Nothin' But Net

Paul Constantine

Saturday, March 18, 2006

1 Introduction

With 3.14 seconds remaining in the last game of the regular season, the ICME Nablas intramural basketball squad trails the EE Fightin' FFT's by 2 points. The inbound pass goes to Murray who jukes EE's Stephen Boyd out of his convex hull and passes to Constantine. Constantine, wide open, throws up a prayer from just outside the three-point line...NOTHIN' BUT NET! The Nablas win! Paul turned in his project! We can all get our grades!

This project has been an adventure and a rite of passage for every student in ICME. I am quite pleased to be finished with it. My project is organized into the following sections: In the first section I describe a general optimization routine that I built in Matlab. The second section derives the model for the hanging net problem and discusses its implementation in Matlab. The third section shows the results of applying my optimization routine to the hanging net problem. And the fourth and final section describes some other directions that I could take this project.

I would like to acknowledge the help, support, and encouragement of my office mates David Gleich, John Bodley, Mike Atkinson, Jeremy Kozdon, Esteban Arcaute, and Will Fong. My conversations with them greatly enhanced my understanding of this problem and the subject of numerical optimization in general.

2 Code

In this section, I will describe an optimization routine for solving a general unconstrained problem. I took my optimization routine for this project directly from [1] and made some slight modifications. I implemented a bisecting line search and incorprated it into a standard BFGS method. In what follows, I will discuss the algorithm for the line search and BFGS method and detail my variations. I will also display and discuss the results from testing my optimization routine on a number of standard test functions.

For notation, assume we want to solve the problem

$$\min_{\mathbf{x}} f(\mathbf{x}).$$

At each iteration k, I may refer to $f_k = f(\mathbf{x}_k)$ and $\nabla f_k = \nabla f(\mathbf{x}_k)$.

2.1 Line Search

The line search that I implemented is given in [1] as Algorithms 3.2 and 3.3. For reference, I provide these algorithms here, including my own parameter choices and slight modifications. Let $\phi(\alpha) = f(x + \alpha p)$ where p is a feasible search direction. Then $\phi'(\alpha) = \nabla f(x+\alpha p)^T p$ is the gradient of f along the direction given by p.

Line Search Algorithm

1: **if** $\alpha = 1$ satisfies the Wolfe conditions **then**
2: $\quad\alpha^* \leftarrow 1$ and **stop**.
3: **end if**
4: Set $\alpha_0 = 0$, $\alpha_1 = 10^{-7}$, and $\alpha_{\max} = 1$.
5: Set $c_1 = 10^{-5}$ and $c_2 = 0.99$.

6: **repeat**
7: Evaluate $\phi(\alpha_i)$.
8: **if** $\phi(\alpha_i) > \phi(0) + c_1\alpha\phi'(0)$ or $[\phi(\alpha_i) \geq \phi(\alpha_{i-1}$ and $i > 1]$ **then**
9: $\alpha^* \leftarrow \text{zoom}(\alpha_{i-1}, \alpha_i)$ and **stop**.
10: **end if**
11: Evaluate $\phi'(\alpha_i)$
12: **if** $|\phi'(\alpha_i)| \leq -c_2\phi'(0)$ **then**
13: $\alpha^* \leftarrow \alpha_i$ and **stop**.
14: **end if**
15: **if** $\phi'(\alpha_i) \geq 0$ **then**
16: $\alpha^* \leftarrow \text{zoom}(\alpha_i, \alpha_{i-1})$ and **stop**.
17: **end if**
18: Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{\max})$.
19: **if** $i > 20$ **then**
20: $\alpha^* \leftarrow 10^{-8}$ and **stop**.
21: **end if**
22: **until** Stopped

My own variations from [1] are noteworthy. I first check if $\alpha = 1$ satisfies the Wolfe conditions. If $\alpha = 1$ consistently satisifies the Wolfe conditions, then we are in a region where we a fortunate enough to get the quadratic convergence of Newton's method. This also speeds up the line search significantly, since I am content - in fact, pleased - to take a step of 1. I chose very conservative values for $c_1 = 10^{-5}$ and $c_2 = 0.99$ to ensure that the α I select will almost always satisfy the Wolfe conditions. Some particular problems may warrant more aggresive choices for these parameters, but my general routine remains conservative. Another item that I consider noteworthy was not my choice but was a feature of the algorithm in [1]. He begins with a very small α and increases it until it satisfies the Wolfe conditions, as opposed to starting with a large α and decreasing. I decided to keep this feature for my general routine since it errs conservative. Admittedly, this will likely result in many more steps. Finally, note that if my line search does not find a suitable α in 20 iterations, it takes a step of $\epsilon = 10^{-8}$.

Bisecting Zoom

1: **repeat**
2: $\alpha_j = (\alpha_{\text{hi}} + \alpha_{\text{lo}})/2$
3: Evaluate $\phi(\alpha_j)$.
4: **if** $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ or $\phi(\alpha_j) \geq \phi(\alpha_{\text{lo}}$ **then**
5: $\alpha_{\text{hi}} \leftarrow \alpha_j$
6: **else**
7: Evaluate $\phi'(\alpha_j)$
8: **if** $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$ **then**
9: $\alpha^* \leftarrow \alpha_j$ and **stop**.
10: **end if**
11: **if** $\phi'(\alpha_j)(\alpha_{\text{hi}} - \alpha_{\text{lo}}) \geq 0$ **then**
12: $\alpha_{\text{hi}} \leftarrow \alpha_{\text{lo}}$
13: **end if**
14: $\alpha_{\text{lo}} \leftarrow \alpha_j$
15: **end if**
16: **if** $j > 20$ **then**
17: $\alpha^* \leftarrow 10^{-8}$ and **stop**.
18: **end if**
19: **until** Stopped

For the zoom method, I chose bisection. This is robust since it lacks possibility for diverging like Newton's method or polynomial interpolation. Again, this follows the theme of using a robust, conservative method for my general routine.

2.2 BFGS

I chose the standard BFGS method for my general optimization routine because of its robustness. Also the method was given in great detail in [1]. In the notation below, H is the approximation to the inverse Hessian of f. Using this as opposed to approximating the actual Hessian allows us to avoid solving a linear system to compute the search direction. I begin with an initial approximation of the identity matrix. At each iteration, the Broyden update incorporates new information from the curvature of the objective function into the approximation for the inverse Hessian.

BFGS Method

1: Given x_0, $\epsilon = 10^{-7}$, and $H_0 = I$.
2: **while** $\|\nabla f_k\|_\infty > \epsilon$ **do**
3: Compute search direction $p_k = -H_k \nabla f_k$.
4: Compute α with the line search algorithm above.
5: $x_{k+1} = x_k + \alpha p_k$.
6: **if** α satisfies the Wolfe conditions **then**
7: $s_k = x_{k+1} - x_k$
8: $y_k = \nabla f_{k+1} - \nabla f_k$
9: $\rho_k = 1/(y_k^T s_k)$
10: $H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$
11: **end if**
12: **end while**

Note that I only update the approximation for the inverse Hessian when the computed steplength satisifies the Wolfe conditions. We are only guaranteed convergence in the approximation of the inverse Hessian if we satisify the Wolfe conditions. Therefore if my steplength does not comply, then I retain the information from the curvature already acquired and start again some small distance ϵ away. I chose to stop iterating when the infinity norm of the gradient was smaller than 10^{-6}. This ensures that each componenent of the gradient is small. And 10^{-6} seems small enough to let the algorithm converge on relatively flat surfaces.

2.3 Test Cases

I tested my general routine on a battery of standard test functions for optimization routines. The Matlab functions for these tests and their computed minimums were passed around, and I acquired them from the information ether of my office. For more information on these test functions, see [2]. Here is a table of the results of my optimization routine applied to the test functions.

Function	Correct Min?	Correct Min Val?	Converged?	Iterations
Rosenbrock	Yes	Yes	Yes	1
Froth	Yes	Yes	Yes	10
Powell Badly Scaled	Yes	Yes	Yes	157
Brown Badly Scaled	Yes	Yes	Yes	12
Beale	Yes	Yes	Yes	15
Jenrich and Sampson	Yes	Yes	Yes	22
Helical Valley	Yes	Yes	Yes	31
Bard	N/A	Yes	Yes	30
Gaussian	Yes	Yes	Yes	3
Meyer	N/A	N/A	No	N/A
Gulf	No	No	Yes	1
3-D Box	Yes	Yes	Yes	15
Powell Singular	Yes	Yes	Yes	24
Wood	Yes	Yes	Yes	26
Kowalik and Osborne	N/A	Yes	Yes	28
Brown and Dennis	N/A	Yes	Yes	26
Osborne 1	N/A	Yes	No	N/A
Biggs EXP6	N/A	Yes	Yes	40
Osborne 2	Yes	Yes	Yes	52
Watson	N/A	Yes	Yes	69
Extended Rosenbrock	Yes	Yes	Yes	95
Extended Powell Singular	Yes	Yes	Yes	29
Penalty I	N/A	Yes	Yes	157
Penalty II	N/A	Yes	Yes	361
Variably Diminished	Yes	Yes	Yes	22
Trigonometric	N/A	Yes	Yes	26
Discrete Boundary Value	N/A	Yes	Yes	18
Discrete Integral Equation	N/A	Yes	Yes	10
Broyden Tridiagonal	N/A	No	Yes	29
Broyden Banded	N/A	Yes	Yes	53

My routine converged on all but two of the test functions, one of which it did acheive the minimum function value. However, to maintain full disclosure, note that when I claim that my routine produced the same results as the given minima and function values, the results were actually within around 10^{-4}. I considered this sufficient accuracy to pass the test. Equipped with a validated optimization routine, we are now ready to hang the net.

3 Problem Definition

The problem as given in class is to minimize the potential energy of a discrete net hanging under the force of gravity from a set of given points with the distance between adjacent nodes fixed. In this section I derive a model for this problem including the constraints, an objective function and its gradient, and boundary conditions. I also discuss an efficient way to implement these functions in Matlab such that they are amenable to my optimization routine. In the last subsection, I briefly consider some of the factors involved in choosing the initial $\mathbf{x}_0$ for the optimization routine.

3.1 The Model

We assume the net is built from a finite number of point mass nodes with massless rigid connectors between adjacent nodes. The connectors have a fixed constant length d. We can impose a Cartesian coordinate system on the set of nodes so that node i has position (x_i, y_i, z_i). The potential energy of node i is given by $m_i g h_i$. Assume we have imposed the grid such that for each node the height is equal to the z component of

the position, i.e. $h_i = z_i$. We can also assume that the mass of each node is the same and equal to 1 so that $m_i = 1$. Also, we can choose the units such that the gravitational constant is equal to 1, or $g = 1$. Then the potential energy of the net is given by sum of heights of each node in the net, i.e.

$$\text{potential energy of net } = \sum_i m_i g h_i = \sum_i z_i$$

Let us choose the grid so that the net hangs from points fixed at height zero. Then the height of each hanging node will be negative. Therefore the sum of heights will also be negative.

For this project, I consider the following type of net. Suppose we have stretched the net perfectly flat so that $z_i = 0$. Then the net looks like a uniform mesh in x and y. In other words there is a set of nodes on a rectangular boundary and each interior node is connected to exactly four adjacent nodes. For simplicity assume $\Delta x = \Delta y$. I have only considered the case of "square" nets such that the number of nodes in the net is a perfect square and length and width of the flattened net are equal. To hang the net, imagine taking a uniform mesh, moving its corners towards the center, and trying to compute how it is a going to hang under a constant force. I could have easily generalized to the case where the width is not equal to the length, but I chose to consider the simpler case.

It will be useful to think of the net as a graph of nodes and edges. Then we can apply some useful graph theoretic tools in the analysis.

3.2 Constraints

Obviously, the minimum of the unconstrained net is $-\infty$; the nodes will fall forever. Therefore we must constrain the nodes to have some meaningful solution. To constrain the distances between adjacent nodes in the net, we can use the 2-norm and write

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} = d$$

for adjacent nodes i and j. For simplicity, assume $d = 1$. Then we can remove the square root and write our constraint

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 = 1$$

for adjacent nodes i and j. To hang the net by its corners, we can constrain the positions of the nodes on the corners of the net $\{\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4\}$. Assuming one corner is pinned at the origin, we have for a given length L and width W,

$$(x_{c_1}, y_{c_1}, z_{c_1}) = (0, 0, 0) \quad (x_{c_2}, y_{c_2}, z_{c_2}) = (L, 0, 0) \quad (x_{c_3}, y_{c_3}, z_{c_3}) = (0, W, 0) \quad (x_{c_4}, y_{c_4}, z_{c_4}) = (L, W, 0)$$

I only considered nets for which $L = W$, but the more general case is straightforward. Choosing L requires a bit of intuition. Suppose there are N nodes in the net where N is a perfect square. Then the length and width of the "flattened" net is $\sqrt{N}$. Therefore, to get a net with some sag in it, we need to choose $L = W < \sqrt{N}$. In my code, I defined a parameter $0 < \beta < 1$ and set

$$L \;=\; W \;=\; \beta\sqrt{N}$$

The total number of constraints in our system is given by the number of edges in the graph of our net $2(N - \sqrt{N})$ plus three constraints for each corner of the net. We can represent the constraints as a function $c : \mathbf{R}^{3N} \rightarrow \mathbf{R}^{2(N-\sqrt{N})-12}$ that takes the position components of each node and returns a vector of the homogeneous version of each constraint. If we let

$$\mathbf{x} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ \vdots \\ x_N \\ y_N \\ z_N \end{bmatrix}$$

then for the set of edges $\{(i,j)\}$ in the graph of the net we have

$$c(\mathbf{x}) = \begin{bmatrix} \vdots \\ (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 - 1 \\ \vdots \\ x_{c_1} \\ y_{c_1} \\ z_{c_1} \\ x_{c_2} - \beta\sqrt{N} \\ y_{c_2} \\ z_{c_2} \\ x_{c_3} \\ y_{c_3} - \beta\sqrt{N} \\ z_{c_3} \\ x_{c_4} - \beta\sqrt{N} \\ y_{c_4} - \beta\sqrt{N} \\ z_{c_4} \end{bmatrix}$$

3.3 Objective Function

Since the distance constraint is nonlinear, we can employ the penalty method and create an objective function that weights the constraint violation by a penalty parameter ρ. The idea behind this method is to take the homogeneous version of each constraint, square it, and add each one to the objective function weighted by ρ. Theoretically, we take the limit of the objective function as $\rho \to \infty$ where we can perfectly satisfy the constraints and find the minimum potential energy. Therefore our objective function can be written

$$f(\mathbf{x}) = \sum_i z_i + \rho c(\mathbf{x})^T c(\mathbf{x})$$

It will be useful to group the constraints into the ones constraining the boundaries and the ones constraining the distances between adjacent nodes. Then we can write

$$f(\mathbf{x}) = -\sum_i z_i + \rho \left(\sum_{i \sim j} ((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 - 1)^2 + \sum_{k=1}^{4} (x_{c_k} - p_{x_{c_k}})^2 + (y_{c_k} - p_{y_{c_k}})^2 + (z_{c_k} - p_{z_{c_k}})^2 \right)$$

where

$$p_{x_{c_2}} = p_{y_{c_3}} = p_{x_{c_4}} = p_{y_{c_4}} = \beta\sqrt{N}$$

and all the other subscripted p's are zero.

Implementing this function in Matlab is tricky. We want to avoid expensive Matlab *for*-loops since we will make many many calls to this function. Matlab is optimized for matrix computations, so we want to express the objective function as a matrix operation. To do this, we create an incidence matrix $\tilde{I}$ which has dimension $2(N - \sqrt{N}) \times N$; each row in $\tilde{I}$ represents an edge in the graph of the net. To form this matrix, iterate through the edges in the graph of the net in some specified order. For each edge k, put a 1 at $\tilde{I}(k,i)$ where i is the node number of one endpoint of edge k, and put a -1 at $\tilde{I}(k,j)$ where j is the other enpoint of edge k. Now if we reshape $\mathbf{x}$ into the form

$$X = \begin{bmatrix} x_1 & y_1 & z_1 \\ & \vdots & \\ x_N & y_N & z_N \end{bmatrix}$$

then we can multply $\tilde{I}X$ to get

$$\tilde{I}X = \begin{bmatrix} & \vdots & \\ x_i - x_j & y_i - y_j & z_i - z_j \\ & \vdots & \end{bmatrix}$$

where each row precisely subtracts the position components of adjacent nodes in the graph of the net. If we then square each element of $\tilde{I}X$, take the row sums, and subtract a $2(N-\sqrt{N})$-vector of ones, then we have the distance constraints of the $c(\mathbf{x})$. The position constraints on the corners of the net are hard coded with parameter β. Suppose they are coded in a function $\mathbf{d}(\mathbf{x},\beta)$. Then with X and $\tilde{I}$ defined as above, we can write the objective function in Matlab as

```
F = @(x)sum(x(3:3:3*N)) + rho*(...
    (sum((I*reshape(x,3,N)').^2,2)-ones(2*(N-sqrt(N)),1))'*...
    (sum((I*reshape(x,3,N)').^2,2)-ones(2*(N-sqrt(N)),1)) +...
    d(x,beta)'*d(x,beta);
```

This is essentially the function I coded for my objective function.

3.4 Gradient Function

Taking the gradient of the objective function with respect to each position component of each of the nodes, we have

$$\nabla_{x_i} f = 0 + 4\rho\sum_{j\sim i}((x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2-1)(x_i-x_j)+2\rho\gamma_{x_i}$$

$$\nabla_{y_i} f = 0 + 4\rho\sum_{j\sim i}((x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2-1)(y_i-y_j)+2\rho\gamma_{y_i}$$

$$\nabla_{z_i} f = 1 + 4\rho\sum_{j\sim i}((x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2-1)(z_i-z_j)+2\rho\gamma_{z_i}$$

where $j\sim i$ represents the sum over the edges connected to node i and

$$\gamma_{x_i} = \begin{cases} x_i & \text{if node } i \text{ is at corner 1} \\ x_i-\beta\sqrt{N} & \text{if node } i \text{ is at corner 2} \\ x_i & \text{if node } i \text{ is at corner 3} \\ x_i-\beta\sqrt{N} & \text{if node } i \text{ is at corner 4} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{y_i} = \begin{cases} y_i & \text{if node } i \text{ is at corner 1} \\ y_i & \text{if node } i \text{ is at corner 2} \\ y_i-\beta\sqrt{N} & \text{if node } i \text{ is at corner 3} \\ y_i-\beta\sqrt{N} & \text{if node } i \text{ is at corner 4} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{z_i} = \begin{cases} z_i & \text{if node } i \text{ is at corner 1} \\ z_i & \text{if node } i \text{ is at corner 2} \\ z_i & \text{if node } i \text{ is at corner 3} \\ z_i & \text{if node } i \text{ is at corner 4} \\ 0 & \text{otherwise} \end{cases}$$

This was an exercise in calculus. However, we face the same problem as with the objective function of wanting to avoid slow Matlab *for*-loops. We want to express this gradient function as a matrix operation so Matlab can optimize its performance. Consider the following rearrangement of ∇f_{x_i}:

$$\nabla_{x_i} f = 0 + 4\rho\sum_{j\sim i}((x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2)(x_i-x_j) - 4\rho\sum_{j\sim i}(x_i-x_j)+2\rho\gamma_{x_i}$$

For the analysis, we focus on the term $\sum_{j\sim i}(x_i-x_j)$. Assuming i represents an interior node with four neighbors this sum can be expanded to

$$\sum_{j\sim i}(x_i-x_j) = 4x_i - x_{n_1} - x_{n_2} - x_{n_3} - x_{n_4}$$

where n_k denotes the neighbors of node i. The Laplacian of the graph of the net is a matrix L with

$$L(i,j) = \begin{cases} d_i & \text{if } i = j \\ -1 & \text{if } i \text{ is connected to } j \\ 0 & \text{otherwise} \end{cases}$$

For interior node i, d_i (the degree of node i) is 4. So if we multiply the first column of X into row i of L, we have

$$L(i,\cdot)X(\cdot,1) \;=\; 4x_i - x_{n_1} - x_{n_2} - x_{n_3} - x_{n_4} \;=\; \sum_{j\sim i}(x_i - x_j)$$

This suggests we can reshape the matrix LX into a vector with the precise components we want for our gradient vector! Since the Laplacian contains the degrees of the nodes on the boundaries, this works for the boundary nodes as well. Fortunately, the Laplacian of the graph of the net can also be defined as $\tilde{I}^T\tilde{I}$ so we can compute

$$LX = \tilde{I}^T\tilde{I}X$$

Next we focus on the other sum in our expression for the gradient, namely

$$\sum_{j\sim i}((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)(x_i - x_j)$$

Note that this is essentially the same as the computation above but scaled by a factor $(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2$. And we already have a spiffy way to compute this quantity! Therefore we create a diagonal matrix D with

$$D_{n(i,j)} = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2$$

where $n(i,j)$ maps the node numbers to an edge number in a way consistent with $\tilde{I}$. Then we can compute

$$\sum_{j\sim i}((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)(x_i - x_j) = (\tilde{I}^T D\tilde{I}X)(i)$$

where the notation $(\tilde{I}^T D\tilde{I}X)(i)$ means the ith component of the vector $\tilde{I}^T D\tilde{I}X$. Therefore we have an efficient way to compute the gradient vector of the objective function sans *for*-loops. Suppose we have hardcoded γ_{x_i}, γ_{y_i}, and γ_{z_i} into a function $\mathbf{d}_{\text{grad}}(\mathbf{x},\beta)$. Then in Matlab it will look like

```
gradF = @(x)reshape([zeros(N,2) ones(N,1)]',3*N,1) + ...
    gamma*4*reshape((I'*diag(sum((I*reshape(x,3,N)').^2,2))*I*reshape(x,3,N)')',3*N,1) - ...
    gamma*4*reshape((I'*I*reshape(x,3,N)')',3*N,1) + ...
    2*gamma*d_grad(x,beta);
```

This is essentially the function I implemented in Matlab.

To test that I had derived and coded my ∇f properly, I compared it with a finite difference approximation of the gradient

$$\nabla^{FD}_{x_i} f(\mathbf{x}) = \frac{f(\mathbf{x} + \epsilon e_{x_i}) - f(\mathbf{x})}{\epsilon}$$

where I chose $\epsilon = \sqrt{\text{eps}}$. See results below for the comparison.

3.5 Initial Condition

To get the initial point $\mathbf{x}_0$, I solved the 2-D Poisson's equation with a constant force function on a square domain $\mathcal{D} = [0, \beta\sqrt{N}] \times [0, \beta\sqrt{N}]$ given by

$$\begin{aligned} u_{xx} + u_{yy} &= -1 \quad \text{on } \mathcal{D} \\ u &= 0 \quad \text{on } \partial\mathcal{D} \end{aligned}$$

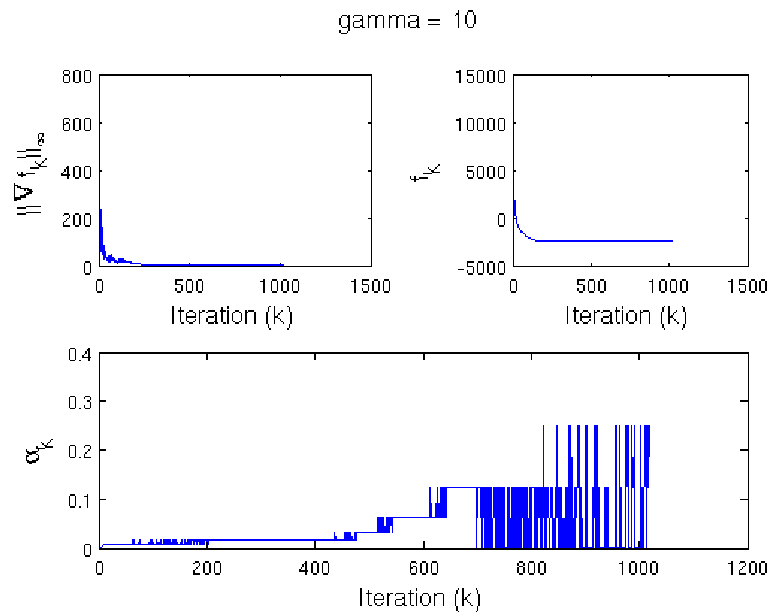
This is the standard engineering model for a membrane, which, heuristically, is the continuous analog of the discrete net problem. I solved this with standard central difference approximations to the second derivatives.

The initial x and y components are then the x and y components of the mesh generated to solve the Poisson equation, and the initial z components are the nummerical solution to the finite difference problem.

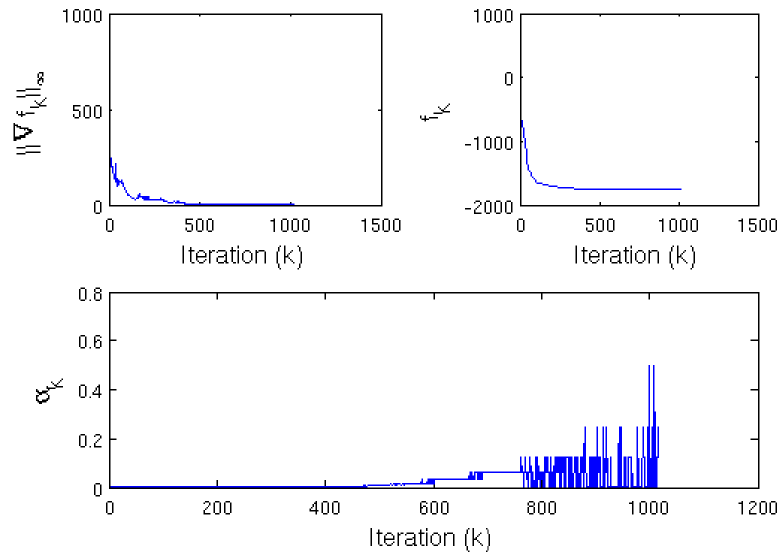
I do not think my relatively flashy choice of $\mathbf{x}_0$ improved my method much. It would have done just as well with the x's and y's initialized to some grid and the z's intialized to zero. In fact, this zero choice would have made much cooler movies.

4 Results

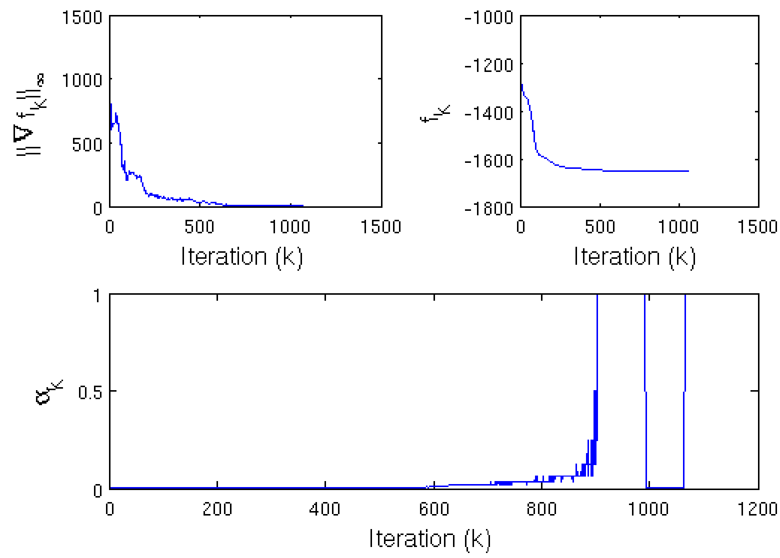
I ran my optimization routine on a net with $17^2 = 289$ nodes and $\beta = 0.7$. I let this run overnight. The wrapper script for my BFGS routine varied my penalty parameter $\rho_k = 10^k$ for $k = 1, 2, 3, 4, 5$. I started with $\mathbf{x}_0$ as described above with $\rho = 10$. Once BFGS converged with those inputs, I used the converged $\mathbf{x}^*_{\rho=10}$ as $\mathbf{x}_0$ for a subsequent BFGS run with $\rho = 100$. I repeated this process until I ran through all of my ρ's. Below are charts for each ρ run. Each chart plots $f(\mathbf{x})$, $\|\nabla f([\mathbf{x})\|_\infty$, and α versus the iteration number.



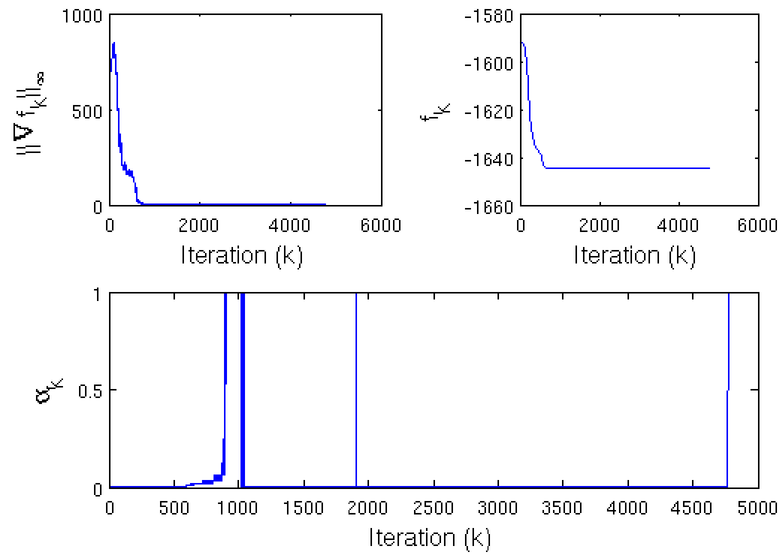
$f(\mathbf{x}^*)$	$\|\nabla f(\mathbf{x}^*)\|_\infty$	$\|\nabla f(\mathbf{x}^*) - \nabla^{FD} f(\mathbf{x}^*)\|_\infty$	Converged?	Iterations	Missed Wolfe
-2471.49	9.80316e-06	2.86559e-05	Yes	1018	181



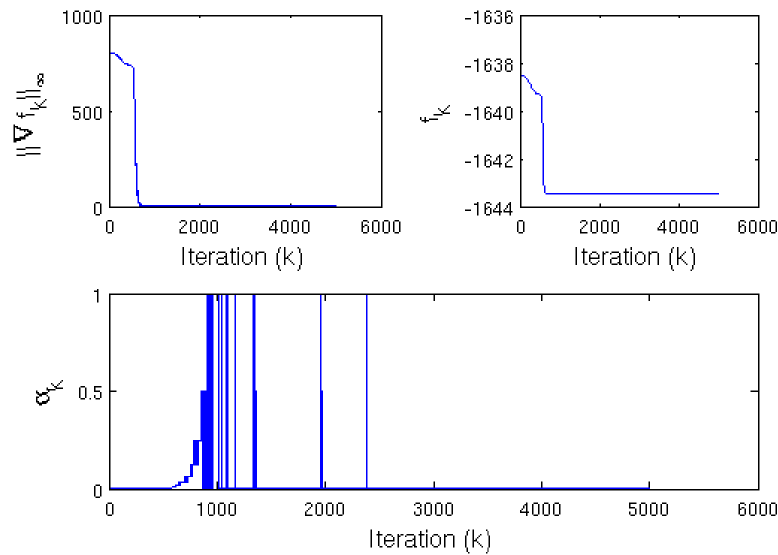
$f(\mathbf{x}^*)$	$\|\nabla f(\mathbf{x}^*)\|_\infty$	$\|\nabla f(\mathbf{x}^*) - \nabla^{FD} f(\mathbf{x}^*)\|_\infty$	Converged?	Iterations	Missed Wolfe
-1751.2	9.7357e-06	2.77673e-05	Yes	1016	132



$f(\mathbf{x}^*)$	$\|\nabla f(\mathbf{x}^*)\|_\infty$	$\|\nabla f(\mathbf{x}^*) - \nabla^{FD} f(\mathbf{x}^*)\|_\infty$	Converged?	Iterations	Missed Wolfe
-1655.61	9.15207e-06	0.000192707	Yes	1065	72



$f(\mathbf{x}^*)$	$\|\nabla f(\mathbf{x}^*)\|_\infty$	$\|\nabla f(\mathbf{x}^*) - \nabla^{FD} f(\mathbf{x}^*)\|_\infty$	Converged?	Iterations	Missed Wolfe
-1644.58	9.5493e-06	0.00181318	Yes	4770	3740



$f(\mathbf{x}^*)$	$\|\nabla f(\mathbf{x}^*)\|_\infty$	$\|\nabla f(\mathbf{x}^*) - \nabla^{FD} f(\mathbf{x}^*)\|_\infty$	Converged?	Iterations	Missed Wolfe
-1643.46	1.81895e-05	0.0181385	No	5001	4005

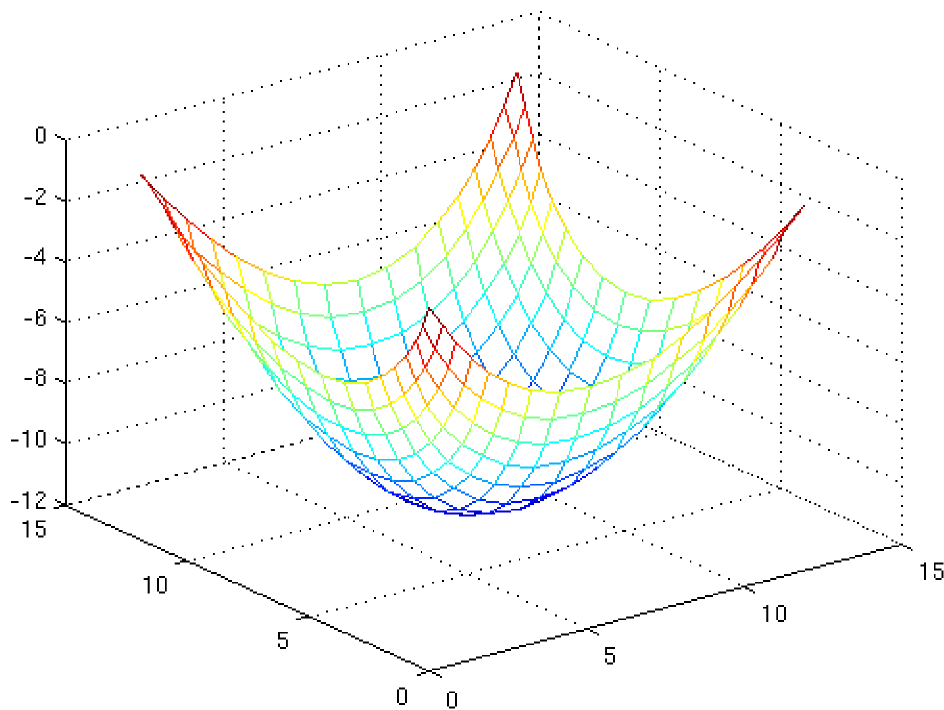
I have a number of observations on this series of plots:

- My last run with $\rho = 100,000$ did not actually converge. My BFGS gives up after 5,000 iterations, and it gave up this time. But as is apparent from the plot of the norm of the gradient, it goes very quickly towards zero.
- The "surfaces" obviously become much "flatter" as ρ increases. This is apparent from the trend in the

Missed Wolfe statistic, which counts the number of iterations in which the routine took a step of size ϵ without updating the inverse Hessian.

- In the smaller ρ's I am encouraged to see the α's increase as the iteration number increases. This implies that the routine is moving toward a Newton region where the natural steplength of 1 satisfies the Wolfe conditions.
- In each of my plots, the decreasing trends in both $f(\mathbf{x}^*)$ and $\|\nabla f(\mathbf{x}^*)\|_\infty$ are very favorable.
- The numerial approximation to the gradient differs significantly from the analytical gradient as ρ increases.
- In the plots for the last two ρ values, the routine seems to find a Newton region around a thousand iterations. Unfortunately it does not converge during these iterations and it just piddles around near the minimum.
- I am encouraged to see that the converged function value is very close between the last two values of ρ. Hopefully this suggests that the penalty method is converging to the correct minimum as $\rho \to \infty$.

We are finally ready for for a visual confirmation of the hanging net. We have examined the theory, derived the model, validated the code, and verified that the numbers are moving in the right direction - downward. So let's see that net hang!



Beautiful. Isn't she?

5 Conclusions

There are many different directions that this project could have gone. I was very interested in finding a PDE that could model this discrete net. I spoke with Professor Peter Pinsky on two occasions trying to derive

such a model, but we did not come to any conclusions. I had hoped to formulate the problem as a physically continuous problem, solve it with the finite element method, and compare the solution to the result of the optimization routine. During our first discussion, Professor Pinsky seemed optimistic since one property of the finite element solution is that it gives the solution with the minimum potential energy of the system. In the end, our discussions inspired me to solve a Poisson equation - the standard model for a membrane - to determine $\mathbf{x}_0$.

I could have used the modified Cholesky factors of an approximation to the Hessian - as opposed to the inverse of the Hessian - and compared these two methods on a number of poorly conditioned problems. An example of a poorly conditioned problem would be one where I had chosen β either very close to 1 or very close to 0.

I could have pursued a number of different methods that may have been more efficient than the method I chose, such as a limited memory BFGS. I believe this would have allowed me to optimize much larger systems in much shorter time periods.

I could have done a thorough study of the choice of parameters for my optimization routine. As it is, I chose conservative parameters and used them throughout my simulation. I could have been much more aggressive with my steplength calculation, including more aggresive c_1 and c_2 parameters and larger initial α's. I also could have compared the performance for different zoom methods, such as cubic or quadratic interpolation or a safeguarded Newton's method.

All of these studies still sound interesting, but alas, there is only so much time; there are lots of other responsibilities and intellectual pursuits to keep me occupied and stimulated. In the end I have found this project extremely useful in solidifying the ideas presented in the course. This project is an indispensable part of the coursework, and I am glad I chose to finish it on time.

References

[1] Nocedal, J., and Wright, S. J., *Numerical Optimization*, Springer-Verlag, New York, 1999.

[2] Gurwitz, C., Klein, L., and Lamba, M., *The More, Garbow and Hillstrom collection of test functions written for Matlab.* Contact gurwitz@sci.brooklyn.cuny.edu for comments.