CHAPTER 10

Root-finding

10.1 Introduction

The next few chapters introduce numerical algorithms for solving some common applied mathematical problems. In each case we present motivating examples, some underpinning theory, and applications in R. This chapter focuses on root-finding, and covers fixed-point iteration, the Newton-Raphson method, the secant method, and the bisection method.

Suppose that $f : \mathbb{R} \to \mathbb{R}$ is a continuous function. A *root* of f is a solution to the equation $f(x) = 0$ (see Figure 10.1 for example). That is, a root is a number $a \in \mathbb{R}$ such that $f(a) = 0$. If we draw the graph of our function, say $y = f(x)$, which is a curve in the plane, a solution of $f(x) = 0$ is the x-coordinate of a point at which the curve crosses the x-axis.

The roots of a function are important algebraically, for example, we use the

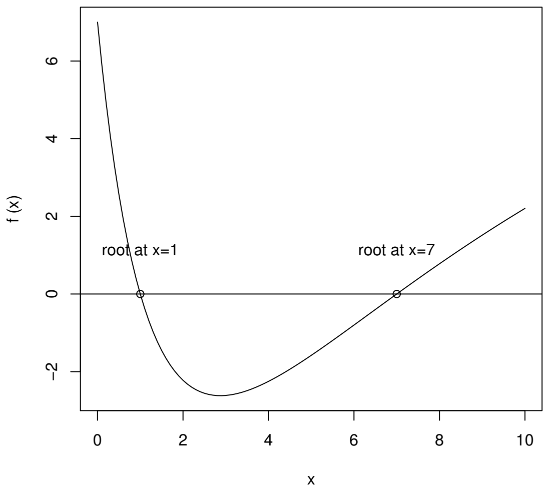

Figure 10.1 *The roots of the function f.*

roots of a polynomial to factorise it. Moreover the solution to a physical problem can often be expressed as the root of a suitable function. Root-finding is also a classical numerical or computational problem, and provides a good introduction to important issues in numerical mathematics.

10.1.1 Example: loan repayments

Suppose that a loan has an initial amount P, a monthly interest rate r, a duration of N months, and a monthly repayment of A. The remaining debt after n months is given by P_n, where

$$\begin{aligned} P_0 &= P; \\ P_{n+1} &= P_n(1+r) - A. \end{aligned}$$

That is, each month you pay interest on the previous balance, then reduce the balance of the loan by amount A. This is a first-order recurrence equation, and has the following solution (check that it works):

$$P_n = P(1+r)^n - A((1+r)^n - 1)/r.$$

Putting $P_N = 0$, we get

$$\frac{A}{P} = \frac{r(1+r)^N}{(1+r)^N - 1}.$$

Suppose that we know P, N, and A, then we can find r by finding the root(s) of the following function:

$$f(x) = \frac{A}{P} - \frac{x(1+x)^N}{(1+x)^N - 1}.$$

Choose some values for P, N, and A and then try finding r analytically. If/when you decide this is too hard, you can try doing it numerically with one of the techniques below.

10.2 Fixed-point iteration

Let $g : \mathbb{R} \to \mathbb{R}$ be a continuous function. A *fixed point* of g is a number a such that $g(a) = a$. That is, a is a solution of the equation $g(x) = x$. Graphically, a fixed point is where the graph $y = g(x)$ of the function crosses the line $y = x$.

The computational problem of finding fixed points of a function is easily reduced to the problem of finding roots. To see this define the function $f(x)$ by the equation $f(x) = c(g(x) - x)$, where c is a non-zero constant, then one clearly has $f(a) = 0$ if and only if $g(a) = a$. So to find the fixed points of the function g we need only find the roots of the associated function f, that is solutions of the equation $f(x) = 0$. Conversely, the problem of finding roots

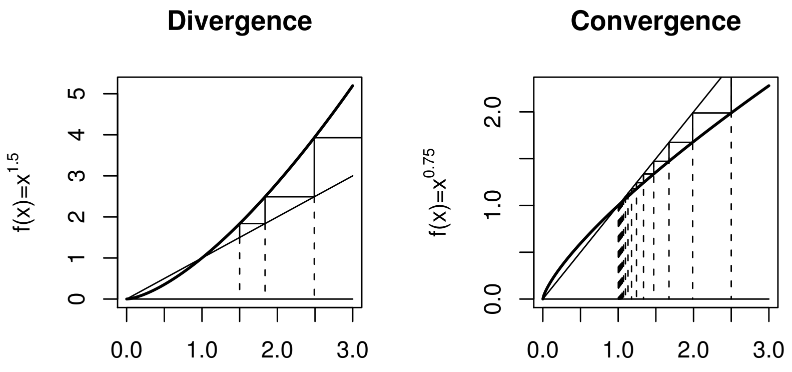

Figure 10.2 *The fixed-point algorithm applied to the function $y = x^{1.5}$, starting at $x_0 = 1.5$, and to the function $y = x^{0.75}$, starting at $x_0 = 2.5$.*

of $f(x) = 0$ is equivalent to the problem of finding fixed points of the function $g(x) = c \cdot f(x) + x$.

Although this is one way to convert from one form of problem to the other, it is not the only way, and in practice some ways are better than others.

The 'fixed-point method' is an iterative method for solving $g(x) = x$. That is, it generates a sequence of points $x_0, x_1, x_2, \ldots$ that (hopefully) converges to some point a such that $g(a) = a$. Starting with our initial guess x_0, we generate the next guess using $x_1 = g(x_0)$ and repeat. This gives the following first-order recurrence relation (also called a difference equation):

Fixed-point method

$$x_{n+1} = g(x_n).$$

If $x_n \to a$ then, given g is continuous, we have

$$a = \lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} g(x_n) = g(\lim_{n\to\infty} x_n) = g(a).$$

So a is a fixed point of g. But does the sequence $\{x_n\}_{n=0}^{\infty}$ always converge? The answer, sadly, is no.

In Figure 10.2 we illustrate the application of the fixed-point method to the functions $g_1(x) = x^{1.5}$ and $g_2(x) = x^{0.75}$, starting to the right of 1 in each case. The dotted lines give the successive values of x_n. The solid lines show how we obtain x_{n+1} from x_n. Both functions have fixed points at 1, but the algorithm diverges when applied to g_1 and converges when applied to g_2. The crucial difference is the value of g' at the fixed point: if $|g'(a)| < 1$ at the fixed point a, then the algorithm will converge, otherwise it diverges. It is also necessary to start relatively 'close' to the fixed point to guarantee that you will converge to it. We will not prove this result here, though Exercise 2 gives an outline of how it is done. You should be able to convince yourself that it is true by looking at the figure.

Even when the method does converge, we still have a (small) problem: $\{x_n\}_{n=0}^{\infty}$ may converge to a, but never actually reach it. We can never avoid this problem, rather we have to accommodate it. The best we can do is ask for an x_n within distance δ of a, for some small $\delta > 0$.

Practically, to avoid iterating forever, we stop when $|x_n - x_{n-1}| \le \epsilon$ for some (user-specified) tolerance ϵ. Noting that $g(a) = a$, and that $g(x) - g(a) \approx g'(a)(x - a)$ for x close to a, we have the following:

$$\begin{aligned}
|x_n - x_{n-1}| \le \epsilon \quad &\Leftrightarrow \quad |g(x_{n-1}) - x_{n-1}| \le \epsilon \\
&\Leftrightarrow \quad |g(x_{n-1}) - g(a) - (x_{n-1} - a)| \le \epsilon \\
&\Rightarrow \quad |x_{n-1} - a| \le \epsilon + |g(x_{n-1}) - g(a)| \approx \epsilon + g'(a)|x_{n-1} - a| \\
&\Rightarrow \quad |x_{n-1} - a| \le \epsilon/(1 - g'(a)).
\end{aligned}$$

Thus, to ensure $|x_n - a| \le \delta$ we need to choose $\epsilon \le \delta(1 - g'(a))$ (approximately). Of course, until we know a we can't find $g'(a)$, so in practice we just choose ϵ to be small.

Note that the fixed-point method can still be used if g' does not exist, provided g is continuous. However, the convergence properties of the method are harder to describe in such a case.

The code below implements the fixed-point algorithm in a function `fixedpoint`. To use it you first need to create a function, `ftn(x)` say, that returns $g(x)$. `fixedpoint(ftn, x0, tol = 1e-9, max.iter = 100)` has four inputs:

`ftn` is the name of a function that takes a single numeric input and returns a single numeric result.

`x0` is the starting point for the algorithm.

`tol` is such that the algorithm will stop if $|x_n - x_{n-1}| \le$ tol, with default 10^{-9}.

`max.iter` is such that the algorithm will stop when n = max.iter, with default 100.

We remark that, because the fixed-point method is not guaranteed to converge, our coding of the algorithm counts how many iterations have been performed, and stops if they exceed some specified maximum. This prevents the function running on forever.

```
# program spuRs/resources/scripts/fixedpoint.r
# loadable spuRs function

fixedpoint <- function(ftn, x0, tol = 1e-9, max.iter = 100) {
  # applies the fixed-point algorithm to find x such that ftn(x) == x
  # we assume that ftn is a function of a single variable
  #
  # x0 is the initial guess at the fixed point
  # the algorithm terminates when successive iterations are
  # within distance tol of each other,
  # or the number of iterations exceeds max.iter

  # do first iteration
  xold <- x0
  xnew <- ftn(xold)
  iter <- 1
  cat("At iteration 1 value of x is:", xnew, "\n")

  # continue iterating until stopping conditions are met
  while ((abs(xnew-xold) > tol) && (iter < max.iter)) {
    xold <- xnew;
    xnew <- ftn(xold);
    iter <- iter + 1
    cat("At iteration", iter, "value of x is:", xnew, "\n")
  }

  # output depends on success of algorithm
  if (abs(xnew-xold) > tol) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    return(xnew)
  }
}
```

10.2.1 Example: finding the root of $f(x) = \log(x) - \exp(-x)$

We consider three approaches to solving the equation $f(x) = \log(x) - \exp(-x) = 0$. First, we put one term on each side of the equation and exponentiate both sides to get

$$x = \exp(\exp(-x)) = g_1(x).$$

Second, we subtract each side from x to get

$$x = x - \log x + \exp(-x) = g_2(x).$$

Finally, we add x to both sides to get

$$x = x + \log x - \exp(-x) = g_3(x).$$

1. Applying the fixed-point method to g_1, we find the sequence appears to con-

verge but it takes 14 iterations for successive guesses to agree to 6 decimal places.

```
> source("../scripts/fixedpoint.r")
> ftn1 <- function(x) return(exp(exp(-x)))
> fixedpoint(ftn1, 2, tol = 1e-06)
```

```
At iteration 1 value of x is: 1.144921
At iteration 2 value of x is: 1.374719
At iteration 3 value of x is: 1.287768
At iteration 4 value of x is: 1.317697
At iteration 5 value of x is: 1.307022
At iteration 6 value of x is: 1.310783
At iteration 7 value of x is: 1.309452
At iteration 8 value of x is: 1.309922
At iteration 9 value of x is: 1.309756
At iteration 10 value of x is: 1.309815
At iteration 11 value of x is: 1.309794
At iteration 12 value of x is: 1.309802
At iteration 13 value of x is: 1.309799
At iteration 14 value of x is: 1.309800
Algorithm converged
[1] 1.309800
```

2. Using g_2, we find the sequence appears to converge and it takes only 6 iterations.

```
> ftn2 <- function(x) return(x - log(x) + exp(-x))
> fixedpoint(ftn2, 2, tol = 1e-06)
```

```
At iteration 1 value of x is: 1.442188
At iteration 2 value of x is: 1.312437
At iteration 3 value of x is: 1.309715
At iteration 4 value of x is: 1.309802
At iteration 5 value of x is: 1.309799
At iteration 6 value of x is: 1.309800
Algorithm converged
[1] 1.309800
```

3. Using g_3, we find the sequence does not appear to converge at all.

```
> ftn3 <- function(x) return(x + log(x) - exp(-x))
> fixedpoint(ftn3, 2, tol = 1e-06, max.iter = 20)
```

```
At iteration 1 value of x is: 2.557812
At iteration 2 value of x is: 3.41949
At iteration 3 value of x is: 4.616252
At iteration 4 value of x is: 6.135946
At iteration 5 value of x is: 7.947946
At iteration 6 value of x is: 10.02051
At iteration 7 value of x is: 12.32510
At iteration 8 value of x is: 14.83673
At iteration 9 value of x is: 17.53383
At iteration 10 value of x is: 20.39797
At iteration 11 value of x is: 23.4134
At iteration 12 value of x is: 26.56671
At iteration 13 value of x is: 29.84637
At iteration 14 value of x is: 33.24243
At iteration 15 value of x is: 36.74626
At iteration 16 value of x is: 40.35030
At iteration 17 value of x is: 44.04789
At iteration 18 value of x is: 47.83317
At iteration 19 value of x is: 51.70089
At iteration 20 value of x is: 55.64637
Algorithm failed to converge
NULL
```

This example illustrates that as a method for finding roots, the fixed-point method has some disadvantages. One needs to convert the problem into fixed-point form, but there are many ways to do this, each of which will have different convergence properties and some of which will not converge at all. We consider the question of the best way of converting a root-finding problem to a fixed-point problem in Exercise 7.

It also turns out that the fixed-point method is relatively slow, in that the error is usually divided by a constant factor at each iteration. Both of our next two algorithms, the Newton-Raphson method and the secant method, converge more quickly because they make informed guesses as to where to find a better approximation to the root.

10.3 The Newton-Raphson method

Suppose our function f is differentiable with continuous derivative f' and a root a. Let $x_0 \in \mathbb{R}$ and think of x_0 as our current 'guess' at a. Now the straight line through the point $(x_0, f(x_0))$ with slope $f'(x_0)$ is the best straight line approximation to the function $f(x)$ at the point x_0 (this is the *meaning* of the derivative). The equation of this straight line is given by

$$f'(x_0) = \frac{f(x_0) - y}{x_0 - x}.$$

Now this straight line crosses the x-axis at a point x_1, which should be a better approximation than x_0 to a. To find x_1 we observe

$$f'(x_0) = \frac{f(x_0) - 0}{x_0 - x_1} \quad \text{and so} \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

In other words, the next best guess x_1 is obtained from the current guess x_0 by subtracting a correction term $f(x_0)/f'(x_0)$ (Figure 10.3).

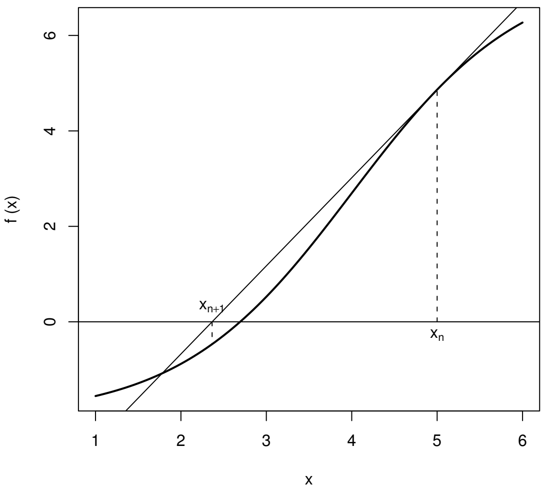

Figure 10.3 *A step in the Newton-Raphson root-finding method.*

Now that we have x_1, we use the same procedure to get the next guess

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

or in general:

Newton-Raphson method

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

Like the fixed-point method, this is a first-order recurrence relation. It can be shown that if f is 'well behaved' at a (which means $f'(a) \neq 0$ and f'' is finite and continuous at a)[1] and you start with x_0 'close enough' to a, then x_n will converge to a quickly. Unfortunately, like the fixed-point method, we don't know if f is well behaved at a until we know a, and we don't know beforehand how close is close enough.

So, we cannot guarantee convergence of the Newton-Raphson algorithm. However, if $x_n \to a$ then, since f and f' are continuous, we have

$$a = \lim_{n\to\infty} x_{n+1} \quad = \quad \lim_{n\to\infty} \left(x_n - \frac{f(x_n)}{f'(x_n)} \right)$$

[1] In fact, we can get away with the slightly less restrictive but more technical condition that $f'(a) \neq 0$ and f' is Lipschitz-continuous in a neighbourhood of a, which means that for some constant c, $|f'(x) - f'(y)| \leq c|x - y|$.

$$= \lim_{n\to\infty} x_n - \frac{f(\lim_{n\to\infty} x_n)}{f'(\lim_{n\to\infty} x_n)} = a - \frac{f(a)}{f'(a)}.$$

Thus, provided $f'(a) \neq \pm\infty$, we must have $f(a) = 0$.

Since we are expecting $f(x_n) \to 0$, a good stopping condition for the Newton-Raphson algorithm is $|f(x_n)| \leq \epsilon$ for some tolerance ϵ. If the sequence $\{x_n\}_{n=0}^{\infty}$ is converging to a root a, then for x close to a we have $f(x) \approx f'(a)(x - a)$. So if $|f(x_n)| \leq \epsilon$ we have $|x - a| \leq \epsilon/f'(a)$ (approximately).

The code below implements the Newton-Raphson algorithm in a function `newtonraphson`. To use it you first need to create a function, `ftn(x)` say, which returns the vector $(f(x), f'(x))$. `newtonraphson(ftn, x0, tol = 1e-9, max.iter = 100)` has four inputs:

`ftn` is the name of a function that takes a single numeric input and returns a numeric vector of length two. If x is the input then the output must be $(f(x), f'(x))$.

`x0` is the starting point for the algorithm.

`tol` is such that the algorithm will stop if $|f(x_n)| \leq$ tol, with default 10^{-9}.

`max.iter` is such that the algorithm will stop when $n =$ max.iter, with default 100.

As for the fixed-point method, because we cannot guarantee convergence, we count the number of iterations and stop if this gets too large. This prevents the program running indefinitely, though of course you have to make sure that you do not stop it too soon, in case it is converging more slowly than you expected. Note that, because our stopping condition only depends on $|f(x_n)|$, and not $|x_n - x_{n-1}|$, we do not have to store the previous iteration, as we did with function `fixedpoint`.

```
# program spuRs/resources/scripts/newtonraphson.r
# loadable spuRs function

newtonraphson <- function(ftn, x0, tol = 1e-9, max.iter = 100) {
  # Newton_Raphson algorithm for solving ftn(x)[1] == 0
  # we assume that ftn is a function of a single variable that returns
  # the function value and the first derivative as a vector of length 2
  #
  # x0 is the initial guess at the root
  # the algorithm terminates when the function value is within distance
  # tol of 0, or the number of iterations exceeds max.iter

  # initialise
  x <- x0
  fx <- ftn(x)
  iter <-  0

  # continue iterating until stopping conditions are met
  while ((abs(fx[1]) > tol) && (iter < max.iter)) {
    x <- x - fx[1]/fx[2]
    fx <- ftn(x)
    iter <-  iter + 1
    cat("At iteration", iter, "value of x is:", x, "\n")
  }

  # output depends on success of algorithm
  if (abs(fx[1]) > tol) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    return(x)
  }
}
```

When applied to the function $\log x - \exp(-x)$ with derivative $1/x + \exp(-x)$, we get impressively fast convergence

```
> source("../scripts/newtonraphson.r")
> ftn4 <- function(x) {
+     fx <- log(x) - exp(-x)
+     dfx <- 1/x + exp(-x)
+     return(c(fx, dfx))
+ }
> newtonraphson(ftn4, 2, 1e-06)

At iteration 1 value of x is: 1.122020
At iteration 2 value of x is: 1.294997
At iteration 3 value of x is: 1.309709
At iteration 4 value of x is: 1.309800
Algorithm converged
[1] 1.309800
```

10.4 The secant method

A problem with the Newton-Raphson algorithm is that it needs the derivative f'. If the derivative is hard to compute or does not exist, then we can use the secant method, which only requires that the function f is continuous.

Like the Newton-Raphson method, the secant method is based on a linear approximation to the function f. Suppose that f has a root at a. For this method we assume that we have two current 'guesses', x_0 and x_1, for the value of a. We will think of x_0 as an older guess and we want to replace the pair x_0, x_1 by the pair x_1, x_2, where x_2 is a new guess.

To find a good new guess x_2 we first draw the straight line from $(x_0, f(x_0))$ to

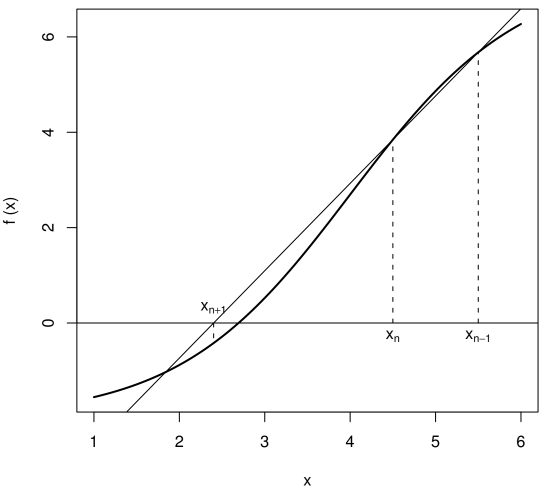

Figure 10.4 *A step in the secant root-finding method.*

$(x_1, f(x_1))$, which is called a secant of the curve $y = f(x)$. Like the tangent, the secant is a linear approximation of the behaviour of $y = f(x)$, in the region of the points x_0 and x_1. As the new guess we will use the x-coordinate x_2 of the point at which the secant crosses the x-axis (Figure 10.4). Now the equation of the secant is given by

$$\frac{y - f(x_1)}{x - x_1} = \frac{f(x_0) - f(x_1)}{x_0 - x_1}$$

and so x_2 can be found from

$$\frac{0 - f(x_1)}{x_2 - x_1} = \frac{f(x_0) - f(x_1)}{x_0 - x_1}$$

which gives

$$x_2 = x_1 - f(x_1)\frac{x_0 - x_1}{f(x_0) - f(x_1)}.$$

Repeating this we get a second-order recurrence relation (each new value depends on the previous two):

Secant method

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

Note that if x_n and x_{n-1} are close together, then

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

so we can view the secant method as an approximation of the Newton-Raphson method, where we substitute $(f(x_n) - f(x_{n-1}))/(x_n - x_{n-1})$ for $f'(x_n)$.

The convergence properties of the secant method are similar to those of the Newton-Raphson method. If f is well behaved at a and you start with x_0 and x_1 sufficiently close to a, then x_n will converge to a quickly, though not quite as fast as the Newton-Raphson method. As for the Newton-Raphson method, we cannot guarantee convergence. Comparing the secant method to the Newton-Raphson method, we see a trade-off: we no longer need to know f' but in return we give up some speed and have to provide two initial points, x_0 and x_1.

The problem of implementing the secant method appears as Exercise 6.

10.5 The bisection method

The Newton-Raphson and secant root-finding methods work by producing a sequence of guesses to the root and, under favourable circumstances, converge rapidly to the root from an initial guess. Unfortunately they cannot be guaranteed to work. A more reliable but slower approach is root-bracketing, which works by first isolating an interval in which the root must lie, and then successively refining the bounding interval in such a way that the root is guaranteed to always lie inside the interval. The canonical example is the bisection method, in which the width of the bounding interval is successively halved.

Suppose that f is a continuous function, then it is easy to see that f has a root in the interval (x_l, x_r) if either $f(x_l) < 0$ and $f(x_r) > 0$ or $f(x_l) > 0$ and $f(x_r) < 0$. A convenient way to verify this condition is to check if $f(x_l)f(x_r) < 0$. The bisection method works by taking an interval (x_l, x_r) that contains a root, then successively refining x_l and x_r until $x_r - x_l \leq \epsilon$, where ϵ is some predefined tolerance. The algorithm is as follows:

```
Bisection method Start with $x_l < x_r$ such that $f(x_l)f(x_r) < 0$.
1. if $x_r - x_l \leq \epsilon$ then stop.
2. put $x_m = (x_l + x_r)/2$; if $f(x_m) = 0$ then stop.
3. if $f(x_l)f(x_m) < 0$ then put $x_r = x_m$ otherwise put $x_l = x_m$.
4. go back to step 1.
```

Note that at every iteration of the algorithm, we know that there is root in the interval (x_l, x_r). Provided we start with $f(x_l)f(x_r) < 0$, the algorithm is guaranteed to converge, with the approximation error reducing by a constant factor $1/2$ at each iteration. If we stop when $x_r - x_l \leq \epsilon$, then we know that both x_l and x_r are within distance ϵ of a root.

Note that the bisection method cannot find a root a if the function f just touches the x-axis at a, that is, if the x-axis is a tangent to the function at a. The Newton-Raphson method will still work in this case. The most popular current root-finding methods use root-bracketing to get close to a root, then switch over to the Newton-Raphson or secant method when it seems safe to do so. This strategy combines the safety of bisection with the speed of the secant method.

Here is an implementation of the bisection method in R. Because this algorithm makes certain assumptions about x_l and x_r, we check that these assumptions hold before the algorithm runs. Also, because the algorithm is guaranteed to converge (provided the initial conditions are met), we do not need to put a bound on the maximum number of iterations. Note that the code has a number of `return` statements. Recall that a function terminates the first time a `return` is executed. In this function, if we detect a problem with the inputs then we print an error message and immediately `return(NULL)`, so that the remainder of the function is not executed.

In Exercise 12 you are asked to generalise `bisection` so that it can deal with the case $f(x_l)f(x_r) > 0$.

```
# program spuRs/resources/scripts/bisection.r
# loadable spuRs function

bisection <- function(ftn, x.l, x.r, tol = 1e-9) {
  # applies the bisection algorithm to find x such that ftn(x) == 0
  # we assume that ftn is a function of a single variable
  #
  # x.l and x.r must bracket the fixed point, that is
  # x.l < x.r and ftn(x.l) * ftn(x.r) < 0
  #
  # the algorithm iteratively refines x.l and x.r and terminates when
  # x.r - x.l <= tol

  # check inputs
  if (x.l >= x.r) {
    cat("error: x.l >= x.r \n")
    return(NULL)
  }
  f.l <- ftn(x.l)
  f.r <- ftn(x.r)
  if (f.l == 0) {
    return(x.l)
  } else if (f.r == 0) {
    return(x.r)
  } else if (f.l * f.r > 0) {
    cat("error: ftn(x.l) * ftn(x.r) > 0 \n")
    return(NULL)
  }
  # successively refine x.l and x.r
  n <- 0
  while ((x.r - x.l) > tol) {
    x.m <- (x.l + x.r)/2
    f.m <- ftn(x.m)
    if (f.m == 0) {
      return(x.m)
    } else if (f.l * f.m < 0) {
      x.r <- x.m
      f.r <- f.m
    } else {
      x.l <- x.m
      f.l <- f.m
    }
    n <- n + 1
    cat("at iteration", n, "the root lies between", x.l, "and", x.r, "\n")
  }

  # return (approximate) root
  return((x.l + x.r)/2)
}
```

Here it is in action. Observe how slow the method is compared to the Newton-Raphson method.

```
> source("../scripts/bisection.r")
> ftn5 <- function(x) return(log(x) - exp(-x))
> bisection(ftn5, 1, 2, tol = 1e-06)

at iteration 1 the root lies between 1 and 1.5
at iteration 2 the root lies between 1.25 and 1.5
at iteration 3 the root lies between 1.25 and 1.375
at iteration 4 the root lies between 1.25 and 1.3125
at iteration 5 the root lies between 1.28125 and 1.3125
at iteration 6 the root lies between 1.296875 and 1.3125
at iteration 7 the root lies between 1.304688 and 1.3125
at iteration 8 the root lies between 1.308594 and 1.3125
at iteration 9 the root lies between 1.308594 and 1.310547
at iteration 10 the root lies between 1.309570 and 1.310547
at iteration 11 the root lies between 1.309570 and 1.310059
at iteration 12 the root lies between 1.309570 and 1.309814
at iteration 13 the root lies between 1.309692 and 1.309814
at iteration 14 the root lies between 1.309753 and 1.309814
at iteration 15 the root lies between 1.309784 and 1.309814
at iteration 16 the root lies between 1.309799 and 1.309814
at iteration 17 the root lies between 1.309799 and 1.309807
at iteration 18 the root lies between 1.309799 and 1.309803
at iteration 19 the root lies between 1.309799 and 1.309801
at iteration 20 the root lies between 1.309799 and 1.309800
[1] 1.309800
```

10.6 Exercises

1. Draw a function $g(x)$ for which the fixed-point algorithm produces the oscillating sequence $1, 7, 1, 7, \ldots$ when started with $x_0 = 7$.
2. (a). Suppose that $x_0 = 1$ and that for $n \geq 0$

$$x_{n+1} = \alpha x_n.$$

Find a formula for x_n. For which values of α does x_n converge, and to what?

(b). Consider the fixed-point algorithm for finding x such that $g(x) = x$:

$$x_{n+1} = g(x_n).$$

Let c be the fixed point, so $g(c) = c$. The first-order Taylor approximation of g about the point c is

$$g(x) \approx g(c) + (x - c)g'(c).$$

Apply this Taylor approximation to the fixed-point algorithm to give a recurrence relation for $x_n - c$.

What condition on the function g at the point c will result in the convergence of x_n to c?

3. Use `fixedpoint` to find the fixed point of $\cos x$. Start with $x_0 = 0$ as your initial guess (the answer is 0.73908513).

Now use `newtonraphson` to find the root of $\cos x - x$, starting with $x_0 = 0$ as your initial guess. Is it faster than the fixed-point method?

4. A picture is worth a thousand words.

The function `fixedpoint_show.r` below is a modification of `fixedpoint` that plots intermediate results. Instead of using the variables `tol` and `max.iter` to determine when the algorithm stops, at each step you will be prompted to enter `"y"` at the keyboard if you want to continue. There are also two new inputs, `xmin` and `xmax`, which are used to determine the range of the plot. `xmin` and `xmax` have defaults `x0 - 1` and `x0 + 1`, respectively.

```
# program spuRs/resources/scripts/fixedpoint_show.r
# loadable spuRs function

fixedpoint_show <- function(ftn, x0, xmin = x0-1, xmax = x0+1) {
  # applies fixed-point method to find x such that ftn(x) == x
  # x0 is the starting point
  # subsequent iterations are plotted in the range [xmin, xmax]

  # plot the function
  x <- seq(xmin, xmax, (xmax - xmin)/200)
  fx <- sapply(x, ftn)
  plot(x, fx, type = "l", xlab = "x", ylab = "f(x)",
    main = "fixed point f(x) = x", col = "blue", lwd = 2)
  lines(c(xmin, xmax), c(xmin, xmax), col = "blue")

  # do first iteration
  xold <- x0
  xnew <- ftn(xold)
  lines(c(xold, xold, xnew), c(xold, xnew, xnew), col = "red")
  lines(c(xnew, xnew), c(xnew, 0), lty = 2, col = "red")

  # continue iterating while user types "y"
  cat("last x value", xnew, " ")
  continue <- readline("continue (y or n)? ") == "y"
  while (continue) {
    xold <- xnew;
    xnew <- ftn(xold);
    lines(c(xold, xold, xnew), c(xold, xnew, xnew), col = "red")
    lines(c(xnew, xnew), c(xnew, 0), lty = 2, col = "red")
    cat("last x value", xnew, " ")
    continue <- readline("continue (y or n)? ") == "y"
  }

  return(xnew)
}
```

Use `fixedpoint_show` to investigate the fixed points of the following functions:

(a). $\cos(x)$ using $x_0 = 1, 3, 6$
(b). $\exp(\exp(-x))$ using $x_0 = 2$
(c). $x - \log(x) + \exp(-x)$ using $x_0 = 2$
(d). $x + \log(x) - \exp(-x)$ using $x_0 = 2$

5. Below is a modification of `newtonraphson` that plots intermediate results, analogous to `fixedpoint_show` above. Use it to investigate the roots of the following functions:

(a). $\cos(x) - x$ using $x_0 = 1, 3, 6$
(b). $\log(x) - \exp(-x)$ using $x_0 = 2$
(c). $x^3 - x - 3$ using $x_0 = 0$
(d). $x^3 - 7x^2 + 14x - 8$ using $x_0 = 1.1, 1.2, \ldots, 1.9$
(e). $\log(x)\exp(-x)$ using $x_0 = 2$.

```
# program spuRs/resources/scripts/newtonraphson_show.r
# loadable spuRs function

newtonraphson_show <- function(ftn, x0, xmin = x0-1, xmax = x0+1) {
  # applies Newton-Raphson to find x such that ftn(x)[1] == 0
  # x0 is the starting point
  # subsequent iterations are plotted in the range [xmin, xmax]
  # plot the function
  x <- seq(xmin, xmax, (xmax - xmin)/200)
  fx <- c()
  for (i in 1:length(x)) {
    fx[i] <- ftn(x[i])[1]
  }
  plot(x, fx, type = "l", xlab = "x", ylab = "f(x)",
    main = "zero f(x) = 0", col = "blue", lwd = 2)
  lines(c(xmin, xmax), c(0, 0), col = "blue")

  # do first iteration
  xold <- x0
  f.xold <- ftn(xold)
  xnew <- xold - f.xold[1]/f.xold[2]
  lines(c(xold, xold, xnew), c(0, f.xold[1], 0), col = "red")

  # continue iterating while user types "y"
  cat("last x value", xnew, " ")
  continue <- readline("continue (y or n)? ") == "y"
  while (continue) {
    xold <- xnew;
    f.xold <- ftn(xold)
    xnew <- xold - f.xold[1]/f.xold[2]
    lines(c(xold, xold, xnew), c(0, f.xold[1], 0), col = "red")
    cat("last x value", xnew, " ")
    continue <- readline("continue (y or n)? ") == "y"
  }

  return(xnew)
}
```

6. Write a program, using both `newtonraphson.r` and `fixedpoint.r` for guidance, to implement the secant root-finding method:

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

First test your program by finding the root of the function $\cos x - x$. Next see how the secant method performs in finding the root of $\log x - \exp(-x)$ using $x_0 = 1$ and $x_1 = 2$. Compare its performance with that of the other two methods.

Write a function `secant_show.r` that plots the sequence of iterates produced by the secant algorithm.

7. Adaptive fixed-point iteration.

To find a root a of f we can apply the fixed-point method to $g(x) = x + cf(x)$, where c is some non-zero constant. That is, given x_0 we put $x_{n+1} = g(x_n) = x_n + cf(x_n)$.

From Taylor's theorem we have

$$\begin{aligned} g(x) &\approx g(a) + (x-a)g'(a) \\ &= a + (x-a)(1+cf'(a)) \end{aligned}$$

so

$$g(x) - a \approx (x-a)(1+cf'(a)).$$

Based on this approximation, explain why $-1/f'(a)$ would be a good choice for c.

In practice we don't know a so cannot find $-1/f'(a)$. At step n of the iteration, what would be your best guess at $-1/f'(a)$? Using this guess for c, what happens to the fixed-point method? (You can allow your guess to change at each step.)

8. The iterative method for finding the fixed point of a function works in very general situations. Suppose $A \subset \mathbb{R}^d$ and $f : A \to A$ is such that for some $0 \le c < 1$ and all vectors $\mathbf{x}, \mathbf{y} \in A$,

$$\|f(\mathbf{x}) - f(\mathbf{y})\|_d \le c\|\mathbf{x} - \mathbf{y}\|_d.$$

It can be shown that for such an f there is a unique point $\mathbf{x}_* \in A$ such that $f(\mathbf{x}_*) = \mathbf{x}_*$. Moreover for any $\mathbf{x}_0 \in A$, the sequence defined by $\mathbf{x}_{n+1} = f(\mathbf{x}_n)$ converges to $\mathbf{x}_*$. Such a function is called a *contraction mapping*, and this result is called the *contraction mapping theorem*, which is one of the fundamental results in the field of *functional analysis*.

Modify the function `fixedpoint(ftn, x0, tol, max.iter)` given in Section 10.2, so that it works for any function `ftn(x)` that takes as input a numerical vector of length $d \ge 1$ and returns a numerical vector of length d. Use your modified function to find the fixed points of the function f below, in the region $[0,2] \times [0,2]$.

$$f(x_1, x_2) = (\log(1 + x_1 + x_2), \log(5 - x_1 - x_2)).$$

9. For $f : \mathbb{R} \to \mathbb{R}$, the Newton-Raphson algorithm uses a sequence of linear approximations to f to find a root. What happens if we use quadratic approximations instead?

Suppose that x_n is our current approximation to f, then a quadratic approximation to f at x_n is given by the second-order Taylor expansion:

$$f(x) \approx g_n(x) = f(x_n) + (x - x_n)f'(x_n) + \tfrac{1}{2}(x - x_n)^2 f''(x_n).$$

Let x_{n+1} be the solution of $g_n(x) = 0$ that is closest to x_n, assuming a solution exists. If $g_n(x) = 0$ has no solution, then let x_{n+1} be the point at which g_n attains either its minimum or maximum. Figure 10.5 illustrates the two cases.

Implement this algorithm in R and use it to find the fixed points of the following functions:

(a). $\cos(x) - x$ using $x_0 = 1, 3, 6$.

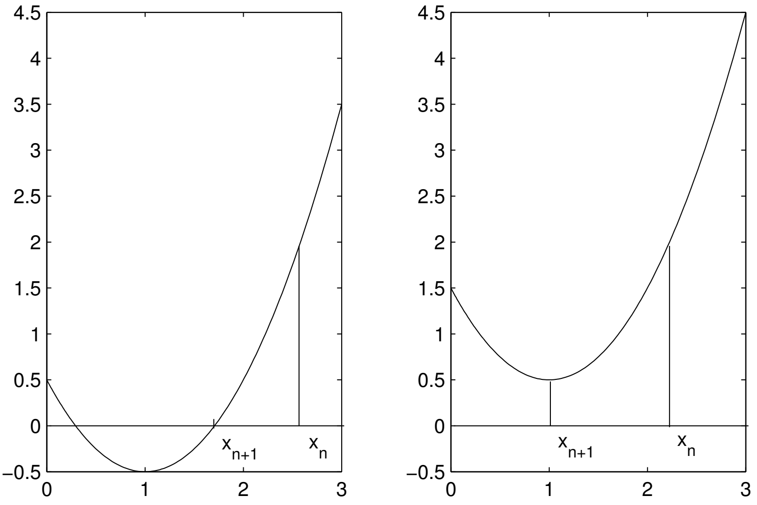

Figure 10.5 *The iterative root-finding scheme of Exercise 9*

(b). $\log(x) - \exp(-x)$ using $x_0 = 2$.

(c). $x^3 - x - 3$ using $x_0 = 0$.

(d). $x^3 - 7x^2 + 14x - 8$ using $x_0 = 1.1, 1.2, \ldots, 1.9$.

(e). $\log(x)\exp(-x)$ using $x_0 = 2$.

For your implementation, assume that you are given a function `ftn(x)` that returns the vector $(f(x), f'(x), f''(x))$. Given x_n, if you rewrite g_n as $g_n(x) = a_2x^2 + a_1x + a_0$ then you can use the formula $(-a_1 \pm \sqrt{a_1^2 - 4a_2a_0})/2a_2$ to find the roots of g_n and thus x_{n+1}. If g_n has no roots then the min/max occurs at the point $g_n'(x) = 0$.

How does this algorithm compare to the Newton-Raphson algorithm?

10. How do we know $\pi = 3.1415926$ (to 7 decimal places)? One way of finding π is to solve $\sin(x) = 0$. By definition the solutions to $\sin(x) = 0$ are $k\pi$ for $k = 0, \pm 1, \pm 2, \ldots$, so the root closest to 3 should be π.

(a). Use a root-finding algorithm, such as the Newton-Raphson algorithm, to find the root of $\sin(x)$ near 3. How close can you get to π? (You may use the function `sin(x)` provided by R.)

The function $\sin(x)$ is *transcendental*, which means that it cannot be written as a rational function of x. Instead we have to write it as an infinite sum:

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

(This is the infinite order Taylor expansion of $\sin(x)$ about 0.) In practice, to calculate $\sin(x)$ numerically we have to truncate this sum, so any numerical calculation of $\sin(x)$ is an approximation. In particular the function `sin(x)` provided by R is only an approximation of $\sin(x)$ (though a very good one).

(b). Put

$$f_n(x) = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

Write a function in R to calculate $f_n(x)$. Plot $f_n(x)$ over the range $[0, 7]$ for a number of values of n, and verify that it looks like $\sin(x)$ for large n.

(c). Choose a large value of n, then find an approximation to π by solving $f_n(x) = 0$ near 3. Can you get an approximation that is correct up to 6 decimal places? Can you think of a better way of calculating π?

11. The astronomer Edmund Halley devised a root-finding method faster than the Newton-Raphson method, but which requires second derivative information. If x_n is our current solution then

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

Let m be a positive integer. Show that applying Halley's method to the function $f(x) = x^m - k$ gives

$$x_{n+1} = \left(\frac{(m-1)x_n^m + (m+1)k}{(m+1)x_n^m + (m-1)k} \right) x_n.$$

Use this to show that, to 9 decimal places, $59^{1/7} = 1.790518691$.

12. The bisection method can be generalised to deal with the case $f(x_l)f(x_r) > 0$, by *broadening* the bracket. That is, we reduce x_l and/or increase x_r, and try again. A reasonable choice for broadening the bracket is to double the width of the interval $[x_l, x_r]$, that is (in pseudo-code)

```
$m \leftarrow (x_l + x_r)/2$
$w \leftarrow x_r - x_l$
$x_l \leftarrow m - w$
$x_r \leftarrow m + w$
```

Incorporate bracket broadening into the function `bisection` given in Section 10.5. Note that broadening is *not guaranteed* to find x_l and x_r such that $f(x_l)f(x_r) \le 0$, so you should include a limit on the number of times it can be tried.

Use your modified function to find a root of

$$f(x) = (x-1)^3 - 2x^2 + 10 - \sin(x),$$

starting with $x_l = 1$ and $x_r = 2$.