Parameter Identification: A Comparison of Methods

Drayton Munster

July 19, 2009

1 The Problem

In this project, we compare a number of methods for solving the parameter identification problem. We are given a data set containing population counts of cells on various days. We fit this data using the logistic growth population model

\[ \dot{y}(t) = q_1 \left(1 - \frac{y(t)}{q_2}\right) y(t), \quad y(0) = q_3, \]

where the parameter set \( q = [q_1, q_2, q_3]^T \) represents the growth rate, capacity and initial population. We will denote the solution obtained with parameter values \( q \) by \( y(t; q) \). Although the logistic equation can be solved analytically by the separation of variables method, we solve it numerically using MATLAB’s ode45, a 4th order Runge-Kutta algorithm with adaptive time stepping. This will allow us to easily generalize our methodology to handle more complex ordinary differential equations in future studies.

We fit our parameters to experimental data using parameter identification methods. Parameter identification utilizes optimization methods to find the best parameter values. Our data is given at discrete times \( t_i \in \{2, 4, \ldots, 14\} \) and five experiments were run. We denote each population data point by \( y_{ie} \) corresponding to data taken at time \( t_i \) and from experiment \( e \). The parameter values that we obtain are usually influenced by our choice of an objective function. We consider two different objective functions for this study. The first seeks to minimize the sum of squares of the difference between our model
at parameter \( q \) and data collected over several experiments

\[
f(q) = \frac{1}{2} \sum_{e=1}^{\text{#experiments}} \sum_{i=1}^{\text{#time samples}} (y(t_i; q) - y^e_i)^2.
\]

The second sum in \( f^1 \) above is the square of the vector 2-norm where the vector consists of the solution at various times. Thus, we introduce the notation

\[
y(q) = [y(t_1; q), y(t_2; q), \ldots, y(t_{\text{#time samples}}; q)]^T
\]

and

\[
y^e = [y^e_1, y^e_2, \ldots, y^e_{\text{#time samples}}]^T.
\]

We also considered two natural variations of the functions above. The first variation attempts to minimize the “worst-case” fit to the data. Thus, we introduce

\[
f_1^1(q) = \max_e \|y(q) - y^e\|_1
\]

\[
f_1^2(q) = \max_e \|y(q) - y^e\|_2
\]

\[
f_1^\infty(q) = \max_e \|y(q) - y^e\|_\infty.
\]

The second variation attempts to minimize the average discrepancy between the model and the data sets.

\[
f_2^1(q) = \sum_{e=1}^{\text{#experiments}} \frac{\|y(q) - y^e\|_1}{\text{#experiments}}
\]

\[
f_2^2(q) = \sum_{e=1}^{\text{#experiments}} \frac{\|y(q) - y^e\|_2}{\text{#experiments}}
\]

\[
f_2^\infty(q) = \sum_{e=1}^{\text{#experiments}} \frac{\|y(q) - y^e\|_\infty}{\text{#experiments}}.
\]

1.1 Sensitivity Analysis

In order to better understand the problem, the sensitivity of our solution to each parameter - that is, how much the solution changes compared to small
changes in the parameters - was determined and plotted. This information gives an indication of which parameters should be computed very accurately. This required the following system of differential equations to be solved:

\[ \dot{y} = q_1 \left(1 - \frac{y}{q_2}\right)y \quad y(0) = q_3 \quad (1) \]

\[ \dot{S}_1 = (q_1 - \frac{2q_1y}{q_2})S_1 + (1 - \frac{y}{q_2})y \quad S_1(0) = 0 \quad (2) \]

\[ \dot{S}_2 = (q_1 - \frac{2q_1y}{q_2})S_2 + \frac{q_1y^2}{q_2} \quad S_2(0) = 0 \quad (3) \]

\[ \dot{S}_3 = (q_1 - \frac{2q_1y}{q_2})S_3 \quad S_3(0) = 1 \quad (4) \]

With \( S_1, S_2, S_3 \) representing the sensitivity with respect to \( q_1, q_2, q_3 \) respectively, in other words \( S_1 = \frac{\partial y}{\partial q_1}, S_2 = \frac{\partial y}{\partial q_2}, S_3 = \frac{\partial y}{\partial q_3} \) and \( \dot{S}_1 \) denotes \( \frac{\partial S_1}{\partial t} \).

Using the ODE45 solver in MATLAB to solve this system with \( q_1 = 0.8055, q_2 = 1.9778, q_3 = 0.4720 \) results in the following plot:

![Figure 1: Scaled Sensitivity over Time](image)

Figure 1 displays the scaled sensitivities over time, that is, for example, \( S_1(t, y, q) \cdot q_1 \). These results are expected based on our knowledge of the logistic growth function. The parameter \( q_1 \) controls the rate at which the
population approaches its long-term value from the initial condition, therefore it is logical that this parameter’s significance increases initially and then becomes less significant as time goes on. Similarly, we know that the population will reach its long-term value regardless of initial condition, therefore $q_3$ will have minimal importance in the long-term behavior of the function but will play a significant role in the first several data points. Since $q_2$ controls the long-term value of the function, it will be less significant toward the beginning and become the dominating parameter towards the end.

2 Methods

2.1 “Brute Force” Search

2.1.1 Method Background

The “brute force” method is simply testing every single possible combination and selecting the best combination, that is, for each choice of $q_i$ the differential equation is solved using ODE45. This solution is then compared to the data using cost function $f_1$ or $f_2$. Since it is not possible to test every possible value on a continuous interval, the interval must be discretized into a finite set of points. Figure 2 demonstrates the grid pattern formed by these finite steps overlaid on an example contour plot:

![Figure 2: Grid Pattern Formed From Discretization](image)
The method implemented here involves a fixed step size between a chosen minimum and maximum value of each parameter. Other implementations can involve a variable step size. This method is extremely time intensive, requiring \( c_{x_1} \times c_{x_2} \times \ldots \times c_{x_n} \) function evaluations where \( c \) is the number of possible choices for variable \( x_n \).

### 2.1.2 Implementation

Implementing the brute force search requires three considerations: how to handle multiple data sets, the cost function used, and the limits or step size.

In the early implementation, the data sets were averaged in order to have a single set to compare against. However, in an effort to increase accuracy, all five data sets were included. Three different norms were used to compare the simulated data and the experimental data: the 1-norm, the 2-norm, and the infinity-norm. Once these were computed for the given iteration, the cost function still had to be determined. As discussed in Section 1, two cost functions were constructed. The first approach attempts to minimize the “worse-case” fit to the data, functions \( f_1^1, f_2^1, f_\infty^1 \), while the second approach attempts to minimize the average of norm values. The code for both of these approaches has been included in Appendix A. In the interest of time, the simulation was run several times, using the results of previous runs to fine-tune the upper and lower bounds on \( q \) as well as the step size. The final parameters used were: \( 0.77 \leq q_1 \leq 0.86, 1.95 \leq q_2 \leq 2.00, 0.44 \leq q_3 \leq 0.49 \) with a step size of 0.001 with all three parameters.
2.1.3 Results

Parameter Values:

<table>
<thead>
<tr>
<th>$f^1$</th>
<th>$q_1$</th>
<th>0.8230</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9700</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4900</td>
</tr>
<tr>
<td>$f^2$</td>
<td>$q_1$</td>
<td>0.8420</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9800</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f^1$</th>
<th>$q_1$</th>
<th>0.8590</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9650</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4650</td>
</tr>
<tr>
<td>$f^2$</td>
<td>$q_1$</td>
<td>0.8090</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9820</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4650</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f^1$</th>
<th>$q_1$</th>
<th>0.7700</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9840</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4900</td>
</tr>
<tr>
<td>$f^2$</td>
<td>$q_1$</td>
<td>0.7700</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9850</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4900</td>
</tr>
</tbody>
</table>

Total Time Elapsed:

<table>
<thead>
<tr>
<th>$f^1$</th>
<th>1530.63 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f^2$</td>
<td>1484.64 seconds</td>
</tr>
</tbody>
</table>

Function Evaluations:\(^1\)

<table>
<thead>
<tr>
<th>$f^1$</th>
<th>236691</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f^2$</td>
<td>236691</td>
</tr>
</tbody>
</table>

2.1.4 Analysis

The brute force search is an extremely inefficient method, expensive in time and function evaluations. In addition, there is a substantial speed\accuracy trade-off. This is because the accuracy is limited to the step sizes used, i.e. a smaller step size is more accurate. However, a smaller step size requires more computations. When implemented with a sufficiently small step size, one can be reasonably confident with the produced answer, stemming from the fact that every possible alternative has been tried.

This method can potentially be useful if derivative information is not available since the method requires only function evaluations. When used

\(^1\)Includes the number of ODE solves and a vector norm.
with slightly coarser step-sizes, this method can also be useful to estimate the general area of the solution and provide a starting point for future methods. 

Summary:

<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only Function Evaluations</td>
<td>Time-Intensive</td>
</tr>
<tr>
<td>Can Show Overall Behavior</td>
<td>Requires Many Function Evaluations</td>
</tr>
<tr>
<td></td>
<td>Speed\Accuracy Tradeoff</td>
</tr>
</tbody>
</table>

2.2 Nelder-Mead Simplex Method

2.2.1 Method Background

The first true optimization algorithm used was an unconstrained optimization technique known as the Nelder-Mead simplex method as implemented in MATLAB’s `fminsearch`. The Nelder-Mead method uses what are called simplices, the simplest polytopes formed by \( n + 1 \) vertices in \( n \)-dimensional space. In one dimension, this is a line segment; in two dimensions, a triangle, etc. The function values are then calculated at the vertices of the simplex and the worst point is replaced by a point generated by reflecting across the centroid (the center of mass if the material is of uniform density) as seen in Figure 3. The function value for this reflected point is then compared to the previous best point. If the reflected point is better, then the simplex is expanded towards the reflected point, otherwise the simplex contracts towards the best point. This process continues until the diameter of the simplex is smaller than a specified tolerance value.

Figure 3: Simplex formed from vertices \( x_1-3 \) and reflected over centroid \( c \) to derive reflected point \( x_r \).
This method is a relatively popular one because it does not require any derivative information, all steps are computed using only function evaluations. Therefore, for functions whose function evaluations are significantly cheaper computationally than their derivatives, this makes a good choice. However, the Nelder-Mead method is a type of greedy algorithm, meaning that it takes the optimal solution for that particular step without regard to a global perspective or future subproblems. This means that the Nelder-Mead method is not guaranteed to find a global extrema and can often become stuck in local extrema. Additionally, convergence can be slow around local maxima due to small steps bouncing around the extremum.

2.2.2 Implementation

Implementing the Nelder-Mead method was relatively simple given the fact that the algorithm was already implemented as the MATLAB function \texttt{fminsearch}. \texttt{fminsearch} requires a cost function that returns a scalar and a starting point. 6 different cost functions were used to compare the findings of \texttt{fminsearch} to the values generated from our brute force search. These methods correspond to the same cost functions used in Section 2.1.2. The code has been included in Appendix B.

2.2.3 Results

Parameter Values:

<table>
<thead>
<tr>
<th></th>
<th>$f^1$</th>
<th></th>
<th>$f^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1^1$</td>
<td>$q_1$</td>
<td>0.8085</td>
<td>$q_1$</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9736</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4999</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$f_1^2$</td>
<td>$q_1$</td>
<td>0.7856</td>
<td>$q_1$</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9666</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.5145</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$f_2^1$</td>
<td>$q_1$</td>
<td>0.8236</td>
<td>$q_1$</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9753</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4658</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

Total Time Elapsed:
2.2.4 Analysis

The Nelder-Mead method is significantly more efficient than the brute force method, requiring almost a thousand times less function evaluations. Similar to the brute force method, Nelder-Mead does not require any derivative information. This makes the method very useful if such information is not available or prohibitively expensive.

However, the Nelder-Mead method does not have global convergence. Since it chooses the optimal solution for each step, it can become stuck on local minima. In this case, it is necessary to restart the search using a different initial guess. It is also necessary to have some information about the function as to place a reasonable initial guess in order to avoid the algorithm being trapped by local minima.

Summary:

<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only Function Evaluations</td>
<td>Greedy Algorithm</td>
</tr>
<tr>
<td>Relatively Fast For Many Functions</td>
<td>Can Get Stuck At Local Minima</td>
</tr>
<tr>
<td></td>
<td>Requires Initial Guess</td>
</tr>
</tbody>
</table>

2.3 Trust Region Method

2.3.1 Method Background

The trust region method works by constructing a quadratic model for the area within a given radius or trust region. The minimum of this model is then calculated and the reduction in cost function is compared to an expected reduction given by the following equation:

\[
\rho = \frac{f(x_k) - f(x_{k+1})}{g(x_k) - g(x_{k+1})}
\]
\[ f(x_k) - f(x_{k+1}) \] represents the actual reduction in the cost function where as 
\[ g(x_k) - g(x_{k+1}) \] represents the predicted reduction by the quadratic model. If 
\( \rho \) is close to one (for example: \( \rho > 0.9 \)), then the model is good and region is 
expanded. If \( \rho \) is sufficiently small (for example: \( \rho < 0.1 \)), then the model is 
a poor predictor and the region shrinks and the initial point does not move. However, if \( \rho \) falls in between these values (i.e. \( 0.1 \leq \rho \leq 0.9 \)), then the 
model’s performance is adequate and the point is accepted but the region is 
neither expanded nor contracted. Convergence occurs when the norm of the 
step or the trust region radius fall below specified tolerance values.

### 2.3.2 Implementation

The trust region method was implemented using the Entrust Optimization 
Suite, which is available at [http://people.sc.fsu.edu/~burkardt/m_src/entrust/entrust.html](http://people.sc.fsu.edu/~burkardt/m_src/entrust/entrust.html). This implementation of the trust region method incorporates an additional step. In order to minimize the quadratic model, Entrust computes the point given by the Newton step as well as the Cauchy (steepest decent). The software then computes a curve known as the Powell dog-leg between the point given by the Newton step and the Cauchy step. The intersection between the trust region and the dog-leg curve is then chosen as the minimal point for the model and that is then compared against expected reductions as shown above. This allows the algorithm to adaptively choose between the Newton and Cauchy steps in order to reduce error and converge more quickly. Figure 4 demonstrates the Newton and Cauchy steps and shows a series of points along the dog-leg curve.
Figure 4: Level curves of the quadratic model as well as the points given by the Newton and Cauchy steps and the Powell dog-leg curve between them.

Entrust was run using the following parameters:

```matlab
options.globalization = 'trust_region';
options.x_lower = [0;0;0];
options.max_iterations = 300;
options.max_fevals = 100;
```

The code used to call Entrust can be found in Appendix C.
2.3.3 Results

Parameter Values:

\[
\begin{array}{c|c}
    q_1 & 0.8055 \\
    q_2 & 1.9778 \\
    q_3 & 0.4720 \\
\end{array}
\]

Total Time Elapsed:

\[
\begin{array}{c|c}
    Entrust & 0.28 \text{ seconds} \\
\end{array}
\]

Function Evaluations:

\[
\begin{array}{|c|c|}
    \hline
    \text{Entrust} & 17 \\
    \hline
\end{array}
\]

2.3.4 Analysis:

The Entrust Trust Region method is the most efficient technique used thus far, both in terms of computational time and function evaluations. This is not surprising given the fact that the Trust Region method makes use of higher order information that other methods neglect. Therefore, Trust Region method depends on having derivative information available and not prohibitively expensive to calculate. While it can potentially be trapped by local minima, the Trust Region method shows some flexibility with large or expanding trust radius to step out of local minima.

Summary:

\[
\begin{array}{|l|l|}
    \hline
    \text{Pros:} & \text{Cons:} \\
    \text{Fast Convergence} & \text{Requires Higher Order Information} \\
    \text{Can Step Out of Local Minima} & \text{Requires Initial Guess} \\
    \text{Requires Fewer Function/Gradient Evaluations} & \\
    \hline
\end{array}
\]

2.4 Gauss-Newton Method

2.4.1 Method Background

The Gauss-Newton method is a method specifically designed for a type of problem known as non-linear least squares problems, that is, a sum of squared
function values. The specific structure given by problems of this form allow one to avoid the computationally expensive second derivative values, or to obtain more accurate approximations.

For data fitting, these problems usually take the form:

\[ r_i(x) = y_i - g(x) \]

with \( r_i \) representing the residual, the difference between the predicted values and the data for parameter vector \( x \) and data set \( i \). The function \( g(x) \) is the data values predicted by the model. In addition to the residual, it is necessary to calculate the Jacobian matrix of \( r \), that is, the matrix of first-order partial derivatives given by:

\[
J = 
\begin{bmatrix}
\frac{\partial r_1}{\partial x_1} & \frac{\partial r_1}{\partial x_2} & \cdots & \frac{\partial r_1}{\partial x_n} \\
\frac{\partial r_2}{\partial x_1} & \frac{\partial r_2}{\partial x_2} & \cdots & \frac{\partial r_2}{\partial x_n} \\
\vdots & \vdots & & \vdots \\
\frac{\partial r_n}{\partial x_1} & \frac{\partial r_n}{\partial x_2} & \cdots & \frac{\partial r_n}{\partial x_n}
\end{bmatrix}
\]

The residual and Jacobian are then used to calculate the cost function, \( f(x) \) to be minimized and its gradient, \( \nabla f \):

\[
f(x) = 0.5 \cdot r^T \cdot r \\
\nabla f = J^T \cdot r
\]

With T representing the transpose of the given matrix or vector. The value of the cost function is used to compare the success of a given iteration in minimization while the norm of the gradient is often one of several stopping conditions. The actual method is an iterative process where

\[
x_{n+1} = x + s
\]

And \( s \) is the step given by the following normal equation:

\[
(J^T J)s = -J^T \cdot r
\]

This process continues until the norm of \( s, ||s|| \), the norm of the gradient, \( ||\nabla f|| \), fall below a predefined tolerance value or the function hits a max iteration or function evaluation.
2.4.2 Implementation

When considering how to implement this method, two different choices for a cost function arose. Given that there are five data sets, the residual vector has to incorporate some method of including all the data. The first method was to create a 35x1 residual vector with the residuals for each data set appended to the end of the previous set. The second method was to perform an additional summation of squared terms on the residuals, effectively transforming the function into a quartic function. The first method allowed for the Jacobian to be contracted out of the sensitivities of each parameter at that point, while the Jacobian for the second method required the sensitivities to be multiplied by $2 \cdot r_i(x)$ due to the squaring of the residuals. While this second method results in a much smaller Jacobian, it has the drawback of flattening the valley around the minimum. Both methods are available in Appendix D.

2.4.3 Results

Parameter Values:

<table>
<thead>
<tr>
<th></th>
<th>Gauss-Newton Method</th>
<th>Quadratic Method</th>
<th>Quartic Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$q_1$</td>
<td>0.8055</td>
<td>0.7926</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>1.9778</td>
<td>1.9811</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>0.4720</td>
<td>0.4811</td>
</tr>
</tbody>
</table>

Total Time Elapsed:

<table>
<thead>
<tr>
<th></th>
<th>Quadratic</th>
<th>0.07 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quartic</td>
<td>4.03 seconds</td>
</tr>
</tbody>
</table>

Function Evaluations:

<table>
<thead>
<tr>
<th></th>
<th>Quadratic</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quartic</td>
<td>113</td>
</tr>
</tbody>
</table>
2.4.4 Analysis

The Gauss-Newton method is very efficient and popular method for solving non-linear least squares problems. For well-formed problems, this method offers rapid convergence with few function evaluations. However, the method did not work as well when used with a function that is very flat in the neighborhood of the minimum. This sort of behavior suggests that the method can be caught in minima or simply bounce around a flat area.

Summary:

<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Fast Convergence</td>
<td>Requires Higher Order Information</td>
</tr>
<tr>
<td>Requires Few Function\Gradient Evaluations</td>
<td>Requires Initial Guess</td>
</tr>
<tr>
<td></td>
<td>Slow Convergence in Flat Areas</td>
</tr>
</tbody>
</table>

2.5 Levenberg-Marquardt Method

2.5.1 Method Background

The Levenberg-Marquardt method is an expansion upon the Gauss-Newton method and is also used with non-linear least squares. It uses the same iterative step process, however has a slightly different form of the normal equation as shown in Equation 5

\[
(J^T J + \lambda I)s = -J^T \cdot r
\]  

(5)

The lambda term is known as the damping factor. When \( \lambda \) is small, the method follows the Gauss-Newton method, however when \( \lambda \) is large, it tends towards the steepest decent direction. The damping factor is changed adaptively over iterations of the method. This allows for the steepest decent direction to take precedence when far from the minimum and the quicker convergence of the Gauss-Newton method to take control when close to the minimum.

2.5.2 Implementation

This implementation uses a method very similar to the trust region method for deciding on expanding or contracting the trust radius, calculating \( \rho \) and
growing or shrinking the damping factor based on the ”gain factor”

$$\rho = \frac{f(x) - f(x + s)}{0.5 \cdot s^T (\lambda s - f'(x))}$$

$\lambda$ is then updated according to the following code:

```plaintext
if $\rho > 0$
    $x = x + s$;
    $\lambda = \lambda \cdot \max\left(\frac{1}{\gamma}, 1 - (\beta - 1)(2\rho - 1)^p\right)$;
    $v = \beta$;
else
    $\lambda = \lambda \cdot v$;
    $v = 2v$;
```

With $\beta = v = 2, \gamma = 3, p = 3$.  
This code is available in Appendix E and uses the GN_Residual method from Appendix D.

### 2.5.3 Results

#### Parameter Values:

<table>
<thead>
<tr>
<th>$q_1$</th>
<th>0.8055</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_2$</td>
<td>1.9778</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0.4720</td>
</tr>
</tbody>
</table>

#### Total Time Elapsed:

| Levenberg-Marquardt | 0.13 seconds |

#### Function Evaluations:

| Levenberg-Marquardt | 9 |

---


3For more discussion of this choice, see Footnote 2.
2.5.4 Analysis

This method was a fairly quick one, however slightly slower than the Gauss-Newton method and requiring more function evaluations due to the damping term. However, this is to be expected. The Gauss-Newton method converges very rapidly near the minimum, while the Levenberg-Marquardt method is a much more robust method and able to quickly converge even when the guess is not an accurate one.

Summary:

<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robust</td>
<td>Requires Initial Guess</td>
</tr>
<tr>
<td>Requires Few Function\Gradient Evaluations</td>
<td>Slightly Slower Convergence Near Minimum</td>
</tr>
<tr>
<td>When Starting Away from Minimum</td>
<td></td>
</tr>
<tr>
<td>Switches Between Optimal Methods</td>
<td></td>
</tr>
</tbody>
</table>
Appendix

A  “Brute Force” Search Code

A.1 Largest Norms

function [FinalNorm1 FinalNorm2 FinalNormInf] = odeProbLargestNormData

tic

% Step Sizes
q1Step = 0.001;
q2Step = 0.001;
q3Step = 0.001;

% Values Over Which to Search, min : qxStep : max
q1 = 0.77 : q1Step : 0.86;
q2 = 1.95 : q2Step : 2.00;
q3 = 0.44: q3Step : 0.49;

% Pre-allocate Matricies For Speed
X = 0 : 0.1 : 14;
Norm1.cost = zeros(1, numel(q1));
Norm2.cost = zeros(1, numel(q1));
NormInf.cost = zeros(1, numel(q1));
loop.Norm1 = zeros(1, numel(q2));
loop.Norm2 = zeros(1, numel(q2));
loop.NormInf = zeros(1, numel(q2));
Norm1.q2index = zeros(1, numel(q1));
Norm2.q2index = zeros(1, numel(q1));
NormInf.q2index = zeros(1, numel(q1));
q3Loop.Norm1 = zeros(1, numel(q3));
q3Loop.Norm2 = zeros(1, numel(q3));
q3Loop.NormInf = zeros(1, numel(q3));
indexq3.LoopNorm1 = zeros(1, numel(q2));
indexq3.LoopNorm2 = zeros(1, numel(q2));
indexq3.LoopNormInf = zeros(1, numel(q2));
Norm1.q3index = zeros(1, numel(q1));
Norm2.q3index = zeros(1, numel(q1));
NormInf.q3index = zeros(1, numel(q1));

% Main Loop
for i = 1:numel(q1)
    for j = 1:numel(q2)
        for k = 1:numel(q3)
            Result = ode45(@(t,y)q1(i) * y - (q1(i) / q2(j)) * y^2,X,q3(k));

            % Calculate Differences
            for n = 1:5
                Difference.set(:,n) = (Result - Data.set(:,n));
            end
        end
    end
end

% Calculate Differences
for n = 1:5
    Difference.set(:,n) = (Result - Data.set(:,n));
end

end
% Take the Norms of Differences and Average
for n = 1:5
    Norm.one(n) = norm(Difference.set(:,n),1);
    Norm.two(n) = norm(Difference.set(:,n),2);
    Norm.inf(n) = norm(Difference.set(:,n),inf);
end
q3Loop.Norm1(k) = max(Norm.one);
q3Loop.Norm2(k) = max(Norm.two);
q3Loop.NormInf(k) = max(Norm.inf);
end
% Store Cost and Value of Optimal q3
[loop.Norm1(j), indexq3Loop.Norm1(j)] = min(q3Loop.Norm1);
[loop.Norm2(j), indexq3Loop.Norm2(j)] = min(q3Loop.Norm2);
[loop.NormInf(j), indexq3Loop.NormInf(j)] = min(q3Loop.NormInf);
end
% Store Cost and Value of Optimal q2 and Its Associated q3
[Norm1.cost(i), Norm1.q2index(i)] = min(loop.Norm1);
Norm1.q3index(i) = indexq3Loop.Norm1(Norm1.q2index(i));
[Norm2.cost(i), Norm2.q2index(i)] = min(loop.Norm2);
Norm2.q3index(i) = indexq3Loop.Norm2(Norm2.q2index(i));
[NormInf.cost(i), NormInf.q2index(i)] = min(loop.NormInf);
NormInf.q3index(i) = indexq3Loop.NormInf(NormInf.q2index(i));
end
% Determine Optimal q1 and Its Associated q2 and q3 Values
[C1, I1] = min(Norm1.cost);
FinalNorm1(1) = (I1 - 1) * q1Step + q1(1);
FinalNorm1(2) = (Norm1.q2index(I1) - 1) * q2Step + q2(1);
FinalNorm1(3) = (Norm1.q3index(I1) - 1) * q3Step + q3(1);
[C2, I2] = min(Norm2.cost);
FinalNorm2(1) = (I2 - 1) * q1Step + q1(1);
FinalNorm2(2) = (Norm1.q2index(I2) - 1) * q2Step + q2(1);
FinalNorm2(3) = (Norm1.q3index(I2) - 1) * q3Step + q3(1);
[CInf, IInf] = min(NormInf.cost);
FinalNormInf(1) = (IInf - 1) * q1Step + q1(1);
FinalNormInf(2) = (Norm1.q2index(IInf) - 1) * q2Step + q2(1);
FinalNormInf(3) = (Norm1.q3index(IInf) - 1) * q3Step + q3(1);

toc
end

A.2 Average Norms

function [FinalNorm1 FinalNorm2 FinalNormInf] = odeProbAvgNormData
tic
% Step Sizes
q1Step = 0.001;
q2Step = 0.001;
q3Step = 0.001;
% Values Over Which to Search, min : qxStep : max
q1 = 0.77 : q1Step : 0.86;
q2 = 1.95; q2Step : 2.00;
q3 = 0.44; q3Step : 0.49;

%Pre-allocate Matrices For Speed
X = 0 : 0.1 : 14;
Norm1.cost = zeros(1, numel(q1) );
Norm2.cost = zeros(1, numel(q1) );
NormInf.cost = zeros(1, numel(q1) );
loop.Norm1 = zeros(1, numel(q2) );
loop.Norm2 = zeros(1, numel(q2) );
loop.NormInf = zeros(1, numel(q2) );
Norm1.q2index = zeros(1, numel(q1) );
Norm2.q2index = zeros(1, numel(q1) );
NormInf.q2index = zeros(1, numel(q1) );
q3Loop.Norm1 = zeros(1, numel(q3) );
q3Loop.Norm2 = zeros(1, numel(q3) );
q3Loop.NormInf = zeros(1, numel(q3) );
indexq3.LoopNorm1 = zeros(1, numel(q2));
indexq3.LoopNorm2 = zeros(1, numel(q2));
indexq3.LoopNormInf = zeros(1, numel(q2));
Norm1.q3index = zeros(1, numel(q1));
Norm2.q3index = zeros(1, numel(q1));
NormInf.q3index = zeros(1, numel(q1));

%Main Loop
for i = 1:numel(q1)
    for j = 1:numel(q2)
        for k = 1:numel(q3)
            [T,Y] = ode45(@(t,y)q1(i) * y - (q1(i) / q2(j)) * y^2,X,q3(k));
            Result = Y(1:20:141);
            Data.set(:,1) = [q3(k) 1.2001 1.774 1.9639 1.9725 1.9778 1.985 1.9328];
            Data.set(:,2) = [q3(k) 1.1915 1.7596 1.9399 1.9535 1.9687 2.022 1.959];
            Data.set(:,3) = [q3(k) 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
            Data.set(:,4) = [q3(k) 1.2371 1.7685 1.8944 1.9720 1.9373 1.8897 2.0558];
            Data.set(:,5) = [q3(k) 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];
            %Calculate Differences
            for n = 1:5
                Difference.set(:,n) = (Result - Data.set(:,n));
            end
            %Take the Norms of Differences and Average
            for n = 1:5
                Norm.one(n) = norm(Difference.set(:,n),1);
                Norm.two(n) = norm(Difference.set(:,n),2);
                Norm.inf(n) = norm(Difference.set(:,n),inf);
            end
            q3Loop.Norm1(k) = mean(Norm.one);
            q3Loop.Norm2(k) = mean(Norm.two);
            q3Loop.NormInf(k) = mean(Norm.inf);
        end
    end
end

%Store Cost and Value of Optimal q3
[loop.Norm1(j), indexq3Loop.Norm1(j)] = min(q3Loop.Norm1);
[loop.Norm2(j), indexq3Loop.Norm2(j)] = min(q3Loop.Norm2);
[loop.NormInf(j), indexq3Loop.NormInf(j)] = min(q3Loop.NormInf);

%Store Cost and Value of Optimal q2 and Its Associated q3
%Determine Optimal q1 and Its Associated q2 and q3 Values
[C1, I1] = min(Norm1.cost);
FinalNorm1(1) = (I1 - 1) * q1Step + q1(1);
FinalNorm1(2) = (Norm1.q2index(I1) - 1) * q2Step + q2(1);
FinalNorm1(3) = (Norm1.q3index(I1) - 1) * q3Step + q3(1);

[C2, I2] = min(Norm2.cost);
FinalNorm2(1) = (I2 - 1) * q1Step + q1(1);
FinalNorm2(2) = (Norm2.q2index(I2) - 1) * q2Step + q2(1);
FinalNorm2(3) = (Norm2.q3index(I2) - 1) * q3Step + q3(1);

[CInf, IInf] = min(NormInf.cost);
FinalNormInf(1) = (IInf - 1) * q1Step + q1(1);
FinalNormInf(2) = (NormInf.q2index(IInf) - 1) * q2Step + q2(1);
FinalNormInf(3) = (NormInf.q3index(IInf) - 1) * q3Step + q3(1);

toc
end

B Nelder-Mead Method

function [AveNorm LargestNorm AveEval LargeEval] = odeProbFMinSearch
tic
AveEval = 0;
LargeEval = 0;

%Perform Nelder-Mead Searches and Return Optimal Q-Values and Function Evaluations
%AveNorm = [AveNorm.norm(1,:), fval, exitflag, aveoutput.norm(1)] = fminsearch(@(x) ... aveNorms1(x(1), x(2), x(3)), [0.8 1.97 0.47]);
%AveNorm.norm(2,:), fval, exitflag, aveoutput.norm(2)] = fminsearch(@(x) ... aveNorms2(x(1), x(2), x(3)), [0.8 1.97 0.47]);
%AveNorm.norm(3,:), fval, exitflag, aveoutput.norm(3)] = fminsearch(@(x) ... aveNormsInf(x(1), x(2), x(3)), [0.8 1.97 0.47]);

[LargestNorm.norm(1,:), fval, exitflag, largeoutput.norm(1)] = fminsearch(@(x) ... LargestNorms1(x(1), x(2), x(3)), [0.8 1.97 0.47]);
[LargestNorm.norm(2,:), fval, exitflag, largeoutput.norm(2)] = fminsearch(@(x) ... LargestNorms2(x(1), x(2), x(3)), [0.8 1.97 0.47]);
[LargestNorm.norm(3,:), fval, exitflag, largeoutput.norm(3)] = fminsearch(@(x) ... LargestNormInf(x(1), x(2), x(3)), [0.8 1.97 0.47]);

%Total the Number of Function Evaluations Per Method
for i = 1:3
    AveEval = AveEval + aveoutput.norm(i).funcCount;
    LargeEval = LargeEval + largeoutput.norm(i).funcCount;
end
toc
function N1 = aveNorms1(q1, q2, q3)
    Norm = zeros(1,5);
    %Given Data Set
    Data.set(:,1) = [q3 1.2001 1.774 1.9726 1.9778 1.985 1.9328];
    Data.set(:,2) = [q3 1.1915 1.7596 1.9399 1.9536 1.9667 2.022 1.969];
    Data.set(:,3) = [q3 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
    Data.set(:,4) = [q3 1.2371 1.7885 1.8944 1.972 1.9373 1.8897 2.0558];
    Data.set(:,5) = [q3 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];
    %Compute Function Values for Supplied q1,q2,q3
    [T,Result] = ode45(@(t,y)q1 * y - (q1 / q2) * y^2,[0:2:14],q3);
    %Calculate Differences
    for i = 1:5
        Difference.set(:,i) = (Result - Data.set(:,i));
    end
    %Take the 1-Norm of Differences and Average
    for i = 1:5
        Norm(i) = norm(Difference.set(:,i),1);
    end
    N1 = mean(Norm);
end

function N2 = aveNorms2(q1, q2, q3)
    Norm = zeros(1,5);
    %Given Data Set
    Data.set(:,1) = [q3 1.2001 1.774 1.9726 1.9778 1.985 1.9328];
    Data.set(:,2) = [q3 1.1915 1.7596 1.9399 1.9536 1.9667 2.022 1.969];
    Data.set(:,3) = [q3 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
    Data.set(:,4) = [q3 1.2371 1.7885 1.8944 1.972 1.9373 1.8897 2.0558];
    Data.set(:,5) = [q3 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];
    %Compute Function Values for Supplied q1,q2,q3
    [T,Result] = ode45(@(t,y)q1 * y - (q1 / q2) * y^2,[0:2:14],q3);
    %Calculate Differences
    for i = 1:5
        Difference.set(:,i) = (Result - Data.set(:,i));
    end
    %Take the 2-Norm of Differences and Average
    for i = 1:5
        Norm(i) = norm(Difference.set(:,i),2);
    end
    N2 = mean(Norm);
end

function NInf = aveNormsInf(q1, q2, q3)
    Norm = zeros(1,5);
    %Given Data Set
    Data.set(:,1) = [q3 1.2001 1.774 1.9726 1.9778 1.985 1.9328];
    Data.set(:,2) = [q3 1.1915 1.7596 1.9399 1.9536 1.9667 2.022 1.969];
    Data.set(:,3) = [q3 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
    Data.set(:,4) = [q3 1.2371 1.7885 1.8944 1.972 1.9373 1.8897 2.0558];
    Data.set(:,5) = [q3 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];
    %Compute Function Values for Supplied q1,q2,q3
    [T,Result] = ode45(@(t,y)q1 * y - (q1 / q2) * y^2,[0:2:14],q3);
    %Calculate Differences
    for i = 1:5
        Difference.set(:,i) = (Result - Data.set(:,i));
    end
    %Take the Inf-Norm of Differences and Average
    for i = 1:5
        Norm(i) = norm(Difference.set(:,i),Inf);
    end
    NInf = mean(Norm);
end
Compute Function Values for Supplied \( q_1, q_2, q_3 \)
\[
[T, \text{Result}] = \text{ode45}(\@t,y) q_1 \cdot y - (q_1 / q_2) \cdot y^2, [0:2:14], q_3);
\]

%Calculate Differences
for \( i = 1:5 \)
    \text{Difference.set}(i,:,:) = (\text{Result} - \text{Data.set}(i,:));
end

%Take the Infinity-Norm of Differences and Average
for \( i = 1:5 \)
    \text{Norm}(i) = \text{norm}(\text{Difference.set}(i,:), \infty);
end
\text{NInf} = \text{mean} (\text{Norm});
end

function \( N_1 = \text{LargestNorms1}(q_1, q_2, q_3) \)
\text{Norm} = \text{zeros}(1,5);

%Given Data Set
\text{Data.set}(1,:,:) = [q_3 1.2001 1.774 1.9639 1.9726 1.9778 1.985 1.9328];
\text{Data.set}(2,:,:) = [q_3 1.1915 1.7596 1.9399 1.9535 1.9687 2.022 1.959];
\text{Data.set}(3,:,:) = [q_3 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
\text{Data.set}(4,:,:) = [q_3 1.2371 1.7685 1.8944 1.972 1.9373 1.8897 2.0588];
\text{Data.set}(5,:,:) = [q_3 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];

%Compute Function Values for Supplied \( q_1, q_2, q_3 \)
\[
[T, \text{Result}] = \text{ode45}(\@t,y) q_1 \cdot y - (q_1 / q_2) \cdot y^2, [0:2:14], q_3);
\]

%Calculate Differences
for \( i = 1:5 \)
    \text{Difference.set}(i,:,:) = (\text{Result} - \text{Data.set}(i,:));
end

%Take the 1-Norm of Differences and Return the Maximum
for \( i = 1:5 \)
    \text{Norm}(i) = \text{norm}(\text{Difference.set}(i,:), 1);
end
\text{N1} = \text{max} (\text{Norm});
end

function \( N_2 = \text{LargestNorms2}(q_1, q_2, q_3) \)
\text{Norm} = \text{zeros}(1,5);

%Given Data Set
\text{Data.set}(1,:,:) = [q_3 1.2001 1.774 1.9639 1.9726 1.9778 1.985 1.9328];
\text{Data.set}(2,:,:) = [q_3 1.1915 1.7596 1.9399 1.9535 1.9687 2.022 1.959];
\text{Data.set}(3,:,:) = [q_3 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
\text{Data.set}(4,:,:) = [q_3 1.2371 1.7685 1.8944 1.972 1.9373 1.8897 2.0588];
\text{Data.set}(5,:,:) = [q_3 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];

%Compute Function Values for Supplied \( q_1, q_2, q_3 \)
\[
[T, \text{Result}] = \text{ode45}(\@t,y) q_1 \cdot y - (q_1 / q_2) \cdot y^2, [0:2:14], q_3);
\]

%Calculate Differences
for \( i = 1:5 \)
    \text{Difference.set}(i,:,:) = (\text{Result} - \text{Data.set}(i,:));
end

%Take the 2-Norm of Differences and Return the Maximum
for \( i = 1:5 \)
N(i) = norm(Difference.set(:,i),2);
end
N2 = max(Norm);
end

function NInf = LargestNormsInf(q1, q2, q3)
Norm = zeros(1,5);
%Given Data Set
Data.set(:,1) = [q3 1.2001 1.774 1.9725 1.9778 1.985 1.9328];
Data.set(:,2) = [q3 1.1915 1.7596 1.9535 1.9687 2.022 1.959];
Data.set(:,3) = [q3 1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
Data.set(:,4) = [q3 1.2371 1.7685 1.8944 1.972 1.9373 1.8897 2.0558];
Data.set(:,5) = [q3 1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];
%Compute Function Values for Supplied q1,q2,q3
[T,Result] = ode45(@(t,y)q1 * y - (q1 / q2) * y^2, [0:2:14], q3);
%Calculate Differences
for i = 1:5
    Difference.set(:,i) = (Result - Data.set(:,i));
end
%Take the Infinity-Norm of Differences and Return the Maximum
for i = 1:5
    Norm(i) = norm(Difference.set(:,i),inf);
end
NInf = max(Norm);
end

C Trust Region Method

function [Optimal] = entrustSearch()
tic
options = [];
options.verbose = 1;
options.max_iterations = 300;
options.max_fevals = 100;
options.globalization = 'trust_region';
options.x_lower = [0;0;0];
Optimal = entrust(@gradSearch, [0.8, 1.98, 0.5], options);
toc
end

function [F, gradF, H] = gradSearch(x, flag)
%Setup
x = 0 : 0.1 : 14;
F = 0;
H = [];
dx(1) = 0;
dx(2) = 0;
dx(3) = 0;
[s1 s2 s3] = Sensitivity2(x(1), x(2), x(3));
%Calculations
% Data Simulation
[T,Y] = ode45(@(t,y)x(1) * y - (x(1) / x(2)) * y^2,X,x(3));
Result = Y(21:20:141)';

Data(1).vec = [1.2001 1.774 1.9639 1.9725 1.9778 1.985 1.9328];
Data(2).vec = [1.1915 1.7596 1.9399 1.9535 1.9687 2.022 1.959];
Data(3).vec = [1.2451 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
Data(4).vec = [1.2371 1.7685 1.8944 1.972 1.9373 1.8897 2.0558];
Data(5).vec = [1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];

for i = 1:5
    for j = 1:7
        Difference(i).vec(j) = (Result(j) - Data(i).vec(j))^2; %#ok<AGROW>
    end
end

% Gradient
% dx(1)
for i = 1:5
    for j = 1:7
        dx(1) = dx(1) + 2 * (Result(j) - Data(i).vec(j)) * s1(j);
    end
end

% dx(2)
for i = 1:5
    for j = 1:7
        dx(2) = dx(2) + 2 * (Result(j) - Data(i).vec(j)) * s2(j);
    end
end

% dx(3)
for i = 1:5
    for j = 1:7
        dx(3) = dx(3) + 2 * (Result(j) - Data(i).vec(j)) * s3(j);
    end
end

% Output
% F
for i = 1:5
    F = F + sum(Difference(i).vec);
end

% Gradient of F
gradF = [dx(1); dx(2); dx(3)];

end

D Gauss-Newton Method

D.1 General Method

function x = Gauss_Newton(fname, x)
tic
    % Call the specific function and return the residual and Jacobian
function [F, J] = GN_Residual(x, flag)
% Setup
X = 0 : 0.1 : 14;
F = zeros(35,1);
J = zeros(35,3);
[s1 s2 s3] = Sensitivity2(x(1), x(2), x(3));
% Calculations
```matlab
%Data Simulation
[T,Y] = ode45(@(t,y)x(1) * y - (x(1) / x(2)) * y^2, X, x(3));
Result = Y(21:20:141)';
Data(1).vec = [1.2001 1.774 1.9639 1.9725 1.9778 1.985 1.9328];
Data(2).vec = [1.1915 1.7596 1.9399 1.9336 1.9687 2.022 1.9581];
Data(3).vec = [1.2461 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
Data(4).vec = [1.2371 1.7685 1.8944 1.972 1.9373 1.8897 2.0558];
Data(5).vec = [1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];

for i = 1:5
    for j = 1:7
        Difference(i).vec(j) = Result(j) - Data(i).vec(j); %#ok<AGROW>
    end
end

%Output

%Jacobian
for i = 1:7:29
    J((i):(6+i),1) = s1;
    J((i):(6+i),2) = s2;
    J((i):(6+i),3) = s3;
end

%F
for i = 1:5
    for j = 1:7
        F(7*i + j - 7) = Difference(i).vec(j);
    end
end

D.3 Modified Jacobian Method

function [F, J] = GN_Jacobian(x, flag)
%Setup
X = 0 : 0.1 : 14;
F = zeros(7,1);
J = zeros(7,3);
Jac = zeros(7,1);
[s1 s2 s3] = Sensitivity2(x(1), x(2), x(3));

%Calculations

%Data Simulation
[T,Y] = ode45(@(t,y)x(1) * y - (x(1) / x(2)) * y^2, X, x(3));
Result = Y(21:20:141)';
Data(1).vec = [1.2001 1.774 1.9639 1.9725 1.9778 1.985 1.9328];
Data(2).vec = [1.1915 1.7596 1.9399 1.9336 1.9687 2.022 1.9581];
Data(3).vec = [1.2461 1.6905 1.9461 1.9971 1.9463 1.9627 2.0606];
Data(4).vec = [1.2371 1.7685 1.8944 1.972 1.9373 1.8897 2.0558];
Data(5).vec = [1.1721 1.7565 1.9472 1.9863 1.9465 1.9524 1.9878];
```
for i = 1:5
    for j = 1:7
        Difference(i).vec(j) = (Result(j) - Data(i).vec(j))^2; %#ok<AGROW>
    end
end

%Jacobian Multiplier
for i = 1:7
    for j = 1:5
        Jac(i) = Jac(i) + (Result(i) - Data(j).vec(i));
    end
end

%Output
%F
for i = 1:7
    for j = 1:5
        F(i,1) = F(i,1) + Difference(j).vec(i);
    end
end

%Jacobian
J(:,1) = 2*s1.*Jac;
J(:,2) = 2*s2.*Jac;
J(:,3) = 2*s3.*Jac;
end

E Levenberg-Marquardt Method

function x = Levenberg_Marquardt(fname, x)
tic
    %Call the specific function and return the residual and Jacobian
    [r, jac] = feval(fname,x);
    f = 0.5 * (r' * r);
    g = jac' * r;
    g_norm = norm(g);
    step = zeros(length(x),1);
    step_norm = 0;
    x = reshape(x, length(x), 1);
    %Initial Settings
    lambda = 10^-5*max(max(jac'*jac));
    beta = 2;
    v = beta;
    gamma = 3;
    p = 3;
    converged = false;
    %Convergence Conditions
    min_step = 0.000000001;
    min_grad = 0.000000001;
    i = 0;
    %Print the Initial Conditions
fprintf(1, '\nInitial Conditions are:\n')
fprintf(1, '\nx:\t%6.4f	%6.4f	%6.4f
', x)
fprintf(1, 'Function Value:\t%6e
', f)
fprintf(1, 'Gradient:\t%5e	%5e	%5e
', g)
fprintf(1, 'Norm of Gradient:\t%5e
', g_norm)
fprintf(1, 'Lambda:\t%6e
', lambda)

%Optimization Loop
while (~converged)
    i = i + 1;
    step = (jac' * jac + (lambda * eye(length(x))))^-1 * -(jac' * r);
    x_p = x + step;

    %Compute new values
    [r_p, jac_p] = feval(fname, x_p);
    f_p = 0.5 * (r_p' * r_p);
    g_p = jac_p' * r_p;

    %Compute Gain Factor
    rho = (f - f_p)/(0.5*step'*(lambda*step - g_p));

    %Update lambda
    if rho > 0
        x = x_p;
        lambda = lambda * max(1/gamma, 1 - (beta - 1)*(2*rho - 1)^p);
        v = beta;
        f = f_p;
        r = r_p;
        g = g_p;
        jac = jac_p;
        g_norm = norm(g);
        step_norm = norm(step);
    fprintf(1, '\nIteration:%g
', i)
    fprintf(1, 'x:\t%6.4f	%6.4f	%6.4f
', x)
    fprintf(1, 'Step:\t%5e	%5e	%5e
', step)
    fprintf(1, 'Step Norm:\t%5e
', step_norm)
    fprintf(1, 'Function Value:\t%6e
', f)
    fprintf(1, 'Gradient:\t%5e	%5e	%5e
', g)
    fprintf(1, 'Norm of Gradient:\t%5e
', g_norm)
    fprintf(1, 'Lambda:\t%6e
', lambda)
    if (step_norm < min_step || g_norm < min_grad)
        converged = true;
    end
    else
        lambda = lambda * v; v = 2*v;
    fprintf(1, '\nIteration:%g
', i)
    end
end

toc
end