Notes 4: Root finding and optimisation

4.1 Root-finding in one dimension

Given a function, $f(x)$, of a single scalar variable root-finding in one dimension is the task of finding the value or values of x which satisfy the equation

$$f(x) = 0. \tag{4.1}$$

Since most equations cannot be solved analytically, numerical approaches are essential. It is important to remember however that since "most" real numbers do not have exact floating-point representations, we are really interested in finding values of x for which Eq. (4.1) is satisfied approximately in the sense that

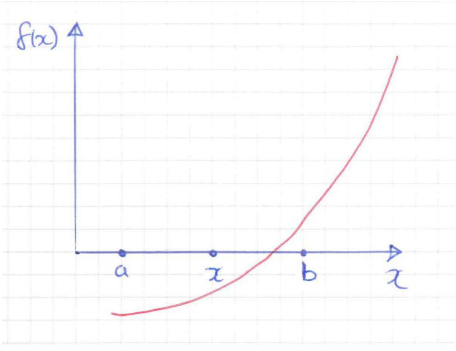
$$|f(x)| < \epsilon_{\rm tol} \tag{4.2}$$

for some pre-specified error tolerance, $\epsilon_{\rm tol}$. All root-finding methods rely on making successive improvements to a reasonable initial guess, x_0, for the position, x_*, of the root. After all, $\mathbb{R}$ is a big place. A root is said to be *bracketed* by the interval $[a, b]$ if $f(a)f(b) < 0$, that is to say, $f(x)$ has different sign at the two ends of the interval. For continuous functions (and as far as floating-point arithmetic is concerned *all* functions are continuous), the Intermediate Value Theorem assures the existence of a root somewhere in the interval $[a, b]$. The first step is always to find an interval which brackets the root you are seeking to find. [1] Any reasonable initial guess for the position of the root should be in this interval. Therefore you should always start by plotting your function. This is true for most analytic tasks but especially so for root finding.

There are many algorithms for root finding (and for the related task of finding extremal values fo functions) although we shall only consider a few of them in this module. These algorithms can be usefully divided into two classes: those which use the derivative of the function $f(x)$ and those which do not. Algorithms which use derivatives tend to converge faster but those which do not tend to be more robust. Convergence time is rarely an issue for one-dimensional root finding on modern computers (not necessarily so for higher dimensions) so the increased speed of derivative methods often does not outweigh the sure-footedness of their less sophisticated non-derivative cousins. In any case, it is common to need to analyse functions for which no analytic form for the derivative is known, for example functions which are defined as the outputs of some numerical or combinatorial function.

4.1.1 Derivative-free methods: bracket and bisection method, Brent's method

We start with methods which do not require any knowledge of the derivative. The *bracketing-and-bisection* algorithm is the most basic root finding method. It involves making successive refinements of the bracketing interval by testing the sign of $f(x)$ at the midpoint of the current interval and then defining a new bracketing interval in the obvious way. At each step, the "best guess" of the position of the root is the midpoint, x:



[1] Note that not every root can be bracketed - a simple counter example is $f(x) = x^2$ which has a root at 0 but no bracketing interval can be chosen. For this reason, the task of finding *all* roots of a nonlinear equation is a-priori a very difficult task.

If we start from a bracketing interval $[a_0, b_0]$, the algorithm consists of the following:

while $b_i - a_i > \epsilon_{\rm tol}$ **do**
  $x = (a_i + b_i)/2.0$;
  if $f(a_i) * f(x) > 0$ **then**
    $a_{i+1} = x$;
    $b_{i+1} = b$;
  else
    $a_{i+1} = a$;
    $b_{i+1} = x$;
  end
end

What is a reasonable value for $\epsilon_{\rm tol}$? A general rule of thumb is to stop when the width of the bracketing interval has decreased to about $(|a_i| + |b_i|)\,\varepsilon_m/2$ where ε_m is the machine precision. You should think about why that is so.

At each step of the algorithm the width, ϵ_i, of the bracketing interval decreases by a factor of two: $\epsilon_{i+1} = \epsilon_i/2$. Hence, $\epsilon_n = \epsilon_0/2^n$. The number of steps needed to reach accuracy $\epsilon_{\rm tol}$ is thus $n = \log_2(\epsilon_0/\epsilon_{\rm tol})$. While the bracketing-and-bisection method is often said to be "slow" it actually converges exponentially fast! Furthermore it cannot fail.

One could try to improve on simple bracketing-and-bisection with a smarter "best guess" of the position of the root at each step. One way to do this is to fit a quadratic, $P(x)$, through the points $(a_i, f(a_i))$, $(x_i, f(x_i))$ and $(b_i, f(b_i))$ where the current interval is $[a_i, b_i]$ and x_i is the current best guess of the position of the root. This quadratic is used to make the next guess, x_{i+1}, of the position of the root by calculating where it crosses zero. That is, we solve $P(x) = 0$. For a quadratic function this can be done analytically. The quadratic formula involves the calculation of (expensive) square roots and generally produces two roots. For these reasons, it is much better in practice to fit x as a function of y and then *evaluate* the resulting quadratic at $y = 0$ since this results in a unique value of x and does not require the computation of square roots. This is simple but clever work-around goes by the grandiose name of *inverse quadratic interpolation*.

The Lagrange interpolating polynomial through the points $(f(a_i), a_i)$, $(f(x_i), x_i)$ and $(f(b_i), b_i)$ is obtained from Eq. (3.1):

$$x = \frac{a_i\,[y - f(x_i)][y - f(b_i)]}{[f(a_i) - f(x_i)][f(a_i) - f(b_i)]} + \frac{x_i\,[y - f(a_i)][y - f(b_i)]}{[f(x_i) - f(a_i)][f(x_i) - f(b_i)]} + \frac{b_i\,[y - f(a_i)][y - f(x_i)]}{[f(b_i) - f(a_i)][f(b_i) - f(x_i)]}. \tag{4.3}$$

Evaluating it at $y = 0$ gives the next best guess for the position of the root:

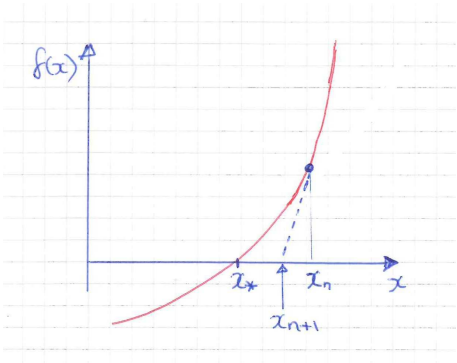
$$x_{i+1} = \frac{a_i\,f(x_i)\,f(b_i)}{[f(a_i) - f(x_i)][f(a_i) - f(b_i)]} + \frac{x_i\,f(a_i)\,f(b_i)}{[f(x_i) - f(a_i)][f(x_i) - f(b_i)]} + \frac{b_i\,f(a_i)\,y - f(x_i)}{[f(b_i) - f(a_i)][f(b_i) - f(x_i)]}. \tag{4.4}$$

The bracketing interval is then reduced in size as before by testing the sign of $f(x_{i+1})$.

Note that it is possible with this procedure to generate a value for x_{i+1} which lies *outside* the interval $[a_i, b_i]$. If this happens, one can resort to a simple bisection of the interval as before. Brent's method combines inverse quadratic interpolation with some checks on proposed refinements of the position of the root and some information about previous estimates of the position, using bisection when necessary, to produce a method which is typically even faster to converge. See [1, chap. 9] for full details. Brent's method is usually the method of choice for generic root-finding problems in one dimension. The idea of inverse quadratic interpolation will come in useful later when we come to the problem of searching for minima.

4.1.2 Derivative methods: Newton-Raphson method

The Newton-Raphson method is the most famous root-finding algorithm. It is a derivative method: it requires the ability to evaluate the derivative $f'(x)$ at any point. The basic idea is the following: at a point, x, which is near to a root, an extrapolation of the tangent line to the curve at x provides an estimate of the position of the root. This is best illustrated geometrically with a picture:



Mathematically, if we are at a point x which is near to a root, x_*, then we wish to find δ such that $f(x+\delta)=0$. This can be done using Taylor's Theorem, (3.2.1). If x is near to the root then δ is small and we can write

$$0 = f(x+\delta) = f(x) + \delta\, f'(x) + O(\delta^2).$$

Neglecting terms of order δ^2 and above and solving for δ we obtain

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

Our improved estimate for the position of the root is then $x+\delta$. This process can be iterated. If our current estimate of the position of the root is x_i, then the next estimate is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.5}$$

Starting from an initial guess, x_0, the algorithm is the following:

while $|f(x_i)| > \epsilon_{\rm tol}$ **do**
  $\delta = -\frac{f(x_i)}{f'(x_i)}$;
  $x_{i+1} = x_i + \delta$;
end

If $f'(x)$ is too difficult (or too expensive) to evaluate analytically for use in Eq. (4.5) then the finite difference formulae developed in Sec. 3.2.1 can be used. This is generally to be discouraged since the additional round-off error (recall Fig. 3.5) can badly degrade the accuracy of the algorithm.

The advantage of Newton-Raphson is that it converges very fast. Let us denote the exact position of the root by x_* and the distance from the root by $\epsilon_i = x_i - x_*$. Using Eq. (4.5, we see that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.6}$$

Since ϵ_i is supposed to be small we can use Taylor's Theorem 3.2.1 :

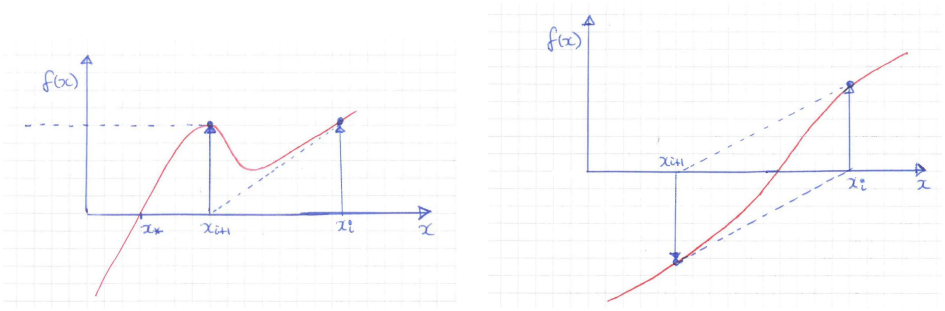
$$\begin{aligned} f(x_*+\epsilon_i) &= f(x_*) + \epsilon_i f'(x_*) + \frac{1}{2}\epsilon_i^2 f''(x_*) + O(\epsilon_i^3) \\ f'(x_*+\epsilon_i) &= f'(x_*) + \epsilon_i f''(x_*) + O(\epsilon_i^2). \end{aligned}$$

Using the fact that $x_*+\epsilon_i = x_i$ and $f(x_*)=0$ we can express the x_i-dependent terms in Eq. (4.6) in terms of ϵ_i and the values of the derivatives of $f(x)$ at the root:

$$\begin{aligned} f(x_i) &= \epsilon_i f'(x_*)\left(1 + \frac{\epsilon_i}{2}\frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_i^3) \\ f'(x_i) &= f'(x_*)\left(1 + \epsilon_i \frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_i^2) \end{aligned}$$

Substituting these into Eq. (4.6) and keeping only leading order terms in ϵ_i we see that

$$\epsilon_{i+1} = \epsilon_i^2 \frac{f''(x_*)}{2\, f'(x_*)}.$$



(A) (B)

Figure 4.1: Some "unfortunate" configurations where the Newton-Raphson algorithm runs into trouble.

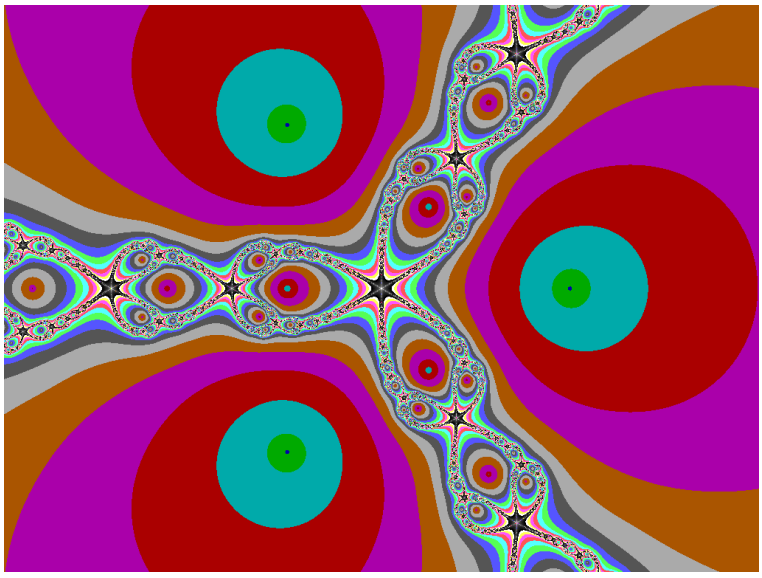


Figure 4.2: (from Wikipedia) Basins of attraction of the roots of Eq. (4.7) in the complex plane. Points are coloured according to the number of iterations of Eq. (4.5) required to reach convergence.

Newton-Raphson iteration therefore converges super-exponentially fast.

The disadvantage of the Newton-Raphson method is that, unlike the bracketing-and-bisection method, it is not guaranteed to converge even if you start near to a bracketed root. Fig. 4.1 shows a couple of unfortunate cases where the Newton-Raphson iteration would fail to converge. In Fig. 4.1(A) an iteration of the method happens to hit on an extremum of $f(x)$. Since $f'(x) = 0$ at an extremum, Eq. (4.5) puts the next guess at infinity! In Fig. 4.1(B), a symmetry between successive iterations sends the method into an infinite loop where the iterations alternate between two values either side of the root.

The potential for complicated dynamics should not come as a surprise to those familiar with the basics of dynamical systems. Eq. (4.5) generalises immediately to complex-valued functions. The result is generally a nonlinear iterated map which can lead to very rich dynamics (periodic cycles, chaos, intermittency etc). For example, Fig. 4.2, shows the basins of attraction in the complex plane of the roots of the polynomial

$$f(z) = z^3 - 1 = 0 \tag{4.7}$$

under the dynamics defined by the Newton-Raphson iteration. One of the great pleasures of nonlinear science is the fact that such beauty and complexity can lurk in the seemingly innocuous task of finding the roots of a simple cubic equation!

4.2 Root-finding in higher dimensions

Root finding in higher dimensions means finding solutions of simultaneous nonlinear equations. For example, in two dimensions, we seek values of x and y which satisfy

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0. \end{aligned}$$

There is no analogue to the concept of bracketing a root in higher dimensions. This makes the process of root finding in higher dimensions very difficult. In order to make a guess about the location of a root in two dimensions, there is no real substitute for tracing out the zero level curves of each function and seeing if they intersect. In dimensions higher than two, the task rapidly starts to seem like searching for a "needle in a haystack". If, however, one does have some insight as to the approximate location of a root in order to formulate a reasonable initial guess, the Newton-Raphson method does generalise to higher dimensions.

Consider the n-dimensional case. We denote the variables by the vector $\mathbf{x} \in \mathbb{R}^n$. We need n functions of $\mathbf{x}$ to have the possibility of isolated roots. We denote these functions by $F_0(\mathbf{x}) \ldots F_{n-1}(\mathbf{x})$. They can be combined into a single vector valued function, $\mathbf{F}(\mathbf{x})$. We seek solutions of the vector equation

$$\mathbf{F}(\mathbf{x}) = 0. \tag{4.8}$$

If the current estimate of the position of the root is $\mathbf{x}_k$ and we take a step $\boldsymbol{\delta}$ then we can expand the i^{th} component of $F(\mathbf{x}_k + \boldsymbol{\delta})$ using the multivariate generalisation of Taylor's Theorem:

$$F_i(\mathbf{x}_k + \boldsymbol{\delta}) = F_i(\mathbf{x}_k) + \sum_{i=0}^{n-1} J_{ij}(\mathbf{x}_k)\, \delta_j + O(\delta^2) \tag{4.9}$$

where

$$J_{ij}(\mathbf{x}) = \frac{\partial F_i}{\partial x_j}(\mathbf{x})$$

is the (i,j) component of the Jacobian matrix, $\mathbf{J}$, of $\mathbf{F}$ evaluated at $\mathbf{x}$. We wish to choose $\boldsymbol{\delta}$ so that $\mathbf{x}_k + \boldsymbol{\delta}$ is as close as possible to being a root. Setting $F_i(\mathbf{x}_k + \boldsymbol{\delta}) = 0$ in Eq. (4.9) for each i, we conclude that the components of $\boldsymbol{\delta}$ should satisfy the set of linear equations

$$\mathbf{J}(\mathbf{x}_k)\, \boldsymbol{\delta} = \mathbf{F}(\mathbf{x}_k). \tag{4.10}$$

This set of linear equations can be solved using standard numerical linear algebra algorithms:

$$\boldsymbol{\delta} = \mathrm{LinearSolve}(\mathbf{J}(\mathbf{x}_k), \mathbf{F}(\mathbf{x}_k)). \tag{4.11}$$

As for Newton-Raphson iteration in one dimension, the next estimate for the position of the root is $\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{\delta}$. We have already seen how the two-dimensional case of Newton-Raphson iteration for complex-valued functions can already lead to very nontrivial dynamics. In higher dimensions, the scope for non-convergence becomes even greater. Nevertheless, if one starts close to a root, the above algorithm usually works.

4.3 Local optimisation in one dimension

Given a real-valued function, $f(x)$, of a single scalar variable, optimization is the task of finding values of x for which $f(x)$ is extremal. That is to say, we seek values of x at which $f'(x) = 0$. These correspond to maxima and minima. It is sufficient to consider only methods for finding minima since maxima can be found be finding minima of $-f(x)$. General optimisation problems can be divided into two classes: local and global. The distincion rests with whether we are required to find local or global extrema of the function to be optimised. As we shall see, local optimisation is relatively easy. Global optimisation on the other hand is usually very hard. In one-dimension there is a simple procedure for finding local minima (which actually generalises quite well to higher dimensions): evaluate the local gradient and "go downhill" until you can't go downhill any more (to within a pre-specified error tolerance of course). If a local minimum exists such a procedure cannot fail to find it.

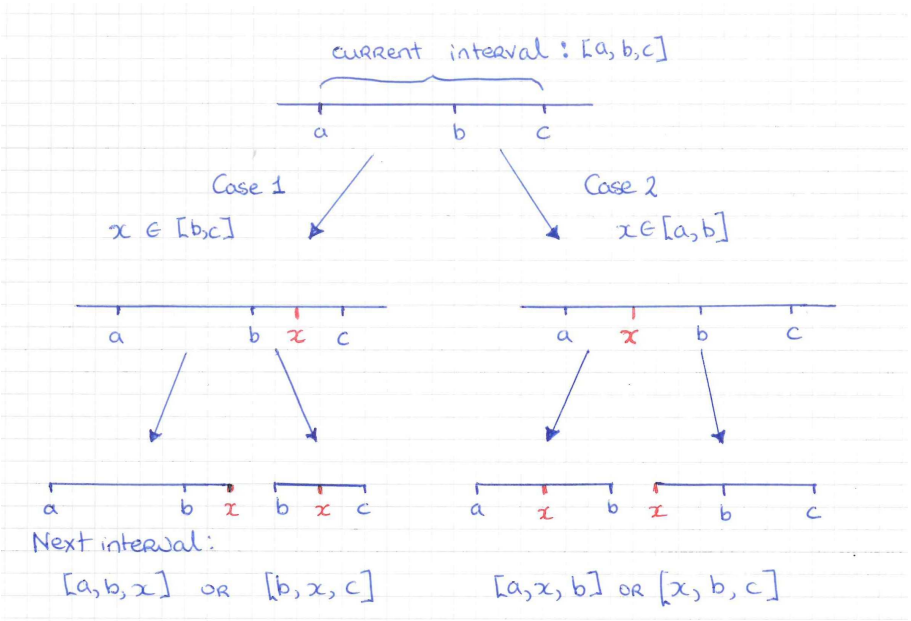


Figure 4.3: Map of possible refinements of a bracketing triple (a, b, c) in the context of the golden section search algorithm for finding a minimum of a function of a single variable.

4.3.1 Golden section search - a bracketing-and-bisection method

Let us begin by adapting the bracketing-and-bisection method of finding roots of $f(x)$ to the problem of finding local minima. In one dimension, a root is said to be *bracketed* by the ordered triple (a, b, c) if $f(b) < f(a)$ and $f(b) < f(c)$. "Ordered" means that $a < b < c$. Starting from a bracketing triple (a_0, b_0, c_0), the idea is to generate successive refinements of the bracketing triple (a, b, c) until $c - b < \epsilon_{\text{tol}}$ where ϵ_{tol} is the pre-specified error tolerance. One refinement strategy would be to evaluate $f(x)$ at the midpoints, x_1 and x_2 of the two respective subintervals, (a, b) and (b, c), test which of the sub-intervals (a, x_1, b), (x_1, b, x_2) and (b, x_2, c) is itself a bracketing subinterval and then set the refinied interval equal to the one which passes this test. This would reduce the bracketing interval by a (fixed) factor of $1/2$ at each step but requires two evaluations of the function $f(x)$ per step. It turns out that it is possible to devise a refinement algorithm which requires only a single evaluation of $f(x)$ per step which retains the attractive property of reducing the bracketing interval by a fixed amount per step. This algorithm is known as the golden section search. The price to be paid for fewer function evaluations is that we can no longer reduce the size of the bracketing interval by a factor of $1/2$ at each step but only by a factor of $(1 + \sqrt{5})/2 \approx 0.618$. The efficiency savings in the reduced number of function evaluations usually more than compensate for the slightly slower rate of interval shrinkage (the convergence is still exponentially fast).

The idea of the golden section search is as follows. At each successive refinement, we pick a new point, x, in the larger of the two intervals (a, b) and (b, c), evaluate $f(x)$ and use it to decide how to shrink the interval. The possible options are sketched in Fig. 4.3. Let us suppose that we choose x to be in the interval (b, c) (Case 1 in Fig. 4.3). After evaluation of $f(x)$ the new bracketing triple is either (a, b, x) or (b, x, c). The size of the new bracketing triple will be either $x - a$ or $c - b$. We require these two lengths to be equal. The reasoning behing this is the following: if they were not not equal, the rate of convergence could be slowed down in the case of a run of bad luck which causes the longer of the two to be selected consecutively. Therefore we chose

$$x - a = c - b \Rightarrow x = a - b + c. \tag{4.12}$$

Note that

$$\begin{aligned} b < x \quad &\Rightarrow \quad b < a - b + c \\ &\Rightarrow \quad b - a < c - b \\ &\Rightarrow \quad x \text{ is in the larger of the two sub-intervals.} \end{aligned}$$

It remains to decide where to choose x within the interval (b, c). A naive suggestion would be to choose x to be the midpoint of (b, c). This, it turns out, would destroy the property that the interval shrinks by a fixed factor at each iteration.

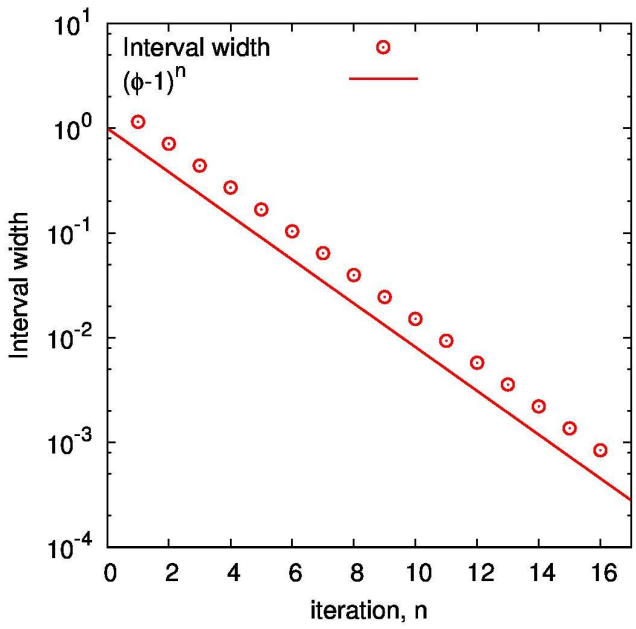


Figure 4.4: Interval width as a function of number of iterations, n, for Golden Section search applied to the function $f(x) = x^{-1}\,e^{x}$ which has a local minimum at $x = 1$. The theoretical convergence rate, $(1-\phi)^n$, (where $\phi = (1+\sqrt{5})/2$ is the Golden Mean) is shown by the solid line.

We characterise the "shape" of a bracketing triple by the ratio

$$w = \frac{c-b}{c-a} \tag{4.13}$$

which measures the proportion of the bracketing interval filled up by the second sub-interval. Obviously the proportion filled by the first sub-interval is

$$1 - w = \frac{b-a}{c-a}. \tag{4.14}$$

In order to ensure that the interval shrinks by a fixed factor at each iteration, the geometry of the refined triple should be the same as that of the current triple. Therefore if the new interval is (b, x, c), we require that

$$\frac{c-x}{c-b} = w.$$

Using Eqs. (4.12), (4.13) and (4.14) we get

$$w = \frac{c-(a-b+c)}{c-b} = \frac{b-a}{c-b} = \frac{\frac{b-a}{c-a}}{\frac{c-b}{c-a}} = \frac{1-w}{w}. \tag{4.15}$$

Rearranging this gives a quadratic equation or the shape ratio, w:

$$w^2 + w - 1 = 0 \quad \Rightarrow w = \frac{\sqrt{5}-1}{2}. \tag{4.16}$$

The appearance of the so-called "Golden mean", $\phi = (\sqrt{5}+1)/2$, in this equation gives the method its name. You might ask what happens if the new interval had been (a, b, x)? It is almost the same. If we define the shape ratio to be

$$w = \frac{x-b}{x-a},$$

we get the quadratic $w^2 - 2w + 1 = 0$ which has only the single root $w = 1$. This is not a valid outcome. However, we can simply swap the order of the large and small subinterval by requiring that

$$w = \frac{b-a}{x-a},$$

and this again returns the Golden mean, Eq. (4.16).

We must not forget that we could have chosen x to be in the interval (a, b) (Case 2 in Fig. 4.3). In this case, a similar argument (left as an exercise) shows that we need $b - a > c - b$ (i.e. x is again in the larger of the two subintervals) and the new interval is either (a, b, x) or (b, x, c) and the shape ratio w which preserves the geometry is again the Golden mean, Eq. (4.16). Gathering together these findings, the algorithm for finding the minimum is the following:

$w = (\sqrt{5} - 1)/2$;
while $c - a > \epsilon_{\text{tol}}$ **do**
  if $|c - b| > |b - a|$ **then**
    $x = b + (1 - w)(c - b)$;
    if $f(b) < f(x)$ **then**
      $(a, b, c) = (a, b, x)$;
    else
      $(a, b, c) = (b, x, c)$;
    end
  else
    $x = b - (1 - w)(b - a)$;
    if $f(b_i) < f(x)$ **then**
      $(a, b, c) = (x, b, c)$;
    else
      $(a, b, c) = (a, x, b)$;
    end
  end
end

4.3.2 Optimisation by parabolic interpolation

The golden section search is in a sense a worst case algorithm which assumes nothing about the function being minimised except that it has a minumum. In many situations, the function $f(x)$ is continuous and differentiable. In this case, the function can be well approximated near its minimum by a second order Taylor expansion which is parabolic in nature. Given a bracketing triple, (a, b, c), one can fit a quadratic through the three points $(a, f(a))$, $(b, f(b))$ and $(c, f(c))$. By analytically calculating the point at which this parabola reaches its minimum one can step to the minimum of the function (or very close to it) in a single step. Using Eq. (3.1) one can show that the value of x for which this parabola is a minimum is

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b) - f(a)] - (b-c)^2[f(b) - f(a)]}{(b-a)[f(b) - f(c)] - (b-c)[f(b) - f(a)]}. \tag{4.17}$$

The value of $f(x_*)$ can then be used, as in the Golden section search to define a refined bracketing triple. This is referred to as parabolic minimisation and can yield very fast convergence for smooth functions. In practice, Eq. (4.17) can be foiled if the points happen to become collinear or happen to hit upon a parabolic maximum. Therefore parabolic minimisation algorithms usually need to perform additional checking and may resort to bisection if circumstances require.

4.4 Local optimisation in higher dimensions

Searching for local extrema in higher dimensional spaces is obviously more difficult than in one dimension but it does not suffer from the general intractability of higher dimensional root-finding which we mentioned in Sec. 4.2. The reason is because the concept of "going downhill" generalises to higher dimensions via the local gradient operator. Therefore, one always has at least the possibility to take steps in the right direction. Higher dimension optimisation algorithms can be grouped according to whether or not they require explicit evaluation of the local gradient of the function to be minimised.

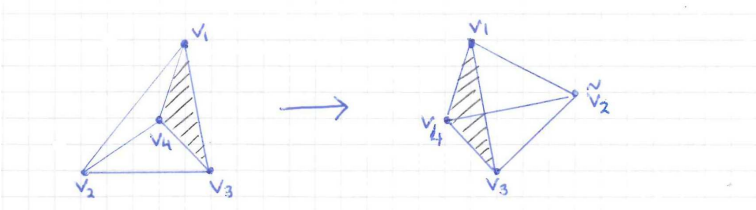
4.4.1 A derivative-free method: Nelder-Mead downhill simplex algorithm

Computing derivatives of multivariate functions can be complicated and expensive. The Nelder-Mead algorithm is a derivative-free method for nonlinear multivariate optimisation which works remarkably well on a large variety of problems,

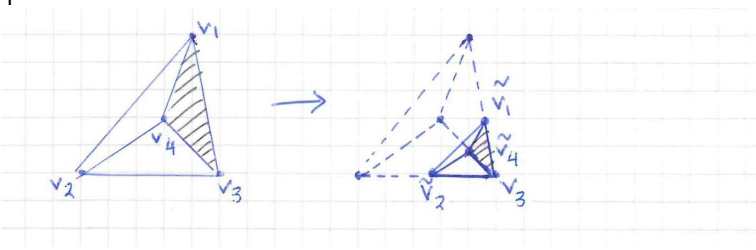
including non-smooth functions (since it doesn't require derivatives). It is conceptually and geometrically very simple to understand and to use even if it can be a bit slow. Everyone should know about this algorithm!

A simplex is a special polytope of N + 1 vertices in N dimensions. Examples of simplices include a line segment on a line, a triangle on a plane, a tetrahedron in three-dimensional space and so forth. The idea of the Nelder-Mead algorithm is to explore the search space using a simplex. The function to be minimised is evaluated on each of the vertices of the simplex. By comparing the values of the function on the vertices, the simplex can get a sense of which direction is downhill even in high dimensional spaces. At each step in the algorithm is uses some geometrical rules to generate test points, evaluates the objective function at these test points and uses the results to define a new simplex which has moved downhill with respect to the original. For this reason the algorithm is often called the downhill simplex algorithm or the amoeba method. The idea is best conveyed from a movie: see [2]

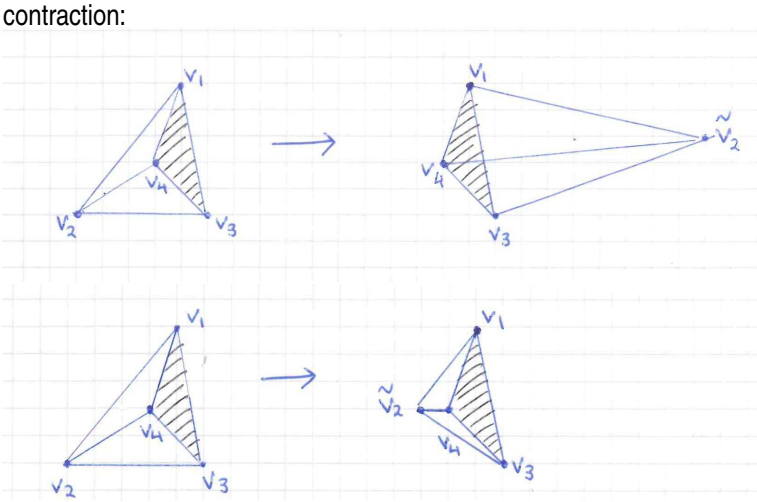
The simplest way to generate test points is to replace the worst point with a point reflected through the centroid of the remaining N points. If this point is better than the best current point then we throw away the previous worst point and update with the new one:



This reflects the simplex through the centroid in a direction which goes downhill. In this picture, and those below, we assume that the highest value of f is attained on the vertex v_2 and the lowest value of f is attained on the vertex v_3. The vertices of the new simplex which have changed during a move are denoted with tildes. If the point obtained by reflection of the worst vertex through the centroid isn't much better than the previous value then we shrink the simplex towards the best point:



These two moves, which preserve the shape of the simplex, constituted the earliest variant of the downhill simplex method designed by Spendley, Hext and Himsworth in [3]. Nelder and Mead added two additional moves [4], expansion and contraction:



These additional moves allow the simplex to change shape and to "squeeze through" narrow holes. The algorithm is stopped when the volume of the simplex gets below a predefined threshold, $\epsilon_{\rm tol}$. For a full discussion of the method see [1, chap. 10] or [5].

4.4.2 A derivative method: the conjugate gradient algorithm

In this section we consider methods for minimising $f(\mathbf{x})$ with $\mathbf{x} = (x_1, x_2 \ldots x_n) \in \mathbb{R}^n$ which make use of the gradient of $f(\mathbf{x})$. That is, we can evaluate $\nabla f(\mathbf{x})$ at any point $\mathbf{x}$. Recall that the gradient operator is simply the vector of partial

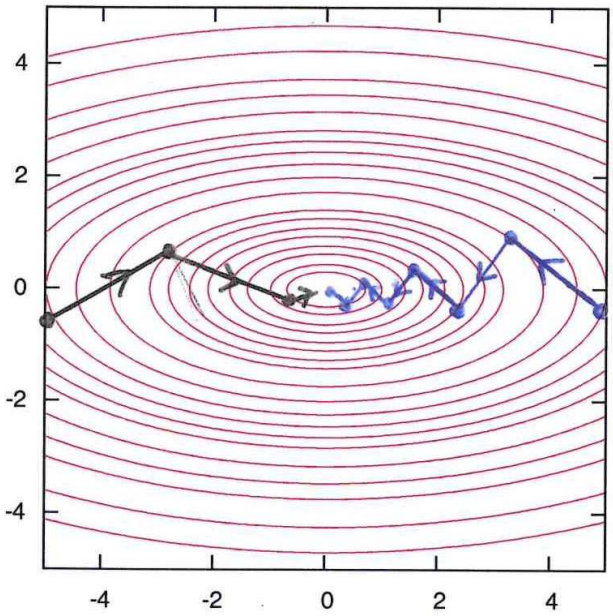


Figure 4.5: Path followed by the Method of Steepest Descent (right path) and Conjugate Gradient Method (left path) during the minimisation of a quadratic function in two dimensions containing a long narrow(ish) valley.

derivatives:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \dots \frac{\partial f}{\partial x_n}(\mathbf{x}) \right).$$

The gradient of f at $\mathbf{x}$ points in the direction in which f increases most quickly. Therefore $-\nabla f(\mathbf{x})$ points in the direction in which f decreases most quickly. The ability to calculate the gradient therefore means that we always know which direction is "downhill".

The concept of line minimisation allows us to think of minimisation problems in multiple dimensions in terms of repeated one-dimensional minimisations. Given a point $\mathbf{x}$ and a vector $\mathbf{u}$, the line minimiser of f from $\mathbf{x}$ in the direction $\mathbf{u}$ is the point $x_* = \mathbf{x} + \lambda_{\min} \mathbf{u}$ where

$$\lambda_{\min} = \arg\min_{\lambda} f(\mathbf{x} + \lambda \mathbf{u}).$$

The important point is that this is a one-dimensional minimisation over λ which can be done, for example, using the golden section search described in Sec. 4.3. Evaluating $f(\mathbf{x} + \lambda_{\min} \mathbf{u})$ gives the minimum value of f along the line in $\mathbb{R}^n$ parameterised by $\mathbf{x} + \lambda \mathbf{u}$. A potential strategy for minimising a function in $\mathbb{R}^n$ is to perform sequential line minimisations in different directions until the value of $f(x)$ stops decreasing. This strategy will work but for it to work well we need to find a "good" set of directions in which to perform the line minimisations.

How should we choose the next direction after each line minimisation? This is where being able to compute the gradient is very useful. Firstly we remark that if $\mathbf{x}_*$ is a line minimiser of f in the direction $\mathbf{u}$, then the gradient of f at $\mathbf{x}_*$, $\nabla f(\mathbf{x}_*)$, must be perpendicular to $\mathbf{u}$. If this were not the case then the directional derivative of f in the direction of $\mathbf{u}$ at xv_* would not be zero and $\mathbf{x}_*$ would not be a line minimum. Therefore, after we arrive at a line minimum, $\mathbf{x}_*$, $-\nabla f(\mathbf{x}_*)$, points directly downhill from $\mathbf{x}_*$ and provides a natural choice of direction for the next line minimisation. Iterating this process gives a method known as the *Method of Steepest Descent*.

While the Method of Steepest Descent works well for many functions it can be slow to converge. The problem with this method is visualised in Fig. 4.5. Since the method is forced to make right-angled turns, it can be very inefficient at making its way to the bottom of narrow valleys. This is illustrated by the (blue) path on the right side of Fig. 4.5. It would be better to allow the algorithm to make steps in directions which are not parallel to the local gradient as in the left path in Fig. 4.5. We are then back to our original question of how to choose the direction for the next step. It turns out that a very good choice is to select the new direction in such a way that the infinitessimal *change* in the gradient is parallel to the local gradient

(or perpendicular to the direction along which we have just minimised). This leads to a method known as the Conjugate Gradient Algorithm.

Underlying this algorithm is the assumption that the function to be approximated can be approximated by a quadratic form centred on some (fixed) point $\mathbf{p}$. The point $\mathbf{p}$ should be thought of as the origin of the coordinate system. Before discussing the method in detail, let us explore some consequences of this assumption. The approximating quadratic form is determined by the multivariate Taylor's Theorem:

$$\begin{aligned} f(\mathbf{p}+\mathbf{x}) &\approx f(\mathbf{p}) + \sum_{i=0}^{n-1} x_i \frac{\partial f}{\partial x_i}(\mathbf{p}) + \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_i x_j \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p}) \\ &= c - \mathbf{b}\cdot\mathbf{x} + \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x}, \end{aligned} \tag{4.18}$$

where

$$\mathbf{b} = -\nabla f(\mathbf{p})$$

and the matrix $\mathbf{A}$ has components

$$A_{ij} = \frac{\partial f}{\partial x_i \partial x_j}(\mathbf{p}).$$

$\mathbf{A}$ is known as the Hessian matrix of f at $\mathbf{p}$. To the extent that the approximation in Eq. (4.18) is valid, the gradient of f at $\mathbf{x}$ is

$$\nabla f(\mathbf{p}+\mathbf{x}) = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}. \tag{4.19}$$

The infinitessimal variation in the gradient as we move in some direction is therefore

$$\delta(\nabla f) = \mathbf{A}\cdot(\delta\mathbf{x}). \tag{4.20}$$

Let us now return to the discussion of how to choose directions for line minimisation. Suppose that we have just done a minimisation in the direction $\mathbf{u}$. We wish to choose a new direction $\mathbf{v}$ such that the change in gradient upon going in the direction $\mathbf{v}$ is perpendicular to $\mathbf{u}$. From Eq. (4.20), we see that

$$\begin{aligned} \mathbf{u}\cdot\delta(\nabla f) &= 0 \\ \Rightarrow \mathbf{u}\cdot\mathbf{A}\cdot(\delta\mathbf{x}) &= 0. \end{aligned}$$

Hence $\mathbf{v}$ should point in the direction of this variation $\delta(\mathbf{x})$ such that

$$\mathbf{u}\cdot\mathbf{A}\cdot\mathbf{v} = 0. \tag{4.21}$$

Vectors $\mathbf{u}$ and $\mathbf{v}$ satisfying this condition are said to be *conjugate* with respect to the Hessian matrix, $\mathbf{A}$. Note the difference with the Method of Steepest Descent which selected $\mathbf{v}$ such that $\mathbf{u}\cdot\mathbf{v} = 0$. The conjugate gradient method constructs a sequence of directions, each of which is conjugate to all previous directions and solves the line minimisations analytically along these direction (this is possible since we are minimising along a section of a quadratic form). A procedure for doing this was provided by Fletcher and Reeves [6]. It works by constructing two sequences of vectors, starting from an arbitrary initial vector $\mathbf{h}_0 = \mathbf{g}_0$, using the recurrence relations

$$\begin{aligned} \mathbf{g}_{i+1} &= \mathbf{g}_i - \lambda_i \mathbf{A}\cdot\mathbf{h}_i \\ \mathbf{h}_{i+1} &= \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \end{aligned} \tag{4.22}$$

where

$$\begin{aligned} \lambda_i &= \frac{\mathbf{g}_i\cdot\mathbf{h}_i}{\mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_i} \\ \gamma_i &= \frac{\mathbf{g}_{i+1}\cdot\mathbf{g}_{i+1}}{\mathbf{g}_i\cdot\mathbf{g}_i}. \end{aligned} \tag{4.23}$$

For each $i = 0 \ldots n-1$, the vectors constructed in this way have the properties

$$\begin{aligned} \mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_j &= 0 \\ \mathbf{g}_i\cdot\mathbf{g}_j &= 0 \\ \mathbf{g}_i\cdot\mathbf{h}_j &= 0 \end{aligned}$$

for each $j < i$. In particular, the $\mathbf{h}_i$ are mutually conjugate as we required. These formulae are not supposed to be obvious. The proofs of these statements are out of scope but we might try to look at them when we come to the linear algebra section of the module. For those who are curious see [7]. The result is a very efficient algorithm for finding minima of functions in multiple dimensiona which has been adapted to myriad uses since its discovery.

The"fly in the ointment" in this discussion is the fact that we are assuming throughout that we know the Hessian matrix, $\mathbf{A}$, whereas we actually do not. This is where the real magic comes in: it is possible to contruct the vectors in Eq. (4.22) without knowing $\mathbf{A}$ by exploiting the fact that we have the ability do line minimisations (for example using the golden section search). This is done as follows:

- suppose we are at a point $\mathbf{p}_i$ and set $\mathbf{g}_i = -\nabla f(\mathbf{p}_i)$.
- given a direction $\mathbf{h}_i$ we move along this direction to the line minimum of f in the direction $\mathbf{h}_i$ from $\mathbf{p}_i$ and define $\mathbf{p}_{i+1}$ to be this line minimiser.
- now set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{p}_{i+1})$.
- it turns out that the vector $\mathbf{g}_{i+1}$ constructed in this way is the same as the one obtained from the construction Eqs. (4.22) and (4.23)! Note that the next direction, $\mathbf{h}_{i+1}$, is constructed from $\mathbf{g}_{i+1}$.

To see why this works, we use Eq. (4.19). According to the above prescription

$$\mathbf{g}_i = -\nabla f(\mathbf{p}_i) = \mathbf{b} - \mathbf{A}\cdot\mathbf{p}_i \tag{4.24}$$

$$\mathbf{g}_{i+1} = -\nabla f(\mathbf{p}_{i+1}) = \mathbf{b} - \mathbf{A}\cdot\mathbf{p}_{i+1} \tag{4.25}$$

where $\mathbf{p}_{i+1}$ is the line minimiser of f in the direction $\mathbf{h}_i$ from $\mathbf{p}_i$:

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \lambda_{\min}\mathbf{h}_i, \tag{4.26}$$

for some value of $\lambda_{\min}$ which we will need to find. Substituting Eq. (4.26) into Eq. (4.25) we find

$$\begin{aligned}\mathbf{g}_{i+1} &= \mathbf{b} - \mathbf{A}\cdot(\mathbf{p}_i + \lambda_{\min}\mathbf{h}_i)\\ &= \mathbf{b} - \mathbf{A}\cdot\mathbf{p}_i - \lambda_{\min}\mathbf{A}\cdot\mathbf{h}_i\\ &= \mathbf{g}_i - \lambda_{\min}\mathbf{A}\cdot\mathbf{h}_i.\end{aligned}$$

This is the vector given in Eq. (4.22) provided that $\lambda_{\min}$ has the correct value. To find $\lambda_{\min}$ we note that since $\mathbf{p}_{i+1}$ is a line minimum in the direction $\mathbf{h}_i$ we have

$$0 = \nabla f(\mathbf{p}_{i+1})\cdot\mathbf{h}_i = \mathbf{g}_{i+1}\cdot\mathbf{h}_i.$$

Substituting our value for $\mathbf{g}_{i+1}$ into this relation we have

$$(\mathbf{g}_i - \lambda_{\min}\mathbf{A}\cdot\mathbf{h}_i)\cdot\mathbf{h}_i = 0 \Rightarrow \lambda_{\min} = \frac{\mathbf{g}_i\cdot\mathbf{h}_i}{\mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_i}$$

which is exactly what is required by Eq. (4.23). We have therefore constructed the set of directions, $\mathbf{g}_i$ and $\mathbf{h}_i$ required by the Conjugate Gradient Algorithm, Eqs. (4.22) and (4.23), without making reference to the Hessian matrix $\mathbf{A}$!

4.5 Global optimisation: heuristic methods

Global optimisation is generally a much harder problem than local optimisation. The reason is because nonlinear functions typically have lots of local maxima and minima. The archetypal example of such a function with many local minima is the famous "travelling salesman" problem: given a list of n cities and the list of distances between each city, find the route of minimal lenght which starts from one city and visits all remaining $n-1$ cities exactly once and returns to the original city.

Without knowing global information about the function to be optimised, one can never know whether a particular local minimum which has been found could be improved upon by looking elsewhere. In the absence of any additional information, the only fool proof strategy is to simply evaluate f for all allowable values of its arguments and select the best one. While this might be possible for some problems where the number of possible values for arguments is finite, it will always be slow for large problems and of no help at all for continuous search spaces. In the case of the travelling salesman problem there are $n!$ possible choices of path.

With the exception of so-called convex optimisation problems for which it is known a-priori that there is only a single minimum (in which case a local minimum is also a global minimum), there are very few provably convergent algorithms for global optimisation problems. For this reason algorithms for global optimisation usually make heavy use of heuristic algorithms. Heuristic algorithms are approaches which, although not proven to converge, are known from experience to find answers which are close to optimal. Most of these methods are stochastic search methods of one kind or another. Examples include simulated annealing, Monte Carlo sampling, parallel tempering, genetic algorithms and swarm-based search. This is a huge area of research in its own right which we will not get into in this module.

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes: The art of scientific computing (3rd ed.)*. Cambridge University Press, New York, NY, USA, 2007.

[2] Nelder-mead method, 2014. http://en.wikipedia.org/wiki/Nelder-Mead_method#mediaviewer/File:Nelder_Mead2.gif.

[3] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, 4(4):441–461, 1962.

[4] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[5] S. Singer and J. Nelder. Nelder-mead algorithm. *Scholarpedia*, 4(7):2928, 2009. http://www.scholarpedia.org/article/Nelder-Mead_algorithm.

[6] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 1964.

[7] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.