CSC 395 Homework 2 Spring 2024

Due: Friday, March 1

In this homework, you will implement several classic line search algorithms for unconstrained optimization in higher dimensions: Gradient Descent and Newton's Method (again). This assignment is split into two parts, one for each method, and for each you will be expected to submit an individual file showcasing your exploration of these topics. You will submit these files through gradescope.

Make sure that the code you submit is cleanly written and well commented. Whenever questions are asked in the assignment, you should include your answers as comments in your code that are placed in an appropriate location within your file.

Part 1. **Gradient Descent**

In class, we introduced Gradient Descent, a line search algorithm for finding a local minimum of a higher dimensional function $f(\vec{x})$. The core idea was to step in the direction of steepest descent: $-\nabla f$. This gives the update rule

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k \nabla f_k,$$

where $\nabla f_k = \nabla f(\vec{x}_k)$ is the gradient of f at $\vec{x}_k$, and α_k is the step size.

There were a number of conditions required to choose a "good" step size, and we discussed several methods for choose α_k. Your first task in this assignment will be to implement the Gradient Descent method with a number of these variations in a python file titles `hw2p1.py`. For this part, please ensure that your functions take as input functions `f` and `df`, which represent the function $f(\vec{x})$ and its gradient $\nabla f(\vec{x})$ respectively. You will need to take additional inputs, but the details are left to you. You should make sure your functions employ both of the common stopping conditions discussed in class: limiting the number of iterations, and using an error tolerance as a stopping threshold (i.e., stop when the norm of the difference between two consecutive guesses is within the specified tolerance).

Note that $\vec{x}$ should be represented as a `numpy` array, as should the gradient of f, and the stopping condition will require you to compute the norm of the consecutive vector guesses. The norm can be computed several ways in `numpy`: you could compute it directly as $||\vec{x}|| = \sqrt{\vec{x}^T\vec{x}}$ (using `math.sqrt` and the command `x@x`), or you could use the built in `numpy.linalg.norm` function.

Implement Gradient Descent with the choices for α_k listed below. You may choose to implement these as different functions, as a single function with a flag for switching behavior, as a single function that takes a general `alpha` function it calls at each step, or any format you choose so long as it is clear how you have implemented the method.

Once you have implemented each variation, you will be asked test it on a some simple problems like

$$\begin{aligned} f_1(x_1,x_2) &= x_1^2+x_2^2, \\ f_2(x_1,x_2) &= 10^6x_1^2+x_2^2, \\ f_3(x_1,x_2,x_3,x_4,x_5) &= x_1^2+x_2^2+x_3^2+x_4^2+x_5^2. \end{aligned}$$

Specific testing instructions are provided for each variation below. Perform your testing in the `main` block of your file (`if __name__ == __main__:`).

1. Implement a simple version of Gradient Descent that uses a constant step size. You should test this variation on the two problems f_1 and f_2. Try setting a small error tolerance and testing with both a very small step size and a very large step size. What happens? You should notice that small step sizes will converge slowly, while large step sizes can overstep.

2. In class, we discussed that certain objective functions of the form

   $$f(\vec{x}) = \frac{1}{2}\vec{x}^T A\vec{x} - \vec{b}^t\vec{x}$$

   have an optimal step size defined by

   $$\alpha_k = \frac{\nabla f_k^T \nabla f_k}{\nabla f_k^T A \nabla f_k}, \quad \nabla f(\vec{x}) = A\vec{x} - \vec{b}.$$

   Mistake 1: For the problem involving the matrix A and vector b, I choose a bad matrix A. The optimal choice for alpha that we derived only works when the input matrix is symmetric and positive definite. The A that I gave you was neither of these things. So your code will end up diverging. I know some of you spent a while debugging this, and I apologize.

   Note that the $\vec{x}$ that minimizes the given $f(\vec{x})$ in this case will be the vector that satisfies $A\vec{x} = \vec{b}$. Consider the following matrix A and vector $\vec{b}$ that should produce the provided solution $\vec{x}$:

   $$A = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix} \quad \to \quad \vec{x} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}.$$

   Fix 1: A = [[2,-1,0],[-1,2,-1],[0,-1,2]] and b = [2,0,2] will produce x=[2,2,2], and should work since this A is symmetric positive definite.

   Implement this choice for α_k and test it using the given matrix A and vector $\vec{b}$.

3. In class we introduced a more general technique called backtracking, where you start with an initial step size (usually taken to be something like $\alpha = 1$), and shrink it by a contraction factor ρ until the "sufficient decrease" Wolfe condition is met:

   $$f(\vec{x}_k + \alpha_k\vec{p}_k) \le f(\vec{x}_k) + c\alpha_k \nabla f_k^T \vec{p}_k,$$

   where $\vec{p}_k = -\nabla f_k$ for Gradient Descent.

   For this problem, test using $\rho = 0.9$ and $c = 0.1$. Try out all three functions: f_1, f_2, and f_3.

Part 2. **Newton's Method Revisited**

In a new file titled `hw2p2.py`, define Newton's method as you did in the first homework, but now all x values should be vectors, and you will need ∇f instead of $f'(x)$ since we are in higher dimensions. Recall that Newton's Method for minimizing a function $f(\vec{x})$ in higher dimensions uses the update rule

$$\vec{x}_{k+1} = \vec{x}_k - [\nabla^2 f_k]^{-1} \nabla f_k,$$

where $\nabla^2 f_k$ is the Hessian matrix for $f(\vec{x})$ defined as

$$\nabla^2 f_k = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Now, in the main block of this new file, complete the following tasks.

1. Use your Newton's Method function to again find the $\vec{x}$ that minimizes the f_1, f_2, and f_3 from before.

2. For f_1 and f_2, compare the number of iterations it takes Newton's method to converge versus the various attempts you made previously using Gradient Descent. Does Newton's Method seem to have the same issues with scaling?

3. For f_3, compute the error at each iteration (the correct solution is $\vec{x} = \vec{0}$), and perform a convergence study (similar to Part 4 of the first homework). You should see that Newton's Method gives quadratic convergence.

Mistake 2:
You might notice that your Newton's method takes only a single step to get to the correct solution for f1, f2, and f3. Why? Remember that optimizing a function with Newton's method is the same as finding a 0 of its derivative, which can be done exactly for linear functions. All three f1, f2, and f3 are quadratic functions, so their derivatives are linear. This means Newton's method should take only a single step. I will again need to fix this to give you a more meaningful test.

Fix 2:
Try f4(x1,x2) = cos(x1) + sin(x2) for Newton's method. This should have a local minimum at x1=pi and x2=3*pi/2. Search starting somewhere near that point (try something like x0 = [3,4]). This should converge quadratically with Newton's method.