A Solver for $f(x) = 0$

Nathaniel Grossman
Department of Mathematics
University of California, Los Angeles
Los Angeles, California 90024

Abstract

A Brent-type solver for real equations $f(x) = 0$, with enhancements for root bracketing and low resolution curve sketching, is implemented in Forth-83 augmented by floating point words.

Introduction

Solution of the equation $f(x) = 0$ is one of the core problems of mathematics. Even in the case where f is a real valued function of a real variable, it is far from solved. If f satisfies certain mild hypotheses, algorithms that never fail are available for the solution of $f(x) = 0$, but they may require an unacceptably long time to run. This paper presents an algorithm due to R. P. Brent and predecessors [2], that 'usually' will produce a root in a 'reasonable' time.

The description offered by Kahan [8] suggests that the Brent-type algorithm is the basis for the highly-successful SOLVE routine embedded in the HP–34C calculator and, presumably, in the HP–15C and other Hewlett-Packard products. Comparisons run between our implementation and that in the HP–15C calculator show that the HP–15C is more robust, evidently because of the care taken by the Hewlett-Packard engineers in the handling of extreme cases when they designed the mathematical function algorithms built into the calculator. Nevertheless, the implementation that we present in the accompanying Forth screens performs well.

Mathematical foundation

The Brent algorithm rests upon an important mathematical principle: If f is a real valued function continuous on the closed interval $[a, b]$ and if the values $f(a)$ and $f(b)$ have opposite signs, then the equation $f(x) = 0$ has at least one root r lying within the open interval (a, b). The simplest application of this principle, the bisection method, is described in almost every elementary textbook on numerical analysis. Its virtue lies in its surety of convergence to an approximation of any desired precision to a root r, but its failing is that this convergence is slow, sometimes unacceptably so if the desired precision is high. If the function f is differentiable, the faster-converging Newton's method can be invoked, but there are pitfalls. In general, when nothing more than its continuity is known about the function f, the applicable root-seeking methods are few. Since the bisection method is certain to yield a root of a continuous function once a suitable bracketing interval $[a, b]$ is known, the problem of root finding reduces ultimately to the search for bracketing intervals.

Because the bisection method converges so slowly, various devices have been suggested for speeding up convergence to a root r of f. Brent [2], building on the work of predecessors, has proposed a root-seeking algorithm that ordinarily runs faster than the bisection method and makes fewer evaluations of the function. Brent's algorithm uses bisection when driven to do so, but prefers to use the more rapidly converging secant method or the even more rapidly convergent technique of inverse quadratic interpolation. Comparison of convergence speeds may be made by counting function calls. The bisection method, beginning on the bracketing interval $[a, b]$ and isolating the root r_0 within $[r_0 - 2 \cdot \texttt{toll}, r_0 + 2 \cdot \texttt{toll}]$, will require essentially $\log_2 [(b - a)/\texttt{toll}]$ steps, so the same number of function calls because each step needs the function value at one new midpoint. Brent proved that his method will converge within at most about $\{\log_2[(b - a)/\texttt{toll}]\}^2$ function evaluations. However, this bound, seemingly greater than that for the bisection method, is calculated for arbitrary continuous functions.When the functions treated are smooth—say continuously differentiable—then Brent's algorithm is far faster than bisection and almost as fast as Newton's Method, without the danger of divergence inherent in the latter.

Fuller description of Brent's algorithm can be found in various sources on numerical analysis, but no one book gives the whole story in all its gory details. We have found useful details given by Brent [2], Forsythe, et al. [4], Kahan [8], and Press, et al. [9]. These intricate details are unfortunately much too long to reproduce here—or even to summarize. With regret, we must refer the reader who wishes to understand the workings of Brent's method fully to the sources just cited, with the hope that the comments in our source screens prove to be some help in unraveling the knots.

Brent's algorithm on the computer

Implementations of Brent's algorithm are available in the literature: ALGOL in [2], FORTRAN in [4] and [9], and Pascal (machine translated from FORTRAN) in [9]. We know of no other implementation in Forth. (Of course, the algorithm is implemented internally in certain Hewlett-Packard calculators.) The present implementation is written in Forth-83, and calls upon a floating point enhancement that follows the protocols described by Duncan and Tracy [3].

We have developed and tested our Forth program using an experimental floating point enhancement written by Martin Tracy which can display 9–10 significant figures as required by the FVG Standard [3]. This implementation is precise, and the examples that we have run in comparison to the HP–15C `SOLVE` always show agreement of roots within one unit in the final digit of the mantissa. The screens contain some protections against underflow, but without a doubt much less attention to exceptional cases than is built into the HP–15C.

The main Forth words

The Forth words provided analyze a function defined according to the syntax

```
: <function> ( r - f(r) )  etc etc etc ;
```

for which the input and output are each a real number (that is, a machine floating point number). There are three principal words: `SKETCHING`, `[ROOT]`, and `[SOLVING]`.

For convenience in analyzing functions for the existence of roots, multiple roots, maxima and minima, etc., we provide a low-resolution graphing capability via the gerund `SKETCHING`, whose syntax is

```
a b SKETCHING <function>.
```

The word `<function>` must already be defined as described. The output is a sketch of the graph of $y = < \texttt{function} > (x)$ on the interval $[a, b]$. This interval is divided into 79 equal subintervals, with the function evaluated at the partition and end points. (The number 79 can easily be adjusted to the Forth-83 Standard screen width of 63 subintervals.) The ordinates are normalized so that the maximum absolute value of y is given the arbitrary value 10, with all other values adjusted proportionally. An x-axis is drawn, but no points are labelled; the interval [a, printed as a reminder. We were persuaded to include a sketching word by the arguments of Press, et al. [9].

The word `[ROOT]` is useful for searching out a root bracketing interval when the function is already available. This word is not a gerund and it is defined to work on a generic function represented by the `DEFER`red word `FUNC`. Therefore, the current function must be vectored in:

```
' <function> IS FUNC  a b [ROOT]
```

is the sequence of words to enter, with $[a, b]$ a first guess at a root bracketing interval. If `<function>` fails to have opposite signs at a and b, `[ROOT]` moves one endpoint and then the other outward alternately, for a maximum of 50 moves. This bracketer can fail, which is why `SKETCHING` will be handy: consider $x^2 - 1$ on $[a, b] = [-2, 2]$. If `[ROOT]` finds a bracketing interval $[x1, x2]$, it stores these endpoints into the `FVARIABLE`s `X1` and `X2`, from which they can be retrieved as needed. If no bracketing interval is found, a suitable message is displayed.

The solving itself is carried out by the gerund `[SOLVING]`, with the syntax

```
a b [SOLVING] <function>
```

Beginning with $[a, b]$, a search is made for a bracketing interval, although a single guess at the root may be made, as in

```
a [SOLVING] <function>
```

and a second endpoint will be supplied automatically. If no brackets are found, a suitable message will be displayed and the root search terminated. Otherwise, the Brent algorithm will be invoked and the search completed. A soothing message assures the user that the search has begun and that the machine has not gone off into Never-Never Land. The result of the search appears on the stack as two floating point numbers `<function>`(r) r, with r an approximate root of the equation $<$ `function` $>(x) = 0$. If r_0 is the exact real root, then r lies in the interval $[r_0 - 2 \cdot \mathtt{tol1}, r_0 + 2 \cdot \mathtt{tol1}]$, where `tol1` is retained in the `FVARIABLE TOL1`.

As the Brent algorithm is made up of steps that are never less effective than bisection, and often more effective, it is clear that the algorithm is certain to terminate in a finite number of steps, either by not finding a bracketing interval (for which search we have limited the number of steps by fiat) or by the natural termination of the root approximation. The program can not run away, provided of course that `<function>` makes only legitimate Forth calls; this is the usual Forth admonition against self-destruction. Timings will depend upon the hardware and Forth software, but calls to the iteration cycle, inverse quadratic interpolation, inverse linear interpolation, and bisection will be found counted respectively in the `VARIABLE`s `#ITERS`, `#QUAD`, `#LIN`, and `#BISECT`. Counting the number of functional evaluations is optional. To take a count, include the words `1 #FUNC +!` in the definition of `<function>`; if the definition is already in the dictionary, then analyze `<function>'`, defined by

```
: <function>' 1 #FUNC +!  <function> ;
```

All the counts can be called up at once by the word `TOTUP`.

The Forth screens

The screens are based upon standard Forth-83, with the fixed point enhancements following the protocols proposed by Duncan and Tracy [3]. We believe that all of our nonstandard words are documented there with the exceptions noted in the following screen comments.

Screen 4. The words `DEFER...IS` are used to vector execution, and they are defined and illustrated in [1]. The word `F#BYTES` is an optional constant in the FVG Standard [3], leaving on the stack the number of bytes in a real number as it is represented upon the floating point stack.

In analogy to the actions of `C,` and `,`, `F,` appends the floating point number atop the stack to the end of the dictionary.

Screen 8. The phrase `n1 n2 TAB` will position the cursor at the position *n*1 to the right and *n*2 down from the upper left corner, which is the position (0,0).

Screen 11. `BELL` sounds an audible alarm.

Screen 12. `DARK` clears the screen.

Screens 14 and 15. We have chosen to carry out the inverse parabolic interpolation by means of Neville's algorithm [9] rather than by the subalgorithm proposed by Brent and adopted in the various implementations already mentioned above.

Certain other features require comment. The Brent algorithm invokes a mixed sort of tolerance to decide when it has converged. A relative tolerance is called upon when the root is 'large' and another, absolute tolerance is used when the root is 'small'. The absolute tolerance is specified in the `FVARIABLE TOL`, and it is set at compiling time. (We have specified 2.5E – 10, based on the display characteristics of the system that we used for development.) This number can also be altered during interpretation. The relative tolerance is based on the number called 'machine epsilon', which is the smallest positive machine number (call it *eps*) such that $1 + eps > 1$. In fact, the exact value of *eps* need not be used: the word `MACHEPS?` on Screen 21 computes *eps* up to a possible factor of 2 and stores it, and it runs each time that `[SOLVING]` is called. Running `MACHEPS?` suggests that the floating point enhancement that we have used in developing this program carries 2–3 decimal guard digits, hidden from printout but found by experiment to be active in comparisons like `FØ<`.

Examples

Some examples will illustrate the use of the solver words. For the techniques of root hunting, see elementary textbooks on numerical analysis. The *HP–15C Owner's Handbook* [7] and the *HP–15C Advanced Functions Handbook* [6] offer useful hints for root finding in a general setting, not just with the HP–15C calculator. For terminology and numerical analysis lore in general, Henrici's text [5] is a splendid source.

Warning: There is as yet no declaration from the Forth Standards Team to guide floating point implementers. Roundoff, truncation, overflow and underflow, floating point literals, arithmetic coprocessors, and many other matters are treated at present according to the implementer's preference. Consequently, roots and function values found on floating point implementations other than the one used by the author may differ slightly from those printed here.

Example 1. The equation $e^x - 2 = 0$ has a unique solution: log 2, the natural logarithm of 2. Define

```
: EEE FALN 2E F-  ;
```

and the counting enhancement

```
: EEE' 1 #FUNC +! EEE  ;
```

Then

```
ØE 1E [SOLVING] EEE'  F. E.
```

prints out

```
.69314718ØE-ØØ1   -4.22ØØ4452E-Ø1Ø
```

for the values of the root (log 2) and the value of the function at the root. Executing `TOL1 E.` prints 1.3...E – 010. However, the final 0 in the root is shaky, as the last digit should be the roundoff of 08. Without knowing more of the inner workings of the floating point package, we withhold further analysis.

The TOTUP is #FUNC = 65, #ITERS = 39, #QUAD = 15, #LIN = 9, and #BIS = 15.

Example 2. The equation $e^x - 5x + 3 = 0$ is treated in [8]. The HP–15C SOLVE finds one of the two roots of this equation in [1.25, 2]. We define

```
: HP   FDUP FALN FSWAP 5E F* F- 3E F+  ;
```

Then 1.25E 2E [SOLVING] HP returns the warning message

```
NO ROOT BRACKETED AFTER 50 TRIES
```

We resort to sketching: 1.25E 2E SKETCHING HP yields a 'convex' curve that is positive at 1.25 and 2, but is negative about halfway between. There are two roots. We try for one with the phrase

```
1.25E 1.6E [SOLVING] HP F. E.
```

and are rewarded with printout

```
1.468829260 0.00000000E+000
```

for the root and the function value at that root.

Example 3. We try to solve $f(x) = 0$, where

$$f(x) = x \cos x - \sin x.$$

To this end, we extend the dictionary by

```
: XC-S FDUP FDUP FCOS F* FSWAP FSIN F-  ;
```

The function $f(x)$ has a unique root between $-\pi/4$ and $\pi/4$, as can be seen by pencil-sketching the curves $y = x$ and $y = \tan x$ on the same axes: the root is $x = 0$. SKETCHING also suggests the existence of a root in this interval. If we run the phrase (in which PI/4 is an FCONSTANT that leaves $\pi/4$ on the stack)

```
PI/4 FNEGATE PI/4 [SOLVING] XC-S F. E.
```

we are rewarded with the root .000000000 and the corresponding function value −1.69...E−021. This is amazing accuracy, but TOTUP tells more of the story: #ITERS = 2, #QUAD = 0, #LIN = 2, and #BIS = 0. The function has odd symmetry in the selected interval, so linear interpolation, effectively equivalent to bisection, converges to the root immediately.

To see the workings better, we try

```
0E PI/4 [SOLVING] XC-S F. E.
```

and see −.000013581 and 1.54...E−012. Having set the left endpoint a equal to the root in the interval, we retrieve a putative root that is outside the interval (not necessarily a fault) but clearly not the root 0. What happened? The answer tells us something about the construction of the floating point implementation within which we are working. When x is 'small enough', the quantities $x \cos x$ and $\sin x$ are so close in value that their difference is represented by the machine number 0. How small is 'small enough' can be deduced from the power series expansion of $f(x)$ for small x; it is

$$f(x) = -x^3/3 + \text{5th and higher powers.}$$

From the definition of TOL1 in Screen 22, we see that the root will be located within essentially the tolerance in TOL if $|b|$ is small, certainly the case when the root being converged upon is

0. The variable **TOL** is set at compilation to the value 2.5E − 010. Hence, the machine number $|f(x)| <$ **TOL** provided that $x^3 < 7.5E-010$, so always if $|x| < 10^{-3}$. The number −.000013581 returned for the root is within this bound, and its effect upon the stopping criterion is the same as that of 0.

The phenomenon observed here is effectively *cancellation*. We can avoid it by refining the definition of **XC-S**. Introduce the refined word

```
: XC-S#  ( r — r' )
         FDUP 1E-003 F<
         IF
           FDUP FDUP F* F* 3E F/ FNEGATE
         ELSE
           XC-S
         THEN ;
```

This modified function will not introduce spurious roots.

Example 4. Sometimes there are multiple occurrences of a root. For example, the polynomial $(x-2)^2$ has the root $x = 2$ and this root may be counted with multiplicity 2 because each occurrence gives one factor $x-2$. The nicest functions $f(x)$, those called *analytic*, always behave this way: each root r of $f(x) = 0$ is associated to a factor $(x-r)^m$ and the integer m is called the multiplicity of r. When a root is found by application of some algorithm and when 'nothing' is known about the behavior of the function near that root, a check should be made for multiplicity, either by sketching or by examining the modified function obtained by deflation. The function obtained by deflating $f(x)$ at the root r is $g(x) = f(x)/(x-r)$. If r is a multiple root of $f(x)$, then r will also be a root of $g(x)$. Because of roundoff and truncation errors introduced by machine representations, it can sometimes be difficult to decide whether a function has a multiple root or just simple roots very close together.

We illustrate another use of deflation by continuing the solution of the equation in Example 2. **SKETCHING** showed that the function **HP** had two simple roots in [1.25, 2], so that the root-bracketer failed. We got around that problem by using the sketch to narrow the search interval to the bracketing [1.25, 1.6], finding the root 1.468829260. To find the second root, we can search on [1.6, 2], but we can also deflate and search on the original interval. To avoid possible input-output aberrations, we introduce an **FVARIABLE ROOT1**, then execute

```
1.25E 1.6E [SOLVING] HP  ROOT1 F! FDROP
```

Now we extend the dictionary by the definition

```
: HP#  FDUP HP  FSWAP  ROOT1 F@  F- F/ ;
```

the deflation of **HP** at the root found first. To be sure, we try **1.25E 2E SKETCHING HP#** and observe that the deflated function has a single axis-crossing in [1.25, 2]. Then **1.25E 2E [SOLVING] HP# F. E.** yields the root 1.743751990 with corresponding function value −6.35...E − 010. For comparison, **1.6E 2E [SOLVING] HP** yields the same root 1.743751990 with function value −2.32...E − 010. Deflation is especially useful when roots of polynomials are sought, because each deflation cuts the work of evaluation substantially. Henrici's text [5] gives examples and points out treacherous quicksand waiting for the unwary.

References

1. A. Anderson and M. Tracy, *Mastering Forth*, Brady, Bowie, MD, 1984.
2. R. P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, NJ, 1973
3. R. Duncan and M. Tracy, "FVG Standard Floating Point Extension," *Dr. Dobb's Journal*, **#95** (September, 1984), 110–115
4. G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, NJ, 1977
5. P. Henrici, *Essentials of Numerical Analysis, with Pocket Calculator Demonstrations*, John Wiley & Sons, New York, 1982
6. *HP–15C Advanced Functions Handbook*, Hewlett-Packard Company, Corvallis, OR, 1984
7. *HP–15C Owner's Handbook*, Hewlett-Packard Company, Corvallis, OR, 1984
8. W. H. Kahan, "Personal Calculator Has Key To Solve Any Equation $f(x) = 0$," *Hewlett-Packard Journal*, (1979), 20–26
9. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes*, Cambridge University Press, New York, 1986

```
\ Scr# 0 Brent-style root finder: solves f(x) = Ø    NG Ø8/14/86
(In the manner of HP-34C, HP-15C key SOLVE, without minima.)
When f(x) is a 'continuous' function that has opposite signs
at a and b, [SOLVING] (screen 33) returns a root of f(x) = Ø in
[a,b] to within a tolerance  4*macheps*|x| + tol, where macheps
is the relative machine precision (screen 21) and tol can be
set by user. Enhancements include low-resolution graphing and
bracketing interval search. See references on screen 34.

           Copyright, 1986, by Nathaniel Grossman

The reader is authorized to make one machine-readable copy of
this program for his or her personal use, but further
distribution of such copies either as gifts, 'freeware', or by
sale is not authorized.
    All commercial rights are reserved by Nathaniel Grossman.

\ Scr# 1 Solver, v1.Ø: F-83; requires floating point NG Ø8/14/86

: MARKER   ;   \ to FORGET from

 2 13 THRU   \ lores function plotter
14 16 THRU   \ inverse interpolators
17 2Ø THRU   \ root bracketer
21 34 THRU   \ root finder
\  35        \ references

2.5E-1Ø  TOL F!     \ tolerance is based on max of 9 sig figs
                    \ and can be user set after compilation

\   Various Fortran presentations of the Brent algorithm would
\ use the same words in different ways. The conventions follow
\ Forsythe, Malcolm, and Moler (see screen 34).

\ Scr# 2 Floating point (and other) extensions      NG Ø7/27/86

: S>F   ( n --- r )    \ float an integer
    SD FLOAT   ;

: F2DUP   ( r r' --- r r' r r' )
    FOVER FOVER   ;

: F+!   ( r addr ---    )
    DUP >R  F@ F+  R> F!   ;

: D>S   ( d --- n )   \ d=n if d < 32768
    DROP   ;
```

```
\ Scr# 3 Constants and variables                       NG Ø7/27/86

VARIABLE #HOR  ( #pixels horizontally)   8Ø #HOR !
VARIABLE #VER  ( #pixels vertically  )   21 #VER !

ØE FCONSTANT FØ

FVARIABLE X1       ( bracket left  )
FVARIABLE X2       ( bracket right )
FVARIABLE XNOW     ( current x )
FVARIABLE YMAX     ( max y[x] )
FVARIABLE YMIN     ( min y[x] )
FVARIABLE DELTAX   ( x step )
FVARIABLE DELTAY   ( y step )


\ Scr# 4 Floating point arrays| deferred function    NG Ø7/27/86

: FARRAY
    CREATE   ( n ---    )   \ allot and initialize for n reals
      Ø ?DO  FØ F,  LOOP
    DOES>    ( n --- addr_of_nth )
      SWAP F#BYTES * +   ;

#HOR @ FARRAY Y-VALUES   \ holds f[x] for y-scaling


DEFER FUNC   \ completion: ' <function> IS FUNC
                \ but handled later by gerunds



\ Scr# 5 Bracket handling                              NG Ø8/Ø2/86
\ Adjust first guess for interval containing root.
\ Inspired by HP SOLVE protocols, but works fully only when
\ the stack contains ONLY the guess(es): the HP user stack
\ always has depth exactly 4

: NO_GUESSES?   ( ?   --- ? )
\ change in depth by pushing real is F#BYTES/2
    DEPTH ( Forth-83 DEPTH )  F#BYTES 2/ <
      IF   \ not at least one real on stack
        CR CR
        ABORT" : SPECIFY AT LEAST ONE BRACKET BOUND AND RERUN!"
      THEN   ;
```

```
\ Scr# 6 Bracket handling, cont.                        NG Ø8/Ø2/86


: ONE_GUESS?   ( ? [r] --- [r1] [r2] )
\ compare HP-15C Handbook, p. 192
    DEPTH  F#BYTES 2/ =   \ exactly one real r on stack
      IF   \
        FDUP FØ=
          IF     1E-7                 \ [Ø,1e-7]
          ELSE   FDUP 1.ØØØØØØ1E F*   \ [r,r(1+1e-7)]
          THEN
      THEN   ;



\ Scr# 7 Bracket handling, cont.                        NG Ø8/Ø2/86

: X1=X2?   ( r1 r2 --- r3 r4 )
    F2DUP F=
    IF
      FDROP ONE_GUESS?
    THEN   ;

: BRACKETED?   ( ? --- ?? )
   NO_GUESSES?   \ prompt for a guess or two
   ONE_GUESS?    \ concoct a second guess
   X1=X2?   ;    \ concoct two distinct guesses



\ Scr# 8 Bracket handling, cont.| make x-axis           NG Ø7/27/86

: MAKE_X1<X2   ( r1 r2 --- r1' r2' )
\ swap if necessary so that r1 < r2
    F2DUP F< NOT
    IF  FSWAP  THEN   ;

: X-STEP   ( x1 x2 ---    )
    FSWAP F-  #HOR @ 1- S>F  F/
    DELTAX F!   ;

: MAKE_X-AXIS   (   ---    )
    #HOR @ Ø DO
      I 11 TAB  ASCII - EMIT
    LOOP   ;
```

```
\ Scr# 9 y-scaling                                  NG Ø7/27/86

: Y-EXTREME?   ( r --- r )
    FDUP
    YMAX F@  FMAX  YMAX F!
    FDUP
    YMIN F@  FMIN  YMIN F!   ;

: Y_SCALE   (   ---   )
\ NOT protected against the identically-zero function, but why
\ are you trying to graph that and machine-seek its roots?
    YMAX F@ FABS  YMIN F@ FABS  FMAX   \ largest excursion
    #VER @ 1- 2/  S>F  F/              \ scaled vertical step
    DELTAY F!   ;


\ Scr# 10 Fill scaling array                        NG Ø7/27/86

: FILL_Y-VALUES   (   ---   )
\ compute FUNC[x] along x-axis, track max and min, fill array
    #HOR @ Ø DO
      XNOW F@  DELTAX F@  XNOW F+!
      FUNC
      Y-EXTREME?   \ update extreme for later y-scaling
      I Y-VALUES F!
    LOOP   ;





\ Scr# 11 Lores screen plotting words               NG Ø7/27/86

: PLACE_*   (   ---   )   \ run thru array, plot to screen
    BELL   ( call operator's eyes back to screen )
    #HOR @ Ø DO
      I Y-VALUES F@  DELTAY F@
      F/  FIX ( truncates toward Ø )  D>S     \ pixel y-offset
      #VER @ 1+ 2/ SWAP -         \ screen y-offset from x-axis
      I SWAP TAB  ASCII * EMIT    \ plot a *
    LOOP   ;

: .[X1,X2]   \ print bracket values in brackets below plot
    Ø #VER @ 2+ TAB
    ASCII [ EMIT  SPACE
    X1 F@ E.  ASCII , EMIT  SPACE  X2 F@ E.
    ASCII ] EMIT   ;
```

```
\ Scr# 12 Initialize lores plotter                    NG Ø7/28/86

: SKETCH_INIT   ( x1 x2 ---   )
    BRACKETED?
    X1=X2?
    MAKE_X1<X2
    -1E1ØØ YMAX F!  1E1ØØ YMIN F!   \ impossible starting values
    F2DUP  X2 F!
    FDUP  XNOW F!  X1 F!
    X-STEP                          \ scale x-axis
    DARK
    MAKE_X-AXIS
    Ø 11 TAB   ;



\ Scr# 13 Lores function sketcher                     NG Ø7/27/86

\ syntax:  a b SKETCHING <function>

: SKETCHING   ( x1 x2 ---   )
    ' IS FUNC   \ ' is NOT a misprint for [']
    SKETCH_INIT
    FILL_Y-VALUES
    Y_SCALE
    PLACE_*
    .[X1,X2]   ;



\ Scr# 14 Inverse parabolic interpolation            NG Ø7/31/86
\ Aitkin-Neville scheme, value at Ø of the parabolic function
\ taking values real XA,XB,XC at real arguments FA,FB,FC

FVARIABLE P/Q  FVARIABLE PEE  FVARIABLE QUE

FVARIABLE XA  FVARIABLE XB  FVARIABLE XC
FVARIABLE FA  FVARIABLE FB  FVARIABLE FC

: XA@   XA F@   ;    : XA!   XA F!   ;
: XB@   XB F@   ;    : XB!   XB F!   ;
: XC@   XC F@   ;    : XC!   XC F!   ;
: FA@   FA F@   ;    : FA!   FA F!   ;
: FB@   FB F@   ;    : FB!   FB F!   ;
: FC@   FC F@   ;    : FC!   FC F!   ;
```

```
\ Scr# 15 Inverse parabolic interpolation, cont.     NG Ø7/31/86
\ Aitkin-Neville scheme, value at Ø of the parabolic function
\ taking real values XA-XB,Ø,XC-XB at real arguments FA,FB,FC

: INV_PARAB   (   ---    )   \ Brent's algorithm defines p/q
    FB@  XA@ XB@ F- F*
    FB@ FA@ F-  F/
    FC@ F*
    FB@  XB@ XC@ F- F*
    FC@ FB@ F-  F/
    FA@ F*  F-  FDUP PEE F!
    FC@ FA@ F-  FDUP QUE F!
    F/  P/Q F!   ;


\ Scr# 16 Inverse linear interpolation               NG Ø7/31/86
\ value at Ø of the linear function taking the real values
\ XA-XB,Ø at FA,FB.

: INV_LIN   (   ---   )   \ Brent's algorithm defines p/q
    FB@  XA@ XB@ F-  F*
    FDUP PEE F!   \ store p
    FB@ FA@ F-
    FDUP QUE F!   \ store q
    F/  P/Q F!   ;

: BRENT'S_I   (   --- r )  \ r = b + p/q (= Brent's i)
    XB@ P/Q F@  F+   ;


\ Scr# 17 Root bracketer: variables, initialization  NG Ø7/28/86

VARIABLE ALAS?
VARIABLE #TRIES
VARIABLE MAX#TRIES

FVARIABLE BLOAT

: ROOT_INIT   ( x1 x2 ---   )
    BRACKETED?  X1=X2?  MAKE_X1<X2
    -1 ALAS? !  Ø #TRIES !  5Ø MAX#TRIES !  1.6E BLOAT F!
    X2 F!  X1 F!   ;

: ALAS
    CR CR  ." NO ROOT BRACKETED AFTER " MAX#TRIES @ .
    ." TRIES" CR  BELL  ABORT   ;
```

```
\ Scr# 18 Interval expanders                            NG Ø7/28/86

: <-X1   (   ---   )   \ move x1 to left
    X1 F@  FDUP
    X2 F@  F-
    BLOAT F@  F* F+    \ x1+BLOAT*(x1-x2)
    X1 F!   ;

: X2->   (   ---   )   \ move x2 to right
    X2 F@  FDUP
    X1 F@  F-
    BLOAT F@  F* F+    \ x2+BLOAT*(x2-x1)
    X2 F!   ;



\ Scr# 19 Function comparisons                          NG Ø7/28/86

: ANTISIGNS?   (   --- b )
\ true if FUNC takes different signs at x1 and x2
\ assuming that f(x1)*f(x2) is NOT Ø
    X1 F@ FUNC  X2 F@ FUNC
    FDUP FABS F/   ( protection against underflow )
    F*  FØ   ;

: X1_NEARER_TO_ROOT?   (   --- b )
\ true if FUNC(x1) is closer to Ø than FUNC(x2)
    X1 F@ FABS  X2 F@ FABS  F<   ;



\ Scr# 20 Root bracketer                                NG Ø7/28/86

: [ROOT]   ( x1 x2 ---   )
\ leaves root brackets, if found, in X1 and X2
    ROOT_INIT
    MAX#TRIES @ Ø DO
      1 #TRIES +!
      ANTISIGNS?
        IF  Ø ALAS? ! LEAVE  THEN   \ cue warning, leave loop
      X1_NEARER_TO_ROOT?
        IF  <-X1  ELSE  X2->  THEN
    LOOP
    ALAS? @
      IF  ALAS  THEN   ;            \ cued warning
```

```
\ Scr# 21 Machine epsilon                              NG Ø8/Ø2/86

FVARIABLE MACHEPS   \ approx least machine r>Ø so 1+r > 1

: MACHEPS?    (    ---    )    \ find macheps
    1E
    BEGIN
      1E F2DUP F+ F<
    WHILE
      Ø.5E F*
    REPEAT
    MACHEPS F!    ;




\ Scr# 22 Tolerances, bounds                           NG Ø8/Ø2/86

FVARIABLE TOL    \ desired length: interval of root uncertainty

: HALF-LENGTH    (    --- r )
    XC@ XB@ F-  Ø.5E F*    ;

: TOL1    (    --- r )    \ r = 2*macheps*|b|+Ø.5*tol
    2E MACHEPS F@  XB@ FABS   F* F*
    TOL F@  Ø.5E   F*
    F+    ;

: |PEE|    (    --- r )    PEE F@ FABS    ;
: |QUE|    (    --- r )    QUE F@ FABS    ;


\ Scr# 23 Interpolation flagger                        NG Ø8/Ø1/86
\ The next approximate root will be calculated by interpolation
\ if that number is inside the latest bracketing interval and
\ "not too close" to the endpoints. Otherwise, bisection will
\ be called. The decision depends on the size of the change
\ p/q to be made in b.

: ACCEPT_INTERP?    (    --- b )
\ 2|p| < min( [3*|half-length|-tol1]*|q|,|tol1*q|)
\ true -> inverse-interpolate   false -> bisect
    2E |PEE| F*    \ 2|p|
    3E HALF-LENGTH FABS F*  TOL1 F-  |QUE|  F*
    TOL F@ FABS  |QUE|  F*
    FMIN
    F<   ;
```

```
\ Scr# 24 Initialize from given root brackets       NG Ø8/Ø4/86

FVARIABLE NOW-LENGTH   \ length of current bracketing interval
FVARIABLE PRE-LENGTH   \ length of previous bracketing interval

: INIT_FROM_A_AND_B   ( a b ---   )
    FDUP XB!  FUNC FB!
    FDUP XA!  FUNC FA!   ;

: INIT_C   (   ---   )
    XA@ XC!
    FA@ FC!
    XB@ XA@ F- FABS FDUP
    NOW-LENGTH F!  PRE-LENGTH F!   ;


\ Scr# 25 Data cycler, quadratic-interpolator flag   NG Ø8/Ø4/86

: CYCLE_DATA    (   ---   )
\ Cycle:  xa -> xc -> xb -> xa, fa -> fc -> fb -> fa
    XA@ XB@ XC@
    FROT
    XC! XB! XA!
    FA@ FB@ FC@
    FROT
    FC! FB! FA!   ;

: USE_QUADRATIC?   (   --- flag )
\ true if now-approx and pre-approx are distinct
    XA@ XC@  F=   ;


\ Scr# 26 Convergence, bisection flags               NG Ø8/Ø4/86

: CONVERGED?   (   --- flag )
\ true if either |c-b|/2 < toll or fb = Ø
    HALF-LENGTH FABS  TOL1  F<
    FB@  FØ=
    OR   ;

: BISECT?   (   ---   flag )
\ true if either |fa| < |fb| or |pre-length| < toll
    FA@ FABS  FB@ FABS F<
    PRE-LENGTH F@ FABS  TOL1  F<
    OR   ;
```

```
\ Scr# 27 Endpoint updater                           NG Ø8/Ø4/86

: ENDPOINTER   (   ---   )
\ now-data -> pre-data; new now-data
    XB@ XA!  FB@ FA!
    TOL F@  NOW-LENGTH F@ FABS  F<
      IF    NOW-LENGTH F@
      ELSE  TOL F@
            ØE HALF-LENGTH F< NOT
              IF  FNEGATE  THEN
      THEN
    XB@ F+  FDUP  XB!  FUNC FB!   ;




\ Scr# 28 Node renamers                              NG Ø8/1Ø/86

: INTER-SWITCH   (   ---   )
    XA@ FDUP  XC!  FUNC FC!
    XB@ XA@ F-  FDUP
    NOW-LENGTH F!  PRE-LENGTH F!   ;

: EXTER-SWITCH   (   ---   )
    FC@ FABS  FB@ FABS  F<
      IF  CYCLE_DATA  THEN   ;

: INTER/EXTER-SWITCH   (   ---   )
    ØE FB@ F<  ØE FC@ F<  =   ( true: both true or both false )
      IF  INTER-SWITCH  ELSE  EXTER-SWITCH  THEN   ;


\ Scr# 29 Initialize the rootfinder                  NG Ø8/1Ø/86

\ optional counters:
VARIABLE #FUNC    \ to count # of function evaluations
VARIABLE #ITERS   \ to count # of iterations in PRE-SOLVER
VARIABLE #LIN     \ to count # of inverse linear interpolations
VARIABLE #QUAD    \ to count # of inverse parabolic interps
VARIABLE #BIS     \ to count # of bisections

\ should have #ITERS = #LIN + #QUAD + #BIS

: INIT_SOLVER   ( a b ---   )
    Ø #LIN !  Ø #QUAD !  Ø #BIS !  Ø #ITERS !  Ø #FUNC !
    MACHEPS?
    INIT_FROM_A_AND_B
    INIT_C   ;
```

```
\ Scr# 30 Select among bisct'n & inverse intrpltrs   NG Ø8/1Ø/86

: LINEAR/QUADRATIC_INTERPOLATION   (   ---   )
\ with optional counters
     BISECT?
        IF    HALF-LENGTH FDUP    1 #BIS +!
              NOW-LENGTH F!  PRE-LENGTH F!
        ELSE
              FA@ FC@ F=
                IF    INV_LIN     1 #LIN +!
                ELSE  INV_PARAB   1 #QUAD +!
                THEN
         THEN   ;



\ Scr# 31 Set vars after interpolation performed     NG Ø8/1Ø/86
: INTERPOLATION_IF_APPROPRIATE   (   ---   )

    ACCEPT_INTERP?
       IF    P/Q F@  NOW-LENGTH F!
       ELSE  HALF-LENGTH FDUP
             NOW-LENGTH F!  PRE-LENGTH F!
       THEN   ;

: SOOTHER
    ." Seeking root in "                          \ optional user
    ASCII [ EMIT SPACE XA@ E. ASCII , EMIT        \  soother
    SPACE XB@ E. ASCII ] EMIT   CR CR     ;       \  message



\ Scr# 32 The solver: unadorned core                NG Ø8/1Ø/86

: PRE-SOLVER   ( a b --- r ) \ r approximates the root
    INIT_SOLVER  CR CR
    SOOTHER
    BEGIN
      CONVERGED? NOT
    WHILE
      1 #ITERS +!
      LINEAR/QUADRATIC_INTERPOLATION
      INTERPOLATION_IF_APPROPRIATE
      ENDPOINTER
      INTER/EXTER-SWITCH
    REPEAT
    XB@   ;
```