Computers & Geosciences Vol. 14, No. 4, pp. 489–503, 1988
Printed in Great Britain.

0098-3004/88 $3.00 + 0.00

ITERATIVE METHOD AND FORTRAN CODE FOR NONLINEAR CURVE FITTING

MARIJKE VAN HEESWIJK and CHRISTOPHER G. FOX
NOAA/MRRD, Hatfield Marine Science Center, Newport, OR 97365, U.S.A.

(*Received* 19 *June* 1987; *revised* 7 *December* 1987)

Abstract—A FORTRAN subroutine is presented which allows the iterative, least-squares fitting of discrete, observational data by a nonlinear function. The function may have any number of parameters, provided that partial derivatives with respect to each of the parameters can be calculated. A brief discussion of the mathematical theory is presented, instructions for use of the subroutine are provided, and examples and performance tests of the algorithm are discussed.

Key Words: FORTRAN, Nonlinear curve fitting, Iterative least squares, Regression.

INTRODUCTION

Fitting mathematical functions to observational data is a fundamental statistical technique for building numerical models of natural systems. The method of least-squares provides an unbiased estimation of the data structure for linear functions and some other specialized functional forms. Usually, these special forms are not adequate to describe the observational data, and the fitting of more complex, nonlinear functions is required.

One method for fitting nonlinear functions consists of applying an appropriate transformation to the data to linearize its form, and then determining the least-squares solution for the transformed data. After calculating the regression coefficients, the relationship can be rewritten in its original form. Because the linear least-squares method minimizes the error residuals of the dependent variable, the transformation of the data results in the distortion of the error field associated with the measured variable. In a few situations, for example the fitting of the Fourier spectrum, such a transformation can tend to linearize the error field (Fox, 1985, Appendix A). For most data sets, however, the measurement error is constant over the data set, and transformation results in a biased least-squares fit.

The technique described in this paper, allows any differentiable function to be fitted to data without the use of transformations of the variables and the resulting distortion of the error field. The method minimizes the linear estimator residuals using an iterative, least-squares technique. Mathematical functions of any form can be applied, provided the partial derivatives with respect to each of the coefficients can be determined. The mathematical basis of the technique is reviewed. A FORTRAN subroutine and complete user instructions are provided in the Appendix, and examples of the method are illustrated.

NOTATION

In reviewing the iterative method for nonlinear curve fitting, the notation used by Menke (1984) will be followed. In this notation, the matrix equation $\mathbf{d} = \mathbf{Gm}$ relates N data points to N mathematical expressions. Data vector $\mathbf{d}$ is represented by $\mathbf{d} = (d_1, d_2, \ldots, d_N)$, the model parameter vector by $\mathbf{m} = (m_1, m_2, \ldots, m_M)$, and data kernel matrix $\mathbf{G}$ by $\mathbf{G} = g_{ij}$, where $i = 1, 2, \ldots, N$, and $j = 1, 2, \ldots, M$. If the relationship between data point d_i and the corresponding function is $d_i = m_1 + m_2 x_i$ for example, then vector $\mathbf{d} = (d_1, d_2, \ldots, d_N)$, $\mathbf{m} = (m_1, m_2)$, and matrix

$$\mathbf{G} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ . & . \\ . & . \\ 1 & x_N \end{pmatrix}.$$

MATHEMATICS

Two methods of solving nonlinear problems are reviewed by example. One of those methods is used in the FORTRAN subroutine. The example used is a function of the power law form which can be written as

$$d = m_1 x^{m_2} \tag{1}$$

where d is the dependent variable, x the independent variable, and m_1 and m_2 unknown parameters of the function. The first method by which this equation can be solved is by linearizing the expression and applying the least-squares inversion method to determine m_1 and m_2. One way linearity can be achieved is by transforming the equation by taking the logarithm of both sides of the expression to yield

$$\log(d) = \log(m_1) + m_2 \log(x). \tag{2}$$

Parameters m_1 and m_2 then can be derived in log–log space using the least-squares inversion.

The difficulty with this approach is that transformation of the nonlinear function into log–log space in order to solve for the unknowns changes the error distribution of the data. If the data to be fitted by the nonlinear expression have a Gaussian distribution of error, their log transformed equivalents will have a nonGaussian distribution of error. Because least-squares inversions should only be applied to data with a Gaussian error distribution, results of a least-squares inversion on log-transformed data are biased.

The second method by which the equation can be solved in linear space is by expanding Equation (1) about parameter estimates using a Taylor series expansion. The unknown parameters then can be solved iteratively, using the least-squares inversion technique to improve parameter estimates after each iteration. The method is not new and was described succinctly by Scarborough (1930). The technique also is described in detail by Draper and Smith (1966) and is reviewed briefly here.

In general, the least-squares inversion technique determines model parameters that minimize the sum of the squares of the residuals $\Sigma_{i=1}^{N} \varepsilon_i^2$. The residual ε_i at each observed data point d_i^{obs} equals $\varepsilon_i = d_i^{\text{obs}} - d_i^{\text{pre}}$, where d_i^{pre} is the predicted value of data point i, using the best estimates for the model parameters.

If the best estimate for the unknown parameters is

$$m_{k,l+1}^{\text{est}} = m_{k,l}^{\text{est}} + \Delta m_{k,l+1}, \qquad k = 1, 2, \ldots; \quad l = 0, 1, 2, \ldots \tag{3}$$

where $\Delta m_{k,l+1}$ is the parameter increment [the difference between the $(l + 1)$ estimate and the (l) estimate], then Equation (1) can be written as

$$d_i^{\text{obs}} = d_i^{\text{pre}} + \varepsilon_i \tag{4}$$

where

$$d_i^{\text{pre}} = f(x_i, m_{1,l}^{\text{est}}, m_{2,l}^{\text{est}}) + \Delta m_{1,l+1}(\partial f/\partial m_{1,l}^{\text{est}}) + \Delta m_{2,l+1}(\partial f/\partial m_{2,l}^{\text{est}}) + \text{higher order terms.} \tag{5}$$

Expression (5) is the Taylor series expansion of d_i^{pre} about the estimated parameter value $m_{k,l}^{\text{est}}$. Ignoring the higher order terms, Equations (4) and (5) can be combined and rewritten as

$$d_i^{\text{obs}} - f(x_i, m_{1,l}^{\text{est}}, m_{2,l}^{\text{est}}) \approx \Delta m_{1,l+1}(\partial f_i/\partial m_{1,l}^{\text{est}}) + \Delta m_{2,l+1}(\partial f_i/\partial m_{2,l}^{\text{est}}) + \varepsilon_i. \tag{6}$$

Equation (6) is linear in parameter increments $\Delta m_{i,l+1}$, which can be calculated by least-squares inversion. In matrix notation, this indicates the system of equations $\mathbf{d} = \mathbf{Gm}$ can be solved for vector $\mathbf{m}$ by minimizing the sum of the squares of the residuals $\Sigma_{i=1}^{N} \varepsilon_i^2$. In this format, vector $\mathbf{d}$ contains the observed data points minus the zeroth order Taylor series term. Matrix $\mathbf{G}$ contains the partial derivatives with respect to each of the parameter estimates, and vector $\mathbf{m}$ contains the unknown parameter increments $\Delta m_{1,l+1}$ and $\Delta m_{2,l+1}$.

Expression (6) can be solved for $\Delta m_{i,l+1}$ by the least-squares method if an initial estimate for parameter $m_{k,l}^{\text{est}}$ is supplied for the first iteration. At the completion of each iteration, $m_{k,l}^{\text{est}}$ is updated according to Equation (3) and used as a new initial estimate in the following iteration. The iterations continue until parameter increment $\Delta m_{k,l+1}$ reaches a selected small value. At this point, best estimate $m_{k,l}^{\text{est}}$ has reached the true parameters m_k within the error allowed by the user. Convergence to a solution does not occur always, however. If, after a particular iteration, the sum of the squares of the residuals does not decrease, parameter increment $\Delta m_{i,l+1}$ is halved to force convergence. Bisecting the parameter increment can be repeated any number of times during an iteration. Even this technique, however, does not ensure convergence.

How well the iterative method fits a data set depends on how appropriate a function is selected and the closeness of the initial estimates to the true parameters. Because a solution to a nonlinear problem is sought, it is possible for the solution to converge to a local minimum or maximum instead of an absolute minimum. If the initial estimates are close to the true parameters, however, it is more likely the solution will converge on the absolute minimum. Once a solution to the nonlinear problem has been determined, it is not known if the solution is the result of convergence to an absolute minimum or not. The only possible check is to fit the function using many different initial estimates and visually inspect the possibly different fits.

The iterative technique is selected for nonlinear curve fitting and used in subroutine CURVEFIT (see Appendix). The technique does not distort the error field, as is the situation for the transformation method described. Even though the error field is not distorted, it is not possible to give confidence limits or a goodness-of-fit estimate for the fitted curve, because the error distribution of the unknown parameters is not known.

DISCUSSION OF SUBROUTINE CURVEFIT

Subroutine CURVEFIT (see Appendix) allows the user to fit any differentiable function to a data set, applying the theory explained in the previous section. To use subroutine CURVEFIT, the user first must inspect the data set and select an appropriate function to represent the data. Whenever a new functional model is used, subroutine CURVEFIT must be adapted for the selected function. Careful attention has to be given to the parts of the subroutine that are flagged by the word "NOTE". Those notes help the user make changes at the correct locations in the routine.

Partial derivatives must be calculated with respect to each of the unknown parameters for the selected function. The function itself is entered into FUNCTION F0 and inserted at the end of subroutine CURVEFIT. Partial derivatives are inserted in order into FUNCTION F1, FUNCTION F2, etc. It is necessary for the user to keep track of the order of insertion, because elements in arrays PARAM and MIN must be in the same order as the partial derivatives in the functions. Array PARAM contains initial estimates of the unknown parameters when subroutine CURVEFIT is called first and returns the final estimates when the iterations are terminated successfully. Array MIN contains the iteration cut-off values for each of the parameters in array PARAM. The routine is configured currently for functions of up to eight unknowns. The subroutine can be expanded, however, to calculate an unlimited number of parameters. The locations at which these modifications are necessary are noted.

To prevent division by zero during program execution, the user must supply a value for variable TEST. When values become smaller than TEST an error message occurs. Depending on the type of nonlinear function selected, some variables or parts of expressions may need to be checked for exponents larger than the computer can store. If such a situation arises, the user can check the size of exponents against variable IEX, which is user supplied.

Subroutine CURVEFIT calls three subroutines, each of which are listed after the main subroutine. Subroutines GJINV and GAUSS are from Menke (1984), and have not been altered. Subroutine GJINV inverts a square matrix, whereas subroutine GAUSS solves a matrix equation. A third subroutine, TESTVAL, checks for zero divides and unreasonably large exponents that may possibly occur at three different locations in subroutine CURVEFIT. Depending on what type of function the user selects, subroutine TESTVAL might not be needed at all, or only a zero divide or an exponent may need to be checked.

The user should be aware of possible problems that might be encountered when utilizing subroutine CURVEFIT. If the initial estimates for the unknown coefficients are too far from the true values, subroutine CURVEFIT may be unable to converge to a solution. The inability to converge will occur in the form of a matrix inversion error, residual values that exceed storage limits, or variables with exponents exceeding IEX. If any of these error conditions occur during program execution, the problem may be solved by selecting better initial estimates for the parameters or, in some situations, selecting larger iteration cut-off values. If the user needs to fit curves to a large number of data sets, it is advisable to write a subroutine in which initial estimates for each data set are determined in some automated fashion. Usually, the initial estimates are derived by using the least-squares method on appropriately transformed data.

Subroutine CURVEFIT is written in single precison, except for variables containing residuals, which are in double precision. The increased precision is required to allow residuals with large exponents to be calculated without causing an overflow problem during program execution. If subroutine CURVEFIT is used for data sets where the independent and dependent variables or the parameters are large, variables may obtain values which exceed single precision storage limits. If this condition occurs, subroutines CURVEFIT, TESTVAL, GAUSS, and GJINV must be changed to double precision. Depending on the type of computer used, the subroutines may need to be compiled with a special option, which specifies that double precision variables are allowed to have large exponents at the expense of an increase in precision. For a VAX/VMS operating system this option is the G_FLOATING option.

EXAMPLES

Two examples are presented to demonstrate the use of subroutine CURVEFIT. In each example, iterative fits are made to synthetic data sets, generated by adding normally distributed random noise of a given amplitude to a known function. By comparing the fitted result to the known underlying function, the performance of the algorithm can be evaluated over differing signal to noise ratios. In addition, the sensitivity of the result to the selection of initial parameter estimates also can be examined.

The first example fits an unnormalized, Gaussian-form exponential function:

$$f(x) = m_1 \exp(-(x - m_2)^2/2m_3). \quad (7)$$

The partial derivatives for this equation are:

$$\partial f/\partial m_1 = \exp(-(x - m_2)^2/2m_3) \quad (8)$$

$$\partial f/\partial m_2 = (m_1(x - m_2)/m_3) \exp(-(x - m_2)^2/2m_3) \quad (9)$$

and

$$\partial f/\partial m_3 = (m_1(x - m_2)^2/2m_3) \exp(-(x - m_2)^2/2m_3). \quad (10)$$

The FORTRAN program illustrated in the Appendix uses this first example. The synthetic data illustrated in Figure 1A represent the sum of Function (7) (with $m_1 = 100$, $m_2 = 250$, and $m_3 = 8000$) and random noise of distribution $N(0, 100)$. For the data of Figure 1C, the variance of the noise has been increased to $N(0, 400)$, or twice the standard deviation.

Subroutine CURVEFIT is used to determine the curves that best fit each of the synthetic data sets. The parameters of the underlying curve were used as initial estimates. Obviously using these initial estimates should give an optimal fit. In practice, the user would be unaware of the parameters of the true function. Figure 1B shows how closely the fitted curve approximates the true curve when fitted to the data in

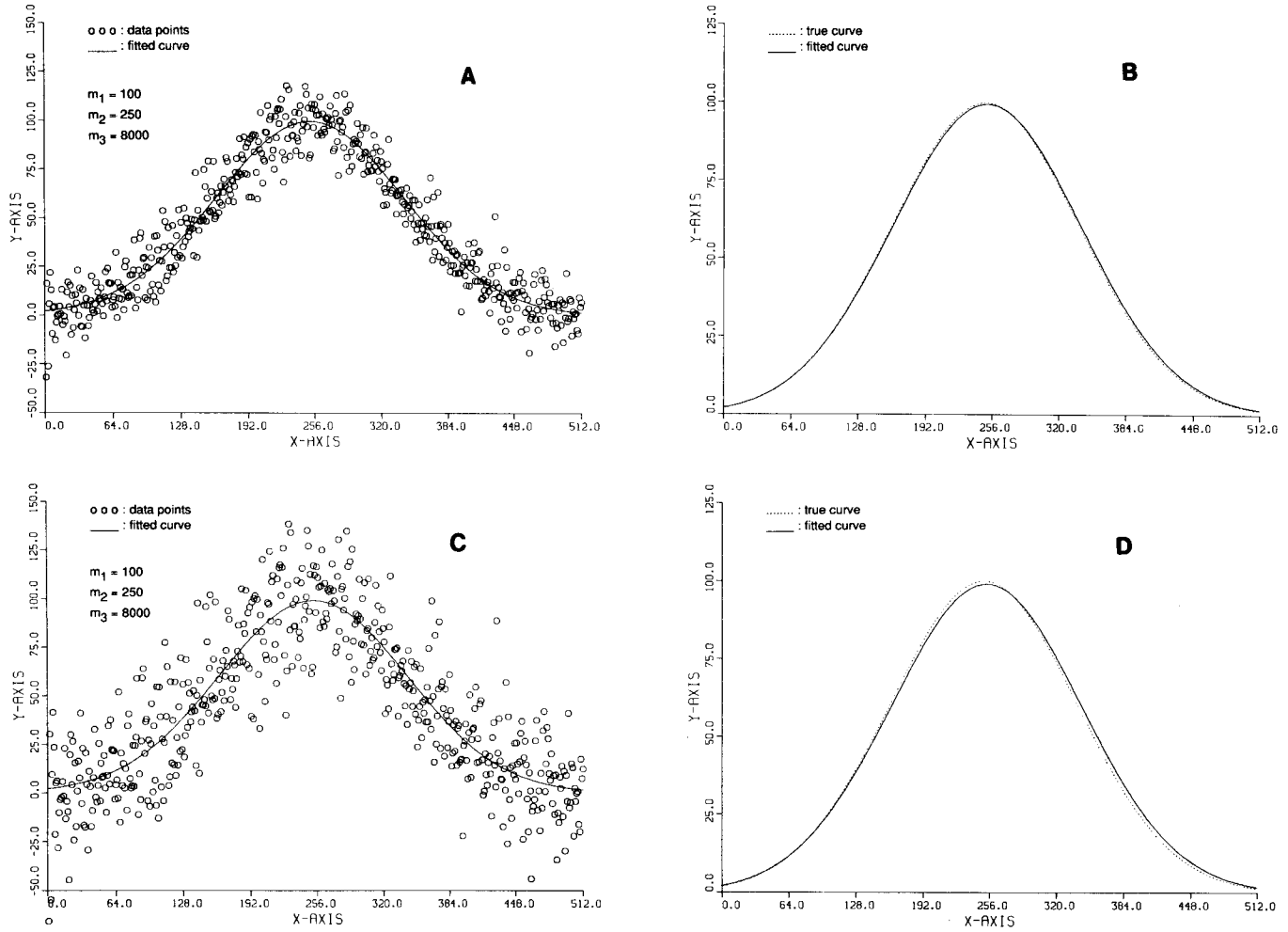

Figure 1. Synthetic data sets and fitted curves for function $f(x) = m_1 \exp[-(x - m_2)^2/2m_3]$. A—Synthetic data set generated by adding noise with distribution $N(0, 100)$ to true curve with parameters $m_1 = 100$, $m_2 = 250$, and $m_3 = 8000$. Curve fitted to data is shown. Initial estimates used to obtain this curve are those of true curve. Iteration cut-off values are 0.1 for all parameters, IEX = 25, and TEST = 0.0000001. B—Comparison of true curve and curve fitted to data in Figure 1A. C—Synthetic data set generated by adding noise with distribution $N(0, 400)$ to true curve with parameters $m_1 = 100$, $m_2 = 250$, and $m_3 = 8000$. Curve fitted to data is shown. Initial estimates used to obtain this curve are those of true curve. Iteration cut-off values are 0.1 for all parameters, IEX = 25, and TEST = 0.0000001. D—Comparison of true curve and curve fitted to data in Figure in 1C.

Table 1. Listing of curve fitting results to data displayed in Figures 1 and 2. Note how results deteriorate for higher noise levels

Figure #	Noise level	Iteration cut-off	Initial estimate			Final Estimate			True Parameter		
			m_1	m_2	m_3	m_1	m_2	m_3	m_1	m_2	m_3
1a	N(0, 100)	0.1	100	250	8000	99.5	251.3	8137.4	100	250	8000
1c	N(0, 400)	0.1	100	250	8000	99.0	252.6	8276.0	100	250	8000
2a	N(0, 640000)	0.05	0	300	0.2	-74.1	302.1	0.19	0	300	0.2
2c	N(0, 2560000)	0.05	0	300	0.2	-311.6	300.5	0.19	0	300	0.2

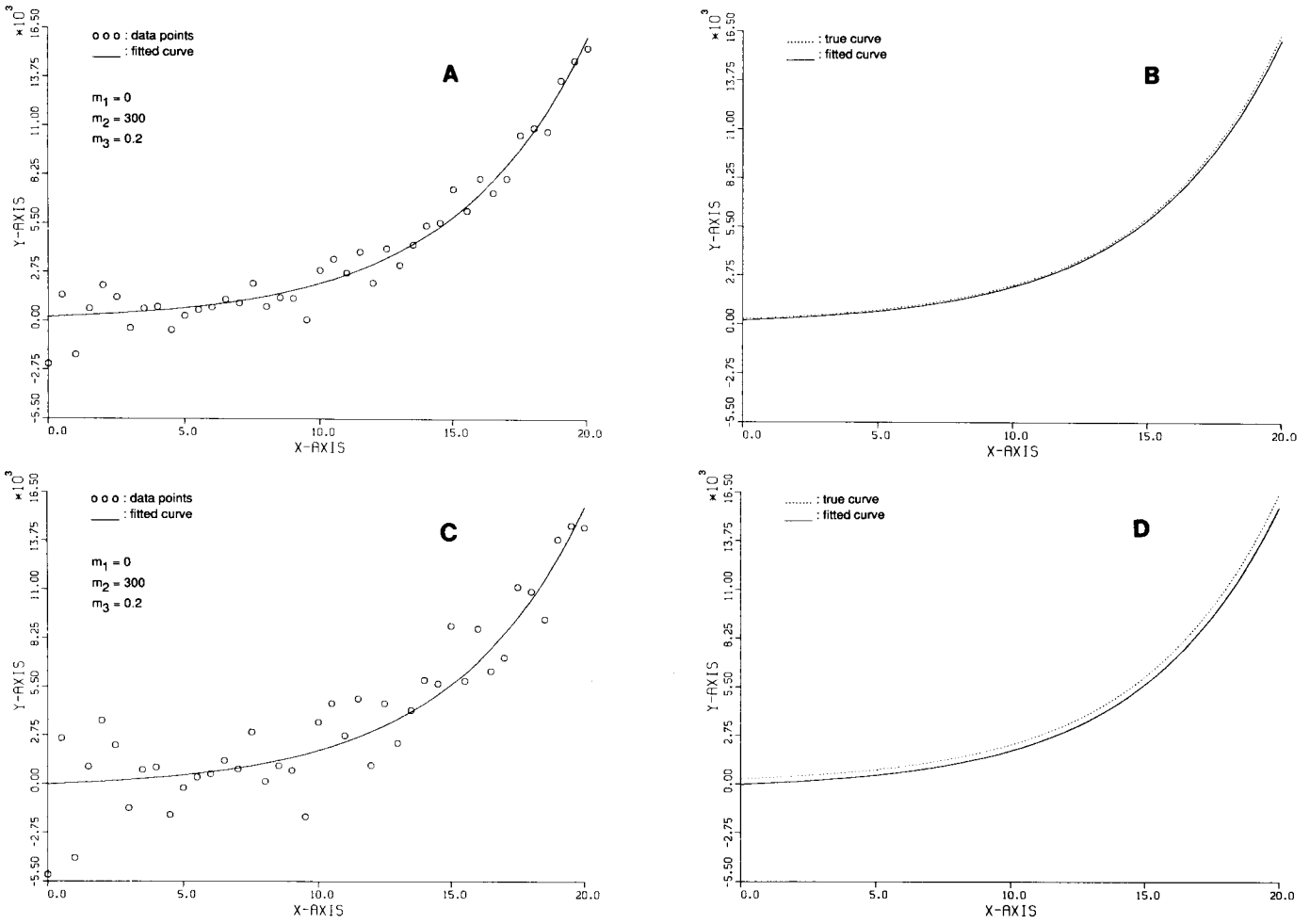

Figure 2. Synthetic data sets and fitted curves for function $f(x) = m_1 + m_2 \exp(m_3 x)$. A—Synthetic data set generated by adding noise with distribution $N(0, 640000)$ to true curve with parameters $m_1 = 0$, $m_2 = 300$, and $m_3 = 0.2$. Curve fitted to data is shown. Initial estimates used to obtain this curve are those of true curve. Iteration cut-off values are 0.05 for all parameters, IEX = 33, and TEST = 0.0000001. B—Comparison of true curve and curve fitted to data in Figure 2A. C—Synthetic data set generated by adding noise with distribution $N(0, 2560000)$ to true curve with parameters $m_1 = 0$, $m_2 = 300$, and $m_3 = 0.2$. Curve fitted to data is shown. Initial estimates used to obtain this curve are those of true curve. Iteration cut-off values are 0.05 for all parameters, IEX = 33, and TEST = 0.0000001. D—Comparison of true curve and curve fitted to data in Figure 2C.

Figure 1A. Figure 1D shows the same comparison for the data in Figure 1C. The higher noise level and thus lower signal to noise ratio of the latter data set does cause a degradation of the fitted curve. Both fits, however, are good. Table 1 lists the final parameter estimates to which each of the fitted curves converged.

The iteration cut-off values used for curve fitting in Figure 1 are 0.1 for all three parameters. Variable IEX is set to 25 and TEST equal to 0.0000001. To obtain a feeling for the sensitivity to initial estimates for Function (7), the curve fitting routine also was used with extreme initial estimates. It was determined that for the data in Figure 1A initial estimates ranging between 44 and 153% of the true parameters converged to the underlying curve displayed in Figure 1B. For Figure 1C initial estimates ranging between 41 and 154% of the true parameters converged to the underlying curve of Figure 1D. Initial estimate extremes that lead to convergence will differ with signal to noise ratios and the form of the underlying function. Function (7) seems relatively insensitive to initial estimates, as opposed to the next example, which is highly sensitive to initial estimates.

The second example (not shown in Appendix) is a standard exponential function of the form

$$f(x) = m_1 + m_2 \exp(m_3 x). \tag{11}$$

The partial derivatives for this equation are:

$$\partial f/\partial m_1 = 1 \tag{12}$$

$$\partial f/\partial m_2 = \exp(m_3 x) \tag{13}$$

and

$$\partial f/\partial m_3 = m_2 \exp(m_3 x). \tag{14}$$

As in the first example, two synthetic data sets are generated, and displayed in Figures 2A and 2C. The true curve to which random noise has been added has parameters $m_1 = 0$, $m_2 = 300$, and $m_3 = 0.2$. The noise level in Figure 2A is $N(0, 640000)$ and $N(0, 2560000)$ for Figure 2C. The data in Figures 2A and 2C are fitted using subroutine CURVEFIT, and as for the first example, the parameters of the true curve are used as initial estimates. Again the computed curve closely approximates the underlying function, with some degradation of the results with the higher noise level. Table 1 lists the final parameter estimates to which each of the fitted curves of Figure 2 converged.

The iteration cut-off values used for curve fitting in Figure 2 are 0.05 for all three parameters. Variable IEX is set to 33 and TEST equal to 0.0000001. To obtain a feeling for the sensitivity to initial estimates for Function (11), the routine again was used with extreme initial estimates. The second example proves extremely sensitive to the initial estimate of the third parameter m_3. Only combinations of initial estimates where the third parameter estimate is within 5% of the true value converge successfully. Different successful initial estimate combinations for the second example converge to close, but different final estimates. This was not the situation for example one, for which a wide range of initial estimates converged to identical final estimates. Function (11) is apparently more sensitive to the initial estimates than Function (7). In practice this indicates that depending on the nonlinear function selected, the initial estimates may need to be close to the actual parameters in order to converge on the correct solution.

SUMMARY

A technique has been presented for fitting differentiable functions of any form to discrete data. The method applies an iterative technique to untransformed data, allowing unbiased estimation of nonlinear functions. A FORTRAN subroutine and complete user instructions are provided, along with the basic mathematical theory and examples of the use and performance of the algorithm. Various pitfalls which may be encountered in using the routine also are discussed. Use of this technique will allow the development of complex numerical models of natural systems. The proper use of the method depends upon the selection of the underlying mathematical function and initial parameter estimates for describing the phenomenon under study.

Acknowledgments—M.v.H. thanks Bill Menke for opening up the world of inverse theory to her, and both authors thank two anonymous reviewers for helpful comments. This work was funded under NOAA's VENTS program, NOAA/PMEL contribution No. 929.

REFERENCES

Draper, N. R., and Smith, H., 1966, Applied regression analysis: John Wiley & Sons, New York, 407 p.

Fox, C. G., 1985, Description, analysis, and prediction of sea-floor roughness using spectral models: Tech. Rept. TR 279, Naval Oceanographic Office, Bay St. Louis, NSTL MS 39522-5001, 218 p.

Menke, W., 1984, Geophysical data analysis: discrete inverse theory: Academic Press, Orlando, Florida, 260 p.

Scarborough, J. B., 1930, Numerical mathematical analysis: The Johns Hopkins Press, Baltimore, Maryland, 416 p.

Appendix overleaf

APPENDIX

```
      SUBROUTINE CURVEFIT(X,Y,FIT,N,PARAM,MIN,L,TEST,IEX,ILIST,IERR)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!  This subroutine fits a non-linear function to any data set, using
!  an iterative method as described by Draper and Smith (1966)*, p. 267.
!  Any non-linear function can be fitted, as long as partial
!  derivatives with respect to each of its parameters can be found.
!  Initial estimates for the parameters have to be supplied to the
!  subroutine.  The best estimates for the parameter increments are found
!  according to the least squares technique described by Menke (1984)**,
!  equation (3.12) p.41.
!
!  *Applied Regression Analysis, by N. R. Draper and H. Smith, John
!   Wiley and Sons, 407 p., 1966.
!  **Geophysical data analysis: discrete inverse theory, by William Menke,
!   Academic Press, 260 p., 1984.
!
!  Explanation of arguments:
!    X     = array with independent variables (sent, not altered)
!    Y     = array with dependent variables; these values to be fitted by
!            non-linear function (sent, not altered)
!    FIT   = array with values of the non-linear function fitted to the
!            data in array Y (returned)
!    N     = number of entries in arrays X,Y,FIT (sent, not altered)
!    PARAM = array with parameters that characterize the non-linear
!            function to be fitted; contains initial estimates when subr.
!            first called, contains final parameters for the fitted
!            non-linear function upon return (sent, altered, returned)
!    MIN   = array with iteration cut-off values for the parameters in
!            array PARAM.  The first value in MIN corresponds to the
!            first parameter in PARAM, etc. (sent, not altered)
!    L     = number of parameters to solve for (i.e. number of entries
!            in arrays MIN and PARAM) (sent, not altered)
!    TEST  = division by a number smaller than test is flagged as an
!            error (sent, not altered)
!    IEX   = value to check exponents in the functions against.
!            Exponents with absolute values exceeding IEX are flagged
!            as an error.  This check may not be necessary for
!            applications other than some exponential functions (sent,
!            not altered)
!    ILIST = 1 for summary of iteration process, 0 for no listing
!            (sent, not altered)
!    IERR  = 0 for no error; any other number means error (returned)
!
!  NOTE: the functions at the end of this program section need to
!  be altered, depending on the desired non-linear function to be
!  fitted.  The number of partial derivatives changes with the number
!  of parameters that are needed to characterize the non-linear
!  function.  Currently the program allows curve fitting for a function
!  with 8 parameters, and the data set to be fitted has a maximum of 1024
!  entries.  If the data set is larger or if the fitting function has
!  more than 8 parameters, parameters IS1 and IS2 at the start of
!  this subroutine will have to be altered accordingly.  For more than
!  8 parameters read comments in the code as to where to make
!  additional changes.
!
!  NOTE: currently an error condition will arise if the solution does not
!  converge after 100 iterations.  Parameter ICON at the beginning of the
!  subroutine can be reset if more or less iterations are desired.
!
!  NOTE: currently the program does not allow for more than 10
!  parameter increment bisections per iteration.  Parameter LBIS at the
!  beginning of the subroutine can be reset if more or less bisections are
!  desired.
!
!  NOTE: currently the subroutine is set up to find an exponential fit
!  to data.  PARAM(1) is the first unknown parameter, PARAM(2) the
!  second, and PARAM(3) the third.  This means function F1 should
!  contain the partial derivative with respect to the first parameter,
!  function F2 the partial with respect to the second parameter, etc.
!
!  NOTE: there is the possibility that certain parameters are zero when
!  passed to the functions.  This creates a problem if the same
!  parameter is in the denominator of one of the functions.  In
!  addition it is possible that a parameter passed to a function
!  causes the exponent in the function to be so large that the
!  computer can not store its value.  Either one of these possibilities
```

```
!  can occur for some exponential fits.  To prevent the program from
!  crashing under these circumstances, subroutine TESTVAL is called from
!  three locations inside subroutine CURVEFIT to prevent zero divides or
!  very large exponents.  The user will have to adapt the checks in
!  subroutine TESTVAL to the non-linear function selected.
!
!  NOTE: parameter KEX may need to be changed, if the computer on
!  which this subroutine is used is not a 32 bit computer.  KEX is
!  used to check the sum of the squares of the residuals to prevent
!  this value from going out of range.
!
!  This subroutine calls subroutines GAUSS, GJINV (from Menke (1984),
!  p.214 and p.218) and TESTVAL.
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      PARAMETER(IS1=1024,IS2=8)  !* NOTE: change for more than 8
                                 !* param. or data set larger than 1024
      PARAMETER(ICON=100)        !* NOTE: change for number of iterations
                                 !* other than 100.
      PARAMETER(LBIS=10)         !* NOTE: change for number of bisections
                                 !* other than 10.
      PARAMETER(KEX=200)         !* NOTE: 307 is the maximum exponent a
                                 !* REAL*8 variable can hold in a 32 bit
                                 !* computer.  Parameter KEX may need to
                                 !* be changed for other types of
                                 !* computers.

      REAL*4 X(1),Y(1),FIT(1),PARAM(1),MIN(1)
      REAL*4 D(IS1),G(IS1,IS2),M(IS2),GTD(IS2),GTG(IS2,IS2)
      REAL*4 GTGINV(IS2,IS2),PAROLD(IS2)
      REAL*8 SSQOLD,SSQNEW,ROOTMSQ

!  Initialize variables
      IERR=0  !* Error flag
      ITER=0  !* Iteration counter
      NBIS=0  !* Bisection counter

!  Check if the problem is underdetermined
      IF(N.LT.L) THEN
        IERR=1
        PRINT*,'the problem is underdetermined -- use more data'
        PRINT*,'points for the number of parameters requested'
        RETURN
      ENDIF

!  Check if parameters IS1 and IS2 are large enough
      IF(N.GT.IS1) THEN
        IERR=1
        PRINT*,'parameter IS1 is too small for the number of data'
        PRINT*,'points requested'
        RETURN
      ENDIF
      IF(L.GT.IS2) THEN
        IERR=1
        PRINT*,'parameter IS2 is too small for the number of unknowns'
        PRINT*,'to be solved'
        RETURN
      ENDIF

!  Check for zero divide and/or excessively large exponents
      CALL TESTVAL(X,N,PARAM,TEST,IEX,IERR)
      IF(IERR.NE.0) THEN
        PRINT*,'error: zero divide or absolute value of exponent'
     >    ,' exceeding ',IEX
        RETURN
      ENDIF

!  Form vector D; i.e. a data point minus the isolated Taylor Series
!  term for that data point.  In addition calculate the sum of the
!  squares of the residuals.
      SSQOLD=0.D0
      DO I=1,N
        D(I)=Y(I)-FO(X(I),PARAM)
        SSQOLD=SSQOLD+D(I)**2
        IF(SSQOLD.GT.1.D0*DEXP(DBLE(KEX))) THEN
          IERR=1
          PRINT*,'the sum of the squares of the residuals is too '
     >      ,'large; run the program again with better initial '
```

```
     >      ,'estimates for the parameters.'
            RETURN
          ENDIF
        END DO
        ROOTMSQ=DSQRT(SSQOLD)  !* RMS residual

!  Allow for printing of iteration information if ILIST=1.
      IF(ILIST.EQ.1) PRINT 10
   10 FORMAT(' ITERATION   # OF BISECT.   RMS ERROR ',
     > '   PARAMETERS 1,2,3  ETC.')
      IF(ILIST.EQ.1) PRINT 20,ITER,NBIS,ROOTMSQ,(PARAM(I), I=1,L)
   20 FORMAT(I6,10X,I3,8X,G10.2,8(3X,G10.4)) !* NOTE:adapt this format
                                             !* statement for more
                                             !* than 8 param.

!  The matrix formation in the following program segment follows
!  equation 3.12 on page 41 of Geophysical Data Analysis: Discrete
!  Inverse Theory by Menke (1984).

!  Form matrix G
   23 DO I=1,N
        DO J=1,L
          IF(J.EQ.1) G(I,J)=F1(X(I),PARAM)
          IF(J.EQ.2) G(I,J)=F2(X(I),PARAM)
          IF(J.EQ.3) G(I,J)=F3(X(I),PARAM)
          IF(J.EQ.4) G(I,J)=F4(X(I),PARAM)
          IF(J.EQ.5) G(I,J)=F5(X(I),PARAM)
          IF(J.EQ.6) G(I,J)=F6(X(I),PARAM)
          IF(J.EQ.7) G(I,J)=F7(X(I),PARAM)  !* NOTE: add additional
          IF(J.EQ.8) G(I,J)=F8(X(I),PARAM)  !* calls to functions for
        END DO                              !* more than 8 param.
      END DO

!  Form matrix GTD
      DO I=1,L
      SUM=0.0
        DO J=1,N
          GTD(I)=SUM+D(J)*G(J,I)
          SUM=GTD(I)
        END DO
      END DO

!  Form matrix GTG
      DO I=1,L
        DO K=1,L
        SUM=0.0
          DO J=1,N
            GTG(I,K)=SUM+G(J,I)*G(J,K)
            SUM=GTG(I,K)
          END DO
        END DO
      END DO

!  Find the inverse of matrix GTG
      CALL GJINV(GTG,GTGINV,L,IS2,TEST,IERR)
      IF(IERR.NE.0) THEN
        PRINT*,'error in matrix inversion'
        RETURN
      ENDIF
!  Find  vector M (i.e. increment PARAM(1), increment PARAM(2), etc.)
      DO I=1,L
      SUM=0.0
        DO J=1,L
          M(I)=SUM+GTGINV(I,J)*GTD(J)
          SUM=M(I)
        END DO
      END DO

!  Create new, corrected parameters
      DO I=1,L
        PAROLD(I)=PARAM(I)      !* Keep the old parameter value stored
        PARAM(I)=PAROLD(I)+M(I) !* Add param. corrections to old param.
      END DO

!  Check for zero divide and/or excessively large exponents
      CALL TESTVAL(X,N,PARAM,TEST,IEX,IERR)
      IF(IERR.NE.0) THEN
        PRINT*,'error: zero divide or absolute value of exponent'
     >   ,' exceeding ',IEX
```

```
        RETURN
      ENDIF

!  Compute new sum of squares of residuals
25    SSQNEW=0.D0
      DO I=1,N
        SSQNEW=SSQNEW+(Y(I)-FO(X(I),PARAM))**2
          IF(SSQNEW.GT.1.D0*DEXP(DBLE(KEX))) THEN
          IERR=1
          PRINT*,'the sum of the squares of the residuals is too '
     >    ,'large; run the program again with better initial '
     >    ,'estimates for the parameters.'
          RETURN
        ENDIF
      END DO

!  Test for convergent solution (SSQNEW<SSQOLD).  If not convergent,
!  bisect the corrections and recompute corrected parameters.
      IF(SSQNEW.LT.SSQOLD) GO TO 30  !* Solution is converging
      DO I=1,L
        M(I)=0.5*M(I)           !* Bisect the correction
        PARAM(I)=PAROLD(I)+M(I) !* Recompute corrected param. using
      END DO                    !* bisected correction terms

!  Check for zero divide and/or excessively large exponents
      CALL TESTVAL(X,N,PARAM,TEST,IEX,IERR)
      IF(IERR.NE.0) THEN
        PRINT*,'error: zero divide or absolute value of exponent'
     >  ,' exceeding ',IEX
        RETURN
      ENDIF

      NBIS=NBIS+1 !* Update the bisection counter
      IF(NBIS.GT.LBIS) GO TO 30  !* Maximum allowable no. of bisections
                                 !* exceeded.  Continue the program.
      GO TO 25  !* Check if the bisection leads to convergence.  If not,
                !* bisect the already bisected correction again.

!  Test if parameter corrections are smaller than the iteration
!  cut-off values
30    DO I=1,L
        IF(ABS(PARAM(I)-PAROLD(I)).GT.MIN(I)) GO TO 40 !* Correction
      END DO                                           !* terms still too
                                                       !* large; continue
                                                       !* iteration
      ROOTMSQ=DSQRT(SSQNEW)  !* RMS residual
      GO TO 50  !* Correction terms smaller than the cut-off values;
                !* terminate iterations

!  Start a new iteration
40    ITER=ITER+1  !* Update iteration counter
      IF(ITER.GT.ICON) THEN   !* Do not allow more than ICON iterations
        PRINT*,'no more than ',ICON,' iterations are allowed'
        IERR=1
        RETURN
      ENDIF
      SSQOLD=SSQNEW
      ROOTMSQ=DSQRT(SSQOLD)  !* RMS residual
      NBIS=0  !* Reinitialize the bisection counter

!  Print information of the completed iteration if ILIST=1.
      IF(ILIST.EQ.1) PRINT 20,ITER,NBIS,ROOTMSQ,(PARAM(I), I=1,L)

!  Form a new vector D using corrected parameters.
      DO I=1,N
        D(I)=Y(I)-FO(X(I),PARAM)
      END DO
      GO TO 23  !* Start matrix formation for the new iteration

!  The last iteration has been completed; compute the final non-linear fit.
50    ITER=ITER+1  !* Update iteration counter
      DO I=1,N     !* Compute the fitted function
        FIT(I)=FO(X(I),PARAM)
      END DO

!  Print information of the last iteration if ILIST=1.
      IF(ILIST.EQ.1) THEN
        PRINT 20,ITER,NBIS,ROOTMSQ,(PARAM(I), I=1,L)
        PRINT *,'FINAL PARAMETERS ARE',(PARAM(I), I=1,L)
```

```
        PRINT *,'FOR ITERATION CUT-OFFS',(MIN(I), I=1,L)
        PRINT *,' '
      ENDIF

      RETURN
      END

!  Functions to calculate non-linear fit to the data, and partial
!  derivatives with respect to the parameters that characterize the
!  nonlinear function.
!
!  NOTE: all functions below will change if a different non-linear
!  function is selected by the user.

      FUNCTION F0(X,PARAM)   !* Calculate the function
      REAL*4 PARAM(1)
      F0=PARAM(1)*EXP(-0.5*(X-PARAM(2))**2/PARAM(3))
      RETURN
      END

      FUNCTION F1(X,PARAM)   !* Partial derivative wrt 1st parameter
      REAL*4 PARAM(1)
      F1=EXP(-0.5*(X-PARAM(2))**2/PARAM(3))
      RETURN
      END

      FUNCTION F2(X,PARAM)   !* Partial derivative wrt 2nd parameter
      REAL*4 PARAM(1)
      F2=(X-PARAM(2))/PARAM(3)*PARAM(1)*EXP(-0.5*(X-PARAM(2))**2/
     > PARAM(3))
      RETURN
      END

      FUNCTION F3(X,PARAM)   !* Partial derivative wrt 3rd parameter
      REAL*4 PARAM(1)
      F3=0.5*PARAM(1)/PARAM(3)**2*(X-PARAM(2))**2*
     > EXP(-0.5*(X-PARAM(2))**2/PARAM(3))
      RETURN
      END

      FUNCTION F4(X,PARAM)   !* Partial derivative wrt 4th parameter
      REAL*4 PARAM(1)
      F4=0.
      RETURN
      END

      FUNCTION F5(X,PARAM)   !* Partial derivative wrt 5th parameter
      REAL*4 PARAM(1)
      F5=0.
      RETURN
      END

      FUNCTION F6(X,PARAM)   !* Partial derivative wrt 6th parameter
      REAL*4 PARAM(1)
      F6=0.
      RETURN
      END

      FUNCTION F7(X,PARAM)   !* Partial derivative wrt 7th parameter
      REAL*4 PARAM(1)
      F7=0.
      RETURN
      END

      FUNCTION F8(X,PARAM)   !* Partial derivative wrt 8th parameter
      REAL*4 PARAM(1)
      F8=0.
      RETURN
      END

!  NOTE: More partials will be added here if the selected function
!  has more than 8 unknown parameters.  There is one partial
!  derivative for each parameter.


      SUBROUTINE TESTVAL(X,N,PARAM,TEST,IEX,IERR)
!  subroutine TESTVAL, to test if denominators become too small (i.e.
!  zero divide), and if exponents become too large in any of the
!  functions of subroutine CURVEFIT.
```

```
!
!  Explanation of subr. arguments:
!    X     = array with independent variables (sent, not altered)
!    N     = number of elements in array X (sent, not altered)
!    PARAM = array with function parameters (sent, not altered)
!    TEST  = division by a number smaller than TEST is flagged as an
!            error (sent, not altered)
!    IEX   = the absolute value of exponents with values larger than IEX
!            are flagged as an error (sent, not altered)
!    IERR  = 0 for no error, 1 if denominator zero, 2 if exponent
!            larger than IEX, 3 if both the denominator is zero and
!            the exponent exceeds IEX.
!
!  NOTE: For applications other than some exponential functions more or
!  less exponents and denominators may need to be checked.  Which
!  denominators and exponents need to be checked depends on the
!  non-linear function selected.
!
!  NOTE: If an exponent needs to be checked, it is not always necessary
!  to check the absolute value of the exponent.  Most computers will deal
!  with large negative exponents.  It is better not to check the absolute
!  value of an exponent unless necessary, because it will make the program
!  more restrictive.

      REAL*4 X(1),PARAM(1)

      IERR=0  !* Initialize error flag
      JERR=0

!  Check parameters to prevent zero divide in the functions
      IF(ABS(PARAM(3)).LT.TEST) THEN !* NOTE: this program segment may need
        IERR=1                       !* to be altered or omitted if a
      ENDIF                          !* different function is selected.

!  Check to prevent function exponent from being too large
      DO I=1,N                                    !* NOTE: this program
        EXPON=-0.5*(X(I)-PARAM(2))**2/PARAM(3) !* segment may need to be
        IF(ABS(EXPON).GT.FLOAT(IEX)) THEN         !* altered or omitted if a
          JERR=2                                  !* different function is
          GO TO 5                                 !* selected.
        ENDIF
      END DO

  5   IERR=IERR+JERR
      RETURN
      END

      SUBROUTINE GJINV(A,B,N,NSTORE,TEST,IERROR)
      REAL*4 A(NSTORE,NSTORE),B(NSTORE,NSTORE),TEST
      INTEGER N,NSTORE,IERROR
!  subroutine GJINV, to invert a square matrix AB=I by Gauss-Jordan
!  reduction -- from Menke (1984), p.218
!
!  protocol:
!
!  A: (sent, altered) a square matrix
!  B: (returned) the inverse of A
!  N: (sent) the size of A and B (currently must be less than or equal
!     to 100)
!  NSTORE: (sent) the dimensioned size of A and B in the calling routine
!  TEST: (sent) division by numbers smaller than test generate an error
!        condition
!  IERROR: (returned) error flag, zero on no error
!
!  subroutines needed: GAUSS

      DIMENSION C(100)

      DO 10 ICOL=1,N  !* Build inverse columnwise by solving AB=I

        DO 1 I=1,N   !* C is ICOL column of identity matrix
          C(I)=0.0
  1     CONTINUE
        C(ICOL)=1.0

        IF(ICOL.EQ.1) THEN   !* Triangularize A on first call
          ITRIAG=0
        ELSE
          ITRIAG=1
        ENDIF
```

```
      CALL GAUSS(A,C,N,NSTORE,TEST,IERROR,ITRIAG)

      IF(IERROR.NE.0) THEN  !* Return on error
        RETURN
      ENDIF

      DO 2 I=1,N  !* Move solution into matrix inverse
        B(I,ICOL)=C(I)
2     CONTINUE

10    CONTINUE

      RETURN
      END

      SUBROUTINE GAUSS(A,VEC,N,NSTORE,TEST,IERROR,ITRIAG)
      REAL A(NSTORE,NSTORE),VEC(NSTORE),TEST
      INTEGER N,NSTORE,IERROR,ITRIAG
!  subroutine GAUSS: Solves Ax=VEC (where A is square) by
!  Gauss-Jordan reduction.   from Menke (1984), p. 214
!
!  protocol:
!
!  A: (sent, altered) a real square N by N matrix
!  VEC: (sent, altered, returned) set to RHS of matrix equation
!       before call.  Solution is returned in VEC after call.
!  N: (sent) the size of A, VEC (currently limited to 100)
!  NSTORE: (sent) the dimension of A in the calling pgm
!  TEST: (sent) division by a number smaller than test is flagged
!        as an error
!  IERROR: (returned) error flag, zero on no error
!  ITRIAG: (sent) matrix triangularization omitted if ITRIAG=1
!          (note that matrix A must be preserved from triangularizing
!          call if ITRIAG=1 option is used)

      DIMENSION LINE(100), ISUB(100)  !* Must be increased if N>100

      SAVE ISUB,L1

      IET=0    !* Initial error flags, one for triangularization,
      IEB=0    !*                     one for back-solving

!  Triangularize the matrix A, replacing the zero elements of the
!  triangularized matrix with the coefficients needed to transform
!  the vector VEC

      IF (ITRIAG.EQ.0) THEN           !* Triangularize matrix

        DO 1 J=1,N                    !* LINE is an array of flags.
          LINE(J)=0                   !* Elements of A are not moved
                                      !* during pivoting. LINE=0 flags
1       CONTINUE                      !* pivoting.  LINE=0 flags unused
                                      !* lines

        DO 30 J=1,N-1                 !* Triangularize matrix by
                                      !* partial pivoting
          BIG=0.0                     !* Find biggest element in j-th
                                      !* column of unused part of A
          DO 40 L1=1,N
            IF(LINE(L1).EQ.0) THEN
              TESTA=ABS(A(L1,J))
              IF(TESTA.GT.BIG) THEN
                I=L1
                BIG=TESTA
              ENDIF
            ENDIF
40        CONTINUE

          IF(BIG.LE.TEST) THEN     !* Test for division by 0
            IET=1
            GO TO 250
          ENDIF

          LINE(I)=1  !* Selected unused line becomes used line
          ISUB(J)=I  !* ISUB points to j-th row of triangularized matrix
          SUM=1.0/A(I,J)  !* Reduce matrix towards triangle

          DO 10 K=1,N
            IF(LINE(K).EQ.0) THEN
```

```
            B=A(K,J)*SUM
            DO 20 L=J+1,N
              A(K,L)=A(K,L)-B*A(I,L)
20          CONTINUE
            A(K,J)=B
          ENDIF
10      CONTINUE
30    CONTINUE

      DO 32 J=1,N  !* Find last unused row and set its pointer.  This
                   !* row contains the apex of the triangle.
        IF(LINE(J).EQ.0) THEN
          L1=J  !* Apex of triangle
          ISUB(N)=J
          GOTO 35  !* Break loop
        ENDIF
32    CONTINUE
35    CONTINUE

    ENDIF  !* Done triangularizing

    !* Start backsolving
    DO 100 I=1,N  !* Invert pointers. LINE(I) now gives row # in
                  !* triang. matrix of i-th row of actual matrix
      LINE(ISUB(I))=I
100 CONTINUE

    DO 320 J=1,N-1   !* Transform the vector to match triangularized
                     !* matrix
      B=VEC(ISUB(J))
      DO 310 K=1,N
        IF(LINE(K).LE.J) THEN
          CONTINUE  !* Skip elements outside of triangle
        ELSE
          VEC(K)=VEC(K)-A(K,J)*B
        ENDIF
310   CONTINUE
320 CONTINUE

    B=A(L1,N)  !* Apex of triangle
    IF(ABS(B).LE.TEST) THEN  !* check for division by 0
      IEB=2
      GO TO 250
    ENDIF
    VEC(ISUB(N))=VEC(ISUB(N))/B
    DO 50 J=N-1,1,-1      !* back-solve rest of triangle
      SUM=VEC(ISUB(J))
      DO 60 J2=J+1,N
        SUM=SUM-VEC(ISUB(J2))*A(ISUB(J),J2)
60    CONTINUE
      B=A(ISUB(J),J)
      IF(ABS(B).LE.TEST) THEN      !* test for division by zero
        IEB=2
        GO TO 250
      ENDIF
      VEC(ISUB(J))=SUM/B       !* solution returned in vec
50  CONTINUE

 !* put the solution vector into the proper order

    DO 230 I=1,N       !* reorder solution
      DO 210 K=I,N     !* search for i-th solution element
        IF(LINE(K).EQ.I) THEN
          J=K
          GO TO 220  !* break loop
        ENDIF
210   CONTINUE
220   CONTINUE
      B=VEC(J)    !* swap solution and pointer elements
      VEC(J)=VEC(I)
      VEC(I)=B
      LINE(J)=LINE(I)
230 CONTINUE

250 IERROR=IET+IEB  !* set final error flag

    RETURN
    END
```