On the Structure of Zero Finders.

*Gaston H Gonnet**

Department of Computer Science
University of Waterloo

Research Report CS-76-15

March 1976

Abstract.

Zero finders written in FORTRAN usually impose a false and inconvenient structure on the program. Desirable software and numerical features of a zero finder are discussed. A zero finder with a new structure is presented together with a number of examples that illustrate its flexibility. A similar routine, that uses an extended secant sheme, for systems of nonlinear equations is presented and discussed.

Key words and phrases: nonlinear equations, zero finding, structure in FORTRAN, systems of nonlinear equations, secant method.

CR Categories: 5.15.

(*) This work was supported by the Department of Communications, National Research Council and Defense Research Board of the government of Canada and by the University of Waterloo.

1. Introduction.

One of the basic tools in the art of programming is the ability to isolate problems and solve them in an efficient and reliable way. Using these reliable bricks, the software architect can build more complex structures. This is the case with elementary functions such as: sqrt, exp, arctan, etc. However for several well known reasons, this is not the case for zero finders. Most of the currently used zero finders present the same structure, namely: the zero finder accepts as a parameter, among others, an external function; upon exit, it returns a "root" of the function. This scheme has several drawbacks, the most important being: the function has to be written in a separate subprogram generating an extra level of code which has to share variables with the calling sequence; stopping criteria, range control, etc., are fixed and often do not fit the user's needs. After many years of fruitful numerical analysis and structured programming, we should be able to devise a more flexible and general routine.

Section 2 contains a list of desired features; section 3 presents ROOT1, an algorithm with some of the desired properties; and section 4 presents ROOT, an extension of ROOT1 to systems of equations.

2. Desired Properties.

In this section we discuss a number of desired or undesired features that we want to include or avoid in a zero finder. Most of them are obvious, while others may be controversial. Our main interest is focused on the software properties rather than the numerical ones, which are extensively discussed in the current literature (Dahlquist [5], Ralston [9]).

2.1 - Structure

The most common structure of zero finders forces the code to be divided into a calling program and a subroutine, thus requiring three levels:

```
program calls zero-finder (..params..,function)
    zero-finder calls function
        function
```

In languages like FORTRAN, where the subprograms cannot easily share global variables and/or be defined in-line, this division becomes inconvenient. Since the function belongs logically to the calling program, its parameters must be transferred via common statements, and one must suffer the usual nuisances.

Some related problems arise when we try: A) to find a zero of a function that, for its evaluation, requires the zero of another function; B) to solve several different equations in parallel, (for example one iteration per function or like cooperating processes); C) to find a zero of a function with certain accuracy and, later on, wish to increase the accuracy without losing the previous information. Note that, with the usual structure, A will require two different copies of the zero finder, and B and C are not possible to satisfy.

Most of the problems regarding structure and control of the variables are related to the fact that the zero finder retains control throughout the whole process.

2.2 - Self starting

The most reliable algorithms are based on mixed strategies that start with two points, whose function values are opposite in sign, and can guarantee some kind of convergence (Brent [2], Bus [4]). Sometimes we are faced with a problem for which we know there exists a solution near a certain point, but providing an interval with a change in sign is a very difficult task, equivalent to finding the zero. This will usually imply function evaluations whose numerical information is lost. In an extreme case we

may be faced with solving a hidden $f(x)^2=0$; here we can't provide such an interval, although the solution is well defined. Our goal will be achieved with a subprogram that either starts with two points opposite in sign and, consequently, has good reliability properties, or starts with a single point, performs its best, and upon a change in sign enters the reliable mode.

2.3 - Accuracy

It is generally agreed that five parameters may be needed to control accuracy (or running time of the zero finder). These are: tolerances (absolute and/or relative) in the argument and in the function value, and number of iterations. A logical "OR" of these conditions is used to halt the execution. This is clearly rigid since we may want to test for other conditions or for a different logical expression of the conditions (e.g. $|f(x)| <$ epsf .and. $|x\text{-root}| <$ epsx).

2.4 - Evaluation savings

In some cases we know the function value at certain points (poles, singularities, limits, etc.), that we would like to feed into the zero finder as initial information to improve the first estimates. This may imply function evaluation savings and avoid function evaluations at points where numerical difficulties exist.

2.5 - Derivatives

Derivatives of the function may come almost free with the evaluation of the function, may be as hard to compute as the function, or may be impossible to calculate. For example, comparing Newton's iteration with a direct parabolic interpolation, the evaluation of the derivative in Newton's method should take no more than 13.7% of the function evaluation time (see appendix IV). Consequently, to obtain the desired generality, our method should not request the derivatives of the function.

2.6 - Numerical reliability.

This means that if the algorithm converges, it does so to the root of the function. This implies that if the algorithm provides an answer, it is correct. It is not surprising to find, even nowadays, published algorithms that contain numerical bugs. One example that illustrates this point is the bisection algorithm, that usually uses: xmed = $(a+b)/2$. In a floating point number system, it is not always true that $a \leqslant (a+b)/2 \leqslant b$.

2.7 - Software reliability.

The algorithm should prevent and handle most of the exception conditions. Typical exceptions are caused by: division by zero, exponent overflows and underflows, destruction of storage, infinite loops, etc. No matter which data are fed into the subprogram, the zero finder should always provide an answer (raising an exception indicator is considered a valid answer, too) (Goodenough [6]).

2.8 - Efficiency

By efficiency we mean that the algorithm should perform as few function evaluations as possible. In the author's opinion, the set of functions that are fast to compute, yet do not allow direct solution, is rather "small". Hence, it is reasonable to assume that the function evaluation will require more time to compute than the internal processing of the zero finder. Naturally we should improve the efficiency of the zero finder itself; but in case of a trade off between internal efficiency and number of function evaluations we should minimize the latter.

2.9 - Range control

We should have the possibility of controlling the range, or ranges, of acceptable values of the unknown variable.

2.10 - Portability

The subprogram should be written in a portable subset of FORTRAN (Ryder [10]). Machine dependent constants should be avoided or automatically generated.

2.11 - Calling sequence

Usual strategies to solve most of the points mentioned so far require adding parameters and switches to the calling sequence. Certainly, long calling statements are disgusting. This also deteriorates the efficiency, because the subprogram uses all those parameters even though we may not be concerned about the corresponding cases.

3. The structure of ROOT1.

The main new feature of this subprogram is its structure. To understand this structure, note that the zero finder at each call returns a suggested estimate of the root.

A simple typical calling sequence is:

```
      W(1) = 0
      XNEW = initial-guess
    1 X = XNEW
      FX = function-evaluation
      XNEW = ROOT1(X,FX,FERR,XERR,W)
      IF(not-satisfied)GO TO 1
```

where X is the argument, FX contains the function value, FERR and XERR are output variables containing the function and argument error, and W is an auxiliary vector used by ROOT1 to store its information to make it reusable. ROOT1 returns the new guess of the root of the function.

The method used is a combined strategy of direct parabolic interpolation (Muller [8]), secant method and bisection, each of them applied when possible and in that priority. Direct parabolic interpolation was preferred over inverse (Brent [2]), although the direct method requires more computation, because the inverse is deteriorated by big second or third derivatives of the function, while the direct is independent of the second derivative. The order of convergence in both cases is 1.839.... The order of convergence of the direct parabolic interpolation near a double zero, when it is possible to apply, is 1.2337... which compares favourably to the order of the inverse which is 1.

The first time ROOT1 is called, it records the value and returns a solution as if $f(x) = -x+a$. The second time it applies the secant method and then uses direct parabolic interpolation, secant or bisection. If bisection is possible it is always forced after 30 iterations. After 80 iterations the algorithm raises an exception condition to stop.

All the status and numerical information about the function we are solving is contained in the vector W (dimensioned 9). There is no side effect in any call to ROOT1 besides the output variables. The initialization or reinitialization of the vector W to solve a new function is commanded by setting W(1)=0.

The output parameter FERR contains |FX|. XERR contains the shortest distance between arguments that contain a change in sign, or 1.e32 if no change in sign has appeared. In case of any type of error, FERR and XERR are set to zero. Testing for FERR and XERR alone, provides a graceful exit in the calling program in case of any type of error (including excessive number of iterations).

Now we will analyze some of the problems described earlier and how they may be solved with this scheme.

3.1 - Known starting values.

At most three (or four with a change in sign) pairs of argument and corresponding function values will be kept by ROOT1, so more points will be useless.

```
      .
      .
      W(1) = 0
      XNEW = ROOT1(X1,FX1,FERR,XERR,W)
      XNEW = ROOT1(X2,FX2,FERR,XERR,W)
      XNEW = ROOT1(X3,FX3,FERR,XERR,W)
    1 X = XNEW
      FX = function-evaluation
      XNEW = ROOT1(X,FX,FERR,XERR,W)
      IF(not-satisfied)GO TO 1
      .
      .
```

Here we feed three pairs of function values that are known to the programmer; the suggested values in the first two calls are merely discarded.

3.2 - Control of accuracy.

Since this is done outside the subprogram with the aid of the variables FERR and XERR, any sophistication is possible without adding any extra overhead to ROOT1.

3.3 - Nested calls.

This is a different scheme to solve a 2×2 system of nonlinear equations.

```
      .
      .
      WX(1) = 0
      X = initial-guess-for-x
      .
      .
   10 WY(1) = 0
      Y = initial-guess-for-y
   20 Y = ROOT1(Y,G(X,Y),FERR,XERR,WY)
      IF(not-satisfied)GO TO 20
      .
      .
      X = ROOT1(X,F(X,Y),FERR,XERR,WX)
      IF(not-satisfied)GO TO 10
```

3.4 - Range Control

```
      .
      .
    1 X = XNEW
      XNEW = ROOT1(X,F(X),FERR,XERR,W)
      IF(XNEW is-out-of-range) correction-on-XNEW
      IF(not-satisfied)GO TO 1
```

Since the returned value is only a suggested value, we are free to change it in any way before we feed it to ROOT1. When we have a single range, a simple and useful correction is:

$$XNEW = A*X + (1-A)*XNEW \quad \text{with } 0<A<1$$

This may be needed in case we do not provide an initial interval with a change in sign. When bisection is possible, the suggested values are guaranteed to fall inside the interval with the change in sign.

Appendix I shows the code for ROOT1. Because of the use of the auxiliary vector W, the routine becomes rather difficult to understand. Help in understanding it may be provided by performing the change in names mentioned in the comments.

This subroutine was extensively debugged over a large sample of functions where it performed extremely well.

3.5 - Performance.

Table I compares results obtained with ROOT1 against the values obtained for similar routines by Bus and Dekker [4].

Algorithms A, M and R are described by Bus and Dekker [4]; B was published by Brent [2] and C by Anderson and Björck [1].

Table I

Total number of function evaluations to solve all the functions in each group.

Function Group	I	II	III	IV	V
Algorithm functions in group	14	12	6	1	5
A	136	171	3118	>5000	
M	137	199	959	27	
R	123	163	1036	23	
B	124	223	808	18	
C	126	185	2720	969	
ROOT1 (+)	98	118	240	36	62
ROOT1 (++)	117	131	305	36	65

Group I consists of various instances of functions with a simple zero in the interval considered.
Group II is composed of functions with a high order inflection point near the zero.
Group III are powers of x (3,5,7,9,19,25)
Group IV is a function for which all derivatives vanish at its root.
Group V consists of the function $x \times \log(n \times x) + 1/(4n)$ that has two separate zeros close together and near a region where the function is not defined ($x \leqslant 0$). This is the case where it is very difficult to provide an interval with a change in sign. It was tested for n=50,100,150,200 and 250 with a single starting value, x=1.

Notes:
a) ROOT1 was run with function value tolerance of 1.E-14 (+) and with argument tolerance of $(1+|X|)\times 1.E\text{-}14$ (++).
b) ROOT1 was designed to work on the function value rather than on the argument interval. This explains in part the difference between the two rows of results.
c) For ROOT1, except for group IV, the value at one end of the interval was fed into the zero finder without calculation. This accounts for one function evaluation per function.
d) Group I in Bus and Dekker [4] included three polynomials of degree 2. These are solved exactly by ROOT1 with only three function evaluations. To make a more fair comparison these functions were excluded from group I.
e) All the computations were done in a Honeywell 6060 with 63 bits accuracy in floating point. Due to the short range in exponent of the H6060 floating point system, some real zeros appeared, which were caused by exponent underflow while evaluating x^n for $n>7$.

It is clear from the results that the algorithm presented behaves somewhat better than the other algorithms referenced for a large variety of functions, even without regard to its structural advantages.

4. The Structure of ROOT, a system solver.

This subprogram has the same structure as ROOT1 with the corresponding changes for systems of equations.

The method applied is a secant scheme extended to systems of equations. In this case there are no known extensions of the bisection and parabolic interpolation schemes, and algorithms that achieve comparable reliability characteristics and convergence are not known.

However, when convergence occurs, the algorithm is very efficient. Note that in each iteration, only n (dimension) function evaluations are performed against n function and $n\times n$ derivative evaluations in Newton's scheme or n and $n\times(n+1)/2$ respectively in the method described by Brown [3]. Moreover no derivatives are required, unlike Brown [3] that suggests the use of two function evaluations to estimate each derivative. Appendix II shows the code of ROOT and Appendix III shows the calculation procedure.

4.1 - Example of use.

```
      DO 10 I=1,IA
10    W(I,1) = 0.0
      XNEW(1) = initial-guess
      XNEW(2) = initial-guess
     . . . . .
20    DO 30 I=1,N
30    X(I) = XNEW(I)
      FX(1) = F1(X(1),X(2),...)
      FX(2) = F2(X(1),X(2),...)
     . . . .
      CALL ROOT(N,X,FX,ERR,XNEW,W,IA)
      IF(not-satisfied)GO TO 20
```

N is the number of functions in the system. X and FX are the input vectors containing the argument and function values. ERR is an output variable that contains the sum of $|FX(i)|$. XNEW is an output vector that contains the suggested zero of the system. W (dimensioned IA × IB) is a work matrix where ROOT stores its information. The input variable IA should be at least 2N+2, and IB (the other dimension of W) should be at least N+2.

The initialization or reinitialization of the vector W to solve a new system of equations is commanded by setting the first column of W to 0.

There is no parameter similar to XERR in this case for obvious reasons. Besides this all the rest of the remarks made about ROOT1 apply to ROOT.

4.2 - Performance.

Table II compares ROOT with the Newton method extended to systems of nonlinear equations and with the algorithm described by Brown [3]. The systems of equations tested are described in Brown [3].

Table II

Number of function and derivative evaluations needed to solve a system of nonlinear equations.

Function	I	II	III
System dimension	2	3	2
Algorithm [3]	8/12	18/36	6/9
Newton [3]	8/16	21/63	8/16
ROOT	18/0	60/0	12/0

A) nn/mm denotes nn function evaluations and mm derivative evaluations, without making distinction in which function or which partial derivative is evaluated.
B) ROOT was run until ERR (the sum of |FX(i)|) was smaller than 1.E-9, which gives more accuracy than the results presented in [3]. Actually the estimates produced in the last iteration would yield ERR<1.E-14 .

5. Conclusions.

These routines were developed for a computer network simulator in which they are used. Efficiency and reliability were the primary goals, since the routines were repeatedly used and buried several levels deep within the simulation program. The simplicity of their design, gaining generality, is the key to their efficiency and reliability. Avoiding the artificial splitting between main line and function evaluation, has resulted in simpler, clearer and more structured programs.

The general scheme presented for zero finders has been similarly applied to the solution of ordinary differential equations (Krogh [7]). It can also be used for minimization, quadrature, etc., that present the same structure and, to some extent, the same difficulties.

6. Acknowlegments.

For their patience, encouragement and precise suggestions, the author would like to thank Prof. D.E. Morgan and Prof. M.A. Malcolm.

7. References.

[1] Anderson N., and Björck Å. A new high order method of regula falsi for computing a root of an equation. BIT 13, (1973). pp. 253-264.

[2] Brent R.B. An algorithm with guaranteed convergence for finding a zero of a function. Computer Journal, Vol 14-4, (Apr. 1971). pp. 422-425. or: *Minimization Without Derivatives.* Prentice-Hall, Englewood Cliffs, 1973.

[3] Brown K.M., and Conte S.D. The Solution of Simultaneous Nonlinear Equations. Proceedings A.C.M. National Conference (1967). pp. 111-114.

[4] Bus J.C.P., and Dekker T.J. Two Efficient Algorithms with guaranteed convergence for finding a zero of a function. TOMS Vol 1-4 (Dec 1975). pp. 330-345.

[5] Dahlquist G., and Björck Å. *Numerical Methods.* Prentice-Hall, Englewood Cliffs, 1974. Ch. 6.

[6] Goodenough J.B. Exception handling issues and a proposed notation. CACM Vol 18-12. (Dec 1975). pp. 683-696

[7] Krogh F.T. An integrator design. JPL Technical Memorandum 33-479. Pasadena. (May 1971).

[8] Muller D.E. A Method of Solving Algebraic Equations Using an Automatic Computer. Mathematical Tables and Other Aids to Computation, Vol 10, (1956). pp. 208-215.

[9] Ralston A. *A First Course in Numerical Analysis.* Mc Graw-Hill, New York, 1965. Ch. 8.

[10] Ryder B.G., and Hall A.D. The PFORT Verifier User's Guide. Bell Labs, Murray Hill, NJ.

8. Appendix I

```
C     SOLVES  F(X)=0  BY REPETITIVE CALLS
      DOUBLE PRECISION FUNCTION ROOT1(X, FX, FERR, XERR, Q)
      DOUBLE PRECISION X,FX,FERR,XERR,Q(9),D,DABS,DECR,DSIGN,
     1       EPS,P,R,V,U,W,Z
C
C     X IS THE ARGUMENT
C     FX IS THE VALUE OF THE FUNCTION AT X
C     FERR IS THE FUNCTION ERROR OF THE LATEST ARGUMENT
C     XERR IS THE WIDTH OF THE ARGUMENT INTERVAL WHEN BISECTION
C          IS POSSIBLE; OTHERWISE IS 1.E32.
C     Q IS AN AUXILIARY VECTOR OF SIZE 9
C          (Q(1)=0 PROVIDES INITIALIZATION)
C     TO OBTAIN AN UNDERSTANDABLE PROGRAM, SUBSTITUTE:
C     Q(1,2,3,4,5,6,7,8,9)->FXA,XA,FXB,XB,FXC,XC,FXO,XO,ITER.
C
C     IF (REAL-ZERO) (NOTHING-TO-DO)
      IF(FX .EQ. 0.0)GO TO 810
      FERR = DABS(FX)
C     IF(FIRST-TIME) THEN (INITIALIZE; ESTIMATE-FIRST-ROOT; EXIT)
      IF(Q(1) .NE. 0.0)GO TO 20
         DO 10 I=3,9
   10    Q(I) = 0.0
         Q(1) = FX
         Q(2) = X
         ROOT1 = X+FX
         XERR = 1.E32
         RETURN
   20 Q(9) = Q(9)+1.0
C     IF (TOO-MANY-ITERATIONS) THEN (ERROR)
      IF(Q(9) .GT. 80.)GO TO 800
C     IF (REPEATED X) THEN (ERROR)
      IF((Q(9) .GE. 2.0 .AND. X .EQ. Q(4)) .OR. X .EQ. Q(2))GO TO 800
C     (PUSH X->A->B->C)
      DO 30 I=1,4
      J = 5-I
   30 Q(J+2) = Q(J)
      Q(1) = FX
      Q(2) = X
C     IF(CHANGE-IN-SIGN) THEN (STORE-OPPOSITE-VALUE)
      IF(Q(1)*DSIGN(1.D0,Q(3)) .GE. 0.0)GO TO 40
         Q(7) = Q(3)
         Q(8) = Q(4)
```

```
C     CALCULATE XERR
   40 XERR = 1.E32
      IF(Q(7) .NE. 0.0)XERR = DABS(Q(8)-Q(2))
C     IF(30<ITERATIONS AND POSSIBLE) THEN (BISECT)
      IF(Q(9) .GT. 30. .AND. Q(7) .NE. 0.0)GO TO 70
      V = (Q(3)-Q(1)) / (Q(4)-Q(2))
C     IF(3-OR-MORE-POINTS) THEN (TRY-MULLER)
      IF(Q(5) .EQ. 0.0)GO TO 50
         U = (Q(5)-Q(3)) / (Q(6)-Q(4))
         W = Q(4)-Q(2)
         Z = (Q(6)-Q(2)) / W
         R = (Z+1.0)*V-U
      IF(R .EQ. 0.0)GO TO 50
         P = 2.0*Z*Q(1)/R
         D = 2.0*P/(W*R)*(V-U)
         IF(D .LT. -1.0)GO TO 50
         ROOT1 = Q(2) - P/(1.0+DSQRT(1.0+D))
      IF(Q(7) .EQ. 0.0 .OR. (Q(2) .LT. ROOT1 .AND.
     1       ROOT1 .LT. Q(8)) .OR. (Q(8) .LT. ROOT1
     2       .AND. ROOT1 .LT. Q(2)))RETURN
C     (APPLY-SECANT-ITERATION)
   50 IF(Q(1) .EQ. Q(3) .AND. Q(7) .NE. 0.0)GO TO 70
      IF(Q(1) .EQ. Q(3))GO TO 800
      DECR = Q(1)/V
C     IF(RELATIVE DECREMENT < EPSILON) PRODUCE EPSILON*8 STEP
      IF(DABS(DECR)*4.6D18 .LT. DABS(Q(2)))DECR =
     1       DSIGN(1.74D-18*Q(2),DECR)
      ROOT1 = Q(2)-DECR
C     IF(OUT-OF-RANGE) THEN (APPLY-BISECTION)
      IF(Q(7) .EQ. 0.0 .OR. (Q(2) .LT. ROOT1 .AND.
     1       ROOT1 .LT. Q(8)) .OR. (Q(8) .LT. ROOT1
     2       .AND. ROOT1 .LT. Q(2)))RETURN
C     APPLY BISECTION
   70 ROOT1 = Q(2) + (Q(8)-Q(2))/2.0
      RETURN
  800 PRINT,"ERROR IN ROOT1"
  810 FERR = 0.0
      XERR = 0.0
      RETURN
      END
```

9. Appendix II

```
      SUBROUTINE ROOT(N,X,FX,ERR,XNEW,Q,IA)
C
C     N IS THE NUMBER OF FUNCTIONS
C     X IS THE ARGUMENT VECTOR
C     FX IS THE FUNCTION VALUES VECTOR
C     ERR IS AN ESTIMATE OF THE ERROR
C     XNEW IS THE SUGGESTED SOLUTION VECTOR
C     Q(IA,IB) IS A WORK AND STORAGE AREA OF ROOT
C     THE FIRST COLUMN OF Q SHOULD BE 0 FOR INITIALIZATION
C     IA IS THE FIRST DIMENSION OF Q AND SHOULD BE
C     GREATER OR EQUAL THAN 2*N+2, IB (THE SECOND DIMENSION)
C     SHOULD BE AT LEAST N+2
      DOUBLE PRECISION X(1),FX(1),ERR,XNEW(1),Q(IA,1),AUX,DABS,
     1     DAMP,SUMP
C
C     Q(1::N,1::NP1) CONTAINS FX VECTORS COLUMNWISE;
C     Q(NP1::N2,1::NP1) CONTAINS X VECTORS COLUMNWISE;
C     Q(NORM,1::NP1) CONTAINS SUM OF DABS(FX(I));
C     Q(NTIM,1::NP1) CONTAINS NUMBER OF TIMES USED IN COMPUTATIONS.
C     Q(1::N,NP2) IS USED AS A WORK VECTOR
C
      NP1 = N+1
      NP2 = N+2
      N2 = 2*N
      NTIM = N2+2
      NORM = N2+1
      NAGE = (N+3)/2
      IF(N .LT. 1 .OR. NTIM .GT. IA)GO TO 800
      ERR = 0.0
      DO 5 I=1,N
    5 ERR = ERR + DABS(FX(I))
      IF(ERR .EQ. 0.0)RETURN
      IF(Q(NORM,1) .NE. 0.0)GO TO 100
        DO 10 I=1,N
        IN = I+N
        DO 20 J=1,NP1
        Q(IN,J) = 0.0
   20   Q(I,J) = 0.0
        Q(IN,I) = 1.0
        Q(I,I) = 100.0
        Q(NTIM,I) = N
        Q(NORM,I) = 1.E32
```

```
      Q(IN,NP1) = X(I)
   10 Q(I,NP1) = FX(I)
      Q(NORM,NP1) = ERR
      Q(NTIM,NP1) = 0
      DAMP = 0.99
      GO TO 150
C
  100 JSUS = 1
      DO 110 I=2,NP1
      IF(Q(NTIM,I) .GE. N2)Q(NORM,I) = 1.E32
      IF(Q(NTIM,JSUS) .LT. NAGE)JSUS = I
      IF(Q(NTIM,I) .GE. NAGE .AND. Q(NORM,I) .GT. Q(NORM,JSUS))
     1    JSUS = I
  110 CONTINUE
C
      DO 120 I=1,N
      IN = I+N
      Q(IN,JSUS) = X(I)
  120 Q(I,JSUS) = FX(I)
      Q(NORM,JSUS) = ERR
      Q(NTIM,JSUS) = 0
      JSMA = 1
      DAMP = 0.0
      DO 130 J=1,NP1
      IF(Q(NORM,J) .GT. 1.E31)DAMP = 0.99
      IF(Q(NORM,J) .LT. Q(NORM,JSMA))JSMA = J
  130 CONTINUE
      IF(JSMA .EQ. NP1)GO TO 150
C
      DO 140 I=1,NTIM
      AUX = Q(I,JSMA)
      Q(I,JSMA) = Q(I,NP1)
  140 Q(I,NP1) = AUX
C
  150 DO 160 I=1,N
  160 Q(I,NP2) = Q(I,NP1)
C     LINEAR EQUATION SOLVER (SHOULD NOT DESTROY Q)
      CALL LEQS(Q,N,IA,Q(1,NP2))
      SUMP = 0.0
      DO 170 I=1,N
  170 SUMP = SUMP + Q(I,NP2)
      IF(DABS(1.0-SUMP) .LT. 1.D-10)GO TO 800
      DO 200 I=1,N
      IN = I+N
      XNEW(I) = Q(IN,NP1)
```

```
      DO 210 J=1,N
  210 XNEW(I) = XNEW(I) - Q(IN,J)*Q(J,NP2)
C     IF(SYSTEM-NOT-COMPLETE) DAMP-SOLUTION
  200 XNEW(I) = XNEW(I)/(1.0-SUMP)*(1.0-DAMP) + Q(IN,NP1)*DAMP
      DO 230 J=1,NP1
  230 Q(NTIM,J) = Q(NTIM,J)+1.0
      RETURN
  800 PRINT,"ERROR IN ROOT"
      ERR = 0.0
      RETURN
      END
```

The subroutine LEQS solves a system of linear equations. Its use should be:

```
CALL LEQS(A,N,IA,B)
```

to solve the system (N × N) A.X=B. A is dimensioned IA × IA. The resulting solution appears in B.

10. Appendix III

Secant iteration extended to system of equations.

At each step of the iteration we calculate a new argument vector, x_g, possible solution of the system, based on n+1 previous points.

Let y_j and x_j be corresponding function and argument column vectors; let Y and X be the matrices $(y_1,y_2,...,y_n)$ and $(x_1,x_2,...,x_n)$. Let C be a constant column vector and W=(1,1,...,1) a row vector of dimension n . Let A be an nx_n matrix. We have then:

$y_j = A.x_j + C$
$Y = A.X + C.W$
$p = n+1$
$C = y_p - A.x_p$
$(Y-y_p.W) = A.(X-x_p.W)$
$A = (Y-y_p.W).(X-x_p.W)^{-1}$
Let x_g be the searched solution, then
$A.x_g + C = 0$
$x_g = -A^{-1}.C$
$= -A^{-1}.(y_p-A.x_p)$
$= x_p - A^{-1}.y_p$
$= x_p - (X-x_p.W).(Y-y_p.W)^{-1}.y_p$
Let $V = (Y-y_p.W)^{-1}.y_p$ and let $|Z|$ denote W.Z (the sum of its elements)
$x_g = x_p.(1+|V|) - X.V$
Let P be the solution of $Y.P = y_p$
then:
$V = P / (1-|P|)$ and
$x_g = (x_p - X.P)/(1-|P|)$

The evaluation of x_g requires, at each step, the solution of a system of linear equations of order n , a matrix-vector product and other O(n) operations. When a new point is evaluated and control is returned to ROOT, we select the n+1 vectors whose $||y||$ is smallest, provided that they are used at least (n+3)/2 times, and not more than 2n . We order them so $||y_p||$ is the smallest of the norms.

11. Appendix IV.

The orders of convergence of Newton's method and direct parabolic interpolation are 2 and 1.83928675... respectively. To obtain the same accuracy at the same cost, the relation between the costs per iteration should be:

$$\ln(2)/\ln(1.839...) = 1.1374...$$

since both methods require the function evaluation, the derivative evaluation in Newton's method should cost no more than 13.7%.