NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A STABLE VARIANT OF THE SECANT METHOD FOR SOLVING NONLINEAR EQUATIONS

W. B. Gragg*
Department of Mathematics
University of California, San Diego

G. W. Stewart†
Departments of Computer Science and Mathematics
Carnegie-Mellon University
Pittsburgh, Pennsylvania

April 1974

*Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR 71-2006. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

†Supported in part by the Office of Naval Research under Contract N00014-67-A-0128-0018.

ABSTRACT

The usual successive secant method for solving systems of nonlinear equations suffers from two kinds of instabilities. First the formulas used to update the current approximation to the inverse Jacobian are numerically unstable. Second, the directions of search for a solution may collapse into a proper affine subspace, resulting at best in slowed convergence and at worst in complete failure of the algorithm. In this report it is shown how the numerical instabilities can be avoided by working with factorizations of matrices appearing in the algorithm. Moreover, these factorizations can be used to detect and remedy degeneracies among the directions. A second part of this report documents and lists a program implementing the algorithm described in the first part.

PART I

1. Introduction

In this paper we shall be concerned with the successive secant method for solving the system of nonlinear equations

$$(1.1) \qquad f(x) = 0,$$

where f is a mapping from some domain in real n-space into real n-space ($f: D \subset \mathbb{R}^n \to \mathbb{R}^n$). Given approximations $x_1, x_2, \ldots, x_{n+1}$ to a solution of (1.1), a new approximation x_* is generated as follows. Let $\ell: \mathbb{R}^n \to \mathbb{R}^n$ be the affine function that interpolates f at $x_1, x_2, \ldots, x_{n+1}$; that is

$$(1.2) \qquad f_i := f(x_i) = \ell(x_i) \qquad (i=1,2,\ldots,n+1).$$

Then x_* is taken to be the zero of the function ℓ. If the points $x_1, x_2, \ldots, x_{n+1}$ are affinely independent then ℓ is uniquely defined. The approximation x_* will be uniquely defined provided the vectors $f_1, f_2, \ldots, f_{n+1}$ are affinely independent (cf. (1.4) below). The method derives its name from the fact that the i-th coordinate function of ℓ represents the secant hyperplane interpolating the i-th coordinate function of f.

Various formulas can be written for the approximation x_* (see [2] for the a detailed discussion of secant methods and their convergence theory). We shall use the following representation. Let X be the $n \times (n+1)$ matrix ($X \in \mathbb{R}^{n \times (n+1)}$) defined by

$$X := (x_1, x_2, \ldots, x_{n+1}),$$

and let

$$F := (f_1, f_2, \ldots, f_{n+1}).$$

Define the operator Δ by

$$\Delta X = (x_2 - x_1, x_3 - x_1, \ldots, x_{n+1} - x_1).$$

Then it is easily verified that the function ℓ defined by

$$\ell(x) = f_1 + \Delta F(\Delta X)^{-1}(x - x_1) \tag{1.3}$$

satisfies (1.2). It follows from solving the equation $\ell(x) = 0$ that

$$x_* = x_1 - \Delta X(\Delta F)^{-1} f_1. \tag{1.4}$$

The existence of the inverses in (1.3) and (1.4) is guaranteed by the affine independence of the columns of X and F.

The new approximation x_* will not in general be an exact zero of f, and the process must be repeated iteratively. This may be done in several ways. We shall be concerned with the successive variant in which x_* replaces one of the points x_i. Conventionally this is done in one of two ways. Either x_* replaces x_{n+1}, or x_* replaces that column of X for which the corresponding column of F has largest norm. In any case the iterative process generates sequences of matrices $X_1, X_2, \ldots$ and a corresponding sequence $F_1, F_2, \ldots$ with X_{k+1} differing from X_k in only a single column (in practice it may be necessary to permute the columns of X_k before inserting $x_*^{(k)}$; see Section 4.2 below).

When f is differentiable, the matrix $\Delta F(\Delta X)^{-1}$ in (1.4) may be regarded as an approximation to the Jacobian f' of f. Thus the secant formula (1.4) is a discretization of Newton's method, a method that under appropriate conditions converges quadratically to a zero of f. The convergence theory for the successive secant method suggests that if the matrices ΔX_k remain uniformly nonsingular, then n steps of the secant method will be roughly comparable to one step of Newton's method (see [2] and [3]). This has important computational consequences. The ab initio calculation of $(\Delta F)^{-1}f_1$ requires $O(n^3)$ operations (see, e.g., [5]), and therefore n steps of the secant method will require $O(n^4)$ operations, which may be prohibitively large. The usual cure for this problem is to calculate $(\Delta F_{k+1})^{-1}$ directly from $(\Delta F_k)^{-1}$ (actually the inverses of slightly different matrices are calculated). Since F_k and F_{k+1} are simply related, this can be done in $O(n^2)$ operations, giving a satisfactory $O(n^3)$ operation count for n steps of the successive secant method (for the first such implementation see [4]).

The method outlined above has two serious defects. First the scheme for updating $(\Delta F)^{-1}$ is numerically unstable. Second, the columns of the matrices X_k may tend to collapse into proper affine subspaces of $\mathbb{R}^n$, resulting in the prediction of wild points or at least in slowed convergence. The first problem arises whenever ΔF_k is ill-conditioned. In this case $(\Delta F_k)^{-1}$ is computed inaccurately and these inaccuracies transmit themselves to subsequent inverses, even though the corresponding ΔF's are well conditioned. The same problem occurs in linear programming (see, e.g., [1]), and one could adopt the usual solution of periodically reinverting ΔF. However, this entails extra work for the reinversion and extra storage to hold the matrix F. Moreover, one must face the tricky problem of deciding when to reinvert.

The problem of degeneracy among the columns of X arises, among other occasions, when one of the component functions of f is linear. Then the linear component and the corresponding component of ℓ, call it ℓ_i, are identical. It follows that x^* lies in the proper affine subspace defined by $\ell_i(x) = 0$.

Ultimately all the column of some X_k must lie in this subspace, and ΔX_k will be singular. The matrix ΔF_k may not be singular, but it will almost certainly be ill-conditioned, and the prediction $x_*^{(k)}$ will be spurious. Moreover, as noted above, the inaccuracies in $(\Delta F_k)^{-1}$ will propogate themselves via the update formulas.

The purpose of this paper is to show how the two problems mentioned above can be resolved by generating and updating QR factorizations of the matrices X_k and F_k. The factorization of F permits the $O(n^2)$ solution of the equation $\Delta Fz = f_1$, which is equivalent to forming $(\Delta F)^{-1}f_1$. The factorization of X enables one to detect degeneracies in the columns of X. Moreover, the factorization can be used to alter a column of X in such a way as to reduce or remove the degeneracy. The factorizations of X_{k+1} and F_{k+1} can be obtained from those of X_k and F_k in $O(n^2)$ operations.

In the next section we shall introduce the factorizations, show how they may be used to execute a step of the secant method, and show how they may be updated. We shall also show that the updating method is numerically stable. In Section 3, we shall show how the factorization can be used to detect and remove degeneracies in X. In Section 4 some comments on the practicalities of implementing these methods are given, and in Section 5 some numerical examples. Part Two of this report consists of a documented program implementing the method presented in Part One.

2. Factorization

In this section we shall be concerned with the stable implementation of a single secant step. Suppose that at step k we are given nonsingular matrices P_k and Q_k such that the matrices Y_k and G_k defined by

(2.1) $$X_k = P_k^T Y_k$$

and

(2.2) $$G_k = Q_k F_k$$

are upper trapezoidal, i.e. zero below the diagonal. (Numerically the matrices P_k and Q_k will be very nearly orthogonal, but we need not assume so.) Because premultiplication by a matrix acts column by column on the multiplicand, we have

$$\Delta X_k = P_k^T(\Delta Y_k)$$

and

$$\Delta G_k = Q_k(\Delta F_k).$$

Moreover, the matrices ΔY_k and ΔG_k are upper Hessenberg, i.e. zero below the first subdiagonal.

Now let $x_*^{(k)}$ be the vector obtained from a single secant step:

(2.3) $$x_*^{(k)} = x_1^{(k)} - \Delta X_k(\Delta F_k)^{-1} f_1^{(k)}.$$

If we set $y_*^{(k)} = P_k^{-T} x_*^{(k)}$, then (2.3) can be written in the form

(2.4) $$y_*^{(k)} = y_1^{(k)} - \Delta Y_k(\Delta G_k)^{-1} g_1^{(k)},$$

where $y_1^{(k)}$ and $g_1^{(k)}$ are the first columns of Y_k and G_k. Equation (2.4) suggests the following algorithm.

(2.5)

1. Solve the system $\Delta G_k z = g_1^{(k)}$
2. $y_*^{(k)} = y_1^{(k)} - \Delta Y_k z$
3. $x_*^{(k)} = P_k^T y_*^{(k)}$
4. $f_*^{(k)} = f(x_*^{(k)})$
5. $g_*^{(k)} = Q_k f_*^{(k)}$

This algorithm produces not only the secant approximation $x_*^{(k)}$ but also the function value $f_*^{(k)}$ and its Q-transform $g_*^{(k)}$. Excepting step 4, the bulk of the work done by the algorithm is concentrated in step 1. Since ΔG_k is an upper Hessenberg matrix, step 1 can be accomplished by standard techniques in $O(n^2)$ operations [5, p. 218]. Thus a knowledge of the factorizations (2.1) and (2.2) allows us to compute a secant approximation in $O(n^2)$ operations.

Of course $x_*^{(k)}$ must replace a column of X_k and $f_*^{(k)}$ replace the corresponding column of F_k. This amounts to replacing the same columns of Y_k and G_k by $y_*^{(k)}$ and $g_*^{(k)}$ to give new matrices Y_k^* and G_k^*. In principle algorithm (2.5) can be applied to these new matrices to give another approximation. In practice, however, G_k^* will no longer be upper trapezoidal and step 1 of (2.5) cannot be effected in $O(n^2)$ operations. To circumvent this difficulty we shall show how to construct orthogonal matrices R_k and S_k such that

$$Y_{k+1} := R_k Y_k^*$$

and

$$G_{k+1} := S_k G_k^*$$

are upper trapezoidal. If we then set

$$P_{k+1} := R_k P_k$$

and

$$Q_{k+1} := S_k Q_k,$$

then the relations (2.1) and (2.2) will be satisfied with k replaced by k+1, and algorithm (2.5) may be efficiently reapplied.

For definiteness we shall deal with the computation of R_k and illustrate the general procedure by a specific example. For numerical reasons that will be discussed in Section 4, the order of the columns of Y and G cannot be assigned arbitrarily. This means that although $y_*^{(k)}$ may replace, say, column ℓ of Y, it may have to be inserted at some other position, say in column m. In the specific case where $n = 7$, $\ell = 1$, and $m = 3$, we shift column 2 into column 1, shift column 3 into column 2 and overwrite column 3 with $y_*^{(k)}$. This gives a matrix Y_k^* whose nonzero elements have the distribution

$$(2.6)\qquad \begin{array}{cccccccc} x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ 0 & x & x & x & x & x & x & x \\ 0 & 0 & x & x & x & x & x & x \\ 0 & 0 & x^3 & 0^3 & x & x & x & x \\ 0 & 0 & x^2 & 0 & 0^2 & x & x & x \\ 0 & 0 & x^1 & 0 & 0 & 0^1 & x & x \end{array} \quad .$$

The matrix R_k is computed as the product of 9 plane rotations or Householder transformations: $R_k = H_9H_8\ldots H_2H_1$. In the first stage, the transformations H_1, H_2, and H_3 are chosen in the usual way (see [5, p. 47]) to introduce zeros into the elements of the "stalactite" in column 3. These transformations will enter nonzero elements in the zero positions labled 1, 2, and 3, so that the matrix will be in Hessenberg form:

$$\begin{array}{cccccccc}
x & x & x & x & x & x & x & x \\
x^4 & x & x & x & x & x & x & x \\
0 & x^5 & x & x & x & x & x & x \\
0 & 0 & x^6 & x & x & x & x & x \\
0 & 0 & 0 & x^7 & x & x & x & x \\
0 & 0 & 0 & 0 & x^8 & x & x & x \\
0 & 0 & 0 & 0 & 0 & x^9 & x & x
\end{array} .$$

Now the transformations $H_4,\ldots,H_9$ are chosen to introduce zeros in the elements labeled 4,...,9, bringing the matrix to trapezoidal form. The matrix $P_{k+1} = H_9\ldots H_1 P_k$ can be formed directly by multiplying the transformations into P_k as they are generated. The matrix G_k^* also has the form (2.6) and is updated similarly.

The procedure sketched above is perfectly general. If column ℓ is to be deleted and a vector inserted in column m the vectors between column ℓ (exclusive) and m (inclusive) are shifted one column toward column ℓ and the new vector is inserted. The matrix is then reduced to triangular form as illustrated above. From the standpoint of operations, the case $\ell = m = 1$ is the worst, requiring the introduction of 2n-3 zeros. In all cases the operation count for the updating is $O(n^2)$.

The method is extremely stable in the sense that there are small matrices Z_k and H_k such that $P_k^T Y_k = X_k + Z_k$ and $Q_k(F_k+H_k) = G_k$. This implies that if no further rounding errors are made in algorithm (2.5), the value of $x_*^{(k)}$ is the value that would have been obtained by taking a secant step with the slightly perturbed matrices $X_k + Z_k$ and $F_k + H_k$.

The derivation of H_k is typical. The errors for each column are independent of one another, and it is sufficient to follow the history of a single column from its insertion as $g_*^{(k)}$. Now $g_*^{(k)}$ is computed according to (2.5.5). It follows from standard rounding error assumptions [5] that the computed $g_*^{(k)}$ satisfies

$$g_*^{(k)} = Q_k f_*^{(k)} + e_*^{(k)},$$

where

$$\|e_*^{(k)}\| \le n^{3/2}\|Q_k\| \, \|f_*^{(k)}\| \, \epsilon.$$

Here $\|\cdot\|$ denotes the spectral norm [5, p. 57] and ϵ is a small constant that depends on the arithmetic used to compute $g_*^{(k)}$. It follows that

$$g_*^{(k)} = Q_k(f_*^{(k)} + h_*^{(k)})$$

where

$$\|h_*^{(k)}\| = \|Q_k^{-1}e_*^{(k)}\| \le n^{3/2}\|Q_k\|\,\|Q_k^{-1}\|\,\|f_*^{(k)}\|\,\epsilon. \tag{2.7}$$

Now the matrices Q_k are computed as the product of orthogonal matrices (see Section 4.4 below) and will themselves be very nearly orthogonal (for detailed error analyses of orthogonal transformations see [5]). It follows that certainly

$$\|h_*^{(k)}\| \le 2n^{3/2}\|f_*^{(k)}\|\,\epsilon. \tag{2.8}$$

Thus when $g_*^{(k)}$ is inserted in G_k, the error bound for the corresponding column of H_k^* is satisfactorily small.

As the matrix G_k^* and the subsequent G's are updated, the column of H corresponding to the inserted $g_*^{(k)}$ will grow, but very slowly as an elementary error analysis will show. Even this slow growth might be intolerable over a large number of iterations, but after about n iterations the column is discarded (this may be forced if necessary), and its replacement is born anew with little error. It is true that the matrices P_k and Q_k will slowly deviate from orthogonality, but orthogonality is not required in the above analysis. All that is needed is that P_k and Q_k be well conditioned so that in the case of Q_k we may pass from (2.7) to (2.8). Since P_k and Q_k are computed as products of orthogonal matrices, their condition cannot deteriorate in any reasonable number of iterations.

Two points in the above analysis bear stressing. First the matrices Z_k and H_k are uniformly bounded, provided no column is retained longer than a fixed number of iterations and the matrices P_k and Q_k remain well conditioned. In effect we can use and update the factorizations as long as we like. This is especially important in parameterized problems in which the factorizations from the solution of one problem are used to start the solution of a nearby problem (cf. Section 4.5). The second point is that the analysis implies that the error in any column will be small compared with the norm of that column. Even if the columns vary widely in size (in the matrix G they will), the error associated with a large column cannot overwhelm a small column.

3. Detecting and Correcting Degeneracy

As was pointed out in Section 1, the columns of X will be affinely dependent whenever ΔX is singular. In this section we shall show how the factorization of X introduced in the last section can be used to tell when ΔX is singular and if necessary remove the singularity by altering a column of X. The method to be used cannot be justified with complete rigor, although a suggestive theorem can be proved.

Actually we shall work with the matrices Y and ΔY, which are the ones that are at hand. There is some ambiguity in speaking of the singularity of

ΔY, since its columns may vary widely in size. For the sake of uniformity we shall instead examine the matrix A obtained from ΔY by scaling its columns so they have 2-norm unity:

$$(3.1) \qquad A := \left(\frac{y_2 - y_1}{\|y_2 - y_1\|}, \frac{y_3 - y_1}{\|y_3 - y_1\|}, \ldots, \frac{y_{n+1} - y_1}{\|y_{n+1} - y_1\|} \right).$$

There is more than just convention in this choice. The convergence proofs for the secant method require a uniform upper bound on the condition of the matrices A generated by the iteration.

The method for correcting degeneracies may be justified heuristically as follows. If A is nearly singular, then it has approximate left and right null vectors; that is there are vectors u and v with $\|u\| = \|v\| = 1$ such that $\|Au\|$ and $\|v^T A\|$ are small; say they are less than some fixed tolerance α. Now to say that $\|v^T A\|$ is small is to say that v is almost orthogonal to each column of A. Thus the condition of A may be improved by replacing some column with the vector v. However, it is important that v not replace a column that is already independent of the other columns of A. The vector u may be used to find a suitable column. Let u_ν be the component of u that is largest in absolute value: $|u_\nu| \geq |u_i|$ $(i=1,2,\ldots,n)$. Then the ν-th column of A is given by

$$(3.2) \qquad a_\nu = \frac{Au}{u_\nu} - \sum_{i \neq \nu} \frac{u_i}{u_\nu} a_i .$$

Since $|u_\nu| \geq n^{-1/2}$, the vector Au/u_ν is negligible, and (3.2) effectively expresses a_ν as a linear combination of the other columns of A. Thus ν should replace a_ν to give a new matrix A_1.

If A_1 is nearly singular, the process may be reapplied to give a matrix A_2, and so on. The following theorem shows that if α is not too large the sequence of matrices A_k so generated must terminate. We establish the result for rectangular matrices with an eye to applications to least squares problems.

Theorem 3.1. Let $A_0 \in \mathbb{R}^{m\times n}$ $(m \geq n)$ have columns of norm unity. Given $\alpha > 0$, generate a sequence $A_0, A_1, \ldots$ of matrices as follows. Let A_k be given and suppose that there are vectors u_k and v_k satisfying

$$\|u_k\| = \|v_k\| = 1, \tag{3.3}$$

and

$$\|A_k u_k\|,\ \|A_k^T v_k\| \leq \alpha \tag{3.4}$$

Let $u_\nu^{(k)}$ be a maximal component of u_k: $|u_\nu^{(k)}| \geq |u_i^{(k)}|$ $(i=1,2,\ldots,n)$. The matrix A_{k+1} is then the matrix obtained by replacing the ν-th column of A_k by v_k. If there are no vectors u_k and v_k satisfying (3.3) and (3.4), end the sequence with A_k. Then if

$$a < \frac{1}{\sqrt{n}(1+\sqrt{n})} \tag{3.5}$$

the sequence terminates with some A_k where $k < n$.

Proof. We shall show that in passing from A_k to A_{k+1}, the column that was thrown out must be a column of A_0. This is clearly true for the matrix A_0 itself. Assuming its truth for $A_0, A_1, \ldots, A_{k-1}$, we can by rearranging the columns of A_k write A_k in the form

$$A_k = (v_0, v_1, \ldots, v_{k-1}, a_k^{(k)}, \ldots, a_n^{(k)}),$$

where $a_k^{(k)}, \ldots, a_n^{(k)}$ are columns of A_0. Thus we must show that $u_i^{(k)}$ $(i=1,2,\ldots,k)$ cannot be maximal.

The case $i = 1$ is typical. Write A_k in the form $A_k = (v_0, A_2^{(k)})$. Then it follows from (3.4) that

$$\|v_0^T A_2^{(k)}\| \leq \sqrt{n-1}\ \alpha.$$

But if we write $u_k = (u_1^{(k)}, w_k^T)^T$

$$\begin{aligned} \alpha \geq |v_0^T A u_k| &= |v_0^T v_0 u_1^{(k)} + v_0^T A_2^{(k)} w_k| \\ &\geq |u_1^{(k)}| - \|v_0^T A_2^{(k)}\| \, \|w_k\| \\ &\geq |u_1^{(k)}| - \sqrt{n-1}\ \alpha. \end{aligned}$$

The inequality (3.5) then implies that $|u_1^{(k)}| < n^{-1/2}$ and $u_1^{(k)}$ cannot be maximal.

Now either the sequence terminates before $k = n-1$, or we must arrive at the matrix A_{n-1}. Since at this point all the columns of A_0 but one have been replaced, the matrix A_{n-1} satisfies $A_{n-1}^T A_{n-1} = (I + E)$, where $|e_{ij}| \leq \alpha$. Thus

$$\|E\| \leq n\alpha.$$

For any vector u with $\|u\| = 1$, we have

$$\begin{aligned} \|A_{n-1}u\|^2 &= |u^T A_{n-1}^T A_{n-1} u| = |u^T (I + E) u| \\ &\geq 1 - |u^T E u| \geq 1 - n\alpha \cdot \alpha > \alpha^2 \end{aligned}$$

and the sequence terminates with A_{n-1}. $\square$

So far as the secant method is concerned, the main problem is to compute the vectors u and v associated with the matrix A defined by (3.1). Since A is upper Hessenberg this can be done efficiently by a variant of the inverse power method. The motivation for the method is that if A is nearly singular then

A^{-1} will be large. Unless the elements of A^{-1} are specially distributed, the vector $u' = A^{-1}e$ will be large for almost any choice of e with $\|e\| = 1$. If we set $u = u'/\|u'\|$, then $\|Au\| = \|e\|/\|u'\| = 1/\|u'\|$ is small.

Because A is upper Hessenberg, it can be reduced by orthogonal transformations to triangular form in $O(n^2)$ operations; that is we can cheaply compute an orthogonal matrix R such that

$$B = RA$$

is upper triangular. We then solve the system $Bu' = e$. Since $\|Au'\| = \|R^T Bu'\| = \|R^T e\| = \|e\|$, we can work with the vector $u' = B^{-1}e$ rather than $A^{-1}e$. The components of e are taken to be $\pm 1/\sqrt{n}$, where the signs are chosen to enhance the size of the solution. Specifically,

(3.6)
```
1. $u'_n = n^{-1/2}/b_{nn}$
2. For $i = n-1, n-2, \dots, 1$
    1. $\sigma = -\Sigma_{j=i+1}^{n} b_{ij}u'_j$
    2. $u'_i = [\sigma + \mathrm{sign}(\sigma)n^{-1/2}]/b_{ii}$.
```

The vector v is obtained by solving the system $B^T w = e$ in a manner analogous to (3.6) and setting $v = R^T w/\|R^T w\|$.

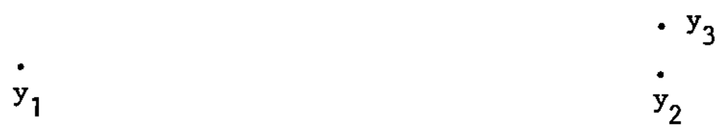
If $\|u'\|$ is large then a column of A, say the ν-th, must be replaced. From the definition of A, this amounts to replacing the $(\nu+1)$-st column of Y by $y_1 + \lambda v$, where λ is arbitrary. We are now in a position to describe our overall algorithm for detecting and removing degeneracies.

(3.7)

```
1. Form A according to (3.1)
2. Calculate u' as described above
3. If $\|u'\| \geq$ tol
    1. Find $\nu$ so that $|u_\nu| \geq |u_i|$ $(i=1,2,\ldots,n)$
    2. Calculate v as described above
    3. $y^* = y_1 + \min\{\|y_i - y_1\|\ i=2,\ldots,n+1\}v$
    4. Insert $y^*$ in Y, throwing out column $\nu+1$
    5. Go to 1
4. ...
```

As we mentioned at the beginning of this section, the above algorithm cannot be justified with complete rigor. Here we summarize the difficulties.

<u>Statement</u> 1. In the formation of A, the vector y_1 has been given a special role as a pivot. If another column of Y is used as a pivot, a different matrix A will be obtained. For example, if y_1, y_2, and y_3 are situated as shown



and y_1 is the pivot, then the vectors may well be judged to be affinely dependent. On the other hand if y_2 is the pivot, they will definitely be judged independent, since $y_1 - y_2$ and $y_3 - y_2$ are orthogonal. We have chosen y_1 as a pivot because the ordering imposed on the columns of Y and G creates the presumption that $x_1 = P^T y_1$ is nearer the zero of f than are the other columns of X (see Section 4.2).

Statement 3. If $\|u'\|$ is large, then A is certainly nearly singular. However it is conceivable that A could be nearly singular and the algorithm for computing u' fail to give a large vector. We feel that this is extremely unlikely (it is equivalent to the failure of the widely used inverse power method for finding eigenvectors [5, p. 619]).

The value of tol should not be too large, otherwise slow convergence or wild predictions may result. On the other hand, Theorem 3.1 below suggests that it should not be too small. We have used a value of 100 in our numerical experiments (for n = 100, the bound (3.5) gives α^{-1} 110).

Statement 3.3. The form of y^* shows that our method for removing degeneracies amounts to taking a "side step" from y_1 along the direction v. The length of the side step is arbitrary. We have chosen the distance between y_1 and y_2 as the length, since x_1 and x_2 are presumed to be the points nearest the zero of f.

Statement 3.5. With tol suitably chosen, the only way this statement could cause an infinite loop is for $\|Av\|$ to be repeatedly smaller than tol. This is unlikely; however, the fastidious user might place an upper bound on the number of attempts to remove the degeneracy in A. Alternatively he can replace only previously untouched vectors.

4. Practical Details

In this section we shall consider some of the practical problems that will arise when the method is implemented. For more detail the reader is referred to the programs in Part Two of this report.

1. Economics. Since the matrices X and F are never used by the algorithm, it is necessary to store only the matrices Y, P, G, and Q. The number of nonzero elements in these matrices is about $3n^2$; however, if they are stored conventionally as separate arrays, they will require about $4n^2$ locations. Since the lower part of the array in which G (or Y) is stored is zero, this part of the array can be used as a workspace in which ΔG and ΔY are formed and manipulated.

In assessing the amount of work involved, we assume that plane rotations are used for all reductions. We shall count the number of rotations and the number of multiplications, which multiplications corresponds roughly to the number of data accesses. The results are summarized below, where only the leading term of the count is given.

a. Secant Step

   rot = n-1,   mult = $3n^2$.

b. Function Evaluation

   rot = 0,   mult = $2n^2$.

c. Insertion and Updating (worst case in which y^* is inserted in the first column replacing y_{n+1})

   rot = n-1,   mult = $12n^2$.

d. Insertion and Updating (typical case in which y^* is inserted in the first column replacing y_{n+1})

   rot = n-1,   mult = $6n^2$.

e. Checking Degeneracy (computation of u)

   rot = n-1,   mult = $2.5n^2$.

f. <u>Fixing Degeneracy</u> (computation of v, evaluation of g^*, insertion of y^* and g^* [typical case])

$$\text{rot} = 2n-2, \qquad \text{mult} = 14.5n^2.$$

Thus a typical iteration without degeneracy will consist of a + b + 2d + e, or 3n-3 rotations and $19.5n^2$ multiplications. With degeneracy, a typical iteration will require 5n-5 rotations and $34n^2$ multiplications.

2. <u>Order of the columns of</u> Y <u>and</u> G. In forming ΔG preliminary to the computation of g^*, the vector g_1 is subtracted from the other columns of G. If $\|g_1\|$ is much larger than $\|g_i\|$, then the vector g_i will be overwhelmed by g_1. To avoid this we order the columns of G so that $\|g_1\| \le \|g_2\| \le \ldots \le \|g_{n+1}\|$. The matrix Y inherits this order, and since $\|f_i\| = \|g_i\|$, it may be presumed that when the process is converging, the vector x_i is nearer the solution than x_{i+1}. The ordering has the advantage that it gives a favorable operation count for the updates in the case when y^* replaces the column for which the norm of g is largest.

3. <u>Communication with the user</u>. The user must of course furnish code to evaluate the function f, which is customarily done in a subprogram provided by the user. After the secant prediction y^* has been calculated the user must decide whether the process has converged. If it has not, he must decide whether the predicted point is acceptable and if not what to do about it. Since no single strategy is likely to be effective in all cases, we have left a blank section in our implementation of the algorithm where the user may code his own decisions.

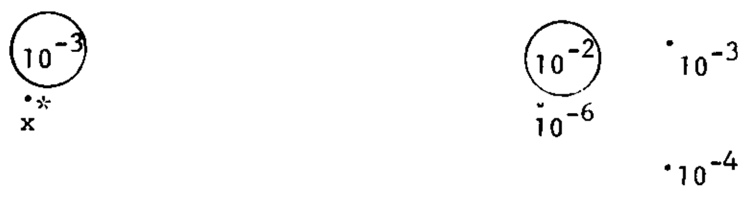
4. Obtaining initial factorizations. The updating algorithm can be used to obtain the factorizations (2.1) and (2.2) at the start of the algorithm. The user of course must furnish n+1 vectors $x_1, x_2, \ldots, x_{n+1}$ in the matrix X. At the k-th (k=0,1,...,n) step of the initialization procedure, assume that the factorizations of the matrices $X^{|k} = (x_1, \ldots, x_k)$ and $F^{|k} = (f_1, f_2, \ldots, f_k)$ are known; i.e.

$$X^{|k} = P^T Y^{|k}, \quad G^{k} = QF^{|k},$$

where $Y^{|k} = (y_1, \ldots, y_k)$ and $G^{|k} = (g_1, \ldots, g_k)$ are upper trapezoidal. Calculate the vectors $y_{k+1} = Px_{k+1}$ and $g_{k+1} = Qf_{k+1}$. Append a column to $Y^{|k}$ and $G^{|k}$ and insert y_{k+1} and g_{k+1}, making sure that the columns just appended are the ones to be discarded, and update as usual. After the n-th step all the vectors in X and F will have been incorporated into the factorization.

5. Using an old Jacobian. When a sequence of closely related problems are being solved, the solution of one may be a good approximation to that of the next. Moreover the approximation to the old Jacobian implicitly contained in the matrices Y, P, G, and Q may also be a good approximation to the new Jacobian. Unfortunately the new iteration cannot simply be started with the old matrices Y, P, G, and Q, as the following hypothetical example shows.

Consder the case illustrated below in which the numbers associated with the points give the norms of the function values.



The point labeled 10^{-6} is the converged value for the old iteration. When the process is restarted with the new function, the point will have a much higher function value, say the circled 10^{-2}. Consequently the prediction x^* will be far removed from the original points, and when y^* is inserted into Y, the array will be judged to be degenerate. Moreover the function value at x^* will have a norm (10^{-3} in the example) which is out of scale with the old values. Thus both the G and the Y arrays must rescaled before they can be used with the new function.

Our method of scaling consists of two steps. First the columns of ΔY are scaled so that their norms are equal to $\|y^*-y_1\|$. The modification is extended to G by linearity. Then, with g_1' denoting the new g value at y_1, the columns of G are increased by $g_1'-g_1$. This scaling technique is described below. The notation Insert(g,i,j) means insert g into column i of G, throwing out column j, then update as usual.

```
1. Calculate the new value $g_1'$ corresponding to $y_1$
2. $y^* = y_1 - \Delta Y(\Delta G)^{-1} g_1'$
3. For i=2,3,...,n+1
    1. $\omega_i = \|y^*-y_1\| / \|y_i-y_1\|$
    2. $y_i \leftarrow y_i + \omega_i(y_i-y_1)$
    3. $g_i \leftarrow g_i + \omega_i(g_i-g_1)$
4. Insert($g_1'-g_1$,1,1), multiplying the update transform-
   ations into $g_1$
5. $g_i = g_i + (g_1'-g_1)$, (i=2,3,...,n+1)
6. Insert($g_1'$,1,1)
```

It should be noted that statements 3.2 and 3.3 do not destroy the upper triangularity of the matrices Y and G, since only the first elements of y_1 and

g_1 are nonzero. Statements 4, 5, and 6 are a circumlocution designed to avoid excessive updating. Statement 4 transforms the system so that $g_1'-g_1$ is nonzero in only its first component, after which G may be altered without destroying its upper triangularity (statement 5). Statement 6 places g_1' in its rightful position.

The y^* predicted by the scaled Y and G will be the same as the y^* of statement 1. The columns of G need no longer be in order of increasing norm; but since all but the first represent old data, they should be discarded as soon as possible.

6. Incorporating linearities. As was mentioned in Section 1, degeneracies are certain to develop when some of the component functions are linear. Since the procedure for removing degeneracies is about as expensive as a secant step, it is important to be able to deal directly with such linearities. This may be done as follows.

Assume that $f: \mathbb{R}^{n+\ell} \rightarrow \mathbb{R}^n$, and that the equation $f(x) = 0$ is supplemented by ℓ linear equations of the form

$$Ax = b, \tag{4.1}$$

where $A \in \mathbb{R}^{(n+\ell)\times\ell}$ is of full rank. Suppose that we are given a unitary matrix U such that

$$AU = (0\ \ T) \tag{4.2}$$

where T is square. Set $\hat{x} = U^T x$ and partition $\hat{x}$ in the form $\hat{x} = (\hat{x}_1^T, \hat{x}_2^T)^T$, where $\hat{x}_2 \in \mathbb{R}^{\ell}$. Then from (4.1) and (4.2)

$$T\hat{x}_2 = b. \tag{4.3}$$

Since A is of full rank, T is nonsingular and any solution of the system (4.1) must have $\hat{x}_2 = T^{-1}b$.

Define the function $\hat{f}$: $\mathbb{R}^n \to \mathbb{R}^n$ by

$$\hat{f}(\hat{x}_1) = f\left[U\begin{pmatrix}\hat{x}_1\\ T^{-1}b\end{pmatrix}\right].$$

Then $\hat{f}(\hat{x}_1) = 0$ if and only if

$$x = U\begin{pmatrix}\hat{x}_1\\ T^{-1}b\end{pmatrix}$$

satisfies $f(x) = 0$ and $Ax = b$. The secant method may now be applied to $\hat{f}$.

The matrix U required by this process may be obtained in the usual way as the product of Householder transformations [5]. When this is done, the matrix T will be triangular, which makes the equation (4.3) easy to solve.

5. Numerical Examples and Conclusions

The algorithm described in the above sections has been tried on a variety of problems. Here we summarize the results of three tests that exhibit the typical behavior of the algorithm.

The first example involves the function whose i-th component is given by

$$f_i(x) = i - \sum_{j=1}^{i} x_i + q_i \sum_{j=i}^{n} (1-x_i)^2$$

This function has a solution at $\hat{x} = (1,1,\ldots,1)^T$. At the solution its Jacobian is the lower triangular matrix whose nonzero elements are all -1, a nicely conditioned matrix. The numbers q_i may be chosen ad libitum to make the function more or less nonlinear. Table one summarizes the results of applying the above algorithm ot this function with $n = 15$ and $q_i = .3$ $(i=1,2,\ldots,n)$. The initial estimate was the point $(0.8, 1.2, 0.8, 1.2,\ldots,0.8)^T$. The remaining 15 points required by the algorithm were obtained by adding alternately $\pm$ .05 to the successive components of the initial estimate. The results are summarized in Table 1, where $\|e\|$ denotes the Euclidean norm of the error in the current iterate, $\|f\|$ denotes the Euclidean norm of the current function value, and $\|\mu\|$ denotes the norm of the vector u used to check degeneracies. Of the starting values only the central one is reported. At three points it was necessary to rectify a degeneracy; otherwise the convergence is routine (the iteration was terminated when $\|f\| \leq 10^{-6}$).

The second example uses the same function with $n = 5$, $q_1 = q_2 = q_3 = q_4 = .5$ and $q_5 = 0$. The starting points are generated in the same way as for the first example. Since the fifth component of the function is linear, degeneracy can be expected in the iteration. It occurs at the seventh step ($\|\mu\| = 4.6 \cdot 10^3$) and is handled easily.

The third example tests the algorithm for reusing old information. The function depends on a parameter s and is defined by

$$f_i(x) = i \cdot s - \sum_{j=1}^{i} x_i + q_i \sum_{j=i}^{n} (s-x_i)^2.$$

With $n = 5$ and $q_i = .3$ the zero $(s,s,s,s,s)^T$ was found for s = 1.0, 1.2, 1.4, 1.6, 1.8, 2.0. The information from one solution was used to start the next. The results are summarized in Table three. The last three solutions are atypical in that they require effectively only a single iteration to converges. This is because the error vectors and the function values were the same at each new starting point, and this information had been preserved from the last solution.

These examples are given principally to illustrate the behavior of the algorithm. Additional experiments suggest that the local behavior of the method is quite good. Indeed if one believes that the algorithm for fixing degeneracies will work, one can apply the theory in [3] to give local convergence proofs. However, we believe it is too early to make general claims about the algorithm. For example, we do not know if damping techniques can be used to make it effective on problems where it otherwise would not work. It is hoped that the program described and listed in Part II of this report will help interested researchers to investigate the algorithm and compare it with others.

Table 1

$\|e\|$	$\|f\|$	$\|u\|$
$7.7\cdot10^{-1}$	$9.0\cdot10^{-1}$	$2.7\cdot10^{0}$
$1.3\cdot10^{-1}$	$3.1\cdot10^{-1}$	$1.2\cdot10^{2}$
$7.5\cdot10^{-1}$	$2.8\cdot10^{1}$	$1.4\cdot10^{1}$
$1.2\cdot10^{-2}$	$1.3\cdot10^{-2}$	$5.7\cdot10^{1}$
$2.9\cdot10^{-3}$	$4.7\cdot10^{-3}$	$6.2\cdot10^{2}$
$9.8\cdot10^{-3}$	$4.3\cdot10^{-1}$	$1.3\cdot10^{1}$
$2.4\cdot10^{-4}$	$2.8\cdot10^{-4}$	$1.5\cdot10^{2}$
$3.0\cdot10^{-3}$	$1.0\cdot10^{-2}$	$1.2\cdot10^{1}$
$1.1\cdot10^{-5}$	$3.3\cdot10^{-5}$	$2.4\cdot10^{1}$
$1.6\cdot10^{-6}$	$4.6\cdot10^{-6}$	$4.3\cdot10^{1}$
$4.3\cdot10^{-7}$	$1.5\cdot10^{-6}$	$2.5\cdot10^{1}$
$1.2\cdot10^{-7}$	$4.2\cdot10^{-7}$	$2.8\cdot10^{1}$

Table 2

$\|e\|$	$\|f\|$	$\|u\|$
$4.5\cdot10^{-1}$	$4.5\cdot10^{-1}$	$1.6\cdot10^{0}$
$7.9\cdot10^{-2}$	$1.1\cdot10^{-1}$	$2.6\cdot10^{1}$
$1.0\cdot10^{-2}$	$8.2\cdot10^{-3}$	$2.5\cdot10^{1}$
$3.6\cdot10^{-3}$	$4.1\cdot10^{-3}$	$7.5\cdot10^{1}$
$3.2\cdot10^{-4}$	$2.6\cdot10^{-4}$	$7.2\cdot10^{1}$
$1.0\cdot10^{-4}$	$1.3\cdot10^{-4}$	$1.2\cdot10^{1}$
$2.9\cdot10^{-6}$	$2.3\cdot10^{-6}$	$4.6\cdot10^{3}$
$1.0\cdot10^{-4}$	$3.4\cdot10^{-4}$	$5.0\cdot10^{0}$
$5.4\cdot10^{-8}$	$1.1\cdot10^{-7}$	$4.7\cdot10^{0}$

Table 3

$\|e\|$	$\|f\|$	$\|u\|$
$4.5\cdot10^{-1}$	$4.1\cdot10^{-1}$	$1.6\cdot10^{0}$
$4.3\cdot10^{-2}$	$6.5\cdot10^{-2}$	$2.6\cdot10^{1}$
$4.7\cdot10^{-3}$	$3.1\cdot10^{-3}$	$2.3\cdot10^{1}$
$1.4\cdot10^{-3}$	$1.2\cdot10^{-3}$	$1.7\cdot10^{2}$
$3.7\cdot10^{-3}$	$1.1\cdot10^{-2}$	$4.5\cdot10^{0}$
$2.9\cdot10^{-5}$	$3.9\cdot10^{-5}$	$6.9\cdot10^{0}$
$2.8\cdot10^{-6}$	$3.8\cdot10^{-6}$	$4.2\cdot10^{0}$
$7.0\cdot10^{-8}$	$9.0\cdot10^{-8}$	$5.6\cdot10^{0}$
$4.5\cdot10^{-1}$	$1.5\cdot10^{0}$	$1.0\cdot10^{1}$
$6.6\cdot10^{-2}$	$9.1\cdot10^{-2}$	$6.4\cdot10^{0}$
$2.5\cdot10^{-3}$	$2.1\cdot10^{-3}$	$9.6\cdot10^{1}$
$9.7\cdot10^{-4}$	$1.0\cdot10^{-3}$	$1.5\cdot10^{1}$
$2.5\cdot10^{-5}$	$2.3\cdot10^{-5}$	$1.1\cdot10^{2}$
$1.0\cdot10^{-3}$	$8.1\cdot10^{-4}$	$2.8\cdot10^{2}$
$9.9\cdot10^{-4}$	$8.1\cdot10^{-4}$	$2.4\cdot10^{0}$
$2.7\cdot10^{-7}$	$4.2\cdot10^{-7}$	$1.0\cdot10^{1}$
$4.5\cdot10^{-1}$	$1.5\cdot10^{0}$	$1.0\cdot10^{1}$
$5.1\cdot10^{-2}$	$6.7\cdot10^{-2}$	$3.3\cdot10^{0}$
$2.3\cdot10^{-3}$	$2.5\cdot10^{-3}$	$7.1\cdot10^{0}$
$1.7\cdot10^{-4}$	$1.7\cdot10^{-4}$	$1.9\cdot10^{1}$
$1.0\cdot10^{-6}$	$7.2\cdot10^{-7}$	$6.8\cdot10^{1}$
$4.5\cdot10^{-1}$	$1.5\cdot10^{0}$	$1.4\cdot10^{2}$
$6.7\cdot10^{-1}$	$1.5\cdot10^{0}$	$9.7\cdot10^{0}$
$1.2\cdot10^{-7}$	$1.5\cdot10^{-7}$	$1.5\cdot10^{1}$
$4.5\cdot10^{-1}$	$1.5\cdot10^{0}$	$1.5\cdot10^{1}$
$1.1\cdot10^{-7}$	$2.0\cdot10^{-7}$	$2.8\cdot10^{1}$
$4.5\cdot10^{-1}$	$1.5\cdot10^{0}$	$2.8\cdot10^{1}$
$8.0\cdot10^{-8}$	$1.5\cdot10^{-7}$	$5.6\cdot10^{1}$

PART II

1. Introduction

In this second part of this report we shall describe and list a program implementing the method described in Part I. Since the program is quite complex, the description is divided into two sections. The first section tells the casual user what he needs to know to use the program; the second section describes the program and its subroutines in greater detail and presupposes a familiarity with Part I.

2. Usage

SSM is a FORTRAN subroutine designed to solve the system of equations

$$f(x) = 0,$$

$$Ax = b,$$

where $f: \mathbb{R}^{n+\ell} \to \mathbb{R}^n$ and $A \in \mathbb{R}^{\ell \times (n+\ell)}$ (thus n is the number of nonlinear equations and ℓ is the number of linear equations in the system). The user must supply to the program the matrix A, the vector b and a subroutine to evaluate the function f. The user must also supply a set of n+1 estimates of the solution; however if a sequence of closely related problems is being solved, the output from the solution of one problem can be used in place of the estimates for the next problem. The user must also supply a section of code in SSM to check convergence.

Calling SSM. Information is transfered to SSM by the arguments in the subroutine call and by a common block. The calling sequence is

```
CALL SSM(X,F,N,L,EVAL,NEWJAC,NEWA,NEWB,FAIL).
```

The parameters in the calling sequence are

X(N+L)	a real array (of minimum dimension n) that on return contains the solution
F(N)	a real array that on return contains the value of f at X
N	n, which must be greater than one
L	ℓ, which may be zero
EVAL	the name of a user coded subroutine to evaluate f
NEWJAC	a logical variable which when true indicates that the user has provided a set of n+1 estimates in the common array Y. NEWJAC can be false only after SSM has been called at least once, in which case it tells SSM to use the results of the last run to start the current run
NEWA	A logical variable, which if true indicates that the coefficients of the system Ax = b have just been placed in the common array A. If the same coefficients are to be used in subsequent runs, NEWA must be false.
NEWB	A logical variable, which if true indicates that the elements of the righthand side of the system Ax = b have been placed in the common array B. If the same right-hand side is to be used in subsequent calls, NEWB must be false. If NEWA is true, SSM assumes that NEWB is also true.

FAIL	An integer which on return contains an error indicator. If FAIL is zero all has gone well. Otherwise FAIL contains an error trace (see §3 below).

The common block is

```
COMMON/SSMCOM/A(L,N+L+2),B(L),Y(N+L+2,N+1)
```

where the dimensions given are the minimal ones. As explained above A and B contain the coefficients and righthand side of the linear system and the columns of Y contain n+1 estimates of the solution. All of this information is altered by the system. If it is desired to use it later then NEWJAC, NEWA, or NEWB, whichever are appropriate, must be set to false.

The subroutine EVAL. The user must furnish a subroutine to evaluate the function. Its calling sequence is

```
CALL EVAL(X,F,FAIL),
```

The arguments are

X(N+L)	an array containing the point x to be evaluated
F(L)	an array that on return contains f(x)
FAIL	an integer that is initially zero. If a failure occurs it should be set to any integer from 1 through 99. This will cause SSM to abort. The last two digits in FAIL will contain the number set in EVAL.

If further information must be communicated to EVAL, this may be done through common statements.

Convergence and other tests. In its main loop, SSM produces a new approximation to the solution which must be tested for acceptability. Since no fixed strategy is likely to be satisfactory for all problems, the user is required to furnish his own tests in the section labled 500. This is also the place to insert *ad hoc* damping techniques and tests to insure that the iteration does not continue too long. Additional information can be communicated to this section by extending the argument list of SSM or by a common block.

In coding this section it is important to realize that SSM works in a coordinate system different from the x-f coordinate system of the user: call it the y-g coordinate system. To each n-vector y there corresponds a n+ℓ vector x satisfying $Ax = b$, which can be retrieved by the statement

```
CALL EVAL G(Y,X,F,G,GNRM,.TRUE.,EVAL,FAIL)
```

The vector x corresponding to y is returned in the array X. The arguments F,G,GNRM, and FAIL are irrelevent in this context. To each function value f there corresponds a value g. Given y, the set of vectors x, f, and g can be retrieved by the statement

```
CALL EVAL G(Y,X,F,G,GNRM,.FALSE.,EVAL,FAIL)
```

On return GNRM contains the Euclidean norm of g, which is approximately equal to the Euclidean norm of f. If FAIL is nonzero on return, it contains the value to which it was set in EVAL. The f value corresponding to a given g can be found by multiplying g by the transpose of the nxn matrix contained in the array Q.

When SSM enters the section labled 500, the arrays YY and GG contain the point from which the prediction was launched and its g-value, YS contains the predicted point, and DY contains the difference DY = YS - YY. The array element NORM(1) contains the Euclidean norm of GG. The arrays X, F, SN, CS, and GS may be used for scratch.

In this section the user must decide whether or not to continue the iteration. If he decides to continue he must provide an acceptable prediction in YS and its corresponding g value in GS, then transfer control to statement 600. It should be stressed that the value of YS need not be the same as the value that was input to the section. For example, YS may be taken to be YY + λDY, where λ is chosen so that the norm of GS is not too large.

Either convergence or an error may make the user decide to terminate the iteration. On normal convergence the user should first execute the statement

```
CALL EVAL G(YS,X,F,GS,GNRM,.FALSE.,EVAL,FAIL),
```

in order to place the converged x and f in X and F, and then return. On an error the user should return after executing the statement

```
FAIL = FAIL + k
```

where k = 10000·i (i=8,9,...).

<u>Parameters set in SSM</u>. Five parameters contained in the common block SECPRM are set at the beginning of SSM. The variable TOL contains a tolerance for detecting degeneracies (see §I.3). The variable NTRY contains an upper bound on the number of attempts to rectify degeneracies and is currently set to n. The variable SCL is set to .1 to handle a rather unlikely error in the

subroutine CHKFIX. The variable UTBND is set to N+3 and insures that any given point will not be used too long. The only parameter the user should have to fool with is MCHEPS, which contains the largest floating point number for which the computed value of 1. + MCHEPS is equal to 1. (Only a rough estimate of the value is needed; e.g. if the floating point fraction contains 27 bits then MCHEPS may be taken to be 10^{-8}.)

<u>Minimal dimensions</u>. SSM will of course not work if its arrays are too small for the problem. Here follows a list of subscripted variables in SSM with their minimal dimensions.

```
X(N+L),F(N+L),A(L,N+L+2),B(L),Y(N+L+2,N+1)
G(N+L+2,N+2),MARK(N+1),NORM(N+1),
P(N,N),Q(N,N).
```

In addition, the first dimensions of Y and G must be equal. The second argument in EVALG must be dimensioned at least N+L. All other arrays in the program must be dimensioned at least N.

3. <u>Program Details</u>

<u>General considerations</u>. The program consists of eight subroutines: SSM, the controlling program; CHKFIX, which detects and rectifies degeneracies; INSERT, which modifies and updates the matrices Y and G; SECSTP, which makes a secant prediction; EVALG, which calls the user coded function EVAL to get a function value; REDUCE, which accomplishes the reduction described in §1.2; HESRED, which triangularizes a Hessenberg matrix in G; and ROT which computes plane rotations.

These subroutines are linked by three common blocks. The block SECCOM contains variables that must be visible to the user. The block SECPRM contains parameters whose values should seldom have to be reset. The block SECVAR contains the remaining variables that are shared by the program.

The array names follow the nomenclature of Part I. In addition, the array NORM contains the Euclidean norms of the columns of G. The array MARK contains integers associated with the columns of Y and G that tell INSERT which columns must be thrown out (specifically if MARK(I) $\geq$ OUTBND, then CHKFIX and INSERT will attempt to discard column I before others with MARK $<$ OUTBND).

The program is provided with an error tracing feature that operates as follows. Each subroutine is assigned a power of ten, its failno. If an error occurs in a given subroutine, it executes the statement FAIL = FAIL + i*failno, where i=1,2,...,9. The calling subrouinte regards the return of a nonzero value in FAIL as an error and does the same thing. In this way the program is aborted with an integer in FAIL whose digits tell where an error occurred and how the program got there.

We shall now give a brief description of each of the subroutines.

SSM(X,F,NN,LL,EVAL,NEWJAC,NEWA,NEWB,FAIL); failno = 10^4. The calling sequence for this program has already been discussed. After some initialization, SSM checks for a new matrix of coefficients in A. If there is one, Householder transformations $H_1, H_2, \ldots, H_\ell$ are determined so that $AH_1 \ldots H_\ell = (0\ T)$ where T is upper triangular. The matrix A is overwritten in the array A by $H_1, \ldots H_\ell$ and by T (this requires two extra columns). If either NEWA or NEWB is true, the system $T\hat{x}_2 = b$ is solved, the solution overwriting b.

The iteration may be started either by using the Jacobian from a previous iteration or by building up a new Jacobian. The first alternative is effected by a straightforward implementation of the technique described in Section I.4.5. If the Jacobian has to be built up, it is done by the technique described in Section I.4.4.

In the main loop, the directions are checked for independence and a secant step is taken. After SSM emerges from the user coded testing section, the new point is inserted into the Y and G arrays (see the description of INSERT), the values in the array MARK are increased by unity to prevent a point from hanging on too long, and the loop is begun again.

CHKFIX(EVAL,FAIL); failno = 10^3. This is a fairly straightforward implementation of the algorithm described in (1.3.7), with some special features. The transpose of the Hessenberg matrix A is formed in the lower part of the array G starting in row three. If the columns of A are zero, the minimum in (I.3.7.3.3) is taken to be SCL$*\|y_1\|$. The matrix A is reduced to triangular form by HESRED, and all diagonal elements of A that are too small are set equal to MACHEPS.

The column to be thrown out is restricted by the array MARK. If some MARK(I) $\geq$ OUTBND) then the column K that is thrown out must satisfy MARK(K) $\geq$ OUTBND; otherwise any column with MARK ≥ 0 may be thrown out. The new column is given a MARK of zero and the elements of the array MARK are increased by unity.

INSERT(YS,GS,GNRM,OT,M). This subroutine inserts YS and GS in Y and G, treating Y and G as N by M arrays. The index of the column to be thrown out is specified by OT. If OT is zero, then the column of largest NORM is chosen, subject to the same MARK restrictions that govern CHKFIX. The new columns are inserted just before the first column of larger norm and are given a MARK of zero. The matrices Y, P, G, and Q are updated by REDUCE.

SECSTP(YY,GG,YS,DY,FAIL); failno = 10^2. This subroutine calculates DY = - ΔY*(ΔG)$^{-1}$*GG and the secant prediction YS = YY + DY. As in CHKFIX the lower part of G is used as a scratch array to contain the transpose of ΔG, which is reduced to triangular form by HESRED.

EVALG(YP,XP,FV,GV,GNRM,ONLYX,EVAL,FAIL). Given the point YP, this subroutine finds the corresponding x-vector XP, calls EVAL to obtain a function value FV, and converts FV into a vector GV in the g-coordinate system. If ONLYX is true, the routine returns before calling EVAL.

REDUCE(Y,P,IN,N,M). This subroutine reduces a matrix Y of dimension N$\times$M with a stalactite to triangular form via the method described in §I.2. The stalactite is assumed to be in column IN. The transformations are accumulated in P.

HESRED. This subroutine reduces a Hessenberg matrix to triangular form using plane rotations. The matrix is stored in the lower part of G starting in row three. The rotations are returned in the arrays CS and SN.

ROT(A,B,CS,SN,R). This subroutine computes plane rotations for REDUCE and HESRED.

4. Program

```
      SUBROUTINE SSM(X,F,NN,LL,EVAL,NEWJAC,NEWA,NEWB,FAIL)
C
C     PARAMETERS IN THE CALLING SEQUENCE.
C
      REAL F(20),X(20)
      INTEGER FAIL,LL,NN
      LOGICAL NEWA,NEWB,NEWJAC
      EXTERNAL EVAL
C
C     GLOBAL VARIABLES.
C
      COMMON /SECCOM/A(20,22),B(20),Y(22,21)
      COMMON /SECVAR/CS(20),G(22,22),L,LM1,MARK(21),N,N1,N2,
     1               NL,NL1,NL2,NM1,NM2,NORM(21),P(20,20),
     2               Q(20,20),RSQN,SN(20)
      COMMON /SECPRM/MCHEPS,NTRY,OUTBND,SCL,TOL
      REAL A,B,CS,G,MCHEPS,NORM,P,Q,RSQN,SCL,SN,TOL,Y
      INTEGER L,LM1,MARK,N,N1,N2,NL,NL1,NL2,NM1,NM2,NTRY,OUTBND
C
C     VARIABLES INTERNAL TO SSM.
C
      REAL DY(20),GNRM,GG(20),GS(20),MAX,OMEGA,OMEGA1,
     1     S,T,YY(20),YS(20)
      INTEGER I,I1,II,J,JJ,K,KK,KM1,NK
C
C     SET UP VALUES IN SECPRM.
C
      TOL = 100.
      NTRY = NN
      MCHEPS = 1.E-8
      SCL = .1
      OUTBND = N+3
C
C     INITIALIZATION.
C
      L = LL
      LM1 = L-1
      N = NN
      N1 = N+1
      N2 = N+2
      NL = N+L
      NL1 = NL+1
      NL2 = NL+2
      NM1 = N-1
      NM2 = N-2
      RSQN = 1./SQRT(FLOAT(N))
      FAIL = 0
C
C     CHECK FOR LINEAR SYSTEMS.
C
      IF(L.EQ.0) GO TO 200
C
C        PROCESS THE LINEAR SYSTEM.
C
         IF(.NOT.NEWA) GO TO 180
```

```
C
C        REDUCE THE MATRIX OF THE LINEAR SYSTEM BY
C        HOUSEHOLDER TRANSFORMATIONS.
C
         DO 170 KK=1,L
           K = L-KK+1
           NK = N+K
           MAX = 0.
           DO 110 J=1,NK
             MAX = AMAX1(MAX,ABS(A(K,J)))
  110      CONTINUE
           IF(MAX .NE. 0.) GO TO 120
             FAIL = 10000
             RETURN
  120      S = 0.
           DO 130 J=1,NK
             A(K,J) = A(K,J)/MAX
             S = S + A(K,J)**2
  130      CONTINUE
           S = SQRT(S)
           IF(A(K,NK) .LT. 0.) S = -S
           A(K,NK) = A(K,NK) + S
           A(K,NL1) = S*A(K,NK)
           A(K,NL2) = -MAX*S
           IF(K .EQ. 1) GO TO 170
           KM1 = K-1
           DO 160 I=1,KM1
             T = 0.
             DO 140 J=1,NK
               T = T + A(I,J)*A(K,J)
  140        CONTINUE
             T = T/A(K,NL1)
             DO 150 J=1,NK
               A(I,J) = A(I,J) - T*A(K,J)
  150        CONTINUE
  160      CONTINUE
  170    CONTINUE
  180  IF(.NOT.(NEWA .OR. NEWB)) GO TO 200
C
C        SOLVE THE TRIANGULAR SYSTEM FOR THE CONSTANT
C        PART OF THE TRANSFORMED SYSTEM.
C
         B(L) = B(L)/A(L,NL2)
         IF(L.EQ.1) GO TO 200
         DO 195 II=1,LM1
           I = L-II
           I1 = I+1
           DO 190 J=I1,L
             NJ = N+J
             B(I) = B(I) - A(I,NJ)*B(J)
  190      CONTINUE
           B(I) = B(I)/A(I,NL2)
  195    CONTINUE
C
C     CHECK THE STATUS OF THE APPROXIMATE JACOBIAN.
```

```
C
  200 IF(NEWJAC) GO TO 300
C
C       RESCALE THE OLD APPROXIMATE JACOBIAN.
C
        MARK(1) = 0.
        DO 205 I=1,N
          YY(I) = 0.
          MARK(I+1) = OUTBND
  205   CONTINUE
        YY(1) = Y(1,1)
        CALL EVALG(YY,X,F,GG,GNRM,.FALSE.,EVAL,FAIL)
        IF(FAIL .EQ. 0) GO TO 210
          FAIL = FAIL + 20000
          RETURN
  210   CALL SECSTP(YY,GG,YS,DY,FAIL)
        IF(FAIL .EQ. 0) GO TO 215
          FAIL = FAIL + 30000
          RETURN
  215   S = (YS(1) - Y(1,1))**2
        DO 220 I=2,N
          S = S + YS(I)**2
  220   CONTINUE
        S = SQRT(S)
        DO 240 J=2,N1
          T = (Y(1,J) - Y(1,1))**2
          DO 225 I=2,N
            T = T + Y(I,J)**2
  225     CONTINUE
          IF(T .NE. 0) GO TO 230
            FAIL = 40000
            RETURN
  230     OMEGA = S/SQRT(T)
          OMEGA1 = 1. - OMEGA
          NORM(J) = SQRT((OMEGA1*NORM(1))**2 +
     1                   2.*OMEGA1*OMEGA*G(1,1)*G(1,J) +
     2                   (OMEGA*NORM(J))**2)
          Y(1,J) = OMEGA1*Y(1,1) + OMEGA*Y(1,J)
          G(1,J) = OMEGA1*G(1,1) + OMEGA*G(1,J)
          NJ = MIN0(N,J)
          DO 235 I=2,NJ
            Y(I,J) = OMEGA*Y(I,J)
            G(I,J) = OMEGA*G(I,J)
  235     CONTINUE
  240   CONTINUE
        DO 245 I=1,N
          G(I,N2) = GG(I)
  245   CONTINUE
        GG(1) = GG(1) - G(1,1)
        CALL INSERT(YY,GG,0.,1,N2)
        DO 250 J=2,N1
          NORM(J) = SQRT(G(1,1)**2 +
     1                   2.*G(1,1)*G(1,J) +
     2                   NORM(J)**2)
          G(1,J) = G(1,J) + G(1,1)
```

```
    250   CONTINUE
          DO 255 I=1,N
            GG(I) = G(I,N2)
    255   CONTINUE
          CALL INSERT(YY,GG,GNRM,1,N1)
          GO TO 400
    300 CONTINUE
C
C         THE APPROXIMATE JACOBIAN IS TO BE FORMED FROM A
C         NEW SET OF POINTS.  BUILD UP THE MATRICES Y,P,G, AND Q.
C
          IF(L .EQ. 0) GO TO 325
C
C           THERE ARE LINEAR EQUATIONS. TRANSFORM
C           THE POINTS.
C
            DO 320 KK=1,L
              K = L-KK+1
              NK = N+K
              DO 315 J=1,N1
                T = 0.
                DO 305 I=1,NK
                  T = T + A(K,I)*Y(I,J)
    305         CONTINUE
                T = T/A(K,NL1)
                DO 310 I=1,NK
                  Y(I,J) = Y(I,J) - T*A(K,I)
    310         CONTINUE
    315       CONTINUE
    320     CONTINUE
    325   DO 350 I=1,N
            DO 340 J=1,N
              P(I,J) = 0.
              Q(I,J) = 0.
    340     CONTINUE
            P(I,I) = 1.
            Q(I,I) = 1.
    350   CONTINUE
          DO 380 K=1,N1
            DO 370 I=1,N
              YY(I) = 0.
              DO 360 J=1,N
                YY(I) = YY(I) + P(I,J)*Y(J,K)
    360       CONTINUE
    370     CONTINUE
            CALL EVALG(YY,X,F,GG,GNRM,.FALSE.,EVAL,FAIL)
            IF(FAIL .EQ. 0) GO TO 373
              FAIL = FAIL + 50000
              RETURN
    373     NORM(K) = GNRM
            CALL INSERT(YY,GG,GNRM,K,K)
    380   CONTINUE
C
C       MAIN LOOP.  OBTAIN A SET OF AFFINELY INDEPENDENT
C       POINTS AND THEN TAKE A SECANT STEP.
```

```
C
  400 CALL CHKFIX(EVAL,FAIL)
      IF(FAIL .EQ. 0) GO TO 405
        FAIL = FAIL + 60000
        RETURN
  405 DO 410 I=1,N
        YY(I) = 0.
        GG(I) = 0.
  410 CONTINUE
      YY(1) = Y(1,1)
      GG(1) = G(1,1)
      CALL SECSTP(YY,GG,YS,DY,FAIL)
      IF(FAIL .EQ. 0) GO TO 500
        FAIL = FAIL + 70000
        RETURN
C
C     ON ENTRY TO THIS PART OF THE PROGRAM, YS CONTAINS
C     A NEW POINT.  IT IS THE RESPONSIBILITY OF THE
C     USER TO PROVIDE CODE THAT DETERMINES WHETHER YS IS
C     ACCEPTABLE AND WHETHER THE ITERATION HAS CONVERGED.
C     ON EXIT (OTHER THAN A RETURN), YS AND GS MUST
C     CONTAIN AN ACCEPTABLE POINT AND ITS VALUE.
C     THE SAMPLE SECTION BELOW RETURNS IF  THE NORM
C     OF THE FUNCTION IS LESS THAN OR EQUAL TO 1.0E-6.
C     BEFORE RETURNING INSERT AND CHKFIX ARE CALLED TO
C     INSURE THAT THE LATEST APPROXIMATION TO THE
C     JACOBIAN IS CONTAINED IN THE ARRAYS Y,P,G, AND Q.
C
  500 CALL EVALG(YS,X,F,GS,GNRM,.FALSE.,EVAL,FAIL)
      IF(FAIL .EQ. 0) GO TO 510
        FAIL = FAIL + 80000
        RETURN
  510 IF(GNRM .GT. 1.E-6) GO TO 600
        CALL INSERT(YS,GS,GNRM,0,N1)
        CALL CHKFIX(EVAL,FAIL)
        IF(FAIL .NE. 0) FAIL = FAIL + 90000
        RETURN
C
C     INSERT THE NEW POINT AND GO BACK FOR ANOTHER.
C
  600 CALL INSERT(YS,GS,GNRM,0,N1)
      DO 610 I=1,N1
        MARK(I) = MARK(I) + 1
  610 CONTINUE
      GO TO 400
      END
      SUBROUTINE CHKFIX(EVAL,FAIL)
C
C     PARAMETERS IN THE CALLING SEQUENCE.
C
      INTEGER FAIL
      EXTERNAL EVAL
C
C     GLOBAL VARIABLES.
C
      COMMON /SECCOM/A(20,22),B(20),Y(22,21)
      COMMON /SECVAR/CS(20),G(22,22),L,LM1,MARK(21),N,N1,N2,
     1               NL,NL1,NL2,NM1,NM2,NORM(21),P(20,20),
     2               Q(20,20),RSQN,SN(20)
      COMMON /SECPRM/MCHEPS,NTRY,OUTBND,SCL,TOL
      REAL A,B,CS,G,MCHEPS,NORM,P,Q,RSQN,SCL,SN,TOL,Y
      INTEGER L,LM1,MARK,N,N1,N2,NL,NL1,NL2,NM1,NM2,NTRY,OUTBND
C
C     VARIABLES INTERNAL TO CHKFIX.
C
      REAL F(20),GNRM,GS(20),MINNRM,NRM,S,T,U(20),UMAX,UNRM,V(20),
     1     VNRM,X(20),YS(20)
      INTEGER I,I1,II,IM1,J,J1,JU,OUT,OUTSET,TRY
      LOGICAL NRMSET
      EQUIVALENCE (CS(1),YS(1)),(SN(1),GS(1)),(U(1),X(1)),
     1            (V(1),F(1))
C
C     TRY NTRY TIMES TO OBTAIN AN AFFINELY INDEPENDENT
C     SET OF DIRECTIONS.
C
      DO 600 TRY=1,NTRY
C
C       DETERMINE WHICH VECTORS MAY BE THROWN OUT.
C
        OUTSET = 0
        DO 10 I=1,N1
          OUTSET = MAX0(MARK(I),OUTSET)
   10   CONTINUE
        IF(OUTSET .LT. OUTBND) OUTSET = 0
C
C       FORM THE TEST MATRIX IN THE SCRATCH AREA OF G.
C
  100   NRMSET = .FALSE.
        DO 130 J=2,N1
          J1 = J+1
          G(J1,1) = Y(1,J) - Y(1,1)
          NRM = G(J1,1)**2
          JU = MIN0(J,N)
          DO 110 I=2,JU
            G(J1,I) = Y(I,J)
            NRM = NRM + G(J1,I)**2
  110     CONTINUE
          IF(NRM .EQ. 0.) GO TO 130
            NRM = SQRT(NRM)
            IF(.NOT. NRMSET) MINNRM = NRM
            NRMSET = .TRUE.
      MINNRM = AMIN1(NRM,MINNRM)
      DO 120 I=1,JU
       G(J1,I) = G(J1,I)/NRM
 120  CONTINUE
 130 CONTINUE
    IF(.NOT. NRMSET) MINNRM = SCL*Y(1,1)
    IF(MINNRM .NE. 0.) GO TO 200
     FAIL = 1000
     RETURN
C
C   SOLVE FOR U AND TEST FOR U LARGE.
C
 200 CALL HESRED
    DO 210 I=1,N
     IF(ABS(G(I+2,I)) .LT. MCHEPS) G(I+2,I) = MCHEPS
 210 CONTINUE
    U(N) = RSQN/G(N2,N)
    UNRM = U(N)**2
    DO 230 II=2,N
     I = N-II+1
     I1 = I+1
     S = 0.
     DO 220 J=I1,N
      S = S - G(J+2,I)*U(J)
 220  CONTINUE
     U(I) = RSQN
     IF(S .LT. 0.) U(I) = -RSQN
     U(I) = (U(I) + S)/G(I+2,I)
     UNRM = UNRM + U(I)**2
 230 CONTINUE
    UNRM = SQRT(UNRM)
    IF(UNRM .LE. TOL) RETURN
C
C   THE DIRECTIONS ARE AFFINELY DEPENDENT.  DETERMINE
C   WHICH ONE TO THROW OUT.
C
 300 UMAX = 0.
    DO 310 I=2,N1
     IF(MARK(I).LT.OUTSET .OR. UMAX.GT.ABS(U(I-1)))
   1        GO TO 310
      OUT = I
      UMAX = ABS(U(I-1))
 310 CONTINUE
C
C   SOLVE FOR V.
C
 400 V(1) = 1./G(3,1)
    VNRM = V(1)**2
    DO 420 I=2,N
     S = 0.
     IM1 = I-1
     DO 410 J=1,IM1
      S = S - G(I+2,J)*V(J)
 410 CONTINUE
     V(I) = 1.
```

```
          IF(S .LT. 0.) V(I) = -1.
          V(I) = (V(I) + S)/G(I+2,I)
          VNRM = VNRM + V(I)**2
  420   CONTINUE
        VNRM = SQRT(VNRM)
        DO 430 II=1,NM1
          I = N-II
          T = CS(I)*V(I) - SN(I)*V(I+1)
          V(I+1) = (CS(I)*V(I+1) + SN(I)*V(I))/VNRM
          V(I) = T
  430   CONTINUE
        V(1) = V(1)/VNRM
C
C       COMPUTE THE NEW POINT AND INSERT IT.
C
  500   YS(1) = Y(1,1) + MINNRM*V(1)
        DO 510 I=2,N
          YS(I) = MINNRM*V(I)
  510   CONTINUE
        CALL EVALG(YS,X,F,GS,GNRM,.FALSE.,EVAL,FAIL)
        IF(FAIL .EQ. 0) GO TO 520
          FAIL = FAIL + 2000
          RETURN
  520   CALL INSERT(YS,GS,GNRM,OUT,N1)
        DO 530 I=2,N1
          MARK(I) = MARK(I) + 1
  530   CONTINUE
  600 CONTINUE
      FAIL = 3000
      RETURN
      END
      SUBROUTINE INSERT(YS,GS,GNRM,OT,M)
C
C     PARAMETERS IN THE CALLING SEQUENCE.
C
      REAL GNRM,GS(20),YS(20)
      INTEGER M,OT
C
C     GLOBAL VARIABLES.
C
      COMMON /SECCOM/A(20,22),B(20),Y(22,21)
      COMMON /SECVAR/CS(20),G(22,22),L,LM1,MARK(21),N,N1,N2,
     1               NL,NL1,NL2,NM1,NM2,NORM(21),P(20,20),
     2               Q(20,20),RSQN,SN(20)
      COMMON /SECPRM/MCHEPS,NTRY,OUTBND,SCL,TOL
      REAL A,B,CS,G,MCHEPS,NORM,P,Q,RSQN,SCL,SN,TOL,Y
      INTEGER L,LM1,MARK,N,N1,N2,NL,NL1,NL2,NM1,NM2,NTRY,OUTBND
C
C     VARIABLES INTERNAL TO INSERT.
C
      REAL MAXNRM
      INTEGER I,IN,IN1,INM1,IU,J,JJ,OUT,OUTSET
C
C     INITIALIZE THE Y AND G ARRAYS.
C
      IU = MIN0(M,NM1)
      DO 10 I=1,IU
        G(I+1,I) = 0.
        G(I+2,I) = 0.
        Y(I+1,I) = 0.
        Y(I+2,I) = 0.
   10 CONTINUE
C
C     DETERMINE WHICH COLUMN IS TO BE THROWN OUT.
C
  100 OUT = OT
      IF(OUT .NE. 0) GO TO 150
C
C       AMONG THE POSSIBLE CANDIDATES CHOOSE THE COLUMN
C       WITH LARGEST G NORM.
C
        OUTSET = 1
        DO 110 I=1,M
          OUTSET = MAX0(MARK(I),OUTSET)
  110   CONTINUE
        IF(OUTSET .LT. OUTBND) OUTSET = 0
        OUT = M
        MAXNRM = 0.
        DO 120 I=1,M
          IF(MAXNRM.GT.NORM(I) .OR. MARK(I).LT.OUTSET)
     1          GO TO 120
            MAXNRM = NORM(I)
            OUT = I
  120   CONTINUE
  150 CONTINUE
C
```

```
C     THE VECTORS ARE TO BE INSERTED JUST BEFORE THE
C     FIRST COLUMN OF LARGER NORM.
C
      DO 160 IN=1,M
        IF(GNRM .LE. NORM(IN)) GO TO 200
  160 CONTINUE
      IN = M+1
C
C     SHIFT THE COLUMNS AND INSERT THE NEW COLUMN.
C
  200 IF(IN .EQ. OUT) GO TO 260
C
C       SHIFT THE COLUMNS
C
        IF(IN .GT. OUT) GO TO 230
C
C         RIGHT SHIFT.
C
          IN1 = IN+1
          DO 220 JJ=IN1,OUT
            J = OUT-JJ+IN1
            DO 210 I=1,N
              Y(I,J) = Y(I,J-1)
              G(I,J) = G(I,J-1)
  210       CONTINUE
            MARK(J) = MARK(J-1)
            NORM(J) = NORM(J-1)
  220     CONTINUE
          GO TO 260
  230   CONTINUE
C
C         LEFT SHIFT.
C
          IN = IN-1
          IF(IN .EQ. OUT) GO TO 260
          INM1 = IN-1
          DO 250 J=OUT,INM1
            DO 240 I=1,N
              Y(I,J) = Y(I,J+1)
              G(I,J) = G(I,J+1)
  240       CONTINUE
            MARK(J) = MARK(J+1)
            NORM(J) = NORM(J+1)
  250     CONTINUE
  260 CONTINUE
C
C     INSERT THE NEW COLUMNS.
C
      DO 270 I=1,N
        Y(I,IN) = YS(I)
        G(I,IN) = GS(I)
  270 CONTINUE
      NORM(IN) = GNRM
C
C     REDUCE THE MATRICES.
```

```
C
   300 CALL REDUCE(Y,P,IN,N,M)
       CALL REDUCE(G,Q,IN,N,M)
       MARK(IN) = 0
       RETURN
       END
```

```
      SUBROUTINE SECSTP(YY,GG,YS,DY,FAIL)
C
C     PARAMETERS IN THE CALLING SEQUENCE.
C
      REAL DY(20),GG(20),YS(20),YY(20)
      INTEGER FAIL
C
C     GLOBAL VARIABLES.
C
      COMMON /SECCOM/A(20,22),B(20),Y(22,21)
      COMMON /SECVAR/CS(20),G(22,22),L,LM1,MARK(21),N,N1,N2,
     1              NL,NL1,NL2,NM1,NM2,NORM(21),P(20,20),
     2              Q(20,20),RSQN,SN(20)
      COMMON /SECPRM/MCHEPS,NTRY,OUTBND,SCL,TOL
      REAL A,B,CS,G,MCHEPS,NORM,P,Q,RSQN,SCL,SN,TOL,Y
      INTEGER L,LM1,MARK,N,N1,N2,NL,NL1,NL2,NM1,NM2,NTRY,OUTBND
C
C     VARIABLES INTERNAL TO SECSTP.
C
      REAL S
      INTEGER I,I1,II,J,JL,JU
C
C     FORM THE G-DIFFERENCE MATRIX IN THE LOWER PART OF G.
C
      DO 20 J=1,N
         JU = MIN0(N,J+1)
         DO 10 I=1,JU
            G(J+2,I) = G(I,J+1)
   10    CONTINUE
         G(J+2,1) = G(J+2,1) - G(1,1)
         YS(J) = GG(J)
   20 CONTINUE
C
C     SOLVE THE G-DIFFERENCE SYSTEM.
C
  100 CALL HESRED
      DO 110 I=1,NM1
         I1 = I+1
         T = YS(I)*CS(I) + YS(I1)*SN(I)
         YS(I1) = YS(I1)*CS(I) - YS(I)*SN(I)
         YS(I) = T
  110 CONTINUE
      IF(G(N2,N) .NE. 0.) GO TO 115
        FAIL = 100
        RETURN
  115 YS(N) = YS(N)/G(N2,N)
      DO 130 II=2,N
         I = N-II+1
         I1 = I+1
         DO 120 J=I1,N
            YS(I) = YS(I) - G(J+2,I)*YS(J)
  120    CONTINUE
         IF(G(I+2,I) .NE. 0.) GO TO 125
           FAIL = 200
           RETURN
  125    YS(I) = YS(I)/G(I+2,I)
  130 CONTINUE
C
C     CALCULATE DY.
C
  200 S = 0.
      DO 220 I=1,N
         S = S + YS(I)
         JL = MAX0(I,2)
         DY(I) = 0.
         DO 210 J=JL,N1
              DY(I) = DY(I) - Y(I,J)*YS(J-1)
  210    CONTINUE
  220 CONTINUE
      DY(1) = DY(1) + S*Y(1,1)
C
C     CALCULATE YS.
C
  300 DO 310 I=1,N
         YS(I) = YY(I) + DY(I)
  310 CONTINUE
      RETURN
      END
      SUBROUTINE EVALG(YP,XP,FV,GV,GNRM,ONLYX,EVAL,FAIL)
C
C     PARAMETERS IN THE CALLING SEQUENCE.
C
      REAL GNRM,FV(20),GV(20),XP(20),YP(20)
      INTEGER FAIL
      LOGICAL ONLYX
      EXTERNAL EVAL
C
C     GLOBAL VARIABLES.
C
      COMMON /SECCOM/A(20,22),B(20),Y(22,21)
      COMMON /SECVAR/CS(20),G(22,22),L,LM1,MARK(21),N,N1,N2,
     1               NL,NL1,NL2,NM1,NM2,NORM(21),P(20,20),
     2               Q(20,20),RSQN,SN(20)
      COMMON /SECPRM/MCHEPS,NTRY,OUTBND,SCL,TOL
      REAL A,B,CS,G,MCHEPS,NORM,P,Q,RSQN,SCL,SN,TOL,Y
      INTEGER L,LM1,MARK,N,N1,N2,NL,NL1,NL2,NM1,NM2,NTRY,OUTBND
C
C     VARIABLES LOCAL TO EVALG.
C
      REAL T
      INTEGER I,J,K,NI,NK
C
C     TRANSFORM YP INTO THE X COORDINATE SYSTEM.
C
      DO 20 I=1,N
        XP(I) = 0.
        DO 10 J=1,N
          XP(I) = XP(I) + P(J,I)*YP(J)
   10   CONTINUE
   20 CONTINUE
C
C     IF THERE ARE LINEAR EQUATIONS, SET THE LAST OF XP
C     TO THE CONSTANT PART AND TRANSFORM INTO THE INITIAL
C     X COORDINATE SYSTEM.
C
      IF(L .EQ. 0) GO TO 100
        DO 30 I=1,L
          NI = N+I
          XP(NI) = B(I)
   30   CONTINUE
        DO 60 K=1,L
          NK = N+K
          T = 0.
          DO 40 I=1,NK
            T = T + A(K,I)*XP(I)
   40     CONTINUE
          T = T/A(K,NL1)
          DO 50 I=1,NK
            XP(I) = XP(I) - T*A(K,I)
   50     CONTINUE
   60   CONTINUE
C
C     IF ONLY XP IS REQUIRED, RETURN.
```

```
C
  100 IF(ONLYX) RETURN
C
C     EVALUATE THE FUNCTION
C
      CALL EVAL(XP,FV,FAIL)
      IF(FAIL .NE. 0) RETURN
C
C     TRANSFORM FV INTO THE G COORDINATE SYSTEM.
C
  200 GNRM = 0.
      DO 220 I=1,N
        GV(I) = 0.
        DO 210 J=1,N
          GV(I) = GV(I) + Q(I,J)*FV(J)
  210   CONTINUE
        GNRM = GNRM + GV(I)**2
  220 CONTINUE
      GNRM = SQRT(GNRM)
      RETURN
      END
      SUBROUTINE REDUCE(Y,P,IN,N,M)
C
C     PARAMETERS IN THE CALLING SEQUENCE.
C
      REAL Y(22,21),P(20,20)
      INTEGER IN,M,N
C
C     VARIABLES INTERNAL TO REDUCE.
C
      REAL CS,R,SN,T
      INTEGER I,I1,II,IN2,IU,J
      IN2 = IN+2
      IF(IN+1.GE.N) GO TO 50
C
C       REDUCE THE STALAGTITE.
C
        DO 40 II=IN2,N
          I1 = N-II+IN2
          I = I1-1
          IF(Y(I1,IN) .EQ. 0.) GO TO 40
          CALL ROT(Y(I,IN),Y(I1,IN),CS,SN,R)
          Y(I1,IN) = 0.
          Y(I,IN) = R
          IF(I.GT.M) GO TO 20
            DO 10 J=I,M
              T = CS*Y(I,J) + SN*Y(I1,J)
              Y(I1,J) = CS*Y(I1,J) - SN*Y(I,J)
              Y(I,J) = T
   10       CONTINUE
   20     CONTINUE
          DO 30 J=1,N
            T = CS*P(I,J) + SN*P(I1,J)
            P(I1,J) = CS*P(I1,J) - SN*P(I,J)
            P(I,J) = T
   30     CONTINUE
   40   CONTINUE
   50 CONTINUE
C
C     REDUCE FROM HESSENBERG TO TRAPEZIODAL FORM.
C
      IU = MIN0(M,N-1)
      DO 100 I=1,IU
        I1 = I+1
        IF(Y(I1,I) .EQ. 0.) GO TO 100
        CALL ROT(Y(I,I),Y(I1,I),CS,SN,R)
        Y(I,I) = R
        Y(I1,I) = 0.
        IF(I1 .GT. M) GO TO 80
          DO 70 J=I1,M
            T = CS*Y(I,J) + SN*Y(I1,J)
            Y(I1,J) = CS*Y(I1,J) - SN*Y(I,J)
            Y(I,J) = T
   70     CONTINUE
   80   CONTINUE
        DO 90 J=1,N
```

```
           T = CS*P(I,J) + SN*P(I1,J)
           P(I1,J) = CS*P(I1,J) - SN*P(I,J)
           P(I,J) = T
   90    CONTINUE
  100 CONTINUE
      RETURN
      END
      SUBROUTINE HESRED
C
C     GLOBAL VARIABLES.
C
      COMMON /SECCOM/A(20,22),B(20),Y(22,21)
      COMMON /SECVAR/CS(20),G(22,22),L,LM1,MARK(21),N,N1,N2,
     1               NL,NL1,NL2,NM1,NM2,NORM(21),P(20,20),
     2               Q(20,20),RSQN,SN(20)
      COMMON /SECPRM/MCHEPS,NTRY,OUTBND,SCL,TOL
      REAL A,B,CS,G,MCHEPS,NORM,P,Q,RSQN,SCL,SN,TOL,Y
      INTEGER L,LM1,MARK,N,N1,N2,NL,NL1,NL2,NM1,NM2,NTRY,OUTBND
C
C     VARIABLES INTERNAL TO HESRED.
C
      REAL R,T
      INTEGER I,K,K1,K3
      DO 20 K=1,NM1
        K1 = K+1
        CALL ROT(G(K+2,K),G(K+2,K1),CS(K),SN(K),R)
        G(K+2,K) = R
        G(K+2,K1) = 0.
        K3 = K+3
        DO 10 I=K3,N2
          T = CS(K)*G(I,K) + SN(K)*G(I,K1)
          G(I,K1) = CS(K)*G(I,K1) - SN(K)*G(I,K)
          G(I,K) = T
   10   CONTINUE
   20 CONTINUE
      RETURN
      END
      SUBROUTINE ROT(A,B,CS,SN,R)
      REAL A,B,CS,SN,R,AA,BB,ETA
      ETA = AMAX1(ABS(A),ABS(B))
      IF(ETA .NE. 0.) GO TO 10
        R = 0.
        CS = 1.
        SN = 0.
        RETURN
   10 CONTINUE
      AA = A/ETA
      BB = B/ETA
      R = SQRT(AA**2 + BB**2)
      CS = AA/R
      SN = BB/R
      R = R*ETA
      RETURN
      END
```

References

1. R. H. Bartels, J. Stoer, and Ch. Zenger, A realization of the simplex method based on triangular decomposition, in Handbook for Automatic Computation II. Linear Algebra (J. H. Wilkinson and C. Reinsch, eds.), 152-190, Springer, New York, 1971.

2. R. P. Brent, On maximizing the efficiency of algorithms for solving systems of nonlinear equations, IBM Research RC 3725, Yorktown Heights, 1972.

3. J. M. Ortega and W. C. Rheinboldt, Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, New York, 1970.

4. P. Wolfe, The secant method for simultaneous nonlinear equations, Comm. ACM 2 (1959) 12-13.

5. J. H. Wilkinson, The Algebraic Eigenvalue Problem, Clarendon, Oxford, 1965.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A STABLE VARIANT OF THE SECANT METHOD FOR SOLVING NONLINEAR EQUATIONS		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) W. B. Gragg and G. W. Stewart		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-67-A-0128-0018
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE April, 1974
		13. NUMBER OF PAGES 57
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Mathematics Program Washington, D. C. 20360		15. SECURITY CLASS. *(of this report)* unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The usual successive secant method for solving systems of nonlinear equations suffers from two kinds of instabilities. First the formulas used to update the current approximation to the inverse Jacobian are numerically unstable. Second, the directions of search for a solution may collapse into a proper affine subspace, resulting at best in slowed convergence and at worst in complete failure of the algorithm. In this report it is shown how the numerical instabilities can be avoided by working with factorizations of matricies appearing in the algorithm. Moreover, these factorizations can be used to detect and remedy degenaracies among the directions. A second part lists a program implementing the algorithm described.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*