Chapter 3

Iterative search methods

3.1 Limitations of algebra and calculus

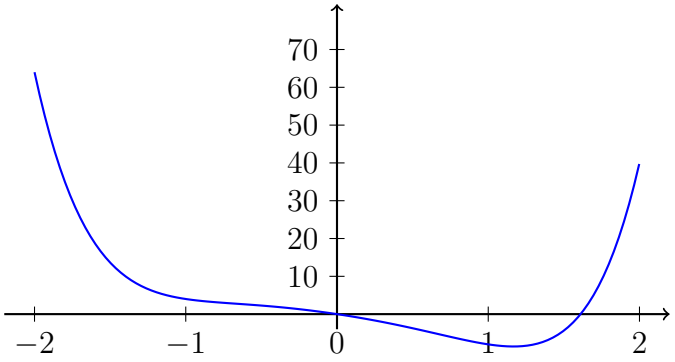
Often, it is nice to have closed-form solutions to a problem. For example, the closed-form solutions for regression modelling used in Chapter 2 are straightforward to implement and reasonably efficient.

Although a noble goal, in some cases, a closed-form solution can be difficult, computationally demanding, or even impossible. We will not dwell on the precise meaning of "closed-form", but roughly speaking, it means an expression that can be written down using everyday functions. The expression $\mathbf{a} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}$ employed in Chapter 2 is an example.

To see this idea concretely, suppose you wanted to find a minimiser of

$$f(x) = x^6 - 3x^2 - 6x.$$

Its graph is shown below on the interval $[-2, 2]$.



A cursory inspection of the graph reveals what looks like a local minimiser somewhere in the interval $[1, 2]$. We may attempt to find the minimiser using calculus:

$$\begin{aligned} f'(x) = 0 &\iff 6x^5 - 6x - 6 = 0 \\ &\iff x^5 - x - 1 = 0. \end{aligned}$$

Now all that needs to be done is to solve this equation to find the minimiser.

Unfortunately, this is not possible. A major result in mathematics known as the *Abel-Ruffini theorem* tells us that the solution to the equation $x^5 - x - 1 = 0$ cannot be expressed using arithmetic, powers and roots. So, no matter how hard you try, it is not possible to find a "closed-form" solution to this equation. In fact, this property applies to almost all polynomials of degree 5 or more. This alone is enough to suggest that algebraic techniques are not always applicable.

Let's consider one more example in the context of regression. In Section 2.4, we attempted to fit a small set of data to the function $f(x) = ae^{bx}$. In doing so, we arrived at the following equations:

$$\begin{aligned} (a-5)e^b + ae^{3b} + ae^{5b} - 6e^{2b} &= 4, \\ (a-10)e^b + 2ae^{3b} + 3ae^{5b} - 18e^{2b} &= 4. \end{aligned}$$

Solving this is not completely intangible: let $z = e^b$, so that the equations become:

$$\begin{aligned} (a-5)z + az^3 + az^5 - 6z^2 &= 4, \\ (a-10)z + 2az^3 + 3az^5 - 18z^2 &= 4. \end{aligned}$$

If one could solve polynomials of degree 5 or more, then the first expression could be used to find z in terms of a, and then the second could be solved for a. Alas, by the Abel-Ruffini theorem, we should not expect it to be possible. By the end of this chapter, you will see how to obtain $a \approx 3.307$ and $b \approx 0.200$.

Before continuing, let's take a slightly more in-depth look at the regression models we obtained.

In general, to fit a function $f(\mathbf{x})$ to a set of data points without linearisation, the objective function is the sum of squared residuals,

$$\|\mathbf{y} - f(\mathbf{x})\|^2, \tag{3.1}$$

whereas with linearisation, the objective function is

$$\|\ln(\mathbf{y}) - \ln(f(\mathbf{x}))\|^2. \tag{3.2}$$

where $f(\mathbf{x})$ denotes the n-vector with components $f(\mathbf{x})_i = f(x_i)$. The minimiser for the objective function (3.1) can be quite different from the minimiser for objective function (3.2).

We will start by observing the two approaches using the same data set as in Chapter 2:

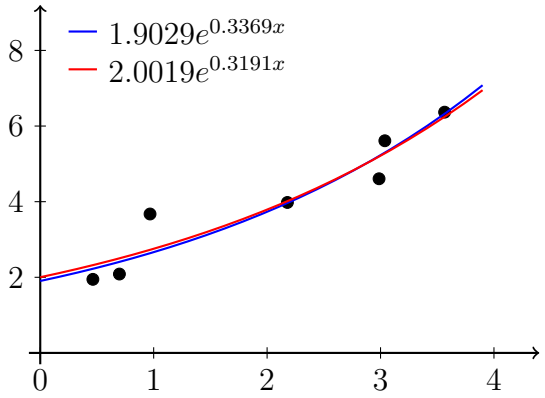
i	1	2	3	4	5	6	7
x_i	0.464	0.698	0.968	2.179	2.987	3.038	3.565
y_i	1.946	2.086	3.674	3.979	4.606	5.611	6.367

Our first attempt was to fit the data to a function of the form

$$f(x) = ae^{bx}.$$

In Chapter 2, by applying linearisation to minimise (3.2), we found $a \approx 1.9029$ and $b \approx 0.3369$. If we use the methods of this chapter and minimise (3.1) directly, we will find $a \approx 2.0019$ and $b \approx 0.3191$.

The two functions are plotted below.



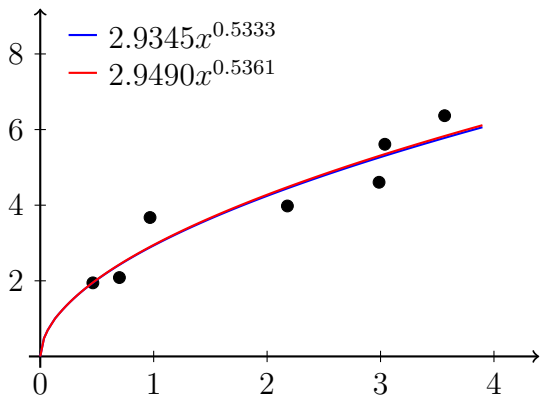
They are quite similar, but still distinct, and there may be reason to prefer one over the other.

If, instead, we were to fit a power function

$$f(x) = ax^b,$$

as in Chapter 2, the coefficients we find by employing linearisation are $a \approx 2.9345$ and $b \approx 0.5333$. and if were we to minimise (3.1) instead, we would find $a \approx 2.9490$ and $b \approx 0.5361$.

Once again, these are quite similar, which can be seen in the plot below.



On the other hand, consider the following data set:

i	1	2	3	4	5	6	7	8	9	10
x_i	0.488	0.635	1.393	2.734	3.162	4.074	4.529	4.567	4.788	4.824
y_i	2.182	3.185	4.216	3.186	3.216	4.396	8.191	9.095	11.795	12.514

Suppose we wish to fit this data to a power function

$$f(x) = ax^b,$$

so we need to find the coefficients a and b.

By using linearisation, we find that the power function that minimises (3.2) is

$$p_1(x) = 3.2001x^{0.5465},$$

whereas, without linearisation, the power function that minimises (3.1) is

$$p_2(x) = 0.0608x^{3.3226}.$$

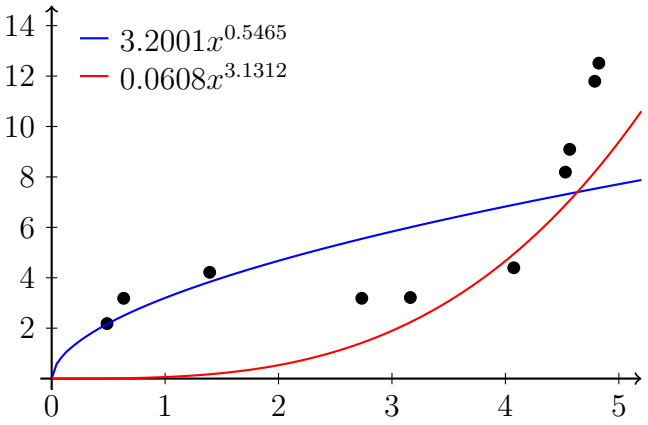
These are quite different, as are the values of each objective function. We have

$$\|\mathbf{y} - p_1(\mathbf{x})\|^2 \approx 66.7645, \quad \|\mathbf{y} - p_2(\mathbf{x})\|^2 \approx 40.7457,$$

and

$$\|\ln(\mathbf{y}) - \ln(p_1(\mathbf{x}))\|^2 \approx 1.4815, \quad \|\ln(\mathbf{y}) - \ln(p_2(\mathbf{x}))\|^2 \approx 75.8938.$$

The graphs of the functions $p_1(x)$ and $p_2(x)$ are displayed below.



This illustrates that the solutions to the linearised problems are not necessarily close to the 'real' solutions.

3.2 Iterative algorithms

The main idea of an iterative algorithm is to repeat an operation or function to approximate a solution until a desired level of tolerance is reached. This concept is not limited to optimisation problems. For example, here is an iterative approach to calculate a decimal approximation of $\sqrt{2}$:

- Start with an initial estimate of $x_0 = 1$.
- Repeatedly calculate $x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right)$.
- Stop once the desired accuracy is obtained.

The essential feature of this procedure is that

$$\lim_{n\to\infty} x_n = \sqrt{2},$$

so that each value of x_n is an approximation to $\sqrt{2}$. The table below shows the first 5 iterations.

n	Calculation	Result (8 d.p.)
0	—	1
1	$\frac{1}{2}\left(1 + \frac{2}{1}\right)$	1.5
2	$\frac{1}{2}\left(1.5 + \frac{2}{1.5}\right)$	1.41666667
3	$\frac{1}{2}\left(1.41666667 + \frac{2}{1.41666667}\right)$	1.41421569
4	$\frac{1}{2}\left(1.41421569 + \frac{2}{1.41421569}\right)$	1.41421356
5	$\frac{1}{2}\left(1.41421356 + \frac{2}{1.41421356}\right)$	1.41421356

We find that the first 8 decimal places for iterations 4 and 5 are the same, so we might consider this a reasonable stopping point.

More often, the stopping condition will be defined in terms of the difference between successive iterations. For example, "stop when $|x_{n+1} - x_n| < 10^{-5}$", which means to stop when the difference between two iterations is less than 0.00001. In general, if a procedure is defined to stop when $|x_{n+1} - x_n| < \varepsilon$, then ε is called the **absolute tolerance** for the procedure.

When using a tolerance, the algorithm is best handled using a while loop. Here is how we might present the algorithm for approximating $\sqrt{2}$ using the absolute tolerance as an input value:

Input: desired tolerance, *tolerance*
Output: an approximation to $\sqrt{2}$
Algorithm:
    let $previous = \infty$
    let $estimate = 1$
    while $|estimate - previous| \geqslant tolerance$ **do**
        let $previous = estimate$
        let $estimate = \frac{1}{2}(previous + \frac{2}{previous})$
    end while
    return *estimate*

Alternatively, one might consider a specified number of iterations. A for loop is most suitable to apply a given number of iterations. Here is how we might present the algorithm for approximating $\sqrt{2}$ using the number of iterations as an input value:

Input: number of iterations, m
Output: an approximation to $\sqrt{2}$
Algorithm:
let $estimate = 1$
for $i = 1$ **to** m
let $estimate = \frac{1}{2}(estimate + \frac{2}{estimate})$
end for
return $estimate$

You might like to think about how to adjust these to stop when the desired tolerance is reached *or* the number of iterations has been applied, whichever comes first.

This algorithm is actually an application of Newton's method to finding a root of the function $f(x) = x^2 - 2$ (see Section 3.4). A more general approach to approximate $\sqrt{s}$ for any $s \geqslant 0$ is:

Input: a non-negative number s; the desired tolerance, *tolerance*
Output: an approximation to $\sqrt{s}$
Algorithm:
let $previous = \infty$
let $estimate = 1$
while $|estimate - previous| \geqslant tolerance$ **do**
let $previous = estimate$
let $estimate = \frac{1}{2}(previous + \frac{s}{previous})$
end while
return $estimate$

To avoid an explicit description of the stopping condition, we will often describe an algorithm using a repeat statement, like so:

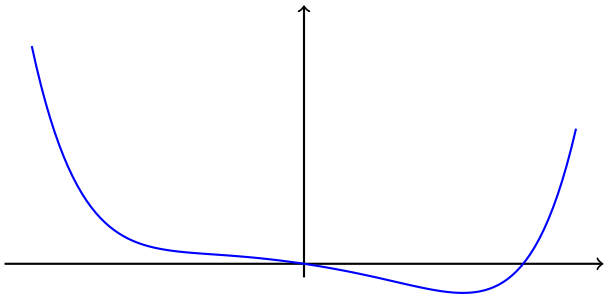
Input: a non-negative number s
Output: an approximation to $\sqrt{s}$
Algorithm:
let $estimate = 1$
repeat
let $estimate = \frac{1}{2}(previous + \frac{s}{previous})$
until the desired accuracy is obtained
return $estimate$

In that case, the exit condition of the loop must be decided at the time of implementation. Practice your MATLAB skills by implementing each of these algorithms as MATLAB functions.

It should also be noted that stopping conditions for higher dimensional approximations need more careful consideration; this will be discussed later.

3.3 The golden section and Fibonacci methods

For this section, consider a function $f\colon \mathbb{R} \to \mathbb{R}$ which is **unimodal** on an interval $[a, d] \subseteq \mathbb{R}$. That is, there is *exactly one* minimiser of f in the interval $[a, d]$.



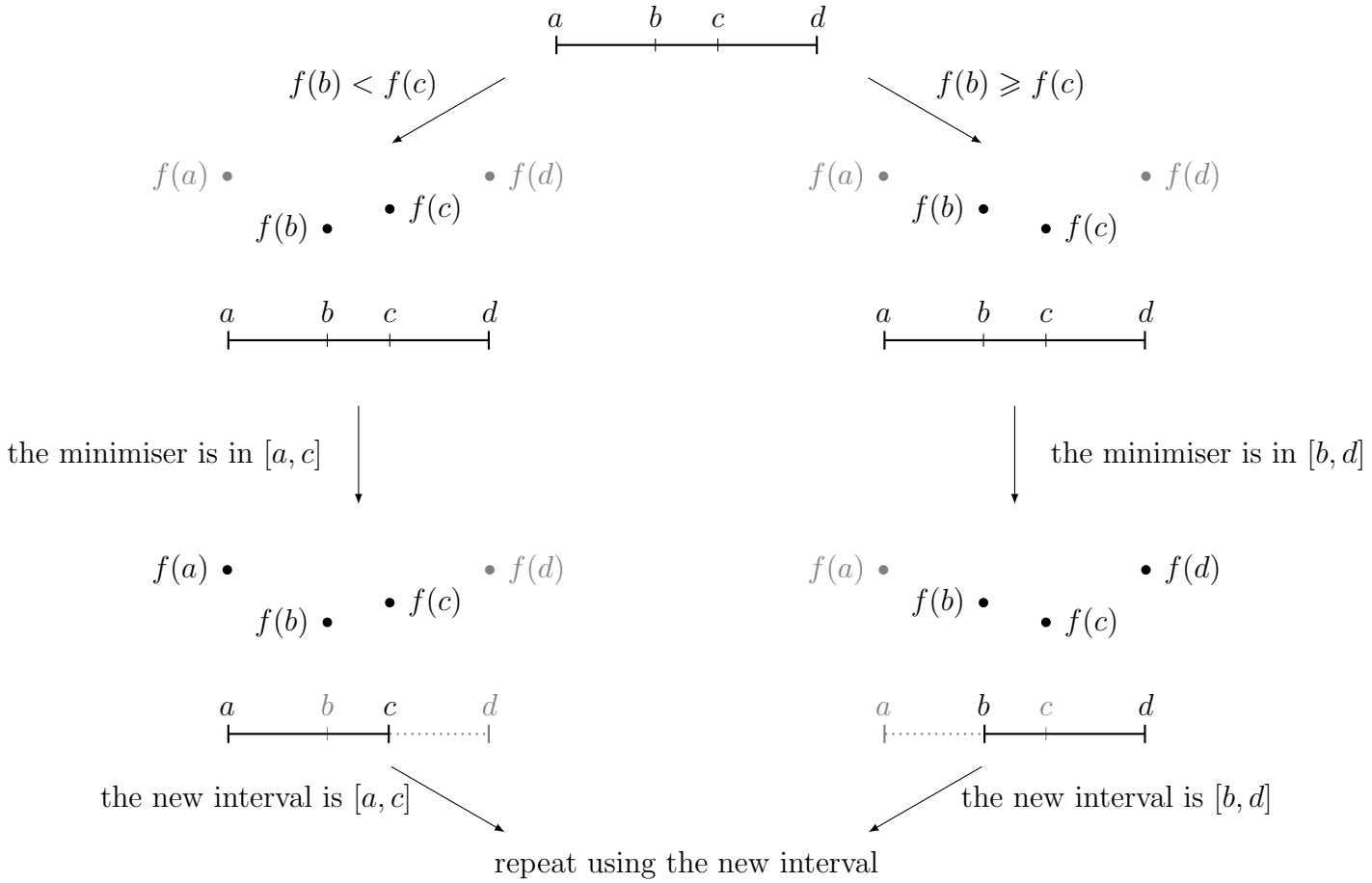
We aim to start with the interval $[a, d]$, and then set up a sequence of smaller and smaller intervals that contain the minimiser. The main idea is simple: first construct two smaller intervals, one of which is guaranteed to contain the minimiser. For the next iteration, choose the interval that is known to contain the minimiser. There are two questions that need to be addressed:

- What is the best way to construct the smaller intervals?
- How do we determine which interval contains the minimiser?

The basis for the algorithms in this section will be to choose two points b and c in the interval $[a, d]$, with $b < c$, and take as candidates the two (overlapping) intervals $[a, c]$ and $[b, d]$. One of two cases must occur:

- $f(b) < f(c)$, in which case the minimiser is in the interval $[a, c]$, or
- $f(b) \geqslant f(c)$, in which case the minimiser is in the interval $[b, d]$.

See below for a digram.



This forms the basis for the algorithms in this section:

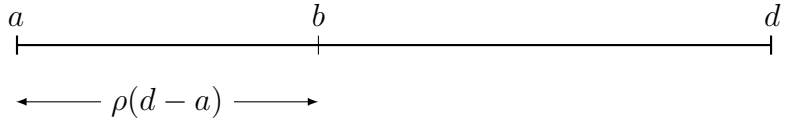
Input: a and d with $a < d$; function f unimodal on $[a, d]$
Output: an interval containing the minimiser of f over $[a, d]$
Algorithm:
  repeat
    let b and c be points in (a, d) with $b < c$
    if $f(b) < f(c)$
      let $d = c$
    else
      let $a = b$
    end if
  until the desired accuracy is obtained
  return $[a, d]$

To determine how one chooses the points b and c, we will make one core decision: we aim to minimise the number of function evaluations by re-using as many possible of the points a, b, c and d at each iteration. That is, after evaluating $f(b)$ and $f(c)$,

- if $f(b) < f(c)$, keep a and replace d with c; also replace c with b and find a new value of b.
- if $f(b) \geqslant f(c)$, keep d and replace a with b; also replace b with c and find a new value of c.

The result of this is that, with suitable implementation, only one new function evaluation is required at each step.

For the first method, known as the **golden section method**, a constant factor $\rho < \frac{1}{2}$ is chosen so that, at each step, the point b is positioned at a proportional distance ρ to the right of a.

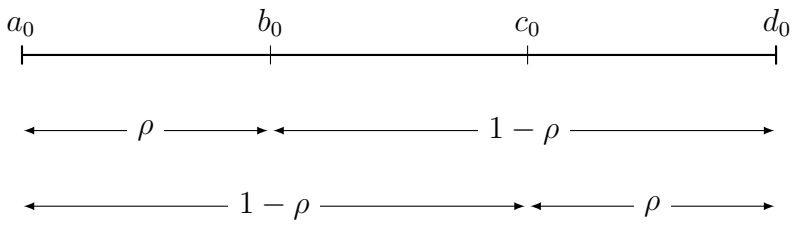


We will now show that there is just one reasonable choice of ρ.

Let a_0, b_0, c_0 and d_0 denote the initial values of the variables a, b, c and d. The values of a_0 and d_0 are given as input, but the values of b_0 and d_0 are determined by the algorithm. Since there is no a priori reason to favour one interval over another, we would like both candidate intervals $[a_0, c_0]$ and $[b_0, d_0]$ to be the same size. So, define b_0 and c_0 by

$$b_0 = a_0 + \rho(d_0 - a_0),$$
$$c_0 = d_0 - \rho(d_0 - a_0).$$

Note also that we have $c_0 - a_0 = (1 - \rho)(d_0 - a_0)$.



Let a_1, b_1, c_1 and d_1 denote the values of the variables a, b, c and d at the next iteration. Since we have decided that ρ is constant, the next iterates of b and c are:

$$\begin{aligned} b_1 &= a_1 + \rho(d_1 - a_1), \\ c_1 &= d_1 - \rho(d_1 - a_1). \end{aligned}$$

First assume that $f(b_0) < f(c_0)$, giving $a_1 = a_0$ and $d_1 = c_0$. To minimise the number of function evaluations, we demand that $c_1 = b_0$, which gives

$$\begin{aligned} & c_1 = b_0 \\ \iff \quad & d_1 - \rho(d_1 - a_1) = a_0 + \rho(d_0 - a_0) && \text{(by definition of } b_0 \text{ and } c_1) \\ \iff \quad & c_0 - \rho(c_0 - a_0) = a_0 + \rho(d_0 - a_0) && \text{(since } d_1 = c_0) \\ \iff \quad & (1-\rho)(c_0 - a_0) = \rho(d_0 - a_0) \\ \iff \quad & (1-\rho)^2(d_0 - a_0) = \rho(d_0 - a_0) && \text{(since } c_0 - a_0 = (1-\rho)(d_0 - a_0)) \\ \iff \quad & (1-\rho)^2 = \rho \\ \iff \quad & \rho^2 - 3\rho + 1 = 0. \end{aligned}$$

So $\rho = \frac{3 \pm \sqrt{5}}{2}$, but taking the positive root will make ρ larger than 1, so we discard that option. Hence,

$$\rho = \frac{3 - \sqrt{5}}{2} \approx 0.381966.$$

A similar computation instead assuming $f(b_0) \geqslant f(c_0)$ will yield the same value of ρ, which completes all we need for the golden section method:

Input: a and d with $a < d$; function f unimodal on $[a, d]$
Output: an interval containing the minimiser of f over $[a, d]$.
Algorithm:
    let $\rho = \frac{3-\sqrt{5}}{2}$
    repeat
        let $b = a + \rho(d - a)$
        let $c = d - \rho(d - a)$
        if $f(b) < f(c)$
            let $d = c$
        else
            let $a = b$
        end if
    until the desired accuracy is obtained
    return $[a, d]$

The version of the algorithm above has not actually implemented the reduced number of function evaluations, since the if statement makes two evaluations of f (evaluating $f(b)$ and $f(c)$); we leave that implementation as an exercise.

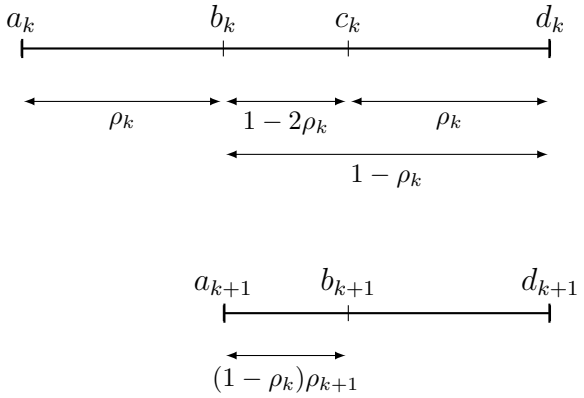
At each step, the interval decreases in length by a factor $\phi = \frac{1}{1-\rho}$ which is the **golden ratio**. In total, this means that after N iterations, the size of the interval is $(1-\rho)^N$ times the size of the initial interval. This can be used to determine the number of steps needed to find the minimiser to a given accuracy.

In contrast to the golden section method, which leaves ρ constant, for the **Fibonacci method**, we allow ρ to vary at each iteration.

Let ρ_k denote the fractional length used at each iteration, so that b_k and c_k are given by

$$b_k = a_k + \rho_k(d_k - a_k),$$
$$c_k = d_k - \rho_(d_k - a_k).$$

The diagram below depicts the relationship between the fractional distance ρ_k and the fractional distance ρ_{k+1}.



From the diagram above, one can deduce that ρ_{k+1} is related to ρ_k by

$$1 - 2\rho_k = (1 - \rho_k)\rho_{k+1} \implies \rho_{k+1} = \frac{1 - 2\rho_k}{1 - \rho_k} = 1 - \frac{\rho_k}{1 - \rho_k}.$$

For a given number of iterations N, the size of the interval after N steps is

$$(1 - \rho_1)(1 - \rho_2) \cdots (1 - \rho_N)(d - a).$$

So, to minimise the size of the final interval, we have the following optimisation problem:

$$\begin{aligned} \text{minimise} \quad & R = (1 - \rho_1)(1 - \rho_2) \cdots (1 - \rho_N) \\ \text{subject to} \quad & \rho_{k+1} = 1 - \tfrac{\rho_k}{1 - \rho_k}, \qquad \text{for each } 1 \leqslant k \leqslant N, \\ & 0 \leqslant \rho_k \leqslant \tfrac{1}{2}, \qquad \text{for each } 1 \leqslant k \leqslant N. \end{aligned}$$

It is shown in Chong and Żak [3] that the solution to this is

$$\rho_k = 1 - \frac{f_{N-k+2}}{f_{N-k+3}},$$

where f_i is the i-th **Fibonacci number**, defined by $f_1 = f_2 = 1$ and $f_{i+1} = f_i + f_{i-1}$. This results in $R = 1/f_{N+2}$, which is better than $R = (1-\rho)^N$ for the golden section method, but requires N to be chosen in advance.

There is also an anomaly in the final iteration that requires replacing $\rho_N = \frac{1}{2}$ with $\rho_N = \frac{1}{2} - \varepsilon$, where ϵ is small. This is because taking $\rho_N = \frac{1}{2}$ results in $b = c$, which means there are not enough points to perform a comparison.

3.4 Root-finding methods

A **root** of a function $f\colon \mathbb{R} \to \mathbb{R}$ is a value $x \in \mathbb{R}$ such that $f(x) = 0$. If f is continuous on an interval $[a, b]$ such that $f(a)$ and $f(b)$ have opposite signs, then there is at least one root of f in the interval $[a, b]$. Under those conditions, a **root-finding method** is an algorithm used to locate one of those roots. This can be combined with the FONC, which imples that if m is a minimiser of a differentiable function $f\colon \mathbb{R} \to \mathbb{R}$, then $f'(m) = 0$. Thus, if the derivative of f is continuous, then root-finding methods can be applied to find a root of f', and this can then be refined to find a local minimiser.

The **bisection method** is a relatively simple root-finding method: start with a function f and an interval $[a, c]$ for which $f(a)$ and $f(c)$ have oposite signs. Calculate the midpoint $b = (a + c)/2$, then:

- if $f(a)$ and $f(b)$ have opposite signs, the next iteration is $[a, b]$,
- if $f(b)$ and $f(c)$ have opposite signs, the next iteration is $[b, c]$,
- if $f(b) = 0$ then b is a root of f.

A slight adjustment of the bisection method is more suitable for finding a minimiser. As in the previous section, we assume that $f\colon \mathbb{R} \to \mathbb{R}$ is unimodal on an interval $[a, c] \subseteq \mathbb{R}$, and now assume further that f is **continuously differentiable** (i.e., it has a continuous derivative). Calculate the midpoint $b = (a + c)/2$, and then:

- if $f'(b) < 0$ then f is decreasing at b, so the minimiser is in $[b, c]$,
- if $f'(b) > 0$ then f is increasing at b, so the minimiser is in $[a, b]$,
- if $f'(b) = 0$ then the minimiser is b.

The bisection method for finding a minimiser is:

Input: a and c with $a < c$; continuously differentiable function f unimodal on $[a, c]$
Output: an interval containing the minimiser of f over $[a, c]$
Algorithm:
    repeat
        let $b = (a + c)/2$
        if $f'(b) < 0$
            let $a = b$
        else if $f'(b) > 0$
            let $c = b$
        else
            return $[b, b]$
        end if
    until the desired accuracy is obtained
    return $[a, c]$

Since the size of the interval is being halved at each step, after N iterations, the size of the interval is $(1/2)^N$ times the size of the initial interval. This is an improvement over the Fibonacci method, as $(1/2)^N < 1/f_{N+2}$, but the Fibonacci method does not depend on any derivatives.

Up next is **Newton's method**, which is based on the first order Taylor approximation of f,

$$f(x + h) \approx f(x) + hf'(x).$$

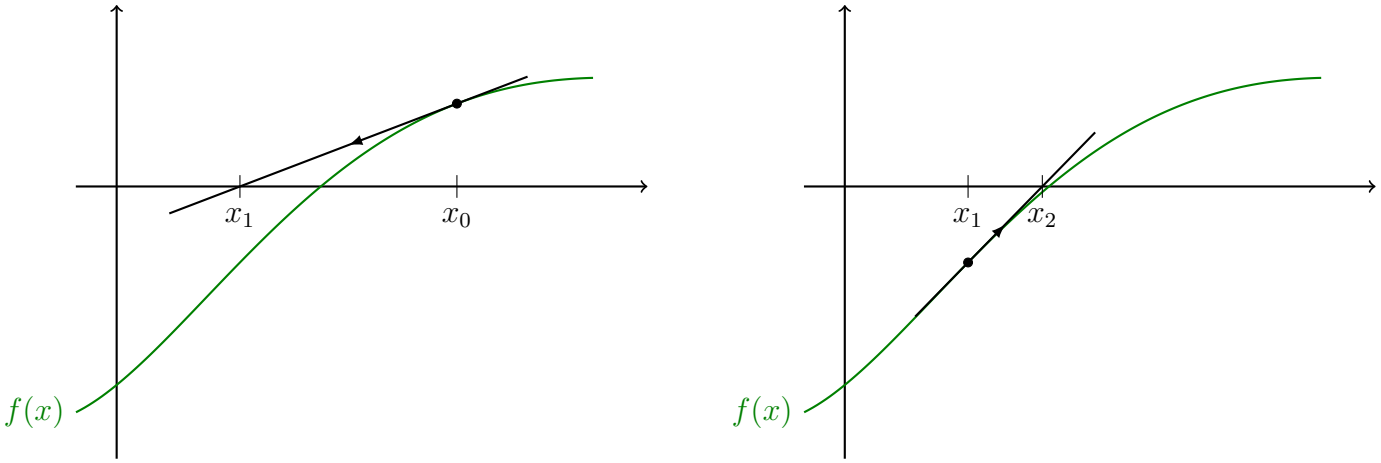
To find a *root*, we set the right-hand side of this expression to 0 and solve for h, giving

$$f(x) + hf'(x) = 0 \implies h = -\frac{f(x)}{f'(x)} \implies x + h = x - \frac{f(x)}{f'(x)}.$$

Defining x_{n+1} as $x_n + h$ results in the update procedure:

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

The value x_0 must also be given as input, which is an initial 'guess' for the root. Graphically, the method can be seen as iteratively moving towards the x-axis along tangent lines of the graph of $f(x)$.



To convert this to an optimisation algorithm, the function must be twice differentiable, as the derivative will be in the numerator and the second derivative in the denominator of the term being subtracted. That is, to find a local minimiser, we will compute:

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

However, because this amounts to solving $f'(x) = 0$ without any further caveats, Newton's method may find a minimiser, a maximiser, or a stationary point of inflection. For this reason, it is best employed when it is known that the function is convex, implying that it has no local maximisers nor stationary points of inflection.

Newton's method for finding a local minimiser is:

Input: an initial guess x; twice differentiable function f
Output: an approximation to a local minimiser of f
Algorithm:
    repeat
        let $x = x - f'(x)/f''(x)$
    until the desired accuracy is obtained
    return x

Example: Newton's method in MATLAB

Let $f\colon \mathbb{R} \to \mathbb{R}$ be given by

$$f(x) = x^4 - 5x + 3.$$

Define a function file `f.m` containing:

```
function [y,dydx,dydx2] = f(x)
y = x^4 -5*x+3;
dydx = 4*x^3 - 5;
dydx2 = 12*x^2;
end
```

A script that implements the iterative procedure with 100 iterations is:

```
x(1) = 2;
M = 100;
for n = 1:M
    [f,fp,fpp]=f(x(n));
    x(n+1) = x(n) - fp/fpp;
end
fprintf('The minimiser is %f\n', x(M+1));
```

This will return a numerical approximation for $(5/4)^{1/3} \approx 1.07722$, as expected. Can you make the implementation more memory efficient?

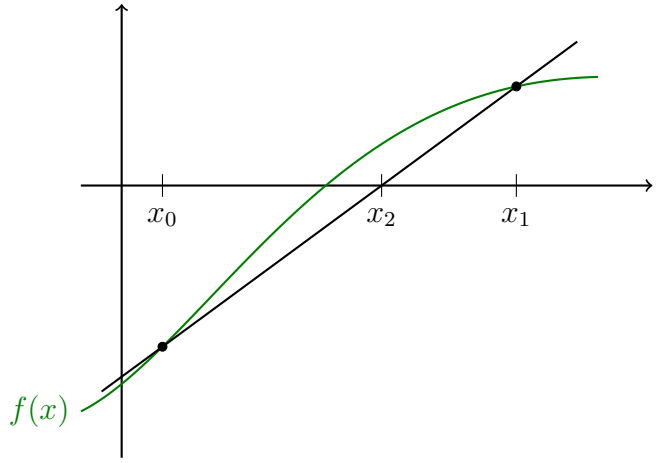
The **secant method** is similar to Newton's method, but it uses an approximation of the first derivative,

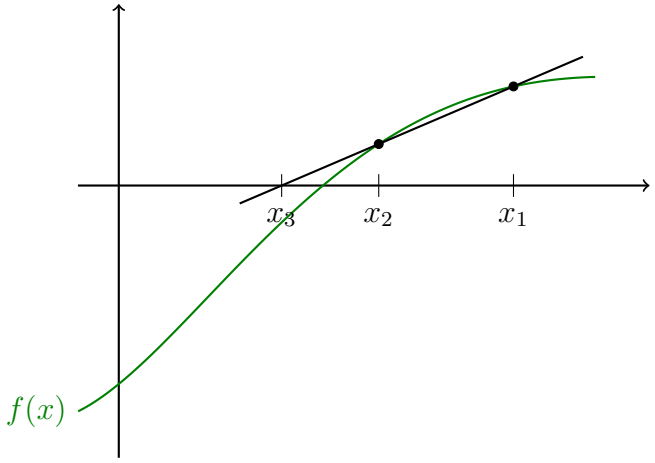
$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

Substituting this into the update rule for Newton's method gives

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

Observe that this no longer relies on computing the derivative of f, but does rely on the previous two iterations to compute x_{n+1}. Graphically, this method can be viewed as taking successive lines between points on the graph of $f(x)$, and using the intersection of that line with the x-axis as the next point.





To apply the secant method to minimisation, simply replace $f(x)$ with $f'(x)$. Note that we need only compute the first derivative, while Newton's method also needs the second derivative.

The secant method for minimisation is then:

Input: two initial points x_0 and x_1; continuously differentiable function f
Output: an approximation to a local minimiser of f
Algorithm:

repeat
let $x_{new} = x_1 - f'(x_1) \cdot (x_1 - x_0)/(f'(x_1) - f'(x_0))$
let $x_0 = x_1$
let $x_1 = x_{new}$
until the desired accuracy is obtained
return x_1