The Fundamentals of FFT-Based Signal Analysis and Measurement

Michael Cerna and Audrey F. Harvey

Introduction

The Fast Fourier Transform (FFT) and the power spectrum are powerful tools for analyzing and measuring signals from plug-in data acquisition (DAQ) devices. For example, you can effectively acquire time-domain signals, measure the frequency content, and convert the results to real-world units and displays as shown on traditional benchtop spectrum and network analyzers. By using plug-in DAQ devices, you can build a lower cost measurement system and avoid the communication overhead of working with a stand-alone instrument. Plus, you have the flexibility of configuring your measurement processing to meet your needs.

To perform FFT-based measurement, however, you must understand the fundamental issues and computations involved. This application note serves the following purposes.

- Describes some of the basic signal analysis computations,
- Discusses antialiasing and acquisition front ends for FFT-based signal analysis,
- Explains how to use windows correctly,
- Explains some computations performed on the spectrum, and
- Shows you how to use FFT-based functions for network measurement.

The basic functions for FFT-based signal analysis are the FFT, the Power Spectrum, and the Cross Power Spectrum. Using these functions as building blocks, you can create additional measurement functions such as frequency response, impulse response, coherence, amplitude spectrum, and phase spectrum.

FFTs and the Power Spectrum are useful for measuring the frequency content of stationary or transient signals. FFTs produce the average frequency content of a signal over the entire time that the signal was acquired. For this reason, you should use FFTs for stationary signal analysis or in cases where you need only the average energy at each frequency line. To measure frequency information that is changing over time, use joint time-frequency functions such as the Gabor Spectrogram.

This application note also describes other issues critical to FFT-based measurement, such as the characteristics of the signal acquisition front end, the necessity of using windows, the effect of using windows on the measurement, and measuring noise versus discrete frequency components.

Basic Signal Analysis Computations

The basic computations for analyzing signals include converting from a two-sided power spectrum to a single-sided power spectrum, adjusting frequency resolution and graphing the spectrum, using the FFT, and converting power and amplitude into logarithmic units.

The power spectrum returns an array that contains the two-sided power spectrum of a time-domain signal. The array values are proportional to the amplitude squared of each frequency component making up the time-domain signal. A plot of the two-sided power spectrum shows negative and positive frequency components at a height

$$\frac{A_k^2}{4}$$

where A_k is the peak amplitude of the sinusoidal component at frequency k. The DC component has a height of A_0^2 where A_0 is the amplitude of the DC component in the signal.

Figure 1 shows the power spectrum result from a time-domain signal that consists of a 3 Vrms sine wave at 128 Hz, a 3 Vrms sine wave at 256 Hz, and a DC component of 2 VDC. A 3 Vrms sine wave has a peak voltage of $3.0 \bullet \sqrt{2}$ or about 4.2426 V. The power spectrum is computed from the basic FFT function. Refer to the Computations Using the FFT section later in this application note for an example this formula.

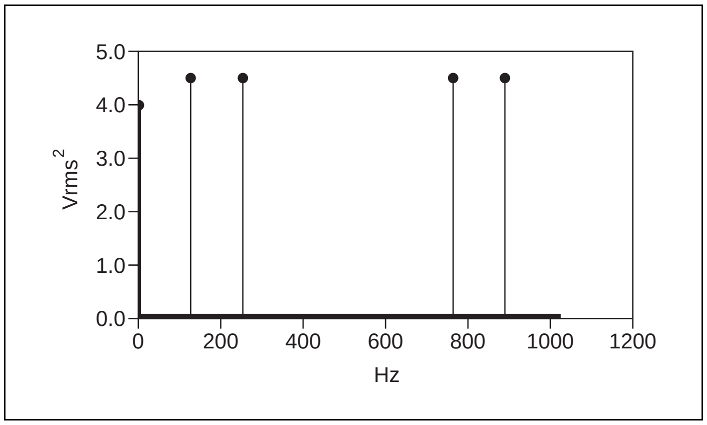


Figure 1. Two-Sided Power Spectrum of Signal

Converting from a Two-Sided Power Spectrum to a Single-Sided Power Spectrum

Most real-world frequency analysis instruments display only the positive half of the frequency spectrum because the spectrum of a real-world signal is symmetrical around DC. Thus, the negative frequency information is redundant. The two-sided results from the analysis functions include the positive half of the spectrum followed by the negative half of the spectrum, as shown in Figure 1.

In a two-sided spectrum, half the energy is displayed at the positive frequency, and half the energy is displayed at the negative frequency. Therefore, to convert from a two-sided spectrum to a single-sided spectrum, discard the second half of the array and multiply every point except for DC by two.

$$G_{AA}(i) = S_{AA}(i), \text{ i = 0 (DC)}$$

$$G_{AA}(i) = 2 \bullet S_{AA}(i), \text{ i = 1 to } \frac{N}{2} - 1$$

where $S_{AA}(i)$ is the two-sided power spectrum, $G_{AA}(i)$ is the single-sided power spectrum, and N is the length of the two-sided power spectrum. The remainder of the two-sided power spectrum S_{AA}

$$\left(\frac{N}{2} \text{ through } N-1\right)$$

is discarded.

The non-DC values in the single-sided spectrum are then at a height of

$$\frac{A_k^2}{2}$$

This is equivalent to

$$\left(\frac{A_k}{\sqrt{2}}\right)^2$$

where

$$\frac{A_k}{\sqrt{2}}$$

is the root mean square (rms) amplitude of the sinusoidal component at frequency k. Thus, the units of a power spectrum are often referred to as quantity squared rms, where quantity is the unit of the time-domain signal. For example, the single-sided power spectrum of a voltage waveform is in volts rms squared.

Figure 2 shows the single-sided spectrum of the signal whose two-sided spectrum Figure 1 shows.

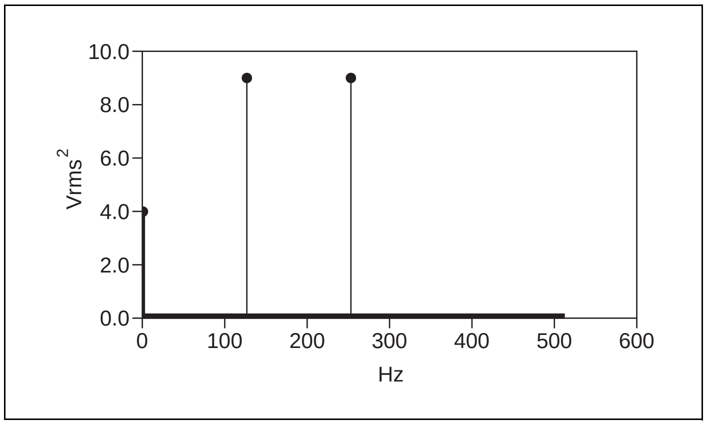


Figure 2. Single-Sided Power Spectrum of Signal in Figure 1

As you can see, the level of the non-DC frequency components are doubled compared to those in Figure 1. In addition, the spectrum stops at half the frequency of that in Figure 1.

Adjusting Frequency Resolution and Graphing the Spectrum

Figures 1 and 2 show power versus frequency for a time-domain signal. The frequency range and resolution on the x-axis of a spectrum plot depend on the sampling rate and the number of points acquired. The number of frequency points or lines in Figure 2 equals

$$\frac{N}{2}$$

where *N* is the number of points in the acquired time-domain signal. The first frequency line is at 0 Hz, that is, DC. The last frequency line is at

$$\frac{F_s}{2} - \frac{F_s}{N}$$

where F_s is the frequency at which the acquired time-domain signal was sampled. The frequency lines occur at Δf intervals where

$$\Delta f = \frac{F_s}{N}$$

Frequency lines also can be referred to as frequency bins or FFT bins because you can think of an FFT as a set of parallel filters of bandwidth Δf centered at each frequency increment from

$$\text{DC to } \frac{F_s}{2} - \frac{F_s}{N}$$

Alternatively you can compute Δf as

$$\Delta f = \frac{1}{N \bullet \Delta t}$$

where Δt is the sampling period. Thus $N \bullet \Delta t$ is the length of the time record that contains the acquired time-domain signal. The signal in Figures 1 and 2 contains 1,024 points sampled at 1.024 kHz to yield $\Delta f = 1$ Hz and a frequency range from DC to 511 Hz.

The computations for the frequency axis demonstrate that the sampling frequency determines the frequency range or bandwidth of the spectrum and that for a given sampling frequency, the number of points acquired in the time-domain signal record determine the resolution frequency. To increase the frequency resolution for a given frequency range, increase the number of points acquired at the same sampling frequency. For example, acquiring 2,048 points at 1.024 kHz would have yielded $\Delta f = 0.5$ Hz with frequency range 0 to 511.5 Hz. Alternatively, if the sampling rate had been 10.24 kHz with 1,024 points, Δf would have been 10 Hz with frequency range from 0 to 5.11 kHz.

Computations Using the FFT

The power spectrum shows power as the mean squared amplitude at each frequency line but includes no phase information. Because the power spectrum loses phase information, you may want to use the FFT to view both the frequency and the phase information of a signal.

The phase information the FFT yields is the phase relative to the start of the time-domain signal. For this reason, you must trigger from the same point in the signal to obtain consistent phase readings. A sine wave shows a phase of –90° at the sine wave frequency. A cosine shows a 0° phase. In many cases, your concern is the relative phases between components, or the phase difference between two signals acquired simultaneously. You can view the phase difference between two signals by using some of the advanced FFT functions. Refer to the *FFT-Based Network Measurement* section of this application note for descriptions of these functions.

The FFT returns a two-sided spectrum in complex form (real and imaginary parts), which you must scale and convert to polar form to obtain magnitude and phase. The frequency axis is identical to that of the two-sided power spectrum. The amplitude of the FFT is related to the number of points in the time-domain signal. Use the following equation to compute the amplitude and phase versus frequency from the FFT.

$$\text{Amplitude spectrum in quantity peak} = \frac{\text{Magnitude [FFT(A)]}}{N} = \frac{\sqrt{[\text{real}[\text{FFT(A)}]]^2 + [\text{imag}[\text{FFT(A)}]]^2}}{N}$$

$$\text{Phase spectrum in radians} = \text{Phase [FFT(A)]} = \text{arctangent}\left(\frac{\text{imag}[\text{FFT(A)}]}{\text{real}[\text{FFT(A)}]}\right)$$

where the arctangent function here returns values of phase between $-\pi$ and $+\pi$, a full range of 2π radians.

Using the rectangular to polar conversion function to convert the complex array

$$\frac{\text{FFT(A)}}{N}$$

to its magnitude (r) and phase (ø) is equivalent to using the preceding formulas.

The two-sided amplitude spectrum actually shows half the peak amplitude at the positive and negative frequencies. To convert to the single-sided form, multiply each frequency other than DC by two, and discard the second half of the array. The units of the single-sided amplitude spectrum are then in quantity peak and give the peak amplitude of each sinusoidal component making up the time-domain signal. For the single-sided phase spectrum, discard the second half of the array.

To view the amplitude spectrum in volts (or another quantity) rms, divide the non-DC components by the square root of two after converting the spectrum to the single-sided form. Because the non-DC components were multiplied by two to convert from two-sided to single-sided form, you can calculate the rms amplitude spectrum directly from the two-sided amplitude spectrum by multiplying the non-DC components by the square root of two and discarding the second half of the array. The following equations show the entire computation from a two-sided FFT to a single-sided amplitude spectrum.

$$\text{Amplitude spectrum in volts rms} = \sqrt{2} \bullet \frac{\text{Magnitude[FFT(A)]}}{N} \quad \text{for } i = 1 \text{ to } \frac{N}{2} - 1$$

$$= \frac{\text{Magnitude[FFT(A)]}}{N} \quad \text{for } i = 0 \text{ (DC)}$$

where *i* is the frequency line number (array index) of the FFT of A.

The magnitude in volts rms gives the rms voltage of each sinusoidal component of the time-domain signal.

To view the phase spectrum in degrees, use the following equation.

$$\text{Phase spectrum in degrees} = \frac{180}{\pi} \bullet \text{Phase FFT(A)}$$

The amplitude spectrum is closely related to the power spectrum. You can compute the single-sided power spectrum by squaring the single-sided rms amplitude spectrum. Conversely, you can compute the amplitude spectrum by taking the square root of the power spectrum. The two-sided power spectrum is actually computed from the FFT as follows.

$$\text{Power spectrum } S_{AA}(f) = \frac{\text{FFT(A)} \bullet \text{FFT*(A)}}{N}$$

where FFT*(A) denotes the complex conjugate of FFT(A). To form the complex conjugate, the imaginary part of FFT(A) is negated.

When using the FFT in LabVIEW, be aware that the speed of the power spectrum and the FFT computation depend on the number of points acquired. If *N* is a power of two, LabVIEW uses the efficient FFT algorithm. Otherwise, LabVIEW actually uses the discrete Fourier transform (DFT), which takes considerably longer. LabWindows/CVI requires that *N* be a factor of two and thus always uses the FFT. Typical benchtop instruments use FFTs of 1,024 and 2,048 points.

So far, you have looked at display units of volts peak, volts rms, and volts rms squared, which is equivalent to mean-square volts. In some spectrum displays, the rms qualifier is dropped for Vrms, in which case V implies Vrms, and V^2 implies Vrms^2, or mean-square volts.

Converting to Logarithmic Units

Most often, amplitude or power spectra are shown in the logarithmic unit decibels (dB). Using this unit of measure, it is easy to view wide dynamic ranges; that is, it is easy to see small signal components in the presence of large ones. The decibel is a unit of ratio and is computed as follows.

$$\text{dB} = 10\log_{10} P / P_r$$

where P is the measured power and P_r is the reference power.

Use the following equation to compute the ratio in decibels from amplitude values.

$$\text{dB} = 20\log_{10} A / A_r$$

where A is the measured amplitude and A_r is the reference amplitude.

When using amplitude or power as the amplitude-squared of the same signal, the resulting decibel level is exactly the same. Multiplying the decibel ratio by two is equivalent to having a squared ratio. Therefore, you obtain the same decibel level and display regardless of whether you use the amplitude or power spectrum.

As shown in the preceding equations for power and amplitude, you must supply a reference for a measure in decibels. This reference then corresponds to the 0 dB level. Several conventions are used. A common convention is to use the reference 1 Vrms for amplitude or 1 Vrms squared for power, yielding a unit in dBV or dBVrms. In this case, 1 Vrms corresponds to 0 dB. Another common form of dB is dBm, which corresponds to a reference of 1 mW into a load of 50 Ω for radio frequencies where 0 dB is 0.22 Vrms, or 600 Ω for audio frequencies where 0 dB is 0.78 Vrms.

Antialiasing and Acquisition Front Ends for FFT-Based Signal Analysis

FFT-based measurement requires digitization of a continuous signal. According to the Nyquist criterion, the sampling frequency, F_s, must be at least twice the maximum frequency component in the signal. If this criterion is violated, a phenomenon known as aliasing occurs. Figure 3 shows an adequately sampled signal and an undersampled signal. In the undersampled case, the result is an aliased signal that appears to be at a lower frequency than the actual signal.

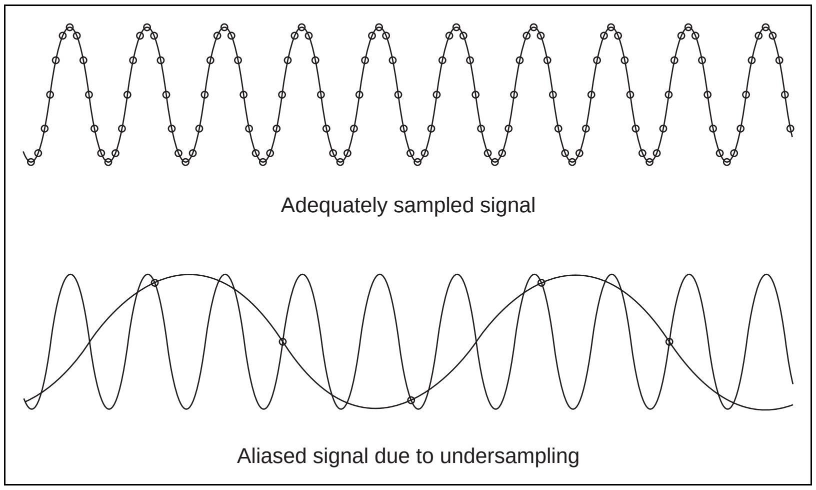

Adequately sampled signal
Aliased signal due to undersampling

Figure 3. Adequate and Inadequate Signal Sampling

When the Nyquist criterion is violated, frequency components above half the sampling frequency appear as frequency components below half the sampling frequency, resulting in an erroneous representation of the signal. For example, a component at frequency

$$\frac{F_s}{2} < f_0 < F_s$$

appears as the frequency $F_s - f_0$.

Figure 4 shows the alias frequencies that appear when the signal with real components at 25, 70, 160, and 510 Hz is sampled at 100 Hz. Alias frequencies appear at 10, 30, and 40 Hz.

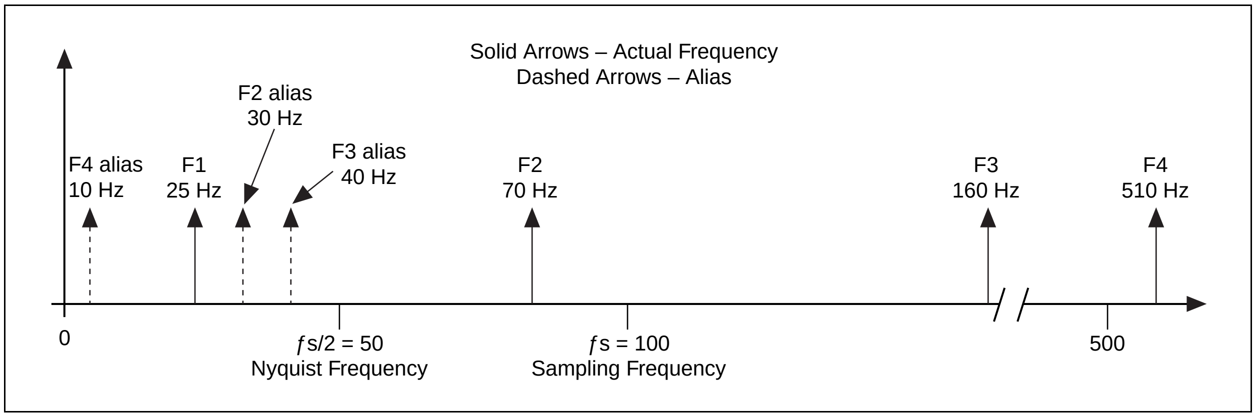


Figure 4. Alias Frequencies Resulting from Sampling a Signal at 100 Hz That Contains Frequency Components Greater than or Equal to 50 Hz

Before a signal is digitized, you can prevent aliasing by using antialiasing filters to attenuate the frequency components at and above half the sampling frequency to a level below the dynamic range of the analog-to-digital converter (ADC). For example, if the digitizer has a full-scale range of 80 dB, frequency components at and above half the sampling frequency must be attenuated to 80 dB below full scale.

These higher frequency components, do not interfere with the measurement. If you know that the frequency bandwidth of the signal being measured is lower than half the sampling frequency, you can choose not to use an antialiasing filter. Figure 5 shows the input frequency response of the National Instruments PCI-4450 Family dynamic signal acquisition boards, which have antialiasing filters. Note how an input signal at or above half the sampling frequency is severely attenuated.

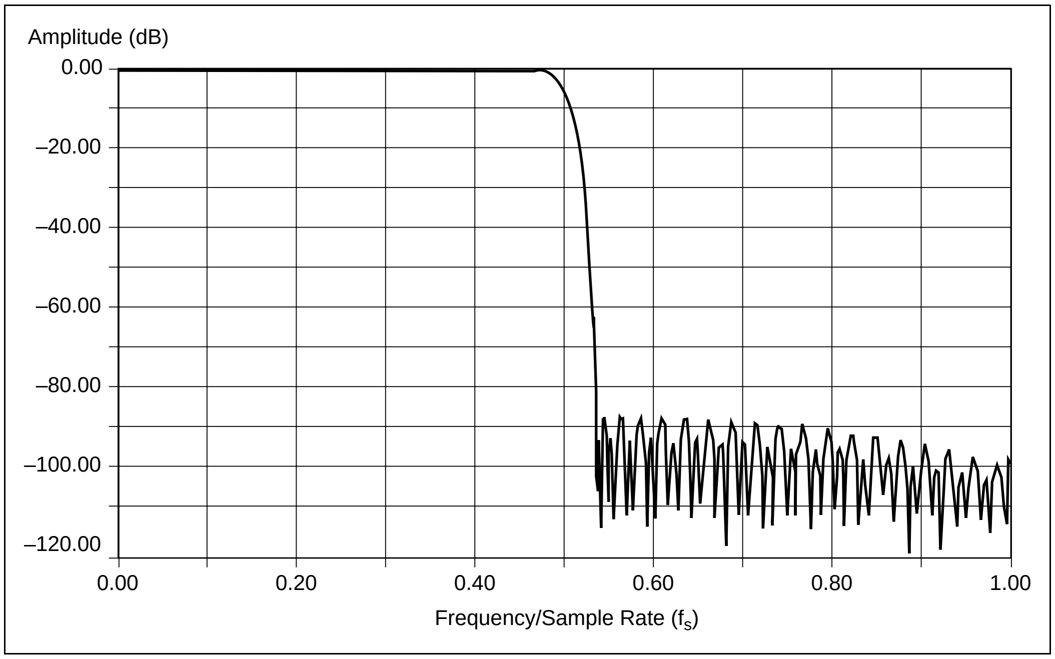


Figure 5. Bandwidth of PCI-4450 Family Input Versus Frequency, Normalized to Sampling Rate

Limitations of the Acquisition Front End

In addition to reducing frequency components greater than half the sampling frequency, the acquisition front end you use introduces some bandwidth limitations below half the sampling frequency. To eliminate signals at or above half of the sampling rate to less than the measurement range, antialiasing filters start to attenuate frequencies at some point below half the sampling rate. Because these filters attenuate the highest frequency portion of the spectrum, typically you want to limit the plot to the bandwidth you consider valid for the measurement.

For example, in the case of the PCI-4450 Family sample shown in Figure 5, amplitude flatness is maintained to within ±0.1 dB, at up to 0.464 of the sampling frequency to 20 kHz for all gain settings, +1 dB to 95 kHz, and then the input gain starts to attenuate. The –3 dB point (or half-power bandwidth) of the input occurs at 0.493 of the input spectrum. Therefore, instead of showing the input spectrum all the way out to half the sampling frequency, you may want to show only 0.464 of the input spectrum. To do this, multiply the number of points acquired by 0.464, respectively, to compute the number of frequency lines to display.

The characteristics of the signal acquisition front end affect the measurement. The National Instruments PCI-4450 Family dynamic signal acquisition boards and the NI 4551 and NI 4552 dynamic signal analyzers are excellent acquisition front ends for performing FFT-based signal analysis measurements. These boards use delta-sigma modulation technology, which yields excellent amplitude flatness, high-performance antialiasing filters, and wide dynamic range as shown in Figure 5. The input channels are also simultaneously sampled for good multichannel measurement performance.

At a sampling frequency of 51.2 kHz, these boards can perform frequency measurements in the range of DC to 23.75 kHz. Amplitude flatness is ±0.1 dB maximum from DC to 23.75 kHz. Refer to the *PCI-4451/4452/4453/4454 User Manual* for more information about these boards.

Calculating the Measurement Bandwidth or Number of Lines for a Given Sampling Frequency

The dynamic signal acquisition boards have antialiasing filters built into the digitizing process. In addition, the cutoff filter frequency scales with the sampling rate to meet the Nyquist criterion as shown in Figure 5. The fast cutoff of the antialiasing filters on these boards means that the number of useful frequency lines in a 1,024-point FFT-based spectrum is 475 lines for ±0.1 dB amplitude flatness.

To calculate the measurement bandwidth for a given sampling frequency, multiply the sampling frequency by 0.464 for the ±0.1 dB flatness. Also, the larger the FFT, the larger the number of frequency lines. A 2,048-point FFT yields twice the number of lines listed above. Contrast this with typical benchtop instruments, which have 400 or 800 useful lines for a 1,024- point or 2,048-point FFT, respectively.

Dynamic Range Specifications

The signal-to-noise ratio (SNR) of the PCI-4450 Family boards is 93 dB. SNR is defined as

$$\mathrm{SNR} = 10\log_{10}\left(\frac{V_s^2}{V_n^2}\right)\mathrm{dB}$$

where V_s and V_n are the rms amplitudes of the signal and noise, respectively. A bandwidth is usually given for SNR. In this case, the bandwidth is the frequency range of the board input, which is related to the sampling rate as shown in Figure 5. The 93 dB SNR means that you can detect the frequency components of a signal that is as small as 93 dB below the full-scale range of the board. This is possible because the total input noise level caused by the acquisition front end is 93 dB below the full-scale input range of the board.

If the signal you monitor is a narrowband signal (that is, the signal energy is concentrated in a narrow band of frequencies), you are able to detect an even lower level signal than –93 dB. This is possible because the noise energy of the board is spread out over the entire input frequency range. Refer to the *Computing Noise Level and Power Spectral Density* section later in this application note for more information about narrowband versus broadband levels.

The spurious-free dynamic range of the dynamic signal acquisition boards is 95 dB. Besides input noise, the acquisition front end may introduce spurious frequencies into a measured spectrum because of harmonic or intermodulation distortion, among other things. This 95 dB level indicates that any such spurious frequencies are at least 95 dB below the full-scale input range of the board.

The signal-to-total-harmonic-distortion (THD)-plus-noise ratio, which excludes intermodulation distortion, is 90 dB from 0 to 20 kHz. THD is a measure of the amount of distortion introduced into a signal because of the nonlinear behavior of the acquisition front end. This harmonic distortion shows up as harmonic energy added to the spectrum for each of the discrete frequency components present in the input signal.

The wide dynamic range specifications of these boards is largely due to the 16-bit resolution ADCs. Figure 6 shows a typical spectrum plot of the PCI-4450 Family dynamic range with a full-scale 997 Hz signal applied. You can see that the harmonics of the 997 Hz input signal, the noise floor, and any other spurious frequencies are below 95 dB. In contrast, dynamic range specifications for benchtop instruments typically range from 70 dB to 80 dB using 12-bit and 13-bit ADC technology.

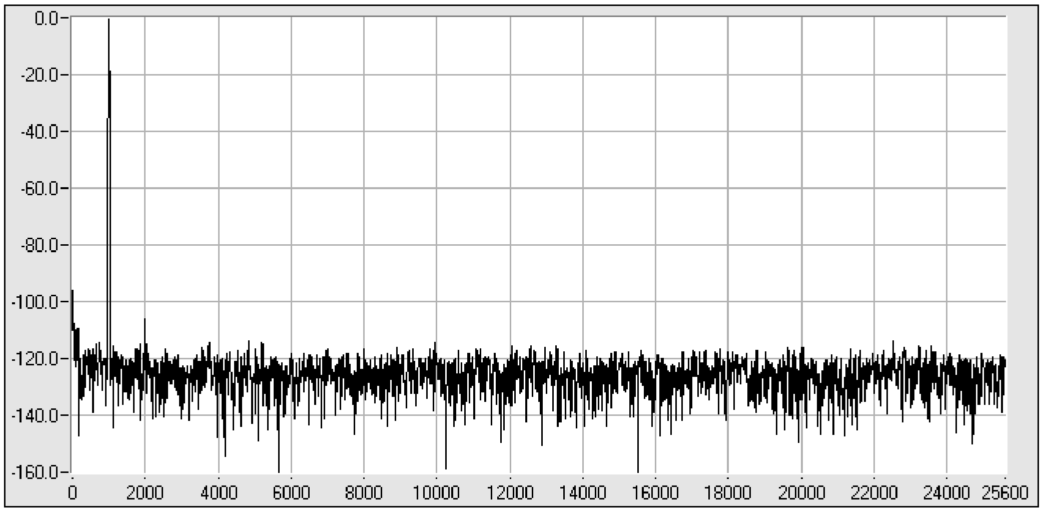


Figure 6. PCI-4450 Family Spectrum Plot with 997 Hz Input at Full Scale (Full Scale = 0 dB)

Using Windows Correctly

As mentioned in the Introduction, using windows correctly is critical to FFT-based measurement. This section describes the problem of spectral leakage, the characteristics of windows, some strategies for choosing windows, and the importance of scaling windows.

Spectral Leakage

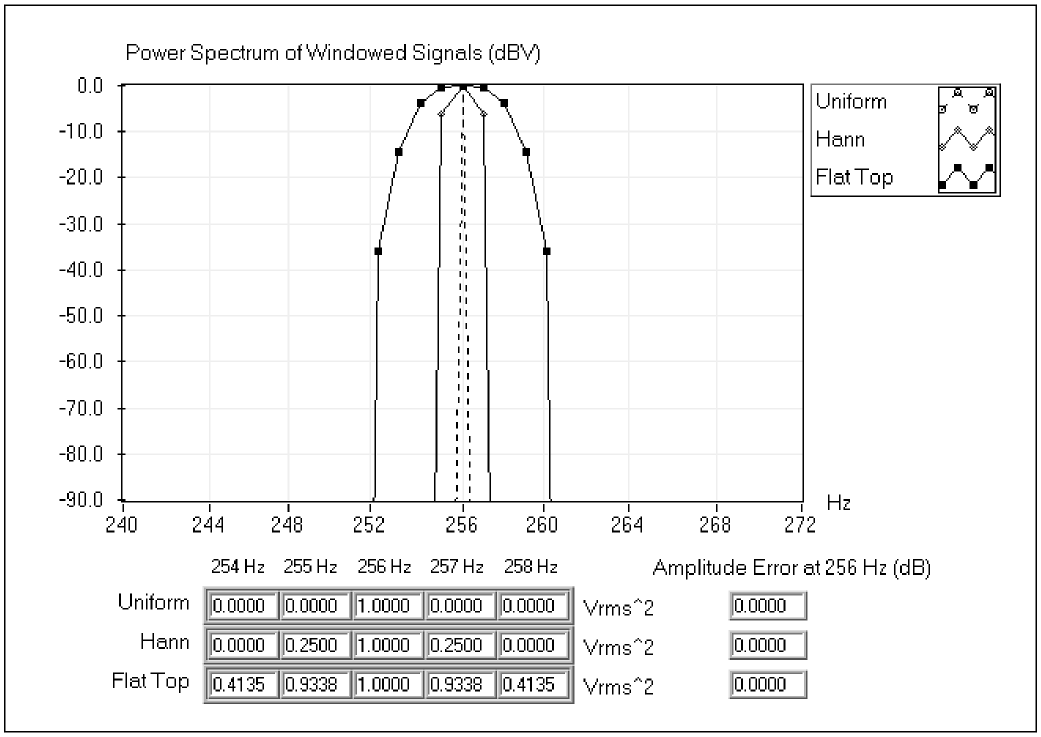
For an accurate spectral measurement, it is not sufficient to use proper signal acquisition techniques to have a nicely scaled, single-sided spectrum. You might encounter spectral leakage. Spectral leakage is the result of an assumption in the FFT algorithm that the time record is exactly repeated throughout all time and that signals contained in a time record are thus periodic at intervals that correspond to the length of the time record. If the time record has a nonintegral number of cycles, this assumption is violated and spectral leakage occurs. Another way of looking at this case is that the nonintegral cycle frequency component of the signal does not correspond exactly to one of the spectrum frequency lines.

There are only two cases in which you can guarantee that an integral number of cycles are always acquired. One case is if you are sampling synchronously with respect to the signal you measure and can therefore deliberately take an integral number of cycles.

Another case is if you capture a transient signal that fits entirely into the time record. In most cases, however, you measure an unknown signal that is stationary; that is, the signal is present before, during, and after the acquisition. In this case, you cannot guarantee that you are sampling an integral number of cycles. Spectral leakage distorts the measurement in such a way that energy from a given frequency component is spread over adjacent frequency lines or bins. You can use windows to minimize the effects of performing an FFT over a nonintegral number of cycles.

Figure 7 shows the effects of three different windows — none (Uniform), Hanning (also commonly known as Hann), and Flat Top — when an integral number of cycles have been acquired, in this figure, 256 cycles in a 1,024-point record. Notice that the windows have a main lobe around the frequency of interest. This main lobe is a frequency domain characteristic of windows. The Uniform window has the narrowest lobe, and the Hann and Flat Top windows introduce some spreading. The Flat Top window has a broader main lobe than the others. For an integral number of cycles, all windows yield the same peak amplitude reading and have excellent amplitude accuracy.

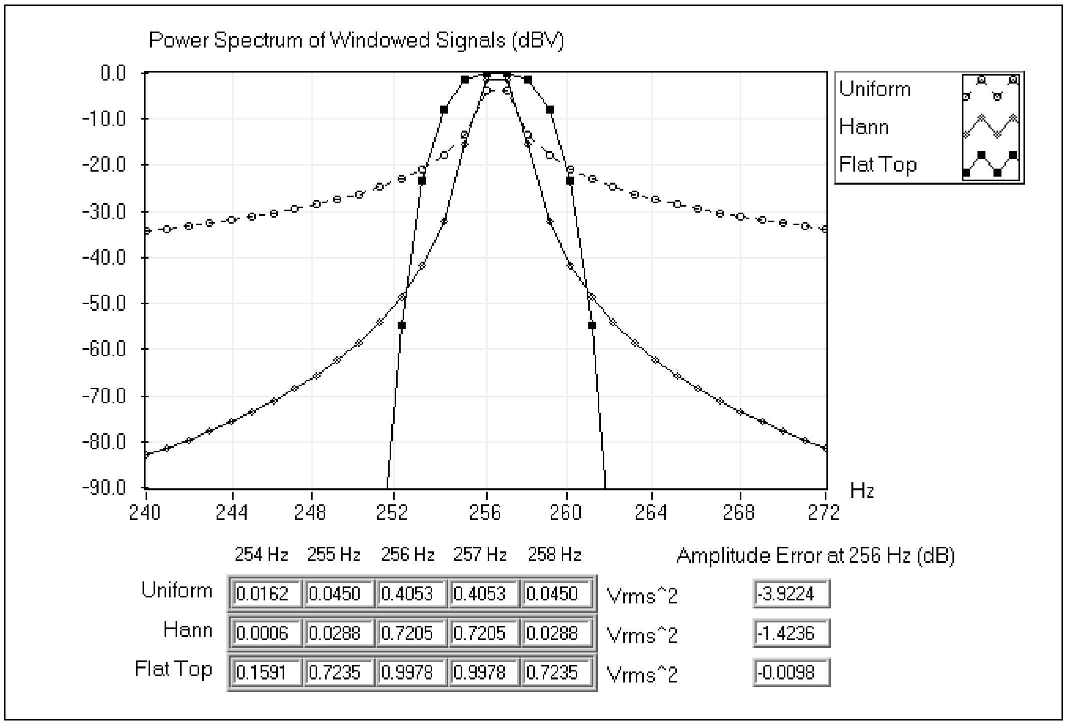
Figure 7 also shows the values at frequency lines of 254 Hz through 258 Hz for each window. The amplitude error at 256 Hz is 0 dB for each window. The graph shows the spectrum values between 240 and 272 Hz. The actual values in the resulting spectrum array for each window at 254 through 258 Hz are shown below the graph. Δf is 1 Hz.


Power Spectrum of Windowed Signals (dBV)
Uniform
Hann
Flat Top
Hz
Amplitude Error at 256 Hz (dB)

	254 Hz	255 Hz	256 Hz	257 Hz	258 Hz		Amplitude Error at 256 Hz (dB)
Uniform	0.0000	0.0000	1.0000	0.0000	0.0000	Vrms^2	0.0000
Hann	0.0000	0.2500	1.0000	0.2500	0.0000	Vrms^2	0.0000
Flat Top	0.4135	0.9338	1.0000	0.9338	0.4135	Vrms^2	0.0000

Figure 7. Power Spectrum of 1 Vrms Signal at 256 Hz with Uniform, Hann, and Flat Top Windows

Figure 8 shows the leakage effects when you acquire 256.5 cycles. Notice that at a nonintegral number of cycles, the Hann and Flat Top windows introduce much less spectral leakage than the Uniform window. Also, the amplitude error is better with the Hann and Flat Top windows. The Flat Top window demonstrates very good amplitude accuracy but also has a wider spread and higher side lobes than the Hann window.


Power Spectrum of Windowed Signals (dBV)
Uniform
Hann
Flat Top
Hz
Amplitude Error at 256 Hz (dB)

	254 Hz	255 Hz	256 Hz	257 Hz	258 Hz		Amplitude Error at 256 Hz (dB)
Uniform	0.0162	0.0450	0.4053	0.4053	0.0450	Vrms^2	-3.9224
Hann	0.0006	0.0288	0.7205	0.7205	0.0288	Vrms^2	-1.4236
Flat Top	0.1591	0.7235	0.9978	0.9978	0.7235	Vrms^2	-0.0098

Figure 8. Power Spectrum of 1 Vrms Signal at 256.5 Hz with Uniform, Hann, and Flat Top Windows

In addition to causing amplitude accuracy errors, spectral leakage can obscure adjacent frequency peaks. Figure 9 shows the spectrum for two close frequency components when no window is used and when a Hann window is used.

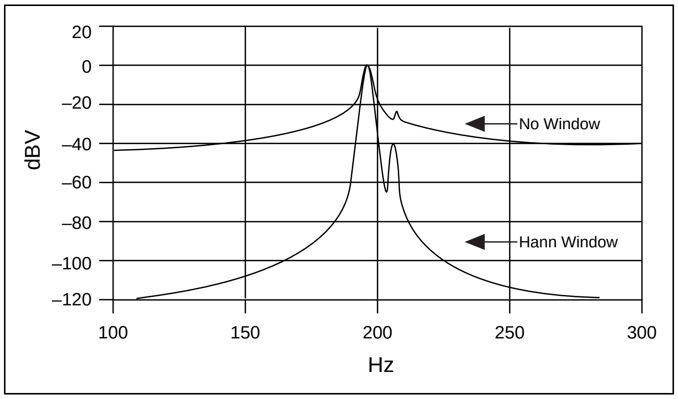


Figure 9. Spectral Leakage Obscuring Adjacent Frequency Components

Window Characteristics

To understand how a given window affects the frequency spectrum, you need to understand more about the frequency characteristics of windows. The windowing of the input data is equivalent to convolving the spectrum of the original signal with the spectrum of the window as shown in Figure 10. Even if you use no window, the signal is convolved with a rectangular-shaped window of uniform height, by the nature of taking a snapshot in time of the input signal. This convolution has a sine function characteristic spectrum. For this reason, no window is often called the Uniform or Rectangular window because there is still a windowing effect.

An actual plot of a window shows that the frequency characteristic of a window is a continuous spectrum with a main lobe and several side lobes. The main lobe is centered at each frequency component of the time-domain signal, and the side lobes approach zero at

$$\Delta f = \frac{F_s}{N} \text{i}$$

intervals on each side of the main lobe.

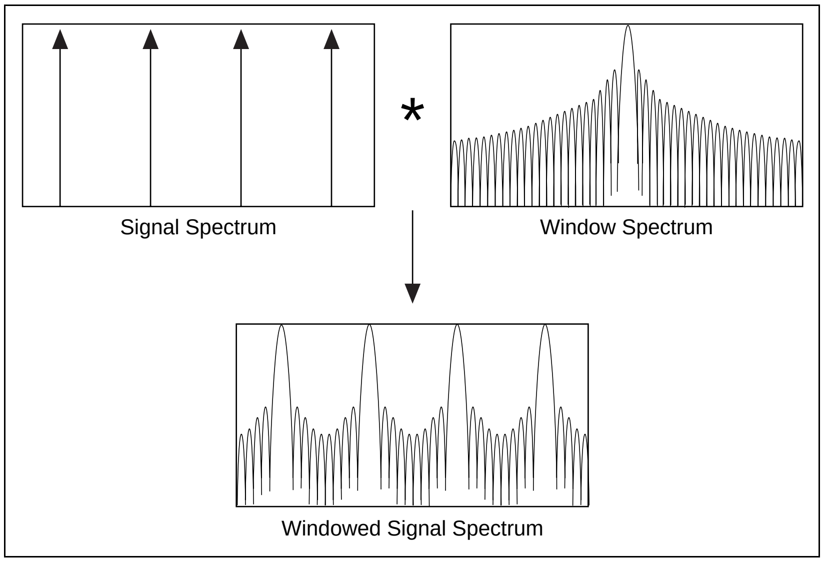


Figure 10. Frequency Characteristics of a Windowed Spectrum

An FFT produces a discrete frequency spectrum. The continuous, periodic frequency spectrum is sampled by the FFT, just as the time-domain signal was sampled by the ADC. What appears in each frequency line of the FFT is the value of the continuous convolved spectrum at each FFT frequency line. This is sometimes referred to as the picket-fence effect because the FFT result is analogous to viewing the continuous windowed spectrum through a picket fence with slits at intervals that corresponds to the frequency lines.

If the frequency components of the original signal match a frequency line exactly, as is the case when you acquire an integral number of cycles, you see only the main lobe of the spectrum. Side lobes do not appear because the spectrum of the window approaches zero at Δf intervals on either side of the main lobe. Figure 7 illustrates this case.

If a time record does not contain an integral number of cycles, the continuous spectrum of the window is shifted from the main lobe center at a fraction of Δf that corresponds to the difference between the frequency component and the FFT line frequencies. This shift causes the side lobes to appear in the spectrum. In addition, there is some amplitude error at the frequency peak, as shown in Figure 8, because the main lobe is sampled off center (the spectrum is smeared).

Figure 11 shows the frequency spectrum characteristics of a window in more detail. The side lobe characteristics of the window directly affect the extent to which adjacent frequency components bias (leak into) adjacent frequency bins. The side lobe response of a strong sinusoidal signal can overpower the main lobe response of a nearby weak sinusoidal signal.

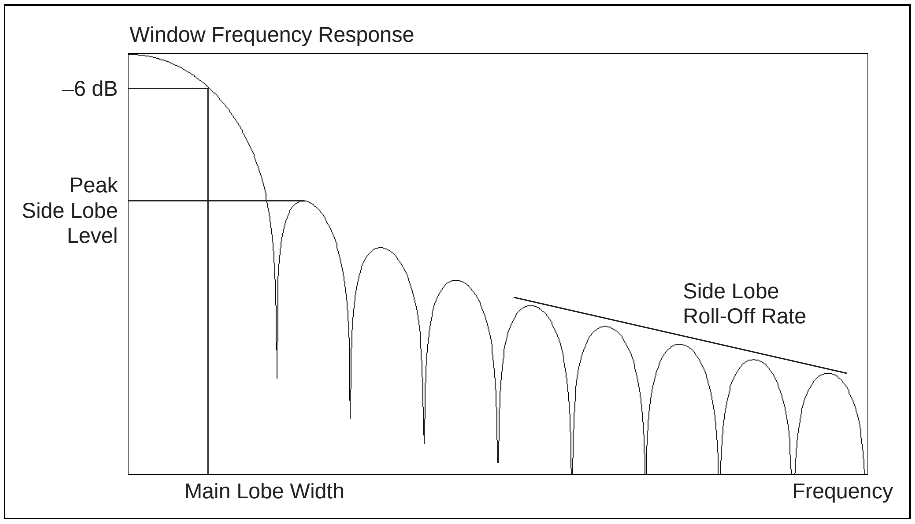


Figure 11. Frequency Response of a Window

Another important characteristic of window spectra is main lobe width. The frequency resolution of the windowed signal is limited by the width of the main lobe of the window spectrum. Therefore, the ability to distinguish two closely spaced frequency components increases as the main lobe of the window narrows. As the main lobe narrows and spectral resolution improves, the window energy spreads into its side lobes, and spectral leakage worsens. In general, then, there is a trade off between leakage suppression and spectral resolution.

Defining Window Characteristics

To simplify choosing a window, you need to define various characteristics so that you can make comparisons between windows. Figure 11 shows the spectrum of a typical window. To characterize the main lobe shape, the –3 dB and –6 dB main lobe width are defined to be the width of the main lobe (in FFT bins or frequency lines) where the window response becomes 0.707 (–3 dB) and 0.5 (–6 dB), respectively, of the main lobe peak gain.

To characterize the side lobes of the window, the maximum side lobe level and side lobe roll-off rate are defined. The maximum side lobe level is the level in decibels relative to the main lobe peak gain, of the maximum side lobe. The side lobe roll-off rate is the asymptotic decay rate, in decibels per decade of frequency, of the peaks of the side lobes. Table 1 lists the characteristics of several window functions and their effects on spectral leakage and resolution.

Table 1. Characteristics of Window Functions

Window	–3 dB Main Lobe Width (bins)	–6 dB Main Lobe Width (bins)	Maximum Side Lobe Level (dB)	Side Lobe Roll-Off Rate (dB/decade)
Uniform (None)	0.88	1.21	-13	20
Hanning (Hann)	1.44	2.00	-32	60
Hamming	1.30	1.81	-43	20
Blackman-Harris	1.62	2.27	-71	20
Exact Blackman	1.52	2.13	-67	20
Blackman	1.68	2.35	-58	60
Flat Top	2.94	3.56	-44	20

Strategies for Choosing Windows

Each window has its own characteristics, and different windows are used for different applications. To choose a spectral window, you must guess the signal frequency content. If the signal contains strong interfering frequency components distant from the frequency of interest, choose a window with a high side lobe roll-off rate. If there are strong interfering signals near the frequency of interest, choose a window with a low maximum side lobe level.

If the frequency of interest contains two or more signals very near to each other, spectral resolution is important. In this case, it is best to choose a window with a very narrow main lobe. If the amplitude accuracy of a single frequency component is more important than the exact location of the component in a given frequency bin, choose a window with a wide main lobe. If the signal spectrum is rather flat or broadband in frequency content, use the Uniform window (no window). In general, the Hann window is satisfactory in 95% of cases. It has good frequency resolution and reduced spectral leakage.

The Flat Top window has good amplitude accuracy, but because it has a wide main lobe, it has poor frequency resolution and more spectral leakage. The Flat Top window has a lower maximum side lobe level than the Hann window, but the Hann window has a faster roll off-rate. If you do not know the nature of the signal but you want to apply a window, start with the Hann window. Figures 7 and 8 contrast the characteristics of the Uniform, Hann, and Flat Top Windows windows.

If you are analyzing transient signals such as impact and response signals, it is better not to use the spectral windows because these windows attenuate important information at the beginning of the sample block. Instead, use the Force and Exponential windows. A Force window is useful in analyzing shock stimuli because it removes stray signals at the end of the signal. The Exponential window is useful for analyzing transient response signals because it damps the end of the signal, ensuring that the signal fully decays by the end of the sample block.

Selecting a window function is not a simple task. In fact, there is no universal approach for doing so. However, Table 2 can help you in your initial choice. Always compare the performance of different window functions to find the best one for the application. Refer to the references at the end of this application note for more information about windows.

Table 2. Initial Window Choice Based on Signal Content

Signal Content	Window
Sine wave or combination of sine waves	Hann
Sine wave (amplitude accuracy is important)	Flat Top
Narrowband random signal (vibration data)	Hann
Broadband random (white noise)	Uniform
Closely spaced sine waves	Uniform, Hamming
Excitation signals (Hammer blow)	Force
Response signals	Exponential
Unknown content	Hann

Scaling Windows

Windows are useful in reducing spectral leakage when using the FFT for spectral analysis. However, because windows are multiplied with the acquired time-domain signal, they introduce distortion effects of their own. The windows change the overall amplitude of the signal. The windows used to produce the plots in Figures 7 and 8 were scaled by dividing the windowed array by the coherent gain of the window. As a result, each window yields the same spectrum amplitude result within its accuracy constraints.

You can think of an FFT as a set of parallel filters, each Δf in bandwidth. Because of the spreading effect of a window, each window increases the effective bandwidth of an FFT bin by an amount known as the equivalent noise-power bandwidth of the window. The power of a given frequency peak is computed by adding the adjacent frequency bins around a peak and is inflated by the bandwidth of the window. You must take this inflation into account when you perform computations based on the spectrum. Refer to the Computations on the Spectrum section for sample computations.

Table 3 lists the scaling factor (or coherent gain), the noise power bandwidth, and the worst-case peak amplitude accuracy caused by off-center components for several popular windows.

Table 3. Correction Factors and Worst-Case Amplitude Errors for Windows

Window	Scaling Factor (Coherent Gain)	Noise Power Bandwidth	Worst-Case Amplitude Error (dB)
Uniform (none)	1.00	1.00	3.92
Hann	0.50	1.50	1.42
Hamming	0.54	1.36	1.75
Blackman-Harris	0.42	1.71	1.13
Exact Blackman	0.43	1.69	1.15
Blackman	0.42	1.73	1.10
Flat Top	0.22	3.77	< 0.01

Computations on the Spectrum

When you have the amplitude or power spectrum, you can compute several useful characteristics of the input signal, such as power and frequency, noise level, and power spectral density.

Estimating Power and Frequency

The preceding windowing examples demonstrate that if you have a frequency component in between two frequency lines, it appears as energy spread among adjacent frequency lines with reduced amplitude. The actual peak is between the two frequency lines. In Figure 8, the amplitude error at 256.5 Hz is due to the fact that the window is sampled at ±0.5 Hz around the center of its main lobe rather than at the center where the amplitude error would be 0. This is the picket-fence effect explained in the Window Characteristics section of this application note.

You can estimate the actual frequency of a discrete frequency component to a greater resolution than the Δf given by the FFT by performing a weighted average of the frequencies around a detected peak in the power spectrum.

$$\text{Estimated Frequency} = \frac{\sum_{i=j-3}^{j+3} (\text{Power}(i) \bullet i \bullet \Delta f)}{\sum_{i=i-3}^{j+3} \text{Power}(i)}$$

where *j* is the array index of the apparent peak of the frequency of interest and

$$\Delta f = \frac{F_s}{N}$$

The span *j* ±3 is reasonable because it represents a spread wider than the main lobes of the windows listed in Table 3.

Similarly, you can estimate the power in Vrms^2 of a given peak discrete frequency component by summing the power in the bins around the peak (computing the area under the peak)

$$\text{Estimated Power} = \frac{\sum_{i=j-3}^{j+3} \text{Power}(i)}{\text{noise power bandwidth of window}}$$

Notice that this method is valid only for a spectrum made up of discrete frequency components. It is not valid for a continuous spectrum. Also, if two or more frequency peaks are within six lines of each other, they contribute to inflating the estimated powers and skewing the actual frequencies. You can reduce this effect by decreasing the number of lines spanned by the preceding computations. If two peaks are that close, they are probably already interfering with one another because of spectral leakage.

Similarly, if you want the total power in a given frequency range, sum the power in each bin included in the frequency range and divide by the noise power bandwidth of the windows.

Computing Noise Level and Power Spectral Density

The measurement of noise levels depends on the bandwidth of the measurement. When looking at the noise floor of a power spectrum, you are looking at the narrowband noise level in each FFT bin. Thus, the noise floor of a given power spectrum depends on the Δf of the spectrum, which is in turn controlled by the sampling rate and number of points. In other words, the noise level at each frequency line reads as if it were measured through a Δf Hz filter centered at that frequency line. Therefore, for a given sampling rate, doubling the number of points acquired reduces the noise power that appears in each bin by 3 dB. Discrete frequency components theoretically have zero bandwidth and therefore do not scale with the number of points or frequency range of the FFT.

To compute the SNR, compare the peak power in the frequencies of interest to the broadband noise level. Compute the broadband noise level in Vrms^2 by summing all the power spectrum bins, excluding any peaks and the DC component, and dividing the sum by the equivalent noise bandwidth of the window. For example, in Figure 6 the noise floor appears to be more than 120 dB below full scale, even though the PCI-4450 Family dynamic range is only 93 dB. If you were to sum all the bins, excluding DC, and any harmonic or other peak components and divide by the noise power bandwidth of the window you used, the noise power level compared to full scale would be around –93 dB from full scale.

Because of noise-level scaling with Δf, spectra for noise measurement are often displayed in a normalized format called power or amplitude spectral density. This normalizes the power or amplitude spectrum to the spectrum that would be measured by a 1 Hz-wide square filter, a convention for noise-level measurements. The level at each frequency line then reads as if it were measured through a 1 Hz filter centered at that frequency line.

Power spectral density is computed as

$$\text{Power spectral density} = \frac{\text{Power Spectrum in } V_{rms}{}^2}{\Delta f \times \text{Noise Power Bandwidth of Window}}$$

The units are then in $\frac{V_{rms}{}^2}{\mathrm{Hz}}$ or $\frac{V^2}{\mathrm{Hz}}$

Amplitude spectral density is computed as:

$$\text{Amplitude Spectral Density} = \frac{\text{Amplitude Spectrum in } V_{rms}}{\sqrt{\Delta f \times \text{Noise Power Bandwidth of Window}}}$$

The units are then in $\frac{V_{rms}}{\sqrt{\mathrm{Hz}}}$ or $\frac{V}{\sqrt{\mathrm{Hz}}}$.

The spectral density format is appropriate for random or noise signals but inappropriate for discrete frequency components because the latter theoretically have zero bandwidth.

FFT-Based Network Measurement

When you understand how to handle computations with the FFT and power spectra, and you understand the influence of windows on the spectrum, you can compute several FFT-based functions that are extremely useful for network analysis. These include the transfer, impulse, and coherence functions. Refer to the Frequency Response and Network Analysis section of this application note for more information about these functions. Refer to the Signal Sources for Frequency Response Measurement section for more information about Chirp signals and broadband noise signals.

Cross Power Spectrum

One additional building block is the cross power spectrum. The cross power spectrum is not typically used as a direct measurement but is an important building block for other measurements.

The two-sided cross power spectrum of two time-domain signals A and B is computed as

$$\text{Cross Power Spectrum } S_{AB}(f) = \frac{\text{FFT(B)} \times \text{FFT*(A)}}{N^2}$$

The cross power spectrum is in two-sided complex form. To convert to magnitude and phase, use the Rectangular-To-Polar conversion function. To convert to a single-sided form, use the same method described in the Converting from a Two-Sided Power Spectrum to a Single-Sided Power Spectrum section of this application note. The units of the single-sided form are in volts (or other quantity) rms squared.

The power spectrum is equivalent to the cross power spectrum when signals A and B are the same signal. Therefore, the power spectrum is often referred to as the auto power spectrum or the auto spectrum. The single-sided cross power spectrum yields the product of the rms amplitudes of the two signals, A and B, and the phase difference between the two signals.

When you know how to use these basic blocks, you can compute other useful functions, such as the Frequency Response function.

Frequency Response and Network Analysis

Three useful functions for characterizing the frequency response of a network are the transfer, impulse response, and coherence functions.

The frequency response of a network is measured by applying a stimulus to the network as shown in Figure 12 and computing the transfer function from the stimulus and response signals.

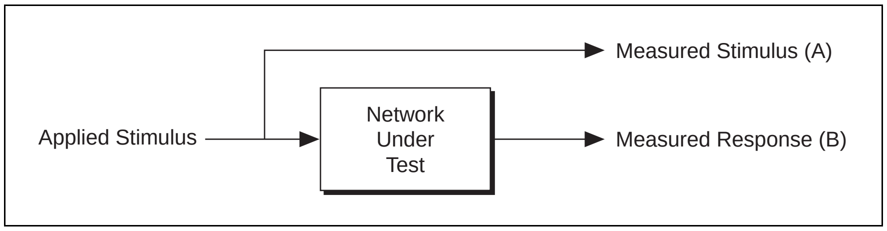


Figure 12. Configuration for Network Analysis

Transfer Function

The transfer function gives the gain and phase versus frequency of a network and is typically computed as

$$\text{Transfer Function H(f)} = \frac{\text{Cross Power Spectrum (Stimulus, Response)}}{\text{Power Spectrum (Stimulus)}} = \frac{S_{AB}(f)}{S_{AA}(f)}$$

where A is the stimulus signal and B is the response signal.

The transfer function is in two-sided complex form. To convert to the frequency response gain (magnitude) and the frequency response phase, use the Rectangular-To-Polar conversion function. To convert to single-sided form, simply discard the second half of the array.

You may want to take several transfer function readings and then average them. To do so, average the cross power spectrum, S_{AB}(f), by summing it in the complex form then dividing by the number of averages, before converting it to magnitude and phase, and so forth. The power spectrum, S_{AA}(f), is already in real form and is averaged normally.

Impulse Response Function

The impulse response function of a network is the time-domain representation of the transfer function of the network. It is the output time-domain signal generated when an impulse is applied to the input at time t = 0.

To compute the impulse response of the network, take the inverse FFT of the two-sided complex transfer function as described in the *Transfer Function* section of this application note.

$$\text{Impulse Response (f)} = \text{Inverse FFT (Transfer Function H(f))} = \text{Inverse FFT}\left(\frac{S_{AB}(f)}{S_{AA}(f)}\right)$$

The result is a time-domain function. To average multiple readings, take the inverse FFT of the averaged transfer function.

Coherence Function

The coherence function is often used in conjunction with the transfer function as an indication of the quality of the transfer function measurement and indicates how much of the response energy is correlated to the stimulus energy. If there is another signal present in the response, either from excessive noise or from another signal, the quality of the network response measurement is poor. You can use the coherence function to identify both excessive noise and causality, that is, identify which of the multiple signal sources are contributing to the response signal. The coherence function is computed as

$$\text{Coherence Function (f)} = \frac{[\text{Magnitude}(\text{Averaged } S_{AB}(f))]^2}{\text{Averaged } S_{AA}(f) \bullet \text{Averaged } S_{BB}(f)}$$

The result is a value between zero and one versus frequency. A zero for a given frequency line indicates no correlation between the response and the stimulus signal. A one for a given frequency line indicates that the response energy is 100 percent due to the stimulus signal; in other words, there is no interference at that frequency.

For a valid result, the coherence function requires an average of two or more readings of the stimulus and response signals. For only one reading, it registers unity at all frequencies. To average the cross power spectrum, S_{AB}(f), average it in the complex form then convert to magnitude and phase as described in the Transfer Function section of this application note. The auto power spectra, S_{AA}(f) and S_{BB}(f), are already in real form, and you average them normally.

Signal Sources for Frequency Response Measurements

To achieve a good frequency response measurement, significant stimulus energy must be present in the frequency range of interest. Two common signals used are the chirp signal and a broadband noise signal. The chirp signal is a sinusoid swept from a start frequency to a stop frequency, thus generating energy across a given frequency range. White and pseudorandom noise have flat broadband frequency spectra; that is, energy is present at all frequencies.

It is best not to use windows when analyzing frequency response signals. If you generate a chirp stimulus signal at the same rate you acquire the response, you can match the acquisition frame size to match the length of the chirp. No window is generally the best choice for a broadband signal source. Because some stimulus signals are not constant in frequency across the time record, applying a window may obscure important portions of the transient response.

Conclusion

There are many issues to consider when analyzing and measuring signals from plug-in DAQ devices. Unfortunately, it is easy to make incorrect spectral measurements. Understanding the basic computations involved in FFT-based measurement, knowing how to prevent antialiasing, properly scaling and converting to different units, choosing and using windows correctly, and learning how to use FFT-based functions for network measurement are all critical to the success of analysis and measurement tasks. Being equipped with this knowledge and using the tools discussed in this application note can bring you more success with your individual application.

References

Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform” in *Proceedings of the IEEE* Vol. 66, No. 1, January 1978.

Audio Frequency Fourier Analyzer (AFFA) User Guide, National Instruments, September 1991.

Horowitz, Paul, and Hill, Winfield, *The Art of Electronics*, 2nd Edition, Cambridge University Press, 1989.

Nuttall, Albert H. “Some Windows with Very Good Sidelobe Behavior,” *IEEE Transactions on Acoustics, Speech, and Signal Processing* Vol. 29, No. 1, February 1981.

Randall, R.B., and Tech, B. *Frequency Analysis*, 3rd Edition, Bruël and Kjær, September 1979.

The Fundamentals of Signal Analysis, Application Note 243, Hewlett-Packard, 1985.

340555B-01 Jul00

Chapter 7. Random Numbers

7.0 Introduction

It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce "random" numbers. More than perverse, it may seem to be a conceptual impossibility. Any program, after all, will produce output that is entirely predictable, hence not truly "random."

Nevertheless, practical computer "random number generators" are in common use. We will leave it to philosophers of the computer age to resolve the paradox in a deep way (see, e.g., Knuth [1] §3.5 for discussion and references). One sometimes hears computer-generated sequences termed *pseudo-random*, while the word *random* is reserved for the output of an intrinsically random physical process, like the elapsed time between clicks of a Geiger counter placed next to a sample of some radioactive element. We will not try to make such fine distinctions.

A working, though imprecise, definition of randomness in the context of computer-generated sequences, is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don't, then at least one of them is not (from your point of view) a good generator.

The above definition may seem circular, comparing, as it does, one generator to another. However, there exists a body of random number generators which mutually do satisfy the definition over a very, very broad class of applications programs. And it is also found empirically that statistically identical results are obtained from random numbers produced by physical processes. So, because such generators are known to exist, we can leave to the philosophers the problem of defining them.

A pragmatic point of view, then, is that randomness is in the eye of the beholder (or programmer). What is random enough for one application may not be random enough for another. Still, one is not entirely adrift in a sea of incommensurable applications programs: There is a certain list of statistical tests, some sensible and some merely enshrined by history, which on the whole will do a very good job of ferreting out any correlations that are likely to be detected by an applications program (in this case, yours). Good random number generators ought to pass all of these tests; or at least the user had better be aware of any that they fail, so that he or she will be able to judge whether they are relevant to the case at hand.

As for references on this subject, the one to turn to first is Knuth [1]. Then try [2]. Only a few of the standard books on numerical methods [3-4] treat topics relating to random numbers.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10. [4]

7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think "random numbers" are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

System-Supplied Random Number Generators

Most C implementations have, lurking within, a pair of library routines for initializing, and then generating, "random numbers." In ANSI C, the synopsis is:

```
#include <stdlib.h>
#define RAND_MAX ...

void srand(unsigned seed);
int rand(void);
```

You initialize the random number generator by invoking `srand(seed)` with some arbitrary `seed`. Each initializing value will typically result in a different random sequence, or a least a different starting point in some one enormously long sequence. The *same* initializing value of `seed` will always return the *same* random sequence, however.

You obtain successive random numbers in the sequence by successive calls to `rand()`. That function returns an integer that is typically in the range 0 to the largest representable positive value of type `int` (inclusive). Usually, as in ANSI C, this largest value is available as `RAND_MAX`, but sometimes you have to figure it out for yourself. If you want a random `float` value between 0.0 (inclusive) and 1.0 (exclusive), you get it by an expression like

```
x = rand()/(RAND_MAX+1.0);
```

Now our first, and perhaps most important, lesson in this chapter is: be *very, very* suspicious of a system-supplied `rand()` that resembles the one just described. If all scientific papers whose results are in doubt because of bad `rand()`s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist. System-supplied `rand()`s are almost always *linear congruential generators*, which generate a sequence of integers $I_1, I_2, I_3, \ldots$, each between 0 and $m - 1$ (e.g., `RAND_MAX`) by the recurrence relation

$$I_{j+1} = aI_j + c \pmod{m} \tag{7.1.1}$$

Here m is called the *modulus*, and a and c are positive integers called the *multiplier* and the *increment* respectively. The recurrence (7.1.1) will eventually repeat itself, with a period that is obviously no greater than m. If $m, a,$ and c are properly chosen, then the period will be of maximal length, i.e., of length m. In that case, all possible integers between 0 and $m - 1$ occur at some point, so any initial "seed" choice of I_0 is as good as any other: the sequence just takes off from that point.

Although this general framework is powerful enough to provide quite decent random numbers, its implementation in many, if not most, ANSI C libraries is quite flawed; quite a number of implementations are in the category "totally botched." Blame should be apportioned about equally between the ANSI C committee and the implementors. The typical problems are these: First, since the ANSI standard specifies that `rand()` return a value of type `int` — which is only a two-byte quantity on many machines — `RAND_MAX` is often *not* very large. The ANSI C standard requires only that it be at least 32767. This can be disastrous in many circumstances: for a Monte Carlo integration (§7.6 and §7.8), you might well want to evaluate 10^6 different points, but actually be evaluating the same 32767 points 30 times each, not at all the same thing! You should categorically reject any library random number routine with a two-byte returned value.

Second, the ANSI committee's published rationale includes the following mischievous passage: "The committee decided that an implementation should be allowed to provide a `rand` function which generates the best random sequence possible in that implementation, and therefore mandated no standard algorithm. It recognized the value, however, of being able to generate the same pseudo-random sequence in different implementations, and so *it has published an example*. . . . [emphasis added]" The "example" is

```
unsigned long next=1;

int rand(void) /* NOT RECOMMENDED (see text) */
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next=seed;
}
```

This corresponds to equation (7.1.1) with $a = 1103515245$, $c = 12345$, and $m = 2^{32}$ (since arithmetic done on `unsigned long` quantities is guaranteed to return the correct low-order bits). These are not particularly good choices for a and c (the period is only 2^{30}), though they are not gross embarrassments by themselves. The real botches occur when implementors, taking the committee's statement above as license, try to "improve" on the published example. For example, one popular 32-bit PC-compatible compiler provides a `long` generator that uses the above congruence, but swaps the high-order and low-order 16 bits of the returned value. Somebody probably thought that this extra flourish added randomness; in fact it ruins the generator. While these kinds of blunders can, of course, be fixed, there remains a fundamental flaw in simple linear congruential generators, which we now discuss.

The linear congruential method has the advantage of being very fast, requiring only a few operations per call, hence its almost universal use. It has the disadvantage that it is not free of sequential correlation on successive calls. If k random numbers at a time are used to plot points in k dimensional space (with each coordinate between 0 and 1), then the points will not tend to "fill up" the k-dimensional space, but rather will lie on $(k-1)$-dimensional "planes." There will be *at most* about $m^{1/k}$ such planes. If the constants m, a, and c are not very carefully chosen, there will be *many fewer than that.* If m is as bad as 32768, then the number of planes on which triples of points lie in three-dimensional space will be no greater than about the cube root of 32768, or 32. Even if m is close to the machine's largest representable integer, e.g., $\sim 2^{32}$, the number of planes on which triples of points lie in three-dimensional space is usually no greater than about the cube root of 2^{32}, about 1600. You might well be focusing attention on a physical process that occurs in a small fraction of the total volume, so that the discreteness of the planes can be very pronounced.

Even worse, you might be using a generator whose choices of $m, a,$ and c have been botched. One infamous such routine, `RANDU`, with $a = 65539$ and $m = 2^{31}$, was widespread on IBM mainframe computers for many years, and widely copied onto other systems [1]. One of us recalls producing a "random" plot with only 11 planes, and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random." Figure that out.

Correlation in k-space is not the only weakness of linear congruential generators. Such generators often have their low-order (least significant) bits much less random than their high-order bits. If you want to generate a random integer between 1 and 10, you should always do it using high-order bits, as in

```
j=1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

and never by anything resembling

```
j=1+(rand() % 10);
```

(which uses lower-order bits). Similarly you should never try to take apart a "`rand()`" number into several supposedly random pieces. Instead use separate calls for every piece.

Portable Random Number Generators

Park and Miller [1] have surveyed a large number of random number generators that have been used over the last 30 years or more. Along with a good theoretical review, they present an anecdotal sampling of a number of inadequate generators that have come into widespread use. The historical record is nothing if not appalling.

There is good evidence, both theoretical and empirical, that the simple multiplicative congruential algorithm

$$I_{j+1} = aI_j \pmod{m} \tag{7.1.2}$$

can be as good as any of the more general linear congruential generators that have $c \neq 0$ (equation 7.1.1) — *if* the multiplier a and modulus m are chosen exquisitely carefully. Park and Miller propose a "Minimal Standard" generator based on the choices

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2147483647 \tag{7.1.3}$$

First proposed by Lewis, Goodman, and Miller in 1969, this generator has in subsequent years passed all new theoretical tests, and (perhaps more importantly) has accumulated a large amount of successful use. Park and Miller do not claim that the generator is "perfect" (we will see below that it is not), but only that it is a good minimal standard against which other generators should be judged.

It is not possible to implement equations (7.1.2) and (7.1.3) directly in a high-level language, since the product of a and $m - 1$ exceeds the maximum value for a 32-bit integer. Assembly language implementation using a 64-bit product register is straightforward, but not portable from machine to machine. A trick due to Schrage [2,3] for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits (including a sign bit) is therefore extremely interesting: It allows the Minimal Standard generator to be implemented in essentially any programming language on essentially any machine.

Schrage's algorithm is based on an *approximate factorization* of m,

$$m = aq + r, \quad \text{i.e.,} \quad q = [m/a],\ r = m \bmod a \tag{7.1.4}$$

with square brackets denoting integer part. If r is small, specifically $r < q$, and $0 < z < m - 1$, it can be shown that both $a(z \bmod q)$ and $r[z/q]$ lie in the range $0, \ldots, m - 1$, and that

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is } \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases} \tag{7.1.5}$$

The application of Schrage's algorithm to the constants (7.1.3) uses the values $q = 127773$ and $r = 2836$.

Here is an implementation of the Minimal Standard generator:

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

float ran0(long *idum)
```
"Minimal" random number generator of Park and Miller. Returns a uniform random deviate between 0.0 and 1.0. Set or reset `idum` to any integer value (except the unlikely value `MASK`) to initialize the sequence; `idum` must not be altered between calls for successive deviates in a sequence.
```
{
    long k;
    float ans;

    *idum ^= MASK;                    XORing with MASK allows use of zero and other
    k=(*idum)/IQ;                         simple bit patterns for idum.
    *idum=IA*(*idum-k*IQ)-IR*k;       Compute idum=(IA*idum) % IM without over-
    if (*idum < 0) *idum += IM;           flows by Schrage's method.
    ans=AM*(*idum);                   Convert idum to a floating result.
    *idum ^= MASK;                    Unmask before return.
    return ans;
}
```

The period of `ran0` is $2^{31} - 2 \approx 2.1 \times 10^9$. A peculiarity of generators of the form (7.1.2) is that the value 0 must never be allowed as the initial seed — it perpetuates itself — and it never occurs for any nonzero initial seed. Experience has shown that users always manage to call random number generators with the seed `idum=0`. That is why `ran0` performs its exclusive-or with an arbitrary constant both on entry and exit. If you are the first user in history to be proof against human error, you can remove the two lines with the $\wedge$ operation.

Park and Miller discuss two other multipliers a that can be used with the same $m = 2^{31} - 1$. These are $a = 48271$ (with $q = 44488$ and $r = 3399$) and $a = 69621$ (with $q = 30845$ and $r = 23902$). These can be substituted in the routine `ran0` if desired; they may be slightly superior to Lewis *et al.*'s longer-tested values. No values other than these should be used.

The routine `ran0` is a Minimal Standard, satisfactory for the majority of applications, but we do not recommend it as the final word on random number generators. Our reason is precisely the simplicity of the Minimal Standard. It is not hard to think of situations where successive random numbers might be used in a way that accidentally conflicts with the generation algorithm. For example, since successive numbers differ by a multiple of only 1.6×10^4 out of a modulus of more than 2×10^9, very small random numbers will tend to be followed by smaller than average values. One time in 10^6, for example, there will be a value $< 10^{-6}$ returned (as there should be), but this will *always* be followed by a value less than about 0.0168. One can easily think of applications involving rare events where this property would lead to wrong results.

There are other, more subtle, serial correlations present in `ran0`. For example, if successive points (I_i, I_{i+1}) are binned into a two-dimensional plane for $i = 1, 2, \ldots, N$, then the resulting distribution fails the χ^2 test when N is greater than a few $\times 10^7$, much less than the period $m - 2$. Since low-order serial correlations have historically been such a bugaboo, and since there is a very simple way to remove them, we think that it is prudent to do so.

The following routine, `ran1`, uses the Minimal Standard for its random value, but it shuffles the output to remove low-order serial correlations. A random deviate derived from the jth value in the sequence, I_j, is output not on the jth call, but rather on a randomized later call, $j + 32$ on average. The shuffling algorithm is due to Bays and Durham as described in Knuth [4], and is illustrated in Figure 7.1.1.

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran1(long *idum)
```
"Minimal" random number generator of Park and Miller with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with `idum` a negative integer to initialize; thereafter, do not alter `idum` between successive deviates in a sequence. `RNMX` should approximate the largest floating value that is less than 1.
```
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy) {                 Initialize.
        if (-(*idum) < 1) *idum=1;           Be sure to prevent idum = 0.
        else *idum = -(*idum);
        for (j=NTAB+7;j>=0;j--) {            Load the shuffle table (after 8 warm-ups).
            k=(*idum)/IQ;
            *idum=IA*(*idum-k*IQ)-IR*k;
            if (*idum < 0) *idum += IM;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ;                            Start here when not initializing.
    *idum=IA*(*idum-k*IQ)-IR*k;              Compute idum=(IA*idum) % IM without over-
    if (*idum < 0) *idum += IM;                 flows by Schrage's method.
    j=iy/NDIV;                               Will be in the range 0..NTAB-1.
    iy=iv[j];                                Output previously stored value and refill the
    iv[j] = *idum;                              shuffle table.
    if ((temp=AM*iy) > RNMX) return RNMX;    Because users don't expect endpoint values.
    else return temp;
}
```

The routine `ran1` passes those statistical tests that `ran0` is known to fail. In fact, we do not know of any statistical test that `ran1` fails to pass, except when the number of calls starts to become on the order of the period m, say $> 10^8 \approx m/20$.

For situations when even longer random sequences are needed, L'Ecuyer [6] has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. The basic idea is simply to add the two sequences, modulo the modulus of

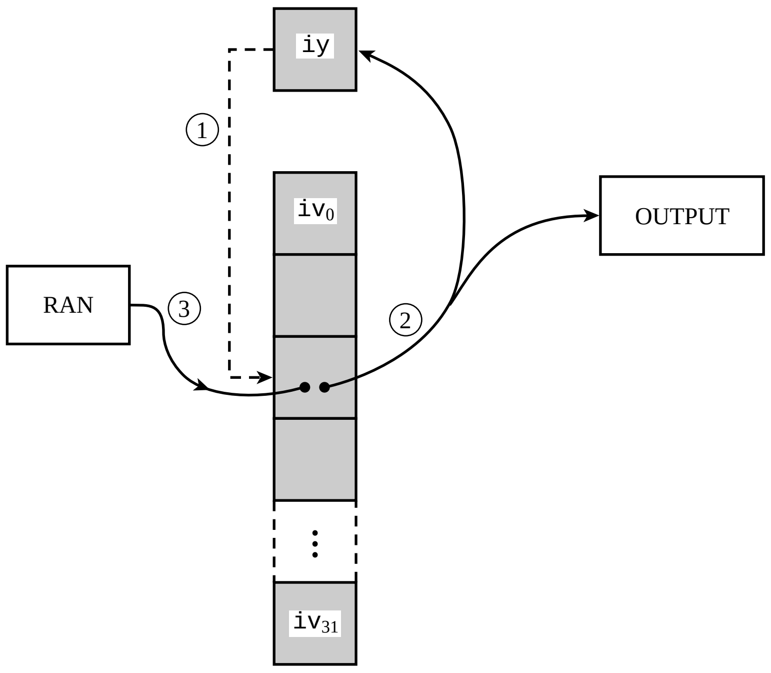


Figure 7.1.1. Shuffling procedure used in `ran1` to break up sequential correlations in the Minimal Standard generator. Circled numbers indicate the sequence of events: On each call, the random number in `iy` is used to choose a random element in the array `iv`. That element becomes the output random number, and also is the next `iy`. Its spot in `iv` is refilled from the Minimal Standard routine.

either of them (call it m). A trick to avoid an intermediate value that overflows the integer wordsize is to subtract rather than add, and then add back the constant $m - 1$ if the result is ≤ 0, so as to wrap around into the desired interval $0, \ldots, m - 1$.

Notice that it is not necessary that this wrapped subtraction be able to reach all values $0, \ldots, m - 1$ from *every* value of the first sequence. Consider the absurd extreme case where the value subtracted was only between 1 and 10: The resulting sequence would still be no less random than the first sequence by itself. As a practical matter it is only necessary that the second sequence have a range covering *substantially* all of the range of the first. L'Ecuyer recommends the use of the two generators $m_1 = 2147483563$ (with $a_1 = 40014$, $q_1 = 53668$, $r_1 = 12211$) and $m_2 = 2147483399$ (with $a_2 = 40692$, $q_2 = 52774$, $r_2 = 3791$). Both moduli are slightly less than 2^{31}. The periods $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$ and $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ share only the factor 2, so the period of the combined generator is $\approx 2.3 \times 10^{18}$. For present computers, period exhaustion is a practical impossibility.

Combining the two generators breaks up serial correlations to a considerable extent. We nevertheless recommend the additional shuffle that is implemented in the following routine, `ran2`. We think that, within the limits of its floating-point precision, `ran2` provides perfect random numbers; a practical definition of "perfect" is that we will pay $1000 to the first reader who convinces us otherwise (by finding a statistical test that `ran2` fails in a nontrivial way, excluding the ordinary limitations of a machine's floating-point representation).

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran2(long *idum)
```
Long period ($> 2 \times 10^{18}$) random number generator of L'Ecuyer with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with `idum` a negative integer to initialize; thereafter, do not alter `idum` between successive deviates in a sequence. `RNMX` should approximate the largest floating value that is less than 1.
```
{
    int j;
    long k;
    static long idum2=123456789;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0) {                           Initialize.
        if (-(*idum) < 1) *idum=1;              Be sure to prevent idum = 0.
        else *idum = -(*idum);
        idum2=(*idum);
        for (j=NTAB+7;j>=0;j--) {               Load the shuffle table (after 8 warm-ups).
            k=(*idum)/IQ1;
            *idum=IA1*(*idum-k*IQ1)-k*IR1;
            if (*idum < 0) *idum += IM1;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ1;                              Start here when not initializing.
    *idum=IA1*(*idum-k*IQ1)-k*IR1;              Compute idum=(IA1*idum) % IM1 without
    if (*idum < 0) *idum += IM1;                    overflows by Schrage's method.
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;              Compute idum2=(IA2*idum) % IM2 likewise.
    if (idum2 < 0) idum2 += IM2;
    j=iy/NDIV;                                  Will be in the range 0..NTAB-1.
    iy=iv[j]-idum2;                             Here idum is shuffled, idum and idum2 are
    iv[j] = *idum;                                  combined to generate output.
    if (iy < 1) iy += IMM1;
    if ((temp=AM*iy) > RNMX) return RNMX;       Because users don't expect endpoint values.
    else return temp;
}
```

L'Ecuyer [6] lists additional short generators that can be combined into longer ones, including generators that can be implemented in 16-bit integer arithmetic.

Finally, we give you Knuth's suggestion [4] for a portable routine, which we have translated to the present conventions as `ran3`. This is not based on the linear congruential method at all, but rather on a *subtractive method* (see also [5]). One might hope that its weaknesses, if any, are therefore of a highly different character from the weaknesses, if any, of `ran1` above. If you ever suspect trouble with one routine, it is a good idea to try the other in the same application. `ran3` has one nice feature: if your machine is poor on integer arithmetic (i.e., is limited to 16-bit integers), you can declare `mj`, `mk`, and `ma[]` as `float`, define `mbig` and `mseed` as 4000000 and 1618033, respectively, and the routine will be rendered entirely floating-point.

```
#include <stdlib.h>                         Change to math.h in K&R C.
#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)
According to Knuth, any large MBIG, and any smaller (but still large) MSEED can be substituted
for the above values.

float ran3(long *idum)
Returns a uniform random deviate between 0.0 and 1.0. Set idum to any negative value to
initialize or reinitialize the sequence.
{
    static int inext,inextp;
    static long ma[56];                     The value 56 (range ma[1..55]) is special and
    static int iff=0;                           should not be modified; see Knuth.
    long mj,mk;
    int i,ii,k;

    if (*idum < 0 || iff == 0) {            Initialization.
        iff=1;
        mj=labs(MSEED-labs(*idum));         Initialize ma[55] using the seed idum and the
        mj %= MBIG;                             large number MSEED.
        ma[55]=mj;
        mk=1;
        for (i=1;i<=54;i++) {               Now initialize the rest of the table,
            ii=(21*i) % 55;                 in a slightly random order,
            ma[ii]=mk;                      with numbers that are not especially random.
            mk=mj-mk;
            if (mk < MZ) mk += MBIG;
            mj=ma[ii];
        }
        for (k=1;k<=4;k++)                  We randomize them by "warming up the gener-
            for (i=1;i<=55;i++) {               ator."
                ma[i] -= ma[1+(i+30) % 55];
                if (ma[i] < MZ) ma[i] += MBIG;
            }
        inext=0;                            Prepare indices for our first generated number.
        inextp=31;                          The constant 31 is special; see Knuth.
        *idum=1;
    }
    Here is where we start, except on initialization.
    if (++inext == 56) inext=1;             Increment inext and inextp, wrapping around
    if (++inextp == 56) inextp=1;               56 to 1.
    mj=ma[inext]-ma[inextp];                Generate a new random number subtractively.
    if (mj < MZ) mj += MBIG;                Be sure that it is in range.
    ma[inext]=mj;                           Store it,
    return mj*FAC;                          and output the derived uniform deviate.
}
```

Quick and Dirty Generators

One sometimes would like a "quick and dirty" generator to embed in a program, perhaps taking only one or two lines of code, just to *somewhat* randomize things. One might wish to process data from an experiment not always in exactly the same order, for example, so that the first output is more "typical" than might otherwise be the case.

For this kind of application, all we really need is a list of "good" choices for m, a, and c in equation (7.1.1). If we don't need a period longer than 10^4 to 10^6, say, we can keep the value of $(m-1)a+c$ small enough to avoid overflows that would otherwise mandate the extra complexity of Schrage's method (above). We can thus easily embed in our programs

```
unsigned long jran,ia,ic,im;
float ran;
...
jran=(jran*ia+ic) % im;
ran=(float) jran / (float) im;
```

whenever we want a quick and dirty uniform deviate, or

```
jran=(jran*ia+ic) % im;
j=jlo+((jhi-jlo+1)*jran)/im;
```

whenever we want an integer between `jlo` and `jhi`, inclusive. (In both cases `jran` was once initialized to any seed value between 0 and `im-1`.)

Be sure to remember, however, that when `im` is small, the kth root of it, which is the number of planes in k-space, is even smaller! So a quick and dirty generator should never be used to select points in k-space with $k > 1$.

With these caveats, some "good" choices for the constants are given in the accompanying table. These constants (i) give a period of maximal length `im`, and, more important, (ii) pass Knuth's "spectral test" for dimensions 2, 3, 4, 5, and 6. The increment `ic` is a prime, close to the value $(\frac{1}{2}-\frac{1}{6}\sqrt{3})$`im`; actually almost any value of `ic` that is relatively prime to `im` will do just as well, but there is some "lore" favoring this choice (see [4], p. 84).

An Even Quicker Generator

In C, if you multiply two `unsigned long int` integers on a machine with a 32-bit long integer representation, the value returned is the low-order 32 bits of the true 64-bit product. If we now choose $m = 2^{32}$, the "mod" in equation (7.1.1) is free, and we have simply

$$I_{j+1} = aI_j + c \tag{7.1.6}$$

Knuth suggests $a = 1664525$ as a suitable multiplier for this value of m. H.W. Lewis has conducted extensive tests of this value of a with $c = 1013904223$, which is a prime close to $(\sqrt{5}-2)m$. The resulting in-line generator (we will call it `ranqd1`) is simply

```
unsigned long idum;
...
idum = 1664525L*idum + 1013904223L;
```

This is about as good as any 32-bit linear congruential generator, entirely adequate for many uses. And, with only a single multiply and add, it is *very* fast.

To check whether your machine has the desired integer properties, see if you can generate the following sequence of 32-bit values (given here in hex): 00000000, 3C6EF35F, 47502932, D1CCF6E9, AAF95334, 6252E503, 9F2EC686, 57FE6C2D, A3D95FA8, 81FDBEE7, 94F0AF1A, CBF633B1.

If you need floating-point values instead of 32-bit integers, and want to avoid a divide by floating-point 2^{32}, a dirty trick is to mask in an exponent that makes the value lie between 1 and 2, then subtract 1.0. The resulting in-line generator (call it `ranqd2`) will look something like

Constants for Quick and Dirty Random Number Generators

overflow at	im	ia	ic	overflow at	im	ia	ic
	6075	106	1283		86436	1093	18257
2^{20}					121500	1021	25673
	7875	211	1663		259200	421	54773
2^{21}				2^{27}			
	7875	421	1663		117128	1277	24749
2^{22}					121500	2041	25673
	6075	1366	1283		312500	741	66037
	6655	936	1399	2^{28}			
	11979	430	2531		145800	3661	30809
2^{23}					175000	2661	36979
	14406	967	3041		233280	1861	49297
	29282	419	6173		244944	1597	51749
	53125	171	11213	2^{29}			
2^{24}					139968	3877	29573
	12960	1741	2731		214326	3613	45289
	14000	1541	2957		714025	1366	150889
	21870	1291	4621	2^{30}			
	31104	625	6571		134456	8121	28411
	139968	205	29573		259200	7141	54773
2^{25}				2^{31}			
	29282	1255	6173		233280	9301	49297
	81000	421	17117		714025	4096	150889
	134456	281	28411	2^{32}			
2^{26}							

```
    unsigned long idum,itemp;
    float rand;
#ifdef vax
    static unsigned long jflone = 0x00004080;
    static unsigned long jflmsk = 0xffff007f;
#else
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;
#endif
    ...
    idum = 1664525L*idum + 1013904223L;
    itemp = jflone | (jflmsk & idum);
    rand = (*(float *)&itemp)-1.0;
```

The hex constants 3F800000 and 007FFFFF are the appropriate ones for computers using the IEEE representation for 32-bit floating-point numbers (e.g., IBM PCs and most UNIX workstations). For DEC VAXes, the correct hex constants are, respectively, 00004080 and FFFF007F. Notice that the IEEE mask results in the floating-point number being constructed out of the 23 low-order bits of the integer, which is not ideal. (Your authors have tried very hard to make *almost all* of the material in this book machine and compiler independent — indeed, even programming language independent. This subsection is a rare aberration. Forgive us. Once in a great while the temptation to be *really dirty* is just irresistible.)

Relative Timings and Recommendations

Timings are inevitably machine dependent. Nevertheless the following table is indicative of the *relative* timings, for typical machines, of the various uniform generators discussed in this section, plus ran4 from §7.5. Smaller values in the table indicate faster generators. The generators ranqd1 and ranqd2 refer to the "quick and dirty" generators immediately above.

Generator	Relative Execution Time
ran0	$\equiv 1.0$
ran1	≈ 1.3
ran2	≈ 2.0
ran3	≈ 0.6
ranqd1	≈ 0.10
ranqd2	≈ 0.25
ran4	≈ 4.0

On balance, we recommend ran1 for general use. It is portable, based on Park and Miller's Minimal Standard generator with an additional shuffle, and has no known (to us) flaws other than period exhaustion.

If you are generating more than 100,000,000 random numbers in a single calculation (that is, more than about 5% of ran1's period), we recommend the use of ran2, with its much longer period.

Knuth's subtractive routine ran3 seems to be the timing winner among portable routines. Unfortunately the subtractive method is not so well studied, and not a standard. We like to keep ran3 in reserve for a "second opinion," substituting it when we suspect another generator of introducing unwanted correlations into a calculation.

The routine ran4 generates *extremely* good random deviates, and has some other nice properties, but it is slow. See §7.5 for discussion.

Finally, the quick and dirty in-line generators ranqd1 and ranqd2 are very fast, but they are somewhat machine dependent, and at best only as good as a 32-bit linear congruential generator ever is — in our view not good enough in many situations. We would use these only in very special cases, where speed is critical.

CITED REFERENCES AND FURTHER READING:

Park, S.K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192–1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §§3.2–3.3. [4]

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31, pp. 742–774. [6]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

7.2 Transformation Method: Exponential and Normal Deviates

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2.1}$$

The probability distribution $p(x)$ is of course normalized, so that

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

Now suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

Exponential Deviates

As an example, suppose that $y(x) \equiv -\ln(x)$, and that $p(x)$ is as given by equation (7.2.1) for a uniform deviate. Then

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy \tag{7.2.5}$$

which is distributed exponentially. This exponential distribution occurs frequently in real problems, usually as the distribution of waiting times between independent Poisson-random events, for example the radioactive decay of nuclei. You can also easily see (from 7.2.4) that the quantity y/λ has the probability distribution $\lambda e^{-\lambda y}$.

So we have

```
#include <math.h>

float expdev(long *idum)
```
Returns an exponentially distributed, positive, random deviate of unit mean, using `ran1(idum)` as the source of uniform deviates.
```
{
    float ran1(long *idum);
    float dum;

    do
        dum=ran1(idum);
    while (dum == 0.0);
    return -log(dum);
}
```

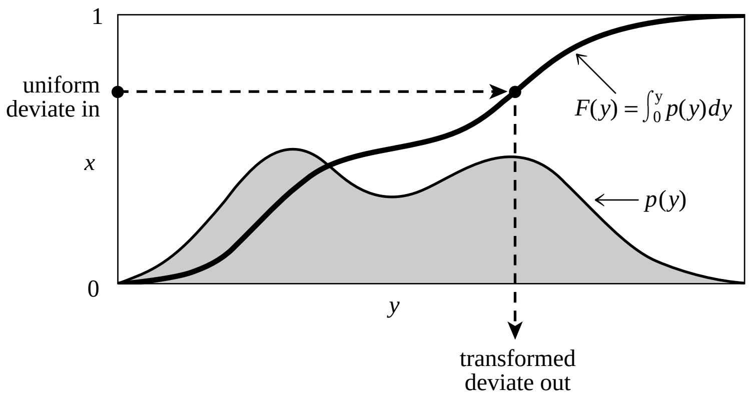


Figure 7.2.1. Transformation method for generating a random deviate y from a known probability distribution $p(y)$. The indefinite integral of $p(y)$ must be known and invertible. A uniform deviate x is chosen between 0 and 1. Its corresponding y on the definite-integral curve is the desired deviate.

Let's see what is involved in using the above *transformation method* to generate some arbitrary desired distribution of y's, say one with $p(y) = f(y)$ for some positive function f whose integral is 1. (See Figure 7.2.1.) According to (7.2.4), we need to solve the differential equation

$$\frac{dx}{dy} = f(y) \tag{7.2.6}$$

But the solution of this is just $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$. The desired transformation which takes a uniform deviate into one distributed as $f(y)$ is therefore

$$y(x) = F^{-1}(x) \tag{7.2.7}$$

where F^{-1} is the inverse function to F. Whether (7.2.7) is feasible to implement depends on whether the *inverse function of the integral of f(y)* is itself feasible to compute, either analytically or numerically. Sometimes it is, and sometimes it isn't.

Incidentally, (7.2.7) has an immediate geometric interpretation: Since $F(y)$ is the area under the probability curve to the left of y, (7.2.7) is just the prescription: choose a uniform random x, then find the value y that has that fraction x of probability area to its left, and return the value y.

Normal (Gaussian) Deviates

Transformation methods generalize to more than one dimension. If x_1, x_2, $\dots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \dots)$ $dx_1 dx_2 \dots$, and if $y_1, y_2, \dots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \dots) dy_1 dy_2 \dots = p(x_1, x_2, \dots) \left| \frac{\partial(x_1, x_2, \dots)}{\partial(y_1, y_2, \dots)} \right| dy_1 dy_2 \dots \tag{7.2.8}$$

where $|\partial(\quad)/\partial(\quad)|$ is the Jacobian determinant of the x's with respect to the y's (or reciprocal of the Jacobian determinant of the y's with respect to the x's).

An important example of the use of (7.2.8) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$\begin{aligned} y_1 &= \sqrt{-2 \ln x_1} \cos 2\pi x_2 \\ y_2 &= \sqrt{-2 \ln x_1} \sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi} \text{arctan} \frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right] \left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.2.12}$$

Since this is the product of a function of y_2 alone and a function of y_1 alone, we see that each y is independently distributed according to the normal distribution (7.2.9).

One further trick is useful in applying (7.2.10). Suppose that, instead of picking uniform deviates x_1 and x_2 in the unit square, we instead pick v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares, $R^2 \equiv v_1^2 + v_2^2$ is a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1 axis can serve as the random angle $2\pi x_2$. What's the advantage? It's that the cosine and sine in (7.2.10) can now be written as $v_1/\sqrt{R^2}$ and $v_2/\sqrt{R^2}$, obviating the trigonometric function calls!

We thus have

```
#include <math.h>

float gasdev(long *idum)
Returns a normally distributed deviate with zero mean and unit variance, using ran1(idum)
as the source of uniform deviates.
{
    float ran1(long *idum);
    static int iset=0;
    static float gset;
    float fac,rsq,v1,v2;

    if (*idum < 0) iset=0;                   Reinitialize.
    if (iset == 0) {                         We don't have an extra deviate handy, so
        do {
            v1=2.0*ran1(idum)-1.0;           pick two uniform numbers in the square ex-
            v2=2.0*ran1(idum)-1.0;               tending from -1 to +1 in each direction,
            rsq=v1*v1+v2*v2;                 see if they are in the unit circle,
        } while (rsq >= 1.0 || rsq == 0.0);       and if they are not, try again.
        fac=sqrt(-2.0*log(rsq)/rsq);
        Now make the Box-Muller transformation to get two normal deviates. Return one and
        save the other for next time.
        gset=v1*fac;
        iset=1;                                Set flag.
        return v2*fac;
    } else {                                   We have an extra deviate handy,
        iset=0;                                so unset the flag,
        return gset;                           and return it.
    }
}
```

See Devroye [1] and Bratley [2] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §9.1. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 116ff.

7.3 Rejection Method: Gamma, Poisson, Binomial Deviates

The *rejection method* is a powerful, general technique for generating random deviates whose distribution function $p(x)dx$ (probability of a value occurring between x and $x + dx$) is known and computable. The rejection method does *not* require that the cumulative distribution function [indefinite integral of $p(x)$] be readily computable, much less the inverse of that function — which was required for the transformation method in the previous section.

The rejection method is based on a simple geometrical argument:

Draw a graph of the probability distribution $p(x)$ that you wish to generate, so that the area under the curve in any range of x corresponds to the desired probability of generating an x in that range. If we had some way of choosing a random point *in two dimensions*, with uniform probability in the *area* under your curve, then the x value of that random point would have the desired distribution.

Now, on the same graph, draw any other curve $f(x)$ which has finite (not infinite) area and lies everywhere *above* your original probability distribution. (This is always possible, because your original curve encloses only unit area, by definition of probability.) We will call this $f(x)$ the *comparison function*. Imagine now that you have some way of choosing a random point in two dimensions that is uniform in the area under the comparison function. Whenever that point lies outside the area under the original probability distribution, we will *reject* it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will *accept* it. It should be obvious that the accepted points are uniform in the accepted area, so that their x values have the desired distribution. It

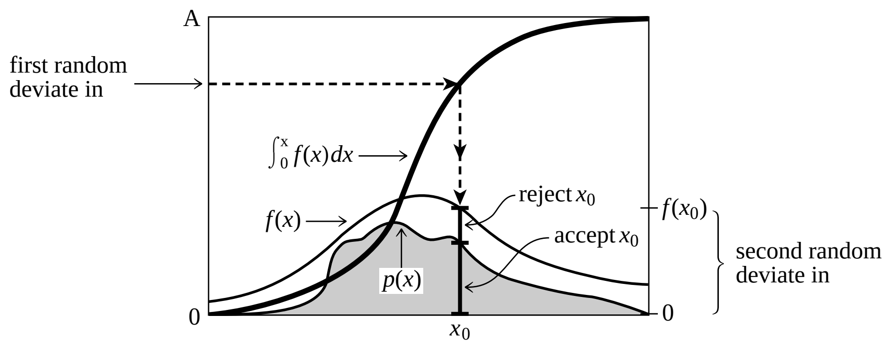


Figure 7.3.1. Rejection method for generating a random deviate x from a known probability distribution $p(x)$ that is everywhere less than some other function $f(x)$. The transformation method is first used to generate a random deviate x of the distribution f (compare Figure 7.2.1). A second uniform deviate is used to decide whether to accept or reject that x. If it is rejected, a new deviate of f is found; and so on. The ratio of accepted to rejected points is the ratio of the area under p to the area between p and f.

should also be obvious that the fraction of points rejected just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. For example, a comparison function whose area is less than 2 will reject fewer than half the points, even if it approximates the probability function very badly at some values of x, e.g., remains finite in some region where $p(x)$ is zero.

It remains only to suggest how to choose a uniform random point in two dimensions under the comparison function $f(x)$. A variant of the transformation method (§7.2) does nicely: Be sure to have chosen a comparison function whose indefinite integral is known analytically, and is also analytically invertible to give x as a function of "area under the comparison function to the left of x." Now pick a uniform deviate between 0 and A, where A is the total area under $f(x)$, and use it to get a corresponding x. Then pick a uniform deviate between 0 and $f(x)$ as the y value for the two-dimensional point. You should be able to convince yourself that the point (x, y) is uniformly distributed in the area under the comparison function $f(x)$.

An equivalent procedure is to pick the second uniform deviate between zero and one, and accept or reject according to whether it is respectively less than or greater than the ratio $p(x)/f(x)$.

So, to summarize, the rejection method for some given $p(x)$ requires that one find, once and for all, some reasonably good comparison function $f(x)$. Thereafter, each deviate generated requires two uniform random deviates, one evaluation of f (to get the coordinate y), and one evaluation of p (to decide whether to accept or reject the point x, y). Figure 7.3.1 illustrates the procedure. Then, of course, this procedure must be repeated, on the average, A times before the final deviate is obtained.

Gamma Distribution

The gamma distribution of integer order $a > 0$ is the waiting time to the ath event in a Poisson random process of unit mean. For example, when $a = 1$, it is just the exponential distribution of §7.2, the waiting time to the first event.

A gamma deviate has probability $p_a(x)dx$ of occurring with a value between x and $x + dx$, where

$$p_a(x)dx = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx \qquad x > 0 \tag{7.3.1}$$

To generate deviates of (7.3.1) for small values of a, it is best to add up a exponentially distributed waiting times, i.e., logarithms of uniform deviates. Since the sum of logarithms is the logarithm of the product, one really has only to generate the product of a uniform deviates, then take the log.

For larger values of a, the distribution (7.3.1) has a typically "bell-shaped" form, with a peak at $x = a$ and a half-width of about $\sqrt{a}$.

We will be interested in several probability distributions with this same qualitative form. A useful comparison function in such cases is derived from the *Lorentzian distribution*

$$p(y)dy = \frac{1}{\pi}\left(\frac{1}{1+y^2}\right)dy \tag{7.3.2}$$

whose inverse indefinite integral is just the tangent function. It follows that the x-coordinate of an area-uniform random point under the comparison function

$$f(x) = \frac{c_0}{1 + (x - x_0)^2/a_0^2} \tag{7.3.3}$$

for any constants $a_0, c_0,$ and x_0, can be generated by the prescription

$$x = a_0 \tan(\pi U) + x_0 \tag{7.3.4}$$

where U is a uniform deviate between 0 and 1. Thus, for some specific "bell-shaped" $p(x)$ probability distribution, we need only find constants a_0, c_0, x_0, with the product a_0c_0 (which determines the area) as small as possible, such that (7.3.3) is everywhere greater than $p(x)$.

Ahrens has done this for the gamma distribution, yielding the following algorithm (as described in Knuth [1]):

```
#include <math.h>

float gamdev(int ia, long *idum)
```
Returns a deviate distributed as a gamma distribution of integer order ia, i.e., a waiting time to the iath event in a Poisson process of unit mean, using ran1(idum) as the source of uniform deviates.
```
{
    float ran1(long *idum);
    void nrerror(char error_text[]);
    int j;
    float am,e,s,v1,v2,x,y;

    if (ia < 1) nrerror("Error in routine gamdev");
    if (ia < 6) {                               Use direct method, adding waiting
        x=1.0;                                      times.
        for (j=1;j<=ia;j++) x *= ran1(idum);
        x = -log(x);
    } else {                                    Use rejection method.
```

```
        do {
            do {
                do {                                    These four lines generate the tan-
                    v1=ran1(idum);                          gent of a random angle, i.e., they
                    v2=2.0*ran1(idum)-1.0;                  are equivalent to
                } while (v1*v1+v2*v2 > 1.0);                y = tan(π * ran1(idum)).
                y=v2/v1;
                am=ia-1;
                s=sqrt(2.0*am+1.0);
                x=s*y+am;                               We decide whether to reject x:
            } while (x <= 0.0);                         Reject in region of zero probability.
            e=(1.0+y*y)*exp(am*log(x/am)-s*y);          Ratio of prob. fn. to comparison fn.
        } while (ran1(idum) > e);                       Reject on basis of a second uniform
    }                                                       deviate.
    return x;
}
```

Poisson Deviates

The Poisson distribution is conceptually related to the gamma distribution. It gives the probability of a certain integer number m of unit rate Poisson random events occurring in a given interval of time x, while the gamma distribution was the probability of waiting time between x and $x + dx$ to the mth event. Note that m takes on only integer values ≥ 0, so that the Poisson distribution, viewed as a continuous distribution function $p_x(m)dm$, is zero everywhere except where m is an integer ≥ 0. At such places, it is infinite, such that the integrated probability over a region containing the integer is some finite number. The total probability at an integer j is

$$\text{Prob}(j) = \int_{j-\epsilon}^{j+\epsilon} p_x(m)dm = \frac{x^j e^{-x}}{j!} \tag{7.3.5}$$

At first sight this might seem an unlikely candidate distribution for the rejection method, since no continuous comparison function can be larger than the infinitely tall, but infinitely narrow, *Dirac delta functions* in $p_x(m)$. However, there is a trick that we can do: Spread the finite area in the spike at j uniformly into the interval between j and $j + 1$. This defines a continuous distribution $q_x(m)dm$ given by

$$q_x(m)dm = \frac{x^{[m]}e^{-x}}{[m]!}dm \tag{7.3.6}$$

where $[m]$ represents the largest integer less than m. If we now use the rejection method to generate a (noninteger) deviate from (7.3.6), and then take the integer part of that deviate, it will be as if drawn from the desired distribution (7.3.5). (See Figure 7.3.2.) This trick is general for any integer-valued probability distribution.

For x large enough, the distribution (7.3.6) is qualitatively bell-shaped (albeit with a bell made out of small, square steps), and we can use the same kind of Lorentzian comparison function as was already used above. For small x, we can generate independent exponential deviates (waiting times between events); when the sum of these first exceeds x, then the number of events that would have occurred in waiting time x becomes known and is one less than the number of terms in the sum.

These ideas produce the following routine:

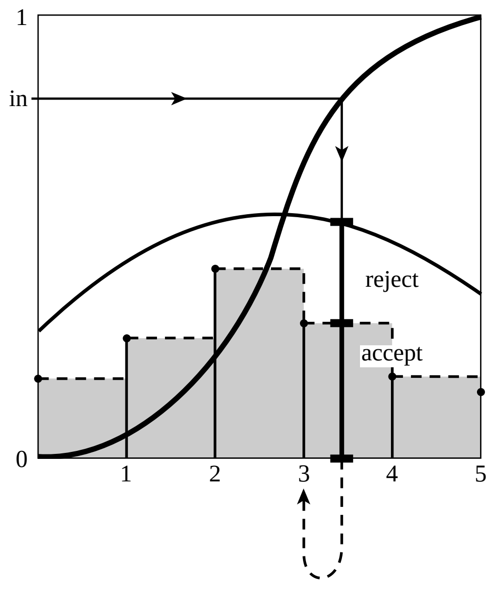


Figure 7.3.2. Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

```
#include <math.h>
#define PI 3.141592654

float poidev(float xm, long *idum)
```
Returns as a floating-point number an integer value that is a random deviate drawn from a Poisson distribution of mean `xm`, using `ran1(idum)` as a source of uniform random deviates.
```
{
    float gammln(float xx);
    float ran1(long *idum);
    static float sq,alxm,g,oldm=(-1.0);        oldm is a flag for whether xm has changed
    float em,t,y;                                   since last call.

    if (xm < 12.0) {                       Use direct method.
        if (xm != oldm) {
            oldm=xm;
            g=exp(-xm);                    If xm is new, compute the exponential.
        }
        em = -1;
        t=1.0;
        do {                               Instead of adding exponential deviates it is equiv-
            ++em;                              alent to multiply uniform deviates. We never
            t *= ran1(idum);                   actually have to take the log, merely com-
        } while (t > g);                       pare to the pre-computed exponential.
    } else {                               Use rejection method.
        if (xm != oldm) {                  If xm has changed since the last call, then pre-
            oldm=xm;                           compute some functions that occur below.
            sq=sqrt(2.0*xm);
            alxm=log(xm);
            g=xm*alxm-gammln(xm+1.0);
            The function gammln is the natural log of the gamma function, as given in §6.1.
        }
        do {
            do {                           y is a deviate from a Lorentzian comparison func-
                y=tan(PI*ran1(idum));          tion.
                em=sq*y+xm;                   em is y, shifted and scaled.
            } while (em < 0.0);               Reject if in regime of zero probability.
            em=floor(em);                     The trick for integer-valued distributions.
            t=0.9*(1.0+y*y)*exp(em*alxm-gammln(em+1.0)-g);
            The ratio of the desired distribution to the comparison function; we accept or
            reject by comparing it to another uniform deviate. The factor 0.9 is chosen so
            that t never exceeds 1.
        } while (ran1(idum) > t);
    }
    return em;
}
```

Binomial Deviates

If an event occurs with probability q, and we make n trials, then the number of times m that it occurs has the binomial distribution,

$$\int_{j-\epsilon}^{j+\epsilon} p_{n,q}(m)dm = \binom{n}{j} q^j (1-q)^{n-j} \tag{7.3.7}$$

The binomial distribution is integer valued, with m taking on possible values from 0 to n. It depends on *two* parameters, n and q, so is correspondingly a bit harder to implement than our previous examples. Nevertheless, the techniques already illustrated are sufficiently powerful to do the job:

```
#include <math.h>
#define PI 3.141592654

float bnldev(float pp, int n, long *idum)
Returns as a floating-point number an integer value that is a random deviate drawn from
a binomial distribution of n trials each of probability pp, using ran1(idum) as a source of
uniform random deviates.
{
    float gammln(float xx);
    float ran1(long *idum);
    int j;
    static int nold=(-1);
    float am,em,g,angle,p,bnl,sq,t,y;
    static float pold=(-1.0),pc,plog,pclog,en,oldg;

    p=(pp <= 0.5 ? pp : 1.0-pp);
    The binomial distribution is invariant under changing pp to 1-pp, if we also change the
    answer to n minus itself; we'll remember to do this below.
    am=n*p;                               This is the mean of the deviate to be produced.
    if (n < 25) {                         Use the direct method while n is not too large.
        bnl=0.0;                              This can require up to 25 calls to ran1.
        for (j=1;j<=n;j++)
            if (ran1(idum) < p) ++bnl;
    } else if (am < 1.0) {                If fewer than one event is expected out of 25
        g=exp(-am);                           or more trials, then the distribution is quite
        t=1.0;                                accurately Poisson. Use direct Poisson method.
        for (j=0;j<=n;j++) {
            t *= ran1(idum);
            if (t < g) break;
        }
        bnl=(j <= n ? j : n);
    } else {                              Use the rejection method.
```

```
        if (n != nold) {                If n has changed, then compute useful quanti-
            en=n;                           ties.
            oldg=gammln(en+1.0);
            nold=n;
        } if (p != pold) {              If p has changed, then compute useful quanti-
            pc=1.0-p;                       ties.
            plog=log(p);
            pclog=log(pc);
            pold=p;
        }
        sq=sqrt(2.0*am*pc);             The following code should by now seem familiar:
        do {                                rejection method with a Lorentzian compar-
            do {                            ison function.
                angle=PI*ran1(idum);
                y=tan(angle);
                em=sq*y+am;
            } while (em < 0.0 || em >= (en+1.0));       Reject.
            em=floor(em);               Trick for integer-valued distribution.
            t=1.2*sq*(1.0+y*y)*exp(oldg-gammln(em+1.0)
                -gammln(en-em+1.0)+em*plog+(en-em)*pclog);
        } while (ran1(idum) > t);       Reject. This happens about 1.5 times per devi-
        bnl=em;                             ate, on average.
    }
    if (p != pp) bnl=n-bnl;             Remember to undo the symmetry transforma-
    return bnl;                             tion.
}
```

See Devroye [2] and Bratley [3] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 120ff. [1]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §X.4. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3].

7.4 Generation of Random Bits

The C language gives you useful access to some machine-level bitwise operations such as << (left shift). This section will show you how to put such abilities to good use.

The problem is how to generate single random bits, with 0 and 1 equally probable. Of course you can just generate uniform random deviates between zero and one and use their high-order bit (i.e., test if they are greater than or less than 0.5). However this takes a lot of arithmetic; there are special-purpose applications, such as real-time signal processing, where you want to generate bits very much faster than that.

One method for generating random bits, with two variant implementations, is based on "primitive polynomials modulo 2." The theory of these polynomials is beyond our scope (although §7.7 and §20.3 will give you small tastes of it). Here,

```
    psdes(&lword,&irword);                    "Pseudo-DES" encode the words.
    itemp=jflone | (jflmsk & irword);         Mask to a floating number between 1 and
    ++(*idum);                                    2.
    return (*(float *)&itemp)-1.0;            Subtraction moves range to 0. to 1.
}
```

~~The accompanying table gives data for verifying that `ran4` and `psdes` work correctly on your machine. We do not advise the use of `ran4` unless you are able to reproduce the hex values shown. Typically, `ran4` is about 4 times slower than `ran0` (§7.1), or about 3 times slower than `ran1`.~~

~~Values for Verifying the Implementation of `psdes`~~

~~`idum`~~	~~before `psdes` call~~		~~after `psdes` call (hex)~~		~~`ran4(idum)`~~	
	~~`lword`~~	~~`irword`~~	~~`lword`~~	~~`irword`~~	~~VAX~~	~~PC~~
~~−1~~	~~1~~	~~1~~	~~604D1DCE~~	~~509C0C23~~	~~0.275898~~	~~0.219120~~
~~99~~	~~1~~	~~99~~	~~D97F8571~~	~~A66CB41A~~	~~0.208204~~	~~0.849246~~
~~−99~~	~~99~~	~~1~~	~~7822309D~~	~~64300984~~	~~0.034307~~	~~0.375290~~
~~99~~	~~99~~	~~99~~	~~D7F376F0~~	~~59BA89EB~~	~~0.838676~~	~~0.457334~~

~~Successive calls to `ran4` with arguments −1, 99, −99, and 99 should produce exactly the `lword` and `irword` values shown. Masking conversion to a returned floating random value is allowed to be machine dependent; values for VAX and PC are shown.~~

~~CITED REFERENCES AND FURTHER READING:~~

~~*Data Encryption Standard*, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards).~~ ~~[1]~~

~~*Guidelines for Implementing and Using the NBS Data Encryption Standard*, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards).~~ ~~[2]~~

~~*Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard*, 1980, NBS Special Publication 500–20 (Washington: U.S. Department of Commerce, National Bureau of Standards).~~ ~~[3]~~

~~Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley).~~ ~~[4]~~

~~Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 6.~~ ~~[5]~~

~~Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press).~~ ~~[6]~~

7.6 Simple Monte Carlo Integration

Inspirations for numerical methods can spring from unlikely sources. “Splines” first were flexible strips of wood used by draftsmen. “Simulated annealing” (we shall see in §10.9) is rooted in a thermodynamic analogy. And who does not feel at least a faint echo of glamor in the name “Monte Carlo method”?

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V. Call them $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f\, dV \approx V \langle f \rangle \; \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.6.1}$$

Here the angle brackets denote taking the arithmetic mean over the N sample points,

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \left\langle f^2 \right\rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

The "plus-or-minus" term in (7.6.1) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error.

Suppose that you want to integrate a function g over a region W that is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V that *includes* W and that *can* easily be sampled (Figure 7.6.1), and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.6.1). And well they should: points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.6.1) takes this into account.

General purpose routines for Monte Carlo integration are quite complicated (see §7.8), but a worked example will show the underlying simplicity of the method. Suppose that we want to find the weight and the position of the center of mass of an object of complicated shape, namely the intersection of a torus with the edge of a large box. In particular let the object be defined by the three simultaneous conditions

$$z^2 + \left(\sqrt{x^2 + y^2} - 3\right)^2 \le 1 \tag{7.6.3}$$

(torus centered on the origin with major radius $= 4$, minor radius $= 2$)

$$x \ge 1 \qquad y \ge -3 \tag{7.6.4}$$

(two faces of the box, see Figure 7.6.2). Suppose for the moment that the object has a constant density ρ.

We want to estimate the following integrals over the interior of the complicated object:

$$\int \rho\, dx\, dy\, dz \qquad \int x\rho\, dx\, dy\, dz \qquad \int y\rho\, dx\, dy\, dz \qquad \int z\rho\, dx\, dy\, dz \tag{7.6.5}$$

The coordinates of the center of mass will be the ratio of the latter three integrals (linear moments) to the first one (the weight).

In the following fragment, the region V, enclosing the piece-of-torus W, is the rectangular box extending from 1 to 4 in x, -3 to 4 in y, and -1 to 1 in z.

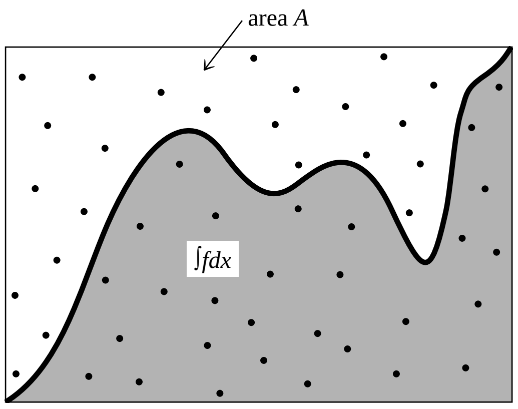


Figure 7.6.1. Monte Carlo integration. Random points are chosen within the area A. The integral of the function f is estimated as the area of A multiplied by the fraction of random points that fall below the curve f. Refinements on this procedure can improve the accuracy of the method; see text.

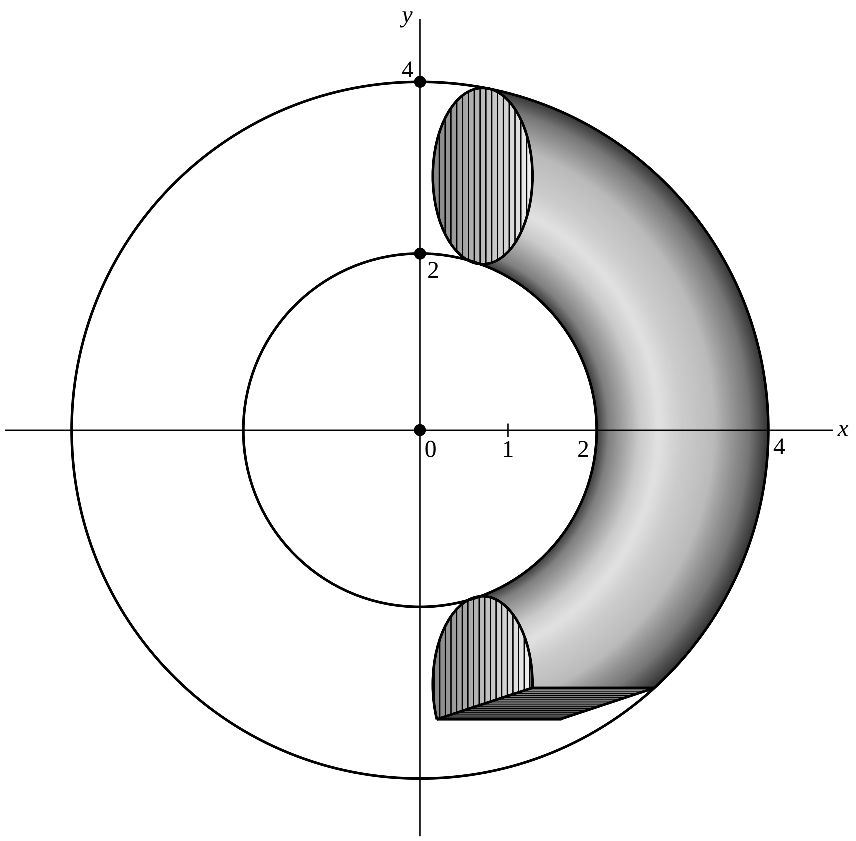


Figure 7.6.2. Example of Monte Carlo integration (see text). The region of interest is a piece of a torus, bounded by the intersection of two planes. The limits of integration of the region cannot easily be written in analytically closed form, so Monte Carlo is a useful technique.

```
#include "nrutil.h"
...
n=...                                     Set to the number of sample points desired.
den=...                                   Set to the constant value of the density.
sw=swx=swy=swz=0.0;                       Zero the various sums to be accumulated.
varw=varx=vary=varz=0.0;
vol=3.0*7.0*2.0;                          Volume of the sampled region.
for(j=1;j<=n;j++) {
    x=1.0+3.0*ran2(&idum);                Pick a point randomly in the sampled re-
    y=(-3.0)+7.0*ran2(&idum);                 gion.
    z=(-1.0)+2.0*ran2(&idum);
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {       Is it in the torus?
        sw += den;                        If so, add to the various cumulants.
        swx += x*den;
        swy += y*den;
        swz += z*den;
        varw += SQR(den);
        varx += SQR(x*den);
        vary += SQR(y*den);
        varz += SQR(z*den);
    }
}
w=vol*sw/n;                               The values of the integrals (7.6.5),
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);        and their corresponding error estimates.
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

A change of variable can often be extremely worthwhile in Monte Carlo integration. Suppose, for example, that we want to evaluate the same integrals, but for a piece-of-torus whose density is a strong function of z, in fact varying according to

$$\rho(x, y, z) = e^{5z} \tag{7.6.6}$$

One way to do this is to put the statement

```
den=exp(5.0*z);
```

inside the `if (...)` block, just before `den` is first used. This will work, but it is a poor way to proceed. Since (7.6.6) falls so rapidly to zero as z decreases (down to its lower limit -1), most sampled points contribute almost nothing to the sum of the weight or moments. These points are effectively wasted, almost as badly as those that fall outside of the region W. A change of variable, exactly as in the transformation methods of §7.2, solves this problem. Let

$$ds = e^{5z}dz \qquad \text{so that} \qquad s = \frac{1}{5}e^{5z}, \quad z = \frac{1}{5}\ln(5s) \tag{7.6.7}$$

Then $\rho dz = ds$, and the limits $-1 < z < 1$ become $.00135 < s < 29.682$. The program fragment now looks like this

```
#include "nrutil.h"
...
n=...                                   Set to the number of sample points desired.
sw=swx=swy=swz=0.0;
varw=varx=vary=varz=0.0;
ss=0.2*(exp(5.0)-exp(-5.0))             Interval of s to be random sampled.
vol=3.0*7.0*ss                          Volume in x,y,s-space.
for(j=1;j<=n;j++) {
    x=1.0+3.0*ran2(&idum);
    y=(-3.0)+7.0*ran2(&idum);
    s=0.00135+ss*ran2(&idum);           Pick a point in s.
    z=0.2*log(5.0*s);                   Equation (7.6.7).
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {
        sw += 1.0;                      Density is 1, since absorbed into definition
        swx += x;                           of s.
        swy += y;
        swz += z;
        varw += 1.0;
        varx += x*x;
        vary += y*y;
        varz += z*z;
    }
}
w=vol*sw/n;                             The values of the integrals (7.6.5),
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);      and their corresponding error estimates.
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

If you think for a minute, you will realize that equation (7.6.7) was useful only because the part of the integrand that we wanted to eliminate (e^{5z}) was both integrable analytically, and had an integral that could be analytically inverted. (Compare §7.2.) In general these properties will not hold. Question: What then? Answer: Pull out of the integrand the "best" factor that *can* be integrated and inverted. The criterion for "best" is to try to reduce the remaining integrand to a function that is as close as possible to constant.

The limiting case is instructive: If you manage to make the integrand f *exactly* constant, and if the region V, of known volume, *exactly* encloses the desired region W, then the average of f that you compute will be exactly its constant value, and the error estimate in equation (7.6.1) will exactly vanish. You will, in fact, have done the integral exactly, and the Monte Carlo numerical evaluations are superfluous. So, backing off from the extreme limiting case, *to the extent* that you are able to make f approximately constant by change of variable, and *to the extent* that you can sample a region only slightly larger than W, you will increase the accuracy of the Monte Carlo integral. This technique is generically called *reduction of variance* in the literature.

The fundamental disadvantage of simple Monte Carlo integration is that its accuracy increases only as the square root of N, the number of sampled points. If your accuracy requirements are modest, or if your computer budget is large, then the technique is highly recommended as one of great generality. In the next two sections we will see that there are techniques available for "breaking the square root of N barrier" and achieving, at least in some cases, higher accuracy with fewer function evaluations.

CITED REFERENCES AND FURTHER READING:
Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).
Shreider, Yu. A. (ed.) 1966, *The Monte Carlo Method* (Oxford: Pergamon).
Sobol', I.M. 1974, *The Monte Carlo Method* (Chicago: University of Chicago Press).
Kalos, M.H., and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

~~7.7 Quasi- (that is, Sub-) Random Sequences~~

~~We have just seen that choosing N points uniformly randomly in an n-dimensional space leads to an error term in Monte Carlo integration that decreases as $1/\sqrt{N}$. In essence, each new point sampled adds linearly to an accumulated sum that will become the function average, and also linearly to an accumulated sum of squares that will become the variance (equation 7.6.2). The estimated error comes from the square root of this variance, hence the power $N^{-1/2}$.~~

~~Just because this square root convergence is familiar does not, however, mean that it is inevitable. A simple counterexample is to choose sample points that lie on a Cartesian grid, and to sample each grid point exactly once (in whatever order). The Monte Carlo method thus becomes a deterministic quadrature scheme — albeit a simple one — whose fractional error decreases at least as fast as N^{-1} (even faster if the function goes to zero smoothly at the boundaries of the sampled region, or is periodic in the region).~~

~~The trouble with a grid is that one has to decide *in advance* how fine it should be. One is then committed to completing all of its sample points. With a grid, it is not convenient to "sample *until*" some convergence or termination criterion is met. One might ask if there is not some intermediate scheme, some way to pick sample points "at random," yet spread out in some self-avoiding way, avoiding the chance clustering that occurs with uniformly random points.~~

~~A similar question arises for tasks other than Monte Carlo integration. We might want to search an n-dimensional space for a point where some (locally computable) condition holds. Of course, for the task to be computationally meaningful, there had better be continuity, so that the desired condition will hold in some finite n-dimensional neighborhood. We may not know *a priori* how large that neighborhood is, however. We want to "sample *until*" the desired point is found, moving smoothly to finer scales with increasing samples. Is there any way to do this that is better than uncorrelated, random samples?~~

~~The answer to the above question is "yes." Sequences of n-tuples that fill n-space more uniformly than uncorrelated random points are called *quasi-random sequences*. That term is somewhat of a misnomer, since there is nothing "random" about quasi-random sequences: They are cleverly crafted to be, in fact, *sub*-random. The sample points in a quasi-random sequence are, in a precise sense, "maximally avoiding" of each other.~~

~~A conceptually simple example is *Halton's sequence* [1]. In one dimension, the jth number H_j in the sequence is obtained by the following steps: (i) Write j as a number in base b, where b is some prime. (For example $j = 17$ in base $b = 3$ is 122.) (ii) Reverse the digits and put a radix point (i.e., a decimal point base b) in front of~~

Chapter 12. Fast Fourier Transform

12.0 Introduction

A very large class of important computational problems falls under the general rubric of "Fourier transform methods" or "spectral methods." For some of these problems, the Fourier transform is simply an efficient computational tool for accomplishing certain common manipulations of data. In other cases, we have problems for which the Fourier transform (or the related "power spectrum") is itself of intrinsic interest. These two kinds of problems share a common methodology.

Largely for historical reasons the literature on Fourier and spectral methods has been disjoint from the literature on "classical" numerical analysis. Nowadays there is no justification for such a split. Fourier methods are commonplace in research and we shall not treat them as specialized or arcane. At the same time, we realize that many computer users have had relatively less experience with this field than with, say, differential equations or numerical integration. Therefore our summary of analytical results will be more complete. Numerical algorithms, per se, begin in §12.2. Various applications of Fourier transform methods are discussed in Chapter 13.

A physical process can be described either in the *time domain*, by the values of some quantity h as a function of time t, e.g., $h(t)$, or else in the *frequency domain*, where the process is specified by giving its amplitude H (generally a complex number indicating phase also) as a function of frequency f, that is $H(f)$, with $-\infty < f < \infty$. For many purposes it is useful to think of $h(t)$ and $H(f)$ as being two different *representations* of the *same* function. One goes back and forth between these two representations by means of the *Fourier transform* equations,

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)e^{2\pi ift}dt \\ h(t) &= \int_{-\infty}^{\infty} H(f)e^{-2\pi ift}df \end{aligned} \tag{12.0.1}$$

If t is measured in seconds, then f in equation (12.0.1) is in cycles per second, or Hertz (the unit of frequency). However, the equations work with other units too. If h is a function of position x (in meters), H will be a function of inverse wavelength (cycles per meter), and so on. If you are trained as a physicist or mathematician, you are probably more used to using *angular frequency* ω, which is given in *radians* per sec. The relation between ω and f, $H(\omega)$ and $H(f)$ is

$$\omega \equiv 2\pi f \qquad H(\omega) \equiv [H(f)]_{f=\omega/2\pi} \tag{12.0.2}$$

and equation (12.0.1) looks like this

$$H(\omega) = \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt$$
$$h(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} H(\omega)e^{-i\omega t}d\omega \qquad (12.0.3)$$

We were raised on the ω-convention, but we changed! There are fewer factors of 2π to remember if you use the f-convention, especially when we get to discretely sampled data in §12.1.

From equation (12.0.1) it is evident at once that Fourier transformation is a *linear* operation. The transform of the sum of two functions is equal to the sum of the transforms. The transform of a constant times a function is that same constant times the transform of the function.

In the time domain, function $h(t)$ may happen to have one or more special symmetries It might be *purely real* or *purely imaginary* or it might be *even*, $h(t) = h(-t)$, or *odd*, $h(t) = -h(-t)$. In the frequency domain, these symmetries lead to relationships between $H(f)$ and $H(-f)$. The following table gives the correspondence between symmetries in the two domains:

If . . .	then . . .
$h(t)$ is real	$H(-f) = [H(f)]^*$
$h(t)$ is imaginary	$H(-f) = -[H(f)]^*$
$h(t)$ is even	$H(-f) = H(f)$ [i.e., $H(f)$ is even]
$h(t)$ is odd	$H(-f) = -H(f)$ [i.e., $H(f)$ is odd]
$h(t)$ is real and even	$H(f)$ is real and even
$h(t)$ is real and odd	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and even	$H(f)$ is imaginary and even
$h(t)$ is imaginary and odd	$H(f)$ is real and odd

In subsequent sections we shall see how to use these symmetries to increase computational efficiency.

Here are some other elementary properties of the Fourier transform. (We'll use the "$\Longleftrightarrow$" symbol to indicate transform pairs.) If

$$h(t) \Longleftrightarrow H(f)$$

is such a pair, then other transform pairs are

$$h(at) \Longleftrightarrow \frac{1}{|a|}H(\frac{f}{a}) \qquad \text{"time scaling"} \qquad (12.0.4)$$

$$\frac{1}{|b|}h(\frac{t}{b}) \Longleftrightarrow H(bf) \qquad \text{"frequency scaling"} \qquad (12.0.5)$$

$$h(t-t_0) \Longleftrightarrow H(f)\, e^{2\pi i f t_0} \qquad \text{"time shifting"} \qquad (12.0.6)$$

$$h(t)\, e^{-2\pi i f_0 t} \Longleftrightarrow H(f-f_0) \qquad \text{"frequency shifting"} \qquad (12.0.7)$$

With two functions $h(t)$ and $g(t)$, and their corresponding Fourier transforms $H(f)$ and $G(f)$, we can form two combinations of special interest. The *convolution* of the two functions, denoted $g * h$, is defined by

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau)h(t-\tau)\, d\tau \tag{12.0.8}$$

Note that $g * h$ is a function in the time domain and that $g * h = h * g$. It turns out that the function $g * h$ is one member of a simple transform pair

$$g * h \Longleftrightarrow G(f)H(f) \qquad \text{“Convolution Theorem”} \tag{12.0.9}$$

In other words, the Fourier transform of the convolution is just the product of the individual Fourier transforms.

The *correlation* of two functions, denoted Corr(g, h), is defined by

$$\text{Corr}(g,h) \equiv \int_{-\infty}^{\infty} g(\tau+t)h(\tau)\, d\tau \tag{12.0.10}$$

The correlation is a function of t, which is called the *lag*. It therefore lies in the time domain, and it turns out to be one member of the transform pair:

$$\text{Corr}(g,h) \Longleftrightarrow G(f)H^*(f) \qquad \text{“Correlation Theorem”} \tag{12.0.11}$$

[More generally, the second member of the pair is $G(f)H(-f)$, but we are restricting ourselves to the usual case in which g and h are real functions, so we take the liberty of setting $H(-f) = H^*(f)$.] This result shows that multiplying the Fourier transform of one function by the complex conjugate of the Fourier transform of the other gives the Fourier transform of their correlation. The correlation of a function with itself is called its *autocorrelation*. In this case (12.0.11) becomes the transform pair

$$\text{Corr}(g,g) \Longleftrightarrow |G(f)|^2 \qquad \text{“Wiener-Khinchin Theorem”} \tag{12.0.12}$$

The *total power* in a signal is the same whether we compute it in the time domain or in the frequency domain. This result is known as *Parseval's theorem*:

$$\text{Total Power} \equiv \int_{-\infty}^{\infty} |h(t)|^2\ dt = \int_{-\infty}^{\infty} |H(f)|^2\ df \tag{12.0.13}$$

Frequently one wants to know “how much power” is contained in the frequency interval between f and $f + df$. In such circumstances one does not usually distinguish between positive and negative f, but rather regards f as varying from 0 (“zero frequency” or D.C.) to $+\infty$. In such cases, one defines the *one-sided power spectral density (PSD)* of the function h as

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \qquad 0 \le f < \infty \tag{12.0.14}$$

so that the total power is just the integral of $P_h(f)$ from $f = 0$ to $f = \infty$. When the function $h(t)$ is real, then the two terms in (12.0.14) are equal, so $P_h(f) = 2\,|H(f)|^2$.

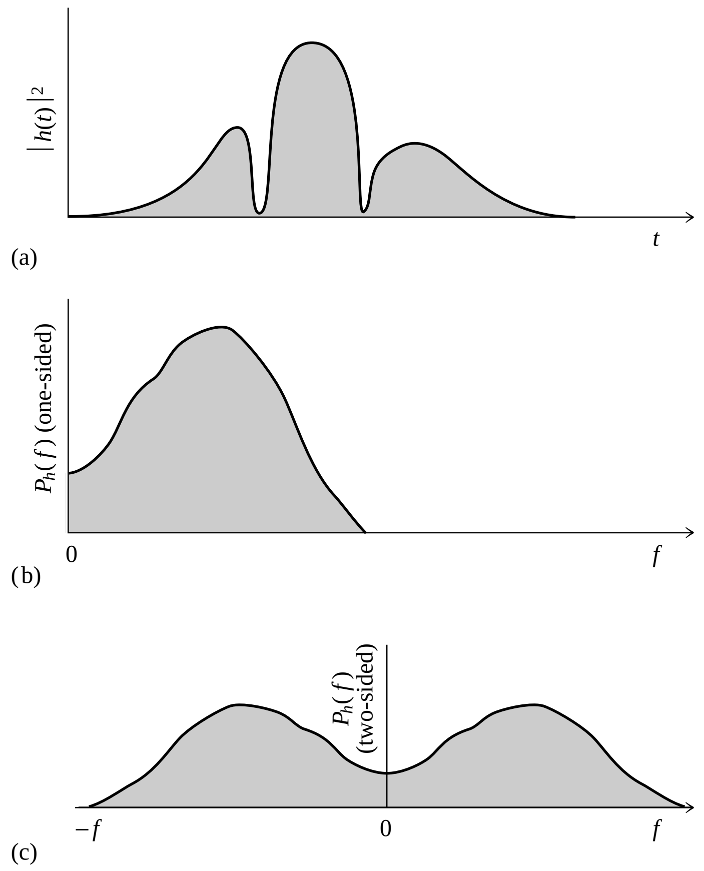


Figure 12.0.1. Normalizations of one- and two-sided power spectra. The area under the square of the function, (a), equals the area under its one-sided power spectrum at positive frequencies, (b), and also equals the area under its two-sided power spectrum at positive and negative frequencies, (c).

Be warned that one occasionally sees PSDs defined without this factor two. These, strictly speaking, are called *two-sided power spectral densities*, but some books are not careful about stating whether one- or two-sided is to be assumed. We will always use the one-sided density given by equation (12.0.14). Figure 12.0.1 contrasts the two conventions.

If the function $h(t)$ goes endlessly from $-\infty < t < \infty$, then its total power and power spectral density will, in general, be infinite. Of interest then is the *(one- or two-sided) power spectral density per unit time*. This is computed by taking a long, but finite, stretch of the function $h(t)$, computing its PSD [that is, the PSD of a function that equals $h(t)$ in the finite stretch but is zero everywhere else], and then dividing the resulting PSD by the length of the stretch used. Parseval's theorem in this case states that the integral of the one-sided PSD-per-unit-time over positive frequency is equal to the mean square amplitude of the signal $h(t)$.

You might well worry about how the PSD-per-unit-time, which is a function of frequency f, converges as one evaluates it using longer and longer stretches of data. This interesting question is the content of the subject of "power spectrum estimation," and will be considered below in §13.4–§13.7. A crude answer for now is: The

PSD-per-unit-time converges to finite values at all frequencies *except* those where $h(t)$ has a discrete sine-wave (or cosine-wave) component of finite amplitude. At those frequencies, it becomes a delta-function, i.e., a sharp spike, whose width gets narrower and narrower, but whose area converges to be the mean square amplitude of the discrete sine or cosine component at that frequency.

We have by now stated all of the analytical formalism that we will need in this chapter with one exception: In computational work, especially with experimental data, we are almost never given a continuous function $h(t)$ to work with, but are given, rather, a list of measurements of $h(t_i)$ for a discrete set of t_i's. The profound implications of this seemingly unimportant fact are the subject of the next section.

CITED REFERENCES AND FURTHER READING:

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.1 Fourier Transform of Discretely Sampled Data

In the most common situations, function $h(t)$ is sampled (i.e., its value is recorded) at evenly spaced intervals in time. Let Δ denote the time interval between consecutive samples, so that the sequence of sampled values is

$$h_n = h(n\Delta) \qquad n = \dots, -3, -2, -1, 0, 1, 2, 3, \dots \tag{12.1.1}$$

The reciprocal of the time interval Δ is called the *sampling rate*; if Δ is measured in seconds, for example, then the sampling rate is the number of samples recorded per second.

Sampling Theorem and Aliasing

For any sampling interval Δ, there is also a special frequency f_c, called the *Nyquist critical frequency*, given by

$$f_c \equiv \frac{1}{2\Delta} \tag{12.1.2}$$

If a sine wave of the Nyquist critical frequency is sampled at its positive peak value, then the next sample will be at its negative trough value, the sample after that at the positive peak again, and so on. Expressed otherwise: *Critical sampling of a sine wave is two sample points per cycle.* One frequently chooses to measure time in units of the sampling interval Δ. In this case the Nyquist critical frequency is just the constant 1/2.

The Nyquist critical frequency is important for two related, but distinct, reasons. One is good news, and the other bad news. First the good news. It is the remarkable fact known as the *sampling theorem*: If a continuous function $h(t)$, sampled at an interval Δ, happens to be *bandwidth limited* to frequencies smaller in magnitude than f_c, i.e., if $H(f) = 0$ for all $|f| \geq f_c$, then the function $h(t)$ is *completely determined* by its samples h_n. In fact, $h(t)$ is given explicitly by the formula

$$h(t) = \Delta \sum_{n=-\infty}^{+\infty} h_n \frac{\sin[2\pi f_c(t - n\Delta)]}{\pi(t - n\Delta)} \tag{12.1.3}$$

This is a remarkable theorem for many reasons, among them that it shows that the "information content" of a bandwidth limited function is, in some sense, infinitely smaller than that of a general continuous function. Fairly often, one is dealing with a signal that is known on physical grounds to be bandwidth limited (or at least approximately bandwidth limited). For example, the signal may have passed through an amplifier with a known, finite frequency response. In this case, the sampling theorem tells us that the entire information content of the signal can be recorded by sampling it at a rate Δ^{-1} equal to twice the maximum frequency passed by the amplifier (cf. 12.1.2).

Now the bad news. The bad news concerns the effect of sampling a continuous function that is *not* bandwidth limited to less than the Nyquist critical frequency. In that case, it turns out that all of the power spectral density that lies outside of the frequency range $-f_c < f < f_c$ is spuriously moved into that range. This phenomenon is called *aliasing*. Any frequency component outside of the frequency range $(-f_c, f_c)$ is *aliased* (falsely translated) into that range by the very act of discrete sampling. You can readily convince yourself that two waves $\exp(2\pi i f_1 t)$ and $\exp(2\pi i f_2 t)$ give the same samples at an interval Δ if and only if f_1 and f_2 differ by a multiple of $1/\Delta$, which is just the width in frequency of the range $(-f_c, f_c)$. There is little that you can do to remove aliased power once you have discretely sampled a signal. The way to overcome aliasing is to (i) know the natural bandwidth limit of the signal — or else enforce a known limit by analog filtering of the continuous signal, and then (ii) sample at a rate sufficiently rapid to give at least two points per cycle of the highest frequency present. Figure 12.1.1 illustrates these considerations.

To put the best face on this, we can take the alternative point of view: If a continuous function has been competently sampled, then, when we come to estimate its Fourier transform from the discrete samples, we can *assume* (or rather we *might as well* assume) that its Fourier transform is equal to zero outside of the frequency range in between $-f_c$ and f_c. Then we look to the Fourier transform to tell whether the continuous function *has* been competently sampled (aliasing effects minimized). We do this by looking to see whether the Fourier transform is already approaching zero as the frequency approaches f_c from below, or $-f_c$ from above. If, on the contrary, the transform is going towards some finite value, then chances are that components outside of the range have been folded back over onto the critical range.

Discrete Fourier Transform

We now estimate the Fourier transform of a function from a finite number of its sampled points. Suppose that we have N consecutive sampled values

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1 \tag{12.1.4}$$

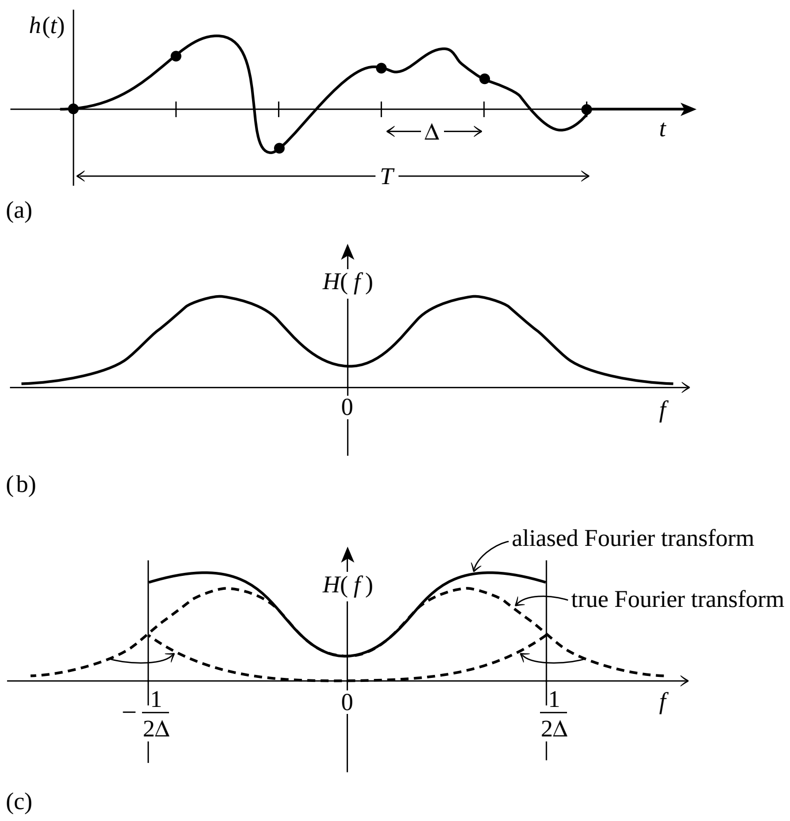


Figure 12.1.1. The continuous function shown in (a) is nonzero only for a finite interval of time T. It follows that its Fourier transform, whose modulus is shown schematically in (b), is not bandwidth limited but has finite amplitude for all frequencies. If the original function is sampled with a sampling interval Δ, as in (a), then the Fourier transform (c) is defined only between plus and minus the Nyquist critical frequency. Power outside that range is folded over or "aliased" into the range. The effect can be eliminated only by low-pass filtering the original function *before sampling*.

so that the sampling interval is Δ. To make things simpler, let us also suppose that N is even. If the function $h(t)$ is nonzero only in a finite interval of time, then that whole interval of time is supposed to be contained in the range of the N points given. Alternatively, if the function $h(t)$ goes on forever, then the sampled points are supposed to be at least "typical" of what $h(t)$ looks like at all other times.

With N numbers of input, we will evidently be able to produce no more than N independent numbers of output. So, instead of trying to estimate the Fourier transform $H(f)$ at all values of f in the range $-f_c$ to f_c, let us seek estimates only at the discrete values

$$f_n \equiv \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2} \tag{12.1.5}$$

The extreme values of n in (12.1.5) correspond exactly to the lower and upper limits of the Nyquist critical frequency range. If you are really on the ball, you will have noticed that there are $N + 1$, not N, values of n in (12.1.5); it will turn out that the two extreme values of n are not independent (in fact they are equal), but all the others are. This reduces the count to N.

The remaining step is to approximate the integral in (12.0.1) by a discrete sum:

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} h_k \; e^{2\pi i f_n t_k}\Delta = \Delta \sum_{k=0}^{N-1} h_k \; e^{2\pi i k n/N} \tag{12.1.6}$$

Here equations (12.1.4) and (12.1.5) have been used in the final equality. The final summation in equation (12.1.6) is called the *discrete Fourier transform* of the N points h_k. Let us denote it by H_n,

$$H_n \equiv \sum_{k=0}^{N-1} h_k \; e^{2\pi i k n/N} \tag{12.1.7}$$

The discrete Fourier transform maps N complex numbers (the h_k's) into N complex numbers (the H_n's). It does not depend on any dimensional parameter, such as the time scale Δ. The relation (12.1.6) between the discrete Fourier transform of a set of numbers and their continuous Fourier transform when they are viewed as samples of a continuous function sampled at an interval Δ can be rewritten as

$$H(f_n) \approx \Delta H_n \tag{12.1.8}$$

where f_n is given by (12.1.5).

Up to now we have taken the view that the index n in (12.1.7) varies from $-N/2$ to $N/2$ (cf. 12.1.5). You can easily see, however, that (12.1.7) is periodic in n, with period N. Therefore, $H_{-n} = H_{N-n} \quad n = 1, 2, \ldots$. With this conversion in mind, one generally lets the n in H_n vary from 0 to $N-1$ (one complete period). Then n and k (in h_k) vary exactly over the same range, so the mapping of N numbers into N numbers is manifest. When this convention is followed, you must remember that zero frequency corresponds to $n = 0$, positive frequencies $0 < f < f_c$ correspond to values $1 \leq n \leq N/2 - 1$, while negative frequencies $-f_c < f < 0$ correspond to $N/2+1 \leq n \leq N-1$. The value $n = N/2$ corresponds to *both* $f = f_c$ and $f = -f_c$.

The discrete Fourier transform has symmetry properties almost exactly the same as the continuous Fourier transform. For example, all the symmetries in the table following equation (12.0.3) hold if we read h_k for $h(t)$, H_n for $H(f)$, and H_{N-n} for $H(-f)$. (Likewise, "even" and "odd" in time refer to whether the values h_k at k and $N-k$ are identical or the negative of each other.)

The formula for the discrete *inverse* Fourier transform, which recovers the set of h_k's exactly from the H_n's is:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \; e^{-2\pi i k n/N} \tag{12.1.9}$$

Notice that the only differences between (12.1.9) and (12.1.7) are (i) changing the sign in the exponential, and (ii) dividing the answer by N. This means that a routine for calculating discrete Fourier transforms can also, with slight modification, calculate the inverse transforms.

The discrete form of Parseval's theorem is

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{12.1.10}$$

There are also discrete analogs to the convolution and correlation theorems (equations 12.0.9 and 12.0.11), but we shall defer them to §13.1 and §13.2, respectively.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.2 Fast Fourier Transform (FFT)

How much computation is involved in computing the discrete Fourier transform (12.1.7) of N points? For many years, until the mid-1960s, the standard answer was this: Define W as the complex number

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

Then (12.1.7) can be written as

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

In other words, the vector of h_k's is multiplied by a matrix whose (n,k)th element is the constant W to the power $n \times k$. The matrix multiplication produces a vector result whose components are the H_n's. This matrix multiplication evidently requires N^2 complex multiplications, plus a smaller number of operations to generate the required powers of W. So, the discrete Fourier transform appears to be an $O(N^2)$ process. These appearances are deceiving! The discrete Fourier transform can, in fact, be computed in $O(N \log_2 N)$ operations with an algorithm called the *fast Fourier transform,* or *FFT.* The difference between $N \log_2 N$ and N^2 is immense. With $N = 10^6$, for example, it is the difference between, roughly, 30 seconds of CPU time and 2 weeks of CPU time on a microsecond cycle time computer. The existence of an FFT algorithm became generally known only in the mid-1960s, from the work of J.W. Cooley and J.W. Tukey. Retrospectively, we now know (see [1]) that efficient methods for computing the DFT had been independently discovered, and in some cases implemented, by as many as a dozen individuals, starting with Gauss in 1805!

One "rediscovery" of the FFT, that of Danielson and Lanczos in 1942, provides one of the clearest derivations of the algorithm. Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length $N/2$. One of the two is formed from the even-numbered points of the original N, the other from the odd-numbered points. The proof is simply this:

$$\begin{aligned}F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\ &= F_k^e + W^k\, F_k^o\end{aligned} \tag{12.2.3}$$

In the last line, W is the same complex constant as in (12.2.1), F_k^e denotes the kth component of the Fourier transform of length $N/2$ formed from the even components of the original f_j's, while F_k^o is the corresponding transform of length $N/2$ formed from the odd components. Notice also that k in the last line of (12.2.3) varies from 0 to N, not just to $N/2$. Nevertheless, the transforms F_k^e and F_k^o are periodic in k with length $N/2$. So each is repeated through two cycles to obtain F_k.

The wonderful thing about the *Danielson-Lanczos Lemma* is that it can be used recursively. Having reduced the problem of computing F_k to that of computing F_k^e and F_k^o, we can do the same reduction of F_k^e to the problem of computing the transform of *its* $N/4$ even-numbered input data and $N/4$ odd-numbered data. In other words, we can define F_k^{ee} and F_k^{eo} to be the discrete Fourier transforms of the points which are respectively even-even and even-odd on the successive subdivisions of the data.

Although there are ways of treating other cases, by far the easiest case is the one in which the original N is an integer power of 2. In fact, we categorically recommend that you *only* use FFTs with N a power of two. If the length of your data set is not a power of two, pad it with zeros up to the next power of two. (We will give more sophisticated suggestions in subsequent sections below.) With this restriction on N, it is evident that we can continue applying the Danielson-Lanczos Lemma until we have subdivided the data all the way down to transforms of length 1. What is the Fourier transform of length one? It is just the identity operation that copies its one input number into its one output slot! In other words, for every pattern of $\log_2 N$ e's and o's, there is a one-point transform that is just one of the input numbers f_n

$$F_k^{eoeeoeo\cdots oee} = f_n \qquad \text{for some } n \tag{12.2.4}$$

(Of course this one-point transform actually does not depend on k, since it is periodic in k with period 1.)

The next trick is to figure out which value of n corresponds to which pattern of e's and o's in equation (12.2.4). The answer is: Reverse the pattern of e's and o's, then let $e = 0$ and $o = 1$, and you will have, *in binary* the value of n. Do you see why it works? It is because the successive subdivisions of the data into even and odd are tests of successive low-order (least significant) bits of n. This idea of *bit reversal* can be exploited in a very clever way which, along with the Danielson-Lanczos

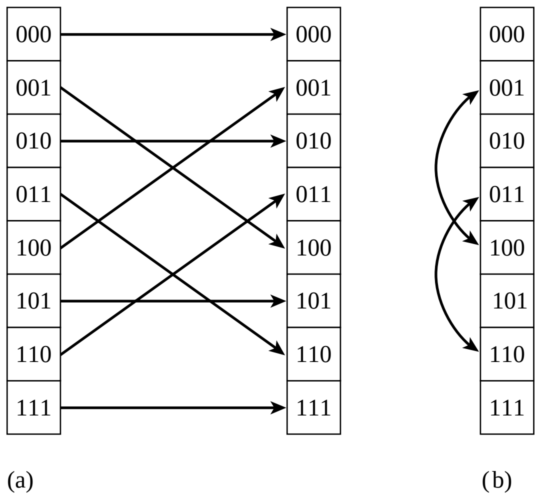


Figure 12.2.1. Reordering an array (here of length 8) by bit reversal, (a) between two arrays, versus (b) in place. Bit reversal reordering is a necessary part of the fast Fourier transform (FFT) algorithm.

Lemma, makes FFTs practical: Suppose we take the original vector of data f_j and rearrange it into bit-reversed order (see Figure 12.2.1), so that the individual numbers are in the order not of j, but of the number obtained by bit-reversing j. Then the bookkeeping on the recursive application of the Danielson-Lanczos Lemma becomes extraordinarily simple. The points as given are the one-point transforms. We combine adjacent pairs to get two-point transforms, then combine adjacent pairs of pairs to get 4-point transforms, and so on, until the first and second halves of the whole data set are combined into the final transform. Each combination takes of order N operations, and there are evidently $\log_2 N$ combinations, so the whole algorithm is of order $N \log_2 N$ (assuming, as is the case, that the process of sorting into bit-reversed order is no greater in order than $N \log_2 N$).

This, then, is the structure of an FFT algorithm: It has two sections. The first section sorts the data into bit-reversed order. Luckily this takes no additional storage, since it involves only swapping pairs of elements. (If k_1 is the bit reverse of k_2, then k_2 is the bit reverse of k_1.) The second section has an outer loop that is executed $\log_2 N$ times and calculates, in turn, transforms of length $2, 4, 8, \ldots, N$. For each stage of this process, two nested inner loops range over the subtransforms already computed and the elements of each transform, implementing the Danielson-Lanczos Lemma. The operation is made more efficient by restricting external calls for trigonometric sines and cosines to the outer loop, where they are made only $\log_2 N$ times. Computation of the sines and cosines of multiple angles is through simple recurrence relations in the inner loops (cf. 5.5.6).

The FFT routine given below is based on one originally written by N. M. Brenner. The input quantities are the number of complex data points (nn), the data array (data[1..2*nn]), and isign, which should be set to either ± 1 and is the sign of i in the exponential of equation (12.1.7). When isign is set to -1, the routine thus calculates the inverse transform (12.1.9) — except that it does not multiply by the normalizing factor $1/N$ that appears in that equation. You can do that yourself.

Notice that the argument nn is the number of *complex* data points. The actual length of the real array (data[1..2*nn]) is 2 times nn, with each complex value occupying two consecutive locations. In other words, data[1] is the real part of f_0, data[2] is the imaginary part of f_0, and so on up to data[2*nn-1], which is the real part of f_{N-1}, and data[2*nn], which is the imaginary part of f_{N-1}. The FFT routine gives back the F_n's packed in exactly the same fashion, as nn complex numbers.

The real and imaginary parts of the zero frequency component F_0 are in data[1] and data[2]; the smallest nonzero positive frequency has real and imaginary parts in data[3] and data[4]; the smallest (in magnitude) nonzero negative frequency has real and imaginary parts in data[2*nn-1] and data[2*nn]. Positive frequencies increasing in magnitude are stored in the real-imaginary pairs data[5], data[6] up to data[nn-1], data[nn]. Negative frequencies of increasing magnitude are stored in data[2*nn-3], data[2*nn-2] down to data[nn+3], data[nn+4]. Finally, the pair data[nn+1], data[nn+2] contain the real and imaginary parts of the one aliased point that contains the most positive and the most negative frequency. You should try to develop a familiarity with this storage arrangement of complex spectra, also shown in Figure 12.2.2, since it is the practical standard.

This is a good place to remind you that you can also use a routine like four1 *without modification* even if your input data array is zero-offset, that is has the range data[0..2*nn-1]. In this case, simply decrement the pointer to data by one when four1 is invoked, e.g., four1(data-1,1024,1);. The real part of f_0 will now be returned in data[0], the imaginary part in data[1], and so on. See §1.2.

```
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(float data[], unsigned long nn, int isign)
```

Replaces data[1..2*nn] by its discrete Fourier transform, if isign is input as 1; or replaces data[1..2*nn] by nn times its inverse discrete Fourier transform, if isign is input as -1. data is a complex array of length nn or, equivalently, a real array of length 2*nn. nn MUST be an integer power of 2 (this is not checked for!).

```
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;          Double precision for the trigonomet-
    float tempr,tempi;                             ric recurrences.

    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {                       This is the bit-reversal section of the
        if (j > i) {                               routine.
            SWAP(data[j],data[i]);             Exchange the two complex numbers.
            SWAP(data[j+1],data[i+1]);
        }
        m=nn;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    Here begins the Danielson-Lanczos section of the routine.
    mmax=2;
    while (n > mmax) {                         Outer loop executed log2 nn times.
        istep=mmax << 1;
        theta=isign*(6.28318530717959/mmax);   Initialize the trigonometric recurrence.
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
```

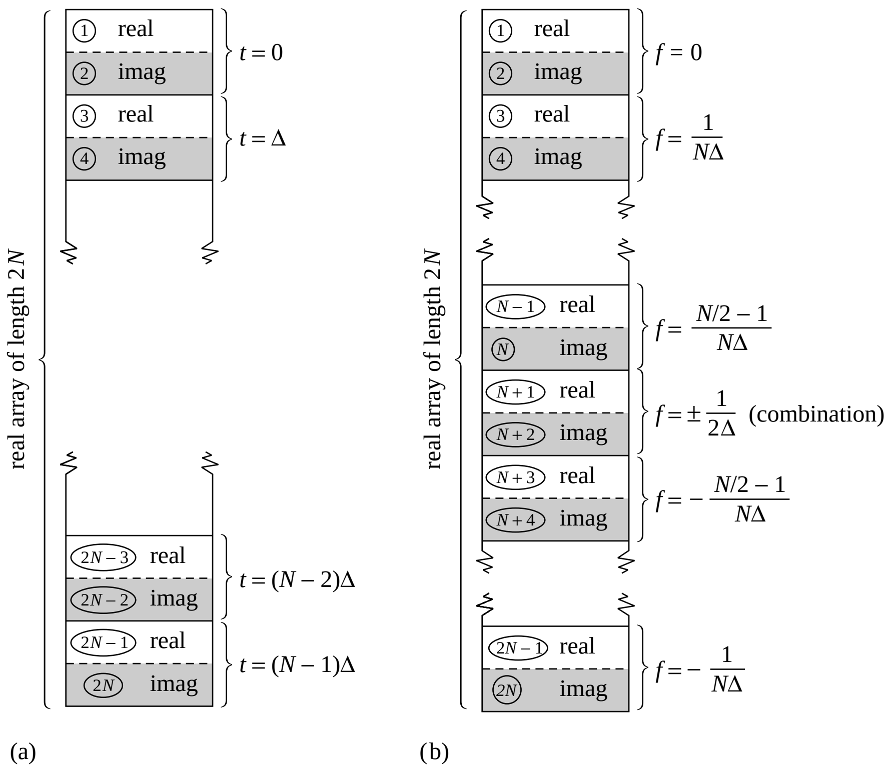


Figure 12.2.2. Input and output arrays for FFT. (a) The input array contains N (a power of 2) complex time samples in a real array of length $2N$, with real and imaginary parts alternating. (b) The output array contains the complex Fourier spectrum at N values of frequency. Real and imaginary parts again alternate. The array starts with zero frequency, works up to the most positive frequency (which is ambiguous with the most negative frequency). Negative frequencies follow, from the second-most negative up to the frequency just below zero.

```
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {                    Here are the two nested inner loops.
            for (i=m;i<=n;i+=istep) {
                j=i+mmax;                          This is the Danielson-Lanczos for-
                tempr=wr*data[j]-wi*data[j+1];         mula:
                tempi=wr*data[j+1]+wi*data[j];
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;           Trigonometric recurrence.
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

(A double precision version of `four1`, named `dfour1`, is used by the routine `mpmul` in §20.6. You can easily make the conversion, or else get the converted routine from the *Numerical Recipes* diskette.)

Chapter 13. Fourier and Spectral Applications

13.0 Introduction

Fourier methods have revolutionized fields of science and engineering, from radio astronomy to medical imaging, from seismology to spectroscopy. In this chapter, we present some of the basic applications of Fourier and spectral methods that have made these revolutions possible.

Say the word "Fourier" to a numericist, and the response, as if by Pavlovian conditioning, will likely be "FFT." Indeed, the wide application of Fourier methods must be credited principally to the existence of the fast Fourier transform. Better mousetraps stand aside: If you speed up *any* nontrivial algorithm by a factor of a million or so, the world will beat a path towards finding useful applications for it. The most direct applications of the FFT are to the convolution or deconvolution of data (§13.1), correlation and autocorrelation (§13.2), optimal filtering (§13.3), power spectrum estimation (§13.4), and the computation of Fourier integrals (§13.9).

As important as they are, however, FFT methods are not the be-all and end-all of spectral analysis. Section 13.5 is a brief introduction to the field of time-domain digital filters. In the spectral domain, one limitation of the FFT is that it always represents a function's Fourier transform as a polynomial in $z = \exp(2\pi i f \Delta)$ (cf. equation 12.1.7). Sometimes, processes have spectra whose shapes are not well represented by this form. An alternative form, which allows the spectrum to have poles in z, is used in the techniques of linear prediction (§13.6) and maximum entropy spectral estimation (§13.7).

Another significant limitation of all FFT methods is that they require the input data to be sampled at evenly spaced intervals. For irregularly or incompletely sampled data, other (albeit slower) methods are available, as discussed in §13.8.

So-called wavelet methods inhabit a representation of function space that is neither in the temporal, nor in the spectral, domain, but rather something in-between. Section 13.10 is an introduction to this subject. Finally §13.11 is an excursion into numerical use of the Fourier sampling theorem.

13.1 Convolution and Deconvolution Using the FFT

We have defined the *convolution* of two functions for the continuous case in equation (12.0.8), and have given the *convolution theorem* as equation (12.0.9). The theorem says that the Fourier transform of the convolution of two functions is equal to the product of their individual Fourier transforms. Now, we want to deal with the discrete case. We will mention first the context in which convolution is a useful procedure, and then discuss how to compute it efficiently using the FFT.

The convolution of two functions $r(t)$ and $s(t)$, denoted $r * s$, is mathematically equal to their convolution in the opposite order, $s * r$. Nevertheless, in most applications the two functions have quite different meanings and characters. One of the functions, say s, is typically a signal or data stream, which goes on indefinitely in time (or in whatever the appropriate independent variable may be). The other function r is a "response function," typically a peaked function that falls to zero in both directions from its maximum. The effect of convolution is to smear the signal $s(t)$ in time according to the recipe provided by the response function $r(t)$, as shown in Figure 13.1.1. In particular, a spike or delta-function of unit area in s which occurs at some time t_0 is supposed to be smeared into the shape of the response function itself, but translated from time 0 to time t_0 as $r(t - t_0)$.

In the discrete case, the signal $s(t)$ is represented by its sampled values at equal time intervals s_j. The response function is also a discrete set of numbers r_k, with the following interpretation: r_0 tells what multiple of the input signal in one channel (one particular value of j) is copied into the identical output channel (same value of j); r_1 tells what multiple of input signal in channel j is additionally copied into output channel $j + 1$; r_{-1} tells the multiple that is copied into channel $j - 1$; and so on for both positive and negative values of k in r_k. Figure 13.1.2 illustrates the situation.

Example: a response function with $r_0 = 1$ and all other r_k's equal to zero is just the identity filter: convolution of a signal with this response function gives identically the signal. Another example is the response function with $r_{14} = 1.5$ and all other r_k's equal to zero. This produces convolved output that is the input signal multiplied by 1.5 and delayed by 14 sample intervals.

Evidently, we have just described in words the following definition of discrete convolution with a response function of finite duration M:

$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k}\, r_k \tag{13.1.1}$$

If a discrete response function is nonzero only in some range $-M/2 < k \le M/2$, where M is a sufficiently large even integer, then the response function is called a *finite impulse response (FIR)*, and its *duration* is M. (Notice that we are defining M as the number of nonzero *values* of r_k; these values span a time interval of $M - 1$ sampling times.) In most practical circumstances the case of finite M is the case of interest, either because the response really has a finite duration, or because we choose to truncate it at some point and approximate it by a finite-duration response function.

The *discrete convolution theorem* is this: If a signal s_j is *periodic* with period N, so that it is completely determined by the N values $s_0, \ldots, s_{N-1}$, then its

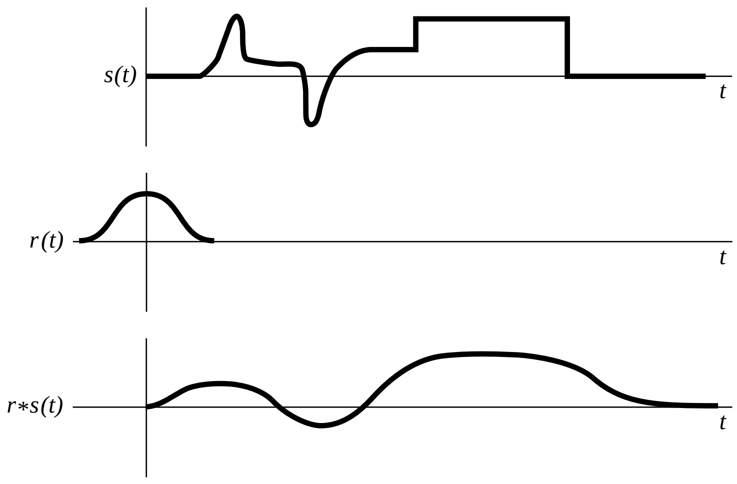


Figure 13.1.1. Example of the convolution of two functions. A signal $s(t)$ is convolved with a response function $r(t)$. Since the response function is broader than some features in the original signal, these are "washed out" in the convolution. In the absence of any additional noise, the process can be reversed by deconvolution.

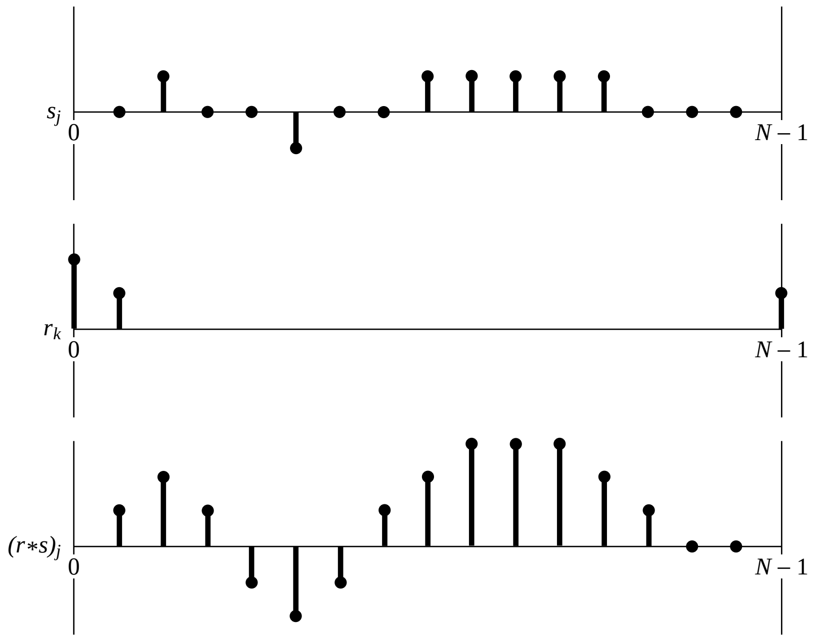


Figure 13.1.2. Convolution of discretely sampled functions. Note how the response function for negative times is wrapped around and stored at the extreme right end of the array r_k.

discrete convolution with a response function *of finite duration* N is a member of the discrete Fourier transform pair,

$$\sum_{k=-N/2+1}^{N/2} s_{j-k}\, r_k \quad \Longleftrightarrow \quad S_n R_n \tag{13.1.2}$$

Here $S_n,\ (n = 0, \dots, N-1)$ is the discrete Fourier transform of the values $s_j,\ (j = 0, \dots, N-1)$, while $R_n,\ (n = 0, \dots, N-1)$ is the discrete Fourier transform of the values $r_k,\ (k = 0, \dots, N-1)$. These values of r_k are the same ones as for the range $k = -N/2+1, \dots, N/2$, but in wrap-around order, exactly as was described at the end of §12.2.

Treatment of End Effects by Zero Padding

The discrete convolution theorem presumes a set of two circumstances that are not universal. First, it assumes that the input signal is periodic, whereas real data often either go forever without repetition or else consist of one nonperiodic stretch of finite length. Second, the convolution theorem takes the duration of the response to be the same as the period of the data; they are both N. We need to work around these two constraints.

The second is very straightforward. Almost always, one is interested in a response function whose duration M is much shorter than the length of the data set N. In this case, you simply extend the response function to length N by padding it with zeros, i.e., define $r_k = 0$ for $M/2 \le k \le N/2$ and also for $-N/2+1 \le k \le -M/2+1$. Dealing with the first constraint is more challenging. Since the convolution theorem rashly assumes that the data are periodic, it will falsely "pollute" the first output channel $(r * s)_0$ with some wrapped-around data from the far end of the data stream s_{N-1}, s_{N-2}, etc. (See Figure 13.1.3.) So, we need to set up a buffer zone of zero-padded values at the end of the s_j vector, in order to make this pollution zero. How many zero values do we need in this buffer? Exactly as many as the most negative index for which the response function is nonzero. For example, if r_{-3} is nonzero, while $r_{-4}, r_{-5}, \dots$ are all zero, then we need three zero pads at the end of the data: $s_{N-3} = s_{N-2} = s_{N-1} = 0$. These zeros will protect the first output channel $(r * s)_0$ from wrap-around pollution. It should be obvious that the second output channel $(r * s)_1$ and subsequent ones will also be protected by these same zeros. Let K denote the number of padding zeros, so that the last actual input data point is s_{N-K-1}.

What now about pollution of the very *last* output channel? Since the data now end with s_{N-K-1}, the last output channel of interest is $(r * s)_{N-K-1}$. This channel can be polluted by wrap-around from input channel s_0 unless the number K is also large enough to take care of the most positive index k for which the response function r_k is nonzero. For example, if r_0 through r_6 are nonzero, while $r_7, r_8 \dots$ are all zero, then we need at least $K = 6$ padding zeros at the end of the data: $s_{N-6} = \dots = s_{N-1} = 0$.

To summarize — we need to pad the data with a number of zeros *on one end* equal to the maximum positive duration *or* maximum negative duration of the response function, *whichever is larger*. (For a symmetric response function of duration M, you will need only $M/2$ zero pads.) Combining this operation with the

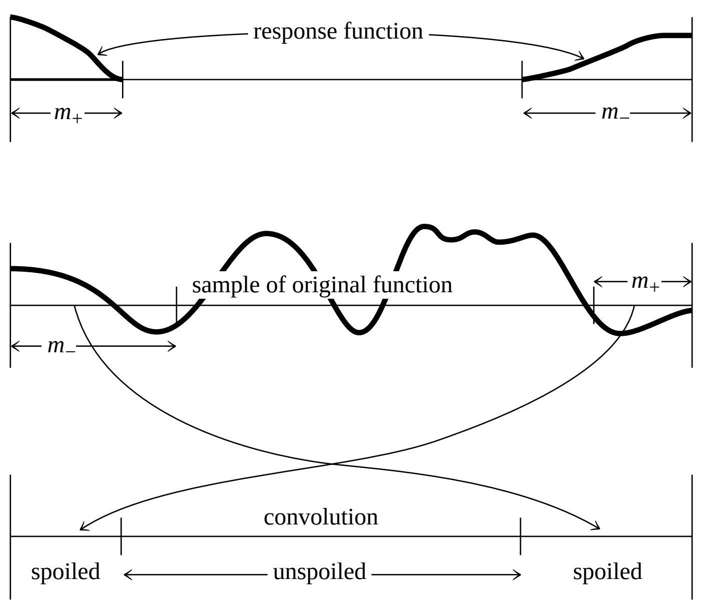


Figure 13.1.3. The wrap-around problem in convolving finite segments of a function. Not only must the response function wrap be viewed as cyclic, but so must the sampled original function. Therefore a portion at each end of the original function is erroneously wrapped around by convolution with the response function.

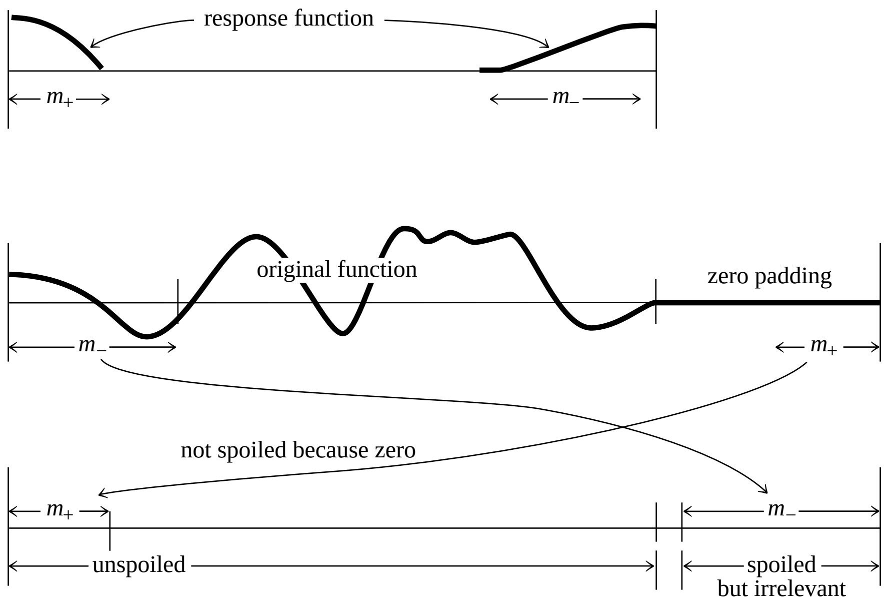


Figure 13.1.4. Zero padding as solution to the wrap-around problem. The original function is extended by zeros, serving a dual purpose: When the zeros wrap around, they do not disturb the true convolution; and while the original function wraps around onto the zero region, that region can be discarded.

padding of the response r_k described above, we effectively insulate the data from artifacts of undesired periodicity. Figure 13.1.4 illustrates matters.

Use of FFT for Convolution

The data, complete with zero padding, are now a set of real numbers s_j, $j = 0, \ldots, N-1$, and the response function is zero padded out to duration N and arranged in wrap-around order. (Generally this means that a large contiguous section of the r_k's, in the middle of that array, is zero, with nonzero values clustered at the two extreme ends of the array.) You now compute the discrete convolution as follows: Use the FFT algorithm to compute the discrete Fourier transform of s and of r. Multiply the two transforms together component by component, remembering that the transforms consist of complex numbers. Then use the FFT algorithm to take the inverse discrete Fourier transform of the products. The answer is the convolution $r*s$.

What about *deconvolution*? Deconvolution is the process of *undoing* the smearing in a data set that has occurred under the influence of a known response function, for example, because of the known effect of a less-than-perfect measuring apparatus. The defining equation of deconvolution is the same as that for convolution, namely (13.1.1), except now the left-hand side is taken to be known, and (13.1.1) is to be considered as a set of N linear equations for the unknown quantities s_j. Solving these simultaneous linear equations in the time domain of (13.1.1) is unrealistic in most cases, but the FFT renders the problem almost trivial. Instead of multiplying the transform of the signal and response to get the transform of the convolution, we just divide the transform of the (known) convolution by the transform of the response to get the transform of the deconvolved signal.

This procedure can go wrong *mathematically* if the transform of the response function is exactly zero for some value R_n, so that we can't divide by it. This indicates that the original convolution has truly lost all information at that one frequency, so that a reconstruction of that frequency component is not possible. You should be aware, however, that apart from mathematical problems, the process of deconvolution has other practical shortcomings. The process is generally quite sensitive to noise in the input data, and to the accuracy to which the response function r_k is known. Perfectly reasonable attempts at deconvolution can sometimes produce nonsense for these reasons. In such cases you may want to make use of the additional process of *optimal filtering*, which is discussed in §13.3.

Here is our routine for convolution and deconvolution, using the FFT as implemented in `four1` of §12.2. Since the data and response functions are real, not complex, both of their transforms can be taken simultaneously using `twofft`. Note, however, that two calls to `realft` should be substituted if `data` and `respns` have very different magnitudes, to minimize roundoff. The data are assumed to be stored in a `float` array `data[1..n]`, with `n` an integer power of two. The response function is assumed to be stored in wrap-around order in a sub-array `respns[1..m]` of the array `respns[1..n]`. The value of `m` can be any *odd* integer less than or equal to `n`, since the first thing the program does is to recopy the response function into the appropriate wrap-around order in `respns[1..n]`. The answer is provided in `ans`.

```
#include "nrutil.h"

void convlv(float data[], unsigned long n, float respns[], unsigned long m,
    int isign, float ans[])
```

Convolves or deconvolves a real data set `data[1..n]` (including any user-supplied zero padding) with a response function `respns[1..n]`. The response function must be stored in wrap-around order in the first `m` elements of `respns`, where `m` is an odd integer $\leq$ `n`. Wrap-around order means that the first half of the array `respns` contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element `respns[m]`. On input `isign` is $+1$ for convolution, -1 for deconvolution. The answer is returned in the first `n` components of `ans`. However, `ans` must be supplied in the calling program with dimensions `[1..2*n]`, for consistency with `twofft`. `n` MUST be an integer power of two.

```
{
    void realft(float data[], unsigned long n, int isign);
    void twofft(float data1[], float data2[], float fft1[], float fft2[],
        unsigned long n);
    unsigned long i,no2;
    float dum,mag2,*fft;

    fft=vector(1,n<<1);
    for (i=1;i<=(m-1)/2;i++)                Put respns in array of length n.
        respns[n+1-i]=respns[m+1-i];
    for (i=(m+3)/2;i<=n-(m-1)/2;i++)        Pad with zeros.
        respns[i]=0.0;
    twofft(data,respns,fft,ans,n);          FFT both at once.
    no2=n>>1;
    for (i=2;i<=n+2;i+=2) {
        if (isign == 1) {
            ans[i-1]=(fft[i-1]*(dum=ans[i-1])-fft[i]*ans[i])/no2;      Multiply FFTs
            ans[i]=(fft[i]*dum+fft[i-1]*ans[i])/no2;                   to convolve.
        } else if (isign == -1) {
            if ((mag2=SQR(ans[i-1])+SQR(ans[i])) == 0.0)
                nrerror("Deconvolving at response zero in convlv");
            ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/mag2/no2;Divide FFTs
            ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/mag2/no2;             to deconvolve.
        } else nrerror("No meaning for isign in convlv");
    }
    ans[2]=ans[n+1];                        Pack last element with first for realft.
    realft(ans,n,-1);                       Inverse transform back to time domain.
    free_vector(fft,1,n<<1);
}
```

Convolving or Deconvolving Very Large Data Sets

If your data set is so long that you do not want to fit it into memory all at once, then you must break it up into sections and convolve each section separately. Now, however, the treatment of end effects is a bit different. You have to worry not only about spurious wrap-around effects, but also about the fact that the ends of each section of data *should* have been influenced by data at the nearby ends of the immediately preceding and following sections of data, but were not so influenced since only one section of data is in the machine at a time.

There are two, related, standard solutions to this problem. Both are fairly obvious, so with a few words of description here, you ought to be able to implement them for yourself. The first solution is called the *overlap-save method*. In this technique you pad only the very beginning of the data with enough zeros to avoid wrap-around pollution. After this initial padding, you forget about zero padding altogether. Bring in a section of data and convolve or deconvolve it. Then throw out the points at each end that are polluted by wrap-around end effects. Output only the

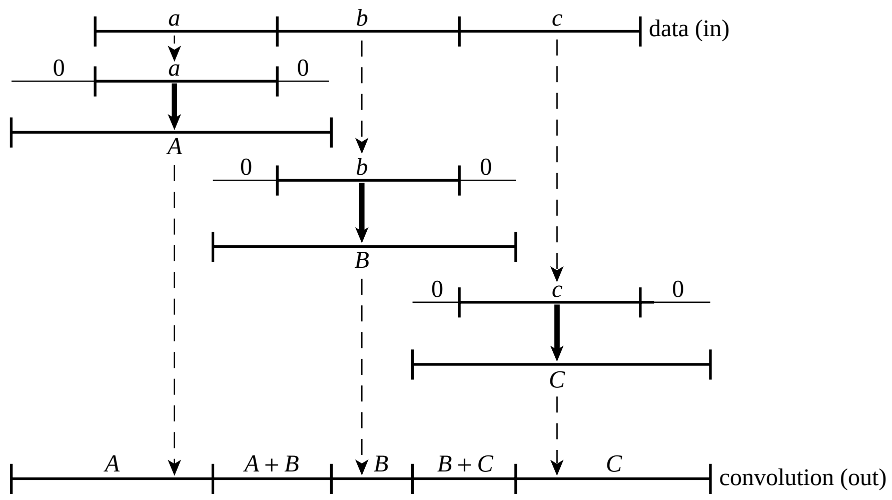


Figure 13.1.5. The overlap-add method for convolving a response with a very long signal. The signal data is broken up into smaller pieces. Each is zero padded at both ends and convolved (denoted by bold arrows in the figure). Finally the pieces are added back together, including the overlapping regions formed by the zero pads.

remaining good points in the middle. Now bring in the next section of data, but not all new data. The first points in each new section overlap the last points from the preceding section of data. The sections must be overlapped sufficiently so that the polluted output points at the end of one section are recomputed as the first of the unpolluted output points from the subsequent section. With a bit of thought you can easily determine how many points to overlap and save.

The second solution, called the *overlap-add method*, is illustrated in Figure 13.1.5. Here you *don't* overlap the input data. Each section of data is disjoint from the others and is used exactly once. However, you carefully zero-pad it at both ends so that there is no wrap-around ambiguity in the output convolution or deconvolution. Now you overlap *and add* these sections of output. Thus, an output point near the end of one section will have the response due to the input points at the beginning of the next section of data properly added in to it, and likewise for an output point near the beginning of a section, *mutatis mutandis*.

Even when computer memory is available, there is some slight gain in computing speed in segmenting a long data set, since the FFTs' $N \log_2 N$ is slightly slower than linear in N. However, the log term is so slowly varying that you will often be much happier to avoid the bookkeeping complexities of the overlap-add or overlap-save methods: If it is practical to do so, just cram the whole data set into memory and FFT away. Then you will have more time for the finer things in life, some of which are described in succeeding sections of this chapter.

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

13.2 Correlation and Autocorrelation Using the FFT

Correlation is the close mathematical cousin of convolution. It is in some ways simpler, however, because the two functions that go into a correlation are not as conceptually distinct as were the data and response functions that entered into convolution. Rather, in correlation, the functions are represented by different, but generally similar, data sets. We investigate their "correlation," by comparing them both directly superposed, and with one of them shifted left or right.

We have already defined in equation (12.0.10) the correlation between two continuous functions $g(t)$ and $h(t)$, which is denoted $\text{Corr}(g, h)$, and is a function of *lag* t. We will occasionally show this time dependence explicitly, with the rather awkward notation $\text{Corr}(g, h)(t)$. The correlation will be large at some value of t if the first function (g) is a close copy of the second (h) but lags it in time by t, i.e., if the first function is shifted to the right of the second. Likewise, the correlation will be large for some negative value of t if the first function *leads* the second, i.e., is shifted to the left of the second. The relation that holds when the two functions are interchanged is

$$\text{Corr}(g, h)(t) = \text{Corr}(h, g)(-t) \tag{13.2.1}$$

The discrete correlation of two sampled functions g_k and h_k, each periodic with period N, is defined by

$$\text{Corr}(g, h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{13.2.2}$$

The *discrete correlation theorem* says that this discrete correlation of two real functions g and h is one member of the discrete Fourier transform pair

$$\text{Corr}(g, h)_j \Longleftrightarrow G_k H_k{}^* \tag{13.2.3}$$

where G_k and H_k are the discrete Fourier transforms of g_j and h_j, and the asterisk denotes complex conjugation. This theorem makes the same presumptions about the functions as those encountered for the discrete convolution theorem.

We can compute correlations using the FFT as follows: FFT the two data sets, multiply one resulting transform by the complex conjugate of the other, and inverse transform the product. The result (call it r_k) will formally be a complex vector of length N. However, it will turn out to have all its imaginary parts zero since the original data sets were both real. The components of r_k are the values of the correlation at different lags, with positive and negative lags stored in the by now familiar wrap-around order: The correlation at zero lag is in r_0, the first component;

the correlation at lag 1 is in r_1, the second component; the correlation at lag -1 is in r_{N-1}, the last component; etc.

Just as in the case of convolution we have to consider end effects, since our data will not, in general, be periodic as intended by the correlation theorem. Here again, we can use zero padding. If you are interested in the correlation for lags as large as $\pm K$, then you must append a buffer zone of K zeros at the end of both input data sets. If you want all possible lags from N data points (not a usual thing), then you will need to pad the data with an equal number of zeros; this is the extreme case. So here is the program:

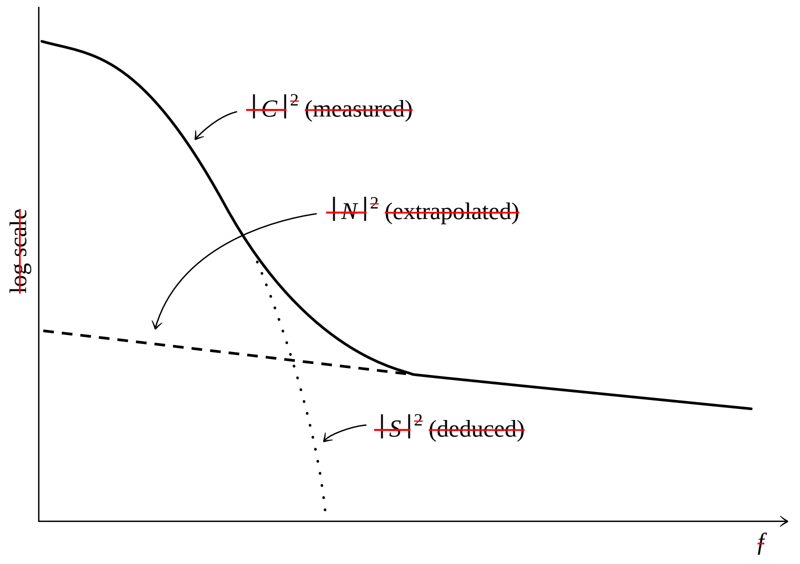
```
#include "nrutil.h"

void correl(float data1[], float data2[], unsigned long n, float ans[])
```
Computes the correlation of two real data sets `data1[1..n]` and `data2[1..n]` (including any user-supplied zero padding). `n` MUST be an integer power of two. The answer is returned as the first `n` points in `ans[1..2*n]` stored in wrap-around order, i.e., correlations at increasingly negative lags are in `ans[n]` on down to `ans[n/2+1]`, while correlations at increasingly positive lags are in `ans[1]` (zero lag) on up to `ans[n/2]`. Note that `ans` must be supplied in the calling program with length at least `2*n`, since it is also used as working space. Sign convention of this routine: if `data1` lags `data2`, i.e., is shifted to the right of it, then `ans` will show a peak at positive lags.
```
{
    void realft(float data[], unsigned long n, int isign);
    void twofft(float data1[], float data2[], float fft1[], float fft2[],
        unsigned long n);
    unsigned long no2,i;
    float dum,*fft;

    fft=vector(1,n<<1);
    twofft(data1,data2,fft,ans,n);          Transform both data vectors at once.
    no2=n>>1;                               Normalization for inverse FFT.
    for (i=2;i<=n+2;i+=2) {
        ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/no2;      Multiply to find
        ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/no2;                   FFT of their cor-
    }                                                              relation.
    ans[2]=ans[n+1];                        Pack first and last into one element.
    realft(ans,n,-1);                       Inverse transform gives correlation.
    free_vector(fft,1,n<<1);
}
```

As in `convlv`, it would be better to substitute two calls to `realft` for the one call to `twofft`, if `data1` and `data2` have very different magnitudes, to minimize roundoff error.

The *discrete autocorrelation* of a sampled function g_j is just the discrete correlation of the function with itself. Obviously this is always symmetric with respect to positive and negative lags. Feel free to use the above routine `correl` to obtain autocorrelations, simply calling it with the same `data` vector in both arguments. If the inefficiency bothers you, routine `realft` can, of course, be used to transform the `data` vector instead.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §13–2.



~~Figure 13.3.1. Optimal (Wiener) filtering. The power spectrum of signal plus noise shows a signal peak added to a noise tail. The tail is extrapolated back into the signal region as a "noise model." Subtracting gives the "signal model." The models need not be accurate for the method to be useful. A simple algebraic combination of the models gives the optimal filter (see text).~~

~~Don't waste your time on this line of thought. The scheme converges to a signal of $S(f) = 0$. Converging iterative methods do exist; this just isn't one of them.~~

~~You can use the routine `four1` (§12.2) or `realft` (§12.3) to FFT your data when you are constructing an optimal filter. To apply the filter to your data, you can use the methods described in §13.1. The specific routine `convlv` is not needed for optimal filtering, since your filter is constructed in the frequency domain to begin with. If you are also deconvolving your data with a known response function, however, you can modify `convlv` to multiply by your optimal filter just before it takes the inverse Fourier transform.~~

~~CITED REFERENCES AND FURTHER READING:~~

~~Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).~~

~~Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).~~

~~Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press~~).

13.4 Power Spectrum Estimation Using the FFT

In the previous section we "informally" estimated the power spectral density of a function $c(t)$ by taking the modulus-squared of the discrete Fourier transform of some finite, sampled stretch of it. In this section we'll do roughly the same thing, but with considerably greater attention to details. Our attention will uncover some surprises.

The first detail is power spectrum (also called a power spectral density or PSD) normalization. In general there is *some* relation of proportionality between a measure of the squared amplitude of the function and a measure of the amplitude of the PSD. Unfortunately there are several different conventions for describing the normalization in each domain, and many opportunities for getting wrong the relationship between the two domains. Suppose that our function $c(t)$ is sampled at N points to produce values $c_0 \dots c_{N-1}$, and that these points span a range of time T, that is $T = (N-1)\Delta$, where Δ is the sampling interval. Then here are several different descriptions of the total power:

$$\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“sum squared amplitude”} \tag{13.4.1}$$

$$\frac{1}{T}\int_0^T |c(t)|^2\ dt \approx \frac{1}{N}\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“mean squared amplitude”} \tag{13.4.2}$$

$$\int_0^T |c(t)|^2\ dt \approx \Delta \sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“time-integral squared amplitude”} \tag{13.4.3}$$

PSD estimators, as we shall see, have an even greater variety. In this section, we consider a class of them that give estimates at discrete values of frequency f_i, where i will range over integer values. In the next section, we will learn about a different class of estimators that produce estimates that are continuous functions of frequency f. Even if it is agreed always to relate the PSD normalization to a particular description of the function normalization (e.g., 13.4.2), there are at least the following possibilities: The PSD is

- defined for discrete positive, zero, and negative frequencies, and its sum over these is the function mean squared amplitude
- defined for zero and discrete positive frequencies only, and its sum over these is the function mean squared amplitude
- defined in the Nyquist interval from $-f_c$ to f_c, and its integral over this range is the function mean squared amplitude
- defined from 0 to f_c, and its integral over this range is the function mean squared amplitude

It *never* makes sense to integrate the PSD of a sampled function outside of the Nyquist interval $-f_c$ and f_c since, according to the sampling theorem, power there will have been aliased into the Nyquist interval.

It is hopeless to define enough notation to distinguish all possible combinations of normalizations. In what follows, we use the notation $P(f)$ to mean *any* of the above PSDs, stating in each instance how the particular $P(f)$ is normalized. Beware the inconsistent notation in the literature.

The method of power spectrum estimation used in the previous section is a simple version of an estimator called, historically, the *periodogram*. If we take an N-point sample of the function $c(t)$ at equal intervals and use the FFT to compute its discrete Fourier transform

$$C_k = \sum_{j=0}^{N-1} c_j \, e^{2\pi ijk/N} \qquad k = 0, \dots, N-1 \tag{13.4.4}$$

then the periodogram estimate of the power spectrum is defined at $N/2 + 1$ frequencies as

$$\begin{aligned} P(0) = P(f_0) &= \frac{1}{N^2} |C_0|^2 \\ P(f_k) &= \frac{1}{N^2} \left[|C_k|^2 + |C_{N-k}|^2 \right] \qquad k = 1, 2, \dots, \left(\frac{N}{2} - 1 \right) \\ P(f_c) = P(f_{N/2}) &= \frac{1}{N^2} \left| C_{N/2} \right|^2 \end{aligned} \tag{13.4.5}$$

where f_k is defined only for the zero and positive frequencies

$$f_k \equiv \frac{k}{N\Delta} = 2f_c \frac{k}{N} \qquad k = 0, 1, \dots, \frac{N}{2} \tag{13.4.6}$$

By Parseval's theorem, equation (12.1.10), we see immediately that equation (13.4.5) is normalized so that the sum of the $N/2 + 1$ values of P is equal to the mean squared amplitude of the function c_j.

We must now ask this question. In what sense is the periodogram estimate (13.4.5) a "true" estimator of the power spectrum of the underlying function $c(t)$? You can find the answer treated in considerable detail in the literature cited (see, e.g., [1] for an introduction). Here is a summary.

First, is the *expectation value* of the periodogram estimate equal to the power spectrum, i.e., is the estimator correct on average? Well, yes and no. We wouldn't really expect one of the $P(f_k)$'s to equal the continuous $P(f)$ at *exactly* f_k, since f_k is supposed to be representative of a whole frequency "bin" extending from halfway from the preceding discrete frequency to halfway to the next one. We *should* be expecting the $P(f_k)$ to be some kind of average of $P(f)$ over a narrow window function centered on its f_k. For the periodogram estimate (13.4.6) that window function, as a function of s the frequency offset *in bins*, is

$$W(s) = \frac{1}{N^2} \left[\frac{\sin(\pi s)}{\sin(\pi s/N)} \right]^2 \tag{13.4.7}$$

Notice that $W(s)$ has oscillatory lobes but, apart from these, falls off only about as $W(s) \approx (\pi s)^{-2}$. This is not a very rapid fall-off, and it results in significant *leakage* (that is the technical term) from one frequency to another in the periodogram estimate. Notice also that $W(s)$ happens to be zero for s equal to a nonzero integer. This means that if the function $c(t)$ is a pure sine wave of frequency exactly equal to one of the f_k's, then there will be *no* leakage to adjacent f_k's. But this is not the characteristic case! If the frequency is, say, one-third of the way between two adjacent f_k's, then the leakage will extend *well* beyond those two adjacent bins. The solution to the problem of leakage is called *data windowing*, and we will discuss it below.

Turn now to another question about the periodogram estimate. What is the variance of that estimate as N goes to infinity? In other words, as we take more sampled points from the original function (either sampling a longer stretch of data at the same sampling rate, or else by resampling the same stretch of data with a faster sampling rate), then how much more accurate do the estimates P_k become? The unpleasant answer is that the periodogram estimates *do not become more accurate at all!* In fact, the variance of the periodogram estimate at a frequency f_k is always equal to the square of its expectation value at that frequency. In other words, the standard deviation is always 100 percent of the value, independent of N! How can this be? Where did all the information go as we added points? It all went into producing estimates at a greater number of discrete frequencies f_k. If we sample a longer run of data using the same sampling rate, then the Nyquist critical frequency f_c is unchanged, but we now have finer frequency resolution (more f_k's) within the Nyquist frequency interval; alternatively, if we sample the same length of data with a finer sampling interval, then our frequency resolution is unchanged, but the Nyquist range now extends up to a higher frequency. In neither case do the additional samples reduce the variance of any one particular frequency's estimated PSD.

You don't have to live with PSD estimates with 100 percent standard deviations, however. You simply have to know some techniques for reducing the variance of the estimates. Here are two techniques that are very nearly identical mathematically, though different in implementation. The first is to compute a periodogram estimate with finer discrete frequency spacing than you really need, and then to sum the periodogram estimates at K consecutive discrete frequencies to get one "smoother" estimate at the mid frequency of those K. The variance of that summed estimate will be smaller than the estimate itself by a factor of exactly $1/K$, i.e., the standard deviation will be smaller than 100 percent by a factor $1/\sqrt{K}$. Thus, to estimate the power spectrum at $M+1$ discrete frequencies between 0 and f_c inclusive, you begin by taking the FFT of $2MK$ points (which number had better be an integer power of two!). You then take the modulus square of the resulting coefficients, add positive and negative frequency pairs, and divide by $(2MK)^2$, all according to equation (13.4.5) with $N = 2MK$. Finally, you "bin" the results into summed (not averaged) groups of K. This procedure is very easy to program, so we will not bother to give a routine for it. The reason that you sum, rather than average, K consecutive points is so that your final PSD estimate will preserve the normalization property that the sum of its $M+1$ values equals the mean square value of the function.

A second technique for estimating the PSD at $M+1$ discrete frequencies in the range 0 to f_c is to partition the original sampled data into K segments each of $2M$ consecutive sampled points. Each segment is separately FFT'd to produce a periodogram estimate (equation 13.4.5 with $N \equiv 2M$). Finally, the K periodogram estimates are averaged at each frequency. It is this final averaging that reduces the variance of the estimate by a factor K (standard deviation by $\sqrt{K}$). This second technique is computationally more efficient than the first technique above by a modest factor, since it is logarithmically more efficient to take many shorter FFTs than one longer one. The principal advantage of the second technique, however, is that only $2M$ data points are manipulated at a single time, not $2KM$ as in the first technique. This means that the second technique is the natural choice for processing long runs of data, as from a magnetic tape or other data record. We will give a routine later for implementing this second technique, but we need first to return to the matters of leakage and data windowing which were brought up after equation (13.4.7) above.

Data Windowing

The purpose of data windowing is to modify equation (13.4.7), which expresses the relation between the spectral estimate P_k at a discrete frequency and the actual underlying continuous spectrum $P(f)$ at nearby frequencies. In general, the spectral power in one "bin" k contains leakage from frequency components that are actually s bins away, where s is the independent variable in equation (13.4.7). There is, as we pointed out, quite substantial leakage even from moderately large values of s.

When we select a run of N sampled points for periodogram spectral estimation, we are in effect multiplying an infinite run of sampled data c_j by a window function in time, one that is zero except during the total sampling time $N\Delta$, and is unity during that time. In other words, the data are windowed by a square window function. By the convolution theorem (12.0.9; but interchanging the roles of f and t), the Fourier transform of the product of the data with this square window function is equal to the convolution of the data's Fourier transform with the window's Fourier transform. In fact, we determined equation (13.4.7) as nothing more than the square of the discrete Fourier transform of the unity window function.

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 = \frac{1}{N^2}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N}\right|^2 \qquad (13.4.8)$$

The reason for the leakage at large values of s, is that the square window function turns on and off so rapidly. Its Fourier transform has substantial components at high frequencies. To remedy this situation, we can multiply the input data $c_j,\ j = 0,\dots,N-1$ by a window function w_j that changes more gradually from zero to a maximum and then back to zero as j ranges from 0 to N. In this case, the equations for the periodogram estimator (13.4.4–13.4.5) become

$$D_k \equiv \sum_{j=0}^{N-1} c_j w_j\, e^{2\pi i j k/N} \qquad k = 0,\dots,N-1 \qquad (13.4.9)$$

$$\begin{aligned}
P(0) = P(f_0) &= \frac{1}{W_{ss}}\left|D_0\right|^2 \\
P(f_k) &= \frac{1}{W_{ss}}\left[\left|D_k\right|^2 + \left|D_{N-k}\right|^2\right] \qquad k = 1,2,\dots,\left(\frac{N}{2}-1\right) \\
P(f_c) = P(f_{N/2}) &= \frac{1}{W_{ss}}\left|D_{N/2}\right|^2
\end{aligned} \qquad (13.4.10)$$

where W_{ss} stands for "window squared and summed,"

$$W_{ss} \equiv N \sum_{j=0}^{N-1} w_j^2 \qquad (13.4.11)$$

and f_k is given by (13.4.6). The more general form of (13.4.7) can now be written in terms of the window function w_j as

$$\begin{aligned} W(s) &= \frac{1}{W_{ss}} \left| \sum_{k=0}^{N-1} e^{2\pi isk/N} w_k \right|^2 \\ &\approx \frac{1}{W_{ss}} \left| \int_{-N/2}^{N/2} \cos(2\pi sk/N) w(k - N/2)\, dk \right|^2 \end{aligned} \tag{13.4.12}$$

Here the approximate equality is useful for practical estimates, and holds for any window that is left-right symmetric (the usual case), and for $s \ll N$ (the case of interest for estimating leakage into nearby bins). The continuous function $w(k-N/2)$ in the integral is meant to be some smooth function that passes through the points w_k.

There is a lot of perhaps unnecessary lore about choice of a window function, and practically every function that rises from zero to a peak and then falls again has been named after someone. A few of the more common (also shown in Figure 13.4.1) are:

$$w_j = 1 - \left| \frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right| \equiv \text{“Bartlett window”} \tag{13.4.13}$$

(The “Parzen window” is very similar to this.)

$$w_j = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi j}{N}\right)\right] \equiv \text{“Hann window”} \tag{13.4.14}$$

(The “Hamming window” is similar but does not go exactly to zero at the ends.)

$$w_j = 1 - \left(\frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right)^2 \equiv \text{“Welch window”} \tag{13.4.15}$$

We are inclined to follow Welch in recommending that you use either (13.4.13) or (13.4.15) in practical work. However, at the level of this book, there is effectively *no difference* between any of these (or similar) window functions. Their difference lies in subtle trade-offs among the various figures of merit that can be used to describe the narrowness or peakedness of the spectral leakage functions computed by (13.4.12). These figures of merit have such names as: *highest sidelobe level (dB), sidelobe fall-off (dB per octave), equivalent noise bandwidth (bins), 3-dB bandwidth (bins), scallop loss (dB), worst case process loss (dB).* Roughly speaking, the principal trade-off is between making the central peak as narrow as possible versus making the tails of the distribution fall off as rapidly as possible. For details, see (e.g.) [2]. Figure 13.4.2 plots the leakage amplitudes for several windows already discussed.

There is particularly a lore about window functions that rise smoothly from zero to unity in the first small fraction (say 10 percent) of the data, then stay at unity until the last small fraction (again say 10 percent) of the data, during which the window function falls smoothly back to zero. These windows will squeeze a little bit of extra narrowness out of the main lobe of the leakage function (never as

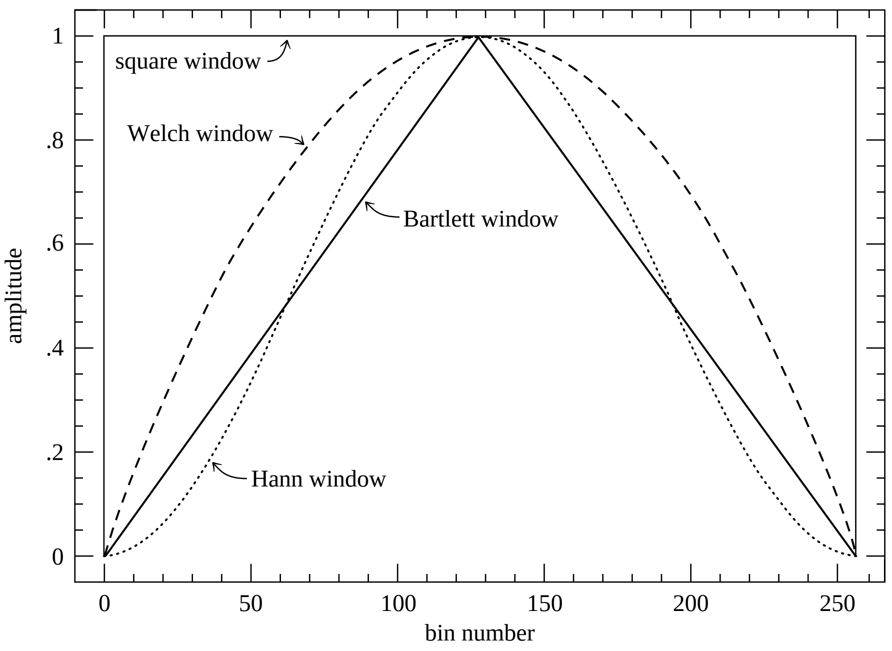


Figure 13.4.1. Window functions commonly used in FFT power spectral estimation. The data segment, here of length 256, is multiplied (bin by bin) by the window function before the FFT is computed. The square window, which is equivalent to no windowing, is least recommended. The Welch and Bartlett windows are good choices.

much as a factor of two, however), but trade this off by widening the leakage tail by a significant factor (e.g., the reciprocal of 10 percent, a factor of ten). If we distinguish between the *width* of a window (number of samples for which it is at its maximum value) and its *rise/fall time* (number of samples during which it rises and falls); and if we distinguish between the *FWHM* (full width to half maximum value) of the leakage function's main lobe and the *leakage width* (full width that contains half of the spectral power that is not contained in the main lobe); then these quantities are related roughly by

$$(\text{FWHM in bins}) \approx \frac{N}{(\text{window width})} \tag{13.4.16}$$

$$(\text{leakage width in bins}) \approx \frac{N}{(\text{window rise/fall time})} \tag{13.4.17}$$

For the windows given above in (13.4.13)–(13.4.15), the effective window widths and the effective window rise/fall times are both of order $\frac{1}{2}N$. Generally speaking, we feel that the advantages of windows whose rise and fall times are only small fractions of the data length are minor or nonexistent, and we avoid using them. One sometimes hears it said that flat-topped windows "throw away less of the data," but we will now show you a better way of dealing with that problem by use of overlapping data segments.

Let us now suppose that we have chosen a window function, and that we are ready to segment the data into K segments of $N = 2M$ points. Each segment will be FFT'd, and the resulting K periodograms will be averaged together to obtain a

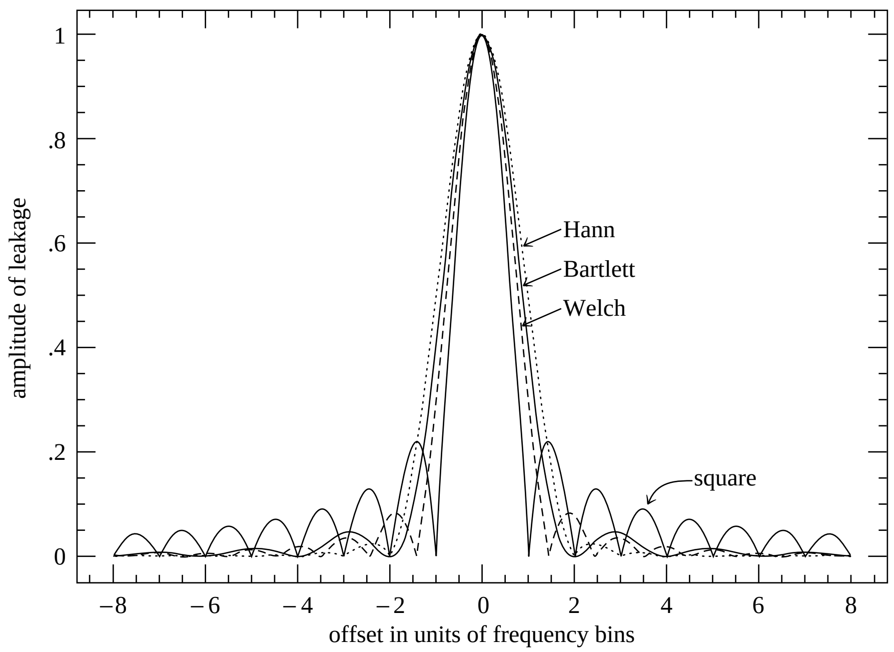


Figure 13.4.2. Leakage functions for the window functions of Figure 13.4.1. A signal whose frequency is actually located at zero offset "leaks" into neighboring bins with the amplitude shown. The purpose of windowing is to reduce the leakage at large offsets, where square (no) windowing has large sidelobes. Offset can have a fractional value, since the actual signal frequency can be located between two frequency bins of the FFT.

PSD estimate at $M + 1$ frequency values from 0 to f_c. We must now distinguish between two possible situations. We might want to obtain the smallest variance from a fixed amount of computation, without regard to the number of data points used. This will generally be the goal when the data are being gathered in real time, with the data-reduction being computer-limited. Alternatively, we might want to obtain the smallest variance from a fixed number of available sampled data points. This will generally be the goal in cases where the data are already recorded and we are analyzing it after the fact.

In the first situation (smallest spectral variance per computer operation), it is best to segment the data without any overlapping. The first $2M$ data points constitute segment number 1; the next $2M$ data points constitute segment number 2; and so on, up to segment number K, for a total of $2KM$ sampled points. The variance in this case, relative to a single segment, is reduced by a factor K.

In the second situation (smallest spectral variance per data point), it turns out to be optimal, or very nearly optimal, to overlap the segments by one half of their length. The first and second sets of M points are segment number 1; the second and third sets of M points are segment number 2; and so on, up to segment number K, which is made of the Kth and $K + 1$st sets of M points. The total number of sampled points is therefore $(K + 1)M$, just over half as many as with nonoverlapping segments. The reduction in the variance is not a full factor of K, since the segments are not statistically independent. It can be shown that the variance is instead reduced by a factor of about $9K/11$ (see the paper by Welch in [3]). This is, however, significantly better than the reduction of about $K/2$ that would have resulted if the same *number* of data points were segmented without overlapping.

We can now codify these ideas into a routine for spectral estimation. While we generally avoid input/output coding, we make an exception here to show how data are read sequentially in one pass through a data file (referenced through the parameter `FILE *fp`). Only a small fraction of the data is in memory at any one time. Note that `spctrm` returns the power at M, not $M + 1$, frequencies, omitting the component $P(f_c)$ at the Nyquist frequency. It would also be straightforward to include that component.

```
#include <math.h>
#include <stdio.h>
#include "nrutil.h"
#define WINDOW(j,a,b) (1.0-fabs((((j)-1)-(a))*(b))) /* Bartlett */
/* #define WINDOW(j,a,b) 1.0 */ /* Square */
/* #define WINDOW(j,a,b) (1.0-SQR((((j)-1)-(a))*(b))) */ /* Welch */

void spctrm(FILE *fp, float p[], int m, int k, int ovrlap)
Reads data from input stream specified by file pointer fp and returns as p[j] the data's power
(mean square amplitude) at frequency (j-1)/(2*m) cycles per gridpoint, for j=1,2,...,m,
based on (2*k+1)*m data points (if ovrlap is set true (1)) or 4*k*m data points (if ovrlap
is set false (0)). The number of segments of the data is 2*k in both cases: The routine calls
four1 k times, each call with 2 partitions each of 2*m real data points.
{
    void four1(float data[], unsigned long nn, int isign);
    int mm,m44,m43,m4,kk,joffn,joff,j2,j;
    float w,facp,facm,*w1,*w2,sumw=0.0,den=0.0;

    mm=m+m;                                     Useful factors.
    m43=(m4=mm+mm)+3;
    m44=m43+1;
    w1=vector(1,m4);
    w2=vector(1,m);
    facm=m;
    facp=1.0/m;
    for (j=1;j<=mm;j++) sumw += SQR(WINDOW(j,facm,facp));
    Accumulate the squared sum of the weights.
    for (j=1;j<=m;j++) p[j]=0.0;                Initialize the spectrum to zero.
    if (ovrlap)                                 Initialize the "save" half-buffer.
        for (j=1;j<=m;j++) fscanf(fp,"%f",&w2[j]);
    for (kk=1;kk<=k;kk++) {
    Loop over data set segments in groups of two.
        for (joff = -1;joff<=0;joff++) {        Get two complete segments into workspace.
            if (ovrlap) {
                for (j=1;j<=m;j++) w1[joff+j+j]=w2[j];
                for (j=1;j<=m;j++) fscanf(fp,"%f",&w2[j]);
                joffn=joff+mm;
                for (j=1;j<=m;j++) w1[joffn+j+j]=w2[j];
            } else {
                for (j=joff+2;j<=m4;j+=2)
                    fscanf(fp,"%f",&w1[j]);
            }
        }
        for (j=1;j<=mm;j++) {                   Apply the window to the data.
            j2=j+j;
            w=WINDOW(j,facm,facp);
            w1[j2] *= w;
            w1[j2-1] *= w;
        }
        four1(w1,mm,1);                         Fourier transform the windowed data.
        p[1] += (SQR(w1[1])+SQR(w1[2]));        Sum results into previous segments.
        for (j=2;j<=m;j++) {
            j2=j+j;
            p[j] += (SQR(w1[j2])+SQR(w1[j2-1])
                +SQR(w1[m44-j2])+SQR(w1[m43-j2]));
        }
        den += sumw;
    }
    den *= m4;                                  Correct normalization.
    for (j=1;j<=m;j++) p[j] /= den;             Normalize the output.
    free_vector(w2,1,m);
    free_vector(w1,1,m4);
}
```

CITED REFERENCES AND FURTHER READING:

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Harris, F.J. 1978, *Proceedings of the IEEE*, vol. 66, pp. 51–83. [2]

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), paper by P.D. Welch. [3]

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

~~13.5 Digital Filtering in the Time Domain~~

~~Suppose that you have a signal that you want to filter digitally. For example, perhaps you want to apply *high-pass* or *low-pass* filtering, to eliminate noise at low or high frequencies respectively; or perhaps the interesting part of your signal lies only in a certain frequency band, so that you need a *bandpass* filter. Or, if your measurements are contaminated by 60 Hz power-line interference, you may need a *notch filter* to remove only a narrow band around that frequency. This section speaks particularly about the case in which you have chosen to do such filtering in the time domain.~~

~~Before continuing, we hope you will reconsider this choice. Remember how convenient it is to filter in the Fourier domain. You just take your whole data record, FFT it, multiply the FFT output by a filter function $\mathcal{H}(f)$, and then do an inverse FFT to get back a filtered data set in time domain. Here is some additional background on the Fourier technique that you will want to take into account.~~

- ~~Remember that you must define your filter function $\mathcal{H}(f)$ for both positive and negative frequencies, and that the magnitude of the frequency extremes is always the Nyquist frequency $1/(2\Delta)$, where Δ is the sampling interval. The magnitude of the smallest nonzero frequencies in the FFT is $\pm 1/(N\Delta)$, where N is the number of (complex) points in the FFT. The positive and negative frequencies to which this filter are applied are arranged in wrap-around order.~~
- ~~If the measured data are real, and you want the filtered output also to be real, then your arbitrary filter function should obey $\mathcal{H}(-f) = \mathcal{H}(f)^*$. You can arrange this most easily by picking an $\mathcal{H}$ that is real and even in f.~~

Chapter 14. Statistical Description of Data

14.0 Introduction

In this chapter and the next, the concept of *data* enters the discussion more prominently than before.

Data consist of numbers, of course. But these numbers are fed into the computer, not produced by it. These are numbers to be treated with considerable respect, neither to be tampered with, nor subjected to a numerical process whose character you do not completely understand. You are well advised to acquire a reverence for data that is rather different from the "sporty" attitude that is sometimes allowable, or even commendable, in other numerical tasks.

The analysis of data inevitably involves some trafficking with the field of *statistics*, that gray area which is not quite a branch of mathematics — and just as surely not quite a branch of science. In the following sections, you will repeatedly encounter the following paradigm:

- apply some formula to the data to compute "a statistic"
- compute where the value of that statistic falls in a probability distribution that is computed on the basis of some "null hypothesis"
- if it falls in a very unlikely spot, way out on a tail of the distribution, conclude that the null hypothesis is *false* for your data set

If a statistic falls in a *reasonable* part of the distribution, you must not make the mistake of concluding that the null hypothesis is "verified" or "proved." That is the curse of statistics, that it can never prove things, only disprove them! At best, you can substantiate a hypothesis by ruling out, statistically, a whole long list of competing hypotheses, every one that has ever been proposed. After a while your adversaries and competitors will give up trying to think of alternative hypotheses, or else they will grow old and die, and *then your hypothesis will become accepted*. Sounds crazy, we know, but that's how science works!

In this book we make a somewhat arbitrary distinction between data analysis procedures that are *model-independent* and those that are *model-dependent*. In the former category, we include so-called *descriptive statistics* that characterize a data set in general terms: its mean, variance, and so on. We also include statistical tests that seek to establish the "sameness" or "differentness" of two or more data sets, or that seek to establish and measure a degree of *correlation* between two data sets. These subjects are discussed in this chapter.

In the other category, model-dependent statistics, we lump the whole subject of fitting data to a theory, parameter estimation, least-squares fits, and so on. Those subjects are introduced in Chapter 15.

Section 14.1 deals with so-called *measures of central tendency*, the moments of a distribution, the median and mode. In §14.2 we learn to test whether different data sets are drawn from distributions with different values of these measures of central tendency. This leads naturally, in §14.3, to the more general question of whether two distributions can be shown to be (significantly) different.

In §14.4–§14.7, we deal with *measures of association* for two distributions. We want to determine whether two variables are "correlated" or "dependent" on one another. If they are, we want to characterize the degree of correlation in some simple ways. The distinction between parametric and nonparametric (rank) methods is emphasized.

Section 14.8 introduces the concept of data smoothing, and discusses the particular case of Savitzky-Golay smoothing filters.

This chapter draws mathematically on the material on special functions that was presented in Chapter 6, especially §6.1–§6.4. You may wish, at this point, to review those sections.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill).

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*].

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth

When a set of values has a sufficiently strong central tendency, that is, a tendency to cluster around some particular value, then it may be useful to characterize the set by a few numbers that are related to its *moments*, the sums of integer powers of the values.

Best known is the *mean* of the values $x_1, \ldots, x_N$,

$$\overline{x} = \frac{1}{N} \sum_{j=1}^{N} x_j \tag{14.1.1}$$

which estimates the value around which central clustering occurs. Note the use of an overbar to denote the mean; angle brackets are an equally common notation, e.g., $\langle x \rangle$. You should be aware that the mean is not the only available estimator of this quantity, nor is it necessarily the best one. For values drawn from a probability distribution with very broad "tails," the mean may converge poorly, or not at all, as the number of sampled points is increased. Alternative estimators, the *median* and the *mode*, are mentioned at the end of this section.

Having characterized a distribution's central value, one conventionally next characterizes its "width" or "variability" around that value. Here again, more than one measure is available. Most common is the *variance*,

$$\mathrm{Var}(x_1 \ldots x_N) = \frac{1}{N-1}\sum_{j=1}^{N}(x_j - \overline{x})^2 \tag{14.1.2}$$

or its square root, the *standard deviation*,

$$\sigma(x_1 \ldots x_N) = \sqrt{\mathrm{Var}(x_1 \ldots x_N)} \tag{14.1.3}$$

Equation (14.1.2) estimates the mean squared deviation of x from its mean value. There is a long story about why the denominator of (14.1.2) is $N-1$ instead of N. If you have never heard that story, you may consult any good statistics text. Here we will be content to note that the $N-1$ *should* be changed to N if you are ever in the situation of measuring the variance of a distribution whose mean $\overline{x}$ is known *a priori* rather than being estimated from the data. (We might also comment that if the difference between N and $N-1$ ever matters to you, then you are probably up to no good anyway — e.g., trying to substantiate a questionable hypothesis with marginal data.)

As the mean depends on the first moment of the data, so do the variance and standard deviation depend on the second moment. It is not uncommon, in real life, to be dealing with a distribution whose second moment does not exist (i.e., is infinite). In this case, the variance or standard deviation is useless as a measure of the data's width around its central value: The values obtained from equations (14.1.2) or (14.1.3) will not converge with increased numbers of points, nor show any consistency from data set to data set drawn from the same distribution. This can occur even when the width of the peak looks, by eye, perfectly finite. A more robust estimator of the width is the *average deviation* or *mean absolute deviation*, defined by

$$\mathrm{ADev}(x_1 \ldots x_N) = \frac{1}{N}\sum_{j=1}^{N}|x_j - \overline{x}| \tag{14.1.4}$$

One often substitutes the sample median x_{med} for $\overline{x}$ in equation (14.1.4). For any fixed sample, the median in fact minimizes the mean absolute deviation.

Statisticians have historically sniffed at the use of (14.1.4) instead of (14.1.2), since the absolute value brackets in (14.1.4) are "nonanalytic" and make theorem-proving difficult. In recent years, however, the fashion has changed, and the subject of *robust estimation* (meaning, estimation for broad distributions with significant numbers of "outlier" points) has become a popular and important one. Higher moments, or statistics involving higher powers of the input data, are almost always less robust than lower moments or statistics that involve only linear sums or (the lowest moment of all) counting.

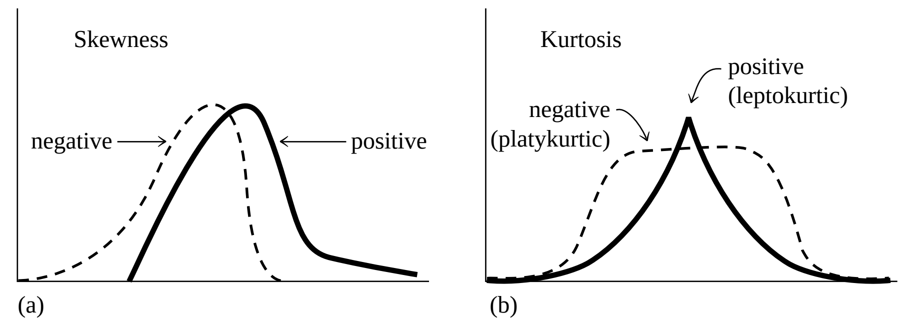


Figure 14.1.1. Distributions whose third and fourth moments are significantly different from a normal (Gaussian) distribution. (a) Skewness or third moment. (b) Kurtosis or fourth moment.

That being the case, the *skewness* or *third moment*, and the *kurtosis* or *fourth moment* should be used with caution or, better yet, not at all.

The skewness characterizes the degree of asymmetry of a distribution around its mean. While the mean, standard deviation, and average deviation are *dimensional* quantities, that is, have the same units as the measured quantities x_j, the skewness is conventionally defined in such a way as to make it *nondimensional*. It is a pure number that characterizes only the shape of the distribution. The usual definition is

$$\text{Skew}(x_1 \dots x_N) = \frac{1}{N}\sum_{j=1}^{N}\left[\frac{x_j - \overline{x}}{\sigma}\right]^3 \qquad (14.1.5)$$

where $\sigma = \sigma(x_1 \dots x_N)$ is the distribution's standard deviation (14.1.3). A positive value of skewness signifies a distribution with an asymmetric tail extending out towards more positive x; a negative value signifies a distribution whose tail extends out towards more negative x (see Figure 14.1.1).

Of course, any set of N measured values is likely to give a nonzero value for (14.1.5), even if the underlying distribution is in fact symmetrical (has zero skewness). For (14.1.5) to be meaningful, we need to have some idea of *its* standard deviation as an estimator of the skewness of the underlying distribution. Unfortunately, that depends on the shape of the underlying distribution, and rather critically on its tails! For the idealized case of a normal (Gaussian) distribution, the standard deviation of (14.1.5) is approximately $\sqrt{15/N}$ when $\overline{x}$ is the true mean, and $\sqrt{6/N}$ when it is estimated by the sample mean, (14.1.1). In real life it is good practice to believe in skewnesses only when they are several or many times as large as this.

The kurtosis is also a nondimensional quantity. It measures the relative peakedness or flatness of a distribution. Relative to what? A normal distribution, what else! A distribution with positive kurtosis is termed *leptokurtic*; the outline of the Matterhorn is an example. A distribution with negative kurtosis is termed *platykurtic*; the outline of a loaf of bread is an example. (See Figure 14.1.1.) And, as you no doubt expect, an in-between distribution is termed *mesokurtic*.

The conventional definition of the kurtosis is

$$\text{Kurt}(x_1 \dots x_N) = \left\{\frac{1}{N}\sum_{j=1}^{N}\left[\frac{x_j - \overline{x}}{\sigma}\right]^4\right\} - 3 \qquad (14.1.6)$$

where the -3 term makes the value zero for a normal distribution.

The standard deviation of (14.1.6) as an estimator of the kurtosis of an underlying normal distribution is $\sqrt{96/N}$ when σ is the true standard deviation, and $\sqrt{24/N}$ when it is the sample estimate (14.1.3). However, the kurtosis depends on such a high moment that there are many real-life distributions for which the standard deviation of (14.1.6) as an estimator is effectively infinite.

Calculation of the quantities defined in this section is perfectly straightforward. Many textbooks use the binomial theorem to expand out the definitions into sums of various powers of the data, e.g., the familiar

$$\text{Var}(x_1 \dots x_N) = \frac{1}{N-1}\left[\left(\sum_{j=1}^{N} x_j^2\right) - N\overline{x}^2\right] \approx \overline{x^2} - \overline{x}^2 \tag{14.1.7}$$

but this can magnify the roundoff error by a large factor and is generally unjustifiable in terms of computing speed. A clever way to minimize roundoff error, especially for large samples, is to use the *corrected two-pass algorithm* [1]: First calculate $\overline{x}$, then calculate $\text{Var}(x_1 \dots x_N)$ by

$$\text{Var}(x_1 \dots x_N) = \frac{1}{N-1}\left\{\sum_{j=1}^{N}(x_j - \overline{x})^2 - \frac{1}{N}\left[\sum_{j=1}^{N}(x_j - \overline{x})\right]^2\right\} \tag{14.1.8}$$

The second sum would be zero if $\overline{x}$ were exact, but otherwise it does a good job of correcting the roundoff error in the first term.

```
#include <math.h>

void moment(float data[], int n, float *ave, float *adev, float *sdev,
    float *var, float *skew, float *curt)
```
Given an array of `data[1..n]`, this routine returns its mean `ave`, average deviation `adev`, standard deviation `sdev`, variance `var`, skewness `skew`, and kurtosis `curt`.
```
{
    void nrerror(char error_text[]);
    int j;
    float ep=0.0,s,p;

    if (n <= 1) nrerror("n must be at least 2 in moment");
    s=0.0;                                  First pass to get the mean.
    for (j=1;j<=n;j++) s += data[j];
    *ave=s/n;
    *adev=(*var)=(*skew)=(*curt)=0.0;       Second pass to get the first (absolute), sec-
    for (j=1;j<=n;j++) {                        ond, third, and fourth moments of the
        *adev += fabs(s=data[j]-(*ave));        deviation from the mean.
        ep += s;
        *var += (p=s*s);
        *skew += (p *= s);
        *curt += (p *= s);
    }
    *adev /= n;
    *var=(*var-ep*ep/n)/(n-1);              Corrected two-pass formula.
    *sdev=sqrt(*var);                       Put the pieces together according to the con-
    if (*var) {                                 ventional definitions.
        *skew /= (n*(*var)*(*sdev));
        *curt=(*curt)/(n*(*var)*(*var))-3.0;
    } else nrerror("No skew/kurtosis when variance = 0 (in moment)");
}
```

Semi-Invariants

The mean and variance of independent random variables are additive: If x and y are drawn independently from two, possibly different, probability distributions, then

$$\overline{(x+y)} = \overline{x} + \overline{y} \qquad \text{Var}(x+y) = \text{Var}(x) + \text{Var}(x) \tag{14.1.9}$$

Higher moments are not, in general, additive. However, certain combinations of them, called *semi-invariants*, are in fact additive. If the centered moments of a distribution are denoted M_k,

$$M_k \equiv \left\langle (x_i - \overline{x})^k \right\rangle \tag{14.1.10}$$

so that, e.g., $M_2 = \text{Var}(x)$, then the first few semi-invariants, denoted I_k are given by

$$\begin{aligned} &I_2 = M_2 \qquad I_3 = M_3 \qquad I_4 = M_4 - 3M_2^2 \\ &I_5 = M_5 - 10M_2M_3 \qquad I_6 = M_6 - 15M_2M_4 - 10M_3^2 + 30M_2^3 \end{aligned} \tag{14.1.11}$$

Notice that the skewness and kurtosis, equations (14.1.5) and (14.1.6) are simple powers of the semi-invariants,

$$\text{Skew}(x) = I_3/I_2^{3/2} \qquad \text{Kurt}(x) = I_4/I_2^2 \tag{14.1.12}$$

A Gaussian distribution has all its semi-invariants higher than I_2 equal to zero. A Poisson distribution has all of its semi-invariants equal to its mean. For more details, see [2].

Median and Mode

The median of a probability distribution function $p(x)$ is the value x_{med} for which larger and smaller values of x are equally probable:

$$\int_{-\infty}^{x_{\text{med}}} p(x)\,dx = \frac{1}{2} = \int_{x_{\text{med}}}^{\infty} p(x)\,dx \tag{14.1.13}$$

The median of a distribution is estimated from a sample of values $x_1, \ldots, x_N$ by finding that value x_i which has equal numbers of values above it and below it. Of course, this is not possible when N is even. In that case it is conventional to estimate the median as the mean of the unique *two* central values. If the values x_j $j = 1, \ldots, N$ are sorted into ascending (or, for that matter, descending) order, then the formula for the median is

$$x_{\text{med}} = \begin{cases} x_{(N+1)/2}, & N \text{ odd} \\ \frac{1}{2}(x_{N/2} + x_{(N/2)+1}), & N \text{ even} \end{cases} \tag{14.1.14}$$

If a distribution has a strong central tendency, so that most of its area is under a single peak, then the median is an estimator of the central value. It is a more robust estimator than the mean is: The median fails as an estimator only if the area in the tails is large, while the mean fails if the first moment of the tails is large; it is easy to construct examples where the first moment of the tails is large even though their area is negligible.

To find the median of a set of values, one can proceed by sorting the set and then applying (14.1.14). This is a process of order $N \log N$. You might rightly think that this is wasteful, since it yields much more information than just the median (e.g., the upper and lower quartile points, the deciles, etc.). In fact, we saw in §8.5 that the element $x_{(N+1)/2}$ can be located in of order N operations. Consult that section for routines.

The *mode* of a probability distribution function $p(x)$ is the value of x where it takes on a maximum value. The mode is useful primarily when there is a single, sharp maximum, in which case it estimates the central value. Occasionally, a distribution will be *bimodal*, with two relative maxima; then one may wish to know the two modes individually. Note that, in such cases, the mean and median are not very useful, since they will give only a "compromise" value between the two peaks.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 2.

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*], vol. 1, §10.15

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Chan, T.F., Golub, G.H., and LeVeque, R.J. 1983, *American Statistician*, vol. 37, pp. 242–247. [1]

Cramér, H. 1946, *Mathematical Methods of Statistics* (Princeton: Princeton University Press), §15.10. [2]

~~14.2 Do Two Distributions Have the Same Means or Variances?~~

~~Not uncommonly we want to know whether two distributions have the same mean. For example, a first set of measured values may have been gathered before some event, a second set after it. We want to know whether the event, a "treatment" or a "change in a control parameter," made a difference.~~

~~Our first thought is to ask "how many standard deviations" one sample mean is from the other. That number may in fact be a useful thing to know. It does relate to the strength or "importance" of a difference of means *if that difference is genuine*. However, by itself, it says nothing about whether the difference *is* genuine, that is, statistically significant. A difference of means can be very small compared to the standard deviation, and yet very significant, if the number of data points is large. Conversely, a difference may be moderately large but not significant, if the data are sparse. We will be meeting these distinct concepts of *strength* and *significance* several times in the next few sections.~~

~~A quantity that measures the significance of a difference of means is not the number of standard deviations that they are apart, but the number of so-called *standard errors* that they are apart. The standard error of a set of values measures the accuracy with which the sample mean estimates the population (or "true") mean. Typically the standard error is equal to the sample's standard deviation divided by the square root of the number of points in the sample.~~