The Basics of Optimization

Spring 2013

If Matlab stalls or crashes, press "Ctrl-C" to stop execution; to reset graphics, type "close all"

You will be writing Matlab code to optimize some functions in 1 and 2 dimensions. Some example functions for you to optimize are in `/course/cs157/pub/stencils/optimization` – copy these functions to a directory in your home folder, and navigate to this directory in Matlab (as indicated by the "Current Folder" bar at the top of the Matlab window). The 1-dimensional functions start with a lowercase "f" followed by a number; the 2-dimensional functions start with a lowercase "g" followed by a number. The functions are all intended to be evaluated on inputs between -10 and 10, and are all set up to plot themselves when called with no inputs. Try it: run `f1` or pick an input like 5 and evaluate f1 at 5, as `f1(5)` (which will also plot the function).

1 Binary search

Your first task is to write a function

```
function x=binarySearchForMin(func,L,U,eps)
```

That takes as input a convex 1-dimensional function to be minimized, `func`, a lower bound `L`, an upper bound `U`, and a step size `eps` (for "epsilon"). Remember that a convex function is one which is "below all its secants", that is, $\frac{\texttt{func}(x)+\texttt{func}(y)}{2} \geq \texttt{func}(\frac{x+y}{2})$, for all inputs x and y; in one dimension this means that convex functions decrease until their minimum, and then increase. (Can you see why?)

When we saw this problem on homework 3, the inputs to the function were integers, but now they are real numbers, which is why we need a variable `eps`. The variable `eps` will serve two purposes: first, to figure out whether our binary search should go left or right from a point m, we evaluate `func(m)`, and `func(m+eps)` and go right if the first is bigger, left if the second is bigger. (In binary search, m should probably be $\frac{L+U}{2}$.) Second, if $U - L \leq$ `eps` this means that we have already restricted the search space to an interval of size `eps`, and it is safe to stop. You are now ready to write the function `binarySearchForMin`.

Function handles in Matlab: Remember that your code here takes a *function* as its first input argument. Different languages have different ways of passing around "pointers to functions". Matlab calls them "function handles", and you can read Matlab's documentation by typing `doc function_handle`. The basics are: if `func` is a function handle (inside `binarySearchForMin`, for example) then you can use it as a regular function, as in `a=func(b)`; to *create* a handle to a function such as `f1`, use the `@` operator. Thus to call `binarySearchForMin` on the function `f1`, searching for a minimum between -10 and 10, to a precision of 0.0001, run:

```
binarySearchForMin(@f1,-10,10,0.0001)
```

2 Golden Section Search

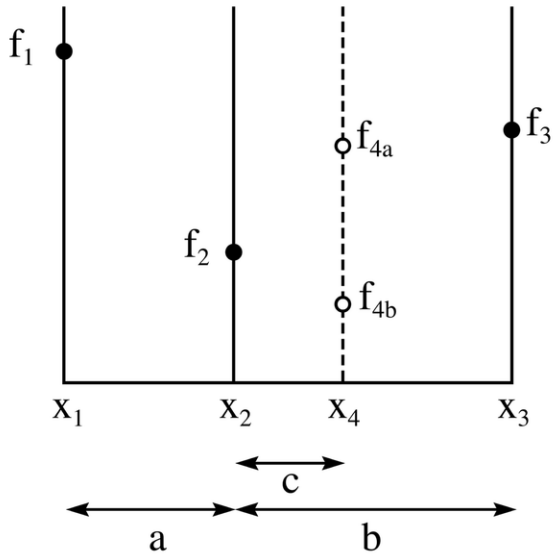
One slightly awkward thing about the above code is the comparison of `func(m)` and `func(m+eps)`, which may suffer from precision issues, and involves "two function evaluations per decision". Phrased another way, to halve the size of the search interval the above code needs two function evaluations, so to improve precision by a factor of p requires $2\log_2 p$ function evaluations. Here we will see a way to slightly improve this.

Recall the golden ratio, denoted by ϕ (the Greek letter "phi"), defined as $\phi = \frac{\sqrt{5}+1}{2}$ which has the property that $1 + \phi = \phi^2$. At any point in the algorithm (except the start), the algorithm will have evaluated the function at three points $x_1 \le x_2 \le x_3$ as illustrated in the diagram (or the mirror image of the diagram, in which case x_2 is closer to x_3 than to x_1, in which case some of the following statements will be flipped). Denote the distance between x_1 and x_3 by d. Then the distance between x_2 and x_3 is d/ϕ, and the distance between x_2 and x_1 is d/ϕ^2. Further, the function has an optimum in the interval $[x_1, x_3]$, meaning that the function is *at least as small* at x_2 as at either endpoint.

In the *next* iteration of the algorithm, construct the point x_4 so that the distance between x_4 and x_3 is d/ϕ^3, and so that x_4 lies in whichever interval $[x_1, x_2]$ or $[x_2, x_3]$ is larger (actually, an easy way to do this is to make $x_4 = x_1 + x_3 - x_2$). There are two cases: if $f(x_4) > f(x_3)$ then recurse on the three values x_1, x_2, x_4; otherwise recurse on the three values x_2, x_4, x_3. Stop when the size of the search interval is less than `eps`.

Write the function `goldenSectionSearchMin`, taking the same arguments as the binary search function: the lower end of the search interval, `L`, the upper end of the search interval, `U`, and the accuracy `eps`. (To initialize the recursion, let $x_1 = L$, $x_3 = U$, and $x_2 = x_3 - \frac{x_3 - x_1}{\phi}$.)

Analysis: note that with each iteration the size of the interval decreases by a factor of ϕ, using one function evaluation. So to improve the precision by a factor of p takes $\log_\phi p$ iterations now, which is 30% better than the $2\log_2 p$ of the binary search algorithm.



3 Gradient Descent in 1 Dimension

In one dimension, gradient descent will seem a bit silly, but its strength is that it generalizes naturally to many dimensions, while binary search does not.

In one dimension, the gradient of a function is just its derivative, which we can approximate discretely around x as $\frac{f(x+\epsilon)-f(x)}{\epsilon}$. The simplest variant of gradient descent minimizes a function `func` given an initial guess for x as follows: given a scaling factor `scale`, repeatedly update x by subtracting `scale` times the gradient of your function at x, until the updates are less than `eps`. Implement this in the function

```
function x=gradientDescent(func,x,scale,eps)
```

4 Gradient Descent in 2 Dimensions

In higher dimensions, the gradient tells us the "direction in which the function increases the fastest". Given a 2-dimensional function $f(x, y)$, its gradient will now be a two-element vector, giving us a direction in (x, y) space. The discrete approximation of the gradient is the pair

$$\left(\frac{f(x+\epsilon, y) - f(x, y)}{\epsilon}\ ,\ \frac{f(x, y+\epsilon) - f(x, y)}{\epsilon}\right)$$

In any number of dimensions, gradient descent works as above: given a scale factor, repeatedly update your current guess by subtracting `scale` times the current gradient, until updates are smaller than `eps`. Write code

```
function [x,y]=gradientDescent2D(func,x,y,scale,eps)
```

that takes an initial guess (x, y) and applies the gradient descent algorithm. Try it out on one of the two-dimensional test functions `g1`, `g2`, ...