Week 3: Tutorial: Non-Linear Least Squares fitting to Data

Start Octave, go to your work directory of choice, add the Utility Subroutine to your path (the *addpath* command). Verify you can access utilities. (Asking for help on a particular subroutine will test the path).

We now extend our data modeling capabilities to use non-linear fitting methods. We will use the Octave routine called *leasqr*. This uses the Marquardt-Levenberg method to find the parameters that fit the model to the data. This also uses the method of least squares, which is a way to minimize the deviation with respect to each and every parameter you want to optimize. This is a very powerful, general and reasonably robust method to find the optimum agreement between data and a model (or theory).

The text by Hansen has a full section devoted to this routine and how useful it is.

You should verify that you now have access to this package:

pkg list % will show what packages you have installed and available for loading

look for optim and struct.

If you do not have the optim package, then download that package from Octave Forge website:

1. Start octave

2. Install the packages (optim requires struct as a dependency). This only needs to be done once.
pkg install -forge struct
pkg install -forge optim

3. Load the packages into the current octave session. When Octave is restarted, step 3 will need to be repeated. This can either be done by hand or put in any startup configuration file (octaverc or whatever)
pkg load struct
pkg load optim

Note that you need to install struct first and then optim. After installing you still need to load optim.

Others found that the download was hard to install. Therefore we suggest that you include the optim and the stats packages (or all packages for that matter) in you initial installation. If you did not do this you can just reinstall Octave with the packages selected. This avoids the need to install the packages directly.

Once you have optim, begin your script file with the following:

```
close all;  %this closes all figures, ensuring that you start with a clean slate
clear all;  %this removes all variables from the workspace
% For the non linear optimizer
pkg load optim     % The routine is called leasqr from the package optim
```

From the command line you can use:

```
>> help leasqr
```

This will test if the pkg load optim worked successfully; if successful, help on leasqr will be displayed.

This will list all of your packages, and struct and optim should be in the list:

```
>> pkg list %
```

A.1 Tutorial 1 script. (Reprise from Tutorial 1)

Bring in your script from Tutorial 1, Section C, where you brought in the kinetic data (from experiment one) that is mostly characterized by exponential decay. Your script should contain something similar to the following:

```
fn =  'KBP240nm26degrees.txt';   % make a slightly shorter name for the file
[ds]= get_OO_Data(fn);  % use either one of our local routines or one of Octaves for data import
time = ds(:, 1);  % optionally time = (time-time(1))/1000. Converts to relative seconds.
Abs = ds(:, 2);
plot(time, Abs ,'c-');   % data is so dense a solid line makes sense.
```

Set nlo and nhi, so that you pick the region best described by a single exponential decay. nlo from your previous work should work well here too. Here now we can set nhi = lengh(Abs); so that we get all of the data:

```
t_red = time(nlo:nhi);   % a reduced data set that is mostly single exponential
Abs_red=Abs(nlo:nhi);
```

Alternatively, if you know you are going to the end of the data you can write:

```
t_red = time(nlo:end);   % a reduced data set that is mostly single exponential
Abs_red=Abs(nlo:end);  % end is not set to a value; collect the data to the end of the vector.
```

Compare your reduced data with the full data by plotting:

```
plot(time, Abs, 'c-', t_red, Abs_red, 'k-')  % Black will be on top of the cyan full data set.
```

A.2 Set up a model for the data and choose parameters that give a reasonable fit.

The model is:

$$A = A_{inf} - \Delta A e^{-r(t-t_o)} \quad \text{or, equivalently,} \quad A = A(t_o) + \Delta A\left(1 - e^{-r(t-t_o)}\right)$$

You have 3 adjustable parameters: $Params = \{A_{inf}, \Delta A, r\}$ For any non-linear method to work, you must give good initial guesses to those parameters. And you need a subroutine, which will be a function (and still a *.m file) to actually do the simulation. You might notice that the time to start is $t \geq t_o$, where the minimum time is t_o, which is $t(nlo)$ in your script.

The relation between Absorbance and time is considered non-linear because one of the parameters that you are trying to find (i.e. the rate) is not linearly related to the absorbance. For linear least squares to be used all parameters (in the model) must be linearly related to the data.

Now we must choose good values for the three parameters. (Then we will write a function to do the simulation.)

A_{inf} , as the name implies, is $A_{inf} \sim Abs(end)$. We did this before (in Tutorial 1) and it is close enough, because we are now going to let *leasqr* find the best value for it. ΔA *or* ΔA_∞ is the difference in absorbance between infinite time and time zero (time(nlo), or t_red(1)). And r is the decay rate constant (sometimes called k). It is straightforward to eyeball the rate from the data, and a very useful thing to know in general. This is the method all engineers use to estimate a decay rate. It is like extracting the half-life, but it is the thirds-life time. Here is how it works: Pick any point on the curve (away from the infinite time end), draw a horizontal line through that point. Note the time where the horizontal line crosses the data (that is T_1). Draw a horizontal line through the end point. Now break that interval between those two lines into thirds. So you draw two more horizontal lines equally spaced between the two you have already drawn. Now find the time where the horizontal line closer to infinite time horizontal line crosses the data and note that time (as T_2). The 1/3rd time then is $T_2 - T_1$, and the rate constant (r) is the reciprocal $r \approx \frac{1}{T_2 - T_1}$. This is a pretty good way to estimate the decay rate, and with some practice can be done by eye without actually drawing all the horizontal lines. After all, it is just an eyeball method; nothing precise is needed at this point.

Put these three estimates into a vector called Params (or whatever name you like), as show above: $Params = \{A_{inf}, \Delta A, r\}$ eg P = [1.5 .9 1/100]

From the command line you can simulate the curve. With the parameters you have chosen try:

```
x = t_red   % set the reduced time to a generic variable x:
xr = x – x(1);   % this is the reduced time, with the first time removed so that xr(1)=0
Yhat = P(1) – P(2) * exp( -P(3) * xr);
plot(t_red, Yhat)   % plot Yhat over the data
```

How does the plot look? How well did you do in your estimates? Make sure this works and gives a nice plot.

Now we will convert this computation to a stand-alone function. First we need to write the function to simulate the exponential decay. Start a new .m file. The first line is

```
function [Yhat]= SimExp(x, P)
```

This line defines a function called SimExp, which expects 2 input arguments (x and P), and returns Yhat. Note that x and P are locally used variables within this function. You will be calling this function from your script using the actual variable names (t_red and Params) from the script itself. These will not necessarily match these local names. In fact, if you look at your workspace, these local variables (x and P) will not show up; you will be able to see only the variables from your script. The point of making this sort of function is that you will be able to use it from any script that you write, and send it the appropriate inputs for each script.

Now you need to put in your new function routine comments to explain what SimExp does, what x and P are and what Yhat is. We can work through this in class. Hopefully the above equation tells you what all these variables are. Below the first line, the comments should include the top line with a "%" in front to complete the explanation: % function [Yhat]= SimExp(x, P). The remainder of the comments will describe what the input and output arguments are and how to use the function. These comments become a help file for the code. When you type help SimExp from the command line you will see all of the help lines you wrote into the function. This is very handy.

Then you need one (or two) line of code (to actually compute Yhat):

```
xr = x – x(1);   % this is the reduced time, with the first time removed so that xr(1)=0
Yhat = P(1) – P(2) * exp( -P(3) * xr);
```

If you did not do this above: Take a moment to compare this line to the original equation above. Try this line on the command line. It should work, once x is properly defined.

Save it in a file with the same name as the function; this is essential and required. SimExp was just my shot at a name for this function routine, you might use something else.

Now test if the new function exists: Type in the command window:

```
>> help SimExp
```

Your help in the file should be printed out in the command window so you will know what to do in the future and how to use the function.

Now that you have successfully created the function, and verified that you can access it, we'll put in the call to the function in your script file:

```
Yhat = SimExp(t_red, Params)
```

Note that the arguments are named according to their names in the calling script. (You are not using the names which are the locally defined names in the function itself.)

If we did a half decent job in guessing the parameters, we should have a result that is close to the data. Add this to your script:

```
plot(time, Abs, 'c-', t_red, Abs_red, 'k-', t_red, Yhat, 'r-')
```

You can also use this line directly on the command line to see the result.

The model fit is a red solid line based on rough guesses for the parameters.

Now change the parameters, if you don't like them, to get a better fit. The turnaround is so fast that you can do a pretty good job even before we let *leasqr* go at it.

If you are happy with the model then you are ready to read the help on leasqr.
This is the line of code you will use:

```
[Yopt, Popt, cvc, Nitr ] = leasqr(t_red, Abs_red , Params, 'SimExp', 1e-11, 200);
```

Check out the help *leasqr* to explain what the arguments and returned variables are. The returned argument cvc tells you if it converged; this is very important. If cvc doesn't equal 1, then you'll need to try again with better starting parameters (Params). Often I just try again from where it left off, using Popt (which are better parameters than Params, we hope).

```
if (~logical(cvc) )  % if it did not converge try again; but only once
    [Yopt, Popt, cvc, Nitr ] = leasqr(t_red, Abs_red , Popt, 'SimExp', 1e-11, 200);
end
```

You should get a converged result, and now you should make a plot that overlays the result with the original data. If this did not work, you might try backing off the convergence tolerance, now set to 1e-11. Create a new figure for this plot so that you can compare it with the previous plot. Then, use the two lines of code that we used earlier, but this time substitute Popt for Params and Yopt for Yhat:

```
Yopt = SimExp(t_red, Popt)
plot(time, Abs, 'c-', t_red, Abs_red, 'k-', t_red, Yopt, 'r-')
```

This will redo the plot and replace your initial guess with the final guess. Now you can look at the results and see if you like the fit.

The Variance of the Fit is that due to Abs_red – Yopt

```
VarFit = var( (Abs_red – Yopt) ,  1);
```

The optimized fit then should be satisfactory and give you the best estimate to the rate constant, and from here we are ready to figure out the errors in the parameters.

Note for Matlab Users:

Matlab has a routine similar to leasqr. It is called nlinfit and the syntax looks like this:

```
Popt = nlinfit(t_red, Abs_red,"ML_ModExp",Params);
```

Function ML_ModExp is the same as ModExp but the x and P are switched:

i.e; function Yhat = ML_ModExp(P,x) % the first line of code; all the rest is the same.

After finding Popt, then run: Yopt = ML_ModExp(Popt,x)

End note for Matlab Users.

If you have experimental data that does not fit this simple model, please come and talk with me about it. The model can be easily modified to include aberrations, such as a sloping line of Abs near the infinite time, which can happen to experimental spectra. To demonstrate the slopping baseline problem, load this data set:

```
ds = load('Lab1DS');
```

A.2B Estimating the Errors in the fit and the parameters.

Now we will develop a way to estimate the errors in parameters.

We find the errors by looking at how the errors came about from the case where the parameters were linearly related to the data. In that case we defined a model matrix which was the counterpart to the parameter of interest. For example, in the linear least squares

$$y = mx + b$$

A row of the model matrix $M_k = [x_k \quad 1] = \left[\frac{\partial y_k}{\partial m} \frac{\partial y_k}{\partial b} \right]$. This is a single row, and the entire model matrix is developed because each row is evaluated at each value of x.

Then the variance-covariance matrix is $VcV = V \cdot \sigma_{yfit}^2$, and V is the reduced-variance-covariance matrix $V = (M^t \cdot M)^{-1}$. V is a 2 by 2 matrix in this case. For the nonlinear case we use exactly the same formulation. But now M depends on more than just the x values. It can depend on the parameters as well. But we evaluate the derivatives at the optimum values of the parameters. That is the best we can do to estimate the errors. For the case of the model above we just need to evaluate the three derivatives, working from the original expression

$$A = A_{inf} - \Delta A e^{-r(t-t_o)}$$

$$M_k = \left[\frac{\partial A_k}{\partial A_{inf}} \quad \frac{\partial A_k}{\partial \Delta A} \frac{\partial A_k}{\partial r} \right] = [1 \quad -e^{-r(t-t_o)} \quad (t-t_0)\Delta A e^{-r(t-t_o)}]$$

This matrix is surprisingly easy to code in Octave. First, make sure the time vector is a column vector:

```
t_red = t_red(:);
x_red = t_red – t_red(1);
z= exp(-r * x_red);   % here r is Popt(3)
M = [ t_red.^0     -z      x_red.*Popt(2) .*z ];
```

After you have the variables you need in the right place, the computation using M is very compact:

V= inv(M'*M);

V is now the reduced-variance-covariance matrix, and can be used as it was for the linear least squares case. Check that V is a 3 by 3 symmetric matrix.

The Variance in the Parameters then:

VarPar = VarFit*V; % these will be for Ainf DeltaA and most importantly the rate constant r

Use this to report the error in the exponential rate constant r. How confident is the answer at 95% confidence?

The variance in the model fit: VarYf = diag(M*VarPar*M')

Now, when looking at the actual numbers for the confidence in the model, using t*sqrt(VarYf), you will see they are very small. What do you make of that and is that really believable?

Use Ebars to put error bars on the data or confidence intervals on the model. The data is so tight you only want to put about 10 to 15 error bars total on your curve.

A.3 Week 3 Lab1DS data

Now consider the data set, which is in the Week3 tutorial folder, called Lab1DS. This is a data set from the kinetics lab. You can bring the data in

ds = load("Lab1DS");

This data set should have two column, time and Absorbance, as was the case with other files. However if you look at this data it has a sloping baseline, that happens from time to time (and I really don't know why, but it looks like thermal drift). Try to fit it with the exponential fit model you have already used, just to verify for yourself that this type of data is hard to analyze. Your write-up (for Experiment 1) tells you to remove this sloping baseline before you analyze it. Instead of doing that, we will include the slope in the model

The model is:

$$A = A_{\infty} - \Delta A\left(e^{-r(t-t_o)}\right) + m(t - t_o)$$

Now you have 4 adjustable parameters: $Params = \{A_{inf}\ , \Delta A\ ,\ r,\ m\}$

You can modify your existing model function or make a new model function.

Your new line of code should look about like this:

Yhat = P(1) – P(2) * exp(-P(3) * xr) + P(4)*xr;

To run the *leasqr* fitting routine you should make a guess at m, but you should **not** just set it to zero (that causes the program difficulties).

You should prepare a single script file that goes through these three sections, A.1, A.2 and A.3, and fits the two data sets to the proper model function and finds the optimum fit. Also turn in your two model functions (the first with 3 parameter and the second with 4 input parameters).

In addition to the fitting you should report the parameters from the 3 model fit with the errors you found in those parameters. You should use Ebars to plot the confidence intervals (at 95% confidence) and comment on the size of the confidence error bars. Space out the error bars so that there are 10 to 20 of them on the fit. Do you think the confidence intervals truly reflect the confidence in the parameters? Why or why not?