Introduction

This is coursework 1 with a total weight of 50%. It is due Friday, 16 November 2018, 16:00. Submission is electronic, via the Assignments section of the module's Canvas page. See last page of this document for detailed instructions.

A) Derivatives and their use

1. Compute the derivative with respect to t of
$$y = -\frac{1}{2}gt^2 + ut + c\,.$$
(4 marks)

   This equation gives the trajectory for the height y of a ball thrown upwards at time $t = 0$, starting at height c, with initial upward velocity u, subject to gravity of strength g. Substituting $g = 10$, $u = 5$ and $c = 1$, and using your expression for the derivative of y, find the maximum height reached by the ball, and the time when the ball reaches this maximum height. (Units are in meters and seconds.)

2. Find the positions and type (i.e. maximum or minimum) of the stationary points of the function $f(x) = 2x^3 - 3x^2 - 36x + 2$, using differentiation.
(6 marks)

3. Consider the sigmoid function $f(x) = \dfrac{1}{1 + \exp(-m(x - x_0))}$.

   (a) Give the limits for $x \to \infty$ and $x \to -\infty$ (no proof necessary). (4 marks)

   (b) By calculating the first and second derivatives (chain rule and product rule!), identify the stationary point *of the derivative* of $f(x)$? Given this information and the value of the derivative at this point, describe the role of x_0 and of m.
(8 marks)

   (c) Plot $f(x)$ in the range $[-5, 5]$ for $m = 1$ and $x_0 = 1$. Does the plot confirm what you found out above? Explain.
(4 marks)

4. Let
$$E = -\sum_{i=1}^{2}\sum_{j=1}^{2} w_{ij}x_ix_j\ .$$
Compute the partial derivatives $\dfrac{\partial E}{\partial x_1}$ and $\dfrac{\partial E}{\partial w_{12}}$.
(4 marks)

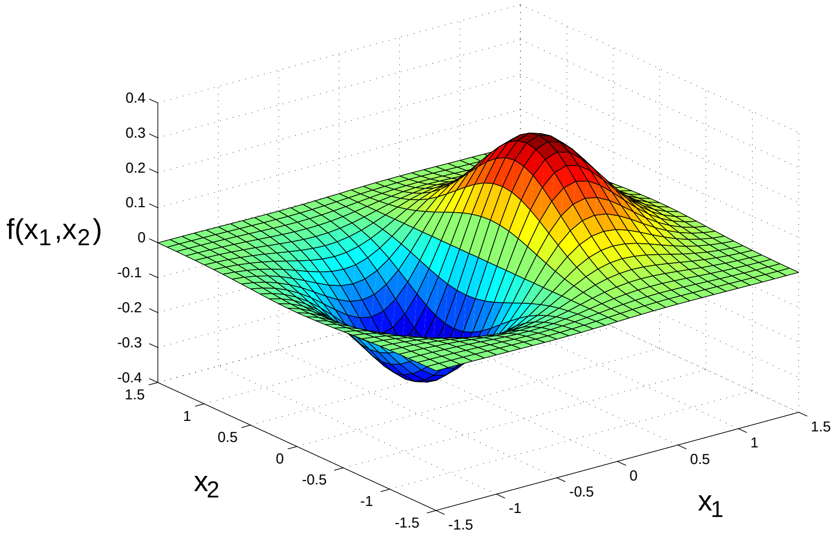
B) Gradient ascent and hill-climbing

This problem is a first example of how to investigate a scientific question with Matlab or Python (NB: You should feel free to use another programming language, however, you will have to do with the templates provided. You are encouraged to discuss it with me beforehand but please be assured there will be no penalty for using a language other than Matlab or Python).

Aims: In this worksheet you will experiment with the gradient ascent algorithm and compare its performance to a hill-climbing algorithm and examine the influence of the learning rate and other user specified parameters.

Introduction: Optimisation

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



If we want to find the highest (or lowest) value of f, the idea is to search the space of all possible values of $\underline{x}$ (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 optimum.

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

The search procedure is known as function optimisation and a great many algorithms are used for it. Two 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: generally the longer spent searching, the better the optimum that is found. Which factor is prioritised therefore depends on the needs of the optimiser: do you want an ok solution quickly or do you have time to look for a really good one?

In this worksheet we will implement two optimisation methods: gradient ascent and hill-climbing. By optimising two functions, we will explore the interplay of the time taken to reach a solution versus the solution's quality and see how the parameters used affect this balance.

Algorithm 1: Gradient Ascent

The idea of gradient ascent is very simple. Starting from a random position in the search-space, take a step in the steepest direction uphill and repeat. Since the gradient points in the steepest direction uphill the algorithm is:

1. Choose a random starting position $\underline{x}$
2. Evaluate the gradient at $\underline{x}$
3. Do $\underline{x}_{\text{new}} = \underline{x} + \eta \cdot \nabla f(\underline{x})$ where η is a learning rate and ∇ denotes the gradient operator
4. Repeat from 2 until a stopping condition is met

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

1. 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 to 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 online and changed adaptively.
2. 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.

3. 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.

Algorithm 2: 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 $\underline{x}_{\text{new}}$ ”near” $\underline{x}$, i.e., ”mutate” $\underline{x}$
3. If $f(\underline{x}_{\text{new}}) > f(x)$, then set $\underline{x} = \underline{x}_{\text{new}}$ 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.

1. In step 2: The mutation process can be divided into 2 parts: which dimensions of 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 others 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.
2. In step 3: When to set $\underline{x} = \underline{x}_{\text{new}}$ is normally if $f(\underline{x}_{\text{new}}) > f(\underline{x})$. However, we often allow so-called neutral moves. These are moves where $f(\underline{x}_{\text{new}}) = f(\underline{x})$. Also one can allow downward moves, i.e. when $f(\underline{x}_{\text{new}}) < f(\underline{x})$ but usually only with a certain probability, e.g. only allow 10% of downward moves. This brings more randomness into the algorithm’s behaviour but also allows it to ”escape” from local maxima.

Task 1: Gradient ascent and Hill-climbing with a simple function

(40 marks)

Download the file `coursework1template.{m,py}` from the module site and copy it to your directory. The file contains functions `SimpleLandscape`, `SimpleLandscapeGrad`, `ComplexLandscape` and `ComplexLandscapeGrad`. These functions take an (x,y) position as input and return the height and gradient vector, respectively, of two 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

With respect to the three variables in the gradient ascent algorithm, we will initially:

1. use a constant learning rate of 0.1;
2. stop after a maximum of 50 iterations of the algorithm;
3. as we know the functions we will be optimising, the exact gradient will be used.

To perform gradient ascent you need to first calculate the gradient at a point and then use this gradient to update the point you are at. Use the function `SimpleLandscapeGrad` or `ComplexLandscapeGrad` which returns the elements of the gradient of the corresponding function 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 out the call to `GradAscent`, then complete the function `Mutate`. Initially, use the following mutation procedure:

1. Randomly choose one element of the vector $\underline{x}$ to mutate;
2. Mutate this by adding a random number in the range (-MaxMutate, MaxMutate);
3. if $f(\underline{x}_{\text{new}}) > f(x)$ set $\underline{x} = \underline{x}_{\text{new}}$.

Initially, use MaxMutate=1 and stop the algorithm after 50 iterations. To implement the steps above, first generate a random value MutDist between (-MaxMutate, MaxMutate). You have learned to do this in a recent seminar. Next, use an appropriate random function to decide which element of StartPoint to change, and add MutDist to it.

Questions

1. 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.
2. Now test the algorithms systematically by starting from a grid of points covering the landscape. You will have to replace the line generating a random starting position with some suitable code. NB: commenting out plotting commands makes the program run faster. Calculate how many starting points lead to the maximum, then calculate the mean number of iterations it takes those that reached the maximum to get there. You will find the function `pcolor` to be useful to summarise your results visually. To get a scale bar, use `colorbar`. Experiment with shading and colormap to see their effects. Document your results in the form of a small report with figures, figure captions and explanations.
3. Explain any differences in results between the 2 algorithms. Experiment with changing the learning rate and range of mutation to see what happens.

Marks: Half the marks are for implementation, i.e. running the algorithms from different starting points, running them systematically and experimenting with changing learning rate and max mutation size. The other half of the marks are for presenting, describing, explaining and analysing the results, especially ”odd” results and relating them to how the algorithms work – see submission format instructions below.

Task 2: Gradient ascent and Hill-climbing with a complex function

(30 marks)

Now try the complex function `ComplexLandscape`. For this function, you will need to change your procedure for generating a random starting point so that both x and y are in the range [-3, 7].

Questions

1. Try some 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?
2. 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. Again use `pcolor` to summarise your results. 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 (e.g. try lowering the learning rate to 0.01? How does the result change if you double NumSteps?) and comment on any observations.

Marks: Again, half marks for implementation and experimenting with parameters, the other half for description and analysis of results, relating observations to the way the algorithms work.

Submission format

For Part A, you can scan handwritten solutions or typeset them. It won't make any difference to my mark, provided I can read your handwriting! For Part B, your submission must include:

- code: namely, working versions of `coursework1template.{m,py}` for both simple and complex functions (feel free to change filename!)
- report (PDF format): illustrating the numerical experiments you have done, what you have found out and how it relates to how the algorithms work. The report part should **not exceed 6 pages** (figures included) and should be readable (fonts no smaller than 10pt please, 11pt or 12pt better!). Concise writing is good. Do not waste space repeating how gradient ascent or hill climbing works. Neither should you provide a running commentary of your code. Instead make sure your code is suitably commented (no page limit there!).

Total marks: 100.