Research Report
Pages 19

Safeguarded Zero-Finding Methods

Sumanth Hegde and Saurav Kasera

Under the guidance of:

Margaret H. Wright
Silver Professor of Computer Science
Chair, Computer Science Department
Courant Institute of Mathematical Sciences
New York University

Safeguarded Zero-Finding Methods

Sumanth Hegde, Saurav Kasera & Margaret Wright
{ snhegde, skasera } @ nyu.edu, mhw@cs.nyu.edu

Abstract

The purpose of this research project is to analyze safeguarded zero finding methods. Root finding is a very common and important problem. There are many methods for root finding but none of them are fool proof, especially when the nature of the function is unknown. We assume a black box that evaluates the function for different values during the course of this project. We digress sometimes trying to analyze the nature of specific functions to know the associated problems these might present to our black box model. We are presented with the problem of applying safeguards to conventional methods that would make our root calculation robust. We will analyze many basic methods like bisection method, which is the safest as it always converges but is slow and will also analyze other complex but efficient methods. These efficient methods might not work in all cases and that is where we will detect exceptional conditions and try to put safeguards on them. Another related question that we try to answer is how effective are these safeguards and where they might fail. Our role in this project is more of analysts. We make extensive use of the material from Kahan[1] and Brent[2].

1. Introduction

Zero finding is same as solving equations. The general form of an equation is $f(x) = c$. When we move all the terms of the right hand side to left hand side, we get an equation of the form $f(x) - c = 0$ which further presents us with the problem of finding the values of x where $f(x) - c = 0$. These are often called the roots of the equation. When there is only one independent variable, the problem is called one-dimensional root finding problem. In this project, we primarily deal with the problem of solving one-dimensional non–linear equations using safeguarded methods. For a short historical perspective on safeguarded methods refer [1]. We illustrate all the numerical examples by Octave [11], an open source free mathematical tool.

Root finding in general

Except for the case of linear equations, most of the algorithms for solving non-linear equations are iterative. They start with one or two initial guess values and improve upon the guess value in successive iterations. For smoothly varying functions, most of the algorithms converge to the right value given a good enough guess (a useful and a necessary criterion for a good enough guess would be a guess ***x*** where ***f(x)*** is defined). The initial guess is often very critical because a good guess often leads to decreased computational effort and increased understanding. Hamming's motto, "The purpose of computing is insight, not numbers", is strongly applicable to zero finding problems. But it is observed that this is not as easy as it seems. Estimating good guesses often requires deep insight and intuition. Guessing becomes more difficult in multi–dimensional cases.

We now look at some of the conventional methods of root finding and the drawbacks, which they suffer.

2. Root Finding Methods & Safeguarding Needs

Bisection Method

The bisection method is the simplest of all. It is guaranteed to converge for all continuous functions. Further, it is always possible to bind the number of steps required for a given accuracy. The method requires two guesses that have opposite signs for ***f(x)***. The algorithm proceeds by taking the mid point of the two guesses and then iterating with the best bracket for the next round. The disadvantage is linear convergence, as it doesn't take advantage of the nature of functions. It also requires two initial guesses with opposite signs. For a function of the form ***1 / (x – c)***, the bisection method will readily converge to ***c***, but as seen it is nowhere near the correct value (the value ***f(x)*** will shoot up as ***x*** approaches ***c***).

The function: ***1 / (x – 2)***, when evaluated using bisection method under Octave, proved this point. The shooting up of ***f(x)*** is quiet expected, but the resulting root converging to ***c*** is not.

Table 1 shows the values for first 50 iterations using bisection method. As we can see the root converges to ***c*** = 2.0000, which is not correct.

0.00000	2.01172	1.99999	2.00000	2.00000
0.00000	2.00195	2.00000	2.00000	2.00000
5.00000	1.99707	2.00000	2.00000	2.00000
2.50000	1.99951	2.00000	2.00000	2.00000
1.25000	2.00073	2.00000	2.00000	2.00000
1.87500	2.00012	2.00000	2.00000	2.00000
2.18750	1.99982	2.00000	2.00000	2.00000
2.03125	1.99997	2.00000	2.00000	2.00000
1.95312	2.00005	2.00000	2.00000	2.00000
1.99219	2.00001	2.00000	2.00000	2.00000

Table 1: Iterate Values for *f(x) = 1 / (x - 2)* with Bisection Method

The function ***1 / (x – 2)*** is shown in fig. 1.

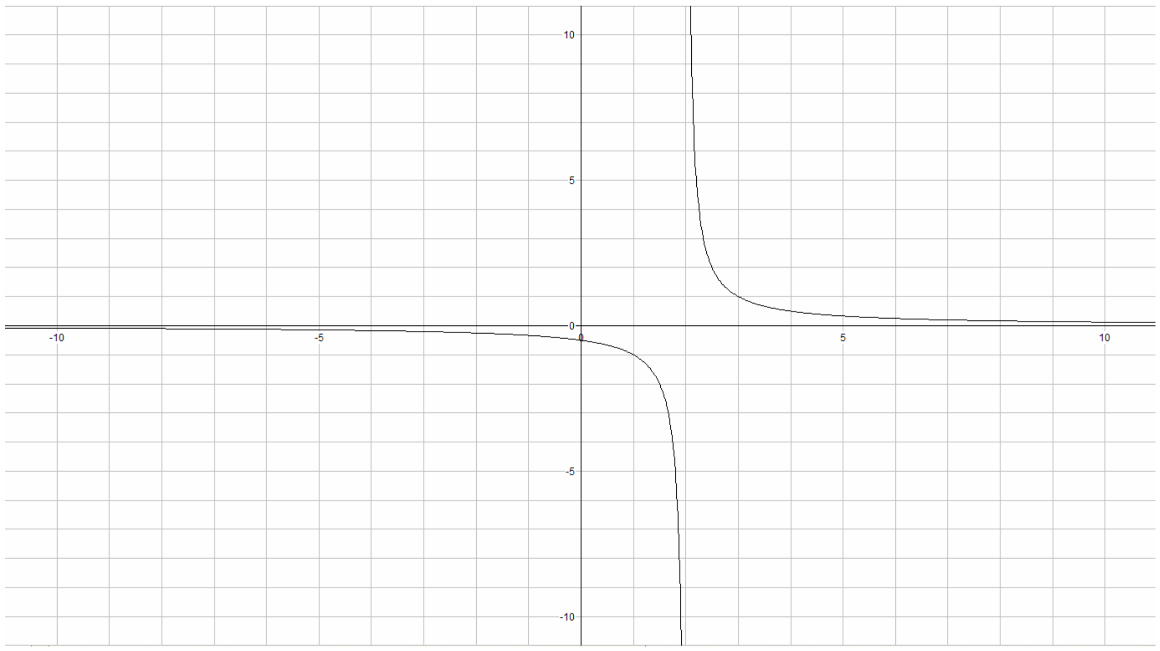

Figure 1: ***f(x) = 1 / (x - 2)***

In fact, this function doesn't have a root, so it is important that we safeguard bisection to detect cases where we don't have any roots. A good idea might be to bound the number of iterations and check the value of ***f(x)*** during the last few iterations. If we get a constant value of ***x*** for more than three to four iterations, put that value of ***x*** in ***f(x)*** and evaluate it. If the evaluated value is not zero then the function may not have a root. Also a non-zero increasing or constant value of x might indicate that the function doesn't have roots. The question of guessing the bound is more intuitive.

Newton's Method

The Newton's method takes advantage of Taylor's expansion for the equation ***f(x)*** where $f(x_0) + (x - x_0) f'(x_0) = 0$.

An estimate of the root, ***x****, can be found from values of the function and its derivative:

$$\hat{x}^* = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The formula can then be used iteratively to obtain improving estimates of the root:

$$\hat{x}^*_{n+1} = \hat{x}^*_n - \frac{f(\hat{x}^*_n)}{f'(\hat{x}^*_n)}$$

Newton's method has quadratic convergence and is very good if the functions are well behaved and when exact arithmetic is used (no round off or truncation). It requires only one guess value with the only constraint that the function should be defined at that point. Obviously, accurate guesses converge fast to the solution but not always.

The disadvantages of Newton's method are that higher-order roots can cause convergence to be slow, and the sequence may take undesirable jumps between roots or take a very large step upon encountering an inflection point.

Newton's Method is not perfect; there are cases where it fails. Both ***f(x)*** and ***f'(x)*** must be continuous and smooth. One case where it fails is when ***f '(x)*** is zero. If ***f'(x)*** is infinite then it also fails because then $x_n+1 = x_n$ and the algorithm stops finding new points.

Let's try and find a root to the equation $f(x) = e^x - 2x = 0$. Notice that $f'(x) = e^x - 2$ so that,

$$x_{n+1} = x_n - \frac{e^{x_n} - 2x_n}{e^{x_n} - 2}$$

If we try an initial value $x_0 = 1$, we find that $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 1\ldots$ In other words, Newton's Method fails to produce a solution. Why is this? Because there is no solution to be found! We could rewrite a solution as $e^x = 2x$. Fig. 2 shows that there is no such solution.

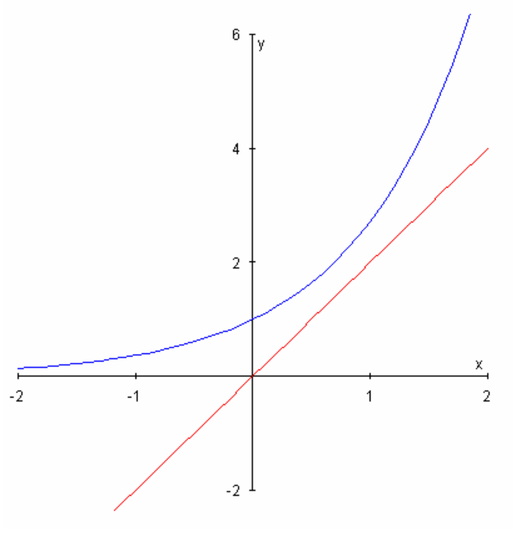

Figure 2: There is no root for equation $f(x) = exp(x) - 2x$

There are also some functions or regions of functions that will cause the algorithm to shoot off to infinity. For example consider function $f(x) = x^{1/3}$.

Fig. 3 shows how this can happen. Newton's method will fail here even when we have roots. The previous guess values calculated would cause the algorithm to shoot the next iterate value, at each step.

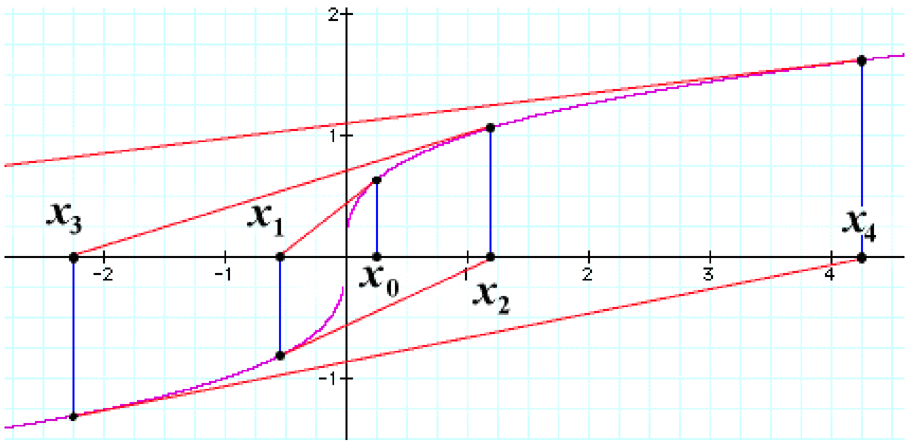

Figure 3: Newton method shoots up for functions like this

Leap-frogging Newton's method

Using Newton's method as an intermediate step, this iterative method approximates numerically the solution of $f(x) = 0$. The order of convergence of the method at a simple root is cubic and the computational efficiency in general is less, but close to that of Newton's method. Like Newton's method, this method [5] requires only function and first derivative evaluations. The method can easily be implemented on computer algebra systems where high machine precision is available.

But even this method, though easy to implement even for high machine precision requirement, will fail for functions, where Newton's method fails. The only advantage of this method is its cubic order of convergence.

Secant Method

The secant method starts with two guesses x_m and x_n. Then we approximate a secant line through $(x_m, f(x_m))$ and $(x_n, f(x_n))$. An improved x_p is the point at which the secant line intersects the x-axis. We repeat the iterations till the required accuracy or machine precision is obtained. The primary advantage of secant method is that it has super linear convergence and it does not require the calculation of the derivative, as was in the case of Newton's. All said and done, the secant method too fails. The most common problem is its oscillatory behavior.

Consider $f(x) = cos(x/3)$, refer fig.4. If we choose x_0 = 600 and x_1 = 1000 as the initial guesses for this curve, the secant method infinitely oscillates between two new guess values c and d.

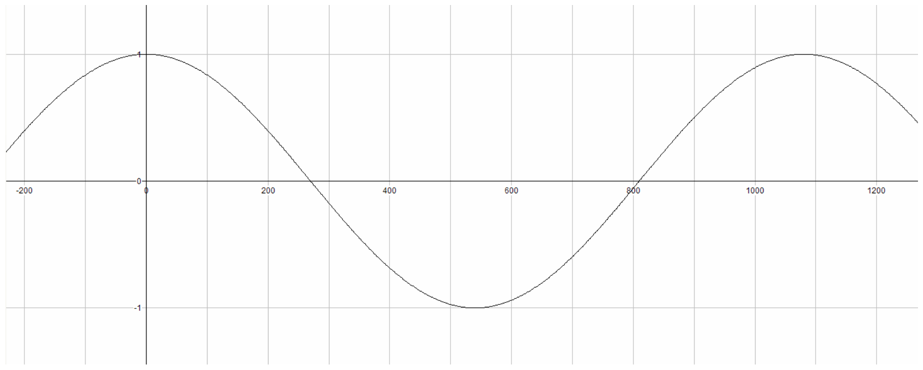

Figure 4: *f(x) = cos(x/3)*

3. Safeguarding the root finding methods

Now it is clear from our above examples that traditional root finding methods are not robust enough for root finding. Therefore, we need to tread a middle approach taking the fast convergence property of algorithms such as Newton's, Secant and Inverse Interpolation and the safety of bisection. This is what has given birth to safeguarded methods that are hybrid of bisection (safe) and other fast convergence methods. If we were given an infinite precision arithmetic facility for root finding, life would be very easy. But this is not the case in real life. Therefore, we add further safeguards that take care of the errors created due to rounding and truncation. Another issue that needs to be addressed is for the functions that don't have a root. We will analyze these issues in details now. We make extensive references to the paper on Solve key for HP34C by Kahan [1].

Many of the solvers we use today are very naïve in their root finding approach. After all they are computing devices and use one of these classical methods for root finding. They don't have the ability to judge the behavior of a function. None of the equation solvers can understand ***f(x)*** and its behavior. Also the point of termination makes the matters worse. How should the solver know when to stop? The easiest and the most widely used method is to do only a finite number of iterations. But this poses 2 problems:

1) How can one be sure that, the value of the last iteration is the root? The correct value may be found in the next 2 or 3 iterations. But solver stops after the prescribed number of steps, even though it was approaching the correct value.

2) How does one know that the value is at least approximation to the correct value? The function might not have root at all.

A good explanation of why equation solving is provably impossible is given in [1]. These questions with profound implications are not answerable by the current methods known to us. We try to make an attempt to answer these questions, though the exact answers are yet to be found.

According to Kahan[1], three major cases need to be safeguarded:

1. When a function ***f(x)*** might not vanish or might not change sign, because no such root exists.

2. When the value of the iterates dither.
3. When the sample value neither diminishes in magnitude nor changes the sign.

In these above cases, most of the methods will fail. These cases are major generalizations. It should be noted that some methods have safeguards against some of these cases, but not all. We provide an analysis of these cases below.

When a function f(x) might not vanish or might not change sign, because no such root exists

There are certain functions, which does not vanish at all. One such function is $\boldsymbol{f(x) = x^2 - 2x + 10}$. The function is shown in fig. 5.

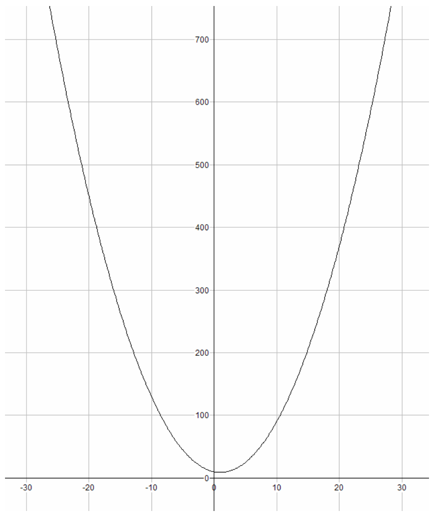

Figure 5: $f(x) = x^2 - 2x + 10$

As we can see, it will never touch the $\boldsymbol{x}$-axis. A way to safeguard these root finding methods for such functions is to stop the calculation where $|\boldsymbol{f(x)}|$ appears to be stationary, near either a local positive minimum of $|\boldsymbol{f(x)}|$ or where the $\boldsymbol{f(x)}$ appears to be constant. This exact method is used in HP-34C equation solver [1].

When the value of the iterates dither

If we use secant method, some functions creates problem. The iterate values will dither.

Consider a function formulated by Kahan [1], ref fig. 6. The secant iteration will cycle endlessly through the values α, β, γ and δ.

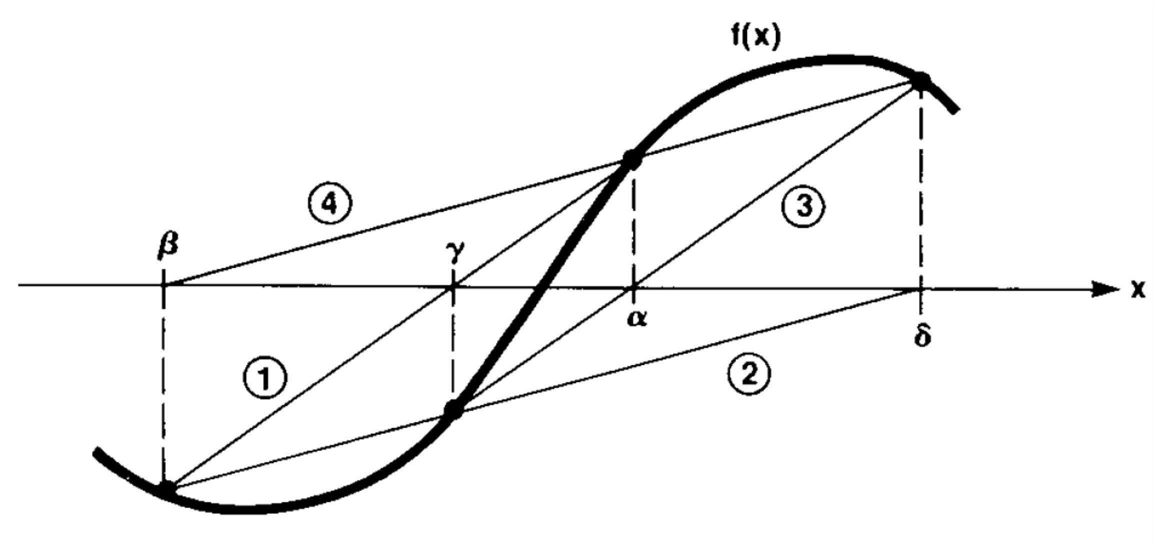

Figure 6: Secant iteration will cycle endlessly

To safeguard the method from such conditions, having discovered two values of ***f(x)*** with opposite signs, we constrain each successive new guess to lie strictly between every two previous guesses at which ***f(x)*** took opposite signs, thereby forcing successive guesses to converge to a place where ***f*** vanishes or reverses sign. This can be accomplished by slightly bending the secant occasionally.

When the value neither diminishes nor changes the sign

When using secant method sometimes we run into problems, when successive samples of ***f(x)*** neither diminish in magnitude nor changes the sign. To safeguard such cases, we need to throw the secant method away and use quadratic interpolation of three points ***[a, f(a)]***, ***[b, f(b)]*** and ***[c, f(c)]***, where ***a*** and ***b*** are previous guesses and c is the current guess, and set ***c*** to the point where the quadratic's derivative vanishes.

Safeguarded root finding methods such as Brent's Method [2], uses this idea to accomplish a hybrid root finding scheme.

The function in fig. 7 as given in Kahan[1] will fall into this category.

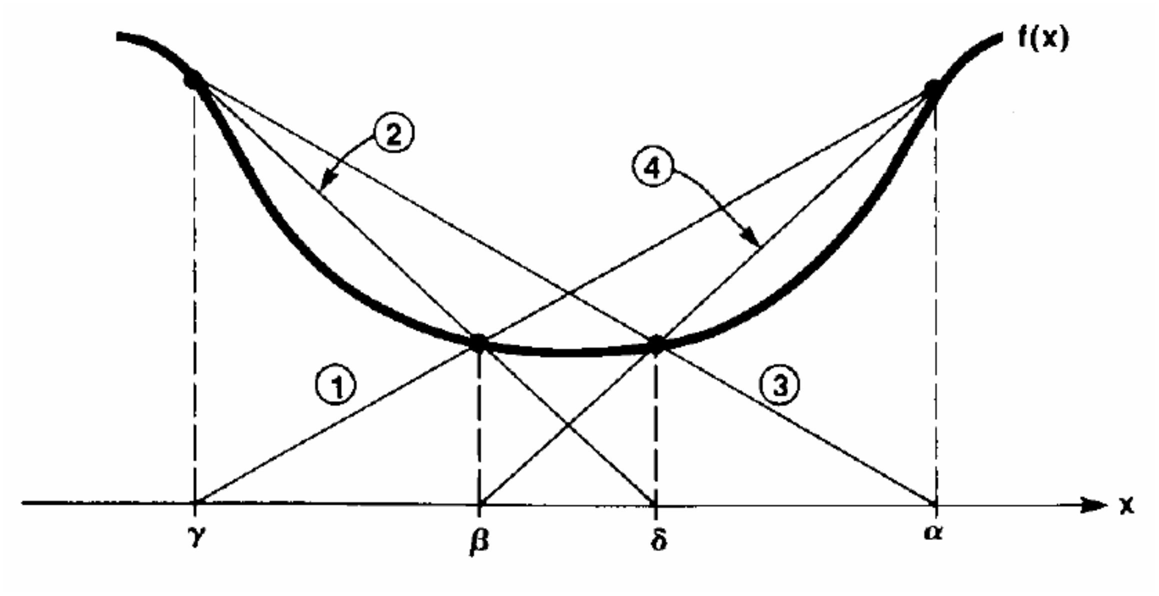

Figure 7: The secant method fails here because of non-diminishing successive samples

4. Safeguarded Methods

Brent's Method

When searching for a real simple root of a real valued function, it is possible to take advantage of the special geometry of the problem, where the function crosses the axis from negative to positive or vice-versa. Brent's method is effectively a safeguarded secant method with inverse quadratic interpolation technique that always keeps a point where the function is positive and one where it is negative so that the root is always bracketed. At any given step, a choice is made between an interpolated (secant) step and bisection in such a way that eventual convergence is guaranteed.

Brent's method is a modification of Dekkar's algorithm. The algorithm is proved to be no worse than the bisection and often converges super linearly for well-behaved functions. For full description refer [2]. The main highlight of the method is that bisection is done at least once in every $2 \log_2 (|b - c| / \delta)$ consecutive steps, where ***b*** & ***c*** are two points such that $|f(b)| \leq |f(c)|$, and δ is the tolerance. ***e*** is the value of p/q at the step before the last one, $p = \pm (a-b)s$ and $q = \pm(1-s)$, where $s = f(a)/f(b)$ on the interval **[a,b]**. If $|e| < \delta$ or $|p/q| \geq \frac{1}{2}|e|$ then it does a bisection, otherwise it does either a bisection or an interpolation. Refer [2] for details.

According to [12], the convergence rate of Secant method is 1.639 and that of inverse quadratic interpolation is 1.839. A confusing question, why Brent's method first uses Secant method instead of inverse quadratic interpolation even though the latter has a better convergence rate, is being worked on.

When Brent's method was tested in Octave for the function ***$f(x) = x^3 - 3x + 6$,*** the value converged to the right value i.e. *-2.35532020386406*. Newton's method will fail for this function.

Even though essentially all of the theory for solving nonlinear equations and local minimization is based on smooth functions, Brent's method is sufficiently robust that you can even get a good estimate for a zero crossing for discontinuous functions.

Safeguarded Newton's method

This method [9] uses Newton's method, but maintains an interval containing the root (as in bisection method). The idea behind this safeguarded method is very simple. If Newton iterate falls outside the interval, switch to bisection for that iteration and then continue with Newton's method.

A general algorithm as given in [9] for this method is:

```
safeNewton <- function(v, f, df, tol=1e-6) {
   newx <- v[2] - f(v[2])/df(v[2])
   # could also do this to v[1] to see if we get a better bracket that way...
   # test to see if we still have a bracket
   f1 <- f(v[1])
   f2 <- f(v[2])
   fn <- f(newx)
   # need to check that [ f1, fn] or [fn, f2] is a bracket and that the bracket is smaller...
   # not going to rely on v being sorted; old will be old side of bracket, v[2] will be new guess
   old <- NULL
   if  (f1*fn < 0) { old <- v[1] }
   if  (f2*fn <0)  { old <- v[2] }
   new <- newx
   if (is.null(old)) {  # do a bisection instead
   middle <- mean(v)
      fm <- f(middle)
      if (fm*f1 < 0) { old <- v[1] } else { old <- v[2] }
      new <- middle
   }
   # pick the smaller interval: old - newx or old-middle since Newton might be safe, but still a bad idea
   #  if ( abs(old-newx) < abs(old-middle) ) { new <- newx } else { new <- middle }  # a logic hole in
here... don't use it.
   c(old, new)  # return the new bracket
}
```

5. Conclusion

We first analyzed some of the conventional root finding methods and their limitations. Bisection always converges but is slow. Newton has quadratic convergence but may fail in some of the cases. Secant is a good alternative to Newton but it oscillates in some of the cases and fails to converge. We then tried to analyze how one can modify present methods to work in most of the situations. We looked at different safeguarding ideas used in the Solve key of HP34C as explained by Kahan [1]. The Solve key works but not always and that provably seem true. In the last section we looked at two safeguarded methods, Brent's algorithm and safeguarded Newton's method. Brent's algorithm [2], which is a combination of secant, inverse quadratic interpolation and bisection, has a guaranteed convergence. As noted by Kahan in his classical paper on the Solve Key of HP34C [1], "root solving is provably impossible", we very much agree with it but not in its entirety because we still want to do more research on this before we fully agree to a statement as strong as that. Our future work would be more in knowing other safeguarded algorithms and developing one of our own.

References

[1] William M. Kahan, Personal Calculator Has Key to Solve Any Equation f(x)=0, *Hewlett-Packard Journal,* December 1979, pp. 20-26.

[2] R. P. Brent, An algorithm with guaranteed convergence for finding a zero of a function, Computer *Journal 14*, 1971, pp. 422-425.

[3] R.P. Brent, Multiple-Precision Zero-Finding Methods And The Complexity of Elementary Function Evaluation, *Analytic Computational Complexity (edited by J. F. Traub)*, Academic Press, New York, 1975, pp. 151–176.

[4] W.B. Gragg and G.W. Stewart, A Stable Variant of the Secant Method for Solving Nonlinear Equations, *SIAM Journal for Numerical Analysis,* Vol. 13, No.6, December 1976, pp. 890-903.

[5] A.B. Kasturiarachi, Leap-frogging Newton's Method, *International Journal for Mathematical Education Sci. Technology,* Vol. 33, No. 4, 2002, pp. 521–527.

[6] Peter Linz and Richard L.C. Wang, Exploring Numerical Methods, *Jones & Bartlett Publishers; Package edition*, December 2002, pp. 171 – 207.

[7] Sou-Cheng Choi, Lecture Notes of Scientific Computing, Lecture 9, Jul 2004.

[8] S.E. Pav, Numerical Methods Course Notes, Ver 0.1, *UCSD Math174, Fall'04,* Dec '04.

[9] Math/Comp 3411 - Numerical Analysis, Andrew Irwin, Mount Allison University.

[10] Martin P. Gelfand, A Numerical Supplement for Undergraduate Physics Using GNU Octave and Gnuplot, January 2005.

[11] Dr. Padma Raghavan, Lecture notes for Introduction to Numerical Analysis I, Fall 2005, Pennsylvania State University.

[12] Micheal T. Heath, http://www.cse.uiuc.edu/heath/scicomp/notes/chap05.pdf.

[13] GNU Octave: the technical computing software, http://pioneer.netserv.chula.ac.th/ %7Eskrung/statistics/Lecture001.html

[14] UBC Calculus Online Course Notes, http://www.ugrad.math.ubc.ca/coursedoc/ math100/ notes/approx/newton.html

[15] Public Presentations of Math 198 Projects, 2004, http://archive.ncsa.uiuc.edu/ Classes/MATH198/alexrein/explanation.htm

[16] Newton's Method, http://www.asp.ucar.edu/colloquium/1992/notes/part1/node79.html

[17] GNU Octave, http://www.octave.org.

Appendix

Octave Scripts

1. Script for Bisection Method

```
#this script takes the inputs a, b, eps, func
# a = lower bound
# b = upper bound
# eps = acceptable error
# func = the function in question

#syntax bisec(1,2,10^-4,@f1)
#where f1 is the name of another m file with the function inside.

# If func(a) and func(b) are different signs (ie negative and posaitive) and func is
continuous,
# then the zero theorem states that a root must exist between them.
# Using this script will then calculate the roots of the function based on the bisection
method,
# storing each iteration through the process as elements in the arrays A, B, and C
respectively.

function[] = bisect(a,b,eps,func)

i=1;
c=(a+b)/2;
n=ceil((log(b-a)-log(eps))/log(2));

A(i,1)=a;
B(i,1)=b;
C(i,1)=c;

ax=feval(func,a);
bx=feval(func,b);
cx=feval(func,c);

Fab=ax*bx;
Fac=ax*cx;

%Check the zero theorem

if Fab > 0 #if D >0 there is no solution
disp('no solution')

elseif Fab == 0 #If D == 0 the solution is already found
#check to see which of the two functions a or b is a solution for
disp('either a or b is a solution to your function')

if ax == 0 #if a is the solution display the value of a
disp('a')
disp(a)

else
disp('b') #if b is the solution display the value of b
disp(b)

end

else #use bisection method to discover the roots

i=i+1;

while i<=n+1 & Fac ~= 0

if Fac > 0
```

```
a = c;
elseif Fac < 0
b = c;
else
a = c;
b = c;
end

A(i,1)=a;
B(i,1)=b;
C(i,1)=c;

c = (a + b)/2;
Fac=feval(func,a)*feval(func,c);

i=i+1;
end

disp('n')
disp(n)

A
B
C

disp('value at theoretical root')
feval(func,c)

end
```

2. Script for Newton's Method

```
function result = newtroot ()

# usage: newtroot (fname, x)
#
#   fname : a string naming a function f(x).
#   x     : initial guess
maxit = 100;

tol = 1e-4;
f      = 'myf';
fprime = 'mydfdx';

xplot = linspace(-2,2, 1000);
fplot = feval(f, xplot);
plot(xplot, fplot, 'k-');

xn = input('First guess: ')
fn = feval(f,xn);
dfndx = feval(fprime, xn);

while( (abs(fn)>tol) & (maxit>=0) )
  xold = xn;
  fold = fn;
  xn = xn - (fn/dfndx);
  printf("%f\n",xn);
  fn = feval(f, xn);
  dfndx = feval(fprime, xn);
  hold on;
  plot([xold,xold], [0, fold], 'r-*');
  plot([xold,xn],   [fold, 0], 'b-*');
  maxit = maxit-1;
end
endfunction
```

3. Script for Secant Method

```
function value=Secant(fun,aIn,bIn,eps,nmax)

% The secant method for finding the zero of fun.
%       fun     the function defined as an mfile.
% For example fun.m could be:
%       function value=fun(x)
%           value=x^2-2;
%       aIn     left end point
%       bIn     right endpoint
%       eps     stopping tolerance
%       nmax    maximum number of steps

    a = aIn;
    b = bIn;

    fa = feval( fun,a );
    fb = feval( fun,b );

    if abs(fa) > abs(fb)
       temp =a;  a =b;  b= temp;
       temp =fa; fa=fb; fb=temp;
    end

    if isreal(a) & isreal(b)
        fprintf('    n\t x_n\t\t f(x_n)\n');
        fprintf('    0  %15.12f %15.12f\n',a,fa);
        fprintf('    1  %15.12f %15.12f\n',b,fb);
    else
        fprintf('    n\t real x_n\t imag x_n\t f(x_n)\n');
        fprintf('    0  %15.12f %15.12f %15.12f\n',real(a),imag(a),abs(fa));
       fprintf('    0  %15.12f %15.12f %15.12f\n',real(b),imag(b),abs(fb));
    end

    for n=2:nmax

       if abs(fa) > abs(fb)
           temp =a;  a =b;  b= temp;
           temp =fa; fa=fb; fb=temp;
       end

       d = (b-a)/(fb-fa);
       b  = a;
       fb = fa;
       d  = d*fa;

       if isreal(b)
           fprintf('%5.0f  %15.12f %15.12f\n', ...
              n,b,fb);
       else
           fprintf('%5.0f  %15.12f %15.12f %15.12f\n', ...
              n,real(b),imag(b),abs(fb));
       end

       if abs(d) < eps
           value=b;
           disp('Converged');
           return;
       end

       a  = a-d;
       fa = feval( fun,a );

    end
    disp('Failed to converge');
```

4. Script for Brent's Method

```
function result = brent ( f, xa, xb )
%
%  BRENT carries out Brent's method for seeking a real root of a nonlinear function.
%  This is a "stripped down" version with little error checking.
%
%  When the iteration gets going:
%    XB will be the latest iterate;
%    XA will be the previous value of XB;
%    XC will be a point with sign ( F(XC) ) = - sign ( F(XB) )
%
FATOL = 0.00001;
XATOL = 0.00001;
XRTOL = 0.00001;
ITMAX = 10;

it = 0;

fxa = feval ( f, xa );
fxb = feval ( f, xb );

xc = xa;
fxc = fxa;
d = xb - xa;
e = d;

format long

while ( it <= ITMAX )

  it = it + 1;

  [ xa, xb, xc; fxa, fxb, fxc ]

  if ( abs ( fxc ) < abs ( fxb ) )
    xa = xb;
    xb = xc;
    xc = xa;
    fxa = fxb;
    fxb = fxc;
    fxc = fxa;
  end

  xtol = 2.0 * XRTOL * abs ( xb ) + 0.5 * XATOL;

  xm = 0.5 * ( xc - xb );

  if ( abs ( xm ) <= xtol )
    'Interval small enough for convergence.'
    result = xb;
    return
  end

  if ( abs ( fxb ) <= FATOL )
    'Function small enough for convergence.'
    result = xb;
    return
  end
%
%  See if a bisection is forced.
%
  if ( abs ( e ) < xtol | abs ( fxa ) <= abs ( fxb ) )

    d = xm;
    e = d;

  else
```

```
    s = fxb / fxa;
%
%  Linear interpolation.
%
    if ( xa == xc )

      p = 2.0 * xm * s;
      q = 1.0 - s;
%
%  Inverse quadratic interpolation.
%
    else

      q = fxa / fxc;
      r = fxb / fxc;
      p = s * ( 2.0 * xm * q * ( q - r ) - ( xb - xa ) * ( r - 1.0 ) );
      q = ( q - 1.0 ) * ( r - 1.0 ) * ( s - 1.0 );

    end

    if ( p > 0.0 )
      q = - q;
    else
      p = - p;
    end

    s = e;
    e = d;

    if ( 2.0 * p >= 3.0 * xm * q - abs ( xtol * q ) | p >= abs ( 0.5 * s * q ) )
      d = xm;
      e = d;
    else
      d = p / q;
    end

  end
%
%  Save old XB, FXB
%
  xa = xb;
  fxa = fxb;
%
%  Compute new XB, FXB,
%
  if ( abs ( d ) > xtol )
    xb = xb + d;
  elseif ( xm > 0.0 )
    xb = xb + xtol;
  elseif ( xm <= 0.0 )
    xb = xb - xtol;
  end

  fxb = feval ( f, xb );

  if ( sign ( fxb ) == sign ( fxc ) )
    xc = xa;
    fxc = fxa;
    d = xb - xa;
    e = d;
  end

end

'Maximum number of steps taken.'
result = xc;
```