency (x) axis of each FFT goes from DC to the Nyquist rate and the amplitude (y) axis is normalized from 0 to -80 dB. The following paragraphs will highlight the effects of each pre-processing option.

Figures 4a-4d are 512-point FFTs using 3/4 zero-filling with the different windowing functions performed. The effect of 3/4 zero-filling which causes a large interpolation of the spectral leakage terms are clearly illustrated in the Rectangular window. Component 3's odd order harmonics are totally hidden in the sidelobes of Component 1 because the normalized amplitude of Component 3 is below the amplitude of the sidelobes for a Rectangular window. By using the Hanning window in Figure 4b the spectral leakage terms are reduced, thus revealing the fifth and seventh harmonics of Component 3, However, the increased spectral component bandwidth due to the windowing function has combined the fundamentals of Components 2 and 3 into one spectral term and the third

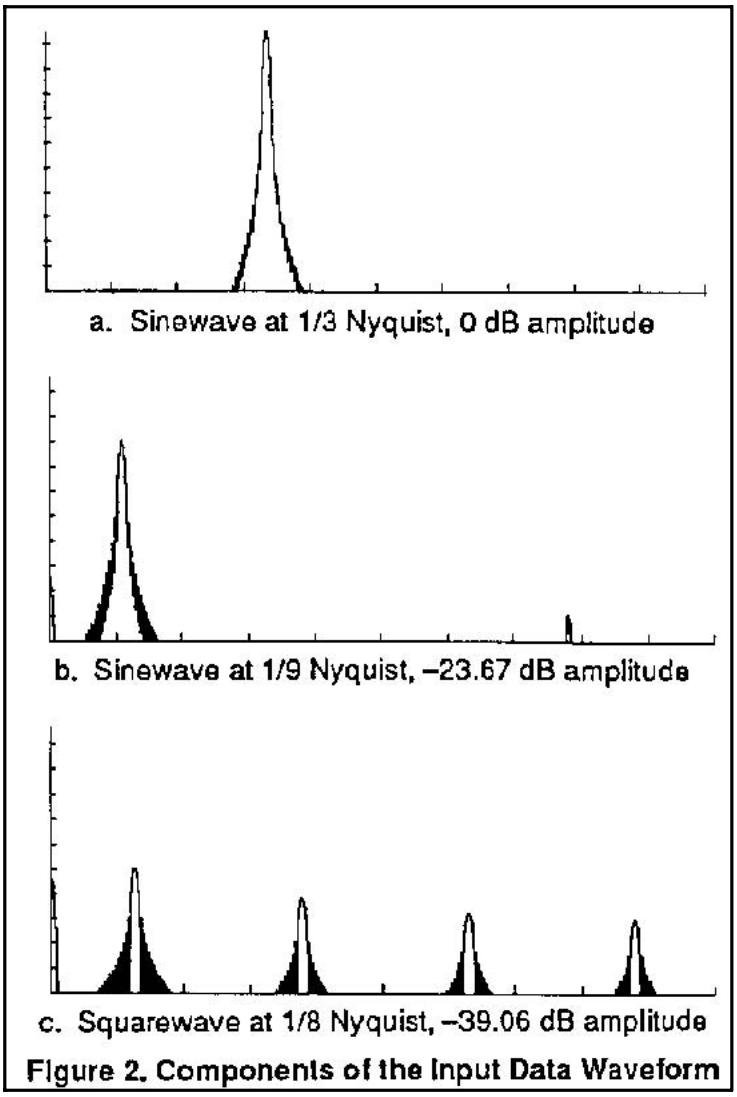

Figure 2. Components of the Input Data Waveform

harmonic of Component 3 with Component 1. Figures 4c and 4d show the effects of the different bandwidths of spectral components caused by the Hamming and Blackman windowing functions.

Figures 5a-5d are 512-point FFTs using 1/2 zero-filling. In Figure 5a, again the harmonics of Component 3 are hidden by the large amplitude sidelobes of Component 1. The Blackman window in Figure 5d reveals all of the harmonics of Component 3 due to the large sidelobe suppression in the windowing function. One problem with Figure 5d is that the large spectral bandwidth of the Blackman window has almost combined the fundamental of Component 3 with Component 2, whereas the Rectangular window in Figure 5a reveals a hint of 2 different spectral components.

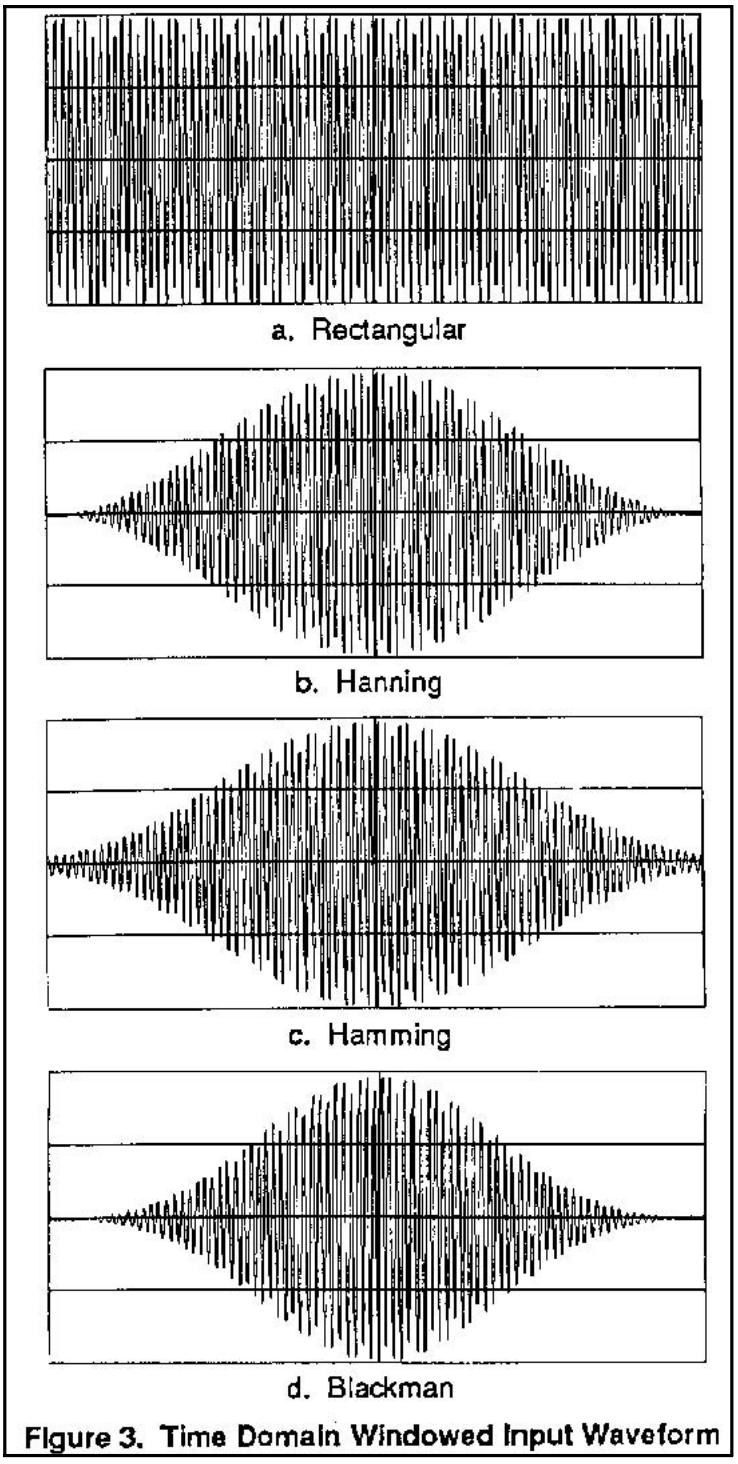

Figure 3. Time Domain Windowed Input Waveform

Figures 6a-6d are 512-point FFTs using no zero-filling. With no zero-filling or full input data resolution, the spectral components are easier to distinguish because of the lack of interpolation of the sidelobes. Even with no zero-filling, the Rectangular window (Fig. 6a) fails to illustrate all the harmonics of the low amplitude square wave in Component 3, while the other windowing functions clearly show most of the spectral components. However, the Hamming windowing function (Fig. 6c) still hides the third harmonic of Component 3 in the sidelobes of Component 1.

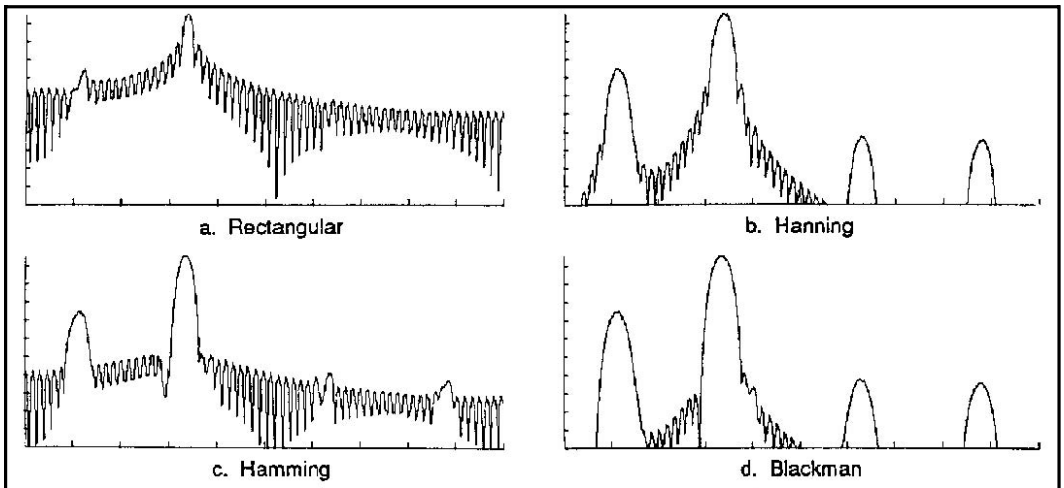

Figure 4. FFT with 512 Points and 3/4 Zero-Filling

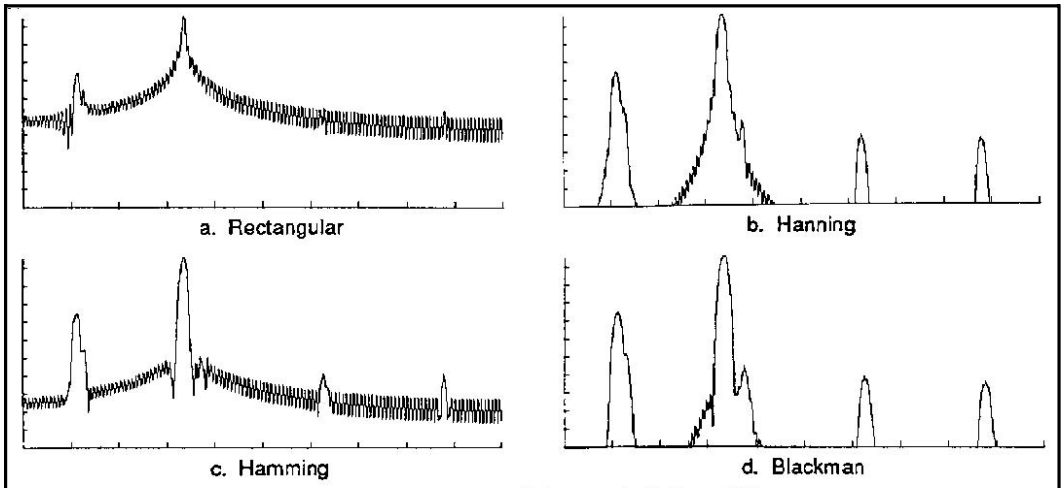

Figure 5. FFT with 512 Points and 1/2 Zero-Filling

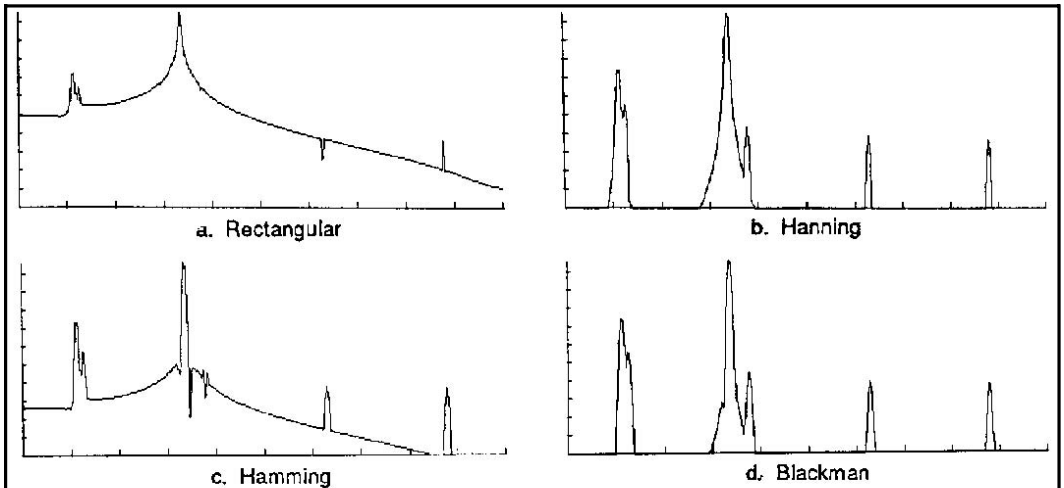

Figure 6. FFT with 512 Points and No Zero-Filling

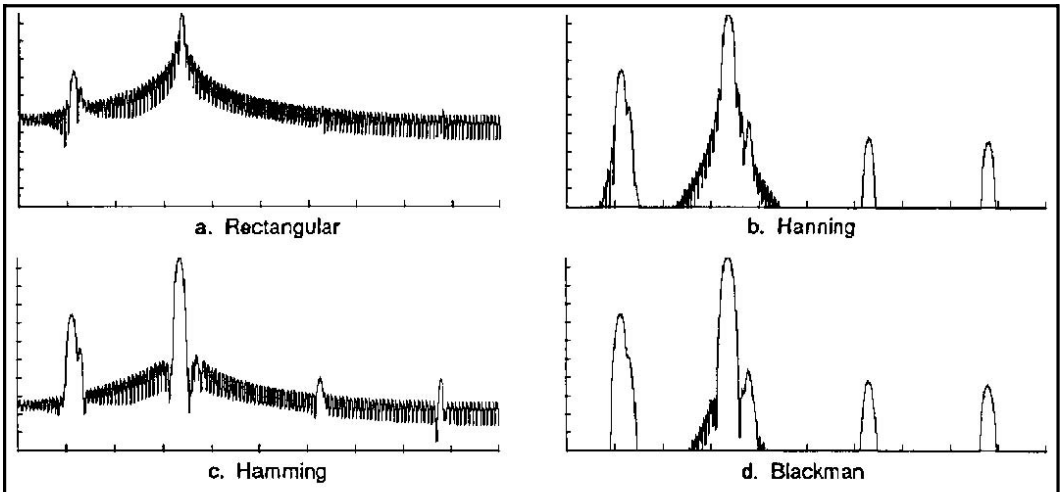

Figure 7. FFT with 1024 Points and 3/4 Zero-Filling

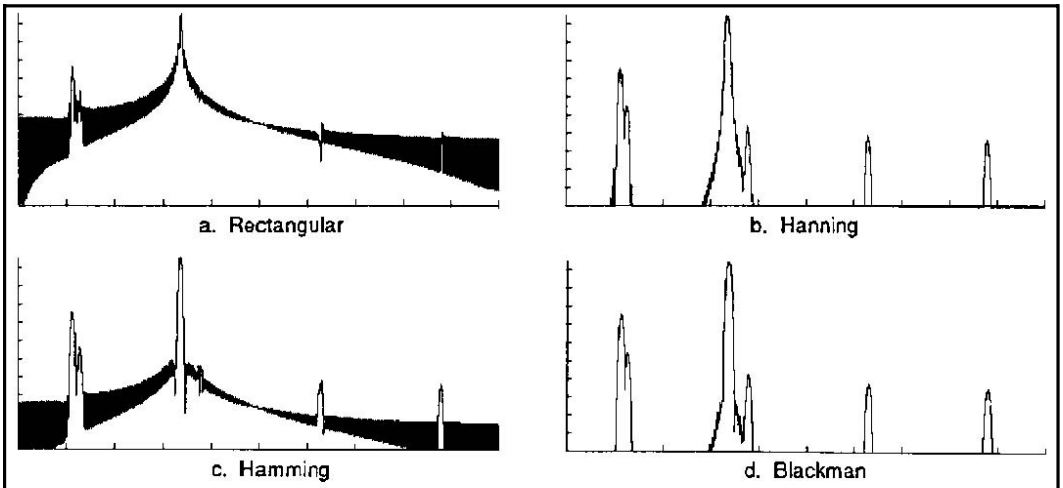

Figure 8. FFT with 1024 Points and 1/2 Zero-Filling

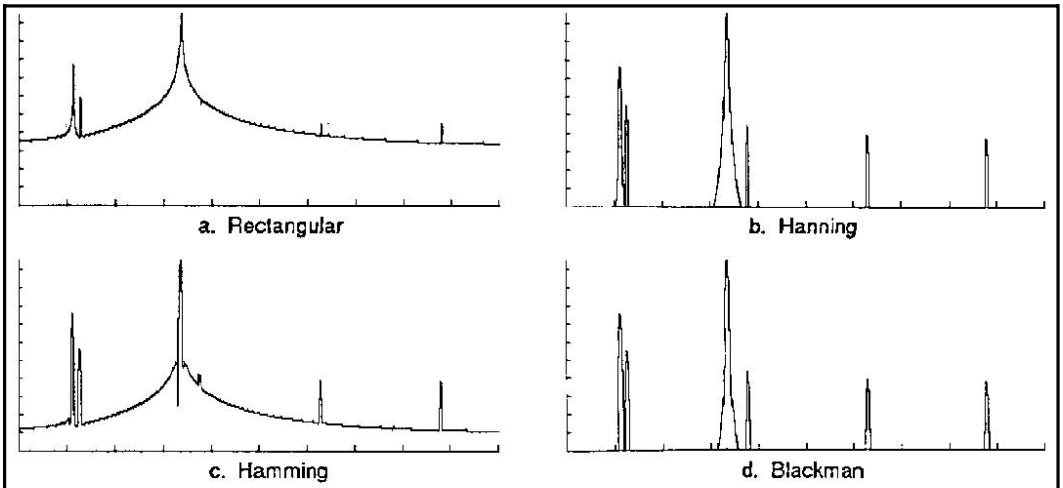

Figure 9. FFT with 1024 Points and No Zero-Filling

Figures 7a-7d and Figures 8a-8d are provided to show the resultant spectrums for 1024-point FFTs using 3/4 and 1/2 zero-filling, respectively. The effects of the pre-processing options are similar to those described in the previous paragraphs for the 512-point FFTs except the frequency resolution is doubled. Therefore, the bandwidth of each spectral component is reduced, which brings out the distinction between the fundamental of Component 3 and Component 2. Also, there is a distinction between the third harmonic of Component 3 and Component 1 in most of the windowing functions except the Rectangular window (Figures 7a & 8a).

Finally, Figures 9a-9d are 1024-point FFTs using no zero-filling. These spectrums best illustrate all of the different spectral components in the input data waveform. The Rectangular window in Figure 9a still hides the harmonics of Component 3 in the sidelobes of Component 1. Figures 9b and 9c best illustrate the difference in the sidelobes caused by the Hamming and Hanning windowing functions. The Hanning windowing function causes the sidelobes to fall off with a steep slope, whereas the Hamming window sidelobes are suppressed but do not drop as sharply from the large spectral components.

Summary

The utilization of various pre-processing options such as the number of data points, "zero filling," and windowing functions permit one to enhance the spectral interpretation of the sampled signal. However, one must be cognizant of the tradeoffs in terms of processing time, frequency resolution, and spectral bandwidth that these options proffer. Each option must be judiciously selected when performing a spectral analysis.

References

1) Ramirez, Robert W., *The FFT Fundamentals and Concepts*, Prentice-Hall Inc, Englewood Cliffs, N.J., 1985.

2) Cooley, P.M. and Tukey, J.W. "An Algorithm for the Machine Computation of Complex Fourier Series," *Mathematics of Computation*, vol 19, April 1965.

3) DeFatta, David J, et al., Digital Signal Processing: A System Design Approach, John Wiley & Sons, New York, 1988.

4) Higgins, Richard J , *Digital Signal Processing in VLSI*, Prentice-Hall Inc, Englewood Cliffs, N.J., 1990.

5) Harris, Frederic J, "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE*,
Vol 66, No. 1, Jan 1978.

6) Oppenheim, Alan V. and Schafer, Ronald W., *Digital Signal Processing*, Prentice-Hall Inc, Englewood Cliffs, N.J., 1975.

7) Fante, Ronald L., *Signal Analysis and Estimation*, John Wiley and Sons, New York, 1988.

Appendix

The following is a "C" listing for performing the FFT algorithm.

```
/*function fft() - performs a fast fourier transform
 * calling sequence:
 *    fft(real, imag, sum, points, power);
 * input parameters:
 *     float far *real - pointer to array of real data points
 *     float far *imag - pointer to array of imaginary data
 *     float sum - The sum of the windowing factors
 *   short points - Number of points in the transform
 *     short power - Power to raise 2 for the number of pts
 * output parameters:
 *     float far *real - pointer to array of magnitude values
 *     float far *imag - pointer to array of phase values
 * returns:
 *     none
 */
void
fft(float far *real, float far *imag, float sum,
   short points, short power)
   {
   short n2, j, l, i. i2, lb, nu2, n_pts1;
   register short k = 0, k2;
   float * pr_k, * pi_k, * pr_k2, * pi_k2;
   float arg, arg1, nu1, num_pts;
   float c, s;
   float tr, ti;
```

```
    n2 = points>> 1;
    num_pts = (float)points;
    nu1 = (float)power - 1.0f;;
    argl = (float)TWO_PI / num_pts:
    N_pts1 = points - 1;
    for (I = 0; I < power; 1++)
        {
      nu2 = (short)pow(2.0f, (double)nu1);
      for (k = 0; k < n_pts1; k += n2)
          {
          j = k / nu2;
        for (i = 0, ib = 0; i < power; i++)
           {
/* bit swap the data in j and put into ib */
           i2 = j >> 1;
           ib = (ib << 1) + j - (i2 << 1);
           j = i2:
          }
        arg = arg 1 * (float)ib;

/* use the big identity sin2 + cos2 = 1 */
        c = (float)cos((double)arg);
        s = (float)sqrt((double)1.0f - (double)(c * c));

   /* calculate real & imaginary components of transform */

        for (i = 0; i < n2; i++, k++)
           {
           k2 = k+ n2;
           pr_k2 = real + k2;
           pi_k2 = imag + k2;
           pr_k - real + k;
           pi_k - imag + k;
           tr = *pr_k2 * c + *pi_k2 * s;
           ti = *pi_k2 * c - *pr_k2 * s;
           *pr_k2 = *pr_k - tr;
           *pi_k2 = *pi_k - ti;
           *pr_k += tr;
           *pi_k += ti;
           }
          }
```

```
      k = 0;
      nu1 -= 1.0f;
      n2 >>= 1;
      }
    for (k = 0; k < points; k++)
      {
      j = k;           /* temp storage place */
      for (i = 0, ib = 0; i < power; i++)
         {
/* bit swap the data in j and put into ib */
        i2 = j >> 1;
        ib = (ib << 1) + j - (i2 << 1);
        j = i2;
        }
     if(ib > k)
        {
        swap (real + k, real + ib);
        swap (imag + k, inrag + ib);
        }
      }
   for (i = 0; i < points; I++)
      real[i] = 2.0f / sum * (float)sqrt((double)
       ((imag[i] * imag[i] + (real[i] * real[i])));
   }

/*function swap() - swaps two float in memory.
 * calling sequence:
 *     swap(x1, x2);

 *input parameters:
 *     float far *x1 - pointer to first variable
 *     float far *x2 - pointer to second variable

 *output parameters:
 *     *(float far *x1) - original x2 value
 *     *(float far *x2) - original x1 value
 *
 * returns:
 *     none
 *
 */
```

```
void
swap(float far *x1, float far *x2 )
    {
    float temp_x;

    temp_x = *x1;
    *x1 = *x2;
    *x2 = temp_x;
    }
```