Nonlinear Least Squares Curve Fitting

Jack Merrin

February 12, 2017

1 Summary

There is no general solution for the minimization of chi-squared to find the best estimates of the parameters when the model is nonlinear. A number of methods have been developed that can often find the best estimates and their uncertainties with arbitrary precision. Iterative methods are usually used to minimize a nonlinear chi-squared. Another brute force method, called the grid method, tests every possible set of parameters with a certain precision. This is usually not possible when the parameter space or the number of parameters are large. Commercial curve fitting software usually implements an iterative method called the Levenberg-Marquardt method. If you have a nonlinear curve fitting software you can also apply it to linear models also if you want, but there is no guarantee you get the analytic solution. In this lesson, we discuss how to implement the grid method for simple problems. We give an example of Newton's method as a basic iterative method. We explore the gradient descent method, Gauss-Newton method, and the Levenberg Marquardt method both theoretically and in MATLAB. Users of curve fitting software should understand the underlying algorithms and be at least able to check if the software is performing the calculation correctly.

2 The minimization of nonlinear data

We minimize the χ^2 error function in least squares theory.

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - f(x_i; \mathbf{a}))^2}{\sigma_i^2}$$

We showed in a previous lesson that if f is a linear model, or

$$f(x_i; \mathbf{a}) = a_1 X_1(x_i) + a_2 X_2(x_i) + \cdots + a_M X_M(x_i)$$

Then χ^2 has a unique global minimum and we can analytically derive expressions for the best estimates of the model parameters and their uncertainties, $\mathbf{a}$ and σ_{a_j}.

When f is not of the above form then it is nonlinear. An example is

$$f(x_i; \mathbf{a}) = a_1 \sin(a_2 x_i) + a_3 e^{a_4 x_i}$$

When you take the derivative of χ^2 for a nonlinear model, you get a nonlinear equation which may have no general analytical solution in terms of linear algebra. One then has to resort to different methods. One typically resorts to iterative methods that require an initial guess for $\mathbf{a}$ and update rules for the parameters that move towards the minimum of χ^2. One repeats this a number of times until the algorithm converges to some minimum of χ^2. This may be a local minimum of the actual global minimum. There is not a general test to show that a local minimum is actually the global minimum. It is always a good idea to start with different initial guesses and see if the same minimum is reached. This is no guarantee, but it is generally tolerated.

We want to work on some nonlinear data in this lesson so we can take a simple nonlinear model. For example, a common nonlinear model is a mass oscillating on a spring with damping.

$$m\ddot{y} + 2b\dot{y} + ky = 0$$

For a certain choice of constants, $b^2 - km = 0$, the motion is called critically damped and obeys the following solution.

$$y = (A + Bt)\exp(-Ct)$$

We can add some random noise to that function and take the average of five samples to generate our data set. This way we know how close different approach fit to the actual parameters. Now we generate a dataset in MATLAB and save it as a .mat file.

Example 2.1. *Generate some sample data that represent critical damping.*

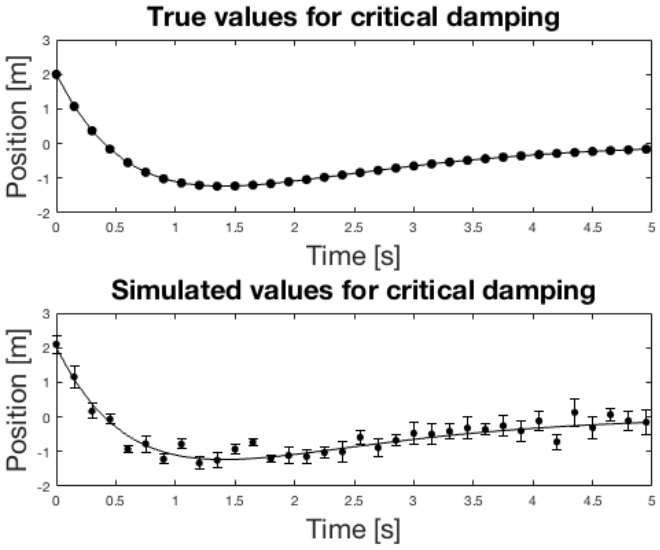
Solution 2.1. *We take five simulated measurements and average them.*

```
clear;figure(1);clf;
A = 2;
B = -5;
C = 1;
xi = 0:0.15:5;
Ni = 5;
ymodel = (A+B.*xi).*exp(-C.*xi);
yi = [];
sigi = [];

for i = 1:length(xi)
    temp = [];
    for j = 1:Ni
        temp(j) = (A + B.*xi(i)).*exp(-C.*xi(i)) + normrnd(0,0.5);
    end
    yi(i) = mean(temp);
    sigi(i) = std(temp)/sqrt(Ni);
end

subplot(2,1,1);
```

```
plot(xi,ymodel,'.-k','markersize',20);
axis([0 5  -2 3]);
hold on;
xlabel( 'Time [s]','fontsize',20);
ylabel( 'Position [m]','fontsize',20);
title('True values for critical damping','fontsize',20);
subplot(2,1,2)
errorbar(xi,yi, sigi,'.k','markersize',15);
xlabel( 'Time [s]','fontsize',20);
ylabel( 'Position [m]','fontsize',20);
title('Simulated values for critical damping','fontsize',20);
axis([0 5  -2 3]);
hold on;
plot(xi,ymodel,'-k','markersize',20)
save criticaldampingdata xi yi sigi
```



3 Brute force grid method

If you have the computational power then there is a brute force approach to curve fitting. This method is called the grid method. Suppose from the graph of some linear data you can tell that A is between 3 and 4 and B is between 0.5 and 1.5. Then you can calculate χ^2 for every value in between to 0.01 or 0.001 resolution. You then just find the least one by direct search in Matlab. Of course, those might take 10,000 or 100,000 calculations but you will get the same answer as the analytical result. You can do a coarse grid to find local minima and then do a fine grid to find the global minimum. One advantage of the grid method is that you don't need to know much theory and it is pretty obvious how it works. If you cannot narrow the parameter space to include the global minimum then you might require too many calculations to perform the optimization.

We will now study some iterative methods that can be used to minimize chi-squared. Grid method is not really an iterative method like the rest. You try to cast a net over the parameter surface. You see which loop of the net produces the minimum, then you cast a finer net over that loop and rapidly converge towards a minimum. This method is good because you get out of it what you expect. When you have lots of parameters it might not be possible to cast a fine enough net to progress because there are too many combinations.

Example 3.1. *Implement the grid method on Matlab to fit a straight line. Compare the results to*

Solution 3.1. *We can implement the grid method to fit a straight line model. It works just the same with nonlinear modesl*

```
%Grid Method

clear; clf;

A = 10;
B = 3;
Ni = 15;
xi = 1:2:15

for i = 1:length(xi)
    tx = [];
    for j = 1:Ni
        tx(j) = A + B*xi(i) + normrnd(0,3);
    end

    ei(i) = std(tx)/sqrt(Ni);
    yi(i) = sum(tx)/Ni;
end

errorbar(xi,yi,ei,'.k', 'markersize',10);
xlabel('trial')
ylabel('measurement')
hold on;
[A sigA B sigB] = lineABx(xi,yi,ei)

A
sigA
B
sigB

a = 9.5:0.0001:10.5;
b = 2.9:0.0001:3.1;
```

```
for j = 1:length(a)
    for k = 1:length(b)
        chisquared(j,k) = sum( 1./ei.^2.*(yi-b(k).*xi - a(j)).^2 );
    end
end

g = min(min(chisquared));
[I J] = find(chisquared == g);
AA = a(I);
BB = b(J);

A
AA
B
BB
```

4 Important matrix math

We can represent χ^2 in terms of vectors. Let

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - f(x_i; \mathbf{a}))^2}{\sigma_i^2}$$

We define the residual vector.

$$\mathbf{r} = y_i - f_i$$

We define the weight matrix which is diagonal.

$$W_{ij} = \frac{1}{\sigma_i^2}\delta_{ij}$$

So we can write χ^2 as

$$\chi^2(\mathbf{a}) = \mathbf{r^T W r}$$

We go back and forth between column and row vectors using the transpose property. Some properties of transpose are

$$\begin{aligned}
(\mathbf{A^T})^\mathbf{T} &= \mathbf{A} \\
\mathbf{A^T} = \mathbf{A} \rightarrow \mathbf{A} &\quad \textit{symmetric} \\
(\mathbf{A} + \mathbf{B})^T &= \mathbf{A}^T + \mathbf{B}^T \\
(\mathbf{AB})^\mathbf{T} &= \mathbf{B^T A^T}
\end{aligned}$$

When you multiply an NxM matrix with an MxL matrix you get an NxL matrix. You can use size(A) command in Matlab to see how your matrices are arranged. You can use the transpose operator A' to convert a column vector to a row vector and vice versa. You may have to check your expression if you haven't kept track of your transposes.

We will be taking partial derivatives of χ^2 with respect to the parameters which is the Jacobian matrix.

$$\frac{\partial \mathbf{f}}{\partial \mathbf{a}} = \mathbf{J}$$
$$\frac{\partial \mathbf{r}}{\partial \mathbf{a}} = -\mathbf{J}$$

We also have the Hessian matrix which is defined as

$$H_{ij} = \frac{\partial^2 \chi^2}{\partial a_j \partial a_k}$$

Here is an important result from Matrix calculus for a symmetric matrix $\mathbf{W}$.

$$\frac{\partial(\mathbf{h^T W h})}{\partial \mathbf{x}} = 2\mathbf{h^T W}\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$$

5 Newton's method

We will explore Newton's method because it is a simple example of an iterative method which we will be using to minimize χ^2. Consider the following diagram.

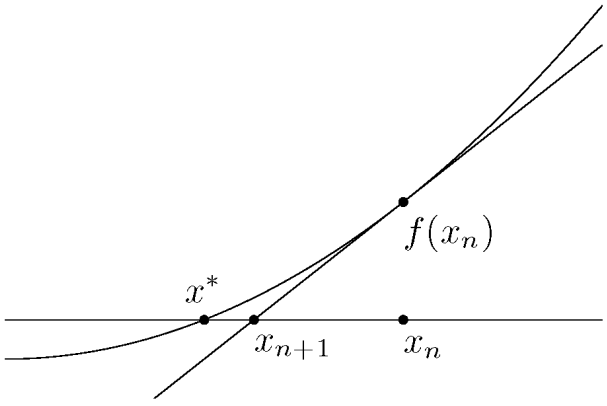


Figure 1: Illustration of Newton's method

We have

$$f'(x) \approx \frac{f(x_n)}{x_n - x_{n+1}}$$

We can rearrange the expression as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

So if we know x_1 the initial guess, we can find x_2. Then we can plug back in x_2 in the next iteration to find x_3 and so on. We get closer and closer to the point where $f(x^*) = 0$ with each iteration.

Example 5.1. *Find the square root of two to high precision.*

Solution 5.1. *We can try out Newton's method to calculate the square root of 2. If we have the equation,*

$$x^2 = 2$$

Then we can write $f(x) = x^2 - 2$. *This function goes to zero at* $\sqrt{2}$ *so we can apply Newton's method.*

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

```
clear; format long;
N = 10;
x(1) = 2.
for i = 1:N
     x(i+1) = x(i) - (x(i)^2 -2)/(2*x(i));
end
```

If we look at the contents of x *we find,*

n	x_n
1	*2*
2	*1.500*
3	*1.416666666666667*
4	*1.414215686274510*
5	*1.414213562374690*
6	*1.414213562373095*
7	*1.414213562373095*

The update function just produces fractions or rational numbers if you choose an initial guess like 2. In old times, before the calculator one could calculate the square root of any number to sufficient accuracy by finding the fraction, then doing long division in the last step. It is quite a good method, 15 digits are perfect after the 6th iteration.

There are a number of things that can go wrong with Newton's method. Each successive update may result in oscillation or the updates may be directed to another solution you are not looking for. The best way to find a root is to make a plot of $f(x)$ near the root and

plan your initial guess such that it climbs down the hill to the root avoiding any bumps or being too far away from the minimum. Also, your function may have a restricted domain or singularities where division by zero occurs. These will obviously crash Newton's method.

6 Gauss-Newton method

$$r_i = y_i - f(x_i; \mathbf{a})$$

The Gauss-Newton algorithm is another iterative curve fitting method. One can arrive at the update formula by considering a Taylor expansion of $f(x_i; \mathbf{a})$ with respect to an update $\boldsymbol{\delta}$.

$$\begin{aligned} f(x_i; \mathbf{a} + \boldsymbol{\delta}) &\approx f(x_i, \mathbf{a}) + \frac{\partial f(x_i; \mathbf{a})}{\partial \mathbf{a}} \boldsymbol{\delta} \\ f(x_i; \mathbf{a} + \boldsymbol{\delta}) &= \mathbf{f} + \mathbf{J}\boldsymbol{\delta} \end{aligned}$$

At the minimum of χ^2, then

$$\frac{\partial \chi^2}{\partial \boldsymbol{\delta}} = 0$$

Then we have

$$\begin{aligned} \chi^2(\mathbf{a} + \boldsymbol{\delta}) &\approx (\mathbf{y} - \mathbf{f}(x_i; \mathbf{a}) - \mathbf{J}\boldsymbol{\delta})^T \mathbf{W} (\mathbf{y} - \mathbf{f}(x_i; \mathbf{a}) - \mathbf{J}\boldsymbol{\delta}) \\ \chi^2(\mathbf{a} + \boldsymbol{\delta}) &\approx (\mathbf{r} - \mathbf{J}\boldsymbol{\delta})^T \mathbf{W} (\mathbf{r} - \mathbf{J}\boldsymbol{\delta}) \end{aligned}$$

We want to take the derivative of χ^2 with respect to $\boldsymbol{\delta}$ and set it equal to zero. Remember the important derivative.

$$0 = (\mathbf{r} - \mathbf{J}\boldsymbol{\delta})^T \mathbf{W} (-\mathbf{J})$$

We can take the transpose of that expression to get

$$0 = \mathbf{J}^T \mathbf{W} (\mathbf{r} - \mathbf{J}\boldsymbol{\delta})$$

Remember that $\mathbf{W}$ is symmetric. Solving for $\boldsymbol{\delta}$ gives the Gauss Newton update rule.

$$\begin{aligned} \boldsymbol{\delta} &= (\mathbf{J}^T \mathbf{W} \mathbf{J})^{-1} \mathbf{J}^T \mathbf{W} \mathbf{r} \\ \mathbf{a}_{s+1} &= \mathbf{a}_s + \boldsymbol{\delta} \end{aligned}$$

To implement the Gauss-Newton method we only need to code for the update rule. Gauss-Newton seems to be more picky about the initial guess than other methods.

Example 6.1. *Implement the Gauss Newton method to fit the model data*

Solution 6.1. *We change the initial guess and update the delta rule.*

```
clear;clf;figure(1);
load modeldata.mat

errorbar(xi, baryi, barsigi,'.k');
hold on;
xlabel('time [s]', 'fontsize',20);
ylabel('distance [m]', 'fontsize',20);

N = length(xi);
W = zeros(N,N);
for i = 1:N
    W(i,i) = 1/barsigi(i)^2;
end
%some initial random guess
A(1) = 1.4
B(1) = -4.3
C(1) = 1.3


gamma = 0.00001; %Arbitrary step size
S = 10000;  %Number of steps

for s = 1:S
    for i = 1:N
        J(i,1) = exp(-C(s)*xi(i));
        J(i,2) = xi(i)*exp(-C(s)*xi(i));
        J(i,3) = (A(s) + B(s)*xi(i) )*(-xi(i))*exp(-C(s)*xi(i));
        ri(i) = (baryi(i) - (A(s)+ B(s)*xi(i))*exp(-C(s)*xi(i)));
    end
    % we really only have to change one line of code for delta
    delta = inv(J'*W*J)*J'*W*ri';
    A(s+1)=A(s)+delta(1);
    B(s+1)=B(s)+delta(2);
    C(s+1)=C(s)+delta(3);
end;

ymodel = (A(end)+B(end).*xi).*exp(-C(end).*xi);
plot(xi, ymodel,'r');
title('Gauss Newton fits','fontsize',20);
```

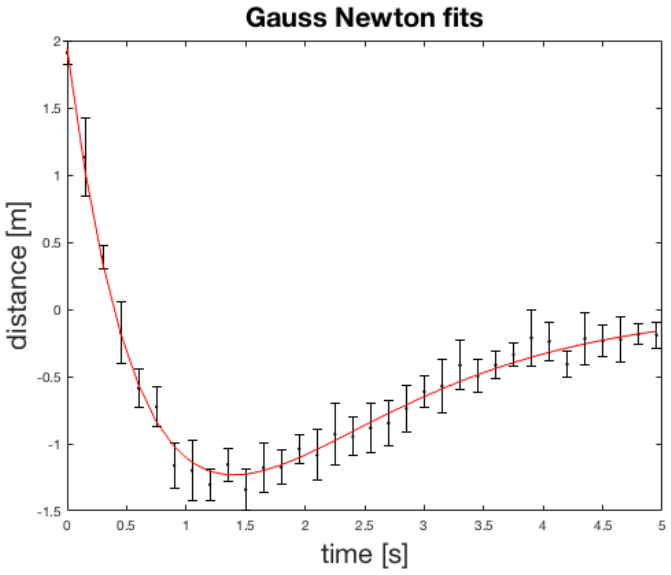


Figure 2: Gauss Newton works

7 Gradient decent method

Imagine χ^2 as a multidimensional hyper-surface of the possible fit parameters, $\mathbf{a} = (a_1, a_2, ..., a_N)$. A choice of $\mathbf{a}$ corresponds to a location on the hypersurface. We can use the result from vector calculus that $-\nabla\chi^2$ will point towards a local minimum. We can then step in that direction and repeat the process. We take an arbitrary step size of $\gamma = 0.00001$.The gradient of chi squared is calculated as

$$\nabla\chi^2 = \frac{\partial(\mathbf{r^T W r})}{\partial \mathbf{x}} = 2\mathbf{r^T W}\frac{\partial \mathbf{r}}{\partial \mathbf{x}} = -2\mathbf{r}^T\mathbf{WJ}$$

The gradient points up the slope of a function so our update rule should point in the opposite direction. We choose the step size to be proportional to γ which is something we set arbitrarily. We can make it 10^{-5} and see if the calculation converges.

$$\mathbf{a}_{s+1} = \mathbf{a}_s + \gamma\mathbf{r^T WJ}$$

Example 7.1. *Implement gradient descent with the model data described earlier in this chapter.*

Solution 7.1. *We load the data and do the calculation*

```
%Imprementing gradiate descent
clear;clf;figure(1);
load modeldata.mat

errorbar(xi, yi, sigi,'.k');
```

```
hold on;
xlabel('time [s]', 'fontsize',20);
ylabel('distance [m]', 'fontsize',20);

N = length(xi);
W = zeros(N,N);
for i = 1:N
    W(i,i) = 1/sigi(i)^2;
end
%some initial random guess
A(1) = .2
B(1) = -.2
C(1) = .3


gamma = 0.00001; %Arbitrary step size
S = 10000;  %Number of steps

for s = 1:S
    for i = 1:N
        J(i,1) = exp(-C(s)*xi(i));
        J(i,2) = xi(i)*exp(-C(s)*xi(i));
        J(i,3) = (A(s) + B(s)*xi(i) )*(-xi(i))*exp(-C(s)*xi(i));
        ri(i) = (yi(i) - (A(s)+ B(s)*xi(i))*exp(-C(s)*xi(i)));
    end
    delta = 2*gamma*ri*W*J;
    A(s+1)=A(s)+delta(1);
    B(s+1)=B(s)+delta(2);
    C(s+1)=C(s)+delta(3);
end;

ymodel = (A(end)+B(end).*xi).*exp(-C(end).*xi);
plot(xi, ymodel,'r');
title('Gradient descent fit','fontsize',20);
```

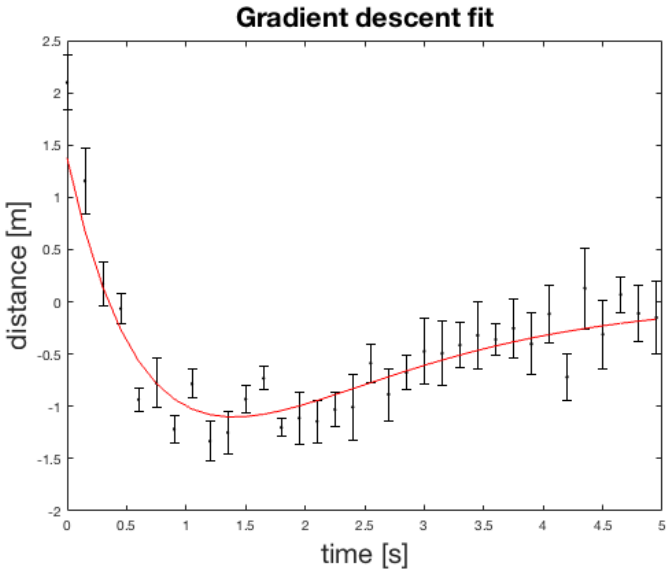


Figure 3: Gradient descent works

8 Levenberg Marquadt Method

The Levenberg-Marquardt algorithm is one of the most powerful curve fitting methods available. Most commercial software packages use it in one form or another. It is robust, meaning that it will converge to a local minimum for a wide range of initial guesses. It does not always find the global minimum. The Levenberg-Marquardt algorithm can be thought of as a combination of the Gauss-Newton method and the gradient descent method. To derive the Levenberg-Marquardt algorithm one starts along the same lines as the Gauss-Newton method and we arrive at

$$(\mathbf{J^T W J})\boldsymbol{\delta} = \mathbf{J^T W r}$$

The Levenberg-Marquardt method slightly modifies this equation to

$$(\mathbf{J^T W J} + \lambda \operatorname{diag}(\mathbf{J^T W J}))\boldsymbol{\delta} = \mathbf{J^T W r}$$

The diag symbol means only the components along the diagonal are non-zero. The update $\boldsymbol{\delta}$ can be solved for by an inverse matrix. The Levenberg-Marquardt update rule is given by

$$\begin{aligned} \boldsymbol{\delta} &= [\mathbf{J^T W J} + \lambda \mathrm{diag} \mathbf{J^T W J}]^{-1} \mathbf{J^T W r} \\ \mathbf{a}_{s+1} &= \mathbf{a}_s + \boldsymbol{\delta} \end{aligned}$$

Example 8.1. *Implement the Levenberg Marquardt algorithm in Matlab*

Solution 8.1. *Here is the code*

```
clear; figure(1);clf;
A = 2;
B = -5;
C = 1;
xi = 0:0.1:5;
Ni = 5;
ymodel = (A+B.*xi).*exp(-C.*xi);
yi = [];
sigi = [];

for i = 1:length(xi)
    temp = [];
    for j = 1:Ni
        temp(j) = (A + B.*xi(i)).*exp(-C.*xi(i)) + normrnd(0,0.5);
    end
    yi(i) = mean(temp);
    sigi(i) = std(temp)/sqrt(Ni);
end

subplot(2,1,1);
plot(xi,ymodel,'.-k','markersize',20);
axis([0 5  -2 3]);
hold on;
xlabel( 'Time [s]','fontsize',20);
ylabel( 'Position [m]','fontsize',20);
title('True values for critical damping','fontsize',20);
subplot(2,1,2)
errorbar(xi,yi, sigi,'.k','markersize',15);
xlabel( 'Time [s]','fontsize',20);
ylabel( 'Position [m]','fontsize',20);
title('Simulated values for critical damping','fontsize',20);
axis([0 5  -2 3]);
hold on;
plot(xi,ymodel,'-k','markersize',20)
save criticaldampingdata xi yi sigi

errorbar(xi, yi, sigi,'.k');
hold on;
xlabel('time [s]', 'fontsize',20);
ylabel('distance [m]', 'fontsize',20);

N = length(xi);
W = zeros(N,N);
for i = 1:N
```

```
    W(i,i) = 1/sigi(i)^2;
end
%some initial random guess
A(1) = 1;
B(1) = -4;
C(1) = 1.3;

S = 10000;
lambda = 0.01;

for s = 1:S
    for i = 1:N
        J(i,1) = exp(-C(s)*xi(i));
        J(i,2) = xi(i)*exp(-C(s)*xi(i));
        J(i,3) = (A(s) + B(s)*xi(i) )*(-xi(i))*exp(-C(s)*xi(i));
        ri(i) = (yi(i) - (A(s)+ B(s)*xi(i))*exp(-C(s)*xi(i)));
    end
    Q = J'*W*J;
    QQ = zeros(3,3);
    QQ(1,1) = Q(1,1)*(1+lambda);
    QQ(2,2) = Q(2,2)*(1+lambda);
    QQ(3,3) = Q(3,3)*(1+lambda);
    delta = inv(Q+QQ)*J'*W*ri';
    A(s+1)=A(s)+delta(1);
    B(s+1)=B(s)+delta(2);
    C(s+1)=C(s)+delta(3);
end;

ymodel = (A(end)+B(end).*xi).*exp(-C(end).*xi);
plot(xi, ymodel,'-.');
title('Levenberg Marquardt fits','fontsize',20);
```

9 Nonlinear least squares uncertainties

If you can have already found the optimal parameters by one of the various methods we are about to discuss then it is straightforward to calculate the estimated parameter uncertainties. Recall from the generalized linear theory, we had the matrix D_{jk}. There is a second way to calculate this matrix just take

$$D_{jk} = (1/2)\frac{\partial^2 \chi^2}{\partial a_j \partial a_k}$$

Now we can find C_{jk} and that leads to the parameter uncertainties $a_j = \sqrt{C_{jj}}$. One should verify that this method works by testing in on the $y = A + Bx$ model to generate D_{jk}.

For nonlinear models we can write out the matrix elements by calculating the derivatives.

$$D_{jk} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[\left(\frac{\partial f}{\partial a_j} \right) \left(\frac{\partial f}{\partial a_k} \right) + (f - y_i) \frac{\partial^2 f}{\partial a_j \partial a_k} \right]$$

Example 9.1. *Find the parameter estimate for B and its uncertainty in the straight line model through the origin by calculating D_{jk}. The formula for χ^2 when we fit the line through the origin was*

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - Bx)^2}{\sigma_i^2}$$
$$\chi^2 = S_{wyy} - 2BS_{wxy} + B^2 S_{wxx}$$

Solution 9.1. *We can easily find* $\mathbf{D}$ *it is a 1 by 1 matrix.*

$$\mathbf{D} = (1/2) \frac{d^2}{dB^2} \chi^2 = S_{wxx}$$

Since $\mathbf{C}$ *is the inverse of* $\mathbf{D}$ *we have*

$$\mathbf{C} = \frac{1}{S_{wxx}}$$

Therefore

$$\sigma_B = \frac{1}{\sqrt{S_{wxx}}}$$

This is exactly what we got before with this model using different methods.

Example 9.2. *Calculate D_{jk} for the critical damping problem and then find the uncertainties in the estimates*

Solution 9.2. *For this task I definitely want to use Mathematica. I notice that D_{jk} is symmetric so we only have to calculate six quantities. To find the derivatives in Mathematica I type the following.*

- $D[D[(1/SS^2) * (y - (A + B * t) * Exp[-C * t])^2, A], A]$
- $D[D[(1/SS^2) * (y - (A + B * t) * Exp[-C * t])^2, B], B]$
- $D[D[(1/SS^2) * (y - (A + B * t) * Exp[-C * t])^2, C], C]$
- $D[D[(1/SS^2) * (y - (A + B * t) * Exp[-C * t])^2, A], B]$
- $D[D[(1/SS^2) * (y - (A + B * t) * Exp[-C * t])^2, A], C]$

- $D[D[(1/SS^2) * (y - (A + B * t) * Exp[-C * t])^2, B], C]$

The results are implemented in MATLAB

```
wi = sigi.^(-2);
A=A(end);
B=B(end);
C=C(end);

E1 = exp(-C.*xi);
E2 = exp(-2.*C.*xi);
P1 = (A + B.*xi);
X2 = xi.*xi;
D(1,1) = sum( wi.*E2);
D(2,2) = sum( wi.*X2.*E2);
D(3,3) = sum( wi.*(E2.*X2.*P1.^2)-E1.*X2.*P1.*(-E1.*P1 + yi));
D(1,2) = sum( wi.*E2.*xi);
D(1,3) = sum( -wi.*E2.*xi.*P1 + wi.*E1.*xi.*(-E1.*P1 + yi));
D(2,3) = sum( -wi.*E2.*X2.*P1 + wi.*E1.*X2.*(-E1.*P1 +yi));

D
M = inv(D)
A
sqrt(M(1,1))
B
sqrt(M(2,2))
C
sqrt(M(3,3))
```

With five samples the values are usually pretty close to the true parameters. If you increase the number of samples Ni then you get even closer to the true values. I was expecting that the error bars would be larger and enclose the true values, but this is not always the case. The curve fit (dotted-dashed line) is visually pretty close to the true model, yet small differences remain. One can experiment with the code to try different starting models and initial guesses. This exercise was meant to show the LM method and how to implement it in MATLAB. You can easily now adapt the code to your curve fitting functions.

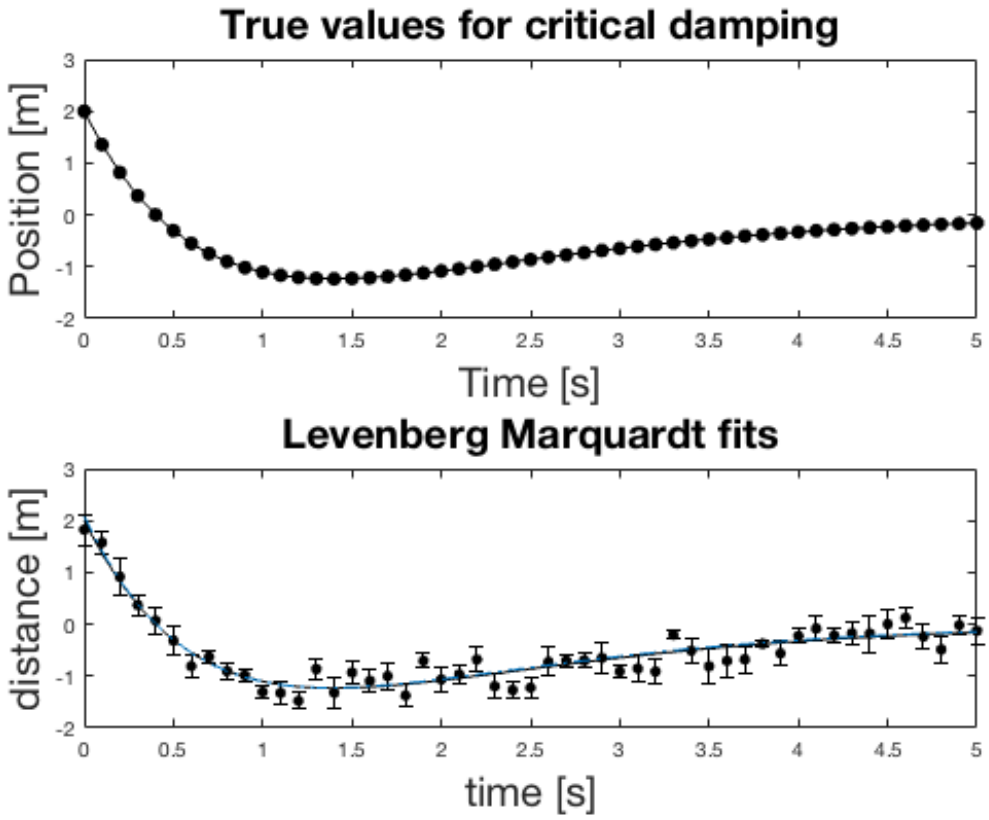


Figure 4: Levenberg Marquardt works. For this plot, the agreement with the true values is reasonable. A = 2.077(93), B = -5.17(12), and C = 1.026(16)

We have to get into further discussions of statistics later to determine what it means to have a good fit. This depends on the value of chi-squared and the number of degrees of freedom. For the meantime, we are stuck with chi by eye.

Visit www.sciencemathmastery.com for more lessons on error analysis, probability and statistics.