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 $\mathbf{a}_{\rm cur}$ to the minimizing ones $\mathbf{a}_{\rm min}$ in a single leap, namely

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

(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 $\mathbf{a}_{\rm cur}$. 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,

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

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.

Numerical Recipes: The Art of Scientific Computing, Third Edition, by W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Version 3.04 (2011). Text is Copyright © 1988-2007 by Cambridge University Press. Computer source code is Copyright © 1987-2007 by Numerical Recipes Software. Hardcover book ISBN 978-0-521-88068-8 is published by Cambridge University Press, http://www.cambridge.org. This electronic edition is published by Numerical Recipes Software, http://numerical.recipes. Permission is granted for authorized subscribers to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any public server computer, is strictly prohibited.

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

$$\frac{\partial \chi^2}{\partial a_k} = -2 \sum_{i=0}^{N-1} \frac{[y_i - y(x_i|\mathbf{a})]}{\sigma_i^2} \frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \qquad k = 0, 1, \ldots, M-1 \tag{15.5.6}$$

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 $\mathbf{a}$, 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 $\mathbf{a}$. 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 $\mathbf{a}$ (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 χ^2.
        }
        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 $B_0, E_0, G_0, B_1, E_1, G_1, \ldots, B_{K-1}, E_{K-1}, G_{K-1}$.

fit_examples.h

```
void fgauss(const Doub x, VecDoub_I &a, Doub &y, VecDoub_O &dyda) {
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,...,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]