Chapter 3

Nonlinear Equations in One Variable

This chapter is concerned with finding solutions to the scalar, nonlinear equation

$$f(x) = 0,$$

where the variable x runs in an interval $[a,b]$. The topic provides us with an opportunity to discuss various issues and concepts that arise in more general circumstances.

There are many canned routines that do the job of finding a solution to a nonlinear scalar equation; the one in MATLAB is called `fzero`; type `help fzero` to see how this function is used and what input parameters it requires.

Following an extended introduction to our topic in Section 3.1 are three sections, 3.2–3.4, each devoted to a different method or a class of methods for solving our nonlinear equation. A closely related problem is that of minimizing a function in one variable, and this is discussed in Section 3.5.

3.1 Solving nonlinear equations

Referring to our prototype problem introduce above, $f(x) = 0$, let us further assume that the function f is *continuous* on the interval, denoted $f \in C[a,b]$. Throughout our discussion we denote a solution of the equation (called *root*, or *zero*) by x^*.

Note: Why introduce nonlinear equations before introducing their easier comrades, the linear ones?! Because one linear equation in one unknown is just too easy and not particularly illuminating, and systems of equations bring a wave of complications with them. We have a keen interest in solving scalar nonlinear equations not only because such problems arise frequently in many applications, but also because it is an opportunity to discuss various issues and concepts that arise in more general circumstances and highlight ideas that are extensible well beyond just one equation in one unknown.

Example 3.1. Here are a few simple functions and their roots.

1. $f(x) = x - 1$, on the interval $[a,b] = [0,2]$.

   Obviously there is one root for this linear equation: $x^* = 1$.

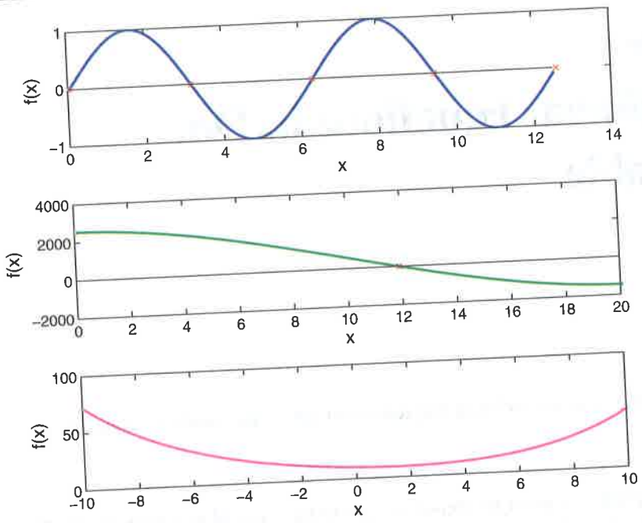

Figure 3.1. *Graphs of three functions and their real roots (if there are any):* (i) $f(x) = \sin(x)$ *on* $[0,4\pi]$, (ii) $f(x) = x^3 - 30x^2 + 2552$ *on* $[0,20]$, *and* (iii) $f(x) = 10\cosh(x/4) - x$ *on* $[-10,10]$.

2. $f(x) = \sin(x)$.

   Since $\sin(n\pi) = 0$ for any integer n, we have

   (a) On the interval $[a,b] = [\frac{\pi}{2}, \frac{3\pi}{2}]$ there is one root, $x^* = \pi$.

   (b) On the interval $[a,b] = [0,4\pi]$ there are five roots; see Figure 3.1.

3. $f(x) = x^3 - 30x^2 + 2552, \quad 0 \le x \le 20.$

   In general, a cubic equation with complex coefficients has three complex roots. But if the polynomial coefficients are real and x is restricted to be real and lie in a specific interval, then there is no general a priori rule as to how many (real) roots to expect. A rough plot of the function on the interval $[0,20]$ is given in Figure 3.1. Based on the plot we suspect there is precisely one root x^* in this interval.

4. $f(x) = 10\cosh(x/4) - x, \quad -\infty < x < \infty,$

   where the function cosh is defined by $\cosh(t) = \frac{e^t + e^{-t}}{2}$.

   Not every equation has a solution. This one has no real roots.

Figure 3.1 indicates that the function $\phi(x) = 10\cosh(x/4) - x$ has a minimum. To find the minimum we differentiate and equate to 0. Let $f(x) = \phi'(x) = 2.5\sinh(x/4) - 1$ (where, analogous to cosh, we define $\sinh(t) = \frac{e^t - e^{-t}}{2}$). This function should have a zero in the interval $[0,4]$. For more on finding a function's minimum, see Section 3.5. ∎

Example 3.2. Here is the MATLAB script that generates Figure 3.1.

```
t = 0:.1:4*pi;
tt = sin(t);
ax = zeros(1,length(t));
xrt = 0:pi:4*pi;
yrt = zeros(1,5);
subplot(3,1,1)
plot(t,tt,'b',t,ax,'k',xrt,yrt,'rx');
xlabel('x')
ylabel('f(x)')

t = 0:.1:20;
tt = t.^3 - 30*t.^2 + 2552;
ax = zeros(1,length(t));
subplot(3,1,2)
plot(t,tt,'b',t,ax,'k',11.8615,0,'rx');
xlabel('x')
ylabel('f(x)')

t = -10:.1:10;
tt = 10 * cosh(t ./4) - t;
ax = zeros(1,length(t));
subplot(3,1,3)
plot(t,tt,'b',t,ax,'k');
xlabel('x')
ylabel('f(x)')
```

This script should not be too difficult to read like text. Note the use of array arguments, for instance, in defining the array `tt` in terms of the array `t`. ∎

Iterative methods for finding roots

It is not realistic to expect, except in special cases, to find a solution of a nonlinear equation by using a closed form formula or a procedure that has a finite, small number of steps. Indeed, it is enough to consider finding roots of polynomials to realize how rare closed formulas are: they practically exist only for very low order polynomials. Thus, one has to resort to *iterative* methods: starting with an initial iterate x_0, the method generates a sequence of iterates $x_1, x_2, \ldots, x_k, \ldots$ that (if all works well) converge to a root of the given, continuous function. In general, our methods will require rough knowledge of the root's location. To find more than one root, we can fire up the same method starting from different initial iterates x_0.

To find approximate locations of roots we can *roughly plot* the function, as done in Example 3.1. (Yes, it sounds a little naive, but sometimes complicated things can be made simple by one good picture.) Alternatively, we can *probe* the function, i.e., evaluate it at several points, looking for locations where $f(x)$ changes sign.

If $f(a) \cdot f(b) < 0$, i.e., f changes sign on the interval $[a,b]$, then by the Intermediate Value Theorem given on page 10 there is a number $c = x^*$ in this interval for which $f(c) = s = 0$. To see this intuitively, imagine trying to graph a scalar function from a positive to a negative value, without lifting the pen (because the function is continuous): somewhere the curve has to cross the x-axis!

Such simple procedures for roughly locating roots are unfortunately not easy to generalize to several equations in several unknowns.

Stopping an iterative procedure

Typically, an iterative procedure starts with an initial iterate x_0 and yields a sequence of iterates $x_1, x_2, \ldots, x_k, \ldots$. Note that in general we *do not* expect the iterative procedure to produce the exact solution x^* exactly. We would conclude that the series of iterates converges if the values of $|f(x_k)|$ and/or of $|x_k - x_{k-1}|$ decrease towards 0 sufficiently fast as the iteration counter k increases.

Correspondingly, one or more of the following general criteria are used to terminate such an iterative process successfully after n iterations:

$$|x_n - x_{n-1}| < \texttt{atol} \quad \text{and/or}$$
$$|x_n - x_{n-1}| < \texttt{rtol}|x_n| \quad \text{and/or}$$
$$|f(x_n)| < \texttt{ftol},$$

where `atol`, `rtol`, and `ftol` are user-specified constants.

Usually, but not always, the second, relative criterion is more robust than the first, absolute one. A favorite combination uses one tolerance value `tol`, which reads

$$|x_n - x_{n-1}| < \texttt{tol}(1 + |x_n|).$$

The third criterion is independent of the first two; it takes the function value into account. The function $f(x)$ may in general be very flat, or very steep, or neither, near its root.

Desired properties of root finding algorithms

When assessing the qualities of a given root finding algorithm, a key property is its efficiency. In determining an algorithm's efficiency it is convenient to concentrate on the number of function evaluations, i.e., the evaluations of $f(x)$ at the iterates $\{x_k\}$, required to achieve convergence to a given accuracy. Other details of the algorithm, which may be considered as overhead, are then generally neglected.[5] Now, if the function $f(x)$ is as simple to evaluate as those considered in Example 3.1, then it is hard to understand why we concentrate on this aspect alone. But in these circumstances any of the algorithms considered in this chapter is very fast indeed. What we really keep in the back of our minds is a possibility that the evaluation of $f(x)$ is rather costly. For instance, think of simulating a space shuttle returning to earth, with x being a control parameter that affects the distance $f(x)$ of the shuttle's landing spot from the location of the reception committee awaiting this event. The calculation of $f(x)$ for each value of x involves a precise simulation of the flight trajectory for the given x, and it may then be very costly. An algorithm that does not require too many such function evaluations is then sought.

Desirable qualities of a root finding algorithm are the following:

- Efficient—requires a small number of function evaluations.
- Robust—fails rarely, if ever. Announces failure if it does fail.
- Requires a minimal amount of additional data such as the function's derivative.
- Requires f to satisfy only minimal smoothness properties.
- Generalizes easily and naturally to many equations in many unknowns.

No algorithm we are aware of satisfies all of these criteria. Moreover, which of these is most important to honor depends on the application. So we study several possibilities in the following sections.

[5] An exception is the number of evaluations of the derivative $f'(x)$ required, for instance, by Newton's method in Section 3.3.

3.2 Bisection method

The method developed in this section is simple and safe and requires minimal assumptions on the function $f(x)$. However, it is also slow and hard to generalize to higher dimensions.

Suppose that for a given $f(x)$ we know an interval $[a,b]$ where f changes sign, i.e., $f(a) \cdot f(b) < 0$. The Intermediate Value Theorem given on page 10 then assures us that there is a root x^* such that $a \le x^* \le b$. Now evaluate $f(p)$, where $p = \frac{a+b}{2}$ is the midpoint, and check the sign of $f(a) \cdot f(p)$. If it is negative, then the root is in $[a,p]$ (by the same Intermediate Value Theorem), so we can set $b \leftarrow p$ and repeat; else $f(a) \cdot f(p)$ is positive, so the root must be in $[p,b]$, hence we can set $a \leftarrow p$ and repeat. (Of course, if $f(a) \cdot f(p) = 0$ exactly, then p is the root and we are done.)

In each such iteration the interval $[a,b]$ where x^* is trapped shrinks by a factor of 2; at the kth step, the point x_k is the midpoint p of the kth subinterval trapping the root. Thus, after a total of n iterations, $|x^* - x_n| \le \frac{b-a}{2} \cdot 2^{-n}$. Therefore, the algorithm is guaranteed to converge. Moreover, if required to satisfy

$$|x^* - x_n| \le \texttt{atol}$$

for a given absolute error tolerance $\texttt{atol} > 0$, we may determine the number of iterations n needed by demanding $\frac{b-a}{2} \cdot 2^{-n} \le \texttt{atol}$. Multiplying both sides by $2^n/\texttt{atol}$ and taking log, we see that it takes

$$n = \left\lceil \log_2 \left(\frac{b-a}{2\,\texttt{atol}} \right) \right\rceil$$

iterations to obtain a value $p = x_n$ that satisfies

$$|x^* - p| \le \texttt{atol}.$$

The following MATLAB function does the job:

```
function [p,n] = bisect(func,a,b,fa,fb,atol)
%
% function [p,n] = bisect(func,a,b,fa,fb,atol)
%
% Assuming fa = func(a), fb = func(b), and fa*fb < 0,
% there is a value root in (a,b) such that func(root) = 0.
% This function returns in p a value such that
%       | p - root | < atol
% and in n the number of iterations required.

% check input
if (a >= b) | (fa*fb >= 0) | (atol <= 0)
  disp('something wrong with the input: quitting');
  p = NaN; n=NaN;
  return
end

n = ceil ( log2 (b-a) - log2 (2*atol));
for k=1:n
  p = (a+b)/2;
  fp = feval(func,p);
  if fa * fp < 0
    b = p;
    fb = fp;
  else
    a = p;
```

```
    fa = fp;
  end
end
p = (a+b)/2;
```

Example 3.3. Using our MATLAB function `bisect` for two of the instances appearing in Example 3.1, we find

- for func$(x) = x^3 - 30x^2 + 2552$, starting from the interval $[0, 20]$ with tolerance 1.e-8 gives $x^* \approx 11.86150151$ in 30 iterations;
- for func$(x) = 2.5\sinh(x/4) - 1$, starting from the interval $[-10, 10]$ with tolerance 1.e-10 gives $x^* \approx 1.5601412791$ in 37 iterations.

Here is the MATLAB script for the second function instance:

```
format long g
[x,n] = bisect('fex3',-10,10,fex3(-10),fex3(10),1.e-10)
function f = fex3(x)
f = 2.5 * sinh (x/4) - 1;
```

Please make sure that you can produce the corresponding script and results for the first instance of this example. ■

The stopping criterion in the above implementation of the bisection method is *absolute*, rather than *relative*, and it relates to the values of x rather than to those of f.

Note that in the function `bisect`, if an evaluated `p` happens to be an exact root, then the code can fail. Such an event would be rather rare in practice, unless we purposely, not to say maliciously, aim for it (e.g., by starting from $a = -1$, $b = 1$, for $f(x) = \sin(x)$). Adding the line

```
if abs(fp) < eps, n = k; return, end
```

just after evaluating `fp` would handle this exception.

We note here that the situation where a root is known to be bracketed so decisively as with the bisection method is not common.

Recursive implementation

The bisection method is a favorite example in elementary computer programming courses, because in addition to its conceptual simplicity it admits a natural presentation in *recursive* form. In MATLAB this can look as follows:

```
function [p] = bisect_recursive (func,a,b,fa,fb,atol)
p = (a+b)/2;
if b-a < atol
  return
else
  fp = feval(func,p);
  if fa * fp < 0
    b = p;
    fb = fp;
  else
    a = p;
```



```
    fa = fp;
  end
  p = bisect_recursive (func,a,b,fa,fb,atol);
end
```

Here then is an incredibly short, yet complete, method implementation. However, what makes recursion unappealing for effective implementation in general is the fact that it is wasteful in terms of storage and potentially suboptimal in terms of CPU time. A precise characterization of the reasons for that is beyond the scope of our discussion, belonging more naturally in an introductory book on programming techniques.

Specific exercises for this section: Exercises 1–2.

3.3 Fixed point iteration

The methods discussed in the present section and in the next two, unlike the previous one, have direct extensions to more complicated problems, e.g., to systems of nonlinear equations and to more complex functional equations.

Our problem

$$f(x) = 0$$

can be written as

$$x = g(x).$$

We will discuss how this can be done in a moment. Given the latter formulation, we are looking for a *fixed point*, i.e., a point x^* satisfying

$$x^* = g(x^*).$$

In the *fixed point iteration* process we define a sequence of iterates $x_1, x_2, \ldots, x_k, \ldots$ by

$$x_{k+1} = g(x_k), \quad k = 0, 1, \ldots,$$

starting from an initial iterate x_0. If such a sequence converges, then the limit must be a fixed point.

The convergence properties of the fixed point iteration depend on the choice of the function g. Before we see how, it is important to understand that for a given problem $f(x) = 0$, we can define many functions g (not all of them "good," in a sense that will become clear soon). For instance, we can consider any of the following, and more:

- $g(x) = x - f(x)$,
- $g(x) = x + 2f(x)$,
- $g(x) = x - f(x)/f'(x)$ (assuming f' exists and $f'(x) \neq 0$).

Thus, we are really considering not one method but a *family of methods*. The *algorithm* of fixed point iteration for finding the roots of $f(x)$ that appears on the following page includes the selection of an appropriate $g(x)$.

Algorithm: Fixed Point Iteration.
Given a scalar continuous function in one variable, $f(x)$, select a function $g(x)$ such that x satisfies $f(x)=0$ if and only if $g(x)=x$. Then:

1. Start from an *initial guess* x_0.
2. For $k=0,1,2,\ldots$, set
$$x_{k+1}=g(x_k)$$
until x_{k+1} satisfies termination criteria.

Suppose that we have somehow determined the continuous function $g \in C[a,b]$, and let us consider the fixed point iteration. Obvious questions arise:

1. Is there a fixed point x^* in $[a,b]$?
2. If yes, is it unique?
3. Does the sequence of iterates converge to a root x^*?
4. If yes, how fast?
5. If not, does this mean that no root exists?

Fixed point theorem

To answer the first question above, suppose that there are two values $a<b$ such that $g(a)\ge a$ and $g(b)\le b$. If $g(a)=a$ or $g(b)=b$, then a fixed point has been found, so now assume $g(a)>a$ and $g(b)<b$. Then for the continuous function

$$\phi(x)=g(x)-x$$

we have $\phi(a)>0$, $\phi(b)<0$. (Note that ϕ *does not* have to coincide with the function f that we have started with; there are lots of ϕ's for a given f.) Hence, by the Intermediate Value Theorem given on page 10, just as before, there is a root $a<x^*<b$ such that $\phi(x^*)=0$. Thus, $g(x^*)=x^*$, so x^* is a fixed point. We have established the existence of a root: $f(x^*)=0$.

Next, suppose that g is not only continuous but also differentiable and that there is a positive number $\rho<1$ such that

$$|g'(x)|\le\rho, \quad a<x<b.$$

Then the root x^* is unique in the interval $[a,b]$, for if there is also y^* which satisfies $y^*=g(y^*)$, then

$$|x^*-y^*|=|g(x^*)-g(y^*)|=|g'(\xi)(x^*-y^*)|\le\rho|x^*-y^*|,$$

where ξ is an intermediate value between x^* and y^*. Obviously, this inequality can hold with $\rho<1$ only if $y^*=x^*$.

We can summarize our findings so far as the *Fixed Point Theorem*.

Theorem: Fixed Point.
If $g\in C[a,b]$ and $a\le g(x)\le b$ for all $x\in[a,b]$, then there is a fixed point x^* in the interval $[a,b]$.

If, in addition, the derivative g' exists and there is a constant $\rho<1$ such that the derivative satisfies
$$|g'(x)|\le\rho \quad \forall\, x\in(a,b),$$
then the fixed point x^* is unique in this interval.

Convergence of the fixed point iteration

Turning to the fixed point iteration and the third question on the preceding page, similar arguments establish convergence (now that we know that there is a unique solution): under the same assumptions we have

$$|x_{k+1}-x^*|=|g(x_k)-g(x^*)|\le\rho|x_k-x^*|.$$

This is a **contraction** by the factor ρ. So

$$|x_{k+1}-x^*|\le\rho|x_k-x^*|\le\rho^2|x_{k-1}-x^*|\le\cdots\le\rho^{k+1}|x_0-x^*|.$$

Since $\rho<1$, we have $\rho^k\to 0$ as $k\to\infty$. This establishes convergence of the fixed point iteration.

Example 3.4. For the function $g(x)=e^{-x}$ on $[0.2,1.2]$, Figure 3.2 shows the progression of iterates towards the fixed point x^* satisfying $x^*=e^{-x^*}$.

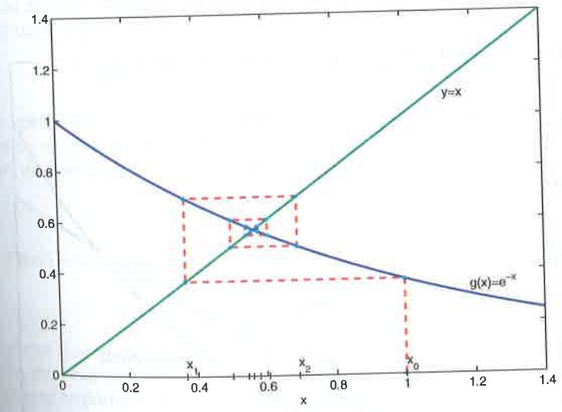

Figure 3.2. *Fixed point iteration for $x=e^{-x}$, starting from $x_0=1$. This yields $x_1=e^{-1}$, $x_2=e^{-e^{-1}}$, ... Convergence is apparent.*

Note the contraction effect. ∎

To answer the question about the speed of convergence, it should be clear from the bounds derived before Example 3.4 that the smaller ρ is, the faster the iteration converges. In case of the

bisection method, the speed of convergence does not depend on the function f (or any g, for that matter), and it is in fact identical to the speed of convergence obtained with $\rho \equiv 1/2$. In contrast, for the fixed point iteration, there is dependence on $|g'(x)|$. This, and the (negative) answer to our fifth question on page 46, are demonstrated next.

Example 3.5. Consider the function

$$f(x) = \alpha \cosh(x/4) - x,$$

where α is a parameter. For $\alpha = 10$ we saw in Example 3.1 that there is no root. But for $\alpha = 2$ there are actually two roots. Indeed, setting

$$g(x) = 2\cosh(x/4)$$

and plotting $g(x)$ and x as functions of x, we obtain Figure 3.3, which suggests not only that there are two fixed points (i.e., roots of f)—let's call them x_1^* and x_2^*—but also that we can bracket them, say, by

$$2 \le x_1^* \le 4, \quad 8 \le x_2^* \le 10.$$

Next we apply our trusted routine `bisect`, introduced in the previous section, to $f(x)$ with $\alpha = 2$. This yields

$$x_1^* \approx 2.35755106, \quad x_2^* \approx 8.50719958,$$

correct to an absolute tolerance 1.e-8, in 27 iterations for each root. (You should be able to explain why precisely the same number of iterations was required for each root.)

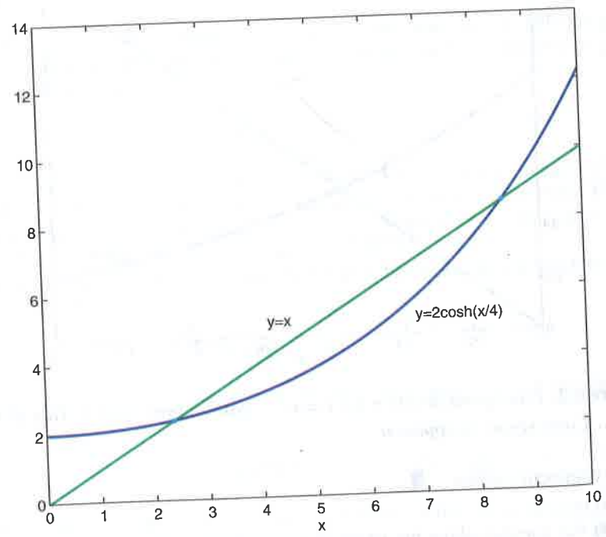

Figure 3.3. *The functions x and $2\cosh(x/4)$ meet at two locations.*

Now, it is very tempting, and rather natural here, to use the same function g defined above in a fixed point iteration, thus defining

$$x_{k+1} = 2\cosh(x_k/4).$$

For the first root, on the interval $[2,4]$ we have the conditions of the Fixed Point Theorem holding. In fact, near x_1^*, $|g'(x)| < 0.4$, so we expect faster convergence than with the bisection method (why?). Indeed, starting from $x_0 = 2$ the method requires 16 iterations, and starting from $x_0 = 4$ it requires 18 iterations to get to within 1.e-8 of the root x_1^*.

For the second root, however, on the interval $[8,10]$ the conditions of the Fixed Point Theorem do not hold! In particular, $|g'(x_2^*)| > 1$. Thus, a fixed point iteration using this g *will not* converge to the root x_2^*. Indeed, starting with $x_0 = 10$ the iteration diverges quickly, and we obtain *overflow* after 3 iterations, whereas starting this fixed point iteration with $x_0 = 8$ we do obtain convergence, but to x_1^* rather than to x_2^*.

It is important to realize that the root x_2^* is there, even though the fixed point iteration with the "natural g" does not converge to it. Not everything natural is good for you! There are other choices of g for the purpose of fixed point iteration that perform better here. ∎

Note: A discussion of *rates of convergence* similar to that appearing here is also given in Section 7.3. There it is more crucial, because here we will soon see methods that converge faster than a rate of convergence can suitably quantify.

Rate of convergence

Suppose now that at a particular root of a given fixed point iteration, $\rho = |g'(x^*)|$ satisfies $0 < \rho < 1$. Starting with x_0 sufficiently close to x^* we can write $x_k - x^* \approx g'(x^*)(x_{k-1} - x^*)$. Hence we get

$$|x_k - x^*| \approx \rho |x_{k-1} - x^*| \approx \cdots \approx \rho^k |x_0 - x^*|.$$

To quantify the speed of convergence in terms of ρ we can ask, how many iterations does it take to reduce the error by a fixed factor, say, 10?

To answer this we set $|x_k - x^*| \approx 0.1|x_0 - x^*|$, obtaining

$$\rho^k \approx 0.1.$$

Taking $\log_{10}$ of both sides yields $k \log_{10} \rho \approx -1$. Let us define the *rate of convergence* as

$$rate = -\log_{10} \rho. \tag{3.1}$$

Then it takes about $k = \lceil 1/rate \rceil$ iterations to reduce the error by more than an order of magnitude. Obviously, the smaller ρ the higher the convergence rate and correspondingly, the smaller the number of iterations required to achieve the same error reduction effect.

Example 3.6. The bisection method is not exactly a fixed point iteration, but it corresponds to an iterative method of a similar sort with $\rho = 0.5$. Thus its convergence rate according to (3.1) is

$$rate = -\log_{10} 0.5 \approx 0.301.$$

This yields $k = 4$, and indeed the error reduction factor with this many bisections is 16, which is more than 10.

For the root x_1^* of Example 3.5 we have $\rho \approx 0.3121$ and

$$rate = -\log_{10} 0.3121 \approx 0.506.$$

Here it takes only two iterations to roughly reduce the error by a factor of 10 if starting close to x^*.

For the root x_2^* of Example 3.5 we obtain from (3.1) a *negative* rate of convergence, as is appropriate for a case where the method does not converge. ∎

Of course, it makes no sense to calculate ρ or the convergence rate so precisely as in Example 3.6: we did this only so that you can easily follow the algebra with a calculator. Indeed, such accurate estimation of ρ would require knowing the solution first! Moreover, there is nothing magical about an error reduction by a particularly precise factor. The rate of convergence should be considered as a rough indicator only. It is of importance, however, to note that the rate becomes negative for $\rho > 1$, indicating no convergence of the fixed point iteration, and that it becomes infinite for $\rho = 0$, indicating that the error reduction per iteration is faster than by any constant factor. We encounter such methods next.

Specific exercises for this section: Exercises 3–4.

3.4 Newton's method and variants

The bisection method requires the function f to be merely continuous (which is good) and makes no further use of further information on f such as availability of its derivatives (which causes it to be painfully slow at times). At the other end of the scale is *Newton's method*, which requires more knowledge and smoothness of the function f but which converges much faster in appropriate circumstances.

Newton's method

Newton's method is the most basic fast method for root finding. The principle we use below to derive it can be directly extended to more general problems.

Assume that the first and second derivatives of f exist and are continuous: $f \in C^2[a,b]$. Assume also that f' can be evaluated with sufficient ease. Let x_k be a current iterate. By Taylor's expansion on page 5 we can write

$$f(x) = f(x_k) + f'(x_k)(x - x_k) + f''(\xi(x))(x - x_k)^2/2,$$

where $\xi(x)$ is some (unknown) point between x and x_k.

Now, set $x = x^*$, for which $f(x^*) = 0$. If f were linear, i.e., $f'' \equiv 0$, then we could find the root by solving $0 = f(x_k) + f'(x_k)(x^* - x_k)$, yielding $x^* = x_k - f(x_k)/f'(x_k)$. For a nonlinear function we therefore define the next iterate by the same formula, which gives

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \ldots.$$

This corresponds to neglecting the term $f''(\xi(x^*))(x^* - x_k)^2/2$ when defining the next iterate; however, if x_k is already close to x^*, then $(x^* - x_k)^2$ is very small, so we would expect x_{k+1} to be much closer to x^* than x_k is.

A *geometric interpretation* of Newton's method is that x_{k+1} is the x-intercept of the tangent line to f at x_k; see Figure 3.4.

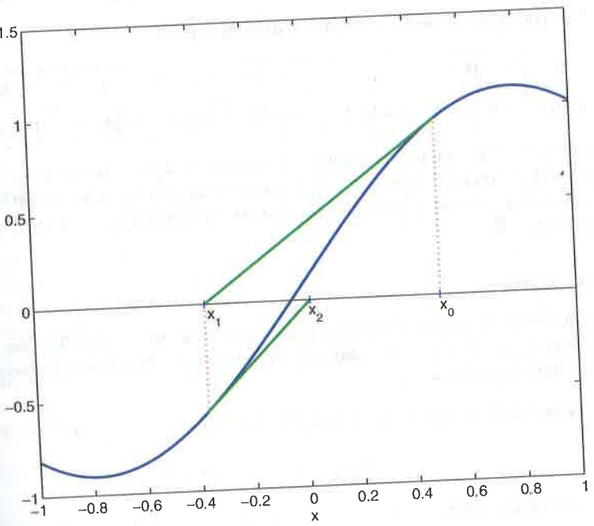

Figure 3.4. *Newton's method: the next iterate is the x-intercept of the tangent line to f at the current iterate.*

Algorithm: Newton's Method.
Given a scalar differentiable function in one variable, $f(x)$:

1. Start from an *initial guess* x_0.
2. For $k = 0, 1, 2, \ldots$, set
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$
until x_{k+1} satisfies termination criteria.

Example 3.7. Consider the same function as in Example 3.5, given by

$$f(x) = 2\cosh(x/4) - x.$$

The Newton iteration here is

$$x_{k+1} = x_k - \frac{2\cosh(x_k/4) - x_k}{0.5\sinh(x_k/4) - 1}.$$

We use the same absolute tolerance of 1.e-8 and the same four initial iterates as in Example 3.5.

- Starting from $x_0 = 2$ requires 4 iterations to reach x_1^* to within the given tolerance.
- Starting from $x_0 = 4$ requires 5 iterations to reach x_1^* to within the given tolerance.
- Starting from $x_0 = 8$ requires 5 iterations to reach x_2^* to within the given tolerance.
- Starting from $x_0 = 10$ requires 6 iterations to reach x_2^* to within the given tolerance.