1 Introduction

The purpose of this project was to implement an unconstrained optimization routine designed to minimize the number of function calls. I elected to implement a quasi-Newton method in Python.

2 Solver

My solver consists of two major pieces: a quasi-Newton method to find the search direction, and simple backtracking to find the step size. I compute the gradient using finite differences with a fixed step size, and measure convergence based on the 2-norm of the gradient.

2.1 Quasi-Newton Methods

The solver I chose to implement is a simple quasi-Newton solver with a backtracking line search. Quasi-Newton, as the name implies, attempts to use Hessian information for a local quadratic fit, which informs the step direction. The update formula for quasi-Newton is

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \boldsymbol{H}_k \nabla f(\boldsymbol{x}_k), \tag{1}$$

where $\boldsymbol{H}_k$ is an approximation to the inverse of the Hessian $\nabla^2 f$. We start with a symmetric positive-definite approximation of $\nabla^2 f$ (in practice $\boldsymbol{H}_0 = \boldsymbol{I}$) and perform successive rank one updates on $\boldsymbol{H}_k$ based on gradient information. Since the Hessian is the second derivative of f, we can consider $\nabla^2 f$ to be the derivative of the gradient ∇f. From a truncation of the Taylor series of ∇f, we find

$$\left[\nabla^2 f(\boldsymbol{x}_k)\right](\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k). \tag{2}$$

Many schemes have been devised to satisfy the quasi-Newton condition (Eq. 2) given above. Belegundo and Chandrupatla recommend the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) scheme when using an approximate line search. Adopting the notation $\boldsymbol{\delta}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{\gamma}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$, the BFGS update is

$$\boldsymbol{\nabla^2 f}_{k+1}^{BFGS} = H - \left(\frac{\boldsymbol{\delta\gamma}^T H + H\boldsymbol{\gamma\delta}^T}{\boldsymbol{\delta}^T\boldsymbol{\gamma}}\right) + \left(1 + \frac{\boldsymbol{\gamma}^T H \boldsymbol{\gamma}}{\boldsymbol{\delta}^T\boldsymbol{\gamma}}\right)\frac{\boldsymbol{\delta\delta}^T}{\boldsymbol{\delta}^T\boldsymbol{\gamma}}. \tag{3}$$

Comparing Formula 3 with the Wikipedia article, I think Belegundo and Chandrupatla mistakenly gave the Hessian update formula in place of the inverse formula – I haven't checked their math thoroughly, but using the above for the inverse Hessian gives garbage results. Rather than muck around with direct matrix inversions, we can perform a rank one update on the inverse directly, resulting in

$$\boldsymbol{H}_{k+1}^{BFGS} = \left(\boldsymbol{I} - \frac{\boldsymbol{\delta\gamma}^T}{\boldsymbol{\delta}^T\boldsymbol{\gamma}}\right) H_k \left(\boldsymbol{I} - \frac{\boldsymbol{\gamma\delta}^T}{\boldsymbol{\delta}^T\boldsymbol{\gamma}}\right) + \frac{\boldsymbol{\delta\delta}^T}{\boldsymbol{\delta}^T\boldsymbol{\gamma}}. \tag{4}$$

Applying Formula 4 directly results in some nasty numerical precision errors – in cases where $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$ are nearly orthogonal, their dot product approaches zero, leading to issues of numerical precision. In practice, I limit the value $1/\boldsymbol{\delta}^T\boldsymbol{\gamma} \leqslant 1000$ to avoid such issues – this was a technique I found in the SciPy implementation of BFGS.

2.2 Line Search

The Quasi-Newton method gives us a search direction, but unlike true Newton's method we do not solve the quadratic problem to find the step size. For my line search, I used a simple backtracking method. I start with a fixed initial step size $\alpha = 1.5$ and repeatedly sample points along the line with geometrically smaller step sizes (factors of 1/2) until a desired reduction in function value is achieved. I found that this method works decently well in practice, but is outperformed by SciPy's implementation of the Wolfe conditions.

3 Results

3.1 Finite Differences

For the project submission, I employed Finite Differences to approximate the gradient. The results of this version of the solver are given below.

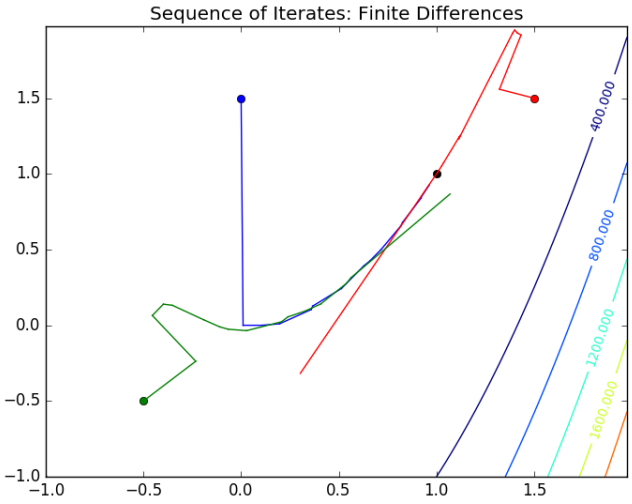

Figure 1: Trajectories of solver in function domain. Note that the solver tends to get 'lost' at various points in the domain – the green trajectory takes an excursion off to the left before settling back into a path to the optimum value, while the red trajectory overshoots considerably before returning to the optimum. Comparing against SciPy, I believe that my linesearch is the problem, as my solver takes roughly the same turns as SciPy, but different step sizes. It bothers me that my line search is quite so poor, but I really need to get back to working on other things...

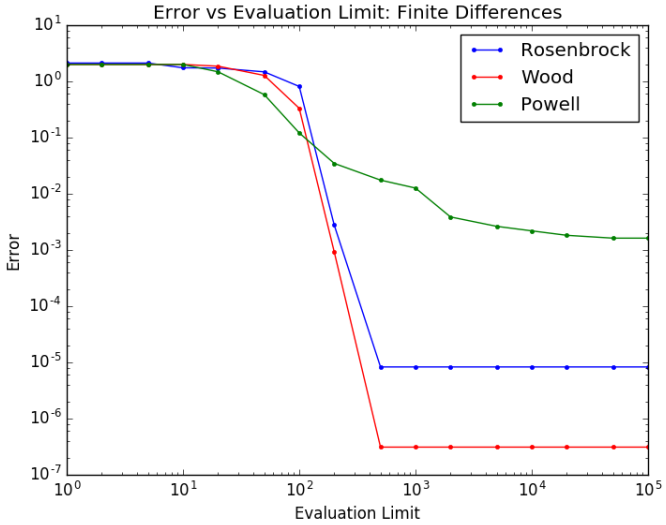

Figure 2: Convergence plots for solver on various objective functions. The solver 'bottoms out' at some point, due to numerical precision issues. This differs based on the function – my convergence criterion is based on the 2-norm of the gradient, so if the function is shallow near the optimum, the solver will quit further away than if the function were steep.

3.2 Automatic Differentiation

In working on the project, I also implemented my solver using automatic differentiaion based on the 'ad' package in Python. While this version was unable to interface with the submission code, I'm showing the results here because they're interesting to compare.

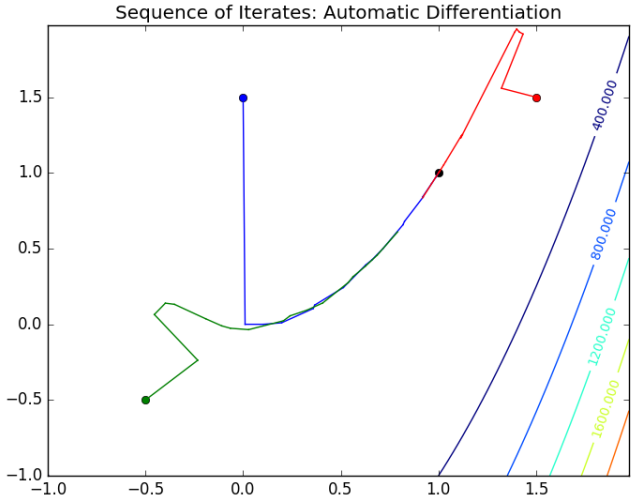

Figure 3: Qualitatively, automatic differentiation performs similarly. Some of the excursions in the function domain are still dominated by a poor line search, note that the red overshoot is far less though. A more accurate approximation of the gradient leads to a better line search, and faster convergence.

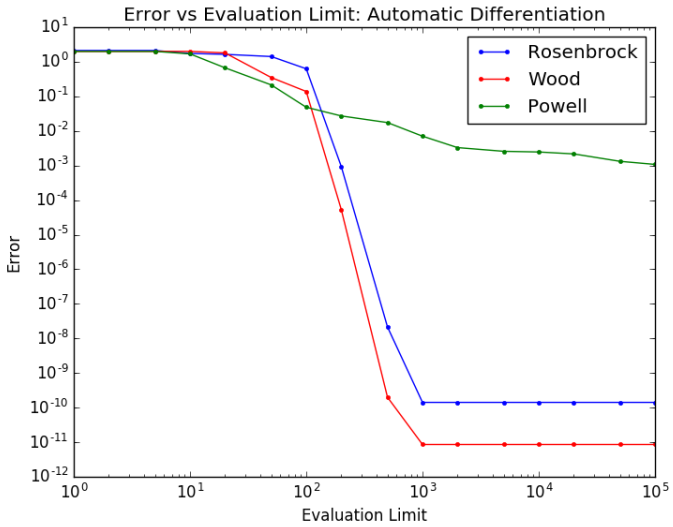

Figure 4: Note that automatic differentiation is able to reach a tighter tolerance than the finite difference implementation; this is likely due to both the greater accuracy and my convergence criterion, which is based on the 2-norm of the gradient.

References

[1] Belegundo, A.D., Chandrupatla, T.R., *Optimization Concepts and Applications in Engineering.*.

[2] Jones, E., Oliphant, T., Peterson, P., *SciPy: Open Source Scientific Tools for Python*, 2001.

[3] Wikipedia contributors, *Quasi-Newton method*, retrieved 14 April 2016 4:00 UTC

[4] Lee, A., *ad 1.3.2: Fast, transparent first- and second-order automatic differentiation*