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 has 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

$$\sigma^2 = \sum_{i=1}^{N} [y_i - y(x_i)]^2 / (N - M) \tag{15.1.6}$$

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 (15.1.5) with respect to the parameters a_k, we obtain equations that must hold at the chi-square minimum,

$$0 = \sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i^2} \right) \left(\frac{\partial y(x_i; \ldots a_k \ldots)}{\partial a_k} \right) \qquad k = 1, \ldots, M \tag{15.1.7}$$

Equation (15.1.7) 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 (15.1.7) and its specializations.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapters 1–4.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), §VI.C. [1]

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 (15.1.5), which in this case is

$$\chi^2(a,b) = \sum_{i=1}^{N} \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2 \qquad (15.2.2)$$

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.

$$\begin{aligned} 0 &= \frac{\partial \chi^2}{\partial a} = -2 \sum_{i=1}^{N} \frac{y_i - a - bx_i}{\sigma_i^2} \\ 0 &= \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=1}^{N} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2} \end{aligned} \qquad (15.2.3)$$

These conditions can be rewritten in a convenient form if we define the following sums:

$$\begin{aligned} &S \equiv \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \quad S_x \equiv \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \quad S_y \equiv \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ &S_{xx} \equiv \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \quad S_{xy} \equiv \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \qquad (15.2.4)$$

With these definitions (15.2.3) becomes

$$\begin{aligned} aS + bS_x &= S_y \\ aS_x + bS_{xx} &= S_{xy} \end{aligned} \qquad (15.2.5)$$

The solution of these two equations in two unknowns is calculated as

$$\begin{aligned} \Delta &\equiv SS_{xx} - (S_x)^2 \\ a &= \frac{S_{xx}S_y - S_xS_{xy}}{\Delta} \\ b &= \frac{SS_{xy} - S_xS_y}{\Delta} \end{aligned} \qquad (15.2.6)$$

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

$$\sigma_f^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial f}{\partial y_i}\right)^2 \tag{15.2.7}$$

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

$$\text{Cov}(a, b) = -S_x/\Delta \tag{15.2.10}$$

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

$$Q = \texttt{gammq}\left(\frac{N-2}{2}, \frac{\chi^2}{2}\right) \tag{15.2.12}$$

Here gammq is our routine for the incomplete gamma function $Q(a, x)$, §6.2. 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 (15.1.6) 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

$$\chi^2 = (1 - r^2)\text{NVar}\,(y_1 \dots y_N) \qquad (15.2.13)$$

where

$$\text{NVar}\,(y_1 \dots y_N) \equiv \sum_{i=1}^{N} (y_i - \overline{y})^2 \qquad (15.2.14)$$

For data with varying weights σ_i, the above equations remain valid if the sums in equation (14.5.1) are weighted by $1/\sigma_i^2$.

The following function, fit, carries out exactly the operations that we have discussed. When the weights σ are known in advance, the calculations exactly correspond to the formulas above. However, when weights σ 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

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \qquad i = 1, 2, \dots, N \qquad (15.2.15)$$

and

$$S_{tt} = \sum_{i=1}^{N} t_i^2 \qquad (15.2.16)$$

Then, as you can verify by direct substitution,

$$b = \frac{1}{S_{tt}} \sum_{i=1}^{N} \frac{t_i y_i}{\sigma_i} \qquad (15.2.17)$$

$$a = \frac{S_y - S_x b}{S} \qquad (15.2.18)$$

$$\sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{SS_{tt}}\right) \tag{15.2.19}$$

$$\sigma_b^2 = \frac{1}{S_{tt}} \tag{15.2.20}$$

$$\mathrm{Cov}(a,b) = -\frac{S_x}{SS_{tt}} \tag{15.2.21}$$

$$r_{ab} = \frac{\mathrm{Cov}(a,b)}{\sigma_a \sigma_b} \tag{15.2.22}$$

```
#include <math.h>
#include "nrutil.h"

void fit(float x[], float y[], int ndata, float sig[], int mwt, float *a,
    float *b, float *siga, float *sigb, float *chi2, float *q)
```
Given a set of data points x[1..ndata],y[1..ndata] with individual standard deviations sig[1..ndata], fit them to a straight line $y = a + bx$ by minimizing χ^2. Returned are 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). If mwt=0 on input, then the standard deviations are assumed to be unavailable: q is returned as 1.0 and the normalization of chi2 is to unit standard deviation on all points.
```
{
    float gammq(float a, float x);
    int i;
    float wt,t,sxoss,sx=0.0,sy=0.0,st2=0.0,ss,sigdat;

    *b=0.0;
    if (mwt) {                                  Accumulate sums ...
        ss=0.0;
        for (i=1;i<=ndata;i++) {                ...with weights
            wt=1.0/SQR(sig[i]);
            ss += wt;
            sx += x[i]*wt;
            sy += y[i]*wt;
        }
    } else {
        for (i=1;i<=ndata;i++) {                ...or without weights.
            sx += x[i];
            sy += y[i];
        }
        ss=ndata;
    }
    sxoss=sx/ss;
    if (mwt) {
        for (i=1;i<=ndata;i++) {
            t=(x[i]-sxoss)/sig[i];
            st2 += t*t;
            *b += t*y[i]/sig[i];
        }
    } else {
        for (i=1;i<=ndata;i++) {
            t=x[i]-sxoss;
            st2 += t*t;
            *b += t*y[i];
        }
    }
    *b /= st2;                                  Solve for a, b, σa, and σb.
    *a=(sy-sx*(*b))/ss;
    *siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
    *sigb=sqrt(1.0/st2);
    *chi2=0.0;                                          Calculate χ².
    *q=1.0;
    if (mwt == 0) {
        for (i=1;i<=ndata;i++)
            *chi2 += SQR(y[i]-(*a)-(*b)*x[i]);
        sigdat=sqrt((*chi2)/(ndata-2));                 For unweighted data evaluate typ-
        *siga *= sigdat;                                    ical sig using chi2, and ad-
        *sigb *= sigdat;                                    just the standard deviations.
    } else {
        for (i=1;i<=ndata;i++)
            *chi2 += SQR((y[i]-(*a)-(*b)*x[i])/sig[i]);
        if (ndata>2) *q=gammq(0.5*(ndata-2),0.5*(*chi2));      Equation (15.2.12).
    }
}
```

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 6.

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,

$$\chi^2(a,b) = \sum_{i=1}^{N} \frac{(y_i - a - bx_i)^2}{\sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2} \tag{15.3.2}$$

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 χ^2-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

$$a = \left[\sum_i w_i(y_i - bx_i)\right] \Bigg/ \sum_i w_i \tag{15.3.4}$$

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., the routine `brent`) 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.

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,

$$y(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_M x^{M-1} \tag{15.4.1}$$

is a polynomial of degree $M - 1$. Or, the functions could be sines and cosines, in which case their general linear combination is a harmonic series.

The general form of this kind of model is

$$y(x) = \sum_{k=1}^{M} a_k X_k(x) \tag{15.4.2}$$

where $X_1(x), \dots, X_M(x)$ 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

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.3}$$

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 **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 **A** is called the *design matrix* of the fitting problem. Notice that in general **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.

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, $a_1, \dots, a_M$, by **a**.

$$
\begin{array}{cc}
 & \longleftarrow \text{basis functions} \longrightarrow \\
 & \begin{array}{cccc} X_1(\;) & X_2(\;) & \cdots & X_M(\;) \end{array} \\
\begin{array}{c} \uparrow \\ \text{data points} \\ \downarrow \end{array}
\begin{array}{c} x_1 \\ \\ x_2 \\ \\ \vdots \\ \\ \\ \\ x_N \end{array}
&
\left(
\begin{array}{cccc}
\dfrac{X_1(x_1)}{\sigma_1} & \dfrac{X_2(x_1)}{\sigma_1} & \cdots & \dfrac{X_M(x_1)}{\sigma_1} \\
\dfrac{X_1(x_2)}{\sigma_2} & \dfrac{X_2(x_2)}{\sigma_2} & \cdots & \dfrac{X_M(x_2)}{\sigma_2} \\
\vdots & & & \vdots \\
\vdots & & & \vdots \\
\vdots & & & \vdots \\
\dfrac{X_1(x_N)}{\sigma_N} & \dfrac{X_2(x_N)}{\sigma_N} & \cdots & \dfrac{X_M(x_N)}{\sigma_N}
\end{array}
\right)
\end{array}
$$

Figure 15.4.1. Design matrix 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.

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 (15.1.7) to the case of the model (15.4.2), this condition yields the M equations

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_k(x_i) \qquad k = 1, \dots, M \tag{15.4.6}$$

Interchanging the order of summations, we can write (15.4.6) as the matrix equation

$$\sum_{j=1}^{M} \alpha_{kj} a_j = \beta_k \tag{15.4.7}$$

where

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \qquad \text{or equivalently} \qquad [\alpha] = \mathbf{A}^T \cdot \mathbf{A} \tag{15.4.8}$$

an $M \times M$ matrix, and

$$\beta_k = \sum_{i=1}^{N} \frac{y_i X_k(x_i)}{\sigma_i^2} \qquad \text{or equivalently} \qquad [\beta] = \mathbf{A}^T \cdot \mathbf{b} \tag{15.4.9}$$

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

$$[\alpha]\cdot\mathbf{a}=[\beta] \qquad \text{or as} \qquad \left(\mathbf{A}^T\cdot\mathbf{A}\right)\cdot\mathbf{a}=\mathbf{A}^T\cdot\mathbf{b} \tag{15.4.10}$$

The inverse matrix $C_{jk}\equiv[\alpha]^{-1}_{jk}$ is closely related to the probable (or, more precisely, *standard*) uncertainties of the estimated parameters **a**. To estimate these uncertainties, consider that

$$a_j=\sum_{k=1}^{M}[\alpha]^{-1}_{jk}\beta_k=\sum_{k=1}^{M}C_{jk}\left[\sum_{i=1}^{N}\frac{y_iX_k(x_i)}{\sigma_i^2}\right] \tag{15.4.11}$$

and that the variance associated with the estimate a_j can be found as in (15.2.7) from

$$\sigma^2(a_j)=\sum_{i=1}^{N}\sigma_i^2\left(\frac{\partial a_j}{\partial y_i}\right)^2 \tag{15.4.12}$$

Note that α_{jk} is independent of y_i, so that

$$\frac{\partial a_j}{\partial y_i}=\sum_{k=1}^{M}C_{jk}X_k(x_i)/\sigma_i^2 \tag{15.4.13}$$

Consequently, we find that

$$\sigma^2(a_j)=\sum_{k=1}^{M}\sum_{l=1}^{M}C_{jk}C_{jl}\left[\sum_{i=1}^{N}\frac{X_k(x_i)X_l(x_i)}{\sigma_i^2}\right] \tag{15.4.14}$$

The final term in brackets is just the matrix $[\alpha]$. Since this is the matrix inverse of $[C]$, (15.4.14) reduces immediately to

$$\sigma^2(a_j)=C_{jj} \tag{15.4.15}$$

In other words, the diagonal elements of $[C]$ 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. An alternative, and preferred, technique involves QR decomposition (§2.10, §11.3, and §11.6) of the design matrix **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 routine below introduces one bookkeeping trick that is 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 routine the total number of parameters a_k is denoted ma (called M above). As input to the routine, you supply an array ia[1..ma], whose components are either zero or nonzero (e.g., 1). Zeros indicate that you want the corresponding elements of the parameter vector a[1..ma] to be held fixed at their input values. Nonzeros 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.

```
#include "nrutil.h"

void lfit(float x[], float y[], float sig[], int ndat, float a[], int ia[],
    int ma, float **covar, float *chisq, void (*funcs)(float, float [], int))
```
Given a set of data points x[1..ndat], y[1..ndat] with individual standard deviations sig[1..ndat], use χ^2 minimization to fit for some or all of the coefficients a[1..ma] of a function that depends linearly on a, $y = \sum_i \mathtt{a}_i \times \mathtt{afunc}_i(x)$. The input array ia[1..ma] indicates by nonzero entries those components of a that should be fitted for, and by zero entries those components that should be held fixed at their input values. The program returns values for a[1..ma], $\chi^2 = \mathtt{chisq}$, and the covariance matrix covar[1..ma][1..ma]. (Parameters held fixed will return zero covariances.) The user supplies a routine funcs(x,afunc,ma) that returns the ma basis functions evaluated at $x = \mathtt{x}$ in the array afunc[1..ma].
```
{
    void covsrt(float **covar, int ma, int ia[], int mfit);
    void gaussj(float **a, int n, float **b, int m);
    int i,j,k,l,m,mfit=0;
    float ym,wt,sum,sig2i,**beta,*afunc;

    beta=matrix(1,ma,1,1);
    afunc=vector(1,ma);
    for (j=1;j<=ma;j++)
        if (ia[j]) mfit++;
    if (mfit == 0) nrerror("lfit: no parameters to be fitted");
    for (j=1;j<=mfit;j++) {                 Initialize the (symmetric) matrix.
        for (k=1;k<=mfit;k++) covar[j][k]=0.0;
        beta[j][1]=0.0;
    }
    for (i=1;i<=ndat;i++) {                 Loop over data to accumulate coefficients of
                                                the normal equations.
        (*funcs)(x[i],afunc,ma);
        ym=y[i];
        if (mfit < ma) {                    Subtract off dependences on known pieces
            for (j=1;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=1;l<=ma;l++) {
            if (ia[l]) {
                wt=afunc[l]*sig2i;
                for (j++,k=0,m=1;m<=l;m++)
                    if (ia[m]) covar[j][++k] += wt*afunc[m];
                beta[j][1] += ym*wt;
            }
        }
    }
    for (j=2;j<=mfit;j++)                   Fill in above the diagonal from symmetry.
        for (k=1;k<j;k++)
            covar[k][j]=covar[j][k];
    gaussj(covar,mfit,beta,1);              Matrix solution.
    for (j=0,l=1;l<=ma;l++)
        if (ia[l]) a[l]=beta[++j][1];       Partition solution to appropriate coefficients
    *chisq=0.0;                                 a.
    for (i=1;i<=ndat;i++) {                 Evaluate χ² of the fit.
        (*funcs)(x[i],afunc,ma);
        for (sum=0.0,j=1;j<=ma;j++) sum += a[j]*afunc[j];
        *chisq += SQR((y[i]-sum)/sig[i]);
    }
    covsrt(covar,ma,ia,mfit);               Sort covariance matrix to true order of fitting
    free_vector(afunc,1,ma);                    coefficients.
    free_matrix(beta,1,ma,1,1);
}
```

That last call to a function `covsrt` is only for the purpose of spreading the covariances back into the full `ma` $\times$ `ma` covariance matrix, in the proper rows and columns and with zero variances and covariances set for variables which were held frozen.

The function `covsrt` is as follows.

```
#define SWAP(a,b) {swap=(a);(a)=(b);(b)=swap;}

void covsrt(float **covar, int ma, int ia[], int mfit)
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;
    float swap;

    for (i=mfit+1;i<=ma;i++)
        for (j=1;j<=i;j++) covar[i][j]=covar[j][i]=0.0;
    k=mfit;
    for (j=ma;j>=1;j--) {
        if (ia[j]) {
            for (i=1;i<=ma;i++) SWAP(covar[i][k],covar[i][j])
            for (i=1;i<=ma;i++) SWAP(covar[k][i],covar[j][i])
            k--;
        }
    }
}
```

Solution by Use of Singular Value Decomposition

In some applications, the normal equations are perfectly adequate for linear least-squares problems. However, in many 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? The 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 $[\alpha]$, 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 **A** (equation 15.4.4) and the vector **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 `svdcmp` and `svbksb` are designed to solve. The solution, which is given by equation (2.6.12), can be rewritten as follows: If **U** and **V** enter the SVD decomposition of **A** according to equation (2.6.1), as computed by `svdcmp`, then let the vectors $\mathbf{U}_{(i)}$ $i = 1,\dots,M$ denote the *columns* of **U** (each one a vector of length N); and let the vectors $\mathbf{V}_{(i)}; i = 1,\dots,M$ denote the *columns* of **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

$$\mathbf{a} = \sum_{i=1}^{M} \left(\frac{\mathbf{U}_{(i)}\cdot\mathbf{b}}{w_i} \right) \mathbf{V}_{(i)} \tag{15.4.17}$$

where the w_i are, as in §2.6, the singular values calculated by `svdcmp`.

Equation (15.4.17) says that the fitted parameters **a** are linear combinations of the columns of **V**, with coefficients obtained by forming dot products of the columns of **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 **V**. In fact, equation (15.4.17) can be written in a form displaying these errors as

$$\mathbf{a} = \left[\sum_{i=1}^{M} \left(\frac{\mathbf{U}_{(i)} \cdot \mathbf{b}}{w_i}\right) \mathbf{V}_{(i)}\right] \pm \frac{1}{w_1}\mathbf{V}_{(1)} \pm \cdots \pm \frac{1}{w_M}\mathbf{V}_{(M)} \qquad (15.4.18)$$

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 **a** (see §15.6).

It follows that the variance in the estimate of a parameter a_j is given by

$$\sigma^2(a_j) = \sum_{i=1}^{M} \frac{1}{w_i^2}[\mathbf{V}_{(i)}]_j^2 = \sum_{i=1}^{M} \left(\frac{V_{ji}}{w_i}\right)^2 \qquad (15.4.19)$$

whose result should be identical with (15.4.14). As before, you should not be surprised at the formula for the covariances, here given without proof,

$$\text{Cov}(a_j, a_k) = \sum_{i=1}^{M} \left(\frac{V_{ji}V_{ki}}{w_i^2}\right) \qquad (15.4.20)$$

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. (You might argue for $\sqrt{N}$, or a constant, instead of N as the multiple; that starts getting into hardware-dependent questions.)

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 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. In the following routine, the point at which this kind of editing would occur is indicated.

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 an extra array of size $N \times M$ to store the whole design matrix. This storage is overwritten by the matrix **U**. Storage is also required for the $M \times M$ matrix **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.

In the routine that follows, the matrices u,v and the vector w are input as working space. The logical dimensions of the problem are ndata data points by ma basis functions (and fitted parameters). If you care only about the values a of the fitted parameters, then u,v,w contain no useful information on output. If you want probable errors for the fitted parameters, read on.

```
#include "nrutil.h"
#define TOL 1.0e-5                        Default value for single precision and vari-
                                              ables scaled to order unity.
void svdfit(float x[], float y[], float sig[], int ndata, float a[], int ma,
    float **u, float **v, float w[], float *chisq,
    void (*funcs)(float, float [], int))
```
Given a set of data points x[1..ndata],y[1..ndata] with individual standard deviations sig[1..ndata], use χ^2 minimization to determine the coefficients a[1..ma] of the fitting function $y = \sum_i \text{a}_i \times \text{afunc}_i(x)$. Here we solve the fitting equations using singular value decomposition of the ndata by ma matrix, as in §2.6. Arrays u[1..ndata][1..ma], v[1..ma][1..ma], and w[1..ma] provide workspace on input; on output they define the singular value decomposition, and can be used to obtain the covariance matrix. The program returns values for the ma fit parameters a, and χ^2, chisq. The user supplies a routine funcs(x,afunc,ma) that returns the ma basis functions evaluated at $x =$ x in the array afunc[1..ma].
```
{
    void svbksb(float **u, float w[], float **v, int m, int n, float b[],
        float x[]);
    void svdcmp(float **a, int m, int n, float w[], float **v);
    int j,i;
    float wmax,tmp,thresh,sum,*b,*afunc;

    b=vector(1,ndata);
    afunc=vector(1,ma);
    for (i=1;i<=ndata;i++) {              Accumulate coefficients of the fitting ma-
        (*funcs)(x[i],afunc,ma);              trix.
        tmp=1.0/sig[i];
        for (j=1;j<=ma;j++) u[i][j]=afunc[j]*tmp;
        b[i]=y[i]*tmp;
    }
    svdcmp(u,ndata,ma,w,v);               Singular value decomposition.
    wmax=0.0;                             Edit the singular values, given TOL from the
    for (j=1;j<=ma;j++)                       #define statement, between here ...
        if (w[j] > wmax) wmax=w[j];
    thresh=TOL*wmax;
    for (j=1;j<=ma;j++)
        if (w[j] < thresh) w[j]=0.0;      ...and here.
    svbksb(u,w,v,ndata,ma,b,a);
    *chisq=0.0;                           Evaluate chi-square.
    for (i=1;i<=ndata;i++) {
        (*funcs)(x[i],afunc,ma);
        for (sum=0.0,j=1;j<=ma;j++) sum += a[j]*afunc[j];
        *chisq += (tmp=(y[i]-sum)/sig[i],tmp*tmp);
    }
    free_vector(afunc,1,ma);
    free_vector(b,1,ndata);
}
```

Feeding the matrix v and vector w output by the above program into the following short routine, you easily obtain variances and covariances of the fitted parameters a. The square roots of the variances are the standard deviations of the fitted parameters. The routine straightforwardly implements equation (15.4.20) above, with the convention that singular values equal to zero are recognized as having been edited out of the fit.

```
#include "nrutil.h"

void svdvar(float **v, int ma, float w[], float **cvm)
To evaluate the covariance matrix cvm[1..ma][1..ma] of the fit for ma parameters obtained
by svdfit, call this routine with matrices v[1..ma][1..ma], w[1..ma] as returned from
svdfit.
{
    int k,j,i;
    float sum,*wti;

    wti=vector(1,ma);
    for (i=1;i<=ma;i++) {
        wti[i]=0.0;
        if (w[i]) wti[i]=1.0/(w[i]*w[i]);
    }
    for (i=1;i<=ma;i++) {        Sum contributions to covariance matrix (15.4.20).
        for (j=1;j<=i;j++) {
            for (sum=0.0,k=1;k<=ma;k++) sum += v[i][k]*v[j][k]*wti[k];
            cvm[j][i]=cvm[i][j]=sum;
        }
    }
    free_vector(wti,1,ma);
}
```

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 "non-parameters," 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.

Here are two examples for user-supplied routines funcs. The first one is trivial and fits a general polynomial to a set of data:

```
void fpoly(float x, float p[], int np)
Fitting routine for a polynomial of degree np-1, with coefficients in the array p[1..np].
{
    int j;

    p[1]=1.0;
    for (j=2;j<=np;j++) p[j]=p[j-1]*x;
}
```

The second example is slightly less trivial. It is used to fit Legendre polynomials up to some order `nl-1` through a data set.

```
void fleg(float x, float pl[], int nl)
Fitting routine for an expansion with nl Legendre polynomials pl, evaluated using the recurrence
relation as in §5.5.
{
    int j;
    float twox,f2,f1,d;

    pl[1]=1.0;
    pl[2]=x;
    if (nl > 2) {
        twox=2.0*x;
        f2=x;
        d=1.0;
        for (j=3;j<=nl;j++) {
            f1=d++;
            f2 += twox;
            pl[j]=(f2*pl[j-1]-f1*pl[j-2])/d;
        }
    }
}
```

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, $X_1(\mathbf{x}), \dots, X_M(\mathbf{x})$. The χ^2 merit function is now

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(\mathbf{x}_i)}{\sigma_i} \right]^2 \tag{15.4.24}$$

All of the preceding discussion goes through unchanged, with x replaced by $\mathbf{x}$. In fact, if you are willing to tolerate a bit of programming hack, you can use the above programs without any modification: In both `lfit` and `svdfit`, the only use made of the array elements `x[i]` is that each element is in turn passed to the user-supplied routine `funcs`, which duly gives back the values of the basis functions at that point. If you set `x[i]=i` before calling `lfit` or `svdfit`, and independently provide `funcs` with the true vector values of your data points (e.g., in global variables), then `funcs` can translate from the fictitious `x[i]`'s to the actual data points before doing its work.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapters 8–9.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

15.5 Nonlinear Models

We now consider fitting when the model depends *nonlinearly* on the set of M unknown parameters $a_k, k = 1, 2, \dots, M$. 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

$$\chi^2(\mathbf{a}) \approx \gamma - \mathbf{d} \cdot \mathbf{a} + \frac{1}{2}\mathbf{a} \cdot \mathbf{D} \cdot \mathbf{a} \qquad (15.5.1)$$

where $\mathbf{d}$ is an M-vector and $\mathbf{D}$ is an $M \times M$ matrix. (Compare equation 10.6.1.) If the approximation is a good one, we know how to jump from the current trial parameters $\mathbf{a}_{\rm cur}$ to the minimizing ones $\mathbf{a}_{\rm min}$ in a single leap, namely

$$\mathbf{a}_{\rm min} = \mathbf{a}_{\rm cur} + \mathbf{D}^{-1} \cdot \left[-\nabla\chi^2(\mathbf{a}_{\rm cur})\right] \qquad (15.5.2)$$

(Compare equation 10.7.4.)

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 $\mathbf{a}_{\rm cur}$. In that case, about all we can do is take a step down the gradient, as in the steepest descent method (§10.6). In other words,

$$\mathbf{a}_{\rm next} = \mathbf{a}_{\rm cur} - \text{constant} \times \nabla\chi^2(\mathbf{a}_{\rm cur}) \qquad (15.5.3)$$

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 10.7 and 10.6 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.

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

$$\chi^2(\mathbf{a}) = \sum_{i=1}^{N} \left[\frac{y_i - y(x_i; \mathbf{a})}{\sigma_i} \right]^2 \tag{15.5.5}$$

The gradient of χ^2 with respect to the parameters $\mathbf{a}$, which will be zero at the χ^2 minimum, has components

$$\frac{\partial \chi^2}{\partial a_k} = -2 \sum_{i=1}^{N} \frac{[y_i - y(x_i; \mathbf{a})]}{\sigma_i^2} \frac{\partial y(x_i; \mathbf{a})}{\partial a_k} \qquad k = 1, 2, \dots, M \tag{15.5.6}$$

Taking an additional partial derivative gives

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i; \mathbf{a})}{\partial a_k} \frac{\partial y(x_i; \mathbf{a})}{\partial a_l} - [y_i - y(x_i; \mathbf{a})] \frac{\partial^2 y(x_i; \mathbf{a})}{\partial a_l \partial a_k} \right] \tag{15.5.7}$$

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 $[\alpha] = \frac{1}{2}\mathbf{D}$ in equation (15.5.2), in terms of which that equation can be rewritten as the set of linear equations

$$\sum_{l=1}^{M} \alpha_{kl} \, \delta a_l = \beta_k \tag{15.5.9}$$

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 $[\alpha]$, 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 $\partial y/\partial a_k$, 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

$$\alpha_{kl} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i; \mathbf{a})}{\partial a_k} \frac{\partial y(x_i; \mathbf{a})}{\partial a_l} \right] \tag{15.5.11}$$

This expression more closely resembles its linear cousin (15.4.8). You should understand that minor (or even major) fiddling with $[\alpha]$ 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 $[\alpha]$ is defined.

Levenberg-Marquardt Method

Marquardt [1] has 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 *Marquardt method*) works very well in practice and has become the standard of nonlinear least-squares routines.

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 $[\alpha]$ 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

$$\sum_{l=1}^{M} \alpha'_{kl}\, \delta a_l = \beta_k \tag{15.5.14}$$

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 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 which is then rejected, the value of λ being then increased. For sufficiently large λ the matrix $[\alpha']$ 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 un-steep degenerate valleys.

These considerations suggest that, in practice, one might as well stop iterating on the first or second occasion that χ^2 decreases by a negligible amount, say either less than 0.01 absolutely or (in case roundoff prevents that being reached) some fractional amount like 10^{-3}. Don't stop after a step where χ^2 *increases*: 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

$$[C] \equiv [\alpha]^{-1} \tag{15.5.15}$$

which, as before, is the estimated covariance matrix of the standard errors in the fitted parameters **a** (see next section).

The following pair of functions encodes Marquardt's method for nonlinear parameter estimation. Much of the organization matches that used in `lfit` of §15.4. In particular the array `ia[1..ma]` must be input with components one or zero corresponding to whether the respective parameter values `a[1..ma]` are to be fitted for or held fixed at their input values, respectively.

The routine `mrqmin` performs one iteration of Marquardt's method. It is first called (once) with `alamda` < 0, which signals the routine to initialize. `alamda` is set on the first and all subsequent calls to the suggested value of λ for the next iteration; `a` and `chisq` are always given back as the best parameters found so far and their χ^2. When convergence is deemed satisfactory, set `alamda` to zero before a final call. The matrices `alpha` and `covar` (which were used as workspace in all previous calls) will then be set to the curvature and covariance matrices for the converged parameter values. The arguments `alpha`, `a`, and `chisq` must not be modified between calls, nor should `alamda` be, except to set it to zero for the final call. When an uphill step is taken, `chisq` and `a` are given back with their input (best) values, but `alamda` is set to an increased value.

The routine `mrqmin` calls the routine `mrqcof` for the computation of the matrix $[\alpha]$ (equation 15.5.11) and vector β (equations 15.5.6 and 15.5.8). In turn `mrqcof` calls the user-supplied routine `funcs(x,a,y,dyda)`, which for input values `x` $\equiv x_i$ and `a` $\equiv$ **a** calculates the model function `y` $\equiv y(x_i; \mathbf{a})$ and the vector of derivatives `dyda` $\equiv \partial y/\partial a_k$.

```
#include "nrutil.h"

void mrqmin(float x[], float y[], float sig[], int ndata, float a[], int ia[],
    int ma, float **covar, float **alpha, float *chisq,
    void (*funcs)(float, float [], float *, float [], int), float *alamda)
```
Levenberg-Marquardt method, attempting to reduce the value χ^2 of a fit between a set of data points `x[1..ndata]`, `y[1..ndata]` with individual standard deviations `sig[1..ndata]`, and a nonlinear function dependent on `ma` coefficients `a[1..ma]`. The input array `ia[1..ma]` indicates by nonzero entries those components of `a` that should be fitted for, and by zero entries those components that should be held fixed at their input values. The program returns current best-fit values for the parameters `a[1..ma]`, and χ^2 = `chisq`. The arrays `covar[1..ma][1..ma]`, `alpha[1..ma][1..ma]` are used as working space during most iterations. Supply a routine `funcs(x,a,yfit,dyda,ma)` that evaluates the fitting function `yfit`, and its derivatives `dyda[1..ma]` with respect to the fitting parameters `a` at `x`. On the first call provide an initial guess for the parameters `a`, and set `alamda<0` for initialization (which then sets `alamda=.001`). If a step succeeds `chisq` becomes smaller and `alamda` decreases by a factor of 10. If a step fails `alamda` grows by a factor of 10. You must call this

routine repeatedly until convergence is achieved. Then, make one final call with alamda=0, so that covar[1..ma][1..ma] returns the covariance matrix, and alpha the curvature matrix. (Parameters held fixed will return zero covariances.)

```
{
    void covsrt(float **covar, int ma, int ia[], int mfit);
    void gaussj(float **a, int n, float **b, int m);
    void mrqcof(float x[], float y[], float sig[], int ndata, float a[],
        int ia[], int ma, float **alpha, float beta[], float *chisq,
        void (*funcs)(float, float [], float *, float [], int));
    int j,k,l;
    static int mfit;
    static float ochisq,*atry,*beta,*da,**oneda;

    if (*alamda < 0.0) {                    Initialization.
        atry=vector(1,ma);
        beta=vector(1,ma);
        da=vector(1,ma);
        for (mfit=0,j=1;j<=ma;j++)
            if (ia[j]) mfit++;
        oneda=matrix(1,mfit,1,1);
        *alamda=0.001;
        mrqcof(x,y,sig,ndata,a,ia,ma,alpha,beta,chisq,funcs);
        ochisq=(*chisq);
        for (j=1;j<=ma;j++) atry[j]=a[j];
    }
    for (j=1;j<=mfit;j++) {                 Alter linearized fitting matrix, by augmenting di-
        for (k=1;k<=mfit;k++) covar[j][k]=alpha[j][k];      agonal elements.
        covar[j][j]=alpha[j][j]*(1.0+(*alamda));
        oneda[j][1]=beta[j];
    }
    gaussj(covar,mfit,oneda,1);             Matrix solution.
    for (j=1;j<=mfit;j++) da[j]=oneda[j][1];
    if (*alamda == 0.0) {                   Once converged, evaluate covariance matrix.
        covsrt(covar,ma,ia,mfit);
        covsrt(alpha,ma,ia,mfit);           Spread out alpha to its full size too.
        free_matrix(oneda,1,mfit,1,1);
        free_vector(da,1,ma);
        free_vector(beta,1,ma);
        free_vector(atry,1,ma);
        return;
    }
    for (j=0,l=1;l<=ma;l++)                 Did the trial succeed?
        if (ia[l]) atry[l]=a[l]+da[++j];
    mrqcof(x,y,sig,ndata,atry,ia,ma,covar,da,chisq,funcs);
    if (*chisq < ochisq) {                  Success, accept the new solution.
        *alamda *= 0.1;
        ochisq=(*chisq);
        for (j=1;j<=mfit;j++) {
            for (k=1;k<=mfit;k++) alpha[j][k]=covar[j][k];
            beta[j]=da[j];
        }
        for (l=1;l<=ma;l++) a[l]=atry[l];
    } else {                                Failure, increase alamda and return.
        *alamda *= 10.0;
        *chisq=ochisq;
    }
}
```

Notice the use of the routine covsrt from §15.4. This is merely for rearranging the covariance matrix covar into the order of all ma parameters. The above routine also makes use of

```
#include "nrutil.h"

void mrqcof(float x[], float y[], float sig[], int ndata, float a[], int ia[],
    int ma, float **alpha, float beta[], float *chisq,
    void (*funcs)(float, float [], float *, float [], int))
```
Used by `mrqmin` to evaluate the linearized fitting matrix `alpha`, and vector `beta` as in (15.5.8), and calculate χ^2.
```
{
    int i,j,k,l,m,mfit=0;
    float ymod,wt,sig2i,dy,*dyda;

    dyda=vector(1,ma);
    for (j=1;j<=ma;j++)
        if (ia[j]) mfit++;
    for (j=1;j<=mfit;j++) {             Initialize (symmetric) alpha, beta.
        for (k=1;k<=j;k++) alpha[j][k]=0.0;
        beta[j]=0.0;
    }
    *chisq=0.0;
    for (i=1;i<=ndata;i++) {            Summation loop over all data.
        (*funcs)(x[i],a,&ymod,dyda,ma);
        sig2i=1.0/(sig[i]*sig[i]);
        dy=y[i]-ymod;
        for (j=0,l=1;l<=ma;l++) {
            if (ia[l]) {
                wt=dyda[l]*sig2i;
                for (j++,k=0,m=1;m<=l;m++)
                    if (ia[m]) alpha[j][++k] += wt*dyda[m];
                beta[j] += dy*wt;
            }
        }
        *chisq += dy*dy*sig2i;          And find χ².
    }
    for (j=2;j<=mfit;j++)               Fill in the symmetric side.
        for (k=1;k<j;k++) alpha[k][j]=alpha[j][k];
    free_vector(dyda,1,ma);
}
```

Example

The following function `fgauss` is an example of a user-supplied function `funcs`. Used with the above routine `mrqmin` (in turn using `mrqcof`, `covsrt`, and `gaussj`), it fits for the model

$$y(x) = \sum_{k=1}^{K} B_k \exp\left[-\left(\frac{x - E_k}{G_k}\right)^2\right] \tag{15.5.16}$$

which is a sum of K Gaussians, each having a variable position, amplitude, and width. We store the parameters in the order $B_1, E_1, G_1, B_2, E_2, G_2, \ldots, B_K, E_K, G_K$.

```
#include <math.h>

void fgauss(float x, float a[], float *y, float dyda[], int na)
```
$y(x;a)$ 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[i] $= B_k$, a[i+1] $= E_k$, a[i+2] $= G_k$, $k = 1, ..., $na/3. The dimensions of the arrays are a[1..na], dyda[1..na].
```
{
    int i;
    float fac,ex,arg;

    *y=0.0;
    for (i=1;i<=na-1;i+=3) {
        arg=(x-a[i+1])/a[i+2];
        ex=exp(-arg*arg);
        fac=a[i]*ex*2.0*arg;
        *y += a[i]*ex;
        dyda[i]=ex;
        dyda[i+1]=fac/a[i+2];
        dyda[i+2]=fac*arg/a[i+2];
    }
}
```

More Advanced Methods for Nonlinear Least Squares

The Levenberg-Marquardt algorithm can be implemented 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 [3] can be found in MINPACK [4]. Another algorithm for nonlinear least-squares keeps the second-derivative term we dropped in the Levenberg-Marquardt method whenever it would be better to do so. These methods are called "full Newton-type" methods and are reputed to be more robust than Levenberg-Marquardt, but more complex. One implementation is the code NL2SOL [5].

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 11.

Marquardt, D.W. 1963, *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, pp. 431–441. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.2 (by J.E. Dennis).

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Moré, J.J. 1977, in *Numerical Analysis*, Lecture Notes in Mathematics, vol. 630, G.A. Watson, ed. (Berlin: Springer-Verlag), pp. 105–116. [3]

Moré, J.J., Garbow, B.S., and Hillstrom, K.E. 1980, *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74. [4]

Dennis, J.E., Gay, D.M, and Welsch, R.E. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 348–368; *op. cit.*, pp. 369–383. [5].