MATH3027: Optimization (UK 21/22)

Week 4: Computer Lab 1

Prof. Richard Wilkinson
School of Mathematical Sciences
University of Nottingham, United Kingdom
Please send any comments or mistakes to
r.d.wilkinson@nottingham.ac.uk

This module is about the theory and *practice* of optimization. Many (if not most) of the optimization problems you'll encounter in your careers cannot be solved analytically (i.e., by pen and paper). Instead, we have to resort to numerical methods and the power of computers to crunch through large numbers of calculations.

In the computer lab sessions we will look at how to implement some of the methods we study in the lectures. Any new material that is presented here may be relevant to the coursework (which counts for 40% of the module mark), but will not be tested in the exam: the exam will only cover things in the lecture notes.

Choice of programming language

You are free to choose which ever programming language you like for this module. I would recommend you use one of

- R
- Python
- MATLAB

(or possibly Julia if you know what you're doing). R and Python have a great advantage over MATLAB, in that are free *open source* languages. MATLAB is very powerful but expensive - you may not have access to it after you graduate. Generally speaking, R is the dominant language for the field of statistics, whereas Python dominates in machine learning. I will demonstrate in R, and can probably help if you get stuck in Python, but I don't know MATLAB and don't have it installed on my computer. Your choice of language isn't going to matter hugely, as in terms of the calculations we are doing the differences between the languages don't matter much.

We will start the semester by running the computer labs as online sessions, as this allows me to demonstrate code and teach concepts more easily than if you are spread across a

huge computer room. If it becomes necessary during the coursework, we can revert to the face-2-face sessions with the help of PhD student demonstrators.

Lab 1

This session is just about reminding you how to do the basics in your chosen language. You should make sure you can

- Do basic vector and matrix calculations, e.g., multiply matrices, find inverses and eigenvalues etc.
- Define a function of one or more inputs
- Write for and while loops.
- Plot some functions.

If you have forgotten the basics, please go and do an introductory online tutorial, such as this (link) online tutorial for R.

Write code to compute

$$\sum_{n=1}^{10^7} 1/n$$

Try first writing a for loop to do the computation, and then do it without writing a loop (i.e., first form the vector $(1, 1/2, 1/3, ...)$ and then sum it). Time both methods. Which is faster?

We know this sum diverges as the number of terms tends to infinity. Write a while loop to find when the sum first exceeds 18.

Consider the Branin function.

$$f(\mathbf{x}) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s \text{ for } x_1 \in [-5, 10], x_2 \in [0, 15]$$

The parameters a, b, c, r, s, t are fixed constants, which have default values $a = 1, b = \frac{5.1}{4\pi^2}, c = \frac{5}{\pi}, r = 6, s = 10$ and $t = \frac{1}{8\pi}$.

- Write a function that takes x and the parameters (a, b etc) as an input, and returns the value of $f(\mathbf{x})$. Write the function so that the parameter values are by default set to the values above, but so that they can be changed by the user, i.e., `f(x)` should use the default values above, but `f(x, a=2)` should use $a = 2$, with the other parameters set to their defaults. Make sure the function returns an error if $\mathbf{x}$ is outside of the domain of f.

- Plot the function over its domain.
- Calculate (mathematically) the gradient of the function (with respect to $\mathbf{x}$), and the Hessian matrix.
- We will now use a finite difference approximation to compute the gradients, which can be a useful way to check your calculation above. To approximate the first derivative of a function $g(x)$ we can use

$$g'(x) \approx \frac{g(x+h) - g(x-h)}{2h}$$

for small h. Use this approximation to approximate the gradient of f at $\mathbf{x} = (1,1)^\top$. What happens as h gets smaller? Plot the error in this approximation vs h, using a log-scale for both axes. What happens if h becomes too small? Why does this happen?

Demonstrate why we prefer this central difference method to the forward difference approximation

$$g'(x) \approx \frac{g(x+h) - g(x)}{h}$$

Note that if $\dim(\mathbf{x}) > 1$ we need to perturb one input variable at a time, e.g.,

$$\frac{\partial g}{\partial x_i}(\mathbf{x}) \approx \frac{g(\mathbf{x} + h\mathbf{e}_i) - g(\mathbf{x} - h\mathbf{e}_i)}{2h}.$$

- The numDeriv package in R has functions to compute numerical estimates of derivatives. For example, the gradient and Hessian. Install this package and use it to check your calculations.
- Use one of the default optimization methods in your language of choice to find the minima of f. In R, you can use the command `optim` (the help page can be found in R using `?optim`).

Create a random 5×5 matrix, A say. Write a function to evaluate the quadratic form

$$f(\mathbf{x}) = \mathbf{x}^\top A \mathbf{x}.$$

Compute the gradient vector and Hessian matrix using numerical approximation, and check your answer against the true answer derived in the lecture notes.

The forward difference approach is a first order method, and the error in the derivative scales as $O(h)$, whereas the central difference method is second order, and the error scales as $O(h^2)$. However, these are theoretical error rates, which don't apply on a computer when using finite precision computation. The problem that occurs is that $f(x)$ and $f(x+h)$ will agree for many significant digits. On a computer, we can only store a

limited number of significant digits. Suppose your machine is accurate to 10 significant digits. If $f(x)$ and $f(x+h)$ agree to the first 9 significant digits, when you use a finite difference approximation, your answer can at best be accurate to a single digit!

Another way to compute derivatives is using the *complex step* method. You can read about it at https://nhigham.com/2020/10/06/what-is-the-complex-step-approximation/. The basic idea is that by computing $f(x+\mathrm{i}h)$ we get an estimate of $f(x)$ and $f'(x)$. If we start with a Taylor expansion we get

$$f(x+\mathrm{i}h) = f(x) + \mathrm{i}hf'(x) - \frac{1}{2}h^2 f''(x) + O(h^3)$$

where $\mathrm{i} = \sqrt{-1}$. Then for small h we have the approximations

$$f(x) \approx \mathrm{Re} f(x+\mathrm{i}h) \quad f'(x) \approx \mathrm{Im}\frac{f(x+\mathrm{i}h)}{h}.$$

These both have error $O(h^2)$. Can you see why?

For the function $f(x) = \exp(x)$, try the complex step method for different values of h. Note that you will need first to understand how to create complex numbers in R using the `complex` command.

Automatic differentiation

Many of the huge advances made in artificial intelligence and machine learning over the past 5 years are built upon automatic differentiation software. This allows users to automatically differentiate any function coded on a computer. The wikipedia page gives a good introduction. If you are using Python, have a look at Jax, which is an easy to use automatic differentiation package. TensorFlow is Google's autodiff language, and can be used from Python or R, but is difficult to get started with. MATLAB also has autodiff capability.

If you finish the tasks early, and are interested, take a look at one of these two approaches.