AMS 595 Final Project

Ashler Herrick

December 2020

1 Project Objectives

The goal of the project was to implement the methods described in Steven Boyd and Lieven Vandenberghe's book "Convex Optimization" in C++ to optimize convex functions.[1] The original goal was to implement gradient descent, Newton's method and an interior point method in C++, however, the task turned out to be much more complicated than I had assumed. Gradient descent was the only method that was fully implemented, however, there was progress made on Newton's method as well. Interior point method's proved to be outside the scope of my ability for this project.

The motivation for the project was due to a recent interest taken in optimization theory, with the goal of gaining a deeper understanding from both a numerical and theoretical perspective. Implementation of the methods requires an understanding of the concepts to a degree which a mere reading of the textbook will not give. Further, when using a language such as C++, essentially everything must be coded from scratch. Operations such as vector addition and scalar multiplication must be coded in using arrays or vectors, rather than being implemented by default as they are in Python.

The approach was rather straightforward: implement the methods described in Convex Optimization in C++. The book contains descriptions of convex optimization algorithms described in three or four steps, along with the computational analysis of the algorithm. It does not provide how to perform each step. For example, the calculation of the gradient vector or hessian matrix was not provided in the text, so I looked to Numerical Recipes: The Art of Scientific Computing for a reference on implementing these.[2] The book does provide code in C++, but I thought it better to only use the ideas and formulas, and code everything from scratch.

2 Techniques and Tools

It is important to first define what convex optimization is. A convex optimization problem of the form

$$\begin{array}{lll} \text{minimize} & f_0(x) & \\ \text{subject to} & f_i(x), \leq 0 & i = 1, ..., m \\ & a_i^T x = b_i, & i = 1, ..., p \end{array} \tag{1}$$

for which the objective $f_0(x)$ is convex, the inequality constraint functions $f_i(x), i \neq 0$ must be convex, and the equality constraint functions $h_i(x) = a_i^T x - b_i$ must be affine.[1] A set C is convex if:

- $C \subseteq S$ where S is an ordered field.
- For all $x, y \in C$ we have $(1-t)x + ty \in C$ for $0 \leq t \leq 1$.

Loosely speaking, this means that for any point in the set, we can draw a line to any other point in the set, and that line will be wholly contained within the set. A function f is convex if

- f is function from a convex set to $\mathbb{R}$.
- For all $x, y \in \mathbf{dom}(f)$ we have $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$.

Again, loosely speaking, this just means the function has upward curvature.

These definitions are why gradient descent works, because for all bounded convex functions there exists a global minimum. If the function is unbounded on the set and one extended the set to include $\pm\infty$ then there always exists a global minimum. Convex functions have the property that if f is convex, then $-f$ is concave, therefore convex optimization methods will work for both the minimization of convex functions and the maximization of concave functions.

As was mentioned before, the initial goal was to implement three convex optimization methods, each for a different class of problem. The method of gradient descent only works for unconstrained problems, Newton's method works for equality constrained problems and interior point methods work for inequality constrained problems. The constraints dictate the amount of complexity required with the method, i.e gradient descent is the simplest and interior point methods are the most complex, and the constraints dictate what method will work for another set of constraints, that is, gradient descent will not work for equality or inequality constrained optimization but interior point methods will work for all classes of convex optimization problems.

To implement gradient descent, one first starts with an objective function that is convex, then calculates and ”moves” along the negative gradient until the gradient is zero. The algorithm is iterative, so it doesn't really ”move”, but instead successively updates the optimal value until the stopping condition of the gradient sufficiently close to zero is reached, at which point the algorithm declares the point optimal.

2.1 Calculation of the Gradient Vector

Before we can perform gradient descent, we need to determine what the gradient of f is. For a function $f : \mathbb{R}^n \to \mathbb{R}$ we have the definition

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

When subsitution some small value for h instead of formally calculating the limit, this is called a finite difference approximation. It is more numerically stable to use a centralized finite difference, that is take

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x+h) - f(x-h)}{2h}, \text{ for } h = x_i\sqrt{\epsilon},$$

as defined in Numerical Recipes.[2]

2.2 Gradient Descent

Now that we can calculate the gradient vector, we can effectively perform gradient descent. The algorithm defined in the code works by performing the following steps:

- Define, $\alpha \in (0, 0.5), \beta \in (0, 1), \nu = 2^{-8}$.
- Start at an initial x, calculate $\nabla f(x)$
- Update $x := x - t\nabla f(x)$, and $t := \beta t$.

while $\tilde{f}(t) = f(x - t\nabla f(x)) \geq f(x) - \alpha t||\nabla f(x)||_2^2$ or $||\nabla f(x)|| > \nu$, and the total iterations is less than 100. Note that $||x||_p$ means the L^p-norm of x.

There are multiple exit conditions because during testing I found that one was often not enough, that is, the gradient would not be close zero but $\tilde{f}$ would be sufficiently small, or the algorithm would not converge at all, which happened for functions with large partial derivatives, for example e^x or $-\log(x)$.

2.3 Calculation of the Hessian Matrix

Newton's Method was another method I initially wanted to implement, but proved to be outside the scope of my abilities. Newton's method uses information about the curvature of the function in order to determine the step direction and length. This requires a calculation of the Hessian matrix

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

Notice that since $\frac{\partial^2 f}{\partial x_1 \partial x_2} = \frac{\partial^2 f}{\partial x_2 \partial x_1}$ the Hessian is symmetric, and the diagonal is the second derivative with respect to one variable. We can calculate the Hessian matrix using finite difference approximations, which I implemented in the C++ code. The function takes a vector x for which it computes the Hessian matrix at x. First a two-dimensional vector is created with length x and width x, and is then filled with zeros in order to access the elements with with the vector.at() function. The diagonal elements are calculated with the formula

$$\frac{\partial^2 f}{\partial x_i^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

and the upper triangular off-diagonal elements are calculated with the formula

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \approx \frac{(f(x_i+h, x_j+h) - f(x_i+h, x_j-h)) - (f(x_i-h, x_j+h) - f(x_i-h, x_j-h))}{4h^2}.$$

Since the matrix is symmetric we can save operations and only calculate the upper triangular portion of the matrix and then simply set $H_{ij} = Hji$ where H_{ij} is the jth entry in the ith row vector.

3 Conclusions

As implemented, the gradient descent function works well for well-behaved functions. By well-behaved I mean the gradient calculation doesn't lead to overflow, and that there is a global minimum. For example, the function e^{-x} is convex but doesn't have a minimum. The function determines the minimum for this function to be at $x^* = 3.15275$ and $f(x^*) = .0427346$. We can tweak the values of α and β to get an x^* further out, but the result is largely the same. We see that gradient descent fails in this case, as we know the function to decrease as x tends to infinity.

An example of an ill-behaved function is a non-quadratic $f(x_1, x_2) = e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} + e^{-x_1-0.1}$. This is provided as an example in Boyd's Convex Optimization, where it is solved with gradient descent. Clearly, we don't need gradient descent to optimize this function, we can see that the minimum is zero, but it is an instructive case to consider as it somewhat pathological. The problem I noticed is that the gradient calculation would very quickly get huge. We know the minimum of this function to be $x^* = (0, 0)$ but even with an initial value of $x_0 = (.1, .1)$ the gradient at x_0 is $(1.27192, 1.82712)$ which moves x_1 to $x_1 = (-1.17195, -1.72712)$ for which the gradient is $(46.9505, -149.605)$ at which point x_2 will move so far from zero that we get overflow in the gradient calculation. I managed to remedy this in a somewhat hacky way of simply setting a maximum step size of 1. This does limit overflow, but it can also make convergence slower. If the function is far away from the minimum then it will take many iterations to reach it, although one could simply rerun the program with the previous "solution" as a new initial value until the true minimum is reached.

One of the barriers to implementing Newton's method was the calculation of the Hessian matrix. I did manage to solve this and as far as I can tell it works effectively. Newton's method requires not only the Hessian matrix but also the *inverse* of the Hessian. This proved to be outside the scope of my ability. There are packages that can be implemented to invert matrices in C++, but I wanted to use minimal libraries and code everything myself. I did do research on implementing it via LU-decomposition and Cholesky factorization, which I may eventually implement as a hobby project.

References

[1] Lieven Vandenberghe Steven Boyd. *Convex Optimization.* Cambridge University Press, 2004.

[2] William T. Vetterling Brian P. Flannery William H. Press, Saul A. Teukolsky. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 3rd edition, 2007.