Solving the Inverse Problem through Optimization Methods

Melanie Ferreri and Christine Wolf

August 12, 2016

Introduction

The inverse problem on electrical networks involves taking a graph and the response matrix produced by that graph and recovering the conductances of the edges. This problem has solutions for certain families of graphs, but they tend not to work for any arbitrary connected graph. Thus, we created a computer program that would allow a graph and response matrix to be fed in and produce the corresponding conductances, which is in essence solving the inverse problem.

Suppose we are given a known graph of a network G and an experimental response matrix Λ, which may or may not contain some error. Then we attempt to recover the original conductances of the network. We do this by using various optimization methods, such as gradient descent, the BFGS method, and the Gauss-Newton method, to minimize the two-norm of the difference between the experimental Λ and a valid response matrix generated by a guess of the true conductance vector. We also applied the program to graphs with different conductances that map to the same response matrix in an attempt to recover all solution sets.

Contents

1 Definitions and Notation

Graph of the Network
We denote a graph G by the notation $G : (\partial V, \text{ int } V)$, where ∂V represents the boundary vertices of a graph (denoted visually with solid dots) and int V represents the interior vertices (denoted visually with outlined dots). If ∂V consists of m vertices and int V consists of n vertices, then we order the vertices such that v: $\{\ 1 \dots m, m+1 \dots m+n\ \}$. Aside from this condition, the ordering of the vertices is arbitrary.

The Kirchhoff Matrix
Suppose we are given a graph that represents some electrical network, where the weights on each edge correspond to conductances on the network. We can format this information in matrix form, which we call the Kirchhoff Matrix, K. The entries of K are given by

$$K_{ij} = \begin{cases} -\gamma_{ij} & \text{for } i\eta j \\ \sum_{i\eta j}^{n} \gamma_{ij} & \text{for } i = j \\ 0 & \text{otherwise} \end{cases}$$

We say $i\eta j$ if there is an edge between i and j.

The Response Matrix
Due to the symmetry of the matrix K, we can partition K into the following four blocks:

$$K = \left[\begin{array}{c|c} A & B \\ \hline B^T & C \end{array}\right]$$

where if we let p represent the number of boundary vertices and q represent the number of interior vertices, A is $p \times p$, B is $p \times q$, and C is $q \times q$. Taking the Schur Complement of this matrix, we find the response matrix, which we denote as Λ, by the following equation: (see (1) for further explanation of the Schur Complement)

$$\Lambda = A - BC^{-1}B^T$$

***L*-Lipschitz and *β*-smooth**
A function f is L-Lipschitz if

$$\forall x, y\ |f(x) - f(y)| \leq L\|x - y\| \text{ for some } L$$

A function f is β-smooth if

$$\forall x, y\ \|\nabla f(x) - \nabla f(y)\| \leq \beta\|x - y\| \text{ for some } \beta$$

2 The Inverse Problem

If the forward problem is generating the response matrix Λ from a given graph with known conductances, then we define the inverse problem as finding unknown conductances from a given Λ. The main problem we will be discussing in this paper is a variation of the inverse problem on electrical networks. For this problem, we take a graph of vertices specified as boundary or interior, with some unknown conductances on its edges. If we set the voltages on the boundary vertices, it generates a response, which we can encode in a response matrix based on the indexing of the vertices. For the inverse problem we want to recover the conductances of the edges based on this response matrix. To see a more detailed and comprehensive explanation of the inverse problem, see (1).

Now suppose we have a graph and a response matrix Λ_0 that corresponds to the graph but may or may not be a mathematically correct response matrix. For example, if Λ_0 is measured experimentally, the values will have some experimental error. We want to find the vector $\boldsymbol{\gamma}$ containing the conductance of each edge in the graph such that $\Lambda(\boldsymbol{\gamma})$ is as close as possible to Λ_0. We chose to do this by finding the result of the equation:

$$\min_{\boldsymbol{\gamma}\geq 0} \frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$$

with the norm being the Frobenius norm. That is, we found the $\boldsymbol{\gamma}$ that minimized the square of the 2 norm of the difference between the given and calculated response matrices. Because $\boldsymbol{\gamma}$ directly produces the Kirchhoff matrix, and the response matrix can be calculated from the Kirchhoff matrix by taking its Schur complement, testing this equation with a given value of $\boldsymbol{\gamma}$ is fairly simple to do. However, the complex process of recovering $\boldsymbol{\gamma}$ when given only $\Lambda(\boldsymbol{\gamma})$ makes the idea of finding the gradient of the function, setting it equal to zero, and solving for $\boldsymbol{\gamma}$ impractical at best. Therefore, we tried using optimization methods instead to find the solution to the equation. These methods generally utilize an iterative checking of the result of the equation when plugging in different guesses, which are then strategically updated. This takes advantage of the ease of computing the result of the equation when given a guess, and thus proved effective in solving our problem.

3 Optimization Methods

We experimented with several different methods of finding the solution to the inverse problem through finding the $\boldsymbol{\gamma}$ that solves the equation

$$\min_{\boldsymbol{\gamma}\geq 0} \frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$$

All of these methods require that we find the gradient of the function, so we found an efficient method to do this.

3.1 Computing the Gradient of $\frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$

Let

$$f(\boldsymbol{\gamma}) = \frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$$

We will compute the gradient of $f(\boldsymbol{\gamma})$. By the chain rule, to get the k^{th} component of the gradient vector for a function $g \circ h(\boldsymbol{\gamma})$ we have

$$\frac{\partial g \circ h}{\partial \gamma_k} = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial g}{\partial y_{ij}} \cdot \frac{\partial h_{ij}}{\partial \gamma_k}$$

where n is the dimension of the matrices and $y = h(\boldsymbol{\gamma})$, for any functions g and h. If we take the functions to be $h(\boldsymbol{\gamma}) = \Lambda_0 - \Lambda(\boldsymbol{\gamma})$ and $g(y) = \frac{1}{2}\|y\|^2 = \frac{1}{2}\sum_{i,j} y_{ij}^2$, then

$$g \circ h(\boldsymbol{\gamma}) = \frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2 = f(\boldsymbol{\gamma})$$

Thus, to take the gradient of $f(\boldsymbol{\gamma})$ we can apply the chain rule formula. We have that to find the partial derivative of g with respect to y_{ij}, we can say

$$\frac{\partial g}{\partial y_{ij}} = \frac{\partial}{\partial y_{ij}}(\frac{1}{2}\sum_{p,q} y_{pq}^2) = \frac{\partial}{\partial y_{ij}}(\frac{1}{2}y_{ij}^2) = y_{ij}$$

since all the other components of the matrix y, the y_{pq}'s where $p \neq i$ or $q \neq j$, are constant with respect to y_{ij}. Since $y = h(\boldsymbol{\gamma})$, we have

$$\frac{\partial g}{\partial y_{ij}} = y_{ij} = h(\boldsymbol{\gamma})_{ij} = (\Lambda_0 - \Lambda(\boldsymbol{\gamma}))_{ij}$$

To finish finding the gradient, we then needed to find a way to compute $\frac{\partial h_{ij}}{\partial \boldsymbol{\gamma}_k}$. We have

$$\frac{\partial h_{ij}}{\partial \boldsymbol{\gamma}_k} = \frac{\partial}{\partial \boldsymbol{\gamma}_k}((\Lambda_0 - \Lambda(\boldsymbol{\gamma}))_{ij}) = -\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma})_{ij}$$

Thus, we computed the matrix $\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma})$. In our first attempt to do this, we took advantage of the result in Dr. Morrow's paper "Derivative of Λ" that for any response matrix Λ, the directional derivatives can be computed through the following formula:

$$D_\epsilon \Lambda = \epsilon_A - \epsilon_B C^{-1} B^T - BC^{-1}\epsilon_B^T + BC^{-1}\epsilon_C C^{-1} B^T$$

where B and C are the block components of the Kirchhoff matrix corresponding to Λ, and ϵ represents the direction of the derivative. Because we wanted the gradient, we constructed ϵ to be in the unit directions. While this gave us the correct matrix, because of the volume of computations that gradient descent required us to perform we wanted a less computationally intense method in order to optimize our algorithm. Thus, we used another formula developed by Dr. Morrow to give us the same result with less computations.

Theorem 3.1. If we have a Kirchhoff matrix

$$K = \begin{bmatrix} A & B \\ B^T & C \end{bmatrix}$$

and we let

$$D = \begin{bmatrix} I \\ -C^{-1}B^T \end{bmatrix}$$

then the response matrix Λ corresponding to the Kirchhoff matrix can be given by

$$\Lambda = D^T K D$$

and

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma}) = D^T \left[\frac{\partial K}{\partial \boldsymbol{\gamma}_k}\right] D$$

Proof. By the definition of the Kirchhoff matrix, we can write

$$K = \begin{bmatrix} A & B \\ B^T & C \end{bmatrix}$$

Since we have

$$D = \begin{bmatrix} I \\ -C^{-1}B^T \end{bmatrix}$$

$$D^T = \begin{bmatrix} I & -BC^{-1} \end{bmatrix}$$

then we have that

$$D^T K D = \begin{bmatrix} I & -BC^{-1} \end{bmatrix} \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \begin{bmatrix} I \\ -C^{-1}B^T \end{bmatrix} = \begin{bmatrix} A - BC^{-1}B^T \end{bmatrix} = \Lambda$$

since $A - BC^{-1}B^T$ is exactly the Schur complement of K.
Now we will show that

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma}) = D^T \left[\tfrac{\partial K}{\partial \boldsymbol{\gamma}_k}\right] D$$

By the product rule, we have that

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma}) = \frac{\partial}{\partial \boldsymbol{\gamma}_k}(D^TKD) = \Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}D^T\Big)KD + D^T\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}K\Big)D + D^TK\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}D\Big)$$

We have that

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}D = \frac{\partial}{\partial \boldsymbol{\gamma}_k}\begin{bmatrix} I \\ -C^{-1}B^T\end{bmatrix} = \begin{bmatrix} 0 \\ \frac{\partial}{\partial \boldsymbol{\gamma}_k}(-C^{-1}B^T)\end{bmatrix}$$

and

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}D^T = \frac{\partial}{\partial \boldsymbol{\gamma}_k}\begin{bmatrix} I & -BC^{-1}\end{bmatrix} = \begin{bmatrix} 0 & \frac{\partial}{\partial \boldsymbol{\gamma}_k}(-BC^{-1})\end{bmatrix}$$

Thus, since

$$KD = \begin{bmatrix} A & B \\ B^T & C\end{bmatrix}\begin{bmatrix} I \\ -C^{-1}B^T\end{bmatrix} = \begin{bmatrix} A - BC^{-1}B^T \\ 0\end{bmatrix} = \begin{bmatrix} \Lambda \\ 0\end{bmatrix}$$

and

$$D^TK = \begin{bmatrix} I & -BC^{-1}\end{bmatrix}\begin{bmatrix} A & B \\ B^T & C\end{bmatrix} = \begin{bmatrix} A - BC^{-1}B^T & 0\end{bmatrix} = \begin{bmatrix} \Lambda & 0\end{bmatrix}$$

by the definition of Λ, we have

$$\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}D^T\Big)KD = \begin{bmatrix} 0 & \frac{\partial}{\partial \boldsymbol{\gamma}_k}(-BC^{-1})\end{bmatrix}\begin{bmatrix} \Lambda \\ 0\end{bmatrix} = 0$$

and

$$D^TK\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}D\Big) = \begin{bmatrix} \Lambda & 0\end{bmatrix}\begin{bmatrix} 0 \\ \frac{\partial}{\partial \boldsymbol{\gamma}_k}(-C^{-1}B^T)\end{bmatrix} = 0$$

Therefore, we have that

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma}) = \Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}D^T\Big)KD + D^T\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}K\Big)D + D^TK\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}D\Big) = 0 + D^T\Big(\frac{\partial}{\partial \boldsymbol{\gamma}_k}K\Big)D + 0 = D^T\left[\tfrac{\partial K}{\partial \boldsymbol{\gamma}_k}\right]D$$

Thus,

$$\frac{\partial}{\partial \boldsymbol{\gamma}_k}\Lambda(\boldsymbol{\gamma}) = D^T \left[\tfrac{\partial K}{\partial \boldsymbol{\gamma}_k}\right] D$$

as desired. □

Finding $\left[\frac{\partial K}{\partial \boldsymbol{\gamma}_k}\right]$ is actually fairly easy. Because the vector $\boldsymbol{\gamma}$ simply contains all the non-zero γ_{ij}'s in an arbitrarily imposed ordering, $\boldsymbol{\gamma}_k$ represents a particular γ_{ij}. This γ will by the definition of the Kirchhoff matrix appear in exactly the spots K_{ij} and K_{ji} as $-\gamma_{ij}$, as well as along the diagonal in K_{ii} and K_{jj} as part of the positive sum of the conductances appearing in the corresponding row. Thus, because all other terms are constant with respect to $\boldsymbol{\gamma}_k$, we have that $\left[\frac{\partial K}{\partial \boldsymbol{\gamma}_k}\right]$ is a matrix of mostly zeros, but with -1's at K_{ij} and K_{ji} and 1's at K_{ii} and K_{jj}, which is simple to create.

Putting together all of this information gives us that the gradient of the function

$$f(\boldsymbol{\gamma}) = \frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$$

is the vector where the k^{th} component is given by

$$\frac{\partial f}{\partial \boldsymbol{\gamma}_k} = \sum_{i=1}^{n}\sum_{j=1}^{n}(\Lambda_0 - \Lambda(\boldsymbol{\gamma}))_{ij} \cdot \Big(- \Big(D^T \left[\tfrac{\partial K}{\partial \boldsymbol{\gamma}_k} \right] D \Big)_{ij} \Big)$$

We can compute each of the matrices present in the sum fairly simply with knowledge of $\boldsymbol{\gamma}$ and take their outer product in order to get the gradient that we want.

3.2 Gradient Descent

We first approached this problem using the optimization method of gradient descent. This method uses the recursive formula

$$x_k = x_{k-1} - t_k \nabla f(x_{k-1})$$

to choose the next guess, where x_k is the k^{th} guess, f is the function to minimize, and t_k is some constant factor that can be incremented as needed. In our case, we have

$$f(\boldsymbol{\gamma}) = \frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$$

In order to utilize this method, we evaluated the gradient of $f(\boldsymbol{\gamma})$, which can compute using the formula described above.

In order to quantify the efficiency of the gradient descent algorithm, we sought to prove that the number of iterations of the method required to show that $\|\nabla f(x_k)\| < \epsilon$ has an upper bound. To do so, we proved the following theorem.

Theorem 3.2. Suppose f is β-smooth, and x is incremented by gradient descent such that

$$x_k = x_{k-1} - \frac{1}{\beta}\nabla f(x_{k-1})$$

Let f^* be the minimum value of f. Then $\min\limits_{k=1\ldots n} \|\nabla f(x_k)\|^2 \leq \frac{2\beta}{n}(f(x_0) - f^*)$.

Proof. By the definition of gradient descent, we have

$$x_k = x_{k-1} - \frac{1}{\beta}\nabla f(x_{k-1})$$

Because f is β smooth, by the definition of β smooth we have

$$\forall y, f(y) \leq f(\bar{x}) + \langle \nabla f(\bar{x}), y - \bar{x}\rangle + \frac{\beta}{2}\|y - \bar{x}\|^2$$

We can let $\bar{x} = x_{k-1}$, and since this is true for all y, we can let $y = x_k$. Substituting these into the formula gives

$$f(x_k) \leq f(x_{k-1}) + \langle \nabla f(x_{k-1}), x_k - x_{k-1}\rangle + \frac{\beta}{2}\|x_k - x_{k-1}\|^2$$

Because $x_k = x_{k-1} - \frac{1}{\beta}\nabla f(x_{k-1})$, $x_k - x_{k-1} = -\frac{1}{\beta}\nabla f(x_{k-1})$, so the formula can be rewritten as

$$f(x_k) \leq f(x_{k-1}) + \langle \nabla f(x_{k-1}), -\frac{1}{\beta}\nabla f(x_{k-1})\rangle + \frac{\beta}{2}\| - \frac{1}{\beta}\nabla f(x_{k-1})\|^2$$

which can be simplified to

$$f(x_k) \leq f(x_{k-1}) - \frac{1}{\beta}\|\nabla f(x_{k-1})\|^2 + \frac{\beta}{2}\cdot\frac{1}{\beta^2}\|\nabla f(x_{k-1})\|^2$$

$$f(x_k) \leq f(x_{k-1}) - \frac{1}{\beta}\|\nabla f(x_{k-1})\|^2 + \frac{1}{2\beta}\|\nabla f(x_{k-1})\|^2$$

$$f(x_k) \leq f(x_{k-1}) - \frac{1}{2\beta}\|\nabla f(x_{k-1})\|^2$$

Thus, we have

$$\frac{1}{2\beta}\|\nabla f(x_{k-1})\|^2 \leq f(x_{k-1}) - f(x_k)$$

If we sum both sides up from $k = 1$ to $k = n$, all the middle terms on the right hand side cancel, leaving us with

$$\frac{1}{2\beta}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq \sum_{k=1}^{n} f(x_{k-1}) - f(x_k)$$

$$\frac{1}{2\beta}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq f(x_0) - f(x_n)$$

If we let f^* be the optimal value of $\min_x f(x)$, we can say that

$$\frac{1}{2\beta}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq f(x_0) - f(x_n) \leq f(x_0) - f^*$$

Dividing both sides by n gives us the average value of $\|\nabla f(x_{k-1})\|^2$ on left side, which is clearly greater than or equal to the minimum value of $\|\nabla f(x_{k-1})\|^2$, so we have

$$\frac{1}{2\beta}\min_k \|\nabla f(x_k)\|^2 \leq \frac{1}{n\cdot 2\beta}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq \frac{1}{n}(f(x_0) - f^*)$$

Rearranging the equation gives us

$$\min_k \|\nabla f(x_k)\|^2 \leq \frac{2\beta}{n}(f(x_0) - f^*)$$

Thus, the function is bounded above, as desired. □

We can also show that the function converges if we increment t_k using the following rules – we initialize t_k to some $\hat{t} > 0$, then repeat $t_k := bt_k$ until

$$f(x_k) < f(x_{k-1}) - at_k\|\nabla f(x_{k-1})\|^2$$

with $0 < a \leq \frac{1}{2}$ and $0 < b < 1$. This is shown in the following theorem.

Theorem 3.3. Suppose f is β-smooth, and x is incremented by gradient descent such that

$$x_k = x_{k-1} - t_k\nabla f(x_{k-1})$$

with t_k as described above. Let f^* be the minimum value of f. Then

$$\min_{k=1\ldots n}\|\nabla f(x_k)\|^2 \leq \frac{2\beta}{n}(f(x_0) - f^*)$$

Proof. By the definition of gradient descent, we have

$$x_k = x_{k-1} - t_k\nabla f(x_{k-1})$$

Because f is β smooth, by the definition of β smooth we have

$$\forall y, f(y) \leq f(\bar{x}) + \langle \nabla f(\bar{x}), y - \bar{x} \rangle + \frac{\beta}{2}\|y - \bar{x}\|^2$$

We can let $\bar{x}$ be x_{k-1}, and since this is true for all y, we can let $y = x_k$. Substituting these into the formula gives

$$f(x_k) \leq f(x_{k-1}) + \langle \nabla f(x_{k-1}), x_k - x_{k-1} \rangle + \frac{\beta}{2}\|x_k - x_{k-1}\|^2$$

Because $x_k = x_{k-1} - t_k \nabla f(x_{k-1})$, $x_k - x_{k-1} = -t_k \nabla f(x_{k-1})$, so the formula can be rewritten as

$$f(x_k) \leq f(x_{k-1}) + \langle \nabla f(x_{k-1}), -t_k \nabla f(x_{k-1}) \rangle + \frac{\beta}{2}\| - t_k \nabla f(x_{k-1})\|^2$$

which can be simplified to

$$f(x_k) \leq f(x_{k-1}) - t_k\|\nabla f(x_{k-1})\|^2 + \frac{\beta}{2} \cdot t_k^2 \|\nabla f(x_{k-1})\|^2$$

$$f(x_k) \leq f(x_{k-1}) - t_k(1 - \frac{\beta}{2} \cdot t_k)\|\nabla f(x_{k-1})\|^2$$

In order to place a bound on this function, we must find a lower bound t_{min} for t_k. We have that if $t_k < \frac{1}{\beta}$, then

$$f(x_k) \leq f(x_{k-1}) - t_k(1 - \frac{\beta}{2} \cdot t_k)\|\nabla f(x_{k-1})\|^2 < f(x_{k-1}) - t_k(1 - \frac{\beta}{2} \cdot \frac{1}{\beta})\|\nabla f(x_{k-1})\|^2 = f(x_{k-1}) - \frac{t_k}{2}\|\nabla f(x_{k-1})\|^2$$

so

$$f(x_k) < f(x_{k-1}) - \frac{t_k}{2}\|\nabla f(x_{k-1})\|^2$$

We know by the Lipschitz condition that

$$f(x_k) \geq f(x_{k-1}) - \frac{1}{\beta}\|\nabla f(x_{k-1})\|^2$$

Since we have $0 < a \leq \frac{1}{2}$ we know that since $t_k := bt_k$ until

$$f(x_k) < f(x_{k-1}) - at_k\|\nabla f(x_{k-1})\|^2$$

we have $t_k = b^i \hat{t}$ for some positive integer i, so since $0 < b < 1$, t_k is decreasing until the condition is satisfied. Thus, we can show that $\forall k$, $t_k \geq t_{min} = \min\{\hat{t}, \frac{b}{\beta}\}$. Suppose $\hat{t} \leq \frac{b}{\beta} < \frac{1}{\beta}$. Then

$$f(x_k) < f(x_{k-1}) - \frac{\hat{t}}{2}\|\nabla f(x_{k-1})\|^2 < f(x_{k-1}) - at_k\|\nabla f(x_{k-1})\|^2$$

since $a \leq \frac{1}{2}$, so $t_k = \hat{t}$ for all k, so we have $t_{min} = \hat{t}$. Suppose $\hat{t} > \frac{b}{\beta}$. If $\hat{t}$ satisfies the equation

$$f(x_k) < f(x_{k-1}) - \frac{\hat{t}}{2}\|\nabla f(x_{k-1})\|^2$$

then the same argument as above holds. Suppose the equation does not hold. Then we must have $\hat{t} \geq \frac{1}{\beta}$, so $b\hat{t} \geq \frac{b}{\beta}$. If the equation still doesn't hold, we have $b\hat{t} \geq \frac{1}{\beta}$, so $b^2\hat{t} \geq \frac{b}{\beta}$. Because $b < 1$ we can continue this argument until we get that the equation is satisfied for some b^i, but $b^i\hat{t} \geq \frac{b}{\beta}$, and $b^{i+j} = b^i$ for all j. Thus, since t is decreasing we get $t_k \geq \frac{b}{\beta}$ for all k. Therefore, we have $t_k \geq t_{min} = \min\{\hat{t}, \frac{b}{\beta}\}$, so $0 < t_{min} \leq \frac{b}{\beta}$.

Because we know $0 < t_{min} \leq \frac{b}{\beta}$, we have that

$$f(x_k) \leq f(x_{k-1}) - t_{min}\Big(1 - \frac{\beta}{2} \cdot \frac{1}{\beta}\Big)\|\nabla f(x_{k-1})\|^2$$

Simplifying the coefficient outside the 2-norm,

$$f(x_k) \leq f(x_{k-1}) - \frac{1}{2}t_{min}\|\nabla f(x_{k-1})\|^2$$

Rearranging the equation, we have

$$\frac{t_{min}}{2}\|\nabla f(x_{k-1})\|^2 \leq f(x_{k-1}) - f(x_k)$$

Summing both sides up from $k = 1$ to $k = n$ causes all the middle terms on the right hand side cancel. Thus, we are left with

$$\frac{t_{min}}{2}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq \sum_{k=1}^{n} f(x_{k-1}) - f(x_k)$$

$$\frac{t_{min}}{2}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq f(x_0) - f(x_n)$$

If we let f^* be the value given by finding $\min\limits_x f(x)$, we can say that

$$\frac{t_{min}}{2}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq f(x_0) - f(x_n) \leq f(x_0) - f^*$$

Dividing both sides by n gives us the average value of $\|\nabla f(x_{k-1})\|^2$ on left side, which is clearly greater than or equal to the minimum value of $\|\nabla f(x_{k-1})\|^2$, so we have

$$\frac{t_{min}}{2}\min_k \|\nabla f(x_k)\|^2 \leq \frac{t_{min}}{2n}\sum_{k=1}^{n}\|\nabla f(x_{k-1})\|^2 \leq \frac{1}{n}(f(x_0) - f^*)$$

Rearranging the equation gives us

$$\min_k \|\nabla f(x_k)\|^2 \leq \frac{2}{t_{min}} \cdot \frac{1}{n}(f(x_0) - f^*)$$

Thus, since t_{min} is a non-zero constant, the function is bounded above, as desired. $\square$

Since both of these values for the constant of gradient descent yield that $\min\limits_{k=1\ldots n}\|\nabla f(x_k)\|^2 \leq \frac{C}{n}$ for some constant C, for any ϵ we can find a value of n such that $\frac{C}{n} < \epsilon$, so we have that $\|\nabla f(x_k)\| < \epsilon$ has a finite upper bound, as we wanted.

3.3 BFGS Method

In addition to gradient descent, we sought to employ other algorithms in order to compare their effectiveness in solving the minimization problem. One of the others we tried was the BFGS method. This method approximates Newton's method, which has quadratic convergence, for super-linear convergence that is in practice faster than gradient descent, although this has not been proven. The

BFGS method requires updating a guess, similarly to gradient descent, but the guess is updating according to the following equation:

$$x_k = x_{k-1} - t_k H_{k-1} \nabla f(x_{k-1})$$

where t_k is the same as in gradient descent and H_k is an approximation of the Hessian, which is used in Newton's method. H_k can be approximated by starting with some matrix of the proper dimensions (the same height and width as the height of x) that is positive definite, say the identity matrix, and updating it according to the formula

$$H_k = (I - \rho_{k-1} s_{k-1} y_{k-1}^T) H_{k-1} (I - \rho_{k-1} y_{k-1} s_{k-1}^T) + \rho_{k-1} s_{k-1} s_{k-1}^T$$

with

$$s_{k-1} = x_k - x_{k-1}$$
$$y_{k-1} = \nabla f_k - \nabla f_{k-1}$$
$$\rho_{k-1} = \frac{1}{y_{k-1}^T s_{k-1}}$$

The gradient ∇f and the factor t_k can be computed as in gradient descent.

3.4 Gauss-Newton Method

Another alternative optimization method we tried is the Gauss-Newton method, which is essentially a modification of Newton's line search. In this method, we used an upper bounding function and minimized that, taking the minimizing value x to be our new guess. To get this upper bounding function, because

$$\frac{1}{2}\|\Lambda_0 - \Lambda(\gamma)\|^2$$

is a composition of two functions we utilized the following theorem.

Theorem 3.4. If h is L-Lipschitz and c is β-smooth, then

$$h(c(x)) \leq h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})) + \frac{L\beta}{2}\|x - \bar{x}\|^2$$

Proof. Because h is L-Lipschitz, by definition we have that

$$|h(c(x)) - h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x}))| \leq L\|c(x) - c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})\|$$

Since c is β-smooth, we have that

$$\forall x, y \ \|\nabla c(x) - \nabla c(y)\| \leq \beta\|x - y\| \text{ for some } \beta$$

which is equivilent to

$$c(x) - (c(y) + \nabla c(y)(x - y)) \leq \frac{\beta}{2}\|x - y\|$$

for all x and y. Thus, we have that

$$L\|c(x) - (c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x}))\| \leq L\|\frac{\beta}{2}\|x - \bar{x}\|^2\|$$

Because $\frac{\beta}{2}\|x - \bar{x}\|^2$ is a real number, we have

$$L\|\frac{\beta}{2}\|x - \bar{x}\|^2\| = L\Big|\frac{\beta}{2}\|x - \bar{x}\|^2\Big| = \frac{L\beta}{2}\|x - \bar{x}\|^2$$

Therefore, since

$$h(c(x)) - h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})) \le |h(c(x)) - h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x}))|$$

we have that

$$h(c(x)) - h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})) \le \frac{L\beta}{2}\|x - \bar{x}\|^2$$

so

$$h(c(x)) \le h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})) + \frac{L\beta}{2}\|x - \bar{x}\|^2$$

as desired. □

The result of this theorem is useful in minimizing our function $\frac{1}{2}\|\Lambda_0 - \Lambda(\boldsymbol{\gamma})\|^2$. If we take $h(y) = \frac{1}{2}\|y\|^2$ and $c(\boldsymbol{\gamma}) = \Lambda_0 - \Lambda(\boldsymbol{\gamma})$, we can find the minimum of the function $h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})) + \frac{L\beta}{2}\|x - \bar{x}\|^2$ by taking the derivative.

3.4.1 Derivative of $h(c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})) + \frac{L\beta}{2}\|x - \bar{x}\|^2$

Let h be a function $h : \mathbb{R}^{n\times n} \to \mathbb{R}$ and $c : \mathbb{R}^m \to \mathbb{R}^{n\times n}$
We want to differentiate

$$h(c(\bar{x}) + \nabla c(x)(x - \bar{x})) + \frac{L\beta}{2}\|x - \bar{x}\|^2$$

First, we will consider the term $h(c(\bar{x}) + \nabla c(x)(x - \bar{x}))$. By the chain rule, we have

$$\frac{\partial h \circ f}{\partial x_k} = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial h}{\partial y_{ij}} \cdot \frac{\partial f_{ij}}{\partial x_k}$$

Where $f = c(\bar{x}) + \nabla c(\bar{x})(x - \bar{x})$. Next, we consider the term $\frac{L\beta}{2}\|x - \bar{x}\|^2$. Using the chain rule again, we set $h = \frac{L\beta}{2}\|y\|^2$ and $f = x - \bar{x}$
Thus,

$$\frac{\partial h}{\partial y} = L\beta \cdot y_{ij}$$

and

$$\frac{\partial f}{\partial x_k} = \begin{bmatrix} \frac{\partial}{\partial x_k}(x_1 - \bar{x}_1) \\ \vdots \\ \frac{\partial}{\partial x_k}(x_n - \bar{x}_n) \end{bmatrix}$$

So

$$\frac{\partial h \circ f}{\partial x_k} = \sum_{i=1}^{n} \beta L(x - \bar{x})_i \cdot \frac{\partial f_i}{\partial x_k}$$

Since $\frac{\partial f}{\partial x_k}$ is a vector of all zeros except for 1 at index k, the sum over i is 0 except for when $i = k$.
So

$$\frac{\partial h \circ f}{\partial x_k} = \beta L(x - \bar{x})_k$$

Thus, the full derivative (general form) is

$$\frac{\partial h \circ f}{\partial x_k} = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial h}{\partial y_{ij}} \cdot \frac{\partial f_{ij}}{\partial x_k} + \beta L(x - \bar{x})_k$$

Because $h(y) = \frac{1}{2}\|y\|^2$ and $c(\boldsymbol{\gamma}) = \Lambda_0 - \Lambda(\boldsymbol{\gamma})$, we can simplify the derivative. As shown in the computing the gradient section, we have that

$$\frac{\partial h}{\partial y_{ij}} = y_{ij}$$

and since $y_{ij} = f(x)_{ij}$, we can say that

$$\frac{\partial f_{ij}}{\partial x_k} = \frac{\partial}{\partial x_k}(\Lambda_0 - \Lambda(\bar{x}) + \nabla(\Lambda_0 - \Lambda(\bar{x}))(x - \bar{x}))_{ij} = -\Big(\frac{\partial}{\partial x_k}\Lambda(\bar{x})\Big)_{ij}$$

and

$$y_{ij} = f(x)_{ij} = (c(\bar{x})+\nabla c(\bar{x})(x-\bar{x}))_{ij} = (\Lambda_0-\Lambda(\bar{x})+\nabla(\Lambda_0-\Lambda(\bar{x}))(x-\bar{x}))_{ij} = (\Lambda_0-\Lambda(\bar{x}))_{ij}-(\nabla\Lambda(\bar{x})(x-\bar{x}))_{ij}$$

Because we have that

$$(\nabla\Lambda(\bar{x})(x - \bar{x}))_{ij} = \sum_{r=1}^{n} \frac{\partial \Lambda}{\partial x_r}(\bar{x})(x_r - \bar{x}_r)$$

we can write the full derivative for our specific case as

$$\frac{\partial h \circ f}{\partial x_k} = \sum_{i=1}^{n}\sum_{j=1}^{n}\Big((\Lambda(\bar{x}) - \Lambda_0)_{ij} + \sum_{r=1}^{n}\frac{\partial \Lambda_{ij}}{\partial x_r}(\bar{x})(x_r - \bar{x}_r)_{ij}\Big)\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})\Big) + \beta L(x_k - \bar{x}_k)$$

We can set this equal to zero to find the value of $x - \bar{x}$ that minimizes the function. We have that

$$\frac{\partial h \circ f}{\partial x_k} = \sum_{i=1}^{n}\sum_{j=1}^{n}\Big((\Lambda(\bar{x}) - \Lambda_0)_{ij} + \sum_{r=1}^{n}\frac{\partial \Lambda_{ij}}{\partial x_r}(\bar{x})(x_r - \bar{x}_r)_{ij}\Big)\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})\Big) + \beta L(x_k - \bar{x}_k) = 0$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(\Lambda(\bar{x}) - \Lambda_0)_{ij}\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})\Big) + \sum_{i=1}^{n}\sum_{j=1}^{n}\Big(\sum_{r=1}^{n}\frac{\partial \Lambda_{ij}}{\partial x_r}(\bar{x})(x_r - \bar{x}_r)_{ij}\Big)\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})\Big) + \beta L(x_k - \bar{x}_k) = 0$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\Big(\sum_{r=1}^{n}\frac{\partial \Lambda_{ij}}{\partial x_r}(\bar{x})(x_r - \bar{x}_r)_{ij}\Big)\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})\Big) + \beta L(x_k - \bar{x}_k) = -\sum_{i=1}^{n}\sum_{j=1}^{n}(\Lambda(\bar{x}) - \Lambda_0)_{ij}\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})\Big)$$

This puts the constant terms on the right side, which is exactly $-\nabla(\frac{1}{2}\|\Lambda_0 - \Lambda(\bar{x})\|^2)$ and thus can be computed as shown in the previous section. We can now put the equation in the form $Az = b$, where $z = x - \bar{x}$, b is the constant vector $-\nabla(\frac{1}{2}\|\Lambda_0 - \Lambda(\bar{x})\|^2)$, and A is the matrix of the coefficients for z, with A_{ij} being the coefficient for the j^{th} variable in the k^{th} equation. Thus, since $\frac{\partial \Lambda_{ij}}{\partial x_k}(\bar{x})$ is a scalar term, we can define A as follows:

$$A_{st} = \begin{cases} \sum_{i=1}^{n}\sum_{j=1}^{n}\Big(\frac{\partial \Lambda_{ij}}{\partial x_s}(\bar{x})\Big)\cdot\Big(\frac{\partial \Lambda_{ij}}{\partial x_t}(\bar{x})\Big) & \text{if } s \neq t \\ \sum_{i=1}^{n}\sum_{j=1}^{n}\Big(\frac{\partial \Lambda_{ij}}{\partial x_s}(\bar{x})\Big)^2 + \beta L & \text{if } s = t \end{cases}$$

Because of the βL factor added to the diagonal, A can always be made invertible, so we can solve for z by finding

$$z = A^{-1}b$$

with A and b as described above. Because $z = x - \bar{x}$, we can pick our new guess x by taking

$$x = \bar{x} + z$$

We can continue to increment x in this way, with $\bar{x}$ being the previous guess and x becoming the new $\bar{x}$ in the next iteration, until our original function $\frac{1}{2}\|\Lambda_0 - \Lambda(x)\|^2$ is as close to zero as we require.

4 The Program

Recall that we sought to solve the inverse problem by finding

$$\min_{\gamma \geq 0} \frac{1}{2} \|\Lambda_0 - \Lambda(\gamma)\|^2$$

through determining the set of conductances γ that produce the desired response matrix. In order to do this, we take an initial guess and repeatedly improve the guess through iteration, gradient calculation, and corresponding correction. To quickly attain this repetition, we developed a program in MATLAB that intakes an experimental response matrix Λ, which may have some error, and outputs a closest set of conductances (least squares solution) that maps to a valid Λ.

We used multiple different algorithms to attain this goal in the most effective way possible. Each of the three algorithms we used, gradient descent, BFGS, and Gauss-Newton, has its own advantages and disadvantages, so we were able to use each of them at different times or various combinations of the algorithms in order to obtain the best result.

4.1 Implementation of Algorithms

We provide a brief outline of the way we implemented the three main algorithms used in our program. The full MATLAB code can be found on page 21.

Gradient Descent Algorithm

```
Take experimental response matrix eLambda and initial guess, where the guess corresponds
    to gamma, graph information representing the number of boundary and interior nodes,
    and initial value of t
define guess
while (norm of gradient > precision value, some number close to 0) {
   take gradient of eLambda - Lambda(x) norm squared
   calculate step size t, with method described in gradient descent section, taking a = b
       = 0.5
   nextguess = guess - t * gradient
   if (nextguess < 0) {
      nextguess = 0
   }
   guess = nextguess;
}
return guess
```

The gradient descent algorithm normally converged the most slowly of the algorithms we tried. However, its consistency and the convergence rates we were able to prove about it made it a good choice when we needed a steady method that would be less likely to cause a matrix to initially go singular or to move about wildly even when the guess was very inaccurate.

BFGS Algorithm

```
Take experimental response matrix eLambda and initial guess, where the guess corresponds
    to gamma, graph information representing the number of boundary and interior nodes,
    and initial value of t
define guess
while (norm of gradient > precision value, some number close to 0) {
   take gradient of eLambda - Lambda(x) norm squared
```

```
    calculate step size t, with method described in gradient descent section
    calculate s, y, and p, and using those calculate Hk with the method described in the
        BFGS section
    nextguess = guess - t * Hk * gradient
    if (nextguess < 0) {
        nextguess = 0
    }
    guess = nextguess;
}
return guess
```

When we implemented the BFGS algorithm, as expected it tended to converge much more quickly than gradient descent when it worked properly. However, the guesses tended to fluctuate a bit wildly early in the process. This sometimes caused problems, particularly when the initial guess is somewhat far from the solution, because it could cause the matrix to become singular to working precision. We found an effective solution was to run gradient descent to improve the guess and then to run BFGS for minimal convergence times.

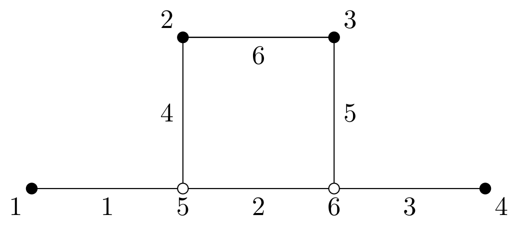
Gauss-Newton Algorithm

```
Take experimental response matrix eLambda and initial guess, where the guess corresponds
    to gamma, and graph information representing the number of boundary and interior nodes
define guess
while (eLambda - Lambda(x) norm squared > precision value, some number close to 0) {
    create the matrix A that represents the series of equations of the form A(nextguess -
        guess) = b given by setting the gradient of the upper bounding function described
        in the Gauss-Newton section equal to zero
    calculate the vector b, which is the negative of the gradient of eLambda -
        Lambda(guess) norm squared
    calculate difference = (nextguess - guess) by finding A^(-1)b
    nextguess = guess + difference
    if (nextguess < 0) {
        nextguess = 0
    }
    guess = nextguess;
}
return guess
```

Note that the Gauss-Newton algorithm equation contains the Lipschitz and beta smoothness constants, L and β respectively, and causes their product to be added onto the diagonal entries of the matrix A. While for guaranteed convergence we should have determined their values through a backtracking line search algorithm similar to the one used to find t_k in the gradient descent algorithm, we found that setting L and β both equal to 1 we tended to get faster convergence, and thus we implemented our program in that way.

4.2 Example Network

Take the following graph, which contains 4 boundary vertices and 2 interior vertices. We call this the “top hat graph.”



The Kirchhoff matrix corresponding to this graph is as follows:

$$K = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 10 & -6 & 0 & -4 & 0 \\ 0 & -6 & 11 & 0 & 0 & -5 \\ 0 & 0 & 0 & 3 & 0 & -3 \\ -1 & -4 & 0 & 0 & 7 & -2 \\ 0 & 0 & -5 & -3 & -2 & 10 \end{bmatrix}$$

We then generate the response matrix Λ, which is equal to the Schur complement of K, and enter it into a variable called `eLambda` in our MATLAB program. Note that in actual application, the response matrix is found experimentally and may or may not contain experimental error. In this case, `eLambda` is equal to the following response matrix:

$$\Lambda_0 = \begin{bmatrix} 0.8485 & -0.6061 & -0.1515 & -0.0909 \\ -0.6061 & 7.5758 & -6.6061 & -0.3636 \\ -0.1515 & -6.6061 & 8.3485 & -1.5909 \\ -0.0909 & -0.3636 & -1.5909 & 2.0455 \end{bmatrix}$$

We also enter a initial guess `x0` of the vector of conductances for the network. In general, we use a vector of all ones as the first guess. The program then iterates through the optimization algorithm, replacing the guess vector with a closer one until the norm of the gradient of

$$\frac{1}{2}\|\Lambda_0 - \Lambda(\gamma)\|^2$$

is very close to zero; we usually use 10^{-8} as the cutoff. The program then outputs the final guess vector that produces a gradient of approximately 0. In this case, the program returns the matrix

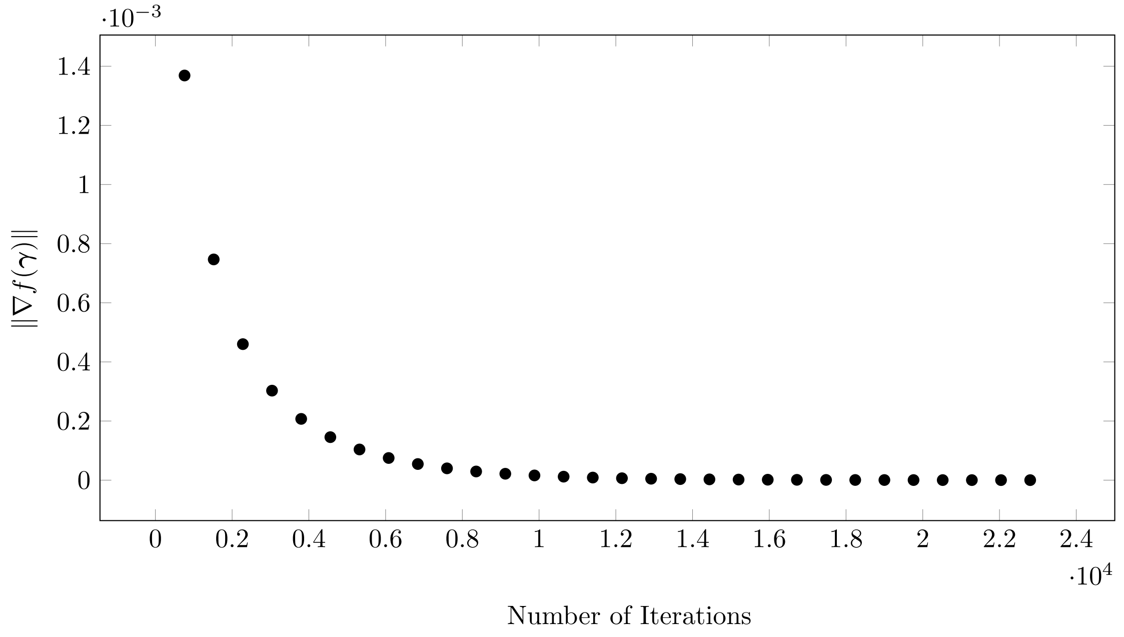
$$\begin{bmatrix} 1.0000 & 5.0000 & 1.0000 \\ 5.0000 & 6.0000 & 2.0000 \\ 4.0000 & 6.0000 & 3.0000 \\ 2.0000 & 5.0000 & 4.0000 \\ 3.0000 & 6.0000 & 5.0000 \\ 2.0000 & 3.0000 & 6.0000 \end{bmatrix}$$

wherein the first two columns indicate the i and j vertices for each edge ij, and the third column gives the corresponding conductance γ_{ij}.

5 Comparing Methods

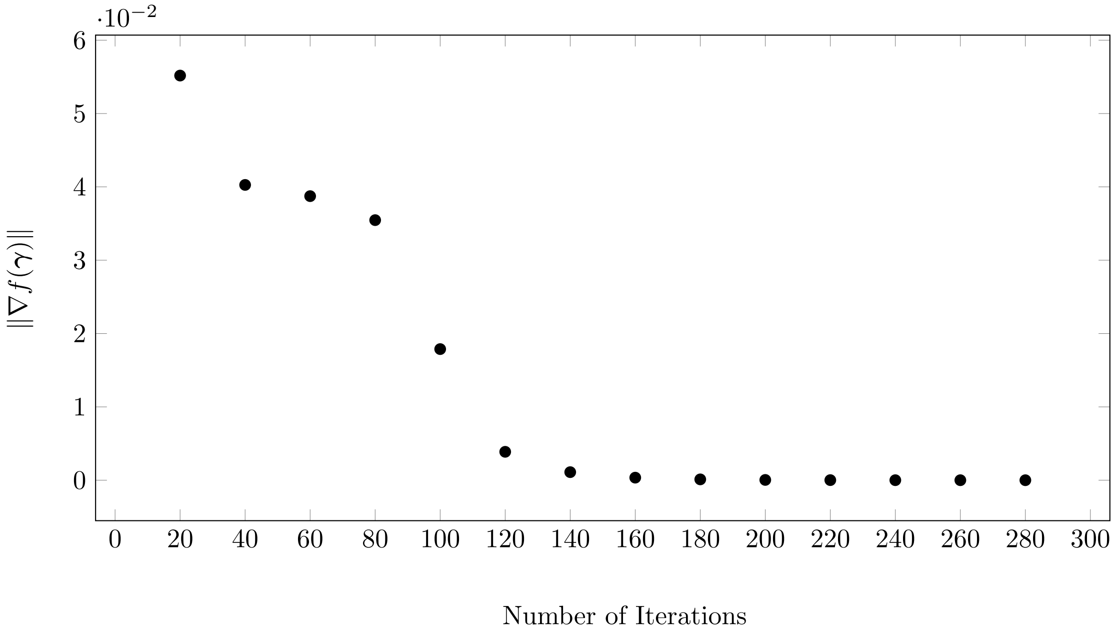
In order to determine which of our three methods were most reliable and/or efficient, we looked at the norm of the gradient of our function at various intervals throughout the iterative process for each method. We then graphed the data on scatter plots to see how each method approaches zero and which methods are fastest. For the labeling of the axes for these scatter plots, let $f(\gamma) = \frac{1}{2}\|\Lambda_0 - \Lambda(\gamma)\|^2$, and note that each of the methods were run until the condition $\|\nabla f(\gamma)\| < 10^{-8}$ was met. The following plots were each generated using the same graph and initial guess from the example in section 4.1.

5.1 Gradient Descent Method



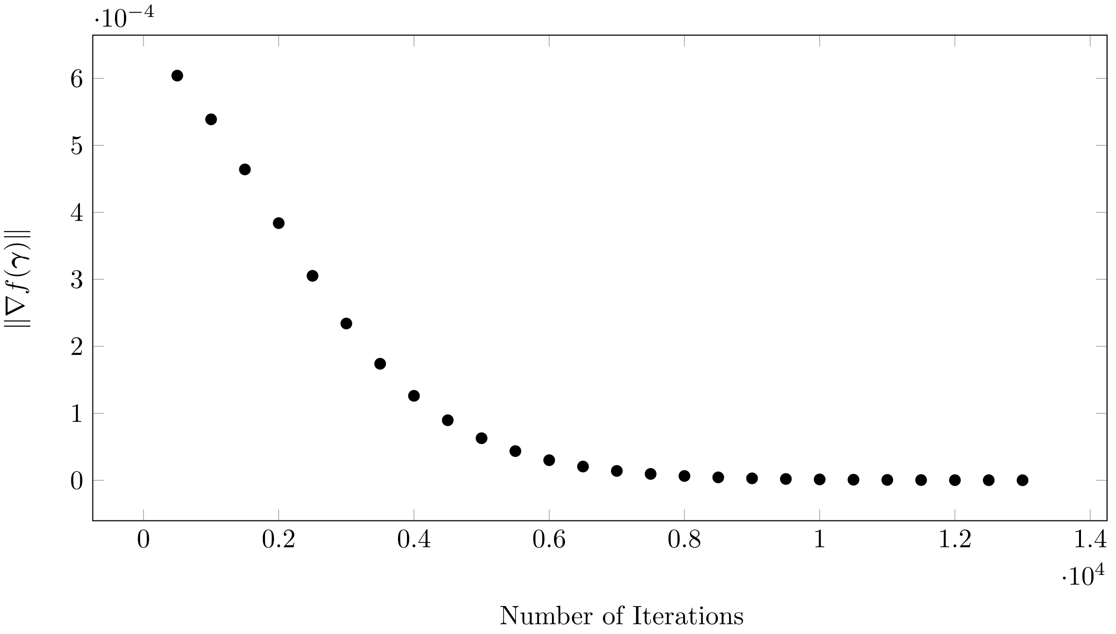
This method took 22,870 iterations to converge to a solution. From the graph we see that the norm approached zero smoothly and gradually without any unexpected spikes or plateaus, which is consistent with the proofs of rate of convergence presented in the Gradient Descent section.

5.2 BFGS Method



This method took 285 iterations to complete, a dramatic reduction compared to gradient descent. However, it follows a less smooth path; it makes a sharp initial descent, then plateaus, and then drops off again and steadily approaches zero. We tested this method on other graphs as well and saw this pattern emerge as a characteristic of the BFGS method. While BFGS converges in the fewest iterations of the three methods, it is the least stable and can often fail to converge if the initial guess is far away from the solution.

5.3 Gauss-Newton Method



Similar to gradient descent, the Gauss-Newton method follows a steady curve, but converges faster than gradient descent with just 13,377 iterations.

5.4 Combining Methods

While the gradient descent method clearly takes many more iterations to converge to a solution than the other methods, it produces a much smoother curve and reliably converges almost regardless of how close the original guess is.The BFGS method is quickest and also converges reliably with a close initial guess, so can be very useful in cases when a close guess is available. The Gauss-Newton method is somewhat of an intermediate option between the first two. The methods might be most useful when employed in succession; gradient descent or the Gauss-Newton Method could be used first to get close to convergence, then the BFGS method could be used to reduce the number of iterations toward the end. This can easily be implemented by setting the precision value of the first method, say gradient descent, to some larger value. Then at the end of the method, call the second method, say BFGS, using the final guess produced by the first method, and return the result given by the second method.

6 n-to-1 Graphs

In some cases, a single graph can be written with multiple sets of valid conductances that map to the exact same response matrix. We call these n-to-1 graphs, n being the number of different sets corresponding to one Λ. We are interested in using our program to recover all solution sets of such graphs, or at least establish a lower bound on n by finding some number of solutions.

Many of the currently known n-to-1 graphs resulted from careful construction in order to allow the n solutions to be obtained by finding the solutions of a polynomial of degree n that also satisfy positivity conditions, or in some other such specific manner. However, there is currently no good way of determining whether a randomly chosen graph is n-to-1 or not.

6.1 Running the Program on n-to-1 Graphs

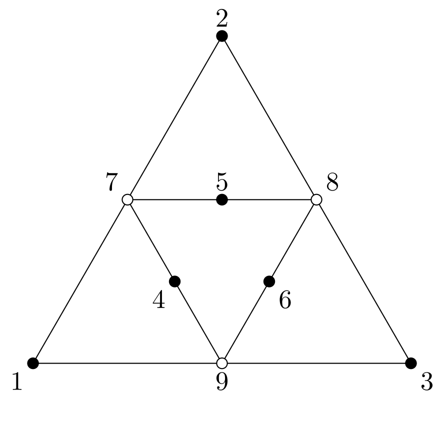
One of our original goals was to build a program that would be able to recover all solution sets for n-to-1 graphs. In order to do this, we used the three optimization methods described earlier, and tried multiple different first guesses with the expectation that the final guess would converge to whichever of the n solution sets was closest to the original guess. However, coming up with graphs to try this on proved more difficult than expected. Particularly when $n > 2$, constructing a valid n-to-1 graph is an especially delicate process because there is a very small range of solutions that will all map to valid Kirchhoff matrices. Because of this, we cannot just assign random $\boldsymbol{\gamma}$ values to the graph and expect to retrieve other valid solution sets.

Instead, we often used pre-constructed n-to-1 graphs for which we already knew all n solution sets to test our program. We found that in most cases the program will return only the solution that was originally used to construct Λ_0. In general, for higher n's, the program would only return a different solution set if the original guess was extremely close to it already. This is slightly problematic, since our program should ideally be able to find all solution sets even if we have no prior knowledge of them.

6.2 Examples

Not all network configurations can be n-to-1. These types of graphs are typically constructed using star-K transformations and the quadrilateral rule to create a system of equations with multiple solutions (5). Note that the following networks are examples of 2- and 3-to-1 graphs but are not the only network shapes that can be 2- or 3-to-1.

6.2.1 2-to-1 Graph



This is an example of a 2-to-1 graph, as shown in (3). For this graph, we were able to arbitrarily assign a set of conductances 1 through 12 and run the program to recover another set of conductances, not already known, that correspond to the same response matrix. This was achieved by varying our initial guess and re-running the program until we came up with a guess that converged to a new solution.
For this graph, setting the original guess

$$x_0 = \begin{bmatrix} 1 & 7 & 1 \\ 1 & 9 & 1 \\ 3 & 9 & 1 \\ 3 & 8 & 1 \\ 2 & 8 & 1 \\ 2 & 7 & 1 \\ 4 & 7 & 1 \\ 4 & 9 & 1 \\ 6 & 9 & 1 \\ 6 & 8 & 1 \\ 5 & 8 & 1 \\ 5 & 7 & 1 \end{bmatrix} \text{ yields } x_{\text{final}} = \begin{bmatrix} 1.0000 & 7.0000 & 1.0000 \\ 1.0000 & 9.0000 & 2.0000 \\ 3.0000 & 9.0000 & 3.0000 \\ 3.0000 & 8.0000 & 4.0000 \\ 2.0000 & 8.0000 & 5.0000 \\ 2.0000 & 7.0000 & 6.0000 \\ 4.0000 & 7.0000 & 7.0000 \\ 4.0000 & 9.0000 & 8.0000 \\ 6.0000 & 9.0000 & 9.0000 \\ 6.0000 & 8.0000 & 10.0000 \\ 5.0000 & 8.0000 & 11.0000 \\ 5.0000 & 7.0000 & 12.0000 \end{bmatrix}$$

However, modifying our initial guess as follows,

$$x_0 = \begin{bmatrix} 1 & 7 & 100 \\ 1 & 9 & 1 \\ 3 & 9 & 100 \\ 3 & 8 & 1 \\ 2 & 8 & 100 \\ 2 & 7 & 1 \\ 4 & 7 & 100 \\ 4 & 9 & 1 \\ 6 & 9 & 100 \\ 6 & 8 & 1 \\ 5 & 8 & 100 \\ 5 & 7 & 1 \end{bmatrix} \text{ yields } x_{\text{final}} = \begin{bmatrix} 1.0000 & 7.0000 & 1.2544 \\ 1.0000 & 9.0000 & 1.7217 \\ 3.0000 & 9.0000 & 3.7219 \\ 3.0000 & 8.0000 & 3.2420 \\ 2.0000 & 8.0000 & 6.8232 \\ 2.0000 & 7.0000 & 4.1198 \\ 4.0000 & 7.0000 & 8.7810 \\ 4.0000 & 9.0000 & 6.8869 \\ 6.0000 & 9.0000 & 11.1657 \\ 6.0000 & 8.0000 & 8.1050 \\ 5.0000 & 8.0000 & 15.0110 \\ 5.0000 & 7.0000 & 8.2397 \end{bmatrix}$$

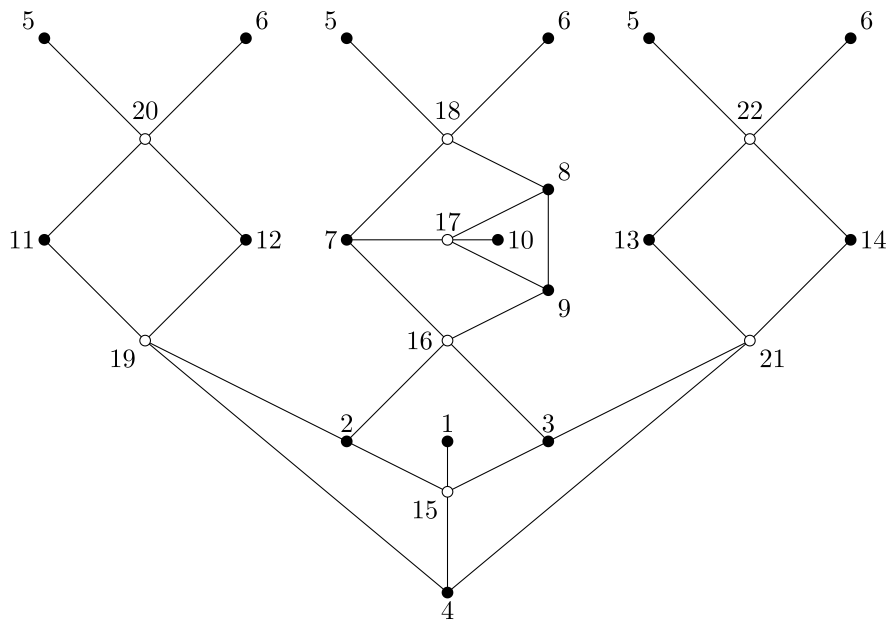
Note that both of the solutions outputted by the program give the same response matrix

$$\Lambda = \begin{bmatrix} 2.7797 & -0.2308 & -0.2727 & -0.9965 & -0.4615 & -0.8182 \\ -0.2308 & 8.7821 & -0.6667 & -1.6154 & -4.6026 & -1.6667 \\ -0.2727 & -0.6667 & 6.0576 & -1.0909 & -1.4667 & -2.5606 \\ -0.9965 & -1.6154 & -1.0909 & 10.2063 & -3.2308 & -3.2727 \\ -0.4615 & -4.6026 & -1.4667 & -3.2308 & 13.4282 & -3.6667 \\ -0.8182 & -1.6667 & -2.5606 & -3.2727 & -3.6667 & 11.9848 \end{bmatrix}$$

so we know that both solutions are valid.

The strategy for choosing x_0 is not fully refined, but we have developed some intuition with regards to what normally works. In order to find different solutions, we need to choose x_0 such that taking the path of steepest descent of the gradient would lead to a new solution. Thus, choosing guesses that are very diverse and distinct in both magnitude and ratios between entries seems to work best for recovering multiple solutions. For example, picking one x_0 to have similar or same values for all edges, such as the first example initial guess, and another x_0 to have alternating large and small values like the second initial guess tends to yield good results.

6.2.2 3-to-1 Graph



This graph is called the “threehands” graph with one inversion and has three known solutions that are listed in Grigoriev’s paper (4). We used the given numbers in our program and attempted to recover all three of the solutions. In practice, the output usually converged to the first solution unless the original guess was set to be extremely close to another known solution. It’s also worth mentioning that on graphs of this size, the program takes a rather large amount of time (hours at least) to run to completion, so testing various initial guesses can quickly become time-consuming. However, we were at least able to get the program to converge to the known solutions by setting the initial guess equal to the floor of the other known solution, though this did not significantly reduce runtime.

7 Remaining

While we managed to accomplish many of our initial goals, much remains to be pursued by future researchers. In particular, we were unable to make significant progress on using our program on n-to-1 graphs. While we had some success with the 2-to-1 graph, we were unable to develop an effective method for recovering all the solutions for n-to-1 graphs with $n > 2$. Additionally, our current method relies on manually entering initial guesses and checking to see whether the solutions are different enough to be considered distinct. Some sort of automatization and possibly algorithmic approach to this process would be worthwhile. Even a smart way of picking an initial guess for 1-to-1 graphs that minimizes the possibility of a matrix going computationally singular could be helpful.

Also, further experimentation with the effectiveness of combining of different algorithms could yield new insights. We scratched the surface through informal experimentation with combining BFGS and gradient descent in order to avoid singularity while still taking advantage of the speed of convergence of BFGS, but this area would benefit from more rigorous testing.

8 MATLAB Code

Main.m

```
% This program was written for the University of Washington 2016 Math REU by Melanie
    Ferreri and Christine Wolf.
% This is the main method in which we enter our Lambda and first guess, then call on the
    function to generate a least squares solution for the inverse problem.

% Here we enter the numbers of boundary (dVNum) and interior (intV) edges on the graph.
dVNum = 4;
intV = 2;
dimK = dVNum + intV;

% Example network: top hat graph

% actual values from which we generated eLambda
a = [1 5 1;
    5 6 2;
    4 6 3;
    2 5 4;
    3 6 5;
    2 3 6];

% vector of all ones we input as the first guess
x0 = [1 5 1;
    5 6 1;
    4 6 1;
    2 5 1;
    3 6 1;
    2 3 1];

% Generate Kirchhoff matrix from actual set of values a
m = makeKirchhoff(a, dimK);
% Generate Lambda from the Kirchhoff matrix
l = getLambda(m, dVNum, dimK);

% Note that you can also manually input a response matrix with error and the program will
    still give you a closest valid set of gammas

format long;

% Run optimization method of your choice. Can also combine methods.
xFinal = getXk(x0, dimK, dVNum, l, 3);

% Print final guess
xFinal
```

getXkGradientDescent.m

```matlab
% Takes an original x guess, dimK, dVNum, empirical lambda, and scaling factor t as
    parameters and returns an x guess that produces a response matrix close enough to the
    desired one by the desired precision using gradient descent.
function x = getXkGradientDescent(x0, dimK, dVNum, eLambda, t)
   gradX = getGradient(x0, dimK, eLambda, dVNum);
   x = x0;
   prev = x;
   total = norm(gradX);
   min = false;
   while total > 0.0000001 % can change precision value
      gradX = getGradient(prev, dimK, eLambda, dVNum);
      % increments t by 0.5 if it satisfies the backtracking line search conditions
      if min == false
         factor = incrementT(x, prev, dimK, dVNum, eLambda, t, gradX);
         t = factor * t;
         if factor == 1
            min = true;
         end
      end
      x = prev - t * gradX; % gradient descent
      x = max(0, x);
      prev = x;
      total = norm(gradX);
      % print current guess
      x
   end
end
```

getXkBFGS.m

```matlab
% Takes an original x guess, dimK, dVNum, empirical lambda, and scaling factor t as
    parameters and returns an x guess that produces a response matrix close enough to the
    desired one by the desired precision using BFGS.
function x = getXkBFGS(x0, dimK, dVNum, eLambda, t)
   gradX = getGradient(x0, dimK, eLambda, dVNum);
   x = x0;
   prev = x;
   heightX = size(x, 1);
   % positive definite matrix of proper dimensions
   H = eye(heightX);
   total = norm(gradX);
   min = false;
   while total > 0.0000001 % can change precision value
      p = -1 * H * gradX;
      % increments t by 0.5 if it satisfies the backtracking line search conditions
      if min == false
         factor = incrementT(x, prev, dimK, dVNum, eLambda, t, gradX);
         t = factor * t;
         if factor == 1 % incrementing t has finished
            min = true;
         end
      end
      x = prev + t * p;
      x = max(0, x);
      gradXNew = getGradient(x, dimK, eLambda, dVNum);
```

```
        y = gradXNew - gradX;
        gradX = gradXNew;
        s = x - prev;
        % removes vertex information from s and y
        s = s(1:heightX, 3);
        y = y(1:heightX, 3);
        H = getHk(H, s, y);
        prev = x;
        total = norm(gradX);
        % print current guess
        x
    end
end
```

getXkGaussNewton.m

```
% Takes an original x guess, dimK, dVNum, empirical lambda, and t (which is taken so that
    the method can be easily interchanged with the other getXk methods without switching
    parameters) as parameters and returns an x guess that produces a response matrix close
    enough to the desired one by the desired precision using the Gauss-Newton method.
function x = getXkGaussNewton(x0, dimK, dVNum, eLambda, t)
    x = x0;
    prev = x;
    xHeight = size(x, 1);
    A = zeros(xHeight, xHeight); % dimension # of edges
    % beta and Lipschitz constants
    beta = 1;
    L = 1;
    K = makeKirchhoff(x0, dimK);
    % get norm of function to minimize
    total = norm(getGradient(prev, dimK, eLambda, dVNum));
    % Uncomment following line to use backtracking line search to find L
    % min = false;
    while total > 0.0000001 % can change precision value
        gradX = getGradient(x, dimK, eLambda, dVNum);
        % Uncomment following lines to use backtracking line search to find L
        % increments L by 0.5 if it satisfies the backtracking line search conditions
        % if min == false
        %    factor = incrementT(x, prev, dimK, dVNum, eLambda, t, gradX);
        %    L = factor * L;
        %    if factor == 1 % incrementing L has finished
        %        min = true;
        %    end
        % end
        b = -1 * getGradient(prev, dimK, eLambda, dVNum); % constant vector
        for i = 1:xHeight
            for j = i:xHeight
                % derivatives of lambda with respect to x_i and x_j
                lam1 = getLambdaDerivative(prev(i, 1), prev(i, 2), K, dimK, dVNum);
                lam2 = getLambdaDerivative(prev(j, 1), prev(j, 2), K, dimK, dVNum);
                lam = (lam1.*lam2);
                A(i, j) = sum(lam(:));
                A(j, i) = A(i, j);
            end
        end
        for k = 1:xHeight
            % adds beta*L to the diagonal entries
            A(k, k) = A(k, k) + beta * L;
        end
        % removes information about vertices
        b = b(1:xHeight, 3);
        % adds vertices information and has values (x - prev) + prev
        x = [prev(1:xHeight, 1:2) (A \ b + prev(1:xHeight, 3))];
        prev = x;
        K = makeKirchhoff(prev, dimK);
        total = norm(gradX);
        % print current guess
        x
    end
end
```

---

getGradient.m

```
% Takes a 3xn matrix x, dimK, empirical lambda, and dVNum as parameters and returns the
    gradient of the norm squared of the difference between the experimental lambda and the
    response matrix corresponding to the given x.
function gradient = getGradient(x, dimK, eLambda, dVNum)
    xHeight = size(x,1);
    % eLambda is empirical lambda
    K = makeKirchhoff(x, dimK);
    LambdaX = getLambda(K, dVNum, dimK);
    leftside = eLambda - LambdaX;
    % rightside is actually just d of x_k
    % now need to get d of x_k for each x (3rd entry) in x vector
    gradient = zeros(xHeight, 3);
    for i = 1:xHeight
        d = getLambdaDerivative(x(i,1),x(i,2),K,dimK,dVNum);
        % for every boundary-to-boundary edge, normalize vector
        if (x(i,1) <= dVNum && x(i,2) <= dVNum)
            d = d/norm(d);
        end
        A = (leftside.*d);
        % -sum because we want the derivative of -lambda
        gradient(i,3) = -sum(A(:));
    end
end
```

---

getLambdaDerivative.m

```
% Takes the vertices on either end of the edge corresponding to the desired gamma, a
    Kirchhoff matrix, and dimK and dVNum as parameters and returns the derivative of the
    response matrix corresponding to the Kirchhoff matrix with respect to the desired
    gamma.
function deriv = getLambdaDerivative( vertex1, vertex2, K, dimK, dVNum )
    % number of interior vertices
    B = K(1:dVNum, (dVNum + 1):dimK);
    C = K((dVNum + 1):dimK, (dVNum + 1):dimK);
    % dim D is supposed to be dVNum row by dimK col
    I = eye(dVNum);
    D2 = -C \ transpose(B);
    D = [I ; D2];
    % creates the matrix dK/dgamma
    dKMaker = [vertex1 vertex2 1];
    dKMatrix = makeKirchhoff(dKMaker, dimK);
    deriv = transpose(D) * dKMatrix * D;
end
```

getHk.m

```
% Takes the previous H matrix and vectors s and y, and returns the value for Hk.
function H = getHk(prevH, s, y)
    r = 1/((y.') * s + 0.0001);
    I = eye(size(prevH, 1));
    H = (I - r * s * (y.')) * prevH * (I - r * y * (s.')) + r * s * (s.');
end
```

incrementT.m

```
% Takes vectors x and prev, ints for dimK and dVNum, the experimental lambda,t, and the
    gradient of x as parameters. Returns 0.5 if the condition for backtracking line search
    is satisfied, and 1 otherwise.
function beta = incrementT(x, prev, dimK, dVNum, eLambda, t, gradX)
    % Difference for L(x_new)
    new = getNormDifference(x, dimK, dVNum, eLambda);

    % Difference for L(x_prev)
    orig = getNormDifference(prev, dimK, dVNum, eLambda);

    % checks backtracking line search condition
    if(orig <= (new - 0.5*t*(norm(gradX))^2))
        beta = 0.5;
    else
        beta = 1;
    end
end
```

getNormDifference.m

```
% Takes a vector x containing conductance values, numbers for dimK and dVNum, and
    empirical lambda as parameters, returning the square of the norm of the difference
    between the experimental lambda and the response matrix generated from x.
function difNorm = getNormDifference(x, dimK, dVNum, eLambda)
    K = makeKirchhoff(x, dimK);
    LambdaX = getLambda(K, dVNum, dimK);
    leftside = eLambda - LambdaX;
    difNorm = (norm(leftside))^2;
end
```

getLambda.m

```
% Takes a Kirchhoff matrix, dVNum, and dimK as parameters and returns the corresponding
    response matrix.
function L = getLambda(K, dVNum, dimK)
    A = K(1:dVNum, 1:dVNum);
    B = K(1:dVNum, (dVNum + 1):dimK);
    C = K((dVNum + 1):dimK, (dVNum + 1):dimK);
    L = A - B*(C\(B.')); % Schur Complement
end
```

makeKirchhoff.m

```
% Takes a matrix containing the nodes an edge connects in its first and second columns and
    the conductance value in the third and dimK as parameters and returns the
    corresponding Kirchhoff matrix
function m = makeKirchhoff( x, dimK )
    % argument should be 3xn (x vector with ij specified in first 2 col)
    m = zeros(dimK);
   % data for diagonal values
    d = zeros(dimK, 1);

   % sets values and makes symmetrical and row sum zero
    for i = 1:size(x,1)
        a = x(i,1); % i coordinate in matrix
        b = x(i,2); % j coordinate in matrix
        m(a,b) = - x(i,3);
        m(b,a) = - x(i,3);
        d(a) = d(a) + x(i,3); % updates diagonal values
        d(b) = d(b) + x(i,3);
    end
    % loop to get row sums and add them to the diagonal
    for i = 1:(dimK)
        m(i,i) = d(i);
    end
end
```

Acknowledgement

This paper was written during the University of Washington Mathematics REU under the supervision of Jim Morrow. We would like to thank everyone involved in putting this program together and allowing us to perform this research. Special thanks to Courtney Kempton and Jim Morrow for many hours of explanations and guidance, from giving insights on proofs to helping us derive important formulas.

References

[1] Edward B. Curtis and James A. Morrow. *Inverse Problems for Electrical Network*. World Scientific. 2000.

[2] Dmitriy Drusvyatskiy and Courtney Kempton. *An accelerated algorithm for minimizing convex compositions*. University of Washington Department of Mathematics. 2016.

[3] Jennifer French and Shen Pan. *2^n to 1 Graphs*. University of Washington Math REU. 2004.

[4] Ilya Grigoriev. *Three 3-to-1 Graphs with Positive Conductivities*. University of Washington Math REU. 2006.

[5] Courtney Kempton. *n-1 graphs*. University of Washington Math REU. 2011.

[6] James A. Morrow. *Derivative of* Λ. University of Washington Math REU. 2013.

[7] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization, Second Edition*. Springer. 2006.