Chapter 6

Root Finding

We were introduced to the concept of finding roots of a function in basic algebra courses. For example, we learned the quadratic formula so that we could factor quadratic polynomials, and we found x-intercepts provided key points when we sketched graphs. In many practical applications, though, we are presented with problems involving very complicated functions for which basic algebraic techniques cannot be used, and we must resort to computational methods. In this chapter we consider the problem of computing roots of a general nonlinear function of one variable. We describe and analyze several techniques, including the well known Newton iteration. We show how the Newton iteration can be applied to the rather special, but important, problem of calculating square roots. We also discuss how to calculate real roots in the special case when the nonlinear function is a polynomial of arbitrary degree. The chapter ends with a discussion of implementation details, and we describe how to use some of MATLAB's built-in root finding functions.

6.1 Roots and Fixed Points

A **root** of a function f is a point, $x = x_*$, where the equation

$$f(x_*) = 0$$

is satisfied. A point $x = x_*$ is a **simple root** if the following properties are satisfied:

- $f(x_*) = 0$ (that is, x_* is a root of $f(x)$),
- the function $f'(x)$ exists everywhere on some open interval containing x_*, and
- $f'(x_*) \neq 0$.

Geometrically these conditions imply that, at a simple root x_*, the graph of $f(x)$ crosses the x-axis. However this geometric interpretation can be deceiving because it is possible to have a function $f(x)$ whose graph crosses the x-axis at a point x_*, but also have $f'(x_*) = 0$. In such a situation x_* is *not* a simple root of $f(x)$; see Example 6.1.2.

Example 6.1.1. The point $x_* = 1$ is a root of each of the functions

$$f(x) = x - 1, \quad f(x) = \ln(x), \quad f(x) = (x-1)^2, \quad \text{and} \quad f(x) = \sqrt{x-1}.$$

It appears from the graphs of these functions, shown in Fig. 6.1, that $x_* = 1$ is a simple root for $f(x) = x - 1$ and $f(x) = \ln(x)$, but to be sure we must check that the above three conditions are satisfied. In the case of $f(x) = \ln(x)$, we see that

$$\begin{aligned} &f(1) = \ln(1) = 0 \\ &f'(x) = \tfrac{1}{x} \quad \text{exists on, for example, the open interval } (\tfrac{1}{2}, \tfrac{3}{2}) \\ &f'(1) = 1 \neq 0. \end{aligned}$$

Thus $x_* = 1$ is a simple root for $f(x) = \ln(x)$.

In the case of $f(x) = (x-1)^2$, we see that $f'(1) = 0$, so $x_* = 1$ is not a simple root for this function. (A similar situation occurs for $(x-1)^n$ for any even integer $n > 0$.) In the case of $f(x) = \sqrt{x-1}$, the function $f'(x)$ does not exist for $x \leq 1$, so $x_* = 1$ is also not a simple root for this function.

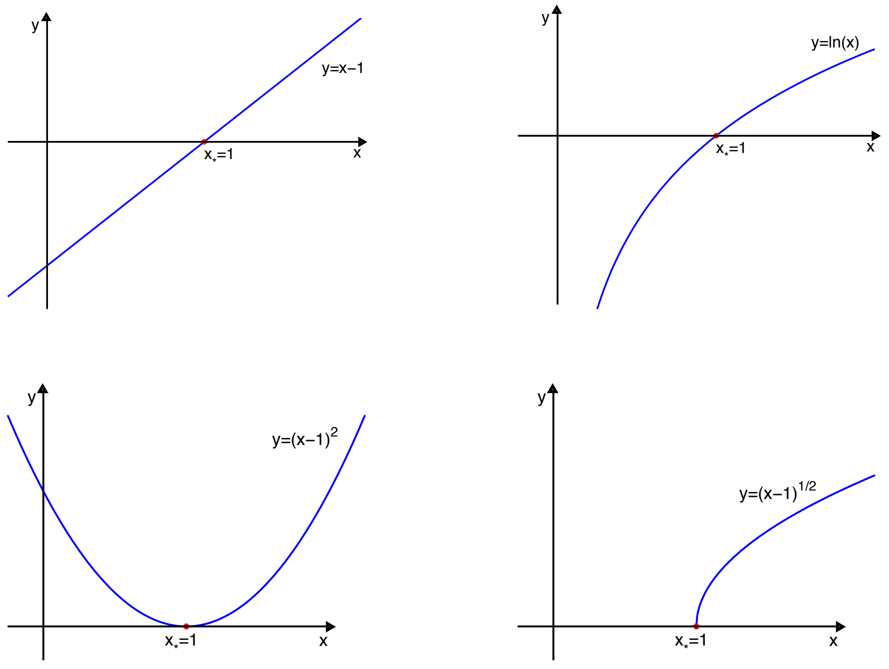

Figure 6.1: Graphs of the functions given in Example 6.1.1. $x_* = 1$ is a simple root of $f(x) = x - 1$ and $f(x) = \ln(x)$. It is a root, but it is *not* simple, for $f(x) = (x-1)^2$ and $f(x) = \sqrt{x-1}$.

Example 6.1.2. The graph of the function $f(x) = (x-1)^3$ is shown in Fig. 6.2. Observe that the graph crosses the x-axis at $x_* = 1$. However, $f'(x) = 3(x-1)^2$, and so $f'(x_*) = f'(1) = 0$. Thus, although $x_* = 1$ is root of this function, it is *not* a simple root. In fact, a similar situation occurs for $f(x) = (x-1)^n$ for any odd integer $n > 1$.

Since $f(x) = (x-1)^3$ crosses the x−axis, many methods for simple roots work also for this function though, in some cases, more slowly.)

Typically, the first issue that needs to be addressed when attempting to compute a simple root is to make an initial estimate of its value. One way to do this is to apply one of the most important theorems from Calculus, the Intermediate Value Theorem (see Problem 6.1.1). This theorem states that if a function f is continuous on the closed interval $[c, d]$, and if $f(a) \cdot f(b) < 0$, then there is a point $x_* \in (c, d)$ where $f(x_*) = 0$. That is, if the function f is continuous and changes sign on an interval, then it must have at least one root in that interval. We exploit this property in Section 6.3.3 when we discuss the bisection method for finding roots, but in this section we begin with a related problem. (Of course, we can use our graphing calculator and its zoom feature for obtaining a good approximation to a root for functions that are simple enough to graph accurately.)

Root finding problems often occur in **fixed–point** form; that is, the problem is to find a fixed–point $x = x_*$ that satisfies the equation

$$x_* = g(x_*).$$

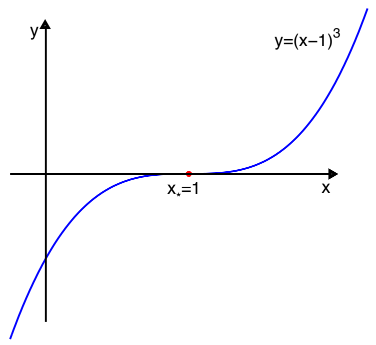

Figure 6.2: Graph of the function, $f(x) = (x-1)^3$, discussed in Example 6.1.2. In this case the graph crosses the x-axis at the root $x_* = 1$, but it is *not* a simple root because $f'(x_*) = f'(1) = 0$.

This type of problem sometimes arises directly when finding an equilibrium of some process. We can convert a problem written in fixed–point form, $x = g(x)$, to the standard form of solving the equation $f(x) = 0$ simply by defining $f(x) \equiv x - g(x) = 0$. Then, we know that the root is simple as long as $f'(x_*) = 1 - g'(x_*) \neq 0$. However, first we consider this problem in its natural fixed–point form.

Example 6.1.3. Consider the equation

$$x = 5 - \frac{3}{x+2}.$$

This is in fixed–point form $x = g(x)$, where $g(x) = 5 - \dfrac{3}{x+2}$. Geometrically, the fixed–points are the values x_* where the graphs of $y = x$ and $y = g(x)$ intersect. Fig. 6.3 shows a plot of the graphs of $y = x$ and $y = g(x)$ for this example. Notice that there are two locations where the graphs intersect, and thus there are two fixed–points.

We can convert the fixed–point form to the standard form $f(x) = 0$:

$$f(x) = x - g(x) = x - 5 + \frac{3}{x+2} = 0.$$

For this particular example it is easy to algebraically manipulate the equations to find the fixed–points of $x = g(x)$, or equivalently, the roots of $f(x) = 0$. In particular,

$$x - 5 + \frac{3}{x+2} = 0 \quad \Rightarrow \quad x^2 - 3x - 7 = 0,$$

and using the quadratic equation we find that the fixed–points are $x_* = \dfrac{3 \pm \sqrt{37}}{2}$.

Problem 6.1.1. The Intermediate Value Theorem. *Let the function f be continuous on the closed interval $[a, b]$. Show that for any number v between the values $f(a)$ and $f(b)$ there exists at least one point $c \in (a, b)$ such that $f(c) = v$.*

Problem 6.1.2. *Consider the function $f(x) = x - e^{-x}$. Use the Intermediate Value Theorem (see Problem 6.1.1) to prove that the equation $f(x) = 0$ has a solution $x_* \in (0, 1)$.*

Problem 6.1.3. The Mean Value Theorem. *Let the function f be continuous and differentiable on the closed interval $[a, b]$. Show that there exists a point $c \in (a, b)$ such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

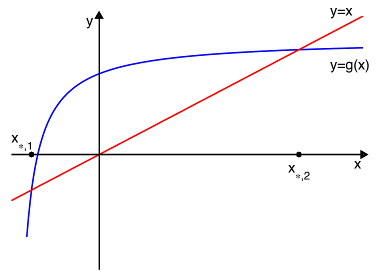

Figure 6.3: Graphs of $y = x$ and $y = g(x)$ for the function $g(x)$ discussed in Example 6.1.3. In this case there are two fixed–points, which are denoted as $x_{*,1}$ and $x_{*,2}$.

Problem 6.1.4. *Consider the function $f(x) = x - e^{-x}$ on the interval $[a, b] = [0, 1]$. Use the Mean Value Theorem (see Problem 6.1.3) to find a point $c \in (0, 1)$ where*

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

.

Problem 6.1.5. *Consider the functions $f(x) = (x - 1)(x + 2)^2$ and $F(x) = x\sqrt{x + 2}$.*

1. *What are the roots of f?*
2. *What are the simple roots of f?*
3. *What are the roots of F?*
4. *What are the simple roots of F?*

Problem 6.1.6. *Consider the equation $x = \cos\left(\frac{\pi}{2}x\right)$. Convert this equation into the standard form $f(x) = 0$ and hence show that it has a solution $x_* \in (0, 1)$ radians. Is this solution simple? [Hint: Use the Intermediate Value Theorem (see Problem 6.1.1) on $f(x)$.]*

6.2 Fixed–Point Iteration

A commonly used technique for computing a solution x_* of an equation $x = g(x)$, and hence a root of $f(x) = x - g(x) = 0$, is to define a sequence $\{x_n\}_{n=0}^{\infty}$ of approximations of x_* according to the iteration

$$x_{n+1} = g(x_n), \quad n = 0, 1, \ldots$$

Here x_0 is an initial estimate of x_* and g is *the fixed–point iteration function.* If the function g is continuous and the sequence $\{x_n\}_{n=0}^{\infty}$ converges to x_*, then x_* satisfies $x_* = g(x_*)$. In practice, to find a good starting point x_0 for the fixed–point iteration we may either:

- Estimate x_* from a graph of $f(x) = x - g(x)$. That is, we attempt to estimate where the graph of $f(x)$ crosses the x-axis.
- Alternatively, we estimate the intersection point of the graphs of $y = x$ and $y = g(x)$.

Unfortunately, even if we begin with a very accurate initial estimate, the fixed–point iteration does not always converge. Before we present mathematical arguments that explain the convergence properties of the fixed–point iteration, we try to develop an understanding of the convergence behavior through a series of examples.

6.2.1 Examples of Fixed–Point Iteration

In this subsection we explore the convergence behavior of the fixed–point iteration through a series of examples.

Example 6.2.1. Suppose we want to find $x = x_*$ to satisfy the following equivalent equations:

Fixed-point form: $x = g(x)$	Root form: $f(x) = x - g(x) = 0$
$x = e^{-x}$	$x - e^{-x} = 0$

The graph of $f(x) = x - e^{-x}$ is shown in Fig. 6.4.

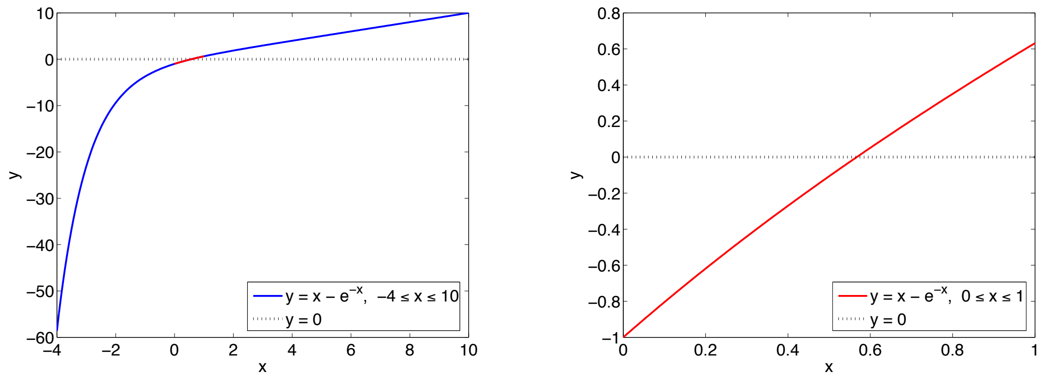

Figure 6.4: Graph of $f(x) = x - e^{-x}$. The left plot shows the graph on the interval $-4 \le x \le 10$, and the right plot zooms in on the region where the graph of f crosses the x-axis (the line $y = 0$).

(a) From the Intermediate Value Theorem (Problem 6.1.1) there is a root of the function f in the interval $(0, 1)$ because:

- $f(x)$ is continuous on $[0, 1]$, and
- $f(0) = -1 < 0$ and $f(1) = 1 - e^{-1} > 0$, and so $f(0) \cdot f(1) < 0$.

(b) From the graph of the derivative $f'(x) = 1 + e^{-x}$, shown in Fig. 6.5, we observe that this root is simple because $f'(x)$ is continuous and positive for all $x \in [0, 1]$.

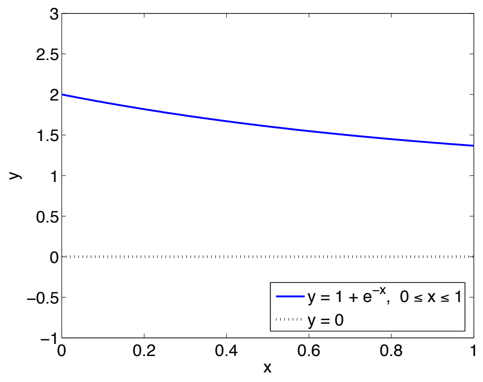

Figure 6.5: Graph of $f'(x) = 1 + e^{-x}$. Notice that f' is continuous and positive for $0 \le x \le 1$.

(c) From the graph of $f(x)$ shown in Fig. 6.4, we see that $x_0 = 0.5$ is a reasonable estimate of a root of $f(x) = 0$, and hence a reasonable estimate of a fixed–point of $x = g(x)$.

(d) Alternatively, the value x_* we seek can be characterized as the intersection point of the line $y = x$ and the curve $y = g(x) \equiv e^{-x}$ as in Fig. 6.6. From this graph we again see that $x_0 = 0.5$ is a reasonable initial estimate of x_*.

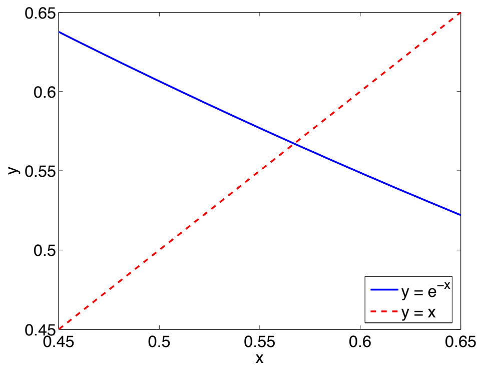

Figure 6.6: Graphs of $y = e^{-x}$ and $y = x$. The point where these graphs intersect is a fixed–point of $x = e^{-x}$, and hence a root of $x - e^{-x} = 0$.

(e) Of course we are relying rather too much on graphical evidence here. As we saw in Section 2.3.1 graphical evidence can be misleading. This is especially true of the evidence provided by low resolution devices such as graphing calculators. But we can easily prove there is a unique fixed–point of the equation $x = e^{-x}$. The curve $y = x$ is strictly monotonic increasing and continuous everywhere and the curve $y = e^{-x}$ is strictly monotonic decreasing and continuous everywhere. We have already seen these curves intersect so there is at least one fixed–point, and the monotonicity properties show that the fixed–point is simple and unique.

(f) If we apply the fixed–point iteration, $x_{n+1} = g(x_n)$, to this problem, we obtain:

$$\begin{array}{lclclcl}
x_0 & = & & & & & 0.5000 \\
x_1 & = & g(x_0) & = & e^{-x_0} & \approx & 0.6065 \\
x_2 & = & g(x_1) & = & e^{-x_1} & \approx & 0.5452 \\
x_3 & = & g(x_2) & = & e^{-x_2} & \approx & 0.5797 \\
x_4 & = & g(x_3) & = & e^{-x_3} & \approx & 0.5601 \\
x_5 & = & g(x_4) & = & e^{-x_4} & \approx & 0.5712 \\
x_6 & = & g(x_5) & = & e^{-x_5} & \approx & 0.5649 \\
x_7 & = & g(x_6) & = & e^{-x_6} & \approx & 0.5684 \\
x_8 & = & g(x_7) & = & e^{-x_7} & \approx & 0.5664 \\
x_9 & = & g(x_8) & = & e^{-x_8} & \approx & 0.5676 \\
x_{10} & = & g(x_9) & = & e^{-x_9} & \approx & 0.5669 \\
x_{11} & = & g(x_{10}) & = & e^{-x_{10}} & \approx & 0.5673 \\
x_{12} & = & g(x_{11}) & = & e^{-x_{11}} & \approx & 0.5671
\end{array}$$

These computations were performed using double precision arithmetic, but we show only the first four digits after the decimal point. The values x_n appear to be converging to $x_* = 0.567\cdots$, and this could be confirmed by computing additional iterations.

Example 6.2.2. It is often the case that a fixed–point problem does not have a unique form. Consider the fixed–point problem from the previous example, $x = e^{-x}$. If we take natural logarithms of both sides of this equation and rearrange we obtain $x = -\ln(x)$ as an alternative fixed–point equation. Although both these equations have the same fixed-points, the corresponding fixed–point iterations, $x_{n+1} = g(x_n)$, behave very differently. Table 6.1 lists the iterates computed using each of the fixed–point iteration functions $g(x) = e^{-x}$ and $g(x) = -\ln(x)$. In each case we used $x_0 = 0.5$ as an initial estimate of x_*. The computations were performed using double precision arithmetic, but we show only the first four digits following the decimal point. The asterisks in the table are used to indicate that the logarithm of a negative number (e.g., $x_5 = \ln(x_4) \approx \ln(-0.0037)$) is not defined when using real arithmetic.

	Iteration Function	
	$x_{n+1} = e^{-x_n}$	$x_{n+1} = -\ln(x_n)$
n	x_n	x_n
0	0.5000	0.5000
1	0.6065	0.6931
2	0.5452	0.3665
3	0.5797	1.0037
4	0.5601	-0.0037
5	0.5712	******
6	0.5649	******
7	0.5684	******
8	0.5664	******
9	0.5676	******
10	0.5669	******
11	0.5673	******
12	0.5671	******

Table 6.1: Computing the root of $x - e^{-x} = 0$ by fixed–point iteration. The asterisks indicate values that are not defined when using only real arithmetic. Note, if complex arithmetic is used as in MATLAB, these values are defined and are complex numbers.

As we saw in the previous example, the iterates computed using the iteration function $g(x) = e^{-x}$ converge to $x_* = 0.5671\cdots$. But here we see that the iterates computed using the iteration function $g(x) = -\ln(x)$ diverge. Why should these two fixed–point iterations behave so differently? The difficulty is not that we started from $x_0 = 0.5$. We would observe similar behavior starting from any point sufficiently close to $x_* = 0.5671\cdots$.

Recall that the fixed–point can be characterized by the point where the curves $y = x$ and $y = g(x)$ intersect. The top two plots in Fig. 6.7 show graphs of these intersecting curves for the two fixed–point functions $g(x) = e^{-x}$ and $g(x) = -\ln(x)$. In the bottom two plots of this same figure, we show, geometrically, how the fixed–point iteration proceeds:

$$x_0 \to g(x_0) \to x_1 \to g(x_1) \to x_2 \to g(x_2) \to \cdots$$

We observe:

- In the case of $g(x) = e^{-x}$, near the intersection point, the slope of $g(x)$ is relatively flat, and each evaluation $g(x_n)$ brings us closer to the intersection point.
- In the case of $g(x) = -\ln(x)$, near the intersection point, the slope of $g(x)$ is relatively steep, and each evaluation $g(x_n)$ pushes us away from the intersection point.

In general, if the slope satisfies $|g'(x)| < 1$ near $x = x_*$, and if the initial estimate x_0 is "close enough" to x_*, then we can expect the fixed–point iteration to converge. Moreover, by looking at these plots we might suspect that very flat slopes (that is, $|g'(x)| \approx 0$) result in faster convergence, and less flat slopes (that is, $|g'(x)| \lessapprox 1$) result in slower convergence. In the next subsection we provide mathematical arguments to confirm these geometrical observations.

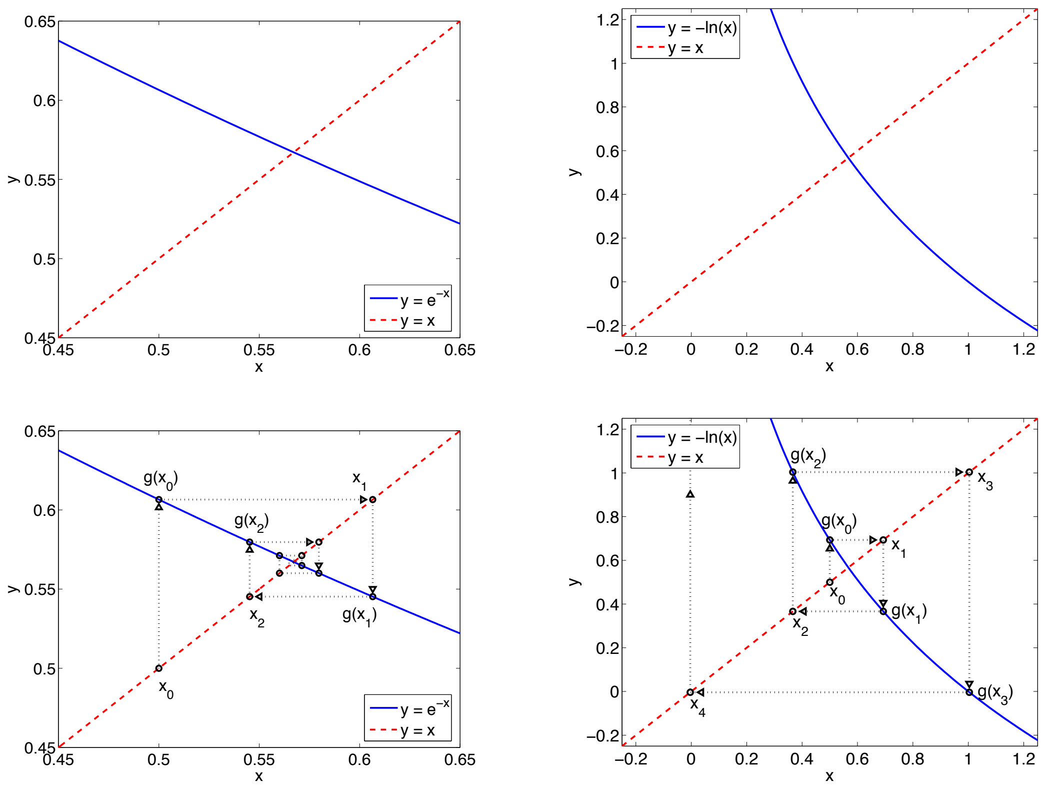

Figure 6.7: Graphs of $y = g(x)$ and $y = x$, where $g(x) = e^{-x}$ (left) and $g(x) = -\ln(x)$ (right). The point where these graphs intersect is the desired fixed–point. The bottom two plots show how the fixed–point iteration proceeds: $x_0 \to g(x_0) \to x_1 \to g(x_1) \to x_2 \to g(x_2) \to \cdots$ In the case of $g(x) = e^{-x}$ these values converge toward the fixed–point, but in the case of $g(x) = -\ln(x)$ these values move away from the fixed–point.

Example 6.2.3. Consider the cubic polynomial $f(x) = x^3 - x - 1$, whose three roots comprise one real root and a complex conjugate pair of roots. From the graph of f, shown in Fig. 6.8, we see that the one real root is $x_* \approx 1.3$. In Section 6.5 we look at the specific problem of computing roots of a polynomial, but here we consider using a fixed–point iteration. The first thing we need to do is algebraically manipulate the equation $f(x) = 0$ to obtain an equation of the form $x = g(x)$. There are many ways we might do this; consider, for example:

- Approach 1.

  $$x^3 - x - 1 = 0 \quad \Rightarrow \quad x^3 = x + 1. \quad \Rightarrow \quad x = \sqrt[3]{x+1}$$

  In this case we have the fixed–point iteration: $x_{n+1} = \sqrt[3]{x_n + 1}$.

- Approach 2.

  $$x^3 - x - 1 = 0 \quad \Rightarrow \quad x = x^3 - 1.$$

  In this case we have the fixed–point iteration: $x_{n+1} = x_n^3 - 1$.

Fig. 6.9 displays plots showing the intersection points of the graphs of $y = x$ and $y = g(x)$ for each of the above fixed–point iteration functions.

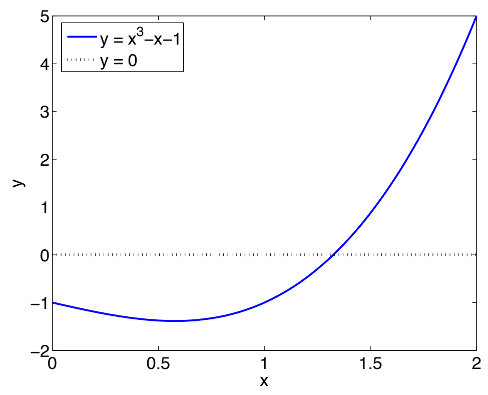

Figure 6.8: Graph of $f(x) = x^3 - x - 1$.

From the discussion in the previous example, we might expect the fixed–point iteration

$$x_{n+1} = \sqrt[3]{x_n + 1}$$

to converge very quickly because near the intersection point the slope of $g(x) = \sqrt[3]{x+1}$ is very flat. On the other hand, we might expect the fixed–point iteration

$$x_{n+1} = x_n^3 - 1$$

to diverge because near the intersection point the slope of $g(x) = x^3 - 1$ is very steep. This is, in fact, what we observe in our computational results. Starting from the point $x_0 = 1.0$ for each of these iteration functions, we compute the results in Table 6.2. The entry HUGE in the table corresponds to a number clearly too large to be of interest; the iteration ultimately overflows on any computer or calculator! The iterates generated using the iteration function $g(x) = \sqrt[3]{x+1}$ converge to the root $x_* = 1.3247\cdots$, while the iterates generated using the iteration function $g(x) = x^3 - 1$ diverge.

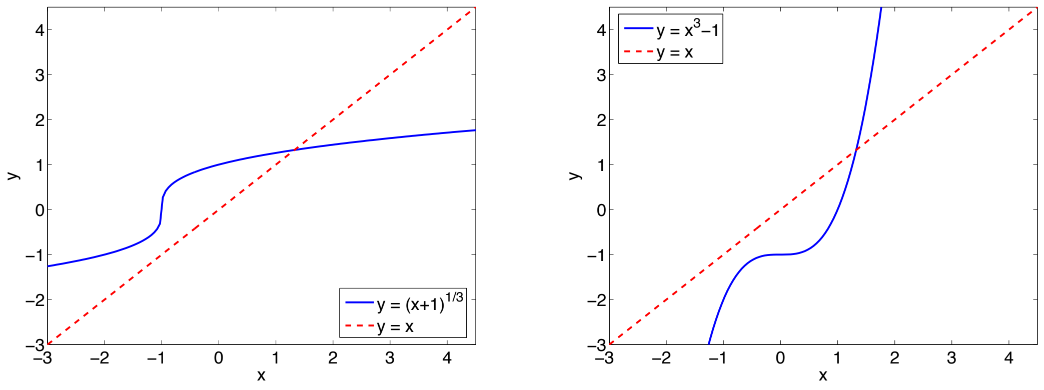

Figure 6.9: Graphs of $y = g(x)$ and $y = x$, where $g(x) = \sqrt[3]{x+1}$ (left) and $g(x) = x^3 - 1$ (right).

Problem 6.2.1. *Assume that the function g is continuous and that the sequence $\{x_n\}_{n=0}^{\infty}$ converges to a fixed–point x_*. Employ a limiting argument to show that x_* satisfies $x_* = g(x_*)$.*

	Iteration Function	
	$x_{n+1} = \sqrt[3]{x_n + 1}$	$x_{n+1} = x_n^3 - 1$
n	x_n	x_n
0	1.000	1.000
1	1.260	0.000
2	1.312	-1.000
3	1.322	-2.000
4	1.324	-9.000
5	1.325	-730.0
6	1.325	HUGE

Table 6.2: Computing the real root of $x^3 - x - 1 = 0$ by fixed–point iteration.

Problem 6.2.2. *Consider the function $f(x) = x - e^{-x}$. The equation $f(x) = 0$ has a root x_* in $(0,1)$. Use Rolle's Theorem (see Problem 4.2.1) to prove that this root x_* of $f(x)$ is simple.*

Problem 6.2.3.
1. *Show algebraically by rearrangement that the equation $x = -\ln(x)$ has the same roots as the equation $x = e^{-x}$*
2. *Show algebraically by rearrangement that the equation $x = \sqrt[3]{x+1}$ has the same roots as the equation $x^3 - x - 1 = 0$*
3. *Show algebraically by rearrangement that the equation $x = x^3 - 1$ has the same roots as the equation $x^3 - x - 1 = 0$*

Problem 6.2.4. *Prove that the equation $x = -\ln(x)$ has a unique simple real root.*

Problem 6.2.5. *Prove that the equation $x^3 - x - 1 = 0$ has a unique simple real root.*

Problem 6.2.6. *For $f(x) = x^3 - x - 1$, show that the rearrangement $x^3 - x - 1 = x(x^2 - 1) - 1 = 0$ leads to a fixed–point iteration function $g(x) = \dfrac{1}{x^2 - 1}$, and show that the rearrangement $x^3 - x - 1 = (x-1)(x^2 + x + 1) - x = 0$ leads to the iteration function $g(x) = 1 + \dfrac{x}{x^2 + x + 1}$. How do the iterates generated by each of these fixed–point functions behave? [In each case compute a few iterates starting from $x_0 = 1.5$.]*

6.2.2 Convergence Analysis of Fixed–Point Iteration

In the previous subsection we observed that the convergence behavior of the fixed–point iteration appears to be related to the magnitude of the slope of the iteration function, $|g'(x)|$, near the fixed–point. In this subsection we solidify these observations with mathematical arguments.

Given an initial estimate x_0 of the fixed–point x_* of the equation $x = g(x)$, we need two properties of the iteration function g to ensure that the sequence of iterates $\{x_n\}_{n=1}^{\infty}$ defined by

$$x_{n+1} = g(x_n), \quad n = 0, 1, \ldots,$$

converges to x_*:

- $x_* = g(x_*)$; that is, x_* is a fixed–point of the iteration function g.

- For any n, let's define $e_n = x_n - x_*$ to be the **error** in x_n considered as an approximation to x_*. Then, for some constant $N > 0$ and all $n > N$ the sequence of absolute errors $\{|e_n|\}_{n=0}^{\infty}$ decreases monotonically to 0; that is, for each such value of n, the iterate x_{n+1} is closer to the point x_* than was the previous iterate x_n.

Assuming that the derivative g' is continuous wherever we need it to be, from the Mean Value Theorem (Problem 6.1.3), we have

$$\begin{aligned} e_{n+1} &= x_{n+1} - x_* \\ &= g(x_n) - g(x_*) \\ &= g'(\eta_n) \cdot (x_n - x_*) \\ &= g'(\eta_n) \cdot e_n \end{aligned}$$

where η_n is a point between x_* and x_n.

Summarizing, the *key equation*

$$e_{n+1} = g'(\eta_n) \cdot e_n$$

states that the new error e_{n+1} is a factor $g'(\eta_n)$ times the old error e_n.

Now, the new iterate x_{n+1} is closer to the root x_* than is the old iterate x_n if and only if $|e_{n+1}| < |e_n|$. This is guaranteed if and only if $|g'(\eta_n)| \le G < 1$ for some constant G. So, informally, if $|g'(\eta_n)| \le G < 1$ for each value of η_n arising in the iteration, the absolute errors tend to zero and the iteration converges.

More specifically:

- If for some constant $0 < G < 1$, $|g'(x)| \le G$ (the **tangent line condition**) for all $x \in I = [x_* - r, x_* + r]$ then, for every starting point $x_0 \in I$, the sequence $\{x_n\}_{n=0}^{\infty}$ is *well defined* and *converges* to the fixed–point x_*. (Note that, as $x_n \to x_*$, $g'(x_n) \to g'(x_*)$ and so $g'(\eta_n) \to g'(x_*)$ since η_n is "trapped" between x_n and x_*.)

- The smaller the value of $|g'(x_*)|$, the more rapid is the ultimate rate of convergence. Ultimately (that is, when the iterates x_n are so close to x_* that $g'(\eta_n) \approx g'(x_*)$) each iteration reduces the magnitude of the error by a factor of about $|g'(x_*)|$.

- If $g'(x_*) > 0$, convergence is ultimately one–sided. Because, when x_n is close enough to x_* then $g'(\eta_n)$ has the same sign as $g'(x_*) > 0$ since we assume that $g'(x)$ is continuous near x_*. Hence the error e_{r+1} has the same sign as the error e_r for all values of $r > n$.

- By a similar argument, if $g'(x_*) < 0$, ultimately (that is, when the iterates x_n are close enough to x_*) convergence is two–sided because the signs of the errors e_r and e_{r+1} are opposite for each value of $r > n$.

We have chosen the interval I to be *centered* at x_*. This is to simplify the theory. Then, if x_0 lies in the interval I so do all future iterates x_n. The disadvantage of this choice is that we don't know the interval I until we have calculated the answer x_*! However, what this result does tell us is what to expect if we start close enough to the solution. If $|g'(x_*)| \le G < 1$, $g'(x)$ is continuous in a neighborhood of x_*, and x_0 is close enough to x_* then the fixed–point iteration

$$x_{n+1} = g(x_n), \quad n = 0, 1, \ldots,$$

converges to x_*. The convergence rate is *linear* if $g'(x_*) \ne 0$ and *superlinear* if $g'(x_*) = 0$; that is, ultimately

$$e_{n+1} \approx g'(x_*) e_n$$

Formally, if the errors satisfy

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^p} = K$$

where K is a positive constant, then the convergence rate is p. The case $p = 1$ is labeled linear, $p = 2$ quadratic, $p = 3$ cubic, etc.; any case with $p > 1$ is superlinear. The following examples illustrate this theory.

Example 6.2.4. Consider the fixed–point problem

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

Here we have $g(x) = e^{-x}$, $g'(x) = -e^{-x}$, and from Examples 6.2.1 and 6.2.2, we know this problem has a fixed–point $x_* = 0.5671\cdots$. Although we have already observed that the fixed–point iteration converges for this problem, we can now explain mathematically why this should be the case.

- If $x > 0$ then $-1 < g'(x) < 0$. Thus, for all $x \in (0, \infty)$, $|g'(x)| < 1$. In particular, we can choose an interval, $I \approx [0.001, 1.133]$ which is approximately centered symmetrically about x_*. Since $|g'(x)| < 1$ for all $x \in I$, then for any $x_0 \in I$, the fixed–point iteration will converge.
- Since $g'(x_*) = -0.567\cdots < 0$, convergence is ultimately two–sided. This can be observed geometrically from Fig. 6.7, where we see x_0 is to the left of the fixed–point, x_1 is to the right, x_2 is to the left, and so on.
- Ultimately, the error at each iteration is reduced by a factor of approximately

  $$\frac{|e_{n+1}|}{|e_n|} \approx |g'(x)| \approx |-0.567|,$$

  as can be seen from the computed results shown in Table 6.3.

n	x_n	e_n	e_n/e_{n-1}
0	0.5000	-0.0671	—
1	0.6065	0.0394	-0.587
2	0.5452	-0.0219	-0.556
3	0.5797	0.0126	-0.573
4	0.5601	-0.0071	-0.564
5	0.5712	0.0040	-0.569

Table 6.3: Errors and error ratios for fixed–point iteration with $g(x) = e^{-x}$

Example 6.2.5. Consider the fixed–point problem

$$x = \sqrt[3]{x+1}.$$

Here we have $g(x) = \sqrt[3]{x+1}$, $\quad g'(x) = \dfrac{1}{3(x+1)^{2/3}}$, and from Example 6.2.3, we know this problem has a fixed–point $x_* = 1.3247\cdots$. An analysis of the convergence behavior of the fixed–point iteration for this problem is as follows:

- If $x \geq 0$ then $x + 1 \geq 1$, and hence $0 \leq \dfrac{1}{x+1} \leq 1$. Therefore $0 \leq g'(x) = \dfrac{1}{3(x+1)^{2/3}} \leq \dfrac{1}{3}$. So $|g'(x)| \leq \dfrac{1}{3}$ for $x \in [0, \infty)$. In particular, choose an interval, $I \approx [0, 2.649]$ which is approximately symmetrically centered around x_*. Since $|g'(x)| < 1$ for all $x \in I$, then for any $x_0 \in I$, the fixed–point iteration will converge.
- Since $g'(x_*) \approx 0.189 > 0$, convergence is ultimately one–sided, and, since $|g'(x_*)|$ is quite small, convergence is speedy. This can be observed from the results shown in Table 6.2, where we see that all of $x_0, x_1, x_2, \ldots$ are to the left of (i.e., less than) x_*.

n	x_n	e_n	e_n/e_{n-1}
0	1.000	-0.325	—
1	1.260	-0.065	0.200
2	1.312	-0.012	0.192
3	1.322	-0.002	0.190
4	1.324	-0.000	0.190

Table 6.4: Errors and error ratios for fixed–point iteration with $g(x) = \sqrt[3]{x+1}$

- The error at each iteration is ultimately reduced by a factor of approximately

$$\frac{|e_{n+1}|}{|e_n|} \approx |g'(x)| \approx |0.189|,$$

as can be seen from the computed results shown in Table 6.4.

Problem 6.2.7. *For the divergent sequences shown in Tables 6.1 and 6.2 above, argue why*

$$|x_* - x_{n+1}| > |x_* - x_n|$$

whenever x_n is sufficiently close, but not equal, to x_*.

Problem 6.2.8. *Analyze the convergence, or divergence, of the fixed–point iteration for each of the iteration functions*

1. $g(x) \equiv \dfrac{1}{x^2 - 1}$
2. $g(x) \equiv 1 + \dfrac{x}{x^2 + x + 1}$

See Problem 6.2.6 for the derivation of these iteration functions. Does your analysis reveal the behavior of the iterations that you computed in Problem 6.2.6?

Problem 6.2.9. *Consider the fixed–point equation $x = 2\sin x$ for which there is a root $x_* \approx 1.90$ radians. Why would you expect the fixed–point iteration $x_{n+1} = 2\sin x_n$ to converge from a starting point x_0 close enough to x_*? Would you expect convergence to be ultimately one–sided or two–sided? Why?*
Use the iteration $x_{n+1} = 2\sin x_n$ starting (i) from $x_0 = 2.0$ and (ii) from $x_0 = 1.4$. In each case continue until you have 3 correct digits in the answer. Discuss the convergence behavior of these iterations in the light of your observations above.

6.3 Root Finding Methods

Let us turn now to finding roots – that is, points x_* such that $f(x_*) = 0$. The first two methods we consider, the Newton and secant iterations, are closely related to the fixed–point iteration discussed in Section 6.2. We then describe the bisection method, which has a slower rate of convergence than the Newton and secant iterations, but has the advantage that convergence can be guaranteed for relative accuracies greater than ϵ_{DP}. The final method we discuss is rather different than the first three, and is based on inverse interpolation.

6.3.1 The Newton Iteration

Suppose $f(x)$ is a given function, and we wish to compute a simple root of $f(x) = 0$. A systematic procedure for constructing an equivalent fixed–point problem, $x = g(x)$, is to choose the **Newton Iteration Function**

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

The associated fixed–point iteration $x_{n+1} = g(x_n)$ leads to the well–known **Newton Iteration**

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

Notice that if x_* satisfies $x_* = g(x_*)$, and if $f'(x_*) \neq 0$, then $f(x_*) = 0$, and hence x_* is a simple root of $f(x) = 0$.

How does this choice of iteration function arise? Here are two developments:

- An algebraic derivation can be obtained by expanding $f(z)$ around x using a Taylor series:

  $$f(z) = f(x) + (z - x)f'(x) + (z - x)^2 \frac{f''(x)}{2} + \cdots$$

  A Taylor polynomial approximation is obtained by terminating the infinite series after a finite number of terms. In particular, a linear polynomial approximation is given by

  $$f(z) \approx f(x) + (z - x)f'(x).$$

  Setting $z = x_{n+1}$, $f(x_{n+1}) = 0$ and $x = x_n$ we obtain

  $$0 \approx f(x_n) + (x_{n+1} - x_n)f'(x_n).$$

  Rearranging this relation gives the Newton iteration. Note, this is "valid" because we assume x_{n+1} is much closer to x_* than is x_n, so, usually, $f(x_{n+1})$ is much closer to zero than is $f(x_n)$.

- For a geometric derivation, observe that the curve $y = f(x)$ crosses the x-axis at $x = x_*$. In point-slope form, the tangent to $y = f(x)$ at the point $(x_n, f(x_n))$ is given by $y - f(x_n) = f'(x_n)(x - x_n)$. Let x_{n+1} be defined as the value of x where this tangent crosses the x-axis (see Fig. 6.10). Then the point $(x_{n+1}, 0)$ is on the tangent line, so we obtain

  $$0 - f(x_n) = f'(x_n)(x_{n+1} - x_n).$$

  Rearranging this equation, we obtain the Newton iteration.

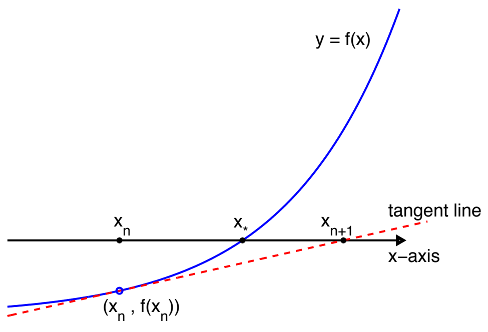

Figure 6.10: Illustration of Newton's method. x_{n+1} is the location where the tangent line to $y = f(x)$ through the point $(x_n, f(x_n))$ crosses the x-axis.

A short computation shows that, for the Newton iteration function g,

$$g'(x) = 1 - \frac{f'(x)}{f'(x)} + \frac{f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2}$$

If we assume that the root x_* is simple, then $f(x_*) = 0$ and $f'(x_*) \neq 0$, so

$$g'(x_*) = \frac{f(x_*)f''(x_*)}{f'(x_*)^2} = 0$$

From the previous section we know that, when x_n is close enough to x_*, then $e_{n+1} \approx g'(x_*)e_n$. So $g'(x_*) = 0$ implies e_{n+1} is very much smaller in magnitude than e_n when $|e_n|$ is itself sufficiently small, indeed convergence will be at least superlinear.

Example 6.3.1. Suppose $f(x) = x - e^{-x}$, and consider using Newton's method to compute a root of $f(x) = 0$. In this case, $f'(x) = 1 + e^{-x}$, and the Newton iteration is

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

In Table 6.5 we give the iterates, errors and error ratios for the Newton iteration . The correct digits for the iterates, x_n, are underlined. Notice that the number of correct digits approximately doubles for each iteration. Also observe that the magnitude of the ratio

$$\frac{|e_n|}{|e_{n-1}|^2}$$

is essentially constant, indicating quadratic convergence.

n	x_n	e_n	e_n/e_{n-1}	e_n/e_{n-1}^2
0	$0.\underline{5}00000000000000$	-6.7E-02	—	—
1	$0.\underline{56}6311003197218$	-8.3E-04	1.2E-02	0.1846
2	$0.\underline{567143}165034862$	-1.3E-07	1.5E-04	0.1809
3	$0.\underline{5671432904097}81$	2.9E-13	-2.3E-06	-0.1716

Table 6.5: Errors and error ratios for the Newton iteration applied to $f(x) = x - e^{-x}$.

Example 6.3.2. Suppose $f(x) = x^3 - x - 1$, and consider using Newton's method to compute a root of $f(x) = 0$. In this case, $f'(x) = 3x^2 - 1$, and the Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^3 - x_n - 1}{3x_n^2 - 1} = \frac{2x_n^3 + 1}{3x_n^2 - 1}\,.$$

In Table 6.6 we give the iterates, errors and error ratios for the Newton iteration . The correct digits for the iterates, x_n, are underlined. For this example, after the first iteration (once we are "close enough" to the root), we see again that the number of correct digits approximately doubles for each iteration. Moreover, the magnitude of the ratio

$$\frac{|e_n|}{|e_{n-1}|^2}$$

is essentially constant, indicating quadratic convergence.

n	x_n	e_n	e_n/e_{n-1}	e_n/e_{n-1}^2
0	1.00000000000000	-3.2E-01	—	—
1	1.50000000000000	1.8E-01	-5.4E-01	1.7E-00
2	1.34782608695652	2.3E-02	1.3E-01	7.5E-01
3	1.32520039895091	4.8E-04	2.1E-02	9.0E-01
4	1.32471817399905	2.2E-07	4.5E-04	9.3E-01
5	1.32471795724479	4.4E-14	2.0E-07	9.3E-01

Table 6.6: Errors and error ratios for the Newton iteration applied to $f(x) = x^3 - x - 1$.

These examples suggest that at each Newton iteration the number of correct digits in x_n approximately doubles: this behavior is typical for quadratically convergent iterations. Can we confirm this theoretically? First observe by definition of fixed–points and fixed–point iteration, we can represent the error as

$$\begin{aligned} e_{n+1} &= x_{n+1} - x_* \\ &= g(x_n) - g(x_*)\,. \end{aligned}$$

If we now expand $g(x_n)$ around x_* using a Taylor series, we obtain

$$\begin{aligned} e_{n+1} &= x_{n+1} - x_* \\ &= g(x_n) - g(x_*) \\ &= \left(g(x_*) + (x_n - x_*)g'(x_*) + (x_n - x_*)^2 \frac{g''(x_*)}{2} + \cdots \right) - g(x_*) \\ &= e_n g'(x_*) + e_n^2 \frac{g''(x_*)}{2} + \cdots \end{aligned}$$

where $e_n = x_n - x_*$. Recall that for Newton's iteration, $g'(x_*) = 0$. Then if we neglect the higher order terms not shown here, and assume $g''(x_*) \neq 0$ as it is in most cases, we obtain the error equation

$$e_{n+1} \approx \frac{g''(x_*)}{2} \cdot e_n^2$$

(For example, for the iteration in Table 6.5 $\dfrac{e_{n+1}}{e_n^2} \approx \dfrac{g''(x_*)}{2} \approx 0.1809$ and we have quadratic convergence.)

Now, let's check that this error equation implies ultimately doubling the number of correct digits. Dividing both sides of the error equation by x_* leads to the relation

$$r_{n+1} \approx \frac{x_*\, g''(x_*)}{2} \cdot r_n^2$$

where the relative error r_n in e_n is defined by $r_n \equiv \dfrac{e_n}{x_*}$. Suppose that x_n is correct to about t decimal digits, so that $r_n \approx 10^{-t}$ and choose d to be the integer such that

$$\frac{x_* g''(x_*)}{2} \approx \pm 10^d$$

where the sign of $\pm$ is chosen according to the sign of the left hand side. Then,

$$r_{n+1} \approx \frac{x_*\, g''(x_*)}{2} \cdot r_n^2 \approx \pm 10^d \cdot 10^{-2t} = \pm 10^{d-2t}$$

As the iteration converges, t grows and, ultimately, $2t$ will become much larger than d so then $r_{n+1} \approx \pm 10^{-2t}$. Hence, for sufficiently large values t, when r_n is accurate to about t decimal digits then r_{n+1} is accurate to about $2t$ decimal digits. This approximate doubling of the number of correct digits at each iteration is evident in Tables 6.5 and 6.6. When the error e_n is sufficiently small that the error equation is accurate then e_{n+1} has the same sign as $g''(x_*)$. Then, convergence for the Newton iteration is one–sided: from above if $g''(x_*) > 0$ and from below if $g''(x_*) < 0$.

Summarizing — for a simple root x_* of f:

- The Newton iteration converges when started from any point x_0 that is "close enough" to the root x_* assuming $g'(x)$ is continuous in a closed interval containing x_*, because the Newton iteration is a fixed–point iteration with $|g'(x_*)| < 1$
- When the error e_n is sufficiently small that the error equation is accurate, the number of correct digits in x_n approximately doubles at each Newton iteration.
- When the error e_n is sufficiently small that the error equation is accurate, the Newton iteration converges from one side.

Note, the point x_0 that is "close enough" to guarantee convergence in the first point above may not provide "an error e_n that is sufficiently small that the error equation is accurate" in the second and third points above. The following example illustrates this matter.

What happens if f has a multiple root? Let's consider a simple case when $f(x) = (x - x_*)^r F(x)$ where $F(x_*) \neq 0$ and $r > 1$. Then, differentiating $f(x)$ twice and inserting the results into the formula $g'(x) = \dfrac{f(x)f''(x)}{f'(x)^2}$ derived above, we get $g'(x) \approx \dfrac{r-1}{r}$ when $(x - x_*)$ is small. So, at any point x close enough to the root x_*, we have $g'(x) > 0$ and $|g'(x)| < 1$, implying one sided linear convergence. Even though we have chosen to simplify the analysis by making a special choice of $f(x)$, our conclusions are correct more generally.

Example 6.3.3. Consider the equation $f(x) \equiv x^{20} - 1 = 0$ with real roots ± 1. If we use the Newton iteration the formula is

$$x_{n+1} = x_n - \frac{x_n^{20} - 1}{20x_n^{19}}$$

Starting with $x_0 = 0.5$, which we might consider to be "close" to the root $x_* = 1$ we compute $x_1 = 26213$, $x_2 = 24903$, $x_3 = 23658$ to five digits. The first iteration takes an enormous leap across the root then the iterations "creep" back towards the root. So, we didn't start "close enough" to get quadratic convergence; the iteration is converging linearly. If, instead, we start from $x_0 = 1.5$, to five digits we compute $x_1 = 1.4250$, $x_2 = 1.3538$ and $x_3 = 1.2863$, and convergence is again linear. If we start from $x_0 = 1.1$, to five digits we compute $x_1 = 1.0532$, $x_2 = 1.0192$, $x_3 = 1.0031$, and $x_4 = 1.0001$. We observe quadratic convergence in this last set of iterates.

Problem 6.3.1. *Construct the Newton iteration by deriving the tangent equation to the curve $y = f(x)$ at the point $(x_n, f(x_n))$ and choosing the point x_{n+1} as the place where this line crosses the x-axis. (See Fig. 6.10.)*

Problem 6.3.2. *Assuming that the root x_* of f is simple, show that $g''(x_*) = \dfrac{f''(x_*)}{f'(x_*)}$. Hence, show that*

$$\frac{e_{n+1}}{e_n^2} \approx \frac{f''(x_*)}{2f'(x_*)}$$

Problem 6.3.3. *For the function $f(x) = x^3 - x - 1$, write down the Newton iteration function g and calculate $g'(x)$ and $g''(x)$. Table 6.6 gives a sequence of iterates for this Newton iteration function. Do the computed ratios behave in the way that you would predict?*

Problem 6.3.4. *Consider computing the root x_* of the equation $x = \cos(x)$, that is computing the value x_* such that $x_* = \cos(x_*)$. Use a calculator for the computations required below.*

1. *Use the iteration $x_{n+1} = \cos(x_n)$ starting from $x_0 = 1.0$ and continuing until you have computed x_* correctly to three digits. Present your results in a table. Observe that the iteration converges. Is convergence from one side? Viewing the iteration as being of the form $x_{n+1} = g(x_n)$, by analyzing the iteration function g show that you would expect the iteration to converge and that you can predict the convergence behavior.*

2. *Now use a Newton iteration to compute the root x_* of $x = \cos(x)$ starting from the same value of x_0. Continue the iteration until you obtain the maximum number of digits of accuracy available on your calculator. Present your results in a table, marking the number of correct digits in each iterate. How would you expect the number of correct digits in successive iterates to behave? Is that what you observe? Is convergence from one side? Is that what you would expect and why?*

Problem 6.3.5. *Some computers perform the division $\dfrac{N}{D}$ by first computing the reciprocal $\dfrac{1}{D}$ and then computing the product $\dfrac{N}{D} = N \cdot \dfrac{1}{D}$. Consider the function $f(x) \equiv D - \dfrac{1}{x}$. What is the root of $f(x)$? Show that, for this function $f(x)$, the Newton iteration function $g(x) = x - \dfrac{f(x)}{f'(x)}$ can be written so that evaluating it does not require divisions. Simplify this iteration function so that evaluating it uses as few adds, subtracts, and multiplies as possible.*

Use the iteration that you have derived to compute $\dfrac{N}{D} = \dfrac{2}{3}$ without using any divisions.

Problem 6.3.6. *Consider the case when $f(x) = (x - x_*)^r F(x)$ where $F(x_*) \neq 0$ and $r > 1$. Show that $g'(x) = \dfrac{f(x)f''(x)}{f'(x)^2} \approx \dfrac{r-1}{r}$ when $(x - x_*)$ is small.*

6.3.2 Secant Iteration

The major disadvantage of the Newton iteration is that the function f' must be evaluated at each iteration. This derivative may be difficult to compute or may be far more expensive to evaluate than the function f. One quite common circumstance where it is difficult to derive f' is when f is defined by a computer program. Recently, the technology of Automatic Differentiation has made computing automatically the values of derivatives defined in such complex ways a possibility but methods that do not require values of $f'(x)$ remain in widespread use.

When the cost of evaluating the derivative is the problem, this cost can often be reduced by using instead the iteration

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

where m_n is an approximation of $f'(x_n)$. For example, we could use a *difference quotient* (recall the discussion in Section 5.1 on approximating derivatives),

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

This difference quotient uses the already–computed values $f(x)$ from the current and the previous iterate. It yields the **secant iteration**:

$$\begin{aligned}
x_{n+1} &= x_n - \frac{f(x_n)}{\left(\dfrac{f(x_n)-f(x_{n-1})}{x_n - x_{n-1}}\right)} \\
&= x_n - \left(\frac{f(x_n)}{f(x_n)-f(x_{n-1})}\right)(x_n - x_{n-1}) \quad \text{[preferred computational form]} \\
&= \frac{x_n f(x_{n-1}) - x_{n-1} f(x_n)}{f(x_{n-1}) - f(x_n)} \quad \text{[computational form to be avoided]}
\end{aligned}$$

This iteration requires two starting values x_0 and x_1. Of the three expressions for the iterate x_{n+1}, the second expression

$$x_{n+1} = x_n - \left(\frac{f(x_n)}{f(x_n)-f(x_{n-1})}\right)(x_n - x_{n-1}), \quad n = 1, 2, \ldots$$

provides the "best" implementation. It is written as a *correction* of the most recent iterate in terms of a scaling of the difference of the current and the previous iterate. All three expressions suffer from cancelation; in the last, cancelation can be catastrophic.

When the error e_n is small enough the secant iteration converges, and it can be shown that the error ultimately behaves like $|e_{n+1}| \approx K|e_n|^p$ where $p = \frac{1+\sqrt{5}}{2} \approx 1.618$ and K is a positive constant. (Derivation of this result is not trivial.) The value $\frac{1+\sqrt{5}}{2}$ is the well-known golden ration whcih we will meet again in the next chapter. Since $p > 1$, the error exhibits *superlinear convergence* but at a rate less than that exemplified by quadratic convergence. However, this superlinear convergence property has a similar effect on the behavior of the error as the quadratic convergence property of the Newton iteration. As long as $f''(x)$ is continuous in an interval containing x_* and the root x_* is simple, we have the following properties:

- If the initial iterates x_0 and x_1 are both sufficiently close to the root x_* then the secant iteration is guaranteed to converge to x_*.
- When the iterates x_n and x_{n-1} are sufficiently close to the root x_* then the secant iteration converges superlinearly; that is, the number of correct digits increases faster than linearly at each iteration. Ultimately the number of correct digits increases by a factor of approximately 1.6 per iteration.

As with the Newton iteration we may get convergence on larger intervals containing x_* than we observe superlinearity.

Example 6.3.4. Consider solving the equation $f(x) \equiv x^3 - x - 1 = 0$ using the secant iteration starting with $x_0 = 1$ and $x_1 = 2$. Observe in Table 6.7 that the errors are tending to zero at an increasing rate but not as fast as in quadratic convergence where the number of correct digits doubles at each iteration. The ratio $|e_i|/|e_{i-1}|^p$ where $p = \frac{1+\sqrt{5}}{2}$ is tending to a limit until limiting precision is encountered.

Problem 6.3.7. *Derive the secant iteration geometrically as follows: Write down the formula for the straight line joining the two points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$, then for the next iterate x_{n+1} choose the point where this line crosses the x-axis. (The figure should be very similar to Fig. 6.10.)*

Problem 6.3.8. *In the three expressions for the secant iteration above, assume that the calculation of $f(x_n)$ is performed just once for each value of x_n. Write down the cost per iteration of each form in terms of arithmetic operations (adds, multiplies, subtracts and divides) and function, f, evaluations.*

n	x_n	e_n	$\lvert e_n\rvert/\lvert e_{n-1}\rvert^p$
0	1.0	-0.32471795724475	–
1	2.0	0.67528204275525	–
2	1.16666666666666	-0.15805129057809	0.298
3	1.25311203319502	-0.07160592404973	1.417
4	1.33720644584166	0.01248848859691	0.890
5	1.32385009638764	-0.00086786085711	1.043
6	1.32470793653209	-0.00001002071266	0.901
7	1.32471796535382	0.00000000810907	0.995
8	1.32471795724467	-0.00000000000008	0.984
9	1.32471795724475	-0.00000000000000	–

Table 6.7: Secant iterates, errors and error ratios

6.3.3 Bisection

The methods described so far choose an initial estimate for the root of $f(x)$, then use an iteration that converges in appropriate circumstances. However, normally there is no guarantee that these iterations will converge from an arbitrary initial estimate. Here we discuss iterations where the user must supply more information but the resulting iteration is guaranteed to converge.

The interval $[a, b]$ is a **bracket for a root** of $f(x)$ if $f(a)\cdot f(b) \leq 0$. Clearly, when f is continuous a bracket contains a root of $f(x)$. For, if $f(a) \cdot f(b) = 0$, then either a or b or both is a root of $f(x) = 0$. And, if $f(a) \cdot f(b) < 0$, then $f(x)$ starts with one sign at a and ends with the opposite sign at b, so by continuity $f(x) = 0$ must have a root between a and b. (Mathematically, this last argument appeals to the Intermediate Value Theorem, see Problem 6.1.1.)

Let $[a, b]$ be an initial bracket for a root of $f(x)$. The bisection method refines this bracket (that is, makes it smaller) as follows. Let $m = \dfrac{a+b}{2}$; this divides the interval $[a, b]$ into two subintervals $[a, m]$ and $[m, b]$ each of half of the length of $[a, b]$. We know that at least one of these subintervals must be a bracket for a root (see Problem 6.3.10). If $f(a) \cdot f(m) \leq 0$, then there is a root in $[a, m]$, and this is a new (smaller) bracket of the root. If $f(m) \cdot f(b) \leq 0$, then there is a root in $[m, b]$, and this is a new (smaller) bracket of the root. Thus, one iteration of the bisection process refines an initial bracket $[a, b]$ to a new bracket $[a_{\text{new}}, b_{\text{new}}]$ of half the length. This process can be repeated with the newly computed bracket, until it becomes sufficiently small. Fig. 6.11 illustrates one iteration of the bisection method.

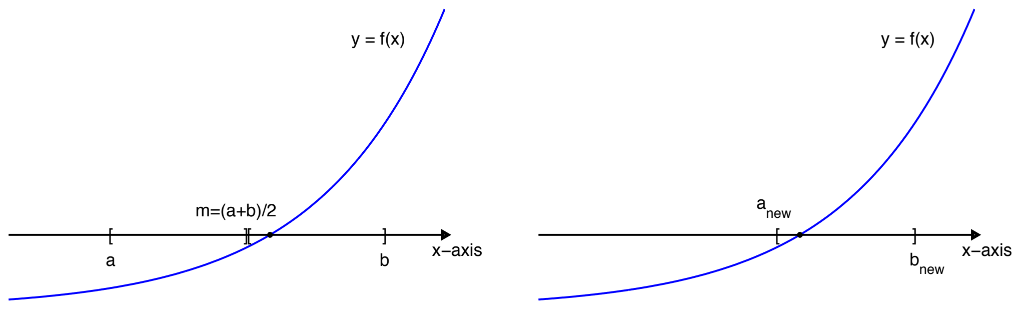

Figure 6.11: Illustration of the bisection method. The left plot shows the initial bracket, $[a, b]$, and the midpoint of this interval, $m = (a + b)/2$. Since the root is in $[m, b]$, the refined bracket is $[a_{\text{new}}, b_{\text{new}}]$, where $a_{\text{new}} := m$ and $b_{\text{new}} := b$.

The simple approach used by the bisection method means that once we determine an initial bracket, the method is guaranteed to converge. Although the Newton and secant iterations require the initial guess to be "close enough" to the root in order to guarantee convergence, there is no such requirement here. The end points of the initial bracket can be very far from the root – as long as the function is continuous, the bisection method will converge.

The disadvantage of bisection is that when the Newton and secant iterations converge, they do so much more quickly than bisection. To explain the convergence behavior of bisection, we let $[a_n, b_n]$ denote the bracket at the nth iteration, and we define the approximation of the root to be the midpoint of this interval, $x_n = m_n = (a_n + b_n)/2$. It is difficult to get a precise formula for the error, but we can say that because bisection simply cuts the interval in half at each iteration, the magnitude of the error, $|e_n| = |x_n - x_*|$, is *approximately* halved at each iteration. Note that it is possible that x_{n-1} (the midpoint of $[a_{n-1}, b_{n-1}]$) is closer to the root than is x_n (the midpoint of $[a_n, b_n]$), so the error does not necessarily decrease at each iteration. Such a situation can be observed in Fig. 6.11, where the midpoint of $[a, b]$ is closer to the root than is the midpoint of $[a_{\text{new}}, b_{\text{new}}]$. However, since the bracket is being refined, bisection eventually converges, and on average we can expect that $\frac{|e_n|}{|e_{n-1}|} \approx \frac{1}{2}$. Thus, on average, the bisection method converges linearly to a root. Note that this implies that $|e_n| \approx 2^{-n}$, or equivalently

$$-\log_2(|e_n|) \approx n\,.$$

Thus, the (approximate) linear convergence of bisection can be illustrated by showing that the points $(n, -\log_2(|e_n|))$ lie essentially on a straight line of slope one (see Example 6.3.5 and Problem 6.3.12).

Example 6.3.5. Consider solving the equation $f(x) \equiv x^3 - x - 1 = 0$ using the bisection method starting with the initial bracket $[a_0, b_0]$ where $a_0 = 1$ and $b_0 = 2$ and terminating when the length of the bracket $[a_n, b_n]$ is smaller than 10^{-4}. Observe in Table 6.8 that the errors are tending to zero but irregularly. The ratio e_n/e_{n-1} is not tending to a limit. However, Fig. 6.12 illustrates the approximate linear convergence rate, since the points $(n, -\log_2(|e_n|))$ lie essentially on a straight line of slope one.

n	x_n	e_n	e_n/e_{n-1}
0	1.5000000000000000	0.1752820427552499	–
1	1.2500000000000000	-0.0747179572447501	-0.426
2	1.3750000000000000	0.0502820427552499	-0.673
3	1.3125000000000000	-0.0122179572447501	-0.243
4	1.3437500000000000	0.0190320427552499	-1.558
5	1.3281250000000000	0.0034070427552499	0.179
6	1.3203125000000000	-0.0044054572447501	-1.293
7	1.3242187500000000	-0.0004992072447501	0.113
8	1.3261718750000000	0.0014539177552499	-2.912
9	1.3251953125000000	0.0004773552552499	0.328
10	1.3247070312500000	-0.0000109259947501	-0.023
11	1.3249511718750000	0.0002332146302499	-21.345
12	1.3248291015625000	0.0001111443177499	0.477
13	1.3247680664062500	0.0000501091614999	0.451

Table 6.8: Bisection iterates, errors and error ratios for Example 6.3.5. The initial bracket was taken to be $[a_0, b_0] = [1, 2]$, and so the initial estimate of the root is $x_0 = 1.5$.

Problem 6.3.9. *Consider using the bisection method to compute the root of $f(x) = x - e^{-x}$ starting with initial bracket $[0, 1]$ and finishing with a bracket of length 2^{-10}. How many iterations will be needed?*

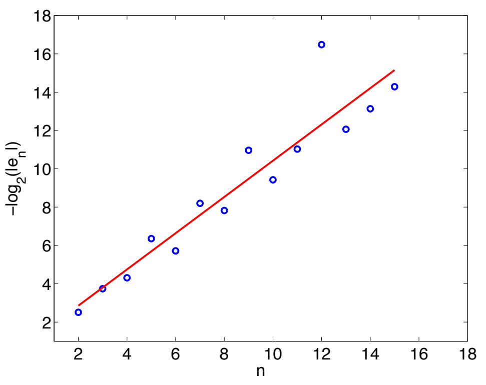

Figure 6.12: Plot of $-\log_2(|e_n|)$ versus n for Example 6.3.5. The points $(n, -\log_2(|e_n|))$ are marked by circles, and the solid line has slope one showing the approximate linear behavior of the points.

Problem 6.3.10. *Let the interval $[a, b]$ be a bracket for a root of the function f, and let $m = \dfrac{a+b}{2}$ be the midpoint of $[a, b]$. Show that if neither $[a, m]$ nor $[m, b]$ is a bracket, then neither is $[a, b]$. [Hint: If neither $[a, m]$ nor $[m, b]$ is a bracket, then $f(a) \cdot f(m) > 0$ and $f(m) \cdot f(b) > 0$. Conclude that $f(a) \cdot f(b) > 0$; that is, conclude that $[a, b]$ is not a bracket.] Now, proceed to use this contradiction to show that one of the intervals $[a, m]$ and $[m, b]$ must be a bracket for a root of the function f.*

Problem 6.3.11. *(Boundary Case) Let $[a, b]$ be a bracket for a root of the function f. In the definition of a bracket, why is it important to allow for the possibility that $f(a)\cdot f(b) = 0$? In what circumstance might we have $f(a)\cdot f(b) = 0$ without having either $f(a) = 0$ or $f(b) = 0$? Is this circumstance likely to occur in practice? How would you modify the algorithm to avoid this problem?*

Problem 6.3.12. *Let $\{[a_n, b_n]\}_{n=0}^{\infty}$ be the sequence of intervals (brackets) produced when bisection is applied to the function $f(x) \equiv x - e^{-x}$ starting with the initial interval (bracket) $[a_0, b_0] = [0, 1]$. Let x_* denote the root of $f(x)$ in the interval $[0, 1]$, and let $e_n = x_n - x_*$, where $x_n = (a_n + b_n)/2$. Fig. 6.13 depicts a plot of $-\log_2(e_n)$ versus n. Show that this plot is consistent with the assumption that, approximately, $e_n = c \cdot 2^{-n}$ for some constant c.*

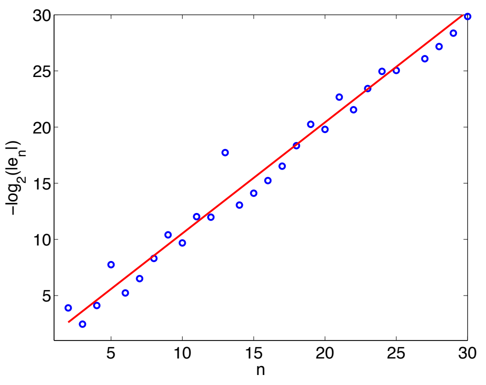

Figure 6.13: Plot of $-\log_2(|e_n|)$ versus n for Problem 6.3.12. The points $(n, -\log_2(|e_n|))$ are marked by circles, and the solid line of slope 1 shows the approximate linear behavior of the points.

6.3.4 Quadratic Inverse Interpolation

A rather different approach to root finding is to use inverse interpolation. The idea is based on the observation that if f has an inverse function, f^{-1}, then

$$f(x) = y \quad \Leftrightarrow \quad x = f^{-1}(y)\,.$$

This relationship implies that if $f^{-1}(y)$ is available, then the root x_* of $f(x) = 0$ is easily computed by evaluating $x_* = f^{-1}(0)$. Unfortunately, it may be very difficult, and in most cases impossible, to find a closed form expression for $f^{-1}(y)$. Instead of attempting to construct it explicitly, we could try to construct an approximation using an interpolating polynomial, $p(y) \approx f^{-1}(y)$, and then use $p(0)$ as an approximation of x_*. But what degree polynomial should we use, and how do we choose the interpolation points to produce a good approximation? Here, we discuss the most commonly used variant based on *quadratic* inverse interpolation, where the interpolation points are constructed iteratively.

To describe the method, consider a case where we have three points x_{n-2}, x_{n-1} and x_n and we have evaluated the function f at each of these points giving the values $y_{n-i} = f(x_{n-i})$, $i = 0, 1, 2$. Assume that the function f has an inverse function f^{-1}, so that $x_{n-i} = f^{-1}(y_{n-i})$, $i = 0, 1, 2$. Now, make a quadratic (Lagrange form) interpolating function to this inverse data:

$$\begin{aligned} p_2(y) \;=\; & \frac{(y-y_{n-1})(y-y_{n-2})}{(y_n-y_{n-1})(y_n-y_{n-2})} f^{-1}(y_n) \\ & + \frac{(y-y_n)(y-y_{n-2})}{(y_{n-1}-y_n)(y_{n-1}-y_{n-2})} f^{-1}(y_{n-1}) \\ & + \frac{(y-y_{n-1})(y-y_n)}{(y_{n-2}-y_{n-1})(y_{n-2}-y_n)} f^{-1}(y_{n-2})\,. \end{aligned}$$

Next, evaluate this expression at $y = 0$ giving

$$\begin{aligned} x_{n+1} = p_2(0) \;=\; & \frac{y_{n-1}y_{n-2}}{(y_n-y_{n-1})(y_n-y_{n-2})} x_n \\ & + \frac{y_n y_{n-2}}{(y_{n-1}-y_n)(y_{n-1}-y_{n-2})} x_{n-1} \\ & + \frac{y_{n-1}y_n}{(y_{n-2}-y_{n-1})(y_{n-2}-y_n)} x_{n-2}\,. \end{aligned}$$

Discard the point x_{n-2} and continue with the same process to compute x_{n+2} using the "old" points x_{n-1} and x_n, and the new point x_{n+1}. Proceeding in this way, we obtain an iteration reminiscent of the secant iteration for which the error behaves approximately like $e_{n+1} = K e_n^{1.44}$. Note that this rate of convergence occurs only when all the x_i currently involved in the formula are close enough to the root x_* of the equation $f(x) = 0$; that is, in the limit quadratic inverse interpolation is a superlinearly convergent method.

Problem 6.3.13. *Derive the formula for the error in interpolation associated with the interpolation formula for $p_2(y)$. Write the error term in terms of derivatives of $f(x)$ rather than of $f^{-1}(y)$, recalling that $\left(f^{-1}\right)'(y) = \dfrac{1}{f'(x)}$.*

6.4 Calculating Square Roots

How does a computer or a calculator compute the square root of a positive real number? In fact, how did engineers calculate square roots before computers were invented? There are many approaches, one dating back to the ancient Babylonians! For example, we could use the identity

$$\sqrt{a} = e^{\frac{1}{2}\ln a}$$

and then use tables of logarithms to find approximate values for $\ln(a)$ and e^b. In fact, some calculators still use this identity along with good routines to calculate logarithms and exponentials.

In this section we describe how to apply root finding techniques to calculate square roots. For any real value $a > 0$, the function $f(x) = x^2 - a$ has two square roots, $x_* = \pm\sqrt{a}$. The Newton iteration applied to $f(x)$ gives

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - \frac{x_n^2 - a}{2x_n} \\ &= \frac{\left(x_n + \dfrac{a}{x_n}\right)}{2} \end{aligned}$$

Note that we only need to perform simple addition, multiplication and division operations to compute x_{n+1}. The (last) algebraically simplified expression is less expensive to compute than the original expression because it involves just one divide, one add, and one division by two, whereas the original expression involves a multiply, two subtractions, a division and a multiplication by two. [When using IEEE Standard floating–point arithmetic, multiplication or division by a power of two of a machine floating–point number involves simply increasing or reducing, respectively, that number's exponent by one, known as a binary shift. This is a low cost operation relative to the standard arithmetic operations.]

From the iteration $x_{n+1} = \dfrac{1}{2}\left(x_n + \dfrac{a}{x_n}\right)$ we observe that since $a > 0$, x_{n+1} has the same sign as x_n for all n. Hence, if the Newton iteration converges, it converges to that root with the same sign as x_0.

Now consider the error in the iterates.

$$e_n \equiv x_n - \sqrt{a}$$

Algebraically, we have

$$\begin{aligned} e_{n+1} &= x_{n+1} - \sqrt{a} &&= \frac{\left(x_n + \dfrac{a}{x_n}\right)}{2} - \sqrt{a} \\ &= \frac{(x_n - \sqrt{a})^2}{2x_n} &&= \frac{e_n^2}{2x_n} \end{aligned}$$

From this formula, we see that when $x_0 > 0$ we have $e_1 = x_1 - \sqrt{a} > 0$ so that $x_1 > \sqrt{a} > 0$. It follows similarly that $e_n \geq 0$ for all $n \geq 1$. Hence, if x_n converges to $\sqrt{a}$ then it converges from above. To prove that the iteration converges, note that

$$0 \leq \frac{e_n}{x_n} = \frac{x_n - \sqrt{a}}{x_n - 0} = \frac{\text{distance from } x_n \text{ to } \sqrt{a}}{\text{distance from } x_n \text{ to } 0} < 1$$

for $n \geq 1$, so

$$e_{n+1} = \frac{e_n^2}{2x_n} = \left(\frac{e_n}{x_n}\right)\frac{e_n}{2} < \frac{e_n}{2}$$

Hence, for all $n \geq 1$, the error e_n is reduced by a factor of at least $\dfrac{1}{2}$ at each iteration. Therefore the iterates x_n converge to $\sqrt{a}$. Of course, "quadratic convergence rules!" as this is the Newton iteration, so when the error is sufficiently small the number of correct digits in x_n approximately doubles with each iteration. However, if x_0 is far from the root we may not have quadratic convergence initially, and, at worst, the error may only be approximately halved at each iteration.

Not surprisingly, the Newton iteration is widely used in computers' and calculators' mathematical software libraries as a part of the built-in function for determining square roots. Clearly, for this approach to be useful a method is needed to ensure that, from the start of the iteration, the error is sufficiently small that “quadratic convergence rules!”. Consider the floating–point number written in the form

$$a = S(a) \cdot 2^{e(a)}$$

where the significand, $S(a)$, satisfies $1 \le S(a) < 2$ and the exponent $e(a)$ is an integer. (As we saw in sectionFPnumbers, any normalized positive floating point number may be written this way.) If the exponent $e(a)$ is even then

$$\sqrt{a} = \sqrt{S(a)} \cdot 2^{e(a)/2}$$

Hence, we can compute $\sqrt{a}$ from this expression by computing the square root of the significand $S(a)$ and halving the exponent $e(a)$. Similarly, if $e(a)$ is odd then $e(a)-1$ is even, so $a = \{2S(a)\} \cdot 2^{e(a)-1}$ and

$$\sqrt{a} = \sqrt{2S(a)} \cdot 2^{(e(a)-1)/2}$$

Hence, we can compute $\sqrt{a}$ from this expression by computing the square root of twice the significand, $2S(a)$, and halving the adjusted exponent, $e(a)-1$.

Combining these two cases we observe that the significand is in the range

$$1 \le S(a) < 2 \le 2S(a) < 4$$

So, for all values $a > 0$ computing $\sqrt{a}$ is reduced to simply modifying the exponent and computing $\sqrt{b}$ for a value b in the interval $[1, 4)$; here b is either $S(a)$ or it is $2S(a)$ depending on whether the exponent $e(a)$ of a is even or odd, respectively.

This process of reducing the problem of computing square roots of numbers a defined in the semi-infinite interval $(0, \infty)$ to a problem of computing square roots of numbers b defined in the (small) finite interval [1,4) is a form of **range reduction**. Range reduction is used widely in simplifying the computation of mathematical functions because, generally, it is easier (and more accurate and efficient) to find a good approximation to a function at all points in a small interval than at all points in the original, larger interval.

The Newton iteration to compute $\sqrt{b}$ is

$$X_{n+1} = \frac{\left(X_n + \dfrac{b}{X_n}\right)}{2}, \quad n = 0, 1, \ldots$$

where X_0 is chosen carefully in the interval $[1, 2)$ to give convergence in a small (known) number of iterations. On many computers a **seed table** is used to generate the initial iterate X_0. Given the value of a, the value of $b =(S(a)$ or $2S(a))$ is computed, then the arithmetic unit supplies the leading bits (binary digits) of b to the seed table which returns an estimate for the square root of the number described by these leading bits. This provides the initial estimate X_0 of $\sqrt{b}$. Starting from X_0, the iteration to compute $\sqrt{b}$ converges to machine accuracy in a *known* maximum number of iterations (usually 3 or 4 iterations). The seed table is constructed so the number of iterations needed to achieve machine accuracy is fixed and known for all numbers for which each seed from the table is used as the initial estimate. Hence, there is no need to check whether the iteration has converged, resulting in some additional savings in computational time.

Problem 6.4.1. *Use the above range reduction strategy (based on powers of 2) combined with the Newton iteration for computing each of $\sqrt{5}$ and $\sqrt{9}$. [Hint: $5 = \frac{5}{4} \cdot 2^2$] (In each case compute b then start the Newton iteration for computing $\sqrt{b}$ from $X_0 = 1.0$). Using a calculator, continue the iteration until you have computed the maximum number of correct digits available. Check for (a) the predicted behavior of the iterates and (b) quadratic convergence.*

Problem 6.4.2. *Consider the function $f(x) = x^3 - a$ which has only one real root, namely $x = \sqrt[3]{a}$. Show that a simplified Newton iteration for this function may be derived as follows*

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - \frac{1}{3}\left(\frac{x_n^3 - a}{x_n^2}\right) \\ &= \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right) \end{aligned}$$

Problem 6.4.3. *In the cube root iteration above show that the second (algebraically simplified) expression for the cube root iteration is less expensive to compute than the first. Assume that the value of $\frac{1}{3}$ is precalculated and in the iteration a multiplication by this value is used where needed.*

Problem 6.4.4. *For $a > 0$ and the cube root iteration*

1. *Show that*
$$e_{n+1} = \frac{(2x_n + \sqrt[3]{a})}{3x_n^2} e_n^2$$
when $x_0 > 0$

2. *Hence show that $e_1 \geq 0$ and $x_1 > \sqrt[3]{a}$ and, in general, that $x_n \geq \sqrt[3]{a}$ for all $n \geq 1$*

3. *When $x_n \geq \sqrt[3]{a}$ show that*
$$0 \leq \frac{(2x_n + \sqrt[3]{a})e_n}{3x_n^2} < 1$$

4. *Hence, show that the error is reduced at each iteration and that the Newton iteration converges from above to $\sqrt[3]{a}$ from any initial estimate $x_0 > 0$.*

Problem 6.4.5. *We saw that $\sqrt{x}$ can be computed for any $x > 0$ by range reduction from values of $\sqrt{X}$ for $X \in [1, 4)$. Develop a similar argument that shows how $\sqrt[3]{x}$ can be computed for any $x > 0$ by range reduction from values of $\sqrt[3]{X}$ for $X \in [1, 8)$. What is the corresponding range of values for X for computing $\sqrt[3]{x}$ when $x < 0$?*

Problem 6.4.6. *Use the range reduction derived in the previous problem followed by a Newton iteration to compute each of the values $\sqrt[3]{10}$ starting the iteration from $X_0 = 1.0$, and $\sqrt[3]{-20}$ starting the iteration from $X_0 = -1.0$.*

Problem 6.4.7. *Let $a > 0$. Write down the roots of the function*

$$f(x) = a - \frac{1}{x^2}$$

Write down the Newton iteration for computing the positive root of $f(x) = 0$. Show that the iteration formula can be simplified so that it does not involve divisions. Compute the arithmetic costs per iteration with and without the simplification; recall that multiplications or divisions by two (binary shifts) are less expensive than arithmetic. How would you compute $\sqrt{a}$ from the result of the iteration, without using divisions or using additional iterations?

Problem 6.4.8. *Let $a > 0$. Write down the real root of the function*

$$f(x) = a - \frac{1}{x^3}$$

Write down the Newton iteration for computing the real root of $f(x) = 0$. Show that the iteration formula can be simplified so that it does not involve a division at each iteration. Compute the arithmetic costs per iteration with and without the simplification; identify any parts of the computation that can be computed a priori, *that is they can be computed just once independently of the number of iterations. How would you compute $\sqrt[3]{a}$ from the result of the iteration, without using divisions or using additional iterations?*

6.5 Roots of Polynomials

Specialized versions of the root–finding iterations described in previous sections may be tailored for computing the roots of a polynomial

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

of degree n. These methods exploit the fact that we know precisely the number of roots for a polynomial of exact degree n; namely, there are n roots counting multiplicities. In many cases of interest some, or all, of these roots are real.

In this section we concentrate on how the Newton iteration can be used for this purpose. The following issues need to be considered:

- To use a Newton iteration to find a root of a function f requires values of both $f(x)$ and its derivative $f'(x)$. For a polynomial p_n, it is easy to determine both of these values efficiently. In Section 6.5.1, we describe an algorithm called *Horner's rule* that can be used to evaluate polynomials efficiently, and, in general, accurately.

- Once a root x_* of $p_n(x)$ has been found, we may want to divide the factor $(x - x_*)$ out of $p_n(x)$ to obtain a polynomial $p_{n-1}(x)$ of degree $n-1$. We can then find another root of $p_n(x)$ by computing a root of $p_{n-1}(x)$. (If we don't remove the factor $(x - x_*)$ and we use $p_n(x)$ again then we may converge to x_* once more, and we will not know whether this is a valid root.) In Section 6.5.2, we describe how *synthetic division* can be used for this deflation process.

- Finally in Section 6.5.3 we consider what effect errors have on the computed roots, and in particular, discuss the *conditioning* of the roots of a polynomial.

6.5.1 Horner's Rule

The derivation of Horner's rule begins by noting that a polynomial p_n can be expressed as a sequence of nested multiplies. For example, a cubic polynomial

$$p_3(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

may be written as the following sequence of nested multiplies

$$p_3(x) = ((((a_3)x + a_2)x + a_1)x + a_0)\,.$$

Notice that if we use the first equation to evaluate $p_3(x)$, then we must perform two exponentiations (or equivalently additional multiplications), three multiplications, and three additions. On the other hand, if we use the second equation, then we need only three multiplications and three additions. In general, if $p_n(x)$ is a polynomial of degree n, then by using Horner's rule to evaluate $p_n(x)$ we avoid all exponentiation operations, and require only n multiplications and n additions.

To develop an algorithm for Horner's rule, it is helpful to visualize the nested multiplies as a sequence of operations:

$$\begin{array}{rcrcl} & & t_3(x) & = & a_3 \\ & & t_2(x) & = & a_2 + x \cdot t_3(x) \\ & & t_1(x) & = & a_1 + x \cdot t_2(x) \\ p_3(x) & = & t_0(x) & = & a_0 + x \cdot t_1(x) \end{array}$$

where we use $t_i(x)$ to denote "temporary" functions.

Example 6.5.1. Consider evaluating the cubic

$$\begin{array}{rcl} p_3(x) & = & 2 - 5x + 3x^2 - 7x^3 \\ & = & (2) + x \cdot \{(-5) + x \cdot [(3) + x \cdot (-7)]\} \end{array}$$

at $x = 2$. This nested multiplication form of $p_3(x)$ leads to the following sequence of operations for evaluating $p_3(x)$ at any point x:

$$\begin{array}{rcrcl} & & t_3(x) & = & (-7) \\ & & t_2(x) & = & (3) + x \cdot t_3(x) \\ & & t_1(x) & = & (-5) + x \cdot t_2(x) \\ p_3(x) & \equiv & t_0(x) & = & (2) + x \cdot t_1(x) \end{array}$$

Evaluating this sequence of operations for $x = 2$ gives

$$\begin{array}{rcrcl} & & t_3(x) & = & (-7) \\ & & t_2(x) & = & (3) + 2 \cdot (-7) = -11 \\ & & t_1(x) & = & (-5) + 2 \cdot (-11) = -27 \\ p_3(x) & \equiv & t_0(x) & = & (2) + 2 \cdot (-27) = -52 \end{array}$$

which you may check by evaluating the original cubic polynomial directly.

To efficiently evaluate the derivative $p_3'(x)$ we differentiate the sequence of operations above with respect to x to obtain

$$\begin{array}{rcrcl} & & t_3'(x) & = & 0 \\ & & t_2'(x) & = & t_3(x) + x \cdot t_3'(x) \\ & & t_1'(x) & = & t_2(x) + x \cdot t_2'(x) \\ p_3'(x) & \equiv & t_0'(x) & = & t_1(x) + x \cdot t_1'(x) \end{array}$$

Merging alternate lines of the derivative sequence with that for the polynomial sequence produces a further sequence for evaluating the cubic $p_3(x)$ and its derivative $p_3'(x)$ simultaneously, namely

$$\begin{array}{rcrcl} & & t_3'(x) & = & 0 \\ & & t_3(x) & = & a_3 \\ & & t_2'(x) & = & t_3(x) + x \cdot t_3'(x) \\ & & t_2(x) & = & a_2 + x \cdot t_3(x) \\ & & t_1'(x) & = & t_2(x) + x \cdot t_2'(x) \\ & & t_1(x) & = & a_1 + x \cdot t_2(x) \\ p_3'(x) & \equiv & t_0'(x) & = & t_1(x) + x \cdot t_1'(x) \\ p_3(x) & \equiv & t_0(x) & = & a_0 + x \cdot t_1(x) \end{array}$$

Example 6.5.2. Continuing with Example 6.5.1

$$\begin{array}{rcrcl} & & t_3'(x) & = & 0 \\ & & t_3(x) & = & (-7) \\ & & t_2'(x) & = & t_3(x) + x \cdot t_3'(x) \\ & & t_2(x) & = & (3) + x \cdot t_3(x) \\ & & t_1'(x) & = & t_2(x) + x \cdot t_2'(x) \\ & & t_1(x) & = & (-5) + x \cdot t_2(x) \\ p_3'(x) & \equiv & t_0'(x) & = & t_1(x) + x \cdot t_1'(x) \\ p_3(x) & \equiv & t_0(x) & = & (2) + x \cdot t_1(x) \end{array}$$

and evaluating at $x = 2$ we have

$$\begin{array}{rcrcl} & & t_3'(x) & = & 0 \\ & & t_3(x) & = & (-7) \\ & & t_2'(x) & = & (-7) + 2 \cdot 0 = -7 \\ & & t_2(x) & = & (3) + 2 \cdot (-7) = -11 \\ & & t_1'(x) & = & (-11) + 2 \cdot (-7) = -25 \\ & & t_1(x) & = & (-5) + 2 \cdot (-11) = -27 \\ p_3'(x) & \equiv & t_0'(x) & = & (-27) + 2 \cdot (-25) = -77 \\ p_3(x) & \equiv & t_0(x) & = & (2) + 2 \cdot (-27) = -52 \end{array}$$

Again you may check these values by evaluating the original polynomial and its derivative directly at $x = 2$.

Since the current temporary values $t_i'(x)$ and $t_i(x)$ are used to determine only the next temporary values $t_{i-1}'(x)$ and $t_{i-1}(x)$, we can write the new values $t_{i-1}'(x)$ and $t_{i-1}(x)$ over the old values $t_i'(x)$ and $t_i(x)$ as follows

$$\begin{array}{rcrcl} & & tp & = & 0 \\ & & t & = & a_3 \\ & & tp & = & t + x \cdot tp \\ & & t & = & a_2 + x \cdot t \\ & & tp & = & t + x \cdot tp \\ & & t & = & a_1 + x \cdot t \\ p_3'(x) & \equiv & tp & = & t + x \cdot tp \\ p_3(x) & \equiv & t & = & a_0 + x \cdot t \end{array}$$

For a polynomial of degree n the regular pattern of this code segment is captured in the pseudocode in Fig. 6.14. (We could program round the multiplication of the third line above since we know that the result is zero.)

Problem 6.5.1. *Use Horner's rule to compute the first two iterates x_1 and x_2 in Table 6.6.*

Problem 6.5.2. *Write out Horner's rule for evaluating a quartic polynomial*

$$p_4(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

and its derivative $p_4'(x)$. For the polynomial $p_4(x) = 3x^4 - 2x^2 + 1$, compute $p_4(-1)$ and $p_4'(-1)$ using Horner's scheme.

Problem 6.5.3. *What is the cost in arithmetic operations of a single Newton iteration when computing a root of the polynomial $p_n(x)$ using the code segment in Fig. 6.14 to compute both of the values $p_n(x)$ and $p_n'(x)$.*

Horner's Rule to evaluate $p_n(x)$ and $p'_n(x)$

```
Input:   coefficients, $a_i$ of $p(x)$
         scalar, $x$
Output:  $t = p_n(x)$ and $tp = p'_n(x)$

  $tp := 0$
  $t := a_n$
  for $i = n - 1$ downto 0 do
      $tp := t + x * tp$
      $t := a_i + x * t$
  next $i$
```

Figure 6.14: Pseudocode to implement Horner's rule for simultaneously evaluating $p_n(x)$ and $p'_n(x)$

Problem 6.5.4. *Modify the code segment computing the values $p_n(x)$ and $p'_n(x)$ in Fig. 6.14 so that it also computes the value of the second derivative $p''_n(x)$. (Fill in the assignments to* tpp *below.)*

```
$tpp :=?$
$tp := 0$
$t := a_n$
for $i = n - 1$ downto 0 do
    $tpp :=?$
    $tp := t + x * tp$
    $t := a_i + x * t$
next $i$
$p''_n(x) := tpp$
$p'_n(x) := tp$
$p_n(x) := t$
```

6.5.2 Synthetic Division

Consider a cubic polynomial p_3 and assume that we have an approximation α to a root x_* of $p_3(x)$. In synthetic division, we divide the polynomial

$$p_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

by the linear factor $x - \alpha$ to produce a quadratic polynomial

$$q_2(x) = b_3x^2 + b_2x + b_1$$

as the quotient and a constant b_0 as the remainder. The defining relation is

$$\frac{p_3(x)}{x-\alpha} = q_2(x) + \frac{b_0}{x-\alpha}$$

To determine the constant b_0, and the coefficients b_1, b_2 and b_3 in $q_2(x)$, we write this relation in the form

$$p_3(x) = (x - \alpha)q_2(x) + b_0$$

Now we substitute in this expression for q_2 and expand the product $(x-\alpha)q_2(x)+b_0$ as a sum of powers of x. As this must equal $p_3(x)$ we write the two power series together:

$$\begin{array}{rclllllll} (x-\alpha)q_2(x)+b_0 & = & (b_3)x^3 & + & (b_2-\alpha b_3)x^2 & + & (b_1-\alpha b_2)x & + & (b_0-\alpha b_1) \\ p_3(x) & = & a_3x^3 & + & a_2x^2 & + & a_1x & + & a_0 \end{array}$$

The left hand sides of these two equations must be equal for any value x, so we can equate the coefficients of the powers x^i on the right hand sides of each equation (here α is considered to be fixed). This gives

$$\begin{array}{rcl} a_3 & = & b_3 \\ a_2 & = & b_2-\alpha b_3 \\ a_1 & = & b_1-\alpha b_2 \\ a_0 & = & b_0-\alpha b_1 \end{array}$$

Solving for the unknown values b_i, we have

$$\begin{array}{rcl} b_3 & = & a_3 \\ b_2 & = & a_2+\alpha b_3 \\ b_1 & = & a_1+\alpha b_2 \\ b_0 & = & a_0+\alpha b_1 \end{array}$$

We observe that $t_i(\alpha)=b_i$; that is, Horner's rule and synthetic division are different interpretations of the same relation. The intermediate values $t_i(\alpha)$ determined by Horner's rule are the coefficients b_i of the polynomial quotient $q_2(x)$ that is obtained when $p_3(x)$ is divided by $(x-\alpha)$. The result b_0 is the value $t_0(\alpha)=p_3(\alpha)$ of the polynomial $p_3(x)$ when it is evaluated at the point $x=\alpha$. So, when α is a root of $p_3(x)=0$, the value of $b_0=0$.

Synthetic division for a general polynomial $p_n(x)$, of degree n in x, proceeds analogously. Dividing $p_n(x)$ by $(x-\alpha)$ produces a quotient polynomial $q_{n-1}(x)$ of degree $n-1$ and a constant b_0 as remainder. So,

$$\frac{p_n(x)}{x-\alpha}=q_{n-1}(x)+\frac{b_0}{x-\alpha}$$

where

$$q_{n-1}(x)=b_nx^{n-1}+b_{n-1}x^{n-2}+\cdots+b_1$$

Since

$$p_n(x)=(x-\alpha)q_{n-1}(x)+b_0,$$

we find that

$$\begin{array}{rclllllll} (x-\alpha)q_{n-1}(x)+b_0 & = & b_nx^n & + & (b_{n-1}-\alpha b_n)x^{n-1} & + & \cdots & + & (b_0-\alpha b_1) \\ p_n(x) & = & a_nx^n & + & a_{n-1}x^{n-1} & + & \cdots & + & a_0 \end{array}$$

Equating coefficients of the powers x^i,

$$\begin{array}{rcl} a_n & = & b_n \\ a_{n-1} & = & b_{n-1}-\alpha b_n \\ & \vdots & \\ a_0 & = & b_0-\alpha b_1 \end{array}$$

Solving these equations for the constant b_0 and the unknown coefficients b_i of the polynomial $q_{n-1}(x)$ leads to

$$\begin{array}{rcl} b_n & = & a_n \\ b_{n-1} & = & a_{n-1}+\alpha b_n \\ & \vdots & \\ b_0 & = & a_0+\alpha b_1 \end{array}$$

How does the synthetic division algorithm apply to the problem of finding the roots of a polynomial? First, from its relationship to Horner's rule, for a given value of α synthetic division efficiently determines the value of the polynomial $p_n(\alpha)$ (and also of its derivative $p_n'(\alpha)$ using the technique described in the pseudocode given in Fig. 6.14). Furthermore, suppose that we have found a good approximation to the root, $\alpha \approx x_*$. When the polynomial $p_n(x)$ is evaluated at $x = \alpha$ via synthetic division the remainder $b_0 \approx p_n(x_*) = 0$. We treat the remainder b_0 as though it is precisely zero. Therefore, approximately

$$p_n(x) = (x - x_*)q_{n-1}(x)$$

where the quotient polynomial q_{n-1} has degree one less than the polynomial p_n. The polynomial q_{n-1} has as its $(n-1)$ roots values that are close to each of the remaining $(n-1)$ roots of $p_n(x) = 0$. We call q_{n-1} the *deflated polynomial* obtained by removing a linear factor $(x - \alpha)$ approximating $(x - x_*)$ from the polynomial p_n. This process therefore reduces the problem of finding the roots of $p_n(x) = 0$ to

- Find one root α of the polynomial $p_n(x)$
- Deflate the polynomial p_n to derive a polynomial of one degree lower, q_{n-1}
- Repeat this procedure with q_{n-1} replacing p_n

Of course, this process can be "recursed" to find all the real roots of a polynomial equation. If there are only real roots, eventually we reach a polynomial of degree two. Then, we may stop and use the quadratic formula to "finish off".

To use this approach to find all the real *and* all the complex roots of a polynomial equation, we must use complex arithmetic in both the Newton iteration and in the synthetic division algorithm, and we must choose starting estimates for the Newton iteration that lie in the complex plane; that is, the initial estimates x_0 must not be on the real line. Better, related approaches exist which exploit the fact that, for a polynomial with real coefficients, the complex roots arise in conjugate pairs. The two factors corresponding to each complex conjugate pair taken together correspond to an irreducible quadratic factor of the original polynomial. These alternative approaches aim to compute these irreducible quadratic factors directly, in real arithmetic, and each irreducible quadratic factor may then be factored using the quadratic formula.

Example 6.5.3. Consider the quartic polynomial

$$p_4(x) = x^4 - 5x^2 + 4$$

which has roots ± 1 and ± 2. Assume that we have calculated the root $\alpha = -2$ and that we use the synthetic division algorithm to deflate the polynomial $p_4(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ to compute the cubic polynomial $q_3(x) = b_4x^3 + b_3x^2 + b_2x + b_1$ and the constant b_0. The algorithm is

$$\begin{array}{rcl} b_4 & = & a_4 \\ b_3 & = & a_3 + \alpha b_4 \\ b_2 & = & a_2 + \alpha b_3 \\ b_1 & = & a_1 + \alpha b_2 \\ b_0 & = & a_0 + \alpha b_1 \end{array}$$

Substituting the values of the coefficients of the quartic polynomial p_4 and evaluating at $\alpha = -2$ we compute

$$\begin{array}{rcl} b_4 & = & (1) \\ b_3 & = & (0) + (-2)(1) = -2 \\ b_2 & = & (-5) + (-2)(-2) = -1 \\ b_1 & = & (0) + (-2)(-1) = 2 \\ b_0 & = & (4) + (-2)(2) = 0 \end{array}$$

So, $b_0 = 0$ as it should be since $\alpha = -2$ is exactly a root and we have

$$q_3(x) = (1)x^3 + (-2)x^2 + (-1)x + (2)$$

You may check that $q_3(x)$ has the remaining roots ± 1 and 2 of $p_4(x)$.

Example 6.5.4. We'll repeat the previous example but using the approximate root $\alpha = -2.001$ and working to four significant digits throughout. Substituting this value of α we compute

$$\begin{array}{rcl} b_4 & = & (1) \\ b_3 & = & (0) + (-2.001)(1) = -2.001 \\ b_2 & = & (-5) + (-2.001)(-2.001) = -0.9960 \\ b_1 & = & (0) + (-2.001)(-0.9960) = 1.993 \\ b_0 & = & (4) + (-2.001)(1.993) = 0.0120 \end{array}$$

Given that the error in α is only 1 in the fourth significant digit, the value of b_0 is somewhat larger than we might expect. If we assume that $\alpha = -2.001$ is close enough to a root, which is equivalent to assuming that $b_0 = 0$, the reduced cubic polynomial is $q_3(x) = x^3 - 2.001x^2 - 0.996x + 1.993$. Using MATLAB, to four significant digits the cubic polynomial $q_3(x)$ has roots 2.001, 0.998 and -0.998, approximating the roots 2, 1 and -1 of $p_4(x)$, respectively. These are somewhat better approximations to the true roots of the original polynomial than one might anticipate from the size of the discarded remainder b_0.

Problem 6.5.5. *The cubic polynomial $p_3(x) = x^3 - 7x + 6$ has roots 1, 2 and -3. Deflate p_3 by synthetic division to remove the root $\alpha = -3$. This will give a quadratic polynomial q_2. Check that $q_2(x)$ has the appropriate roots. Repeat this process but with an approximate root $\alpha = -3.0001$ working to five significant digits throughout. What are the roots of the resulting polynomial $q_2(x)$.*

Problem 6.5.6. *Divide the polynomial $q_{n-1}(x) = b_n x^{n-1} + b_{n-1}x^{n-2} + \cdots + b_1$ by the linear factor $(x - \alpha)$ to produce a polynomial quotient $r_{n-2}(x)$ and remainder c_1*

$$\frac{q_{n-1}(x)}{x-\alpha} = r_{n-2}(x) + \frac{c_1}{x-\alpha}$$

where $r_{n-2}(x) = c_n x^{n-2} + c_{n-1}x^{n-3} + \cdots + c_2$.

(a) Solve for the unknown coefficients c_i in terms of the values b_i.

(b) Derive a code segment, a loop, that interleaves the computations of the b_i's and c_i's.

Problem 6.5.7. *By differentiating the synthetic division relation $p_n(x) = (x-\alpha)q_{n-1}(x) + b_0$ with respect to the variable x, keeping the value of α fixed, and equating the coefficients of powers of x on both sides of the differentiated relation derive Horner's algorithm for computing the value of the derivative $p_n'(x)$ at any point $x = \alpha$.*

6.5.3 Conditioning of Polynomial Roots

So far, we have considered calculating the roots of a polynomial by repetitive iteration and deflation, and we have left the impression that both processes are simple and safe although the computation of Example 6.5.4 should give us pause for thought. Obviously there are errors when the iteration is not completed, that is the root is not computed exactly, and these errors are clearly propagated into the deflation process which itself introduces further errors into the remaining roots. The overall process is difficult to analyze but a simpler question which we can in part answer is whether the computed polynomial roots are sensitive to these errors. Particularly we might ask the closely related question "Are the polynomial roots sensitive to small changes in the coefficients of the polynomial?" That is, is polynomial root finding an inherently ill–conditioned process? In many applications one algorithmic choice is to reduce a "more complicated" computational problem to one of computing the roots of a high degree polynomial. If the resulting problem is likely to be ill–conditioned, this approach should be considered cautiously.

Consider a polynomial

$$p_n(x) = a_0 + a_1 x + \cdots + a_n x^n \,.$$

Assume that the coefficients of this polynomial are perturbed to give the polynomial

$$P_n(x) = A_0 + A_1 x + \cdots + A_n x^n \,.$$

Let z_0 be a root of $p_n(x)$ and Z_0 be the "corresponding" root of $P_n(x)$; usually, for sufficiently small perturbations of the coefficients a_i there is such a correspondence. By Taylor series

$$P_n(z_0) \approx P_n(Z_0) + (z_0 - Z_0)P_n'(Z_0)$$

and so, since $p_n(z_0) = 0$,

$$P_n(z_0) - p_n(z_0) \approx P_n(Z_0) + (z_0 - Z_0)P_n'(Z_0)\,.$$

Observing that $P_n(Z_0) = 0$, we have

$$z_0 - Z_0 \approx \frac{\sum_{i=0}^{n}(A_i - a_i)z_0^i}{P_n'(Z_0)}\,.$$

So, the error in the root can be large when $P_n'(Z_0) \approx 0$ such as occurs when Z_0 is close to a cluster of roots of $P_n(x)$. We observe that P_n is likely to have a cluster of at least m roots whenever p_n has a multiple root of multiplicity m. [Recall that at a multiple root α of $P_n(x)$ we have $P_n'(\alpha) = 0$.]

One way that the polynomial P_n might arise is essentially directly. That is, if the coefficients of the polynomial p_n are not representable as computer numbers at the working precision then they are rounded; that is, the difference $|a_i - A_i| \approx \max\{|a_i|, |A_i|\}\epsilon_{\text{WP}}$. In this case, a reasonable estimate is

$$z_0 - Z_0 \approx \frac{P_n(z_0)}{P_n'(Z_0)}\epsilon_{\text{WP}}$$

so that even small perturbations that arise in the storage of computer numbers can lead to a significantly sized error. The other way $P_n(x)$ might arise is implicitly as a consequence of *backward error analysis.* For some robust algorithms for computing roots of polynomials, we can show that the roots computed are the roots of a polynomial $P_n(x)$ whose coefficients are usually close to the coefficients of $p_n(x)$

Example 6.5.5. Consider the following polynomial of degree 12:

$$\begin{aligned} p_{12}(x) &= (x-1)(x-2)^2(x-3)(x-4)(x-5)(x-6)^2(x-7)^2(x-8)(x-9) \\ &= x^{12} - 60x^{11} + 1613x^{10} - 25644x^9 + \cdots \end{aligned}$$

If we use, for example, MATLAB (see Section 6.6) to compute the roots of the power series form of $p_{12}(x)$, we obtain the results shown in Table 6.9. Observe that each of the double roots 2, 6 and 7 of the polynomial $p_{12}(x)$ is approximated by a pair of close real roots near 2, 6 and 7. Also note that the double roots are approximated much less accurately than are the simple roots of similar size, and the larger roots are calculated somewhat less accurately than the smaller ones.

Example 6.5.5 and the preceding analysis indicates that multiple and clustered roots are likely to be ill–conditioned, though this does not imply that isolated roots of the same polynomial are well–conditioned. The numerator in the expression for $z_0 - Z_0$ contains "small" coefficients $a_i - A_i$ by definition so, at first glance, it seems that a near-zero denominator gives rise to the only ill–conditioned case. However, this is not so; as a famous example due to Wilkinson shows, the roots of a polynomial of high degree may be ill–conditioned even when all its roots are well–spaced.

exact root	computed root
9	9.00000000561095
8	7.99999993457306
7	7.00045189374144
7	6.99954788307491
6	6.00040397500841
6	5.99959627548739
5	5.00000003491181
4	3.99999999749464
3	3.00000000009097
2	2.00000151119609
2	1.99999848881036
1	0.999999999999976

Table 6.9: Roots of the polynomial $p_{12}(x) = (x-1)(x-2)^2(x-3)(x-4)(x-5)(x-6)^2(x-7)^2(x-8)(x-9)$. The left column shows the exact roots, and the right column shows the computed roots of the power series form, $p_{12}(x) = x^{12} - 60x^{11} + 1613x^{10} - 25644x^9 + \cdots$ as computed with MATLAB's `roots` function (see Section 6.6)
.

Example 6.5.6. Wilkinson devised an example of an ill–conditioned polynomial of degree 20 with the integer roots $1, 2, \ldots, 20$, that is

$$\begin{aligned} p_{20}(x) &= (x-1)(x-2)\cdots(x-20) \\ &= x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + \cdots \end{aligned}$$

To demonstrate ill–conditioning in this case, suppose we use MATLAB (see Section 6.6) to construct the power series form of $p_{20}(x)$, and then compute its roots. Some of the coefficients are sufficiently large integers that they require more digits than are available in MATLAB DP numbers; so, they are rounded. The computed roots of $p_{20}(x)$ are shown in Table 6.10.

Observe that the computed roots are all real but the least accurate differ from the true (integer) roots in the fourth decimal place (recall that MATLAB DP uses about 15 decimal digits in its arithmetic). A calculation of the residuals (that is, evaluations of the computed polynomial) using the MATLAB function `polyval` for the true (integer) roots and for the computed roots shows that both are large. (The true roots have large residuals, when they should mathematically be zero, because of the effects of rounding.) The residuals for the computed roots are about 2 orders of magnitude larger than the residuals for the true roots. This indicates that MATLAB performed satisfactorily – it was responsible for losing only about 2 of the available 15 digits of accuracy. So, the errors in the computed roots are the result mainly of the ill–conditioning of the original polynomial. [Note: The results that we give are machine dependent though you should get approximately the same values using DP arithmetic with any compiler on any computer implementing the IEEE Standard.]

Example 6.5.7. Next, we increase the degree of the polynomial to 22 giving $p_{22}(x)$ by adding the (next two) integer roots 21 and 22. Then, we follow the same computation. This time our computation, shown in Table 6.11, gives a mixture of real and complex conjugate pairs of roots with much larger errors than the previous example.

Observe that some of the large roots 21, 20, 15, 12, 11 and 10 of $p_{22}(x)$ are each computed to no more than three correct digits. Worse, each of the pairs of real roots 18 and 19, 16 and 17, and 13 and 14 of $p_{22}(x)$ become complex conjugate pairs of roots of $P_{22}(x)$. They are not even close to being real!

exact root	computed root
20	20.0003383927958
19	18.9970274109358
18	18.0117856688823
17	16.9695256538411
16	16.0508983668794
15	14.9319189435064
14	14.0683793756331
13	12.9471623191454
12	12.0345066828504
11	10.9836103930028
10	10.0062502975018
9	8.99831804962360
8	8.00031035715436
7	6.99996705935380
6	6.00000082304303
5	5.00000023076008
4	3.99999997423112
3	3.00000000086040
2	1.99999999999934
1	0.999999999999828

Table 6.10: Roots of the polynomial $p_{20}(x)$ in Example 6.5.6. The left column shows the exact roots, and the right column shows the computed roots of the power series form of $p_{20}(x)$ as computed with MATLABś `roots` function (see Section 6.6).

exact root	computed root
22	22.0044934641580
21	20.9514187857428
20	20.1656378752978
19	18.5836448660249 + 0.441345126232861i
18	18.5836448660249 - 0.441345126232861i
17	16.3917246244130 + 0.462094344544644i
16	16.3917246244130 - 0.462094344544644i
15	15.0905963295356
14	13.4984691077096 + 0.371867167940926i
13	13.4984691077096 - 0.371867167940926i
12	11.6990708240321
11	11.1741488784148
10	9.95843441261174
9	9.00982455499210
8	7.99858075332924
7	7.00011857014703
6	5.99999925365778
5	4.99999900194002
4	4.00000010389144
3	2.99999999591073
2	2.00000000004415
1	1.00000000000012

Table 6.11: Roots of the polynomial $p_{22}(x)$ in Example 6.5.7. The left column shows the exact roots, and the right column shows the computed roots of the power series form of $p_{22}(x)$ as computed with MATLAB's `roots` function (see Section 6.6).

Problem 6.5.8. *Consider the quadratic polynomial $p_2(x) = x^2 - 2x + 1$ with double root $x = 1$. Perturb the quadratic to the form $P_2(x) = x^2 - 2(1+\epsilon)x + 1$ where ϵ is small in magnitude compared to 1 but may be either positive or negative. Approximately, by how much is the root of the quadratic perturbed? Does the root remain a double root? If not, do the roots remain real?*

6.6 Matlab Notes

By now our familiarity with MATLAB should lead us to suspect that there are built-in functions that can be used to compute roots, and it is not necessary to write our own implementations of the methods discussed in this chapter. The MATLAB functions that are relevant to the topics discussed in this chapter include:

`roots`	used to compute all roots of $p(x) = 0$ were $p(x)$ is a polynomial
`fzero`	used to find a root of $f(x) = 0$ for a general function of one variable

It is, however, instructive to implement some of the methods; at the very least, such exercises can help to improve our programming skills. We therefore begin this section by developing implementations of fixed–point, Newton and bisection methods. We then conclude this section with a thorough discussion of the built-in MATLAB functions `roots` and `fzero`. We emphasize that it is typically best to use one of these built-in functions, which have been thoroughly tested, rather than one of our own implementations.

6.6.1 Fixed–Point Iteration

Recall that the basic fixed–point iteration is given by

$$x_{n+1} = g(x_n), \quad n = 0, 1, 2, ...$$

The iterates, x_{n+1}, can be computed using a `for` loop. To begin the iteration, though, we need an initial guess, x_0, and we need a function, $g(x)$. If $g(x)$ is simple, then it is usually best to define it as an anonymous function, and if $g(x)$ is complicated, then it is usually best to define it as a function m-file.

Example 6.6.1. Suppose we want to compute 12 iterations of $g(x) = e^{-x}$, using the initial guess $x_0 = 0.5$ (see Table 6.1). Since $g(x)$ is a very simple function, we define it as an anonymous function, and use a `for` loop to generate the iterates:

```
g = @(x) exp(-x);
x = 0.5;
for n = 1:12
  x = g(x);
end
```

If we want to test the fixed–point iteration for several different $g(x)$, then it would be helpful to have a MATLAB program (that is, a function m-file) that works for arbitrary $g(x)$. In order to write this function, we need to consider a few issues:

- In the previous example, we chose, somewhat arbitrarily, to compute 12 iterations. For other problems this may be too many or two few iterations. To increase flexibility, we should:

- Stop the iteration if the computed solution is sufficiently accurate. This can be done, for example, by stopping the iteration if the relative change between successive iterates is less than some specified tolerance. That is, if

  $$|x_{n+1} - x_n| \leq \text{tol} * |x_n|.$$

  This can be implemented by inserting the following conditional statement inside the `for` loop:

  ```
  if ( abs(x(n+1)-x(n)) <= tol*max(abs(x(n)), abs(x(n+1))))
    break
  end
  ```

  A value of `tol` must be either defined in the code, or specified by the user. The command `break` terminates execution of the `for` loop.
- Allow the user to specify a maximum number of iterations, and execute the loop until the above convergence criterion is satisfied, or until the maximum number of iterations is reached.

• It might be useful to return all of the computed iterations. This can be done easily by creating a vector containing $x_0, x_1, \cdots$.

Combining these items with the basic code given in Example 6.6.1 we obtain the following function:

```
function [x, flag] = FixedPoint(g, x0, tol, nmax)
%
%     [x, flag] = FixedPoint(g, x0, tol, nmax);
%
% Find a fixed--point of g(x) using the iteration:  x(n+1) = g(x(n))
%
% Input:  g - an anonymous function or function handle,
%        x0 - an initial guess of the fixed--point,
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%         flag - set to 0 for normal exit and to 1
%                for exceeding maximum number of iterations
%
x(1) = x0; flag = 1;
for n = 1:nmax
  x(n+1) = g(x(n));
  if ( abs(x(n+1)-x(n)) <= tol*max(abs(x(n)), abs(x(n+1))))
    flag = 0;
    break
  end
end
x = x(:);
```

The final statement in the function, `x = x(:)`, is used to ensure `x` is returned as a column vector. This, of course, is not necessary, and is done only for cosmetic purposes.

Example 6.6.2. To illustrate how to use the function `FixedPoint`, consider $g(x) = e^{-x}$ with the initial guess $x_0 = 0.5$.

(a) If we want to compute exactly 12 iterations, we can set `nmax = 12` and `tol = 0`. That is, we use the statements:

```
g = @(x) exp(-x);
[x, flag] = FixedPoint(g, 0.5, 0, 12);
```

The same result could be computed using just one line of code:

```
[x, flag] = FixedPoint(@(x) exp(-x), 0.5, 0, 12);
```

Note that on exit, `flag = 1`, indicating that the maximum number of iterations has been exceeded, and that the iteration did not converge within the specified tolerance. Of course this is not surprising since we set `tol = 0`!

(b) If we want to see how many iterations it takes to reach a certain stopping tolerance, say `tol =` 10^{-8}, then we should choose a relatively large value for `nmax`. For example, we could compute

```
[x, flag] = FixedPoint(@(x) exp(-x), 0.5, 1e-8, 1000);
```

On exit, we see that `flag = 0`, and so the iteration has converged within the specified tolerance. The number of iterations is then `length(x) - 1` (recall the first entry in the vector is the initial guess). For this example, we find that it takes 31 iterations.

Problem 6.6.1. *Implement* `FixedPoint` *and use it to:*

(a) Produce a table corresponding to Table 6.1 starting from the value $x_0 = 0.6$

(b) Produce a table corresponding to Table 6.2 starting from the value $x_0 = 0.0$

(c) Produce a table corresponding to Table 6.2 starting from the value $x_0 = 2.0$

Recall that it is possible to produced formatted tables in MATLAB *using the* `sprintf` *and* `disp` *commands; see Section 1.3.5.*

Problem 6.6.2. *Implement* `FixedPoint` *and use it to compute 10 iterations using each* $g(x)$ *given in Problem 6.2.6, with* $x_0 = 1.5$*.*

Problem 6.6.3. *Rewrite the function* `FixedPoint` *using* `while` *instead of the combination of* `for` *and* `if`*. Note that it is necessary to implement a counter to determine when the number of iterations has reached* `nmax`*.*

6.6.2 Newton and Secant Methods

The Newton and secant methods are used to find roots of $f(x) = 0$. Implementation of these methods is very similar to the fixed–point iteration. For example, Newton's method can be implemented as follows:

```
function x = Newton(f, df, x0, tol, nmax)
%
%     x = Newton(f, df, x0, tol, nmax);
%
% Use Newton's method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%        df - an anonymous function or function handle for f'(x)
%        x0 - an initial guess of the root,
%       tol - a (relative) stopping tolerance
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%
x(1) = x0;
for n = 1:nmax
  x(n+1) = x(n) - f(x(n)) / df(x(n));
  if ( abs(x(n+1)-x(n)) <= tol*abs(x(n)) )
    break
  end
end
x = x(:);
```

Note that in Newton's method we must input two functions, $f(x)$ and $f'(x)$. It might be prudent to include a check, and to print an error message, if `df(x(n))` is zero, or smaller in magnitude than a specified value.

Example 6.6.3. Suppose we want to find a root of $f(x) = 0$ where $f(x) = x - e^{-x}$, with initial guess $x_0 = 0.5$. Then, using `Newton`, we can compute an approximation of this root using the following MATLAB statements:

```
f = @(x) x - exp(-x);
df = @(x) 1 + exp(-x);
x = Newton(f, df, 0.5, 1e-8, 10);
```

or, in one line of code, using

```
x = Newton(@(x) x - exp(-x), @(x) 1 + exp(-x), 0.5, 1e-8, 10);
```

For this problem, only 4 iterations of Newton's method are needed to converge to the specified tolerance.

Problem 6.6.4. *Using the* `Fixed Point` *code above as a template, modify* `Newton` *to check more carefully for convergence and to use a flag to indicate whether convergence has occurred before the maximum number of iterations has been exceeded. Also include in your code a check for small derivative values.*

Problem 6.6.5. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Test both codes at a simple root of a quadratic polynomial $f(x)$ that is concave up, choosing the initial iterates x_0 (and, in the case of secant, x_1) close to this root. Construct tables containing the values of the iterates, x_n and corresponding function values, $f(x_n)$. Is there any pattern to the signs of the consecutive $f(x)$ values?*

Problem 6.6.6. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Use both codes to compute the root of $f(x) = x - e^{-x}$. For Newton, use the initial guess $x_0 = 0.5$, and for secant use $x_0 = 0$ and $x_1 = 1$. Check the ratios of the errors in successive iterates as in Table 6.7 to determine if they are consistent with the convergence rates discussed in this chapter. Repeat the secant method with initial iterates $x_0 = 1$ and $x_1 = 0$.*

Problem 6.6.7. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Use both methods to compute the root of $f(x) = \ln(x-3) + \sin(x) + 1$, $x > 3$. For Newton, use the initial guess $x_0 = 4$, and for secant use $x_0 = 3.25$ and $x_1 = 4$. Check the ratios of the errors in successive iterates as in Table 6.7 to determine if they are consistent with the convergence rates discussed in this chapter.*

6.6.3 Bisection Method

Recall that bisection begins with an initial bracket, $[a, b]$ containing the root, and proceeds as follows:

- First compute the midpoint, $m = (a + b)/2$.
- Determine if a root is in $[a, m]$ or $[m, b]$. If the root is in $[a, m]$, rename m as b, and continue. Similarly, if the root is in $[m, b]$, rename m as a and continue.
- Stop if the length of the interval becomes sufficiently small.

Note that we can determine if the root is in $[a, m]$ by checking to see if $f(a) \cdot f(m) \leq 0$. With these observations, an implementation of the bisection method could have the form:

```
function x = Bisection0(f, a, b, tol, nmax)
%
%     x = Bisection0(f, a, b, tol, nmax);
%
% Use bisection method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%     [a,b] - an initial bracket containing the root
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%
x(1) = a;   x(2) = b;
for n = 2:nmax
  m = (a+b)/2;   x(n+1) = m;
  if f(a)*f(m) < 0
    b = m;
  else
    a = m;
  end
  if ( abs(b-a) <= tol*abs(a) )
    break
  end
end
x = x(:);
```

Note that this implementation requires that we evaluate $f(x)$ twice for every iteration (to compute $f(a)$ and $f(b)$). Since this is typically the most expensive part of the method, it is a good idea to reduce the number of function evaluations as much as possible. The bisection method can be implemented so that only one function evaluation is needed at each iteration. In the algorithm below we have taken care to keep track of whether the maximum number of iterations has been exceeded and to check for zeros and avoid underflows in our treatment of the function values generated during the iteration.

Problem 6.6.8. *Use the* `Bisection` *code to compute a root of* $f(x) = x - e^{-x}$*. Use the initial bracket* $[0, 1]$*. Produce a table of results similar to Table 6.8.*

Problem 6.6.9. *Using the* `Bisection`*,* `Secant` *and* `Newton` *codes, experimentally compare the convergence behavior of each method for the function* $f(x) = x - e^{-x}$*.*

```
function [x, flag] = Bisection(f, a, b, tol, nmax)
%
%     [x, flag]  = Bisection(f, a, b, tol, nmax);
%
% Use bisection method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%     [a,b] - an initial bracket containing the root
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...,
%             with last iterate the final approximation to the root
%         flag - set to 0 for normal exit and to 1
%                for exceeding maximum number of iterations
%
x(1) = a;  x(2) = b;  fa = f(a);  fb = f(b);  flag = 1;
if (sign(fa) ~= 0 & sign(fb) ~= 0)
  for n = 2:nmax
    c = (a+b)/2;  x(n+1) = c;  fc = f(c);
    if sign(fc) == 0
      flag = 0;
      break
    end
    if sign(fa) ~= sign(fc)
      b = c;  fb = fc;
    else
      a = c;  fa = fc;
    end
    if (abs(b-a) <= tol*max(abs(a), abs(b)))
      flag = 0;
      break
    end
  end
else
  if sign(fa) == 0
    x(2) = a; x(1) = b;
  end
  flag = 0;
end
x = x(:);
```

6.6.4 The `roots` and `fzero` Functions

As previously mentioned, MATLAB provides two built-in root finding methods, `roots` (to find roots of polynomials) and `fzero` (to find a root of a more general function). In this section we discuss in more detail how to use these built-in MATLAB functions.

Computing a solution of $f(x) = 0$.

The MATLAB function `fzero` is used to find zeros of a general function of one variable. The implementation uses a combination of bisection, secant and inverse quadratic interpolation. The basic calling syntax for `fzero` is:

```
x = fzero(fun, x0)
```

where `fun` can be an inline or anonymous function, or a handle to a function M–file. The initial guess, `x0` can be a scalar (that is, a single initial guess), or it can be a bracket (that is, a vector containing two values) on the root. Note that if `x0` is a single initial guess, then `fzero` first attempts to find a bracket by sampling on successively wider intervals containing `x0` until a bracket is determined. If a bracket for the root is known, then it is usually best to supply it, rather than a single initial guess.

Example 6.6.4. Consider the function $f(x) = x - e^{-x}$. Because $f(0) < 0$ and $f(1) > 0$, a bracket for a root of $f(x) = 0$ is $[0, 1]$. This root can be found as follows:

```
x = fzero(@(x) x-exp(-x), [0, 1])
```

We could have used a single initial guess, such as:

```
x = fzero(@(x) x-exp(-x), 0.5)
```

but as mentioned above, it is usually best to provide a bracket if one is known.

Problem 6.6.10. *Use `fzero` to find a root of $f(x) = 0$, with $f(x) = \sin x$. For initial guess, use $x_0 = 1$, $x_0 = 5$, and the bracket $[1, 5]$. Explain why a different root is computed in each case.*

Problem 6.6.11. *Consider the function $f(x) = \frac{1}{x} - 1$, and note that $f(1) = 0$. Thus, we might expect that $x_0 = 0.5$ is a reasonable initial guess for `fzero` to compute a solution of $f(x) = 0$. Try this, and see what `fzero` computes. Can you explain what happens here? Now use the initial guess to be the bracket $[0.5, 1.5]$. What does `fzero` compute in this case?*

If a root, x_*, has multiplicity greater than one, then $f'(x_*) = 0$, and hence the root is not simple. `fzero` can find roots which have odd multiplicity, but it cannot find roots of even multiplicity, as illustrated in the following example.

Example 6.6.5. Consider the function $f(x) = x^2 e^x$, which has a root $x_* = 0$ of multiplicity 2.

(a) If we use the initial guess $x_0 = 1$, then

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, 1)
```

computes $x \approx -925.8190$. At first this result may seem completely ridiculous, but if we compute

```
f(xstar)
```

the result is 0. Notice that $\lim_{x\to-\infty} x^2 e^x = 0$, thus it appears that `fzero` "finds" $x \approx -\infty$.

(b) If we try the initial guess $x_0 = -1$, then

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, -1)
```

then `fzero` fails because it cannot find an interval containing a sign change.

(c) We know the root is $x_* = 0$, so we might try the bracket $[-1, 1]$. However, if we compute

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, [-1, 1])
```

an error occurs because the initial bracket does not satisfy the sign change condition, $f(-1)f(1) < 0$.

Problem 6.6.12. *The functions $f(x) = x^2 - 4x + 4$ and $f(x) = x^3 - 6x^2 + 12x - 8$ satisfy $f(2) = 0$. Attempt to find this root using* `fzero`*. Experiment with a variety of initial guesses. Why does* `fzero` *have trouble finding the root for one of the functions, but not for the other?*

It is possible to reset certain default parameters used by `fzero` (such as stopping tolerance and maximum number of iterations), and it is possible to obtain more information on exactly what is done during the computation of the root (such as number of iterations and number of function evaluations) by using the calling syntax

```
[x, fval, exitflag, output] = fzero(fun, x0, options)
```

where

- `x` is the computed approximation of the root.
- `fval` is the function value of the computed root, $f(x_*)$. Note that if a root is found, then this value should be close to zero.
- `exitflag` provides information about what condition caused `fzero` to terminate; see `doc fzero` for more information.
- `output` is a structure array that contains information such as which algorithms were used (e.g., bisection, interpolation, secant), number of function evaluations and number of iterations.
- `options` is an input parameter that can be used, for example, to reset the stopping tolerance, and to request that results of each iteration be displayed in the command window. The options are set using the built-in MATLAB function `optimset`.

The following examples illustrate how to use `options` and `output`.

Example 6.6.6. Consider $f(x) = x - e^{-x}$. We know there is a root in the interval $[0, 1]$. Suppose we want to investigate how the cost of computing the root is affected by the choice of the initial guess. Since the cost is reflected in the number of function evaluations, we use `output` for this purpose.

(a) Using the initial guess $x_0 = 0$, we can compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), 0);
output
```

we see that `output` displays the information:

```
intervaliterations: 10
        iterations: 6
         funcCount: 27
         algorithm: 'bisection, interpolation'
           message: 'Zero found in the interval [-0.64, 0.64]'
```

The important quantity here is `funcCount`, which indicates that a total of 27 function evaluations was needed to compute the root.

(b) On the other hand, if we use $x_0 = 0.5$, and compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), 0.5);
output
```

we see that 18 function evaluations are needed.

(c) Finally, if we use the bracket $[0, 1]$, and compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1]);
output
```

we see that only 8 function evaluations are needed.

Notice from this example that when we provide a bracket, the term `interval iterations`, which is returned by the structure `output`, is 0. This is because `fzero` does not need to do an initial search for a bracket, and thus the total number of function evaluations is substantially reduced.

Example 6.6.7. Again, consider $f(x) = x - e^{-x}$, which has a root in $[0, 1]$. Suppose we want to investigate how the cost of computing the root is affected by the choice of stopping tolerance on x. To investigate this, we must first use the MATLAB function `optimset` to define the structure `options`.

(a) Suppose we want to set the tolerance to 10^{-4}. Then we can use the command:

```
options = optimset('TolX', 1e-4);
```

If we then execute the commands:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1], options);
output
```

we see that only 5 function evaluations are needed.

(b) To increase the tolerance to 10^{-8}, we can use the command:

```
options = optimset('TolX', 1e-8);
```

Executing the commands:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1], options);
output
```

shows that 7 function evaluations are needed to compute the approximation of the root.

Example 6.6.8. Suppose we want to display results of each iteration when we compute an approximation of the root of $f(x) = x - e^{-x}$. For this, we use `optimset` as follows:

```
options = optimiset('Display', 'iter');
```

If we then use `fzero` as:

```
x = fzero(@(x) x-exp(-x), [0, 1], options);
```

then the following information is displayed in the MATLAB command window:

```
 Func-count    x          f(x)            Procedure
    2               1      0.632121        initial
    3          0.6127     0.0708139        interpolation
    4         0.56707  -0.000115417        interpolation
    5        0.567144   9.44811e-07        interpolation
    6        0.567143   1.25912e-11        interpolation
    7        0.567143  -1.11022e-16        interpolation
    8        0.567143  -1.11022e-16        interpolation

Zero found in the interval [0, 1]
```

Notice that a total of 8 function evaluations were needed, which matches (as it should) the number found in part (c) of Example 6.6.6.

Problem 6.6.13. *Note that we can use* `optimset` *to reset* `TolX` *and to request display of the iterations in one command, such as:*

```
options = optimset('TolX', 1e-7, 'Display', 'iter');
```

Use this set of options with $f(x) = x - e^{-x}$, and various initial guesses (e.g., 0, 1 and [0,1]) in `fzero` *and comment on the computed results.*

Roots of polynomials

The MATLAB function `fzero` cannot, in general, be effectively used to compute roots of polynomials. For example, it cannot compute the roots of $f(x) = x^2$ because $x = 0$ is a root of even multiplicity. However, there is a special purpose method, called `roots`, that can be used to compute all roots of a polynomial.

Recall from our discussion of polynomial interpolation (section 4.5) MATLAB assumes polynomials are written in the canonical form

$$p(x) = a_1x^n + a_2x_{n-1} + \cdots + a_nx + a_{n+1},$$

and that they are represented with a vector (either row or column) containing the coefficients:

$$a = \begin{bmatrix} a_1 & a_2 & \cdots & a_n & a_{n+1} \end{bmatrix}.$$

If such a vector is defined, then the roots or $p(x)$ can be computed using the built-in function `roots`. Because it is beyond the scope of this book, we have not discussed the basic algorithm used by `roots` (which computes the eigenvalues of a *companion matrix* defined by the coefficients of $p(x)$), but it is a very simple function to use. In particular, execution of the statement

```
r = roots(a);
```

produces a (column) vector `r` containing the roots of the polynomial defined by the coefficients in the vector `a`.

Example 6.6.9. Consider $x^4 - 5x^2 + 4 = 0$. The following MATLAB commands:

```
a = [1 0 -5 0 4];
r = roots(a)
```

compute a vector `r` containing the roots 2, 1, -2 and -1.

The MATLAB function `poly` can be used to create the coefficients of a polynomial with given roots. That is, if `r` is a vector containing the roots of a polynomial, then

```
a = poly(r);
```

constructs a vector containing the coefficients of a polynomial whose roots are given by the entries in the vector `r`. Basically, `poly` multiplies out the factored form

$$p(x) = (x - r_1)(x - r_2) \cdots (x - r_n)$$

to obtain the canonical power series form

$$p(x) = x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1}.$$

Example 6.6.10. The roots of $(2x + 1)(x - 3)(x + 7) = 0$ are obviously $-\frac{1}{2}, 3$ and -7. Executing the MATLAB statements

```
r = [-1/2, 3, -7];
a = poly(r)
```

constructs the vector `a = [1, 4.5, -19, -10.5]`.

Problem 6.6.14. *Consider the polynomial given in Example 6.5.5. Use the* MATLAB *function* `poly` *to construct the coefficients of the power series form of* $p_{12}(x)$*. Then compute the roots using the* MATLAB *function* `roots`*. Use the* `format` *command so that computed results display 15 digits.*

Problem 6.6.15. *Consider the polynomial* $p_{20}(x)$ *given in Example 6.5.6. Compute the results shown in that example using the* MATLAB *functions* `poly` *and* `roots`*. Use the* `format` *command so that computed results display 15 digits.*

Problem 6.6.16. *Consider the polynomial* $p_{22}(x)$ *given in Example 6.5.7. Compute the results shown in that example using the* MATLAB *functions* `poly` *and* `roots`*. Use the* `format` *command so that computed results display 15 digits.*