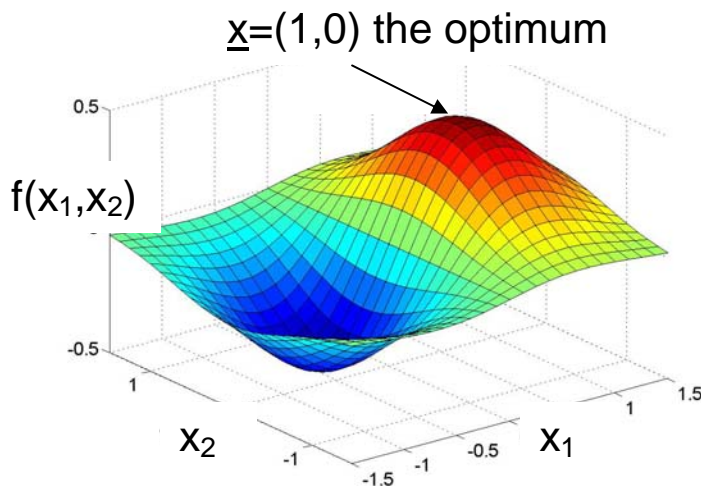


Formal Computational Skills 4: Optimisation

Aims: The aim of this worksheet is to experiment with the gradient ascent algorithm and compare its performance to a hill-climbing algorithm. You should also see the influence of the learning rate and several other user specified parameters

Optimisation

Suppose we have a function $f(x_1, x_2, \dots, x_n)$ which takes n variables as inputs and gives a 1-dimensional output (ie a number). We can visualise this as an $n+1$ dimensional space where $\underline{x} = (x_1, x_2, \dots, x_n)^T$ is a point on an n -D plane and $f(\underline{x})$ is the height above the plane. So for a 2D \underline{x} we would have:



Suppose we want to find the highest (or lowest) value of f . The idea is to search the space of all possible values of \underline{x} (therefore known as the *search-space*) until we find it. Instances of \underline{x} are often referred to as *solutions*.

In this case we would search for $\underline{x} = (1, 0)$ as $f(1, 0)$ is the highest point or the *optimum*

If there is more than one peak, the highest is called the *global* optimum and the others *local* or *false* optima

The procedure of searching is known as *function optimisation* and a great many techniques and algorithms are used for it. 2 issues that are important in determining which method to use are (1) the time it takes to find an optimum and (2) the height (relative to the global optimum) of the optimum found. These issues are often in opposition since generally, the longer spent, searching the better optimum is found, and so which is optimised depends on the needs of the optimiser (do you want an ok solution quickly or do you have the time to look for a really good one?). In this worksheet we will examine the performance of 2 common methods: gradient ascent and hill-climbing (discussed in lecture 4); over a range of different functions (which can be thought of as landscapes).

Gradient Ascent

The idea of gradient ascent is very simple. Starting from random position in the search-space, take a step in the steepest direction up-hill and repeat. Since the gradient $\nabla f(\underline{x})$ points in the steepest direction uphill the algorithm is:

1. Choose a random starting position \underline{x}
2. Evaluate the gradient at \underline{x} : $\nabla f(\underline{x})$
3. Do $\underline{x}_{\text{new}} = \underline{x} + \eta \nabla f(\underline{x})$
4. Repeat from 2 until a stopping condition is met

There are therefore 3 potential areas that can be varied/investigated by the user:

The learning rate η : this controls the size of the step taken and is usually set to a value much less than 1. It will affect the time to get a good solution and the quality of a solution. Theory suggests that a larger learning rate will get to a solution quicker but it may miss better optima. In most practical algorithms, it is set on-line and changed adaptively.

Stopping condition: It is often difficult to know when to stop searching – is the optimum you are at the best or not? – and the question is sometimes answered by practical issues of how much time one has. Typically, an upper limit is set on the number of steps of the algorithm or function evaluations, though if the solution stops moving for a length of time, the algorithm may be terminated prematurely.

Evaluating the gradient: In most cases, as the function being optimised is unknown, the gradient cannot be evaluated mathematically and must be estimated. This can range from simply seeing if it is higher/lower on either side, to sophisticated (and time-consuming) algorithms which approximate 1st and 2nd order derivatives.

Here we will use a constant learning rate initially set to 0.1 (though you will experiment by changing this value) and will stop after a maximum of 50 iterations of the algorithm. As we know the functions we will be optimising, the exact gradient will be used.

Hill-climbing

Hill climbing is another very simple algorithm which proceeds as follows:

1. Choose a random starting position \underline{x}
2. Randomly select a new point \underline{x}_{new} 'near' \underline{x} , that is, *mutate* \underline{x}
3. If $f(\underline{x}_{new}) > f(\underline{x})$, set $\underline{x} = \underline{x}_{new}$ else repeat from 2.
4. Repeat steps 2 and 3 until a stopping condition is met

It is similar to gradient ascent in structure and can be seen as a form of this algorithm. The main difference however is in the level of stochasticity. In the form we will use it, once learning rate etc have been decided on, the gradient ascent algorithm is largely deterministic. That is, given a particular starting point it will always end up at the same point. In contrast, the randomness introduced by step 2 (and sometimes in step 3) makes hill-climbing much more stochastic. Together with the stopping condition (which has the same issues as for gradient ascent), the issues in steps 2 and 3 determine how the algorithm behaves.

Step 2: The mutation process can be divided into 2 parts: which dimensions of \underline{x} to change and how much to change each of them. Some schemes ensure a small random movement in only one dimension at a time, while in principle allow a move to any part of the search-space, albeit with vanishingly small probabilities. Usually the process is set so that mutations to points near \underline{x} are more likely than those further away.

Step 3: When to set $\underline{x} = \underline{x}_{new}$ is normally if $f(\underline{x}_{new}) > f(\underline{x})$. However, we often allow so called *neutral* moves where $f(\underline{x}_{new}) = f(\underline{x})$ and often allow downward moves ie when $f(\underline{x}_{new}) < f(\underline{x})$ but only with a certain probability, ie only allow 10% of downward moves. This brings more randomness into the algorithm's behaviour but also allows it to 'escape' from local minima.

Here we will use following mutation procedure: randomly choose one element of the vector \underline{x} to mutate; mutate this by adding a random number in the range (-MaxMutate, MaxMutate); if $f(\underline{x}_{new}) > f(\underline{x})$ set $\underline{x} = \underline{x}_{new}$. We will start with MaxMutate=1 though you will experiment with this value. As with gradient ascent, stop the algorithm after 50 iterations.

Task 1: Gradient ascent and Hill-climbing with a simple function [5 marks]

Download the file GradientAscentEG.m from the web-site and copy it to your directory. The file contains functions SimpleLandscape and SimpleLandscapeGrad, and ComplexLandscape and ComplexLandscapeGrad. These functions take an (x,y) position as input and return the height or gradient vector of 2 functions of x and y: one simple and one more complicated.

It also has the outline of functions which perform gradient descent/hill-climbing. Sections you need to complete are marked with TO DO in the comments.

1.1 Gradient Ascent: To perform gradient ascent you need to first calculate the gradient at a point (line 28) and then use this gradient to update the point you are at (line 30). Use the

function SimpleLandscapeGrad which returns the elements of the gradient as a vector. Try starting the algorithm from a few random points and observe the behaviour.

1.2 Hill-climbing: Now try hill-climbing on the same landscape. Comment/uncomment lines 15-16, then write the function mutate. As described earlier, this will have 2 components: First generate a random value MutDist between (-MaxMutate, MaxMutate). Look at line 13 for an example of this. Next, use rand to decide which element of StartPoint to change, and add MutDist to it; Now evaluate the new point and update if $f(\underline{x}_{new}) > f(\underline{x})$.

Questions:

What do you notice about the 2 algorithms in terms of the height they get to and the time it takes them to get there? To test this, change the hill climbing and gradient ascent functions so that they return 2 parameters: a 1 or 0 specifying whether the global optimum was reached together with how many iterations the algorithm performed. NB to stop the algorithm after the maximum has been reached, use an 'if' statement to test whether the maximum height has been reached, then use the function 'break' to break out of the 'for' loop.

Now test the algorithms systematically by starting from a grid of points covering the landscape. Do this by replacing the line generating a random starting position with a loop eg:

```
GridPoints=[-2:0.25:2]
for i = 1:length(GridPoints)
    for j = 1:length(GridPoints)
        StartPt=[ GridPoints(i) GridPoints(j)];
        [MaxReached(i, j), Iters(i, j)] = GradAscent(StartPt,NumSteps,LRate);
        if(MaxReached(i, j)) SumItersToMax = SumItersToMax + Iters(i,j); end
    end
end
end
```

NB commenting out plotting commands makes the program run faster. Calculate how many starting points lead to the maximum using sum(sum(MaxReached)), then calculate the mean number of iterations it takes those that reached the maximum, to get there. Use 'pcolor' to graph Times and MaxReached eg: pcolor(GridPoints,GridPoints,MaxReached). To get a scale bar do: colorbar. Experiment with shading and colormap to see their effects.

Explain any differences in results between the 2 algorithms. Experiment with changing the learning rate and range of mutation to see what happens.

Task 2: Gradient ascent and Hill-climbing with a complex function [5 marks]

Now try the complex function, ComplexLandscape. To plot it, rather than line 4, use the function DrawComplexLandscape. You will need to change GridPoints and/or your procedure for generating a random starting point so that both x and y are in the range [-3, 7]. You can however delete the parts of your algorithms which check if points are within the range.

Questions:

Try some random starting positions. If gradient ascent keeps failing as its in a flat bit of the landscape, change the range of random starting positions. What do you notice about the 2 algorithms in terms of their performance? How does this differ from their performance on the simple landscape?

Now test your algorithms systematically over the new ranges. This time however, as there are many optima that the algorithms could find, simply return the height that they achieve after NumSteps iterations and do eg pcolor(MaxReached) to see a plot of the heights reached. The function colorbar will give you a scale. This plot will show you the *basins of attraction* of each of the optima that is, the set of points which lead to a given optima. Experiment with the learning rate and range of mutation and comment on any observations. In particular, try lowering the learning rate to 0.01. How does the result change if you double NumSteps?