MATHEMATICS OF COMPUTATION
VOLUME 44, NUMBER 169
JANUARY 1985, PAGES 151-165

A Rapid Robust Rootfinder

By Richard I. Shrager

Abstract. A numerical algorithm is presented for solving one nonlinear equation in one real variable. Given F_X with brackets A and B, i.e., $\text{sign}(F_A) = -\text{sign}(F_B) \neq 0$, the algorithm finds a zero of F, $A < X < B$. Alternately, a crossover pair is found, i.e., (X, Y): $\text{sign}(F_X) = -\text{sign}(F_Y) \neq 0$ where there is no floating-point number in the system between X and Y. This feature allows full use of machine precision. Optionally, a tolerance TOL > 0 may be given, to permit termination when $|Y - X| <$ TOL. The method, once rapid convergence sets in, is alternation of one linear interpolation or extrapolation with one inverse quadratic interpolation. The resulting asymptotic convergence rate is competitive with other methods that refine both brackets and do not require dF/dX. Other merits of the algorithm are: robust calculation, efficient three-point interpolation, and superior behavior in bad cases. The algorithm is tested and compared with others.

1. The Problem, and an Outline of the Paper. Given a real-valued function F of one real variable X, with brackets A and B such that $\text{sign}(F_A) = -\text{sign}(F_B) \neq 0$,

(1) find X such that $F_X = 0$ (we call such X a machine zero of F), or

(2) find X and Y such that $\text{sign}(F_X) = -\text{sign}(F_Y) \neq 0$ and either

(a) $|X - Y| <$ TOL, where TOL is a user-supplied tolerance, or

(b) X and Y have no representable number between them (we call such X and Y a crossover pair).

The rootfinder has no access to derivatives of F. It is to be superlinearly convergent for smooth F with simple roots, and reasonably efficient even in the worst of cases. The root is to be found by successively refining the bracket $[A, B]$ until one of the termination criteria (1, 2a, or 2b, above) is met.

The contents of the paper are as follows:

Section 2: Background. Some current algorithms of Bus, Dekker, and Brent are discussed.

Section 3: Structure and correctness. The general flow of Algorithm S is used to show that it must terminate.

Section 4: Robust calculation. Overflow never occurs and underflow is safeguarded, permitting X and F to span the range of the machine.

Section 5: Machine issues. Floating-point numbers can be specified precisely with standard, civilized computer expressions.

Section 6: Good cases. The various strategy sequences during rapid convergence are described.

Received February 8, 1982; revised May 16, 1983 and February 3, 1984.

1980 *Mathematics Subject Classification*. Primary 65H05, 65-04, 65D05, 65G05, 68E05.

Section 7: Efficient interpolation. A specialized but inexpensive 3-point interpolation is presented.

Section 8: Bad cases. Modified bisection avoids the worst costs of interval bisection. More frequent bisection is encouraged in high cost problems. Multiple of the linear interpolation step is a buffer strategy between bisection and the other strategies.

Section 9: Algorithm S is compared to others.

Appendix A: Machine-dependent constants and procedures.

Appendix B: Procedures called by Algorithm S.

Appendix C: Algorithm S.

2. Background. Before 1969, most rootfinders found in computer libraries fell into one of two categories: "slow but sure," or "fast but risky." A slow but sure method is interval bisection (IB), where X is generated from the current bracket $[A, B]$ by $X \leftarrow (A + B)/2$. In the undesirable "slow and somewhat risky" category is linear interpolation (LI), in which X is generated, e.g., by $R \leftarrow F_A/(F_A - F_B)$, then $X \leftarrow (1 - R) * A + R * B$. X usually lies strictly between A and B, but in most problems, one bracket is refined while the other remains far from the root. This leads to the double disadvantage of linear convergence with no guarantee of the root's location (especially when no machine zero is found). The LI strategy is upgraded to "slow but sure" by alternating it with IB. The most common "fast but risky" method is the secant method, which achieves an order of convergence of 1.618 because it always uses the last two iterates to generate X by linear extrapolation (LE). Here again, without a machine zero, convergence is not guaranteed for general F. In addition, it is possible for the method to oscillate or diverge. However, the order of convergence is an industry standard.

In 1969, Dekker [3] published an algorithm employing all three strategies (LI, LE, and IB) to the effect that both brackets converge with order 1.618. For most problems with simple roots, the order of strategies is (LI, LI, LE) repeated. IB is reserved for cases where the linear strategies do not work well. In 1973, Brent showed an example [1, p. 49] where Dekker's algorithm can be induced to make very small refinements in the bracket for many iterations, because there is no built-in requirement that IB will ever be used. Brent presented an algorithm [1, p. 58] with two refinements. First, IB is used with a frequency that guarantees the equivalent of an IB step every two steps. Therefore, Brent's algorithm should never be much slower than IB. Second, Brent introduced inverse quadratic interpolation (QI), which is used about every third step during rapid convergence, resulting in noticeable improvement in performance [1, Table 4.2], [3, Table 1] for simple roots in unexceptional neighborhoods. In 1975, Bus and Dekker [2] published two algorithms. The first is similar to Dekker's algorithm with a counter to insure that bisection is achieved at least once in every four steps. The second algorithm strives for frequent use of rational interpolation, achieving a convergence order of 1.839 in the best cases with insured bisection every five steps. In addition, Bus and Dekker disputed Brent's claim that the cost of Brent's algorithm is, at worst, twice the cost of IB, but they gave no examples.

At this point, we will adopt the Bus and Dekker names for the various algorithms to avoid confusion, adding our own to the list:

Algorithm	Reference	Strategies
A	Dekker [3]	LI, LE, IB
B	Brent [1]	LI, LE, QI, IB
M	Bus & Dekker [2]	LI, LE, R3, IB
R	Bus & Dekker [2]	LI, LE, R3, IB
S	Shrager	LI, LE, QI, ML, MB

The various strategies are

LI: linear interpolation using the current brackets.
LE: linear extrapolation, constrained between the brackets.
QI: inverse quadratic interpolation, similarly constrained.
R3: 3-point rational interpolation.
IB: interval bisection.
MB: modified bisection, explained in Section 8.
ML: multiple LI step, explained in Section 8.

3. Algorithm S: Structure and Correctness. The variable S (for Strategy) in Algorithm S, indicates which of 5 strategies has been selected in each iteration. Based on S, X is generated by one of 5 methods selected in the first CASE statement in Appendix C. Regardless of the strategy used, X is eventually generated by one of two statements:

$$X \leftarrow \text{COMBIN}(A, B, R), \quad \text{where } 0 \leqslant R \leqslant .5, \quad \text{or}$$
$$X \leftarrow \text{COMBIN}(\text{FPNMED}(A, B), \text{COMBIN}(A, B, .5), Q).$$

The values produced by the expressions COMBIN(A, B, R) and FPNMED(A, B) are required to lie strictly between A and B if possible, and to equal A otherwise. Therefore X generated either way will satisfy this requirement. $X = A$ produces the crossover termination. The general structure of the iterative loop is shown in flow-chart form in Figure 1. The 5-way branch mentioned above occurs in Box 1. Box 2 contains two more 5-way branches, one for SIGN(F_X) = SIGN(F_B) called "success" (the second CASE statement in Appendix C), and another for SIGN(F_X) = SIGN(F_A). These ten branches are simply individual decisions about the next value of S, with an occasional auxiliary calculation. Regardless of which branch is taken, one of the brackets has been replaced by X by the end of the iteration.

Since the bracket is refined in every iteration, and the floating-point numbers are a finite set, then even with TOL = 0 and no machine zero of F, a termination will be reached (i.e., a crossover) in a finite number of iterations, though possibly in as many iterations as there are numbers in the initial $[A, B]$. Brent showed that Algorithm A could be induced to approach this limit. We show in Section 8 that Algorithm S cannot be so induced, and in fact, that it has by far the mildest worst case of any current algorithm.

While Algorithm S is correct with respect to termination, and it can refine a bracket to machine precision when required, it cannot guarantee the accuracy of the

root in some seemingly simple problems, nor can any of the rootfinders. For example, consider $F_x = G_x - 1.5$, with a root $X_* = 1.5$, where G_X is smooth and computed correct to the last bit. For $0 \ll dG/dX < 1$, the number of machine zeros increases as $2/(dG/dX)$ from subtractive roundoff. When roundoff or underflow produce many machine zeros where there should be only one root, Algorithm S will stop with the first encountered zero. For some F, it may be possible to determine the proper sign for any X even though the magnitudes are too small to resolve, in which case a very small number with the proper sign may "stand in" for the true F, avoiding the large interval of machine zeros. An analogous method can allow Algorithm S to proceed when true F would be too large to compute.

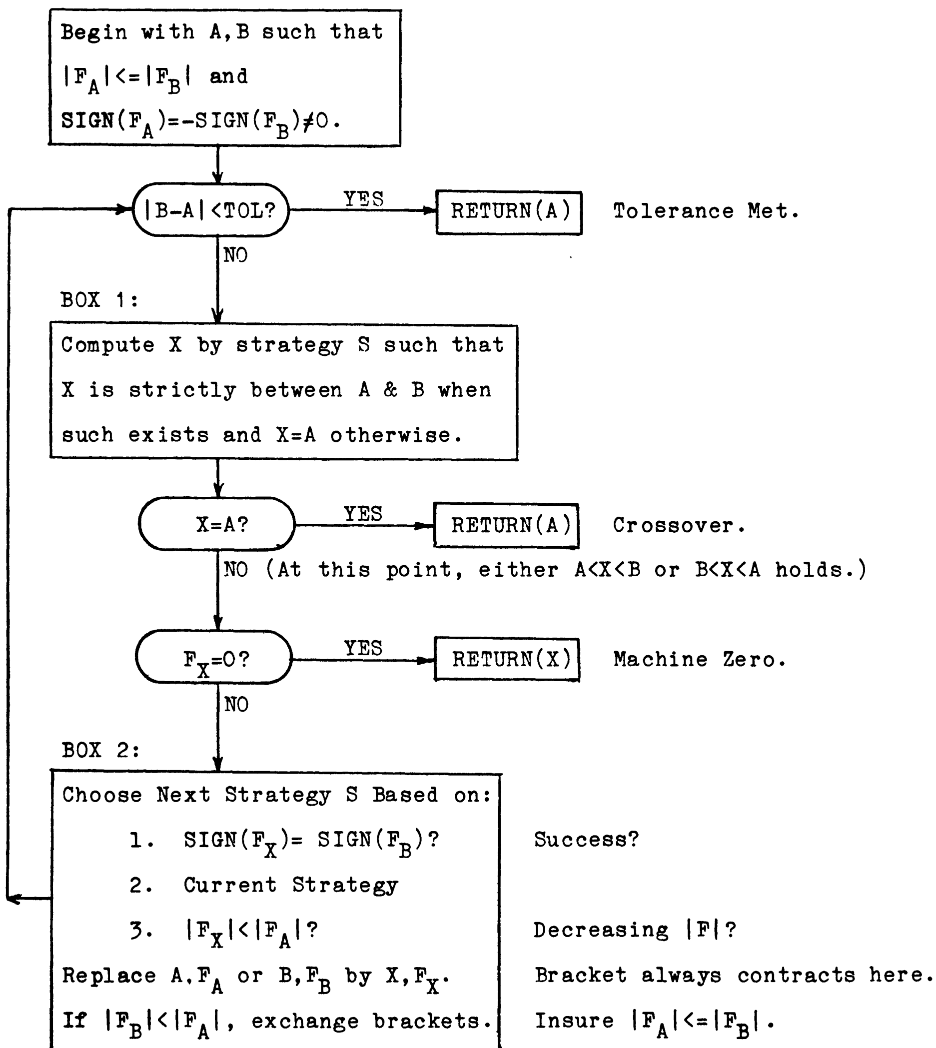


FIGURE 1
General flow chart of Algorithm S, *indicating features that guarantee correctness.*

4. Robust Calculation. There are two basic rules observed in Algorithm S: overflow and division by zero shall never occur, and underflow is safeguarded. Underflow is presumed to yield zero without stopping the program. Certainly, it is more efficient to let some exceptions occur, and trap them with machine interrupts, rather than test in advance or scale to avoid the exceptions every time through the calculations. But for this application, the "strict avoidance" policy is not very expensive; we avoid confronting the confusing array of interrupt handling conventions in current systems; and Algorithm S runs successfully on systems that simply "stop dead" when such exceptions occur. Dekker [4] points out that correctness (eventual termination) is not precluded by overflow as long as the calculation proceeds with some large magnitude having the proper sign. Even this device has its price, because the overflow usually interferes with an accurate determination of X, which in turn interferes with convergence rate.

If an underflowed quantity is likely to become a significant part of subsequent calculations, the calculation is redone with scaled values, so that underflow cannot occur or at least that underflow the second time is guaranteed insignificant. Such underflow is "detected" by a suspicious zero result, or by an incremented quantity that fails to change, e.g., X in the procedure COMBIN in Appendix B.

Here are four arithmetic difficulties encountered in Algorithms B, M, and R which are avoided by the methods of Sections 4 and 5:

1. Algorithms M and R routinely produce intermediate quantities with more extreme magnitudes than the inputs. Even when all X and F are well within machine range, an overflow or underflow will interfere with the accurate calculation of X. The following are two examples on the IBM-370 with magnitudes 10^{-75} to 10^{75}. Algorithms M and R will overflow on $F_X = X^3$ with $A = -10^{13}$ and $B = 2*10^{13}$. They will also underflow persistently on $F_X = X - 10^{-40}$ with $A = 5*10^{-41}$ and $B = 2*10^{-40}$. On this last example, perceptible progress is made only by IB, every fourth or fifth step.

2. Algorithms B, M, and R cannot consistently resolve a root to machine precision. Their achievable bounds for various forms of computer arithmetic are listed in Theorem 1 of [4]. Machine-precision techniques are discussed in Section 5.

3. Some functions, particularly those with horizontal or linear asymptotes at each end, may have no a priori bound on the root. A simple ploy for the user is to set $-A$ and B to huge numbers, close to machine limits. For such problems, Algorithms B, M, and R will overflow on the operation $B - A$. Similarly, if A and B are near poles of F, an overflow may occur on $F_A - F_B$.

4. Some functions have small roots whose trailing digits are below the hard underflow level of the machine. Unless the machine permits soft underflow [8], it will be impossible for Algorithms B, M, and R to refine the brackets near the root, because small, significant increments will underflow. The IB formula $X \leftarrow (A + B)/2$ will not underflow in this situation, but it is unsafe on other than binary machines [5, p. 162].

The following are techniques for preventing harmful exceptions (refer to Appendix A for definitions):

1. If the magnitudes of A and B are huge, then $A + B$ may overflow if SIGN(A) = SIGN(B), and $A - B$ may overflow otherwise. Such operations are few in

Algorithm S, so they may be checked in advance for safety. For example, the test $ABS(B - A) < \text{TOL}$ also takes two other forms: $A < \text{TOL} + B$ and $B < \text{TOL} + A$ depending on the signs of A and B. (Comparisons are assumed exact for all FPNs.) For another example, X may be generated either by $X \leftarrow (1 - R) * A + R * B$ or by $X \leftarrow A + R * (B - A)$. This last formula is used whenever $\text{SIGN}(A) = \text{SIGN}(B)$ because it also avoids the anomaly mentioned in [5, p. 162].

2. The calculation of R involves ratios with known bounds, provided the arithmetic is at least weak by the definition of Dekker [4], with some properties worth mentioning:

Property A: If $A < B$, then $A/B < 1$.

Property B (soft): FPEMAX > FPPREC.

FPEMAX and FPPREC are defined in Appendix A. Property B (soft) insures that the ratio $1/(1 - U)$ does not overflow for $U < 1$. The computation of R in the LE strategy involves the product of ratios:

$$R \leftarrow [1/(1 - U)][(A - C)/(B - A)].$$

The relation $U < 1$ must hold, so Property B (soft) insures the safety of the first ratio. Neglecting roundoff, the relation $|A - C| < |B - A|$ must hold, because the current values (C, A, B) are the previous values (A, X, B), and X is at most half way from A to B. For the following examples of roundoff, recall that the current A and B have at least one FPN between them, otherwise a crossover-pair exit would have occurred. Roundoff is of three types, shown below in their extreme form, where $t_0, t_1, t_2, \ldots$ are consecutive increaing FPNs:

(a) Let $C = t_0$, $B = t_5$, and let A be rounded to t_3, instead of t_2, where the t's are equally spaced:

$$(A - C)/(B - A) = 1.5.$$

(b) Let $B = t_0$, $A = t_2$, $C = t_2$, $C = t_3$, where A is a normalized integer power of the machine base (not the smallest):

$$(A - C)/(B - A) = (1/2) * \text{FPBASE}.$$

(c) Let $0 = C = t_0$, $A = t_1$, $B = t_3$ on a hard underflow machine:

$$(A - C)/(B - A) = (1/2) * \text{FPBASE}^{\text{FPPREC}-1}.$$

On soft underflow machines, the FPNs are equally spaced around zero, so case (c) does not occur. Based on the above observations, Property B (soft) is sufficient for soft underflow machines, but for hard underflow, we need:

Property B (hard): FPEMAX > = 2 * FPPREC.

3. The statement $X \leftarrow A + R * (B - A)$ could yield $X = A$ even though the exact quantity $R * (B - A)$ is a significant fraction of A. That is, $R * (B - A)$ underflows. On machines with soft underflow (see [8] for definitions of underflow), this never happens, because underflow implies a result less than any significant part of any representable number. On machines with hard underflow, the condition is correctable by scaling A and B by a large factor so that the least digit of LARGE * A is well above the underflow level, i.e.,

$$X \leftarrow (\text{LARGE} * A + R * (\text{LARGE} * B - \text{LARGE} * A))/\text{LARGE}.$$

The variable LARGE is an integer power of the machine base to avoid roundoff effects:

Property C: FPF(A) = FPF(LARGE$*A$) where FPF is a function defined in Appendix A.

The above calculation of X can produce $X = A$ only when the increment to A is insignificant, whereupon X is replaced by the number nearest to A in the proper direction, as described in the next section.

5. Machine Issues. The procedures COMBIN and FPNMED specified in Appendix B, are required to return a result strictly between A and B, if possible, and to return A otherwise. Consequently, these procedures must detect when A and B are neighbors in the number system. The procedure NEIBOR, specified in Appendix A, provides such capability, since it returns the number closest to A toward B. (NEIBOR is a realization of a procedure called NEXTAFTER, suggested in [8].) Iterates X of the form $X =$ NEIBOR(A, B) are frequently required in the final stages of refining the bracket, and the relation $B =$ NEIBOR(A, B) is proof that the machine limit of refinement has been reached.

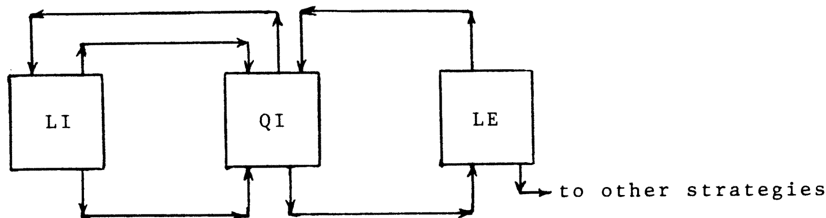
The procedures COMBIN, FPNMED, and NEIBOR are written in terms of primitive procedures FPE, FPF, and FPN. These last three, in the author's view, should be provided for any floating-point system, as they enable many machine-dependent operations to be performed with a minimum of machine-dependent information. See Appendix A for definitions of these procedures and some associated machine-dependent constants. Their use is amply illustrated in Appendices B and C.

Machine dependence of a code should not be worrisome, as long as the machine-dependent parts are clearly indicated and easily adaptable to other machines. In the codes of COMBIN, FPNMED, NEIBOR, and ROOT (Algorithm S), comments that begin COMMENT MACHINE explain the machine-dependent requirements. The procedures FPE, FPF, and FPN will have to be rewritten for each base, precision, and range, but they are simple, and they allow much of the code of their client procedures to remain intact. For example, in NEIBOR, COMBIN, and FPNMED, as long as one is dealing with hard underflow systems, only the constants need be changed. Conversion from hard to soft underflow will require more effort, because the definition of a permissible floating-point number has changed. However, Algorithm S is left unchanged regardless of the change of arithmetic, provided that the properties mentioned in Section 4 are preserved.

As a final word on the civility of code, a floating-point number has 3 parts: sign, fraction, and exponent. It will profit us in the long run to establish expressions that treat those parts separately, just as we have already done with character strings in the newer FORTRANs.

6. Good Cases. At the start of any iteration, A and B bracket the root(s) with $|F_A| \leqslant |F_B|$. Until termination, X always lies strictly between A and B. The subgoal of Algorithm S is to refine the "worst" bracket B at each step. In this context, we regard SIGN(F_X) = SIGN(F_B) as "success" and SIGN(F_X) = SIGN(F_A) as "failure". X is never more than half way from A to B (neglecting roundoff), therefore any success is at least as good as an IB step.

In this paragraph, assume that F is nearly linear in $[A, B]$, and therefore the LI, QI, and LE strategies will produce $|F_X| < |F_A|$ at every iteration. Algorithm S starts with LI. Either X replaces B and the brackets exchange (success), or X replaces A (failure). Either way, QI follows using 3 points (see Section 7 for the exact formula): X from LI has become A, the bracket replaced by X has become C, and the remaining bracket is B. The QI step may either fail or succeed depending on the higher derivatives of F. (See Ostrowski [7] on remainder formulas.) If it succeeds, LI repeats with the latest two points. Otherwise LE is invoked with the latest two points. A geometrical note: if LI fails in a bracket with no zeros of d^2F/dX^2, and QI fails with $|F_X| \leqslant 1/3|F_A|$, the use and the success of LE are guaranteed (the LE iterate is not used if it lies in the B half of the bracket). Therefore LE will succeed when used, in most smooth problems, and QI will once again be invoked with the latest 3 points. As long as $|F_X| < |F_A|$, and one of the strategies QI or LE succeeds, the result is alternation of 2-point and 3-point interpolation, always using the latest available points. A summary of such sequences may be given as a graph, in which the upper paths denote success, and the lower paths denote failure:



Brent [1, p. 54] deduces a weak order of convergence of at least 1.618 for Algorithm B. The same reasoning applies to Algorithm S, which uses the same interpolation formulas in slightly different order. Where Algorithm B tends to use QI every third step, Algorithm S uses it every other step. In cases where QI confers an advantage over LI in convergence rate (e.g., for smooth F and away from zeros in low-order derivatives of F or its inverse $X(F)$), the more frequent use of QI confers a greater advantage.

7. Efficient Inverse Quadratic Interpolation. A useful fact will be stated without proof. Consider two starting points B and C with $|F_B| < |F_C|$. The variable A is computed by linear interpolation or extrapolation which, in either case, can be expressed as:

$$R \leftarrow F_B/(F_B - F_C); \quad A \leftarrow B + R(C - B).$$

Assume, as required in the algorithm, that $\operatorname{sign}(F_A) = -\operatorname{sign}(F_B)$ and $|F_A| < |F_B|$. Then the root X of the inverse quadratic through these three points is computed by:

$$U \leftarrow F_A/F_C; \quad R \leftarrow [F_A/(F_A - F_B)]/(1 - U); \quad X \leftarrow A + R(B - A).$$

In other words, where a linear interpolation is done in 5 arithmetic operations, the subsequent inverse quadratic interpolation requires 8. This economy is available only when A has been linearly interpolated from B and C. For this reason, a QI step in the proposed algorithm is always immediately preceded by an LI or LE step.

Efficiency of interpolation is not a negligible issue. Some programs use root-finding to invert easily-computed functions many thousands of times. In such cases, added overhead can negate fewer F-evaluations. Brent's QI formula is fairly robust (in terms of avoiding arithmetic exceptions) but inefficient (about 20 operations). Inversely, the Bus and Dekker rational formula is efficient (10 operations) but unsafe. The proposed QI formula, with conditions and precautions explained in Section 4, is both robust and efficient.

8. Bad Cases. This section considers three methods of improving efficiency in problems where the rapidly-converging strategies often fail: avoiding the worst costs of IB, increasing the frequency of bisection, and inserting a buffer strategy between bisection and the others.

First we consider problems for which IB does not work well, namely for roots of small magnitude in an initial bracket containing many magnitudes (e.g., binades). As a model bracket we use $[0, 1]$ on a binary machine with p bits of precision, with 2^{-L} as the smallest positive normalized number, and with no unnormalized numbers. (Using other large brackets or other arithmetic rules will not affect the conclusions drawn here.) The function F is assumed to have no machine zero in $[0, 1]$. Cost is the number of F-evaluations required to find a crossover pair. Average cost refers to a population of problems in which the distribution of roots is uniform in $[0, 1]$. We will also be concerned with worst costs of various bisection schemes. These cannot be ignored as curiosities, because a high worst cost indicates the presence of many other high cost roots. Our goal is to reduce all of IB's higher costs.

The average cost of IB is $p + 1$. That is, for $1/2 < X_* < 1$ where X_* is the root, each bisection determines one bit of precision. For $2^{-L} \leqslant 2^{-(n+1)} < X_* < 2^{-n}$, n additional bisections are required to achieve the smaller crossover tolerance, but the interval $[2^{-(n+1)}, 2^{-n}]$ is also smaller and less likely to contain a root. Still, the worst cost of IB is $p + L$, ranging from hundreds to thousands of F-evaluations on the computers listed in [5, Chapter 2]. This is in addition to the often fruitless iterations spent by the various algorithms on strategies other than bisection. As illustrated in groups 7–11 of the test results, both smooth and nonsmooth problems, in intervals of many sizes, can be subject to some of this cost.

A second search technique, which we call table bisection (TB), regards an interval $[A, B]$ as a vector T containing all the floating-point numbers in the interval. I.e., $A = T_0$, $B = T_J$, and the next iterate will be the median value $T_{J/2}$. A procedure, (FPNMED) for producing this number given any A and B is presented in Appendix B.

The average and worst cost of TB are the same: $p + \log_2 L$ samples per run. While this is much better than the worst cost of IB, it exceeds the average cost of IB by a considerable amount. TB should be used only when IB runs into trouble. This is accomplished by a blend of the two methods called modified bisection (MB). Algorithm S has a weight variable q, initialized at .5 and squared after each use. As long as A and B differ in either sign or magnitude, the MB formula

$$X = T + Q*(M - T)$$

is used, where $M = (A + B)/2$ and T is the median floating-point value in (A, B). When A and B are within some factor (2 in the program) the interval bisection formula $X = A + .5*(B - A)$ is used. For example, if the root is very near zero, the

first few values of Q will be 2^{-1}, 2^{-2}, 2^{-4}, 2^{-8}, and very quickly, the vast majority of weight is given to table bisection. The average cost of MB exceeds the average cost of IB by only 0.66 samples per run (a computed average of trials over all binades on a DEC-10 computer), while the worst cost, about $p + 2 * \log_2 L$, is only moderately worse than TB in relative terms.

From Table 3 in [2], it is observed that when the rapidly convergent strategies are not working well, all reported methods behave poorly. These results would have been much worse for a small relative tolerance (the root was zero). One problem, already mentioned, is the choice of bisection strategy. But also important is the fraction of bisection steps overall, which can be quite low for methods described in [2].

This, then, is our second method of improving bad cases: having evolved a reasonably efficient bisection strategy, we must also insure that bisection steps will be taken frequently when they are needed. Accordingly, a counter (MB) in the program records the number of bisection episodes, i.e., the number of times the strategy variable S is changed to 5 from some lesser value. The algorithm, upon selecting $S = 5$, will take at least MB consecutive bisection steps before selecting another strategy. As a result, bisection will occasionally be used when faster methods would work, but the cost will always be moderate relative to total cost. Also, if some maximum permissible number of consecutive nonbisection steps NB (initially 12 in Algorithm S) has been taken, then 4 or more consecutive bisection steps are taken, NB is lowered by one (with a lower limit of 4 to permit superlinear strategies to work if they can), and MB is raised by one with no explicit maximum. Thus the asymptotic bad case behavior of Algorithm S is that of bisection, which is selected an increasing fraction of the time.

For our third method of improving bad cases, we consider what to do after bisection. Bisection was invoked because the other methods failed. Rather than resume a recently ineffective plan, there is a simple test for how well things are likely to go. The hope is that LI is a good strategy. If so, moving a multiple of the LI step from A toward B will succeed; this is the ML strategy. (See Section 6 on success and failure.) In Algorithm S, the initial multiple is 8, it is doubled at each failure of ML and restored to 8 at any success of ML. Whenever ML fails, it accomplishes two goals: first, it eliminates a larger portion of the bracket than LI would have, and second, it warns us that F was not nearly linear. Therefore, more bisection is probably a good idea. On the other hand, if ML succeeds, the repeated sequence ML-LI is asymptotically superlinearly convergent where the repeated sequence IB-LI is not. It is mainly for these reasons that Algorithm S tests better with ML than without it. ML is also used instead of QI or LE when $|F_X| < |F_A|$ fails to hold.

9. Test Results. Test problems from [2] were among those used to compare Algorithms B, M, R, and S. All problems for all methods were run on an IBM-370 computer in double precision (14 base-16 digits with magnitudes 16^{-64} to 16^{64}). The results are presented in Table I. The results are grouped as in [2] (individual results are largely uninteresting), and table entries are average F-evaluations per problem. In addition, two other groups of well-behaved problems are included. Group 5 uses $F_X = X^P - C^P$ for all combinations of:

$$C = (.01, .02, .05, .1, .2, .5, 1, 2, 5), \quad \text{and}$$

$$P = (-6, -3, -1.5, -.75, .75, 1.5, 3, 6)$$

with first guesses near the root X_*:

$$(A = .5X_*, B = 2X_*), (A = .5X_*, B = 1.25X_*),$$
$$(A = .5X_*, B = 1.01X_*), (A = .99X_*, B = 2X_*),$$

288 problems in all. Group 6 uses $F_X = X^P - C$ with the same combinations of inputs as Group 5. However, Group 5 problems all have exact answers $X_* = C$, whereas Group 6 counts only those problems with no exact zero, 184 problems out of 288. In Groups 5 and 6, TOL = 0 was used.

Table I shows, essentially, that Algorithm S does well, though not best, in well-behaved problems (Groups 1, 2, 5, 6) and is, for the most part, superior to the other methods when things go wrong (Groups 3, 4). This last point was illustrated further by several additional groups of problems with TOL = 0:

Group 7. $F_X = (X - 1/C)^3, A = -1, B = 3.$
8. $F_X = (X - 1)/(1 + (X - 1)^2), A = 0, B = C.$
9. $F_X = \mathrm{LOG}(X), A = 1/C, B = C.$
10. $F_X = \mathrm{EXP}(-X^2) - .01, A = 0, B = C.$
11. $F_X = \mathrm{IF}\ X > .7\ \mathrm{THEN}\ 1\ \mathrm{ELSE}\ -1, A = 0, B = C.$

In groups 7 through 11, the parameter C was increased exponentially:

$$C = 2^i, \qquad i = 1, 2, 3, \ldots.$$

Rather than present many tables, we feel that the following observations summarize the results:

1. For small C (< 256 usually), cost of the four algorithms was essentially the same except in groups 7 and 11 as remarked below.

2. In Group 7, the costs of Algorithms B, M, and R were at least 1.7 times the cost of Algorithm S in all cases, because multiple roots require bisection which was used a greater fraction of the time in Algorithm S.

3. The costs of Algorithms B and M increased as $\mathrm{Log}_2 C$ in all groups, while the cost of Algorithm S leveled off for large C. Theoretically, the costs of Algorithm S should increase as $2 * \mathrm{Log}_2(\mathrm{Log}_2 C)$ for large C.

4. The cost of Algorithm R increased as $\mathrm{Log}_2 C$ in Groups 8 and 9, but as $5 * \mathrm{Log}_2 C$ in Groups 10 and 11. In these last two groups, Algorithm R used bisection only about one in every five steps. Throughout Group 11, Algorithm R was at least 3 times as expensive as the other algorithms.

Problems illustrating robustness of arithmetic were not included in this section. For a discussion of those, see Section 4.

TABLE I

Test results in F-evaluations-per-run

Problem	Algorithm			
Group	B	M	R	S
1*	9.12	9.82	9.00	9.35
2*	18.6	16.2	13.4	16.7
3*	94.	114.	126.	15.
4*	18.	18.	22.	7.
5.	8.79	9.13	7.74	8.56
6	11.67	11.90	10.47	11.75
7-11	· · · · · · see text · · · · · ·			

*from Bus and Dekker [2].

```
COMMENT         Appendix A:
   Machine-dependent procedures and constants are defined here.
   The acronyms FPN, FPE, and FPF mean a floating-point number
   x, its exponent e, and its signed fraction f, such that:
      x=f*(base^e), base = machine base, and 1/base<=f<1.
   x=0 yields e=fpemin defined below, and x=0 iff f=0;

INTEGER PROCEDURE fpe(REAL x);  COMMENT Return the FPE of x;
   RETURN ( IF x=0. THEN -128 ELSE
      (('377000000000 LAND abs(x)) LSH -27)-'000000000200);
REAL PROCEDURE fpf(REAL x);  COMMENT Return the FPF of x;
   RETURN ( IF x=0. THEN 0. ELSE (IF x<0. THEN -1. ELSE 1.)*
      ((abs(x) LAND '400777777777) LOR '200000000000));
REAL PROCEDURE fpn(INTEGER e; REAL f);
   COMMENT  Return FPN with FPE e and FPF f. If f=0, return 0;
   RETURN ( IF f=0. THEN 0. ELSE (IF f<0. THEN -1. ELSE 1.)*
      ((abs(f) LAND '400777777777) LOR ((e+128) LSH 27)));

INTEGER fpprec,fpemin,fpemax;   REAL fpbase,fpbsm1,fpbsp1,
   fpfmin,fpfeps,fpfmax,   fpnmin,fpnmax,   huge,small,large;
COMMENT MACHINE dependent constants. These should be stored as
   permanent data or generated once at the start of execution.
fpprec_  27,                   no. of base-digits of precision.
fpemin_-128,                   minimum FPE.
fpemax_ 127,                   maximum FPE.
fpbase_  2.,                   floating-point base.
fpbsm1_(fpbase-1)/(2*fpbase), used by fpnmed in appendix B.
fpbsp1_(fpbase+1)/(2*fpbase), used by fpnmed in appendix B.
fpfmin_1/fpbase,               minimum FPF > 0.
fpfeps_fpn(1-fpprec,fpfmin),  minimal FPF increment, i.e. one
                                  in the last digit of any FPF.
fpfmax_1.-fpfeps,              maximum FPF.
fpnmin_fpn(fpemin,fpfmin),    minimum FPN > 0.
fpnmax_fpn(fpemax,fpfmax),    maximum FPN.
small _fpn(fpemin+fpprec+3,fpfmin),  any very small FPN, but
        large enough that the smallest significant increment
        to it is an FPN. Used only on hard underflow machines.
large _fpn(fpprec+3,fpfmin),  fpbase raised to an integer
        power such that large*fpnmin >= small.
        Used only on hard underflow machines;

REAL PROCEDURE neibor(REAL a,b);    BEGIN "neibor"
   COMMENT Return FPN closest to a toward b. Return a iff a=b;
   INTEGER ea,sa,sb;   REAL aa,ab,fa;
   COMMENT MACHINE: neibor also requires fpfmin,fpfeps,fpfmax,
      fpnmin,fpe,fpf, and fpn defined above;
   aa_abs(a); sa_sign(a); ea_fpe(a); fa_fpf(aa);
   ab_abs(b); sb_sign(b);    RETURN (
      IF sa=sb AND ab>aa THEN
         IF fa<fpfmax THEN sa*fpn(ea,fa+fpfeps)
         ELSE sa*fpn(ea+1,fpfmin)
      ELSE IF a=b THEN a
           ELSE IF fa>fpfmin THEN sa*fpn(ea,fa-fpfeps)
                ELSE IF fa=fpfmin THEN
                        IF aa=fpnmin THEN 0.
                        ELSE sa*fpn(ea-1,fpfmax)
                     ELSE sb*fpnmin)   END "neibor";
```

```
COMMENT          Appendix B
   Procedures called directly by Algorithm S are given here;

REAL PROCEDURE combin(REAL a,b,r);   BEGIN "combin"
COMMENT Return x=(1-r)*a+r*b where a and b are arbitrary and
   0<=r<=.5.  x must be strictly between a and b if possible.
   Return x=a iff a=b or b=neibor(a,b);
REAL x;   COMMENT MACHINE: combin also requires large, small,
            and neibor from appendix A;
x_IF (a GEQ 0.)=(b GEQ 0.) THEN a+r*(b-a) ELSE (1-r)*a+r*b;
COMMENT MACHINE: for soft underflow, the following test and
  BEGIN-block should condense to: IF x=a THEN x_neibor(a,b);
IF x=a THEN BEGIN
   IF abs(a)<small AND abs(b)<1 THEN
      x_(large*a+r*(large*b-large*a))/large;
   IF x=a THEN x_neibor(a,b)   END;
RETURN(IF x=b THEN neibor(b,a) ELSE x)   END "combin";

REAL PROCEDURE fpnmed(REAL ain,bin);   BEGIN "fpnmed"
COMMENT Return the median FPN between ain and bin. I.e. vector
   t=all FPNs in [ain,bin], ain=t(0), bin=t(n). Return t(n/2).
   Return ain iff ain=bin or bin=neibor(ain,bin);
INTEGER ea,eb,edist,ex,sa,sb;   REAL a,b,fa,fb,fx;
COMMENT MACHINE: fpnmed also needs fpemin,fpbsm1,fpbsp1,fpfmin,
   fpfeps,fpnmin,fpe,fpf,fpn, and neibor from appendix A;
IF abs(ain)>abs(bin)      THEN BEGIN b_ain; a_bin END
ELSE IF abs(ain)<abs(bin) THEN BEGIN a_ain; b_bin END
     ELSE RETURN(IF ain=bin THEN ain ELSE 0.);
sb_sign(b);
IF a=0. THEN BEGIN sa_sb; ea_fpemin; fa_fpfmin;
   IF abs(bin)>fpnmin THEN b_neibor(b,a)   END
ELSE BEGIN sa_sign(a); ea_fpe(a); fa_abs(fpf(a)) END;
eb_fpe(b); fb_abs(fpf(b));
COMMENT (edist DIV 2) means edist/2 truncated, not rounded;
IF sa=sb THEN BEGIN edist_eb-ea; ex_eb-(edist DIV 2);
   fx_fa+(fb-fa)/2.;
   IF edist MOD 2 = 1 THEN
      IF fx GEQ fpbsp1 THEN fx_fx-fpbsm1
      ELSE BEGIN fx_fx+fpbsm1; ex_ex-1 END END
ELSE BEGIN edist_1+ea+eb-2*fpemin; ex_eb-(edist DIV 2);
   fx_(fb-fa)/2.;
   IF edist MOD 2 = 0 THEN fx_fx+fpbsp1
   ELSE IF fx GEQ 0. THEN fx_fx+fpfmin
        ELSE BEGIN fx_fx+1.;
           IF fx=1. THEN fx_fpfmin ELSE ex_ex-1 END END;
b_sb*fpn(ex,fx);
RETURN ( IF b=bin THEN ain ELSE b)   END "fpnmed";
```

```
COMMENT          Appendix C: Algorithm S;

REAL PROCEDURE root(REAL ain,bin,tol; REAL PROCEDURE f);
BEGIN "root"   REAL a,b,c,fa,fb,fc,fx,q,r,u,v,x;
INTEGER i,j,la,mb,nb;   LABEL bisect,funct,fail4;
COMMENT MACHINE: root also needs fpnmax from appendix A,
                 and combin,fpnmed from appendix B;
q_.5; v_8.; mb_-2; i_1; nb_12; j_0;   COMMENT initial controls;
s_1; a_ain; fa_f(a); x_bin; fx_f(x);  COMMENT initial brackets;
IF abs(fx) GEQ abs(fa) THEN BEGIN b_x; fb_fx END
ELSE BEGIN b_a; fb_fa; a_x; fa_fx END;
WHILE TRUE DO BEGIN                   COMMENT iterative loop;
   la_IF sign(a)=sign(b) THEN abs(b-a)<tol
      ELSE IF a<b THEN b<tol+a ELSE a<tol+b;
   IF la THEN RETURN(a);              COMMENT tolerance met;
   IF j>nb THEN BEGIN s_5; nb_nb-1 MAX 4; mb_mb+1 MAX 4 END
   ELSE BEGIN r_fa/fb; r_r/(r-1.) END;
   CASE s-1 OF BEGIN   COMMENT select the sth of 5 strategies.
       Do only the sth BEGIN directly below "Do" on this line;
       BEGIN r_r END;                 COMMENT LI;
       BEGIN r_r/(1.-u);              COMMENT QI;
          IF r>.5 THEN GOTO bisect END;
       BEGIN r_u/(1.-u);              COMMENT LE;
          IF abs(a)>abs(b) THEN r_r*((1.-c/a)/(b/a-1.))
          ELSE r_r*((a/b-c/b)/(1.-a/b));
          IF r>.5 THEN BEGIN s_4; GOTO bisect END END;
       BEGIN r_v*r;                   COMMENT ML;
          IF r>.5 THEN GOTO bisect END;
bisect:BEGIN i_i+1; j_-1; r_.5;       COMMENT MB;
          IF q<1. THEN BEGIN
             u_IF abs(a)<abs(b) THEN a/b ELSE b/a;
             IF u<.5 THEN BEGIN
                x_combin(fpnmed(a,b),combin(a,b,r),q);
                q_q*q; GOTO funct END
             ELSE q_1. END END END;   COMMENT end strategies;
   x_combin(a,b,r);                   COMMENT interpolate;
   funct:   IF  x=a  THEN RETURN(x);  COMMENT crossover pair a,b;
   fx_f(x); IF fx=0. THEN RETURN(x);  COMMENT machine zero;
   j_j+1;   la_(abs(fx)<abs(fa));     COMMENT la is boolean;
   IF sign(fx)=sign(fb) THEN BEGIN    COMMENT success;
      CASE s-1 OF BEGIN               COMMENT see previous CASE;
         BEGIN IF la THEN BEGIN s_2; u_fx/fb END END;
         BEGIN s_1 END;
         BEGIN IF la THEN BEGIN s_2; u_fx/fc END ELSE s_1 END;
         BEGIN s_1; v_8 END;
         BEGIN IF i GEQ mb THEN s_4 END END;
      IF la THEN BEGIN b_a; fb_fa; a_x; fa_fx END
      ELSE BEGIN b_x; fb_fx END END   COMMENT end success;
   ELSE BEGIN                         COMMENT failure;
      CASE s-1 OF BEGIN
         BEGIN IF la THEN BEGIN s_2; u_fx/fa END ELSE s_4 END;
         BEGIN IF la THEN BEGIN s_3; u_fx/fa; c_a; fc_fa END
               ELSE s_4 END;
         BEGIN GOTO fail4 END;
  fail4: BEGIN s_5; v_v+v; i_0; j_0; mb_mb+1 END;
         BEGIN s_s END END;
      IF abs(fx) LEQ abs(fb) THEN BEGIN a_x; fa_fx END
      ELSE BEGIN a_b; fa_fb; b_x; fb_fx END END END END "root";
```

Laboratory of Applied Studies
Division of Computer Research and Technology
National Institutes of Health
Bethesda, Maryland 20205

National Institutes of Health
Bethesda, Maryland 20205

1. R. P. BRENT, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

2. J. C. P. BUS & T. J. DEKKER, "Two efficient algorithms with guaranteed convergence for finding a zero of a function," *ACM Trans. Math. Software*, v. 1, no. 4, 1975, pp. 330–345.

3. T. J. DEKKER, "Finding a zero by means of successive linear interpolation," in *Constructive Aspects of the Fundamental Theorem of Algebra* (B. Dejon and P. Henrici, eds.), Wiley-Interscience, London, 1969.

4. T. J. DEKKER, "Correctness proof and machine arithmetic," in *Performance Evaluation of Numerical Software* (L. D. Fosdick, ed.), Elsevier, New York, 1979.

5. G. E. FORSYTHE, M. A. MALCOLM & C. B. MOLER, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, N. J., 1977.

6. W. M. KAHAN, "Personal calculator has key to solve any equation $f(x) = 0$," *Hewlett-Packard J.*, Dec., 1979, pp. 20–26.

7. A. M. OSTROWSKI, *Solution of Equations and Systems of Equations*, Academic Press, New York, 1960; or 2nd ed., 1966.

8. D. STEVENSON & IEEE TASK P754 (A WORKING GROUP), "A proposed standard for binary floating-point arithmetic," *Computer*, v. 12, no. 3, 1981, pp. 51–62.