Chapter 4  Roots of Nonlinear Equations

4.1 The Problem

“Find a root of the equation \( f(x)=0 \)” means “Compute a value of \( x \) for which the equation \( f(x)=0 \) is satisfied.” An equivalent way of stating the problem is “Find a zero of the function \( f(x) \)”, meaning “Find a value of \( x \) for which \( f(x) \) is zero”. In even simpler terms, it means “Solve the equation \( f(x)=0 \) numerically for \( x \)” or “Find where the function \( y=f(x) \) crosses the x axis.”

In numerical computation, “is satisfied” means “is almost exactly satisfied, with small error”, “is zero” means “is very small in magnitude”, etc.

A reasonably typical problem in root finding is

Find all real roots of the equation \( f(x) = \cos(x) - ax = 0 \) for \( a = 0.336 \).

[roots near 1.167637235, -2.74377022, and -2.853722039]

[graph of \( f(x) = \cos(x) - 0.336 \times \) vs. \( x \) for \( x \) on \([-4,+2]\)]

If we plot \( f(x) \) versus \( x \), we see that there is a positive root of the equation \( f(x)=0 \) near \( x = 1.168 \).

However, the function \( f(x) \) either touches the \( x \) axis near \( x=-2.80 \), or crosses it twice, or merely comes very close to it. In fact, \( f(x) \) crosses the \( x \) axis twice near \( x=-2.80 \), so there are two real roots near \( x = -2.80 \). The fact that these roots are “clustered” (very near each other) makes them hard to compute using many methods. Or at least it is hard to compute them both rapidly and accurately. If the constant 0.336 were a little larger, around \( a=0.3365082 \), these two clustered roots would become a double root. [graph for \( a=0.3365082 \)]

If the constant were a little larger still, say 0.337, there would be no negative real roots at all. [graph for \( a=0.337 \)] In such a case where \( f(x) \) comes very near the \( x \) axis but doesn’t touch it, there are often two complex roots with real parts approximately equal to the value of \( x \) where \( f(x) \) comes closest to the \( x \) axis. We will not deal with the case of complex roots for non-polynomial functions \( f(x) \).

Another way to look at this same problem is “Find where the two curves \( g_1(x) = \cos(x) \) and \( g_2(x) = ax \) cross each other.” From the statement of our problem above, \( f(x) \) is equal to \( g_1(x) - g_2(x) \), so when \( g_1(x) = g_2(x) \), the value of \( f(x) \) will obviously be zero. Note that any equation of the form \( g_1(x) = g_2(x) \) can be transformed into an equation of the standard form \( f(x) = 0 \). For example

\[
\ln(x) = x \cos(x)
\]

which is an equation of the form \( g_1(x)=g_2(x) \), is equivalent to

\[
f(x) = \ln(x) - x \cos(x) = 0
\]

In many problems, only the roots in some “physical region” are of interest. The physical region might be \( x>0 \), for example, in which case we could ignore the nonpositive roots. The user must determine what the physical region is. Using too large a physical region could result in answers that are nonsense.
What’s the problem here?

Faced with a problem of finding a real root of \( f(x)=0 \), today’s student will probably say, “That’s not hard! I’ll program the function \( f(x) \) into my graphing calculator, display the function to find a good starting point, zooming in if necessary. Then I’ll ask the calculator to find the root. Why are we studying a problem that has already been mechanized, anyway?”

One good reason for studying root finding is that in a fairly typical situation we need the real root(s) not of just one function but of several thousand functions that are the same in form but have parameter values that differ somewhat. For example, we may need the smallest real root of

\[ f(x) = \cos(x) - ax = 0 \]

for many different values of the parameter “\( a \)”. These values of “\( a \)” often aren’t even known in advance, but will be computed by a program in the process of solving a much larger practical problem.

Therefore we need to be able to

1) Find a reasonably good starting approximation, or two, to the root

2) Select a reasonably good algorithm, programmed in a robust way

3) Specify a reasonable convergence criterion

4) Run the program and assess the results

A mathematically sophisticated student might ask, “What’s the problem here? Just find the inverse function \( x=f^{-1}(y) \) and evaluate it for \( y=0 \). Problem solved!” In other words, “Just solve the equation \( f(x)=0 \) for \( x \).” This is certainly the thing to do if the inverse function can be found in a simple form. Many functions cannot be inverted analytically, however. For example the function \( f(x) = x - e^{-x} \) is monotonically increasing and therefore has an inverse, but the inverse function to this \( f(x) \) cannot be written in closed form in terms of elementary functions, it has been proved. Therefore, in many problems we must proceed “iteratively”, computing more and more accurate approximations to the root, never finding it exactly but eventually finding an approximate answer that is good enough for all practical purposes.
Extraneous roots

Certain algebraic operations can introduce new roots into an equation. These are called extraneous roots.

For example, consider the equation

\[ 0 = f(x) = x - 7 \]

Obviously this equation has one and only one root, \( x = 7 \). However, suppose we transpose to obtain

\[ x = 7 \]  \hspace{1cm} \text{Eq. 4.1-1} \]

and then multiply by \( x \) to get

\[ x^2 = 7x \]

Now the equation has two roots, \( x = 7 \) and \( x = 0 \). The root \( x = 0 \) is an extraneous root because it is not a root of the original equation.

Alternately, suppose we square both sides of Equation 4.1-1 above to get

\[ x^2 = 49 \]

Now the equation has two roots, \( x = 7 \) and \( x = -7 \). The root \( x = -7 \) is extraneous.

The user should check that each calculated "root" \( r \) is really a root of the original equation. Plug each calculated root \( r \) back into the original equation \( f(x) = 0 \) and check that \( f(r) \) is actually equal to zero.
Exercises

1. Which root finding problems below can be solved easily by direct mathematical computation? If it is possible, do it: find all real roots of the equation. If there are an infinite number of real roots, find the two roots of smallest magnitude. Check all computed roots by plugging them into the equation. In each case where you solve for a root or roots, check for extraneous roots and state whether or not you found any.

   Hint: Fiddle with the equation mathematically, trying to solve for $x$ in all places where it appears, in terms of numbers and elementary mathematical functions. If you can’t solve the equation analytically, say, “Apparently this equation cannot be solved analytically for $x$ in terms of elementary functions.” Do not solve graphically.

   Also, sketch each problem both in the form $f(x) = 0$ and also in the form $g_1(x) = g_2(x)$. (If a function has more than two terms in it, divide it into $g_1(x)$ and $g_2(x)$ arbitrarily.)

(a) $f(x) = 3x^2 + 4x - 5 = 0$

(b) $f(x) = \exp(-1/x^2) - 0.6 = 0$ (Recall from Prependix A: $\exp(z)$ means $e^z$.)

(c) $f(x) = \cos(x) - 0.3 = 0$

(d) $f(x) = \cos(x) - 0.3 \times x = 0$

(e) $f(x) = 3x - \sqrt{x + 2} = 0$

(f) $f(x) = x^2 + 8/x = 0$

(g) $f(x) = 2/x^2 - 1/x - 1 = 0$

(h) $f(x) = x^{-6} - x = 0$

(i) $f(x) = 2/x - 1 - x = 0$

(j) $f(x) = x^4 - x = 0$

(k) $f(x) = e^x - x = 0$

(l) $f(x) = e^{-x} - x = 0$
4.2 The Method of Bisection

The basic bisection method

If a function $f(x)$ is continuous for $x$ on the closed interval $[a, b]$ and if $f(x)$ has opposite signs at $x=a$ and $x=b$, then $f(x)$ must cross the x axis at least once in the interval $[a, b]$; that is, the equation $f(x)=0$ must have at least one root in $[a, b]$. If $f(x)$ is continuous and $f(a)$ and $f(b)$ have opposite signs, we say that we have bracketed at least one root in $[a, b]$.

Always bear in mind that the function could cross the x axis, or touch it, more than once in $[a, b]$. For example, consider $f(x) = x^3 - 3x - 2$ on the interval $[a, b] = [-2, 3]$. $f(a) = f(-2) = -4$ and $f(b) = f(3) = 16$. Since $f(x)$ is continuous (it is a polynomial, and all polynomials are continuous for all $x$) and $f(a)$ and $f(b)$ have opposite signs, we know that $f(x)=0$ must have at least one root in the interval $[-2, 3]$. In fact this equation has a simple root at $x=2$ and a double root at $x=-1$.

The method of bisection starts with an initial interval $[a_0, b_0]$ on which $f(x)$ is continuous and such that $f(a_0)$ and $f(b_0)$ have opposite signs. The idea is to find a new interval $[a_1, b_1]$, half as long as $[a_0, b_0]$, that is still known to contain at least one root of $f(x)=0$. To do this, we bisect the initial interval, computing

$$x_{\text{mid}} = (a_0 + b_0) / 2 \quad \text{and} \quad f(x_{\text{mid}}).$$

We examine $f(x_{\text{mid}})$. If $f(x_{\text{mid}})=0$, we have found a root of $f(x)=0$ at $x=x_{\text{mid}}$, and we are finished. (The problem was to find one root, not to find all roots, and we have found one root.)

If $f(x_{\text{mid}})$ has the same sign as $f(a_0)$, then there might be a root, or several roots in the interval $[a_0, x_{\text{mid}}]$, but we can't be sure of that because the function has the same sign at the two endpoints. But if $f(x_{\text{mid}})$ has the same sign as $f(a_0)$, $f(x_{\text{mid}})$ must have the opposite sign to $f(b_0)$, so we know that there must be at least one root in the half-interval $[x_{\text{mid}}, b_0]$. Therefore we discard the half-interval $[a_0, x_{\text{mid}}]$ and assign $x_{\text{mid}}$ as the left end $a_1$ of the new interval $[a_1, b_1]$, and proceed to the next iteration.

Likewise, if $f(x_{\text{mid}})$ has the sign opposite to that of $f(a_0)$, then we know that there must be at least one root in $[a_0, x_{\text{mid}}]$ and we assign $x_{\text{mid}}$ as the right end $b_1$ of the new interval $[a_1, b_1]$ and proceed to the next iteration. In any case, either we have found a root at $x_{\text{mid}}$ or we have cut in half the length of the interval that we know contains at least one root. Iterating on, we can reduce the length of this interval of uncertainty to as small a value as desired and, in doing so, compute the value of the root as accurately as desired.

If an interval cannot be found on which $f(x)$ changes sign, the method of bisection cannot be used. For this reason, roots of even multiplicity of polynomials, such as the root at $x=-1$ of $f(x) = x^3 - 3x - 2 = (x + 1)^2 (x - 2) = 0$ cannot be computed by bisection (note that $x=-1$ is a root of multiplicity two of this polynomial) except by getting lucky and choosing an initial interval such as $[a, b]=[-5, 3]$ such that we hit $x=-1$ by chance, in this case on the first iteration of bisection. [Draw this function and see what we mean.] The root at $x=2$, on the other hand, is of multiplicity one, and can be found using bisection.

We need some sort of convergence criterion. Ideally we would like to be able to specify some number $\text{abstol}$ such that, at the end of our computation of a root, we can be sure that...
Choosing $a_0$, $b_0$, and abstol

We need an initial interval $[a_0, b_0]$ such that $f(a_0)f(b_0) \leq 0$. If we just have one problem to solve, we can sketch $f(x)$ on some suitable interval, or plot a graph on a graphing calculator, and find reasonable values for $a_0$ and $b_0$ from the graph. However we may have hundreds or thousands of similar problems to solve, each with a slightly different function $f(x)$, as we change some of the parameters (constants) in $f(x)$. In that case we may have to do a little mathematical analysis. We might be able to prove that $f(0)$ is always negative and $f(x)$ is always positive at large positive values of $x$. Then we could choose $a_0=0$ and choose $b_0$ equal to some large but reasonable positive value of $x$. See also the section below on Starting Points, where we will discuss how to use Taylor series to get a starting point.

$abstol$ is chosen to be a very small positive number if we want to compute a root to high accuracy, and $abstol$ might be somewhat larger than that if we are willing to settle for a less accurate value of the root. Keep in mind that for a root $r$, $abstol < \text{macheps } |r|$ is an unreasonably small value for $abstol$, because it requests more relative accuracy than the machine can deliver.

How long will it take?

If we know that $f(a_0)$ and $f(b_0)$ have opposite signs (assuming always that $f(x)$ is continuous unless specified otherwise), then we could guess that a root (call the value of a root $r$: $f(r)=0$) is located at $x_{\text{mid}}=(a_0 + b_0) / 2$ and not make an error greater than $|b_0 - a_0| / 2$. After one iteration, we could guess that a root lies at the midpoint of the new interval and not make an error greater than half the length of the new interval, which is one-quarter of the length of the original interval, etc. Therefore, in terms of the initial interval $[a_0, b_0]$, after $n$ iterations

$$|\text{Error}| \leq |b_0 - a_0| / 2^{n+1}$$

There are methods that are usually much faster than bisection, but on the other hand bisection is never extremely slow. If we want to know how many iterations it takes to achieve a maximum error no greater than some $abstol>0$, we can just solve

$$|b_0 - a_0| / 2^{n+1} \leq abstol$$

for the value of $n$, taking the ceiling function of the result because we can’t do a fractional part of one iteration. Using logarithms,

$$2^{n+1} \geq |b_0 - a_0| / abstol$$

$$n+1 \geq \log_2(|b_0 - a_0| / abstol)$$

$$n = \text{ceil}(\log_2(|b_0 - a_0| / abstol)) - 1$$

We could get lucky and hit a root exactly in fewer than this many iterations, so we should write

$$n \leq \text{ceil}(\log_2(|b_0 - a_0| / abstol)) - 1$$

In applying the method of bisection to practical problems, only rarely will we hit a root exactly.
Always bear in mind that every computer program with only finite input will eventually either stop or go into an infinite loop, because the computer only has a finite number of states. This is quite different from a mathematical iteration on real numbers.

**A crude bisection algorithm**

We can test whether $f(a)$ and $f(b)$ have opposite signs by testing their product. A crude algorithm for the method of bisection might go as follows. Assume that we are given real values of $a$, $b$, (with $a<b$) and abstol, and will return any computed root as $r$. (We will refer only to $a$ and $b$, not to $a_0$, $b_0$, $a_1$, $b_1$, etc. because in programming the algorithm we will ordinarily not save all of these values, only the ends $a$ and $b$ of the current interval $[a, b]$.)

1) Evaluate $f(a)$ and $f(b)$. Call these $f_a$ and $f_b$, respectively.
   If $f_a=0$ then $x=a$ is a root; terminate successfully with $r=a$;
   else if $f_b=0$ then $x=b$ is a root; terminate successfully with $r=b$.
   If $f_af_b>0$ then we do not have the required sign change;
   terminate unsuccessfully with an error message.

2) Compute $x_{mid} = (a + b) / 2$.
   If $(b - a) / 2 \leq$ abstol, then terminate successfully
   and return $r=x_{mid}$ as the root.

3) Evaluate $f(x_{mid})$. Call this value $f_{mid}$.
   If $f_{mid}=0$, then terminate successfully and return $r=x_{mid}$ as the root.

4) If $f_af_{mid}>0$
   then replace $a := x_{mid}$
   and $f_a := f_{mid}$
   else replace $b := x_{mid}$
   and $f_b := f_{mid}$

In either case, go to Step 2 after completing Step 4.
Practical aspects of programming the bisection algorithm

The bisection algorithm has been programmed approximately this way thousands of times by thousands of programmers, and it usually works satisfactorily. However if we want to produce a portable and robust piece of mathematical software, we can’t just write a subprogram directly based on the crude algorithm above, but must make several improvements in it.

First, in a computer subprogram implementing this algorithm, we would want to return some flag parameter to indicate what caused the return, perhaps

\[ k\text{Flag} = +1 \text{ for } r=a \text{ or } r=b \text{ in Step 1 above,} \]
\[ k\text{Flag} = -1 \text{ for an unsuccessful return due to } f(a)f(b)>0 \text{ in Step 1,} \]
\[ k\text{Flag} = +2 \text{ for a successful return in Step 2,} \]
\[ k\text{Flag} = +3 \text{ for a successful return in Step 3, and} \]
\[ k\text{Flag} = +4 \text{ for a successful return when the root has been computed to full precision (see below).} \]

Second, multiplying \( f_a f_b \) or \( f_a f_{\text{mid}} \) can cause an unnecessary overflow or underflow. We should avoid this. In any computer language, we could write an If statement similar to

\[
\text{If } f_a > 0.0 \text{ And } f_b > 0.0 \text{ Or } f_a < 0.0 \text{ And } f_b < 0.0 \text{ Then}
\]
\[ \text{signChange = False} \]
\[ \text{Else} \]
\[ \text{signChange = True} \]

for use in Step 1 above and then use the value of the Boolean variable signChange to tell the program how to proceed, and do similarly in Step 4.

Third, we saw in a previous chapter that if a and b are close together, then in floating point arithmetic, \( (a + b) / 2 \) may actually lie outside of the interval \([a, b]\). Therefore instead of computing \( (a + b) / 2 \) for the value of \( x_{\text{mid}} \), we should use the best method of computing the midpoint, which was developed in the chapter on Floating Point:

\[
\text{If } a > 0.0 \text{ And } b > 0.0 \text{ Or } a < 0.0 \text{ And } b < 0.0 \text{ Then}
\]
\[ x_{\text{Mid}} = a + (b - a) / 2.0 \]
\[ \text{Else} \]
\[ x_{\text{Mid}} = (a + b) / 2.0 \]

Fourth, an unsophisticated user might specify a value of abstol so small that it would be much smaller than the gap length between adjacent floating point numbers near the root r, in which case, we could never converge to the requested accuracy. To handle that case, we should add a natural convergence criterion in Step 2 above:

2) Compute \( x_{\text{mid}} \) as shown above.
\[ \text{If } (b-a) / 2 \leq \text{abstol, then terminate successfully and return } r=x_{\text{mid}} \text{ as the root.} \]
\[ \text{If } x_{\text{mid}} \leq a \text{ or } x_{\text{mid}} \geq b \text{ then terminate successfully and return } r=x_{\text{mid}} \text{ as the root.} \]
Using the accurate formula above, $x_{\text{mid}}$ will lie strictly between $a$ and $b$ if there are any floating point numbers there at all; if $x_{\text{mid}}$ does not lie strictly between $a$ and $b$ then there are no floating point numbers between $a$ and $b$, so there is no point in proceeding to compute $f_{\text{mid}}$. We have converged to the maximum precision of which this machine is capable, and should declare victory and exit (with kFlag=$+4$, as suggested above).

Fifth and last, in case something goes wrong, we do not want to iterate forever, so we should add an iteration counter, “iter” say, to count the iterations. If the value of iter exceeds some input limit, maxit, we terminate unsuccessfully (with kFlag=$-2$, say).

Call the bisection algorithm with these five improvements “the improved bisection algorithm”.
Exercises

1. For each problem below, sketch \( y = f(x) \) and show the root, or the first few roots nearest the origin. Also, sketch each part in the form \( g_1(x) = g_2(x) \) and mark the roots (the values of \( x \) at which \( g_1(x) \) and \( g_2(x) \) cross).

   Find a reasonable initial interval \([a_0, b_0]\) for computing the smallest positive real root, if any, of the equation \( f(x) = 0 \), and carry out the first three iterations of the method of bisection.

   (a) \( f(x) = e^{-x} - x = 0 \)   (b) \( f(x) = \cos(x) - 0.3 \ x = 0 \)

   (c) \( f(x) = \tan(x) - 2x = 0 \)   (d) \( f(x) = \exp(-x^2) - 3x = 0 \)

   (e) \( f(x) = x \ln(x) - \cos(x) = 0 \)   (f) \( f(x) = \ln(x) - x \cos(x) = 0 \)

2. For each initial \([a_0, b_0]\) interval and abstol value below, how many iterations of bisection would be necessary, assuming we had a continuous function \( f(x) \) such that \( f(a_0)f(b_0)<0? \)

   (a) \([a_0, b_0] = [-2, 3], \quad \text{abstol} = 1E-7 \)

   (b) \([a_0, b_0] = [1E-8, 1E5], \quad \text{abstol} = 1E-3 \)

3. We start the method of bisection from an initial interval \([a_0, b_0] = [2.2, 2.7]\). We want our final approximation to the root \( r \) to have at least five correct decimal places in it (5D). What is the minimum number of iterations of bisection required to guarantee this? Give a number for your answer, not an expression.

   Hint: What should the value of abstol be, to get five correct decimal places in this problem?

4.(a) Carry out three iterations of the method of bisection for finding a root of the equation

   \( f(x) = 0.4 \ x^3 + x - 1 = 0 \)

   starting from the initial interval \([a_0, b_0] = [0, 1]\).

   (b) How many iterations of the method of bisection would be necessary to localize a root of this equation with an error no greater than 2E-5, starting from \([a_0, b_0] = [0, 1]\)? Give a number for your final answer!

5. Write out the steps of the full bisection algorithm, including kflag and the four improvements given above.
6. Use Taylor series to expand \( f(x) = \exp(-x^2) \) about \( x_0 = 0 \) through terms in \( x^2 \), and also through terms in \( x^3 \):

\[
\exp(-x^2) = f(x) = 1 - x^2 + R_3(x), R_3(x) = R_3(x, \xi_3(x)) = 1 - x^2 + 0 + R_4(x), R_4(x) = R_4(x, \xi_4(x))
\]

Work out the formulae for \( R_3(x) \) and \( R_4(x) \) in detail.

For \( x = 0.4 \), find upper bounds on the magnitudes of \( R_3(x) \) and \( R_4(x) \).

Compute the value of \( f(0.4) \) and from it the actual error.

Verify that the actual error satisfies the bounds you found.

Now use bisection and the known value of the error to compute both \( \xi_3 \) and \( \xi_4 \).

Are the values of \( \xi_3 \) and \( \xi_4 \) unique (is there only one value of each \( \xi \) in \([x_0, x]\))?

7. The method of bisection is applied to finding a root of an equation \( f(x) = 0 \).

The initial interval \([a_0, b_0]\) is \([2.25, 2.75]\) and \( f(a_0) \ast f(b_0) < 0 \).

(a) “In this case the initial midpoint \((a_0 + b_0)/2\) has at least two correct bits in it, as an approximation to the root \( r \); whatever the function \( f(x) \) may be.”

Justify this statement.

(b) Now twenty iterations of the method of bisection are carried out, starting from the initial interval given above. What is the minimum number of correct bits in the final midpoint \((a_{20} + b_{20})/2\), as an approximation to the root \( r \)?

8.(a) For the FPNS defined by \((b, s, m, M, c/r) = (2, 24, -128, +127, \text{rounds})\), find a function \( f(x) \) and initial interval \([a_0, b_0]\) such that the improved bisection algorithm will run as many iterations as possible, terminating on the natural convergence criterion. How many iterations will it run?

(b) Show the first four intervals \([a_n, b_n]\) for your problem in part (a) above.

(c) For arbitrary values of \( b, s, m, \) and \( M \), give an approximate formula for the maximum number of iterations that the improved bisection algorithm could possibly run, on the worst possible problem(s).

9.(a) Give the best algorithm for computing the midpoint \( x_{mid} \) of an interval \([a, b]\) using floating point arithmetic, given the values of \( a \) and \( b \).

(b) For \([a_0, b_0] = [-1.2, +3.4]\), how many iterations of the method of bisection will be required to achieve an absolute convergence tolerance of \( \text{abstol}=1.0 \times 10^{-5} \)?
Problems

1. Program the improved bisection algorithm as a subprogram in some high-level computer language. Test it thoroughly.

2. Use the bisection subprogram from Exercise 6 above to compute the value of machine epsilon, macheps. Hint: Use a function f(x) defined by

\[ f(X) = \begin{cases} 
-1 & \text{if } (1 + X) \text{ is less than or equal to 1, and} \\
+1 & \text{if } (1 + X) \text{ is greater than 1.}
\end{cases} \]

Compute the addition \((1 + X)\) by calling a procedure or subroutine (not a function subprogram!) to do the addition, so that the compiler will not hold the sum in a register using extra precision. Use abstol=0, to get ultimate precision.
Programming Problem: Bisection

Write a subprogram implementing the method of bisection in a robust form, with all of the improvements described in section 4.2. Test it as described below. The bisection routine must be self-contained, including complete documentation that describes what it does and how to use it. It must not contain any problem-dependent constants or functions. Submit only one computer run, in which the main program calls the bisection routine several times, for different functions. Every form of normal exit should be tested, as well as every form of abnormal exit.

Choose one or more test functions. Use the bisection routine to localize at least one positive root and one negative root. In addition, use the routine to solve the two problems below.

1) Use the routine to compute exactly the value of machine epsilon (macheps) by localizing to full machine precision (using abstol=0.0) the discontinuity in the function

\[ f(x) = +1 \text{ if } (1.0 + x) > 1.0, \]
\[ = -1 \text{ if not} \]

Use REAL (float) variables and arithmetic for this task, not type double. The addition in \((1.0 + x)\) must be computed in a separate subprogram such as the procedure AddTwoReals shown in the notes.

Call the resulting root \(x_1\). The desired value of \(x_1\) is the final value of \(b\), not the value of “a” or of xMid. Why is this?

When the value of \(x_1=\text{macheps}\) has been computed, compute the value of \(((1.0 + x_1) - 1.0)\) in the main program, and call it rightGap. (Do the addition and subtraction in a separate subprogram or subprograms.) rightGap is the length of the gap just to the right of 1.0 in this FPNS.

2) Now compute the smallest positive \(x\) for which \((1.0 - x)\) is less than 1.0, using a function similar to the \(f(x)\) shown above, but modified appropriately for the task. For this value of \(x\) (call it \(x_2\)), compute the value of \((1.0 - (1.0 - x_2))\), and call this value leftGap. leftGap is the length of the gap just to the left of 1.0 in this FPNS.

Is the desired value of \(x_2\) equal to the final value of “a” or to the final value of \(b\)? Why?

Print out the following quantities, each with at least eight significant decimal digits. \(x_1\), rightGap, \(x_2\), and leftGap, Compute and print also \(\log_2(\text{rightGap})\), \(\log_2(\text{leftGap})\), \((\text{rightGap} / \text{leftGap}) \, flBase\), \((x_1 - \text{rightGap} / 2.0) \, epsDiff\), and, if epsDiff is nonzero, \(\log_2(epsDiff)\), each with at least eight SDD. The values of \(\log_2(\text{rightGap})\), \(\log_2(\text{leftGap})\), \((\text{rightGap} / \text{leftGap}) \, flBase\), and \(\log_2(epsDiff)\) should all be integers.

For computing macheps, rightGap, and leftGap, it is essential that the bisection subprogram return the final values of both “a” and “b”. In Java this can be done by making “a” and “b” elements in an array of two elements, such as endPoint[0] and endPoint[1], and sending the array endPoint[] to the bisection subprogram as a parameter. You must decide which endpoint to use in each case, “a” or “b”.

13
In your discussion, explain why the following four statements are true.

1) \( \text{flBase} \) is the base of the FPNS.

2) If \( x_1 = \text{rightGap} \), the computer uses chopped floating point arithmetic.

3) If \( \text{epsDiff} \) is zero, the machine uses naive rounding.

4) If \( \text{epsDiff} \) is a positive number much smaller than \( \text{rightGap} \), the machine probably uses symmetric rounding (rounding to even).
   
   Hint: What is the last digit in the floating point representation of 1.0 in this FPNS? Is this digit even or odd?

Follow good programming practices throughout. See Prependix B.

Hand in paper hardcopy of your source code, test data and output.
4.3 Starting Points

If practical we can graph the function for our parameter values, or for typical parameter values, and figure out approximately where the root(s) will lie. A knowledge of the limiting behavior of the function in various regions may be very useful. This knowledge can often be gained from Taylor series. (The reader should be careful not to confuse the point $x_0$ about which a Taylor series is expanded with the starting point $x_0$ of a method for computing a root. They are usually not the same thing.)

For example, near $x=0$,

\[
\begin{align*}
\sin(x) & \approx x \\
(\text{the Taylor series about } x_0=0 \text{ is } & \sin(x) = x - x^3/6 + x^5/120 - ...) \\
\cos(x) & \approx (1 - x^2/2), \text{ or even just } 1 \\
\tan(x) & \approx x \\
\cot(x), \text{ which is exactly equal to } & 1/\tan(x), \text{ is therefore approximately equal to } 1/x \\
\tan^{-1}(x) & \approx x \\
\exp(x) & \approx (1 + x) \\
\text{etc.}
\end{align*}
\]

Near $x=1$,

\[
\ln(x) \approx (x-1)
\]

Near $x=\pi/2$,

\[
\begin{align*}
\sin(x) & \approx (1 - (x - \pi/2)^2/2), \text{ or even just } 1 \\
\cos(x) & \approx (\pi/2 - x) \\
\cot(x) & \approx (\pi/2 - x),
\end{align*}
\]

hence

\[
\tan(x) \approx 1/(\pi/2 - x) \quad \text{(that is, } 1/\text{ (a Taylor series)})
\]

For large positive values of $x$, $\tan^{-1}(x) = \pi/2 - 1/x$.

Etc., etc., etc.!

To approximate the smallest positive root of $f(x) = \cos(x) - ax = 0$ for a positive value of “a”, observe that there must be a root between $x=0$ and $x=\pi/2$. (Why “must” there be a root in this interval?) Let us approximate $\cos(x)$ near zero as above by

\[
\cos(x) \approx 1 - x^2/2
\]

and get the equation
0 = f(x) ≈ (1 - x^2/2) - ax

This gives a quadratic equation

\[ x^2 + 2ax - 2 = 0 \]

which has solutions

\[ x = (1/2)(-2a \pm \sqrt{4a^2 + 4(2)}) \]

For a=0.336, this gives roots at \( x = 1.11758 \) and \(-1.78958\). The positive approximate root is not too far from the true root, which is near 1.16764 (about 4.3% error).

In the case above, we have approximated \( \cos(x) \) by a Taylor polynomial and inserted that polynomial into the equation \( \cos(x) - a x = 0 \). Equivalently, we could have approximated the whole function \( f(x) = \cos(x) - a x \) by a Taylor polynomial and set that equal to zero. The two methods are equivalent and give exactly the same answer. Personally, I prefer the first method.

In most cases, we choose simple values of \( x_0 \) about which to form the Taylor polynomials, so that we will get simple values for the function and its derivatives at \( x = x_0 \). That means we try to choose \( x_0 = 0 \) for \( f(x) = e^x \), \( x_0 = 1 \) for \( f(x) = \ln(x) \), and \( x_0 = 0, \pm \pi/2, \pm \pi \), etc. for trig functions, provided these values of \( x_0 \) are judged to be close enough to the root to give Taylor polynomials that are a good approximation to \( f(x) \) near the root.

If a first-degree Taylor polynomial \( p_1(x) \) is used, then the Taylor series method of finding a starting point may, depending on how the Taylor polynomial \( p_1(x) \) is used, be equivalent to carrying out one iteration of Newton's method starting from the value of \( x_0 \) that was used as the base point for the Taylor series.

**Crude but widely applicable methods**

Some computer programs for finding roots just start with two estimates near each other, evaluate \( f(x) \) at each of those points, and then move in the direction in which \( |f(x)| \) seems to be decreasing. Whenever \( |f(x)| \) decreases, the step size can be increased, perhaps by doubling it. If \( f(x) \) changes sign, a root has been bracketed. If \( |f(x)| \) increases and the sign of \( f(x) \) has not changed, the program may go back and examine that interval with a smaller step size. A method like this is used in the procedure activated by the “SOLVE” button on Hewlett-Packard calculators. The HP algorithm changes the step size by a factor of fifty.

Another reasonable method is to define a “physical region” in which the solution is thought to lie, and then to generate many starting points in this region using a pseudorandom number generator, using each point as the starting point for computing a root.
Exercises

1. (a) Use a Taylor series to compute a good numerical starting point $x_0$ for solving the nonlinear equation

$$\sin(x) = 1 - x .$$

Compute a number for $x_0$!

(b) Sketch on the same graph $g_1(x) = \sin(x)$, $g_2(x) = 1 - x$, and the Taylor polynomial $p_1(x)$.

2. (a) Approximate $\cos(x)$ near $x = \pi/2$ instead of near $x = 0$ as done in the text above, and solve for an approximate value of $x$ for the smallest positive root of $f(x) = \cos(x) - ax = 0$, where “$a” is a positive constant.

For $a = 0.336$, which approximation is more accurate?

Would there be any reason to expect the two approximate solutions for the positive root to bracket the true root? Explain.

(b) Sketch on the same graph $g_1(x) = \cos(x)$, $g_2(x) = 0.336x$, and the Taylor polynomials $p_2(x)$ for $x_0 = 0$ and $p_1(x)$ for $x = \pi/2$.

3. Find a method of approximating the positive root(s) of the following equations. In each case sketch the “parts” of each function ($\tan(x)$, 0.3 $x$, etc.) and see where they intersect. Remember that zero is not a positive number!

(a) $\tan(x) = 0.3x$ (approximate the smallest positive root, and the second smallest)

(b) $\tan^{-1}(x) = 0.3x$

(c) $\cot(x) = 9x$ (smallest positive root)

(d) $x = 3e^{-2x}$

4. We wish to compute the positive root of the equation $\cos(x) = x^2$.

(a) Expand $\cos(x)$ in a Taylor series around $x_0 = 0$, through terms in $x^2$. Using only the first (constant) term from this series, compute an approximation to the positive root of the equation above.

(b) Using the terms in the Taylor series through the term in $x^2$, compute another approximation to the positive root.

(c) Use a graph to show why the two approximations computed in parts (a) and (b) “bracket” the root.
5. (a) We wish to find the root $r$ of

$$f(x) = \sin(x) + 0.1 \ x = 0$$

that lies in the interval $[p, 4]$. Draw a graph (preferably a “g-picture” of two functions crossing) that shows this root.

(b) Choose a value $x_0$ about which to approximate $\sin(x)$ near this root in a Taylor series in powers of $(x - x_0)^n$. Why did you choose this value of $x_0$?

(c) Give your Taylor polynomial approximation for $\sin(x)$, and from it get an approximate value for the root $r$, as a number.

6. We wish to calculate the root $r$ near $x=1.26$ of the equation

$$x = \left( e^x - 1 \right)/2 .$$

(1.2564312086 is a correctly rounded value of $r$.)

Use a Taylor series expanded about the point $x_0=0$ to find a good starting point for computing this root.

Problems

1. Program the “crude but widely applicable method” described above as a subprogram in a high-level programming language. Test it.
4.4 Newton’s Method

The method of bisection is never extremely slow, but neither is it very fast unless it happens to hit a root exactly, by chance. Another drawback is that it cannot compute roots of even multiplicity except by luck. We need a method that is usually much faster than bisection and which can compute a root of even multiplicity, although in the latter case the method might not be fast. (Fortunately, multiple roots are not very common.)

After Isaac Newton invented calculus in order to solve physics problems, he applied calculus to the problem of computing roots of equations. In doing so, he used a tactic that has since become widely used:

If faced with a problem that is nonlinear or otherwise too complicated to solve directly, approximate this problem by another problem that is linear or otherwise simpler than the original problem. Solve the simpler problem. Plug the answer from this solution back into the simpler problem and compute a second solution, and repeat this process. (In other words, “iterate”.) Test for convergence.

It will usually be necessary to supply an initial starting value for this process, a “first guess” at the value of the solution. This is often not as hard to do as to find the two starting points making up the initial bracketing interval \([a, b]\) required for bisection.

What Newton did was to select a starting value \(x_0\) and then to approximate the function \(f(x)\) by the first two terms from the Taylor series of \(f(x)\) expanded about the point \(x_0\).

\[
f(x) = 0 \approx f(x_0) + f'(x_0) (x - x_0)
\]

Then he solved this approximate equality for \(x\):

\[
0 = f(x_0) + f'(x_0) (x - x_0)
\]

\[
(x - x_0) = -f(x_0) / f'(x_0)
\]

\[
x = x_0 - f(x_0) / f'(x_0)
\]

Then Newton renamed this value of \(x\) to be \(x_1\):

\[
x_1 = x_0 - f(x_0) / f'(x_0)
\]

and finally he generalized the iteration from “Compute \(x_1\) from \(x_0\)” to “Compute \(x_{n+1}\) from \(x_n\)”:

\[
\begin{align*}
\text{***************} & \\
\text{* } x_{n+1} = x_n - f(x_n) / f'(x_n) & \text{*} \\
\text{* } & \text{*} \\
\text{***************}
\end{align*}
\]

This is Newton’s method for finding a root of one equation in one unknown, \(x\). (It is sometimes called the Newton-Raphson method.) It requires only one initial guess or iterate, \(x_0\). It does not always converge. It could even divide by zero, if the user or the computer program does not check to see if the denominator is zero. It could converge but converge much more slowly than bisection. In many cases, however, given a reasonably good first guess \(x_0\), Newton’s method converges very rapidly to a root, assuming the equation has a root.
Newton’s method is already in a form that is a good form to use for many numerical problems:

(Next approximation) = (Best known approximation) ± (Small correction)

This is a good form because if the “Small correction” is indeed, as we hope, much smaller in magnitude than the “Best known approximation”, then there cannot be a large amount of subtractive cancellation in the ± operation. We will see this idea in several different contexts as we go along.

**Classic Error 4.1:**
Not putting an equation into the form $f(x) = 0$ before trying to solve it using Newton’s method (or the method of bisection, for that matter).

**Exercises**

1. For each $f(x)$ below, choose an appropriate starting point $x_0$ for finding the smallest positive real root, if any, of the equation $f(x)=0$, and then carry out iterations of Newton’s method until the iterates $x_n$ do not change in the first nine significant digits. Underline all of the correct significant digits in each iterate $x_n$.

   Give the values of $f(x_n)$ as well as the values of $x_n$.

   In each case, sketch the function and draw the first two iterations of Newton’s method on your graph.

   (a) $f(x) = 2x^3 + 2x - 8$
   (b) $f(x) = e^x - x$
   (c) $f(x) = \cos(x) - 0.3 x$
   (d) $f(x) = \tan(x) - 2x$
   (e) $f(x) = \exp(-x^2) - 0.1 x$
   (f) $f(x) = x \ln(x) - \cos(x)$
   (g) $f(x) = \ln(x) - x \cos(x)$

2. (a) Write the general formula for Newton’s method of finding a root.

   (b) Using Newton’s method, compute to high accuracy a root of $f(x) = 2x^3 - 3x -1 = 0$ with $x_0=2$. Give all iterates, $x_i$, and give also the values of $f(x_i)$. Underline the correct significant digits in each iterate.

   (c) Sketch a graph of $y=f(x)$ and on it draw the first iteration, only, of Newton’s method.
3. What happens when Newton’s method is applied to the problems below starting from \(x_0=0.2\)? What happens starting from \(x_0=2.0\)? Is Newton’s method converging rapidly? Converging slowly? Diverging? Doing something else?

In each case, sketch the function and draw the first two iterations of Newton’s method on your graph, for one starting point. Carry out several iterations numerically, starting from both \(x_0=0.2\) and \(x_0=2.0\).

(a) \(f(x) = x^{1/3} = 0\) (the cube root of \(x\))
(b) \(f(x) = \tan^{-1}(x) = 0\)
(c) \(f(x) = x^5 = 0\)
(d) \(f(x) = x^2 + 0.05 = 0\)
(e) \(f(x) = x^3 - 5x^2 + 3x + 9.1 = 0\)

4. We wish to solve the equation \(\cos(x) = x\) using Newton’s method.

(a) Rewrite this equation in the form \(f(x) = 0\).

(b) Use Taylor series to find a reasonably good starting point for computing the real root of this equation.

(c) Using the starting point \(x_0 = 0.7\) (\(not\) the starting point you computed in part (b)!), carry out two Newton iterations on this problem, computing \(x_1\) and \(x_2\). Give \(f(x_0), x_1, f(x_1), x_2,\) and \(f(x_2)\).

(d) Draw a graph of your \(f(x)\), and draw the first Newton iteration on it.

(e) Using \(r \approx 0.739085133\) for the true value of the root \(r\), underline the correct significant digits in the values of \(x_0, x_1,\) and \(x_2\) from part (c).

(f) Starting from the initial interval \([a_0, b_0] = [0.7, 0.8]\), how many iterations of the method of bisection would be required to guarantee an answer to this problem accurate to six decimal places?

5.(a) Perform one iteration of Newton’s method for computing the positive root of the equation \(\cos(x) = x^2\), starting from \(x_0=0.8\).

Make it clear what function \(f(x)\) you are using.

(b) Draw a graph showing the Newton iteration you carried out in part (a).
6. When a loan is made for an amount \( A \), to be repaid in \( n \) payments of \( p \) each, the rate \( r \) of compound interest (per payment period) satisfies the equation

\[
A r (1 + r)^n = p ((1 + r)^n - 1)
\]

(a) For \( A = $4500 \), \( p = $150 \) per month, and \( n = 36 \) months, solve for \( r \) to full precision using Newton’s method, starting from \( r_0 = 0.01 \). Show all iterates to as many significant digits as possible, and underline the correct significant digits in each iterate.

(b) For the values of \( A, p, \) and \( n \) in part (a), sketch \( f(x) \) on one graph, and \( g_1(x) \) and \( g_2(x) \) on another graph.

(c)* Find a good formula for a starting point \( r_0 \), for general values of \( A, p, \) and \( n \).

7. Suppose we use Newton’s method to find a root of \( f(x) = \tan^{-1}(x) = 0 \).

What is the critical positive value, \( X_0 \), of \( x_0 \), such that for \( 0 < x_0 < X_0 \) Newton’s method applied to this problem converges, whereas for \( x_0 > X_0 \), Newton’s method for solving this problem diverges.

Hint: What iterates would we get if we started at \( x_0 = X_0 \)?

8. A light ray travels from the point \((0,a)\) on the y axis to the point \((d,-b)\) below the x axis. \((a, b, \) and \( d \) are all positive.) Above the x axis the index of refraction is \( n_1 \); below the x axis the index of refraction is \( n_2 \). \((n_1 \text{ and } n_2 \text{ are each greater than or equal to } 1.0.)\) If the ray makes an angle of \( \theta_1 \) with the y axis when it is above the x axis and an angle \( \theta_2 \) with the y axis when it is below the x axis, then according to Snell’s Law

\[
n_1 \sin(\theta_1) = n_2 \sin(\theta_2) .
\]

The problem is to find the point \((0,x)\) where the ray crosses the x axis.

Draw a graph of the problem. Now find an appropriate function \( f(x) \) such that when \( f(x) = 0 \), that value of \( x \) is the answer to the problem.

How can you choose a good starting point for \( x \), for general values of \( a, b, d, n_1, \) and \( n_2 \)? Solve this problem when \( a=3, d=4, n_1=1, \) and \( n_2=1.3 \) using Newton’s method. Show all iterates to as many significant digits as possible, and underline the correct significant digits in each iterate.

9. What is the largest number \( B \) such for \(|x_0| < B\), Newton's method applied to \( f(x) = \tan^{-1}(x) = 0 \) always converges to the root \( r=0 \)? \( \text{Hint: For } x_0=B, \ x_{n+1} = -x_n . \)

10. If we apply Newton’s method to \( f(x) = \tan^{-1}(x) = 0 \) starting at the initial point \( x_0=B \) from the Exercise just above, the method yields \( x_1=-B, x_2=B, \) etc. This “oscillation” is unstable: if we start at \( x_0=B+d \) for some small positive value of \( d \), Newton’s method will diverge from the \(+B, -B, +B, \ldots \) regime and go off to plus and minus infinity.

Investigate numerically what happens when Newton’s method is applied to

\[
f(x) = 24x^2 - x^4 + 176
\]
starting from $x_0=2$ and also starting from $x_0=2.01$. Does it oscillate? Is the oscillation stable or unstable? (A stable oscillation is one that tends to “come back” to its original form after it is perturbed.) How could you find the “interval of stability” for $x_0$?

11.(a) Carry out two Newton iterations for finding a root of the equation

$$f(x) = 0.4 \, x^3 + x - 1 = 0$$

starting from $x_0=0$.

(b) Draw a graph showing the two Newton iterations from part (a). (Show the tangent lines, etc.)
4.5 Order of Convergence

In most cases, Newton's method converges faster than the method of bisection. (There are exceptions; for example Newton's method is slower than bisection on a problem such as

\[ f(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1 = 0 \]

which has a root \( r=1 \) of multiplicity five, but most roots are "simple" (not multiple) roots.) The reader should have noticed that in most cases once Newton's method gets close to the root, the number of correct significant digits increases more and more rapidly. For example, consider the problem of finding the real root of the equation

\[ f(x) = e^{-x} - 0.2x = 0 \]

using Newton's method and starting from \( x_0=0 \). Using extended precision on an IBM mainframe \( (b=16, s=30, m=-64, M=63, \text{chopped}) \), which is about equivalent to 35 significant decimal digits, gives the results

<table>
<thead>
<tr>
<th>n</th>
<th>( x_n )</th>
<th>( f(x_n) )</th>
<th>( x_{n+1} - x_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.8333</td>
</tr>
<tr>
<td>1</td>
<td>0.833333333333333333333333333333337</td>
<td>0.2679315</td>
<td>0.4222</td>
</tr>
<tr>
<td>2</td>
<td>1.25553991735558098528638636642729</td>
<td>0.33813989E-1</td>
<td>0.6973E-1</td>
</tr>
<tr>
<td>3</td>
<td>1.325270700503657964097368508692294</td>
<td>0.67687574E-3</td>
<td>0.1453E-2</td>
</tr>
<tr>
<td>4</td>
<td>1.326724062440620481664643792073417</td>
<td>0.28051071E-6</td>
<td>0.6028E-6</td>
</tr>
<tr>
<td>5</td>
<td>1.326724665242096624864278799338311</td>
<td>0.48209164E-13</td>
<td>0.1036E-12</td>
</tr>
<tr>
<td>6</td>
<td>1.3267246652422200223635099294698124</td>
<td>0.14239350E-26</td>
<td>0.3060E-26</td>
</tr>
<tr>
<td>7</td>
<td>1.3267246652422200223635099297758079</td>
<td>0.77037198E-33</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>1.3267246652422200223635099297758079</td>
<td>0.77037198E-33</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Applying Newton's method to this problem on this computer, the value of \( f(x_n) \) never gets to zero. We know from looking at functions in floating point arithmetic that there may not be any value of \( x \) for which \( f(x) \) is exactly zero. Obviously \( x_7 \) and \( x_8 \) are as accurate an answer as we are going to get. Taking \( x_8 \) to be the root \( r \), the correct significant digits in \( x_n \) have been underlined. The numbers of correct significant digits in \( x_n \) are 0, 0, 1, 3, 6, 13, 27, 34, and 34. The sequence

3 --> 6 --> 13 --> 27

represents an approximate doubling at each iteration. This approximate doubling of the correct digits in \( x_n \), and the approximate doubling of the exponents in \( f(x_n) \) and \( (x_{n+1} - x_n) \) is characteristic of the behavior of the final iterates from Newton's method on problems having simple roots. It may take a while before this rapid final convergence occurs, of course, or it may never occur, if the multiplicity \( m \) is greater than unity (\( m>1 \)).

When Newton's method is converging rapidly, how can we measure how rapid the convergence is? If the root of \( f(x)=0 \) is \( x=r \) then \( r \) is the true value, and

\[ \text{Error in } x_n \equiv e_n \equiv r - x_n \]
If $e_n$ is converging to zero, how fast is it converging to zero? Very often $|e_n|$ can be expressed approximately as

$$|e_n| \approx C |e_{n-1}|^\alpha$$

where $C$ is some positive constant.

If so then we can define the order of convergence by

**Definition:** If

$$\lim_{n \to \infty} \frac{|e_n|}{|e_{n-1}|^\alpha} = C > 0$$

then $\alpha$ is the order of convergence.

If the above limit exists, then eventually the number of correct digits in $x_n$ is approximately multiplied by $\alpha$ at each iteration. If that is true, we would naturally like to have as large a value of $\alpha$ as possible.

For our Newton example above, we might guess $\alpha=2$ and from our iterates above compute

| n | $e_n$   | $|e_n| / |e_{n-1}|^2$ |
|---|--------|----------------------|
| 0 | 1.326725 |                       |
| 1 | 0.4933913 | 0.2803044            |
| 2 | 0.71184748E-1 | 0.2924179       |
| 3 | 0.14539648E-2 | 0.2869327       |
| 4 | 0.60280158E-6 | 0.2851456       |
| 5 | 0.10359878E-12 | 0.2851057      |
| 6 | 0.30599545E-26 | 0.2851056      |

It appears that the third column is converging to a nonzero constant $C$ around 0.2851, so we would guess that indeed $\alpha=2$ for Newton’s method in this problem. If we used a guess much smaller than $\alpha=2$, say $\alpha=1$, we would get values of $|e_n| / |e_{n-1}|^\alpha$ that would converge to zero, and if we used a guess larger than $\alpha=2$, say $\alpha=3$, the values of $|e_n| / |e_{n-1}|^\alpha$ would diverge to infinity, that is, they would not converge at all.
In most problems we can estimate the order of convergence $\alpha$ by any of several methods:

1) The value of $\alpha$ is the limit of the ratios of the number of correct digits in successive iterates $x_n$:

$$
6/3, \ 13/6, \ 27/13 \ \text{(converging to $\alpha=2$, perhaps)}:
$$
$$
\alpha = \lim_{n \to \infty} \frac{\text{number of correct digits in } x_n}{\text{number of correct digits in } x_{n-1}}
$$

(Note: We say “correct digits” rather than “correct significant digits” in this case because the $x_n$ could be converging to zero, and zero has no significant digits.)

2) The value of $\alpha$ is the limit of the ratios of the exponents in $f(x_n)$:

$$
(-6)/(-3), \ (-13)/(-6), \ (-26)/(-13) \ \text{(converging to $\alpha=2$, perhaps)}, \text{ or}
$$

3) The value of $\alpha$ is the limit of the ratios of the exponents in $(x_{n+1} - x_n)$:

$$
(-2)/(-1), \ (-6)/(-2), \ (-12)/(-6), \ (-26)/(-12) \ \text{(converging to $\alpha=2$, perhaps)}
$$

4) The value of $\alpha$ is the limit as $n$ goes to infinity of

$$
\log(|e_{n+2}/e_{n+1}|) / \log(|e_{n+1}/e_n|) \approx \alpha \quad \text{Eq. 4.5-1}
$$

The logarithms in this formula can be to any base, $e$ or 10 or...

c_{n+2}, c_{n+1}, \text{ and } c_n \text{ are the errors in } x_{n+2}, x_{n+1}, \text{ and } x_n, \text{ respectively. These can be computed from}

$$
c_n = r - x_n
$$

once we have computed the value of the root $r$ accurately by any method.

For our example above, $x_8$ is the most accurate iterate, so we take $r=x_8$ and compute approximate errors $e_n$ as follows.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$x_n$</th>
<th>$e_n = r - x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>1.326724665</td>
</tr>
<tr>
<td>1</td>
<td>0.83333333333333333333333333</td>
<td>0.493391332</td>
</tr>
<tr>
<td>2</td>
<td>1.2555399173555809528638636642729</td>
<td>0.711847478E-1</td>
</tr>
<tr>
<td>3</td>
<td>1.32527070503657964097368508692294</td>
<td>0.145396474E-2</td>
</tr>
<tr>
<td>4</td>
<td>1.3267240624406204816646643792073417</td>
<td>0.602801580E-6</td>
</tr>
<tr>
<td>5</td>
<td>1.326724665242096624864278799338311</td>
<td>0.103598771E-12</td>
</tr>
<tr>
<td>6</td>
<td>1.326724665242200022363509294698124</td>
<td>0.3059945E-26</td>
</tr>
<tr>
<td>7</td>
<td>1.326724665242200022363509297758079</td>
<td>“0.0” (Ran out of digits)</td>
</tr>
<tr>
<td>8</td>
<td>1.326724665242200022363509297758079</td>
<td></td>
</tr>
</tbody>
</table>

For $n=0$, and using $\ln( )$ for $\log( )$, our formula for $\alpha$ then gives

$$
\ln(0.711847478E-1 / 0.493391332) / \ln(0.493391332 / 1.326724665) = 1.95722889
$$
For $n=1$, we get
\[
\ln(0.145396474E-2 / 0.711847478E-1) / \ln(0.711847478E-1 / 0.493391332) = 2.00978105
\]
For $n = 2, 3, \text{ and } 4$ we get $\alpha = 2.00160568, 2.00001797, \text{ and } 2.00000001, \text{ respectively.}$

These approximate values for $\alpha$ appear to be converging to $\alpha = 2$, and in fact they are.

Method 4 above (Eq. 4.5-1) can also be applied by replacing errors $e_n$ by function values $f(x_n)$ or by step sizes $(x_n - x_{n-1})$.

In all of these four methods, we must not use a value of $n$ so large that we have gone beyond the number of digits that the computer or calculator carries.

Now we will state, but not prove, a theorem about the convergence of Newton’s method.

**Theorem:**

1) $r$ is a simple root of $f(x)=0$ (and therefore $f'(r) \neq 0$),
and if
2) $f(x)$ and $f'(x)$ are both continuous in a neighborhood of the root $r$,
and if
3) the initial iterate $x_0$ is sufficiently close to the root $r$,
then Newton’s method will converge to $r$ and it will do so with order of convergence $\alpha = 2$ or greater.

Convergence with $\alpha = 2$ is called **quadratic convergence**. For $\alpha = 2$, eventually the number of correct digits in $x_n$ will approximately double at each iteration. Quadratic convergence is fast, at least at the end, and therefore highly desirable. Its quadratic convergence to simple roots makes Newton’s method a very popular method.

Convergence with $\alpha = 1$ is called **linear convergence**. In linear convergence, the error decreases by approximately a constant factor at each iteration. Linear convergence can be “fast linear convergence”, but usually linear convergence means rather slow convergence, and sometimes extremely slow convergence.

Any order of convergence $\alpha$ that is greater than one is called **superlinear convergence**. Superlinear convergence is almost always better than linear convergence.

For a multiple root, $f'(r) = 0$ and Newton's method will converge only linearly ($\alpha = 1$) if it converges at all.

The theorem above often is not easy to apply. To use it we have to prove that a root exists, that it is a simple root rather than multiple, and then we have to find a starting point that “is sufficiently close to the root” without having any guidance as to just how close "sufficiently close" might be.

We should not think that having superlinear convergence to a simple root always solves all of our problems. Consider the following example from the great twentieth century numerical analyst James H. Wilkinson:

\[
f(x) = x^{20} - 1 = 0, \quad x_0 = 0.5
\]

This problem has two real roots, $r_1 = -1$ and $r_2 = +1$, and both of these roots are simple roots, so we might expect Newton’s method to be very fast. However,
\[
x_1 = 0.5 - (-0.9999990463 / 0.00003814697266) = 0.5 - (-26214.375) = 26214.875
\]
\[
x_2 = 24904.13125
\]
\[
x_3 = 23658.92469
\]
\[
x_4 = 22475.97845
\]

and so on. The initial convergence is very slow. The first iteration threw Newton’s method from a starting point \(x_0 = 0.5\) that is not far from the root \(r = 1.0\) to an \(x_1 = 26214.875\) that is far, far away, and now each iteration is moving \(x\) only about 5% of the distance back toward the root. The reason for this slow progress is that \(f(x)\) has only two real roots, but it also has another eighteen complex roots, all of magnitude equal to one, spaced equidistant on a circle with radius equal to unity, centered at the origin in the complex plane. Once Newton’s method gets out to a point very far from the origin, the twenty roots with magnitude equal to one “appear” much like a single root of multiplicity 20, and convergence will be approximately linear until the Newton iteration has worked its way back to a point reasonably near \(r = 1\), after which the roots will no longer “appear” to be multiple, and quadratic convergence will set in.

Even in the above problem, the final convergence was quadratic. What does it mean if a user has a problem for which \(f'(x)\) is obviously not zero at the root, so that the root is a simple root (multiplicity \(m = 1\)), and yet Newton’s method converges only linearly (\(\alpha = 1\))? Almost always it means that

**the formula for the derivative \(f'(x)\) is wrong!**

### Convergence of the method of bisection

The method of bisection does not converge smoothly; one midpoint may be very close to the root and the next midpoint not so close. As a result, the limit in the definition of order of convergence does not exist for the iterates computed by the method of bisection. In the long run, however, the method of bisection is roughly equivalent to an iteration converging linearly (\(\alpha = 1\)) with constant \(C\) equal to one-half.

### The \(\varepsilon^{1/m}\) rule

As shown above, in floating point a mathematical function \(y = f(x)\) is often not a “thin curve” because of roundoff error; it is a “fat curve” or a “fuzzy fat curve”. Because of this and because \(f'(x) = 0\) at a multiple root \(x = r\), multiple roots cannot be localized as accurately as simple (non-multiple) roots can. The rule that quantifies this was formulated by L. F. Shampine.

For definiteness, consider a cubic polynomial with a triple root (\(m = 3\)). Then we can write

\[
p_3(x) = C (x - r)^3 = C (x^3 - 3xr^2 + 3xr^2 - r^3)
\]

Since we do not know the value \(r\) of the root, we cannot evaluate the polynomial as \(C (x - r)^3\), but will have to use some other form such as the coefficient form.
\[ p_3(x) = c_3 x^3 + c_2 x^2 + c_1 x + c_0 \]

in which case it is obvious that \( c_3 = C, \ c_2 = -3Cr, \ c_1 = 3Cr^2, \) and \( c_0 = -Cr^3. \) For \( x \) very near \( r \) (\( x \approx r \)), the terms \( Cx^3, -3Cr^2, 3Cr^3, \) and \(-Cr^3\) will be almost exactly equal to \( Cr^3, -3Cr^3, +3Cr^3, \) and \(-Cr^3, \) so that the largest magnitude of any term will be almost exactly equal to \( |3Cr^3| \). The relative error in expressing a number in any FPNS is no larger than machinepsilon\(\equiv \epsilon \), so the error in this quantity could be as large as

\[ \epsilon \ |3Cr^3| \]

or perhaps a little larger, due to roundoff in the four multiplications required to evaluate the product \( 3Cr^3 \). Therefore our “fuzzy fat curve” \( y = f(x) \) might be as wide near \( x = r \) as this value, or a little wider. We cannot localize the root more accurately than the interval from the greatest value of \( x \) where the “fuzzy fat curve” is entirely below the \( x \) axis to the smallest value of \( x \) where this curve is entirely above the \( x \) axis. The latter \( x \) value is approximately given by

\[ C (x - r)^3 \approx \epsilon \ |3Cr^3| \]

\(|x - r|\) is the absolute error we will make in localizing the root, and \(|x - r| / |r|\) is the magnitude of the relative error. Canceling the factors of \( C \), we find

\[ |\text{Relative error}| = |x - r| / |r| = (3 \epsilon)^{1/3} \]

The factor of 3 in \((3 \epsilon)\) is not important. The important thing is the dependence on \( \epsilon^{1/3} \). For roots of any multiplicity \( m \), this will generalize to “**The \( \epsilon^{1/m} \) rule**”:

In floating point computations, a root of an equation \( f(x) = 0 \) cannot be computed with a relative error of magnitude much smaller than \( \epsilon^{1/m} \), where \( \epsilon \) is the value of “machine epsilon” for the FPNS being used and \( m \) is the multiplicity of the root:

\[ |\text{Relative error of a root}| \geq \epsilon^{1/m} \]

This applies to any method, whether it be bisection, Newton's method, or whatever. It says that a simple root might be computed with a relative error of magnitude around \( \epsilon \), but a double root probably will have relative error at least as large as \( \sqrt{\epsilon} \), which is much larger. A triple root will have a relative error at least as large as \( \epsilon^{1/3} \), which is even larger, and so on. For example, consider using double precision with \( s = 56 \) bits in the mantissa, in an FPNS that rounds. Then

\[ \epsilon = 2^{-56} \approx 1.39E-17 \]

so that a simple root might be computed with 15 or 16 correct SD, but

\[ \sqrt{\epsilon} \approx 3.75E-9 \]

so a double root will probably have at most 7 or 8 correct SD and

\[ \epsilon^{1/3} \approx 2.40E-6 \]

which means that a triple root will have at most 4 or 5 correct SD.
In computing a multiple root we are losing a fixed fraction \((1 - 1/m)\) of our significant digits. It is very unusual to lose a fixed fraction of the digits in a numerical computation. Usually we lose a fixed number of significant digits, not a fixed fraction of them.

A group of \(m\) roots that are closely clustered will often appear in an FPNS much like a multiple root, and often cannot be computed very accurately.

If the multiplicity \(m\) of a root is known, it might be possible to compute it very accurately even though it is a multiple root. Unfortunately the multiplicity \(m\) is rarely known.

The reason simple roots \((m=1)\) can be computed more accurately than multiple roots \((m\geq2)\) is that the “fuzzy fat curve of points” that represents a function in floating point (see the chapter on Floating Point) cuts across the \(x\) axis at a nonzero angle equal to \(\tan^{-1}(f'(r))\) at a simple root \(r\), but for a multiple root the tangent line has slope zero at the root \(r\), so the fuzzy fat curve runs along the \(x\) axis for a short distance. Look again at Figure 2 in the section on “Functions in Floating Point”, to see this effect.

**Exercises**

1. Derive Eq. 4.5-1. Hint: Start from

   \[ |e_{n+1}| \approx C |e_n|^\alpha \]

   and

   \[ |e_{n+2}| \approx C |e_{n+1}|^\alpha \]

   Eliminate \(C\) and solve for \(\alpha\).

2. (a) Copy the iterates below. Underline the correct significant digits in each iterate. Beside each iterate, write the number of correct significant digits. (The last iterate, 0.04792961, is correctly rounded to all digits shown.)

   0.05109
   0.04722
   0.047896
   0.04792031
   0.04792034
   0.04792961
   0.04792961

   (b) Estimate the value of \(\alpha\) from the ratios of the correct significant digits.

   (c) Estimate the value of \(\alpha\) using Eq. 4.5-1.

3. Below are eight iterates from each of two root finding iterations. Underline the correct digits in each iterate, and write beside each iterate the number of correct SD in that iterate.

   From these numbers, estimate the order of convergence \(\alpha\) in each case.

   (a) 1.108358892  (b) 2.621186280
   1.028849149  2.387801256
1.052702072  2.308668693
1.045546195  2.295247750
1.047692958  2.294407551
1.047048929  2.294395124
1.047242138  2.294395102
1.047184175  2.294395102
(limit = 1.047197551)

(c) Estimate the value of $\alpha$ in parts (a) and (b) above using Eq. 4.5-1.

4. Underline the correct digits in each set of iterates below.

(a) 1.3742135  
1.5109637  
1.5638042  
1.5641963
1.5641963

(b) 1.134962  
1.143826  
1.149370  
1.151235
1.153972
1.154329
1.154461

(c) Give a general definition of “order of convergence”.

(d) Approximately what is the order of convergence in each of parts (a) and (b) above?
5. Below are three convergent sequences of iterates.
   In each part, underline the correct digits in each iterate, and write the number of
   correct digits beside each iterate.
   Then classify each sequence by its order of convergence $\alpha$:
   $\alpha = 1$, linear convergence, or
   $1 < \alpha < 2$, superlinear but not quadratic convergence, or
   $\alpha = 2$, quadratic convergence.

   (a) 2.702900004  
        2.581134113  
        2.526142133  
        2.514147346  
        2.513285460  
        2.513274133  
        2.513274123  
        2.513274123  
   
   (b) 2.452257593  
        2.222460665  
        2.110795890  
        2.094664088  
        2.094395174  
        2.094395102  
        2.094395132  
   
   (c) 2.714974  
        2.464470  
        2.389319  
        2.366774  
        2.360010  
        2.357981  
        2.357190  
   
   (limit = 2.357111)

6.(a) Give the general formula for Newton’s method for computing a root.

(b) Carry out one iteration of Newton’s method for finding a root of the equation

   \[ f(x) = x^3 + x - 3 = 0 \]

   starting from $x_0=1.0$.

(c) Draw a graph showing this Newton iteration.

(d) If we started from $x_0=1.5$, the Newton iterates on this problem would be

   1.5
   1.25806451612903
   1.21470533327416
   1.21341278623619
   1.21341166276308
   1.21341166276223
   1.21341166276223

   Copy these iterates.

   Underline the correct digits in each iterate, and from those results conclude
   whatever you can about the order of convergence $\alpha$ in this case.
7. For each of the following convergent iterations, take the final iterate, which is the most accurate, to be the true value of the limit. Copy the numbers onto your paper. (You may photocopy the numbers, if you wish.) Underline the correct digits in each of the other iterates. Write the number of correct significant digits beside each iterate. From the latter numbers, estimate the value of $\alpha$ for that iteration. If $\alpha = 1$, estimate the value of the constant $C$. Each value of $\alpha$ will be either 1, approximately 1.618, or 2.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.796826155</td>
<td>1.392870019</td>
</tr>
<tr>
<td>1.484615457</td>
<td>1.070634145</td>
</tr>
<tr>
<td>1.372219606</td>
<td>1.048786552</td>
</tr>
<tr>
<td>1.331757099</td>
<td>1.047305285</td>
</tr>
<tr>
<td>1.317190597</td>
<td>1.047204856</td>
</tr>
<tr>
<td>1.311946656</td>
<td>1.047198046</td>
</tr>
<tr>
<td>1.310058837</td>
<td>1.047197585</td>
</tr>
<tr>
<td>1.309379222</td>
<td>1.047197553</td>
</tr>
<tr>
<td>1.309134561</td>
<td>1.047197551</td>
</tr>
<tr>
<td>1.309046483</td>
<td>1.047197551</td>
</tr>
<tr>
<td>(limit $\approx$ 1.308996939)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>4.954394927</td>
</tr>
<tr>
<td>2.111278129</td>
<td>5.307450746</td>
</tr>
<tr>
<td>2.293065371</td>
<td>5.803306326</td>
</tr>
<tr>
<td>2.536762841</td>
<td>5.656156679</td>
</tr>
<tr>
<td>2.759765397</td>
<td>5.676619645</td>
</tr>
<tr>
<td>2.859755163</td>
<td>5.677908301</td>
</tr>
<tr>
<td>2.872292594</td>
<td>5.677895317</td>
</tr>
<tr>
<td>2.872453825</td>
<td>5.677895325</td>
</tr>
<tr>
<td>2.872453851</td>
<td>5.677895325</td>
</tr>
<tr>
<td>2.872453851</td>
<td>5.677895325</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(e)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.253533863</td>
<td>5.339407340</td>
</tr>
<tr>
<td>4.004549197</td>
<td>5.455720163</td>
</tr>
<tr>
<td>3.833425339</td>
<td>5.536472500</td>
</tr>
<tr>
<td>3.787704773</td>
<td>5.522724925</td>
</tr>
<tr>
<td>3.785403484</td>
<td>5.523587526</td>
</tr>
<tr>
<td>3.785398163</td>
<td>5.523598785</td>
</tr>
<tr>
<td>3.785398163</td>
<td>5.523598776</td>
</tr>
<tr>
<td>3.785398163</td>
<td>5.523598776</td>
</tr>
<tr>
<td>3.785398163</td>
<td>5.523598776</td>
</tr>
</tbody>
</table>

8. When computing a triple root (multiplicity=3) of an equation using double precision floating point arithmetic with macheps=1E-15, what is the largest number of correct significant decimal digits that can be expected in the answer? Just give a number for your answer.
9. (a) In what cases does Newton’s method converge slower than quadratically? 
Give a concrete example (a function f(x) and a numerical starting point x₀) in which this occurs. 
What is the order of convergence of Newton’s method in your example? 

(b) Give a concrete example (a function f(x) and a numerical starting point x₀) in which the starting point x₀ for Newton’s method is not equal to a root r of the equation, but the next iterate x₁ is exactly equal to the root.

10. (a) Newton’s method is applied to solving a problem that has a root r of multiplicity m=4, and it converges to this root. What is the order of convergence α of Newton’s method likely to be in this case? 
This problem is solved on a machine with machine epsilon equal to 1E-16.

(b) If the root r is approximately equal to 7E-6, about how large an error should we expect in the computed value of r due to the finite precision of the machine? 
Give your reasoning briefly.

(c) If the root r is approximately equal to 4E+5, about how large an error should we expect in the computed value of r due to the finite precision of the machine? 
Give your reasoning briefly.

11. (a) Suppose we compute a double root r (i.e., multiplicity m=2) that is approximately equal to 6E-8, on a computer on which macheps=1E-16 in double precision. 
Approximately how large an error can we expect to have in the computed root, if we use Newton's method and iterate until no further reduction in |f(x)| occurs? 

(b) What would the order of convergence be in the iteration in part (a)? 

(c) What would the error be in part (a) be if the (double) root were equal to 6E15?

12. In the example in the text above, Newton’s method required seven iterations to get 34 SD accuracy. Starting from [a, b]=[0, 2], how many iterations would the method of bisection require?

13. What is the order of convergence α for Newton’s method operating on f(x) = tan⁻¹(x) = 0 with x₀=0.5? Hint: It is not α=2.

14. Find one or more formulas for computing an approximate value of the order of convergence α, given a sequence of iterates xₙ.
15. (a) Find an equation \( f(x) = 0 \) that has only one root, and that root is a double root, so that Newton’s method will converge only linearly, but for which there is a starting point \( x_0 \) such that if Newton’s method starts at \( x_0, x_1 \) will be exactly equal to the root.

(For this part of this exercise, you can work graphically without specifying the function analytically.)

(b)* Give a function for part (a) analytically and compute the starting point \( x_0 \) fairly accurately.

16. Starting from \( x_0 = 0.0 \), Newton’s method on a certain problem gives the following sequence of values of \( x_n \): 0.0, 0.2, 0.22, 0.222, 0.2222, etc. (That is, after \( n \) iterations, the value of \( x_n \) consists of “n two’s” after the decimal point.)

(a) What is the value of the root \( r \) toward which Newton’s method is converging in this problem? Give the value as an exact fraction.

(b) Using your answer from part (a) as the true value of the root, compute the errors \( e_n \) in the first four iterates, \( x_0, x_1, x_2, \) and \( x_3 \).

(c) What is the order of convergence of Newton’s method in this problem? Explain briefly.

(d) What is the multiplicity \( m \) of the root \( r \) in this problem?

(Hint: consider the ratio of successive errors \( e_n \).)

(e) Find a function \( f(x) \) for which Newton’s method will give this series of iterates \( x_n = 0.0, 0.2, 0.22, \) etc. .

17.* Each sequence of iterates \( x_n \) in Exercise 2.(a) to 1.(e) above was produced by using

\[ |r - x_{n+1}| = C |r - x_n|^\alpha \]

as an exact equation to generate \( x_{n+1} \) from \( x_n \). Knowing the value of \( r \), give formulas for computing accurate values for \( \alpha \) and \( C \). Apply these formulas to each sequence in Exercise 1 above, and get the values of \( r, \alpha, \) and \( C \).

(Hint: Use logarithms to solve the equation just above for \( \alpha \).)
18. (a) We wish to solve an equation \( f_1(x) = 0 \) for a root \( r \), and we know that this root \( r \)
is a simple root, that is, a root of multiplicity \( m=1 \), for which \( f_1'(r) \) is nonzero. The function and all of its derivatives are continuous.

We attempt to apply Newton’s method. Our iteration converges to the desired root \( r \), but it converges only linearly (\( p=1 \)).

What should we conclude?
What should we do next?

(b) We apply Newton’s method (correctly) to finding a multiple root (\( m>1 \)) of a certain problem \( f_2(x) = 0 \). We obtain iterates

\[
x_0 = 0.2, \quad x_1 = 0.22, \quad x_2 = 0.222, \quad x_3 = 0.2222, \text{ etc.}
\]

What is the limit of this sequence \( x_n \), as an exact fraction?
What is the order of convergence \( \alpha \) of this sequence? Explain briefly.

19. We wish to calculate the root \( r \) near \( x=1.26 \) of the equation \( x = (e^x - 1)/2 \).
(1.2564312086 is a correctly rounded value of \( r \).)

(a) Starting from \( x_0=1.4 \), carry out three iterations of Newton’s method.
Give both the Newton iterates \( x_n \) and the function values \( f(x_n) \).

(b) Draw a clear graph of the first Newton iteration.

(c) Underline the correct significant digits in every Newton iterate, \( x_0, x_1, x_2, \) and \( x_3 \).

(d) Draw whatever conclusions you can about the order of convergence of Newton’s method in this problem. Give your reasoning!
Programming Problem: Compound interest and Newton’s method

Suppose a person or a company borrows some amount of money $V$ (the Value of the loan), to be paid back in $n$ equal payments of $P$ each, at a compound rate of interest $r$. Given the value of $r$ and the values of any two of the other three variables, $V$, $n$ and $P$, we can compute the value of the fourth variable easily, but to compute $r$ given $V$, $n$, and $P$ we must solve a nonlinear equation. This equation can be written in various different forms:

\[ f_1(r) = \frac{r}{1 - (1 + r)^n} - \frac{P}{V} = 0 \]
\[ f_2(r) = \left(\frac{P}{V}\right) \left(1 - (1 + r)^n\right) - r = 0 \]
\[ f_3(r) = P \left(\frac{(1 + r)^n - 1}{r (1 + r)^n}\right) - V = 0 \]
\[ f_4(r) = \left(\frac{1}{r}\right) - \frac{1}{r (1 + r)^n} - \frac{V}{P} = 0 \]

Verify by algebraic manipulation that these four equations are all equivalent, in the sense that if a positive value $r$ is a root of any one of the equations, it is a root of all of the equations. We assume that $n \geq 2$, $V>0$, $P>0$, and $nP>V$ in all cases.

For this assignment we will use the values $V=$4500, $n=36$, and $P=$150, meaning that a loan of $4500 is to be repaid in 36 equal monthly payments of $150 each. The problem is to solve for $r$, the rate of compound monthly interest. For these values of $V$, $n$, and $P$, the value of $r$ that satisfies the four equations above is near $r=0.01$, or 1% interest per month. However, we will also want to consider properties of the above equations for other, more general values of $n$, $V$, and $P$, as long as these values satisfy the inequalities above.

Plot each of the four functions $f_i(r)$ above, for $r$ on the interval [0.0001, 0.02]. Make these graphs large and clear. Label the maximum and minimum values of $y=f(r)$ on each graph.

Does each of these four functions $f_i(r)$, $i=1,2,3,4$ appear to have a root near $r=0.01$?

Which of these functions $f_i(r)$ can be evaluated directly, at $r=0$?
Which functions require the use of l’Hospital’s rule to evaluate $f_i(0)$?

Derive general formulas for $f_i(0)$ and $f_i'(0)$, for all four functions. (0/0 is not an acceptable answer!)

Which of the functions $f_i(r)$ have an undesired extraneous root at $r=0$?

For which of the functions $f_i(r)$ could we use $r_0=0$ as a starting point for Newton’s method, and expect to find the positive root? For each such function, would the convergence be monotonic?

From the graphs, or from other evidence, what is the multiplicity $m$ of the positive root?

Which of the four functions $f_i(r)$ appears to be the best choice(s) for solving the problem of computing $r$? Judge the functions on
1) whether or not they have an undesired extraneous root at $r=0$,
2) how easy it is to find a usable starting value $r_0$, for any values of $V$, $P$, and $n$, and
3) how linear the functions are, in some approximate sense (a nearly linear function should make Newton’s method converge faster).

Choose two of the above four functions, one for which $f_i(0)=0$ and one for which $f_i(0)\neq0$.

Program Newton’s method for these two functions, in double precision. Write the program using
variables for \( P, V, \) and \( n \), setting the values of these variables at the top of the program, or reading them in. Use a reasonable starting point \( r_0 \) for each method, but not necessarily the same starting point for both methods. If \( r_0=0 \) is a valid starting point, use that. If not, and if \( r_0=1E-6 \) is a valid starting point, use that. Otherwise, use \( r_0=0.02 \). *(Why do we suggest these starting points? Explain, using your graphs.)*

In each case, iterate until \[ |r_{n+1} - r_n| \leq \text{reltol} \cdot \max(|r_n|, |r_{n+1}|) + \text{abstol} \]

and \[ |f(r_{n+1})|^3 \cdot |f(r_n)| \]

are both satisfied,

with \( \text{reltol}=1E-7 \) and \( \text{abstol}=1E-9 \). This is often a reasonable convergence criterion for using Newton’s method to compute a simple root, although the value of \( \text{abstol} \) depends on the particular problem being solved and must be chosen somewhat carefully based on knowledge of the approximate size of the root. This criterion will almost always cause Newton’s method to continue iterating until the root has been computed at least moderately accurately (the first part of the criterion), and until no more accuracy is possible on this machine (the second part of the criterion).

Limit each application of Newton’s method to a maximum of \( \text{maxit}=15 \) iterations. For each function, print \( r_n, f(r_n), \) and \( f'(r_n) \), and \( (r_{n+1} - r_n) \), at each iteration. Print at least 14 significant digits for each of these four quantities, and preferably at least 16 significant digits.

Taking the final value of \( r_n \) as the exact value of the root, underline the correct significant digits in each iterate \( r_n \), on your output. Draw any conclusions that you can about the order of convergence \( \alpha \) for each function. If the order of convergence is not equal to 2, your results are wrong and the problem is probably that your formula for \( f'(r) \) is wrong.

Hand in a good thorough discussion, covering all of the questions above as well as discussions of “What I Did”, “What Happened”, and “What I Learned”.

Follow good programming practices throughout. See Prependix B.

Hand in paper hardcopy of your source code, output, and discussion.

**Optional extra credit:**

Find another function that is equivalent to the above four functions, that might be a reasonable function to use when computing \( r \).

Find a formula for a value of a starting point \( r_0 \) that is always guaranteed to be greater than the positive root, for which convergence to the positive root is guaranteed.

(This is not guaranteed to be possible.)
4.6 Convergence Criteria

For a given method of computing a root of an equation \( f(x) = 0 \), how do we know when to stop iterating? In other words, we need to choose a convergence criterion.

For the method of bisection, we maintained an interval that we knew contained at least one root, and we used an absolute convergence criterion,

\[ |b_n - a_n| / 2 \leq \text{abstol} \]

because we knew that if this criterion is satisfied, then there must (ignoring roundoff) be a root within a distance \( \text{abstol} \) of the midpoint of the interval \([a_n, b_n]\). An absolute convergence criterion calls for a given number of decimal places (or digits in some base other than ten) to be correct, not a given number of significant digits.

Note that instead of an absolute criterion we could have used a relative convergence criterion, comparing the distance \( |b_n - a_n| / 2 \) to the size of typical values of \(|x|\) in this interval. One possibility would be to require

\[ (|b_n - a_n| / 2) / \max(|a_n|, |b_n|) \leq \text{reltol} \]

Note that using the natural convergence criterion, the denominator of the fraction on the left cannot be zero. Note also that we could cross-multiply and test

\[ |b_n - a_n| / 2 \leq \text{reltol} \times \max(|a_n|, |b_n|) \]

The value of \( \text{reltol} \) should be at least ten or a hundred times larger than machine epsilon, because it is unreasonable to ask for more accuracy than the precision of the machine, or even to ask for accuracy equal to machine precision, since roundoff may produce a relative error in the value of \( f(x) \) that is equal to a small multiple of machine epsilon, or worse.

A relative criterion calls for a given number of significant digits to be correct. This is usually what we want. If the root to which we are converging is at \( x=0 \), however, we can’t use a relative criterion because zero has no significant digits. In that case, an absolute criterion is better. Modern mathematical software often combines absolute and relative convergence criteria into one criterion of a mixed nature:

\[ |b_n - a_n| \leq \text{abstol} + \text{reltol} \times \max(|a_n|, |b_n|) \]

This is the professional way to do it! If a user wants a pure absolute criterion, the user can set \( \text{reltol}=0 \). If the user wants a pure relative criterion, set \( \text{abstol}=0 \). If there is a natural criterion available, as there is in bisection, and if the user wants high accuracy, then both \( \text{abstol} \) and \( \text{reltol} \) can be set equal to zero.

For Newton’s method the choice of a convergence criterion is not so simple. We have no bracketing interval \([a, b]\), only successive iterates \( x_n \). What should we test for smallness to determine whether or not we have converged? There is more than one choice, and many books advocate criteria that are very dangerous. We will consider two criteria here.
1) \[ |f(x_n)| \leq \text{fabsatol} \]

This seems rather reasonable. After all, we are trying to find where \( f(x) = 0 \), so why not test whether the magnitude of \( f(x_n) \) is small? Unfortunately there is a good answer to this question, in the form of another question:

“Small” compared to what?

For example, \( \text{fabsatol} = 1E-8 \) might be fine for the equation

\[
f_1(x) = e^x - x = 0
\]

(assuming we were using a precision of at least ten significant decimal digits), but this value of \( \text{fabsatol} \) would not do at all for the closely related equation

\[
f_2(x) = 1E20 \times (e^x - x) = 0
\]

because there is no reason to think that \( |f_2(x_n)| \) will get as small as \( 1E-8 \) in floating point arithmetic: small roundoff errors in computing \( e^x - x \) will cause \( |f_2(x_n)| \) to be much larger than \( 1E-8 \) even very close to the root. For this reason criterion #1 above should never be used. Any book that advocates using it alone as a convergence criterion is naive and untrustworthy, and should be burned before it can warp any impressionable young minds.

We could sample \( f(x) \) at a few points fairly near the root to get an idea of how large \( |f(x)| \) is likely to be, and compare \( |f(x_n)| \) to this estimate. That would be a reasonable idea, but we won’t pursue it here.

2) \[ |x_{n+1} - x_n| \leq \text{abstol} + \text{reltol} \times |x_{n+1}| \]

This is much better. We are testing the “step size” \( |x_{n+1} - x_n| \), and we have given a good answer to the question “Small compared to what?” “Compared to \( \text{abstol} \) and/or the \( \text{reltol} \) term”, that’s what. Rescaling \( f(x) \) has no effect on our criterion, and rescaling \( x \) is handled by the relative part of the criterion.

Because Newton’s method usually converges quadratically, we usually might as well choose to iterate to full machine precision, or as close to it as we can get. If we choose to do that, we can add to the criterion above

“and \[ |f(x_{n+1})| \approx |f(x_n)| \]”

This says, “As long as \( |f(x_n)| \) is decreasing, keep on iterating, even if \( |x_{n+1} - x_n| \) seems to be small enough.” (Of course, if \( |f(x_{n+1})| > |f(x_n)| \) then we will return \( x_n \), not \( x_{n+1} \), as the best known value of the root.) This combined criterion almost always works well. Finding a convergence criterion for Newton’s method that never works poorly, even for pathological examples, would be a major research project.
4.7  The Secant Method

Newton’s method is an often an excellent method to use if the derivative $f'(x)$ is available, but sometimes it is inconvenient or practically impossible to compute the derivative analytically. In that case, we may try to approximate the first derivative $f'(x)$ by the secant line through the last two points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$:

$$f'(x) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Substituting this approximation for $f'(x)$ into Newton’s method produces the secant method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_n) - f(x_{n-1}) / (x_n - x_{n-1})}$$

$$= x_n - f(x_n) \times \frac{(x_n - x_{n-1})}{(f(x_n) - f(x_{n-1}))}$$

We could collect both terms on the right over a common denominator, but the formula is already in the form

$$(\text{Next approximation}) = (\text{Best known approximation}) - (\text{Small correction})$$

which we know is the best form to use in floating point.

The order of convergence for the secant method can be shown to be equal to the golden ratio, $\phi$:

$$p = \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618034$$

given certain reasonable conditions on $f(x)$, including $f'(r) \neq 0$ (the root is not a multiple root).

The exponents of the values of the errors $(r - x_n)$ or of the function values $f(x_n)$ are often a set of negative Fibonacci numbers:

$$f(x_n) = \ldots \text{E}-1$$
$$= \ldots \text{E}-1$$
$$= \ldots \text{E}-2$$
$$= \ldots \text{E}-3$$
$$= \ldots \text{E}-5$$
$$= \ldots \text{E}-8$$
$$= \ldots \text{E}-13$$
$$= \text{etc.}$$

The ratios of successive Fibonacci numbers $2/1$, $3/2$, $5/3$, $8/5$, $13/8$, ..., approach $\phi$, the golden ratio.

The signs of successive values of $f(x_n)$ from the secant method will eventually almost always follow one of the four following patterns:

$$+ + + + + + + +$$
$$= - - - - - - - -$$
$$+ + - + + - + + -$$
$$= - + - - + - - +$$

Because the last two patterns have periods of three, the secant method has been called "a waltz method" by deBoor. (A waltz is a three-step dance.)
The secant method even does better on some functions than does Newton’s method.

When the secant method gets close to the root, however, the approximation above for \( f'(x) \) is using two points from the “cloud of points” that represent the function \( f(x) \) in floating point, and as can be seen from the graphs of floating point functions presented previously, the approximation for \((x_{n-1},f(x_{n-1}))\) and \((x_n,f(x_n))\) could easily have the wrong sign, and may in many cases be zero. As a result the secant method can behave erratically near the root and may even fail by producing a zero denominator \((f(x_n) - f(x_{n-1}))\) even on a problem in which \( f'(x) \) is not zero anywhere near the root. Hence the secant method can be characterized as “fast but flaky”.

**Exercises**

1. Apply the secant method to \( f(x) = e^{-x} - 0.2 \times x = 0 \), \( x_0 = 1 \), \( x_1 = 1.5 \).
   Carry out iterations on a calculator (or using double precision on a computer) until the method has converged to full machine precision, or very close to it.
   Show both \( x_n \) and \( f(x_n) \) for all of your iterations.
   Draw the first two iterations.
   Underline the correct digits in the iterates \( x_n \).
   List the numbers of significant digits in the \( x_n \) and from those values estimate the order of convergence \( \alpha \) of the secant method.
   Does your answer agree with the value given above for \( \alpha \)?
   Also, apply Eq. 4.5-1 to estimate the value of \( \alpha \) numerically.

2. Apply the secant method to \( f(x) = x^{1/3} = 0 \), \( x_0 = -1 \), \( x_1 = 2 \).
   Draw the first two iterations.
   Does the method converge?
   What would Newton’s method do on this problem?
   Is this problem well-behaved? Graph \( f(x) \) and \( f'(x) \).
   Is \( f(x) \) continuous near the root \( r \)? Is \( f'(x) \) continuous near \( r \)?
4.8 **The Method of Simple Iteration (Fixed Point Iteration)**

A lot of methods, such as Newton’s method, can be characterized as “one-point methods”, meaning that $x_{n+1}$ depends on the properties of the function at only one point, $x_n$. In general these one-point methods can be written as

$$x_{n+1} = g(x_n) \quad \text{Eq. 4.8-1}$$

For Newton’s method, the iteration function $g(x)$ is

$$g(x) = x - \frac{f(x)}{f'(x)}$$

because if that function $g(x)$ is substituted into Eq. 4.8-1 it produces Newton’s method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$  

Newton’s method usually converges quadratically, so this is a very good function $g(x)$ in many cases, but often a simpler function can be found. For example, consider the problem

$$f(x) = e^{-x} - x = 0$$

We can rewrite this problem as

$$x = e^{-x}$$

and from this we can derive a simple iteration function $g(x)$:

$$x_{n+1} = g_1(x_n) = \exp(-x_n)$$

Starting from $x_0=0.5$, we get successive iterates $x_n = 0.6065, 0.5452, 0.5697, 0.5601,...$, and the iteration is clearly converging to the root $r=0.56714329$. The convergence is slow, but it was very easy to find this $g(x)$ function and program it into a calculator. (Later we will see how to speed up the convergence of this process.)

Now we could also have taken the natural log of both sides of the equation above to get

$$\ln(x) = -x$$

and from this derive a different $g(x)$ function,

$$x_{n+1} = g_2(x_n) = -\ln(x_n)$$

Starting from $x_0=0.5$, we get the iterates $x_n = 0.6931, 0.3665, 1.003, -0.003715$, after which the iteration fails because it tries to take the natural log of a negative number, which gives a complex answer. The iteration diverged, and it would have diverged no matter how close we started to the root. For example, starting at $x_0=0.5671$, we get the iterates $x_n = 0.5672, 0.5670, 0.5674, 0.5667, 0.5679, ...$, and eventually a negative number -0.4003 and then the iteration fails. Some functions $g(x)$ converge and some don’t. Most of them, except for special cases such as Newton’s method, converge only linearly if they converge at all. Nevertheless these simple iteration functions $g(x)$ are easy to find, very often we can find one that converges, and, once programmed into a calculator we need only push the “COMPUTE” button once to perform each iteration. For example, on a Sharp EL-5103 calculator the entire program for the $g_1(x)$ iteration function above is
EXP(-A) STO A

The program for Newton’s method is

\[ A - \frac{\text{EXP}(-A) - A}{-\text{EXP}(-A) - 1} \text{ STO A} \]

In return for a more complicated program we get quadratic convergence.

There are an infinite number of iteration functions \( g(x) \) that can be gotten from any single nonlinear equation. Most of the simplest ones are gotten by “solving the equation \( f(x) = 0 \) for \( x \)” in some way or other, often by transposing one term to the other side and then transforming that term into \( x \) by applying some legal algebraic operation to each side of the equation.

If we graph an iteration function \( y = g(x) \), we should bear in mind that we are not trying to find where \( g(x) = 0 \), that is, where the curve \( y = g(x) \) crosses the \( x \)-axis, we are trying to find where \( x = g(x) \), that is, where the curve \( y = g(x) \) crosses the 45-degree line \( y = x \). Do not confuse the \( g \) function with the \( f \) function, or the \( g \) picture (the \( g \) graph) with the \( f \) picture!

\[ f(x) = 0 \quad \text{but} \quad g(x) = x \]

(See graphs in lecture.)

Graphing various examples of \( g(x) \) iteration functions, we can find that we get convergence to a root if we start close enough to the root and if

\[ |g'(x)| \leq C < 1 \]

everywhere in a neighborhood of the root. As might be suspected from this inequality, the smaller \( |g'(x)| \) is near the root, the faster is the convergence. For Newton’s method applied to a simple root \( r \) of any “nice” function, \( g(r) = 0 \). This is one way to explain why Newton’s method usually has superlinear convergence.

**Classic Error 4.2:**
Confusing a “\( g \)-picture” with an “\( f \)-picture” in drawing a graph.
In an \( f \)-picture, used with bisection, Newton’s method, the secant method, etc., we look for a point where the curve \( y = f(x) \) crosses the \( x \)-axis.
In a \( g \)-picture, used with the method of simple iteration, we look for the point where the iteration function \( y = x_{n+1} = g(x_n) \) crosses the 45-degree line \( y = x \). \( f(x) \) and \( g(x) \) are not the same functions. We are searching for a value of \( x \) where \( f(x) = 0 \) but \( g(x) = x \).

**Classic Error 4.3:**
Moving “horizontally to the curve, vertically to the line \( y = x \)” when graphing the method of simple iteration.
Correct is “vertically to the curve, horizontally to the line \( y = x \).”
Exercises

1. (a) An engineer applies the method of simple iteration by iterating

\[ x_{n+1} = g(x_n) = 1 - 0.4 x_n^3 \]

What nonlinear equation is the engineer trying to solve?

(b) Solve the equation to high accuracy using Newton's method. Draw the first Newton iteration.

(c) Underline the correct significant digits in each Newton iterate. What appears to be the order of convergence of Newton's method on this problem?

(d) Carry out ten iterations of the simple iteration in part (a), starting from \( x_0 = 0 \). Using the accurate answer you obtained in part (b) as the true value of the root, underline the correct significant digits in these iterates. What appears to be the order of convergence of this method on this problem?

2. Consider the equation \( f(x) = 2x^3 - 3x - 1 = 0 \).

Find two different functions \( g_1(x) \) and \( g_2(x) \) for the method of simple iteration, neither of which is Newton’s method. Sketch a graph showing each of the two \( g(x) \) functions. Draw two iterations of the method of simple iteration on each graph, starting from \( x_0 = 2 \). Is the iteration converging?

3. (a) Starting from \( x_0 = 0 \), carry out two iterations of the method of simple iteration for finding a root of the equation

\[ f(x) = 0.4 x^3 + x - 1 = 0 \]

using the iteration function

\[ x = g(x) = 1 - 0.4 x^3 \]

(b) Draw a graph of the “g picture” for part (a), showing both iterations clearly.

4. (a) Starting from \( x_0 = 1.4 \), carry out three iterations of the method of simple iteration

\[ x_{n+1} = g(x_n) = (\exp(x_n) - 1) / 2 \]

for this problem. (Reminder: the notation \( \exp(z) \) means the same thing as \( e^z \).)

(b) Is this iteration converging, or diverging? Give your reasoning.

(c) Draw a clear graph showing the first two iterations in part (a).
5. Find at least two different iteration functions \( g(x) \) for each of the following rootfinding problems. (Do not use Newton’s method.)

- Make a graph of each \( g(x) \), and estimate which ones would converge.
- Perform a few iterations with each \( g(x) \) function and state whether or not it seems to be converging.

(Hint: See the material on inverse functions in an early section of these notes.)

(a) \( f(x) = e^x - 0.2 \ x = 0 \)

(b) \( f(x) = 2x - \cos(x) = 0 \)

(c) \( f(x) = x - 2 \ \exp(-x^5) \)

(d) \( f(x) = 3 \ \cos(x) - x = 0 \) (Smallest positive root)

(e) \( f(x) = \tan(x) - 2x = 0 \) (for the root nearest to \( x=1.2 \))

(f) \( f(x) = \tan(x) - 2x = 0 \) (for the root nearest to \( x=4.6 \))

(For part (f), \( g(x)=\tan^{-1}(2x) \) computed by a calculator will not be exactly what we want for \( g(x) \). How can we modify \( \tan^{-1}(2x) \) to get the \( g(x) \) we need? Draw a graph of \( \tan(x) \) to get an idea.)

6.(a) For the problem \( f(x) = x^3 + x - 3 = 0 \)

find an iteration function \( g(x) \) for the method of simple iteration, not Newton’s method. Carry out two iterations of the method of simple iteration, starting from \( x_0=1.3 \).

(b) Does this iteration appear to be converging? Explain briefly.

(c) Draw a graph (a “g-picture”) that shows these two iterations clearly.

7. Consider the iteration \( x_{n+1} = g(x_n) = 1 + 0.1 \ x_n^3 \)

(a) What nonlinear equation is being solved by this iteration?

(b) Sketch this iteration function (draw a large graph!), along with the line \( y = x \), as done in lecture for similar problems.

(c) Draw on the graph, and also describe in words, what will happen if the iteration above is carried out

1. for \( x_0 = 1.1 \),
2. for \( x_0 = 2.2 \),
3. for \( x_0 = 2.4 \),
4. for \( x_0 = 2.5 \).
8. There are four sets of $f(x)$ values given below. Each set resulted from applying one root finding method we have studied to some problem. For each set, give
   1) the order of convergence, if any, and
   2) the root finding method that most likely gave this set of $f(x)$.
   Explain briefly your reasoning for #2 in each case.

   (a)  
   \begin{align*}
   3.7E-1 & \\
   1.4E-1 & \\
   1.9E-1 & \\
   3.5E-4 & \\
   1.2E-7 & \\
   1.5E-14 & \\
   2.3E-28 & \\
   3.7E-6 & \\
   7.4E-7 & \\
   1.5E-8 & \\
   \end{align*}

   (b)  
   \begin{align*}
   2.9E-1 & \\
   5.8E-2 & \\
   1.2E-2 & \\
   2.3E-3 & \\
   4.6E-4 & \\
   9.3E-5 & \\
   1.9E-5 & \\
   3.7E-6 & \\
   7.4E-7 & \\
   1.5E-8 & \\
   \end{align*}

   (c)  
   \begin{align*}
   2.1E-1 & \\
   8.0E-2 & \\
   1.7E-1 & \\
   1.3E-3 & \\
   2.3E-5 & \\
   3.0E-8 & \\
   6.9E-13 & \\
   2.1E-20 & \\
   1.5E-32 & \\
   \end{align*}

   (d)  
   \begin{align*}
   1.3E-1 & \\
   8.1E-1 & \\
   3.7E-1 & \\
   9.5E-2 & \\
   4.2E-1 & \\
   2.6E-2 & \\
   5.9E-3 & \\
   1.6E-3 & \\
   3.0E-3 & \\
   8.6E-4 & \\
   2.7E-3 & \\
   3.6E-4 & \\
   7.4E-5 & \\
   \end{align*}

9. Prove that when Newton’s method converges to a simple root of a “nice”, well-behaved function $f(x)$, $g'(r)=0$. Hint: What is $g(x)$ for Newton’s method applied to $f(x) = 0$?

10. In the analysis of algorithms, it is of interest to find the large positive value of $x$ for which

$$x^a = \ln(x)$$

where “$a$” is some given small positive constant power.

For $a = 0.001$, solve the above equation for $x$.

Hints: The method of simple iteration will converge nicely on this problem, but the values of $x$ are too large for a computer or calculator to store, without special measures being taken. Therefore change the variable from $x$ to $u = \ln(x)$ and solve for $u$. Then use this value of $u$ to write $x$ in floating point form such as $1.28544e2811$ (this is not the right value).
4.9 Is There a Best Method?

Is there a “best method” for finding a root of a nonlinear equation?

No!

(That was a short section, wasn’t it?)

In many cases where a rootfinding problem is part of a larger problem and where the derivative \( f'(x) \) is easily available analytically along with a reasonably good initial iterate \( x_0 \), Newton’s method with some reasonable convergence criterion can be coded into the program and will solve the problem. It would be nice, however, to have a general mathematical software routine to solve a wide variety of rootfinding problems.

Some of the better general purpose routines have been based on a method by T. Dekker, a Dutch computer scientist. Dekker’s idea was to combine the secant method with the method of bisection, in order to get the speed of the secant method together with the stability and guaranteed convergence rate of bisection.

**Dekker's Method:**

Keep an interval \([a_n, b_n]\) that brackets at least one root, as in bisection. Try to apply the secant method to the last two points computed, \((x_{n-1}, f(x_{n-1}))\) and \((x_n, f(x_n))\). If the new iterate \(x_{n+1}\) lies strictly between \(a_n\) and \(b_n\), accept \(x_{n+1}\) and use the bisection sign criterion to replace either \(a_n\) or \(b_n\) by \(x_{n+1}\). If \(x_{n+1}\) does not lie strictly between \(a_n\) and \(b_n\), reject \(x_{n+1}\) and apply the method of bisection, once, to the bracketing interval \([a_n, b_n]\). Then revert back to the secant method.

Dekker’s method usually works very well if an interval \([a_0, b_0]\) can be found on which the function \(f(x)\) changes sign. Dekker’s method usually doesn't do much bisection and is therefore often about as fast as the secant method. An exception occurs if a very tight convergence criterion is used, in which case the secant method will be used until the root has been localized except for "noisy" roundoff errors, after which bisection will be used to grind the answer down into the roundoff noise.

In some cases, however, the secant method can produce points \(x_n\) that consistently lie close to one end of the interval \([a_n, b_n]\), causing very slow convergence. To solve this problem, L. Shampine added an extra proviso to Dekker’s method.

**Shampine's Method:**

If Dekker’s method does not reduce the length of the bracketing interval \([a_n, b_n]\) by at least a factor of eight in any three successive secant iterations, apply three successive iterations of the method of bisection, then revert to Dekker’s method.

Since bisection will reduce the length of the bracketing interval by exactly a factor of eight in three iterations, Shampine’s method is never more than twice as slow as bisection, and it is usually almost as fast as the secant method. Shampine produced subroutine ZEROIN in his book *Numerical Computing: An Introduction*, and there is later software available in connection with his book *Fundamentals of Numerical Computing* by Shampine, Allen, and Pruess (Wiley, 1997). A similar method is used in the SOLVE function that is built into some Hewlett-Packard calculators.
Dekker’s and Shampine’s methods both require an interval \([a, b]\) on which \(f(x)\) changes sign, and we know that this may not be available in some problems, or it might exist but be very hard to find. In this case the programmer may choose to try to minimize \(|f(x)|\) or \((f(x))^2\) using a general method for finding a minimum of a function. If this process produces a point at which \(f(x)\) is zero, that solves the problem of finding a root. If it merely finds a point at which \(|f(x)|\) is small, we are again faced with the problem of answering the question “Small compared to what?” In many cases, fortunately, searching for a minimum of \(|f(x)|\) will find an interval containing a sign change.

Chapter 4: Summary

Starting from an initial interval \([a_0, b_0]\), the number of iterations of bisection required to produce an iterate guaranteed to be no farther than abstol from a root is

\[ n \leq \text{ceil}(\log_2(|b_0 - a_0| / \text{abstol})) - 1 \]

A crude bisection algorithm: [See above]

Improvements in the crude bisection algorithm:

1) Return a condition code, kflag, that tells what caused the return.
2) Avoid performing multiplications that might overflow unnecessarily.
3) Compute the midpoint accurately in floating point:
   - if \(a\) and \(b\) have opposite signs, use \(x_{\text{mid}} = (a+b)/2\);
   - otherwise use \(x_{\text{mid}} = (a+ (b-a)/2)\).
4) Add a natural convergence criterion that adapts automatically to any computer.
5) Implement a limit on the maximum number of iterations that can be performed.

Newton’s method:

\[ x_{n+1} = x_n - f(x_n) / f'(x_n) \]

A good form to use for many numerical problems is

\[(\text{Next approximation}) = (\text{Best known approximation}) \pm (\text{Small correction})\]

**Definition:** If \( \lim \frac{|e_n| / |e_{n-1}|^\alpha}{n \to \infty} = C \neq 0 \)

then \(\alpha\) is the order of convergence.

A practical method for approximating \(\alpha\) is

\[ \alpha = \lim_{n \to \infty} \frac{\text{(number of correct digits in } x_n)}{\text{(number of correct digits in } x_{n+1})} \]

**Theorem:**

If

1) \(r\) is a simple root of \(f(x)=0\) (and therefore \(f'(r) \neq 0\)),

and if

2) \(f(x)\) and \(f'(x)\) are both continuous in a neighborhood of the root \(r\),

and if

3) the initial iterate \(x_0\) is sufficiently close to the root \(r\),

then Newton’s method will converge to \(r\), and it will do so with order of convergence \(\alpha=2\) or greater.

“The \(e^{1/m}\) rule”:
In floating point computations, a root of an equation \( f(x) = 0 \) cannot be computed with a relative error of magnitude much smaller than \( \varepsilon^{1/m} \), where \( \varepsilon \) is the value of “machine epsilon” for the FPNS being used and \( m \) is the multiplicity of the root:

\[
|\text{Relative error of a root}| \geq \varepsilon^{1/m}
\]

The convergence criterion used most often in high quality mathematical software is

\[
|x_{n+1} - x_n| \leq \text{abstol} + \text{reltol} \times |x_{n+1}|
\]

Secant method:

\[
x_{n+1} = x_n - \frac{f(x_n) \times ((x_n - x_{n-1}) / (f(x_n) - f(x_{n-1})))}{f(x_n)}
\]

**Classic Error 4.1:** Not putting an equation into the form \( f(x) = 0 \) before trying to solve it using Newton’s method.

**Classic Error 4.1:** Confusing a “g-picture” with an “f-picture” in drawing a graph.
In an f-picture, used with bisection, Newton’s method, the secant method, etc., we look for a point where the curve \( y = f(x) \) crosses the x-axis.
In a g-picture, used with the method of simple iteration, we look for the point where the iteration function \( y = x_{n+1} = g(x_n) \) crosses the 45-degree line \( y = x \). \( f(x) \) and \( g(x) \) are not the same functions. We are searching for a value of \( x \) where \( f(x) = 0 \) but \( g(x) = x \).

**Classic Error 4.3:** Moving “horizontally to the curve, vertically to the line \( y = x \)” when graphing the method of simple iteration.
Correct is “vertically to the curve, horizontally to the line \( y = x \).”

In general there is no “best method” for computing a root. Dekker’s method is very good for simple roots and Shampine’s method is even better (see above for the details of these two algorithms). Newton’s method is widely applicable and often very fast. When Newton’s method is slow, it can often be accelerated using the methods given in a later chapter.

Hi,

Exam 2 will be held on Wednesday the 12th October in class. The study guide for this exam is as follows:

1. It covers Ch 4 and Ch 5 topics.
2. The handout on roots can be a source for likely exam questions.
3. There will be some T/F, define terms type questions.

Have a good one.