Notes 6: Optimisation

6.1 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.

6.1.1 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 $c - b < \epsilon_{\rm tol}$ where $\epsilon_{\rm tol}$ 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. 6.1. Let us suppose that we choose x to be in the interval (b, c) (Case 1 in Fig. 6.1). 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

$$x - a = c - b \Rightarrow x = a - b + c. \tag{6.1}$$

Note that

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

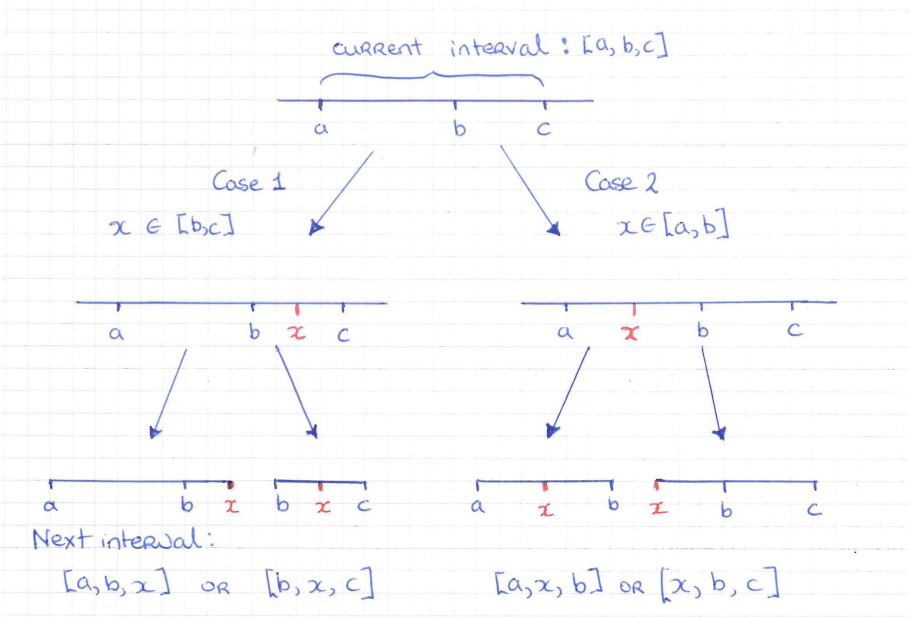


Figure 6.1: Map of possible refinements of a bracketing triple (a, b, c) in the context of the golden section search algorithm for finding a minimum of a function of a single variable.

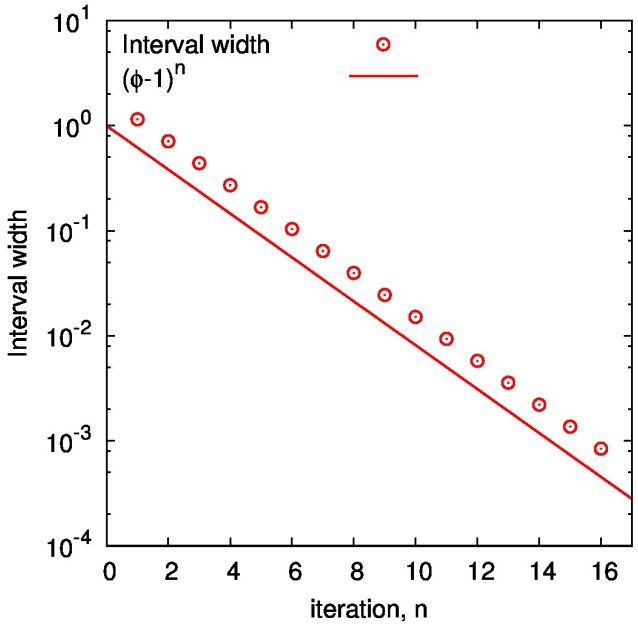


Figure 6.2: Interval width as a function of number of iterations, n, for Golden Section search applied to the function $f(x) = x^{-1}\, e^x$ which has a local minimum at $x = 1$. The theoretical convergence rate, $(1 - \phi)^n$, (where $\phi = (1 + \sqrt{5})/2$ is the Golden Mean) is shown by the solid line.

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

$$w = \frac{c-b}{c-a} \tag{6.2}$$

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

$$1 - w = \frac{b-a}{c-a}. \tag{6.3}$$

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. (6.1), (6.2) and (6.3) we get

$$w = \frac{c-(a-b+c)}{c-b} = \frac{b-a}{c-b} = \frac{\frac{b-a}{c-a}}{\frac{c-b}{c-a}} = \frac{1-w}{w}. \tag{6.4}$$

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

$$w^2 + w - 1 = 0 \quad \Rightarrow w = \frac{\sqrt{5}-1}{2}. \tag{6.5}$$

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. (6.5).

We must not forget that we could have chosen x to be in the interval (a, b) (Case 2 in Fig. 6.1). 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. (6.5). Gathering together these findings, the algorithm for finding the minimum is the following:

$w = (\sqrt{5}-1)/2$;
while $c - a > \epsilon_{\rm tol}$ **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

6.1.2 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

$$x = b - \frac{1}{2}\frac{(b-a)^2\left[f(b)-f(a)\right] - (b-c)^2\left[f(b)-f(a)\right]}{(b-a)\left[f(b)-f(c)\right] - (b-c)\left[f(b)-f(a)\right]}. \tag{6.6}$$

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. (6.6) 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.

6.2 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.

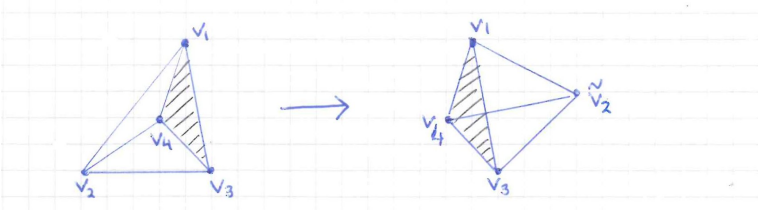
6.2.1 A derivative-free method: Nelder-Mead downhill simplex algorithm

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!

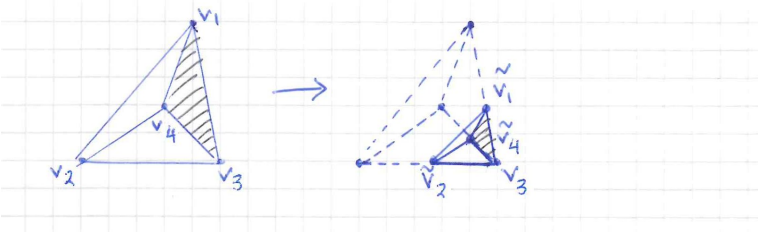
A simplex is a special polytope of N + 1 vertices in N dimensions. Examples of simplices include a line segment on a line, a triangle on a plane, a tetrahedron in three-dimensional space and so forth. The idea of the Nelder-Mead algorithm is to explore the search space using a simplex. The

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 [**?**]

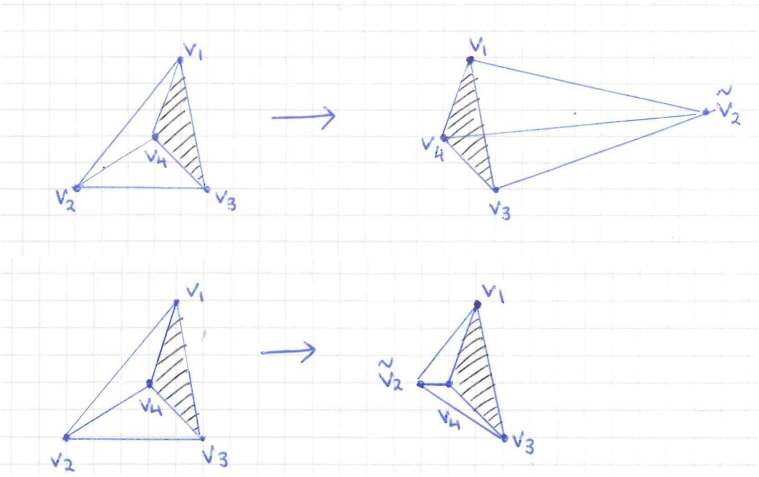
The simplest way to generate test points is to replace the worst point with a point reflected through the centroid of the remaining N points. If this point is better than the best current point then we throw away the previous worst point and update with the new one:



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 [**?**]. Nelder and Mead added two additional moves [**?**], 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 [2, chap. 10] or [**?**].

6.2.2 A derivative method: the conjugate gradient algorithm

In this section we consider methods for minimising $f(\mathbf{x})$ with $\mathbf{x} = (x_1, x_2 \dots x_n) \in \mathbb{R}^n$ 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

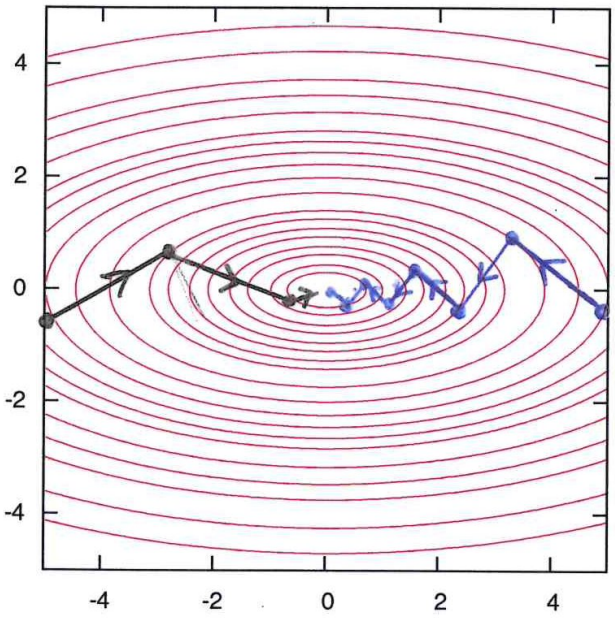


Figure 6.3: Path followed by the Method of Steepest Descent (right path) and Conjugate Gradient Method (left path) during the minimisation of a quadratic function in two dimensions containing a long narrow(ish) valley.

operator is simply the vector of partial 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. 6.1. 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. 6.3. 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. 6.3. 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. 6.3. 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:

$$\begin{aligned} f(\mathbf{p}+\mathbf{x}) &\approx f(\mathbf{p}) + \sum_{i=0}^{n-1} x_i \frac{\partial f}{\partial x_i}(\mathbf{p}) + \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_i x_j \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p}) \\ &= c - \mathbf{b}\cdot\mathbf{x} + \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x}, \end{aligned} \tag{6.7}$$

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. (6.7) is valid, the gradient of f at $\mathbf{x}$ is

$$\nabla f(\mathbf{p}+\mathbf{x}) = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}. \tag{6.8}$$

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

$$\delta(\nabla f) = \mathbf{A}\cdot(\delta\mathbf{x}). \tag{6.9}$$

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. (6.9), 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

$$\mathbf{u}\cdot\mathbf{A}\cdot\mathbf{v} = 0. \tag{6.10}$$

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 [**?**]. It works by constructing two sequences of vectors, starting from an arbitrary initial vector $\mathbf{h}_0 = \mathbf{g}_0$, using the recurrence relations

$$\begin{aligned} \mathbf{g}_{i+1} &= \mathbf{g}_i - \lambda_i \mathbf{A}\cdot\mathbf{h}_i \\ \mathbf{h}_{i+1} &= \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \end{aligned} \tag{6.11}$$

where

$$\begin{aligned} \lambda_i &= \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \\ \gamma_i &= \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i}. \end{aligned} \tag{6.12}$$

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 [**?**]. 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. (6.11) 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. (6.11) and (6.12)! Note that the next direction, $\mathbf{h}_{i+1}$, is constructed from $\mathbf{g}_{i+1}$.

To see why this works, we use Eq. (6.8). According to the above prescription

$$\mathbf{g}_i = -\nabla f(\mathbf{p}_i) = \mathbf{b} - \mathbf{A} \cdot \mathbf{p}_i \tag{6.13}$$

$$\mathbf{g}_{i+1} = -\nabla f(\mathbf{p}_{i+1}) = \mathbf{b} - \mathbf{A} \cdot \mathbf{p}_{i+1} \tag{6.14}$$

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

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \lambda_{\min} \mathbf{h}_i, \tag{6.15}$$

for some value of $\lambda_{\min}$ which we will need to find. Substituting Eq. (6.15) into Eq. (6.14) 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. (6.11) 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. (6.12). We have therefore constructed the set of directions, $\mathbf{g}_i$ and $\mathbf{h}_i$ required by the Conjugate Gradient Algorithm, Eqs. (6.11) and (6.12), without making reference to the Hessian matrix $\mathbf{A}$!

6.2.3 The biconjugate gradient algorithm

The biconjugate gradient algorithm is an iterative method for solving Eq. (5.1) which makes use only of matrix-vector multiplications. Since matrix-vector multiplication can be implemented in $O(n)$ time for sparse matrices, the resulting algorithm is much faster than the LU decomposition discussed in Sec. 5.2 (although more efficient implementations of LU decomposition which take advantage of sparsity do exist, their stability properties are usually inferior to the biconjugate gradient algorithm). The connection to the regular conjugate gradient algorithm which we met in Sec. 6.2.2 will become clearer in due course. We first describe the algorithm.

Start with pair of vectors $\mathbf{g}_0$ and $\bar{\mathbf{g}}_0$ and set $\mathbf{h}_0 = \mathbf{g}_0$ and $\bar{\mathbf{h}}_0 = \bar{\mathbf{g}}_0$. We then construct four sequences of vectors, $\mathbf{g}_k$, $\bar{\mathbf{g}}_k$, $\mathbf{h}_k$ and $\bar{\mathbf{h}}_k$ from the following recurrence:

$$\begin{aligned}
\lambda_k &= \frac{\bar{\mathbf{g}}_k \cdot \mathbf{g}_k}{\bar{\mathbf{h}}_k \cdot \mathbf{A} \cdot \bar{\mathbf{h}}_k} && (6.16)\\
\mathbf{g}_{k+1} &= \mathbf{g}_k - \lambda_k \mathbf{A} \cdot \mathbf{h}_k \quad \bar{\mathbf{g}}_{k+1} = \bar{\mathbf{g}}_k - \lambda_k \mathbf{A}^{\mathrm{T}} \cdot \bar{\mathbf{h}}_k && (6.17)\\
\gamma_k &= \frac{\bar{\mathbf{g}}_{k+1} \cdot \mathbf{g}_{k+1}}{\bar{\mathbf{g}}_k \cdot \bar{\mathbf{g}}_k} \\
\mathbf{h}_{k+1} &= \mathbf{g}_{k+1} + \gamma_k \mathbf{h}_k \quad \bar{\mathbf{h}}_{k+1} = \bar{\mathbf{g}}_{k+1} + \gamma_k \bar{\mathbf{h}}_k. && (6.18)
\end{aligned}$$

The sequences of vectors constructed in this way have the properties of

$$\begin{aligned}
\text{bi-orthogonality:} \quad & \bar{\mathbf{g}}_i \cdot \mathbf{g}_j = \mathbf{g}_i \cdot \bar{\mathbf{g}}_j = 0 && (6.19)\\
\text{bi-conjugacy:} \quad & \bar{\mathbf{h}}_i \cdot \mathbf{A} \cdot \bar{\mathbf{h}}_j = \mathbf{h}_i \cdot \mathbf{A}^{\mathrm{T}} \cdot \bar{\mathbf{h}}_j = 0 && (6.20)\\
\text{mutual orthogonality:} \quad & \bar{\mathbf{g}}_i \cdot \mathbf{h}_j = \mathbf{g}_i \cdot \bar{\mathbf{h}}_j = 0, && (6.21)
\end{aligned}$$

for any $j < i$. These statements are claimed in [2] to be provable by induction although I haven't tried. Note that this recurrence must terminate after n iterations with $\mathbf{g}_n = \bar{\mathbf{g}}_n = 0$. This is because of the orthogonality conditions: after n steps the n previously constructed $\mathbf{g}_k$ must be orthogonal to $\bar{\mathbf{g}}_n$ but this is impossible since the dimension of the space is n. To use Eqs. (6.16)-(6.18) to solve Eq. (5.1), we take an initial guess $\mathbf{x}_0$ and choose our starting vectors to be

$$\mathbf{g}_0 = \bar{\mathbf{g}}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0. \tag{6.22}$$

As we progress through the iteration, we construct a sequence of improved estimates of the solution of Eq. (5.1):

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{h}_k. \tag{6.23}$$

Now we remark that

$$\mathbf{g}_k = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_k. \tag{6.24}$$

Why? By induction, if we assume that Eq. (6.24) is true for k then

$$\begin{aligned}
\mathbf{g}_{k+1} &= \mathbf{g}_k - \lambda_k \mathbf{A} \cdot \mathbf{h}_k \\
&= \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_k - \lambda_k \mathbf{A} \cdot \mathbf{h}_k \\
&= \mathbf{b} - \mathbf{A} \cdot (\mathbf{x}_k + \lambda_k \mathbf{h}_k) \\
&= \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1}
\end{aligned}$$

thus establishing that Eq. (6.24) is true for $k+1$. Since the statement holds for $k = 0$ by the definition (6.22) of $\mathbf{g}_0$, the statement is proven for all k. The importance of Eq. (6.24) is that the vectors $\mathbf{g}_k$ constructed by the bi-conjugate gradient algorithm, Eqs. (6.16)-(6.18), have the meaning of a residual. They measure the distance of our current estimate, $\mathbf{x}_k$, from the true solution. Since we have established that the terminating residual is $\mathbf{g}_n = 0$, $\mathbf{x}_n$ must be the solution of the original linear system.

Let us suppose that $\mathbf{A}$ is symmetric and (in order to avoid zero denominators) positive definite. If we choose $\mathbf{g}_0 = \bar{\mathbf{g}}_0$ then Eqs. (6.16)-(6.18) become equivalent to the "regular" conjugate gradient algorithm, Eqs.(6.11)-(6.12), which we encountered in our study of multi-dimensional optimisation.

To see the connection, note that if $\mathbf{A}$ is symmetric and positive definite, it defines a quadratic form for any vector $\mathbf{b}$ given by

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \cdot \mathbf{x}, \tag{6.25}$$

which has a single minimum at which the gradient, $\nabla f(\mathbf{x})$, is equal to zero. But

$$\nabla f(\mathbf{x}) = \mathbf{A} \cdot \mathbf{x} - \mathbf{b},$$

so minimising the quadratic form is equivalent to finding the solution of the linear system! Historically, the conjugate gradient algorithm in its regular form was invented first in order to solve symmetric positive definite systems of linear equations. The biconjugate gradient algorithm came later to generalise the method to non-symmetric systems of linear equations but the connection with function optimisation is lost in this generalisation. The use of the conjugate gradient algorithm for nonlinear optimisation of functions which can be approximated by quadratic forms came even later still.

6.3 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] Linear multistep method, 2014. http://en.wikipedia.org/wiki/Linear_multistep_method.

[2] 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.

[3] C. Moler. Matlab news & notes may 2003 - stiff differential equations, 2003. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html.

[4] L. F. Shampine and S. Thompson. Stiff systems. Scholarpedia, 2(3):2855, 2007.