THE LEVENBERG-MARQUARDT METHOD: NONLINEAR LEAST SQUARES MINIMIZATION AND CURVE FITTING

December, 2010 Guanqing Ou and Arash Ushani

INTRODUCTION

The solution to the Nonlinear Least Squares Minimization problem is the minimum of a function that follows the form

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) \quad (1)$$

where x is a vector $\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, and r_j are functions that map x from $\mathcal{R}^n$ to $\mathcal{R}$. The r_js are referred to as residuals and $m \geq n$. The Levenberg-Marquardt algorithm provides a method of solving this problem, which can be used to fit a set of data points to a function with nonlinear parameters. The following paper discusses this problem and methods of solving it, including the gradient descent method, the Gauss-Newton method, and the Levenberg-Marquardt method. These three methods are compared based on their rate of convergence and accuracy using an example problem.

NONLINEAR LEAST SQUARES MINIMIZATION

All three methods of solving this problem make use of the gradient and Hessian of the function f. This makes sense, as these terms give us information about the behavior of the function as x changes, and thus can be used to find the values of x for which f is minimized. The following discusses how the gradient and Hessian may be expressed in a way that facilitates numerically finding this minimum.

Let us consider equation (1). If we think of all r_j s as a vector of functions: $r(x) = (r_1(x), r_2(x), \dots, r_m(x))$, then the function f can be rewritten as

$$f(x) = \frac{1}{2}\|r(x)\|^2 \quad (2)$$

This allows us to express the gradient and Hessian of f in a fairly concise manner. Recall that for any function $g(x)$, where $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, the gradient $\nabla g(x) = (\frac{\delta g}{\delta x_1}, \frac{\delta g}{\delta x_2}, \dots, \frac{\delta g}{\delta x_n})$, and the Hessian $H(g)$ is a matrix with dimensions $n \times n$ where the entry $H(g)_{ij}(x) = \frac{\delta^2 g}{\delta x_i \delta x_j}$.

First let us consider $\nabla f(x)$. In the most general form, $\nabla f(x) = \begin{bmatrix} \frac{\delta f}{\delta x_1} \\ \frac{\delta f}{\delta x_2} \\ \vdots \\ \frac{\delta f}{\delta x_n} \end{bmatrix}$. But $f(x) = \frac{1}{2}\|r(x)\|^2$, so we can express each $\frac{\delta f}{\delta x_i}$ as

$$\frac{\delta f}{\delta x_i} = \begin{bmatrix} \frac{\delta r_1}{\delta x_i} & \frac{\delta r_2}{\delta x_i} & \dots & \frac{\delta r_m}{\delta x_i} \end{bmatrix} \begin{bmatrix} \frac{\delta f}{\delta r_1} \\ \frac{\delta f}{\delta r_2} \\ \vdots \\ \frac{\delta f}{\delta r_m} \end{bmatrix} = \sum_{j=1}^{i} \frac{\delta f}{\delta r_j} \left(\frac{\delta r_j}{\delta x_i} \right) \tag{3}$$

Using this equation, we can restate ,$\nabla f(x)$ as

$$\nabla f(x) = \begin{bmatrix} \frac{\delta r_1}{\delta x_1} & \frac{\delta r_2}{\delta x_1} & \dots & \frac{\delta r_m}{\delta x_1} \\ \frac{\delta r_1}{\delta x_2} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots \\ \frac{\delta r_1}{\delta x_n} & \dots & \ddots & \frac{\delta r_m}{\delta x_n} \end{bmatrix} \begin{bmatrix} \frac{\delta f}{\delta r_1} \\ \frac{\delta f}{\delta r_2} \\ \vdots \\ \frac{\delta f}{\delta r_m} \end{bmatrix} \tag{4}$$

We can further simplify this by remembering that the Jacobian of a set of functions $r(x) = r_1(x), r_2(x), \dots, r_m(x))$ is

$$J(x) = \begin{bmatrix} \frac{\delta r_1}{\delta x_1} & \frac{\delta r_1}{\delta x_2} & \dots & \frac{\delta r_1}{\delta x_n} \\ \frac{\delta r_2}{\delta x_1} & \frac{\delta r_2}{\delta x_2} & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta r_m}{\delta x_1} & \dots & \dots & \frac{\delta r_m}{\delta x_n} \end{bmatrix} \tag{5}$$

This is the transpose of the left matrix in equation 4. The right matrix, $\begin{bmatrix} \frac{\delta f}{\delta r_1} \\ \frac{\delta f}{\delta r_2} \\ \vdots \\ \frac{\delta f}{\delta r_m} \end{bmatrix}$, is simply $r(x)$, as $f(x) = \frac{1}{2}\|r(x)\|^2$. Thus,

$$\nabla f(x) = J(x)^T r(x). \tag{6}$$

The Hessian can be found similarly. Recall each term $a_{ij} = H(f)_{ij}(x) = \frac{\delta^2 f}{\delta x_i \delta x_j}$. We already know that $\frac{\delta f}{\delta x_i} = \sum_{j=1}^{i} r(x)\left(\frac{\delta r_j}{\delta x_i}\right)$, to find $H(f)_{ij}(x)$, we simply take the partial derivative to x_j:

$$H(f)_{ij}(x) = \frac{\delta \sum_{j=1}^{i} r_j(x)\left(\frac{\delta r_j}{\delta x_i}\right)}{\delta x_j} \tag{7}$$

which, using the Chain Rule, can be expressed as

$$H(f)_{ij}(x) = \sum_{j=1}^{i} \frac{\delta r_j}{\delta x_j}\left(\frac{\delta r_j}{\delta x_i}\right) + \sum_{j=1}^{i} r_j(x)\left(\frac{\delta^2 r_j}{\delta x_i \delta x_j}\right) \tag{8}$$

If we assume that all $r_j$is linear in x or simply that the residuals $r_j(x)$ are small, the second term can be ignored. And the first term is simply the terms of the Jacobian of $r(x)$ multiplied by each other, so we can say

$$H(x) = \mathbf{J}^{\mathbf{T}}(x)\mathbf{J}(x) \tag{8}$$

With these expressions of the gradient and Hessian of $f(x)$, we can move onto a discussion of the methods used to solve the nonlinear least squares minimization problem.

COMPUTATIONAL APPROACH

Parameter and Curve Fitting

In each of the following methods, there is assumed to exist a function $\hat{y}(t; \mathbf{p})$ where t is an independent variable, and $\mathbf{p}$ is a vector of n parameters that when applied to the function y is presumed to provide a fit of the function to a set of m points (t_i, y_i). The goodness-of-fit between this parameterized function and the set of data points is determined by calculating the chi-squared error

$$\chi^2(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{m}\left[y(t_i) - \frac{\hat{y}(t_i;\mathbf{p})}{w_i}\right]^2 \tag{9}$$

w_i weights the errors, in case the user wants to optimize the function at certain points t_i. A weighting matrix $\mathbf{W}$ is defined with the diagonals equal to $\frac{1}{w_i^2}$.

The goal of any parameter/curve fitting method is to iteratively find a perturbation $\mathbf{h}$ to the vector $\mathbf{p}$ until some $\chi^2(\mathbf{p_i})$ is the minimum possible value for the points (t_i, y_i) and function $\hat{y}(t;\mathbf{p})$.

Equation 9 can be rewritten in terms of matrices:

$$\chi^2(\mathbf{p}) = \frac{1}{2}\big(y - \hat{y}(\mathbf{p})\big)^T\mathbf{W}\,(y - \hat{y}(\mathbf{p})) \tag{10}$$

which, after factoring, gives

$$\chi^2(\mathbf{p}) = \frac{1}{2}y^T\mathbf{W}y - y^T\mathbf{W}\hat{y} + \frac{1}{2}\hat{y}^T\mathbf{W}\hat{y} \tag{11}$$

The Gradient Descent Method

Gradient descent is a fairly simple method, which updates the parameter vector $\mathbf{p}$ by subtracting the scaled gradient at each step, such that

$$\mathbf{p}_{i+1} = \mathbf{p}_i - \lambda\nabla\chi^2(\mathbf{p}). \tag{12}$$

The gradient of χ^2 can be found using equation 12:

$$\begin{aligned}\frac{\delta\chi^2(\mathbf{p})}{\delta\mathbf{p}} &= \big(y - \hat{y}(\mathbf{p})\big)^T\mathbf{W}\frac{\delta}{\delta\mathbf{p}}\big(y - \hat{y}(\mathbf{p})\big) = -\big(y - \hat{y}(\mathbf{p})\big)^T\mathbf{W}\frac{\delta\hat{y}(\mathbf{p})}{\delta\mathbf{p}} \\ &= -\big(y - \hat{y}(\mathbf{p})\big)^T\mathbf{W}\mathbf{J}\end{aligned} \tag{13}$$

So the perturbation $\mathbf{h}$ is simply

$$\mathbf{h} = \lambda\big(y - \hat{y}(\mathbf{p})\big)^T\mathbf{W}\mathbf{J} \tag{14}$$

The Gauss-Newton Method

The Gauss-Newton method assumes that the function is quadratic near the optimal parameter values, and locally approximates the parameters using a first-order Taylor series expansion:

$$\hat{y}(\mathbf{p}+\mathbf{h}) \approx \hat{y}(\mathbf{p}) + \frac{\delta\hat{y}}{\delta\mathbf{p}}\mathbf{h} = \hat{y} + \mathbf{J}\mathbf{h} \tag{15}$$

This allows us to restate equation 11 as follows:

$$\chi^2(\mathbf{p}+\mathbf{h}) \approx \frac{1}{2}\mathrm{y}^{\mathrm{T}}\mathbf{W}\mathrm{y} + \hat{\mathrm{y}}^{\mathbf{T}}\mathbf{W}\hat{\mathrm{y}} - \frac{1}{2}\mathrm{y}^{\mathrm{T}}\mathbf{W}\hat{\mathrm{y}} - (\mathrm{y}-\hat{\mathrm{y}})\mathbf{W}\mathbf{J}\mathbf{h} + \frac{1}{2}\mathbf{h}^{\mathbf{T}}\mathbf{J}^{\mathbf{T}}\mathbf{W}\mathbf{J}\mathbf{h} \quad (16)$$

Since we are assuming that the behavior of the parameter vector is quadratic around the optimal value, to find the perturbation $\mathbf{h}$ that minimizes χ^2, we find where $\frac{\delta\chi^2}{\delta\mathbf{h}} = 0$:

$$\frac{\delta\chi^2(\mathbf{p}+\mathbf{h})}{\delta\mathbf{h}} = -(\mathrm{y}-\hat{\mathrm{y}})\mathbf{W}\mathbf{J} + \frac{1}{2}\mathbf{h}^{\mathbf{T}}\mathbf{J}^{\mathbf{T}}\mathbf{W}\mathbf{J} = 0 \quad (17)$$

Rearrangement gives us:

$$\mathbf{J}^{\mathbf{T}}\mathbf{W}\mathbf{J}\mathbf{h} = \mathbf{J}^{\mathbf{T}}\mathbf{W}(\mathrm{y}-\hat{\mathrm{y}}) \quad (18)$$

The Levenberg-Marquardt Method

Levenberg proposed the combination of the two methods discussed above:

$$[\mathbf{J}^{\mathbf{T}}\mathbf{W}\mathbf{J} + \lambda\mathbf{I}]\mathbf{h} = \mathbf{J}^{\mathbf{T}}\mathbf{W}(y-\hat{\mathrm{y}}) \quad (19)$$

The size of the parameter λ determines whether the Gauss-Newton or gradient descent term on the left dominates. Large values of λ will result in dominance of the gradient, while small values of λwill result in dominance of the Gauss-Newton approximation, and the solution will converge to the local minimum [1]. The gradient descent method allows us to move in roughly the right direction when far away from the solution, while the quadratic approximation made by the Gauss-Newton is more accurate as we move close to the correct parameter values [2]. As before, this process is an iterative process. The value of λ is determined by evaluating size of the error after each iteration. If the error has increased compared to the previous, the value of $\mathbf{h}$ is not updated, λ is increased by a large factor, usually 10 [2]. If, on the other hand, the error has decreased, λ is decreased by the same factor and the process continues.

Marquardt's contribution to the LM method was the recognition that for large values of λ, the Hessian ($\mathbf{J}^{\mathbf{T}}\mathbf{J}$) is not used. As such, he proposed the following adjustment to the Levenberg method:

$$[\mathbf{J}^{\mathbf{T}}\mathbf{W}\mathbf{J} + \lambda\mathrm{diag}(\mathbf{J}^{\mathbf{T}}\mathbf{W}\mathbf{J})\mathbf{I}]\mathbf{h} = \mathbf{J}^{\mathbf{T}}\mathbf{W}(y-\hat{\mathrm{y}}) \quad (20)$$

which scales each component of the gradient with the second derivative. This addition prevents overshooting the solution when gradient descent dominates. If the second derivative for any component is large, when we solve for $\mathbf{h}$ by taking the inverse of this term, the scaled gradient is very small, and the resulting movement in that direction is small. Since a large second derivative

means we're approaching the minimum, this prevents an iteration where gradient descent dominates from moving too far along any direction.

IMPLEMENTATION

The above methods were implemented in MATLAB as shown below.

Gradient Descent

This function takes three inputs, the number of iterations, the initial guess, and the parameters used to generate the data points we're trying to fit to. Two types of functions were explored, with similar results. The first was the sum of sines, the one seen implemented below was in the form:

$$\hat{y} = p_1 e^{-\frac{x}{p_2}} + p_3 x e^{-\frac{x}{p_4}} \tag{21}$$

where p_i is the ith term in the parameter vector **p**.

```
function res = graddescent (iteratns, guess, parameters)
    if nargin < 3
        parameters=[5 2 .2 10];
    end
    if nargin < 2
        guess = [4.98 1.98 .18 9.9]';
    end
    if nargin < 1
        iteratns =100;
    end
```

Lambda is set, which is the scale factor by which the perturbation **h** is multiplied before being added to the previous guess for **p**.

```
    lambda = .001;
    n= length(guess);
```

A grid of evenly spaced points is generated, and eventually used to generate the data points that we're trying to fit to. This grid is evenly spaced for convenience.

```
    xs = linspace(1,100,20);
```

The scaling matrix **W** is generated. It is just the identity matrix, since we want the fit to be equally as good at all points x.

```
    W = eye(length(xs)); %setting weighting all to 1 for now
    p=guess;
    syms x
```

The function y_hat, which is the function to be fitted, is generated by inputting the guess for **p** into the function datapts. "n" is used in the version of this function that is the sum of sines.

```
functn_yhat= datapts(n,p);
```

To calculate the Jacobian numerically, we step **p** forward slightly and use the approximation

$$J_{ij} = \frac{\delta \hat{y}_i}{\delta \mathbf{p_j}} = \frac{\hat{y}(t_i; \mathbf{p} + \delta \mathbf{p_j}) - \hat{y}(t_i; \mathbf{p})}{\delta \mathbf{p_j}} \tag{21}$$

Here, $\delta \mathbf{p_j}$is defined.

```
dp = .00001;
```

The solution is set using the input parameters, and a set of data points are generated using this function and the set of x values from before.

```
func_y_real=datapts(n,parameters);
datapoints=(subs(func_y_real, x, xs))';
```

In each iteration, the guess **p** is used to find points corresponding with the data points. The difference between these two sets of points is calculated.

```
for i = 1:iteratns
    fitfunctnpoints=(subs(functn_yhat, x,xs))';
    diff = datapoints- fitfunctnpoints;
```

The Jacobian is calculated numerically `using the function seen below.`

```
    J= jacobian (dp, p, xs,functn_yhat);
```

The perturbation **h** is calculated as derived in the gradient descent method.

```
    h = lambda*diff'*W*J;
```

A new **p** is calculated, and functn_yhat is updated with this new set of parameters.

```
    p=p+h';
    functn_yhat = datapts(n,p);
end
res=p;
end
```

The Jacobian function implements equation 21.

```
function res = jacobian (dp, p, xs,functn_yhat)
syms x
    J = zeros(length(xs), length(p));
    for j = 1:length(p)
        del_p = zeros(length(p),1);
        del_p(j) = dp;
        p_jac = p+del_p;
        functn_yhat_jac = datapts(length(p_jac), p_jac);
        for i = 1:length(xs)
            newpyhat= subs(functn_yhat_jac, x, xs(i));
            oldpyhat =subs(functn_yhat, x, xs(i));
            J(i,j)=(newpyhat - oldpyhat)/dp;
        end
    end
    res = J;
end
```

Function datapts takes an input of parameters values and generates a function based on those function values. In the sum of sines version, n determines how many sines are summed. These portions of the code is commented out, but kept in this text for reference.

```
function res = datapts (n,p)
    if nargin < 2
       p = [5 2 .2 10];
    end
    if nargin < 1
        n= 2;
    end
    syms x
%     func=0;
    func = n/n*p(1) * exp(-x/p(2))+ p(3) *x *exp(-x/p(4));
%         for i = 1:n
%             func=func+sin(p(i)*x);
%         end
    res=func;
end
```

Gauss-Newton

This method was implemented in the same way, with the only difference being the equation for the perturbation **h**.

Specifically, $\mathbf{J^T W J h} = \mathbf{J^T W}(y - \hat{y})$. It can be seen below that each part of this equation is calculated separately, and **h** found using this relationship.

```
    for i = 1:iteratns
        fitfunctnpoints=(subs(functn_yhat, x,xs))';
        diff = datapoints- fitfunctnpoints;
        J= jacobian (dp, p, xs,functn_yhat);
        JtransWJ = J'*W*J;
```

```
        JtransW = J'*W*diff;
        h = JtransWJ\JtransW;
        p=p+h;
        functn_yhat = datapts(n,p);
    end
```

Levenberg-Marquardt

In addition to the new expression for **h** associated with this method, this implementation also included updating the value of λ depending on what happened to the size of the chi-square error with each iteration.

```
    for i = 1:iteratns
        fitfunctnpoints=(subs(functn_yhat, x,xs))';
        diff = datapoints- fitfunctnpoints;

As before, the Jacobian is calculated.
        J= jacobian (dp, p, xs,functn_yhat);
        JtransWJ= J'*W*J;
        diagonal= diag(diag(JtransWJ));
```

h is equal to the division of $[\mathbf{J^T W J} + \lambda diag(\mathbf{J^T W J})]$ into $\mathbf{J^T W}(y - \hat{y})$

```
        h = linsolve((JtransWJ+lambda*diagonal), J'*W*diff);
        p_temp=p+h;
```

A temporary y_hat function is generated so that we can test what happens to the error with this new set of parameters.

```
        functn_yhat_temp = datapts(n,p_temp);
        newfitfuncpoints = (subs(functn_yhat_temp, x,xs))';
        newdiff= datapoints - newfitfuncpoints;
```

The chi-square error term is calculated with the old and new parameters.

```
        chi_new= 1/2*newdiff'*W*newdiff;
        chi_old=1/2*diff'*W*diff;
```

If the new parameters are better, then they are set as the new guess, and lambda is decreased by a factor of 10.

```
        if chi_new<chi_old
            p=p_temp;
            lambda = lambda/10;
```

Otherwise, the old set of parameters is kept, and lambda is increased by a factor of 10.

```
        else
            p=p;
            lambda = lambda*10;
        end
        functn_yhat = datapts(n,p);
    end
```

EVALUATION AND COMPARISON

To test and compare the performance of the Levenberg-Marquardt method with the gradient descent and Gauss-Newton methods, we generated a data set according to the following parameterization, using the parameter values $(p_1, p_2, p_3, p_4) = (20, 10, 1, 50)$, for t ϵ $[0, 100]$:

$$y = p_1 e^{\,t/p_2} + p_3 t\, e^{\,-t/p_4}$$

We first ran our algorithms starting from an initial guess for the parameter values of $(\mathrm{p}_1, \mathrm{p}_2, \mathrm{p}_3, \mathrm{p}_4) = (5, 2, 0.2, 10)$. The Levenberg-Marquardt method quickly converges on the correct parameterization after 20 iterations. Gradient descent is much slower, slowly approaching the solution after 5000 iterations. If we try to speed it up by increasing λ, the value that determines essentially how large our steps are, we quickly encounter divergence. This is already the case with Gauss-Newton. Our initial guess is too far off the true value of the parameters.

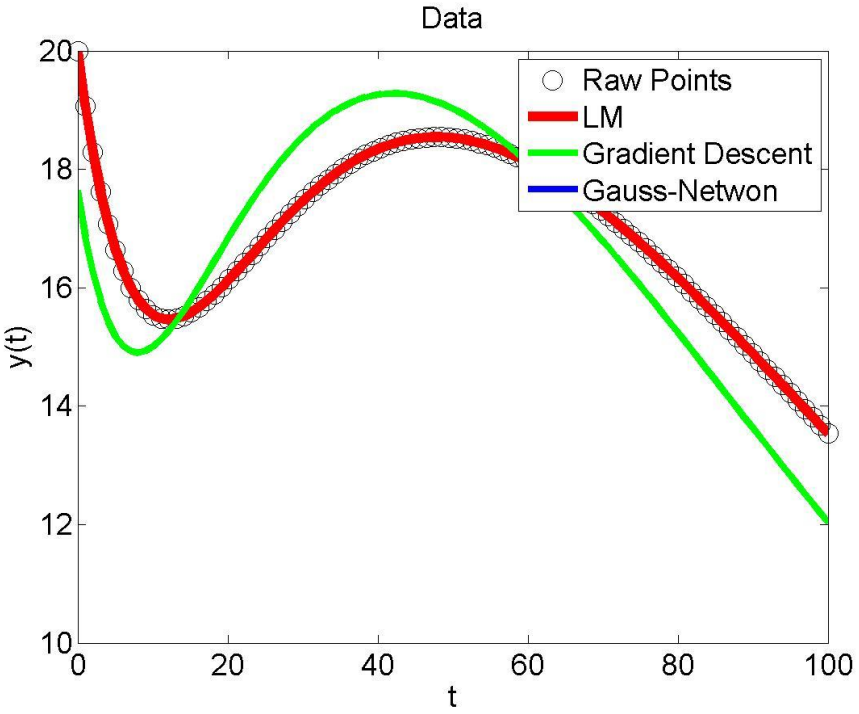

Figure 1. The generated data set (open circles) compared with the results of our three algorithms. Levenberg-Marquardt and Gauss-Newton were run for 20 iterations each. Gradient descent was run for 5000 iterations. The initial guess for the parameter values was $(p_1, p_2, p_3, p_4) = (5, 2, 0.2, 10)$. The results for Gauss-Newton are not visible because the solution diverges.

As a metric to determine how well the three different methods were performing, we calculated the error (both in terms of χ^2 and the overall parameter error) at each iteration. It is clear that the Levenberg-Marquardt method vastly outperforms the gradient descent method in speed and the Gauss-Newton in stability.

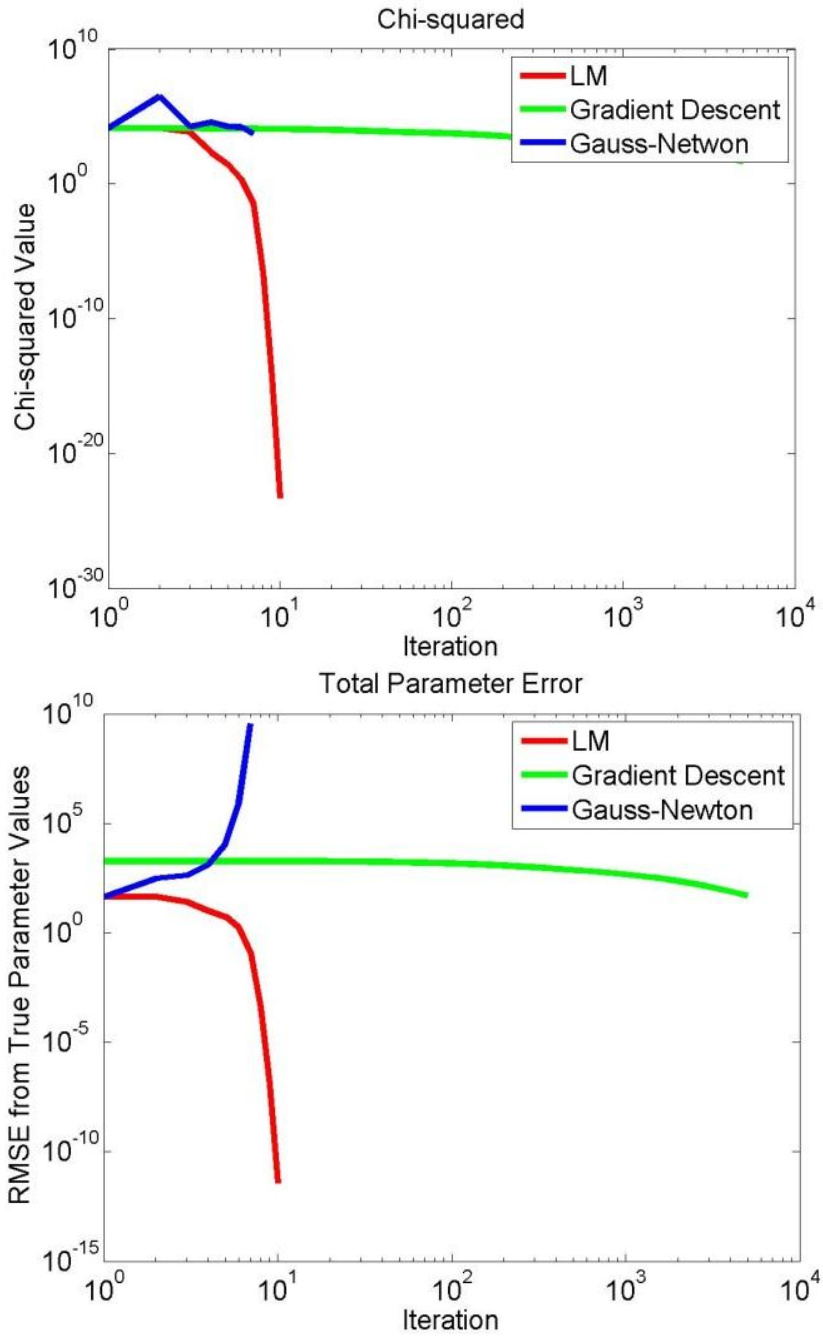

Figure 2. Two different ways of measuring error in the algorithms. On top, we simply measure χ^2 at each iteration. On the bottom, we calculate the root square mean error in the four parameters values. We can see that the Levenberg-Marquardt method is successful at quickly approaching the solution. The gradient descent method, while it moves in the right direction, is extremely slow. Gauss-Newton quickly diverges.

Levenberg-Marquardt's success is due to its adaptive nature: it acts more like Gradient Descent at first, and then morphs into Gauss-Newton as it approaches the solution. We can see this behavior if we consider the value of λ as it changes each iteration.

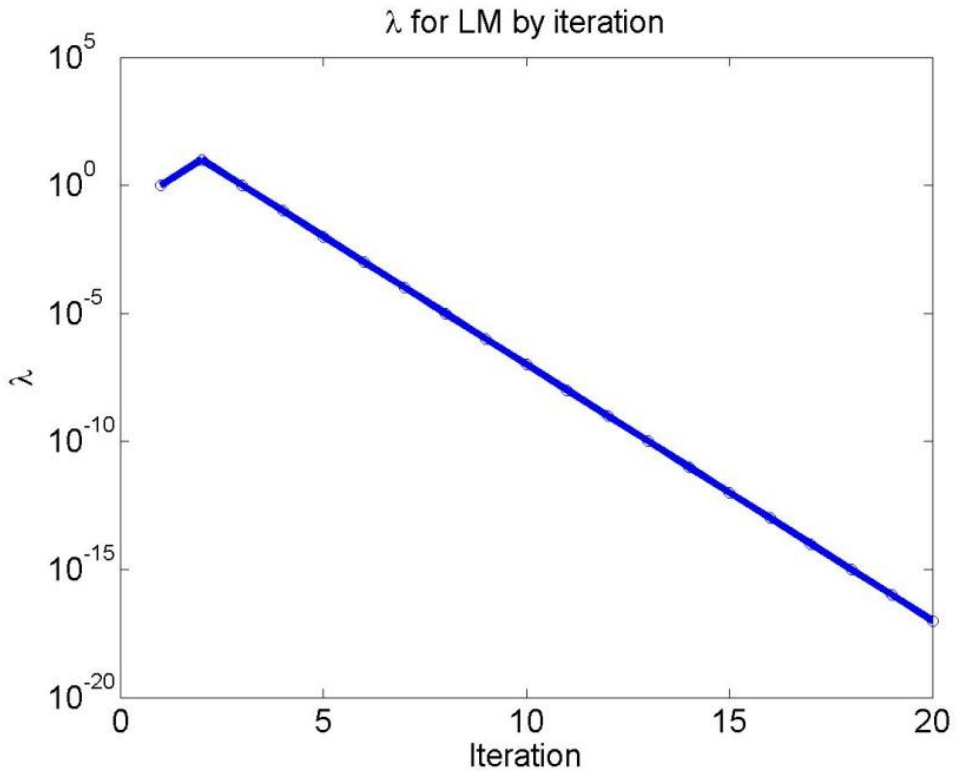

Figure 3. The value of λ at each iteration. High values indicate behavior like that of gradient descent and low values indicate behavior like that of Gauss-Newton.

In order to achieve convergence using the Gauss-Newton method, we used initial parameter values that were closer to the true parameter values: $(p_1, p_2, p_3, p_4) = (18, 12, 1.2, 45)$ (Figure 4).

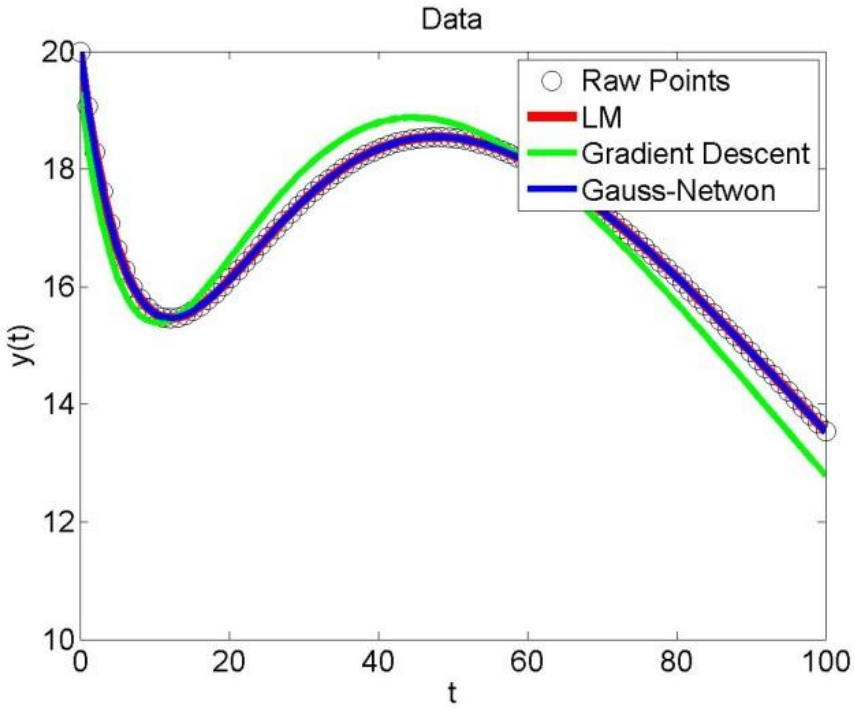

Figure 4. The generated data set compared with the results of our three algorithms. Levenberg-Marquardt and Gauss-Newton were run for 20 iterations each, producing almost identical solutions. Gradient descent was run for 5000 iterations. The initial guess for the parameter values was $(p_1, p_2, p_3, p_4) = (18, 12, 1.2, 45)$.

For this set of initial conditions, we again calculated the error at each iteration. Gradient descent is still slow. Gauss-Newton slightly outperforms Levenberg-Marquardt in this situation. Because Levenberg-Marquardt is adaptive, it takes a few iterations for it to approach the behavior of Gauss-Newton. In exchange for this tradeoff, it is stable over a wider range of initial parameter values, as demonstrated in the above example.

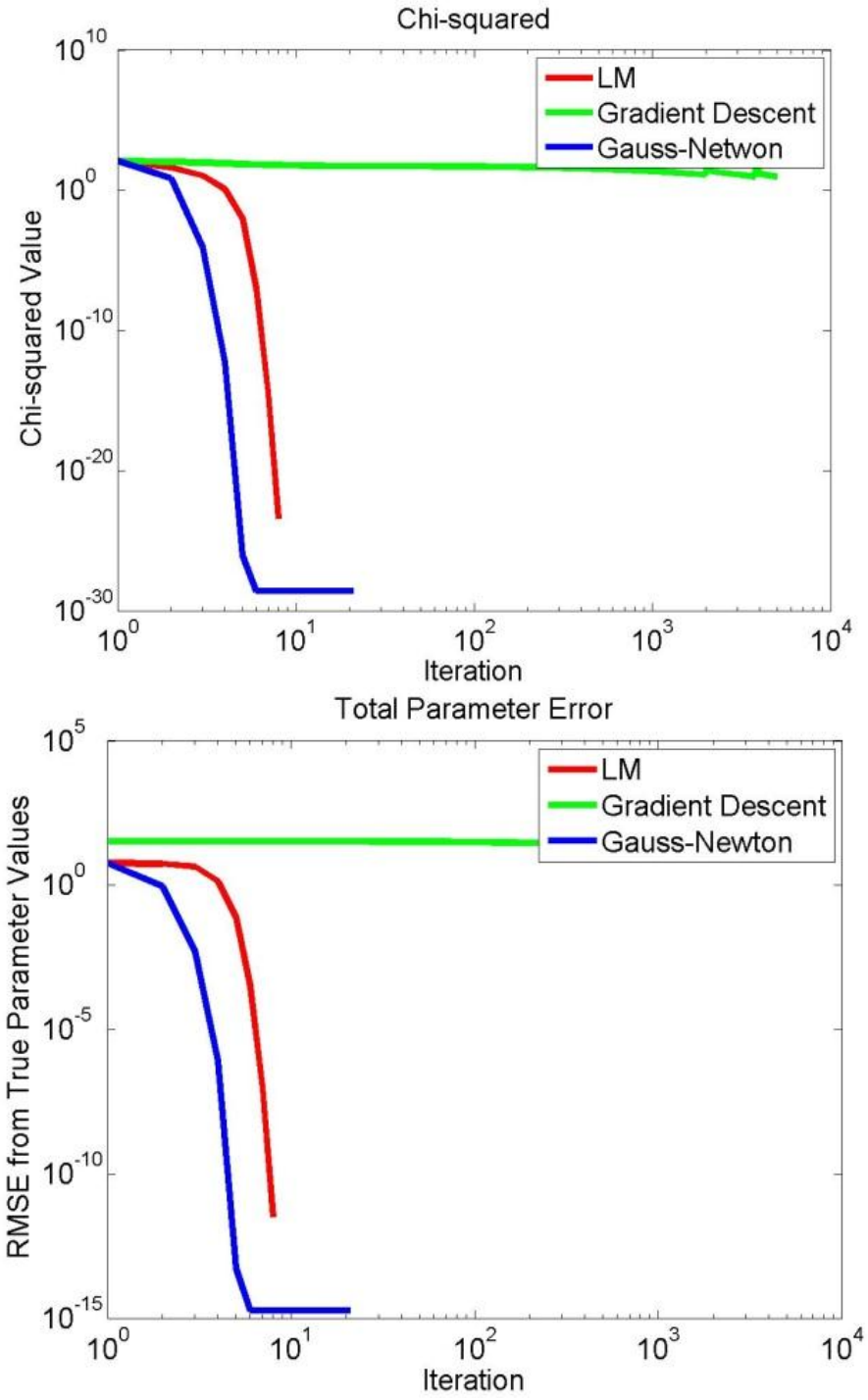

Figure 5. Two different ways of measuring error in the algorithms. On top, we simply measure χ^2at each iteration. On the bottom, we calculate the root square mean error in the four parameters values. We can see that in this case, both the Levenberg-Marquardt and Gauss-Newton methods are successful at quickly approaching the solution. The gradient descent method, while it moves in the right direction, is still extremely slow.

CONCLUSION

The Levenberg-Marquardt algorithm can be used in a variety of applications to find optimal parameter values given a function and a set of data points. This has been used with great effectiveness in the training of neural networks, as well as data analysis of complicated systems like ground water filtration systems and biological simulations of metabolism [3-5]. Its adaptability presents considerable advantages over slow methods such as gradient descent and frequently unstable methods such as Gauss-Newton. The dynamically changing λ value allows Levenberg-Marquardt to have the advantages of both worlds.

REFERENCES

[1]. Gavin H. The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems. 2010. Available at: http://www.duke.edu/~hpgavin/ce281/lm.pdf [Accessed December 1, 2010].

[2]. Ranganathan A. The Levenberg-Marquardt Algorithm. 2004. Available at: http://cronos.rutgers.edu/~meer /TEACH/ADD/ananth.pdf [Accessed December 1, 2010].

[3]. Mendes P, Kell D. Non-linear optimization of biochemical pathways: applications to metabolic engineering and parameter estimation. *Bioinformatics*. 1998;14(10):869 -883.

[4]. Szeliski R. Image mosaicing for tele-reality applications. In: *Applications of Computer Vision, 1994., Proceedings of the Second IEEE Workshop on*.; 1994:44-53.

[5]. Kermani BG, Schiffman SS, Nagle HT. Performance of the Levenberg-Marquardt neural network training method in electronic nose applications. *Sensors and Actuators B: Chemical*. 2005;110(1):13-22.

WORK DISTRIBUTION

Grace did background research and wrote the section describing the various methods, both computationally and in implementation. Arash generated pretty pictures and wrote the evaluation and comparison section. Arash also wrote the Work Distribution section. Both Grace and Arash wrote code implementing all three methods.

Levenberg is responsible for most of the work behind the algorithm. Marquardt made a single change (replaced the identity matrix with the diagonal Hessian) and now receives half the credit for Levenberg's method. Gauss and Newton are both superb role models for any aspiring mathematician and deserve more credit that they could possibly get. I have never heard of Gradient Descent, but I suppose he or she did useful things as well.