integer values, say between $+1000000$ and -1000000 (supposing that you need six significant figures). Now modify equation (13.6.11) by enclosing the sum term in an "integer part of" operator. The discrepancy will now, by definition, be an integer. Experiment with different values of M to find LP coefficients that make the range of the discrepancy as small as you can. If you can get to within a range of ± 127 (and in our experience this is not at all difficult), then you can write it to a file as a single byte. This is a compression factor of 4, compared to 4-byte integer or floating formats.

Notice that the LP coefficients are computed using the *quantized* data, and that the discrepancy is also quantized, i.e., quantization is done both outside and inside the LPC loop. If you are careful in following this prescription, then, apart from the initial quantization of the data, you will not introduce even a single bit of roundoff error into the compression-reconstruction process: While the evaluation of the sum in (13.6.11) may have roundoff errors, the residual that you store is the value that, when added back to the sum, gives *exactly* the original (quantized) data value. Notice also that you do not need to massage the LP coefficients for stability; by adding the residual back in to each point, you never depart from the original data, so instabilities cannot grow. There is therefore no need for `fixrts`, above.

Look at §22.5 to learn about *Huffman coding*, which will further compress the residuals by taking advantage of the fact that smaller values of discrepancy will occur more often than larger values. A very primitive version of Huffman coding would be this: If most of the discrepancies are in the range ± 127, but an occasional one is outside, then reserve the value 127 to mean "out of range," and then record on the file (immediately following the 127) a full-word value of the out-of-range discrepancy. Section 22.5 explains how to do much better.

There are many variant procedures that all fall under the rubric of LPC:

- If the spectral character of the data is time-variable, then it is best not to use a single set of LP coefficients for the whole data set, but rather to partition the data into segments, computing and storing different LP coefficients for each segment.
- If the data are really well characterized by their LP coefficients, and you can tolerate some small amount of error, then don't bother storing all of the residuals. Just do linear prediction until you are outside of tolerances, and then reinitialize (using M sequential stored residuals) and continue predicting.
- In some applications, most notably speech synthesis, one cares only about the spectral content of the reconstructed signal, not the relative phases. In this case, one need not store any starting values at all, but only the LP coefficients for each segment of the data. The output is reconstructed by driving these coefficients with initial conditions consisting of all zeros except for one nonzero spike. A speech synthesizer chip may have of order 10 LP coefficients, which change perhaps 20 to 50 times per second.
- Some people believe that it is interesting to analyze a signal by LPC, even when the residuals x_i are *not* small. The x_i's are then interpreted as the underlying "input signal" that, when filtered through the all-poles filter defined by the LP coefficients (see §13.7), produces the observed "output signal." LPC reveals simultaneously, it is said, the nature of the filter *and* the particular input that is driving it. We are skeptical of these applications; the literature, however, is full of extravagant claims.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), especially the paper by J. Makhoul, "Linear Prediction: A Tutorial Review," reprinted from *Proceedings of the IEEE*, vol. 63, p. 561, 1975.

Burg, J.P. 1968, "A New Analysis Technique for Time Series Data," reprinted in Childers, 1978.[1]

Anderson, N. 1974, "On the Calculation of Filter Coefficients for Maximum Entropy Spectral Analysis," *Geophysics*, vol. 39, pp. 69–72, reprinted in Childers, 1978.[2]

Cressie, N. 1991, "Geostatistical Analysis of Spatial Data," in *Spatial Statistics and Digital Image Analysis* (Washington: National Academy Press).[3]

Press, W.H., and Rybicki, G.B. 1992, "Interpolation, Realization, and Reconstruction of Noisy, Irregularly Sampled Data," *Astrophysical Journal*, vol. 398, pp. 169–176.[4]

13.7 Power Spectrum Estimation by the Maximum Entropy (All-Poles) Method

The FFT is not the only way to estimate the power spectrum of a process, nor is it necessarily the best way for all purposes. To see how one might devise another method, let us enlarge our view for a moment, so that it includes not only real frequencies in the Nyquist interval $-f_c < f < f_c$, but also the entire complex frequency plane. From that vantage point, let us transform the complex f-plane to a new plane, called the *z-transform plane* or *z-plane*, by the relation

$$z \equiv e^{2\pi i f \Delta} \qquad (13.7.1)$$

where Δ is, as usual, the sampling interval in the time domain. Notice that the Nyquist interval on the real axis of the f-plane maps one-to-one onto the unit circle in the complex z-plane.

If we now compare (13.7.1) to equations (13.4.4) and (13.4.6), we see that the FFT power spectrum estimate (13.4.5) for any real sampled function $c_k \equiv c(t_k)$ can be written, except for normalization convention, as

$$P(f) = \left| \sum_{k=-N/2}^{N/2-1} c_k z^k \right|^2 \qquad (13.7.2)$$

Of course, (13.7.2) is not the *true* power spectrum of the underlying function $c(t)$, but only an estimate. We can see in two related ways why the estimate is not likely to be exact. First, in the time domain, the estimate is based on only a finite range of the function $c(t)$, which may, for all we know, have continued from $t = -\infty$ to ∞. Second, in the z-plane of equation (13.7.2), the finite Laurent series offers, in general, only an approximation to a general analytic function of z. In fact, a formal expression for representing "true" power spectra (up to normalization) is

$$P(f) = \left| \sum_{k=-\infty}^{\infty} c_k z^k \right|^2 \qquad (13.7.3)$$

This is an infinite Laurent series that depends on an infinite number of values c_k. Equation (13.7.2) is just one kind of analytic approximation to the analytic function of z represented by (13.7.3), the kind, in fact, that is implicit in the use of FFTs to estimate power spectra by periodogram methods. It goes under several names, including *direct method, all-zero model,* and *moving average (MA) model*. The term "all-zero" in particular refers to the fact that the model spectrum can have zeros in the z-plane, but not poles.

If we look at the problem of approximating (13.7.3) more generally, it seems clear that we could do a better job with a rational function, one with a series of type (13.7.2) in both the numerator and the denominator. Less obviously, it turns out that there are some advantages in an approximation whose free parameters all lie in the *denominator*, namely,

$$P(f) \approx \frac{1}{\left| \sum_{k=-M/2}^{M/2} b_k z^k \right|^2} = \frac{a_0}{\left| 1 + \sum_{k=1}^{M} a_k z^k \right|^2} \tag{13.7.4}$$

Here the second equality brings in a new set of coefficients a_k's, which can be determined from the b_k's using the fact that z lies on the unit circle. The b_k's can be thought of as being determined by the condition that power series expansion of (13.7.4) agree with the first $M + 1$ terms of (13.7.3). In practice, as we shall see, one determines the b_k's or a_k's by another method.

The differences between the approximations (13.7.2) and (13.7.4) are not just cosmetic. They are approximations of very different character. Most notable is the fact that (13.7.4) can have *poles*, corresponding to infinite power spectral density, on the unit z-circle, i.e., at real frequencies in the Nyquist interval. Such poles can provide an accurate representation for underlying power spectra that have sharp, discrete "lines" or delta-functions. By contrast, (13.7.2) can have only zeros, not poles, at real frequencies in the Nyquist interval, and must thus attempt to fit sharp spectral features with, essentially, a polynomial. The approximation (13.7.4) goes under several names: *all-poles model, maximum entropy method (MEM), autoregressive model (AR)*. We need only find out how to compute the coefficients a_0 and the a_k's from a data set, so that we can actually use (13.7.4) to obtain spectral estimates.

A pleasant surprise is that we already know how! Look at equation (13.6.11) for linear prediction. Compare it with linear filter equations (13.5.1) and (13.5.2), and you will see that, viewed as a filter that takes input x's into output y's, linear prediction has a filter function

$$\mathcal{H}(f) = \frac{1}{1 - \sum_{j=0}^{N-1} d_j z^{-(j+1)}} \tag{13.7.5}$$

Thus, the power spectrum of the y's should be equal to the power spectrum of the x's multiplied by $|\mathcal{H}(f)|^2$. Now let us think about what the spectrum of the input x's is, when they are residual discrepancies from linear prediction. Although we will not prove it formally, it is intuitively believable that the x's are independently random and therefore have a flat (white noise) spectrum. (Roughly speaking, any residual correlations left in the x's would have allowed a more accurate linear prediction, and would have been removed.) The overall normalization of this flat spectrum is just the mean square amplitude of the x's. But this is exactly the quantity computed in equation (13.6.13) and returned by the routine `memcof` as `xms`. Thus, the coefficients a_0 and a_k in equation (13.7.4) are related to the LP coefficients returned by `memcof` simply by

$$a_0 = \texttt{xms} \qquad a_k = -\texttt{d}(k-1), \quad k = 1, \dots, M \tag{13.7.6}$$

There is also another way to describe the relation between the a_k's and the autocorrelation components ϕ_k. The Wiener-Khinchin theorem (12.0.13) says that the Fourier transform of the autocorrelation is equal to the power spectrum. In z-transform language, this Fourier transform is just a Laurent series in z. The equation that is to be satisfied by the coefficients in equation (13.7.4) is thus

$$\frac{a_0}{\left| 1 + \sum_{k=1}^{M} a_k z^k \right|^2} \approx \sum_{j=-M}^{M} \phi_j z^j \tag{13.7.7}$$

The approximately equal sign in (13.7.7) has a somewhat special interpretation. It means that the series expansion of the left-hand side is supposed to agree with the right-hand side term-by-term from z^{-M} to z^M. Outside this range of terms, the right-hand side is obviously zero, while the left-hand side will still have nonzero terms. Notice that M, the number of coefficients in the approximation on the left-hand side, can be any integer up to N, the total number of autocorrelations available. (In practice, one often chooses M much smaller than N.) M is called the *order* or *number of poles* of the approximation.

Whatever the chosen value of M, the series expansion of the left-hand side of (13.7.7) defines a certain sort of *extrapolation* of the autocorrelation function to lags larger than M, in fact even to lags larger than N, i.e., *larger than the run of data can actually measure*. It turns out that this particular extrapolation can be shown to have, among all possible extrapolations, the maximum *entropy* in a definable information-theoretic sense. Hence the name *maximum entropy method*, or MEM. The maximum entropy property has caused MEM to acquire a certain "cult" popularity; one sometimes hears that it gives an intrinsically "better" estimate than is given by other methods. Don't believe it. MEM has the very cute property of being able to fit sharp spectral features, but there is nothing else magical about its power spectrum estimates.

The operations count in `memcof` scales as the product of N (the number of data points) and M (the desired order of the MEM approximation). If M were chosen to be as large as N, then the method would be much slower than the $N \log N$ FFT methods of the previous section. In practice, however, one usually wants to limit the order (or number of poles) of the MEM approximation to a few times the number of sharp spectral features that one desires it to fit. With this restricted number of poles, the method will smooth the spectrum somewhat, but this is often a desirable property. While exact values depend on the application, one might take $M = 10$ or 20 or 50 for $N = 1000$ or 10000. In that case, MEM estimation is not much slower than FFT estimation.

We feel obliged to warn you that `memcof` can be a bit quirky at times. If the number of poles or number of data points is too large, roundoff error can be a problem, even in double precision. With "peaky" data (i.e., data with extremely sharp spectral features), the algorithm may suggest split peaks even at modest orders, and the peaks may shift with the phase of the sine wave. Also, with noisy input functions, if you choose too high an order, you will find spurious peaks galore! Some experts recommend the use of this algorithm in conjunction with more conservative methods, like periodograms, to help choose the correct model order and to avoid getting too fooled by spurious spectral features. MEM can be finicky, but it can also do remarkable things. We recommend that you try it out, cautiously, on your own problems. We now turn to the evaluation of the MEM spectral estimate from its coefficients.

The MEM estimation (13.7.4) is a function of continuously varying frequency f. There is no special significance to specific equally spaced frequencies as there was in the FFT case. In fact, since the MEM estimate may have very sharp spectral features, one wants to be able to evaluate it on a very fine mesh near to those features, but perhaps only more coarsely farther away from them. Here is a function that, given the coefficients already computed, evaluates (13.7.4) and returns the estimated power spectrum as a function of $f\Delta$ (the frequency times the sampling interval). Of course, $f\Delta$ should lie in the Nyquist range between $-1/2$ and $1/2$.

```
Doub evlmem(const Doub fdt, VecDoub_I &d, const Doub xms)                linpredict.h
```
Given `d[0..m-1]` and `xms` as returned by `memcof`, this function returns the power spectrum estimate $P(f)$ as a function of `fdt` $= f\Delta$.
```
{
    Int i;
    Doub sumr=1.0,sumi=0.0,wr=1.0,wi=0.0,wpr,wpi,wtemp,theta;

    Int m=d.size();
    theta=6.28318530717959*fdt;
    wpr=cos(theta);                     Set up for recurrence relations.
    wpi=sin(theta);
    for (i=0;i<m;i++) {                 Loop over the terms in the sum.
        wr=(wtemp=wr)*wpr-wi*wpi;
        wi=wi*wpr+wtemp*wpi;
        sumr -= d[i]*wr;                These accumulate the denominator of (13.7.4).
        sumi -= d[i]*wi;
    }
    return xms/(sumr*sumr+sumi*sumi);
}
```

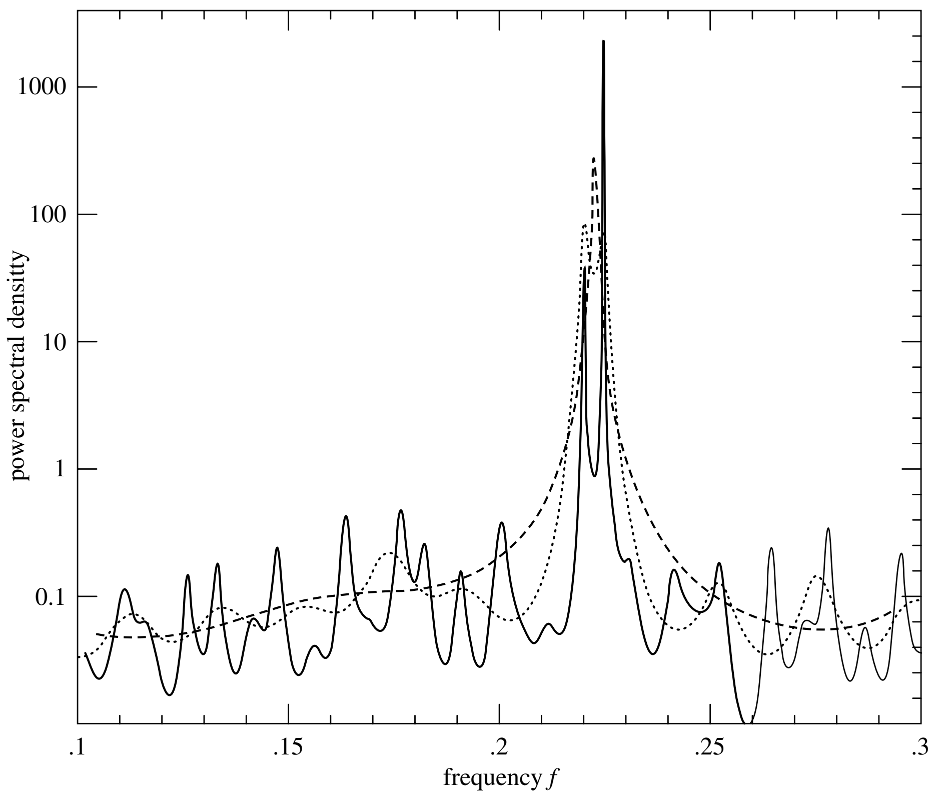


Figure 13.7.1. Sample output of maximum entropy spectral estimation. The input signal consists of 512 samples of the sum of two sinusoids of very nearly the same frequency, plus white noise with about equal power. Shown is an expanded portion of the full Nyquist frequency interval (which would extend from zero to 0.5). The dashed spectral estimate uses 20 poles; the dotted, 40; the solid, 150. With the larger number of poles, the method can resolve the distinct sinusoids, but the flat noise background is beginning to show spurious peaks. (Note logarithmic scale.)

Be sure to evaluate $P(f)$ on a fine enough grid to *find* any narrow features that may be there! Such narrow features, if present, can contain virtually all of the power in the data. You might also wish to know how the $P(f)$ produced by the routines memcof and evlmem is normalized with respect to the mean square value of the input data vector. The answer is

$$\int_{-1/2}^{1/2} P(f\Delta)d(f\Delta) = 2\int_{0}^{1/2} P(f\Delta)d(f\Delta) = \text{mean square value of data} \qquad (13.7.8)$$

Sample spectra produced by the routines memcof and evlmem are shown in Figure 13.7.1.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), Chapter II.

Kay, S.M., and Marple, S.L. 1981, “Spectrum Analysis: A Modern Perspective,” *Proceedings of the IEEE*, vol. 69, pp. 1380–1419.

13.8 Spectral Analysis of Unevenly Sampled Data

Thus far, we have been dealing exclusively with evenly sampled data,

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{13.8.1}$$

where Δ is the sampling interval, whose reciprocal is the sampling rate. Recall also (§12.1) the significance of the Nyquist critical frequency

$$f_c \equiv \frac{1}{2\Delta} \tag{13.8.2}$$

as codified by the sampling theorem: A sampled data set like equation (13.8.1) contains *complete* information about all spectral components for a signal $h(t)$ containing only frequencies below the Nyquist frequency, and scrambled or *aliased* information about any signals containing frequencies larger than the Nyquist frequency. The sampling theorem thus defines both the attractiveness and the limitation of any analysis of an evenly spaced data set.

There are situations, however, where evenly spaced data cannot be obtained. A common case is where instrumental dropouts occur, so that data are obtained only on a (not consecutive integer) subset of equation (13.8.1), the so-called *missing data* problem. Another case, common in observational sciences like astronomy, is that the observer cannot completely control the time of the observations, but must simply accept a certain dictated set of t_i's.

There are some obvious ways to get from unevenly spaced t_i's to evenly spaced ones, as in equation (13.8.1). Interpolation is one way: Lay down a grid of evenly spaced times on your data and interpolate values onto that grid; then use FFT methods. In the missing data problem, you only have to interpolate on missing data points. If a lot of consecutive points are missing, you might as well just set them to zero, or perhaps "clamp" the value at the last measured point. However, the experience of practitioners of such interpolation techniques *is not reassuring*. Generally speaking, such techniques perform poorly. Long gaps in the data, for example, often produce a spurious bulge of power at low frequencies (wavelengths comparable to gaps).

A completely different method of spectral analysis for unevenly sampled data, one that mitigates these difficulties and has some other very desirable properties, was developed by Lomb [1], based in part on earlier work by Barning [2] and Vaníček [3], and additionally elaborated by Scargle [4]. The Lomb method (as we will call it) evaluates data, and sines and cosines, only at times t_i that are actually measured. Suppose that there are N data points $h_i \equiv h(t_i)$, $i = 0, \ldots, N-1$. Then first find the mean and variance of the data by the usual formulas,

$$\bar{h} \equiv \frac{1}{N}\sum_{i=0}^{N-1} h_i \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_{i=0}^{N-1}(h_i - \bar{h})^2 \tag{13.8.3}$$

Now, the Lomb *normalized periodogram* (spectral power as a function of angular frequency $\omega \equiv 2\pi f > 0$) is defined by

$$P_N(\omega) \equiv \frac{1}{2\sigma^2}\left\{ \frac{\left[\sum_j (h_j - \bar{h})\cos\omega(t_j - \tau)\right]^2}{\sum_j \cos^2\omega(t_j - \tau)} + \frac{\left[\sum_j (h_j - \bar{h})\sin\omega(t_j - \tau)\right]^2}{\sum_j \sin^2\omega(t_j - \tau)} \right\} \tag{13.8.4}$$

Here τ is defined by the relation

$$\tan(2\omega\tau) = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j} \tag{13.8.5}$$

The constant τ is a kind of offset that makes $P_N(\omega)$ completely independent of shifting all the t_i's by any constant. Lomb shows that this particular choice of offset has another, deeper, effect: It makes equation (13.8.4) identical to the equation that one would obtain if one estimated the harmonic content of a data set, at a given frequency ω, by linear least-squares fitting to the model

$$h(t) = A\cos\omega t + B\sin\omega t \tag{13.8.6}$$

This fact gives some insight into why the method can give results superior to FFT methods: It weights the data on a "per-point" basis instead of on a "per-time interval" basis, when uneven sampling can render the latter seriously in error.

A very common occurrence is that the measured data points h_i are the sum of a periodic signal and independent (white) Gaussian noise. If we are trying to determine the presence or absence of such a periodic signal, we want to be able to give a quantitative answer to the question, "How significant is a peak in the spectrum $P_N(\omega)$?" In this question, the null hypothesis is that the data values are independent Gaussian random values. A very nice property of the Lomb normalized periodogram is that the viability of the null hypothesis can be tested fairly rigorously, as we now discuss.

The word "normalized" refers to the factor σ^2 in the denominator of equation (13.8.4). Scargle [4] shows that with this normalization, at any particular ω and *in the case of the null hypothesis*, $P_N(\omega)$ has an exponential probability distribution with unit mean. In other words, the probability that $P_N(\omega)$ will be between some positive z and $z + dz$ is $\exp(-z)dz$. It readily follows that, if we scan some M *independent* frequencies, the probability that none give values larger than z is $(1 - e^{-z})^M$. So

$$P(> z) \equiv 1 - (1 - e^{-z})^M \tag{13.8.7}$$

is the false-alarm probability of the null hypothesis, that is, the *significance level* of any peak in $P_N(\omega)$ that we do see. A small value for the false-alarm probability indicates a highly significant periodic signal.

To evaluate this significance, we need to know M. After all, the more frequencies we look at, the less significant is some one modest bump in the spectrum. (Look long enough, find anything!) A typical procedure will be to plot $P_N(\omega)$ as a function of many closely spaced frequencies in some large frequency range. How many of these are independent?

Before answering, let us first see how accurately we need to know M. The interesting region is where the significance is a small (significant) number, $\ll 1$. There, equation (13.8.7) can be series expanded to give

$$P(> z) \approx Me^{-z} \tag{13.8.8}$$

We see that the significance scales linearly with M. Practical significance levels are numbers like 0.05, 0.01, 0.001, etc. An error of even $\pm 50\%$ in the estimated significance is often tolerable, since quoted significance levels are typically spaced apart by factors of 5 or 10. So our estimate of M need not be very accurate.

Horne and Baliunas [5] give results from extensive Monte Carlo experiments for determining M in various cases. In general, M depends on the number of frequencies sampled, the number of data points N, and their detailed spacing. It turns out that M is very nearly equal to N when the data points are approximately equally spaced and when the sampled frequencies "fill" (oversample) the frequency range from 0 to the Nyquist frequency f_c (equation 13.8.2). Further, the value of M is not importantly different for random spacing of the data points than for equal spacing. When a larger frequency range than the Nyquist range is sampled, M increases proportionally. About the only case where M differs significantly from the case of evenly spaced points is when the points are closely clumped, say into groups of three; then (as one would expect) the number of independent frequencies is reduced by a factor of about 3.

The program `period`, below, calculates an effective value for M based on the above rough-and-ready rules and assumes that there is no important clumping. This will be adequate for most purposes. In any particular case, if it really matters, it is not too difficult to compute a better value of M by simple Monte Carlo: Holding fixed the number of data points and their locations t_i, generate synthetic data sets of Gaussian (normal) deviates, find the largest values of $P_N(\omega)$ for each such data set (using the accompanying program), and fit the resulting distribution for M in equation (13.8.7).

Figure 13.8.1 shows the results of applying the method as discussed so far. In the upper figure, the data points are plotted against time. Their number is $N = 100$, and their distribution in t is Poisson random. There is certainly no sinusoidal signal evident to the eye. The

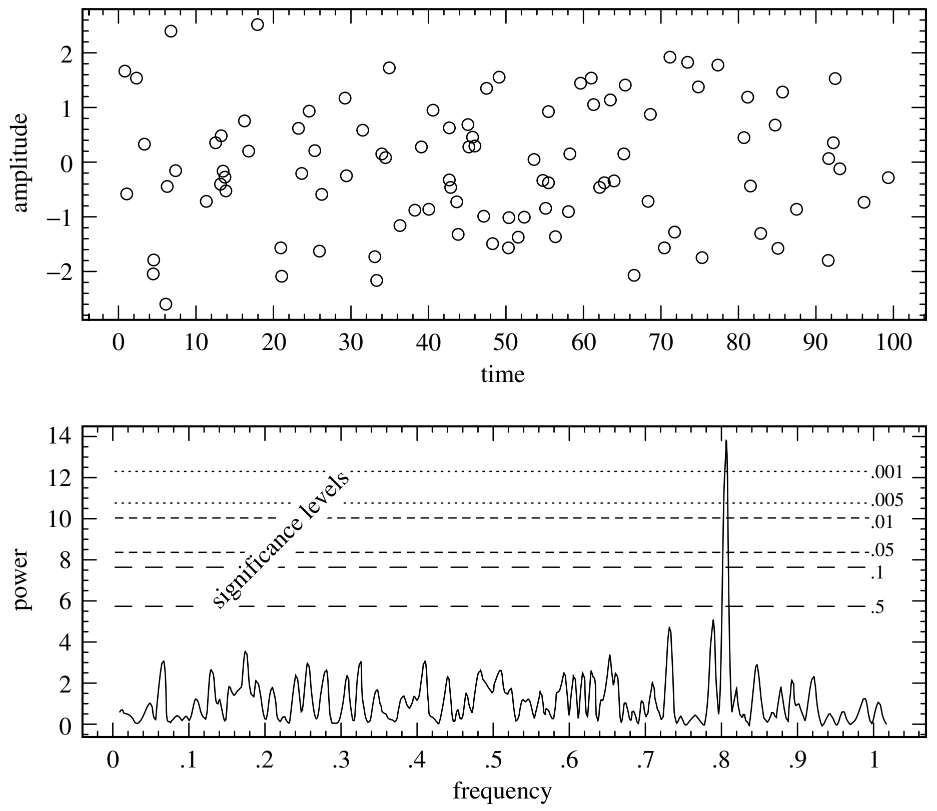


Figure 13.8.1. Example of the Lomb algorithm in action. The 100 data points (upper figure) are at random times between 0 and 100. Their sinusoidal component is readily uncovered (lower figure) by the algorithm, at a significance level $p < 0.001$. If the 100 data points had been evenly spaced at unit interval, the Nyquist critical frequency would have been 0.5. Note that, for these unevenly spaced points, there is no visible aliasing into the Nyquist range.

lower figure plots $P_N(\omega)$ against frequency $f = \omega/2\pi$. The Nyquist critical frequency that would obtain if the points were evenly spaced is at $f = f_c = 0.5$. Since we have searched up to about twice that frequency, and oversampled the f's to the point where successive values of $P_N(\omega)$ vary smoothly, we take $M = 2N$. The horizontal dashed and dotted lines are (respectively from bottom to top) significance levels 0.5, 0.1, 0.05, 0.01, 0.005, and 0.001. One sees a highly significant peak at a frequency of 0.81. That is in fact the frequency of the sine wave that is present in the data. (You will have to take our word for this!)

Note that two other peaks approach but do not exceed the 50% significance level; that is about what one might expect by chance. It is also worth commenting on the fact that the significant peak was found (correctly) *above the Nyquist frequency* and without any significant aliasing down into the Nyquist interval! That would not be possible for evenly spaced data. It is possible here because the randomly spaced data have *some* points spaced much closer than the "average" sampling rate, and these remove ambiguity from any aliasing.

Implementation of the normalized periodogram in code is straightforward, with, however, a few points to be kept in mind. We are dealing with a *slow* algorithm. Typically, for N data points, we may wish to examine on the order of $2N$ or $4N$ frequencies. Each combination of frequency and data point has, in equations (13.8.4) and (13.8.5), not just a few adds or multiplies, but four calls to trigonometric functions; the operations count can easily reach several hundred times N^2. It is highly desirable — in fact results in a factor 4 speedup — to replace these trigonometric calls by recurrences. That is possible only if the sequence of frequencies examined is a linear sequence. Since such a sequence is probably what most users would want anyway, we have built this into the implementation.

At the end of this section we describe a way to evaluate equations (13.8.4) and (13.8.5) — approximately, but to any desired degree of approximation — by a fast method [6] whose operation count goes only as $N \log N$. This faster method should be used for long data sets.

The lowest independent frequency f to be examined is the inverse of the span of the input data, $\max_i(t_i) - \min_i(t_i) \equiv T$. This is the frequency such that the data can include one complete cycle. In subtracting off the data's mean, equation (13.8.4) already assumed that you are not interested in the data's zero frequency piece — which is just that mean value. In an FFT method, higher independent frequencies would be integer multiples of $1/T$. Because we are interested in the statistical significance of any peak that may occur, however, we had better (over-)sample more finely than at interval $1/T$, so that sample points lie close to the top of any peak. Thus, the accompanying program includes an oversampling parameter, called `ofac`; a value `ofac` $\gtrsim 4$ might be typical in use. We also want to specify how high in frequency to go, say f_{hi}. One guide to choosing f_{hi} is to compare it with the Nyquist frequency f_c that would obtain if the N data points were evenly spaced over the same span T, that is, $f_c = N/(2T)$. The accompanying program includes an input parameter `hifac`, defined as f_{hi}/f_c. The number of different frequencies N_P returned by the program is then given by

$$N_P = \frac{\texttt{ofac} \times \texttt{hifac}}{2} N \qquad (13.8.9)$$

(You have to remember to dimension the output arrays to at least this size.)

The trigonometric recurrences should be done in double precision even if you convert the rest of the routine to single precision. The code embodies a few tricks with trigonometric identities, to decrease roundoff errors. If you are an aficionado of such things, you can puzzle it out. A final detail is that equation (13.8.7) will fail because of roundoff error if z is too large; but equation (13.8.8) is fine in this regime.

period.h

```
void period(VecDoub_I &x, VecDoub_I &y, const Doub ofac, const Doub hifac,
    VecDoub_O &px, VecDoub_O &py, Int &nout, Int &jmax, Doub &prob) {
```
Given `n` data points with abscissas `x[0..n-1]` (which need not be equally spaced) and ordinates `y[0..n-1]`, and given a desired oversampling factor `ofac` (a typical value being 4 or larger), this routine fills array `px[0..nout-1]` with an increasing sequence of frequencies (not angular frequencies) up to `hifac` times the "average" Nyquist frequency, and fills array `py[0..nout-1]` with the values of the Lomb normalized periodogram at those frequencies. The arrays `x` and `y` are not altered. The vectors `px` and `py` are resized to `nout` (eq. 13.8.9) if their initial size is less than this; otherwise, only their first `nout` components are filled. The routine also returns `jmax` such that `py[jmax]` is the maximum element in `py`, and `prob`, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of `prob` indicates that a significant periodic signal is present.
```
    const Doub TWOPI=6.283185307179586476;
    Int i,j,n=x.size(),np=px.size();
    Doub ave,c,cc,cwtau,effm,expy,pnow,pymax,s,ss,sumc,sumcy,sums,sumsh,
        sumsy,swtau,var,wtau,xave,xdif,xmax,xmin,yy,arg,wtemp;
    VecDoub wi(n),wpi(n),wpr(n),wr(n);
    nout=Int(0.5*ofac*hifac*n);
    if (np < nout) {px.resize(nout); py.resize(nout);}
    avevar(y,ave,var);                      Get mean and variance of the input data.
    if (var == 0.0) throw("zero variance in period");
    xmax=xmin=x[0];                         Go through data to get the range of abscis-
    for (j=0;j<n;j++) {                         sas.
        if (x[j] > xmax) xmax=x[j];
        if (x[j] < xmin) xmin=x[j];
    }
    xdif=xmax-xmin;
    xave=0.5*(xmax+xmin);
    pymax=0.0;
    pnow=1.0/(xdif*ofac);                   Starting frequency.
    for (j=0;j<n;j++) {                     Initialize values for the trigonometric recur-
        arg=TWOPI*((x[j]-xave)*pnow);           rences at each data point.
        wpr[j]= -2.0*SQR(sin(0.5*arg));
        wpi[j]=sin(arg);
        wr[j]=cos(arg);
        wi[j]=wpi[j];
    }
    for (i=0;i<nout;i++) {                    Main loop over the frequencies to be evalu-
        px[i]=pnow;                                 ated.
        sumsh=sumc=0.0;                       First, loop over the data to get τ and related
        for (j=0;j<n;j++) {                         quantities.
            c=wr[j];
            s=wi[j];
            sumsh += s*c;
            sumc += (c-s)*(c+s);
        }
        wtau=0.5*atan2(2.0*sumsh,sumc);
        swtau=sin(wtau);
        cwtau=cos(wtau);
        sums=sumc=sumsy=sumcy=0.0;            Then, loop over the data again to get the
        for (j=0;j<n;j++) {                         periodogram value.
            s=wi[j];
            c=wr[j];
            ss=s*cwtau-c*swtau;
            cc=c*cwtau+s*swtau;
            sums += ss*ss;
            sumc += cc*cc;
            yy=y[j]-ave;
            sumsy += yy*ss;
            sumcy += yy*cc;
            wr[j]=((wtemp=wr[j])*wpr[j]-wi[j]*wpi[j])+wr[j];    Update the trigono-
            wi[j]=(wi[j]*wpr[j]+wtemp*wpi[j])+wi[j];            metric recurrences.
        }
        py[i]=0.5*(sumcy*sumcy/sumc+sumsy*sumsy/sums)/var;
        if (py[i] >= pymax) pymax=py[jmax=i];
        pnow += 1.0/(ofac*xdif);              The next frequency.
    }
    expy=exp(-pymax);                         Evaluate statistical significance of the max-
    effm=2.0*nout/ofac;                             imum.
    prob=effm*expy;
    if (prob > 0.01) prob=1.0-pow(1.0-expy,effm);
}
```

13.8.1 Fast Computation of the Lomb Periodogram

We here show how equations (13.8.4) and (13.8.5) can be calculated — approximately, but to any desired precision — with an operation count only of order $N_P \log N_P$. The method uses the FFT, but it is in no sense an FFT periodogram of the data. It is an actual evaluation of equations (13.8.4) and (13.8.5), the Lomb normalized periodogram, with exactly that method's strengths and weaknesses. This fast algorithm, due to Press and Rybicki [6], makes feasible the application of the Lomb method to data sets at least as large as 10^6 points; it is already faster than straightforward evaluation of equations (13.8.4) and (13.8.5) for data sets as small as 60 or 100 points.

Notice that the trigonometric sums that occur in equations (13.8.5) and (13.8.4) can be reduced to four simpler sums. If we define

$$S_h \equiv \sum_{j=0}^{N-1} (h_j - \bar{h}) \sin(\omega t_j) \qquad C_h \equiv \sum_{j=0}^{N-1} (h_j - \bar{h}) \cos(\omega t_j) \tag{13.8.10}$$

and

$$S_2 \equiv \sum_{j=0}^{N-1} \sin(2\omega t_j) \qquad C_2 \equiv \sum_{j=0}^{N-1} \cos(2\omega t_j) \tag{13.8.11}$$

then

$$\begin{aligned}
\sum_{j=0}^{N-1}(h_j-\bar{h})\cos\omega(t_j-\tau)&=C_h\cos\omega\tau+S_h\sin\omega\tau\\
\sum_{j=0}^{N-1}(h_j-\bar{h})\sin\omega(t_j-\tau)&=S_h\cos\omega\tau-C_h\sin\omega\tau\\
\sum_{j=0}^{N-1}\cos^2\omega(t_j-\tau)&=\frac{N}{2}+\frac{1}{2}C_2\cos(2\omega\tau)+\frac{1}{2}S_2\sin(2\omega\tau)\\
\sum_{j=0}^{N-1}\sin^2\omega(t_j-\tau)&=\frac{N}{2}-\frac{1}{2}C_2\cos(2\omega\tau)-\frac{1}{2}S_2\sin(2\omega\tau)
\end{aligned} \tag{13.8.12}$$

Now notice that *if* the t_js *were* evenly spaced, then the four quantities S_h, C_h, S_2, and C_2 could be evaluated by two complex FFTs, and the results could then be substituted back through equation (13.8.12) to evaluate equations (13.8.5) and (13.8.4). The problem is therefore only to evaluate equations (13.8.10) and (13.8.11) for unevenly spaced data.

Interpolation, or rather reverse interpolation — we will here call it *extirpolation* — provides the key. Interpolation, as classically understood, uses several function values on a regular mesh to construct an accurate approximation at an arbitrary point. Extirpolation, just the opposite, *replaces* a function value at an arbitrary point by several function values on a regular mesh, doing this in such a way that sums over the mesh are an accurate approximation to sums over the original arbitrary point.

It is not hard to see that the weight functions for extirpolation are identical to those for interpolation. Suppose that the function $h(t)$ to be extirpolated is known only at the discrete (unevenly spaced) points $h(t_i)\equiv h_i$, and that the function $g(t)$ (which will be, e.g., $\cos\omega t$) can be evaluated anywhere. Let $\widehat{t}_k$ be a sequence of evenly spaced points on a regular mesh. Then Lagrange interpolation (§3.2) gives an approximation of the form

$$g(t)\approx\sum_k w_k(t)g(\widehat{t}_k) \tag{13.8.13}$$

where $w_k(t)$ are interpolation weights. Now let us evaluate a sum of interest by the following scheme:

$$\sum_{j=0}^{N-1}h_j g(t_j)\approx\sum_{j=0}^{N-1}h_j\left[\sum_k w_k(t_j)g(\widehat{t}_k)\right]=\sum_k\left[\sum_{j=0}^{N-1}h_j w_k(t_j)\right]g(\widehat{t}_k)\equiv\sum_k\widehat{h}_k\,g(\widehat{t}_k) \tag{13.8.14}$$

Here $\widehat{h}_k\equiv\sum_j h_j w_k(t_j)$. Notice that equation (13.8.14) replaces the original sum by one on the regular mesh. Notice also that the accuracy of equation (13.8.13) depends only on the fineness of the mesh with respect to the function g and has nothing to do with the spacing of the points t_j or the function h; therefore, the accuracy of equation (13.8.14) also has this property.

The general outline of the fast evaluation method is therefore this: (i) Choose a mesh size large enough to accommodate some desired oversampling factor, and large enough to have several extirpolation points per half-wavelength of the highest frequency of interest. (ii) Extirpolate the values h_i onto the mesh and take the FFT; this gives S_h and C_h in equation (13.8.10). (iii) Extirpolate the constant values 1 onto another mesh, and take its FFT; this, with some manipulation, gives S_2 and C_2 in equation (13.8.11). (iv) Evaluate equations (13.8.12), (13.8.5), and (13.8.4), in that order.

There are several other tricks involved in implementing this algorithm efficiently. You can figure most out from the code, but we will mention the following points: (a) A nice way to get transform values at frequencies 2ω instead of ω is to stretch the time-domain data by a factor 2, and then wrap it to double-cover the original length. (This trick goes back to Tukey.) In the program, this appears as a modulo function. (b) Trigonometric identities are used to get from the left-hand side of equation (13.8.5) to the various needed trigonometric functions of $\omega\tau$. C++ identifiers like, e.g., `cwt` and `hs2wt` represent quantities like, e.g., $\cos\omega\tau$ and $\frac{1}{2}\sin(2\omega\tau)$. (c) The function `spread` does extirpolation onto the M most nearly centered mesh points around an arbitrary point; its turgid code evaluates coefficients of the Lagrange interpolating polynomials, in an efficient manner.

fasper.h

```
void fasper(VecDoub_I &x, VecDoub_I &y, const Doub ofac, const Doub hifac,
    VecDoub_O &px, VecDoub_O &py, Int &nout, Int &jmax, Doub &prob) {
```

Given `n` data points with abscissas `x[0..n-1]` (which need not be equally spaced) and ordinates `y[0..n-1]`, and given a desired oversampling factor `ofac` (a typical value being 4 or larger), this routine fills array `px[0..nout-1]` with an increasing sequence of frequencies (not angular frequencies) up to `hifac` times the "average" Nyquist frequency, and fills array `py[0..nout-1]` with the values of the Lomb normalized periodogram at those frequencies. The arrays `x` and `y` are not altered. The vectors `px` and `py` are resized to `nout` (eq. 13.8.9) if their initial size is less than this; otherwise, only their first `nout` components are filled. The routine also returns `jmax` such that `py[jmax]` is the maximum element in `py`, and `prob`, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of `prob` indicates that a significant periodic signal is present.

```
    const Int MACC=4;
    Int j,k,nwk,nfreq,nfreqt,n=x.size(),np=px.size();
    Doub ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt,hs2wt,
        hypo,pmax,sterm,swt,var,xdif,xmax,xmin;
    nout=Int(0.5*ofac*hifac*n);
    nfreqt=Int(ofac*hifac*n*MACC);                Size the FFT as next power of 2 above
    nfreq=64;                                         nfreqt.
    while (nfreq < nfreqt) nfreq <<= 1;
    nwk=nfreq << 1;
    if (np < nout) {px.resize(nout); py.resize(nout);}
    avevar(y,ave,var);                            Compute the mean, variance, and range
    if (var == 0.0) throw("zero variance in fasper");     of the data.
    xmin=x[0];
    xmax=xmin;
    for (j=1;j<n;j++) {
        if (x[j] < xmin) xmin=x[j];
        if (x[j] > xmax) xmax=x[j];
    }
    xdif=xmax-xmin;
    VecDoub wk1(nwk,0.);                          Zero the workspaces.
    VecDoub wk2(nwk,0.);
    fac=nwk/(xdif*ofac);
    fndim=nwk;
    for (j=0;j<n;j++) {                           Extirpolate the data into the workspaces.
        ck=fmod((x[j]-xmin)*fac,fndim);
        ckk=2.0*(ck++);
        ckk=fmod(ckk,fndim);
        ++ckk;
        spread(y[j]-ave,wk1,ck,MACC);
        spread(1.0,wk2,ckk,MACC);
    }
    realft(wk1,1);                                Take the Fast Fourier Transforms.
    realft(wk2,1);
    df=1.0/(xdif*ofac);
    pmax = -1.0;
    for (k=2,j=0;j<nout;j++,k+=2) {               Compute the Lomb value for each fre-
        hypo=sqrt(wk2[k]*wk2[k]+wk2[k+1]*wk2[k+1]);          quency.
        hc2wt=0.5*wk2[k]/hypo;
        hs2wt=0.5*wk2[k+1]/hypo;
        cwt=sqrt(0.5+hc2wt);
        swt=SIGN(sqrt(0.5-hc2wt),hs2wt);
        den=0.5*n+hc2wt*wk2[k]+hs2wt*wk2[k+1];
        cterm=SQR(cwt*wk1[k]+swt*wk1[k+1])/den;
        sterm=SQR(cwt*wk1[k+1]-swt*wk1[k])/(n-den);
        px[j]=(j+1)*df;
        py[j]=(cterm+sterm)/(2.0*var);
        if (py[j] > pmax) pmax=py[jmax=j];
    }
    expy=exp(-pmax);                        Estimate significance of largest peak value.
    effm=2.0*nout/ofac;
    prob=effm*expy;
    if (prob > 0.01) prob=1.0-pow(1.0-expy,effm);
}
```

fasper.h

```
void spread(const Doub y, VecDoub_IO &yy, const Doub x, const Int m) {
```

Given an array yy[0..n-1], extirpolate (spread) a value y into m actual array elements that best approximate the "fictional" (i.e., possibly noninteger) array element number x. The weights used are coefficients of the Lagrange interpolating polynomial.

```
    static Int nfac[11]={0,1,1,2,6,24,120,720,5040,40320,362880};
    Int ihi,ilo,ix,j,nden,n=yy.size();
    Doub fac;
    if (m > 10) throw("factorial table too small in spread");
    ix=Int(x);
    if (x == Doub(ix)) yy[ix-1] += y;
    else {
        ilo=MIN(MAX(Int(x-0.5*m),0),Int(n-m));
        ihi=ilo+m;
        nden=nfac[m];
        fac=x-ilo-1;
        for (j=ilo+1;j<ihi;j++) fac *= (x-j-1);
        yy[ihi-1] += y*fac/(nden*(x-ihi));
        for (j=ihi-1;j>ilo;j--) {
            nden=(nden/(j-ilo))*(j-ihi);
            yy[j-1] += y*fac/(nden*(x-j));
        }
    }
}
```

CITED REFERENCES AND FURTHER READING:

Lomb, N.R. 1976, "Least-Squares Frequency Analysis of Unequally Spaced Data," *Astrophysics and Space Science*, vol. 39, pp. 447–462.[1]

Barning, F.J.M. 1963, "The Numerical Analysis of the Light-Curve of 12 Lacertae," *Bulletin of the Astronomical Institutes of the Netherlands*, vol. 17, pp. 22–28.[2]

Vaníček, P. 1971, "Further Development and Properties of the Spectral Analysis by Least Squares," *Astrophysics and Space Science*, vol. 12, pp. 10–33.[3]

Scargle, J.D. 1982, "Studies in Astronomical Time Series Analysis II. Statistical Aspects of Spectral Analysis of Unevenly Sampled Data," *Astrophysical Journal*, vol. 263, pp. 835–853.[4]

Horne, J.H., and Baliunas, S.L. 1986, "A Prescription for Period Analysis of Unevenly Sampled Time Series," *Astrophysical Journal*, vol. 302, pp. 757–763.[5]

Press, W.H. and Rybicki, G.B. 1989, "Fast Algorithm for Spectral Analysis of Unevenly Sampled Data," *Astrophysical Journal*, vol. 338, pp. 277–280.[6]

13.9 Computing Fourier Integrals Using the FFT

Not uncommonly, one wants to calculate accurate numerical values for integrals of the form

$$I = \int_a^b e^{i\omega t} h(t) dt \ , \tag{13.9.1}$$

or the equivalent real and imaginary parts

$$I_c = \int_a^b \cos(\omega t)h(t)dt \qquad I_s = \int_a^b \sin(\omega t)h(t)dt \ , \tag{13.9.2}$$

and one wants to evaluate this integral for many different values of ω. In cases of interest, $h(t)$ is often a smooth function, but it is not necessarily periodic in $[a, b]$, nor does it necessarily go to zero at a or b. While it seems intuitively obvious that the *force majeure* of the FFT ought to be applicable to this problem, doing so turns out to be a surprisingly subtle matter, as we will now see.

Let us first approach the problem naively, to see where the difficulty lies. Divide the interval $[a, b]$ into M subintervals, where M is a large integer, and define

$$\Delta \equiv \frac{b-a}{M}, \quad t_j \equiv a + j\Delta, \quad h_j \equiv h(t_j), \quad j = 0, \ldots, M \tag{13.9.3}$$

Notice that $h_0 = h(a)$ and $h_M = h(b)$, and that there are $M + 1$ values h_j. We can approximate the integral I by a sum,

$$I \approx \Delta \sum_{j=0}^{M-1} h_j \exp(i\omega t_j) \tag{13.9.4}$$

which is at any rate first-order accurate. (If we centered the h_j's and the t_j's in the intervals, we could be accurate to second order.) Now, for certain values of ω and M, the sum in equation (13.9.4) can be made into a discrete Fourier transform, or DFT, and evaluated by the fast Fourier transform (FFT) algorithm. In particular, we can choose M to be an integer power of 2 and define a set of special ω's by

$$\omega_m \Delta \equiv \frac{2\pi m}{M} \tag{13.9.5}$$

where m has the values $m = 0, 1, \ldots, M/2 - 1$. Then equation (13.9.4) becomes

$$I(\omega_m) \approx \Delta e^{i\omega_m a} \sum_{j=0}^{M-1} h_j e^{2\pi i m j/M} = \Delta e^{i\omega_m a}[\text{DFT}(h_0 \ldots h_{M-1})]_m \tag{13.9.6}$$

Equation (13.9.6), while simple and clear, is emphatically *not recommended* for use: It is likely to give wrong answers!

The problem lies in the oscillatory nature of the integral (13.9.1). If $h(t)$ is at all smooth, and if ω is large enough to imply several cycles in the interval $[a, b]$ — in fact, ω_m in equation (13.9.5) gives exactly m cycles — then the value of I is typically very small, so small that it is easily swamped by first-order, or even (with centered values) second-order, truncation error. Furthermore, the characteristic "small parameter" that occurs in the error term is not $\Delta/(b - a) = 1/M$, as it would be if the integrand were not oscillatory, but $\omega\Delta$, which can be as large as π for ω's within the Nyquist interval of the DFT (cf. equation 13.9.5). The result is that equation (13.9.6) becomes systematically inaccurate as ω increases.

It is a sobering exercise to implement equation (13.9.6) for an integral that can be done analytically and to see just how bad it is. We recommend that you try it.

Let us therefore turn to a more sophisticated treatment. Given the sampled points h_j, we can approximate the function $h(t)$ everywhere in the interval $[a, b]$ by interpolation on nearby h_j's. The simplest case is linear interpolation, using the two nearest h_j's, one to the left and one to the right. A higher-order interpolation, e.g., would be cubic interpolation, using two points to the left and two to the right — except in the first and last subintervals, where we must interpolate with three h_j's on one side, one on the other.

The formulas for such interpolation schemes are (piecewise) polynomial in the independent variable t, but with coefficients that are of course linear in the function values h_j.

Although one does not usually think of it in this way, interpolation can be viewed as approximating a function by a sum of kernel functions (which depend only on the interpolation scheme) times sample values (which depend only on the function). Let us write

$$h(t) \approx \sum_{j=0}^{M} h_j \, \psi\left(\frac{t-t_j}{\Delta}\right) + \sum_{j=\text{endpoints}} h_j \, \varphi_j\left(\frac{t-t_j}{\Delta}\right) \tag{13.9.7}$$

Here $\psi(s)$ is the kernel function of an interior point: It is zero for s sufficiently negative or sufficiently positive and becomes nonzero only when s is in the range where the h_j multiplying it is actually used in the interpolation. We always have $\psi(0) = 1$ and $\psi(m) = 0$, $m = \pm 1, \pm 2, \dots$, since interpolation right on a sample point should give the sampled function value. For linear interpolation, $\psi(s)$ is piecewise linear, rises from 0 to 1 for s in $(-1, 0)$, and falls back to 0 for s in $(0, 1)$. For higher-order interpolation, $\psi(s)$ is made up piecewise of segments of Lagrange interpolation polynomials. It has discontinuous derivatives at integer values of s, where the pieces join, because the set of points used in the interpolation changes discretely.

As already remarked, the subintervals closest to a and b require different (noncentered) interpolation formulas. This is reflected in equation (13.9.7) by the second sum, with the special endpoint kernels $\varphi_j(s)$. Actually, for reasons that will become clearer below, we have included *all* the points in the *first* sum (with kernel ψ), so the φ_j's are actually differences between true endpoint kernels and the interior kernel ψ. It is a tedious, but straightforward, exercise to write down all the $\varphi_j(s)$'s for any particular order of interpolation, each one consisting of differences of Lagrange interpolating polynomials spliced together piecewise.

Now apply the integral operator $\int_a^b dt \exp(i\omega t)$ to both sides of equation (13.9.7), interchange the sums and integral, and make the changes of variable $s = (t - t_j)/\Delta$ in the first sum and $s = (t - a)/\Delta$ in the second sum. The result is

$$I \approx \Delta e^{i\omega a}\left[W(\theta)\sum_{j=0}^{M} h_j e^{ij\theta} + \sum_{j=\text{endpoints}} h_j \alpha_j(\theta)\right] \tag{13.9.8}$$

Here $\theta \equiv \omega\Delta$, and the functions $W(\theta)$ and $\alpha_j(\theta)$ are defined by

$$W(\theta) \equiv \int_{-\infty}^{\infty} ds \, e^{i\theta s} \psi(s) \tag{13.9.9}$$

$$\alpha_j(\theta) \equiv \int_{-\infty}^{\infty} ds \, e^{i\theta s} \varphi_j(s - j) \tag{13.9.10}$$

The key point is that equations (13.9.9) and (13.9.10) can be evaluated, analytically, once and for all, for any given interpolation scheme. Then equation (13.9.8) is an algorithm for applying "endpoint corrections" to a sum that (as we will see) can be done using the FFT, giving a result with high-order accuracy.

We will consider only interpolations that are left-right symmetric. Then symmetry implies

$$\varphi_{M-j}(s) = \varphi_j(-s) \qquad \alpha_{M-j}(\theta) = e^{i\theta M}\alpha_j^*(\theta) = e^{i\omega(b-a)}\alpha_j^*(\theta) \tag{13.9.11}$$

where $*$ denotes complex conjugation. Also, $\psi(s) = \psi(-s)$ implies that $W(\theta)$ is real.

Turn now to the first sum in equation (13.9.8), which we want to do by FFT methods. To do so, choose some N that is an integer power of 2 with $N \geq M+1$. (Note that M need not be a power of 2, so $M = N-1$ is allowed.) If $N > M+1$, define $h_j \equiv 0$, $M+1 < j \leq N-1$, i.e., "zero-pad" the array of h_j's so that j takes on the range $0 \leq j \leq N-1$. Then the sum can be done as a DFT for the special values $\omega = \omega_n$ given by

$$\omega_n \Delta \equiv \frac{2\pi n}{N} \equiv \theta \qquad n = 0, 1, \dots, \frac{N}{2} - 1 \tag{13.9.12}$$

For fixed M, the larger N is chosen, the finer the sampling in frequency space. The value M, on the other hand, determines the *highest* frequency sampled, since Δ decreases with increasing M (equation 13.9.3), and the largest value of $\omega\Delta$ is always just under π (equation 13.9.12). In general it is advantageous to oversample by *at least* a factor of 4, i.e., $N > 4M$ (see below). We can now rewrite equation (13.9.8) in its final form as

$$\begin{aligned}I(\omega_n) = \Delta e^{i\omega_n a}\Big\{&W(\theta)[\text{DFT}(h_0 \ldots h_{N-1})]_n \\ &+ \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 + \ldots \\ &+ e^{i\omega(b-a)}\left[\alpha_0^*(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3} + \ldots\right]\Big\}\end{aligned} \tag{13.9.13}$$

For cubic (or lower) polynomial interpolation, at most the terms explicitly shown above are nonzero; the ellipses $(\ldots)$ can therefore be ignored, and we need explicit forms only for the functions $W, \alpha_0, \alpha_1, \alpha_2, \alpha_3$, calculated with equations (13.9.9) and (13.9.10). We have worked these out for you, in the trapezoidal (second-order) and cubic (fourth-order) cases. Here are the results, along with the first few terms of their power series expansions for small θ:

Trapezoidal order:

$$W(\theta) = \frac{2(1-\cos\theta)}{\theta^2} \approx 1 - \frac{1}{12}\theta^2 + \frac{1}{360}\theta^4 - \frac{1}{20160}\theta^6$$

$$\begin{aligned}\alpha_0(\theta) &= -\frac{(1-\cos\theta)}{\theta^2} + i\frac{(\theta - \sin\theta)}{\theta^2} \\ &\approx -\frac{1}{2} + \frac{1}{24}\theta^2 - \frac{1}{720}\theta^4 + \frac{1}{40320}\theta^6 + i\theta\left(\frac{1}{6} - \frac{1}{120}\theta^2 + \frac{1}{5040}\theta^4 - \frac{1}{362880}\theta^6\right)\end{aligned}$$

$$\alpha_1 = \alpha_2 = \alpha_3 = 0$$

Cubic order:

$$W(\theta) = \left(\frac{6+\theta^2}{3\theta^4}\right)(3 - 4\cos\theta + \cos 2\theta) \approx 1 - \frac{11}{720}\theta^4 + \frac{23}{15120}\theta^6$$

$$\begin{aligned}\alpha_0(\theta) &= \frac{(-42+5\theta^2) + (6+\theta^2)(8\cos\theta - \cos 2\theta)}{6\theta^4} + i\frac{(-12\theta + 6\theta^3) + (6+\theta^2)\sin 2\theta}{6\theta^4} \\ &\approx -\frac{2}{3} + \frac{1}{45}\theta^2 + \frac{103}{15120}\theta^4 - \frac{169}{226800}\theta^6 + i\theta\left(\frac{2}{45} + \frac{2}{105}\theta^2 - \frac{8}{2835}\theta^4 + \frac{86}{467775}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_1(\theta) &= \frac{14(3-\theta^2) - 7(6+\theta^2)\cos\theta}{6\theta^4} + i\frac{30\theta - 5(6+\theta^2)\sin\theta}{6\theta^4} \\ &\approx \frac{7}{24} - \frac{7}{180}\theta^2 + \frac{5}{3456}\theta^4 - \frac{7}{259200}\theta^6 + i\theta\left(\frac{7}{72} - \frac{1}{168}\theta^2 + \frac{11}{72576}\theta^4 - \frac{13}{5987520}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_2(\theta) &= \frac{-4(3-\theta^2) + 2(6+\theta^2)\cos\theta}{3\theta^4} + i\frac{-12\theta + 2(6+\theta^2)\sin\theta}{3\theta^4} \\ &\approx -\frac{1}{6} + \frac{1}{45}\theta^2 - \frac{5}{6048}\theta^4 + \frac{1}{64800}\theta^6 + i\theta\left(-\frac{7}{90} + \frac{1}{210}\theta^2 - \frac{11}{90720}\theta^4 + \frac{13}{7484400}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_3(\theta) &= \frac{2(3-\theta^2) - (6+\theta^2)\cos\theta}{6\theta^4} + i\frac{6\theta - (6+\theta^2)\sin\theta}{6\theta^4} \\ &\approx \frac{1}{24} - \frac{1}{180}\theta^2 + \frac{5}{24192}\theta^4 - \frac{1}{259200}\theta^6 + i\theta\left(\frac{7}{360} - \frac{1}{840}\theta^2 + \frac{11}{362880}\theta^4 - \frac{13}{29937600}\theta^6\right)\end{aligned}$$

The program `dftcor`, below, implements the endpoint corrections for the cubic case. Given input values of ω, Δ, a, b, and an array with the eight values $h_0, \ldots, h_3, h_{M-3}, \ldots, h_M$, it returns the real and imaginary parts of the endpoint corrections in equation (13.9.13), and the factor $W(\theta)$. The code is turgid, but only because the formulas above are complicated. The formulas have cancellations to high powers of θ. It is therefore necessary to compute the right-hand sides in double precision, even when the corrections are desired only to single precision. It is also necessary to use the series expansion for small values of θ. The optimal cross-over value of θ depends on your machine's wordlength, but you can always find it experimentally as the largest value where the two methods give identical results to machine precision.

dftintegrate.h

```
void dftcor(const Doub w, const Doub delta, const Doub a, const Doub b,
    VecDoub_I &endpts, Doub &corre, Doub &corim, Doub &corfac) {
```
For an integral approximated by a discrete Fourier transform, this routine computes the correction factor that multiplies the DFT and the endpoint correction to be added. Input is the angular frequency `w`, stepsize `delta`, lower and upper limits of the integral `a` and `b`, while the array `endpts` contains the first 4 and last 4 function values. The correction factor $W(\theta)$ is returned as `corfac`, while the real and imaginary parts of the endpoint correction are returned as `corre` and `corim`.
```
    Doub a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t,t2,t4,t6,
        cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,tmth2,tth4i;
    th=w*delta;
    if (a >= b || th < 0.0e0 || th > 3.1416e0)
        throw("bad arguments to dftcor");
    if (abs(th) < 5.0e-2) {          Use series.
        t=th;
        t2=t*t;
        t4=t2*t2;
        t6=t4*t2;
        corfac=1.0-(11.0/720.0)*t4+(23.0/15120.0)*t6;
        a0r=(-2.0/3.0)+t2/45.0+(103.0/15120.0)*t4-(169.0/226800.0)*t6;
        a1r=(7.0/24.0)-(7.0/180.0)*t2+(5.0/3456.0)*t4-(7.0/259200.0)*t6;
        a2r=(-1.0/6.0)+t2/45.0-(5.0/6048.0)*t4+t6/64800.0;
        a3r=(1.0/24.0)-t2/180.0+(5.0/24192.0)*t4-t6/259200.0;
        a0i=t*(2.0/45.0+(2.0/105.0)*t2-(8.0/2835.0)*t4+(86.0/467775.0)*t6);
        a1i=t*(7.0/72.0-t2/168.0+(11.0/72576.0)*t4-(13.0/5987520.0)*t6);
        a2i=t*(-7.0/90.0+t2/210.0-(11.0/90720.0)*t4+(13.0/7484400.0)*t6);
        a3i=t*(7.0/360.0-t2/840.0+(11.0/362880.0)*t4-(13.0/29937600.0)*t6);
    } else {                         Use trigonometric formulas.
        cth=cos(th);
        sth=sin(th);
        ctth=cth*cth-sth*sth;
        stth=2.0e0*sth*cth;
        th2=th*th;
        th4=th2*th2;
        tmth2=3.0e0-th2;
        spth2=6.0e0+th2;
        sth4i=1.0/(6.0e0*th4);
        tth4i=2.0e0*sth4i;
        corfac=tth4i*spth2*(3.0e0-4.0e0*cth+ctth);
        a0r=sth4i*(-42.0e0+5.0e0*th2+spth2*(8.0e0*cth-ctth));
        a0i=sth4i*(th*(-12.0e0+6.0e0*th2)+spth2*stth);
        a1r=sth4i*(14.0e0*tmth2-7.0e0*spth2*cth);
        a1i=sth4i*(30.0e0*th-5.0e0*spth2*sth);
        a2r=tth4i*(-4.0e0*tmth2+2.0e0*spth2*cth);
        a2i=tth4i*(-12.0e0*th+2.0e0*spth2*sth);
        a3r=sth4i*(2.0e0*tmth2-spth2*cth);
        a3i=sth4i*(6.0e0*th-spth2*sth);
    }
    cl=a0r*endpts[0]+a1r*endpts[1]+a2r*endpts[2]+a3r*endpts[3];
    sl=a0i*endpts[0]+a1i*endpts[1]+a2i*endpts[2]+a3i*endpts[3];
    cr=a0r*endpts[7]+a1r*endpts[6]+a2r*endpts[5]+a3r*endpts[4];
    sr= -a0i*endpts[7]-a1i*endpts[6]-a2i*endpts[5]-a3i*endpts[4];
    arg=w*(b-a);
    c=cos(arg);
    s=sin(arg);
    corre=cl+c*cr-s*sr;
    corim=sl+s*cr+c*sr;
}
```

Since the use of `dftcor` can be confusing, we also give an illustrative program `dftint` that uses `dftcor` to compute equation (13.9.1) for general a, b, ω, and $h(t)$. Several points within this program bear mentioning: The constants `M` and `NDFT` correspond to M and N in the above discussion. On successive calls, we recompute the Fourier transform only if a or b or $h(t)$ has changed.

Since `dftint` is designed to work for any value of ω satisfying $\omega\Delta < \pi$, not just the special values returned by the DFT (equation 13.9.12), we do polynomial interpolation of degree `MPOL` on the DFT spectrum. You should be warned that a large factor of oversampling ($N \gg M$) is required for this interpolation to be accurate. After interpolation, we add the endpoint corrections from `dftcor`, which can be evaluated for any ω.

While `dftcor` is good at what it does, the routine `dftint` is illustrative only. It is not a general-purpose program, because it does not adapt its parameters `M`, `NDFT`, `MPOL` or its interpolation scheme to any particular function $h(t)$. You will have to experiment with your own application.

dftintegrate.h

```
void dftint(Doub func(const Doub), const Doub a, const Doub b, const Doub w,
    Doub &cosint, Doub &sinint) {
```

Example program illustrating how to use the routine `dftcor`. The user supplies an external function `func` that returns the quantity $h(t)$. The routine then returns $\int_a^b \cos(\omega t)h(t)\,dt$ as `cosint` and $\int_a^b \sin(\omega t)h(t)\,dt$ as `sinint`.

```
    static Int init=0;
    static Doub (*funcold)(const Doub);
    static Doub aold = -1.e30,bold = -1.e30,delta;
    const Int M=64,NDFT=1024,MPOL=6;
```

The values of `M`, `NDFT`, and `MPOL` are merely illustrative and should be optimized for your particular application. `M` is the number of subintervals, `NDFT` is the length of the FFT (a power of 2), and `MPOL` is the degree of polynomial interpolation used to obtain the desired frequency from the FFT.

```
    const Doub TWOPI=6.283185307179586476;
    Int j,nn;
    Doub c,cdft,corfac,corim,corre,en,s,sdft;
    static VecDoub data(NDFT),endpts(8);
    VecDoub cpol(MPOL),spol(MPOL),xpol(MPOL);
    if (init != 1 || a != aold || b != bold || func != funcold) {
```

Do we need to initialize, or is only ω changed?

```
        init=1;
        aold=a;
        bold=b;
        funcold=func;
        delta=(b-a)/M;
        for (j=0;j<M+1;j++)                    Load the function values into the data
            data[j]=func(a+j*delta);               array.
        for (j=M+1;j<NDFT;j++)                 Zero-pad the rest of the data array.
            data[j]=0.0;
        for (j=0;j<4;j++) {                    Load the endpoints.
            endpts[j]=data[j];
            endpts[j+4]=data[M-3+j];
        }
        realft(data,1);
```

`realft` returns the unused value corresponding to $\omega_{N/2}$ in `data[1]`. We actually want this element to contain the imaginary part corresponding to ω_0, which is zero.

```
        data[1]=0.0;
    }
```

Now interpolate on the DFT result for the desired frequency. If the frequency is an ω_n, i.e., the quantity en is an integer, then cdft=data[2*en-2], sdft=data[2*en-1], and you could omit the interpolation.

```
    en=w*delta*NDFT/TWOPI+1.0;
    nn=MIN(MAX(Int(en-0.5*MPOL+1.0),1),NDFT/2-MPOL+1);       Leftmost point for the
    for (j=0;j<MPOL;j++,nn++) {                                 interpolation.
        cpol[j]=data[2*nn-2];
        spol[j]=data[2*nn-1];
        xpol[j]=nn;
    }
    cdft = Poly_interp(xpol,cpol,MPOL).interp(en);
    sdft = Poly_interp(xpol,spol,MPOL).interp(en);
    dftcor(w,delta,a,b,endpts,corre,corim,corfac);           Now get the endpoint cor-
    cdft *= corfac;                                             rection and the mul-
    sdft *= corfac;                                             tiplicative factor W(θ).
    cdft += corre;
    sdft += corim;
    c=delta*cos(w*a);                                  Finally multiply by Δ and exp(iωa).
    s=delta*sin(w*a);
    cosint=c*cdft-s*sdft;
    sinint=s*cdft+c*sdft;
}
```

Sometimes one is interested only in the discrete frequencies ω_m of equation (13.9.5), the ones that have integral numbers of periods in the interval $[a, b]$. For smooth $h(t)$, the value of I tends to be much smaller in magnitude at these ω's than at values in between, since the integral half-periods tend to cancel precisely. (That is why one must oversample for interpolation to be accurate: $I(\omega)$ is oscillatory with small magnitude near the ω_m's.) If you want these ω_m's without messy (and possibly inaccurate) interpolation, you have to set N to a multiple of M (compare equations 13.9.5 and 13.9.12). In the method implemented above, however, N must be at least $M + 1$, so the smallest such multiple is $2M$, resulting in a factor $\sim$2 unnecessary computing. Alternatively, one can derive a formula like equation (13.9.13), but with the last function sample $h_M = h(b)$ omitted from the DFT, but included entirely in the endpoint correction for h_M. Then one can set $M = N$ (an integer power of 2) and get the special frequencies of equation (13.9.5) with no additional overhead. The modified formula is

$$\begin{aligned} I(\omega_m) = \Delta e^{i\omega_m a}\Big\{&W(\theta)\,[\text{DFT}(h_0 \ldots h_{M-1})]_m \\ &+ \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 \\ &+ e^{i\omega(b-a)}\left[A(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3}\right]\Big\} \end{aligned} \tag{13.9.14}$$

where $\theta \equiv \omega_m \Delta$ and $A(\theta)$ is given by

$$A(\theta) = -\alpha_0(\theta) \tag{13.9.15}$$

for the trapezoidal case, or

$$\begin{aligned} A(\theta) &= \frac{(-6 + 11\theta^2) + (6 + \theta^2)\cos 2\theta}{6\theta^4} - i\,\text{Im}[\alpha_0(\theta)] \\ &\approx \frac{1}{3} + \frac{1}{45}\theta^2 - \frac{8}{945}\theta^4 + \frac{11}{14175}\theta^6 - i\,\text{Im}[\alpha_0(\theta)] \end{aligned} \tag{13.9.16}$$

for the cubic case.

Factors like $W(\theta)$ arise naturally whenever one calculates Fourier coefficients of smooth functions, and they are sometimes called attenuation factors [1]. However, the endpoint corrections are equally important in obtaining accurate values of integrals. Narasimhan and Karthikeyan [2] have given a formula that is algebraically equivalent to our trapezoidal formula. However, their formula requires the evaluation of *two* FFTs, which is unnecessary. The basic idea used here goes back at least to Filon [3] in 1928 (before the FFT!). He used Simpson's rule (quadratic interpolation). Since this interpolation is not left-right symmetric, two Fourier transforms are required. An alternative algorithm for equation (13.9.14) has been given by Lyness in [4]; for related references, see [5]. To our knowledge, the cubic-order formulas derived here have not previously appeared in the literature.

Calculating Fourier transforms when the range of integration is $(-\infty, \infty)$ can be tricky. If the function falls off reasonably quickly at infinity, you can split the integral at a large enough value of t. For example, the integration to $+\infty$ can be written

$$\begin{aligned}\int_a^\infty e^{i\omega t} h(t)\,dt &= \int_a^b e^{i\omega t} h(t)\,dt + \int_b^\infty e^{i\omega t} h(t)\,dt \\ &= \int_a^b e^{i\omega t} h(t)\,dt - \frac{h(b)e^{i\omega b}}{i\omega} + \frac{h'(b)e^{i\omega b}}{(i\omega)^2} - \cdots\end{aligned} \tag{13.9.17}$$

The splitting point b must be chosen large enough that the remaining integral over (b, ∞) is small. Successive terms in its asymptotic expansion are found by integrating by parts. The integral over (a, b) can be done using `dftint`. You keep as many terms in the asymptotic expansion as you can easily compute. See [6] for some examples of this idea. More powerful methods, which work well for long-tailed functions but which do not use the FFT, are described in [7-9].

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.3.4.[1]

Narasimhan, M.S. and Karthikeyan, M. 1984, "Evaluation of Fourier Integrals Using a FFT with Improved Accuracy and Its Applications," *IEEE Transactions on Antennas and Propagation*, vol. 32, pp. 404–408.[2]

Filon, L.N.G. 1928, "On a Quadrature Formula for Trigonometric Integrals," *Proceedings of the Royal Society of Edinburgh*, vol. 49, pp. 38–47.[3]

Giunta, G. and Murli, A. 1987, "A Package for Computing Trigonometric Fourier Coefficients Based on Lyness's Algorithm," *ACM Transactions on Mathematical Software*, vol. 13, pp. 97–107.[4]

Lyness, J.N. 1987, in *Numerical Integration*, P. Keast and G. Fairweather, eds. (Dordrecht: Reidel).[5]

Pantis, G. 1975, "The Evaluation of Integrals with Oscillatory Integrands," *Journal of Computational Physics*, vol. 17, pp. 229–233.[6]

Blakemore, M., Evans, G.A., and Hyslop, J. 1976, "Comparison of Some Methods for Evaluating Infinite Range Oscillatory Integrals," *Journal of Computational Physics*, vol. 22, pp. 352–376.[7]

Lyness, J.N., and Kaper, T.J. 1987, "Calculating Fourier Transforms of Long Tailed Functions," *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 1005–1011.[8]

Thakkar, A.J., and Smith, V.H. 1975, "A Strategy for the Numerical Evaluation of Fourier Sine and Cosine Transforms to Controlled Accuracy," *Computer Physics Communications*, vol. 10, pp. 73–79.[9]

13.10 Wavelet Transforms

Like the fast Fourier transform (FFT), the discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is an integer power of 2, transforming it into a numerically different vector of the same length. Also like the FFT, the wavelet transform is invertible and in fact orthogonal — the inverse transform, when viewed as a big matrix, is simply the transpose of the transform.

Both FFT and DWT, therefore, can be viewed as a rotation in function space, from the input space (or time) domain, where the basis functions are the unit vectors $\mathbf{e}_i$, or Dirac delta functions in the continuum limit, to a different domain. For the FFT, this new domain has basis functions that are the familiar sines and cosines. In the wavelet domain, the basis functions are somewhat more complicated and have the fanciful names "mother functions" and "wavelets."

Of course there are an infinity of possible bases for function space, almost all of them uninteresting! What makes the wavelet basis interesting is that, *unlike* sines and cosines, individual wavelet functions are quite localized in space; simultaneously, *like* sines and cosines, individual wavelet functions are quite localized in frequency or (more precisely) characteristic scale. As we will see below, the particular kind of dual localization achieved by wavelets renders large classes of functions and operators sparse, or sparse to some high accuracy, when transformed into the wavelet domain. Analogously with the Fourier domain, where a class of computations, like convolutions, becomes computationally fast, there is a large class of computations — those that can take advantage of sparsity — that becomes computationally fast in the wavelet domain [1].

Unlike sines and cosines, which define a unique Fourier transform, there is not one single unique set of wavelets; in fact, there are infinitely many possible sets. Roughly, the different sets of wavelets make different trade-offs between how compactly they are localized in space, how smooth they are, and whether they have any special boundary conditions. (There are further fine distinctions.)

13.10.1 Daubechies Wavelet Filter Coefficients

A particular set of wavelets is specified by a particular set of numbers, called *wavelet filter coefficients*. Here, we will largely restrict ourselves to wavelet filters in a class discovered by Daubechies [2]. This class includes members ranging from highly localized to highly smooth. The simplest (and most localized) member, often called *DAUB4*, has only four coefficients, $c_0, \dots, c_3$. For the moment we specialize to this case for ease of notation.

Consider the following transformation matrix acting on a column vector of data to its right:

$$\begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & \\
\vdots & \vdots & & & & & \ddots & & & \\
 & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & c_3 & -c_2 & c_1 & -c_0 \\
c_2 & c_3 & & & & & & & c_0 & c_1 \\
c_1 & -c_0 & & & & & & & c_3 & -c_2
\end{bmatrix} \qquad (13.10.1)$$

Here blank entries signify zeroes. Note the structure of this matrix. The first row convolves four consecutive data points with the filter coefficients $c_0 \dots, c_3$; likewise, the third, fifth, and other odd rows. If the even rows followed this pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. (Note how the last two rows wrap around like convolutions with periodic boundary conditions.) Instead of convolving with $c_0, \ldots, c_3$, however, the even rows perform a different convolution, with coefficients $c_3, -c_2, c_1, -c_0$. The action of the matrix, overall, is thus to perform two related convolutions, then to decimate each of them by half (throw away half the values), and interleave the remaining halves.

It is useful to think of the filter $c_0, \ldots, c_3$ as being a smoothing filter, call it H, something like a moving average of four points. Then, because of the minus signs, the filter $c_3, -c_2, c_1, -c_0$, call it G, is *not* a smoothing filter. (In signal processing contexts, H and G are called *quadrature mirror filters* [3].) In fact, the c's are chosen so as to make G yield, insofar as possible, a *zero* response to a sufficiently smooth data vector. This is done by requiring the sequence $c_3, -c_2, c_1, -c_0$ to have a certain number of vanishing moments. When this is the case for p moments (starting with the zeroth), a set of wavelets is said to satisfy an "approximation condition of order p." This results in the output of H, decimated by half, accurately representing the data's "smooth" information. The output of G, also decimated, is referred to as the data's "detail" information [4].

For such a characterization to be useful, it must be possible to reconstruct the original data vector of length N from its $N/2$ smooth or s-components and its $N/2$ detail or d-components. That is effected by requiring the matrix (13.10.1) to be orthogonal, so that its inverse is just the transposed matrix

$$\begin{bmatrix}
c_0 & c_3 & & & \cdots & & & & c_2 & c_1 \\
c_1 & -c_2 & & & \cdots & & & & c_3 & -c_0 \\
c_2 & c_1 & c_0 & c_3 & & & & & & \\
c_3 & -c_0 & c_1 & -c_2 & & & & & & \\
 & & & & \ddots & & & & & \\
 & & & & & c_2 & c_1 & c_0 & c_3 & \\
 & & & & & c_3 & -c_0 & c_1 & -c_2 & \\
 & & & & & & & c_2 & c_1 & c_0 & c_3 \\
 & & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{bmatrix} \tag{13.10.2}$$

One sees immediately that matrix (13.10.2) is inverse to matrix (13.10.1) if and only if these two equations hold,

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2c_0 + c_3c_1 &= 0 \end{aligned} \tag{13.10.3}$$

If additionally we require the approximation condition of order $p = 2$, then two additional relations are required,

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \tag{13.10.4}$$

Equations (13.10.3) and (13.10.4) are four equations for the four unknowns $c_0, \ldots, c_3$, first recognized and solved by Daubechies. The unique solution (up to a left-right reversal) is

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} \qquad & c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} \qquad & c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \tag{13.10.5}$$

In fact, DAUB4 is only the most compact of a sequence of wavelet sets: If we had six coefficients instead of four, there would be three orthogonality requirements in equation (13.10.3) (with offsets of zero, two, and four), and we could require the vanishing of $p = 3$ moments in equation (13.10.4). In this case, DAUB6, the solution coefficients can also be expressed in closed form,

$$\begin{aligned}
c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/16\sqrt{2} & c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/16\sqrt{2}\\
c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/16\sqrt{2} & c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/16\sqrt{2}\\
c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/16\sqrt{2} & c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/16\sqrt{2}
\end{aligned} \tag{13.10.6}$$

For higher p, the coefficients are available only numerically, e.g., tabulated in [5] or [6]. (We use some of these below.) The number of coefficients increases by two each time p is increased by one.

13.10.2 Discrete Wavelet Transform

We have not yet defined the discrete wavelet transform (DWT), but we are almost there: The DWT consists of applying a wavelet coefficient matrix like (13.10.1) *hierarchically*, first to the full data vector of length N, then to the "smooth" vector of length $N/2$, then to the "smooth-smooth" vector of length $N/4$, and so on until only a trivial number of "smooth-...-smooth" components (usually 2 or 4) remain. The procedure is sometimes called a *pyramidal algorithm* [4], for obvious reasons. The output of the DWT consists of these remaining components and all the "detail" components that were accumulated along the way. A diagram should make the procedure clear:

$$\begin{bmatrix} y_0\\ y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7\\ y_8\\ y_9\\ y_{10}\\ y_{11}\\ y_{12}\\ y_{13}\\ y_{14}\\ y_{15} \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} s_0\\ d_0\\ s_1\\ d_1\\ s_2\\ d_2\\ s_3\\ d_3\\ s_4\\ d_4\\ s_5\\ d_5\\ s_6\\ d_6\\ s_7\\ d_7 \end{bmatrix}
\xrightarrow{\text{permute}}
\left[\begin{array}{c} s_0\\ s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ \hline d_0\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7 \end{array}\right]
\xrightarrow{13.10.1}
\left[\begin{array}{c} S_0\\ D_0\\ S_1\\ D_1\\ S_2\\ D_2\\ S_3\\ D_3\\ \hline d_0\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7 \end{array}\right]
\xrightarrow{\text{permute}}
\left[\begin{array}{c} S_0\\ S_1\\ S_2\\ S_3\\ \hline D_0\\ D_1\\ D_2\\ D_3\\ \hline d_0\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7 \end{array}\right]
\xrightarrow{\text{etc.}}
\left[\begin{array}{c} \mathcal{S}_0\\ \mathcal{S}_1\\ \hline \mathcal{D}_0\\ \mathcal{D}_1\\ \hline D_0\\ D_1\\ D_2\\ D_3\\ \hline d_0\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7 \end{array}\right] \tag{13.10.7}$$

If the length of the data vector were a higher power of 2, there would be more stages of applying (13.10.1) (or any other wavelet coefficients) and permuting. The endpoint will always be a vector with two $\mathcal{S}$'s and a hierarchy of $\mathcal{D}$'s, D's, d's, etc. Notice that once d's are generated, they simply propagate through to all subsequent stages.

A value d_i of any level is termed a "wavelet coefficient" of the original data vector; the final values $\mathcal{S}_0, \mathcal{S}_1$ should strictly be called "mother-function coefficients," although the term "wavelet coefficients" is often used loosely for both d's and final $\mathcal{S}$'s. Since the full procedure is a composition of orthogonal linear operations, the whole DWT is itself an orthogonal linear operator.

To invert the DWT, one simply reverses the procedure, starting with the smallest level of the hierarchy and working (in equation 13.10.7) from right to left. The inverse matrix (13.10.2) is of course used instead of the matrix (13.10.1).

As already noted, the matrices (13.10.1) and (13.10.2) embody periodic ("wraparound") boundary conditions on the data vector. One normally accepts this as a minor inconvenience: The last few wavelet coefficients at each level of the hierarchy are affected by data from both ends of the data vector. By circularly shifting the matrix (13.10.1) $N/2$ columns to the left, one can symmetrize the wraparound; but this does not eliminate it. It is in fact possible to eliminate the wraparound completely by altering the coefficients in the first and last few rows of (13.10.1), giving an orthogonal matrix that is purely band-diagonal. This variant can be useful when, e.g., the data vary by many orders of magnitude from one end of the data vector to the other. We discuss it in §13.10.5, below.

Here is a DWT routine, `wt1`, that performs the pyramidal algorithm (or its inverse if `isign` is negative) on some data vector `a[0..n-1]`. Successive applications of the wavelet filter, and accompanying permutations, are performed by the object `wlet`, of class `Wavelet`, to be described below. The routine `wt1` also provides for the possibility of preconditioning and postconditioning steps, which we won't need until a later subsection.

wavelet.h

```
void wt1(VecDoub_IO &a, const Int isign, Wavelet &wlet)
One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm,
replacing a[0..n-1] by its wavelet transform (for isign=1), or performing the inverse operation
(for isign=-1). Note that n MUST be an integer power of 2. The object wlet, of type Wavelet,
is the underlying wavelet filter. Examples of Wavelet types are Daub4, Daubs, and Daub4i.
{
    Int nn, n=a.size();
    if (n < 4) return;
    if (isign >= 0) {                                   Wavelet transform.
        wlet.condition(a,n,1);
        for (nn=n;nn>=4;nn>>=1) wlet.filt(a,nn,isign);
        Start at largest hierarchy, and work toward smallest.
    } else {
        for (nn=4;nn<=n;nn<<=1) wlet.filt(a,nn,isign);
        Start at smallest hierarchy, and work toward largest.
        wlet.condition(a,n,-1);
    }
}
```

The `Wavelet` class is an "abstract base class," meaning that it is really only a promise that specific wavelets that derive from it will contain a method called `filt`, the actual wavelet filter. `Wavelet` also provides a default, null, pre- and postconditioning method. The class `Daub4` is derived from `Wavelet` and is intended for use with `wt1`. Its `filt` method implements the matrices (13.10.1) and (13.10.2), along with the permutation shown in (13.10.7).

wavelet.h

```
struct Wavelet {
    virtual void filt(VecDoub_IO &a, const Int n, const Int isign) = 0;
    virtual void condition(VecDoub_IO &a, const Int n, const Int isign) {}
};

struct Daub4 : Wavelet {
    void filt(VecDoub_IO &a, const Int n, const Int isign) {
    Applies the Daubechies 4-coefficient wavelet filter to data vector a[0..n-1] (for isign=1)
    or applies its transpose (for isign=-1). Used hierarchically by routines wt1 and wtn.
        const Doub C0=0.4829629131445341, C1=0.8365163037378077,
        C2=0.2241438680420134, C3=-0.1294095225512603;
        Int nh,i,j;
        if (n < 4) return;
        VecDoub wksp(n);
        nh = n >> 1;
        if (isign >= 0) {                                Apply filter.
            for (i=0,j=0;j<n-3;j+=2,i++) {
                wksp[i] = C0*a[j]+C1*a[j+1]+C2*a[j+2]+C3*a[j+3];
                wksp[i+nh] = C3*a[j]-C2*a[j+1]+C1*a[j+2]-C0*a[j+3];
            }
            wksp[i] = C0*a[n-2]+C1*a[n-1]+C2*a[0]+C3*a[1];
            wksp[i+nh] = C3*a[n-2]-C2*a[n-1]+C1*a[0]-C0*a[1];
        } else {                                         Apply transpose filter.
            wksp[0] = C2*a[nh-1]+C1*a[n-1]+C0*a[0]+C3*a[nh];
            wksp[1] = C3*a[nh-1]-C0*a[n-1]+C1*a[0]-C2*a[nh];
            for (i=0,j=2;i<nh-1;i++) {
                wksp[j++] = C2*a[i]+C1*a[i+nh]+C0*a[i+1]+C3*a[i+nh+1];
                wksp[j++] = C3*a[i]-C0*a[i+nh]+C1*a[i+1]-C2*a[i+nh+1];
            }
        }
        for (i=0;i<n;i++) a[i]=wksp[i];
    }
};
```

For larger sets of wavelet coefficients, the wraparound of the last rows or columns is a programming inconvenience. An efficient implementation would handle the wraparounds as special cases, outside of the main loop. For now, we will content ourselves with a more general scheme involving some extra arithmetic at run-time.

The following class, Daubs, takes an integer argument n in its constructor and creates a wavelet object with the filter DAUBn. Slightly better than "Hobson's choice," you can choose $n = 4$, 12, or 20. For other values of n you will need to add additional coefficient tables (e.g., from [6]).

wavelet.h

```
struct Daubs : Wavelet {
Structure for initializing and using the DAUBn wavelet filter for any n whose coefficients are
provided (here n = 4, 12, 20).
    Int ncof,ioff,joff;
    VecDoub cc,cr;
    static Doub c4[4],c12[12],c20[20];
    Daubs(Int n) : ncof(n), cc(n), cr(n) {
        Int i;
        ioff = joff = -(n >> 1);
        // ioff = -2; joff = -n + 2;        Alternative centering. (Used by Daub4, above.)
        if (n == 4) for (i=0; i<n; i++) cc[i] = c4[i];
        else if (n == 12) for (i=0; i<n; i++) cc[i] = c12[i];
        else if (n == 20) for (i=0; i<n; i++) cc[i] = c20[i];
        else throw("n not yet implemented in Daubs");
        Doub sig = -1.0;
        for (i=0; i<n; i++) {
            cr[n-1-i]=sig*cc[i];
            sig = -sig;
        }
    }
    void filt(VecDoub_IO &a, const Int n, const Int isign); See below.
};

Doub Daubs::c4[4]=
    {0.4829629131445341,0.8365163037378079,
    0.2241438680420134,-0.1294095225512604};
Doub Daubs::c12[12]=
    {0.111540743350, 0.494623890398, 0.751133908021,
    0.315250351709,-0.226264693965,-0.129766867567,
    0.097501605587, 0.027522865530,-0.031582039318,
    0.000553842201, 0.004777257511,-0.001077301085};
Doub Daubs::c20[20]=
    {0.026670057901, 0.188176800078, 0.527201188932,
    0.688459039454, 0.281172343661,-0.249846424327,
    -0.195946274377, 0.127369340336, 0.093057364604,
    -0.071394147166,-0.029457536822, 0.033212674059,
    0.003606553567,-0.010733175483, 0.001395351747,
    0.001992405295,-0.000685856695,-0.000116466855,
    0.000093588670,-0.000013264203};
```

There is some arbitrariness in how the wavelets at each hierarchical stage are centered over the data they act on. Daubs implements one popular choice, with another shown in commented code. Consult the literature if this matters to you (it rarely does).

The implementation of Daubs::filt() is straightforward:

wavelet.h

```
void Daubs::filt(VecDoub_IO &a, const Int n, const Int isign) {
Applies the previously initialized Daubn wavelet filter to data vector a[0..n-1] (for isign = 1)
or applies its transpose (for isign = −1). Used hierarchically by routines wt1 and wtn.
    Doub ai,ai1;
    Int i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;
    if (n < 4) return;
    VecDoub wksp(n);
    nmod = ncof*n;                      A positive constant equal to zero mod n.
    n1 = n-1;                           Mask of all bits, since n a power of 2.
    nh = n >> 1;
    for (j=0;j<n;j++) wksp[j]=0.0;
    if (isign >= 0) {                   Apply filter.
        for (ii=0,i=0;i<n;i+=2,ii++) {
            ni = i+1+nmod+ioff;         Pointer to be incremented and wrapped around.
            nj = i+1+nmod+joff;
            for (k=0;k<ncof;k++) {
                jf = n1 & (ni+k+1);     We use "bitwise and" to wrap around the
                jr = n1 & (nj+k+1);         pointers.
                wksp[ii] += cc[k]*a[jf];
                wksp[ii+nh] += cr[k]*a[jr];
            }
        }
    } else {                            Apply transpose filter.
        for (ii=0,i=0;i<n;i+=2,ii++) {
            ai = a[ii];
            ai1 = a[ii+nh];
            ni = i+1+nmod+ioff;         See comments above.
            nj = i+1+nmod+joff;
            for (k=0;k<ncof;k++) {
                jf = n1 & (ni+k+1);
                jr = n1 & (nj+k+1);
                wksp[jf] += cc[k]*ai;
                wksp[jr] += cr[k]*ai1;
            }
```

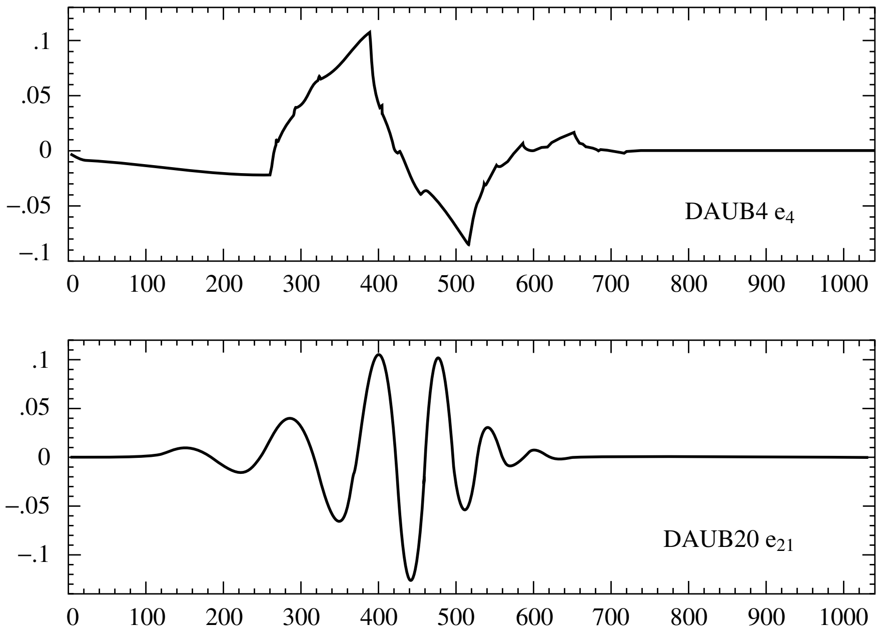


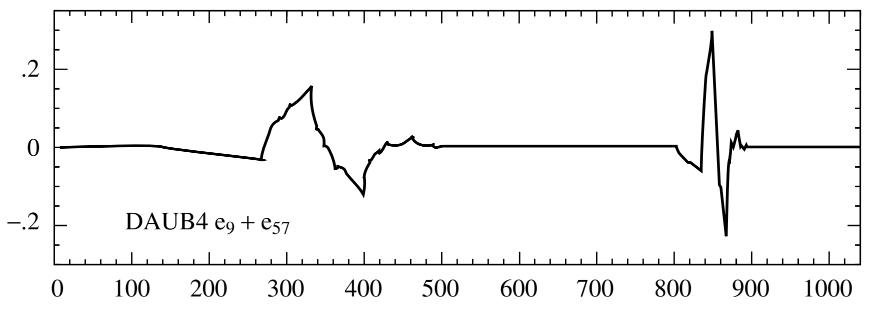
Figure 13.10.1. Wavelet functions, that is, single basis functions from the wavelet families DAUB4 and DAUB20. A complete, orthonormal wavelet basis consists of scalings and translations of either one of these functions. DAUB4 has an infinite number of cusps; DAUB20 would show similar behavior in a higher derivative.

```
        }
    }
    for (j=0;j<n;j++) a[j] = wksp[j];        Copy the results back from workspace.
}
```

13.10.3 What Do Wavelets Look Like?

We are now in a position to actually see some wavelets. To do so, we simply run unit vectors through any of the above discrete wavelet transforms, with `isign` negative so that the inverse transform is performed. Figure 13.10.1 shows the DAUB4 wavelet that is the inverse DWT of a unit vector in component 4 of a vector of length 1024, and also the DAUB20 wavelet that is the inverse of component 21. (One needs to go to a later hierarchical level for DAUB20 to avoid a wavelet with a wrapped-around tail.) Other unit vectors would give wavelets with the same shapes but different positions and scales.

One sees that both DAUB4 and DAUB20 have wavelets that are continuous. DAUB20 wavelets also have higher continuous derivatives. DAUB4 has the peculiar property that its derivative exists only *almost* everywhere. Examples of where it fails to exist are the points $p/2^n$, where p and n are integers; at such points, DAUB4 is left differentiable, but not right differentiable! This kind of discontinuity — at least in some derivative — is a necessary feature of wavelets with compact support, like the Daubechies series. For every increase in the number of wavelet coefficients by two, the Daubechies wavelets gain about *half* a derivative of continuity. (But not exactly half; the actual orders of regularity are irrational numbers!)



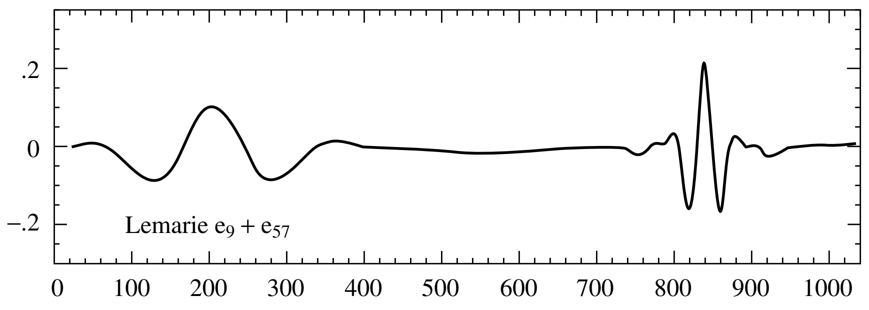


Figure 13.10.2. More wavelets, here generated from the sum of two unit vectors, $\mathbf{e}_9 + \mathbf{e}_{57}$, which are in different hierarchical levels of scale, and also at different spatial positions. DAUB4 wavelets (top) are defined by a filter in coordinate space (equation 13.10.5), while Lemarie wavelets (bottom) are defined by a filter most easily written in Fourier space (equation 13.10.14).

Note that the fact that wavelets are not smooth does not prevent their having exact representations for some smooth functions, as demanded by their approximation order p. The continuity of a wavelet is not the same as the continuity of functions that a set of wavelets can represent. For example, DAUB4 can represent (piecewise) linear functions of arbitrary slope: In the correct linear combinations, the cusps all cancel out. Every increase of two in the number of coefficients allows one higher order of polynomial to be exactly represented.

Figure 13.10.2 shows the result of performing the inverse DWT on the input vector $\mathbf{e}_9 + \mathbf{e}_{57}$, again for the two different particular wavelets. Since 9 lies early in the hierarchical range of 8–15, that wavelet lies on the left side of the picture. Since 57 lies in a later (smaller-scale) hierarchy, it is a narrower wavelet; in the range of 32–63 it is toward the end, so it lies on the right side of the picture. Note that smaller-scale wavelets are taller, so as to have the same squared integral.

13.10.4 Wavelet Filters in the Fourier Domain

The Fourier transform of a set of filter coefficients c_j is given by

$$H(\omega) = \sum_j c_j e^{ij\omega} \qquad (13.10.8)$$

Here H is a function periodic in 2π, and it has the same meaning as before: It is the wavelet filter, now written in the Fourier domain. A very useful fact is that the orthogonality conditions for the c's (e.g., equation 13.10.3 above) collapse to two simple relations in the Fourier domain,

$$\tfrac{1}{2}|H(0)|^2 = 1 \tag{13.10.9}$$

and

$$\tfrac{1}{2}\left[|H(\omega)|^2 + |H(\omega+\pi)|^2\right] = 1 \tag{13.10.10}$$

Likewise, the approximation condition of order p (e.g., equation 13.10.4 above) has a simple formulation, requiring that $H(\omega)$ have a pth order zero at $\omega = \pi$, or (equivalently)

$$H^{(m)}(\pi) = 0 \qquad m = 0, 1, \ldots, p-1 \tag{13.10.11}$$

It is thus relatively straightforward to invent wavelet sets in the Fourier domain. You simply invent a function $H(\omega)$ satisfying equations (13.10.9) – (13.10.11). To find the actual c_j's applicable to a data (or s-component) vector of length N, and with periodic wraparound as in matrices (13.10.1) and (13.10.2), you invert equation (13.10.8) by the discrete Fourier transform

$$c_j = \frac{1}{N}\sum_{k=0}^{N-1} H(2\pi\frac{k}{N})e^{-2\pi ijk/N} \tag{13.10.12}$$

The quadrature mirror filter G (reversed c_j's with alternating signs), incidentally, has the Fourier representation

$$G(\omega) = e^{-i\omega}H^*(\omega+\pi) \tag{13.10.13}$$

where the asterisk denotes complex conjugation.

In general, the above procedure will *not* produce wavelet filters with compact support. In other words, all N of the c_j's, $j = 0, \ldots, N-1$ will in general be nonzero (though they may be rapidly decreasing in magnitude). The Daubechies wavelets, or other wavelets with compact support, are specially chosen so that $H(\omega)$ is a trigonometric polynomial with only a small number of Fourier components, guaranteeing that there will be only a small number of nonzero c_j's.

On the other hand, there is sometimes no particular reason to demand compact support. Giving it up in fact allows the ready construction of relatively smoother wavelets (higher values of p). Even without compact support, the convolutions implicit in the matrix (13.10.1) can be done efficiently by FFT methods.

Lemarie's wavelet (see [4]) has $p = 4$, does not have compact support, and is defined by the choice of $H(\omega)$,

$$H(\omega) = \left[2(1-u)^4\frac{315 - 420u + 126u^2 - 4u^3}{315 - 420v + 126v^2 - 4v^3}\right]^{1/2} \tag{13.10.14}$$

where

$$u \equiv \sin^2\frac{\omega}{2} \qquad v \equiv \sin^2\omega \tag{13.10.15}$$

It is beyond our scope to explain where equation (13.10.14) comes from. An informal description is that the quadrature mirror filter $G(\omega)$ deriving from equation (13.10.14) has the property that it gives identically zero when applied to any function whose odd-numbered samples are equal to the cubic spline interpolation of its even-numbered samples. Since this class of functions includes many very smooth members, it follows that $H(\omega)$ does a good job of truly selecting a function's smooth information content. Sample Lemarie wavelets are shown in Figure 13.10.2.

13.10.5 Daubechies Wavelets on the Interval

The discrete wavelet transforms that we have seen thus far are periodic and thus "live on a circle.". Wavelets close to one edge of the data vector have tails that wrap around to the other edge. Said differently, some components of a discrete wavelet transform depend on data values at both ends of the data vector.

Most of the time, this periodicity is merely something between a curiosity and a minor nuisance, exactly like the discrete Fourier transform's similar periodicity. Similar simple workarounds (e.g., zero-padding of the data) apply. Occasionally, however, the wraparound can produce undesirable effects, for example when the data differ by orders of magnitude at the two ends, or are smooth at one end but unsmooth at the other.

By modifying the coefficients of the wavelet filters near the two ends of the data vector, it is possible to produce wavelets that utilize only local data at each edge, that is, wavelets that "live on the interval" instead of on the circle. For such wavelets, the orthogonal matrix analogous to (13.10.1) is purely band-diagonal, and is identical to (13.10.1) except for modifications in the first and last few rows. Various such constructions have been proposed. Our favorite is that of [7].

One wrinkle needs to be mentioned: We would hope that those modified rows of the new matrix that are "detail filters" have the property of giving exactly zero when applied to smooth polynomial sequences like $1, 1, 1, 1, 1$ or $1, 2, 3, 4, 5$. Indeed, all the period wavelets previously discussed have this property. Alas, this condition, plus orthogonality, imposes too many constraints on the coefficients, and is unachievable. It turns out, however, that a simple linear preconditioning of the first and last few data points (that is, replacing the values by linear combinations of themselves) restores the desired property. The preconditioning is done only once in the transform, *not* at every pyramidal level. This need for preconditioning (with a corresponding postconditioning for the inverse) is the reason that our `Wavelet` abstract class has a method named `condition`. Finally we get to use it in a nontrivial way!

Here is an implementation of DAUB4 wavelets on the interval as a class derived from `Wavelet`, compatible for use in `wt1`. The ugliness of the code reflects only the large number of new coefficients that must be provided. If you want to implement higher DAUBn's on the interval, you'll need even more coefficients, as found in [6] or [5].

```
struct Daub4i : Wavelet {                                                wavelet.h
    void filt(VecDoub_IO &a, const Int n, const Int isign) {
```
Applies the Cohen-Daubechies-Vial 4-coefficient wavelet on the interval filter to data vector `a[0..n-1]` (for `isign=1`) or applies its transpose (for `isign=-1`). Used hierarchically by routines `wt1` and `wtn`.
```
        const Doub C0=0.4829629131445341, C1=0.8365163037378077,
            C2=0.2241438680420134, C3=-0.1294095225512603;
        const Doub R00=0.603332511928053,R01=0.690895531839104,
            R02=-0.398312997698228,R10=-0.796543516912183,R11=0.546392713959015,
            R12=-0.258792248333818,R20=0.0375174604524466,R21=0.457327659851769,
            R22=0.850088102549165,R23=0.223820356983114,R24=-0.129222743354319,
            R30=0.0100372245644139,R31=0.122351043116799,R32=0.227428111655837,
            R33=-0.836602921223654,R34=0.483012921773304,R43=0.443149049637559,
            R44=0.767556669298114,R45=0.374955331645687,R46=0.190151418429955,
            R47=-0.194233407427412,R53=0.231557595006790,R54=0.401069519430217,
            R55=-0.717579999353722,R56=-0.363906959570891,R57=0.371718966535296,
            R65=0.230389043796969,R66=0.434896997965703,R67=0.870508753349866,
            R75=-0.539822500731772,R76=0.801422961990337,R77=-0.257512919478482;
        Int nh,i,j;
        if (n < 8) return;
        VecDoub wksp(n);
        nh = n >> 1;
        if (isign >= 0) {
            wksp[0]  = R00*a[0]+R01*a[1]+R02*a[2];
            wksp[nh] = R10*a[0]+R11*a[1]+R12*a[2];
            wksp[1] = R20*a[0]+R21*a[1]+R22*a[2]+R23*a[3]+R24*a[4];
            wksp[nh+1] = R30*a[0]+R31*a[1]+R32*a[2]+R33*a[3]+R34*a[4];
            for (i=2,j=3;j<n-4;j+=2,i++) {
                wksp[i] = C0*a[j]+C1*a[j+1]+C2*a[j+2]+C3*a[j+3];
                wksp[i+nh] = C3*a[j]-C2*a[j+1]+C1*a[j+2]-C0*a[j+3];
            }
            wksp[nh-2] = R43*a[n-5]+R44*a[n-4]+R45*a[n-3]+R46*a[n-2]+R47*a[n-1];
            wksp[n-2] = R53*a[n-5]+R54*a[n-4]+R55*a[n-3]+R56*a[n-2]+R57*a[n-1];
            wksp[nh-1] = R65*a[n-3]+R66*a[n-2]+R67*a[n-1];
            wksp[n-1] = R75*a[n-3]+R76*a[n-2]+R77*a[n-1];
        } else {
            wksp[0] = R00*a[0]+R10*a[nh]+R20*a[1]+R30*a[nh+1];
            wksp[1] = R01*a[0]+R11*a[nh]+R21*a[1]+R31*a[nh+1];
            wksp[2] = R02*a[0]+R12*a[nh]+R22*a[1]+R32*a[nh+1];
            if (n == 8) {
                wksp[3] = R23*a[1]+R33*a[5]+R43*a[2]+R53*a[6];
                wksp[4] = R24*a[1]+R34*a[5]+R44*a[2]+R54*a[6];
            } else {
                wksp[3] = R23*a[1]+R33*a[nh+1]+C0*a[2]+C3*a[nh+2];
                wksp[4] = R24*a[1]+R34*a[nh+1]+C1*a[2]-C2*a[nh+2];
                wksp[n-5] = C2*a[nh-3]+C1*a[n-3]+R43*a[nh-2]+R53*a[n-2];
                wksp[n-4] = C3*a[nh-3]-C0*a[n-3]+R44*a[nh-2]+R54*a[n-2];
            }
            for (i=2,j=5;i<nh-3;i++) {
                wksp[j++] = C2*a[i]+C1*a[i+nh]+C0*a[i+1]+C3*a[i+nh+1];
                wksp[j++] = C3*a[i]-C0*a[i+nh]+C1*a[i+1]-C2*a[i+nh+1];
            }
            wksp[n-3] = R45*a[nh-2]+R55*a[n-2]+R65*a[nh-1]+R75*a[n-1];
            wksp[n-2] = R46*a[nh-2]+R56*a[n-2]+R66*a[nh-1]+R76*a[n-1];
            wksp[n-1] = R47*a[nh-2]+R57*a[n-2]+R67*a[nh-1]+R77*a[n-1];
        }
        for (i=0;i<n;i++) a[i]=wksp[i];
    }
    void condition(VecDoub_IO &a, const Int n, const Int isign) {
        Doub t0,t1,t2,t3;
        if (n < 4) return;
        if (isign >= 0) {
            t0 = 0.324894048898962*a[0]+0.0371580151158803*a[1];
            t1 = 1.00144540498130*a[1];
            t2 = 1.08984305289504*a[n-2];
            t3 = -0.800813234246437*a[n-2]+2.09629288435324*a[n-1];
            a[0]=t0; a[1]=t1; a[n-2]=t2; a[n-1]=t3;
        } else {
            t0 = 3.07792649138669*a[0]-0.114204567242137*a[1];
            t1 = 0.998556681198888*a[1];
            t2 = 0.917563310922261*a[n-2];
            t3 = 0.350522032550918*a[n-2]+0.477032578540915*a[n-1];
            a[0]=t0; a[1]=t1; a[n-2]=t2; a[n-1]=t3;
        }
    }
};
```

Do you really need wavelets on the interval, instead of ordinary, periodic wavelets? Occasionally, yes. If you look ahead to Figure 13.10.6, which is a graphical display of two-dimensional wavelet coefficients, you can see the difference between

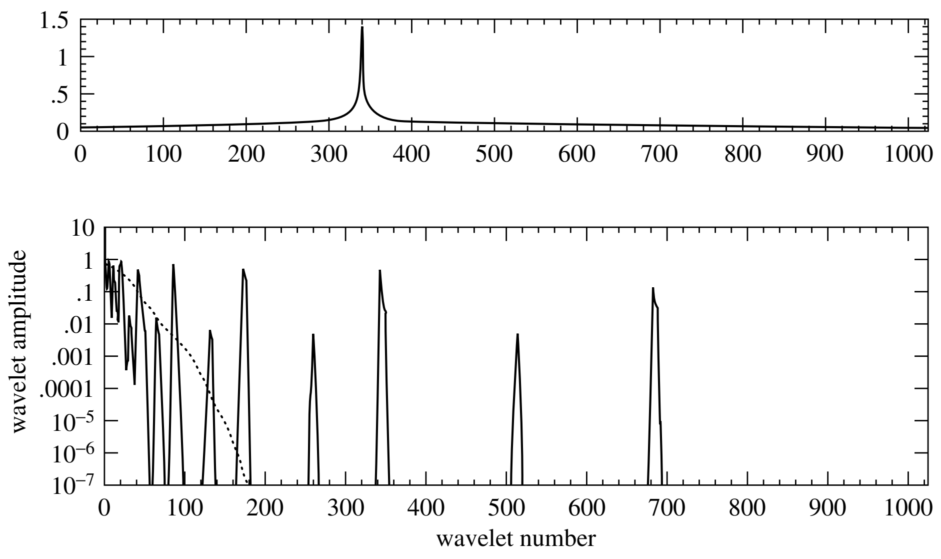


Figure 13.10.3. Top: Arbitrary test function, with cusp, sampled on a vector of length 1024. Bottom: Absolute value of the 1024 wavelet coefficients produced by the discrete wavelet transform of the function. Note log scale. The dotted curve plots the same amplitudes when sorted by decreasing size. One sees that only 130 out of 1024 coefficients are larger than 10^{-4} (or larger than about 10^{-5} times the largest coefficient, whose value is ~ 10).

allowing and suppressing wraparound.

13.10.6 Truncated Wavelet Approximations

Most of the usefulness of wavelets rests on the fact that wavelet transforms can usefully be severely truncated, that is, turned into sparse expansions. The case of Fourier transforms is different: FFTs are ordinarily used without truncation, to compute fast convolutions, for example. This works because the convolution operator is particularly simple in the Fourier basis. There are not, however, any standard mathematical operations that are especially simple in the wavelet basis.

To see how truncation works, consider the simple example shown in Figure 13.10.3. The upper panel shows an arbitrarily chosen test function, smooth except for a square-root cusp, sampled onto a vector of length 2^{10}. The bottom panel (solid curve) shows, on a log scale, the absolute value of the vector's components after it has been run through the DAUB4 discrete wavelet transform. One notes, from right to left, the different levels of hierarchy, 512–1023, 256–511, 128–255, etc. Within each level, the wavelet coefficients are nonnegligible only very near the location of the cusp, or very near the left and right boundaries of the hierarchical range (edge effects).

The dotted curve in the lower panel of Figure 13.10.3 plots the same amplitudes as the solid curve, but sorted into decreasing order of size. One can read off, for example, that the 130th largest wavelet coefficient has an amplitude less than 10^{-5} of the largest coefficient, whose magnitude is ~ 10 (power or square integral ratio less than 10^{-10}). Thus, the example function can be represented quite accurately by only 130, rather than 1024, coefficients — the remaining ones being set to zero.

Note that this kind of truncation makes the vector sparse, but still of logical length 1024. It is very important that vectors in wavelet space be truncated according to the *amplitude* of the components, not their position in the vector. Keeping the first 256 components of the vector (all levels of the hierarchy except the last two) would give an extremely poor, and jagged, approximation to the function. When you compress a function with wavelets, you have to record both the values *and the positions* of the nonzero coefficients.

Generally, compact (and therefore unsmooth) wavelets are better for lower accuracy approximations and for functions with discontinuities (like edges). Smooth (and therefore noncompact) wavelets are better for achieving high numerical accuracy. This makes compact wavelets a good choice for image compression, for example, while it makes smooth wavelets best for fast solution of integral equations.

In real applications of wavelets to compression, components are not starkly "kept" or "discarded." Rather, components may be kept with a varying number of bits of accuracy, depending on their magnitude. The JPEG-2000 image compression standard utilizes wavelets in such a manner.

13.10.7 Wavelet Transform in Multidimensions

A wavelet transform of a d-dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indices), then on its second, and so on. Each transformation corresponds to multiplication by an orthogonal matrix $\mathbf{M}$. Because (illustrating the case $d = 2$)

$$\sum_j M_{nj}\left(\sum_i M_{mi}a_{ij}\right) = \sum_i M_{mi}\left(\sum_j M_{nj}a_{ij}\right) \qquad (13.10.16)$$

the result is independent of the order in which the indices were transformed. The situation is exactly like that for multidimensional FFTs. A routine for effecting the multidimensional DWT can thus be modeled on a multidimensional FFT routine like `fourn`:

wavelet.h

```
void wtn(VecDoub_IO &a, VecInt_I &nn, const Int isign, Wavelet &wlet)
```
Replaces `a` by its `ndim`-dimensional discrete wavelet transform, if `isign` is input as 1. Here `nn[0..ndim-1]` is an integer array containing the lengths of each dimension (number of real values), which must all be powers of 2. `a` is a real array of length equal to the product of these lengths, in which the data are stored as in a multidimensional real array. If `isign` is input as −1, `a` is replaced by its inverse wavelet transform. The object `wlet`, of type `Wavelet`, is the underlying wavelet filter. Examples of `Wavelet` types are `Daub4`, `Daubs`, and `Daub4i`.
```
{
    Int idim,i1,i2,i3,k,n,nnew,nprev=1,nt,ntot=1;
    Int ndim=nn.size();
    for (idim=0;idim<ndim;idim++) ntot *= nn[idim];
    if (ntot&(ntot-1)) throw("all lengths must be powers of 2 in wtn");
    for (idim=0;idim<ndim;idim++) {          Main loop over the dimensions.
        n=nn[idim];
        VecDoub wksp(n);
        nnew=n*nprev;
        if (n > 4) {
            for (i2=0;i2<ntot;i2+=nnew) {
                for (i1=0;i1<nprev;i1++) {
                    for (i3=i1+i2,k=0;k<n;k++,i3+=nprev) wksp[k]=a[i3];
                    Copy the relevant row or column or etc. into workspace.
                    if (isign >= 0) {            Do one-dimensional wavelet transform.
```

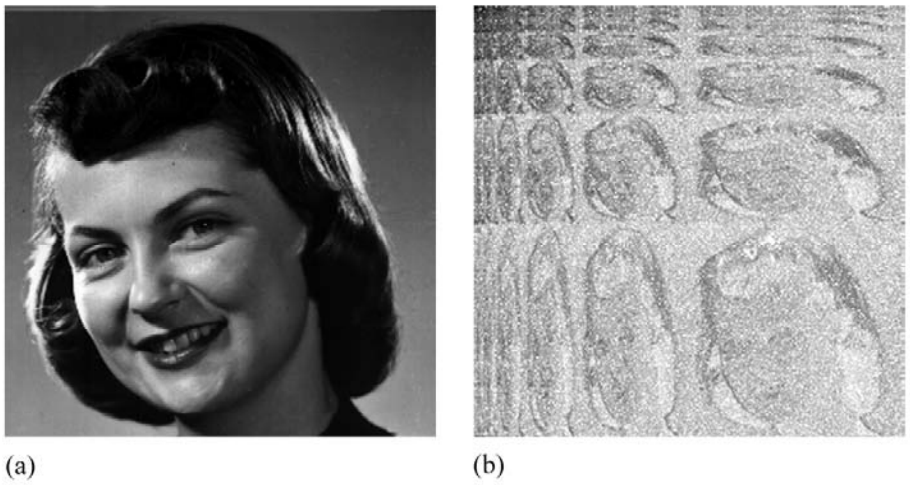


Figure 13.10.4. (a) Two-dimensional array of intensities (i.e., a photograph) and (b) its two-dimensional discrete wavelet transform. Darker pixels represent wavelet components that are larger in magnitude, on a logarithmic scale. Wavelets number from the upper-left corner, where the "smooth" information content is encoded.

```
                        wlet.condition(wksp,n,1);
                        for(nt=n;nt>=4;nt >>= 1) wlet.filt(wksp,nt,isign);
                    } else {                    Or inverse transform.
                        for(nt=4;nt<=n;nt <<= 1) wlet.filt(wksp,nt,isign);
                        wlet.condition(wksp,n,-1);
                    }
                    for (i3=i1+i2,k=0;k<n;k++,i3+=nprev) a[i3]=wksp[k];
                    Copy back from workspace.
                }
            }
        }
        nprev=nnew;
    }
}
```

Here, as before, `wlet` is a `Wavelet` object that embodies a particular wavelet filter and (if required) pre-conditioner.

Figure 13.10.4 shows a sample image and its wavelet transform, represented graphically.

13.10.8 Compression of Images

An immediate application of the multidimensional transform `wtn` is to image compression. The overall procedure is to take the wavelet transform of a digitized image, and then to "allocate bits" among the wavelet coefficients in some highly nonuniform, optimized, manner. As already mentioned, large wavelet coefficients get quantized accurately, while small coefficients are quantized coarsely with only a bit or two — or else are truncated completely. If the resulting quantization levels are still statistically nonuniform, they may then be further compressed by a technique like Huffman coding (§22.5).

While a more detailed description of the "back end" of this process, namely the quantization and coding of the image, is beyond our scope, it is quite straightforward

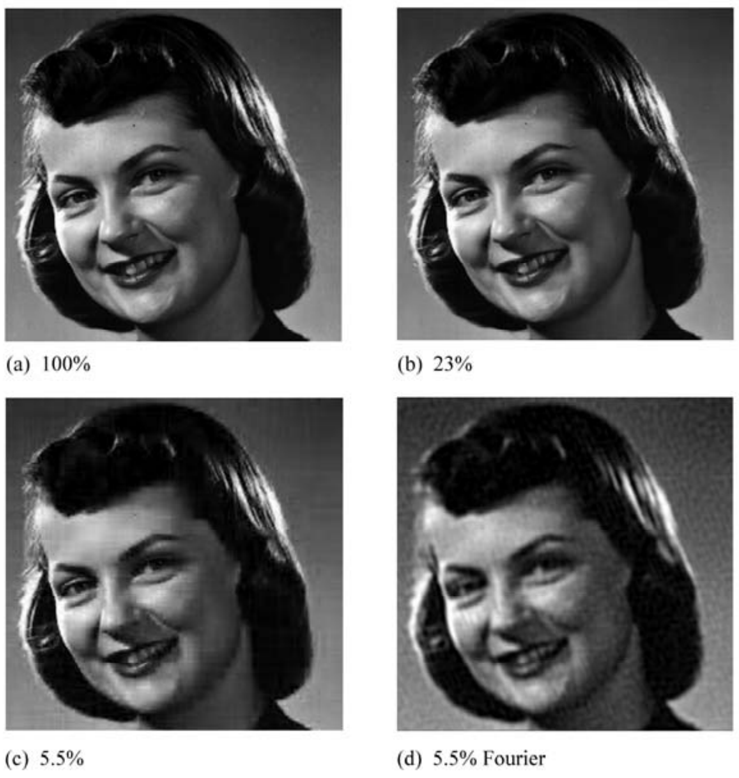


Figure 13.10.5. (a) IEEE test image, 256×256 pixels with 8-bit grayscale. (b) The image is transformed into the wavelet basis; 77% of its wavelet components are set to zero (those of smallest magnitude); it is then reconstructed from the remaining 23%. (c) Same as (b), but 94.5% of the wavelet components are deleted. (d) Same as (c), but the Fourier transform is used instead of the wavelet transform. Wavelet coefficients are better than the Fourier coefficients at preserving relevant details.

to demonstrate the "front-end" wavelet encoding with a simple truncation: We keep (with full accuracy) all wavelet coefficients larger than some threshold, and we delete (set to zero) all smaller wavelet coefficients. We can then adjust the threshold to vary the fraction of preserved coefficients.

Figure 13.10.5 shows a sequence of images that differ in the number of wavelet coefficients that have been kept. The original picture (a), which is an official IEEE test image, has 256 by 256 pixels with an 8-bit grayscale. The two reproductions following are reconstructed with 23% (b) and 5.5% (c) of the 65536 wavelet coefficients. The latter image illustrates the kind of compromises made by the truncated wavelet representation. High-contrast edges (the model's right cheek and hair highlights, e.g.) are maintained at a relatively high resolution, while low-contrast areas (the model's left eye and cheek, e.g.) are washed out into what amounts to large constant pixels. Figure 13.10.5(d) is the result of performing the identical procedure with Fourier, instead of wavelet, transforms: The figure is reconstructed from the 5.5% of 65536 real Fourier components having the largest magnitudes. One sees that, since sines and cosines are nonlocal, the resolution is uniformly poor across the picture; also, the deletion of any components produces a mottled "ringing" everywhere. (Practical Fourier image compression schemes therefore break up an image into small blocks of pixels, 16×16, say, and do rather elaborate smoothing across block boundaries when the image is reconstructed.)

Viewers will sometimes choose (b) over (a), in Figure 13.10.5, as the superior image. The reason is that a "little bit" of wavelet compression has the effect of *denoising* the image. See [8] for a rigorous development.

13.10.9 Fast Solution of Linear Systems

There are interesting applications of wavelets to linear algebra. The basic idea [1] is to think of an integral operator (that is, a large matrix) as a digital image. Suppose that the operator compresses well under a two-dimensional wavelet transform, i.e., that a large fraction of its wavelet coefficients are so small as to be negligible. Then any linear system involving the operator becomes a sparse system in the wavelet basis. In other words, to solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{13.10.17}$$

we first wavelet-transform the operator $\mathbf{A}$ and the right-hand side $\mathbf{b}$ by

$$\widetilde{\mathbf{A}} \equiv \mathbf{W} \cdot \mathbf{A} \cdot \mathbf{W}^T, \qquad \widetilde{\mathbf{b}} \equiv \mathbf{W} \cdot \mathbf{b} \tag{13.10.18}$$

where $\mathbf{W}$ represents the one-dimensional wavelet transform, then solve

$$\widetilde{\mathbf{A}} \cdot \widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} \tag{13.10.19}$$

and finally transform to the answer by the inverse wavelet transform

$$\mathbf{x} = \mathbf{W}^T \cdot \widetilde{\mathbf{x}} \tag{13.10.20}$$

(Note that the routine `wtn` does the complete transformation of $\mathbf{A}$ into $\widetilde{\mathbf{A}}$.)

A typical integral operator that compresses well into wavelets has arbitrary (or even nearly singular) elements near its main diagonal, but becomes smooth away from the diagonal. An example might be

$$A_{ij} = \begin{cases} -1 & \text{if } i = j \\ |i-j|^{-1/2} & \text{otherwise} \end{cases} \tag{13.10.21}$$

Figure 13.10.6 shows a graphical representation of the wavelet transform of this matrix, where i and j range over $0 \ldots 255$, using the DAUB4 wavelet, both in its conventional, periodic, implementation, and its modified form on the interval. Elements larger in magnitude than 10^{-3} times the maximum element are shown as black pixels, while elements between 10^{-3} and 10^{-6} are shown in gray. White pixels are $< 10^{-6}$. The indices i and j each number from the lower left.

In the figure, one sees the hierarchical decomposition into power-of-two sized blocks. At the edges or corners of the various blocks, one sees edge effects caused by the wraparound wavelet boundary conditions. Apart from edge effects, within each block, the nonnegligible elements are concentrated along the block diagonals. This is a statement that, for this type of linear operator, a wavelet is coupled mainly to near neighbors in its own hierarchy (square blocks along the main diagonal) and near neighbors in other hierarchies (rectangular blocks off the diagonal).

The number of nonnegligible elements in a matrix like that in Figure 13.10.6 scales only as N, the linear size of the matrix; as a rough rule of thumb it is about

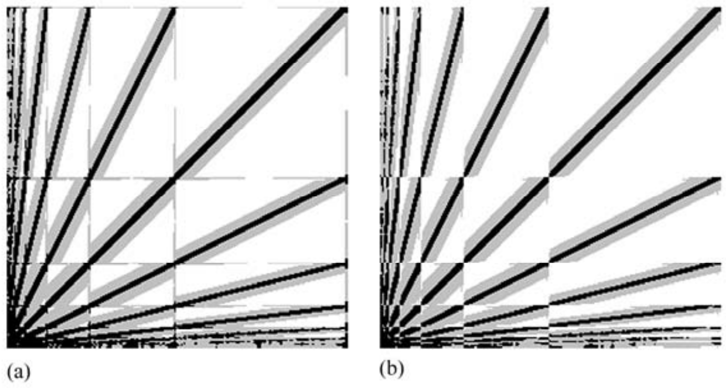


Figure 13.10.6. Wavelet transform of a 256×256 matrix, represented graphically. The original matrix has a discontinuous cusp along its diagonal, decaying smoothly away on both sides of the diagonal. In wavelet basis, the matrix becomes sparse: Components larger than 10^{-3} are shown as black, components larger than 10^{-6} as gray, and smaller-magnitude components are white. The matrix indices i and j number from the lower left. (a) Ordinary DAUB4 (periodic) is used. (b) Modified DAUB4 on the interval is used, eliminating wraparound artifacts and producing a more regular pattern of significant components.

$10N \log_{10}(1/\epsilon)$, where ϵ is the truncation level, e.g., 10^{-6}. For a 2000 by 2000 matrix, then, the matrix is sparse by a factor on the order of 30.

Various numerical schemes can be used to solve sparse linear systems of this "hierarchically band-diagonal" form. Beylkin, Coifman, and Rokhlin [1] make the interesting observations that (1) the product of two such matrices is itself hierarchically band-diagonal (truncating, of course, newly generated elements that are smaller than the predetermined threshold ϵ); and, moreover, that (2) the product can be formed in order N operations.

Fast matrix multiplication enables finding the matrix inverse by Schultz's (or Hotelling's) method; see §2.5.

Other schemes are also possible for fast solution of hierarchically band-diagonal forms. For example, one can use the conjugate gradient method, implemented in §2.7 as `linbcg`.

CITED REFERENCES AND FURTHER READING:

Daubechies, I. 1992, *Wavelets* (Philadelphia: S.I.A.M.).

Strang, G. 1989, "Wavelets and Dilation Equations: A Brief Introduction," *SIAM Review*, vol. 31, pp. 614–627.

Beylkin, G., Coifman, R., and Rokhlin, V. 1991, "Fast Wavelet Transforms and Numerical Algorithms," *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141–183.[1]

Daubechies, I. 1988, "Orthonormal Bases of Compactly Supported Wavelets," *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909–996.[2]

Vaidyanathan, P.P. 1990, "Multirate Digital Filters, Filter Banks, Polyphase Networks, and Applications," *Proceedings of the IEEE*, vol. 78, pp. 56–93.[3]

Mallat, S.G. 1989, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693.[4]

Cohen, A. 1993, "Tables for Wavelet Filters Adapted to Life on an Interval," multiple Web sites; mirrored at `http://www.nr.com/contrib`.[5]

Brewster, M.E. and Beylkin, G. 1994, tables from “Double Precision Wavelet Transform Library,” mirrored at `http://www.nr.com/contrib`.[6]

Cohen, A., Daubechies, I., and Vial, P. 1993, “Wavelets on the Interval and Fast Wavelet Transforms,” *Applied and Computational Harmonic Analysis*, vol. 1, pp. 54–81.[7]

Donoho, D. and Johnstone, I.M. 1994, “Ideal Spatial Adaptation via Wavelet Shrinkage,” *Biometrika*, vol. 81, no. 3, pp. 425–455.[8]

13.11 Numerical Use of the Sampling Theorem

We have met the sampling theorem before, in §4.5 (in relation to the accuracy of the trapezoidal rule for integration); in §6.9, where we implemented an approximating formula for Dawson’s integral due to Rybicki, and in §12.1, where we first saw it in a Fourier context. Now that we have become Fourier sophisticates, we can readily supply a derivation of the formula in §6.9 and illustrate the use of the sampling theorem as a purely numerical tool. Our discussion is identical to Rybicki [1].

For our present purposes, the sampling theorem is most conveniently stated as follows: Consider an arbitrary function $g(t)$ and the grid of sampling points $t_n = \alpha + nh$, where n ranges over the integers and α is a constant that allows an arbitrary shift of the sampling grid. We then write

$$g(t) = \sum_{n=-\infty}^{\infty} g(t_n)\,\text{sinc}\,\frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.1}$$

where $\text{sinc}\,x \equiv \sin x/x$. The summation over the sampling points is called the *sampling representation* of $g(t)$, and $e(t)$ is its error term. The sampling theorem asserts that the sampling representation is exact, that is, $e(t) \equiv 0$, if the Fourier transform of $g(t)$,

$$G(\omega) = \int_{-\infty}^{\infty} g(t)e^{i\omega t}\,dt \tag{13.11.2}$$

vanishes identically for $|\omega| \geq \pi/h$.

When can sampling representations be used to advantage for the approximate numerical computation of functions? In order that the error term be small, the Fourier transform $G(\omega)$ must be sufficiently small for $|\omega| \geq \pi/h$. On the other hand, in order for the summation in (13.11.1) to be approximated by a reasonably small number of terms, the function $g(t)$ itself should be very small outside of a fairly limited range of values of t. Thus we are led to two conditions to be satisfied in order that (13.11.1) be useful numerically: Both the function $g(t)$ and its Fourier transform $G(\omega)$ must rapidly approach zero for large values of their respective arguments.

Unfortunately, these two conditions are mutually antagonistic — the Uncertainty Principle in quantum mechanics. There exist strict limits on how rapidly the simultaneous approach to zero can be in both arguments. According to a theorem of Hardy [2], if $g(t) = O(e^{-t^2})$ as $|t| \to \infty$ and $G(\omega) = O(e^{-\omega^2/4})$ as $|\omega| \to \infty$, then $g(t) \equiv Ce^{-t^2}$, where C is a constant. This can be interpreted as saying that, of all functions, the Gaussian is the most rapidly decaying in both t and ω, and in this sense is the “best” function to be expressed numerically as a sampling representation.

Let us then write for the Gaussian $g(t) = e^{-t^2}$,

$$e^{-t^2} = \sum_{n=-\infty}^{\infty} e^{-t_n^2}\,\text{sinc}\,\frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.3}$$

The error $e(t)$ depends on the parameters h and α as well as on t, but it is sufficient for the present purposes to state the bound,

$$|e(t)| < e^{-(\pi/2h)^2} \tag{13.11.4}$$

which can be understood simply as the order of magnitude of the Fourier transform of the Gaussian at the point where it "spills over" into the region $|\omega| > \pi/h$.

When the summation in (13.11.3) is approximated by one with finite limits, say from $N_0 - N$ to $N_0 + N$, where N_0 is the integer nearest to $-\alpha/h$, there is a further truncation error. However, if N is chosen so that $N > \pi/(2h^2)$, the truncation error in the summation is less than the bound given by (13.11.4), and, since this bound is an overestimate, we shall continue to use it for (13.11.3) as well. The truncated summation gives a remarkably accurate representation for the Gaussian even for moderate values of N. For example, $|e(t)| < 5\times10^{-5}$ for $h = 1/2$ and $N = 7$; $|e(t)| < 2\times10^{-10}$ for $h = 1/3$ and $N = 15$; and $|e(t)| < 7\times10^{-18}$ for $h = 1/4$ and $N = 25$.

One may ask, what is the point of such a numerical representation for the Gaussian, which can be computed so easily and quickly as an exponential? The answer is that many transcendental functions can be expressed as an integral involving the Gaussian, and by substituting (13.11.3) one can often find excellent approximations to the integrals as a sum over elementary functions.

Let us consider as an example the function $w(z)$ of the complex variable $z = x + iy$, related to the complex error function by

$$w(z) = e^{-z^2}\,\mathrm{erfc}(-iz) \tag{13.11.5}$$

having the integral representation

$$w(z) = \frac{1}{\pi i}\int_C \frac{e^{-t^2}\,dt}{t - z} \tag{13.11.6}$$

where the contour C extends from $-\infty$ to ∞, passing below z (see, e.g., [3]). Many methods exist for the evaluation of this function (e.g., [4]). Substituting the sampling representation (13.11.3) into (13.11.6) and performing the resulting elementary contour integrals, we obtain

$$w(z) \approx \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-t_n^2}\,\frac{1 - (-1)^n e^{-\pi i(\alpha - z)/h}}{t_n - z} \tag{13.11.7}$$

where we now omit the error term. One should note that there is no singularity as $z \to t_m$ for some $n = m$, but a special treatment of the mth term will be required in this case (for example, by power series expansion).

An alternative form of equation (13.11.7) can be found by expressing the complex exponential in (13.11.7) in terms of trigonometric functions and using the sampling representation (13.11.3) with z replacing t. This yields

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-t_n^2}\,\frac{1 - (-1)^n \cos\pi(\alpha - z)/h}{t_n - z} \tag{13.11.8}$$

This form is particularly useful in obtaining Re $w(z)$ when $|y| \ll 1$. Note that in evaluating (13.11.7) the complex exponential inside the summation is a constant and needs to be evaluated only once; a similar comment holds for the cosine in (13.11.8).

There are a variety of formulas that can now be derived from either equation (13.11.7) or (13.11.8) by choosing particular values of α. Eight interesting choices are $\alpha = 0, x, iy$, or z, plus the values obtained by adding $h/2$ to each of these. Since the error bound (13.11.3) assumed a real value of α, the choices involving a complex α are useful only if the imaginary part of z is not too large. This is not the place to catalog all 16 possible formulas, and we give only two particular cases that show some of the important features.

First of all let $\alpha = 0$ in equation (13.11.8), which yields,

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-(nh)^2}\,\frac{1 - (-1)^n \cos(\pi z/h)}{nh - z} \tag{13.11.9}$$

This approximation is good over the entire z-plane. As stated previously, one has to treat the case where one denominator becomes small by expansion in a power series. Formulas for the case $\alpha = 0$ were discussed briefly in [5]. They are similar, but not identical, to formulas derived by Chiarella and Reichel [6], using the method of Goodwin [7].

Next, let $\alpha = z$ in (13.11.7), which yields

$$w(z) \approx e^{-z^2} - \frac{2}{\pi i} \sum_{n \text{ odd}} \frac{e^{-(z-nh)^2}}{n} \tag{13.11.10}$$

the sum being over all odd integers (positive and negative). Note that we have made the substitution $n \to -n$ in the summation. This formula is simpler than (13.11.9) and contains half the number of terms, but its error is worse if y is large. Equation (13.11.10) is the source of the approximation formula (6.9.3) for Dawson's integral, used in §6.9.

CITED REFERENCES AND FURTHER READING:

Rybicki, G.B. 1989, "Dawson's Integral and The Sampling Theorem," *Computers in Physics*, vol. 3, no. 2, pp. 85–87.[1]

Hardy, G.H. 1933, "A Theorem Concerning Fourier Transforms," *Journal of the London Mathematical Society*, vol. 8, pp. 227–231.[2]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`.[3]

Gautschi, W. 1970, "Efficient Computation of the Complex Error function," *SIAM Journal on Numerical Analysis*, vol. 7, pp. 187–198.[4]

Armstrong, B.H., and Nicholls, R.W. 1972, *Emission, Absorption and Transfer of Radiation in Heated Atmospheres* (New York: Pergamon).[5]

Chiarella, C., and Reichel, A. 1968, "On the Evaluation of Integrals Related to the Error Function," *Mathematics of Computation*, vol. 22, pp. 137–143.[6]

Goodwin, E.T. 1949, "The Evaluation of Integrals of the Form $\int_{-x}^{+x} f(x)e^{-x^2}dx$," *Proceedings of the Cambridge Philosophical Society*, vol. 45, pp. 241–245.[7]

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 given to the computer, not produced by it. These are numbers to be treated with considerable respect, neither to be tampered with, nor subjected to a computational process whose character you do not completely understand. You are well advised to acquire a reverence for data, 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 wonderful gray area that 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, usually called a *tail test* or *p-value test*:

- 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

[*]"Science advances one funeral at a time." —Max Planck (attributed)

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. Information-theoretic methods are discussed in §14.7. Section 14.9 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 and §6.14. You may wish, at this point, to review those sections.

Bayesian methods make little appearance in this chapter, but become more prominent in the two chapters following this one.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill).

Taylor, J.R. 1997, *An Introduction to Error Analysis*, 2nd ed. (Sausalito, CA: University Science Books).

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press).

Wall, J.V., and Jenkins, C.R. 2003, *Practical Statistics for Astronomers* (Cambridge, UK: Cambridge University Press).

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press).

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 $x_0, \ldots, x_{N-1}$,

$$\overline{x} = \frac{1}{N} \sum_{j=0}^{N-1} x_j \tag{14.1.1}$$

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*,

$$\mathrm{Var}(x_0 \ldots x_{N-1}) = \frac{1}{N-1} \sum_{j=0}^{N-1} (x_j - \overline{x})^2 \tag{14.1.2}$$

or its square root, the *standard deviation*,

$$\sigma(x_0 \ldots x_{N-1}) = \sqrt{\mathrm{Var}(x_0 \ldots x_{N-1})} \tag{14.1.3}$$

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 should 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.)

If we calculate equation (14.1.1) many times with different sets of sampled data (each set having N values), the values $\overline{x}$ will themselves have a standard deviation. This is called the *standard error* of the estimated mean $\overline{x}$. When the underlying distribution is Gaussian, it is given approximately by $\sigma/\sqrt{N}$. Correspondingly, there is a standard error of the estimated variance, equation (14.1.2), which is approximately $\sigma^2\sqrt{2/N}$, and a standard error for the estimated σ, equation (14.1.3), which is approximately $\sigma/\sqrt{2N}$.

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

$$\text{ADev}(x_0 \ldots x_{N-1}) = \frac{1}{N} \sum_{j=0}^{N-1} \left| x_j - \overline{x} \right| \tag{14.1.4}$$

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 more 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.

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

$$\text{Skew}(x_0 \ldots x_{N-1}) = \frac{1}{N} \sum_{j=0}^{N-1} \left[\frac{x_j - \overline{x}}{\sigma} \right]^3 \tag{14.1.5}$$

where $\sigma = \sigma(x_0 \ldots x_{N-1})$ is the distribution's standard deviation (14.1.3). A positive value of skewness signifies a distribution with an asymmetric tail extending out toward more positive x; a negative value signifies a distribution whose tail extends out toward 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 error. 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 error 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). (Yes, using the sample mean is likely to give a more accurate estimate than using the true mean!) 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

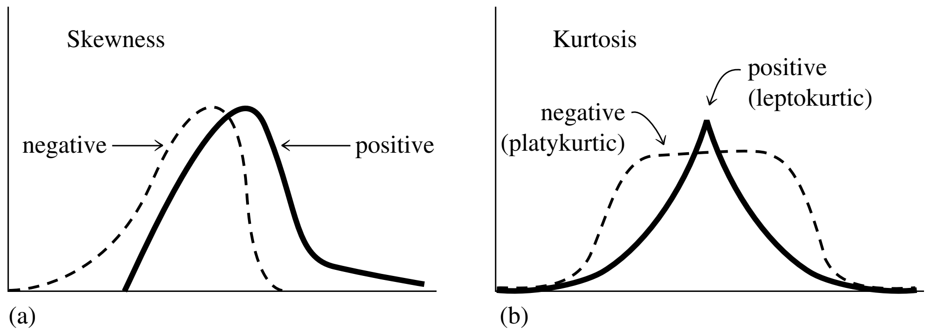


Figure 14.1.1. Distributions whose third and fourth moments are significantly different from a normal (Gaussian) distribution. (a) Skewness or third moment. (b) Kurtosis or fourth moment.

$$\text{Kurt}(x_0 \dots x_{N-1}) = \left\{ \frac{1}{N} \sum_{j=0}^{N-1} \left[\frac{x_j - \overline{x}}{\sigma} \right]^4 \right\} - 3 \tag{14.1.6}$$

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

The standard error 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). (Yes, you are better off using the sample variance.) 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

$$\text{Var}(x_0 \dots x_{N-1}) = \frac{1}{N-1} \left[\left(\sum_{j=0}^{N-1} x_j^2 \right) - N\overline{x}^2 \right] \approx \overline{x^2} - \overline{x}^2 \tag{14.1.7}$$

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 $\text{Var}(x_0 \dots x_{N-1})$ by

$$\text{Var}(x_0 \dots x_{N-1}) = \frac{1}{N-1} \left\{ \sum_{j=0}^{N-1} (x_j - \overline{x})^2 - \frac{1}{N} \left[\sum_{j=0}^{N-1} (x_j - \overline{x}) \right]^2 \right\} \tag{14.1.8}$$

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.

moment.h

```
void moment(VecDoub_I &data, Doub &ave, Doub &adev, Doub &sdev, Doub &var,
    Doub &skew, Doub &curt) {
```
Given an array of `data[0..n-1]`, this routine returns its mean `ave`, average deviation `adev`, standard deviation `sdev`, variance `var`, skewness `skew`, and kurtosis `curt`.
```
    Int j,n=data.size();
    Doub ep=0.0,s,p;
    if (n <= 1) throw("n must be at least 2 in moment");
    s=0.0;                                  First pass to get the mean.
    for (j=0;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=0;j<n;j++) {                         ond, third, and fourth moments of the
        adev += abs(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 != 0.0) {                           ventional definitions.
        skew /= (n*var*sdev);
        curt=curt/(n*var*var)-3.0;
    } else throw("No skew/kurtosis when variance = 0 (in moment)");
}
```

14.1.1 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

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

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

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

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

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

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].

14.1.2 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 $x_0, \ldots, x_{N-1}$ 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, $j = 0, \ldots, N-1$, are sorted into ascending (or, for that matter, descending) order, then the formula for the median is

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

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, including a method for getting a good approximation to the median in a single pass through the data.

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., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill), Chapter 1.

Spiegel, M.R., Schiller, J., and Srinivasan, R.A. 2000, *Schaum's Outline of Theory and Problem of Probability and Statistics*, 2nd ed. (New York: McGraw-Hill).

Stuart, A., and Ord, J.K. 1994, *Kendall's Advanced Theory of Statistics*, 6th ed. (London: Edward Arnold) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*], vol. 1, §10.15

Norusis, M.J. 2006, *SPSS 14.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

Chan, T.F., Golub, G.H., and LeVeque, R.J. 1983, "Algorithms for Computing the Sample Variance: Analysis and Recommendations," *American Statistician*, vol. 37, pp. 242–247.[1]

Cramér, H. 1946, *Mathematical Methods of Statistics* (Princeton, NJ: 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.

14.2.1 Student's t-Test for Significantly Different Means

Applying the concept of standard error, the conventional statistic for measuring the significance of a difference of means is termed *Student's t*. When the two distributions are thought to have the same variance, but possibly different means, then Student's t is computed as follows: First, estimate the standard error of the difference of the means, s_D, from the "pooled variance" by the formula

$$s_D = \sqrt{\frac{\sum_{i\in A}(x_i - \overline{x}_A)^2 + \sum_{i\in B}(x_i - \overline{x}_B)^2}{N_A + N_B - 2}\left(\frac{1}{N_A} + \frac{1}{N_B}\right)} \tag{14.2.1}$$

where each sum is over the points in one sample, the first or second; each mean likewise refers to one sample or the other; and N_A and N_B are the numbers of points in the first and second samples, respectively. Second, compute t by

$$t = \frac{\overline{x}_A - \overline{x}_B}{s_D} \tag{14.2.2}$$

Third, evaluate the p-value or significance of this value of t for Student's distribution with $N_A + N_B - 2$ degrees of freedom, by equation (6.14.11).

The p-value is a number between zero and one. It is the probability that $|t|$ could be this large or larger just by chance, for distributions with equal means. Therefore, a small numerical value of the p-value (0.01 or 0.001) means that the observed difference is "very significant." The function $A(t|\nu)$ in equation (6.14.11) is one minus the p-value.

As a routine, we have

```
void ttest(VecDoub_I &data1, VecDoub_I &data2, Doub &t, Doub &prob)
```
stattests.h

Given the arrays `data1[0..n1-1]` and `data2[0..n2-1]`, returns Student's t as `t`, and its p-value as `prob`, small values of `prob` indicating that the arrays have significantly different means. The data arrays are assumed to be drawn from populations with the same true variance.

```
{
    Beta beta;
    Doub var1,var2,svar,df,ave1,ave2;
    Int n1=data1.size(), n2=data2.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    df=n1+n2-2;                                    Degrees of freedom.
    svar=((n1-1)*var1+(n2-1)*var2)/df;             Pooled variance.
    t=(ave1-ave2)/sqrt(svar*(1.0/n1+1.0/n2));
    prob=beta.betai(0.5*df,0.5,df/(df+t*t));       See equation (6.14.11).
}
```

which makes use of the following routine for computing the mean and variance of a set of numbers,

moment.h

```
void avevar(VecDoub_I &data, Doub &ave, Doub &var) {
Given array data[0..n-1], returns its mean as ave and its variance as var.
    Doub s,ep;
    Int j,n=data.size();
    ave=0.0;
    for (j=0;j<n;j++) ave += data[j];
    ave /= n;
    var=ep=0.0;
    for (j=0;j<n;j++) {
        s=data[j]-ave;
        ep += s;
        var += s*s;
    }
    var=(var-ep*ep/n)/(n-1);            Corrected two-pass formula (14.1.8).
}
```

The next case to consider is where the two distributions have significantly different variances, but we nevertheless want to know if their means are the same or different. (A treatment for baldness has caused some patients to *lose* all their hair and turned others into werewolves, but we want to know if it helps cure baldness *on the average*!) Be suspicious of the unequal-variance t-test: If two distributions have very different variances, then they may also be substantially different in shape; in that case, the difference of the means may not be a particularly useful thing to know.

To find out whether the two data sets have variances that are significantly different, you use the *F-test*, described later on in this section.

The relevant statistic for the unequal-variance t-test is

$$t = \frac{\overline{x}_A - \overline{x}_B}{[\mathrm{Var}(x_A)/N_A + \mathrm{Var}(x_B)/N_B]^{1/2}} \tag{14.2.3}$$

This statistic is distributed *approximately* as Student's t with a number of degrees of freedom equal to

$$\frac{\left[\dfrac{\mathrm{Var}(x_A)}{N_A} + \dfrac{\mathrm{Var}(x_B)}{N_B}\right]^2}{\dfrac{\left[\mathrm{Var}(x_A)/N_A\right]^2}{N_A - 1} + \dfrac{\left[\mathrm{Var}(x_B)/N_B\right]^2}{N_B - 1}} \tag{14.2.4}$$

Expression (14.2.4) is in general not an integer, but equation (6.14.11) doesn't care.

The routine is

stattests.h

```
void tutest(VecDoub_I &data1, VecDoub_I &data2, Doub &t, Doub &prob) {
```
Given the arrays data1[0..n1-1] and data2[0..n2-1], this routine returns Student's t as t, and its p-value as prob, small values of prob indicating that the arrays have significantly different means. The data arrays are allowed to be drawn from populations with unequal variances.
```
    Beta beta;
    Doub var1,var2,df,ave1,ave2;
    Int n1=data1.size(), n2=data2.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    t=(ave1-ave2)/sqrt(var1/n1+var2/n2);
    df=SQR(var1/n1+var2/n2)/(SQR(var1/n1)/(n1-1)+SQR(var2/n2)/(n2-1));
    prob=beta.betai(0.5*df,0.5,df/(df+SQR(t)));
}
```

Our final example of a Student's t-test is the case of *paired samples*. Here we imagine that much of the variance in *both* samples is due to effects that are point-by-point identical in the two samples. For example, we might have two job candidates who have each been rated by the same ten members of a hiring committee. We want to know if the means of the ten scores differ significantly. We first try ttest above, and obtain a value of prob that is not especially significant (e.g., > 0.05). But perhaps the significance is being washed out by the tendency of some committee members always to give high scores and others always to give low scores, which increases the apparent variance and thus decreases the significance of any difference in the means. We thus try the paired-sample formulas,

$$\text{Cov}(x_A, x_B) \equiv \frac{1}{N-1}\sum_{i=0}^{N-1}(x_{Ai} - \overline{x}_A)(x_{Bi} - \overline{x}_B) \tag{14.2.5}$$

$$s_D = \left[\frac{\text{Var}(x_A) + \text{Var}(x_B) - 2\text{Cov}(x_A, x_B)}{N}\right]^{1/2} \tag{14.2.6}$$

$$t = \frac{\overline{x}_A - \overline{x}_B}{s_D} \tag{14.2.7}$$

where N is the number in each sample (number of pairs). Notice that it is important that a particular value of i label the corresponding points in each sample, that is, the ones that are paired. The p-value for the t statistic in (14.2.7) is evaluated for $N-1$ degrees of freedom.

The routine is

stattests.h

```
void tptest(VecDoub_I &data1, VecDoub_I &data2, Doub &t, Doub &prob) {
```
Given the paired arrays data1[0..n-1] and data2[0..n-1], this routine returns Student's t for paired data as t, and its p-value as prob, small values of prob indicating a significant difference of means.
```
    Beta beta;
    Int j, n=data1.size();
    Doub var1,var2,ave1,ave2,sd,df,cov=0.0;
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    for (j=0;j<n;j++) cov += (data1[j]-ave1)*(data2[j]-ave2);
    cov /= (df=n-1);
    sd=sqrt((var1+var2-2.0*cov)/n);
    t=(ave1-ave2)/sd;
    prob=beta.betai(0.5*df,0.5,df/(df+t*t));
}
```

14.2.2 F-Test for Significantly Different Variances

The *F-test* tests the hypothesis that two samples have different variances by trying to reject the null hypothesis that their variances are actually consistent. The statistic F is the ratio of one variance to the other, so values either $\gg 1$ or $\ll 1$ will indicate very significant differences. The distribution of F in the null case is given in equation (6.14.49), which is evaluated using the routine `betai`. In the most common case, we are willing to disprove the null hypothesis (of equal variances) by either very large or very small values of F, so the correct p-value is *two-tailed*, the sum of two incomplete beta functions. It turns out, by equation (6.4.3), that the two tails are always equal; we need compute only one, and double it. Occasionally, when the null hypothesis is strongly viable, the identity of the two tails can become confused, giving an indicated probability greater than one. Changing the probability to two minus itself correctly exchanges the tails. These considerations and equation (6.4.3) give the routine

stattests.h
```
void ftest(VecDoub_I &data1, VecDoub_I &data2, Doub &f, Doub &prob) {
```
Given the arrays `data1[0..n1-1]` and `data2[0..n2-1]`, this routine returns the value of `f`, and its p-value as `prob`. Small values of `prob` indicate that the two arrays have significantly different variances.
```
    Beta beta;
    Doub var1,var2,ave1,ave2,df1,df2;
    Int n1=data1.size(), n2=data2.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    if (var1 > var2) {                      Make F the ratio of the larger variance to the smaller
        f=var1/var2;                            one.
        df1=n1-1;
        df2=n2-1;
    } else {
        f=var2/var1;
        df1=n2-1;
        df2=n1-1;
    }
    prob = 2.0*beta.betai(0.5*df2,0.5*df1,df2/(df2+df1*f));
    if (prob > 1.0) prob=2.-prob;
}
```

CITED REFERENCES AND FURTHER READING:

Spiegel, M.R., Schiller, J., and Srinivasan, R.A. 2000, *Schaum's Outline of Theory and Problem of Probability and Statistics*, 2nd ed. (New York: McGraw-Hill).

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press), Chapter 9.

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press), Chapters 7–8.

Norusis, M.J. 2006, *SPSS 14.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

14.3 Are Two Distributions Different?

Given two sets of data, we can generalize the questions asked in the previous section and ask the single question: Are the two sets drawn from the same distribution function, or from different distribution functions? Equivalently, in proper statistical language, "Can we disprove, to a certain required level of significance, the null hypothesis that two data sets are drawn from the same population distribution function?" Disproving the null hypothesis in effect proves that the data sets are from different distributions. Failing to disprove the null hypothesis, on the other hand, only shows that the data sets can be *consistent* with a single distribution function. One can never *prove* that two data sets come from a single distribution, since, e.g., no practical amount of data can distinguish between two distributions that differ only by one part in 10^{10}.

Proving that two distributions are different, or showing that they are consistent, is a task that comes up all the time in many areas of research: Are the visible stars distributed uniformly in the sky? (That is, is the distribution of stars as a function of declination — position in the sky — the same as the distribution of sky area as a function of declination?) Are educational patterns the same in Brooklyn as in the Bronx? (That is, are the distributions of people as a function of last-grade-attended the same?) Do two brands of fluorescent lights have the same distribution of burn-out times? Is the incidence of chicken pox the same for first-born, second-born, third-born children, etc.?

These four examples illustrate the four combinations arising from two different dichotomies: (1) The data are either continuous or binned. (2) Either we wish to compare one data set to a known distribution, or we wish to compare two equally unknown data sets. The data sets on fluorescent lights and on stars are continuous, since we can be given lists of individual burnout times or of stellar positions. The data sets on chicken pox and educational level are binned, since we are given tables of numbers of events in discrete categories: first-born, second-born, etc.; or 6th grade, 7th grade, etc. Stars and chicken pox, on the other hand, share the property that the null hypothesis is a known distribution (distribution of area in the sky, or incidence of chicken pox in the general population). Fluorescent lights and educational level involve the comparison of two equally unknown data sets (the two brands, or Brooklyn and the Bronx).

One can always turn continuous data into binned data, by grouping the events into specified ranges of the continuous variable(s): declinations between 0 and 10 degrees, 10 and 20, 20 and 30, etc. Binning involves a loss of information, however. Also, there is often considerable arbitrariness as to how the bins should be chosen. Along with many other investigators, we prefer to avoid unnecessary binning of data.

The accepted test for differences between binned distributions is the *chi-square test*. For continuous data as a function of a single variable, the most generally accepted test is the *Kolmogorov-Smirnov test*. We consider each in turn.

14.3.1 Chi-Square Test

Suppose that N_i is the number of events observed in the ith bin, and that n_i is the number expected according to some known distribution. Note that the N_i's are integers, while the n_i's may not be. Then the chi-square statistic is

$$\chi^2 = \sum_i \frac{(N_i - n_i)^2}{n_i} \tag{14.3.1}$$

where the sum is over all bins. A large value of χ^2 indicates that the null hypothesis (that the N_i's are drawn from the population represented by the n_i's) is rather unlikely.

Any term j in (14.3.1) with $0 = n_j = N_j$ should be omitted from the sum. A term with $n_j = 0,\ N_j \neq 0$ gives an infinite χ^2, as it should, since in this case the N_i's cannot possibly be drawn from the n_i's!

The *chi-square probability function* $Q(\chi^2|\nu)$ is an incomplete gamma function, and was already discussed in §6.14 (see equation 6.14.38). Strictly speaking, $Q(\chi^2|\nu)$ is the probability that the sum of the squares of ν random *normal* variables of unit variance (and zero mean) will be greater than χ^2. The terms in the sum (14.3.1) are not exactly the squares of a normal variable. However, if the number of events in each bin is large ($\gg 1$), then the normal distribution is approximately achieved and the chi-square probability function is a good approximation to the distribution of (14.3.1) in the case of the null hypothesis. Its use to estimate the p-value significance of the chi-square test is standard (but see §14.3.2).

The appropriate value of ν, the number of degrees of freedom, bears some additional discussion. If the data are collected with the model n_i's fixed — that is, not later renormalized to fit the total observed number of events ΣN_i — then ν equals the number of bins N_B. (Note that this is *not* the total number of *events*!) Much more commonly, the n_i's are normalized after the fact so that their sum equals the sum of the N_i's. In this case, the correct value for ν is $N_B - 1$, and the model is said to have one constraint (`knstrn=1` in the program below). If the model that gives the n_i's has additional free parameters that were adjusted after the fact to agree with the data, then each of these additional "fitted" parameters decreases ν (and increases `knstrn`) by one additional unit.

We have, then, the following program:

stattests.h
```
void chsone(VecDoub_I &bins, VecDoub_I &ebins, Doub &df,
    Doub &chsq, Doub &prob, const Int knstrn=1) {
```
Given the array `bins[0..nbins-1]` containing the observed numbers of events, and an array `ebins[0..nbins-1]` containing the expected numbers of events, and given the number of constraints `knstrn` (normally one), this routine returns (trivially) the number of degrees of freedom `df`, and (nontrivially) the chi-square `chsq` and the p-value `prob`. A small value of `prob` indicates a significant difference between the distributions `bins` and `ebins`. Note that `bins` and `ebins` are both double arrays, although `bins` will normally contain integer values.
```
    Gamma gam;
    Int j,nbins=bins.size();
    Doub temp;
    df=nbins-knstrn;
    chsq=0.0;
    for (j=0;j<nbins;j++) {
        if (ebins[j]<0.0 || (ebins[j]==0. && bins[j]>0.))
            throw("Bad expected number in chsone");
        if (ebins[j]==0.0 && bins[j]==0.0) {
            --df;                              No data means one less degree of free-
        } else {                                  dom.
            temp=bins[j]-ebins[j];
            chsq += temp*temp/ebins[j];
        }
    }
    prob=gam.gammq(0.5*df,0.5*chsq);         Chi-square probability function. See §6.2.
}
```

Next we consider the case of comparing *two* binned data sets. Let R_i be the number of events in bin i for the first data set and S_i the number of events in the same bin i for the second data set. Then the chi-square statistic is

$$\chi^2 = \sum_i \frac{(R_i - S_i)^2}{R_i + S_i} \tag{14.3.2}$$

Comparing (14.3.2) to (14.3.1), you should note that the denominator of (14.3.2) is *not* just the average of R_i and S_i (which would be an estimator of n_i in 14.3.1). Rather, it is twice the average, the sum. The reason is that each term in a chi-square sum is supposed to approximate the square of a normally distributed quantity with unit variance. The variance of the difference of two normal quantities is the sum of their individual variances, not the average.

If the data were collected in such a way that the sum of the R_i's is necessarily equal to the sum of S_i's, then the number of degrees of freedom is equal to one less than the number of bins, $N_B - 1$ (that is, `knstrn` $= 1$), the usual case. If this requirement were absent, then the number of degrees of freedom would be N_B. Example: A birdwatcher wants to know whether the distribution of sighted birds as a function of species is the same this year as last. Each bin corresponds to one species. If the birdwatcher takes his data to be the first 1000 birds that he saw in each year, then the number of degrees of freedom is $N_B - 1$. If he takes his data to be all the birds he saw on a random sample of days, the same days in each year, then the number of degrees of freedom is N_B (`knstrn` $= 0$). In this latter case, note that he is also testing whether the birds were more numerous overall in one year or the other: That is the extra degree of freedom. Of course, any additional constraints on the data set lower the number of degrees of freedom (i.e., increase `knstrn` to *more positive* values) in accordance with their number.

The program is

stattests.h

```
void chstwo(VecDoub_I &bins1, VecDoub_I &bins2, Doub &df,
    Doub &chsq, Doub &prob, const Int knstrn=1) {
```
Given the arrays `bins1[0..nbins-1]` and `bins2[0..nbins-1]`, containing two sets of binned data, and given the number of constraints `knstrn` (normally 1 or 0), this routine returns the number of degrees of freedom `df`, the chi-square `chsq`, and the p-value `prob`. A small value of `prob` indicates a significant difference between the distributions `bins1` and `bins2`. Note that `bins1` and `bins2` are both `double` arrays, although they will normally contain integer values.
```
    Gamma gam;
    Int j,nbins=bins1.size();
    Doub temp;
    df=nbins-knstrn;
    chsq=0.0;
    for (j=0;j<nbins;j++)
        if (bins1[j] == 0.0 && bins2[j] == 0.0)
            --df;                              No data means one less degree of free-
        else {                                     dom.
            temp=bins1[j]-bins2[j];
            chsq += temp*temp/(bins1[j]+bins2[j]);
        }
    prob=gam.gammq(0.5*df,0.5*chsq);           Chi-square probability function. See §6.2.
}
```

Equation (14.3.2) and the routine `chstwo` both apply to the case where the total number of data points is the same in the two binned sets, or to the case where any difference in the totals is part of what is being tested for. For intentionally unequal sample sizes, the formula analogous to (14.3.2) is

$$\chi^2 = \sum_i \frac{(\sqrt{S/R}\,R_i - \sqrt{R/S}\,S_i)^2}{R_i + S_i} \tag{14.3.3}$$

where

$$R \equiv \sum_i R_i \qquad S \equiv \sum_i S_i \tag{14.3.4}$$

are the respective numbers of data points. It is straightforward to make the corresponding change in chstwo. The fact that R_i and S_i occur in the denominator of equation (14.3.3) with equal weights may seem unintuitive, but the following heuristic derivation shows how this comes about: In the null hypothesis that R_i and S_i are drawn from the same distribution, we can estimate the probability associated with bin i as

$$\widehat{p}_i = \frac{R_i + S_i}{R + S} \tag{14.3.5}$$

The expected number of counts is thus

$$\widehat{R}_i = R\,\widehat{p}_i \qquad \text{and} \qquad \widehat{S}_i = S\,\widehat{p}_i \tag{14.3.6}$$

and the chi-square statistic summing over all observations is

$$\chi^2 = \sum_i \frac{(R_i - \widehat{R}_i)^2}{\widehat{R}_i} + \sum_i \frac{(S_i - \widehat{S}_i)^2}{\widehat{S}_i} \tag{14.3.7}$$

Substituting equations (14.3.6) and (14.3.5) into equation (14.3.7) gives, after some algebra, exactly equation (14.3.3). Although there are $2N_B$ terms in equation (14.3.7), the number of degrees of freedom is actually $N_B - 1$ (minus any additional constraints), the same as equation (14.3.2), because we implicitly estimated $N_B + 1$ parameters, the $\widehat{p}_i$'s and the ratio of the two sample sizes. This number of degrees of freedom must thus be subtracted from the original $2N_B$.

For three or more samples, see equation (14.4.3) and related discussion.

14.3.2 Chi-Square with Small Numbers of Counts

When a significant fraction of bins have small numbers of counts ($\lesssim$ 10, say), then the χ^2 statistics (14.3.1), (14.3.2), and (14.3.3) are not well approximated by a chi-square probability function. Let us quantify this problem and suggest some remedies.

Consider first equation (14.3.1). In the null hypothesis, the count in an individual bin, N_i, is a Poisson deviate of mean n_i, so it occurs with probability

$$p(N_i|n_i) = \exp(-n_i)\frac{n_i^{N_i}}{N_i!} \tag{14.3.8}$$

(cf. equation 6.14.61). We can calculate the mean μ and variance σ^2 of the term $(N_i - n_i)^2/n_i$ by evaluating the appropriate expectation values. There are various analytical ways to do this. The sums, and the answers, are

$$\begin{aligned}\mu &= \sum_{N_i=0}^{\infty} p(N_i|n_i)\,\frac{(N_i-n_i)^2}{n_i} = 1\\ \sigma^2 &= \left\{\sum_{N_i=0}^{\infty} p(N_i|n_i)\left[\frac{(N_i-n_i)^2}{n_i}\right]^2\right\} - \mu^2 = 2 + \frac{1}{n_i}\end{aligned} \tag{14.3.9}$$

Now we can see what the problem is: Equation (14.3.9) says that each term in (14.3.1) adds, on average, 1 to the value of the χ^2 statistic, and slightly more than 2 to its variance. But the variance of the chi-square probability function is *exactly* twice its mean (equation 6.14.37). If a significant fraction of n_i's are small, then quite probable values of the χ^2 statistic will appear to lie farther out on the tail than they actually are, so that the null hypothesis may be rejected even when it is true.

Several approximate remedies are possible. One is simply to rescale the observed χ^2 statistic so as to "fix" its variance, an idea due to Lucy [1]. If we define

$$Y^2 \equiv \nu + \sqrt{\frac{2\nu}{2\nu + \sum_i n_i^{-1}}}\left(\chi^2 - \nu\right) \tag{14.3.10}$$

where ν is the number of degrees of freedom (see discussion above), then Y^2 is asymptotically approximated by the chi-square probability function even when many n_i's are small. The basic idea in (14.3.10) is to subtract off the mean, rescale the difference from the mean, and then add back the mean. Lucy [1] also defines a similar Z^2 statistic by rescaling not the χ^2 sum of all the terms, but the terms individually, using equation (14.3.9) separately for each.

Another possibility, valid when ν is large, is to use the central limit theorem directly. From its mean and standard deviation, we now know that the χ^2 statistic must be approximately the normal distribution,

$$\chi^2 \sim \mathrm{N}\left(\nu, \left[2\nu + \sum_i n_i^{-1}\right]^{1/2}\right) \tag{14.3.11}$$

We can then obtain p-values from equation (6.14.2), computing a complementary error function. (The p-value is one minus that cdf.)

The same ideas go through in the case of two binned data sets, with counts R_i and S_i, and total numbers of counts R and S (equation 14.3.3, with equation 14.3.2 as the special case with $R = S$). Now, in the null hypothesis, and glossing over some technical issues beyond our scope, we can think of $T_i \equiv R_i + S_i$ as being fixed, while R_i is a random variable drawn from the binomial distribution

$$R_i \sim \text{Binomial}\left(T_i, \frac{R}{R+S}\right) \tag{14.3.12}$$

(see equation 6.14.67). Calculating moments over the binomial distribution, one can obtain as analogs of equations (14.3.9)

$$\begin{aligned} \mu &= 1 \\ \sigma^2 &= 2 + \left[\frac{(R+S)^2}{RS} - 6\right]\frac{1}{R_i + S_i} \end{aligned} \tag{14.3.13}$$

Notice that, now, depending on the values of R and S, the variance can be either greater or less than its nominal value 2, and that it is less than 2 for the case $R = S$. The formulas (14.3.9) and (14.3.13) are originally due to Haldane [2] (see also [3]).

Summing over i, one obtains the analogs of equations (14.3.10) and (14.3.11) simply by the replacement,

$$\sum_i n_i^{-1} \longrightarrow \left[\frac{(R+S)^2}{RS} - 6\right]\sum_i \frac{1}{R_i + S_i} \tag{14.3.14}$$

In fact, equation (14.3.9) is a limiting form of equation (14.3.13) in just the same limit that Poisson is a limiting form of binomial, namely

$$S \to \infty, \qquad \frac{R}{R+S} S_i \to n_i, \qquad R_i \to N_i \tag{14.3.15}$$

There are also other ways of treating small-number counts, including the likelihood ratio test [4], the *modified Neyman* χ^2 [5], and the *chi-square-gamma* statistic [5].

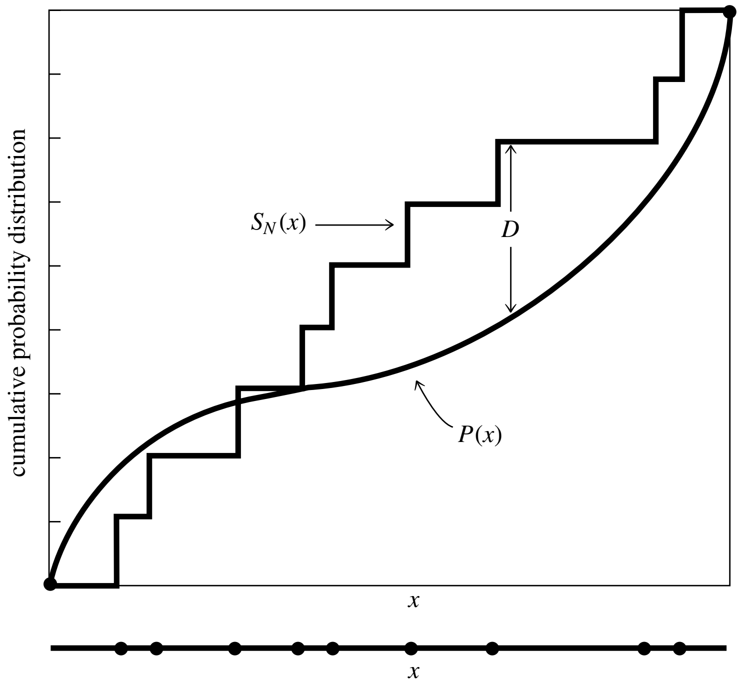


Figure 14.3.1. Kolmogorov-Smirnov statistic D. A measured distribution of values in x (shown as N dots on the lower abscissa) is to be compared with a theoretical distribution whose cumulative probability distribution is plotted as $P(x)$. A step-function cumulative probability distribution $S_N(x)$ is constructed, one that rises an equal amount at each measured point. D is the greatest distance between the two cumulative distributions.

14.3.3 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (or *K–S*) test is applicable to unbinned distributions that are functions of a single independent variable, that is, to data sets where each data point can be associated with a single number (lifetime of each lightbulb when it burns out, or declination of each star). In such cases, the list of data points can be easily converted to an unbiased estimator $S_N(x)$ of the *cumulative* distribution function of the probability distribution from which it was drawn: If the N events are located at values x_i, $i = 0, \ldots, N-1$, then $S_N(x)$ is the function giving the fraction of data points to the left of a given value x. This function is obviously constant between consecutive (i.e., sorted into ascending order) x_i's and jumps by the same constant $1/N$ at each x_i. (See Figure 14.3.1.)

Different distribution functions, or sets of data, give different cumulative distribution function estimates by the above procedure. However, all cumulative distribution functions agree at the smallest allowable value of x (where they are zero) and at the largest allowable value of x (where they are unity). (The smallest and largest values might of course be $\pm\infty$.) So it is the behavior between the largest and smallest values that distinguishes distributions.

One can think of any number of statistics to measure the overall difference between two cumulative distribution functions: the absolute value of the area between them, for example, or their integrated mean square difference. The Kolmogorov-Smirnov D is a particularly simple measure: It is defined as the *maximum value* of the absolute difference between two cumulative distribution functions. Thus, for comparing one data set's $S_N(x)$ to a known cumulative distribution function $P(x)$, the K–S statistic is

$$D = \max_{-\infty<x<\infty} |S_N(x) - P(x)| \tag{14.3.16}$$

while for comparing two different cumulative distribution functions $S_{N_1}(x)$ and $S_{N_2}(x)$, the K–S statistic is

$$D = \max_{-\infty<x<\infty} \left|S_{N_1}(x) - S_{N_2}(x)\right| \tag{14.3.17}$$

What makes the K–S statistic useful is that *its* distribution in the case of the null hypothesis (data sets drawn from the same distribution) can be calculated, at least to a useful approximation, thus giving the p-value significance of any observed nonzero value of D. A central feature of the K–S test is that it is invariant under reparametrization of x; in other words, you can locally slide or stretch the x-axis in Figure 14.3.1, and the maximum distance D remains unchanged. For example, you will get the same significance using x as using $\log x$.

The function that enters into the calculation of the p-value was discussed previously in §6.14, was defined in equations (6.14.56) and (6.14.57), and was implemented in the object `KSdist`. In terms of the function Q_{KS}, the p-value of an observed value of D (as a disproof of the null hypothesis that the distributions are the same) is given approximately [6] by the formula

$$\text{Probability}\ (D > \text{observed}\ ) = Q_{KS}\left(\left[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\right] D\right) \tag{14.3.18}$$

where N_e is the effective number of data points, $N_e = N$ for the case (14.3.16) of one distribution, and

$$N_e = \frac{N_1 N_2}{N_1 + N_2} \tag{14.3.19}$$

for the case (14.3.17) of two distributions, where N_1 is the number of data points in the first distribution and N_2 the number in the second.

The nature of the approximation involved in (14.3.18) is that it becomes asymptotically accurate as the N_e becomes large, but is already quite good for $N_e \geq 4$, as small a number as one might ever actually use. (See [6].)

Here is the routine for the case of one distribution:

```
void ksone(VecDoub_IO &data, Doub func(const Doub), Doub &d, Doub &prob)        kstests.h
```
Given an array `data[0..n-1]`, and given a user-supplied function of a single variable `func` that is a cumulative distribution function ranging from 0 (for smallest values of its argument) to 1 (for largest values of its argument), this routine returns the K–S statistic `d` and the p-value `prob`. Small values of `prob` show that the cumulative distribution function of `data` is significantly different from `func`. The array `data` is modified by being sorted into ascending order.
```
{
    Int j,n=data.size();
    Doub dt,en,ff,fn,fo=0.0;
    KSdist ks;
    sort(data);                         If the data are already sorted into as-
                                            cending order, then this call can be
                                            omitted.
    en=n;
    d=0.0;
    for (j=0;j<n;j++) {                 Loop over the sorted data points.
        fn=(j+1)/en;                    Data's c.d.f. after this step.
        ff=func(data[j]);                         Compare to the user-supplied function.
        dt=MAX(abs(fo-ff),abs(fn-ff));            Maximum distance.
        if (dt > d) d=dt;
        fo=fn;
    }
    en=sqrt(en);
    prob=ks.qks((en+0.12+0.11/en)*d);             Compute p-value.
}
```

While the K-S statistic is intended for use with a continuous distribution, it can also be used for a discrete distribution. In this case, it can be shown that the test is conservative, that is, the statistic returned is no larger than in the continuous case. If you allow discrete variables in the case of two distributions, you have to consider how to deal with ties. The standard way to handle ties is to combine all the tied data points and add them to the cdf at once (see, e.g., [7]). This refinement is included in the routine kstwo.

kstests.h

```
void kstwo(VecDoub_IO &data1, VecDoub_IO &data2, Doub &d, Doub &prob)
```
Given an array data1[0..n1-1], and an array data2[0..n2-1], this routine returns the K–S statistic d and the p-value prob for the null hypothesis that the data sets are drawn from the same distribution. Small values of prob show that the cumulative distribution function of data1 is significantly different from that of data2. The arrays data1 and data2 are modified by being sorted into ascending order.
```
{
    Int j1=0,j2=0,n1=data1.size(),n2=data2.size();
    Doub d1,d2,dt,en1,en2,en,fn1=0.0,fn2=0.0;
    KSdist ks;
    sort(data1);
    sort(data2);
    en1=n1;
    en2=n2;
    d=0.0;
    while (j1 < n1 && j2 < n2) {                     If we are not done...
        if ((d1=data1[j1]) <= (d2=data2[j2]))        Next step is in data1.
            do
                fn1=++j1/en1;
            while (j1 < n1 && d1 == data1[j1]);
        if (d2 <= d1)                                Next step is in data2.
            do
                fn2=++j2/en2;
            while (j2 < n2 && d2 == data2[j2]);
        if ((dt=abs(fn2-fn1)) > d) d=dt;
    }
    en=sqrt(en1*en2/(en1+en2));
    prob=ks.qks((en+0.12+0.11/en)*d);                Compute p-value.
}
```

14.3.4 Variants on the K–S Test

The sensitivity of the K–S test to deviations from a cumulative distribution function $P(x)$ is not independent of x. In fact, the K–S test tends to be most sensitive around the median value, where $P(x) = 0.5$, and less sensitive at the extreme ends of the distribution, where $P(x)$ is near 0 or 1. The reason is that the difference $|S_N(x) - P(x)|$ does not, in the null hypothesis, have a probability distribution that is independent of x. Rather, its variance is proportional to $P(x)[1 - P(x)]$, which is largest at $P = 0.5$. Since the K–S statistic (14.3.16) is the maximum difference over all x of two cumulative distribution functions, a deviation that might be statistically significant at *its own* value of x gets compared to the expected chance deviation at $P = 0.5$ and is thus discounted. A result is that, while the K–S test is good at finding *shifts* in a probability distribution, especially changes in the median value, it is not always so good at finding *spreads*, which more affect the tails of the probability distribution, and which may leave the median unchanged.

One way of increasing the power of the K–S statistic out on the tails is to replace D (equation 14.3.16) by a so-called *stabilized* or *weighted* statistic [8-10], for example the *Anderson-Darling statistic*,

$$D^* = \max_{-\infty<x<\infty} \frac{|S_N(x) - P(x)|}{\sqrt{P(x)[1-P(x)]}} \tag{14.3.20}$$

Unfortunately, there is no simple formula analogous to equation (14.3.18) for this statistic, although Noé [11] gives a computational method using a recursion relation and provides a graph of numerical results. There are many other possible similar statistics, for example

$$D^{**} = \int_{P=0}^{1} \frac{[S_N(x) - P(x)]^2}{P(x)[1-P(x)]} dP(x) \tag{14.3.21}$$

which is also discussed by Anderson and Darling (see [9]).

Another approach, which we prefer as simpler and more direct, is due to Kuiper [12,13]. We already mentioned that the standard K–S test is invariant under reparametrizations of the variable x. An even more general symmetry, which guarantees equal sensitivities at all values of x, is to wrap the x-axis around into a circle (identifying the points at $\pm\infty$), and to look for a statistic that is now invariant under all shifts and parametrizations on the circle. This allows, for example, a probability distribution to be "cut" at some central value of x and the left and right halves to be interchanged, without altering the statistic or its significance.

Kuiper's statistic, defined as

$$V = D_+ + D_- = \max_{-\infty<x<\infty} [S_N(x) - P(x)] + \max_{-\infty<x<\infty} [P(x) - S_N(x)] \tag{14.3.22}$$

is the sum of the maximum distance of $S_N(x)$ *above and below* $P(x)$. You should be able to convince yourself that this statistic has the desired invariance on the circle: Sketch the indefinite integral of two probability distributions defined on the circle as a function of angle around the circle, as the angle goes through several times 360°. If you change the starting point of the integration, D_+ and D_- change individually, but their sum is constant.

Furthermore, there is a simple formula for the asymptotic distribution of the statistic V, directly analogous to equations (14.3.18) – (14.3.19). Let

$$Q_{KP}(\lambda) = 2\sum_{j=1}^{\infty} (4j^2\lambda^2 - 1)e^{-2j^2\lambda^2} \tag{14.3.23}$$

which is monotonic and satisfies

$$Q_{KP}(0) = 1 \qquad Q_{KP}(\infty) = 0 \tag{14.3.24}$$

In terms of this function the p-value is [6]

$$\text{Probability } (V > \text{observed }) = Q_{KP}\left(\left[\sqrt{N_e} + 0.155 + 0.24/\sqrt{N_e}\right] V\right) \tag{14.3.25}$$

Here N_e is N in the one-sample case or is given by equation (14.3.19) in the case of two samples.

Of course, Kuiper's test is ideal for any problem originally defined on a circle, for example, to test whether the distribution in longitude of something agrees with some theory, or whether two somethings have different distributions in longitude. (See also [14].)

We will leave to you the coding of routines analogous to `ksone`, `kstwo`, and `KSdist::qks`. (For $\lambda < 0.4$, don't try to do the sum 14.3.23. Its value is 1, to 7 figures, but the series can require many terms to converge, and loses accuracy to roundoff.)

Two final cautionary notes: First, we should mention that all varieties of the K–S test lack the ability to discriminate some kinds of distributions. A simple example is a probability distribution with a narrow "notch" within which the probability falls to zero. Such a distribution is of course ruled out by the existence of even one data point within the notch, but, because of its cumulative nature, a K–S test would require many data points in the notch before signaling a discrepancy.

Second, we should note that, if you estimate any parameters from a data set (e.g., a mean and variance), then the distribution of the K–S statistic D for a cumulative distribution function $P(x)$ that *uses the estimated parameters* is no longer given by equation (14.3.18). In general, you will have to determine the new distribution yourself, e.g., by Monte Carlo methods.

CITED REFERENCES AND FURTHER READING:

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press), Chapter 14.

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press), Chapter 14.

Lucy, L.B. 2000, "Hypothesis Testing for Meagre Data Sets," *Monthly Notices of the Royal Astronomical Society*, vol. 318, pp. 92–100.[1]

Haldane, J.B.S. 1937, "The Exact Value of the Moments of the Distribution of χ^2, Used as a Test of Goodness of Fit, When Expectations Are Small," *Biometrika*, vol. 29, pp. 133–143.[2]

Read, T.R.C., and Cressie, N.A.C. 1988, *Goodness-of-Fit Statistics for Discrete Multivariate Data* (New York: Springer), pp. 140–144.[3]

Baker, S., and Cousins, R.D. 1984, "Clarification of the Use of Chi-Square and Likelihood Functions in Fits to Histograms," *Nuclear Instruments and Methods in Physics Research*, vol. 221, pp. 437–442.[4]

Mighell, K.J. 1999, "Parameter Estimation in Astronomy with Poisson-Distributed Data. I.The χ^2_γ Statistic," *Astrophysical Journal*, vol. 518, pp. 380–393[5]

Stephens, M.A. 1970, "Use of Kolmogorov-Smirnov, Cramer-von Mises and Related Statistics without Extensive Tables," *Journal of the Royal Statistical Society*, ser. B, vol. 32, pp. 115–122.[6]

Hollander, M., and Wolfe, D.A. 1999, *Nonparametric Statistical Methods*, 2nd ed. (New York: Wiley), p. 183.[7]

Anderson, T.W., and Darling, D.A. 1952, "Asymptotic Theory of Certain Goodness of Fit Criteria Based on Stochastic Processes," *Annals of Mathematical Statistics*, vol. 23, pp. 193–212.[8]

Darling, D.A. 1957, "The Kolmogorov-Smirnov, Cramer-von Mises Tests," *Annals of Mathematical Statistics*, vol. 28, pp. 823–838.[9]

Michael, J.R. 1983, "The Stabilized Probability Plot," *Biometrika*, vol. 70, no. 1, pp. 11–17.[10]

Noé, M. 1972, "The Calculation of Distributions of Two-Sided Kolmogorov-Smirnov Type Statistics," *Annals of Mathematical Statistics*, vol. 43, pp. 58–64.[11]

Kuiper, N.H. 1962, "Tests Concerning Random Points on a Circle," *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, ser. A., vol. 63, pp. 38–47.[12]

Stephens, M.A. 1965, "The Goodness-of-Fit Statistic V_n: Distribution and Significance Points," *Biometrika*, vol. 52, pp. 309–321.[13]

Fisher, N.I., Lewis, T., and Embleton, B.J.J. 1987, *Statistical Analysis of Spherical Data* (New York: Cambridge University Press).[14]

14.4 Contingency Table Analysis of Two Distributions

In this section and the next three sections, we deal with *measures of association* for two distributions. The situation is this: Each data point has two or more different quantities associated with it, and we want to know whether knowledge of one quantity gives us any demonstrable advantage in predicting the value of another quantity. In many cases, one variable will be an "independent" or "control" variable, and another will be a "dependent" or "measured" variable. Then, we want to know if the latter variable *is* in fact dependent on or *associated* with the former variable. If it is, we want to have some quantitative measure of the strength of the association. One often hears this loosely stated as the question of whether two variables are *correlated* or *uncorrelated*, but we will reserve those terms for a particular kind of association (linear, or at least monotonic), as discussed in §14.5 and §14.6.

Notice that, as in previous sections, the different concepts of significance and strength appear: The association between two distributions may be very significant even if that association is weak — if the quantity of data is large enough.

It is useful to distinguish among some different kinds of variables, with different categories forming a loose hierarchy.

- A variable is called *nominal* if its values are the members of some unordered set. For example, "state of residence" is a nominal variable that (in the U.S.) takes on one of 50 values; in astrophysics, "type of galaxy" is a nominal variable with the three values "spiral," "elliptical," and "irregular."
- A variable is termed *ordinal* if its values are the members of a discrete, but ordered, set. Examples are grade in school, planetary order from the Sun (Mercury = 1, Venus = 2, ...), and number of offspring. There need not be any concept of "equal metric distance" between the values of an ordinal variable, only that they be intrinsically ordered.
- We will call a variable *continuous* if its values are real numbers, as are times, distances, temperatures, etc. (Social scientists sometimes distinguish between *interval* and *ratio* continuous variables, but we do not find that distinction very compelling.)

A continuous variable can always be made into an ordinal one by binning it into ranges. If we choose to ignore the ordering of the bins, then we can turn it into a nominal variable. Nominal variables constitute the lowest type of the hierarchy, and therefore the most general. For example, a set of *several* continuous or ordinal variables can be turned, if crudely, into a single nominal variable, by coarsely binning each variable and then taking each distinct combination of bin assignments as a single nominal value. When multidimensional data are sparse, this is often the only sensible way to proceed.

The remainder of this section will deal with measures of association between *nominal* variables. For any pair of nominal variables, the data can be displayed as a *contingency table*, a table whose rows are labeled by the values of one nominal variable, whose columns are labeled by the values of the other nominal variable, and whose entries are nonnegative integers giving the number of observed events for each combination of row and column (see Figure 14.4.1). The analysis of association between nominal variables is thus called *contingency table analysis* or *cross-*

	0. red	1. green	. . .	
0. male	# of red males N_{00}	# of green males N_{01}	. . .	# of males $N_{0\cdot}$
1. female	# of red females N_{10}	# of green females N_{11}	. . .	# of females $N_{1\cdot}$
⋮	⋮	⋮	. . .	⋮
	# of red $N_{\cdot 0}$	# of green $N_{\cdot 1}$	. . .	total # N

Figure 14.4.1. Example of a contingency table for two nominal variables, here sex and color. The row and column marginals (totals) are shown. The variables are "nominal," i.e., the order in which their values are listed is arbitrary and does not affect the result of the contingency table analysis. If the ordering of values has some intrinsic meaning, then the variables are "ordinal" or "continuous," and correlation techniques (§14.5 – §14.6) can be utilized.

tabulation analysis.

The remainder of this section gives an approach, based on the chi-square statistic, that does a good job of characterizing the significance of association but is only so-so as a measure of the strength (principally because its numerical values have no very direct interpretations). We will return to contingency table analysis in §14.7 with an approach, based on the information-theoretic concept of *entropy*, that will say little about the significance of association (use chi-square for that!) but is capable of very elegantly characterizing the strength of an association already known to be significant.

14.4.1 Measures of Association Based on Chi-Square

Some notation first: Let N_{ij} denote the number of events that occur with the first variable x taking on its ith value and the second variable y taking on its jth value. Let N denote the total number of events, the sum of all the N_{ij}'s. Let $N_{i\cdot}$ denote the number of events for which the first variable x takes on its ith value regardless of the value of y; $N_{\cdot j}$ is the number of events with the jth value of y regardless of x. So we have

$$N_{i\cdot} = \sum_j N_{ij} \qquad N_{\cdot j} = \sum_i N_{ij}$$
$$N = \sum_i N_{i\cdot} = \sum_j N_{\cdot j} \tag{14.4.1}$$

In other words, "dot" is a placeholder that means, "sum over the missing index". $N_{\cdot j}$ and $N_{i\cdot}$ are sometimes called the *row and column totals* or *marginals*, but we will use these terms cautiously since we can never keep straight which are the rows and which are the columns!

The null hypothesis is that the two variables x and y have no association. In this case, the probability of a particular value of x given a particular value of y should be the same as the probability of that value of x regardless of y. Therefore, in the null hypothesis, the expected number for any N_{ij}, which we will denote n_{ij}, can be calculated from only the row and column totals,

$$\frac{n_{ij}}{N_{\cdot j}} = \frac{N_{i\cdot}}{N} \qquad \text{which implies} \qquad n_{ij} = \frac{N_{i\cdot}N_{\cdot j}}{N} \tag{14.4.2}$$

Notice that if a column or row total is zero, then the expected number for all the entries in that column or row is also zero; in that case, the never-occurring bin of x or y should simply be removed from the analysis.

The chi-square statistic is now given by equation (14.3.1), which, in the present case, is summed over all entries in the table:

$$\chi^2 = \sum_{i,j} \frac{(N_{ij} - n_{ij})^2}{n_{ij}} \tag{14.4.3}$$

The number of degrees of freedom is equal to the number of entries in the table (product of its row size and column size) minus the number of constraints that have arisen from our use of the data themselves to determine the n_{ij}. Each row total and column total is a constraint, except that this overcounts by one, since the total of the column totals and the total of the row totals both equal N, the total number of data points. Therefore, if the table is of size I by J, the number of degrees of freedom is $IJ - I - J + 1$. Equation (14.4.3), along with the chi-square probability function (§6.2), now give the significance of an association between the variables x and y. Incidentally, the two-sample chi-square test for equality of distributions, equation (14.3.3), is a special case of equation (14.4.3) with $J = 2$ and with the y variable simply a label distinguishing the two samples.

Suppose there is a significant association. How do we quantify its strength, so that (e.g.) we can compare the strength of one association with another? The idea here is to find some reparametrization of χ^2 that maps it into some convenient interval, like 0 to 1, where the result is not dependent on the quantity of data that we happen to sample, but rather depends only on the underlying population from which the data were drawn. There are several different ways of doing this. Two of the more common are called *Cramer's* V and the *contingency coefficient* C.

The formula for Cramer's V is

$$V = \sqrt{\frac{\chi^2}{N \min(I-1, J-1)}} \tag{14.4.4}$$

where I and J are again the numbers of rows and columns, and N is the total number of events. Cramer's V has the pleasant property that it lies between zero and one inclusive, equals zero when there is no association, and equals one only when the association is perfect: All the events in any row lie in one unique column, and vice versa. (In chess parlance, no two rooks, placed on a nonzero table entry, can capture each other.)

In the case of $I = J = 2$, Cramer's V is also referred to as the *phi* statistic.

The contingency coefficient C is defined as

$$C = \sqrt{\frac{\chi^2}{\chi^2 + N}} \tag{14.4.5}$$

It also lies between zero and one, but (as is apparent from the formula) it can never achieve the upper limit. While it can be used to compare the strength of association of two tables with the same I and J, its upper limit depends on I and J. Therefore it can never be used to compare tables of different sizes.

The trouble with both Cramer's V and the contingency coefficient C is that, when they take on values in between their extremes, there is no very direct interpretation of what that value means. For example, you are in Las Vegas, and a friend tells you that there is a small, but significant, association between the color of a croupier's eyes and the occurrence of red and black on his roulette wheel. Cramer's V is about 0.028, your friend tells you. You know what the usual odds against you are (because of the green zero and double zero on the wheel). Is this association sufficient for you to make money? Don't ask us! For a measure of association that is directly applicable to gambling, look at §14.7.

stattests.h

```
void cntab(MatInt_I &nn, Doub &chisq, Doub &df, Doub &prob, Doub &cramrv,
    Doub &ccc)
```

Given a two-dimensional contingency table in the form of an array `nn[0..ni-1][0..nj-1]` of integers, this routine returns the chi-square `chisq`, the number of degrees of freedom `df`, the p-value `prob` (small values indicating a significant association), and two measures of association, Cramer's V (`cramrv`) and the contingency coefficient C (`ccc`).

```
{
    const Doub TINY=1.0e-30;                A small number.
    Gamma gam;
    Int i,j,nnj,nni,minij,ni=nn.nrows(),nj=nn.ncols();
    Doub sum=0.0,expctd,temp;
    VecDoub sumi(ni),sumj(nj);
    nni=ni;                                 Number of rows...
    nnj=nj;                                 ...and columns.
    for (i=0;i<ni;i++) {                    Get the row totals.
        sumi[i]=0.0;
        for (j=0;j<nj;j++) {
            sumi[i] += nn[i][j];
            sum += nn[i][j];
        }
        if (sumi[i] == 0.0) --nni;          Eliminate any zero rows by reducing the num-
    }                                           ber.
    for (j=0;j<nj;j++) {                    Get the column totals.
        sumj[j]=0.0;
        for (i=0;i<ni;i++) sumj[j] += nn[i][j];
        if (sumj[j] == 0.0) --nnj;          Eliminate any zero columns.
    }
    df=nni*nnj-nni-nnj+1;                   Corrected number of degrees of freedom.
    chisq=0.0;
    for (i=0;i<ni;i++) {                    Do the chi-square sum.
        for (j=0;j<nj;j++) {
            expctd=sumj[j]*sumi[i]/sum;
            temp=nn[i][j]-expctd;
            chisq += temp*temp/(expctd+TINY);       Here TINY guarantees that any
        }                                               eliminated row or column will
                                                        not contribute to the sum.
    }
    prob=gam.gammq(0.5*df,0.5*chisq);                    Chi-square probability function.
    minij = nni < nnj ? nni-1 : nnj-1;
    cramrv=sqrt(chisq/(sum*minij));
    ccc=sqrt(chisq/(chisq+sum));
}
```

CITED REFERENCES AND FURTHER READING:

Agresti, A. 2002, *Categorical Data Analysis*, 2nd ed. (New York: Wiley).

Mickey, R.M., Dunn, O.J., and Clark, V.A. 2004, *Applied Statistics: Analysis of Variance and Regression*, 3rd ed. (New York: Wiley).

Norusis, M.J. 2006, *SPSS 14.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

14.5 Linear Correlation

We next turn to measures of association between variables that are ordinal or continuous, rather than nominal. Most widely used is the *linear correlation coefficient*. For pairs of quantities (x_i, y_i), $i = 0, \ldots, N-1$, the linear correlation coefficient r (also called the product-moment correlation coefficient, or *Pearson's r*) is given by the formula

$$r = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_i (x_i - \overline{x})^2}\sqrt{\sum_i (y_i - \overline{y})^2}} \tag{14.5.1}$$

where, as usual, $\overline{x}$ is the mean of the x_i's and $\overline{y}$ is the mean of the y_i's.

The value of r lies between -1 and 1, inclusive. It takes on a value of 1, termed "complete positive correlation," when the data points lie on a perfect straight line with positive slope, with x and y increasing together. The value 1 holds independent of the magnitude of the slope. If the data points lie on a perfect straight line with negative slope, y decreasing as x increases, then r has the value -1; this is called "complete negative correlation." A value of r near zero indicates that the variables x and y are *uncorrelated*.

When a correlation is known to be significant, r is one conventional way of summarizing its strength. In fact, the value of r can be translated into a statement about what residuals (root-mean-square deviations) are to be expected if the data are fitted to a straight line by the least-squares method (see §15.2, especially equation 15.2.13). Unfortunately, r is a rather poor statistic for deciding *whether* an observed correlation is statistically significant and/or whether one observed correlation is significantly stronger than another. The reason is that r is ignorant of the individual distributions of x and y, so there is no universal way to compute its distribution in the case of the null hypothesis.

About the only general statement that can be made is this: If the null hypothesis is that x and y are uncorrelated, and if the distributions for x and y each have enough convergent moments ("tails" die off sufficiently rapidly), and if N is large (typically

> 500), then r is distributed approximately normally, with a mean of zero and a standard deviation of $1/\sqrt{N}$. In that case, the (double-sided) significance of the correlation, that is, the probability that $|r|$ should be larger than its observed value in the null hypothesis, is

$$\operatorname{erfc}\left(\frac{|r|\sqrt{N}}{\sqrt{2}}\right) \tag{14.5.2}$$

where $\operatorname{erfc}(x)$ is the complementary error function, equation (6.2.10), computed by the routines `Erf.erfc` or `erfcc` of §6.2. A small value of (14.5.2) indicates that the two distributions are significantly correlated. (See expression 14.5.9 below for a more accurate test.)

Most statistics books try to go beyond (14.5.2) and give additional statistical tests that can be made using r. In almost all cases, however, these tests are valid only for a very special class of hypotheses, namely that the distributions of x and y jointly form a *binormal* or *two-dimensional Gaussian* distribution around their mean values, with joint probability density

$$p(x,y)\,dxdy = \text{const.} \times \exp\left[-\tfrac{1}{2}(a_{00}x^2 - 2a_{01}xy + a_{11}y^2)\right]\,dxdy \tag{14.5.3}$$

where a_{00}, a_{01}, and a_{11} are arbitrary constants. For this distribution r has the value

$$r = -\frac{a_{01}}{\sqrt{a_{00}a_{11}}} \tag{14.5.4}$$

There are occasions when (14.5.3) may be known to be a good model of the data. There may be other occasions when we are willing to take (14.5.3) as at least a rough-and-ready guess, since many two-dimensional distributions do resemble a binormal distribution, (that is, a two-dimensional Gaussian) at least not too far out on their tails. In either situation, we can use (14.5.3) to go beyond (14.5.2) in any of several directions:

First, we can allow for the possibility that the number N of data points is not large. Here, it turns out that the statistic

$$t = r\sqrt{\frac{N-2}{1-r^2}} \tag{14.5.5}$$

is distributed in the null case (of no correlation) like Student's t-distribution with $\nu = N - 2$ degrees of freedom, whose two-sided significance level is given by $1 - A(t|\nu)$ (equation 6.14.11) [1]. As N becomes large, this significance and (14.5.2) become asymptotically the same, so that one never does worse by using (14.5.5), even if the binormal assumption is not well substantiated.

Second, when N is only moderately large (≥ 10), we can compare whether the difference of two significantly nonzero r's, e.g., from different experiments, is itself significant. In other words, we can quantify whether a change in some control variable significantly alters an existing correlation between two other variables. This is done by using *Fisher's z-transformation* to associate each measured r with a corresponding z:

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right) \tag{14.5.6}$$

Then, each z is approximately normally distributed with a mean value

$$\overline{z} = \frac{1}{2}\left[\ln\left(\frac{1+r_{\text{true}}}{1-r_{\text{true}}}\right) + \frac{r_{\text{true}}}{N-1}\right] \tag{14.5.7}$$

where r_{true} is the actual or population value of the correlation coefficient, and with a standard deviation

$$\sigma(z) \approx \frac{1}{\sqrt{N-3}} \tag{14.5.8}$$

Equations (14.5.7) and (14.5.8), when they are valid, give several useful statistical tests [1]. For example, the significance level at which a measured value of r differs from some hypothesized value r_{true} is given by

$$\text{erfc}\left(\frac{|z-\overline{z}|\sqrt{N-3}}{\sqrt{2}}\right) \tag{14.5.9}$$

where z and $\overline{z}$ are given by (14.5.6) and (14.5.7), with small values of (14.5.9) indicating a significant difference. (Setting $\overline{z} = 0$ makes expression 14.5.9 a more accurate replacement for expression 14.5.2 above.) Similarly, the significance of a difference between two measured correlation coefficients r_1 and r_2 is

$$\text{erfc}\left(\frac{|z_1-z_2|}{\sqrt{2}\sqrt{\frac{1}{N_1-3}+\frac{1}{N_2-3}}}\right) \tag{14.5.10}$$

where z_1 and z_2 are obtained from r_1 and r_2 using (14.5.6), and where N_1 and N_2 are, respectively, the number of data points in the measurement of r_1 and r_2.

All of the significances above are two-sided. If you wish to disprove the null hypothesis in favor of a one-sided hypothesis, such as that $r_1 > r_2$ (where the sense of the inequality was decided *a priori*), then (i) if your measured r_1 and r_2 have the *wrong* sense, you have failed to demonstrate your one-sided hypothesis, but (ii) if they have the right ordering, you can multiply the significances given above by 0.5, which makes them more significant.

But keep in mind: These interpretations of the r statistic can be completely meaningless if the joint probability distribution of your variables x and y is too different from a binormal distribution.

```
void pearsn(VecDoub_I &x, VecDoub_I &y, Doub &r, Doub &prob, Doub &z)
```
stattests.h

Given two arrays `x[0..n-1]` and `y[0..n-1]`, this routine computes their correlation coefficient r (returned as `r`), the p-value at which the null hypothesis of zero correlation is disproved (`prob` whose small value indicates a significant correlation), and Fisher's z (returned as `z`), whose value can be used in further statistical tests as described above.

```
{
    const Doub TINY=1.0e-20;                 Will regularize the unusual case of
    Beta beta;                                  complete correlation.
    Int j,n=x.size();
    Doub yt,xt,t,df;
    Doub syy=0.0,sxy=0.0,sxx=0.0,ay=0.0,ax=0.0;
    for (j=0;j<n;j++) {                      Find the means.
        ax += x[j];
        ay += y[j];
    }
    ax /= n;
    ay /= n;
    for (j=0;j<n;j++) {                              Compute the correlation coefficient.
        xt=x[j]-ax;
        yt=y[j]-ay;
        sxx += xt*xt;
        syy += yt*yt;
        sxy += xt*yt;
    }
    r=sxy/(sqrt(sxx*syy)+TINY);
    z=0.5*log((1.0+r+TINY)/(1.0-r+TINY));           Fisher's z transformation.
    df=n-2;
    t=r*sqrt(df/((1.0-r+TINY)*(1.0+r+TINY)));       Equation (14.5.5).
    prob=beta.betai(0.5*df,0.5,df/(df+t*t));        Student's t probability.
    // prob=erfcc(abs(z*sqrt(n-1.0))/1.4142136);
    For large n, this easier computation of prob, using the short routine erfcc, would give
    approximately the same value.
}
```

CITED REFERENCES AND FURTHER READING:

Taylor, J.R. 1997, *An Introduction to Error Analysis*, 2nd ed. (Sausalito, CA: University Science Books), Chapter 9.

Mickey, R.M., Dunn, O.J., and Clark, V.A. 2004, *Applied Statistics: Analysis of Variance and Regression*, 3rd ed. (New York: Wiley).

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press), Chapter 12.

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley), Chapter 7.

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd rev. ed., reprinted 2000 (New York: Dover), §19.7.

Norusis, M.J. 2006, *SPSS 14.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

Stuart, A., and Ord, J.K. 1994, *Kendall's Advanced Theory of Statistics*, 6th ed. (London: Edward Arnold) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*], §16.28 and §16.33.[1]

14.6 Nonparametric or Rank Correlation

It is precisely the uncertainty in interpreting the significance of the linear correlation coefficient r that leads us to the important concepts of *nonparametric* or *rank correlation*. As before, we are given N pairs of measurements (x_i, y_i). Before, difficulties arose because we did not necessarily know the probability distribution function from which the x_i's or y_i's were drawn.

The key concept of nonparametric correlation is this: If we replace the value of each x_i by the value of its *rank* among all the other x_i's in the sample, that is, $1, 2, 3, \ldots, N$, then the resulting list of numbers will be drawn from a perfectly known distribution function, namely uniformly from the integers between 1 and N, inclusive. Better than uniformly, in fact, since if the x_i's are all distinct, then each integer will occur precisely once. If some of the x_i's have identical values, it is conventional to assign to all these "ties" the mean of the ranks that they would have had if their values had been slightly different. This *midrank* will sometimes be an integer, sometimes a half-integer. In all cases the sum of all assigned ranks will be the same as the sum of the integers from 1 to N, namely $\frac{1}{2}N(N+1)$.

Of course we do exactly the same procedure for the y_i's, replacing each value by its rank among the other y_i's in the sample.

Now we are free to invent statistics for detecting correlation between uniform sets of integers between 1 and N, keeping in mind the possibility of ties in the ranks. There is, of course, some loss of information in replacing the original numbers by ranks. We could construct some rather artificial examples where a correlation could be detected parametrically (e.g., in the linear correlation coefficient r) but could not be detected nonparametrically. Such examples are very rare in real life, however, and the slight loss of information in ranking is a small price to pay for a very major advantage: When a correlation is demonstrated to be present nonparametrically, then it is really there! (That is, to a certainty level that depends on the significance chosen.) Nonparametric correlation is more robust than linear correlation, more resistant to unplanned defects in the data, in the same sort of sense that the median is more robust than the mean. For more on the concept of robustness, see §15.7.

As always in statistics, some particular choices of a statistic have already been invented for us and consecrated, if not beatified, by popular use. We will discuss two, the *Spearman rank-order correlation coefficient* (r_s), and *Kendall's tau* (τ).

14.6.1 Spearman Rank-Order Correlation Coefficient

Let R_i be the rank of x_i among the other x's and S_i be the rank of y_i among the other y's, with ties being assigned the appropriate midrank as described above. Then the rank-order correlation coefficient is defined to be the linear correlation coefficient of the ranks, namely,

$$r_s = \frac{\sum_i (R_i - \overline{R})(S_i - \overline{S})}{\sqrt{\sum_i (R_i - \overline{R})^2}\sqrt{\sum_i (S_i - \overline{S})^2}} \tag{14.6.1}$$

The significance of a nonzero value of r_s is tested by computing

$$t = r_s\sqrt{\frac{N-2}{1-r_s^2}} \tag{14.6.2}$$

which is distributed approximately as Student's distribution with $N-2$ degrees of freedom. A key point is that this approximation does not depend on the original distribution of the x's and y's; it is always the same approximation, and always pretty good.

It turns out that r_s is closely related to another conventional measure of nonparametric correlation, the so-called *sum squared difference of ranks*, defined as

$$D = \sum_{i=0}^{N-1} (R_i - S_i)^2 \tag{14.6.3}$$

(This D is sometimes denoted D^{**}, where the asterisks are used to indicate that ties are treated by midranking.)

When there are no ties in the data, the exact relation between D and r_s is

$$r_s = 1 - \frac{6D}{N^3 - N} \tag{14.6.4}$$

When there are ties, the exact relation is slightly more complicated: Let f_k be the number of ties in the kth group of ties among the R_i's, and let g_m be the number of ties in the mth group of ties among the S_i's. Then it turns out that

$$r_s = \frac{1 - \frac{6}{N^3-N}\left[D + \frac{1}{12}\sum_k (f_k^3 - f_k) + \frac{1}{12}\sum_m (g_m^3 - g_m)\right]}{\left[1 - \frac{\sum_k (f_k^3 - f_k)}{N^3-N}\right]^{1/2}\left[1 - \frac{\sum_m (g_m^3 - g_m)}{N^3-N}\right]^{1/2}} \tag{14.6.5}$$

holds exactly. Notice that if all the f_k's and all the g_m's are equal to one, meaning that there are no ties, then equation (14.6.5) reduces to equation (14.6.4).

In (14.6.2) we gave a t-statistic that tests the significance of a nonzero r_s. It is also possible to test the significance of D directly. The expectation value of D in the null hypothesis of uncorrelated data sets is

$$\overline{D} = \frac{1}{6}(N^3 - N) - \frac{1}{12}\sum_k (f_k^3 - f_k) - \frac{1}{12}\sum_m (g_m^3 - g_m) \tag{14.6.6}$$

its variance is

$$\mathrm{Var}(D) = \frac{(N-1)N^2(N+1)^2}{36}\left[1 - \frac{\sum_k (f_k^3 - f_k)}{N^3-N}\right]\left[1 - \frac{\sum_m (g_m^3 - g_m)}{N^3-N}\right] \tag{14.6.7}$$

and it is approximately normally distributed, so that the significance level is a complementary error function (cf. equation 14.5.2). Of course, (14.6.2) and (14.6.7) are not independent tests, but simply variants of the same test. In the program that follows, we calculate both the significance level obtained by using (14.6.2) and the significance level obtained by using (14.6.7); their discrepancy will give you an idea of how good the approximations are. You will also notice that we break off the task of assigning ranks (including tied midranks) into a separate function, `crank`.

stattests.h

```
void spear(VecDoub_I &data1, VecDoub_I &data2, Doub &d, Doub &zd, Doub &probd,
    Doub &rs, Doub &probrs)
```

Given two data arrays, `data1[0..n-1]` and `data2[0..n-1]`, this routine returns their sum squared difference of ranks as D, the number of standard deviations by which D deviates from its null-hypothesis expected value as `zd`, the two-sided p-value of this deviation as `probd`, Spearman's rank correlation r_s as `rs`, and the two-sided p-value of its deviation from zero as `probrs`. The external routines `crank` (below) and `sort2` (§8.2) are used. A small value of either `probd` or `probrs` indicates a significant correlation (`rs` positive) or anticorrelation (`rs` negative).

```
{
    Beta bet;
    Int j,n=data1.size();
    Doub vard,t,sg,sf,fac,en3n,en,df,aved;
    VecDoub wksp1(n),wksp2(n);
    for (j=0;j<n;j++) {
        wksp1[j]=data1[j];
        wksp2[j]=data2[j];
    }
    sort2(wksp1,wksp2);                Sort each of the data arrays, and convert the en-
    crank(wksp1,sf);                       tries to ranks. The values sf and sg return
    sort2(wksp2,wksp1);                    the sums ∑(f_k^3 − f_k) and ∑(g_m^3 − g_m),
    crank(wksp2,sg);                       respectively.
    d=0.0;
    for (j=0;j<n;j++)                  Sum the squared difference of ranks.
```

```
        d += SQR(wksp1[j]-wksp2[j]);
    en=n;
    en3n=en*en*en-en;
    aved=en3n/6.0-(sf+sg)/12.0;                          Expectation value of D,
    fac=(1.0-sf/en3n)*(1.0-sg/en3n);
    vard=((en-1.0)*en*en*SQR(en+1.0)/36.0)*fac;          and variance of D give
    zd=(d-aved)/sqrt(vard);                              number of standard devia-
    probd=erfcc(abs(zd)/1.4142136);                         tions and p-value.
    rs=(1.0-(6.0/en3n)*(d+(sf+sg)/12.0))/sqrt(fac);      Rank correlation coefficient,
    fac=(rs+1.0)*(1.0-rs);
    if (fac > 0.0) {
        t=rs*sqrt((en-2.0)/fac);                         and its t-value,
        df=en-2.0;
        probrs=bet.betai(0.5*df,0.5,df/(df+t*t));        give its p-value.
    } else
        probrs=0.0;
}
```

stattests.h

```
void crank(VecDoub_IO &w, Doub &s)
```

Given a sorted array `w[0..n-1]`, replaces the elements by their rank, including midranking of ties, and returns as s the sum of $f^3 - f$, where f is the number of elements in each tie.

```
{
    Int j=1,ji,jt,n=w.size();
    Doub t,rank;
    s=0.0;
    while (j < n) {
        if (w[j] != w[j-1]) {                 Not a tie.
            w[j-1]=j;
            ++j;
        } else {                              A tie:
            for (jt=j+1;jt<=n && w[jt-1]==w[j-1];jt++);       How far does it go?
            rank=0.5*(j+jt-1);                This is the mean rank of the tie,
            for (ji=j;ji<=(jt-1);ji++)        so enter it into all the tied entries,
                w[ji-1]=rank;
            t=jt-j;
            s += (t*t*t-t);                   and update s.
            j=jt;
        }
    }
    if (j == n) w[n-1]=n;                     If the last element was not tied, this is its
}                                                rank.
```

14.6.2 Kendall's Tau

Kendall's τ is even more nonparametric than Spearman's r_s or D. Instead of using the numerical difference of ranks, it uses only the relative ordering of ranks: higher in rank, lower in rank, or the same in rank. But in that case we don't even have to rank the data! Ranks will be higher, lower, or the same if and only if the values are larger, smaller, or equal, respectively. On balance, we prefer r_s as being the more straightforward nonparametric test, but both statistics are in general use. In fact, τ and r_s are very strongly correlated and, in most applications, are effectively the same test.

To define τ, we start with the N data points (x_i, y_i). Now consider all $\frac{1}{2}N(N-1)$ *pairs* of data points, where a data point cannot be paired with itself, and where the points in either order count as one pair. We call a pair *concordant* if the relative ordering of the ranks of the two x's (or for that matter the two x's themselves) is the same as the relative ordering of the ranks of the two y's (or for that matter the two y's themselves). We call a pair *discordant* if the relative ordering of the ranks of the two x's is opposite from the relative ordering of the ranks of the two y's. If there is a tie in either the ranks of the two x's or the ranks of the two y's, then we don't call the pair either concordant or discordant. If the tie is in the x's, we will call the pair an "extra y pair." If the tie is in the y's, we will call the pair an "extra x pair." If the tie is in both the x's and the y's, we don't call the pair anything at all. Are you still with us?

Kendall's τ is now the following simple combination of these various counts:

$$\tau = \frac{\text{concordant} - \text{discordant}}{\sqrt{\text{concordant} + \text{discordant} + \text{extra-}y}\ \sqrt{\text{concordant} + \text{discordant} + \text{extra-}x}} \tag{14.6.8}$$

You can easily convince yourself that this must lie between 1 and -1, and that it takes on the extreme values only for complete rank agreement or complete rank reversal, respectively.

More important, Kendall has worked out, from the combinatorics, the approximate distribution of τ in the null hypothesis of no association between x and y. In this case, τ is approximately normally distributed, with zero expectation value and a variance of

$$\text{Var}(\tau) = \frac{4N + 10}{9N(N-1)} \tag{14.6.9}$$

The following program proceeds according to the above description, and therefore loops over all pairs of data points. Beware: This is an $O(N^2)$ algorithm, unlike the algorithm for r_s, whose dominant sort operations are of order $N \log N$. If you are routinely computing Kendall's τ for data sets of more than a few thousand points, you may be in for some serious computing. If, however, you are willing to bin your data into a moderate number of bins, then read on.

stattests.h

```
void kendl1(VecDoub_I &data1, VecDoub_I &data2, Doub &tau, Doub &z, Doub &prob)
```

Given data arrays `data1[0..n-1]` and `data2[0..n-1]`, this program returns Kendall's τ as `tau`, its number of standard deviations from zero as `z`, and its two-sided p-value as `prob`. Small values of `prob` indicate a significant correlation (`tau` positive) or anticorrelation (`tau` negative).

```
{
    Int is=0,j,k,n2=0,n1=0,n=data1.size();
    Doub svar,aa,a2,a1;
    for (j=0;j<n-1;j++) {                       Loop over first member of pair,
        for (k=j+1;k<n;k++) {                   and second member.
            a1=data1[j]-data1[k];
            a2=data2[j]-data2[k];
            aa=a1*a2;
            if (aa != 0.0) {                    Neither array has a tie.
                ++n1;
                ++n2;
                aa > 0.0 ? ++is : --is;
            } else {                            One or both arrays have ties.
                if (a1 != 0.0) ++n1;            An "extra x" event.
                if (a2 != 0.0) ++n2;            An "extra y" event.
            }
        }
    }
    tau=is/(sqrt(Doub(n1))*sqrt(Doub(n2)));     Equation (14.6.8).
    svar=(4.0*n+10.0)/(9.0*n*(n-1.0));          Equation (14.6.9).
    z=tau/sqrt(svar);
    prob=erfcc(abs(z)/1.4142136);               p-value.
}
```

Sometimes it happens that there are only a few possible values each for x and y. In that case, the data can be recorded as a contingency table (see §14.4) that gives the number of data points for each contingency of x and y.

Spearman's rank-order correlation coefficient is not a very natural statistic under these circumstances, since it assigns to each x and y bin a not-very-meaningful midrank value and then totals up vast numbers of identical rank differences. Kendall's tau, on the other hand, with its simple counting, remains quite natural. Furthermore, its $O(N^2)$ algorithm is no longer a problem, since we can arrange for it to loop over pairs of contingency table entries (each containing many data points) instead of over pairs of data points. This is implemented in the program that follows.

Note that Kendall's tau can be applied only to contingency tables where both variables are *ordinal*, i.e., well-ordered, and that it looks specifically for monotonic correlations, not for arbitrary associations. These two properties make it less general than the methods of §14.4, which applied to *nominal*, i.e., unordered, variables and arbitrary associations.

Comparing `kendl1` above with `kendl2` below, you will see that we have changed a number of variables from `int` to `double`. This is because the number of events in a contingency table might be sufficiently large as to cause overflows in some of the integer arithmetic, while the number of individual data points in a list could not possibly be that large (for an $O(N^2)$ routine!).

```
void kendl2(MatDoub_I &tab, Doub &tau, Doub &z, Doub &prob)                    stattests.h
```
Given a two-dimensional table `tab[0..i-1][0..j-1]`, such that `tab[k][l]` contains the number of events falling in bin `k` of one variable and bin `l` of another, this program returns Kendall's τ as `tau`, its number of standard deviations from zero as `z`, and its two-sided p-value as `prob`. Small values of `prob` indicate a significant correlation (`tau` positive) or anticorrelation (`tau` negative) between the two variables. Although `tab` is a `double` array, it will normally contain integral values.
```
{
    Int k,l,nn,mm,m2,m1,lj,li,kj,ki,i=tab.nrows(),j=tab.ncols();
    Doub svar,s=0.0,points,pairs,en2=0.0,en1=0.0;
    nn=i*j;                                Total number of entries in contingency table.
    points=tab[i-1][j-1];
    for (k=0;k<=nn-2;k++) {                Loop over entries in table,
        ki=(k/j);                          decoding a row,
        kj=k-j*ki;                         and a column.
        points += tab[ki][kj];             Increment the total count of events.
        for (l=k+1;l<=nn-1;l++) {          Loop over other member of the pair,
            li=l/j;                        decoding its row
            lj=l-j*li;                     and column.
            mm=(m1=li-ki)*(m2=lj-kj);
            pairs=tab[ki][kj]*tab[li][lj];
            if (mm != 0) {                 Not a tie.
                en1 += pairs;
                en2 += pairs;
                s += (mm > 0 ? pairs : -pairs);        Concordant, or discordant.
            } else {
                if (m1 != 0) en1 += pairs;
                if (m2 != 0) en2 += pairs;
            }
        }
    }
    tau=s/sqrt(en1*en2);
    svar=(4.0*points+10.0)/(9.0*points*(points-1.0));
    z=tau/sqrt(svar);
    prob=erfcc(abs(z)/1.4142136);
}
```

CITED REFERENCES AND FURTHER READING:

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press), Chapter 13.

Lehmann, E.L. 1975, *Nonparametrics: Statistical Methods Based on Ranks* (San Francisco: Holden-Day); reprinted 2006 (New York: Springer).

Hollander, M., and Wolfe, D.A. 1999, *Nonparametric Statistical Methods*, 2nd ed. (New York: Wiley).

Downie, N.M., and Heath, R.W. 1965, *Basic Statistical Methods*, 2nd ed. (New York: Harper & Row), pp. 206–209.

Norusis, M.J. 2006, *SPSS 14.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

14.7 Information-Theoretic Properties of Distributions

In this section we return to nominal distributions, that is to say, to distributions with discrete outcomes that have no meaningful ordering. Information theory [1-3] provides a different, and sometimes very useful, perspective on the nature of such a distribution $\mathbf{p}$ with outcomes i, $0 \le i \le I-1$, and associated probabilities p_i, and on the relation between two or more such distributions. We develop that perspective in this section, starting with a review of some key concepts.

14.7.1 Entropy of a Distribution

Suppose that we make M sequential, independent draws from a distribution $\mathbf{p}$, thus generating a *message* that describes the outcomes, an M-vector of integers i_j, each in the range $0 \le i_j \le I-1$, with $j = 0, \ldots, M-1$. We want to send the message to a waiting confederate, but we first want to compress it (that is, *encode* it) into the most parsimonious format, say into the smallest possible number of bits, B. We can calculate a lower bound on B by equating 2^B, the number of possible different compressed messages, to a statistical estimate of the number of likely input messages. That equation, in the limit of M becoming very large, is

$$2^B \approx \frac{M!}{\prod_i (Mp_i)!} \tag{14.7.1}$$

The rationale for the right-hand side is that our message will contain very nearly Mp_i occurrences of the integer i for each i, so the count of messages will be very nearly the number of ways that we can arrange M objects of I types, with Mp_i of them identical for each type i. Taking the logarithm of equation (14.7.1), using Stirling's approximation on the factorials, and keeping only terms that scale as fast as M, we readily obtain

$$B \approx -M \sum_{i=0}^{I-1} p_i \log_2 p_i \equiv M\, H_2(\mathbf{p}) \tag{14.7.2}$$

where $H_2(\mathbf{p})$ is called the *entropy (in bits)* of the distribution $\mathbf{p}$, a terminology borrowed from statistical physics. The subscript 2 is to remind us that the logarithm has base 2. We can also define an entropy with base e,

$$H(\mathbf{p}) \equiv -\sum_{i=0}^{I-1} p_i \ln p_i = -(\ln 2) \sum_{i=0}^{I-1} p_i \log_2 p_i = (\ln 2)\, H_2(\mathbf{p}) \tag{14.7.3}$$

If $H_2(\mathbf{p})$ is measured in *bits*, then $H(\mathbf{p})$ will be measured in *nats*, with 1 nat = 1.4427 bits. In evaluating (14.7.3), note that

$$\lim_{p \to 0} p \ln p = 0 \tag{14.7.4}$$

The value $H(\mathbf{p})$ lies between 0 and $\ln I$. It is zero only when one of the p_i's is one, all the others zero.

Although we derived B as a lower bound, a central result of information theory is that, in the limit of large M, one can find codes that actually achieve that bound. (Arithmetic coding, described in §22.6, is an example of such a code.) Heuristically, one can interpret equation (14.7.2) as saying that it takes, on average, $-\log_2 p_i$ bits (a positive number, since $p_i < 1$) to encode an outcome i. Thus, the compressed message size is M times the expectation of $-\log_2 p_i$ over outcomes occurring with probability p_i.

Yet a different view of entropy arises if we consider the game of "twenty questions," where by repeated yes/no questions you try to eliminate all except one correct possibility for an unknown object. Better yet, let us consider a generalization of the game, where you are allowed to ask multiple choice questions as well as binary (yes/no) ones. The categories in your multiple choice questions are supposed to be mutually exclusive and exhaustive (as are "yes" and "no").

The value to you of an answer increases with the number of possibilities that it eliminates. More specifically, an answer that eliminates all except a fraction p of the remaining possibilities can be assigned a value $-\ln p$. The purpose of the logarithm is to make the value additive, since, e.g., one question that eliminates all but 1/6 of the possibilities is considered as good as two questions that, in sequence, reduce the number by factors 1/2 and 1/3.

So that is the value of an answer; but what is the value of a question? If there are I possible answers to the question and the fraction of possibilities consistent with answer i is p_i, then the value of the question is the expectation value of the value of the answer, which is just $-\sum_i p_i \ln p_i$ or $H(\mathbf{p})$, as above.

As already mentioned, the entropy is zero only if one of the p_i's is unity, with all the others zero. In this case, the question is valueless, since its answer is preordained. $H(\mathbf{p})$ takes on its maximum value when all the p_i's are equal, in which case the question is sure to eliminate all but a fraction $1/I$ of the remaining possibilities.

A third, still different, view of entropy comes from thinking about bets (or, more politely, "investments"). A *fair bet* on an outcome i of probability p_i is one that has a payoff $o_i = 1/p_i$. This is the unique payoff (per unit wagered) for which, in the long run, the bettor will neither win nor lose, since in expectation value

$$\langle o_i \rangle = p_i o_i = 1 \tag{14.7.5}$$

Suppose you have the opportunity to bet repeatedly on a game offering fair bets on each outcome. This is not very interesting as a money-making proposition. But suppose that you are *clairvoyant* and can know in advance the outcome of each play (although you cannot affect that outcome). Now you're in business! You always put your money on the winning choice of i. How much money can you make?

Since your profit on each (sure thing!) wager scales multiplicatively with your accumulated wealth, the appropriate figure of merit is the the average *doubling rate*, or, equivalently, *e-folding rate*, at which you can increase your capital. Since you always win, but can't control the outcome, this is given by

$$W \equiv \langle \ln o_i \rangle = \langle -\ln p_i \rangle = -\sum_i p_i \ln p_i = H(\mathbf{p}) \qquad (14.7.6)$$

In other words, the entropy of a distribution is the e-folding rate of capital for a fair game about which you have perfect predictive information. While this may seem fanciful, we will see in §14.7.3 how it generalizes to the more realistic case where you have only imperfect, perhaps very small, predictive information.

14.7.2 Kullback-Leibler Distance

Back in the context of message compression, suppose that events occur with a distribution $\mathbf{p}$, that is, p_i, $0 \le i \le I-1$, but we try to compress the message of their outcomes with a code that is optimized for some other distribution $\mathbf{q}$, that is, q_i, $0 \le i \le I-1$. Our code therefore takes about $-\log_2 q_i$ bits, or $-\ln q_i$ nats, to encode outcome i, and the average compressed length per outcome is

$$-\sum_i p_i \ln q_i = H(\mathbf{p}) + \sum_i p_i \ln \frac{p_i}{q_i} \equiv H(\mathbf{p}) + D(\mathbf{p}\|\mathbf{q}) \qquad (14.7.7)$$

The quantity

$$D(\mathbf{p}\|\mathbf{q}) \equiv \sum_i p_i \ln \frac{p_i}{q_i} \qquad (14.7.8)$$

is called the *Kullback-Leibler distance* between $\mathbf{p}$ and $\mathbf{q}$, also called the *relative entropy* between the two distributions. We can easily prove that it is nonnegative, since

$$-D(\mathbf{p}\|\mathbf{q}) = \sum_i p_i \ln\left(\frac{q_i}{p_i}\right) \le \sum_i p_i \left(\frac{q_i}{p_i} - 1\right) = 1 - 1 = 0 \qquad (14.7.9)$$

where the inequality follows from the fact that

$$\ln w \le w - 1 \qquad (14.7.10)$$

(Of course we already knew it had to be nonnegative, because we knew that $H(\mathbf{p})$ was the *smallest* possible compressed message size for the distribution $\mathbf{p}$.) The Kullback-Leibler distance between two distributions is zero only when the two distributions are identical. The Kullback-Leibler distance between any distribution $\mathbf{p}$ and the uniform distribution $\mathbf{U}$ is just the difference between the entropy of $\mathbf{p}$ and the maximum possible entropy $\ln I$, that is,

$$H(\mathbf{p}) + D(\mathbf{p}\|\mathbf{U}) = \ln I \qquad (14.7.11)$$

This is illustrated in Figure 14.7.1. Just like entropy, the Kullback-Leibler distance is measured in bits or nats, depending on whether the logarithms are taken base 2 or e, respectively.

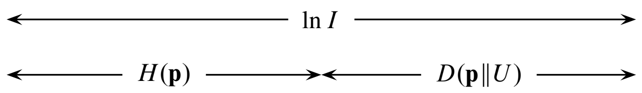


Figure 14.7.1. Relation between the entropy of a distribution $\mathbf{p}$, its Kullback-Leibler distance to the uniform distribution $\mathbf{U}$, and its maximum possible entropy $\ln I$.

Notice that the Kullback-Leibler distance is not symmetric, nor (it turns out) does it satisfy the triangle inequality. So it is not a true metric distance. It is, however, a useful measure of the degree by which some "target" distribution $\mathbf{q}$ differs from some "base" distribution $\mathbf{p}$. We now give a couple of examples of where it naturally occurs.

Example 1. Suppose that we are seeing events drawn from the distribution $\mathbf{p}$, but we want to rule out an alternative hypothesis that they are drawn from $\mathbf{q}$. We might do this by computing a likelihood ratio,

$$\mathcal{L} = \frac{p(\text{Data}|\mathbf{p})}{p(\text{Data}|\mathbf{q})} = \prod_{\text{data}} \frac{p_i}{q_i} \tag{14.7.12}$$

and rejecting the alternative hypothesis $\mathbf{q}$ if this ratio is larger than some large number, say 10^6. (In the above shorthand notation, the product over "data" means that we substitute for i in each factor the particular outcome of that factor's individual data event.) Taking the logarithm of equation (14.7.12), you can easily see that, under hypothesis $\mathbf{p}$, the average increase in $\ln \mathcal{L}$ per data event is just $D(\mathbf{p}\|\mathbf{q})$. In other words, the Kullback-Leibler distance is the expected log-likelihood with which a false hypothesis $\mathbf{q}$ can be rejected, per event. As we might expect, this has something to do with "how different" $\mathbf{q}$ is from $\mathbf{p}$.

As a Bayesian aside, the reason that the above likelihood test is unsatisfyingly asymmetric is that, without the notion of a prior, we have no way to treat hypotheses $\mathbf{p}$ and $\mathbf{q}$ democratically. But suppose that $p(\mathbf{p})$ is the prior probability of $\mathbf{p}$, so that $p(\mathbf{q}) = 1 - p(\mathbf{p})$ is the prior for $\mathbf{q}$. Then the Bayes odds ratio on the two hypotheses is

$$\text{O.R.} = \frac{p(\mathbf{p}|\text{Data})}{p(\mathbf{q}|\text{Data})} = \frac{p(\text{Data}|\mathbf{p})\, p(\mathbf{p})}{p(\text{Data}|\mathbf{q})\, p(\mathbf{q})} = \frac{p(\mathbf{p})}{p(\mathbf{q})} \prod_{\text{data}} \frac{p_i}{q_i} \tag{14.7.13}$$

The figure of merit is now the expected increase in ln(O.R.) if $\mathbf{p}$ is true, *minus* the expected increase (that is, *plus* the expected decrease) if $\mathbf{q}$ is true, which can readily be seen to be

$$p(\mathbf{p})\, D(\mathbf{p}\|\mathbf{q}) + p(\mathbf{q})\, D(\mathbf{q}\|\mathbf{p}) \tag{14.7.14}$$

per data event, which has the appropriate symmetry. We can use expression (14.7.14) to estimate how many data events we will need on average to distinguish between two distributions. Notice that in the case of a uniform ("noninformative") prior, $p(\mathbf{p}) = p(\mathbf{q}) = 0.5$, we get just the symmetrized average of the two Kullback-Leibler distances.

Example 2. Meanwhile, back at the racetrack where we are offered payoffs of o_i on events with probability p_i, $\sum_i p_i = 1$, we want to work out the best way to divide our capital across all the possible outcomes i of each race. Suppose we bet a fraction b_i on outcome i. Analgously to equation (14.7.6), we want to maximize the average e-folding rate,

$$W = \langle \ln(b_i o_i) \rangle = \sum_i p_i \ln(b_i o_i) \tag{14.7.15}$$

subject to the constraint

$$\sum_i b_i = 1 \tag{14.7.16}$$

An easy calculation (using a Lagrange multiplier to impose the constraint) gives the result that the maximum occurs for

$$b_i = p_i \tag{14.7.17}$$

completely independent of the values o_i! This remarkable result is called *proportional betting*, or sometimes *Kelly's formula* [4].

In practice, the distribution $\mathbf{p}$ is imperfectly known, both to you and to the bookie at the track. Suppose that you estimate the outcome probabilities as $\mathbf{q}$, while the bookie's estimate is $\mathbf{r}$. If the bookie is feeling generous, he offers payoffs that are fair bets according to his estimate,

$$o_i = 1/r_i \tag{14.7.18}$$

while you place proportional bets with $b_i = q_i$. Your e-folding rate is now

$$W = \langle \ln(b_i o_i) \rangle = \sum_i p_i \ln \frac{q_i}{r_i} = D(\mathbf{p}\|\mathbf{r}) - D(\mathbf{p}\|\mathbf{q}) \tag{14.7.19}$$

This will be positive if and only if your estimate of the probabilities is better than the bookie's, that is, closer as measured by the Kullback-Leibler distance. Betting, in other words, is a competition between you and the bookie over who can better estimate the true odds.

A more realistic variant is to assume that the bookie offers payoffs of only some fraction $f < 1$ of his reciprocal probability estimates. Then (you can work out), you can win only if

$$D(\mathbf{p}\|\mathbf{r}) - D(\mathbf{p}\|\mathbf{q}) > -\ln f \tag{14.7.20}$$

14.7.3 Conditional Entropy and Mutual Information

We now want to look at the association of two variables. Let us return to the guessing game that was discussed in §14.7.1. Suppose we are deciding what question to ask next in the game and have to choose between two candidates, or possibly want to ask both in one order or another. Suppose that one question, x, has I possible answers, labeled by i, and that the other question, y, has J possible answers, labeled by j. Then the possible outcomes of asking both questions form a contingency table whose entries are the joint outcome probabilities p_{ij}, normalized by

$$\sum_{i=0}^{I-1} \sum_{j=0}^{J-1} p_{ij} \equiv \sum_{i,j} p_{ij} = 1 \tag{14.7.21}$$

We use the same "dot" notation as in §14.4 to denote the row and column sums, so that $p_{i\cdot}$ is the probability of outcome i asking question x only, while $p_{\cdot j}$ is the probability of outcome j asking question y only. The entropies of the questions x and y are thus, respectively,

$$H(x) = -\sum_i p_{i\cdot} \ln p_{i\cdot} \qquad H(y) = -\sum_j p_{\cdot j} \ln p_{\cdot j} \tag{14.7.22}$$

The entropy of the two questions together is

$$H(x,y) = -\sum_{i,j} p_{ij} \ln p_{ij} \tag{14.7.23}$$

Now what is the entropy of the question y *given* x (that is, if x is asked first)? It is the expectation value over the answers to x of the entropy of the restricted y distribution that lies in a single column of the contingency table (corresponding to the x answer):

$$H(y|x) = -\sum_i p_{i\cdot} \sum_j \frac{p_{ij}}{p_{i\cdot}} \ln \frac{p_{ij}}{p_{i\cdot}} = -\sum_{i,j} p_{ij} \ln \frac{p_{ij}}{p_{i\cdot}} \tag{14.7.24}$$

Correspondingly, the entropy of x given y is

$$H(x|y) = -\sum_j p_{\cdot j} \sum_i \frac{p_{ij}}{p_{\cdot j}} \ln \frac{p_{ij}}{p_{\cdot j}} = -\sum_{i,j} p_{ij} \ln \frac{p_{ij}}{p_{\cdot j}} \tag{14.7.25}$$

We can readily prove that the entropy of y given x is never more than the entropy of y alone, i.e., that asking x first can only reduce the usefulness of asking y (in which case the two variables are *associated*):

$$\begin{aligned} H(y|x) - H(y) &= -\sum_{i,j} p_{ij} \ln \frac{p_{ij}/p_{i\cdot}}{p_{\cdot j}} \\ &= \sum_{i,j} p_{ij} \ln \frac{p_{\cdot j}\, p_{i\cdot}}{p_{ij}} \\ &\le \sum_{i,j} p_{ij} \left(\frac{p_{\cdot j}\, p_{i\cdot}}{p_{ij}} - 1 \right) \\ &= \sum_{i,j} p_{i\cdot} p_{\cdot j} - \sum_{i,j} p_{ij} \\ &= 1 - 1 = 0 \end{aligned} \tag{14.7.26}$$

Quantities like $H(x|y)$ or $H(y|x)$ are called *conditional entropies*. You can easily show that

$$H(x,y) = H(x) + H(y|x) = H(y) + H(x|y) \tag{14.7.27}$$

sometimes called the *chain rule for entropies*. It immediately follows that

$$H(x) - H(x|y) = H(y) - H(y|x) \equiv I(x,y) \tag{14.7.28}$$

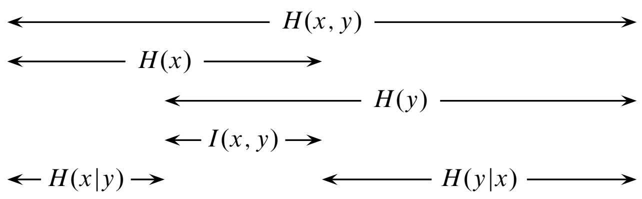


Figure 14.7.2. Relations among the entropies, conditional entropies, and mutual information of two variables. The quantities shown as segment lengths are always positive.

a quantity called the *mutual information* between x and y, given explicitly by

$$I(x, y) = \sum_{i,j} p_{ij} \ln \left(\frac{p_{ij}}{p_{i\cdot} p_{\cdot j}} \right) \tag{14.7.29}$$

Notice that the mutual information is symmetrical, $I(x, y) = I(y, x)$.

Figure 14.7.2 provides a handy way to visualize the additive relations and inequalities among the quantities discussed. As before, all the quantities are measured in bits or nats. Using mutual information, one can make statements like this about the degree of association of two variables: "The variables have information (entropy) 6.5 and 4.2 bits, respectively. However, their mutual information is 3.8 bits, so together they provide only 6.9 bits of information."

As a more detailed example, let us go back to the racetrack one last time. Suppose that you have some *side information* relevant to the outcome, but not completely predictive. That is, x is the random variable of which outcome i wins, while y is a random variable whose value j you know. Instead of a simple set of probabilities p_i, we now have a contingency table of joint outcomes, p_{ij}. How should you bet, and what is your expected e-folding rate?

First, we need to generalize equation (14.7.17). Suppose that b_{ij} is the fraction of assets that we bet on outcome i when our side information has the value j. There are now J separate constraints,

$$\sum_i b_{ij} = 1, \qquad 0 \le j \le J - 1 \tag{14.7.30}$$

For simplicity, let us take the case where the payoffs are for fair bets (but without the side information), $o_i = 1/p_{i\cdot}$. Then we want to maximize

$$W = \left\langle \ln \frac{b_{ij}}{p_{i\cdot}} \right\rangle = \sum_{i,j} p_{ij} \ln \frac{b_{ij}}{p_{i\cdot}} \tag{14.7.31}$$

A simple calculation, now with J distinct Lagrange multipliers, gives the result,

$$b_{ij} = \frac{p_{ij}}{p_{\cdot j}} \tag{14.7.32}$$

This is again proportional betting, except that it is now conditioned on the value j that is known to us. Substituting equation (14.7.32) into (14.7.31) gives

$$W = \sum_{i,j} p_{ij} \ln \left(\frac{p_{ij}}{p_{i\cdot} p_{\cdot j}} \right) = I(x, y) \tag{14.7.33}$$

We see that the expected e-folding rate is exactly the mutual information between x and y. In other words, we can make money if and only if our side information y has nonzero mutual information with the outcome x. As in equation (14.7.20), you can easily work out more realistic cases where the payouts are not fair bets, or are based on inexact estimates of the true probabilities. A special case of equation (14.7.33) is when the side information y predicts the outcome x *perfectly*. Then, $I(x,y) = H(x) = H(y) = H(x,y)$ and we recover exactly equation (14.7.6).

14.7.4 Uncertainty Coefficients

By analogy with the various coefficients of correlation discussed earlier in this chapter, one sometimes sees *uncertainty coefficients* defined from the various entropies defined above (and in Figure 14.7.2). The uncertainty coefficient of y with respect to x, denoted $U(y|x)$, is defined by

$$U(y|x) \equiv \frac{H(y) - H(y|x)}{H(y)} \tag{14.7.34}$$

This measure lies between 0 and 1, with the value 0 indicating that x and y have no association and the value 1 indicating that knowledge of x completely predicts y. For in-between values, $U(y|x)$ gives the fraction of y's entropy $H(y)$ that is lost if x is already known. In our game of "twenty questions," $U(y|x)$ is the fractional loss in the utility of question y if question x is to be asked first.

If we wish to view x as the dependent variable and y as the independent one, then interchanging x and y we can of course define the dependency of x on y,

$$U(x|y) \equiv \frac{H(x) - H(x|y)}{H(x)} \tag{14.7.35}$$

If we want to treat x and y symmetrically, then the useful combination turns out to be

$$U(x,y) \equiv 2\left[\frac{H(y) + H(x) - H(x,y)}{H(x) + H(y)}\right] \tag{14.7.36}$$

If the two variables are completely independent, then $H(x,y) = H(x) + H(y)$, so (14.7.36) vanishes. If the two variables are completely dependent, then $H(x) = H(y) = H(x,y)$, so (14.7.35) equals unity. You can easily show that

$$U(x,y) = \frac{H(x)U(x|y) + H(y)U(y|x)}{H(x) + H(y)} \tag{14.7.37}$$

that is, that the symmetrical measure is just a weighted average of the two asymmetrical measures (14.7.34) and (14.7.35), weighted by the entropy of each variable separately.

Generally we find the entropy measures themselves, in bits or nats, more useful than the uncertainty coefficients derived from them.

CITED REFERENCES AND FURTHER READING:

Shannon, C.E., and Weaver, W. 1949, *The Mathematical Theory of Communication*, reprinted 1998 (Urbana, IL: University of Illinois Press).[1]

Cover, T.M., and Thomas, J.A. 1991, *Elements of Information Theory* (New York: Wiley). [2]

MacKay, D.J.C. 2003, *Information Theory, Inference, and Learning Algorithms* (Cambridge, UK: Cambridge University Press). [3]

Kelly, J. 1956, "A New Interpretation of Information Rate," *Bell System Technical Journal*, vol. 35, pp. 917–926. [4]

14.8 Do Two-Dimensional Distributions Differ?

We here discuss a useful generalization of the K–S test (§14.3) to *two-dimensional* distributions. This generalization is due to Fasano and Franceschini [1], a variant on an earlier idea due to Peacock [2].

In a two-dimensional distribution, each data point is characterized by an (x, y) pair of values. An example near to our hearts is that each of the 19 neutrinos that were detected from Supernova 1987A is characterized by a time t_i and by an energy E_i (see [3]). We might wish to know whether these measured pairs (t_i, E_i), $i = 0 \dots 18$ are consistent with a theoretical model that predicts neutrino flux as a function of both time and energy — that is, a two-dimensional probability distribution in the (x, y) [here, (t, E)] plane. That would be a one-sample test. Or, given two sets of neutrino detections, from two comparable detectors, we might want to know whether they are compatible with each other, a two-sample test.

In the spirit of the tried-and-true one-dimensional K–S test, we want to range over the (x, y)-plane in search of some kind of maximum *cumulative* difference between two two-dimensional distributions. Unfortunately, cumulative probability distribution is not well-defined in more than one dimension! Peacock's insight was that a good surrogate is the *integrated probability in each of four natural quadrants* around a given point (x_i, y_i), namely the total probabilities (or fraction of data) in $(x > x_i, y > y_i)$, $(x < x_i, y > y_i)$, $(x < x_i, y < y_i)$, $(x > x_i, y < y_i)$. The two-dimensional K–S statistic D is now taken to be the maximum difference (ranging both over data points and over quadrants) of the corresponding integrated probabilities. When comparing two data sets, the value of D may depend on which data set is ranged over. In that case, define an effective D as the average of the two values obtained. If you are confused at this point about the exact definition of D, don't fret; the accompanying computer routines amount to a precise algorithmic definition.

Figure 14.8.1 gives a feeling for what is going on. The 65 triangles and 35 squares seem to have somewhat different distributions in the plane. The dotted lines are centered on the triangle that maximizes the D statistic; the maximum occurs in the upper-left quadrant. That quadrant contains only 0.12 of all the triangles, but it contains 0.56 of all the squares. The value of D is thus 0.44. Is this statistically significant?

Even for fixed sample sizes, it is unfortunately not rigorously true that the distribution of D in the null hypothesis is independent of the shape of the two-dimensional distribution. In this respect the two-dimensional K–S test is not as natural as its one-dimensional parent. However, extensive Monte Carlo integrations have shown that the distribution of the two-dimensional D is *very nearly* identical for even quite different distributions, as long as they have the same coefficient of correlation r, defined in the usual way by equation (14.5.1). In their paper, Fasano and Franceschini tabulate Monte Carlo results for (what amounts to) the distribution of D as a function

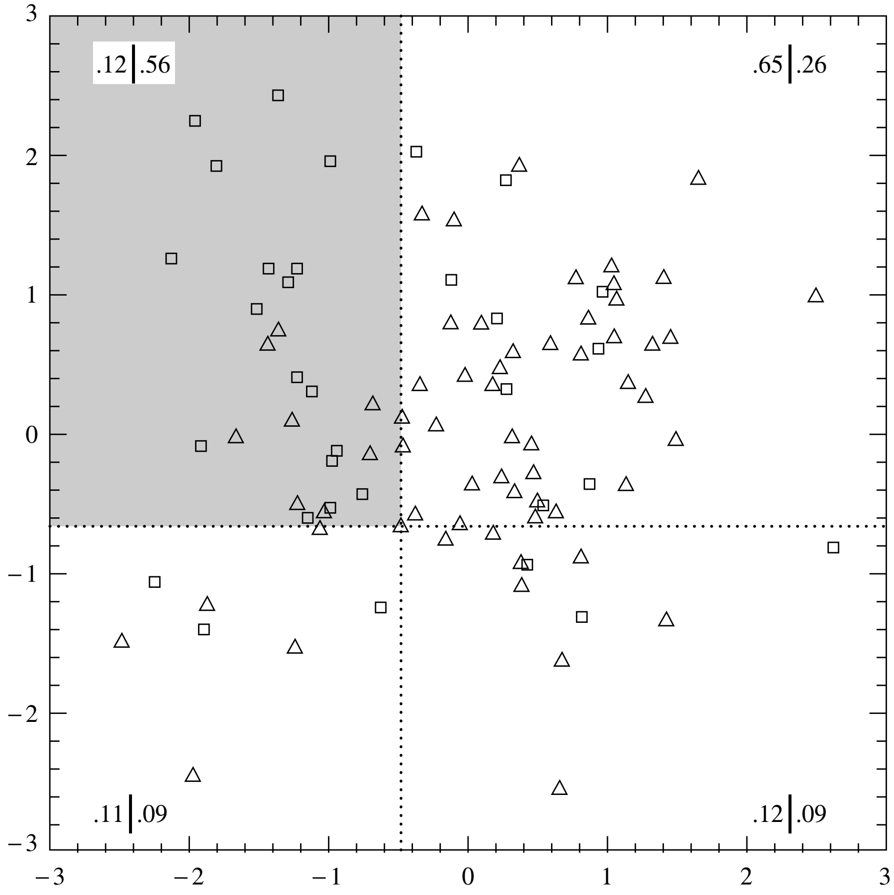


Figure 14.8.1. Two-dimensional distributions of 65 triangles and 35 squares. The two-dimensional K–S test finds that point one of whose quadrants (shown by dotted lines) maximizes the difference between fraction of triangles and fraction of squares. Then, equation (14.8.1) indicates whether the difference is statistically significant, i.e., whether the triangles and squares must have different underlying distributions.

of (of course) D, sample size N, and coefficient of correlation r. Analyzing their results, one finds that the significance levels for the two-dimensional K–S test can be summarized by the simple, though approximate, formulas

$$\text{Probability}\ (D > \text{observed}\ ) = Q_{KS}\left(\frac{\sqrt{N}\,D}{1+\sqrt{1-r^2}(0.25-0.75/\sqrt{N})}\right) \tag{14.8.1}$$

for the one-sample case, and the same for the two-sample case, but with

$$N = \frac{N_1 N_2}{N_1 + N_2}. \tag{14.8.2}$$

The above formulas are accurate enough when $N \gtrsim 20$, and when the indicated probability (significance level) is less than (more significant than) 0.20 or so. When the indicated probability is > 0.20, its value may not be accurate, but the implication that the data and model (or two data sets) are not significantly different is certainly correct. Notice that in the limit of $r \to 1$ (perfect correlation), equations (14.8.1) and (14.8.2) reduce to equations (14.3.18) and (14.3.19): The two-dimensional data lie on a perfect straight line, and the two-dimensional K–S test becomes a one-dimensional K–S test.

The significance level for the data in Figure 14.8.1, by the way, is about 0.001. This establishes to a near-certainty that the triangles and squares were drawn from different distributions. (As in fact they were.)

Of course, if you do not want to rely on the Monte Carlo experiments embodied in equation (14.8.1), you can do your own: Generate a lot of synthetic data sets from your model, each one with the same number of points as the real data set. Compute D for each synthetic data set, using the accompanying computer routines (but ignoring their calculated probabilities), and count what fraction of the time these synthetic D's exceed the D from the real data. That fraction is your significance.

One disadvantage of the two-dimensional tests, by comparison with their one-dimensional progenitors, is that the two-dimensional tests require of order N^2 operations: Two nested loops of order N take the place of an $N \log N$ sort. For desktop computers, this restricts the usefulness of the tests to N less than several thousand.

We now give computer implementations. The one-sample case is embodied in the routine `ks2d1s` (that is, two dimensions, one sample). This routine calls a straightforward utility routine `quadct` to count points in the four quadrants, and it calls a user-supplied routine `quadvl` that must be capable of returning the integrated probability of an analytic model in each of four quadrants around an arbitrary (x, y) point. A trivial sample `quadvl` is shown; realistic `quadvl`s can be quite complicated, often incorporating numerical quadratures over analytic two-dimensional distributions.

kstests_2d.h

```
void ks2d1s(VecDoub_I &x1, VecDoub_I &y1, void quadvl(const Doub, const Doub,
    Doub &, Doub &, Doub &, Doub &), Doub &d1, Doub &prob)
```

Two-dimensional Kolmogorov-Smirnov test of one sample against a model. Given the x and y coordinates of `n1` data points in arrays `x1[0..n1-1]` and `y1[0..n1-1]`, and given a user-supplied function `quadvl` that exemplifies the model, this routine returns the two-dimensional K-S statistic as `d1`, and its p-value as `prob`. Small values of `prob` show that the sample is significantly different from the model. Note that the test is slightly distribution-dependent, so `prob` is only an estimate.

```
{
    Int j,n1=x1.size();
    Doub dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen;
    KSdist ks;
    d1=0.0;
    for (j=0;j<n1;j++) {                          Loop over the data points.
        quadct(x1[j],y1[j],x1,y1,fa,fb,fc,fd);
        quadvl(x1[j],y1[j],ga,gb,gc,gd);
        if (fa > ga) fa += 1.0/n1;
        if (fb > gb) fb += 1.0/n1;
        if (fc > gc) fc += 1.0/n1;
        if (fd > gd) fd += 1.0/n1;
        d1=MAX(d1,abs(fa-ga));
        d1=MAX(d1,abs(fb-gb));
        d1=MAX(d1,abs(fc-gc));
        d1=MAX(d1,abs(fd-gd));
        For both the sample and the model, the distribution is integrated in each of four
        quadrants, and the maximum difference is saved.
    }
    pearsn(x1,y1,r1,dum,dumm);                    Get the linear correlation coefficient r1.
    sqen=sqrt(Doub(n1));
    rr=sqrt(1.0-r1*r1);
    Estimate the probability using the K-S probability function.
    prob=ks.qks(d1*sqen/(1.0+rr*(0.25-0.75/sqen)));
}
```

kstests_2d.h

```
void quadct(const Doub x, const Doub y, VecDoub_I &xx, VecDoub_I &yy, Doub &fa,
    Doub &fb, Doub &fc, Doub &fd)
```

Given an origin (x, y), and an array of nn points with coordinates xx[0..nn-1] and yy[0..nn-1], count how many of them are in each quadrant around the origin, and return the normalized fractions. Quadrants are labeled alphabetically, counterclockwise from the upper right. Used by ks2d1s and ks2d2s.

```
{
    Int k,na,nb,nc,nd,nn=xx.size();
    Doub ff;
    na=nb=nc=nd=0;
    for (k=0;k<nn;k++) {
        if (yy[k] == y && xx[k] == x) continue;
        if (yy[k] > y)
            xx[k] > x ? ++na : ++nb;
        else
            xx[k] > x ? ++nd : ++nc;
    }
    ff=1.0/nn;
    fa=ff*na;
    fb=ff*nb;
    fc=ff*nc;
    fd=ff*nd;
}
```

quadvl.h

```
void quadvl(const Doub x, const Doub y, Doub &fa, Doub &fb, Doub &fc, Doub &fd)
```

This is a sample of a user-supplied routine to be used with ks2d1s. In this case, the model distribution is uniform inside the square $-1 < x < 1$, $-1 < y < 1$. In general, this routine should return, for any point (x, y), the fraction of the total distribution in each of the four quadrants around that point. The fractions, fa, fb, fc, and fd, must add up to 1. Quadrants are alphabetical, counterclockwise from the upper right.

```
{
    Doub qa,qb,qc,qd;
    qa=MIN(2.0,MAX(0.0,1.0-x));
    qb=MIN(2.0,MAX(0.0,1.0-y));
    qc=MIN(2.0,MAX(0.0,x+1.0));
    qd=MIN(2.0,MAX(0.0,y+1.0));
    fa=0.25*qa*qb;
    fb=0.25*qb*qc;
    fc=0.25*qc*qd;
    fd=0.25*qd*qa;
}
```

The routine ks2d2s is the two-sample case of the two-dimensional K–S test. It also calls quadct, pearsn, and KSdist::qks. Being a two-sample test, it does not need an analytic model.

kstests_2d.h

```
void ks2d2s(VecDoub_I &x1, VecDoub_I &y1, VecDoub_I &x2, VecDoub_I &y2, Doub &d,
    Doub &prob)
```

Two-dimensional Kolmogorov-Smirnov test on two samples. Given the x and y coordinates of the first sample as n1 values in arrays x1[0..n1-1] and y1[0..n1-1], and likewise for the second sample, n2 values in arrays x2 and y2, this routine returns the two-dimensional, two-sample K-S statistic as d, and its p-value as prob. Small values of prob show that the two samples are significantly different. Note that the test is slightly distribution-dependent, so prob is only an estimate.

```
{
    Int j,n1=x1.size(),n2=x2.size();
    Doub d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,sqen;
    KSdist ks;
    d1=0.0;
    for (j=0;j<n1;j++) {                        First, use points in the first sample as origins.
        quadct(x1[j],y1[j],x1,y1,fa,fb,fc,fd);
        quadct(x1[j],y1[j],x2,y2,ga,gb,gc,gd);
        if (fa > ga) fa += 1.0/n1;
        if (fb > gb) fb += 1.0/n1;
        if (fc > gc) fc += 1.0/n1;
        if (fd > gd) fd += 1.0/n1;
        d1=MAX(d1,abs(fa-ga));
        d1=MAX(d1,abs(fb-gb));
        d1=MAX(d1,abs(fc-gc));
        d1=MAX(d1,abs(fd-gd));
    }
    d2=0.0;
    for (j=0;j<n2;j++) {                        Then, use points in the second sample as
        quadct(x2[j],y2[j],x1,y1,fa,fb,fc,fd);      origins.
        quadct(x2[j],y2[j],x2,y2,ga,gb,gc,gd);
        if (ga > fa) ga += 1.0/n1;
        if (gb > fb) gb += 1.0/n1;
        if (gc > fc) gc += 1.0/n1;
        if (gd > fd) gd += 1.0/n1;
        d2=MAX(d2,abs(fa-ga));
        d2=MAX(d2,abs(fb-gb));
        d2=MAX(d2,abs(fc-gc));
        d2=MAX(d2,abs(fd-gd));
    }
    d=0.5*(d1+d2);                              Average the K-S statistics.
    sqen=sqrt(n1*n2/Doub(n1+n2));
    pearsn(x1,y1,r1,dum,dumm);                  Get the linear correlation coefficient for each
    pearsn(x2,y2,r2,dum,dumm);                      sample.
    rr=sqrt(1.0-0.5*(r1*r1+r2*r2));
    Estimate the probability using the K-S probability function.
    prob=ks.qks(d*sqen/(1.0+rr*(0.25-0.75/sqen)));
}
```

CITED REFERENCES AND FURTHER READING:

Fasano, G. and Franceschini, A. 1987, "A Multidimensional Version of the Kolmogorov-Smirnov Test," *Monthly Notices of the Royal Astronomical Society*, vol. 225, pp. 155–170.[1]

Peacock, J.A. 1983, "Two-Dimensional Goodness-of-Fit Testing in Astronomy," *Monthly Notices of the Royal Astronomical Society*, vol. 202, pp. 615–627.[2]

Spergel, D.N., Piran, T., Loeb, A., Goodman, J., and Bahcall, J.N. 1987, "A Simple Model for Neutrino Cooling of the LMC Supernova," *Science*, vol. 237, pp. 1471–1473.[3]

14.9 Savitzky-Golay Smoothing Filters

In §13.5 we learned something about the construction and application of digital filters, but little guidance was given on *which particular* filter to use. That, of course, depends on what you want to accomplish by filtering. One obvious use for *low-pass* filters is to smooth noisy data.

The premise of data smoothing is that one is measuring a variable that is both slowly varying and also corrupted by random noise. Then it can sometimes be useful to replace each data point by some kind of local average of surrounding data points. Since nearby points measure very nearly the same underlying value, averaging can reduce the level of noise without (much) biasing the value obtained.

We must comment editorially that the smoothing of data lies in a murky area, beyond the fringe of some better-posed, and therefore more highly recommended, techniques that are discussed elsewhere in this book. If you are fitting data to a parametric model, for example (see Chapter 15), it is almost always better to use raw data than to use data that have been pre-processed by a smoothing procedure. Another alternative to blind smoothing is so-called "optimal" or Wiener filtering, as discussed in §13.3 and more generally in §13.6. Data smoothing is probably most justified when it is used simply as a graphical technique, to guide the eye through a forest of data points all with large error bars, or as a means of making initial *rough* estimates of simple parameters from a graph.

In this section we discuss a particular type of low-pass filter, well-adapted for data smoothing, and termed variously *Savitzky-Golay* [1], *least-squares* [2], or *DISPO* (Digital Smoothing Polynomial) [3] filters. Rather than having their properties defined in the Fourier domain and then translated to the time domain, Savitzky-Golay filters derive directly from a particular formulation of the data smoothing problem in the time domain, as we will now see. Savitzky-Golay filters were initially (and are still often) used to render visible the relative widths and heights of spectral lines in noisy spectrometric data.

Recall that a digital filter is applied to a series of equally spaced data values $f_i \equiv f(t_i)$, where $t_i \equiv t_0 + i\Delta$ for some constant sample spacing Δ and $i = \ldots -2, -1, 0, 1, 2, \ldots$. We have seen (§13.5) that the simplest type of digital filter (the nonrecursive or finite impulse response filter) replaces each data value f_i by a linear combination g_i of itself and some number of nearby neighbors,

$$g_i = \sum_{n=-n_L}^{n_R} c_n f_{i+n} \tag{14.9.1}$$

Here n_L is the number of points used "to the left" of a data point i, i.e., earlier than it, while n_R is the number used to the right, i.e., later. A so-called *causal* filter would have $n_R = 0$.

As a starting point for understanding Savitzky-Golay filters, consider the simplest possible averaging procedure: For some fixed $n_L = n_R$, compute each g_i as the average of the data points from f_{i-n_L} to f_{i+n_R}. This is sometimes called *moving window averaging* and corresponds to equation (14.9.1) with constant $c_n = 1/(n_L + n_R + 1)$. If the underlying function is constant, or is changing linearly with time (increasing or decreasing), then no bias is introduced into the result. Higher points at one end of the averaging interval are on the average balanced by lower points at the other end. A bias is introduced, however, if the underlying function has a nonzero second derivative. At a local maximum, for example, moving window averaging always reduces the function value. In the spectrometric application, a narrow spectral line has its height reduced and its width increased. Since these parameters are themselves of physical interest, the bias introduced is distinctly undesirable.

Note, however, that moving window averaging does preserve the area under a spectral line, which is its zeroth moment, and also (if the window is symmetric with $n_L = n_R$) its mean position in time, which is its first moment. What is violated is the second moment, equivalent to the line width.

The idea of Savitzky-Golay filtering is to find filter coefficients c_n that preserve higher moments. Equivalently, the idea is to approximate the underlying function within the moving window not by a constant (whose estimate is the average), but by a polynomial of higher order, typically quadratic or quartic: For each point f_i, we least-squares fit a polynomial to all $n_L + n_R + 1$ points in the moving window, and then set g_i to be the value of that polynomial at position i. (If you are not familiar with least-squares fitting, you might want to look ahead to Chapter 15.) We make no use of the value of the polynomial at any other point. When we move on to the next point f_{i+1}, we do a whole new least-squares fit using a shifted window.

All these least-squares fits would be laborious if done as described. Luckily, since the process of least-squares fitting involves only a linear matrix inversion, the coefficients of a fitted polynomial are themselves linear in the values of the data. That means that we can do all the fitting in advance, for fictitious data consisting of all zeros except for a single 1, and then do the fits on the real data just by taking linear combinations. This is the key point, then: There are particular sets of filter coefficients c_n for which equation (14.9.1) "automatically" accomplishes the process of polynomial least-squares fitting inside a moving window.

To derive such coefficients, consider how g_0 might be obtained: We want to fit a polynomial of degree M in i, namely $a_0 + a_1 i + \cdots + a_M i^M$, to the values $f_{-n_L}, \ldots, f_{n_R}$. Then g_0 will be the value of that polynomial at $i = 0$, namely a_0. The design matrix for this problem (§15.4) is

$$A_{ij} = i^j \qquad i = -n_L, \ldots, n_R, \quad j = 0, \ldots, M \tag{14.9.2}$$

and the normal equations for the vector of a_j's in terms of the vector of f_i's is in matrix notation

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{a} = \mathbf{A}^T \cdot \mathbf{f} \qquad \text{or} \qquad \mathbf{a} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^T \cdot \mathbf{f}) \tag{14.9.3}$$

We also have the specific forms

$$\left\{\mathbf{A}^T \cdot \mathbf{A}\right\}_{ij} = \sum_{k=-n_L}^{n_R} A_{ki} A_{kj} = \sum_{k=-n_L}^{n_R} k^{i+j} \tag{14.9.4}$$

and

$$\left\{\mathbf{A}^T \cdot \mathbf{f}\right\}_{j} = \sum_{k=-n_L}^{n_R} A_{kj} f_k = \sum_{k=-n_L}^{n_R} k^j f_k \tag{14.9.5}$$

Since the coefficient c_n is the component a_0 when $\mathbf{f}$ is replaced by the unit vector $\mathbf{e}_n$, $-n_L \le n < n_R$, we have

$$c_n = \left\{(\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^T \cdot \mathbf{e}_n)\right\}_0 = \sum_{m=0}^{M} \left\{(\mathbf{A}^T \cdot \mathbf{A})^{-1}\right\}_{0m} n^m \tag{14.9.6}$$

Equation (14.9.6) says that we need only one row of the inverse matrix. (Numerically we can get this by LU decomposition with only a single backsubstitution.)

The function `savgol`, below, implements equation (14.9.6). As input, it takes the parameters `nl` $= n_L$, `nr` $= n_R$, and `m` $= M$ (the desired order). Also input

M	n_L	n_R	Sample Savitzky-Golay Coefficients										
2	2	2				-0.086	0.343	0.486	0.343	-0.086			
2	3	1			-0.143	0.171	0.343	0.371	0.257				
2	4	0		0.086	-0.143	-0.086	0.257	0.886					
2	5	5	-0.084	0.021	0.103	0.161	0.196	0.207	0.196	0.161	0.103	0.021	-0.084
4	4	4		0.035	-0.128	0.070	0.315	0.417	0.315	0.070	-0.128	0.035	
4	5	5	0.042	-0.105	-0.023	0.140	0.280	0.333	0.280	0.140	-0.023	-0.105	0.042

is np, the physical length of the output array c, and a parameter ld that for data fitting should be zero. In fact, ld specifies which coefficient among the a_i's should be returned, and we are here interested in a_0. For another purpose, namely the computation of numerical derivatives (already mentioned in §5.7), the useful choice is ld ≥ 1. With ld $= 1$, for example, the filtered first derivative is the convolution (14.9.1) divided by the stepsize Δ. For ld $= k > 1$, the array c must be multiplied by $k!$ to give derivative coefficients. For derivatives, one usually wants m $= 4$ or larger.

savgol.h

```
void savgol(VecDoub_O &c, const Int np, const Int nl, const Int nr,
    const Int ld, const Int m)
```

Returns in c[0..np-1], in wraparound order (N.B.!) consistent with the argument respns in routine convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data points used, while nr is the number of rightward (future) data points, making the total number of data points used nl $+$ nr $+ 1$. ld is the order of the derivative desired (e.g., ld $= 0$ for smoothed function. For the derivative of order k, you must multiply the array c by $k!$.) m is the order of the smoothing polynomial, also equal to the highest conserved moment; usual values are m $= 2$ or m $= 4$.

```
{
    Int j,k,imj,ipj,kk,mm;
    Doub fac,sum;
    if (np < nl+nr+1 || nl < 0 || nr < 0 || ld > m || nl+nr < m)
        throw("bad args in savgol");
    VecInt indx(m+1);
    MatDoub a(m+1,m+1);
    VecDoub b(m+1);
    for (ipj=0;ipj<=(m << 1);ipj++) {          Set up the normal equations of the desired
        sum=(ipj ? 0.0 : 1.0);                      least-squares fit.
        for (k=1;k<=nr;k++) sum += pow(Doub(k),Doub(ipj));
        for (k=1;k<=nl;k++) sum += pow(Doub(-k),Doub(ipj));
        mm=MIN(ipj,2*m-ipj);
        for (imj = -mm;imj<=mm;imj+=2) a[(ipj+imj)/2][(ipj-imj)/2]=sum;
    }
    LUdcmp alud(a);                             Solve them: LU decomposition.
    for (j=0;j<m+1;j++) b[j]=0.0;
    b[ld]=1.0;
    Right-hand side vector is unit vector, depending on which derivative we want.
    alud.solve(b,b);                            Get one row of the inverse matrix.
    for (kk=0;kk<np;kk++) c[kk]=0.0;            Zero the output array (it may be bigger than
    for (k = -nl;k<=nr;k++) {                       number of coefficients).
        sum=b[0];                               Each Savitzky-Golay coefficient is the dot
        fac=1.0;                                    product of powers of an integer with the
        for (mm=1;mm<=m;mm++) sum += b[mm]*(fac *= k);      inverse matrix row.
        kk=(np-k) % np;                         Store in wraparound order.
        c[kk]=sum;
    }
}
```

As output, savgol returns the coefficients c_n, for $-n_L \le n \le n_R$. These are stored in c in "wraparound order"; that is, c_0 is in c[0], c_{-1} is in c[1], and so on for further negative indices. The value c_1 is stored in c[np-1], c_2 in c[np-2], and so on for positive indices. This order may seem arcane, but it is the natural one where causal filters have nonzero coefficients in low array elements of c. It is also the order required by the function convlv in §13.1, which can be used to apply the digital filter to a data set.

The table on the previous page shows some typical output from savgol. For orders 2 and 4, the coefficients of Savitzky-Golay filters with several choices of n_L and n_R are shown. The central column is the coefficient applied to the data f_i in obtaining the smoothed g_i. Coefficients to the left are applied to earlier data, to the right, to later. The coefficients always add (within roundoff error) to unity. One sees that, as befits a smoothing operator, the coefficients always have a central positive lobe, but with smaller, outlying corrections of both positive and negative sign. In practice, the Savitzky-Golay filters are most useful for much larger values of n_L and n_R, since these few-point formulas can accomplish only a relatively small amount of smoothing.

Figure 14.9.1 shows a numerical experiment using a 33-point smoothing filter, that is, $n_L = n_R = 16$. The upper panel shows a test function, constructed to have six "bumps" of varying widths, all of height 8 units. To this function Gaussian white noise of unit variance has been added. (The test function without noise is shown as the dotted curves in the center and lower panels.) The widths of the bumps (full width at half of maximum, or FWHM) are 140, 43, 24, 17, 13, and 10, respectively.

The middle panel of Figure 14.9.1 shows the result of smoothing by a moving window average. One sees that the window of width 33 does quite a nice job of smoothing the broadest bump, but that the narrower bumps suffer considerable loss of height and increase of width. The underlying signal (dotted) is very badly represented.

The lower panel shows the result of smoothing with a Savitzky-Golay filter of the identical width and degree $M = 4$. One sees that the heights and widths of the bumps are quite extraordinarily preserved. A trade-off is that the broadest bump is less smoothed. That is because the central positive lobe of the Savitzky-Golay filter coefficients fills only a fraction of the full 33-point width. As a rough guideline, best results are obtained when the full width of the degree 4 Savitzky-Golay filter is between 1 and 2 times the FWHM of desired features in the data. (References [3] and [4] give additional practical hints.)

Figure 14.9.2 shows the result of smoothing the same noisy "data" with broader Savitzky-Golay filters of three different orders. Here we have $n_L = n_R = 32$ (65-point filter) and $M = 2, 4, 6$. One sees that, when the bumps are too narrow with respect to the filter size, even the Savitzky-Golay filter must at some point give out. The higher-order filter manages to track narrower features, but at the cost of less smoothing on broad features.

To summarize: Within limits, Savitzky-Golay filtering does manage to provide smoothing without loss of resolution. It does this by assuming that relatively distant data points have some significant redundancy that can be used to reduce the level of noise. The specific nature of the assumed redundancy is that the underlying function should be locally well-fitted by a polynomial. When this is true, as it is for smooth line profiles not too much narrower than the filter width, the performance of

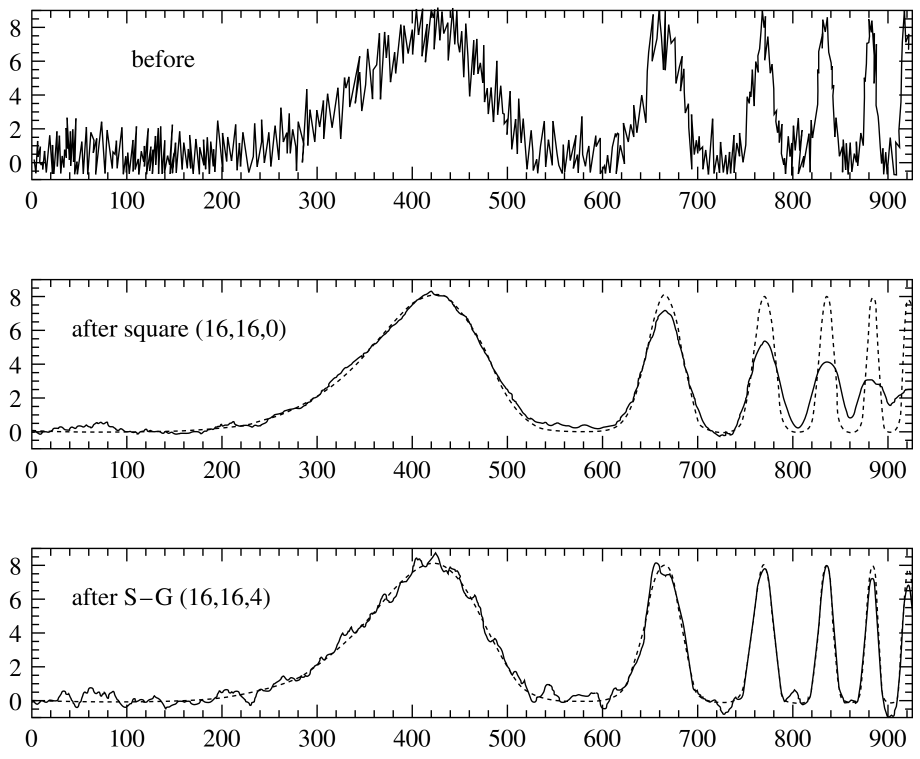


Figure 14.9.1. Top: Synthetic noisy data consisting of a sequence of progressively narrower bumps and additive Gaussian white noise. Center: Result of smoothing the data by a simple moving window average. The window extends 16 points leftward and rightward, for a total of 33 points. Note that narrow features are broadened and suffer corresponding loss of amplitude. The dotted curve is the underlying function used to generate the synthetic data. Bottom: Result of smoothing the data by a Savitzky-Golay smoothing filter (of degree 4) using the same 33 points. While there is less smoothing of the broadest feature, narrower features have their heights and widths preserved.

Savitzky-Golay filters can be spectacular. When it is not true, these filters have no compelling advantage over other classes of smoothing filter coefficients.

A last remark concerns irregularly sampled data, where the values f_i are not uniformly spaced in time. The obvious generalization of Savitzky-Golay filtering would be to do a least-squares fit within a moving window around each data point, one containing a fixed number of data points to the left (n_L) and right (n_R). Because of the irregular spacing, however, there is no way to obtain universal filter coefficients applicable to more than one data point. One must instead do the actual least-squares fits for each data point. This becomes computationally burdensome for larger n_L, n_R, and M.

As a cheap alternative, one can simply pretend that the data points *are* equally spaced. This amounts to virtually shifting, within each moving window, the data points to equally spaced positions. Such a shift introduces the equivalent of an additional source of noise into the function values. In those cases where smoothing is useful, this noise will often be much smaller than the noise already present. Specifically, if the location of the points is approximately random within the window, then a rough

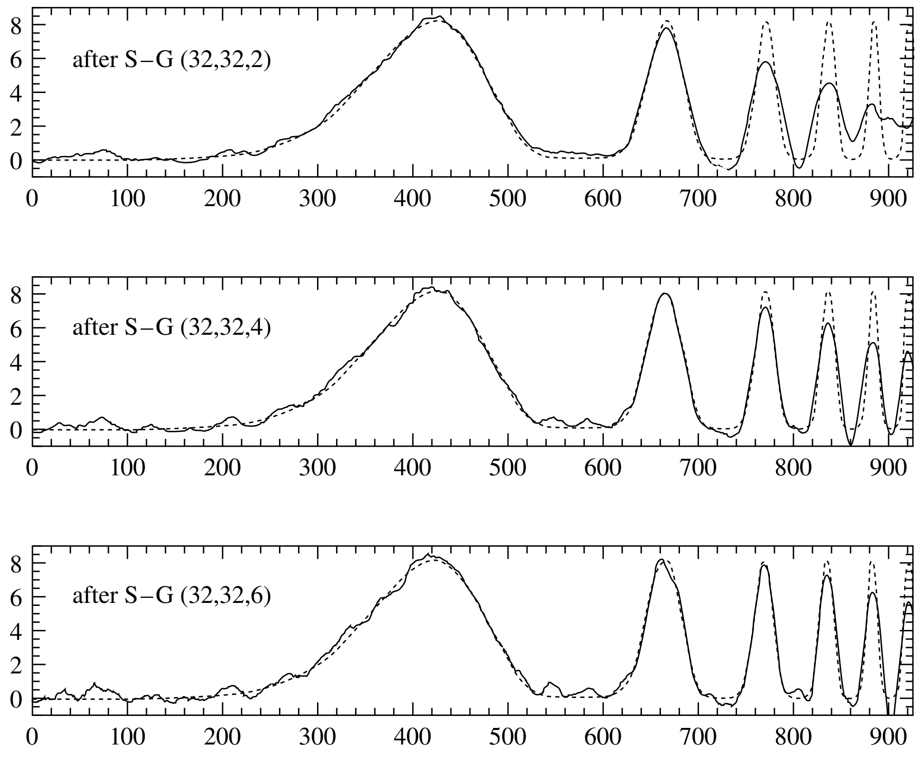


Figure 14.9.2. Result of applying wider 65-point Savitzky-Golay filters to the same data set as in Figure 14.9.1. Top: degree 2. Center: degree 4. Bottom: degree 6. All of these filters are inoptimally broad for the resolution of the narrow features. Higher-order filters do best at preserving feature heights and widths, but do less smoothing on broader features.

criterion is this: If the change in f across the full width of the $N = n_L + n_R + 1$ point window is less than $\sqrt{N/2}$ times the measurement noise on a single point, then the cheap method can be used.

CITED REFERENCES AND FURTHER READING:

Savitzky A., and Golay, M.J.E. 1964, "Smoothing and Differentiation of Data by Simplified Least Squares Procedures," *Analytical Chemistry*, vol. 36, pp. 1627–1639.[1]

Hamming, R.W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall).[2]

Ziegler, H. 1981, "Properties of Digital Smoothing Polynomial (DISPO) Filters," *Applied Spectroscopy*, vol. 35, pp. 88–92.[3]

Bromba, M.U.A., and Ziegler, H. 1981, "Application Hints for Savitzky-Golay Digital Smoothing Filters," *Analytical Chemistry*, vol. 53, pp. 1583–1586.[4]

Modeling of Data

CHAPTER 15

15.0 Introduction

Given a set of observations, one often wants to condense and summarize the data by fitting it to a *model* that depends on adjustable parameters. Sometimes the model is simply a convenient class of functions, such as polynomials or Gaussians, and the fit supplies the appropriate coefficients. Other times, the model's parameters come from some underlying theory that the data are supposed to satisfy; examples are rate coefficients in a complex network of chemical reactions or orbital elements of a binary star. Modeling can also be used as a kind of constrained interpolation, where you want to extend a few data points into a continuous function, but with some underlying idea of what that function should look like.

One very general approach has the following paradigm: You choose or design a *figure-of-merit function* (merit function, for short) that measures the agreement between the data and the model with a particular choice of parameters. In frequentist statistics, the merit function is conventionally arranged so that small values represent close agreement. Bayesians choose as their merit function the probability of the parameters given the data (or often its logarithm) so that larger values represent closer agreement.

In either case, the parameters of the model are then adjusted to find a happy extremum in the merit function, yielding *best-fit parameters*. The adjustment process is thus a problem in minimization in many dimensions. This optimization was the subject of Chapter 10; however, there exist special, more efficient, methods that are specific to modeling, and we will discuss these in this chapter.

There are important issues that go beyond the mere finding of best-fit parameters. Data are generally not exact. They are subject to *measurement errors* (called *noise* in the context of signal processing). Thus, typical data never exactly fit the model that is being used, even when that model is correct. We need the means to assess whether or not the model is appropriate, that is, we need to test the *goodness-of-fit* against some useful statistical standard.

We usually also need to know the accuracy with which parameters are determined by the data set. In frequentist terms, we need to know the standard errors of the best-fit parameters. Alternatively, in Bayesian language, we want to find not just the peak of the joint parameter probability distribution, but the whole distribution.

Or we at least want to be able to sample from that distribution, typically by Markov chain Monte Carlo, as we will discuss at length in §15.8.

It is not uncommon in fitting data to discover that the merit function is not unimodal, with a single minimum. In some cases, we may be interested in global rather than local questions. Not, "how good is this fit?" but rather, "how sure am I that there is not a *very much better* fit in some corner of parameter space?" As we have seen in Chapter 10, especially §10.12, this kind of problem is generally quite difficult to solve.

The important message is that fitting of parameters is not the end-all of model parameter estimation. To be genuinely useful, a fitting procedure should provide (i) parameters, (ii) error estimates on the parameters or a way to sample from their probability distribution, and (iii) a statistical measure of goodness-of-fit. When the third item suggests that the model is an unlikely match to the data, then items (i) and (ii) are probably worthless. Unfortunately, many practitioners of parameter estimation never proceed beyond item (i). They deem a fit acceptable if a graph of data and model "looks good." This approach is known as *chi-by-eye*. Luckily, its practitioners get what they deserve.

15.0.1 Basic Bayes

Because the discussion in this and subsequent chapters will move freely between frequentist and Bayesian methods, this is a good place to compare these two powerfully useful ways of thinking. In §14.0, when we discussed tail, or p-value, tests, we were adopting a frequentist viewpoint. The central frequentist idea is that, given the details of a null hypothesis, there is an implied population (that is, probability distribution) of possible data sets. If the assumed null hypothesis is correct, then the actual, measured, data set is drawn from that population. (We expand on this in §15.6.) It then makes sense to ask questions about how "frequently" some aspect of the measured data occurs in the population. If the answer is "very infrequently," then the hypothesis is rejected. The frequentist viewpoint avoids questions like, "what is the *probability* that this hypothesis is true?" because its focus is on the distribution of data sets, not hypotheses. Indeed, whether by dogma or merely benign neglect, it eschews the machinery needed to handle the concept of a probability distribution of hypotheses.

That machinery is Bayes' theorem, which follows from the standard axioms of probability. Bayes' theorem relates the conditional probabilities of two events, say A and B:

$$P(A|B) = P(A)\,\frac{P(B|A)}{P(B)} \tag{15.0.1}$$

Here $P(A|B)$ is the probability of A *given* that B has occurred, and similarly for $P(B|A)$, while $P(A)$ and $P(B)$ are unconditional probabilities.

Bayesians allow a broader set of uses for probabilities than frequentists. To a Bayesian, $P(A|B)$ is a measure of the degree of plausibility of A (given B) on a scale ranging from zero to one. In this broader view, A and B need not be repeatable events; they can indeed be propositions or hypotheses. In equation (15.0.1), A might be a hypothesis and B might be some data, so that $P(A|B)$ expresses the probability of a hypothesis, given the data. The equations of probability theory thus become a set of consistent rules for conducting inference [1,2]. Interestingly, this viewpoint was universal before the 20th century. The Bernoullis (both of them), Laplace, Gauss, Legendre, and Poisson, among others, made little or no distinction between inference and probability. An opposing frequentist view, that these concepts should be kept separate, became explicit only with the work of Fisher, Box, Kendall, Neyman, and Pearson (among others), much later.

Since plausibility is itself always conditioned on some, perhaps unarticulated, set of assumptions, all Bayesian probabilities are viewed as conditional on some collective background information I. Suppose H is some hypothesis. Even before there exist any explicit data, a Bayesian can assign to H some degree of plausibility $P(H|I)$, called the "Bayesian prior." Now, when some data D_1 comes along, Bayes theorem tells how to reassess the plausibility of H,

$$P(H|D_1 I) = P(H|I)\frac{P(D_1|HI)}{P(D_1|I)} \tag{15.0.2}$$

The factor in the numerator on the right of equation (15.0.2) is calculable as the probability of a data set *given* the hypothesis (comparable to "likelihood" as we will define it in §15.1). The denominator, called the "prior predictive probability" of the data, is in this case merely a normalization constant that can be calculated by the requirement that the probability of all hypotheses should sum to unity. (In other Bayesian contexts, the prior predictive probabilities of two qualitatively different models can be used to assess their relative plausibility.)

If some additional data D_2 come along tomorrow, we can further refine our estimate of H's probability, as

$$P(H|D_2 D_1 I) = P(H|D_1 I)\frac{P(D_2|HD_1 I)}{P(D_2|D_1 I)} \tag{15.0.3}$$

Using the product rule for probabilities, $P(AB|C) = P(A|C)P(B|AC)$, we find that equations (15.0.2) and (15.0.3) imply

$$P(H|D_2 D_1 I) = P(H|I)\frac{P(D_2 D_1|HI)}{P(D_2 D_1|I)} \tag{15.0.4}$$

which shows that we would have gotten the same answer if all the data $D_1 D_2$ had been taken together.

We might wonder, before we adopt the laws of probability as our calculus of inference and thus become Bayesians, whether there are any other alternatives. The answer is, basically, no. Cox [3] showed that making a small number of very reasonable assumptions about "degree of belief" leads inevitably to the axioms of probability, and thus the application of Bayes theorem to the evaluation of hypotheses, given data. Either you become a Bayesian or else you must live in a world with no general calculus of inference.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill), Chapters 6–11.

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press), Chapters 12–13.

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. 2004, *Bayesian Data Analysis*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC).

Sivia, D.S. 1996, *Data Analysis: A Bayesian Tutorial* (Oxford, UK: Oxford University Press).

Jaynes, E.T. 1976, in *Foundations of Probability Theory, Statistical Inference, and Statistical Theories of Science*, W.L. Harper and C.A. Hooker, eds. (Dordrecht: Reidel).[1]

Jaynes, E.T. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel).[2]

Cox, R.T. 1946, "Probability, Frequency, and Reasonable Expectation," *American Journal of Physics*, vol. 14, pp. 1–13.[3]

15.1 Least Squares as a Maximum Likelihood Estimator

Suppose that we are fitting N data points (x_i, y_i), $i = 0, \dots, N-1$, to a model that has M adjustable parameters a_j, $j = 0, \dots, M-1$. The model predicts a functional relationship between the measured independent and dependent variables,

$$y(x) = y(x|a_0 \dots a_{M-1}) \tag{15.1.1}$$

where the notation indicates dependence on the parameters explicitly on the right-hand side, following the vertical bar.

What, exactly, do we want to minimize to get fitted values for the a_j's? The first thing that comes to mind is the familiar least-squares fit,

$$\text{minimize over } a_0 \dots a_{M-1} : \qquad \sum_{i=0}^{N-1} [y_i - y(x_i|a_0 \dots a_{M-1})]^2 \tag{15.1.2}$$

But where does this come from? What general principles is it based on?

To answer these questions, let us start by asking, "*Given a particular set of parameters*, what is the probability that the observed data set should have occurred?" If the y_i's take on continuous values, the probability will always be zero unless we add the phrase, "...plus or minus some small, fixed Δy on each data point." So let's always take this phrase as understood. If the probability of obtaining the data set is too small, then we can conclude that the parameters under consideration are "unlikely" to be right. Conversely, our intuition tells us that the data set should not be too improbable for the correct choice of parameters.

To be more quantitative, suppose that each data point y_i has a measurement error that is independently random and distributed as a normal (Gaussian) distribution around the "true" model $y(x)$. And suppose that the standard deviations σ of these normal distributions are the same for all points. Then the probability of the data set is the product of the probabilities of each point:

$$P(\text{data} \mid \text{model}) \propto \prod_{i=0}^{N-1} \left\{ \exp\left[-\frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma} \right)^2 \right] \Delta y \right\} \tag{15.1.3}$$

Notice that there is a factor Δy in each term in the product.

If we are Bayesians, we proceed by invoking Bayes' theorem, in the form

$$P(\text{model} \mid \text{data}) \propto P(\text{data} \mid \text{model})\, P(\text{model}) \tag{15.1.4}$$

where $P(\text{model}) = P(a_0 \dots a_{M-1})$ is our prior probability distribution on all models. As often as not, we take a constant, *noninformative* prior. The most probable model, then, is the one that maximizes equation (15.1.3) or, equivalently, minimizes the negative of its logarithm,

$$\left[\sum_{i=0}^{N-1} \frac{[y_i - y(x_i)]^2}{2\sigma^2}\right] - N \log \Delta y \tag{15.1.5}$$

Since N, σ, and Δy are all constants, minimizing this equation is equivalent to minimizing (15.1.2).

If we are frequentists, we must get to the same destination by a more tortuous path (as is so often the case when Bayesian and frequentist methods coincide). We are not allowed to think about the notion of probability as applied to parameter sets, because, for frequentists, there is no statistical universe of models from which the parameters are drawn. We instead substitute a dictum: We identify the probability of the data given the parameters (which is computable as above), as the *likelihood* of the parameters given the data. This identification is entirely based on intuition. It has no formal mathematical basis in and of itself. Parameters derived in this way are called *maximum likelihood estimators*.

What we see is that least-squares fitting gives an answer that is both (i) the most probable parameter set in the Bayesian sense, assuming a flat prior, and (ii) the maximum likelihood estimate of the fitted parameters, in both cases *if* the measurement errors are independent and normally distributed with constant standard deviation. Notice that we made no assumption about the linearity or nonlinearity of the model $y(x|a_0 \dots)$ in its parameters $a_0 \dots a_{M-1}$. Just below, we will relax our assumption of constant standard deviations and obtain the very similar formulas for what is called "chi-square fitting" or "weighted least-squares fitting." First, however, let us discuss further our very stringent assumption of a normal distribution.

For a hundred years or so, mathematical statisticians have been in love with the fact that the probability distribution of the sum of a very large number of very small random deviations almost always converges to a normal distribution. (For precise statements of this *central limit theorem*, consult [1] or other standard works on mathematical statistics.) This infatuation tended to focus interest away from the fact that, for real data, the normal distribution is often rather poorly realized, if it is realized at all. We are often taught, rather casually, that, on average, measurements will fall within $\pm\sigma$ of the true value 68% of the time, within $\pm 2\sigma$ 95% of the time, and within $\pm 3\sigma$ 99.7% of the time. Extending this, one would expect a measurement to be off by $\pm 20\sigma$ only one time out of 2×10^{88}. We all know that "glitches" are much more likely than *that*!

In some instances, the deviations from a normal distribution are easy to understand and quantify. For example, in measurements obtained by counting events, the measurement errors are usually distributed as a Poisson distribution, whose cumulative probability function was already discussed in §6.2. When the number of counts going into one data point is large, the Poisson distribution converges toward a Gaussian. However, the convergence is not uniform when measured in fractional accuracy. The more standard deviations out on the tail of the distribution, the larger the number of counts must be before a value close to the Gaussian is realized. The sign of the effect is always the same: The Gaussian predicts that "tail" events are much less likely than they actually (by Poisson) are. This causes such events, when they occur, to skew a least-squares fit much more than they ought.

Other times, the deviations from a normal distribution are not so easy to understand in detail. Experimental points are occasionally just *way off*. Perhaps the power flickered during a point's measurement, or someone kicked the apparatus, or someone wrote down a wrong number. Points like this are called *outliers*. They can easily turn a least-squares fit on otherwise adequate data into nonsense. Their probability of occurrence in the assumed Gaussian model is so small that the maximum likelihood estimator is willing to distort the whole curve to try to bring them, mistakenly, into line.

The subject of *robust statistics* deals with cases where the normal or Gaussian model is a bad approximation, or cases where outliers are important. We will discuss robust methods briefly in §15.7. All the sections between this one and that one assume, one way or the other, a Gaussian model for the measurement errors in the data. It it quite important that you keep the limitations of that model in mind, even as you use the very useful methods that follow from assuming it.

Finally, note that our discussion of measurement errors has been limited to *statistical* errors, the kind that will average away if we only take enough data. Measurements are also susceptible to *systematic* errors that will not go away with any amount of averaging. For example, the calibration of a metal meter stick might depend on its temperature. If we take all our measurements at the same wrong temperature, then no amount of averaging or numerical processing will correct for this unrecognized systematic error.

15.1.1 Chi-Square Fitting

We considered the chi-square statistic once before, in §14.3. Here it arises in a slightly different context.

If each data point (x_i, y_i) has its own, known standard deviation σ_i, then equation (15.1.3) is modified only by putting a subscript i on the symbol σ. That subscript also propagates docilely into (15.1.5), so that the maximum likelihood estimate of the model parameters (and also the Bayesian most probable parameter set) is obtained by minimizing the quantity

$$\chi^2 \equiv \sum_{i=0}^{N-1} \left(\frac{y_i - y(x_i | a_0 \dots a_{M-1})}{\sigma_i} \right)^2 \tag{15.1.6}$$

called the "chi-square."

To whatever extent the measurement errors actually *are* normally distributed, the quantity χ^2 is correspondingly a sum of N squares of normally distributed quantities, each normalized to unit variance. Once we have adjusted the $a_0 \dots a_{M-1}$ to minimize the value of χ^2, the terms in the sum are not all statistically independent. For models that are linear in the a's, however, it turns out that the probability distribution for different values of χ^2 at its minimum can nevertheless be derived analytically, and is the *chi-square distribution for $N - M$ degrees of freedom*. We learned how to compute this probability function using the incomplete gamma function in §6.2. In particular, equation (6.14.39) gives the probability Q that the chi-square should exceed a particular value χ^2 by chance, where $\nu = N - M$ is the *number of degrees of freedom*. The quantity Q, or its complement $P \equiv 1 - Q$, is frequently tabulated in appendices to statistics books, or it can be computed as $P =$ `Chisqdist(`ν`).invcdf(`χ^2`)` by the routine in §6.14.8. It is quite common, and usually not too wrong, to assume that the chi-square distribution holds even for models that are not strictly linear in the a's.

This computed probability gives a quantitative measure for the goodness-of-fit of the model. If Q is a very small probability for some particular data set, then the apparent discrepancies are unlikely to be chance fluctuations. Much more probably either (i) the model is wrong — can be statistically rejected, or (ii) someone has lied to you about the size of the measurement errors σ_i — they are really larger than stated.

While above we were quick to poke fun at the frequentist's foundations for maximum likelihood estimation (or lack thereof), we must now take aim at strict Bayesians: There are no good fully Bayesian methods for assessing goodness-of-fit, that is, for comparing the probability of a best-fit model to that of a nonspecific alternative hypothesis like "the model is wrong." The problem is that the strict application of Bayes theorem requires either (i) a comparison between two well-posed hypotheses (the *odds ratio*), or (ii) a normalization of the probability of the best-fit model against an integral of such probabilities over all possible models (the *normalizing constant*). In most situations neither of these is available. Sensible Bayesians usually fall back to p-value tail statistics like chi-square probability when they really need to know if a model is wrong.

Another important point is that the chi-square probability Q does not directly measure the credibility of the assumption that the measurement errors are normally distributed. It assumes they are. In most, but not all, cases, however, the effect of nonnormal errors is to create an abundance of outlier points. These decrease the probability Q, so that we can add another possible, though less definitive, conclusion to the above list: (iii) the measurement errors may not be normally distributed.

Possibility (iii) is fairly common, and also fairly benign. It is for this reason that reasonable experimenters are often rather tolerant of low probabilities Q. It is not uncommon to deem acceptable on equal terms any models with, say, $Q > 0.001$. This is not as sloppy as it sounds: Truly *wrong* models will often be rejected with vastly smaller values of Q, 10^{-18}, say. However, if day-in and day-out you find yourself accepting models with $Q \sim 10^{-3}$, you really should track down the cause.

If you happen to know the actual distribution law of your measurement errors, then you might wish to *Monte Carlo simulate* some data sets drawn from a particular model, cf. §7.3 – §7.4. You can then subject these synthetic data sets to your actual fitting procedure, so as to determine both the probability distribution of the χ^2 statistic and also the accuracy with which your model parameters are reproduced by the fit. We discuss this further in §15.6. The technique is very general, but it can also be slow.

At the opposite extreme, it sometimes happens that the probability Q is too large, too near to 1, literally too good to be true! Nonnormal measurement errors cannot in general produce this disease, since the normal distribution is about as "compact" as a distribution can be. Almost always, the cause of too good a chi-square fit is that the experimenter, in a "fit" of conservativism, has *overestimated* his or her measurement errors. Very rarely, too good a chi-square signals actual fraud, data that have been "fudged" to fit the model.

A rule of thumb is that a "typical" value of χ^2 for a "moderately" good fit is $\chi^2 \approx \nu$. More precise is the statement that the χ^2 statistic has a mean ν and a standard deviation $\sqrt{2\nu}$ and, asymptotically for large ν, becomes normally distributed.

In some cases the uncertainties associated with a set of measurements are not known in advance, and considerations related to χ^2 fitting are used to derive a value for σ. If we assume that all measurements have the same standard deviation, $\sigma_i = \sigma$, and that the model does fit well, then we can proceed by first assigning an arbitrary constant σ to all points, next fitting for the model parameters by minimizing χ^2, and finally recomputing

$$\sigma^2 = \sum_{i=0}^{N-1} [y_i - y(x_i)]^2/(N-M) \tag{15.1.7}$$

Obviously, this approach prohibits an independent assessment of goodness-of-fit, a fact occasionally missed by its adherents. When, however, the measurement error is not known, this approach at least allows *some* kind of error bar to be assigned to the points.

If we take the derivative of equation (15.1.6) with respect to the parameters a_k, we obtain equations that must hold at the chi-square minimum:

$$0 = \sum_{i=0}^{N-1} \left(\frac{y_i - y(x_i)}{\sigma_i^2}\right)\left(\frac{\partial y(x_i|\ldots a_k \ldots)}{\partial a_k}\right) \qquad k = 0,\ldots,M-1 \tag{15.1.8}$$

Equation (15.1.8) is, in general, a set of M nonlinear equations for the M unknown a_k. Various of the procedures described subsequently in this chapter derive from (15.1.8) and its specializations.

CITED REFERENCES AND FURTHER READING:

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press), Chapters 10–11.[1]

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press), Chapter 6.

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. 2004, *Bayesian Data Analysis*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC), Chapter 8.

15.2 Fitting Data to a Straight Line

A concrete example will make the considerations of the previous section more meaningful. We consider the problem of fitting a set of N data points (x_i, y_i) to a straight-line model

$$y(x) = y(x|a,b) = a + bx \tag{15.2.1}$$

This problem is often called *linear regression*, a terminology that originated, long ago, in the social sciences. We assume that the uncertainty σ_i associated with each measurement y_i is known, and that the x_i's (values of the dependent variable) are known exactly.

To measure how well the model agrees with the data, we use the chi-square merit function (15.1.6), which in this case is

$$\chi^2(a,b) = \sum_{i=0}^{N-1} \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2 \tag{15.2.2}$$

If the measurement errors are normally distributed, then this merit function will give maximum likelihood parameter estimations of a and b; if the errors are not normally distributed, then the estimations are not maximum likelihood but may still be useful in a practical sense. In §15.7, we will treat the case where outlier points are so numerous as to render the χ^2 merit function useless.

Equation (15.2.2) is minimized to determine a and b. At its minimum, derivatives of $\chi^2(a,b)$ with respect to a, b vanish:

$$\begin{aligned} 0 &= \frac{\partial \chi^2}{\partial a} = -2 \sum_{i=0}^{N-1} \frac{y_i - a - bx_i}{\sigma_i^2} \\ 0 &= \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=0}^{N-1} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2} \end{aligned} \tag{15.2.3}$$

These conditions can be rewritten in a convenient form if we define the following sums:

$$\begin{aligned} S &\equiv \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \qquad S_x \equiv \sum_{i=0}^{N-1} \frac{x_i}{\sigma_i^2} \qquad S_y \equiv \sum_{i=0}^{N-1} \frac{y_i}{\sigma_i^2} \\ S_{xx} &\equiv \sum_{i=0}^{N-1} \frac{x_i^2}{\sigma_i^2} \qquad S_{xy} \equiv \sum_{i=0}^{N-1} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \tag{15.2.4}$$

With these definitions (15.2.3) becomes

$$\begin{aligned} aS + bS_x &= S_y \\ aS_x + bS_{xx} &= S_{xy} \end{aligned} \tag{15.2.5}$$

The solution of these two equations in two unknowns is calculated as

$$\begin{aligned} \Delta &\equiv SS_{xx} - (S_x)^2 \\ a &= \frac{S_{xx}S_y - S_xS_{xy}}{\Delta} \\ b &= \frac{SS_{xy} - S_xS_y}{\Delta} \end{aligned} \tag{15.2.6}$$

Equation (15.2.6) gives the solution for the best-fit model parameters a and b.

We are not done, however. We must estimate the probable uncertainties in the estimates of a and b, since obviously the measurement errors in the data must introduce some uncertainty in the determination of those parameters. If the data are independent, then each contributes its own bit of uncertainty to the parameters. Consideration of propagation of errors shows that the variance σ_f^2 in the value of any function will be

$$\sigma_f^2 = \sum_{i=0}^{N-1} \sigma_i^2 \left(\frac{\partial f}{\partial y_i} \right)^2 \tag{15.2.7}$$

For the straight line, the derivatives of a and b with respect to y_i can be directly evaluated from the solution:

$$\begin{aligned} \frac{\partial a}{\partial y_i} &= \frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta} \\ \frac{\partial b}{\partial y_i} &= \frac{S x_i - S_x}{\sigma_i^2 \Delta} \end{aligned} \tag{15.2.8}$$

Summing over the points as in (15.2.7), we get

$$\begin{aligned} \sigma_a^2 &= S_{xx}/\Delta \\ \sigma_b^2 &= S/\Delta \end{aligned} \tag{15.2.9}$$

which are the variances in the estimates of a and b, respectively. We will see in §15.6 that an additional number is also needed to characterize properly the probable uncertainty of the parameter estimation. That number is the *covariance* of a and b, and (as we will see below) is given by

$$\mathrm{Cov}(a, b) = -S_x/\Delta \tag{15.2.10}$$

The coefficient of correlation between the uncertainty in a and the uncertainty in b, which is a number between -1 and 1, follows from (15.2.10) (compare equation 14.5.1),

$$r_{ab} = \frac{-S_x}{\sqrt{S S_{xx}}} \tag{15.2.11}$$

A positive value of r_{ab} indicates that the errors in a and b are likely to have the same sign, while a negative value indicates the errors are anticorrelated, likely to have opposite signs.

We are *still* not done. We must estimate the goodness-of-fit of the data to the model. Absent this estimate, we have not the slightest indication that the parameters a and b in the model have any meaning at all! The probability Q that a value of chi-square as *poor* as the value (15.2.2) should occur by chance is

$$Q = 1 - \texttt{Chisqdist}(N-2)\texttt{.invcdf}\,(\chi^2) \tag{15.2.12}$$

Here `Chisqdist` is our object embodying the chi-square distribution function (see §6.14.8) and `invcdf` is its inverse cumulative distribution function. If Q is larger than, say, 0.1, then the goodness-of-fit is believable. If it is larger than, say, 0.001, then the fit *may* be acceptable if the errors are nonnormal or have been moderately underestimated. If Q is less than 0.001, then the model and/or estimation procedure can rightly be called into question. In this latter case, turn to §15.7 to proceed further.

If you do not know the individual measurement errors of the points σ_i, and are proceeding (dangerously) to use equation (15.1.7) for estimating these errors, then here is the procedure for estimating the probable uncertainties of the parameters a and b: Set $\sigma_i \equiv 1$ in all equations through (15.2.6), and multiply σ_a and σ_b, as obtained from equation (15.2.9), by the additional factor $\sqrt{\chi^2/(N-2)}$, where χ^2 is computed by (15.2.2) using the fitted parameters a and b. As discussed above, this procedure is equivalent to *assuming* a good fit, so you get no independent goodness-of-fit probability Q.

In §14.5 we promised a relation between the linear correlation coefficient r (equation 14.5.1) and a goodness-of-fit measure, χ^2 (equation 15.2.2). For unweighted data (all $\sigma_i = 1$), that relation is

$$\chi^2 = (1-r^2)\sum_{i=0}^{N-1}(y_i - \overline{y})^2 \tag{15.2.13}$$

For data with varying errors σ_i, the above equations remain valid if the sums in equations (15.2.13) and (14.5.1) are weighted by $1/\sigma_i^2$.

The following object, `Fitab`, carries out exactly the operations that we have discussed. You call its constructor either with, or without, errors σ_i. If the σ_i's are known, the calculations exactly correspond to the formulas above. However, when σ_i's are unavailable, the routine *assumes* equal values of σ for each point and *assumes* a good fit, as discussed in §15.1.

The formulas (15.2.6) are susceptible to roundoff error. Accordingly, we rewrite them as follows: Define

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \qquad i = 0, 1, \dots, N-1 \tag{15.2.14}$$

and

$$S_{tt} = \sum_{i=0}^{N-1} t_i^2 \tag{15.2.15}$$

Then, as you can verify by direct substitution,

$$b = \frac{1}{S_{tt}}\sum_{i=0}^{N-1}\frac{t_i y_i}{\sigma_i} \tag{15.2.16}$$

$$a = \frac{S_y - S_x b}{S} \tag{15.2.17}$$

$$\sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{S S_{tt}}\right) \tag{15.2.18}$$

$$\sigma_b^2 = \frac{1}{S_{tt}} \tag{15.2.19}$$

$$\mathrm{Cov}(a,b) = -\frac{S_x}{S S_{tt}} \tag{15.2.20}$$

$$r_{ab} = \frac{\mathrm{Cov}(a,b)}{\sigma_a \sigma_b} \tag{15.2.21}$$

fitab.h

```
struct Fitab {
```

Object for fitting a straight line $y = a + bx$ to a set of points (x_i, y_i), with or without available errors σ_i. Call one of the two constructors to calculate the fit. The answers are then available as the variables a, b, siga, sigb, chi2, and either q or sigdat.

```
    Int ndata;
    Doub a, b, siga, sigb, chi2, q, sigdat;             Answers.
    VecDoub_I &x, &y, &sig;

    Fitab(VecDoub_I &xx, VecDoub_I &yy, VecDoub_I &ssig)
    : ndata(xx.size()), x(xx), y(yy), sig(ssig), chi2(0.), q(1.), sigdat(0.)  {
```

Constructor. Given a set of data points x[0..ndata-1], y[0..ndata-1] with individual standard deviations sig[0..ndata-1], sets a,b and their respective probable uncertainties siga and sigb, the chi-square chi2, and the goodness-of-fit probability q (that the fit would have χ^2 this large or larger).

```
        Gamma gam;
        Int i;
        Doub ss=0.,sx=0.,sy=0.,st2=0.,t,wt,sxoss;
        b=0.0;                                          Accumulate sums ...
        for (i=0;i<ndata;i++) {
            wt=1.0/SQR(sig[i]);                         ...with weights
            ss += wt;
            sx += x[i]*wt;
            sy += y[i]*wt;
        }
        sxoss=sx/ss;
        for (i=0;i<ndata;i++) {
            t=(x[i]-sxoss)/sig[i];
            st2 += t*t;
            b += t*y[i]/sig[i];
        }
        b /= st2;                                       Solve for $a$, $b$, $\sigma_a$, and $\sigma_b$.
        a=(sy-sx*b)/ss;
        siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
        sigb=sqrt(1.0/st2);                             Calculate $\chi^2$.
        for (i=0;i<ndata;i++) chi2 += SQR((y[i]-a-b*x[i])/sig[i]);
        if (ndata>2) q=gam.gammq(0.5*(ndata-2),0.5*chi2);       Equation (15.2.12).
    }

    Fitab(VecDoub_I &xx, VecDoub_I &yy)
    : ndata(xx.size()), x(xx), y(yy), sig(xx), chi2(0.), q(1.), sigdat(0.) {
```

Constructor. As above, but without known errors (sig is not used). The uncertainties siga and sigb are estimated by assuming equal errors for all points, and that a straight line is a good fit. q is returned as 1.0, the normalization of chi2 is to unit standard deviation on all points, and sigdat is set to the estimated error of each point.

```
        Int i;
        Doub ss,sx=0.,sy=0.,st2=0.,t,sxoss;
        b=0.0;                                          Accumulate sums ...
        for (i=0;i<ndata;i++) {
            sx += x[i];                                 ...without weights.
            sy += y[i];
        }
        ss=ndata;
        sxoss=sx/ss;
        for (i=0;i<ndata;i++) {
            t=x[i]-sxoss;
            st2 += t*t;
            b += t*y[i];
        }
        b /= st2;                                       Solve for $a$, $b$, $\sigma_a$, and $\sigma_b$.
        a=(sy-sx*b)/ss;
        siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
        sigb=sqrt(1.0/st2);                             Calculate $\chi^2$.
        for (i=0;i<ndata;i++) chi2 += SQR(y[i]-a-b*x[i]);
        if (ndata > 2) sigdat=sqrt(chi2/(ndata-2));
        siga *= sigdat;
        sigb *= sigdat;
    }
};
```

For unweighted data evaluate typical `sig` using `chi2`, and adjust the standard deviations.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill), Chapter 6.

Devore, J.L. 2003, *Probability and Statistics for Engineering and the Sciences*, 6th ed. (Belmont, CA: Duxbury Press), Chapter 12.

15.3 Straight-Line Data with Errors in Both Coordinates

If experimental data are subject to measurement error not only in the y_i's, but also in the x_i's, then the task of fitting a straight-line model

$$y(x) = a + bx \tag{15.3.1}$$

is considerably harder. It is straightforward to write down the χ^2 merit function for this case,

$$\chi^2(a,b) = \sum_{i=0}^{N-1} \frac{(y_i - a - bx_i)^2}{\sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2} \tag{15.3.2}$$

where $\sigma_{x\,i}$ and $\sigma_{y\,i}$ are, respectively, the x and y standard deviations for the ith point. The weighted sum of variances in the denominator of equation (15.3.2) can be understood both as the variance in the direction of the smallest χ^2 between each data point and the line with slope b, and also as the variance of the linear combination $y_i - a - bx_i$ of two random variables x_i and y_i,

$$\text{Var}(y_i - a - bx_i) = \text{Var}(y_i) + b^2\text{Var}(x_i) = \sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2 \equiv 1/w_i \tag{15.3.3}$$

The sum of the square of N random variables, each normalized by its variance, is thus chi-square distributed.

We want to minimize equation (15.3.2) with respect to a and b. Unfortunately, the occurrence of b in the denominator of equation (15.3.2) makes the resulting equation for the slope $\partial\chi^2/\partial b = 0$ nonlinear. However, the corresponding condition for the intercept, $\partial\chi^2/\partial a = 0$, is still linear and yields

$$a = \left[\sum_i w_i(y_i - bx_i)\right] \Big/ \sum_i w_i \tag{15.3.4}$$

where the w_i's are defined by equation (15.3.3). A reasonable strategy, now, is to use the machinery of Chapter 10 (e.g., a `Brent` object) for minimizing a general one-dimensional function to minimize with respect to b while using equation (15.3.4) at each stage to ensure that the minimum with respect to b is also minimized with respect to a.

Because of the finite error bars on the x_i's, the minimum χ^2 as a function of b will be finite, though usually large, when b equals infinity (line of infinite slope). The angle $\theta \equiv$

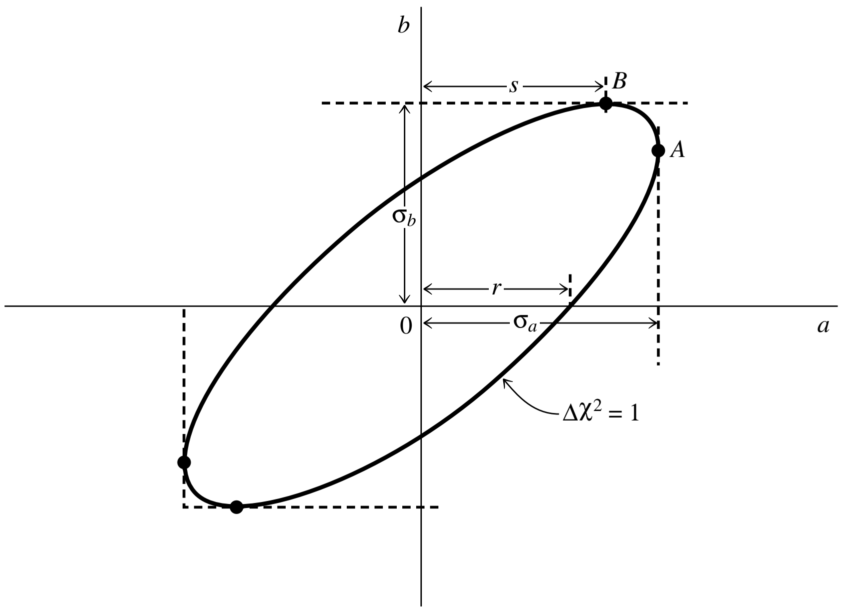


Figure 15.3.1. Standard errors for the parameters a and b. The point B can be found by varying the slope b while simultaneously minimizing the intercept a. This gives the standard error σ_b and also the value s. The standard error σ_a can then be found by the geometric relation $\sigma_a^2 = s^2 + r^2$.

arctan b is thus more suitable as a parametrization of slope than b itself. The value of χ^2 will then be periodic in θ with period π (not 2π!). If any data points have very small σ_y 's but moderate or large σ_x 's, then it is also possible to have a maximum in χ^2 near zero slope, $\theta \approx 0$. In that case, there can sometimes be two χ^2 minima, one at positive slope and the other at negative. Only one of these is the correct global minimum. It is therefore important to have a good starting guess for b (or θ). Our strategy, implemented below, is to scale the y_i 's so as to have variance equal to the x_i 's, and then to do a conventional (as in §15.2) linear fit with weights derived from the (scaled) sum $\sigma_{y\,i}^2 + \sigma_{x\,i}^2$. This yields a good starting guess for b if the data are even *plausibly* related to a straight-line model.

Finding the standard errors σ_a and σ_b on the parameters a and b is more complicated. We will see in §15.6 that, in appropriate circumstances, the standard errors in a and b are the respective projections onto the a- and b-axes of the "confidence region boundary" where χ^2 takes on a value one greater than its minimum, $\Delta\chi^2 = 1$. In the linear case of §15.2, these projections follow from the Taylor series expansion

$$\Delta\chi^2 \approx \frac{1}{2}\left[\frac{\partial^2\chi^2}{\partial a^2}(\Delta a)^2 + \frac{\partial^2\chi^2}{\partial b^2}(\Delta b)^2\right] + \frac{\partial^2\chi^2}{\partial a \partial b}\Delta a \Delta b \tag{15.3.5}$$

Because of the present nonlinearity in b, however, analytic formulas for the second derivatives are quite unwieldy; more important, the lowest-order term frequently gives a poor approximation to $\Delta\chi^2$. Our strategy is therefore to find the roots of $\Delta\chi^2 = 1$ numerically, by adjusting the value of the slope b away from the minimum. In the program below, the general root finder `zbrent` is used. It may occur that there are no roots at all — for example, if all error bars are so large that all the data points are compatible with each other. It is important, therefore, to make some effort at bracketing a putative root before refining it (cf. §9.1).

Because a is minimized at each stage of varying b, successful numerical root finding leads to a value of Δa that minimizes χ^2 for the value of Δb that gives $\Delta\chi^2 = 1$. This (see Figure 15.3.1) directly gives the tangent projection of the confidence region onto the b-axis, and thus σ_b. It does not, however, give the tangent projection of the confidence region onto the a-axis. In the figure, we have found the point labeled B; to find σ_a we need to find the point A. Geometry to the rescue: To the extent that the confidence region is approximated by an ellipse, you can prove (see figure) that $\sigma_a^2 = r^2 + s^2$. The value of s is known from having found the point B. The value of r follows from equations (15.3.2) and (15.3.3) applied at the χ^2 minimum (point O in the figure), giving

$$r^2 = 1 \Big/ \sum_i w_i \tag{15.3.6}$$

Actually, since b can go through infinity, this whole procedure makes more sense in (a, θ) space than in (a, b) space. That is, in fact, how the following program works. Since it is conventional, however, to return standard errors for a and b, not a and θ, we finally use the relation

$$\sigma_b = \sigma_\theta / \cos^2 \theta \tag{15.3.7}$$

We caution that if b and its standard error are both large, so that the confidence region actually includes infinite slope, then the standard error σ_b is not very meaningful. The functor `Chixy` is normally called only by the routine `Fitexy`. However, if you want, you can yourself explore the confidence region by making repeated calls to `Chixy` (whose argument is an angle θ, not a slope b), after a single initializing call to `Fitexy`.

Be aware that the literature on the seemingly straightforward subject of this section is generally confusing and sometimes plain wrong. Deming's [1] early treatment is sound, but its reliance on Taylor expansions gives inaccurate error estimates. References [2-4] are reliable, more recent, general treatments with critiques of earlier work. York [5] and Reed [6] usefully discuss the simple case of a straight line as treated here, but the latter paper has some errors, corrected in [7]. All this commotion has attracted the Bayesians [8-10], who have still different points of view.

A final caution, repeated from §15.0, is that if the goodness-of-fit is not acceptable (returned probability is too small), the standard errors σ_a and σ_b are surely not believable. In dire circumstances, you might try scaling all your x and y error bars by a constant factor until the probability is acceptable (0.5, say), to get more plausible values for σ_a and σ_b.

Implementing code is given in a Webnote [11].

CITED REFERENCES AND FURTHER READING:

Deming, W.E. 1943, *Statistical Adjustment of Data* (New York: Wiley), reprinted 1964 (New York: Dover).[1]

Jefferys, W.H. 1980, "On the Method of Least Squares," *Astronomical Journal*, vol. 85, pp. 177–181; see also vol. 95, p. 1299 (1988).[2]

Jefferys, W.H. 1981, "On the Method of Least Squares — Part Two," *Astronomical Journal*, vol. 86, pp. 149–155; see also vol. 95, p. 1300 (1988).[3]

Lybanon, M. 1984, "A Better Least-Squares Method When Both Variables Have Uncertainties," *American Journal of Physics*, vol. 52, pp. 22–26.[4]

York, D. 1966, "Least-Squares Fitting of a Straight Line," *Canadian Journal of Physics*, vol. 44, pp. 1079–1086.[5]

Reed, B.C. 1989, "Linear Least-Squares Fits with Error in Both Coordinates," *American Journal of Physics*, vol. 57, pp. 642–646; see also vol. 58, p. 189, and vol. 58, p. 1209.[6]

Reed, B.C. 1992, "Linear Least-squares Fits with Errors in Both Coordinates. II: Comments on Parameter Variances," *American Journal of Physics*, vol. 60, pp. 59–62.[7]

Zellner, A. 1971, *An Introduction to Bayesian Inference in Econometrics* (New York: Wiley); reprinted 1987 (Malabar, FL: R. E. Krieger).[8]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer).[9]

Jaynes, E.T. 1991, in *Maximum-Entropy and Bayesian Methods, Proceedings of the 10th International Workshop*, W.T. Grandy, Jr., and L.H. Schick, eds. (Boston: Kluwer).[10]

Macdonald, J.R., and Thompson, W.J. 1992, "Least-Squares Fitting When Both Variables Contain Errors: Pitfalls and Possibilities," *American Journal of Physics*, vol. 60, pp. 66–73.

Numerical Recipes Software 2007, "Code Implementation for Fitexy," *Numerical Recipes Webnote No. 19*, at `http://www.nr.com/webnotes?19` [11]

15.4 General Linear Least Squares

An immediate generalization of §15.2 is to fit a set of data points (x_i, y_i) to a model that is not just a linear combination of 1 and x (namely $a + bx$), but rather a linear combination of *any* M specified functions of x. For example, the functions could be $1, x, x^2, \dots, x^{M-1}$, in which case their general linear combination,

$$y(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{M-1} x^{M-1} \tag{15.4.1}$$

is a polynomial of degree $M - 1$. Or, the functions could be sines and cosines, in which case their general linear combination is a Fourier series. The general form of this kind of model is

$$y(x) = \sum_{k=0}^{M-1} a_k X_k(x) \tag{15.4.2}$$

where the quantities $X_0(x), \dots, X_{M-1}(x)$ are arbitrary fixed functions of x, called the *basis functions*.

Note that the functions $X_k(x)$ can be wildly nonlinear functions of x. In this discussion, "linear" refers only to the model's dependence on its *parameters* a_k.

For these linear models we generalize the discussion of the previous section by defining a merit function

$$\chi^2 = \sum_{i=0}^{N-1} \left[\frac{y_i - \sum_{k=0}^{M-1} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.3}$$

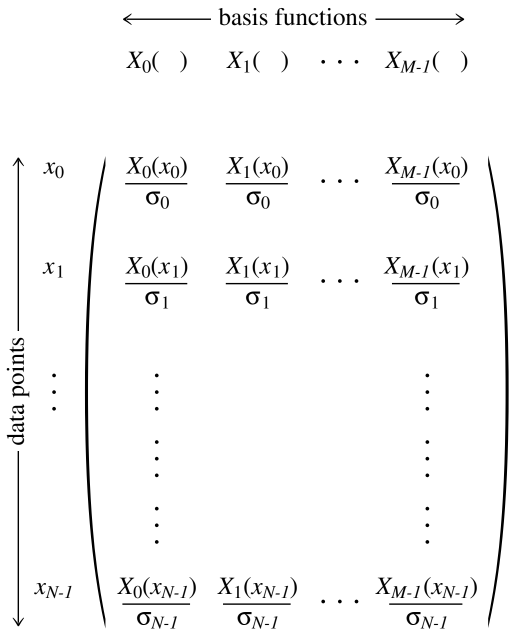
As before, σ_i is the measurement error (standard deviation) of the ith data point, presumed to be known. If the measurement errors are not known, they may all (as discussed at the end of §15.1) be set to the constant value $\sigma = 1$.

Once again, we will pick as best parameters those that minimize χ^2. There are several different techniques available for finding this minimum. Two are particularly useful, and we will discuss both in this section. To introduce them and elucidate their relationship, we need some notation.

Let $\mathbf{A}$ be a matrix whose $N \times M$ components are constructed from the M basis functions evaluated at the N abscissas x_i, and from the N measurement errors σ_i, by the prescription

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{15.4.4}$$

The matrix $\mathbf{A}$ is called the *design matrix* of the fitting problem. Notice that in general $\mathbf{A}$ has more rows than columns, $N \geq M$, since there must be more data points than model parameters to be solved for. (You can fit a straight line to two points, but not a very meaningful quintic!) The design matrix is shown schematically in Figure 15.4.1.



$$
\begin{array}{c|cccc}
 & X_0(\ ) & X_1(\ ) & \cdots & X_{M-1}(\ ) \\
\hline
x_0 & \frac{X_0(x_0)}{\sigma_0} & \frac{X_1(x_0)}{\sigma_0} & \cdots & \frac{X_{M-1}(x_0)}{\sigma_0} \\
x_1 & \frac{X_0(x_1)}{\sigma_1} & \frac{X_1(x_1)}{\sigma_1} & \cdots & \frac{X_{M-1}(x_1)}{\sigma_1} \\
\vdots & \vdots & & & \vdots \\
x_{N-1} & \frac{X_0(x_{N-1})}{\sigma_{N-1}} & \frac{X_1(x_{N-1})}{\sigma_{N-1}} & \cdots & \frac{X_{M-1}(x_{N-1})}{\sigma_{N-1}}
\end{array}
$$

Figure 15.4.1. Design matrix **A** for the least-squares fit of a linear combination of M basis functions to N data points. The matrix elements involve the basis functions evaluated at the values of the independent variable at which measurements are made and the standard deviations of the measured dependent variable. The measured values of the dependent variable do not enter the design matrix.

Also define a vector **b** of length N by

$$b_i = \frac{y_i}{\sigma_i} \tag{15.4.5}$$

and denote the M vector whose components are the parameters to be fitted, $a_0, \dots, a_{M-1}$, by **a**.

15.4.1 Solution by Use of the Normal Equations

The minimum of (15.4.3) occurs where the derivative of χ^2 with respect to all M parameters a_k vanishes. Specializing equation (15.1.8) to the case of the model (15.4.2), this condition yields the M equations

$$0 = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=0}^{M-1} a_j X_j(x_i) \right] X_k(x_i) \qquad k = 0, \dots, M-1 \tag{15.4.6}$$

Interchanging the order of summations, we can write (15.4.6) as the matrix equation

$$\sum_{j=0}^{M-1} \alpha_{kj} a_j = \beta_k \tag{15.4.7}$$

where

$$\alpha_{kj} = \sum_{i=0}^{N-1} \frac{X_j(x_i)X_k(x_i)}{\sigma_i^2} \qquad \text{or, equivalently,} \qquad \boldsymbol{\alpha} = \mathbf{A}^T \cdot \mathbf{A} \tag{15.4.8}$$

an $M \times M$ matrix, and

$$\beta_k = \sum_{i=0}^{N-1} \frac{y_i X_k(x_i)}{\sigma_i^2} \qquad \text{or, equivalently,} \qquad \boldsymbol{\beta} = \mathbf{A}^T \cdot \mathbf{b} \tag{15.4.9}$$

a vector of length M.

The equations (15.4.6) or (15.4.7) are called the *normal equations* of the least-squares problem. They can be solved for the vector of parameters **a** by the standard methods of Chapter 2, notably LU decomposition and backsubstitution, Choleksy decomposition, or Gauss-Jordan elimination. In matrix form, the normal equations can be written as either

$$\boldsymbol{\alpha} \cdot \mathbf{a} = \boldsymbol{\beta} \qquad \text{or as} \qquad \left(\mathbf{A}^T \cdot \mathbf{A}\right) \cdot \mathbf{a} = \mathbf{A}^T \cdot \mathbf{b} \tag{15.4.10}$$

The inverse matrix $\mathbf{C} \equiv \boldsymbol{\alpha}^{-1}$, called the covariance matrix, is closely related to the probable (or, more precisely, *standard*) uncertainties of the estimated parameters **a**. To estimate these uncertainties, consider that

$$a_j = \sum_{k=0}^{M-1} \boldsymbol{\alpha}_{jk}^{-1} \beta_k = \sum_{k=0}^{M-1} C_{jk} \left[\sum_{i=0}^{N-1} \frac{y_i X_k(x_i)}{\sigma_i^2} \right] \tag{15.4.11}$$

and that the variance associated with the estimate a_j can be found as in (15.2.7) from

$$\sigma^2(a_j) = \sum_{i=0}^{N-1} \sigma_i^2 \left(\frac{\partial a_j}{\partial y_i} \right)^2 \tag{15.4.12}$$

Note that α_{jk} is independent of y_i, so that

$$\frac{\partial a_j}{\partial y_i} = \sum_{k=0}^{M-1} C_{jk} X_k(x_i)/\sigma_i^2 \tag{15.4.13}$$

Consequently, we find that

$$\sigma^2(a_j) = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} C_{jk} C_{jl} \left[\sum_{i=0}^{N-1} \frac{X_k(x_i) X_l(x_i)}{\sigma_i^2} \right] \tag{15.4.14}$$

The final term in brackets is just the matrix $\boldsymbol{\alpha}$. Since this is the matrix inverse of $\mathbf{C}$, (15.4.14) reduces immediately to

$$\sigma^2(a_j) = C_{jj} \tag{15.4.15}$$

In other words, the diagonal elements of $\mathbf{C}$ are the variances (squared uncertainties) of the fitted parameters **a**. It should not surprise you to learn that the off-diagonal elements C_{jk} are the covariances between a_j and a_k (cf. 15.2.10); but we shall defer discussion of these to §15.6.

We will now give a routine that implements the above formulas for the general linear least-squares problem, by the method of normal equations. Since we wish to compute not only the solution vector **a** but also the covariance matrix **C**, it is most convenient to use Gauss-Jordan elimination (routine gaussj of §2.1) to perform the linear algebra. The operation count in this application is no larger than that for LU decomposition. If you have no need for the covariance matrix, however, you can save a factor of 3 on the linear algebra by switching to LU decomposition, without computation of the matrix inverse. In theory, since $\mathbf{A}^T \cdot \mathbf{A}$ is positive-definite, Cholesky decomposition is the most efficient way to solve the normal equations. However, in practice, most of the computing time is spent in looping over the data to form the equations, and Gauss-Jordan is quite adequate.

We need to warn you that the solution of a least-squares problem directly from the normal equations is rather susceptible to roundoff error, because the condition number of the matrix $\boldsymbol{\alpha}$ is the square of the condition number of $\mathbf{A}$. An alternative, and preferred, technique involves QR decomposition (§2.10, §11.4, and §11.7) of the design matrix $\mathbf{A}$. This is essentially what we did at the end of §15.2 for fitting data to a straight line, but without invoking all the machinery of QR to derive the necessary formulas. Later in this section, we will discuss other difficulties in the least-squares problem, for which the cure is *singular value decomposition* (SVD), of which we give an implementation. It turns out that SVD also fixes the roundoff problem, so it is our recommended technique for all but "easy" least-squares problems. It is for these easy problems that the following routine, which solves the normal equations, is intended.

The object Fitlin, below, has one "value-added feature" that can be quite useful in practical work: Frequently it is a matter of art to decide which parameters a_k in a model should be fit from the data set, and which should be held constant at fixed values, for example values predicted by a theory or measured in a previous experiment. One wants, therefore, to have a convenient means for "freezing" and "unfreezing" the parameters a_k. In the following code, the total number of parameters a_k is denoted ma (called M above) and is deduced from the size of the vector that is returned by the user-supplied fitting function routine. The Fitlin object maintains a boolean array ia[0..ma-1]. Components that are false indicate that you want the corresponding elements of the parameter vector a[0..ma-1] to be held fixed at their input values. Components that are true indicate parameters that should be fitted for. On output, any frozen parameters will have their variances, and all their covariances, set to zero in the covariance matrix.

fitlin.h

```
struct Fitlin {
```

Object for general linear least-squares fitting by solving the normal equations, also including the ability to hold specified parameters at fixed, specified values. Call constructor to bind data vectors and fitting functions. Then call any combination of hold, free, and fit as often as desired. fit sets the output quantities a, covar, and chisq.

```
    Int ndat, ma;
    VecDoub_I &x,&y,&sig;
    VecDoub (*funcs)(const Doub);
    VecBool ia;

    VecDoub a;
    MatDoub covar;
    Doub chisq;
```

Output values. a is the vector of fitted coefficients, covar is its covariance matrix, and chisq is the value of χ^2 for the fit.

```
    Fitlin(VecDoub_I &xx, VecDoub_I &yy, VecDoub_I &ssig, VecDoub funks(const Doub))
```

```
: ndat(xx.size()), x(xx), y(yy), sig(ssig), funcs(funks) {
```
Constructor. Binds references to the data arrays `xx`, `yy`, and `ssig`, and to a user-supplied function `funks(x)` that returns a `VecDoub` containing `ma` basis functions evaluated at $x =$ `x`. Initializes all parameters as free (not held).
```
    ma = funcs(x[0]).size();
    a.resize(ma);
    covar.resize(ma,ma);
    ia.resize(ma);
    for (Int i=0;i<ma;i++) ia[i] = true;
}

void hold(const Int i, const Doub val) {ia[i]=false; a[i]=val;}
void free(const Int i) {ia[i]=true;}
```
Optional functions for holding a parameter, identified by a value `i` in the range $0,\ldots,$`ma-1`, fixed at the value `val`, or for freeing a parameter that was previously held fixed. `hold` and `free` may be called for any number of parameters before calling `fit` to calculate best-fit values for the remaining (not held) parameters, and the process may be repeated multiple times. Alternatively, you can set the boolean vector `ia` directly, before calling `fit`.

```
void fit() {
```
Solve the normal equations for χ^2 minimization to fit for some or all of the coefficients `a[0..ma-1]` of a function that depends linearly on `a`, $y = \sum_i$ a$_i$ $\times$ funks$_i(x)$. Set answer values for `a[0..ma-1]`, $\chi^2 =$ `chisq`, and the covariance matrix `covar[0..ma-1][0..ma-1]`. (Parameters held fixed by calls to `hold` will return zero covariances.)
```
    Int i,j,k,l,m,mfit=0;
    Doub ym,wt,sum,sig2i;
    VecDoub afunc(ma);
    for (j=0;j<ma;j++) if (ia[j]) mfit++;
    if (mfit == 0) throw("lfit: no parameters to be fitted");
    MatDoub temp(mfit,mfit,0.),beta(mfit,1,0.);
    for (i=0;i<ndat;i++) {                   Loop over data to accumulate coefficients of
        afunc = funcs(x[i]);                     the normal equations.
        ym=y[i];
        if (mfit < ma) {                     Subtract off dependences on known pieces
            for (j=0;j<ma;j++)                   of the fitting function.
                if (!ia[j]) ym -= a[j]*afunc[j];
        }
        sig2i=1.0/SQR(sig[i]);
        for (j=0,l=0;l<ma;l++) {             Set up matrix and r.h.s. for matrix inversion.
            if (ia[l]) {
                wt=afunc[l]*sig2i;
                for (k=0,m=0;m<=l;m++)
                    if (ia[m]) temp[j][k++] += wt*afunc[m];
                beta[j++][0] += ym*wt;
            }
        }
    }
    for (j=1;j<mfit;j++) for (k=0;k<j;k++) temp[k][j]=temp[j][k];
    gaussj(temp,beta);                       Matrix solution.
    for (j=0,l=0;l<ma;l++) if (ia[l]) a[l]=beta[j++][0];
```
Spread the solution to appropriate positions in `a`, and evaluate χ^2 of the fit.
```
    chisq=0.0;
    for (i=0;i<ndat;i++) {
        afunc = funcs(x[i]);
        sum=0.0;
        for (j=0;j<ma;j++) sum += a[j]*afunc[j];
        chisq += SQR((y[i]-sum)/sig[i]);
    }
    for (j=0;j<mfit;j++) for (k=0;k<mfit;k++) covar[j][k]=temp[j][k];
    for (i=mfit;i<ma;i++)                    Rearrange covariance matrix into the correct
        for (j=0;j<i+1;j++) covar[i][j]=covar[j][i]=0.0;       order.
    k=mfit-1;
    for (j=ma-1;j>=0;j--) {
            if (ia[j]) {
                for (i=0;i<ma;i++) SWAP(covar[i][k],covar[i][j]);
                for (i=0;i<ma;i++) SWAP(covar[k][i],covar[j][i]);
                k--;
            }
        }
    }
};
```

Typical use of `Fitlin` will look something like this:

```
const Int npts=...
VecDoub xx(npts),yy(npts),ssig(npts);
...
Fitlin myfit(xx,yy,ssig,cubicfit);
myfit.fit();
```

where (in this example) `cubicfit` is a user-supplied function that might look like this:

```
VecDoub cubicfit(const Doub x) {
    VecDoub ans(4);
    ans[0] = 1.;
    for (Int i=1;i<4;i++) ans[i] = x*ans[i-1];
    return ans;
}
```

15.4.2 Solution by Use of Singular Value Decomposition

In some applications, the normal equations are perfectly adequate for linear least-squares problems. However, in many other cases the normal equations are very close to singular. A zero pivot element may be encountered during the solution of the linear equations (e.g., in `gaussj`), in which case you get no solution at all. Or a very small pivot may occur, in which case you typically get fitted parameters a_k with very large magnitudes that are delicately (and unstably) balanced to cancel out almost precisely when the fitted function is evaluated.

Why does this commonly occur? A mathematical reason is that the condition number of the matrix $\boldsymbol{\alpha}$ is the square of the condition number of $\mathbf{A}$. But an additional physical reason is that, more often than experimenters would like to admit, data do not clearly distinguish between two or more of the basis functions provided. If two such functions, or two different combinations of functions, happen to fit the data about equally well — or equally badly — then the matrix $\boldsymbol{\alpha}$, unable to distinguish between them, neatly folds up its tent and becomes singular. There is a certain mathematical irony in the fact that least-squares problems are *both* overdetermined (number of data points greater than number of parameters) *and* underdetermined (ambiguous combinations of parameters exist); but that is how it frequently is. The ambiguities can be extremely hard to notice a priori in complicated problems.

Enter singular value decomposition (SVD). This would be a good time for you to review the material in §2.6, which we will not repeat here. In the case of an overdetermined system, SVD produces a solution that is the best approximation in the least-squares sense, cf. equation (2.6.10). That is exactly what we want. In the case of an underdetermined system, SVD produces a solution whose values (for us, the a_k's) are smallest in the least-squares sense, cf. equation (2.6.8). That is also what we want: When some combination of basis functions is irrelevant to the fit, that combination will be driven down to a small, innocuous, value, rather than pushed up to delicately canceling infinities.

In terms of the design matrix $\mathbf{A}$ (equation 15.4.4) and the vector $\mathbf{b}$ (equation 15.4.5), minimization of χ^2 in (15.4.3) can be written as

$$\text{find} \quad \mathbf{a} \quad \text{that minimizes} \quad \chi^2 = |\mathbf{A} \cdot \mathbf{a} - \mathbf{b}|^2 \tag{15.4.16}$$

Comparing to equation (2.6.9), we see that this is precisely the problem that routines in the `SVD` object are designed to solve. The solution, which is given by equation (2.6.12), can be rewritten as follows: If $\mathbf{U}$ and $\mathbf{V}$ enter the SVD decomposition of $\mathbf{A}$ according to equation (2.6.1), as computed by `SVD`, then let the vectors $\mathbf{U}_{(i)}$ $i = 0, \ldots, M-1$ denote the *columns* of $\mathbf{U}$ (each one a vector of length N), and let the vectors $\mathbf{V}_{(i)}$ $i = 0, \ldots, M-1$ denote the *columns* of $\mathbf{V}$ (each one a vector of length M). Then the solution (2.6.12) of the least-squares problem (15.4.16) can be written as

$$\mathbf{a} = \sum_{i=0}^{M-1} \left(\frac{\mathbf{U}_{(i)} \cdot \mathbf{b}}{w_i} \right) \mathbf{V}_{(i)} \tag{15.4.17}$$

where the w_i are, as in §2.6, the singular values calculated by `SVD`.

Equation (15.4.17) says that the fitted parameters $\mathbf{a}$ are linear combinations of the columns of $\mathbf{V}$, with coefficients obtained by forming dot products of the columns of $\mathbf{U}$ with the weighted data vector (15.4.5). Though it is beyond our scope to prove here, it turns out that the standard (loosely, "probable") errors in the fitted parameters are also linear combinations of the columns of $\mathbf{V}$. In fact, equation (15.4.17) can be written in a form displaying these errors as

$$\mathbf{a} = \left[\sum_{i=0}^{M-1} \left(\frac{\mathbf{U}_{(i)} \cdot \mathbf{b}}{w_i} \right) \mathbf{V}_{(i)} \right] \pm \frac{1}{w_0} \mathbf{V}_{(0)} \pm \cdots \pm \frac{1}{w_{M-1}} \mathbf{V}_{(M-1)} \tag{15.4.18}$$

Here each $\pm$ is followed by a standard deviation. The amazing fact is that, decomposed in this fashion, the standard deviations are all mutually independent (uncorrelated). Therefore they can be added together in root-mean-square fashion. What is going on is that the vectors $\mathbf{V}_{(i)}$ are the principal axes of the error ellipsoid of the fitted parameters $\mathbf{a}$ (see §15.6).

It follows that the variance in the estimate of a parameter a_j is given by

$$\sigma^2(a_j) = \sum_{i=0}^{M-1} \frac{1}{w_i^2} [\mathbf{V}_{(i)}]_j^2 = \sum_{i=0}^{M-1} \left(\frac{V_{ji}}{w_i} \right)^2 \tag{15.4.19}$$

whose result should be identical to (15.4.14). As before, you should not be surprised at the formula for the covariances, here given without proof,

$$\mathrm{Cov}(a_j, a_k) = \sum_{i=0}^{M-1} \left(\frac{V_{ji} V_{ki}}{w_i^2} \right) \tag{15.4.20}$$

We introduced this subsection by noting that the normal equations can fail by encountering a zero pivot. We have not yet, however, mentioned how SVD overcomes this problem. The answer is: If any singular value w_i is zero, its reciprocal in equation (15.4.18) should be set to zero, not infinity. (Compare the discussion preceding equation 2.6.7.) This corresponds to adding to the fitted parameters **a** a *zero* multiple, rather than some random large multiple, of any linear combination of basis functions that are degenerate in the fit. It is a good thing to do!

Moreover, if a singular value w_i is nonzero but very small, you should also define *its* reciprocal to be zero, since its apparent value is probably an artifact of roundoff error, not a meaningful number. A plausible answer to the question "how small is small?" is to edit in this fashion all singular values whose ratio to the largest singular value is less than N times the machine precision ϵ. (This is a more conservative recommendation than the default in §2.6, which scales as $N^{1/2}$.)

There is another reason for editing even *additional* singular values, ones large enough that roundoff error is not a question. Singular value decomposition allows you to identify linear combinations of variables that just happen not to contribute much to reducing the χ^2 of your data set. Editing these can sometimes reduce the probable error errors on your coefficients quite significantly, while increasing the minimum χ^2 only negligibly. We will learn more about identifying and treating such cases in §15.6.

Generally speaking, we recommend that you always use SVD techniques instead of using the normal equations. SVD's only significant disadvantage is that it requires extra storage of order $N \times M$ for the design matrix and its decomposition. Storage is also required for the $M \times M$ matrix $\mathbf{V}$, but this is instead of the same-sized matrix for the coefficients of the normal equations. SVD can be significantly slower than solving the normal equations; however, its great advantage, that it (theoretically) *cannot fail*, more than makes up for the speed disadvantage.

The following object, Fitsvd, has an interface almost identical to Fitlin, above. An additional optional parameter in the constructor sets the threshold for editing singular values.

```
struct Fitsvd {                                                        fitsvd.h
```
Object for general linear least-squares fitting using singular value decomposition. Call constructor to bind data vectors and fitting functions. Then call fit, which sets the output quantities a, covar, and chisq.
```
    Int ndat, ma;
    Doub tol;
    VecDoub_I *x,&y,&sig;               (Why is x a pointer? Explained in §15.4.4.)
    VecDoub (*funcs)(const Doub);
    VecDoub a;                          Output values. a is the vector of fitted coefficients,
    MatDoub covar;                          covar is its covariance matrix, and chisq is the
    Doub chisq;                             value of χ² for the fit.

    Fitsvd(VecDoub_I &xx, VecDoub_I &yy, VecDoub_I &ssig,
    VecDoub funks(const Doub), const Doub TOL=1.e-12)
    : ndat(yy.size()), x(&xx), xmd(NULL), y(yy), sig(ssig),
    funcs(funks), tol(TOL) {}
```
Constructor. Binds references to the data arrays xx, yy, and ssig, and to a user-supplied function funks(x) that returns a VecDoub containing ma basis functions evaluated at $x =$ x. If TOL is positive, it is the threshold (relative to the largest singular value) for discarding small singular values. If it is ≤ 0, the default value in SVD is used.

```
    void fit() {
```
Solve by singular value decomposition the χ^2 minimization that fits for the coefficients a[0..ma-1] of a function that depends linearly on a, $y = \sum_i \mathtt{a}_i \times \mathtt{funks}_i(x)$. Set answer values for a[0..ma-1], chisq $= \chi^2$, and the covariance matrix covar[0..ma-1][0..ma-1].

```
        Int i,j,k;
        Doub tmp,thresh,sum;
        if (x) ma = funcs((*x)[0]).size();
        else ma = funcsmd(row(*xmd,0)).size();          (Discussed in §15.4.4.)
        a.resize(ma);
        covar.resize(ma,ma);
        MatDoub aa(ndat,ma);
        VecDoub b(ndat),afunc(ma);
        for (i=0;i<ndat;i++) {                          Accumulate coefficients of the
            if (x) afunc=funcs((*x)[i]);                    design matrix.
            else afunc=funcsmd(row(*xmd,i));            (Discussed in §15.4.4.)
            tmp=1.0/sig[i];
            for (j=0;j<ma;j++) aa[i][j]=afunc[j]*tmp;
            b[i]=y[i]*tmp;
        }
        SVD svd(aa);                                    Singular value decomposition.
        thresh = (tol > 0. ? tol*svd.w[0] : -1.);
        svd.solve(b,a,thresh);                          Solve for the coefficients.
        chisq=0.0;                                      Evaluate chi-square.
        for (i=0;i<ndat;i++) {
            sum=0.;
            for (j=0;j<ma;j++) sum += aa[i][j]*a[j];
            chisq += SQR(sum-b[i]);
        }
        for (i=0;i<ma;i++) {                            Sum contributions to covariance
            for (j=0;j<i+1;j++) {                           matrix (15.4.20).
                sum=0.0;
                for (k=0;k<ma;k++) if (svd.w[k] > svd.tsh)
                    sum += svd.v[i][k]*svd.v[j][k]/SQR(svd.w[k]);
                covar[j][i]=covar[i][j]=sum;
            }
        }

    }
```

From here on, code for multidimensional fits, to be discussed in §15.4.4.

```
    MatDoub_I *xmd;
    VecDoub (*funcsmd)(VecDoub_I &);

    Fitsvd(MatDoub_I &xx, VecDoub_I &yy, VecDoub_I &ssig,
    VecDoub funks(VecDoub_I &), const Doub TOL=1.e-12)
    : ndat(yy.size()), x(NULL), xmd(&xx), y(yy), sig(ssig),
    funcsmd(funks), tol(TOL) {}
```

Constructor for multidimensional fits. Exactly the same as the previous constructor, except that `xx` is now a matrix whose rows are the multidimensional data points and `funks` is now a function of a multidimensional data point (as a `VecDoub`).

```
    VecDoub row(MatDoub_I &a, const Int i) {
```

Utility. Returns the row of a `MatDoub` as a `VecDoub`.

```
        Int j,n=a.ncols();
        VecDoub ans(n);
        for (j=0;j<n;j++) ans[j] = a[i][j];
        return ans;
    }
};
```

For degenerate or nearly degenerate problems, if you want to try different singular value thresholds, you call the `Fitsvd` constructor once. Then, as many times as you want, "reach in" and increase `tol`, then call `fit` again and examine the resulting value of `chisq` (and optionally also the covariance matrix). Keep going as long as `chisq` does not increase by too much. To learn what is "too much," see §15.6; but a few $\times 0.1$ is almost always OK.

15.4.3 Examples

Be aware that some apparently nonlinear problems can be expressed so that they are linear. For example, an exponential model with two parameters a and b,

$$y(x) = a\exp(-bx) \tag{15.4.21}$$

can be rewritten as

$$\log[y(x)] = c - bx \tag{15.4.22}$$

which is linear in its parameters c and b. (Of course you must be aware that such transformations do not exactly take Gaussian errors into Gaussian errors.)

Also watch out for "nonparameters," as in

$$y(x) = a\exp(-bx + d) \tag{15.4.23}$$

Here the parameters a and d are, in fact, indistinguishable. This is a good example of where the normal equations will be exactly singular, and where SVD will find a zero singular value. SVD will then make a least-squares choice for setting a balance between a and d (or, rather, their equivalents in the linear model derived by taking the logarithms). However — and this is true whenever SVD gives back a zero singular value — you are better advised to figure out analytically where the degeneracy is among your basis functions, and then make appropriate deletions in the basis set.

We already gave an example of a user-supplied fitting-function routine, `cubicfit`, above. Here are two further examples. First, we trivially generalize `cubicfit` for polynomials of an arbitrary, preset, degree:

fit_examples.h

```
Int fpoly_np = 10;                    Global variable for the degree plus one.

VecDoub fpoly(const Doub x) {
Fitting routine for a polynomial of degree fpoly_np-1.
    Int j;
    VecDoub p(fpoly_np);
    p[0]=1.0;
    for (j=1;j<fpoly_np;j++) p[j]=p[j-1]*x;
    return p;
}
```

The second example is slightly less trivial. It is used to fit Legendre polynomials up to some order `fleg_nl` to a data set. (Note that, for most uses, the data should satisfy $-1 \le x \le 1$.)

fit_examples.h

```
Int fleg_nl = 10;                     Global variable for the degree plus one.

VecDoub fleg(const Doub x) {
Fitting routine for an expansion with nl Legendre polynomials, evaluated using the recurrence
relation as in §5.4.
    Int j;
    Doub twox,f2,f1,d;
    VecDoub pl(fleg_nl);
    pl[0]=1.;
    pl[1]=x;
    if (fleg_nl > 2) {
        twox=2.*x;
        f2=x;
        d=1.;
        for (j=2;j<fleg_nl;j++) {
            f1=d++;
            f2+=twox;
            pl[j]=(f2*pl[j-1]-f1*pl[j-2])/d;
        }
    }
    return pl;
}
```

15.4.4 Multidimensional Fits

If you are measuring a single variable y as a function of more than one variable — say, a *vector* of variables $\mathbf{x}$ — then your basis functions will be functions of a vector, $X_0(\mathbf{x}), \dots, X_{M-1}(\mathbf{x})$. The χ^2 merit function is now

$$\chi^2 = \sum_{i=0}^{N-1} \left[\frac{y_i - \sum_{k=0}^{M-1} a_k X_k(\mathbf{x}_i)}{\sigma_i} \right]^2 \tag{15.4.24}$$

All of the preceding discussion goes through unchanged, with x replaced by $\mathbf{x}$. In fact, we anticipated this in the coding of `Fitsvd`, above, which can do multidimensional general linear fits as easily as one-dimensional. Here is how:

A second, overloaded, constructor in `Fitsvd` substitutes a matrix `xx` for what was previously a vector. The rows of the matrix are the `ndat` data points. The number of columns is the dimensionality of the space (that is, of $\mathbf{x}$). Similarly, the user-supplied function `funks` now takes a vector argument, an $\mathbf{x}$. A simple example (fitting a quadratic function to data in two dimensions) might be

```
    VecDoub quadratic2d(VecDoub_I &xx) {
        VecDoub ans(6);
        Doub x=xx[0], y=xx[1];
        ans[0] = 1;
        ans[1] = x; ans[2] = y;
        ans[3] = x*x; ans[4] = x*y; ans[5] = y*y;
        return ans;
    }
```

Be sure that the argument of your user function has exactly the type "`VecDoub_I &`" (and not, for example, "`VecDoub &`" or "`VecDoub_I`"), since strict C++ compilers are picky about this.

The two constructors in `Fitsvd` communicate to `fit` whether data points are one-dimensional or multidimensional by setting either `xmd` or `x` to `NULL`. This explains the oddity that `x` was bound to the user data as a pointer, while `y` and `sig` were bound as references. (Yes, we know this is a bit of a hack!)

CITED REFERENCES AND FURTHER READING:

Bevington, P.R., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill), Chapter 7.

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press), Chapter 11.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 1995 (Philadelphia: S.I.A.M.).

Monahan, J.F. 2001, *Numerical Methods of Statistics* (Cambridge, UK: Cambridge University Press), Chapter 5.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. 2004, *Bayesian Data Analysis*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC), Chapter 14.

15.5 Nonlinear Models

We now consider fitting when the model depends *nonlinearly* on the set of M unknown parameters $a_k, k = 0, 1, \ldots, M-1$. We use the same approach as in previous sections, namely to define a χ^2 merit function and determine best-fit parameters by its minimization. With nonlinear dependences, however, the minimization must proceed iteratively. Given trial values for the parameters, we develop a procedure that improves the trial solution. The procedure is then repeated until χ^2 stops (or effectively stops) decreasing.

How is this problem different from the general nonlinear function minimization problem already dealt with in Chapter 10? Superficially, not at all. Sufficiently close to the minimum, we expect the χ^2 function to be well approximated by a quadratic form, which we can write as

$$\chi^2(\mathbf{a}) \approx \gamma - \mathbf{d} \cdot \mathbf{a} + \tfrac{1}{2}\mathbf{a} \cdot \mathbf{D} \cdot \mathbf{a} \tag{15.5.1}$$

where $\mathbf{d}$ is an M-vector and $\mathbf{D}$ is an $M \times M$ matrix. (Compare equation 10.8.1.) If the approximation is a good one, we know how to jump from the current trial parameters $\mathbf{a}_{\text{cur}}$ to the minimizing ones $\mathbf{a}_{\text{min}}$ in a single leap, namely

$$\mathbf{a}_{\text{min}} = \mathbf{a}_{\text{cur}} + \mathbf{D}^{-1} \cdot \left[-\nabla\chi^2(\mathbf{a}_{\text{cur}})\right] \tag{15.5.2}$$

(Compare equation 10.9.4.)

On the other hand, (15.5.1) might be a poor local approximation to the shape of the function that we are trying to minimize at $\mathbf{a}_{\text{cur}}$. In that case, about all we can do is take a step down the gradient, as in the steepest descent method (§10.8). In other words,

$$\mathbf{a}_{\text{next}} = \mathbf{a}_{\text{cur}} - \text{constant} \times \nabla\chi^2(\mathbf{a}_{\text{cur}}) \tag{15.5.3}$$

where the constant is small enough not to exhaust the downhill direction.

To use (15.5.2) or (15.5.3), we must be able to compute the gradient of the χ^2 function at any set of parameters $\mathbf{a}$. To use (15.5.2) we also need the matrix $\mathbf{D}$, which is the second derivative matrix (Hessian matrix) of the χ^2 merit function, at any $\mathbf{a}$.

Now, this is the crucial difference from Chapter 10: There, we had no way of directly evaluating the Hessian matrix. We were given only the ability to evaluate the function to be minimized and (in some cases) its gradient. Therefore, we had to resort to iterative methods *not just* because our function was nonlinear, *but also* in order to build up information about the Hessian matrix. Sections 10.9 and 10.8 concerned themselves with two different techniques for building up this information.

Here, life is much simpler. We *know* exactly the form of χ^2, since it is based on a model function that we ourselves have specified. Therefore, the Hessian matrix is known to us. Thus we are free to use (15.5.2) whenever we care to do so. The only reason to use (15.5.3) will be failure of (15.5.2) to improve the fit, signaling failure of (15.5.1) as a good local approximation.

15.5.1 Calculation of the Gradient and Hessian

The model to be fitted is

$$y = y(x|\mathbf{a}) \tag{15.5.4}$$

and the χ^2 merit function is

$$\chi^2(\mathbf{a}) = \sum_{i=0}^{N-1} \left[\frac{y_i - y(x_i|\mathbf{a})}{\sigma_i} \right]^2 \tag{15.5.5}$$

The gradient of χ^2 with respect to the parameters $\mathbf{a}$, which will be zero at the χ^2 minimum, has components

$$\frac{\partial \chi^2}{\partial a_k} = -2 \sum_{i=0}^{N-1} \frac{[y_i - y(x_i|\mathbf{a})]}{\sigma_i^2} \frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \qquad k = 0, 1, \dots, M-1 \tag{15.5.6}$$

Taking an additional partial derivative gives

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = 2 \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \frac{\partial y(x_i|\mathbf{a})}{\partial a_l} - [y_i - y(x_i|\mathbf{a})] \frac{\partial^2 y(x_i|\mathbf{a})}{\partial a_l \partial a_k} \right] \tag{15.5.7}$$

It is conventional to remove the factors of 2 by defining

$$\beta_k \equiv -\frac{1}{2} \frac{\partial \chi^2}{\partial a_k} \qquad \alpha_{kl} \equiv \frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \tag{15.5.8}$$

making $\boldsymbol{\alpha} = \frac{1}{2}\mathbf{D}$ in equation (15.5.2), in terms of which that equation can be rewritten as the set of linear equations:

$$\sum_{l=0}^{M-1} \alpha_{kl}\, \delta a_l = \beta_k \tag{15.5.9}$$

This set is solved for the increments δa_l that, added to the current approximation, give the next approximation. In the context of least squares, the matrix $\boldsymbol{\alpha}$, equal to one-half times the Hessian matrix, is usually called the *curvature matrix*.

Equation (15.5.3), the steepest descent formula, translates to

$$\delta a_l = \text{constant} \times \beta_l \tag{15.5.10}$$

Note that the components α_{kl} of the Hessian matrix (15.5.7) depend both on the first derivatives and on the second derivatives of the basis functions with respect to their parameters. Some treatments proceed to ignore the second derivative without comment. We will ignore it also, but only *after* a few comments.

Second derivatives occur because the gradient (15.5.6) already has a dependence on $\partial y/\partial a_k$, and so the next derivative simply must contain terms involving $\partial^2 y/\partial a_l \partial a_k$. The second derivative term can be dismissed when it is zero (as in the linear case of equation 15.4.8) or small enough to be negligible when compared to the term involving the first derivative. It also has an additional possibility of being ignorably small in practice: The term multiplying the second derivative in equation

(15.5.7) is $[y_i - y(x_i|\mathbf{a})]$. For a successful model, this term should just be the random measurement error of each point. This error can have either sign, and should in general be uncorrelated with the model. Therefore, the second derivative terms tend to cancel out when summed over i.

Inclusion of the second derivative term can in fact be destabilizing if the model fits badly or is contaminated by outlier points that are unlikely to be offset by compensating points of opposite sign. From this point on, we will always use as the definition of α_{kl} the formula

$$\alpha_{kl} = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \frac{\partial y(x_i|\mathbf{a})}{\partial a_l} \right] \tag{15.5.11}$$

This expression more closely resembles its linear cousin (15.4.8). You should understand that minor (or even major) fiddling with $\boldsymbol{\alpha}$ has no effect at all on what final set of parameters $\mathbf{a}$ is reached, but affects only the iterative route that is taken in getting there. The condition at the χ^2 minimum, that $\beta_k = 0$ for all k, is independent of how $\boldsymbol{\alpha}$ is defined.

15.5.2 Levenberg-Marquardt Method

Marquardt [1] put forth an elegant method, related to an earlier suggestion of Levenberg, for varying smoothly between the extremes of the inverse-Hessian method (15.5.9) and the steepest descent method (15.5.10). The latter method is used far from the minimum, switching continuously to the former as the minimum is approached. This *Levenberg-Marquardt method* (also called the *Marquardt method*) works very well in practice if you can guess plausible starting guesses for your parameters. It has become a standard nonlinear least-squares routine.

The method is based on two elementary, but important, insights. Consider the "constant" in equation (15.5.10). What should it be, even in order of magnitude? What sets its scale? There is no information about the answer in the gradient. That tells only the slope, not how far that slope extends. Marquardt's first insight is that the components of the Hessian matrix, even if they are not usable in any precise fashion, give *some* information about the order-of-magnitude scale of the problem.

The quantity χ^2 is nondimensional, i.e., is a pure number; this is evident from its definition (15.5.5). On the other hand, β_k has the dimensions of $1/a_k$, which may well be dimensional, i.e., have units like cm^{-1}, or kilowatt-hours, or whatever. (In fact, each component of β_k can have different dimensions!) The constant of proportionality between β_k and δa_k must therefore have the dimensions of a_k^2. Scan the components of $\boldsymbol{\alpha}$ and you see that there is only one obvious quantity with these dimensions, and that is $1/\alpha_{kk}$, the reciprocal of the diagonal element. So that must set the scale of the constant. But that scale might itself be too big. So let's divide the constant by some (nondimensional) fudge factor λ, with the possibility of setting $\lambda \gg 1$ to cut down the step. In other words, replace equation (15.5.10) by

$$\delta a_l = \frac{1}{\lambda \alpha_{ll}} \beta_l \qquad \text{or} \qquad \lambda\, \alpha_{ll}\, \delta a_l = \beta_l \tag{15.5.12}$$

It is necessary that α_{ll} be positive, but this is guaranteed by definition (15.5.11) — another reason for adopting that equation.

Marquardt's second insight is that equations (15.5.12) and (15.5.9) can be combined if we define a new matrix α' by the following prescription:

$$\begin{aligned} \alpha'_{jj} &\equiv \alpha_{jj}(1+\lambda) \\ \alpha'_{jk} &\equiv \alpha_{jk} \qquad (j \neq k) \end{aligned} \tag{15.5.13}$$

and then replace both (15.5.12) and (15.5.9) by

$$\sum_{l=0}^{M-1} \alpha'_{kl}\,\delta a_l = \beta_k \tag{15.5.14}$$

When λ is very large, the matrix α' is forced into being *diagonally dominant*, so equation (15.5.14) goes over to be identical to (15.5.12). On the other hand, as λ approaches zero, equation (15.5.14) goes over to (15.5.9).

Given an initial guess for the set of fitted parameters **a**, the recommended Marquardt recipe is as follows:

- Compute $\chi^2(\mathbf{a})$.
- Pick a modest value for λ, say $\lambda = 0.001$.
- (†) Solve the linear equations (15.5.14) for $\delta\mathbf{a}$ and evaluate $\chi^2(\mathbf{a}+\delta\mathbf{a})$.
- If $\chi^2(\mathbf{a}+\delta\mathbf{a}) \geq \chi^2(\mathbf{a})$, *increase* λ by a factor of 10 (or any other substantial factor) and go back to (†).
- If $\chi^2(\mathbf{a}+\delta\mathbf{a}) < \chi^2(\mathbf{a})$, *decrease* λ by a factor of 10, update the trial solution $\mathbf{a} \leftarrow \mathbf{a}+\delta\mathbf{a}$, and go back to (†).

Also necessary is a condition for stopping. Iterating to convergence (to machine accuracy or to the roundoff limit) is generally wasteful and unnecessary since the minimum is at best only a statistical estimate of the parameters **a**. As we will see in §15.6, a change in the parameters that changes χ^2 by an amount $\ll 1$ is *never* statistically meaningful.

Furthermore, it is not uncommon to find the parameters wandering around near the minimum in a flat valley of complicated topography. The reason is that Marquardt's method generalizes the method of normal equations (§15.4); hence it has the same problem as that method with regard to near-degeneracy of the minimum. Outright failure by a zero pivot is possible, but unlikely. More often, a small pivot will generate a large correction that is then rejected, the value of λ being then increased. For sufficiently large λ, the matrix $\boldsymbol{\alpha}'$ is positive-definite and can have no small pivots. Thus the method does tend to stay away from zero pivots, but at the cost of a tendency to wander around doing steepest descent in very unsteep degenerate valleys.

These considerations suggest that, in practice, one might as well stop iterating after a few occurrences of χ^2 decreasing by a negligible amount, say either less than 0.001 absolutely or (in case roundoff prevents that being reached) fractionally. Don't stop after a step where χ^2 *increases* more than trivially: That only shows that λ has not yet adjusted itself optimally.

Once the acceptable minimum has been found, one wants to set $\lambda = 0$ and compute the matrix

$$\mathbf{C} \equiv \boldsymbol{\alpha}^{-1} \tag{15.5.15}$$

which, as before, is the estimated covariance matrix of the standard errors in the fitted parameters **a** (see next section).

The following object, Fitmrq, implements Marquardt's method for nonlinear parameter estimation. The user interface is intentionally very close to that of Fitlin in §15.4. In particular, the feature of being able to freeze or unfreeze chosen parameters is available here, too.

One difference from Fitlin is that you have to supply an initial guess for the parameters **a**. Now *that* is a can of worms! When you are fitting for parameters that enter highly nonlinearly, there is no reason in the world that the χ^2 surface should have only a single minimum. Marquardt's method embodies no magical insight into finding the global minimum; it's just a downhill search. Often, it should be the endgame strategy for fitting parameters, preceded by perhaps cruder, and likely problem-specific, methods for getting into the right general basin of convergence.

Another difference between Fitmrq and Fitlin is the format of the user-supplied function funks. Since Fitmrq needs both function and gradient values, funks is now coded as a void function returning answers through arguments passed by reference. An example is given below. You call Fitmrq's constructor once, to bind your data vectors and function. Then (after any optional calls to hold or free) you call fit, which sets values for a, chisq, and covar. The curvature matrix alpha is also available. Note that the original vector of parameter guesses that you send to the constructor is not modified; rather, the answer is returned in a.

fitmrq.h

```
struct Fitmrq {
```
Object for nonlinear least-squares fitting by the Levenberg-Marquardt method, also including the ability to hold specified parameters at fixed, specified values. Call constructor to bind data vectors and fitting functions and to input an initial parameter guess. Then call any combination of hold, free, and fit as often as desired. fit sets the output quantities a, covar, alpha, and chisq.
```
    static const Int NDONE=4, ITMAX=1000;      Convergence parameters.
    Int ndat, ma, mfit;
    VecDoub_I &x,&y,&sig;
    Doub tol;
    void (*funcs)(const Doub, VecDoub_I &, Doub &, VecDoub_O &);
    VecBool ia;
    VecDoub a;                           Output values. a is the vector of fitted coefficients,
    MatDoub covar;                           covar is its covariance matrix, alpha is the cur-
    MatDoub alpha;                           vature matrix, and chisq is the value of χ² for
    Doub chisq;                              the fit.

    Fitmrq(VecDoub_I &xx, VecDoub_I &yy, VecDoub_I &ssig, VecDoub_I &aa,
    void funks(const Doub, VecDoub_I &, Doub &, VecDoub_O &), const Doub
    TOL=1.e-3) : ndat(xx.size()), ma(aa.size()), x(xx), y(yy), sig(ssig),
    tol(TOL), funcs(funks), ia(ma), alpha(ma,ma), a(aa), covar(ma,ma) {
```
Constructor. Binds references to the data arrays xx, yy, and ssig, and to a user-supplied function funks that calculates the nonlinear fitting function and its derivatives. Also inputs the initial parameters guess aa (which is copied, not modified) and an optional convergence tolerance TOL. Initializes all parameters as free (not held).
```
        for (Int i=0;i<ma;i++) ia[i] = true;
    }

    void hold(const Int i, const Doub val) {ia[i]=false; a[i]=val;}
    void free(const Int i) {ia[i]=true;}
```
Optional functions for holding a parameter, identified by a value i in the range 0,...,ma-1, fixed at the value val, or for freeing a parameter that was previously held fixed. hold and free may be called for any number of parameters before calling fit to calculate best-fit values for the remaining (not held) parameters, and the process may be repeated multiple times.
```
    void fit() {
```

Iterate to reduce the χ^2 of a fit between a set of data points x[0..ndat-1], y[0..ndat-1] with individual standard deviations sig[0..ndat-1], and a nonlinear function that depends on ma coefficients a[0..ma-1]. When χ^2 is no longer decreasing, set best-fit values for the parameters a[0..ma-1], and chisq $= \chi^2$, covar[0..ma-1][0..ma-1], and alpha[0..ma-1][0..ma-1]. (Parameters held fixed will return zero covariances.)

```
    Int j,k,l,iter,done=0;
    Doub alamda=.001,ochisq;
    VecDoub atry(ma),beta(ma),da(ma);
    mfit=0;
    for (j=0;j<ma;j++) if (ia[j]) mfit++;
    MatDoub oneda(mfit,1), temp(mfit,mfit);
    mrqcof(a,alpha,beta);                Initialization.
    for (j=0;j<ma;j++) atry[j]=a[j];
    ochisq=chisq;
    for (iter=0;iter<ITMAX;iter++) {
        if (done==NDONE) alamda=0.;          Last pass. Use zero alamda.
        for (j=0;j<mfit;j++) {           Alter linearized fitting matrix, by augmenting di-
            for (k=0;k<mfit;k++) covar[j][k]=alpha[j][k];    agonal elements.
            covar[j][j]=alpha[j][j]*(1.0+alamda);
            for (k=0;k<mfit;k++) temp[j][k]=covar[j][k];
            oneda[j][0]=beta[j];
        }
        gaussj(temp,oneda);              Matrix solution.
        for (j=0;j<mfit;j++) {
            for (k=0;k<mfit;k++) covar[j][k]=temp[j][k];
            da[j]=oneda[j][0];
        }
        if (done==NDONE) {               Converged. Clean up and return.
            covsrt(covar);
            covsrt(alpha);
            return;
        }
        for (j=0,l=0;l<ma;l++)           Did the trial succeed?
            if (ia[l]) atry[l]=a[l]+da[j++];
        mrqcof(atry,covar,da);
        if (abs(chisq-ochisq) < MAX(tol,tol*chisq)) done++;
        if (chisq < ochisq) {            Success, accept the new solution.
            alamda *= 0.1;
            ochisq=chisq;
            for (j=0;j<mfit;j++) {
                for (k=0;k<mfit;k++) alpha[j][k]=covar[j][k];
                    beta[j]=da[j];
            }
            for (l=0;l<ma;l++) a[l]=atry[l];
        } else {                         Failure, increase alamda.
            alamda *= 10.0;
            chisq=ochisq;
        }
    }
    throw("Fitmrq too many iterations");
}


void mrqcof(VecDoub_I &a, MatDoub_O &alpha, VecDoub_O &beta) {
```

Used by fit to evaluate the linearized fitting matrix alpha, and vector beta as in (15.5.8), and to calculate χ^2.

```
    Int i,j,k,l,m;
    Doub ymod,wt,sig2i,dy;
    VecDoub dyda(ma);
    for (j=0;j<mfit;j++) {               Initialize (symmetric) alpha, beta.
        for (k=0;k<=j;k++) alpha[j][k]=0.0;
        beta[j]=0.;
    }
```

```
        chisq=0.;
        for (i=0;i<ndat;i++) {          Summation loop over all data.
            funcs(x[i],a,ymod,dyda);
            sig2i=1.0/(sig[i]*sig[i]);
            dy=y[i]-ymod;
            for (j=0,l=0;l<ma;l++) {
                if (ia[l]) {
                    wt=dyda[l]*sig2i;
                    for (k=0,m=0;m<l+1;m++)
                        if (ia[m]) alpha[j][k++] += wt*dyda[m];
                    beta[j++] += dy*wt;
                }
            }
            chisq += dy*dy*sig2i;       And find χ².
        }
        for (j=1;j<mfit;j++)            Fill in the symmetric side.
            for (k=0;k<j;k++) alpha[k][j]=alpha[j][k];
    }

    void covsrt(MatDoub_IO &covar) {
```
Expand in storage the covariance matrix covar, so as to take into account parameters that are being held fixed. (For the latter, return zero covariances.)
```
        Int i,j,k;
        for (i=mfit;i<ma;i++)
            for (j=0;j<i+1;j++) covar[i][j]=covar[j][i]=0.0;
        k=mfit-1;
        for (j=ma-1;j>=0;j--) {
            if (ia[j]) {
                for (i=0;i<ma;i++) SWAP(covar[i][k],covar[i][j]);
                for (i=0;i<ma;i++) SWAP(covar[k][i],covar[j][i]);
                k--;
            }
        }
    }

};
```

15.5.3 Example

The following function fgauss is an example of a user-supplied function funks. Used with Fitmrq, it fits for the model

$$y(x) = \sum_{k=0}^{K-1} B_k \exp\left[-\left(\frac{x - E_k}{G_k}\right)^2\right] \tag{15.5.16}$$

which is a sum of K Gaussians, each with a variable position, amplitude, and width. We store the parameters in the order $B_0, E_0, G_0, B_1, E_1, G_1, \ldots, B_{K-1}, E_{K-1}, G_{K-1}$.

```
void fgauss(const Doub x, VecDoub_I &a, Doub &y, VecDoub_O &dyda) {          fit_examples.h
```
$y(x;a)$ is the sum of na/3 Gaussians (15.5.16). The amplitude, center, and width of the Gaussians are stored in consecutive locations of a: a[$3k$] $= B_k$, a[$3k+1$] $= E_k$, a[$3k+2$] $= G_k$, $k = 0, \ldots,$ na$/3 - 1$. The dimensions of the arrays are a[0..na-1], dyda[0..na-1].
```
    Int i,na=a.size();
    Doub fac,ex,arg;
    y=0.;
    for (i=0;i<na-1;i+=3) {
        arg=(x-a[i+1])/a[i+2];
        ex=exp(-SQR(arg));
```

```
            fac=a[i]*ex*2.*arg;
            y += a[i]*ex;
            dyda[i]=ex;
            dyda[i+1]=fac/a[i+2];
            dyda[i+2]=fac*arg/a[i+2];
        }
}
```

15.5.4 More Advanced Methods for Nonlinear Least Squares

You will need more capability than `Fitmrq` can supply if either (i) it is converging too slowly, or (ii) it is converging to a local minimum that is not the one you want. Several options are available.

NL2SOL [3] is a highly regarded nonlinear least-squares implementation with many advanced features. For example, it keeps the second-derivative term we dropped in the Levenberg-Marquardt method whenever it would be better to do so, a so-called *full Newton-type* method.

A different variant on the Levenberg-Marquardt algorithm is to implement it as a model-trust region method for minimization (see §9.7 and ref. [2]) applied to the special case of a least-squares function. A code of this kind due to Moré [4] can be found in MINPACK [5].

CITED REFERENCES AND FURTHER READING:

Bevington, P.R., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill), Chapter 8.

Monahan, J.F. 2001, *Numerical Methods of Statistics* (Cambridge, UK: Cambridge University Press), Chapters 5–9.

Seber, G.A.F., and Wild, C.J. 2003, *Nonlinear Regression* (Hoboken, NJ: Wiley).

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. 2004, *Bayesian Data Analysis*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC).

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.2 (by J.E. Dennis).

Marquardt, D.W. 1963, *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, pp. 431–441.[1]

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.).[2]

Dennis, J.E., Gay, D.M, and Welsch, R.E. 1981, "An Adaptive Nonlinear Least-Squares Algorithm," *ACM Transactions on Mathematical Software*, vol. 7, pp. 348–368; *op. cit.*, pp. 369–383.[3].

Moré, J.J. 1977, in *Numerical Analysis*, Lecture Notes in Mathematics, vol. 630, G.A. Watson, ed. (Berlin: Springer), pp. 105–116.[4]

Moré, J.J., Garbow, B.S., and Hillstrom, K.E. 1980, *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74.[5]

15.6 Confidence Limits on Estimated Model Parameters

Several times already in this chapter we have made statements about the standard errors, or uncertainties, in a set of M estimated parameters $\mathbf{a}$. We have given some formulas for computing standard deviations or variances of individual parameters (equations 15.2.9, 15.4.15, and 15.4.19), as well as some formulas for covariances between pairs of parameters (equation 15.2.10; remark following equation 15.4.15; equation 15.4.20; equation 15.5.15).

In this section, we want to be more explicit regarding the precise meaning of these quantitative uncertainties, and to give further information about how quantitative confidence limits on fitted parameters can be estimated. The subject can get somewhat technical, and even somewhat confusing, so we will try to make precise statements, even when they must be offered without proof.

Figure 15.6.1 shows the conceptual scheme of an experiment that "measures" a set of parameters. There is some underlying true set of parameters $\mathbf{a}_{\text{true}}$ that are known to Mother Nature but hidden from the experimenter. These true parameters are statistically realized, along with random measurement errors, as a measured data set, which we will symbolize as $\mathcal{D}_{(0)}$. The data set $\mathcal{D}_{(0)}$ *is* known to the experimenter. He or she fits the data to a model by χ^2 minimization or some other technique and obtains measured, i.e., fitted, values for the parameters, which we here denote $\mathbf{a}_{(0)}$.

Because measurement errors have a random component, $\mathcal{D}_{(0)}$ is not a unique realization of the true parameters $\mathbf{a}_{\text{true}}$. Rather, there are infinitely many other realizations of the true parameters as "hypothetical data sets" each of which *could* have been the one measured, but happened not to be. Let us symbolize these by $\mathcal{D}_{(1)}, \mathcal{D}_{(2)}, \ldots$. Each one, had it been realized, would have given a slightly different set of fitted parameters, $\mathbf{a}_{(1)}, \mathbf{a}_{(2)}, \ldots$, respectively. These parameter sets $\mathbf{a}_{(i)}$ therefore occur with some probability distribution in the M-dimensional space of all possible parameter sets $\mathbf{a}$. The actual measured set $\mathbf{a}_{(0)}$ is one member drawn from this distribution.

Even more interesting than the probability distribution of $\mathbf{a}_{(i)}$ would be the distribution of the difference $\mathbf{a}_{(i)} - \mathbf{a}_{\text{true}}$. This distribution differs from the former one by a translation that puts Mother Nature's true value at the origin. If we knew *this* distribution, we would know everything that there is to know about the quantitative uncertainties in our experimental measurement $\mathbf{a}_{(0)}$.

So the name of the game is to find some way of estimating or approximating the probability distribution of $\mathbf{a}_{(i)} - \mathbf{a}_{\text{true}}$ without knowing $\mathbf{a}_{\text{true}}$ and without having available to us an infinite universe of hypothetical data sets.

15.6.1 Monte Carlo Simulation of Synthetic Data Sets

Although the measured parameter set $\mathbf{a}_{(0)}$ is not the true one, let us consider a fictitious world in which it *was* the true one. Since we hope that our measured parameters are not *too* wrong, we hope that that fictitious world is not too different from the actual world with parameters $\mathbf{a}_{\text{true}}$. In particular, let us hope — no, let us *assume* — that the shape of the probability distribution $\mathbf{a}_{(i)} - \mathbf{a}_{(0)}$ in the fictitious world is the same, or very nearly the same, as the shape of the probability distribution

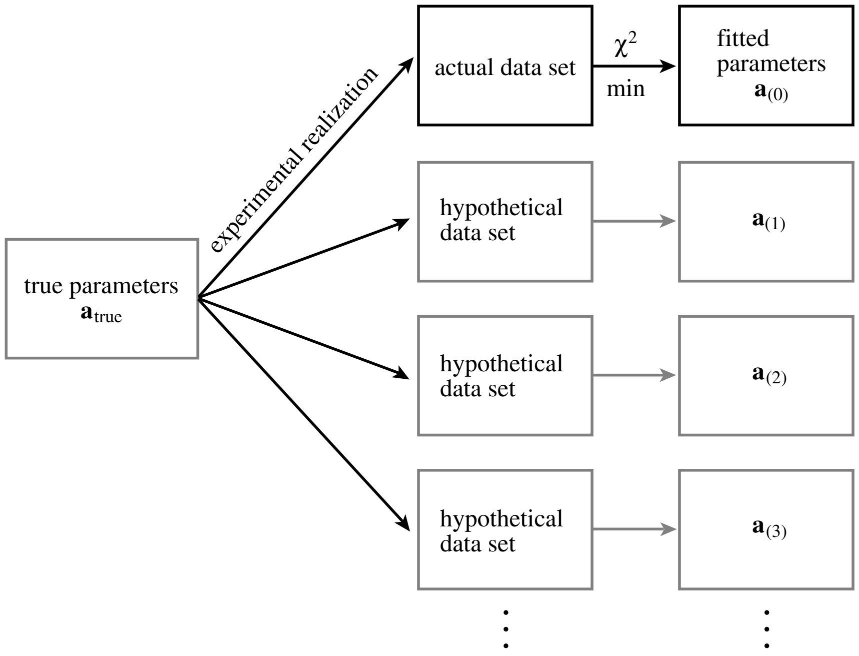


Figure 15.6.1. A statistical universe of data sets from an underlying model. True parameters $\mathbf{a}_{\text{true}}$ are realized in a data set, from which fitted (observed) parameters $\mathbf{a}_{(0)}$ are obtained. If the experiment were repeated many times, new data sets and new values of the fitted parameters would be obtained.

$\mathbf{a}_{(i)} - \mathbf{a}_{\text{true}}$ in the real world. Notice that we are not assuming that $\mathbf{a}_{(0)}$ and $\mathbf{a}_{\text{true}}$ are equal; they are certainly not. We are only assuming that the way in which random errors enter the experiment and data analysis does not vary rapidly as a function of $\mathbf{a}_{\text{true}}$, so that $\mathbf{a}_{(0)}$ can serve as a reasonable surrogate.

Now, often, the distribution of $\mathbf{a}_{(i)} - \mathbf{a}_{(0)}$ in the fictitious world *is* within our power to calculate (see Figure 15.6.2). If we know something about the process that generated our data, given an assumed set of parameters $\mathbf{a}_{(0)}$, then we can usually figure out how to *simulate* our own sets of "synthetic" realizations of these parameters as "synthetic data sets." The procedure is to draw random numbers from appropriate distributions (cf. §7.3 – §7.4) so as to mimic our best understanding of the underlying process and measurement errors in our apparatus. With such random draws, we construct data sets with exactly the same numbers of measured points, and precisely the same values of all control (independent) variables, as our actual data set $\mathcal{D}_{(0)}$. Let us call these simulated data sets $\mathcal{D}^S_{(1)}, \mathcal{D}^S_{(2)}, \ldots$. By construction, these are supposed to have exactly the same statistical relationship to $\mathbf{a}_{(0)}$ as the $\mathcal{D}_{(i)}$'s have to $\mathbf{a}_{\text{true}}$. (For the case where you don't know enough about what you are measuring to do a credible job of simulating it, see below.)

Next, for each $\mathcal{D}^S_{(j)}$, perform exactly the same procedure for estimation of parameters, e.g., χ^2 minimization, as was performed on the actual data to get the parameters $\mathbf{a}_{(0)}$, giving simulated measured parameters $\mathbf{a}^S_{(1)}, \mathbf{a}^S_{(2)}, \ldots$. Each simulated measured parameter set yields a point $\mathbf{a}^S_{(i)} - \mathbf{a}_{(0)}$. Simulate enough data sets and enough derived simulated measured parameters, and you map out the desired probability distribution in M dimensions.

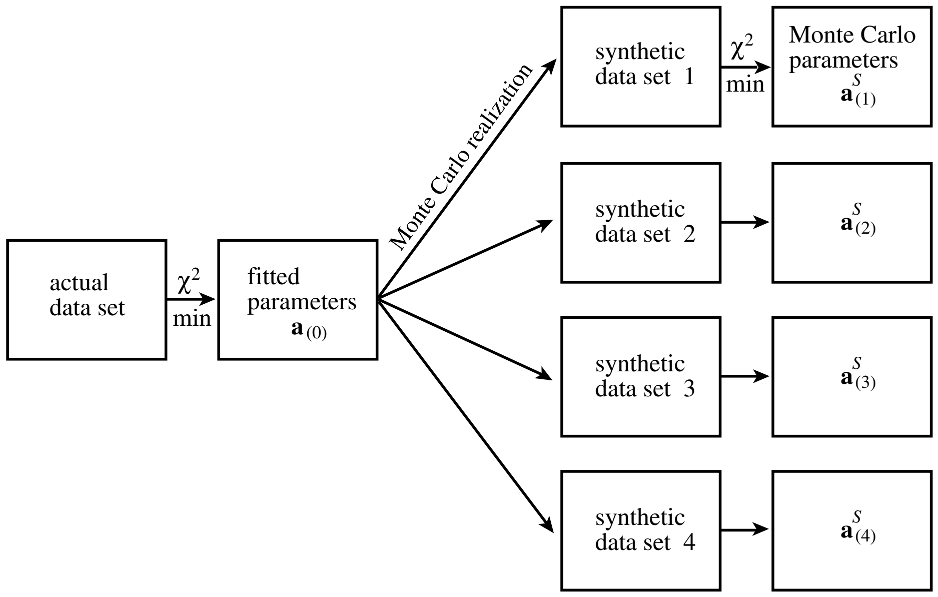


Figure 15.6.2. Monte Carlo simulation of an experiment. The fitted parameters from an actual experiment are used as surrogates for the true parameters. Computer-generated random numbers are used to simulate many synthetic data sets. Each of these is analyzed to obtain its fitted parameters. The distribution of these fitted parameters around the (known) surrogate true parameters is thus studied.

In fact, the ability to do *Monte Carlo simulations* in this fashion has revolutionized many fields of modern experimental science. Not only is one able to characterize the errors of parameter estimation in a very precise way; one can also try out on the computer different methods of parameter estimation, or different data reduction techniques, and seek to minimize the uncertainty of the result according to any desired criteria. Offered the choice between mastery of a five-foot shelf of analytical statistics books and middling ability at performing statistical Monte Carlo simulations, we would surely choose to have the latter skill.

15.6.2 Quick-and-Dirty Monte Carlo: The Bootstrap Method

Here is a powerful technique that can often be used when you don't know enough about the underlying process, or the nature of your measurement errors, to do a credible Monte Carlo simulation. Suppose that your data set consists of *N independent and identically distributed* (or *iid*) "data points." Each data point probably consists of several numbers, e.g., one or more control variables (uniformly distributed, say, in the range that you have decided to measure) and one or more associated measured values (each distributed however Mother Nature chooses). "Iid" means that the sequential order of the data points is not of consequence to the process that you are using to get the fitted parameters $\mathbf{a}$. For example, a χ^2 sum like (15.5.5) does not care in what order the points are added. Even simpler examples are the mean value of a measured quantity and the mean of some function of the measured quantities.

The *bootstrap method* [1] uses the actual data set $\mathcal{D}^S_{(0)}$, with its N data points, to generate any number of synthetic data sets $\mathcal{D}^S_{(1)}, \mathcal{D}^S_{(2)}, \ldots$, also with N data points. The procedure is simply to draw N data points at a time *with replacement* from the set $\mathcal{D}^S_{(0)}$. Because of the replacement, you do not simply get back your original data set each time. You get sets in which a random fraction of the original points, typically $\sim 1/e \approx 37\%$, are replaced by *duplicated* original points. Now, exactly as in the previous discussion, you subject these data sets to the same estimation procedure as was performed on the actual data, giving a set of simulated measured parameters $\mathbf{a}^S_{(1)}, \mathbf{a}^S_{(2)}, \ldots$. These will be distributed around $\mathbf{a}_{(0)}$ in close to the same way that $\mathbf{a}_{(0)}$ is distributed around $\mathbf{a}_{\text{true}}$.

Sounds like getting something for nothing, doesn't it? In fact, it took a while for the bootstrap method to become accepted by statisticians. By now, however, enough theorems have been proved to render the bootstrap reputable (see [2] for references). The basic idea behind the bootstrap is that the actual data set, viewed as a probability distribution consisting of delta functions at the measured values, is in most cases the best — or only — available estimator of the underlying probability distribution. It takes courage, but one can often simply use *that* distribution as the basis for Monte Carlo simulations.

Watch out for cases where the bootstrap's iid assumption is violated. For example, if you have made measurements at evenly spaced intervals of some control variable, then you can *usually* get away with pretending that these are iid uniformly distributed over the measured range. However, some estimators of $\mathbf{a}$ (e.g., ones involving Fourier methods) might be particularly sensitive to all the points on a grid being present. In that case, the bootstrap is going to give a wrong distribution. Also watch out for estimators that look at anything like small-scale clumpiness within the N data points, or estimators that sort the data and look at sequential differences. Obviously the bootstrap will fail on these, too. (The theorems justifying the method are still true, but some of their technical assumptions are violated by these examples.)

For a large class of problems, however, the bootstrap does yield easy, *very quick*, Monte Carlo estimates of the errors in an estimated parameter set.

15.6.3 Confidence Limits

Rather than present all details of the probability distribution of errors in parameter estimation, it is common practice to summarize the distribution in the form of *confidence limits*. The full probability distribution is a function defined on the M-dimensional space of parameters $\mathbf{a}$. A *confidence region* (or *confidence interval*) is just a region of that M-dimensional space (hopefully a small region) that contains a certain (hopefully large) percentage of the total probability distribution. You point to a confidence region and say, e.g., "there is a 99% chance that the true parameter values fall within this region around the measured value."

It is worth emphasizing that you, the experimenter, get to pick both the *confidence level* (99% in the above example) and the shape of the confidence region. The only requirement is that your region does include the stated percentage of probability. Certain percentages are, however, customary in scientific usage: 68.3% (the lowest confidence worthy of quoting), 90%, 95.4%, 99%, and 99.73%. Higher confidence levels are conventionally "ninety-nine point nine ... nine." As for shape, obviously you want a region that is compact and reasonably centered on your mea-

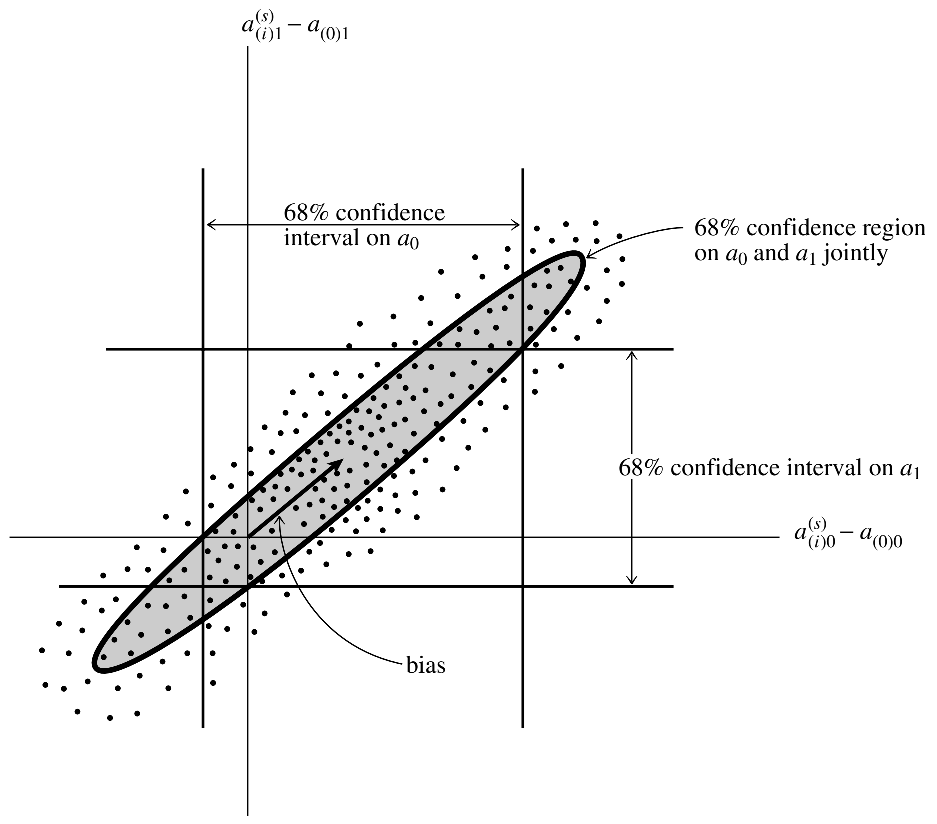

$a^{(s)}_{(i)1} - a_{(0)1}$
68% confidence interval on a_0
68% confidence region on a_0 and a_1 jointly
68% confidence interval on a_1
$a^{(s)}_{(i)0} - a_{(0)0}$
bias

Figure 15.6.3. Confidence intervals in one and two dimensions. The same fraction of measured points (here 68%) lies (i) between the two vertical lines, (ii) between the two horizontal lines, (iii) within the ellipse.

surement $\mathbf{a}_{(0)}$, since the whole purpose of a confidence limit is to inspire confidence in that measured value. In one dimension, the convention is to use a line segment centered on the measured value; in higher dimensions, ellipses or ellipsoids are most frequently used.

You might suspect, correctly, that the numbers 68.3%, 95.4%, and 99.73%, and the use of ellipsoids, have some connection with a normal distribution. That is true historically, but not always relevant nowadays. In general, the probability distribution of the parameters will not be normal, and the above numbers, used as levels of confidence, are purely matters of convention.

Figure 15.6.3 sketches a possible probability distribution for the case $M = 2$. Shown are three different confidence regions that might usefully be given, all at the same confidence level. The two vertical lines enclose a band (horizontal interval) that represents the 68% confidence interval for the variable a_0 without regard to the value of a_1. Similarly the horizontal lines enclose a 68% confidence interval for a_1. The ellipse shows a 68% confidence interval for a_0 and a_1 jointly. Notice that to enclose the same probability as the two bands, the ellipse must necessarily extend outside of both of them (a point we will return to below).

15.6.4 Constant Chi-Square Boundaries as Confidence Limits

When the method used to estimate the parameters $\mathbf{a}_{(0)}$ is chi-square minimization, as in the previous sections of this chapter, then there is a natural choice for the shape of confidence intervals, whose use is almost universal. For the observed data set $\mathcal{D}_{(0)}$, the value of χ^2 is a minimum at $\mathbf{a}_{(0)}$. Call this minimum value $\chi^2_{\min}$. If the vector $\mathbf{a}$ of parameter values is perturbed away from $\mathbf{a}_{(0)}$, then χ^2 increases. The region within which χ^2 increases by no more than a set amount $\Delta\chi^2$ defines some M-dimensional confidence region around $\mathbf{a}_{(0)}$. If $\Delta\chi^2$ is set to be a large number, this will be a big region; if it is small, it will be small. Somewhere in between there will be choices of $\Delta\chi^2$ that cause the region to contain, variously, 68%, 90%, etc., of probability distribution for $\mathbf{a}$'s, as defined above. These regions are taken as the confidence regions for the parameters $\mathbf{a}_{(0)}$.

Very frequently one is interested not in the full M-dimensional confidence region, but in individual confidence regions for some smaller number ν of parameters. For example, one might be interested in the confidence interval of each parameter taken separately (the bands in Figure 15.6.3), in which case $\nu = 1$. In that case, the natural confidence regions in the ν-dimensional subspace of the M-dimensional parameter space are the *projections* of the M-dimensional regions defined by fixed $\Delta\chi^2$ into the ν-dimensional spaces of interest. In Figure 15.6.4, for the case $M = 2$, we show regions corresponding to several values of $\Delta\chi^2$. The one-dimensional confidence interval in a_1 corresponding to the region bounded by $\Delta\chi^2 = 1$ lies between the lines A and A'.

Note that it is the projection of the higher-dimensional region on the lower-dimension space that is used, not the intersection. The intersection would be the band between Z and Z'. It is *never* used. It is shown in the figure only for the purpose of making this cautionary point, that it should not be confused with the projection.

15.6.5 Probability Distribution of Parameters in the Normal Case

You may be wondering why we have, in this section up to now, made no connection at all with the error estimates that come out of the χ^2 fitting procedure, most notably the covariance matrix C_{ij}. The reason is this: χ^2 minimization is a useful means for estimating parameters even if the measurement errors are not normally distributed. While normally distributed errors are required if the χ^2 parameter estimate is to be a maximum likelihood estimator (§15.1), one is often willing to give up that property in return for the relative convenience of the χ^2 procedure. Only in extreme cases, i.e., measurement error distributions with very large "tails," is χ^2 minimization abandoned in favor of more robust techniques, as will be discussed in §15.7.

However, the formal covariance matrix that comes out of a χ^2 minimization has a clear quantitative interpretation only if (or to the extent that) the measurement errors actually are normally distributed. In the case of *non*normal errors, you are "allowed"

- to fit for parameters by minimizing χ^2
- to use a contour of constant $\Delta\chi^2$ as the boundary of your confidence region
- to use Monte Carlo simulation or detailed analytic calculation in determining

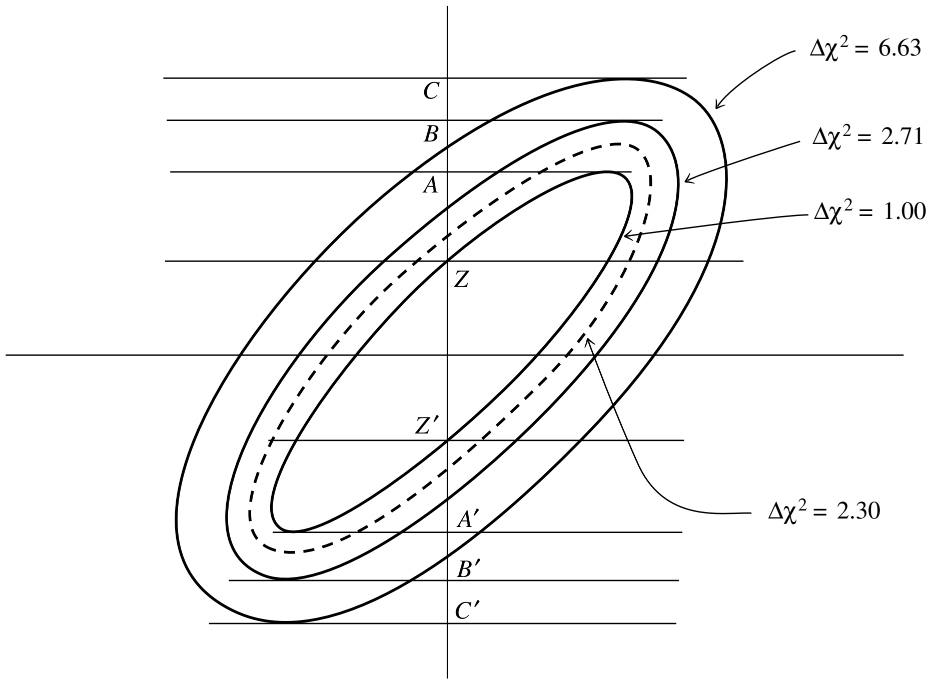


Figure 15.6.4. Confidence region ellipses corresponding to values of chi-square larger than the fitted minimum. The solid curves, with $\Delta\chi^2 = 1.00, 2.71, 6.63$, project onto one-dimensional intervals AA', BB', CC'. These intervals — not the ellipses themselves — contain 68.3%, 90%, and 99% of normally distributed data. The ellipse that contains 68.3% of normally distributed data is shown dashed and has $\Delta\chi^2 = 2.30$. For additional numerical values, see the table on p. 815.

which contour $\Delta\chi^2$ is the correct one for your desired confidence level

- to give the covariance matrix C_{ij} as the "formal covariance matrix of the fit."

You are *not* allowed

- to use formulas that we now give for the case of normal errors, which establish quantitative relationships among $\Delta\chi^2$, C_{ij}, and the confidence level.

Here are the key theorems that hold when (i) the measurement errors are normally distributed, and either (ii) the model is linear in its parameters or (iii) the sample size is large enough that the uncertainties in the fitted parameters **a** do not extend outside a region in which the model could be replaced by a suitable linearized model. [Note that condition (iii) does not preclude your use of a nonlinear routine like `Fitmrq` to *find* the fitted parameters.]

Theorem A. $\chi^2_{\min}$ is distributed as a chi-square distribution with $N - M$ degrees of freedom, where N is the number of data points and M is the number of fitted parameters. This is the basic theorem that lets you evaluate the goodness-of-fit of the model, as discussed above in §15.1. We list it first to remind you that unless the goodness-of-fit is credible, the whole estimation of parameters is suspect.

Theorem B. If $\mathbf{a}^S_{(j)}$ is drawn from the universe of simulated data sets with actual parameters $\mathbf{a}_{(0)}$, then the probability distribution of $\delta\mathbf{a} \equiv \mathbf{a}^S_{(j)} - \mathbf{a}_{(0)}$ is the multivariate normal distribution

$$P(\delta\mathbf{a})\,da_0\ldots da_{M-1} = \text{const.} \times \exp\left(-\tfrac{1}{2}\delta\mathbf{a}\cdot\boldsymbol{\alpha}\cdot\delta\mathbf{a}\right)\,da_0\ldots da_{M-1}$$

where $\boldsymbol{\alpha}$ is the curvature matrix defined in equation (15.5.8).

Theorem C. If $\mathbf{a}^S_{(j)}$ is drawn from the universe of simulated data sets with actual parameters $\mathbf{a}_{(0)}$, then the quantity $\Delta\chi^2 \equiv \chi^2(\mathbf{a}_{(j)}) - \chi^2(\mathbf{a}_{(0)})$ is distributed as a chi-square distribution with M degrees of freedom. Here the χ^2's are all evaluated using the fixed (actual) data set $\mathcal{D}_{(0)}$. This theorem makes the connection between particular values of $\Delta\chi^2$ and the fraction of the probability distribution that they enclose as an M-dimensional region, i.e., the confidence level of the M-dimensional confidence region.

Theorem D. Suppose that $\mathbf{a}^S_{(j)}$ is drawn from the universe of simulated data sets (as above); that its first ν components $a_0,\ldots,a_{\nu-1}$ are held fixed; and that its remaining $M-\nu$ components are varied so as to minimize χ^2. Call this minimum value χ^2_ν. Then $\Delta\chi^2_\nu \equiv \chi^2_\nu - \chi^2_{\min}$ is distributed as a chi-square distribution with ν degrees of freedom. If you consult Figure 15.6.4, you will see that this theorem connects the *projected* $\Delta\chi^2$ region with a confidence level. In the figure, a point that is held fixed in a_1 and allowed to vary in a_0 minimizing χ^2 will seek out the ellipse whose top or bottom edge is tangent to the line of constant a_1, and is therefore the line that projects it onto the smaller-dimensional space.

As a first example, let us consider the case $\nu = 1$, where we want to find the confidence interval of a single parameter, say a_0. Notice that the chi-square distribution with $\nu = 1$ degree of freedom is the same distribution as that of the square of a single normally distributed quantity. Thus $\Delta\chi^2_\nu < 1$ occurs 68.3% of the time (1-σ for the normal distribution), $\Delta\chi^2_\nu < 4$ occurs 95.4% of the time (2-σ for the normal distribution), $\Delta\chi^2_\nu < 9$ occurs 99.73% of the time (3-σ for the normal distribution), etc. In this manner you find the $\Delta\chi^2_\nu$ that corresponds to your desired confidence level. (Additional values are given in the table on the next page.)

Let $\delta\mathbf{a}$ be a change in the parameters whose first component is arbitrary, δa_0, but the rest of whose components are chosen to minimize the $\Delta\chi^2$. Then Theorem D applies. The value of $\Delta\chi^2$ is given in general by

$$\Delta\chi^2 = \delta\mathbf{a}\cdot\boldsymbol{\alpha}\cdot\delta\mathbf{a} \tag{15.6.1}$$

which follows from equation (15.5.8) applied at $\chi^2_{\min}$ where $\beta_k = 0$. Since $\delta\mathbf{a}$ by hypothesis minimizes χ^2 in all but its zeroth component, components 1 through $M-1$ of the normal equations (15.5.9) continue to hold. Therefore, the solution of (15.5.9) is

$$\delta\mathbf{a} = \boldsymbol{\alpha}^{-1}\cdot\begin{pmatrix} c \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \mathbf{C}\cdot\begin{pmatrix} c \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{15.6.2}$$

where c is one arbitrary constant that we get to adjust to make (15.6.1) give the desired left-hand value. Plugging (15.6.2) into (15.6.1) and using the fact that $\mathbf{C}$ and $\boldsymbol{\alpha}$ are inverse matrices of one another, we get

$$c = \delta a_0 / C_{00} \qquad \text{and} \qquad \Delta\chi^2_\nu = (\delta a_0)^2 / C_{00} \tag{15.6.3}$$

or

$\Delta\chi^2$ as a Function of Confidence Level p and Number of Parameters of Interest ν

	ν					
p	1	2	3	4	5	6
68.27%	1.00	2.30	3.53	4.72	5.89	7.04
90%	2.71	4.61	6.25	7.78	9.24	10.6
95.45%	4.00	6.18	8.02	9.72	11.3	12.8
99%	6.63	9.21	11.3	13.3	15.1	16.8
99.73%	9.00	11.8	14.2	16.3	18.2	20.1
99.99%	15.1	18.4	21.1	23.5	25.7	27.9

$$\delta a_0 = \pm\sqrt{\Delta\chi^2_\nu}\,\sqrt{C_{00}} \tag{15.6.4}$$

At last! A relation between the confidence interval $\pm\delta a_0$ and the formal standard error $\sigma_0 \equiv \sqrt{C_{00}}$. Not unreasonably, we find that the 68% confidence interval is $\pm\sigma_0$, the 95% confidence interval is $\pm 2\sigma_0$, etc.

These considerations hold not just for the individual parameters a_i, but also for any linear combination of them: If

$$b \equiv \sum_{k=0}^{M-1} c_i a_i = \mathbf{c}\cdot\mathbf{a} \tag{15.6.5}$$

then the 68% confidence interval on b is

$$\delta b = \pm\sqrt{\mathbf{c}\cdot\mathbf{C}\cdot\mathbf{c}} \tag{15.6.6}$$

However, these simple, normal-sounding numerical relationships do *not* hold in the case $\nu > 1$ [3]. In particular, $\Delta\chi^2 = 1$ is not the boundary, nor does it project onto the boundary, of a 68.3% confidence region when $\nu > 1$. If you want to calculate not confidence intervals in one parameter, but confidence ellipses in two parameters jointly, or ellipsoids in three, or higher, then you must follow the following prescription for implementing Theorems C and D above:

- Let ν be the number of fitted parameters whose joint confidence region you wish to display, $\nu \le M$. Call these parameters the "parameters of interest."
- Let p be the confidence limit desired, e.g., $p = 0.68$ or $p = 0.95$.
- Find Δ (i.e., $\Delta\chi^2$) such that the probability of a chi-square variable with ν degrees of freedom being less than Δ is p. For some useful values of p and ν, Δ is given in the table above. For other values, you can use the `invcdf` method of the `Chisqdist` object in §6.14.8 with p as the argument.
- Take the $M \times M$ covariance matrix $\mathbf{C} = \boldsymbol{\alpha}^{-1}$ of the chi-square fit. Copy the intersection of the ν rows and columns corresponding to the parameters of interest into a $\nu \times \nu$ matrix denoted $\mathbf{C}_{\text{proj}}$.
- Invert the matrix $\mathbf{C}_{\text{proj}}$. (In the one-dimensional case this was just taking the reciprocal of the element C_{00}.)

- The equation for the elliptical boundary of your desired confidence region in the ν-dimensional subspace of interest is

$$\Delta = \delta\mathbf{a}' \cdot \mathbf{C}_{\text{proj}}^{-1} \cdot \delta\mathbf{a}' \tag{15.6.7}$$

  where $\delta\mathbf{a}'$ is the ν-dimensional vector of parameters of interest.

If you are confused at this point, you may find it helpful to compare Figure 15.6.4 and the table on the previous page considering the case $M = 2$ with $\nu = 1$ and $\nu = 2$. You should be able to verify the following statements: (i) The horizontal band between C and C' contains 99% of the probability distribution, so it is a confidence limit on a_1 alone at this level of confidence. (ii) Ditto the band between B and B' at the 90% confidence level. (iii) The dashed ellipse, labeled by $\Delta\chi^2 = 2.30$, contains 68.3% of the probability distribution, so it is a confidence region for a_0 and a_1 jointly, at this level of confidence.

Another point of possible confusion might also be worth airing here. In §15.1.1, when we discussed the use of χ^2 as a goodness-of-fit statistic, we mentioned that a "moderately good" fit could have a χ^2 value that differed by as much as $\pm\sqrt{2\nu}$ from its expected value ν (now the total number of degrees of freedom $N - M$, not ν as used above). Indeed, the suggested tail probability that embodies this advice is $Q = 1 - \texttt{Chisqdist}(\nu)\texttt{.invcdf}(\chi^2)$. Yet, in the discussion above, we seem to be saying that small changes in χ^2, as little as ± 1 or ± 2.71 (see table on the previous page), are significant. Can both statements be true?

Yes. In §15.1.1 we were considering the variation in χ^2 over a population of hypothetical data sets with the same parameter values, $\mathbf{a}_{\text{true}}$ (cf. Figure 15.6.1). These values vary by typically $\pm\sqrt{2\nu}$. By contrast, in the discussion above, we took a single data set and held it fixed. We then asked, essentially as an exercise in propagation of errors, how much uncertainty in the fitted parameter values $\mathbf{a}_0$ was generated by the uncertainties in the data. One way to see that these are quite different concepts is to think about how they should each scale with N, the number of data points. As N gets large, χ^2 scales as N, while its variation over hypothetical data sets scales as $N^{1/2}$, essentially a random walk. Now imagine $\mathbf{a}$ varying around its fitted value $\mathbf{a}_0$ by a small amount, $\mathbf{a} = \mathbf{a}_0 + \delta\mathbf{a}$. The change in χ^2 scales with the number of terms in the sum, N, and quadratically with distance from the minimum,

$$\delta\chi^2 \propto N(\delta\mathbf{a})^2 \tag{15.6.8}$$

As the number of data points increases, we reasonably expect the parameters to become more accurately determined, scaling as

$$\delta\mathbf{a} \propto N^{-1/2} \tag{15.6.9}$$

Combining these two equations, we find that $\delta\chi^2$ for the minimum significant change in parameters $\delta\mathbf{a}$ scales as N^0, that is, as a constant. In fact, Theorems B and C above tell us that this is not just reasonable expectation on our part; it is actually true.

15.6.6 Confidence Limits from Singular Value Decomposition

When you have obtained your χ^2 fit by singular value decomposition (§15.4), the information about the fit's formal errors comes packaged in a somewhat different,

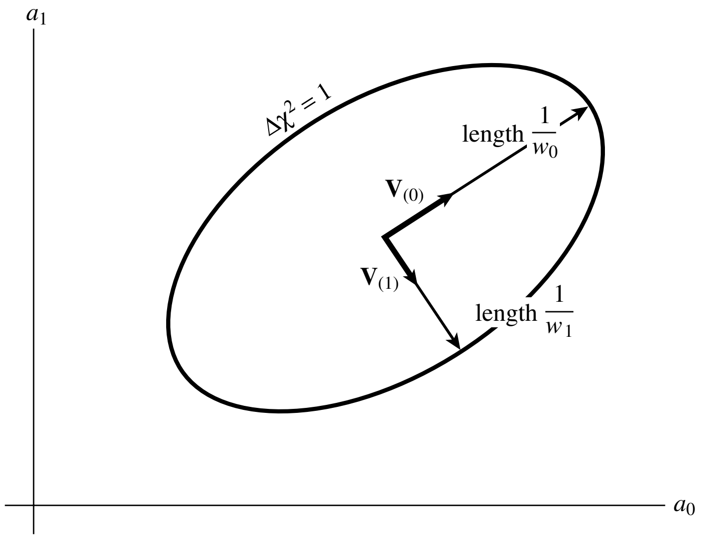


Figure 15.6.5. Relation of the confidence region ellipse $\Delta\chi^2 = 1$ to quantities computed by singular value decomposition. The vectors $\mathbf{V}_{(i)}$ are unit vectors along the principal axes of the confidence region. The semi-axes have lengths equal to the reciprocal of the singular values w_i. If the axes are all scaled by some constant factor α, $\Delta\chi^2$ is scaled by the factor α^2.

but generally more convenient, form. The columns of the matrix $\mathbf{V}$ are an orthonormal set of M vectors that are the principal axes of the $\Delta\chi^2 =$ constant ellipsoids. We denote the columns as $\mathbf{V}_{(0)} \ldots \mathbf{V}_{(M-1)}$. The lengths of those axes are inversely proportional to the corresponding singular values $w_0 \ldots w_{M-1}$; see Figure 15.6.5. The boundaries of the ellipsoids are thus given by

$$\Delta\chi^2 = w_0^2(\mathbf{V}_{(0)} \cdot \delta\mathbf{a})^2 + \cdots + w_{M-1}^2(\mathbf{V}_{(M-1)} \cdot \delta\mathbf{a})^2 \tag{15.6.10}$$

which is the justification for writing equation (15.4.18) above. Keep in mind that it is *much* easier to plot an ellipsoid given a list of its vector principal axes than given its matrix quadratic form: Loop over points $\mathbf{z}$ on a unit sphere in any desired way (e.g., by latitude and longitude) and plot the mapped points

$$\delta\mathbf{a} = \sqrt{\Delta\chi^2} \sum_i \frac{1}{w_i} (\mathbf{z} \cdot \mathbf{V}_{(i)}) \mathbf{V}_{(i)} \tag{15.6.11}$$

The formula for the covariance matrix $\mathbf{C}$ in terms of the columns $\mathbf{V}_{(i)}$ is

$$\mathbf{C} = \sum_{i=0}^{M-1} \frac{1}{w_i^2} \mathbf{V}_{(i)} \otimes \mathbf{V}_{(i)} \tag{15.6.12}$$

or, in components,

$$C_{jk} = \sum_{i=0}^{M-1} \frac{1}{w_i^2} V_{ji} V_{ki} \tag{15.6.13}$$

A method for plotting error ellipses (2-dimensions) or ellipsoids (3-dimensions) from the covariance matrix $\mathbf{C}$ directly, not using its principal axes, is described in §16.1.1.

CITED REFERENCES AND FURTHER READING:

Davison, A.C., and Hinkley, D.V. 1997, *Bootstrap Methods and Their Application* (New York: Cambridge University Press).

Efron, B. 1982, *The Jackknife, the Bootstrap, and Other Resampling Plans* (Philadelphia: S.I.A.M.).[1]

Efron, B., and Tibshirani, R. 1993, *An Introduction to the Bootstrap* (Boca Raton, FL: CRC Press).[2]

Lupton, R. 1993, *Statistics in Theory and Practice* (Princeton, NJ: Princeton University Press), Chapters 10–11.

Avni, Y. 1976, "Energy Spectra of X-Ray Clusters of Galaxies," *Astrophysical Journal*, vol. 210, pp. 642–646.[3]

Lampton, M., Margon, M., and Bowyer, S. 1976, "Parameter Estimation in X-ray Astronomy," *Astrophysical Journal*, vol. 208, pp. 177–190.

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

15.7 Robust Estimation

The concept of *robustness* has been mentioned in passing several times already. In §14.1 we noted that the median was a more robust estimator of central value than the mean; in §14.6 it was mentioned that rank correlation is more robust than linear correlation. The concept of outlier points as exceptions to a Gaussian model for experimental error was discussed in §15.1.

The term "robust" was coined in statistics by G.E.P. Box in 1953. Various definitions of greater or lesser mathematical rigor are possible for the term, but in general, referring to a statistical estimator, it means "insensitive to small departures from the idealized assumptions for which the estimator is optimized" [1,2,3]. The word "small" can have two different interpretations, both important: either fractionally small departures for all data points, or else fractionally large departures for a small number of data points. It is the latter interpretation, leading to the notion of outlier points, that is generally the most stressful for statistical procedures.

Statisticians have developed various sorts of robust statistical estimators. Many, if not most, can be grouped into one of three categories.

M-estimates follow from maximum likelihood arguments very much as equations (15.1.6) and (15.1.8) followed from equation (15.1.3). M-estimates are usually the most relevant class for model fitting, that is, estimation of parameters. We therefore consider these estimates in some detail below.

L-estimates are "linear combinations of order statistics." These are most applicable to estimations of central value and central tendency, though they can occasionally be applied to some problems in estimation of parameters. Two "typical" L-estimates will give you the general idea. They are (i) the median, and (ii) *Tukey's trimean*, defined as the weighted average of the first, second, and third quartile points in a distribution, with weights 1/4, 1/2, and 1/4, respectively.

R-estimates are estimates based on rank tests. For example, the equality or inequality of two distributions can be estimated by the *Wilcoxon test* of computing the mean rank of one distribution in a combined sample of both distribu-

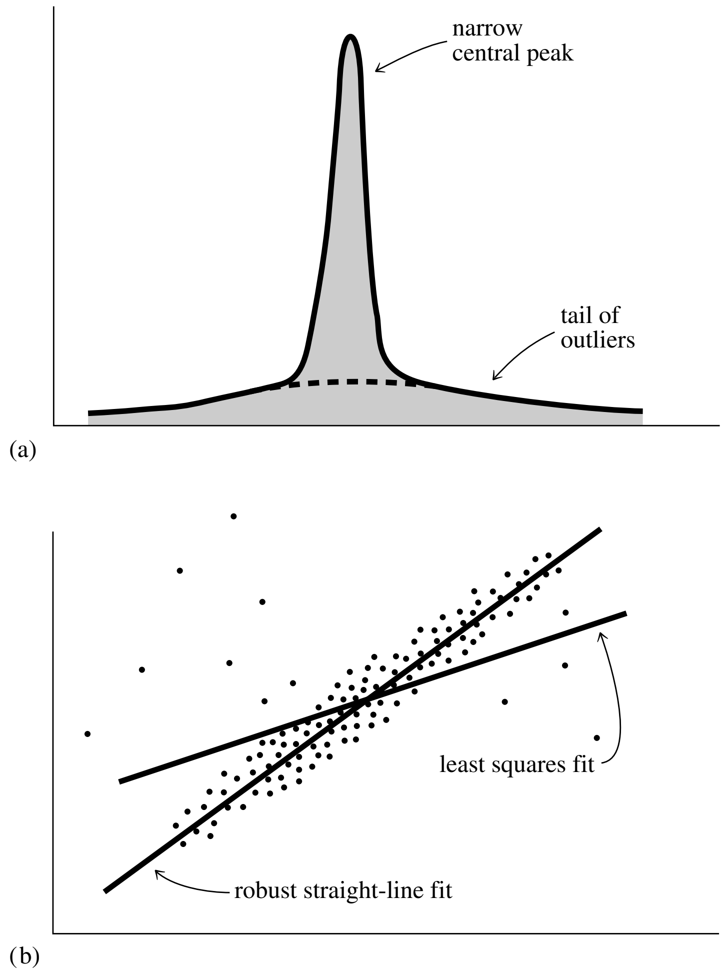


Figure 15.7.1. Examples where robust statistical methods are desirable: (a) A one-dimensional distribution with a tail of outliers; statistical fluctuations in these outliers can prevent accurate determination of the position of the central peak. (b) A distribution in two dimensions fitted to a straight line; nonrobust techniques such as least-squares fitting can have undesired sensitivity to outlying points.

tions. The Kolmogorov-Smirnov statistic (equation 14.3.17) and the Spearman rank-order correlation coefficient (14.6.1) are R-estimates in essence, if not always by formal definition.

Some other kinds of robust techniques, coming from the fields of optimal control and filtering rather than from the field of mathematical statistics, are mentioned at the end of this section. Some examples where robust statistical methods are desirable are shown in Figure 15.7.1.

15.7.1 Estimation of Parameters by Local M-Estimates

Suppose we know that our measurement errors are not normally distributed. Then, in deriving a maximum likelihood formula for the estimated parameters $\mathbf{a}$ in a model $y(x|\mathbf{a})$, we would write instead of equation (15.1.3)

$$P = \prod_{i=0}^{N-1} \{\exp\left[-\rho(y_i, y\,\{x_i|\mathbf{a}\})\right] \Delta y\} \tag{15.7.1}$$

where the function ρ is the negative logarithm of the probability density. Taking the logarithm of (15.7.1) analogously with (15.1.5), we find that we want to minimize the expression

$$\sum_{i=0}^{N-1} \rho(y_i, y\,\{x_i|\mathbf{a}\}) \tag{15.7.2}$$

Very often, it is the case that the function ρ depends not independently on its two arguments, measured y_i and predicted $y(x_i)$, but only on their difference, at least if scaled by some weight factors σ_i that we are able to assign to each point. In this case the M-estimate is said to be *local*, and we can replace (15.7.2) by the prescription

$$\text{minimize over } \mathbf{a} \qquad \sum_{i=0}^{N-1} \rho\left(\frac{y_i - y(x_i|\mathbf{a})}{\sigma_i}\right) \tag{15.7.3}$$

where the function $\rho(z)$ is a function of a single variable $z \equiv [y_i - y(x_i)]/\sigma_i$.

If we now define the derivative of $\rho(z)$ to be a function $\psi(z)$,

$$\psi(z) \equiv \frac{d\rho(z)}{dz} \tag{15.7.4}$$

then the generalization of (15.1.8) to the case of a general M-estimate is

$$0 = \sum_{i=0}^{N-1} \frac{1}{\sigma_i}\psi\left(\frac{y_i - y(x_i)}{\sigma_i}\right)\left(\frac{\partial y(x_i|\mathbf{a})}{\partial a_k}\right) \qquad k = 0, \ldots, M-1 \tag{15.7.5}$$

If you compare (15.7.3) to (15.1.3) and (15.7.5) to (15.1.8), you see at once that the specialization for normally distributed errors is

$$\rho(z) = \tfrac{1}{2}z^2 \qquad \psi(z) = z \qquad \text{(normal)} \tag{15.7.6}$$

If the errors are distributed as a *double* or *two-sided exponential*, namely

$$\text{Prob}\,\{y_i - y(x_i)\} \sim \exp\left(-\left|\frac{y_i - y(x_i)}{\sigma_i}\right|\right) \tag{15.7.7}$$

then, by contrast,

$$\rho(x) = |z| \qquad \psi(z) = \text{sgn}(z) \qquad \text{(double exponential)} \tag{15.7.8}$$

Comparing to equation (15.7.3), we see that in this case the maximum likelihood estimator is obtained by minimizing the *mean absolute deviation*, rather than the mean square deviation. Here the tails of the distribution, although exponentially decreasing, are asymptotically much larger than any corresponding Gaussian.

A distribution with even more extensive — and therefore sometimes even more realistic — tails is the *Cauchy* or *Lorentzian* distribution,

$$\text{Prob}\,\{y_i - y(x_i)\} \sim \frac{1}{1 + \frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma_i}\right)^2} \tag{15.7.9}$$

This implies

$$\rho(z) = \log\left(1 + \frac{1}{2}z^2\right) \qquad \psi(z) = \frac{z}{1 + \frac{1}{2}z^2} \qquad \text{(Lorentzian)} \tag{15.7.10}$$

Notice that the ψ function occurs as a weighting function in the generalized normal equations (15.7.5). For normally distributed errors, equation (15.7.6) says that the more deviant the points, the greater the weight. By contrast, when tails are somewhat more prominent, as in (15.7.7), then (15.7.8) says that all deviant points get the same relative weight, with only the sign information used. Finally, when the tails are even larger, (15.7.10) says the ψ increases with deviation, then starts *decreasing*, so that very deviant points — the true outliers — are not counted at all in the estimation of the parameters.

This general idea, that the weight given individual points should first increase with deviation, then decrease, motivates some additional prescriptions for ψ that do not especially correspond to standard, textbook probability distributions. Two examples are

Andrew's sine

$$\psi(z) = \begin{cases} \sin(z/c) & |z| < c\pi \\ 0 & |z| > c\pi \end{cases} \tag{15.7.11}$$

If the measurement errors happen to be normal after all, with standard deviations σ_i, then it can be shown that the optimal value for the constant c is $c = 2.1$.

Tukey's biweight

$$\psi(z) = \begin{cases} z(1 - z^2/c^2)^2 & |z| < c \\ 0 & |z| > c \end{cases} \tag{15.7.12}$$

where the optimal value of c for normal errors is $c = 6.0$.

15.7.2 Numerical Calculation of M-Estimates

To fit a model by means of an M-estimate, you first decide which M-estimate you want, that is, which matching pair ρ, ψ you want to use. We rather like (15.7.8) or (15.7.10).

You then have to make an unpleasant choice between two fairly difficult problems. Either find the solution of the nonlinear set of M equations (15.7.5), or else minimize the single function in M variables (15.7.3).

Notice that the function (15.7.8) has a discontinuous ψ and a discontinuous derivative for ρ. Such discontinuities frequently wreak havoc on both general nonlinear equation solvers and general function minimizing routines. You might now think of rejecting (15.7.8) in favor of (15.7.10), which is smoother. However, you will find that the latter choice is also bad news for many general equation solving or minimization routines: Small changes in the fitted parameters can drive $\psi(z)$ off its peak into one or the other of its asymptotically small regimes. Therefore, different terms in the equation spring into or out of action (almost as bad as analytic discontinuities).

Don't despair. If your computer is fast enough, or if your patience is great enough, this is an excellent application for the downhill simplex minimization algorithm exemplified in `Amoeba` (§10.5) or `Amebsa` (§10.12). Those algorithms make no assumptions about continuity; they just ooze downhill and will work for virtually any sane choice of the function ρ.

It is very much to your (patience) advantage to find good starting values, however. Often this is done by first fitting the model by the standard χ^2 (nonrobust) techniques, e.g., as described in §15.4 or §15.5. The fitted parameters thus obtained are then used as starting values in `Amoeba`, now using the robust choice of ρ and minimizing the expression (15.7.3).

15.7.3 Fitting a Line by Minimizing Absolute Deviation

Occasionally there is a special case that happens to be much easier than is suggested by the general strategy outlined above. The case of equations (15.7.7) – (15.7.8), when the model is a simple straight line,

$$y(x|a,b) = a + bx \tag{15.7.13}$$

and where the weights σ_i are all equal, happens to be such a case. The problem is precisely the robust version of the problem posed in equation (15.2.1) above, namely fit a straight line through a set of data points. The merit function to be minimized is

$$\sum_{i=0}^{N-1} |y_i - a - bx_i| \tag{15.7.14}$$

rather than the χ^2 given by equation (15.2.2).

The key simplification is based on the following fact: The median c_M of a set of numbers c_i is also the value that minimizes the sum of the absolute deviations

$$\sum_i |c_i - c_M|$$

(Proof: Differentiate the above expression with respect to c_M and set it to zero.)

It follows that, for fixed b, the value of a that minimizes (15.7.14) is

$$a = \text{median}\,\{y_i - bx_i\} \tag{15.7.15}$$

Equation (15.7.5) for the parameter b is

$$0 = \sum_{i=0}^{N-1} x_i \,\text{sgn}(y_i - a - bx_i) \tag{15.7.16}$$

(where sgn(0) is to be interpreted as zero). If we replace a in this equation by the implied function $a(b)$ of (15.7.15), then we are left with an equation in a single variable that can be solved by bracketing and bisection, as described in §9.1. (In fact, it is dangerous to use any fancier method of root finding, because of the discontinuities in equation 15.7.16.)

Here is an object that does all this. It calls `select` (§8.5) to find the median. The bracketing and bisection are built into the routine, as is the linear fit that generates the initial guesses for a and b.

fitmed.h

```
struct Fitmed {
```
Object for fitting a straight line $y = a + bx$ to a set of points (x_i, y_i), by the criterion of least absolute deviations. Call the constructor to calculate the fit. The answers are then available as the variables a, b, and abdev (the mean absolute deviation of the points from the line).
```
    Int ndata;
    Doub a, b, abdev;                                  Answers.
    VecDoub_I &x, &y;

    Fitmed(VecDoub_I &xx, VecDoub_I &yy) : ndata(xx.size()), x(xx), y(yy) {
```
Constructor. Given a set of data points xx[0..ndata-1], yy[0..ndata-1], sets a, b, and abdev.
```
        Int j;
        Doub b1,b2,del,f,f1,f2,sigb,temp;
        Doub sx=0.0,sy=0.0,sxy=0.0,sxx=0.0,chisq=0.0;
        for (j=0;j<ndata;j++) {          As a first guess for a and b, we will find the
            sx += x[j];                      least-squares fitting line.
            sy += y[j];
            sxy += x[j]*y[j];
            sxx += SQR(x[j]);
        }
        del=ndata*sxx-sx*sx;
        a=(sxx*sy-sx*sxy)/del;           Least-squares solutions.
        b=(ndata*sxy-sx*sy)/del;
        for (j=0;j<ndata;j++)
            chisq += (temp=y[j]-(a+b*x[j]),temp*temp);
        sigb=sqrt(chisq/del);            The standard deviation will give some idea of
        b1=b;                                how big an iteration step to take.
        f1=rofunc(b1);
        if (sigb > 0.0) {
            b2=b+SIGN(3.0*sigb,f1);      Guess bracket as 3-σ away, in the downhill di-
            f2=rofunc(b2);                   rection known from f1.
            if (b2 == b1) {
                abdev /= ndata;
                return;
            }
            while (f1*f2 > 0.0) {        Bracketing.
                b=b2+1.6*(b2-b1);
                b1=b2;
                f1=f2;
                b2=b;
                f2=rofunc(b2);
            }
            sigb=0.01*sigb;
            while (abs(b2-b1) > sigb) {
                b=b1+0.5*(b2-b1);        Bisection.
                if (b == b1 || b == b2) break;
                f=rofunc(b);
                if (f*f1 >= 0.0) {
                    f1=f;
                    b1=b;
                } else {
                    f2=f;
                    b2=b;
                }
            }
        }
        abdev /= ndata;
    }

    Doub rofunc(const Doub b) {
```
Evaluates the right-hand side of equation (15.7.16) for a given value of b.
```
        const Doub EPS=numeric_limits<Doub>::epsilon();
        Int j;
        Doub d,sum=0.0;
        VecDoub arr(ndata);
        for (j=0;j<ndata;j++) arr[j]=y[j]-b*x[j];
        if ((ndata & 1) == 1) {
            a=select((ndata-1)>>1,arr);
        } else {
            j=ndata >> 1;
            a=0.5*(select(j-1,arr)+select(j,arr));
        }
        abdev=0.0;
        for (j=0;j<ndata;j++) {
            d=y[j]-(b*x[j]+a);
            abdev += abs(d);
            if (y[j] != 0.0) d /= abs(y[j]);
            if (abs(d) > EPS) sum += (d >= 0.0 ? x[j] : -x[j]);
        }
        return sum;
    }
};
```

15.7.4 Other Robust Techniques

Sometimes you may have a priori knowledge about the probable values and probable uncertainties of some parameters that you are trying to estimate from a data set. In such cases you may want to perform a fit that takes this advance information properly into account, neither completely freezing a parameter at a predetermined value (as in `Fitlin` §15.4) nor completely leaving it to be determined by the data set. The formalism for doing this is called "use of a priori covariances."

A related problem occurs in signal processing and control theory, where it is sometimes desired to "track" (i.e., maintain an estimate of) a time-varying signal in the presence of noise. If the signal is known to be characterized by some number of parameters that vary only slowly, then the formalism of *Kalman filtering* tells how the incoming raw measurements of the signal should be processed to produce best parameter estimates as a function of time. For example, if the signal is a frequency-modulated sine wave, then the slowly varying parameter might be the instantaneous frequency. The Kalman filter for this case is called a *phase-locked loop* and is implemented in the circuitry of modern radio receivers [4,5].

CITED REFERENCES AND FURTHER READING:

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley).[1]

Maronna, R., Martin, D., and Yohai, V. 2006, *Robust Statistics: Theory and Methods* (Hoboken, NJ: Wiley).[2]

Launer, R.L., and Wilkinson, G.N. (eds.) 1979, *Robustness in Statistics* (New York: Academic Press).[3]

Sayed, A.H. 2003, *Fundamentals of Adaptive Filtering* (New York: Wiley-IEEE).[4]

Harvey, A.C. 1989, *Forecasting, Structural Time Series Models and the Kalman Filter* (Cambridge, UK: Cambridge University Press).[5]

15.8 Markov Chain Monte Carlo

In this section and the next we redress somewhat the imbalance, at this point, between frequentist and Bayesian methods of modeling. Like Monte Carlo integration, *Markov chain Monte Carlo* or *MCMC* is a random sampling method. Unlike Monte Carlo integration, however, the goal of MCMC is not to sample a multidimensional region uniformly. Rather, the goal is to visit a point $\mathbf{x}$ with a probability proportional to some given distribution function $\pi(\mathbf{x})$. The distribution $\pi(\mathbf{x})$ is not quite a probability, because it is not necessarily normalized to have unity integral over the sampled region; but it is proportional to a probability.

Why would we want to sample a distribution in this way? The answer is that Bayesian methods, often implemented using MCMC, provide a powerful way of estimating the parameters of a model and their degree of uncertainty. A typical case is that there is a given set of data $\mathbf{D}$, and that we are able to calculate the probability of the data set *given* the values of the model parameters $\mathbf{x}$, that is, $P(\mathbf{D}|\mathbf{x})$. If we assume a prior $P(\mathbf{x})$, then Bayes' theorem says that the (posterior) probability of the model is proportional to $\pi(\mathbf{x}) \equiv P(\mathbf{D}|\mathbf{x})P(\mathbf{x})$, but with an unknown normalizing constant. Because of this unknown constant, $\pi(\mathbf{x})$ is not a normalized probability density. But if we can sample from it, we can estimate any quantity of interest, for example its mean or variance. Indeed, we can readily recover a normalized probability density by observing how often we sample a given volume $d\mathbf{x}$. Often even more useful, we can observe the distribution of any single component or set of components of the vector $\mathbf{x}$, equivalent to *marginalizing* (i.e., integrating over) the other components.

We could in principle obtain all the same information by ordinary Monte Carlo integration over the region of interest, computing the value of $\pi(\mathbf{x}_i)$ at every (uniformly) sampled point $\mathbf{x}_i$. The huge advantage of MCMC is that it "automatically" puts its sample points preferentially where $\pi(\mathbf{x})$ is large (in fact, in direct proportion). In a high-dimensional space, or where $\pi(\mathbf{x})$ is expensive to compute, this can be advantageous by many orders of magnitude.

Two insights, originally due to Metropolis and colleagues in the early 1950s, lead to feasible MCMC methods. The first is the idea that we should try to sample $\pi(\mathbf{x})$ not via unrelated, independent points, but rather by a *Markov chain*, a sequence of points $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$ that, while locally correlated, can be shown to eventually visit every point $\mathbf{x}$ in proportion to $\pi(\mathbf{x})$, the *ergodic* property. Here the word "Markov" means that each point $\mathbf{x}_i$ is chosen from a distribution that depends only on the value of the immediately preceding point $\mathbf{x}_{i-1}$. In other words, the chain has memory extending only to one previous point and is completely defined by a transition probability function of two variables $p(\mathbf{x}_i|\mathbf{x}_{i-1})$, the probability with which $\mathbf{x}_i$ is picked given a previous point $\mathbf{x}_{i-1}$.

The second insight is that if $p(\mathbf{x}_i|\mathbf{x}_{i-1})$ is chosen to satisfy the *detailed balance equation*,

$$\pi(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1) = \pi(\mathbf{x}_2)p(\mathbf{x}_1|\mathbf{x}_2) \tag{15.8.1}$$

then (up to some technical conditions) the Markov chain will in fact sample $\pi(\mathbf{x})$ ergodically. This amazing fact is worthy of some contemplation. Equation (15.8.1) expresses the idea of *physical equilibrium* in the reversible transition

$$\mathbf{x}_1 \longleftrightarrow \mathbf{x}_2 \tag{15.8.2}$$

That is, if $\mathbf{x}_1$ and $\mathbf{x}_2$ occur in proportion to $\pi(\mathbf{x}_1)$ and $\pi(\mathbf{x}_2)$, respectively, then the overall transition rates in each direction, each the product of a population density and a transition probability, are the same. To see that this might have something to do with the Markov chain being ergodic, integrate both sides of equation (15.8.1) with respect to $\mathbf{x}_1$:

$$\int p(\mathbf{x}_2|\mathbf{x}_1)\pi(\mathbf{x}_1)\,d\mathbf{x}_1 = \pi(\mathbf{x}_2)\int p(\mathbf{x}_1|\mathbf{x}_2)\,d\mathbf{x}_1 = \pi(\mathbf{x}_2) \qquad (15.8.3)$$

The left-hand side of equation (15.8.3) is the probability of $\mathbf{x}_2$, computed by integrating over all possible values of $\mathbf{x}_1$ with the corresponding transition probability. The right-hand side is seen to be the desired $\pi(\mathbf{x}_2)$. So equation (15.8.3) says that if $\mathbf{x}_1$ is drawn from π, then *so is its successor* in the Markov chain, $\mathbf{x}_2$.

We also need to show that the equilibrium distribution is rapidly approached from any starting point $\mathbf{x}_0$. While the formal proof is beyond our scope, a heuristic proof is to recognize that, because of ergodicity, even very unlikely values $\mathbf{x}_0$ will be visited by the equilibrium Markov chain once in a great while. Since the chain has no past memory, choosing any such point as a starting point $\mathbf{x}_0$ is equivalent to just picking up the equilibrium distribution chain at that particular point in time, q.e.d. In practice we need to recognize that when we start from a very unlikely point, successor points will themselves be quite unlikely until we rejoin a more probable part of the distribution. There is thus a need to *burn-in* an MCMC chain by stepping through, and discarding, a certain number of points $\mathbf{x}_i$. Below, we discuss how to determine the length of the burn-in.

We can gain a better understanding the nature of the approach to π using concepts from §11.0 and (in the next chapter) §16.3. Heuristically, let us pretend that the states x_i are discrete. Then $p(x_j|x_i) \equiv P_{ij}$ is a transition matrix satisfying equation (16.3.1). The discussion following equation (16.3.4) shows that the matrix $\mathbf{P}^T$ must have at least one unity eigenvalue. In fact, the vector $\boldsymbol{\pi}$ (the discrete form of the distribution $\pi(\mathbf{x})$) is an eigenvector of $\mathbf{P}^T$ with unity eigenvalue, by equation (15.8.3).

Can there be eigenvalues with magnitude greater than unity? No. Suppose to the contrary that $\lambda > 1$ is the largest eigenvalue, with eigenvector $\mathbf{v}$. Then, repeatedly applying $\mathbf{P}^T$,

$$\lim_{n\to\infty} (\mathbf{P}^T)^n \cdot \mathbf{v} = \lambda^n \mathbf{v} \to \infty \times \mathbf{v} \qquad (15.8.4)$$

Any starting distribution that contains even a tiny piece of $\mathbf{v}$ (always possible to arrange) will be driven to have values either < 0 or > 1, which is impossible. Hence it must be that $\lambda \le 1$.

From an arbitrary starting distribution $\mathbf{u}$, repeated steps of $\mathbf{P}^T$ must thus converge to $\boldsymbol{\pi}$ geometrically, with a rate that is asymptotically the magnitude of the second-largest eigenvalue, which will be < 1 if $\boldsymbol{\pi}$ is the unique equilibrium distribution. If the second eigenvalue is small, the distribution $p(x_j|x_i)$ is said to be *rapidly mixing*.

Obviously missing from this discussion, and beyond our scope, is a discussion of degenerate eigenvalues (related to the question of uniqueness) and a continuous, rather than discrete, treatment. In practice, one rarely knows enough about $\mathbf{P}$ to compute useful bounds on the second eigenvalue a priori.

15.8.1 Metropolis-Hastings Algorithm

Unless we can find a transition probability function $p(\mathbf{x}_2|\mathbf{x}_1)$ that satisfies the detailed balance equation (15.8.1), we have no way to proceed. Luckily, Hastings [1], generalizing Metropolis' work, has given a very general prescription:

Pick a *proposal distribution* $q(\mathbf{x}_2|\mathbf{x}_1)$. This can be pretty much anything you want, as long as a succession of steps generated by it can, in principle, reach everywhere in the region of interest. For example, $q(\mathbf{x}_2|\mathbf{x}_1)$ might be a multivariate normal distribution centered on $\mathbf{x}_1$.

Now, to generate a step starting at $\mathbf{x}_1$, first generate a *candidate point* $\mathbf{x}_{2c}$ by drawing from the proposal distribution. Second, calculate an *acceptance probability* $\alpha(\mathbf{x}_1, \mathbf{x}_{2c})$ by the formula

$$\alpha(\mathbf{x}_1, \mathbf{x}_{2c}) = \min\left(1, \frac{\pi(\mathbf{x}_{2c})\, q(\mathbf{x}_1|\mathbf{x}_{2c})}{\pi(\mathbf{x}_1)\, q(\mathbf{x}_{2c}|\mathbf{x}_1)}\right) \tag{15.8.5}$$

Finally, with probability $\alpha(\mathbf{x}_1, \mathbf{x}_{2c})$, accept the candidate point and set $\mathbf{x}_2 = \mathbf{x}_{2c}$; otherwise reject it and leave the point unchanged (that is, $\mathbf{x}_2 = \mathbf{x}_1$). The net result of this process is a transition probability,

$$p(\mathbf{x}_2|\mathbf{x}_1) = q(\mathbf{x}_2|\mathbf{x}_1)\, \alpha(\mathbf{x}_1, \mathbf{x}_2), \qquad (\mathbf{x}_2 \neq \mathbf{x}_1) \tag{15.8.6}$$

To see how this satisfies detailed balance, first multiply equation (15.8.5) by the denominator in the second argument of the min function. Then write down the identical equation, but exchange $\mathbf{x}_1$ and $\mathbf{x}_2$. From these pieces, one writes,

$$\begin{aligned}\pi(\mathbf{x}_1)\, q(\mathbf{x}_2|\mathbf{x}_1)\, \alpha(\mathbf{x}_1, \mathbf{x}_2) &= \min[\pi(\mathbf{x}_1)\, q(\mathbf{x}_2|\mathbf{x}_1),\ \pi(\mathbf{x}_2)\, q(\mathbf{x}_1|\mathbf{x}_2)] \\ &= \min[\pi(\mathbf{x}_2)\, q(\mathbf{x}_1|\mathbf{x}_2),\ \pi(\mathbf{x}_1)\, q(\mathbf{x}_2|\mathbf{x}_1)] \\ &= \pi(\mathbf{x}_2)\, q(\mathbf{x}_1|\mathbf{x}_2)\, \alpha(\mathbf{x}_2, \mathbf{x}_1)\end{aligned} \tag{15.8.7}$$

which, using equation (15.8.6), can be seen to be exactly the detailed balance equation (15.8.1).

It is often possible to choose the proposal distribution $q(\mathbf{x}_2|\mathbf{x}_1)$ in such a way as to simplify equation (15.8.5). For example, if $q(\mathbf{x}_2|\mathbf{x}_1)$ depends only on the absolute difference $|\mathbf{x}_1 - \mathbf{x}_2|$, as in the case of a normal distribution with fixed covariance, then the ratio $q(\mathbf{x}_1|\mathbf{x}_{2c})/q(\mathbf{x}_{2c}|\mathbf{x}_1)$ is just 1. Another case that occurs frequently is when, for some component x of $\mathbf{x}$, $q(x_{2c}|x_1)$ is lognormally distributed with a mode at x_1. In that case the ratio for this component is x_{2c}/x_1 (cf. equation 6.14.31).

15.8.2 Gibbs Sampler

An important special case of the Metropolis-Hastings algorithm is the *Gibbs sampler*. (Historically, the Gibbs sampler was developed independently of Metropolis-Hastings, see [2,5], but we discuss it here in a unified framework.) The Gibbs sampler is based on the fact that a multivariate distribution is uniquely determined by the set of all of its full conditional distributions; but if you don't know what this means, just read on anyway.

A *full conditional distribution* of $\pi(\mathbf{x})$ is obtained by holding all of the components of $\mathbf{x}$ constant *except one* (call it x), and then sampling as a function of x alone. In other words, it is the distribution that you see when you "drill through" $\pi(\mathbf{x})$ along a coordinate direction, and with fixed values of all the other coordinates. We'll denote a full conditional distribution by the notation $\pi(x\,|\,\mathbf{x}^-)$, where $\mathbf{x}^-$ means "values of all the coordinates except one." (To keep the notation readable, we are suppressing an index i that would tell which component of $\mathbf{x}$ is x.)

Suppose that we construct a Metropolis-Hastings chain that allows only the one coordinate x to vary. Then equation (15.8.5) would look like this:

$$\alpha(x_1, x_{2c}|\mathbf{x}^-) = \min\left(1, \frac{\pi(x_{2c}|\mathbf{x}^-)\, q(x_1|x_{2c}, \mathbf{x}^-)}{\pi(x_1|\mathbf{x}^-)\, q(x_{2c}|x_1, \mathbf{x}^-)}\right) \tag{15.8.8}$$

Now let's pick as our proposal distribution,

$$q(x_2|x_1, \mathbf{x}^-) = \pi(x_2|\mathbf{x}^-) \tag{15.8.9}$$

Look what happens: The second argument of the min function becomes 1, so the acceptance probability α is also 1. In other words, if we propose a value x_2 from the full conditional distribution $\pi(x_2|\mathbf{x}^-)$, we can always accept it. The advantage is obvious. The disadvantage is that the full conditional distribution *must* be properly normalized as a probability distribution — otherwise how could we use it as a transition probability? Thus, we will usually need to calculate (either analytically or by numerical integration) the normalizing constant

$$\int \pi(x|\mathbf{x}^-)dx \tag{15.8.10}$$

for every $\mathbf{x}^-$ of interest, and we will need to have a practical algorithm for drawing x_2 from the thus-normalized distribution. Note that these one-dimensional normalizing constants are *much* easier to compute than would be the multidimensional normalizing constant for the whole distribution $\pi(\mathbf{x})$.

The full Gibbs sampler operates as follows: Cycle through each component of $\mathbf{x}$ in turn. (A fixed cyclical order is usually used, but choosing a component randomly each time is also fine.) For each component, hold all the other components fixed and draw a new value x from the full conditional distribution $\pi(x \mid \mathbf{x}^-)$ of all possible values of that component. (This is where you might have to do a numerical integral at each step.) Set the component to the new value and go on to the next component.

You can see that the Gibbs sampler is "more global" than the regular Metropolis-Hastings algorithm. At each step, a component of $\mathbf{x}$ gets reset to a value completely independent of its previous value (independent, at least, in the conditional distribution). If we tried to get behavior like this with regular Metropolis-Hastings, by proposing really big multivariate normal steps, say, we would get nowhere, since the steps would be almost always rejected!

On the other hand, the need to draw from a normalized conditional distribution can be a real killer in terms of computational workload. Gibbs sampling can be recommended enthusiastically when the components of $\mathbf{x}$ have discrete, not continuous, values, and not too many possible values for each component. In that case the normalization is just a sum over not-too-many terms, and the Gibbs sampler can be very efficient. For the case of continuous variables, you are probably better off with regular Metropolis-Hastings, unless your particular problem admits to some fast, tricky way of getting the normalizations.

Don't confuse the Gibbs sampler with the tactic of doing regular Metropolis-Hastings steps along one component at a time. For the latter, we restrict the proposal distribution to proposing a change in a single component, either randomly chosen or else cycling through all the components in a regular order. This is sometimes useful if it lets us compute $\pi(\mathbf{x})$ more efficiently (e.g., using saved pieces from the previous calculation on components that have not changed). What makes this *not* Gibbs is that we calculate an acceptance probability in the regular way, with equation (15.8.5) and the full distribution $\pi(\mathbf{x})$, which need not be normalized.

15.8.3 MCMC: A Worked Example

A number of practical details regarding MCMC are best discussed in the context of a worked example:

> *At the beginning of an experiment, events occur Poisson randomly with a mean rate λ_1, but only every k_1th event is recorded. Then, at time t_c, the mean rate changes to λ_2, but now only every k_2th event is recorded. We are given the times $t_0, \ldots, t_{N-1}$ of the N recorded events. Oh, by the way, the values λ_1, λ_2, k_1, k_2, and t_c are all unknown. We want to find them.*

Let's decompose the separate parts of the calculation into separate objects. First we need an object that represents the point $\mathbf{x}$. Although we've been discussing $\mathbf{x}$ as if it were a vector, it can actually be a mixture of continuous, discrete, boolean, or any other kind of variable. In our example we have both continuous and discrete variables.

```
struct State {                                                  mcmc.h
Worked MCMC example: Structure containing the components of x.
    Doub lam1, lam2;              λ1 and λ2
    Doub tc;                      tc
    Int k1, k2;                   k1 and k2
    Doub plog;                    Set to log P by Plog, below.

    State(Doub la1, Doub la2, Doub t, Int kk1, Int kk2) :
        lam1(la1), lam2(la2), tc(t), k1(kk1), k2(kk2) {}
    State() {};
};
```

The constructor is used to set initial values. (The `plog` variable is not part of $\mathbf{x}$, but it will be used later.)

Next, we need an object for calculating $\pi(\mathbf{x}) = P(\mathbf{D}|\mathbf{x})$, the probability of the data given the parameters. For our example, we need to use a couple of facts about Poisson processes: If a Poisson process has a rate λ, then the waiting time to the kth event is distributed as Gamma(k, λ), that is,

$$p(\tau|k,\lambda) = \frac{\lambda^k}{(k-1)!}\tau^{k-1}e^{-\lambda\tau} \tag{15.8.11}$$

where $\tau = t_{i+k} - t_i$. (Compare equation 6.14.41, and also §7.3.10.) The exponential distribution is a special case with $k = 1$. Further, probabilities for non-overlapping intervals such as $t_{i+k} - t_i$ and $t_{i+2k} - t_{i+k}$ are independent. It follows that, for our example,

$$P(\mathbf{D}|\mathbf{x}) = \prod_{t_i \le t_c} p(t_{i+1} - t_i \,|\, k_1, \lambda_1) \times \prod_{t_i > t_c} p(t_{i+1} - t_i \,|\, k_2, \lambda_2) \tag{15.8.12}$$

where $p(\tau|k,\lambda)$ is as given in (15.8.11), and where t_i is now the ith *recorded* time. (In the words following equation 15.8.11, t_i was the ith event whether recorded or not.)

Actually, as the amount of data gets large, $P(\mathbf{D}|\mathbf{x})$ is likely to over- or underflow, so it is best to calculate $\log P$. It is important to make this calculation as efficient as possible, because it will be done at every step. Particularly important is to minimize the amount of looping over all the data points. In our example, if you take the logarithm of equations (15.8.11) and (15.8.12), you'll see that the individual t_i's enter into $\log P$ only as a sum of intervals and sum of log of intervals, less than and greater than t_c. An efficient way to proceed is thus to digest the data once and store two cumulative sums. Then, given a value t_c, we can find our place in the table of sums by bisection and read off the left and right sums directly. There is thus no loop over the data at all! Life is rarely so good, but when it is, then *carpe diem*. The resulting object looks like this:

mcmc.h
```
struct Plog {
Functor that calculates log P of a State.
    VecDoub &dat;                               Bind to data vector.
    Int ndat;
    VecDoub stau, slogtau;

    Plog(VecDoub &data) : dat(data), ndat(data.size()),
    stau(ndat), slogtau(ndat) {
    Constructor. Digest the data vector for subsequent fast calculation of log P. The data are
    assumed to be sorted in ascending order.
        Int i;
        stau[0] = slogtau[0] = 0.;
        for (i=1;i<ndat;i++) {
            stau[i] = dat[i]-dat[0];            Equal to sum of intervals.
            slogtau[i] = slogtau[i-1] + log(dat[i]-dat[i-1]);
        }
    }

    Doub operator() (State &s) {
    Return log P of s, and also set s.plog.
        Int i,ilo,ihi,n1,n2;
        Doub st1,st2,stl1,stl2, ans;
        ilo = 0;
        ihi = ndat-1;
        while (ihi-ilo>1) {                     Bisection to find where is t_c in the data.
            i = (ihi+ilo) >> 1;
            if (s.tc > dat[i]) ilo=i;
            else ihi=i;
        }
        n1 = ihi;
        n2 = ndat-1-ihi;
        st1 = stau[ihi];
        st2 = stau[ndat-1]-st1;
        stl1 = slogtau[ihi];
        stl2 = slogtau[ndat-1]-stl1;
        Equations (15.8.11) and (15.8.12):
        ans =  n1*(s.k1*log(s.lam1)-factln(s.k1-1))+(s.k1-1)*stl1-s.lam1*st1;
        ans += n2*(s.k2*log(s.lam2)-factln(s.k2-1))+(s.k2-1)*stl2-s.lam2*st2;
        return (s.plog = ans);
    }
};
```

The `Plog` object is the only place that the data enter, and they enter only through the constructor. All other parts of the calculation see the data only through the calculation of $\log P$.

Next we come to the proposal generator, which we call `Proposal`. It doesn't have any contact with the data, or with $\log P$. All it needs to know about is the domain of $\mathbf{x}$ (that is, `State`). It is worth thinking hard about the proposal generator. Although "almost any" generator will work in theory, a poor generator will take longer than the age of the universe to converge, while a good, *rapidly mixing* generator can go like lightning. This is where MCMC starts becoming an art.

Our example is designed to furnish an illustration of this in the interaction between the λ parameters and their corresponding k's. The mean rate of recorded counts is λ/k. Since λ is a continuous variable, we will be proposing relatively small changes in it at each step. Since k is discrete, there is no such thing as a small change, especially when k is small.

If we naively write a generator that proposes random independent changes in λ and k, then, after we have settled down to roughly the right value of λ/k, essentially all proposals for changing k will be rejected. The reason is that the acceptable step in λ required for a change in k from 1 to 2 (say) is so large (doubling λ) that our generator will pick it only, say, every billion years! If we are not smart enough to recognize this problem ahead of time, we can find it experimentally by inspecting the Markov chain as it evolves and noting the proposals to change k are never accepted.

A solution in our case is to have two kinds of steps. The first changes λ (by a small amount) and keeps k fixed. The second changes k and λ, keeping λ/k fixed. We choose randomly between the two kinds of steps, mostly choosing the first kind.

The general issue here is what to do when $\pi(\mathbf{x})$ defines some highly correlated directions among the components in $\mathbf{x}$. If you can recognize these directions, your proposal generator should, at least sometimes, generate proposals along them. Otherwise, it will have to propose very small steps, if they are ever to be accepted. In our example, this latter choice was made impossible by the discreteness in k, forcing us to diagnose and confront the issue directly. So, although `Proposal` doesn't directly have to know about $\log P$, *you* may need a qualitative understanding of $\log P$ when you design `Proposal`.

Since only `Proposal` knows the algorithm by which a proposal is generated, this object must also calculate, and return, the ratio $q(\mathbf{x}_1|\mathbf{x}_{2c})/q(\mathbf{x}_{2c}|\mathbf{x}_1)$, which is needed in equation (15.8.5). Here is an example that proposes small lognormal steps for the variables λ_1, λ_2, and t_c, or else proposes incrementing k_1 and k_2 by 1, 0, or -1, with corresponding changes in the λ's as described above.

mcmc.h

```
struct Proposal {
Functor implementing the proposal distribution.
    Normaldev gau;
    Doub logstep;

    Proposal(Int ranseed, Doub lstep) : gau(0.,1.,ranseed), logstep(lstep) {}

    void operator() (const State &s1, State &s2, Doub &qratio) {
    Given state s1, set state s2 to a proposed candidate. Also set qratio to q(s1|s2)/q(s2|s1).
        Doub r=gau.doub();
        if (r < 0.9) {                          Lognormal steps holding the k's constant.
            s2.lam1 = s1.lam1 * exp(logstep*gau.dev());
            s2.lam2 = s1.lam2 * exp(logstep*gau.dev());
            s2.tc = s1.tc * exp(logstep*gau.dev());
            s2.k1 = s1.k1;
            s2.k2 = s1.k2;
            qratio = (s2.lam1/s1.lam1)*(s2.lam2/s1.lam2)*(s2.tc/s1.tc);
            Factors for lognormal steps.
        } else {                                Steps that change k1 and/or k2.
            r=gau.doub();
            if (s1.k1>1) {
                if (r<0.5) s2.k1 = s1.k1;
                else if (r<0.75) s2.k1 = s1.k1 + 1;
                else s2.k1 = s1.k1 - 1;
            } else {                          k1 = 1 requires special treatment.
                if (r<0.75) s2.k1 = s1.k1;
                else s2.k1 = s1.k1 + 1;
            }
            s2.lam1 = s2.k1*s1.lam1/s1.k1;
            r=gau.doub();                     Now all the same for k2.
            if (s1.k2>1) {
                if (r<0.5) s2.k2 = s1.k2;
                else if (r<0.75) s2.k2 = s1.k2 + 1;
                else s2.k2 = s1.k2 - 1;
            } else {
                if (r<0.75) s2.k2 = s1.k2;
                else s2.k2 = s1.k2 + 1;
            }
            s2.lam2 = s2.k2*s1.lam2/s1.k2;
            s2.tc = s1.tc;
            qratio = 1.;
        }
    }
};
```

(We use the convenient fact that since `Normaldev` is derived from `Ran`, it contains both normal and uniform random number generators.)

How shall we set `logstep`, the size of the proposed lognormal step? A rule of thumb for proposals like this with an adjustable scale is that the average acceptance probability ought to be roughly between 0.1 and 0.4. If it is very much smaller, then decrease the step size parameter; if it is much larger, then increase the step size parameter. In our example, the value `logstep` $= 0.01$ (i.e., proposed changes on the order of $\pm 1\%$) gives good results.

Finally, there is a function that takes a specified number of steps, implementing equation (15.8.5). This short piece of code is about the only "universal" part of MCMC; it has no persistent state and gets all the information it needs via the `State`, `Plog`, and `Proposal` structures. As we have seen, these are all problem-dependent and benefit from cleverness and special tricks.

mcmc.h
```
Doub mcmcstep(Int m, State &s, Plog &plog, Proposal &propose) {
Take m MCMC steps, starting with (and updating) s.
    State sprop;                                  Storage for candidate.
    Doub qratio,alph,ran;
    Int accept=0;
    plog(s);
    for (Int i=0;i<m;i++) {                       Loop over steps.
        propose(s,sprop,qratio);
        alph = min(1.,qratio*exp(plog(sprop)-s.plog));        Equation (15.8.5).
        ran = propose.gau.doub();
        if (ran < alph) {                         Accept the candidate.
            s = sprop;
            plog(s);
            accept++;
        }
    }
    return accept/Doub(m);
}
```

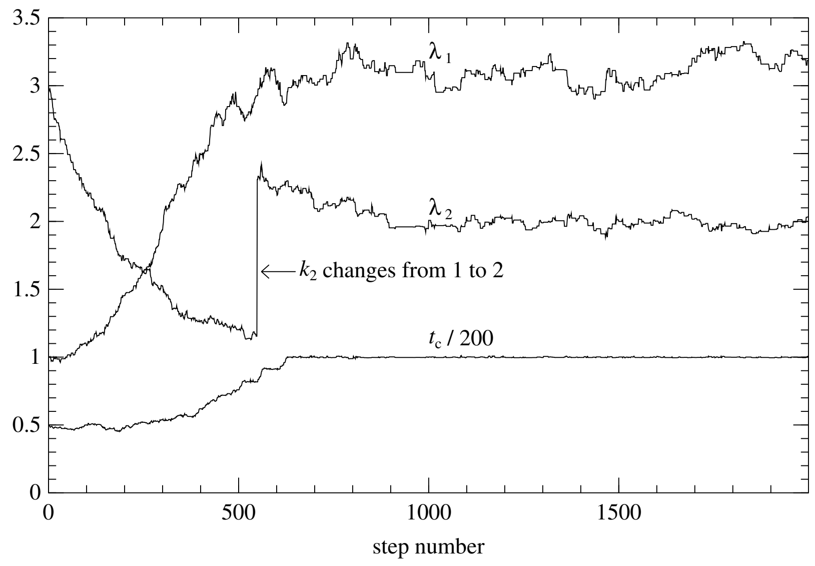


Figure 15.8.1. Evolution of model parameters λ_1, λ_2, and t_c as a function of Markov chain Monte Carlo step. In this example, the burn-in time is seen to be ~ 1000 steps, after which the Markov chain explores the equilibrium distribution.

Let's try it all out. We'll assume $N = 1000$ data points t_i and start $\mathbf{x}$ with the values $\lambda_1 = 1$, $\lambda_2 = 3$, $t_c = 100$, and $k_1 = k_2 = 1$. (Secretly, we know that the data were generated using actual values $3, 2, 200, 1, 2$, respectively.) The random seed is 10102, and the lognormal stepsize is 0.01. We'll take 1000 steps of burn-in, and thereafter store values after every 10 steps. Driver code in `main` for this run is

```
VecDoub times(1000);
...                                Fill the vector times here.
State s(1.,3.,100.,1,1);
Plog plog(times);
Proposal propose(10102,.01);
for (i=0;i<1000;i++) accept = mcmcstep(1,s,plog,propose);   Burn-in.
for (i=0;i<10000;i++) {                                      Production.
    accept = mcmcstep(10,s,plog,propose);
    ...                            Save values, increment averages, etc., here.
}
```

Figure 15.8.1 shows the evolution of the parameters λ_1, λ_2 and t_c. During burn-in, you can see the parameters heading toward equilibrium, mostly monotonically, but with the exception of λ_2, which goes rapidly toward the value 1, with the value $k = 1$. These values indeed replicate the mean rate of the recorded data. Only when it is near convergence (around step 560), does the model discover that the t_i's greater than t_c don't actually fit an exponential distribution ($k_2 = 1$) but *do* fit a gamma distribution with the same mean rate, but with $k_2 = 2$ (the correct answer). Had we not provided `Proposal` with a step that tests for this, we would likely have converged to a wrong answer. More precisely, we would have produced a model whose true burn-in time was, unknowably, a figurative billion years.

Figure 15.8.2 shows how λ_1 and λ_2 distribute themselves during 10^5 steps after step 1000. This is the payoff of MCMC: We learn not just about most likely parameter values, but also details about how well the parameters are determined by this

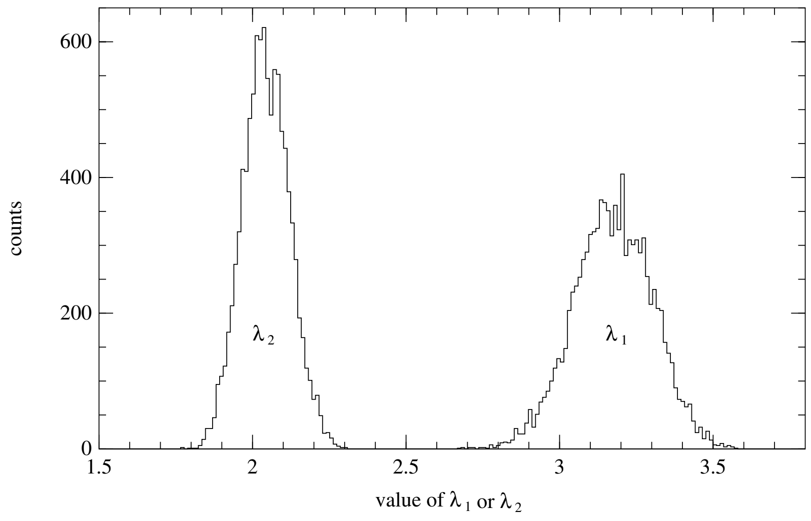


Figure 15.8.2. After burn-in, the MCMC model is run for an additional 10^5 steps. Parameter values are saved every 10 steps, giving the histograms shown for parameters λ_1 and λ_2. These histograms represent the inferred parameter values and their uncertainties. The model data were generated with parameters $\lambda_1 = 3$ and $\lambda_2 = 2$. The inferred values for this particular sample of 1000 data points are seen to be about as accurate as should be expected from their uncertainties.

particular data set. We could also have shown any joint distribution of interest, or computed any average quantity, for example

$$\langle \lambda_1 \rangle = \frac{1}{n-k} \sum_{i=k}^{n-1} (\lambda_1)_i \tag{15.8.13}$$

Here $k = 1000$ is the number of burn-in steps, which we reject; $n = k + 10^5$ is the number of steps that are averaged; and $(\lambda_1)_i$ denotes the value of λ_1 at the ith step. Sums like (15.8.13) are called *ergodic averages*.

Some remarks should be made about equation (15.8.13): One is allowed to average all steps, even though successive steps are not independent samples of $\pi(\mathbf{x})$. One would also be allowed to include in the average only every mth step, where you choose m to be greater than some empirically observed correlation time in the Markov chain. The latter is sometimes recommended as a means of estimating the standard error of $\langle \lambda_1 \rangle$, just as in Monte Carlo integration. (Compare equations 7.7.1 and 7.7.2.) Warning: Doing this in the context of a finite data set is often associated with conceptual error. While it is true that as $n \to \infty$, equation (15.8.13) does converge to a precise value, it is *not* true that this value has anything to do with the actual (population) value of λ_1. Rather, it is just the best apparent (sample) value of λ_1 for this particular data set. The relation of this apparent value to the actual value has nothing to do with the standard error of $\langle \lambda_1 \rangle$, but is instead indicated by the width of the distribution of all the $(\lambda_1)_i$'s.

Figure 15.8.2 illustrates this point well. By running the model for a long time, we could achieve beautifully precise distributions that have extremely well-converged means. But they would not be centered on the (here secretly known) true values 3 and 2. You should run an MCMC model only (i) long enough to be sure (or sure enough) that there was sufficient burn-in, and (ii) long enough to characterize distributions well enough that the error in mean quantities of interest is reasonably small compared with the observed dispersion of those quantities in the Markov chain.

15.8.4 Other Aspects of MCMC

We have said little about how to determine the necessary length of burn-in, other than to advise you to *look at the output* (which is always a good idea). There is in fact a large literature on this subject [2-4]. A number of so-called *convergence diagnostic* tools have been developed. The problem is that, even when you use these tools, you really must *look at the output* anyway; so their added value is often not large. It is always a good idea to have the length of burn-in be *at least* 1 or 2% of the total length of your run, which will be determined by the accuracy that you need in estimating model parameters. Keep in mind that it is easy to construct scary examples of distributions $\pi(\mathbf{x})$ full of false convergence traps. The more you know about your distribution, the better off you will be.

Multiple, independent Markov chains can be run to explore a single distribution $\pi(\mathbf{x})$. On a single processor the only reason to do this would be to meet some unusual need for independent samples of the distribution. However, on machines with multiple processors, this is a natural way of achieving efficient parallelization.

We have assumed that the number of dimensions in $\mathbf{x}$ is fixed. It is possible to have models, however, in which the number of fitted parameters is itself a variable. These *variable dimension models* require special care in the design of proposal distributions that can step between different numbers of dimensions. See the paper by Phillips and Smith in [2] for an introduction.

15.8.5 Importance Sampling and MCMC

In §7.9 we noted that the error in Monte Carlo integration could be reduced by importance sampling, where we write (in the notation of this section)

$$I \equiv \int f(\mathbf{x})\,d\mathbf{x} = \int \frac{f(\mathbf{x})}{p(\mathbf{x})}\,p(\mathbf{x})\,d\mathbf{x} \tag{15.8.14}$$

for an aptly chosen p. We saw that the ideal p would (i) resemble f in functional form, cf. equation (7.9.6), and (ii) admit to a good method for sampling uniformly over $p(\mathbf{x})\,d\mathbf{x}$.

You might think that MCMC provides a great general-purpose way to sample over any p and thus make importance sampling easy to implement in all cases. Unfortunately, no. The problem, once again (as for the Gibbs sampler), is the normalizing constant. MCMC's great virtue is that it samples over a distribution $\pi(\mathbf{x})$ without requiring that it be normalized. If you ask what normalized probability distribution $p(\mathbf{x})$ is actually being sampled over, it is of course

$$p(\mathbf{x}) = \frac{\pi(\mathbf{x})}{\int \pi(\mathbf{x})d\mathbf{x}} \tag{15.8.15}$$

Equation (15.8.14) then becomes

$$I \equiv \int f(\mathbf{x})\,d\mathbf{x} = \int \pi(\mathbf{x})\,d\mathbf{x} \;\times\; \int \frac{f(\mathbf{x})}{\pi(\mathbf{x})}\,p(\mathbf{x})\,d\mathbf{x} \tag{15.8.16}$$

The differential $p(\mathbf{x})\,d\mathbf{x}$ can be sampled over by MCMC, knowing only $\pi(\mathbf{x})$; no problem. The $\pi(\mathbf{x})$ in the denominator of the integrand can also be readily computed. But we have, in general, no easy way to calculate that pesky normalizing constant, $\int \pi(\mathbf{x})\,d\mathbf{x}$.

Sometimes, though not often, you can construct a function $\pi(\mathbf{x})$ that both resembles f and also can be integrated analytically, so that the normalizing constant is knowable. Then, yes, by all means use MCMC to sample $\pi(\mathbf{x})$. In this case the idea of recording only every mth step, after choosing m large enough so that the points thus chosen are independent samples, is not a bad idea after all. In fact you'll have to do this if you expect to use the error estimate in equation (7.9.3) as written.

Finally, if the integral that you really want is

$$J \equiv \frac{\int f(\mathbf{x})\pi(\mathbf{x})\,d\mathbf{x}}{\int \pi(\mathbf{x})\,d\mathbf{x}} = \int f(\mathbf{x})\,p(\mathbf{x})\,d\mathbf{x} \qquad (15.8.17)$$

with $f(\mathbf{x})$ and $\pi(\mathbf{x})$ both known (and $p(\mathbf{x})$ only implied), then MCMC is exactly what you need. It provides uniform samples over $p(\mathbf{x})d\mathbf{x}$, and no calculation of a normalizing constant is needed.

CITED REFERENCES AND FURTHER READING:

Hastings, W.K. 1970, "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika*, vol. 57, pp. 97–109.[1]

Gilks, W.R., Richardson, S., and Spiegelhalter, D.J., eds. 1996, *Markov Chain Monte Carlo in Practice* (Boca Raton, FL: Chapman & Hall/CRC), especially Chapter 1.[2]

Gamerman, D. 1997, *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference* (London: Chapman & Hall). [3]

Neal, R.M. 1993, "Probabilistic Inference Using Markov Chain Monte Carlo Methods," *Technical Report CRG-TR-93-1*, Department of Computer Science, University of Toronto. Available at `http://www.cs.toronto.edu/~radford/ftp/review.pdf`. [4]

Casella, G., and George, E.I. 1992, "Explaining the Gibbs Sampler," *American Statistician*, vol. 46, no. 3, pp. 167–174.[5]

Tanner, M.A. 2005, *Tools for Statistical Inference: Methods for the Exploration of Posterior Distributions and Likelihood Functions*, 3rd ed. (New York: Springer).

Liu, J.S. 2002, *Monte Carlo Strategies in Scientific Computing* (New York: Springer).

Beichl, I., and Sullivan, F. (eds.) 2006, *Computing in Science and Engineering*, special issue on Monte Carlo Methods, vol. 8, no. 2 (March/April), pp. 7–47.

15.9 Gaussian Process Regression

Some types of statistical models do not depend on knowing (or guessing) parameterized functional forms, and thus lie outside of the parameter-fitting paradigm that has thus far occupied our attention. As an alternative to assuming that our data have some functional form, we can assume that they have some statistical property. A common example is to assume that the data, viewed as an entire set, is drawn from some multivariate normal (Gaussian) distribution in a high-dimensional space. That distribution is allowed to have a complicated correlation structure: The individual data points are *not* assumed to be independent. We can then ask, *given* the data points that we observe, what are the most probable values for other quantities of interest, for example the values of variables at points other than the ones measured. Of course, as previously, we are also encouraged to ask not just about the most probable values, but about the whole distribution around the most probable values. This general scheme is called *Gaussian process regression*.

We have already met examples of Gaussian process regression twice before in this book, though under different names. In §3.7 we discussed *kriging* as a multidimensional interpolation technique. Later, in §13.6, we discussed *linear prediction*, mostly in the context of one-dimensional data such as time series. Here we can usefully merge some of the ideas in those two sections.

As we presented it in §3.7, kriging was an interpolation, not a fitting, technique. This was evident from the facts that (i) the interpolated function output by the `Krig` object went exactly through the measured data points, and (ii) we never discussed how to input measurement errors. However, the `Krig` object's constructor did have an argument `err`, introduced with the mysterious remark that you should leave it set to `NULL` until you read §15.9. Well, here we are!

We did incorporate measurement errors in §13.6, although they were there called *noise*. In particular, equations (13.6.6) and (13.6.7) can be used (after some change of notation and algebraic manipulation) to derive the appropriate generalization of equations (3.7.14) and (3.7.15) to the case where the measurements y_i, $i = 0, \ldots, N-1$, have errors characterized by some covariance matrix $\mathbf{\Sigma}$. In most cases $\mathbf{\Sigma}$ will be simply a diagonal matrix with elements σ_i^2, the squares of the individual errors. The answers are

$$\widehat{y}_* = \mathbf{V}_* \cdot (\mathbf{V} - \mathbf{\Sigma}')^{-1} \cdot \mathbf{Y} \tag{15.9.1}$$

and

$$\mathrm{Var}(\widehat{y}_*) = \mathbf{V}_* \cdot (\mathbf{V} - \mathbf{\Sigma}')^{-1} \cdot \mathbf{V}_* \tag{15.9.2}$$

where

$$\mathbf{\Sigma}' \equiv \begin{pmatrix} \mathbf{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \tag{15.9.3}$$

That is, we simply subtract $\mathbf{\Sigma}$ (suitably augmented by bordering zeros) from $\mathbf{V}$ (equation 3.7.13) before inverting the matrix. The argument `err`, input as the σ_i's (not squared), does this for the case of diagonal measurement errors. Note that `err` has type `Doub*`. If your errors are stored in a `VecDoub`, then you'll send `&err[0]` to the `Krig` constructor. (Sorry about this hack. The purpose was to make `NULL` a possible default value.)

So, no new code is needed in this section. In `Krig`, you already have a serviceable multidimensional Gaussian process regression fitting routine, all ready to go.

When you are fitting, rather than interpolating, it is a good idea to pay more attention to the choice of variogram model than we did in §3.7. While for simple applications there is nothing wrong with the power-law model implemented in the `Powvargram` object

$$v(r) = \alpha r^{\beta} \tag{15.9.4}$$

several other models are widely used. These include the *exponential model*,

$$v(r) = b[1 - \exp(-r/a)] \tag{15.9.5}$$

the *spherical model*,

$$v(r) = \begin{cases} b\left(\frac{3}{2}\frac{r}{a} - \frac{1}{2}\frac{r^3}{a^3}\right) & 0 \le r \le a \\ b & a \le r \end{cases} \tag{15.9.6}$$

and various anisotropic models for which $v(\mathbf{r})$ is not just a function of the magnitude r. See [1,2] for derivations and examples.

We should also mention the so-called *nugget effect*, though, in our opinion, its name vastly outshines its utility. If $v(\mathbf{r})$ does not go to zero as $\mathbf{r} \to 0$, but instead goes to some constant v_0, then the resulting variogram describes a distribution that decorrelates by some finite amount in an infinitesimal distance. That is, if you find a gold nugget at location $\mathbf{x}$, there is no certainty that you'll find another one at location $\mathbf{x} + \delta\mathbf{x}$, no matter how small you make $\delta\mathbf{x}$. Some practitioners deem it desirable to allow for a nonzero nugget effect, allowing nonzero values of v_0 when they empirically fit $v(r)$ from a data set. That seems debatable to us; but in deference to such opinion we have given the `Powvargram` constructor an otherwise undocumented argument, `nug`, for feeding in the value v_0 of your choice. (We draw the line at actually fitting for such a parameter!)

Beyond debatable, and actually incorrect, however, is to confuse the nugget effect with the effect of measurement error. They seem superficially similar: Measurement error also decorrelates measured values, even at arbitrarily small distances (even zero). Conceptually, and mathematically, however, they are different. Referring to equation (3.7.13), a nugget effect adds a constant positive value to all the *off-diagonal* v_{ij}'s. Measurement errors, on the other hand, subtract (not necessarily constant) negative values from the *diagonal* v_{ii}'s. These actions do not have equivalent effects on equations (3.7.14) and (3.7.15). This can readily be seen in Figure 15.9.1, which may also help elucidate the difference between kriging interpolation and kriging fitting. Only panel (d) in the figure shows a correct use of kriging for data with measurement errors, that is, kriging fitting with errors σ_i. Panels (b) and (c) show the results of kriging interpolation with and without a nugget effect. One sees that even with a positive nugget, the interpolated curve goes exactly through the data points, which is incorrect when measurement errors are significant. The legitimate use of kriging interpolation (as in §3.7) is for smooth functions that are "exactly" known at scattered points. Kriging fits using σ_i's (this section) are for data with errors.

CITED REFERENCES AND FURTHER READING:

Cressie, N. 1991, *Statistics for Spatial Data* (New York: Wiley).[1]

Wackernagel, H. 1998, *Multivariate Geostatistics*, 2nd ed. (Berlin: Springer).[2]

Isaaks, E.H., and Srivastava, R.M. 1989, *Applied Geostatistics* (New York: Oxford University Press).

Rasmussen, C.E., and Williams, C.K.I. 2006, *Gaussian Processes for Machine Learning* (Cambridge, MA: MIT Press).

Rybicki, G.B., and Press, W.H. 1992, "Interpolation, Realization, and Reconstruction of Noisy, Irregularly Sampled Data," *Astrophys. J.*, vol. 398, pp. 169–176.

Deutsch, C.V., and Journel, A.G. 1992, *GSLIB: Geostatistical Software Library and User's Guide* (New York: Oxford University Press).

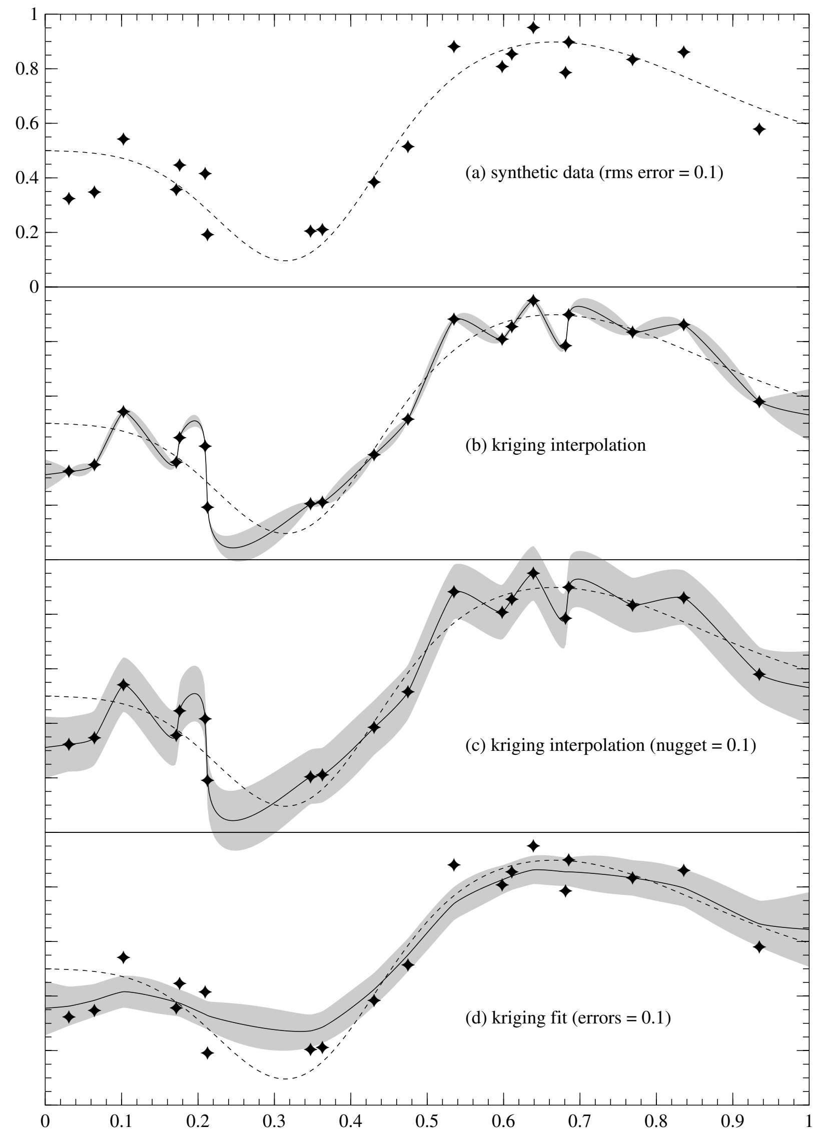


Figure 15.9.1. One-dimensional examples of interpolation and fitting by kriging. (a) Synthetic data points generated from the known curve (dashed line) with Gaussian errors (r.m.s. magnitude 0.1). (b) Result of kriging interpolation. Equation (3.7.14) is plotted as the solid line, while the 1-σ estimated interpolation errors (3.7.15) are shown as the shaded band. The interpolation error is seen to be meaningless for data with measurement errors. (c) Same as (b), but with a nugget effect of $\mathbf{0.1}$. (d) Result of kriging fit (equations 15.9.1 and 15.9.2) using the actual measurement errors. This is the correct use of kriging for data with errors.

CHAPTER 16

Classification and Inference

16.0 Introduction

This chapter groups together a selection of computational techniques whose common feature is that they treat problems of classification and inference on complex models. Given substantially more space, the chapter might have been expanded to be a more complete survey of *machine learning*; but at its present length, it cannot pretend to be such. (A few general references are given below.)

Classification and inference, in a loose sense, are also the goals of many of the purely statistical methods that we already discussed in Chapters 14 and 15, and the line between such techniques and machine learning is a fuzzy one. This chapter's topics tend to have one or both of two characteristics: the underlying model (i) has discrete or combinatorial aspects that distinguish it from "classical" statistical methods, and/or (ii) has empirical or heuristic aspects that make exact statistical treatments unattainable.

There is a list of important topics, related to those in this chapter, that we would have wanted to include if only they could have been reduced to suitable length. *Bayesian networks* is at the top of this list. Section 16.0.1 gives an example of the kind of problem that a Bayesian network can solve. Other significant topics that we must omit include

- genetic algorithms
- neural nets
- kernel methods more general than those discussed in §16.5

16.0.1 Bayesian networks

These are sometimes called *Bayes nets*, *Bayesian learning networks*, or *belief networks*. Here we want only to give a flavor of the method, so that you will know when to consult the references below.

A Bayesian network consists of nodes, each of which can have a value. The values can be {true,false}, or a set of possibilities like {low, medium, high}, or an integer. Figure 16.0.1 shows an example where all the nodes have true/false values.

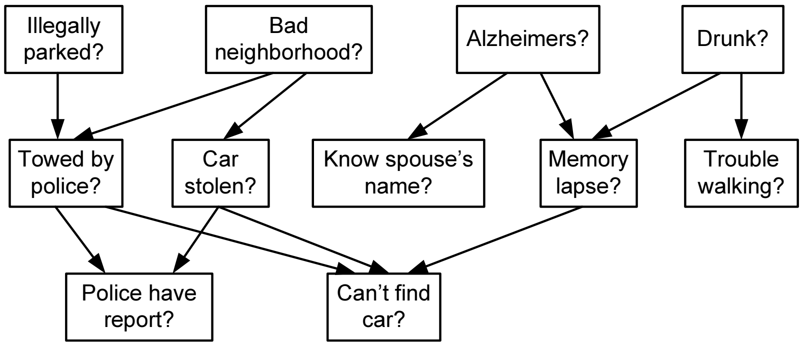


Figure 16.0.1. Example of a Bayesian network. Evidence about any node can be propagated to give probabilistic conclusions about any other node.

Each node in a network has a set of *prior probabilities* or *priors* that give the likelihood of its values absent any additional evidence. If a node has one or more *parents*, then its priors are conditioned on the values of the parents. For example, referring to the Figure, we might have

P(Illegally-Parked = true)
0.20

Bad-N'hood	P(Car-Stolen = true | Bad-N'hood)
true	0.05
false	0.001

Alzheimers	Drunk	P(Memory-Lapse = true | Alzheimers, Drunk)
true	true	0.999
true	false	0.95
false	true	0.50
false	false	0.01

And so on, for all the other nodes.

Things get interesting when we have some evidence to assimilate into the network. For example, you might be coming out of a bar in a bad neighborhood, walking with some difficulty, and be unable to find your car. Is it stolen? The Bayesian network theory gives algorithms for propagating information both up (from "Can't find car?") and down (from "Bad neighborhood?") to get new *posterior* estimates for the probabilities at all nodes, including, here, "Car stolen?" You can also compute in advance the value of new evidence. For example, how much would it help to call the police and see if there is a police report of a towed or recovered, stolen, car?

For more than this brief taste, see [1-3].

CITED REFERENCES AND FURTHER READING:

Hastie, T., Tibshirani, R., and Friedman, J.H. 2003, *The Elements of Statistical Learning* (Berlin: Springer).

Duda, R.O., Hart, P.E., and Stork, D.G. 2000, *Pattern Classification*, 2nd ed. (New York: Wiley).

Witten, I.H., and Frank, E. 2005, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. (San Francisco: Morgan Kaufmann).

Mitchell, T.M. 1997, *Machine Learning* (New York: McGraw-Hill).

Vapnik, V. 1998, *Statistical Learning Theory* (New York: Wiley).

Russell, S., and Norvig, P. 2002, *Artificial Intelligence: A Modern Approach*, 2nd ed. (Upper Saddle River, NJ: Prentice-Hall).

Haykin, S. 1998, *Neural Networks: A Comprehensive Foundation*, 2nd ed. (Upper Saddle River, NJ: Prentice-Hall).

Bishop, C.M. 1996, *Neural Networks for Pattern Recognition* (New York: Oxford University Press).

Korb, K.B., and Nicholson, A.E. 2004, *Bayesian Artificial Intelligence* (Boca Raton, FL: Chapman & Hall/CRC).[1]

Neapolitan, R.E. 1990, *Probabilistic Reasoning in Expert Systems* (New York: Wiley).[2]

Jensen, F.V. 2001, *Bayesian Networks and Decision Graphs* (New York: Springer).[3]

16.1 Gaussian Mixture Models and k-Means Clustering

Gaussian mixture models, so called, are one of the simplest examples of classification by *unsupervised learning*. They are also one of the simplest examples where solution by the *EM (expectation-maximization) algorithm* proves highly successful.

Here is the setup: You are given N data points in an M-dimensional space, usually with M in the range one to a few (say, three or four dimensions, tops). You want to "fit" the data, in this special sense: Find a set of K multivariate Gaussian distributions that best represents the observed distribution of data points. The number K is fixed in advance but the means and covariances of the distributions are unknown,

What makes the exercise "unsupervised" is that you are *not told* which of the N data points come from which of the K Gaussians. Indeed, one of the desired *outputs* is, for each data point n, an estimate of the probability that it came from distribution number k. This probability is denoted $P(k|n)$ or p_{nk}, where (using a zero-based counting scheme) $0 \leq k < K$ and $0 \leq n < N$. The matrix p_{nk} is sometimes called the *responsibility matrix*, because its entries indicate how much "responsibility" component k has for data point n.

Thus, given the data points, say as an $N \times M$ matrix whose rows are vectors of length M, there are a whole bunch of parameters that we want to estimate:

$$\begin{aligned} &\boldsymbol{\mu}_k && \text{(the } K \text{ means, each a vector of length } M\text{)} \\ &\boldsymbol{\Sigma}_k && \text{(the } K \text{ covariance matrices, each of size } M \times M\text{)} \\ P(k|n) \equiv\ &p_{nk} && \text{(the } K \text{ probabilities for each of } N \text{ data points)} \end{aligned} \tag{16.1.1}$$

We will also get some additional estimates as by-products: $P(k)$ denotes the fraction of all data points in component k, that is, the probability that a data point chosen at random is in k; $P(\mathbf{x})$ denotes the probability (actually a probability density) of finding a data point at some position $\mathbf{x}$, where $\mathbf{x}$ is the M-dimensional position vector; and $\mathcal{L}$ denotes the overall likelihood of the estimated parameter set.

In fact, $\mathcal{L}$ is the key to the whole problem. $\mathcal{L}$ is defined, as usual, as proportional to the probability of the data set, *given* all the fitted parameters. We find the best values for the parameters by maximizing the likelihood $\mathcal{L}$. You can also think of this as maximizing the posterior probability of the parameters, given uniform or very broad priors.

Let's work backward from $\mathcal{L}$. Since the data points are (assumed) independent, $\mathcal{L}$ is the product of the probabilities of finding a point at each observed position $\mathbf{x}_n$,

$$\mathcal{L} = \prod_n P(\mathbf{x}_n) \tag{16.1.2}$$

We can split $P(\mathbf{x}_n)$ into its contribution from each of the K Gaussians and write

$$P(\mathbf{x}_n) = \sum_k N(\mathbf{x}_n \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) P(k) \tag{16.1.3}$$

where $N(\mathbf{x} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the multivariate Gaussian density,

$$N(\mathbf{x} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{M/2} \det(\boldsymbol{\Sigma})^{1/2}} \exp[-\tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu}) \cdot \boldsymbol{\Sigma}^{-1} \cdot (\mathbf{x} - \boldsymbol{\mu})] \tag{16.1.4}$$

$P(\mathbf{x}_n)$ is sometimes called the *mixture weight* of the data point $\mathbf{x}_n$. We can "take apart" $P(\mathbf{x}_n)$ into its K individual contributions, giving the individual probabilities

$$p_{nk} \equiv P(k|n) = \frac{N(\mathbf{x}_n \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) P(k)}{P(\mathbf{x}_n)} \tag{16.1.5}$$

Equations (16.1.2) through (16.1.5) are a prescription for calculating $\mathcal{L}$ and the p_{nk}'s, given the data, and given values for the $\boldsymbol{\mu}_k$'s, $\boldsymbol{\Sigma}_k$'s, and $P(k)$. In the language of the EM algorithm, this is called an *expectation step* or *E-step*.

But how do we get the $\boldsymbol{\mu}_k$'s, $\boldsymbol{\Sigma}_k$'s, and $P(k)$?

Suppose we *knew* the p_{nk}'s. A familiar theorem for the one-dimensional Gaussian distribution is that the maximum likelihood estimate of its mean is just the arithmetic mean of a set of points drawn from it. This theorem straightforwardly generalizes to yield maximum likelihood estimates for the means, and covariance matrices, of multivariate Gaussians. A further small generalization is that, since we know only probabilistically whether a particular point is drawn from a particular Gaussian, we should count only the appropriate fraction p_{nk} of each point. These considerations result in the following maximum likelihood estimates:

$$\begin{aligned} \hat{\boldsymbol{\mu}}_k &= \sum_n p_{nk} \mathbf{x}_n \Big/ \sum_n p_{nk} \\ \hat{\boldsymbol{\Sigma}}_k &= \sum_n p_{nk} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k) \otimes (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k) \Big/ \sum_n p_{nk} \end{aligned} \tag{16.1.6}$$

and, in a similar vein,

$$\hat{P}(k) = \frac{1}{N} \sum_n p_{nk} \tag{16.1.7}$$

"Hats" here denote estimators; however, this is a notational nicety that we will henceforth ignore. Equations (16.1.6) and (16.1.7) are the so-called *maximization step* or *M-step* of the EM algorithm.

What we have motivated thus far is that *right at* the maximum likelihood solution, both the E-step and the M-step relations will hold. That is, the maximum likelihood parameters are a stationary point for both E-steps and M-steps. The power of the EM algorithm derives from the more powerful theorem (beyond our scope to prove here) that, starting from *any* parameter values, an iteration of E-step followed by an M-step will increase the likelihood value $\mathcal{L}$; and that repeated iterations will converge to (at least a local) likelihood maximum. Often, happily, the convergence is to the global maximum.

The EM algorithm, in brief, is thus

- Guess starting values for the $\boldsymbol{\mu}_k$'s, $\boldsymbol{\Sigma}_k$'s, and fractions $P(k)$.
- Repeat: An E-step to get new p_{nk}'s and new $\mathcal{L}$, followed by an M-step to get new $\boldsymbol{\mu}_k$'s, $\boldsymbol{\Sigma}_k$'s, and $P(k)$.
- Quit when the value of $\mathcal{L}$ is no longer changing.

One important practical detail is that the values of the Gaussian density function will often be so small as to underflow to zero. It is therefore important to work with logarithms of these densities, rather than the densities themselves, e.g.,

$$\log N(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu})\cdot\boldsymbol{\Sigma}^{-1}\cdot(\mathbf{x}-\boldsymbol{\mu}) - \frac{M}{2}\log(2\pi) - \frac{1}{2}\log\det(\boldsymbol{\Sigma}) \tag{16.1.8}$$

A problem arises with equation (16.1.3), where we need to take the sum of quantities, all of which may be so small as to underflow if ever reconstructed from their logarithms. The solution to this problem is the so-called *log-sum-exp* formula,

$$\log\left(\sum_i \exp(z_i)\right) = z_{\max} + \log\left(\sum_i \exp(z_i - z_{\max})\right) \tag{16.1.9}$$

where the z_i's are the logarithms that we are using to represent small quantities and $z_{\max}$ is their maximum. Equation (16.1.9) guarantees that at least one exponentiation won't underflow, and that any that do could have been neglected anyway.

Figure 16.1.1 shows an example of how the EM algorithm converges to a solution with 1000 two-dimensional data points and four components. As the number of data points increases, the topography of the likelihood space gets smoother, with fewer local minima, so that it becomes more and more likely that the global maximum will be found (as in this case).

You should always inspect an EM solution for reasonableness. If you are getting hung up on an unacceptable local maximum, one strategy is to do a series of independent runs, using K randomly chosen data points as the starting means in each case. (Be sure that you don't duplicate a data point in the starting guesses.) Then pick the best one, i.e., the one that converges to the largest log-likelihood.

Here is a structure that implements the EM algorithm for Gaussian mixture models, given only the data points and initial estimates of the means $\boldsymbol{\mu}_k$. The constructor sets the problem up, and does one initial E-step and M-step. Thereafter, the user alternately calls the `estep()` and `mstep()` routines, until convergence is achieved, as signaled by the return value of `estep()`, the change in log-likelihood, becoming sufficiently small (say, 10^{-6}). The results are then available in the structure members `means`, `resp`, `frac`, and `sig`.

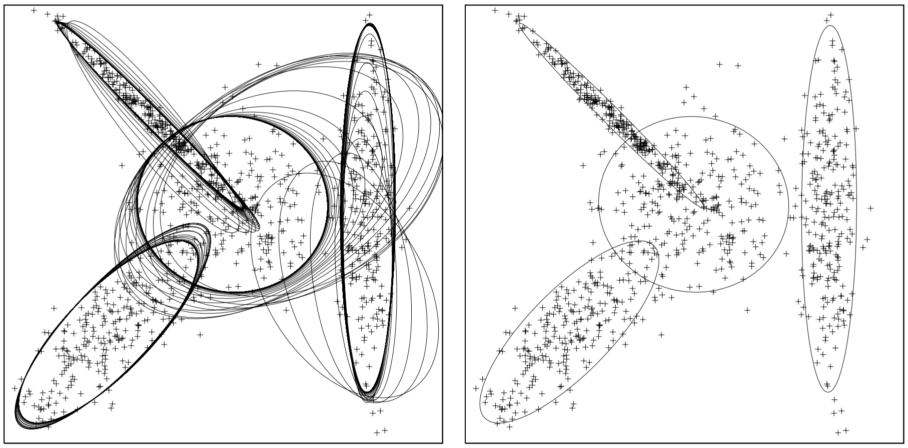


Figure 16.1.1. Example of a Gaussian mixture model in $M = 2$ dimensions, with $N = 1000$ data points and $K = 4$ components. Left: Evolution of the estimated means and covariances, shown as 2-sigma ellipses. The ellipses are plotted after each iteration of an E-step and M-step (see text). Right: The converged result. The leftmost two components converged rapidly. The rightmost component took about 10 iterations to get to near-convergence; only after it had done so did the central component shrink down to a converged result.

gaumixmod.h

```
struct preGaumixmod {
```
For nonwizards, this is basically a `typedev` of `Mat_mm` as an `mm`×`mm` matrix. For wizards, what is going on is that we need to set a static variable `mmstat` *before* defining `Mat_mm`, and this must happen *before* the `Gaumixmod` constructor is invoked.
```
    static Int mmstat;
    struct Mat_mm : MatDoub {Mat_mm() : MatDoub(mmstat,mmstat) {} };
    preGaumixmod(Int mm) {mmstat = mm;}
};
Int preGaumixmod::mmstat = -1;

struct Gaumixmod : preGaumixmod {
```
Solve for a Gaussian mixture model from a set of data points and initial guesses of k means.
```
    Int nn, kk, mm;                  Nos. of data points, components, and dimensions.
    MatDoub data, means, resp;       Local copies of x_n's, mu_k's, and the p_nk's.
    VecDoub frac, lndets;            P(k)'s and log det Sigma_k's.
    vector<Mat_mm> sig;              Sigma_k's
    Doub loglike;                    log L.
    Gaumixmod(MatDoub &ddata, MatDoub &mmeans) : preGaumixmod(ddata.ncols()),
    nn(ddata.nrows()), kk(mmeans.nrows()), mm(mmstat), data(ddata), means(mmeans),
    resp(nn,kk), frac(kk), lndets(kk), sig(kk) {
```
Constructor. Arguments are the data points (as rows in a matrix) and initial guesses for the means (also as rows in a matrix).
```
        Int i,j,k;
        for (k=0;k<kk;k++) {
            frac[k] = 1./kk;                     Uniform prior on P(k).
            for (i=0;i<mm;i++) {
                for (j=0;j<mm;j++) sig[k][i][j] = 0.;
                sig[k][i][i] = 1.0e-10;          See text at end of this section.
            }
        }
        estep();                                 Perform one initial E-step and M-step. User
        mstep();                                     is responsible for calling additional steps
    }                                                until convergence is obtained.
    Doub estep() {
```
Perform one E-step of the EM algorithm.
```
        Int k,m,n;
        Doub tmp,sum,max,oldloglike;
        VecDoub u(mm),v(mm);
        oldloglike = loglike;
        for (k=0;k<kk;k++) {                Outer loop for computing the $p_{nk}$'s.
            Cholesky choltmp(sig[k]);       Decompose $\Sigma_k$ in the outer loop.
            lndets[k] = choltmp.logdet();
            for (n=0;n<nn;n++) {            Inner loop for $p_{nk}$'s.
                for (m=0;m<mm;m++) u[m] = data[n][m]-means[k][m];
                choltmp.elsolve(u,v);       Solve $\mathbf{L}\cdot\mathbf{v} = \mathbf{u}$.
                for (sum=0.,m=0; m<mm; m++) sum += SQR(v[m]);
                resp[n][k] = -0.5*(sum + lndets[k]) + log(frac[k]);
            }
        }
```

At this point we have unnormalized logs of the p_{nk}'s. We need to normalize using log-sum-exp and compute the log-likelihood.

```
        loglike = 0;
        for (n=0;n<nn;n++) {                    Separate normalization for each n.
            max = -99.9e99;                     Log-sum-exp trick begins here.
            for (k=0;k<kk;k++) if (resp[n][k] > max) max = resp[n][k];
            for (sum=0.,k=0; k<kk; k++) sum += exp(resp[n][k]-max);
            tmp = max + log(sum);
            for (k=0;k<kk;k++) resp[n][k] = exp(resp[n][k] - tmp);
            loglike +=tmp;
        }
        return loglike - oldloglike;            When abs of this is small, then we have
    }                                               converged.
    void mstep() {
```

Perform one M-step of the EM algorithm.

```
        Int j,n,k,m;
        Doub wgt,sum;
        for (k=0;k<kk;k++) {
            wgt=0.;
            for (n=0;n<nn;n++) wgt += resp[n][k];
            frac[k] = wgt/nn;                   Equation (16.1.7).
            for (m=0;m<mm;m++) {
                for (sum=0.,n=0; n<nn; n++) sum += resp[n][k]*data[n][m];
                means[k][m] = sum/wgt;          Equation (16.1.6).
                for (j=0;j<mm;j++) {
                    for (sum=0.,n=0; n<nn; n++) {
                        sum += resp[n][k]*
                            (data[n][m]-means[k][m])*(data[n][j]-means[k][j]);
                    }
                    sig[k][m][j] = sum/wgt;     Equation (16.1.6).
                }
            }
        }
    }
};
```

About the only place that Gaumixmod can fail algorithmically (as distinct from converging to a poor, local, solution) is by encountering a zero or negative diagonal element in the Cholesky decomposition. As a result, all sins tend to appear, sometimes confusingly, as exceptions at that point in the code. If you are getting such exceptions, here are some possibilities:

- You have duplicated vectors in your initial guesses for the μ_k's.
- One or more of your μ_k's is so distant from all data points that it is not "attracting" enough of them to solve for the parameters of its component. Try using random data points as starting guesses, or reduce K.
- You may just have too few data points N to support a nondegenerate model with K components. Reduce K or get more data!

- Rarely, you might want to change the constant `1.0e-10` that initializes the diagonal components of $\mathbf{\Sigma}_k$ in the code. (See discussion under "K-Means Clustering," below.)
- You can reduce the number of parameters in $\mathbf{\Sigma}$, as we now discuss.

Occasionally data are too sparse, or too noisy, to give meaningful results for all the components of the covariance matrices $\mathbf{\Sigma}_k$. In such cases, you can impose simpler covariance models by changing the re-estimation formulas for $\mathbf{\Sigma}$ in equation (16.1.6). One step of simplification is to make $\mathbf{\Sigma}$ diagonal, while still allowing different variances for the different dimensions. The re-estimation formula for the diagonal components of $\mathbf{\Sigma}_k$ is then

$$(\widehat{\mathbf{\Sigma}}_k)_{mm} = \sum_n p_{nk}[(\mathbf{x}_n)_m - (\widehat{\boldsymbol{\mu}}_k)_m]^2 \Big/ \sum_n p_{nk} \tag{16.1.10}$$

where subscripts m indicate that particular component of the vector. Set nondiagonal components of $\mathbf{\Sigma}_k$ to zero.

Even more drastic, we can replace $\mathbf{\Sigma}_k$ by a single scalar (that is, spherical) variance by using the re-estimation formula

$$(\widehat{\mathbf{\Sigma}}_k) = \mathbf{1} \times \left(\sum_n p_{nk} |\mathbf{x}_n - \widehat{\boldsymbol{\mu}}_k|^2 \Big/ \sum_n p_{nk} \right) \tag{16.1.11}$$

where $\mathbf{1}$ is the identity matrix.

We have not coded these options in `Gaumixmod`, but they are easy to add.

16.1.1 A Note on the Use of Cholesky Decomposition

It is worth remarking briefly on the use of Cholesky decomposition (§2.9) in this and similar manipulations of multivariate Gaussians.

In the `Gaumixmod` routine above, we need a way of inverting the covariance matrices — or, more precisely, an efficient way to compute expressions like $\mathbf{y} \cdot \mathbf{\Sigma}^{-1} \cdot \mathbf{y}$. Because the covariance matrix $\mathbf{\Sigma}$ is symmetric and positive-definite, the Cholesky decomposition, which has fewer operations than other methods, can be used, giving

$$\mathbf{\Sigma} = \mathbf{L} \cdot \mathbf{L}^T \tag{16.1.12}$$

where $\mathbf{L}$ is a lower triangular matrix, implying

$$Q = \mathbf{y} \cdot \mathbf{\Sigma}^{-1} \cdot \mathbf{y} = \left| \mathbf{L}^{-1} \cdot \mathbf{y} \right|^2 \tag{16.1.13}$$

Since $\mathbf{L}$ is triangular, $\mathbf{L}^{-1} \cdot \mathbf{y}$ can be obtained efficiently by backsubstitution.

Another very convenient use for the decomposition (16.1.12) is in the mundane task of drawing error ellipses, as in Figure 16.1.1 (or, similarly, error ellipsoids in three dimensions). The locus of points $\mathbf{x}$ that are one standard deviation ("1-sigma") away from the mean $\boldsymbol{\mu}$ is given by

$$1 = (\mathbf{x} - \boldsymbol{\mu}) \cdot \mathbf{\Sigma}^{-1} \cdot (\mathbf{x} - \boldsymbol{\mu}) \quad \Rightarrow \quad \left| \mathbf{L}^{-1} \cdot (\mathbf{x} - \boldsymbol{\mu}) \right| = 1 \tag{16.1.14}$$

Now suppose that $\mathbf{z}$ is a point on the unit circle (two dimensions) or unit sphere (three dimensions). Then, by substitution into equation (16.1.14), you can easily see that

$$\mathbf{x} = \mathbf{L} \cdot \mathbf{z} + \boldsymbol{\mu} \tag{16.1.15}$$

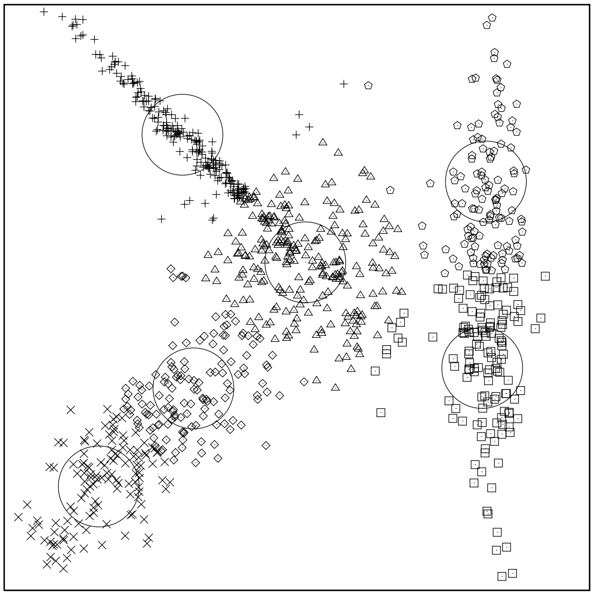


Figure 16.1.2. Output of k-means clustering as applied to the same data as Figure 16.1.1, with $K = 6$ components. The final assignments are shown as different plotted symbols. The centers of the large circles are the locations of the final means. (The radius of those circles is arbitrary, for visibility only.) Unlike Gaussian mixture modeling, k-means clustering can't split a point probabilistically between two components, so many points from the Gaussian at the upper left are mistakenly assigned to the central component. Also, k-means clustering needs more than one component to model a Gaussian with a large aspect ratio, because it clusters by radial distance.

is a point on the 1-sigma locus. Going around the unit circle in $\mathbf{z}$, and using the mapping (16.1.15), gives the desired ellipse. Put a constant 2 in front of the $\mathbf{L}$ in (16.1.15) for 2-sigma ellipses, and so forth.

We already remarked, in §7.4, on the closely related use of Cholesky decomposition to generate multivariate Gaussian deviates from a given covariance matrix.

16.1.2 K-Means Clustering

One interesting simplification of Gaussian mixture modeling has an independent history and is known as *k-means clustering*. We forget about the $\mathbf{\Sigma}_k$ covariances matrices completely, and we forget about probabilistic assignments of data points to components. Instead, each data point gets assigned to one (and only one) of the K components.

The E-step is simply: Assign each data point $\mathbf{x}_n$ to the component k whose mean $\boldsymbol{\mu}_k$ it is closest to, by Euclidean distance.

The M-step is simply: For all k, re-estimate the mean $\boldsymbol{\mu}_k$ as the average of data points $\mathbf{x}_n$ assigned to component k.

The convergence criterion is: Stop when an E-step doesn't change the assignment of any data point (in which case the M-step would also produce unchanged $\boldsymbol{\mu}_k$'s).

Interestingly, convergence is guaranteed — you can't get into an infinite loop of, say, shifting a point back and forth between two components. Despite its simplicity, k-means clustering can be quite useful: It is very fast, and it converges very rapidly. It can be used as a method to reduce a large number of data points to a much smaller number of "centers," which can then be used as starting points for more sophisticated methods.

For example, you might use k-means clustering to get starting values for a Gaussian mixture model that has difficulty converging to a good, global maximum. If K is the ultimate number of components that you want, you might use k-means to get down to $\sim 3 \times K$ components, then (repeatedly) randomly select K of these as starting guesses for the Gaussian model.

Be alert to the fact that k-means clustering has an intrinsically "spherical" view of the world, because of its Euclidean "nearest-to" assignments. If you have components that might have big aspect ratios, be sure to set K large enough so that these can be represented by several different centers. Figure 16.1.2 shows the same input data as Figure 16.1.1, now clustered by k-means. The Gaussians at the lower left and right have broken up into two centers each. The Gaussian at the upper left is only a single component, because it has had many of its points misclassified into the central component. (A Gaussian mixture model would have assigned those points probabilistically to both components.)

Code for k-means classification is similar to, but much shorter than, the previous code for a Gaussian mixture model:

kmeans.h

```
struct Kmeans {
Solve for a k-means clustering model from a set of data points and initial guesses of the means.
Output is a set of means and an assignment of each data point to one component.
    Int nn, mm, kk, nchg;
    MatDoub data, means;
    VecInt assign, count;
    Kmeans(MatDoub &ddata, MatDoub &mmeans) : nn(ddata.nrows()), mm(ddata.ncols()),
    kk(mmeans.nrows()), data(ddata), means(mmeans), assign(nn), count(kk) {
    Constructor. Arguments are the data points (as rows in a matrix), and initial guesses for
    the means (also as rows in a matrix).
        estep();                    Perform one initial E-step and M-step. User is re-
        mstep();                        sponsible for calling additional steps until con-
    }                                   vergence is obtained.
    Int estep() {
    Perform one E-step.
        Int k,m,n,kmin;
        Doub dmin,d;
        nchg = 0;
        for (k=0;k<kk;k++) count[k] = 0;
        for (n=0;n<nn;n++) {
            dmin = 9.99e99;
            for (k=0;k<kk;k++) {
                for (d=0.,m=0; m<mm; m++) d += SQR(data[n][m]-means[k][m]);
                if (d < dmin) {dmin = d; kmin = k;}
            }
            if (kmin != assign[n]) nchg++;
            assign[n] = kmin;
            count[kmin]++;
        }
        return nchg;
    }
    void mstep() {
    Perform one M-step.
        Int n,k,m;
        for (k=0;k<kk;k++) for (m=0;m<mm;m++) means[k][m] = 0.;
        for (n=0;n<nn;n++) for (m=0;m<mm;m++) means[assign[n]][m] += data[n][m];
        for (k=0;k<kk;k++) {
            if (count[k] > 0) for (m=0;m<mm;m++) means[k][m] /= count[k];
        }
    }
};
```

Incidentally, k-means clustering is not only a simplification of Gaussian mixture models; it is actually a limiting case. If the $\mathbf{\Sigma}_k$ matrices are all held fixed as

$$\mathbf{\Sigma}_k = \epsilon\, \mathbf{1} \qquad (16.1.16)$$

with ϵ infinitesimal and $\mathbf{1}$ the identity matrix, then the component k with mean closest to $\mathbf{x}_n$ will be assigned all of the responsibility p_{nk} for that n. The re-estimation of the $\boldsymbol{\mu}_k$'s then is identical to k-means clustering. The theorem that proves that the EM algorithm converges for Gaussian mixtures can easily be modified to prove the convergence of k-means clustering. (Basically, there is a hidden log-likelihood function that can be shown to increase at each step.)

Indeed, we can now explain the obscure constant `1.0e-10` in the initialization part of `Gaumixmod`: It is a value for ϵ that makes that routine's *first* E-step, M-step iteration be one of k-means clustering.

CITED REFERENCES AND FURTHER READING:

McLachlan, G. and Peel, D. 2000, *Finite Mixture Models* (New York: Wiley).

Moore, A.W. 2004, "Clustering with Gaussian Mixtures," at `http://www.cs.cmu.edu/~awm`.

Dempster, A.P., Laird, N.M., and Rubin, D.B. 1977, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society*, Series B, vol. 39, pp. 1-38. [The original paper on EM methods.]

Tanner, M.A. 2005, *Tools for Statistical Inference: Methods for the Exploration of Posterior Distributions and Likelihood Functions*, 3rd ed. (New York: Springer).

16.2 Viterbi Decoding

In this section we discuss models with discrete states, and how to use data to estimate what state a model is in, or what succession of states it traverses by allowed transitions. By *state*, we mean some discrete condition that can be characterized as a node on a directed graph like that in Figure 16.2.1. By *transition*, we mean moving along one of the directed edges of the graph. If you want to characterize a continuous variable in the context of this section, you need to define a set of discrete bins for its possible values, and make these the states.

The setup we describe is slightly more general than its close cousin, the directed graph of *stages and states* that defined the dynamic programming (DP) problem in §10.13. For some applications, the estimation problem of interest does live on a graph that has states and stages, exactly like DP; but for other applications, we need a general directed graph. We'll consider both types below.

Historically, problems involving the estimation of states have arisen in diverse, and often noncommunicating, fields. There are often multiple names for single concepts. (We saw this previously in the Bellman-Dijkstra-Viterbi algorithm for DP.) This history also makes it hard to give, in this section, a unified treatment with a single narrative. A more practical approach is to go through a couple of examples from different fields, and then, afterward, make some comparisons and give some advice.

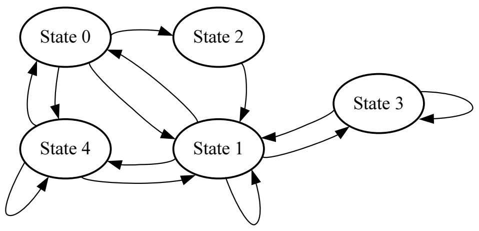


Figure 16.2.1. Graph of a dynamical system with five states. Allowed transitions are shown by arrows.

16.2.1 Error-Correcting Codes and Soft-Decision Decoding

An (N, K) *binary block code* is a list of 2^K binary *codewords*, each of length $N > K$ bits, designed to send a K-bit message in such a way that it can be received correctly even if one or more of the N bits arrive garbled (that is, 0 instead of 1, or vice versa). Two simple examples are shown below. In these particular cases, the message bits are the initial bits of the codeword, but that need not be true in general. Assigning any permutation of the codewords to message words is, effectively, the same code; likewise an arbitrary permutation of the bits in all the codewords (permuting bit-columns in the table).

(6,3) Shortened Hamming	
message	codeword
000	000000
001	001110
010	010101
011	011011
100	100011
101	101101
110	110110
111	111000

(7,4) Hamming	
message	codeword
0000	0000000
0001	0001011
0010	0010111
0011	0011100
0100	0100110
0101	0101101
0110	0110001
0111	0111010
1000	1000101
1001	1001110
1010	1010010
1011	1011001
1100	1100011
1101	1101000
1110	1110100
1111	1111111

Both of the codes shown have the property that their *Hamming distance* is 3. This means that all pairs of codewords differ in at least three bits. This is the property of the code that makes it "error correcting on one bit." If you receive a codeword with one of its bits wrong, then (i) it will not be in the above table, and (ii) there will be a unique codeword in the table that differs from it in one bit position. So, trying each bit position in turn, you can figure out what was the intended codeword.

A longer code, with a larger Hamming distance d, can be error correcting for more than one garbled bit, in fact for $(d-1)/2$ bits (rounding down). An (N, K) code can have d as large as $N - K$. However, trying all possible corrections until you find a valid codeword is a very poor decode strategy!

For so-called *linear codes* it is possible to construct a *parity-check matrix* $\mathbf{P}$ with the property that multiplying it by the vector of received bits (and doing all arithmetic modulo 2) gives a vector, the so-called *syndrome*, that is either all zeros (indicating that the received bits are ok) or else it uniquely corresponds to a mask (termed a *coset leader*) that tells which bits need correcting. So this error-correction algorithm, called *syndrome decoding*, can be summarized as:

- multiply the received bits by the parity-check matrix to get the syndrome,
- do a table lookup of the syndrome to get the coset leader, and
- XOR the coset leader with the received bits to get a valid codeword.

For example, the parity check matrix for the (7, 4) Hamming code, above, is

$$\mathbf{P} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{16.2.1}$$

(Convert codewords to column vectors by reading from left to right.) The lookup table relating syndrome to coset leader is

syndrome	coset leader
000	0000000
001	0000001
010	0000010
011	0001000
100	0000100
101	1000000
110	0100000
111	0010000

This particular code is called a *perfect code* because the number of syndromes exactly equals the number of coset leaders with one nonzero bit, plus 1 for the zero syndrome, a numerological coincidence. There are very few perfect codes, because there are very few sets of integers satisfying

$$1 + \binom{N}{1} + \cdots + \binom{N}{e} = 2^{N-K} \tag{16.2.2}$$

where e is the number of bits corrected. Probably the most nontrivial perfect code is the *Golay code*, with $N = 23$, $K = 12$, and $e = 3$. (Check out the numerology yourself.)

It's no big deal if a code is not perfect. It just means that there are some extra syndromes that correct *some* errors of more than e bits, but not enough to correct all such errors. You include these extra syndromes in the table, and run the algorithm exactly as already described. However, if a code is too far from perfect, you are wasting syndromes without gaining more bits of sure correction.

In practical applications, N and K are larger than these examples. For example, the lowest level of error correction on an audio compact disk (CD) is a $(28, 24)$ *Reed-Solomon* (RS) code, which can correct $e = 2$ bits. (On a CD, bits of the output codewords from many consecutive blocks are then interleaved and further protected by an RS(32,28) code.) Reed-Solomon codes are typically decoded by a more efficient process than syndrome decoding, using the so-called *Berlekamp-Massey* algorithm.

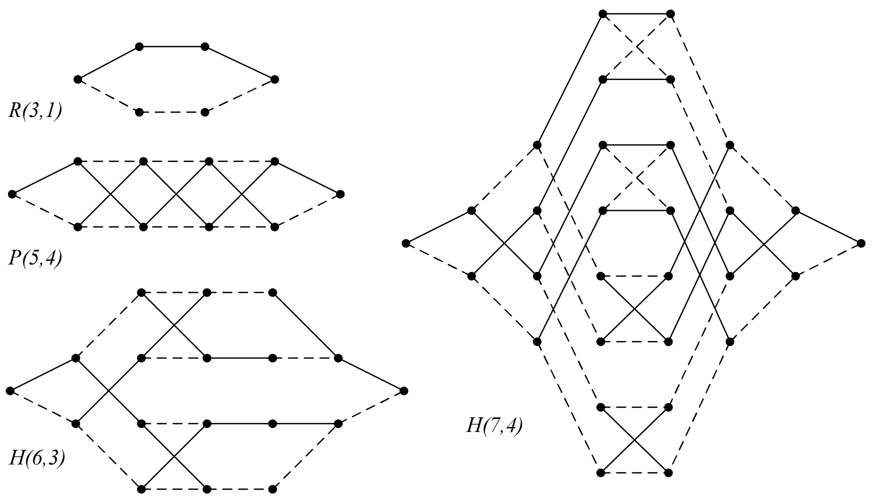


Figure 16.2.2. Trellises associated with four binary codes. The graph is traversed from left to right. A zero is output when any dotted edge is traversed, a one for a solid edge. Every path yields a valid codeword.

Now take a deep breath. Everything we have discussed thus far is what is called *hard-decision decoding* (HDD), meaning that hard decisions are made as to whether each incoming bit is a 1 or 0, with the error-correction algorithm acting on the resulting, possibly garbled, codeword. Virtually all coding theory utilized HDD until the early 1970s. Then came the giant leap forward with the recognition by multiple practitioners that Viterbi's 1967 decoding algorithm (an independent rediscovery of Bellman-Dijkstra, we might now say) could utilize "soft" data about each bit as easily as hard.

To understand *soft-decision decoding* (SDD), let us first note that every binary code can be represented by the kind of stage/state graph that we met in dynamic programming (§10.13), which, in the present context, is called a *trellis*. Figure 16.2.2 shows the trellises for the two codes given explicitly above, as well as for the schoolbook examples of a repetition code ("tell me three times") and a parity code. The latter, with $d = 1$, is error detecting, but not error correcting.

Although arrows are not shown, the trellis is traversed from left to right. Any such path on the trellis generates a valid codeword. A zero bit is emitted when a dotted edge is traversed, a one bit for a solid edge. You encode message bits by deciding whether to branch up or down, when you have a choice. Notice that you don't get such a choice at every stage: "Forced" edges generate precisely the extra codeword bits that the code's redundancy requires.

Although every code has a trellis, it is not so easy to find the *minimal trellis*, the one that has the fewest possible states at its maximum expansion. MacKay [1] gives a brief introduction; many additional references are in [3].

The first great idea behind soft-decision decoding is that we don't need to decide whether an incoming bit is a 0 or 1. Rather, we just need to assign a probability to each possibility (summing to unity, of course). For example, a bit's value may be determined by whether an instantaneous voltage is positive or negative — but the voltage measurement has some Gaussian spread of errors. If the voltage is many standard deviations positive, or negative, then the respective probabilities are very close to one or zero; but if the voltage is only $t = 0.5$ (say) standard deviations away from zero, we may want to assign a probability of 0.6915 to one more favored outcome, and 0.3085 to the less favored, since

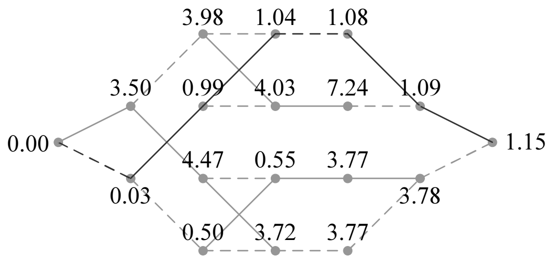
$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{0.5} e^{-z^2/2} dz \approx 0.6915 \qquad (16.2.3)$$

(By the end of this section we will be more sophisticated about the notion of assigning probabilities to transitions.)

The second great idea is that the problem of finding the maximum likelihood path through a trellis — that is, the path with the maximum product of the probabilities at each stage — is just a dynamic programming problem, where the cost of traversing an edge whose probability is p is taken as $-\log(p)$, a positive number, since $0 \leq p \leq 1$. The minimum cost path, with this metric, is the maximum likelihood path. In each stage, all the 0 edges (dotted lines in the figure) get one probability, and all the 1 edges (solid lines) get its complement. The edge probabilities can, and in general will, vary with each bit received (that is, from stage to stage), since the noise and path loss can vary with time.

Take these two ideas together and you have *soft-decision decoding using the Viterbi algorithm*.

The following tableau decodes one codeword for the shortened Hamming (6, 3) code given above. In this example, five of the six bits are received fairly unambiguously, while one bit (the second) is seen to be rather ambiguous. Nevertheless, the algorithm treats all bits equally "softly." Reviewing §10.13 as necessary, you should be able to see where all the numbers in the tableau come from, as well as how the darker path (which is the final "hard" decision for the codeword 011011) is obtained by backtracking. Given the appropriate cost function, the routine `dynpro` in §10.13 does exactly this calculation.



Prob(0) =	.97	.62	.04	.96	.01	.06
Prob(1) =	.03	.38	.96	.04	.99	.94
–log[Prob(0)] =	0.03	0.47	3.22	0.04	4.61	2.81
–log[Prob(1)] =	3.50	0.96	0.05	3.21	0.01	0.06

You might have the "bright idea" of converting the final minimum path length, 1.15 in the above example, into a probability by taking its negative exponential. The result is 0.3166. Does this mean that you'll get the right codeword only 31% of the time? No! Go stand in the corner! You are confusing likelihood with probability. The likelihoods of all eight codewords (not found by the DP algorithm, but computed exhaustively) are, for this example,

codeword	likelihood
000000	0.000014
001110	0.001372
010101	0.000006
011011	0.316126
100011	0.000665
101101	0.000007
110110	0.000001
111000	0.000006

You can see that the likelihoods don't add up to 1, and that the favored path wins over any competitor by a large factor. (Bayesians: We know that this paragraph is making you break out in a rash. We are on your side, and will have more to say about this in §16.3.4.)

In the above example, the second bit was merely ambiguous. What if it had instead been really wrong, indicating, say, 0.99 probability of having a value zero? No problem. Since the underlying code is one-bit error correcting, the DP algorithm will readily decide to traverse the unlikely single edge, since the alternative would be to traverse two or more unlikely edges on other bits. However, if we made the second bit incorrect with probability 0.999999, the algorithm would "correct" two other bits instead, which, under the circumstances, would be the best decision.

You can see that it is not meaningful to say exactly how many bits e a soft-decision decode algorithm can correct. It just makes the best choice determined by the probabilities. As another example, we might consider the simple parity code shown in Figure 16.2.2. With a hard-decision decode, parity does not give enough information to correct a single bit. With a soft-decision algorithm, however, the parity bit can cast the deciding vote if some other bit is wavering too close to an ambiguous 50% probability level.

Soft-decision decoding algorithms are available for essentially all codes in use today, including Reed-Solomon codes and the important *turbo codes* [2] that are beyond our scope. Some important applications (e.g., *trellis coded modulation*), use short trellises whose end states loop around to become identified with their start states. The Viterbi algorithm is applied to long sequences of input symbols that loop through the trellis many times. In trellis coded modulation, the symbols being (softly) decoded are not single bits, but locations in the complex phase plane that comprise a carefully chosen *constellation* centered at the origin (for example, a hexagonal lattice).

CITED REFERENCES AND FURTHER READING:

Lin, S. and Costello, D.J. 2004, *Error Control Coding*, 2nd ed. (Upper Saddle River, NJ: Pearson-Prentice Hall).

Blahut, R.E. 2002, *Algebraic Codes for Data Transmission* (Cambridge, UK: Cambridge University Press).

MacKay, D.J.C. 2003, *Information Theory, Inference, and Learning Algorithms* (Cambridge, UK: Cambridge University Press).[1]

Schlegel, C. and Perez, L. 2000, *Trellis and Turbo Coding*, (Piscataway, NJ: IEEE Press).[2]

"Special Issue on Codes and Complexity", 1996, *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 1649-2064.[3]

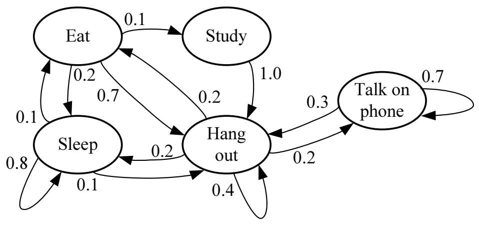


Figure 16.3.1. Example of a Markov model. Transitions occur between states along the directed edges shown. Each outgoing edge is labeled by its probability. The sum of the probabilities on the outgoing edges from each state is 1. This model is called "Teen Life."

16.3 Markov Models and Hidden Markov Modeling

Trellises, like those in §16.2, are directed graphs without any loops, so a path that begins at the leftmost node inevitably ends, after a finite number of stages, at the rightmost node. The more general *Markov model* lives on a graph that can have loops (as in Figure 16.2.1), so some paths can continue indefinitely. Indeed, as a convention, one can add a self-loop (a directed edge connecting a state to itself) to any state that would otherwise have "no way out." Then, all paths can be continued indefinitely, even those whose fate is to remain stuck in a single state.

To turn such a directed graph into a Markov model (also known as a *Markov chain* or *first-order Markov process*), we simply label all of its edges with *transition probabilities*, such that the sum of probabilities over the outgoing edges from each node is 1. Figure 16.3.1 shows an example, a Markov model with five states, that we call "Teen Life."

A single realization of a Markov model is a random path that moves from state to state according to the model's probabilities. These are conveniently organized into a *transition matrix* $\mathbf{A}$ whose element A_{ij} is the probability associated with an $i \to j$ transition, that is, the probability of moving to state j, given state i as the starting point. A valid transition matrix satisfies

$$0 \le A_{ij} \le 1 \qquad \text{and} \qquad \sum_j A_{ij} = 1 \tag{16.3.1}$$

The transition matrix for Teen Life (Figure 16.3.1) is

$$\mathbf{A} = \begin{pmatrix} 0 & 0.7 & 0.1 & 0 & 0.2 \\ 0.2 & 0.4 & 0 & 0.2 & 0.2 \\ 0 & 1.0 & 0 & 0 & 0 \\ 0 & 0.3 & 0 & 0.7 & 0 \\ 0.1 & 0.1 & 0 & 0 & 0.8 \end{pmatrix} \tag{16.3.2}$$

where the states are numbered in the order (Eat, Hang, Study, Talk, and Sleep).

A routine for generating a realization of a Markov model from its $M \times M$ transition matrix, using the `Ran` structure of §7.1 to get random numbers, is straightforward.

markovgen.h

```
void markovgen(const MatDoub_I &atrans, VecInt_O &out, Int istart=0,
    Int seed=1) {
```
Generate a realization of an M-state Markov model, given its $M \times M$ transition matrix atrans. The vector out is filled with integers in the range $0 \ldots M-1$. The starting state is the optional argument istart (defaults to 0). seed is an optional argument that sets the seed of the random number generator.
```
    Int i, ilo, ihi, ii, j, m = atrans.nrows(), n = out.size();
    MatDoub cum(atrans);                Temporary matrix to hold cumulative probabilities.
    Doub r;
    Ran ran(seed);                      Use the random number generator Ran.
    if (m != atrans.ncols()) throw("transition matrix must be square");
    for (i=0; i<m; i++) {                  Fill cum and die if clearly not a transition matrix.
        for (j=1; j<m; j++) cum[i][j] += cum[i][j-1];
        if (abs(cum[i][m-1]-1.) > 0.01)
            throw("transition matrix rows must sum to 1");
    }
    j = istart;                            The current state is kept in j.
    out[0] = j;
    for (ii=1; ii<n; ii++) {               Main loop.
        r = ran.doub()/cum[j][m-1];        Slightly-off normalization gets corrected here.
        ilo = 0;
        ihi = m;
        while (ihi-ilo > 1) {              Use bisection to find location among the cumu-
            i = (ihi+ilo) >> 1;                lative probabilities.
            if (r>cum[j][i-1]) ilo = i;
            else ihi = i;
        }
        out[ii] = j = ilo;                 Set new current state.
    }
}
```

What makes the transition matrix a matrix, and not just a table of probabilities, is its connection to *ensembles* of realizations of the corresponding Markov model. An ensemble can be characterized by the components of a *population vector* $\mathbf{s}_t$ whose components give the number of models in each state at time t. (Here and below, we use t as an integer, discrete time variable. On a trellis it would be called a *stage* instead of a *time*.) For Teen Life, we can give names to the components of $\mathbf{s}_t$ corresponding to the states: (E, H, S, T, Z).

If all the models in the ensemble are evolved by one timestep (one transition), then the population vector $\mathbf{s}_t$ turns into $\mathbf{s}_{t+1}$ by the matrix multiplication

$$\mathbf{s}_{t+1} = \mathbf{A}^T\, \mathbf{s}_t \tag{16.3.3}$$

The transpose operation is needed only because two common conventions are at odds: The time order $i \to j$ for A_{ij} versus the left matrix multiplication of a column vector (whose implicit time ordering is "from" the second index "to" the first). Given a matrix, you can easily tell whether it is intended to be an $\mathbf{A}$ or an $\mathbf{A}^T$, by whether, respectively, its rows or columns sum to unit probability.

Note that we can evolve more than one step at a time, by precomputing powers of $\mathbf{A}^T$. So, $\mathbf{s}_{t+n} = (\mathbf{A}^T)^n\, \mathbf{s}_t$, for example.

Every Markov model has at least one equilibrium distribution of states that remains unaffected when multiplied by $\mathbf{A}^T$. To prove this, we write

$$\mathbf{A}^T\, \mathbf{s}_e = \mathbf{s}_e \qquad \Longleftrightarrow \qquad (\mathbf{A}^T - \mathbf{1})\, \mathbf{s}_e = \mathbf{0} \tag{16.3.4}$$

where $\mathbf{s}_e$ is the sought-after equilibrium state. Equation (16.3.4) can hold if and only if $\mathbf{A}^T - \mathbf{1}$ is a singular matrix. Since the columns of $\mathbf{A}^T$ all sum to 1, the columns of $\mathbf{A}^T - \mathbf{1}$ all sum to zero, and hence are linearly dependent, q.e.d. Equivalently, we have proved that $\mathbf{A}^T$ has at least one eigenvalue equal to 1. The corresponding eigenvector is an equilibrium distribution of states. If there is only one eigenvalue equal to 1, the equilibrium is unique. For the Teen Life model, there is one eigenvalue of 1, and the corresponding eigenvector (normalized so that its components sum to unity) is approximately $(0.099, 0.297, 0.001, 0.198, 0.395)$. (This teenager spends about 39.5% of his/her time sleeping, 19.8% talking on the phone, 0.1% studying, and so forth.)

Do almost all starting distributions converge to a unique equilibrium, in which case the model is said to be *ergodic*? Not necessarily. Two things can go wrong. First, if there is more than one eigenvalue equal to 1, a model will converge to some different linear combination of the corresponding eigenvectors for different starting distributions. Such models are said to fail the test of *irreducibility*. Second, the model may have a periodic limit cycle, so that, for most starting distributions, it doesn't converge at all. Such models are said to fail the test of *aperiodicity*. The theorem (and vocabulary test) is: Irreducibility and aperiodicity imply ergodicity.

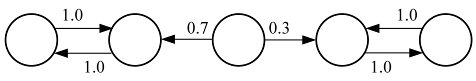
One way to diagnose these conditions is to perform successive squarings of the matrix $\mathbf{A}^T$ to take it to a very high power, say 2^{32}. This requires $O(32\,M^3)$ operations for a model with M states. (While there are more sophisticated methods, none scale better than M^3.) If all M columns in the result are converging to identical vectors, then there is just one eigenvalue (unity), and all starting distributions will converge to its eigenvector (which is in fact the repeated column vector). The model is then ergodic.

Otherwise, locate any rows in the power matrix that are zero, and cross out those rows and their corresponding columns. (These are states that become permanently unpopulated as the model evolves.) Then, check to see if the remaining columns are all eigenvectors with unit eigenvalue. You can do the test by multiplying each such column by the original $\mathbf{A}^T$. If all columns pass the test, then there are multiple equilibria, but all starting distributions will converge to some combination of them. If any column fails the test, then the model has a periodic limit cycle. There are still equilibria, given by the eigenvectors of unit eigenvalue, but a starting state must be very special to evolve to one. Indeed, such states are a set of measure zero, and we can say that the equilibria are *unstable*.

A simple example with multiple equilibria and periodic limit cycles is the transition matrix

$$\mathbf{A}' = \begin{pmatrix} 0 & 1.0 & 0 & 0 & 0 \\ 1.0 & 0 & 0 & 0 & 0 \\ 0 & 0.7 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0 & 1.0 \\ 0 & 0 & 0 & 1.0 & 0 \end{pmatrix} \tag{16.3.5}$$

corresponding to the graph



$\mathbf{A}'^T$ has two eigenvectors with unit eigenvalue (you can guess them from the graph): $(0.5, 0.5, 0, 0, 0)$ and $(0, 0, 0, 0.5, 0.5)$. However, $\mathbf{A}'^T$ to the power 2^{32} does not have these as any of its columns, but is rather

$$(\mathbf{A}'^T)^{2^{32}} = \begin{pmatrix} 1.0 & 0 & 0.7 & 0 & 0 \\ 0 & 1.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0.3 & 0 & 1.0 \end{pmatrix} \tag{16.3.6}$$

thus showing that the model has only unstable equilibria. (The little identity matrix blocks are merely artifacts of the limit cycles having period 2, while 2^{32} is even. In general, you will get some other pattern.)

Successive squaring is a poor way to get the equilibrium distribution for a model that is known (or guessed) to have a single stable equilibrium. A better way, since we already know the eigenvalue, is inverse iteration. Just solve the equation

$$(\mathbf{A}^T - \mathbf{1})\,\mathbf{s}_e = \mathbf{b} \tag{16.3.7}$$

by LU decomposition (§2.3), with the right-hand vector $\mathbf{b} = (1, 1, \dots, 1)$. If your solver complains about the zero pivot instead of substituting a small value for it (which is what we want for this application), then use the matrix $(\mathbf{A}^T - 0.999999 \times \mathbf{1})$ instead. In either case, you will want to renormalize $\mathbf{s}_e$, to make, e.g., its components have unit sum.

You can test for multiple equilibria by perturbing the right-hand side vector and seeing if you get the same $\mathbf{s}_e$. If you do have multiple equilibria, it is probably time to turn to the methods of Chapter 11 and calculate $\mathbf{A}^T$'s eigenvalues and eigenvectors directly.

That is all (in fact, more) than we want to tell about Markov models in general. We turn now to the real business at hand, which is to estimate statistically the state of a "hidden" Markov model, given only partial or imperfect information.

16.3.1 Hidden Markov Models

In a *hidden Markov model* (HMM), we don't get to observe the state of the model directly. Rather, whenever it is in any state i (one of M states), it emits a *symbol* k, chosen probabilistically from a set of K symbols. The probability of emitting symbol number k from state number i is denoted by

$$b_i(k) \equiv P(\text{symbol } k \mid \text{state } i) \qquad (0 \le i < M, \quad 0 \le k < K) \tag{16.3.8}$$

with the normalization condition

$$\sum_{k=0}^{K-1} b_i(k) = 1 \qquad (0 \le i < M) \tag{16.3.9}$$

Thus, when the model evolves through N timesteps, the *hidden states* are a vector of integers,

$$\mathbf{s} = \{s_t\} = (s_0, s_1, \dots s_{N-1}) \tag{16.3.10}$$

each in the range $0 \le s_i < M$, while the *observations* or *data* are a vector of integers,

$$\mathbf{y} = \{y_t\} = (y_0, y_1, \dots y_{N-1}) \tag{16.3.11}$$

each in the range $0 \le y_i < K$.

For the Teen Life example, here is a table of symbols and their probabilities of being emitted from each state, in response to the repeated parental query, "What are you doing?":

			$i = 0$	1	2	3	4
k	symbol	meaning	Eat	Hang	Study	Talk	Sleep
0	o	[silence]	0.2	0.2	0	0.3	0.5
1	s	"I'm studying!"	0	0	1.0	0.2	0
2	b	"I'm busy!"	0	0.6	0	0.4	0
3	g	[grunt]	0.8	0.2	0	0.1	0
4	z	[snore]	0	0	0	0	0.5

The key point is that the emitted symbols give only incomplete, or garbled, state information (e.g., the claim of studying when actually talking on the phone). A state can emit more than one possible symbol, and a symbol can be emitted by more than one possible state. Nevertheless, our goal is to make the best statistical reconstruction of the vector $\mathbf{s}$ from $\mathbf{y}$.

More specifically, at each time t we want to estimate

$$P_t(i) \equiv P(s_t = i \mid \mathbf{y}) \tag{16.3.12}$$

the probability that the actual state of the system is i at time t, given all the data. (If the word "probability" in this context bothers you, you may think of it as a likelihood.)

Let $\alpha_t(i)$ be defined for $t = 0 \dots N-1$ and $i = 0 \dots M-1$ as the probability of the observed data up to time t (that is, $y_0 \dots y_t$), *given* that we are in state i at time t. Since we are not specifying any of the previous states, we must sum over all possible paths that get to state i at time t. Thus,

$$\begin{aligned}
\alpha_0(i) &= b_i(y_0)\\
\alpha_t(i) &= \sum_{i_0,i_i,\dots,i_{t-1}} b_{i_0}(y_0)\; A_{i_0 i_1} b_{i_1}(y_1)\; \dots\; A_{i_{t-1} i} b_i(y_t) \qquad (1 \le t < N)
\end{aligned} \tag{16.3.13}$$

In other words, each transition contributes to the product both a transition probability and a symbol probability, and we sum over all possible combinations of previous states, that is, all possible values of $i_0, i_1, \dots, i_{t-1}$, each in the range $0 \dots M-1$.

Since $\alpha_t(i)$ is the probability of data given state, it can also be interpreted as the likelihood of the state, given the data; or, if we are Bayesians (and we are!), as the unnormalized posterior probability of being in state i, which can be normalized simply by dividing by $\sum_i \alpha_t(i)$. However, equation (16.3.13) seems useless for a big and a little reason. Big: It has exponentially many terms to evaluate. Little: it uses only part of the data (the data earlier in time) to estimate the state i at time t. It is what is called a *forward estimate*.

Amazingly, both problems are easy to fix. It is not hard to see that the $\alpha_t(i)$'s satisfy a recurrence relation that advances them all one step in t:

$$\alpha_{t+1}(j) = \sum_{i=0}^{M-1} \alpha_t(i) A_{ij} b_j(y_{t+1}) \qquad (0 \le j < M) \quad \text{for} \quad t = 0, \ldots, N-2 \tag{16.3.14}$$

One step of this recurrence takes only $O(M^2)$ operations, so the whole table of $\alpha_t(i)$'s can be computed in $O(NM^2)$.

To fix the second problem, we come at the issue from "the other end of time." Let $\beta_t(i)$ be defined for $t = N-1 \ldots 0$ and $i = 0 \ldots M-1$ as the probability of the *future* observed data $(y_{t+1}, y_{t+2}, \ldots y_{N-1})$ again *given* that we are in state i at time t. Analogously to the α's, we have

$$\beta_t(i) = \sum_{i_{t+1},\ldots,i_{N-1}} A_{i i_{t+1}} b_{i_{t+1}}(y_{t+1}) \ \ldots \ A_{i_{N-2} i_{N-1}} b_{i_{N-1}}(y_{N-1}) \tag{16.3.15}$$

with the special case $\beta_{N-1}(i) = 1$. (Because there are no data to the future of $t = N-1$, those data's probability is by definition 1.) In the formula for the $\beta_t(i)$'s there is a factor in the product for each future transition probability and each future symbol probability (fixing the symbols to be the actual y's of course). Just as for the α's, the β's can be interpreted as likelihoods, or unnormalized posterior probabilities. And, wonderfully, equation (16.3.15) can be solved by a *backward* recurrence,

$$\beta_{t-1}(i) = \sum_{j=0}^{M-1} A_{ij} b_j(y_t) \beta_t(j) \qquad (0 \le i < M) \quad \text{for} \quad t = N-1, \ldots, 1 \tag{16.3.16}$$

Calculating all the β's for $t = N-1, N-2, \ldots, 0$ is called a *backward estimate*.

Now here is the big payoff: From the definitions of the α's and β's, the product $\alpha_t(i)\beta_t(i)$ is the unnormalized posterior probability of state i at time t given *all* the data. If we normalize it by dividing by

$$\mathcal{L}_t = \sum_{i=0}^{M-1} \alpha_t(i) \beta_t(i) \tag{16.3.17}$$

we get the desired estimate of the probability of each separate state at each separate time,

$$P_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\mathcal{L}_t} \tag{16.3.18}$$

Further, it follows from the definitions (16.3.13) and (16.3.15) that $\mathcal{L}_t$ is actually independent of t, so we can omit the subscript t and calculate it only once. (In practice, one often renormalizes at each time t for greater numerical stability, however.) The value of $\mathcal{L}$ can be interpreted as the probability (or likelihood) of the whole data set, given the parameters of the model.

Equations (16.3.14) and (16.3.16), taken together, are called the *forward-backward algorithm* for state estimation in a hidden Markov model.

Translating HMM into code, we start with a structure that will hold the various quantities that come into play, and its constructor. You construct an `HMM` structure by specifying a transition probability matrix $\mathbf{A}$ (N.B.: not $\mathbf{A}^T$), a symbol probability matrix $b_{ik} \equiv b_i(k)$, and a vector of observed data $\mathbf{y}$.

hmm.h

```
struct HMM {
Structure for a hidden Markov model and its methods.
    MatDoub a, b;                          Transition matrix and symbol probability matrix.
    VecInt obs;                            Observed data.
    Int fbdone;
    Int mstat, nobs, ksym;                 Number of states, observations, and symbols.
    Int lrnrm;
    MatDoub alpha, beta, pstate;           Matrices α, β, and P_i(t).
    VecInt arnrm, brnrm;
    Doub BIG, BIGI, lhood;
    HMM(MatDoub_I &aa, MatDoub_I &bb, VecInt_I &obs);     Constructor; see below.
    void forwardbackward();                HMM state estimation.
    void baumwelch();                      HMM parameter re-estimation.
    Doub loglikelihood() {return log(lhood)+lrnrm*log(BIGI);}
    Returns the log-likelihood computed by forwardbackward().
};

HMM::HMM(MatDoub_I &aa, MatDoub_I &bb, VecInt_I &obss) :
    a(aa), b(bb), obs(obss), fbdone(0),
    mstat(a.nrows()), nobs(obs.size()), ksym(b.ncols()),
    alpha(nobs,mstat), beta(nobs,mstat), pstate(nobs,mstat),
    arnrm(nobs), brnrm(nobs), BIG(1.e20), BIGI(1./BIG)  {
    Constructor. Input are the transition matrix aa, the symbol probability matrix bb, and the
    observed vector of symbols obss. Local copies are made, so the input quantities need not
    be preserved by the calling program.
    Int i,j,k;
    Doub sum;
    Although space constraints make us generally stingy about printing code for checking input,
    we will save you a lot of grief by doing so in this case. If you get "matrix not normalized"
    errors, you probably have your matrix transposed. Note that normalization errors <1% are
    silently fixed.
    if (a.ncols() != mstat) throw("transition matrix not square");
    if (b.nrows() != mstat) throw("symbol prob matrix wrong size");
    for (i=0; i<nobs; i++) {
        if (obs[i] < 0 || obs[i] >= ksym) throw("bad data in obs");
    }
    for (i=0; i<mstat; i++) {
        sum = 0.;
        for (j=0; j<mstat; j++) sum += a[i][j];
        if (abs(sum - 1.) > 0.01) throw("transition matrix not normalized");
        for (j=0; j<mstat; j++) a[i][j] /= sum;
    }
    for (i=0; i<mstat; i++) {
        sum = 0.;
        for (k=0; k<ksym; k++) sum += b[i][k];
        if (abs(sum - 1.) > 0.01) throw("symbol prob matrix not normalized");
        for (k=0; k<ksym; k++) b[i][k] /= sum;
    }
}
```

Now, to actually do the forward-backward estimation, you call the function forwardbackward. This fills the matrix pstate, so that $\text{pstate}_{ti} = P_t(i)$. It also sets the internal variables lhood and lrnrm so that the function loglikelihood returns the logarithm of $\mathcal{L}$. Don't be surprised at how large in magnitude this (negative) number can be. The probability of any *particular* data set of more than trivial length is astronomically small!

In the following code, the quantities BIG, BIGI, arnrm, brnrm, and lrnrm all relate to dealing with values that would far underflow an ordinary floating format. The basic idea is to renormalize as necessary, keeping track of the accumulated number of renormalizations. At the end, when an α, a β, and an $\mathcal{L}$ are combined, probability values of reasonable magnitude result.

hmm.h

```
void HMM::forwardbackward() {
```
HMM forward-backward algorithm. Using the stored a, b, and obs matrices, the matrices alpha, beta, and pstate are calculated. The latter is the state estimation of the model, given the data.
```
    Int i,j,t;
    Doub sum,asum,bsum;
    for (i=0; i<mstat; i++) alpha[0][i] = b[i][obs[0]];
    arnrm[0] = 0;
    for (t=1; t<nobs; t++) {                    Forward pass.
        asum = 0;
        for (j=0; j<mstat; j++) {
            sum = 0.;
            for (i=0; i<mstat; i++) sum += alpha[t-1][i]*a[i][j]*b[j][obs[t]];
            alpha[t][j] = sum;
            asum += sum;
        }
        arnrm[t] = arnrm[t-1];                  Renormalize the α's as necessary to avoid under-
        if (asum < BIGI) {                          flow, keeping track of how many renormal-
            ++arnrm[t];                             izations for each α.
            for (j=0; j<mstat; j++) alpha[t][j] *= BIG;
        }
    }
    for (i=0; i<mstat; i++) beta[nobs-1][i] = 1.;
    brnrm[nobs-1] = 0;
    for (t=nobs-2; t>=0; t--) {                 Backward pass.
        bsum = 0.;
        for (i=0; i<mstat; i++) {
            sum = 0.;
            for (j=0; j<mstat; j++) sum += a[i][j]*b[j][obs[t+1]]*beta[t+1][j];
            beta[t][i] = sum;
            bsum += sum;
        }
        brnrm[t] = brnrm[t+1];
        if (bsum < BIGI) {                      Similarly, renormalize the β's as necessary.
            ++brnrm[t];
            for (j=0; j<mstat; j++) beta[t][j] *= BIG;
        }
    }
    lhood = 0.;                                 Overall likelihood is lhood with lnorm renormal-
    for (i=0; i<mstat; i++) lhood += alpha[0][i]*beta[0][i];        izations.
    lrnrm = arnrm[0] + brnrm[0];
    while (lhood < BIGI) {lhood *= BIG; lrnrm++;}
    for (t=0; t<nobs; t++) {                    Get state probabilities from α's and β's.
        sum = 0.;
        for (i=0; i<mstat; i++) sum += (pstate[t][i] = alpha[t][i]*beta[t][i]);
```
The next line is an equivalent calculation of sum. But we'd rather have the normalization of the $P_i(t)$'s be more immune to roundoff error. Hence we do the above sum for each value of t.
```
        // sum = lhood*pow(BIGI, lrnrm - arnrm[t] - brnrm[t]);
        for (i=0; i<mstat; i++) pstate[t][i] /= sum;
    }
    fbdone = 1;                                 Flag prevents misuse of baumwelch(), later.
}
```

You may be wondering how well forwardbackward is able to do at predicting the hidden states of Teen Life, given just a long string of output symbols. If we take the prediction to be the state with the highest probability at each time, then this is correct about 78% of the time. Another 17% of the time, the correct state has the second-highest probability, often when the top two probabilities are nearly equal. It is an important property of HMMs that the output is not only a prediction, but also a quantitative assessment of how “sure” the model is of that prediction.

16.3.2 Some Variations on HMM

HMM state estimation with the forward-backward algorithm is a very flexible formalism, and many variants are possible. For example, in decoding codes on a trellis, as we did above, the symbols 0 or 1 are emitted not by the states, but by the transitions between the states. If we want to use HMM for that problem (we will say more about this below), we must replace $b_i(k)$ with $b_{ij}(k)$, the probability of emitting symbol k in a transition from state i to state j. The forward and backward recurrences now become

$$\begin{aligned}\alpha_{t+1}(j) &= \sum_{i=0}^{M-1} \alpha_t(i) A_{ij} b_{ij}(y_{t+1}) \\ \beta_{t-1}(i) &= \sum_{j=0}^{M-1} A_{ij} b_{ij}(y_t) \beta_t(j)\end{aligned} \tag{16.3.19}$$

and we start off the α's with the special rule $\alpha_0(i) = 1$, since (like the case of $\beta_{N-1}(i)$ previously) the probability of the data is 1 before there are any data.

Another variant case is where one or more intermediate states are known exactly. In that case, one or more of the sums over $i_0, i_1, \ldots, i_{t-1}$ in equation (16.3.13) is left out, and the corresponding index on an A and b gets replaced by the known state number. If you track through how this affects the recurrence equation (16.3.14), you'll see that the new procedure is

- calculate the α's forward to, and including, a known state;
- zero all the α values at that time except for that of the known state;
- don't renormalize anything (though you feel tempted to do so); and
- continue forward with the α's for the next timestep.

Proceed similarly for the β's.

The opposite variant is where you have *missing data*, meaning that for some values of t there is no observation of the symbol $\mathbf{y}_t$. In this case, all you need to do is to make a special case for the symbol probability,

$$b_i(y_t) \equiv 1, \qquad (0 \le i < M) \qquad t \in \{\text{missing}\} \tag{16.3.20}$$

meaning that, regardless of state i, the probability of observing the data (meaning no data) at time t is unity. Now proceed as usual to calculate the state probabilities. If you then want to *reconstruct* the missing data, you can calculate its posterior probabilities,

$$P(y_t = k \mid \mathbf{y}) = \sum_{i=0}^{M-1} P_i(t) b_i(k) = \sum_{i=0}^{M-1} \frac{\alpha_i(t)\beta_i(t)}{\mathcal{L}} b_i(k) \qquad t \in \{\text{missing}\} \tag{16.3.21}$$

16.3.3 Bayesian Re-Estimation of the Model Parameters

This is magical. The probability that we were in state i at time t is $\alpha_t(i)\beta_t(i)/\mathcal{L}$. What is the probability, given the data $\mathbf{y}$, that a given transition, say between time

t and time $t+1$, was a transition between state i and state j? We write, applying various of the laws of probability,

$$\begin{aligned}
P(s_t = i, s_{t+1} = j \mid \mathbf{y})
&= P(s_{t+1} = j \mid \mathbf{y},\, s_t = i)P(s_t = i \mid \mathbf{y}) \\
&= \frac{P(\mathbf{y} \mid s_{t+1} = j,\, s_t = i)P(s_{t+1} = j \mid s_t = i)}{\sum_j P(\mathbf{y} \mid s_{t+1} = j,\, s_t = i)P(s_{t+1} = j \mid s_t = i)} P(s_t = i \mid \mathbf{y}) \\
&= \frac{[\alpha_t(i) b_j(y_{t+1}) \beta_{t+1}(j)][A_{ij}]}{\sum_j [\alpha_t(i) b_j(y_{t+1}) \beta_{t+1}(j)][A_{ij}]} \frac{[\alpha_t(i)\beta_t(i)]}{\mathcal{L}} \\
&= \frac{\alpha_t(i) A_{ij} b_j(y_{t+1}) \beta_{t+1}(j)}{\mathcal{L}}
\end{aligned} \tag{16.3.22}$$

Note how the sum over j in the denominator disappears by the recurrence (16.3.16) for $\beta_t(i)$.

So, for a long run of data, we can compute the fraction of the time that a state i transitions to state j as the estimated number of $i \rightarrow j$ transitions divided by the estimated number of i states,

$$\widehat{A}_{ij} = \frac{\sum_t \alpha_t(i) A_{ij} b_j(y_{t+1}) \beta_{t+1}(j)}{\sum_t \alpha_t(i)\beta_t(i)} \tag{16.3.23}$$

noting that the $\mathcal{L}$'s cancel out. The reason for calling this quotient $\widehat{A}_{ij}$ is that it is a *re-estimation* of the transition probability A_{ij}. The corresponding re-estimation of the symbol probability matrix $b_i(k)$ is the fraction of all i states that emit a symbol k, namely

$$\widehat{b}_i(k) = \frac{\sum_t \delta(y_t, k)\alpha_t(i)\beta_t(i)}{\sum_t \alpha_t(i)\beta_t(i)} \tag{16.3.24}$$

where $\delta(j,k)$ is 1 if $j = k$, zero otherwise.

You might think that this process is somehow circular, or that re-estimating A_{ij} and $b_i(k)$ in this fashion only introduces noise that degrades the model. Far from it! Baum and Welch first showed that replacing A_{ij} by $\widehat{A}_{ij}$ and $b_j(k)$ by $\widehat{b}_j(k)$, and then recalculating the probabilities of each state at each time by the forward-backward algorithm, always *increases* $\mathcal{L}$, the overall likelihood of the model. It is, in fact, an EM algorithm (cf. §16.1, and see below). You can continue this cycle of estimating states (forward-backward) and re-estimating model probabilities (Baum-Welch), obtaining further increases in $\mathcal{L}$, until convergence to a maximum is achieved. Equations (16.3.23) and (16.3.24) are known as *Baum-Welch re-estimation*.

So the magic is this: We began by estimating states in a *known* hidden Markov model. We now see that, *just from the data*, we can get not only an estimate of the states, but also an estimate of the model itself, that is, the transition probabilities and symbol probabilities. Like any iterative process, this works best if we have a good initial guess. But it will often converge to a good model from a fairly random initial guess. (You should not start with exactly uniform probabilities, because that creates a symmetry that the iteration finds hard to break.)

The code is straightforward. The updating of $b_i(k)$ comes almost for free as a byproduct of computing the denominator in the update for A_{ij}. Like the forward-backward algorithm, Baum-Welch re-estimation takes $O(NM^2)$ operations.

hmm.h

```
void HMM::baumwelch() {
```
Baum-Welch re-estimation of the stored matrices a and b, using the data obs and the matrices alpha and beta as computed by forwardbackward() (which must be called first). The previous values of a and b are overwritten.
```
    Int i,j,k,t;
    Doub num,denom,term;
    MatDoub bnew(mstat,ksym);
    Doub powtab[10];                    Fill table of powers of BIGI.
    for (i=0; i<10; i++) powtab[i] = pow(BIGI,i-6);
    if (fbdone != 1) throw("must do forwardbackward first");
    for (i=0; i<mstat; i++) {          Looping over i, get denominators and new b.
        denom = 0.;
        for (k=0; k<ksym; k++) bnew[i][k] = 0.;
        for (t=0; t<nobs-1; t++) {
            term = (alpha[t][i]*beta[t][i]/lhood)
                * powtab[arnrm[t] + brnrm[t] - lrnrm + 6];
            denom += term;
            bnew[i][obs[t]] += term;
        }
        for (j=0; j<mstat; j++) {      Inner loop over j gets elements of a.
            num = 0.;
            for (t=0; t<nobs-1; t++) {
                num += alpha[t][i]*b[j][obs[t+1]]*beta[t+1][j]
                    * powtab[arnrm[t] + brnrm[t+1] - lrnrm + 6]/lhood;
            }
            a[i][j] *= (num/denom);
        }
        for (k=0; k<ksym; k++) bnew[i][k] /= denom;
    }
    b = bnew;
    fbdone = 0;                         Don't let this routine be called again until forward-
}                                           backward() has been called.
```

You must always precede a call to baumwelch by a call to forwardbackward, since the latter updates the α and β tables. Also, as you alternate calls to the two functions, you monitor convergence by the value of the log-likelihood calculated by forwardbackward.

Be aware that convergence can be excruciatingly slow! The references describe methods by which convergence can be accelerated in some cases. Common difficulties are when a rare state is not correctly captured by the model, or when the model thinks that there are two states, with nearly identical transition probabilities, when there is really only one. If you have even a plausible guess for the transition probability matrix, you should use it to start. There are many applications where you shouldn't use re-estimation at all: If you have a pretty good model to start with, just use it (via forwardbackward) to analyze your data, and don't even think about re-estimating.

The Baum-Welch re-estimation algorithm, which dates from the mid-1960s, was generalized in the mid-1970s by Dempster, Laird, and Rubin, as the *expectation-maximization (EM) algorithm*, with a variety of applications to problems with missing or censored data. (An example is the Gaussian mixture model in §16.1.) In this more general language, the forward-backward algorithm is an E-step, while Baum-Welch is an M-step. Alas, one small cloud in an otherwise bright sky is that the maximum of $\mathcal{L}$ achieved by multiple EM iterations is only guaranteed to be a local, not a global, maximum.

HMM has found wide application in speech recognition, gene sequence comparison, financial models, and a variety of other fields. The references give specifics.

16.3.4 Comparing the Viterbi Algorithm with HMM

It is important to understand the similarities and differences between the Viterbi algorithm and hidden Markov modeling (its forward-backward algorithm in particular).

When we discussed the Viterbi algorithm in the context of decoding, we made the implicit assumption that a 1 bit was a priori as likely as a 0 bit. It is straightforward to generalize the Viterbi algorithm to include an arbitrary a priori transition probability A_{ij}, just like HMM. In that case, the probability factor on each edge (whose negative logarithm is the edge cost) is the product of two terms, again just like HMM, $A_{ij} b_{ij}(k)$, where now $b_{ij}(k)$ is the probability of observing the observed symbol k given that an $i \rightarrow j$ transition occurred.

We discussed Baum-Welch parameter re-estimation for HMMs in some detail. Re-estimation of the parameters in a Viterbi model, often called *Viterbi training*, is analogous. Take the most probable path output by the algorithm (or ensemble of paths collected from the decodes of many codewords). Count the number of $i \rightarrow j$ transitions seen along these paths and the numbers of each symbol k seen for each pair i, j. Now re-estimate A_{ij} and $b_{ij}(k)$ by the obvious normalizations of these counts.

The Viterbi algorithm and the forward estimation part of the forward-backward algorithm are structurally very similar. In both cases, we sweep forward in time (or by stages) and assign a likelihood (or posterior probability) to each node, based on the data already seen. The difference is that Viterbi assigns to a node the probability of the *single best path* that reaches it, while forward-backward assigns the sum of probabilities over *all possible paths* to that node. Indeed the Viterbi algorithm is sometimes called the *min-sum algorithm* while forward-backward is referred to as the *sum-product algorithm*, just to highlight this distinction. (In the context of coding theory, the forward-backward algorithm is also sometimes called the *BCJR* or *Bahl-Cocke-Jelinek-Raviv* algorithm. In other contexts it is sometimes called *belief propagation*.)

The backward passes of the two algorithms have somewhat different structures. For Viterbi, the backward pass simply consolidates the information about the single most probable path that is already implicit in the node labeling. For forward-backward, as we have seen, the backward pass is needed to get posterior probabilities for each node that use all the data, both ahead of and behind any time t.

If you think you have a choice between using the Viterbi algorithm or using HMM, you should probably think again. Most problems clearly favor only one or the other method. If your desired output is a valid *path* on the graph, then HMM won't do: It might yield a set of highly probable nodes that just don't lie on any single path. For example, you might have the first half of one codeword and the second half of another, with no graph edge connecting the two halves. That is why decoding theory usually starts in the world of Viterbi (although, in some more complicated constructs, it can end up with a foot in each world).

On the other hand, if you care about which *nodes* are visited, then HMM is most likely what you want. In fact, Viterbi can give very poor results. The most probable path is often *very improbable* when compared to the sum of all paths that lead through a particular node, one possibly not on the most probable path. Or, another way of describing this, there may be exponentially many paths with not-too-different probabilities, so the node probabilities are determined by the statistics of where they all channel, not by which one path happens to have the highest probability.

It is very easy to "mine" HMM for alternative possibilities, since it yields seemingly every possible posterior probability that you might want to know. It is quite difficult to get anything from the Viterbi algorithm other than the single most probable path. That is because the enumeration of all possible paths is vastly harder than the enumeration of all possible nodes; the Bellman-Dijkstra-Viterbi algorithm is exquisitely good at keeping only the information that it needs. Data structures for finding more than one probable path rapidly become very complex.

Finally, we must take aim at the myth, occasionally heard, that the Viterbi algorithm, as a pure maximum likelihood (ML) estimate, is somehow "less Bayesian" than HMM. In fact, HMM is also a pure ML estimate if you look only at the state i with the largest $\alpha_t(i)\beta_t(i)$ at each time t, neither normalizing its value nor looking at any other i's. But you are then ignoring a wealth of useful information about the other possible states. (This, in part, is why you should get with the Bayesian program!) We think that both HMM and Viterbi are in fact Bayesian to the core. If there were good ways to enumerate all the other paths and their likelihoods, we would not hesitate to normalize the best-path likelihood and call it a posterior probability. It is only because of the difficulty of this enumeration that it is possible to keep the Viterbi algorithm's Bayesian character "in the closet"; and there is no advantage, that we can see, in doing so.

CITED REFERENCES AND FURTHER READING:

Hsu, H.P. 1997, *Schaum's Outline of Theory and Problems of Probability, Random Variables, and Random Processes* (New York: McGraw-Hill).

Häggström, O. 2002, *Finite Markov Chains and Algorithmic Applications* (Cambridge, UK: Cambridge University Press).

Norris, J.R. 1998, *Markov Chains*, Cambridge Series in Statistical and Probabilistic Mathematics (Cambridge, UK: Cambridge University Press).

MacDonald, I.L. and Zucchini, W. 1997, *Hidden Markov and Other Models for Discrete-Valued Time Series* (Boca Raton, FL: Chapman & Hall/CRC).

McLachlan, G.J. and Krishnan, T. 1996, *The EM Algorithm and Extensions* (New York: Wiley).

Rabiner, L. 1989, "A Tutorial on Hidden Harkov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257-286. [Review article on the use of HMMs in speech recognition.]

Eddy, S.R. 1998, "Profile Hidden Markov Models," *Bioinformatics*, vol. 14, pp. 755-763. [Review article on the use of HMMs in genetics.]

16.4 Hierarchical Clustering by Phylogenetic Trees

Hierarchical clustering is a type of *unsupervised learning*: We seek algorithms that figure out how to cluster an unordered set of input data without ever being given any training data with the "right answer." As the name implies, the output of a hierarchical clustering algorithm is a bunch of fully nested sets. The smallest sets are the individual data elements. The largest set is the whole data set. Intermediate

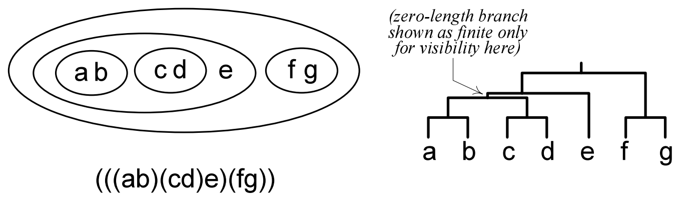

a b
c d
e
f g
(((ab)(cd)e)(fg))
(zero-length branch shown as finite only for visibility here)
a b c d e f g

Figure 16.4.1. Representations of hierarchical classification. Top left: Diagram showing fully nested sets. Bottom left: Equivalent parenthesized expression. Right: Binary tree (with possibly zero branch lengths).

sets are nested; that is, the intersection of any two sets is either the null set, or else the smaller of the two sets.

What you need to get started with hierarchical clustering is either a set of *sequences*, or else a *distance matrix*. *Character-based methods* start with n sequences each m characters long (for example, DNA bases or protein amino acids). A toy example might be the $n = 16$ sequences of $m = 12$ characters,

0.	`CGGTTGGGAGCT`	8.	`GCGCGGTGCAGC`
1.	`AGGTCGTGAGGT`	9.	`AGGCGGTGCGGG`
2.	`TGGTTGGGGTTT`	10.	`GGGCGGGGCGGG`
3.	`TGGGTGCGAGTT`	11.	`GGGCGCTGCGGG`
4.	`ACGTTTGGGTGA`	12.	`GGACGGAGGCTG`
5.	`AAGGTTGGGGAA`	13.	`GGGTGGGAGCTG`
6.	`GTCTTTCGGGTG`	14.	`AGGAGGCTGATG`
7.	`CACTTGCGGGGG`	15.	`TGGCGGATGATG`

It is probably not immediately obvious that these sequences were generated from a balanced five-level binary tree, with `GGGGGGGGGGGG` at the root, and with each daughter node having two random mutations from her parent. We will see below the extent to which some of the algorithms that we discuss can figure this out from the data. A realistic case likely would have significantly longer sequences than this, and fewer mean mutations per character; the number of sequences might be either more, or fewer, than this toy.

The alternative starting point is with an $n \times n$ matrix d_{ij} of distances between all pairs of your n data points, which might now be sequences, points in N-dimensional space, or whatever. You are responsible for making sure that the distance matrix satisfies four conditions:

$$\begin{aligned}
d_{ij} &\geq 0 && \text{(positivity)} \\
d_{ii} &= 0 && \text{(zero self-distance)} \\
d_{ij} &= d_{ji} && \text{(symmetry)} \\
d_{ik} &\leq d_{ij} + d_{jk} && \text{(triangle inequality)}
\end{aligned} \tag{16.4.1}$$

for all i, j, k. We'll discuss below how to get distances from sequences, if that's the way you want to go.

Figure 16.4.1 shows three representations of the same hierarchical clustering of seven data elements. The two representations on the left are self-explanatory. The one on the right, the binary tree, takes explaining on one point: If, in the set diagram, (ab), (cd), and (e) are clustered "democratically," then why does the binary tree

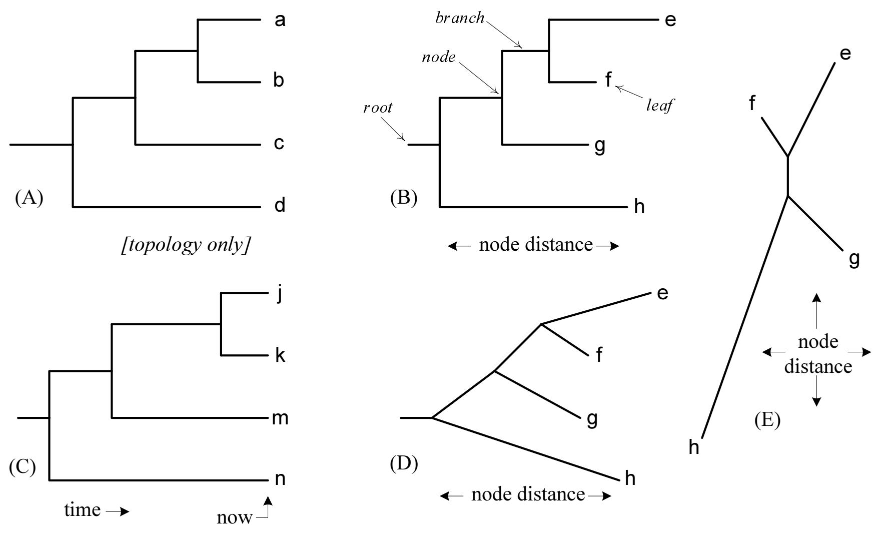


Figure 16.4.2. Types of trees. A *cladogram* (A) has arbitrary branch lengths. Only the topology is intended to be represented. A *phylogram* or *phylogenetic tree* (B) is an additive tree, where the distance between any two nodes/leaves is the sum of lengths of the horizontal connecting branches. An *ultrametric tree* (C) is an additive tree with the property that any node has the same distance to all of its leaves (as when all lengths represent time). Tree (D) is an alternative way of drawing tree (B). Again, only horizontal distances are significant. In an *unrooted tree* (E), line lengths represent distances independent of orientation. The tree (E) is the unrooted depiction of (B) or (D).

select (*e*) arbitrarily as the descendant of a higher node, instead of having three equal descendants from a common node?

The answer is just convention. Binary trees are the adopted common language of hierarchical clustering because (i) they emerge naturally from the concept of mutation events in biology, (ii) they are somewhat easier to represent in a computer than more general trees, and, (iii) they are often easier to prove theorems about. Our binary trees will almost always have the concept of *branch length*, a representation of some measure of difference between a parent node and its child. When we need to connect up democratically some number of nodes greater than two, we do it with zero-length branches. A convention is to view all topologies of nodes so connected as being equivalent.

16.4.1 Tree Basics

Figure 16.4.2 shows several ways of drawing binary trees and introduces some further terminology. The data points are *leaves*, meaning that they are generally taken as terminal nodes on the tree. Trees are often drawn either sideways (root left, leaves right) or upside down (root top, leaves bottom) by comparison with their arboreal namesake whose roots are on the bottom, leaves on top — at least for most trees that we see! A tree without meaningful branch lengths is usually called a *cladogram*. These have a rich historical tradition in pre-molecular biology but are viewed with alarm by most bioscientists today. A tree with meaningful branch lengths representing *distances* (in some metric) between nodes and their children, or between leaves, is called a *phylogram* or *phylogenetic tree*. (However, some authors use the terms cladogram and phylogram interchangeably, while some others use the words merely to distinguish different drawing styles.)

To a mathematician, a phylogenetic tree is an *additive tree*, meaning that the tree's path lengths induce a *distance metric* between any two leaves, namely the sum of path lengths up and down that connect the two leaves through their unique closest common ancestor. In real situations, the data we are given often are not exactly represented by an additive distance metric. Thus, the problem of hierarchical clustering amounts to finding a way of projecting such data onto the set of all additive trees in a useful, or statistically justifiable, way.

Given an additive tree, it is easy to compute its distance matrix d_{ij}, defined now as the matrix of all distances between pairs of leaf nodes. But what about the reverse? Given some symmetric matrix d_{ij}, is it possible to determine whether there exists an additive tree that instantiates it? Yes. One answer is the *four-point condition* for additive trees: Given four distinct leaves i, j, k, l, an additive tree exists if and only if

$$d_{ij} + d_{kl} \le \max(d_{ik} + d_{jl},\ d_{il} + d_{jk}) \qquad (16.4.2)$$

for all choices of i, j, k, l. In words, this is equivalent to the statement: For all distinct i, j, k, l, there is a tie for the maximum of the three sums of the form $d_{ij} + d_{kl}$. Later, when we discuss the neighbor-joining method, we will have a more practical algorithmic answer.

As Figure 16.4.2 illustrates, a tree can either be *rooted* or *unrooted*. An unrooted tree can always be rooted arbitrarily, by choosing any branch, grasping its midpoint between your thumb and forefinger, and then shaking the tree so that all the other branches drop downward into place. (We could give a much more mathematical description, but it would not add any clarity.) Some hierarchical clustering algorithms yield rooted trees, where the root has some meaning with respect to the data; others yield unrooted trees, although they may be drawn as if rooted, simply as a graphical convention. It's important to keep track of which kind of algorithm you are using.

You may wonder why the data points must always be leaves (terminal nodes). Might not some data points actually be good ancestors of others? The answer is again a mixture of history and convention: If the leaves are observed living taxa, then they are by definition alive today. If "today" represents terminal nodes, then they are by definition leaves. What makes this a mere convention is that we can always connect a leaf to an ancestor node by a zero-length branch, so that ancestor-versus-living-taxon becomes a distinction without a difference. A benefit of this convention is that we always know in advance how many internal nodes will be generated by n data points: $n - 1$ if the tree is rooted, or $n - 2$ for an unrooted tree, independent of the tree's topology. (If this isn't obvious, then draw a few pictures.)

If path length denotes, literally, evolutionary time, then a phylogenetic tree has the additional property of being *ultrametric* (refer to Figure 16.4.2). Ultrametric trees are defined as additive trees with the property that the distance from any node to all of its descendant leaves is the same for all such leaves. Clearly this is the case if all the leaves have the same "time distance" from their common ancestor. In the early 1960s, it was proposed that accepted mutation rates might be close enough to constant that, at the molecular level, actual evolutionary data might be close to ultrametric, i.e., that there was a "molecular clock." For most cases, this is no longer believed to be true. For example, mutation rates in *E. coli* have been found to vary by two orders of magnitude. Ultrametric trees are important mathematically, as we will see, but almost never realistic in their own right.

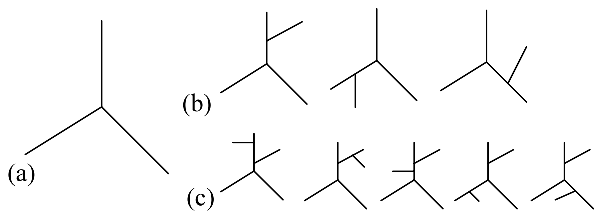


Figure 16.4.3. Tree counting. (a) There is just one unrooted tree with three leaves. (b) There are three ways to add a fourth leaf. (c) For each of the trees in (b), there are five ways to add a fifth leaf. Continuing to add leaves, the number of trees with n leaves is $1 \times 3 \times 5 \times \cdots \times (2n-5)$.

The test, analogous to equation (16.4.2), for whether a given distance matrix is ultrametric is the *three-point condition*,

$$d_{ij} \le \max(d_{ik}, d_{jk}) \tag{16.4.3}$$

for all distinct i, j, k. Equivalently, in words: Among the three distances connecting three distinct leaves, there is a tie for the maximum. Here too there is a more practical algorithmic answer, which we will mention later.

There are a *lot* of possible ways to draw a tree that connects n leaves. As Figure 16.4.3 illustrates, the number of distinct, unrooted, possibilities is

$$N_{\text{trees}}(n) = 1 \times 3 \times 5 \times \cdots \times (2n-5) \equiv (2n-5)!! \tag{16.4.4}$$

The fact that this expression grows super-exponentially with n creates a dilemma in the field of computational phylogenetics: An algorithm that requires an explicit enumeration of, or explicit search over, all possible trees can be useful only for small values of n. Thus, $N_{\text{trees}}(10) \approx 2 \times 10^6$, is easy, but $N_{\text{trees}}(20) \approx 2 \times 10^{20}$ is practically impossible.

16.4.2 Strategies for the Hierarchical Clustering Problem

If you are starting with a set of sequences, then, schematically, the goal of a character-based method is to find the best of all possible trees, given the data, for some definition of "best":

$$(\text{sequences}) \xrightarrow[\text{“best” tree}]{\text{search for}} (\text{tree}) \tag{16.4.5}$$

The two most common definitions of "best" are *maximum parsimony* and *maximum likelihood*, both of which we will say more about below [1]. Character methods are generally limited by the super-exponential explosion in the number of trees. Although the exhaustive search can be limited to some extent, for example by heuristics of various kinds, its long shadow can never be avoided entirely.

Alternatively, if you are starting with a distance matrix, the problem is to find the additive tree whose induced distance matrix (by branch lengths up and down) is closest to d_{ij}, according to some criterion of closeness. This is also an exponentially difficult problem. In practice, however, distance methods almost always use fast heuristic methods that, while provably exact only for the unrealistic case where d_{ij}

already comes from an ultrametric or additive tree, actually work pretty well for distance matrices encountered in practice. In other words, the adopted scheme is

$$\begin{pmatrix}\text{distance}\\ \text{matrix}\end{pmatrix} \xrightarrow[]{\substack{\text{ultrametric tree}\\ \text{heuristic}}} (\text{tree}) \tag{16.4.6}$$

or

$$\begin{pmatrix}\text{distance}\\ \text{matrix}\end{pmatrix} \xrightarrow[]{\substack{\text{additive tree}\\ \text{heuristic}}} (\text{tree}) \tag{16.4.7}$$

The ability of these fast heuristic methods to give "pretty good" solutions to NP-hard problems is remarkable, and only partially understood [2].

The most widely used heuristics are all *agglomerative methods*, meaning that they start by connecting individual data points into small clusters, then connect those clusters, and so forth. Common ultrametric-tree heuristic methods are *UPGMA*, *WPGMA*, *single linkage clustering*, and *complete linkage clustering*. The most widely used additive-tree heuristic method — and probably the most widely used phylogenetic clustering method overall — is the *neighbor-joining (NJ)* method [6]. We will discuss, and implement, all the mentioned heuristic methods below.

There are a few, less-well-developed, distance-based methods that avoid heuristics by finding provably error-bounded methods for transforming an arbitrary distance matrix into the matrix of an additive tree, then exactly constructing the resulting tree [3,4],

$$\begin{pmatrix}\text{distance}\\ \text{matrix}\end{pmatrix} \xrightarrow[]{\substack{\text{find nearby}\\ \text{additive}}} \begin{pmatrix}\text{additive}\\ \text{matrix}\end{pmatrix} \xrightarrow[]{\substack{\text{exact}\\ \text{construction}}} (\text{tree}) \tag{16.4.8}$$

Though more rigorous than the heuristic methods, there is little evidence that these methods produce better results [5].

Evidently, one can always turn a character-based problem into a distance-based one by defining a distance on character sequences,

$$(\text{sequences}) \xrightarrow[]{\substack{\text{define a}\\ \text{distance}}} \begin{pmatrix}\text{distance}\\ \text{matrix}\end{pmatrix} \tag{16.4.9}$$

and then continuing with schemes (16.4.6) or (16.4.7).

The obvious distance between two sequences is their *Hamming distance* $H(i, j)$, which is defined as the number of character positions in which sequence i differs from sequence j, an integer between 0 and m. However, when you are given not just the data, but also a statistical model defining how it was generated (i.e., "evolved"), there is often a *corrected distance transformation* that will give better tree reconstructions [2]. For example, the popular *Cavender-Felsenstein* model (whose discussion is beyond our scope) has the corrected distance transformation

$$d_{ij} = -\tfrac{1}{2}\log\left(1 - 2H(i, j)/m\right) \tag{16.4.10}$$

This expression can be used directly when sequences are long enough, or mutation probabilities small enough, that the argument of the logarithm is never negative. If your data produce a negative argument, then a standard workaround is to use a multiple (1× or 2×) of the largest computable d_{ij} for all uncomputable d_{ij}'s. Corrected distance transformations also exist for general Markov models.

Corrected distance transformations have the defining (and desirable) property that as the sequence length increases, the matrix of observed corrected distances will converge to the distance matrix of an additive tree. (This is not true for the uncorrected Hamming distance, incidentally.) In such a case, the power of an additive-tree heuristic method like neighbor joining is much less mysterious. Corrected distance transformations thus provide a statistical justification for the use of the neighbor-joining method.

16.4.3 Implementation of Agglomerative Methods

The general scheme of an agglomerative method is, first, to initialize n active clusters, each containing one data point, and, second, to repeat the following operations exactly $n-2$ times:

- Find the two active clusters that are closest by some prescribed distance measure.
- Create a new active cluster that combines the two.
- Connect the new cluster, as parent, to the two closest clusters, as children, with some prescription for the two branch lengths.
- Delete the two children from the active list.
- Compute, by some prescription, distances from the new cluster to the active clusters that remain.

Each repetition of these steps reduces the active cluster list by exactly one (one addition, two deletions), so after $n-2$ repetitions there will be exactly two active clusters. You connect these either by a single branch (unrooted case), or by creating a root node between them (rooted case) with some prescription as to the two root branch lengths.

As we now turn to implementing phylogenetic tree routines, a few words of caution are in order. Hamming's dictum, that the purpose of computing is insight, not numbers, applies here: Much of the value of a phylogenetic tree program lies in its graphics and user interface, both areas outside of our scope. If you are working with any significant quantity of real data, you probably want to use a sophisticated package. As we write, PAUP (Phylogenetic Analysis Using Parsimony) [7] is the most widely used commercial package, both for maximum parsimony trees and also for the various heuristic methods. PHYLIP (Phylogeny Inference Package) [8] is a free package for smaller trees ($\lesssim 20$ taxa). TreeView is a widely used, free, program used for drawing trees in various formats. A user guide to the use of these and other programs is [9]. If the insight you desire lies in algorithmics, not production data, then you may read on.

Here is an abstract base class that implements the general agglomerative method, leaving the various "prescriptions" to be specified by particular derived classes, which we give later.

phylo.h
```
struct Phylagglomnode {
Node for phylogenetic tree.
    Int mo,ldau,rdau,nel;                Pointers up and down; no. of elements.
    Doub modist,dep,seq;                 Branch length to parent. See text re. dep and
};                                           seq.

struct Phylagglom{
Abstract base class for constructing an agglomerative phylogenetic tree.
```

```
Int n, root, fsroot;                 No. of data points, root node, forced root.
Doub seqmax, depmax;                 Max.  values of seq, dep over the tree.
vector<Phylagglomnode> t;            The tree.
virtual void premin(MatDoub &d, VecInt &nextp) = 0;
Function called before minimum search.
virtual Doub dminfn(MatDoub &d, Int i, Int j) = 0;
Distance function to be minimized
virtual Doub dbranchfn(MatDoub &d, Int i, Int j) = 0;
Branch length, node i to mother (j is sister).
virtual Doub dnewfn(MatDoub &d, Int k, Int i, Int j, Int ni, Int nj) = 0;
Distance function for newly constructed nodes.
virtual void drootbranchfn(MatDoub &d, Int i, Int j, Int ni, Int nj,
Doub &bi, Doub &bj) = 0;
Sets branch lengths to the final root node.
Int comancestor(Int leafa, Int leafb);         See text discussion of NJ.
Phylagglom(const MatDoub &dist, Int fsr = -1)
    : n(dist.nrows()), fsroot(fsr), t(2*n-1) {}
    Constructor is always called by a derived class.

void makethetree(const MatDoub &dist) {
Routine that actually constructs the tree, called by the constructor of a derived class.
    Int i, j, k, imin, jmin, ncurr, node, ntask;
    Doub dd, dmin;
    MatDoub d(dist);                 Matrix d is initialized with dist.
    VecInt tp(n), nextp(n), prevp(n), tasklist(2*n+1);
    VecDoub tmp(n);
    for (i=0;i<n;i++) {              Initializations on leaf elements.
        nextp[i] = i+1;              nextp and prevp are for looping on the distance
        prevp[i] = i-1;                  matrix even as it becomes sparse.
        tp[i] = i;                   tp points from a distance matrix row to a tree
        t[i].ldau = t[i].rdau = -1;          element.
        t[i].nel = 1;
    }
    prevp[0] = nextp[n-1] = -1;      Signifying end of loop.
    ncurr = n;
    for (node = n; node < 2*n-2; node++) {     Main loop!
        premin(d,nextp);             Any calculations needed before min finding.
        dmin = 9.99e99;
        for (i=0; i>=0; i=nextp[i]) {             Find i, j pair with min distance.
            if (tp[i] == fsroot) continue;
            for (j=nextp[i]; j>=0; j=nextp[j]) {
                if (tp[j] == fsroot) continue;
                if ((dd = dminfn(d,i,j)) < dmin) {
                    dmin = dd;
                    imin = i; jmin = j;
                }
            }
        }
        i = imin; j = jmin;
        t[tp[i]].mo = t[tp[j]].mo = node;         Now set properties of the parent
        t[tp[i]].modist = dbranchfn(d,i,j);           and children.
        t[tp[j]].modist = dbranchfn(d,j,i);
        t[node].ldau = tp[i];
        t[node].rdau = tp[j];
        t[node].nel = t[tp[i]].nel + t[tp[j]].nel;
        for (k=0; k>=0; k=nextp[k]) {             Get new-node distances.
            tmp[k] = dnewfn(d,k,i,j,t[tp[i]].nel,t[tp[j]].nel);
        }
        for (k=0; k>=0; k=nextp[k]) d[i][k] = d[k][i] = tmp[k];
        tp[i] = node;             New node replaces child i in dist. matrix, while child
        if (prevp[j] >= 0) nextp[prevp[j]] = nextp[j];    j gets patched around.
        if (nextp[j] >= 0) prevp[nextp[j]] = prevp[j];
        ncurr--;
    }                                End of main loop.
```

```
        i = 0; j = nextp[0];                      Set properties of the root node.
        root = node;
        t[tp[i]].mo = t[tp[j]].mo = t[root].mo = root;
        drootbranchfn(d,i,j,t[tp[i]].nel,t[tp[j]].nel,
            t[tp[i]].modist,t[tp[j]].modist);
        t[root].ldau = tp[i];
        t[root].rdau = tp[j];
        t[root].modist = t[root].dep = 0.;
        t[root].nel = t[tp[i]].nel + t[tp[j]].nel;
        We now traverse the tree computing seq and dep, hints for where to plot nodes in a
        two-dimensional representation. See text.
        ntask = 0;
        seqmax = depmax = 0.;
        tasklist[ntask++] = root;
        while (ntask > 0) {
            i = tasklist[--ntask];
            if (i >= 0) {                         Meaning, process going down the tree.
                t[i].dep = t[t[i].mo].dep + t[i].modist;
                if (t[i].dep > depmax) depmax = t[i].dep;
                if (t[i].ldau < 0) {              A leaf node.
                    t[i].seq = seqmax++;
                } else {                          Not a leaf node.
                    tasklist[ntask++] = -i-1;
                    tasklist[ntask++] = t[i].ldau;
                    tasklist[ntask++] = t[i].rdau;
                }
            } else {                              Meaning, process coming up the tree.
                i = -i-1;
                t[i].seq = 0.5*(t[t[i].ldau].seq + t[t[i].rdau].seq);
            }
        }
    }
};
```

The Phylagglom structure creates a tree of Phylagglomnodes. Each node carries pointers to its mother and two daughters, and knows its number of elements (original data points), branch length to its mother, and two floating values dep and seq, which we now explain: The final while block in makethetree() does a depth-first traversal of the finished tree. When it reaches a node in the downward direction, it sets dep to the sum of branch lengths to the root node. The variable dep is thus a hint as to where to plot the node in the depth direction. When the traversal reaches a leaf, it sets seq to a sequential numbering of leaves. When it reaches an internal node in the upward direction, it sets its seq value to the average of the seq values of its two children. The value of seq thus becomes a hint as to where to plot a node perpendicular to the depth direction. If you plot nodes by dep and seq, then plotted branches won't cross each other.

Looking at the nested loops, you can see that makethetree() is $O(n^3)$ in time. Actually, it is straightforward to reduce this to $O(n^2)$: With some extra bookkeeping, you can keep the distances in a structure that allows the shortest to be found without an n^2 search. We have not coded this, just to keep the code shorter and simpler.

16.4.4 Algorithms That Are Exact for Ultrametric Trees

Given a distance matrix that is exactly ultrametric, all of the agglomerative algorithms that we now discuss will (modulo some technical details) reconstruct its tree exactly. The reason that we need more than one such algorithm is because their behaviors can be somewhat different on realistic, nonultrametric, data, in the general

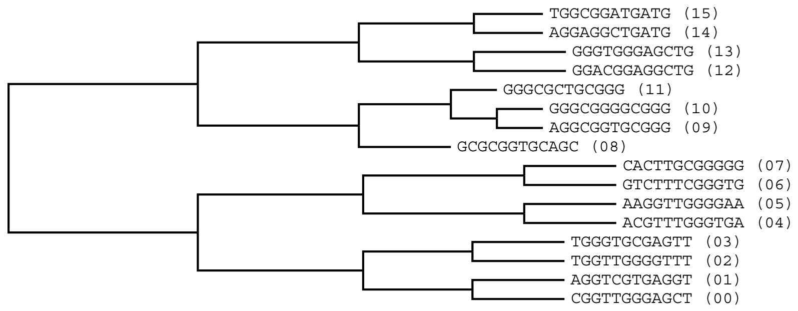


Figure 16.4.4. Example of WPGMA agglomerative clustering on a toy problem. Strings were mutated hierarchically from GGGGGGGGGGGG to produce the input data. WPGMA and related methods (UPGMA, SLC, CLC) yield perfect results for perfectly ultrametric input data, but can deviate badly when that assumption is violated. In this example, however, it does quite well.

scheme of (16.4.6). The different algorithms are distinguished by their prescriptions for computing the distances to new nodes.

The *weighted pair group method using arithmetic averages (WPGMA)* uses this prescription: If a new cluster k is formed from old clusters i and j, then the distance from k to another active cluster p is

$$d_{pk} = d_{kp} = \tfrac{1}{2}(d_{pi} + d_{pj}) \tag{16.4.11}$$

that is, just the average of distances to the two children.

Implementing code, as a class derived from Phylagglom, is

phylo.h

```
struct Phylo_wpgma : Phylagglom {
Derived class implementing the WPGMA method. Only need to define functions that are virtual
in Phylagglom.
    void premin(MatDoub &d, VecInt &nextp) {}          No pre-min calculations.
    Doub dminfn(MatDoub &d, Int i, Int j) {return d[i][j];}
    Doub dbranchfn(MatDoub &d, Int i, Int j) {return 0.5*d[i][j];}
    Doub dnewfn(MatDoub &d, Int k, Int i, Int j, Int ni, Int nj) {
        return 0.5*(d[i][k]+d[j][k]);}                 New-node distance is average.
    void drootbranchfn(MatDoub &d, Int i, Int j, Int ni, Int nj,
        Doub &bi, Doub &bj) {bi = bj = 0.5*d[i][j];}
    Phylo_wpgma(const MatDoub &dist) : Phylagglom(dist)
        {makethetree(dist);}                           This call actually makes the tree.
};
```

Figure 16.4.4 shows the result of applying the WPGMA method to the toy data at the beginning of this section, using the Hamming distance as the distance metric. You can see that the tree captures almost all of the correct underlying topology, erring only in its pairing of 09 and 10 and thus missing the correct pairings 08-09 and 10-11.

The *unweighted pair group method using arithmetic averages (UPGMA)* uses

$$d_{pk} = d_{kp} = \frac{n_i d_{pi} + n_j d_{pj}}{n_i + n_j} \tag{16.4.12}$$

Although, paradoxically, UPGMA looks "weighted" while WPGMA looks "unweighted," the names derive from the fact that the UPGMA formula is equivalent to an *un*weighted average of distances to all of a node's descendant leaves. The UPGMA method is the most widely used of the ultrametric heuristic methods.

Implementing code is

phylo.h
```
struct Phylo_upgma : Phylagglom {
Derived class implementing the UPGMA method. Only need to define functions that are virtual
in Phylagglom.
    void premin(MatDoub &d, VecInt &nextp) {}             No pre-min calculations.
    Doub dminfn(MatDoub &d, Int i, Int j) {return d[i][j];}
    Doub dbranchfn(MatDoub &d, Int i, Int j) {return 0.5*d[i][j];}
    Doub dnewfn(MatDoub &d, Int k, Int i, Int j, Int ni, Int nj) {
        return (ni*d[i][k] + nj*d[j][k]) / (ni+nj);}      Distance is weighted.
    void drootbranchfn(MatDoub &d, Int i, Int j, Int ni, Int nj,
        Doub &bi, Doub &bj) {bi = bj = 0.5*d[i][j];}
    Phylo_upgma(const MatDoub &dist) : Phylagglom(dist)
        {makethetree(dist);}                              This call actually makes the
};                                                            tree.
```

The *single linkage clustering method* and the *complete linkage clustering method* use, respectively, the minimum and maximum distances to the two children,

$$\begin{aligned} d_{pk} = d_{kp} &= \min(d_{pi}, d_{pj}) \qquad \text{(single linkage)} \\ d_{pk} = d_{kp} &= \max(d_{pi}, d_{pj}) \qquad \text{(complete linkage)} \end{aligned} \tag{16.4.13}$$

Implementing code is

phylo.h
```
struct Phylo_slc : Phylagglom {
Derived class implementing the single linkage clustering method.
    void premin(MatDoub &d, VecInt &nextp) {}      No pre-min calculations.
    Doub dminfn(MatDoub &d, Int i, Int j) {return d[i][j];}
    Doub dbranchfn(MatDoub &d, Int i, Int j) {return 0.5*d[i][j];}
    Doub dnewfn(MatDoub &d, Int k, Int i, Int j, Int ni, Int nj) {
        return MIN(d[i][k],d[j][k]);}          New-node distance is min of children.
    void drootbranchfn(MatDoub &d, Int i, Int j, Int ni, Int nj,
        Doub &bi, Doub &bj) {bi = bj = 0.5*d[i][j];}
    Phylo_slc(const MatDoub &dist) : Phylagglom(dist)
        {makethetree(dist);}                       This call actually makes the tree.
};

struct Phylo_clc : Phylagglom {
Derived class implementing the complete linkage clustering method.
    void premin(MatDoub &d, VecInt &nextp) {}          No pre-min calculations.
    Doub dminfn(MatDoub &d, Int i, Int j) {return d[i][j];}
    Doub dbranchfn(MatDoub &d, Int i, Int j) {return 0.5*d[i][j];}
    Doub dnewfn(MatDoub &d, Int k, Int i, Int j, Int ni, Int nj) {
        return MAX(d[i][k],d[j][k]);}          New-node distance is max of children.
    void drootbranchfn(MatDoub &d, Int i, Int j, Int ni, Int nj,
        Doub &bi, Doub &bj) {bi = bj = 0.5*d[i][j];}
    Phylo_clc(const MatDoub &dist) : Phylagglom(dist)
        {makethetree(dist);}                   This call actually makes the tree.
};
```

16.4.5 Neighbor Joining: Exact for Additive Trees

Saitou and Nei's *neighbor-joining method (NJ)* [6] is an agglomerative method with the remarkable property that it exactly reconstructs any additive tree, given that tree's distance matrix (again modulo some technical details). NJ is probably the most widely used agglomerative method, and perhaps the most widely used method for phylogenetic tree construction overall. Real biological trees are almost never close enough to ultrametric to give UPGMA a significant advantage over NJ, so NJ is likely the method, among the fast heuristic methods, that you will want to try first.

The prescriptions for treating NJ within the framework of `Phyloagglom` are slightly more complicated than for the ultrametric heuristics. At each stage of forming a new cluster, we compute an auxiliary quantity,

$$u_i \equiv \frac{1}{n_a - 2} \sum_{j \neq i} d_{ij} \tag{16.4.14}$$

where the sum is over active clusters, whose number is n_a. Then, we find not the minimum distance, per se, but the minimum of the expression

$$d_{ij} - u_i - u_j \tag{16.4.15}$$

When we connect clusters i and j to form a new node k, the branch lengths from i to k and from j to k are

$$\begin{aligned} d_{ik} &= \tfrac{1}{2}(d_{ij} + u_i - u_j) \\ d_{jk} &= \tfrac{1}{2}(d_{ij} + u_j - u_i) \end{aligned} \tag{16.4.16}$$

Finally, the distance between new node k and another node p is

$$d_{pk} = d_{kp} = \tfrac{1}{2}(d_{pi} + d_{pj} - d_{ij}) \tag{16.4.17}$$

(You can now see why `Phylagglom` was coded with some features that were not exercised by the ultrametric heuristics.)

phylo.h

```
struct Phylo_nj : Phylagglom {
Derived class implementing the neighbor joining (NJ) method.
    VecDoub u;
    void premin(MatDoub &d, VecInt &nextp) {
    Before finding the minimum we (re-)calculate the u's.
        Int i,j,ncurr = 0;
        Doub sum;
        for (i=0; i>=0; i=nextp[i]) ncurr++;          Count live entries.
        for (i=0; i>=0; i=nextp[i]) {                  Compute u[i].
            sum = 0.;
            for (j=0; j>=0; j=nextp[j]) if (i != j) sum += d[i][j];
            u[i] = sum/(ncurr-2);
        }
    }
    Doub dminfn(MatDoub &d, Int i, Int j) {
        return d[i][j] - u[i] - u[j];                  NJ finds min of this.
    }
    Doub dbranchfn(MatDoub &d, Int i, Int j) {
        return 0.5*(d[i][j]+u[i]-u[j]);                NJ setting for branch lengths.
    }
    Doub dnewfn(MatDoub &d, Int k, Int i, Int j, Int ni, Int nj) {
        return 0.5*(d[i][k] + d[j][k] - d[i][j]);      NJ new distances.
    }
    void drootbranchfn(MatDoub &d, Int i, Int j, Int ni, Int nj,
    Doub &bi, Doub &bj) {
        Since NJ is unrooted, it is a matter of taste how to assign branch lengths to the root.
        This prescription plots aesthetically.
        bi = d[i][j]*(nj - 1 + 1.e-15)/(ni + nj -2 + 2.e-15);
```

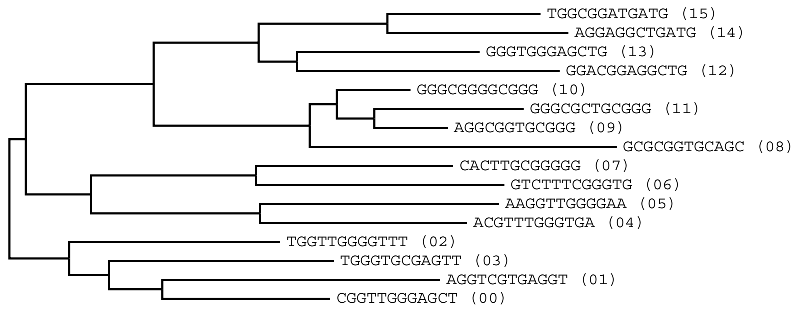


Figure 16.4.5. Same data as previous figure, clustered by the neighbor-joining (NJ) method. The method yields perfect results when the input data are the metric of an additive tree (which these data are not). While displayed here as if rooted, the NJ method outputs an unrooted tree (see next figure).

```
        bj = d[i][j]*(ni - 1 + 1.e-15)/(ni + nj -2 + 2.e-15);
    }
    Phylo_nj(const MatDoub &dist, Int fsr = -1)
        : Phylagglom(dist,fsr), u(n) {makethetree(dist);}
};
```

Computing the u_i's is here coded as an $O(n^2)$ process, but it is repeated $O(n)$ times, so it adds $O(n^3)$ to the workload. It is straightforward to make this $O(n^2)$, in line with the best coding for the rest of the tree construction. When you compute the u_i's, most distances have not changed; you just need to correct those that have. We have not coded this, just to keep the code concise.

It is important to keep in mind that the neighbor-joining method intrinsically produces an *unrooted* tree, regardless of how the graphical output may be drawn. Figure 16.4.5 shows the tree produced by the above code, run on the same toy example as above. It is clear by inspection that, if we want to root the tree at all, we will likely do so at some different point than the one drawn. It is for just this reason that Phylo_nj's constructor has an optional integer argument for specifying a node as an immediate daughter to a "forced" root. (You can't specify a new root by its node number, because it doesn't exist yet.) Also, since you may not know how Phyloagglom has numbered its internal nodes, there is a method that returns the node number of an internal node, given two leaves that have it as their first common ancestor.

phylo.h

```
Int Phylagglom::comancestor(Int leafa, Int leafb) {
Given the node numbers of two leaves, return the node number of their first common ancestor.
    Int i, j;
    for (i = leafa; i != root; i = t[i].mo) {
        for (j = leafb; j != root; j = t[j].mo) if (i == j) break;
        if (i == j) break;
    }
    return i;
}
```

Figure 16.4.6 shows the result of rerooting the tree of Figure 16.4.5 to the common ancestor of leaves 08 and 15. The recovered topology is now seen to be almost identical to that recovered by WPGMA, except for one additional mistake in not giving 02 and 03 a common parent.

The two figures, 16.4.5 and 16.4.6, were produced by lines of code like