1 Introduction

The purpose of this project was to implement a constrained optimization routine designed to minimize the number of function calls. Since infeasible solutions are disqualified, I elected to use an interior point method, as an exterior point method would converge from outside the feasible set, and therefore would always be disqualified for any finitely-converged solution.

2 Solver

My solver consists of two major pieces: a Feasibility Solver, and an Interior Point Solver. Since I had implemented both a Conjugate Gradient[1] and quasi-Newton solver for the previous project, I reused my old code to reduce coding time, and focus on learning new concepts.

2.1 Feasibility Solver

Interior Point Methods require a feasible point as an initial guess, which will not be guaranteed for an arbitrary choice of $\boldsymbol{x}_0$. Thus, my first step is to solve the Feasibility Problem, and find a point $\boldsymbol{x} \in \Omega$. Belegundo and Chandrupatla recommend solving this via an Exterior Point Method, with objective function [1]

$$\hat{f} = \sum_{j=1}^{m} \max(0, g_j + \epsilon)^2. \tag{1}$$

I minimize Equation 1 via a Conjugate Gradient method. I elected to use CG for the Feasibility Solver, as I found that my quasi-Newton method tended to meander a bit in Project 1. Since I don't need particularly accurate convergence with the Feasibility Problem, I aimed to use CG to quickly find the direction towards the feasible set, and use backtracking with a particularly large initial step size to get there quickly. In practice, I'm not sure if this does any better than a quasi-Newton method with a similar line search – I didn't have time to check the two against each other.

2.2 Interior Point Solver

An Interior Point Method maintains feasibility by adding a Barrier Function term to the objective function. I elected to use a log-barrier function of the form

$$B(\boldsymbol{x}) = -\sum_{j=1}^{m} \log[-g_i(\boldsymbol{x})], \tag{2}$$

with some minor modifications. As Belegundo and Chandrupatla recommend, I define $\log(-g_j) = 0$ when $g_j(\boldsymbol{x}) < -1$, in order to assure $B \geqslant 0$.[1] Additionally, when $g_j(\boldsymbol{x}) \geqslant 0$, I define $\log(-g_j) = \infty$. In some cases, my line search extends beyond the feasible set: Setting the Barrier Function to a singular value is a cheap way to encourage the solver to stay within Ω. With $B(\boldsymbol{x})$ defined, we solve a sequence of unconstrained optimization problems

$$\text{min. } f(\boldsymbol{x}) + \frac{1}{r} B(\boldsymbol{x}), \tag{3}$$

parameterized by $r > 0$. I define the problem sequence by starting with an initial value for r (In practice $r_0 = 100$), multiply r by a factor of two for each subproblem, and define a gradient tolerance for convergence $\|\nabla f\|_2 < 1/r$. This allows early problems in the sequence to be somewhat loosely converged, allowing us to get near the optimal point with fewer function calls, saving our computational budget for when we're close

[1] I implemented a CG solver in Project 1, but did not end up using it, and so did not discuss it in the previous writeup. The original intent was to have a solver which would be computationally tractable for high-dimensionality problems, as quasi-Newton requires $O(n^2)$ storage for the inverse Hessian. I figured you folks probably wouldn't throw a particularly high dimensional problem at us though, and shelved the solver until now. In retrospect, L-BFGS probably would have been a better solution for this problem anyway...

enough for tight convergence. In practice, I found that increasing r too much lead to stability issues, so I set a ceiling at $r_{\max} = 1000$ and set the smallest gradient tolerance to machine epsilon (which is never achieved via $1/r$, but set at $r_{\max}$).

3 A Note on Finite Differences

The TA (Jon Cox) recommended I try to call Julia from Python, in order to take advantage of the Dual Numbers package. I thought this was a great idea! Unfortunately, I elected not to figure this out – looking at various Python-Julia interfaces online, most claimed to be fairly 'immature'. I figured since the coders owned up to the fact that their interfaces were not super stable, it would take a fair bit of work to get this kind of functionality working. Additionally, since I'm not learning Julia at the moment, it seemed like waste of time to try hacking together an interface for a language I don't use, for a piece of code I'll only use once (for this project). Sorry guys, you haven't sold Julia to me just yet. (Maybe I'll pick it up once the language matures a bit more – the unicode character thing is pretty sick.)

4 Results

This section presents some results from the solver implementation described above.

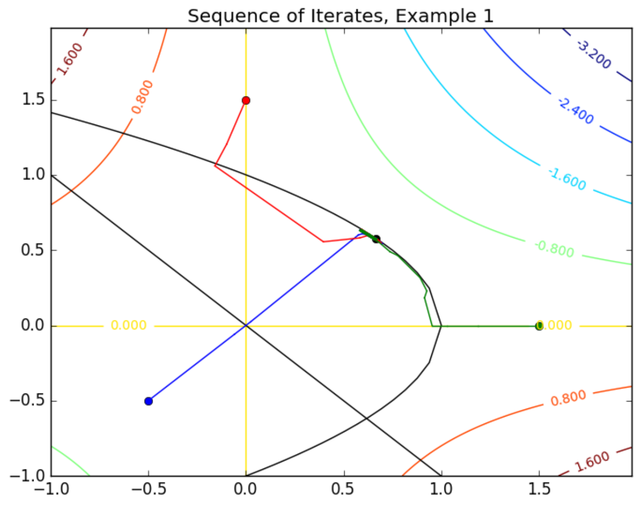

Figure 1: Trajectories of solver in parameter space on Example function 1. This figure clearly shows the two phases of the solver. Notice how for each starting point the solver goes straight for the feasible region, roughly orthogonal to the nearest constraint. Here the Feasibility Solver gets us to the feasible set very quickly! The solver then switches to the Interior Point Solver, which tends to follow the constraints to the optimal point – this is because the Barrier Function creates something of a 'channel' which runs towards $\boldsymbol{x}^*$.

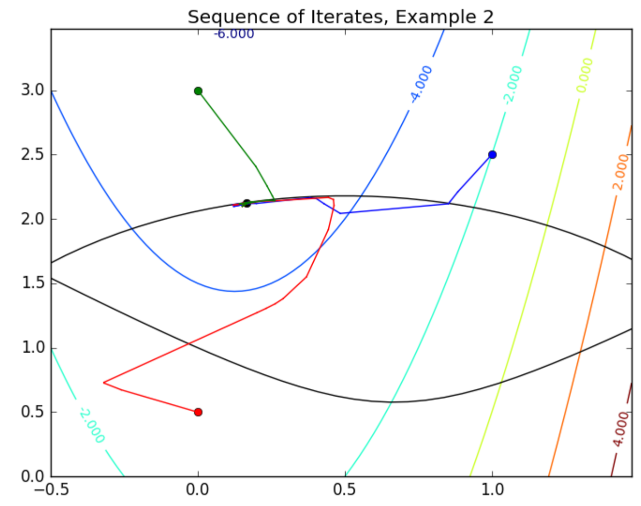

Figure 2: Trajectories of solver in parameter space on Example function 2. The green and blue trajectories in the Feasible Phase behave similar to those in Example 1. However, the red trajectory seems to get lost. I am unsure whether this is an issue with the exterior objective function (Eq. 1) or my CG solver.

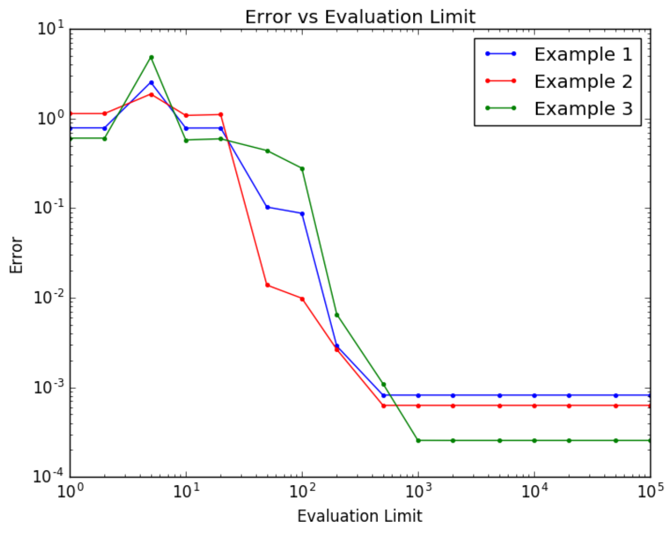

Figure 3: Convergence plots for solver on various objective functions. The error is roughly non-convergent for low iteration counts – the solver needs enough steps to solve the feasibility problem before it can make any real progress. The center of the curve (around an Evaluation Limit of 10^2) shows the rapid convergence rate of a quasi-Newton method. 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.

References

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