An algorithm with guaranteed convergence for finding a zero of a function

R. P. Brent*

Computer Science Department, Stanford University, Stanford, California 94305, USA

An algorithm is presented for finding a zero of a function which changes sign in a given interval. The algorithm combines linear interpolation and inverse quadratic interpolation with bisection. Convergence is usually superlinear, and is never much slower than for bisection. ALGOL 60 procedures are given.

(Received August 1970, Revised March 1971)

1. Introduction

Let f be a real-valued function, defined on the interval $[a, b]$, with $f(a)\,f(b) \leqslant 0$. f need not be continuous on $[a, b]$: for example, f might be a limited-precision approximation to some continuous function (see Forsythe, 1969). We want to find an approximation $\hat{\zeta}$ to a zero ζ of f, to within a given positive tolerance 2δ, by evaluating f at a small number of points. Of course, if f is discontinuous then there may be no zero in $[a, b]$, so we shall be satisfied if f takes both non-negative and non-positive values in $[\hat{\zeta} - 2\delta, \hat{\zeta} + 2\delta] \cap [a, b]$. Clearly, such a $\hat{\zeta}$ may always be found by bisection in about $\log_2 [(b - a)/\delta]$ steps, and this is the best that we can do for arbitrary f. We shall describe an algorithm which is never much slower than bisection, but which has the advantage of superlinear convergence to a simple zero ζ, if we ignore rounding errors and suppose that f is continuously differentiable near ζ. This means that, in practice, convergence is often much faster than for bisection.

2. Dekker's algorithm

The algorithm described here is similar to an algorithm, which we call Dekker's algorithm for short, variants of which have been given by van Wijngaarden, Zonneveld, and Dijkstra (1963), Wilkinson (1967), Peters and Wilkinson (1969), and Dekker (1969). We wish to emphasise that, although these variants of Dekker's algorithm have proved satisfactory in most practical cases, none of them guarantees convergence in less than about $(b - a)/\delta$ function evaluations (examples are given in Section 3 below). Our algorithm, on the other hand, must converge within about $(\log_2 [(b - a)/\delta])^2$ function evaluations. For example, typically we might have $b - a = 1$ and $\delta = 10^{-12}$, giving 10^{12} and 1,600 function evaluations respectively. On well-behaved functions, e.g. polynomials of moderate degree with well-separated roots, our algorithm has proved to be at least as fast as Dekker's, often slightly faster, so there is no extra price to pay for the improvement in the guaranteed rate of convergence. Of course, both our algorithm and Dekker's are much faster than bisection on well-behaved functions.

3. The algorithm

To avoid repetition, we assume that the reader is familiar with Peters and Wilkinson (1969) or Dekker (1969), and merely point out the differences between our algorithm and Dekker's. The algorithm is defined precisely by the ALGOL 60 procedure *zero* given in the Appendix.

At a typical step we have three points a, b and c such that $f(b).f(c) \leqslant 0$, $|f(b)| \leqslant |f(c)|$, and a may coincide with c. The points a, b and c change during the algorithm, but there should be no confusion if we omit subscripts. b is the best approximation so far to ζ, a is the previous value of b, and ζ must lie between b and c (initially $a = c$).

If $f(b) = 0$ then we are finished (the ALGOL procedure given by Dekker (1969) does not recognise this case, and can take a large number of small steps if f vanishes on an interval, which may happen because of underflow).

If $f(b) \neq 0$ then let $m = \frac{1}{2}(c - b)$. We prefer not to return with $\hat{\zeta} = \frac{1}{2}(b + c)$ as soon as $|m| \leqslant 2\delta$, for if superlinear convergence has set in then b is probably a much better approximation to ζ than $\frac{1}{2}(b + c)$ is. Instead, we return with $\hat{\zeta} = b$ if $|m| \leqslant \delta$ (so the error is no more than δ if, as is often true, f is nearly linear between b and c), and otherwise interpolate (or extrapolate) f linearly between a and b, giving a new point i (see Section 4 for inverse quadratic interpolation). To avoid the possibility of overflow or division by zero we find i as $b + p/q$, and the division is not performed if $2|p| \geqslant 3|m.q|$, for then i is not needed anyway. The reason why the simpler criterion $|p| \geqslant |m.q|$ is not used is explained in Section 4. Since $0 < |f(b)| \leqslant |f(a)|$ (see Section 4), we can safely compute $s = f(b)/f(a)$, $p = \pm(a - b)s$, and $q = \mp(1 - s)$.

Define $b'' = \begin{cases} i \text{ if } i \text{ lies between } b \text{ and } b + m \text{ ('interpolation')}, \\ b + m \text{ otherwise ('bisection')}, \end{cases}$

and $\quad b' = \begin{cases} b'' \text{ if } |b - b''| > \delta, \\ b + \delta.\,\mathrm{sign}(m) \text{ otherwise (a 'step of } \delta\text{')}. \end{cases}$

Dekker's algorithm takes b' as the next point at which f is evaluated, forms a new set $\{a, b, c\}$ from the old set $\{b, c, b'\}$, and continues. Unfortunately, it is easy to construct a function f for which steps of δ are taken every time, so about $(b - a)/\delta$ function evaluations are required for convergence. For example, let

$$f(x) = \begin{cases} 2^{x/\delta} \text{ for } a + \delta \leqslant x \leqslant b, \\ -\left(\dfrac{b - a - \delta}{\delta}\right).2^{b/\delta} \text{ for } x = a, \\ \text{arbitrary for } a < x < a + \delta. \end{cases}$$

The first linear interpolation gives the point $b - \delta$, the next (an extrapolation) gives $b - 2\delta$, the next $b - 3\delta$, and so on.

Even if steps of δ are avoided, the asymptotic rate of convergence of successive linear interpolation may be very slow if f has a zero of sufficiently high multiplicity. An example for which convergence is worse than linear (Brent, 1971) is

$$f(x) = \begin{cases} 0 \text{ if } x = 0, \\ x.\exp(-x^{-2}) \text{ if } x \neq 0, \end{cases}$$

on an interval containing the origin. These examples are rather artificial, and unless an extended exponent range is used (see Section 8) we may be saved by underflow, but it is clear that with Dekker's algorithm convergence may occasionally be very slow.

Our main modification of Dekker's algorithm ensures that

*Present Address: P.O. Box 218, Yorktown Heights, New York 10598, USA.

a bisection is done at least once in every $2.\log_2(|b-c|/\delta)$ consecutive steps. The modification is this: let e be the value of p/q at the step before the last one. If $|e| < \delta$ or $|p/q| \geqslant \frac{1}{2}|e|$ then we do a bisection, otherwise we do either a bisection or an interpolation just as in Dekker's algorithm. Thus $|e|$ decreases by at least a factor of two on every second step, and when $|e| < \delta$ a bisection must be done. After a bisection we take $e = m$ for the next step.

A simpler idea is to take e as the value of p/q at the last step, but this slows down convergence for well-behaved functions by causing unnecessary bisections. With the better choice of e, our experience has been that convergence is always at least as fast as for Dekker's algorithm.

4. Inverse quadratic interpolation

If the three current points a, b and c are distinct, we can find the point i by inverse quadratic interpolation, i.e. fitting x as a quadratic in y, instead of by linear interpolation using just a and b. For well-behaved functions this device saves about 0·5 function evaluations per zero on the average. Inverse interpolation is used because with direct quadratic interpolation we have to solve a quadratic equation for i. Cox (1970) gives another way of avoiding this problem. (See also Ostrowski (1966), Ch. 11.)

Care must be taken to avoid overflow or division by zero when computing the new point i. Since b is the most recent approximation to the root and a is the previous value of b, we do a bisection if $|f(b)| \geqslant |f(a)|$. Otherwise we have $|f(b)| < |f(a)| \leqslant |f(c)|$, so a safe way to find i is to compute

$$r_1 = f(a)/f(c),\ r_2 = f(b)/f(c),\ r_3 = f(b)/f(a),$$
$$p = \pm r_3[(c-b)\,r_1(r_1 - r_2) - (b-a)(r_2 - 1)],$$

and

$$q = \mp(r_1 - 1)(r_2 - 1)(r_3 - 1).$$

Then $i = b + p/q$, but as before we do not perform the division unless it is safe to do so (if bisection is to be done then i is not needed anyway). When inverse quadratic interpolation is used, the interpolating parabola cannot be a good approximation to f unless it is single-valued between $(b, f(b))$ and $(c, f(c))$, so it is natural to accept the point i if it lies between b and c and up to three-quarters of the way from b to c (consider the limiting case where the interpolating parabola has a vertical tangent at c and $f(b) = -f(c)$). Thus i will be rejected if

$$2|p| \geqslant \frac{3}{2}\,|(c-b)q|\,.$$

5. Superlinear convergence

Ostrowski (1966) shows that if f is C^2 in a neighbourhood of a simple zero ζ, then successive linear interpolation from a sufficiently good approximation gives superlinear convergence to ζ, with order at least $\frac{1}{2}(1+\sqrt{5}) = 1{\cdot}618\ldots$. We remark that this result holds under the weaker hypothesis that f has a Lipschitz continuous derivative near ζ. In fact, convergence is superlinear, in the sense that $\lim_{n\to\infty} |x_n - \zeta|^{1/n} = 0$, if f is C^1 near ζ. If f' is Lipschitz continuous near ζ then the order is at least 1·618... when inverse quadratic interpolations are performed in place of some of the linear interpolations. For proofs of these results, see Brent (1971).

Ignoring the effect of rounding errors and the tolerance δ, we see, as in Dekker (1969), that the algorithm will eventually stop doing bisections when it is approaching a simple zero of a C^1 function, so convergence will be superlinear. In practice, convergence for well-behaved functions is fast, and the stopping criterion is usually satisfied in a few steps once superlinear convergence sets in.

6. The tolerance

As in Peters and Wilkinson (1969), the tolerance (2δ) is a combination of a relative tolerance (4ε) and an absolute tolerance $(2t)$. At each step we take $\delta = 2\varepsilon|b| + t$, where b is the current best approximation to ζ, ε = *macheps* is the relative machine precision ($\beta^{1-\tau}$ for τ-digit truncated floating-point arithmetic with base β, and half this for rounded arithmetic), and t is a positive absolute tolerance. Since δ depends on b, which could lie anywhere in the given interval, we should replace δ by its positive minimum over the interval in the upper bound for the number of function evaluations required. In the ALGOL procedures the variable *tol* is used for δ.

7. The effect of rounding errors

The ALGOL procedures have been written so that rounding errors in the computation of i, m etc. cannot prevent convergence with the above choice of δ. The number 2ε in the definition of δ (Section 6) may be increased if a higher relative error is acceptable, but it should not be decreased, for then rounding errors might prevent convergence.

The bound for $|\hat{\zeta} - \zeta|$ has to be increased slightly if we take rounding errors into account. Suppose that, for floating-point numbers x and y, the computed arithmetic operations satisfy

$$fl(x \times y) = xy(1+\varepsilon_1)$$

and

$$fl(x \pm y) = x(1+\varepsilon_2) \pm y(1+\varepsilon_3),$$

where $|\varepsilon_i| \leqslant \varepsilon$ for $i = 1, 2$ and 3 (see Wilkinson, 1963). We also assume that $fl(|x|) = |x|$ exactly, for any floating-point number x. The algorithm computes approximations

$$\tilde{m} = fl(0{\cdot}5 \times (c-b))$$

and

$$\tilde{tol} = fl(2 \times \varepsilon \times |b| + t)$$

to m and *tol*, where ζ lies between b and c, and the algorithm terminates with $\hat{\zeta} = b$ only when

$$|\tilde{m}| \leqslant \tilde{tol}$$

(unless $f(b) = 0$, when $\hat{\zeta} = \zeta = b$). Our assumptions give

$$|\tilde{m}| \geqslant \tfrac{1}{2}[|c-b| - \varepsilon(|b| + |c|)]\,(1-\varepsilon),$$

and similarly

$$\tilde{tol} \leqslant (2\varepsilon|b| + t)(1+\varepsilon)^3,$$

so $|\tilde{m}| \leqslant \tilde{tol}$ implies that

$$|c-b| \leqslant \left(\frac{2}{1-\varepsilon}\right)(2\varepsilon|b| + t)(1+\varepsilon)^3 + \varepsilon(|b| + |c|).$$

Since $|\hat{\zeta} - \zeta| \leqslant |c-b|$ and $b = \hat{\zeta}$, this gives

$$|\hat{\zeta} - \zeta| \leqslant 6\varepsilon|\zeta| + 2t,$$

neglecting terms of order εt and $\varepsilon^2|\zeta|$. Usually the error is less than half this bound (see Section 3).

Of course, it is the user's responsibility to consider the effect of rounding errors in the computation of f. The ALGOL procedures only guarantee to find a zero ζ of the *computed* function f to the accuracy discussed above, and ζ may be nowhere near a root of the mathematically defined function that the user is really interested in.

8. Extended exponent range

In some applications the range of f may be larger than is allowed for standard floating-point numbers. Hence, in the Appendix we give an ALGOL procedure (*zero2*) which accepts $f(x)$ represented as a pair $(y(x), z(x))$, where $f(x) = y(x).2^{z(x)}$ (y real, z integer). Thus *zero2* will accept functions in the same representation as is assumed by Peters and Wilkinson (1969), although *zero2* does not require that $1/16 \leqslant |y(x)| < 1$ (unless $y(x) = 0$), and could be simplified slightly if this assumption were made.

9. Practical tests

The ALGOL procedures *zero* (for standard floating-point numbers) and *zero2* (for floating-point with an extended exponent range) were tested using ALGOL W (Wirth and Hoare, 1966) on an IBM 360/67 and a 360/91 with machine precision $16^{-13} \doteq 2{\cdot}5 \times 10^{-16}$. The number of function evaluations for convergence was never more than three times greater than would be needed if bisection were used, even for the pathological functions given in Section 3, and for these functions Dekker's algorithm takes more than 10^6 function evaluations. *Zero2* has been tested extensively with eigenvalue routines, and in this application it usually takes the same or one less function evaluation per eigenvalue than Dekker's algorithm, and considerably less than bisection (numerical results are given in Brent, 1971).

10. Concluding remarks

Our algorithm appears to be at least as fast as Dekker's on well-behaved functions, and, unlike Dekker's, it is guaranteed to converge in a reasonable number of steps for any function. The ALGOL procedures *zero* and *zero2* given in the Appendix have been written to avoid problems with rounding errors or overflow, and floating-point underflow is not harmful as long as the result is set to zero. (A FORTRAN translation of procedure *zero* is given in Brent (1971).)

A recent paper by Cox (1970) gives an algorithm which combines bisection with interpolation, using both f and f'. This algorithm may fail to converge in a reasonable number of steps in the same way as Dekker's. A simple modification, similar to the one that we have given in Section 3 for Dekker's algorithm, will remedy this defect without slowing the rate of convergence for well-behaved functions.

Finally, we note that golden section search and a method of successive parabolic interpolation (Jarratt, 1967) can be combined to give an algorithm for finding a local minimum of a function of one variable, just as bisection and successive linear interpolation can be combined to give an algorithm for finding a zero. The minimisation algorithm always converges nearly as fast as would Fibonacci search, and it converges superlinearly if f has a positive and continuous second derivative near the minimum (Brent, 1971).

Acknowledgement

The author wishes to thank Professors G. E. Forsythe and G. H. Golub for their advice and encouragement, the referee for his helpful comments, and the CSIRO for its support.

Appendix: Algol 60 procedures

```
real procedure zero (a, b, macheps, t, f);
value a, b, macheps, t; real a, b, macheps, t;
real procedure f;
  begin comment:
```

Procedure *zero* returns a zero x of the function f in the given interval $[a, b]$, to within a tolerance $6macheps\ |x| + 2t$, where *macheps* is the relative machine precision and t is a positive tolerance. The procedure assumes that $f(a)$ and $f(b)$ have different signs;

```
  real c, d, e, fa, fb, fc, tol, m, p, q, r, s;
  fa := f(a); fb := f(b);
  int: c := a; fc := fa; d := e := b - a;
  ext: if abs(fc) < abs(fb) then
    begin a := b; b := c; c := a;
    fa := fb; fb := fc; fc := fa
    end;
  tol := 2 × macheps × abs(b) + t; m := 0·5 × (c - b);
  if abs(m) > tol ∧ fb ≠ 0 then
    begin comment: See if a bisection is forced;
    if abs(e) < tol ∨ abs(fa) ⩽ abs(fb) then d := e := m else
      begin s := fb/fa; if a = c then
        begin comment: Linear interpolation;
        p := 2 × m × s; q := 1 - s
        end
      else
        begin comment: Inverse quadratic interpolation;
        q := fa/fc; r := fb/fc;
        p := s × (2 × m × q × (q - r) -
                                    (b - a) × (r - 1));
        q := (q - 1) × (r - 1) × (s - 1)
        end;
      if p > 0 then q := -q else p := -p;
      s := e; e := d;
      if 2 × p < 3 × m × q - abs(tol × q) ∧
                                  p < abs(0·5 × s × q) then
      d := p/q else d := e := m
      end;
    a := b; fa := fb;
    b := b + (if abs(d) > tol then d else if m > 0 then
              tol else -tol);
    fb := f(b);
    go to if fb > 0 ≡ fc > 0 then int else ext
    end;
  zero := b
  end zero
real procedure zero2 (a, b, macheps, t, f);
value a, b, macheps, t; real a, b, macheps, t; procedure f;
  begin comment:
```

Procedure *zero2* finds a *zero* of the function $\tilde{f}$ in the same way as procedure *zero* does, except that the procedure $f(x, y, z)$ returns y (real) and z (integer) so that $\tilde{f}(x) = y.2^z$. Thus underflow and overflow can be avoided with a very large function range;

```
  real procedure pwr2 (x, n); value x, n; real x; integer n;
  comment: This procedure is machine-dependent. It computes
  x.2^n for n ⩽ 0, avoiding underflow in intermediate results;
  pwr2 := if n > -200 then x × 2 ↑ n else
          if n > -400 then (x × 2 ↑ (-200)) ×
                                  2 ↑ (n + 200) else
          if n > -600 then ((x × 2 ↑ (-200)) ×
                                  2 ↑ (-200)) × 2 ↑ (n + 400)
          else 0;
  integer ea, eb, ec;
  real c, d, e, fa, fb, fc, tol, m, p, q, r, s;
  f(a, fa, ea); f(b, fb, eb);
  int: c := a; fc := fa; ec := ea; d := e := b - a;
  ext: if (ec ⩽ eb ∧ pwr2(abs(fc), ec - eb) < abs(fb))
       ∨ (ec > eb ∧ pwr2(abs(fb), eb - ec) ⩾ abs(fc)) then
    begin a := b; fa := fb; ea := eb;
          b := c; fb := fc; eb := ec;
          c := a; fc := fa; ec := ea
    end;
  tol := 2 × macheps × abs(b) + t; m := 0·5 × (c - b);
  if abs(m) > tol ∧ fb ≠ 0 then
    begin if abs(e) < tol ∨
    (ea ⩽ eb ∧ pwr2(abs(fa), ea - eb) ⩽ abs(fb)) ∨
    (ea > eb ∧ pwr2(abs(fb), eb - ea) ⩾ abs(fa)) then
    d := e := m else
      begin s := pwr2(fb, eb - ea)/fa; if a = c then
        begin p := 2 × m × s; q := 1 - s end
      else
        begin q := pwr2(fa, ea - ec)/fc;
        r := pwr2(fb, eb - ec)/fc;
        p := s × (2 × m × q × (q - r) - (b - a) × (r - 1));
        q := (q - 1) × (r - 1) × (s - 1)
        end;
```

```
    if p > 0 then q := −q else p := −p; s := e; e := d;
    if 2 × p < 3 × m × q − abs(tol × q) ∧
        p < abs(0·5 × s × q) then
    d := p/q else d := e := m
    end;
  a := b; fa := fb; ea := eb;
  b := b + (if abs(d) > tol then d else if m > 0 then
            tol else −tol);
  f(b, fb, eb);
  go to if fb ≻ 0 ≡ fc < 0 then int else ext
  end;
zero2 := b
end zero2
```

References

BRENT, R. P. (1971). *Algorithms for minimization without derivatives*, Prentice Hall, Englewood Cliffs, New Jersey (to appear).

COX, M. G. (1970). A bracketing technique for computing a zero of a function, *The Computer Journal*, Vol. 13, pp. 101-102.

DEKKER, T. J. (1969). Finding a zero by means of successive linear interpolation, in *Constructive aspects of the fundamental theorem of algebra*, edited by B. Dejon and P. Henrici, Interscience, New York.

FORSYTHE, G. E. (1969). Remarks on the paper by Dekker, in *Constructive aspects of the fundamental theorem of algebra*, edited by B. Dejon and P. Henrici, Interscience, New York.

JARRATT, P. (1967). An iterative method for locating turning points, *The Computer Journal*, Vol. 10, pp. 82-84.

OSTROWSKI, A. M. (1966). *Solution of equations and systems of equations*, Academic Press, New York.

PETERS, G., and WILKINSON, J. H. (1969). Eigenvalues of $Ax = \lambda Bx$ with band symmetric A and B, *The Computer Journal*, Vol. 12, pp. 398-404.

VAN WIJNGAARDEN, A., ZONNEVELD, J. A., and DIJKSTRA, E. W. (1963). Programs AP 200 and AP 230 de serie AP 200, edited by T. J. Dekker, The Mathematical Centre, Amsterdam.

WILKINSON, J. H. (1963). *Rounding errors in algebraic processes*, HMSO, London.

WILKINSON, J. H. (1967). Two algorithms based on successive linear interpolation, Technical Report CS 60, Computer Science Department, Stanford University.

WIRTH, N., and HOARE, C. A. R. (1966). A contribution to the development of ALGOL, *CACM*, Vol. 9, pp. 413-431.