# **Non-linear least squares.**

In our introduction to linear algebra we have seen that a linear function *f* must fulfill two criteria:

*f*(u + v) = *f*(u) + *f*(v)

*f*(a·u) = a·*f*(u)

from which we derive the 'linear combinations':

*f*(a·u + b·v) = a·*f*(u) + b·*f*(v)

A typical linear function that fulfills these criteria is:

*f*(u) = *k*·u + *c*

which is the form we have encountered so far in the examples of linear least-squares.

In a linear function the independent variable u can only have 1 as exponent and cannot itself be in the exponent. In contrast, in a non-linear function the independent variable can be raised to powers different from 1 (including negative powers) or can itself be in the exponent.

We have already seen an example of non-linear functions in the exponential decay Problem 2 of CHAPTER 6. In that case the non-linear function of time, , was:

t = [0 5 10 15 20 25 30 35 40 45 50 55 60]'; nt = length(t)

yvec = [0.9669 0.5627 0.4608 0.2979 0.3493 0.4414 0.2387 0.2586 ...

0.0988 0.0896 0.1247 0.0378 0.03031]';

OLS\_plot = figure;

set(gcf,'Unit','Normalized','Position',[0 0.2 0.5 0.8])

plot(t,yvec\_n,'sb','MarkerSize',30)

xlim([-1 1.02\*max(t)])

xlabel('Time (minutes)');ylabel('[compound X] (microM)'); hold on

and we found the *decay rate constant* by linearizing the function to:

and solving for and by *linear least squares*.

However, most non-linear function cannot be easily linearized, and in those cases we need an alternative method to identify the *coefficients* of all the independent variables.

For example, in solving the exponential decay problem we could have started with a reasonable guess for as the largest value of , and for from the estimated half-time (~10 min), based on the known relationship:

u1 = yvec(1);

u2 = log(1/2)/10;

and calculated the corresponfing values and the error:

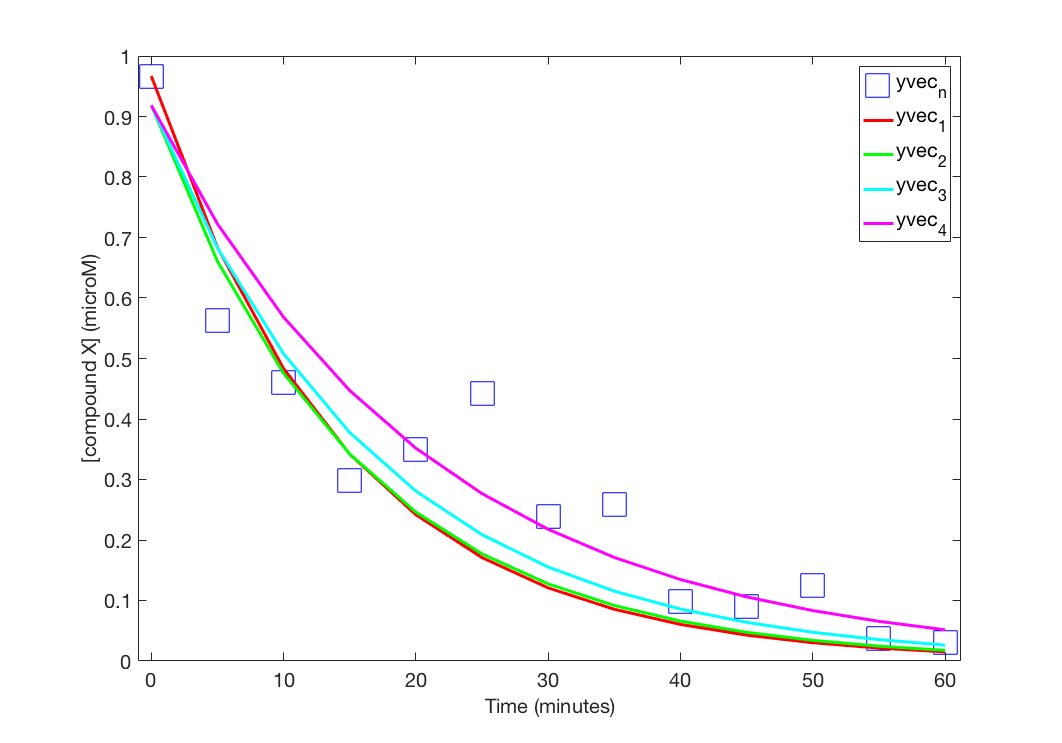
yvec1 = u1\*exp(u2\*t);

resid\_vec1 = yvec1 - yvec;

sse1 = resid\_vec1'\*resid\_vec1

plot(t,yvec1,'-r','LineWidth',3)

By observing how well the *model* derived from our initial *guessed parameters* fits the observed data we might reasonably conclude that we can get a better fit by decreasing a little (i.e., by 5%) both and , and then again just by 10% in two consecutive steps:

u1 = .95\*u1;

u2 = .95\*u2;

yvec2 = u1\*exp(u2\*t);

resid\_vec2 = yvec2 - yvec;

sse2 = resid\_vec2'\*resid\_vec2

plot(t,yvec2,'-g','LineWidth',3)

u2 = .90\*u2;

yvec3 = u1\*exp(u2\*t);

resid\_vec3 = yvec3 - yvec;

sse3 = resid\_vec3'\*resid\_vec3

plot(t,yvec3,'-c','LineWidth',3)

u2 = .90\*u2;

yvec4 = u1\*exp(u2\*t);

resid\_vec4 = yvec4 - yvec;

sse4 = resid\_vec4'\*resid\_vec4

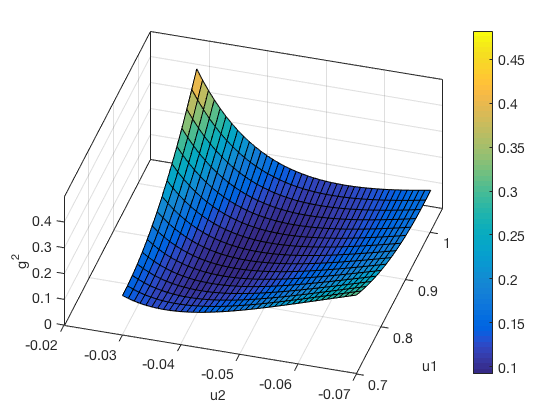
plot(t,yvec4,'-m','LineWidth',3)

legend('yvec\_n','yvec\_1','yvec\_2','yvec\_3','yvec\_4')

In these consecutive attempts, we achieve a progressive reduction in the difference between the observed data and the model, calculated with the coefficients *u1* *=**Y****0*** and *u2 =**k* of the exponential function:

***yvectrial =*** *Y0**exp(k* ***t****)*

By carrying out a small *grid* search we can easily see that in fact there exist values for *u1* and *u2* for which the squared length (the sum of squared errors, *sse*) of the difference vector ***g*** achieves a minimum:



u1 = [0.7:0.02:1.0];

u2 = [-.07:0.001:-.03];

gsq = zeros(length(u1),length(u2));

for i = 1:length(u1)

for j = 1:length(u2)

g = yvec\_n - u1(i)\*exp(u2(j)\*t);

gsq(i,j) = g'\*g;

end

end

residfun = figure;

[X,Y] = meshgrid(u1,u2);

surf(X,Y,gsq')

ylabel('u2');xlabel('u1')

zlabel('g^2');

box on

Thus, we need a method that starting from the initial difference ***g*** between the curve calculated with our guess and the experimental points, will progressively modify our estimate of *u1* *=**Y****0*** and *u2 =**k* until each element of ***g*** becomes as close as possible to 0.

This is the same as finding the *zeros* of the *residual* function:

where ***yvec*** is a vector of values that we know (e.g., it was determined experimentally and is expected to be represented correctly by a model), and ***yvectrial*** is the vector of corresponding model values that we obtain for some guess (or progressive approximation) of the unknown parameters ***u*** in the model. In other words, we are interested in finding the values of ***u*** for which the residual vector ***g*** becomes **0**. These are the *roots* of the equation:



To understand how this is accomplished let's consider a simple residual function *g*(*u*) *= f(u) - y*, in which both *g* and *u* are *scalars*, not vectors. We start with an initial guess *u1* for *u*, which gives us a value *g1**= f(u1) - y*. By taking very small changes around *u1* we calculate the derivative *dg****/****du* at *u1*. Recalling that the derivative is positive if the function increases in the neighborhood of *u1* (as *u* increases) we derive the value ***Δ****u* to be added to *u1* to get a residual *g2* < *g1*:

(*dg****/****du*) *Δu* = -*Δg*

Since we are looking for the *zero* of *g*, the equation becomes:

(*dg****/****du*) *Δu* = -*g*

If we repeat this step starting from *g2* we obtain an even smaller residual *g3*. However, it is important to understand that the residual function *g* may have *local* minima, and that a *wrong* initial guess of *u* may lead us to become *‘trapped’* in one of these minima, away from the *global* minimum of the function.

This *iterative procedure* is known as the ***Newton-Raphson*** method. It means that, for a general *vector* function of several variables , we will solve for the *zeros* of (the values of for which ), starting from an initial guess ***uk*** of the unknowns. At each iteration *k+1* we will get a ***uk+1***, which is a better value than ***uk*** until becomes **0** (with perfect data) or as close as possible to **0** with data containing errors.

In the particular case of the model function , the residual function:

is a *vector*, and ***u*** is also a vector with 2 elements, *u1* *=**Y****0*** and *u2 =**k*, for which we calculate the different values of .

In order to apply the *Newton-Raphson* method to this case, by analogy with our simple *scalar* residual function , in which we calculated the derivative of *g* with respect to a single variable *u*:

here we calculate the *partial derivatives* of with respect to each element of the initial guess ***u0***.

The matrix of the partial derivatives of with respect to each of the variables in ***u*** is called the ***Jacobian***, and is usually represented with the symbol ***J***.

**IMPORTANT**: notice that since is not a number, but a vector, the partial derivatives are themselves vectors. Each *row* of the *Jacobian* contains the partial derivatives of an element (*g1* or *g2* or *g3* or *g4*,… or *gm*) of the vector ***g*** with respect to the variables (*u1*, *u2*, *u3*, *u4*,... *un*). Therefore, each row of the *Jacobian* is a *gradient* of a component of ***g*** with respect to ***u***.

=

The non-linear least-squares process will be an iterative solution of the matrix equation:

by means of the normal equation for linear least-squares:

We notice here that if ***p*** isthe projection of ***g*** onto the *column space* of , , or equivalently, the projection of ***g*** onto the *row space* of , then by the *fundamental theorem of linear algebra*:

despite ***g*** and ***p*** being different. We also notice that the vector is a linear combination of the *rows* in the *Jacobian*, which are the gradients of the components of ***g*** with respect to ***u***.

Thus, is itself a *gradient*, and movement along its direction corresponds to a that would eliminate the residual in one step if ***g*** was a perfectly linear function. For this reason the vector is called the *steepest descent vector*.

If everything is working well the norm of should approach 0 as we approach the zeros of the residual function. Thus, we can follow the progression of this norm as a measure of the *1st order optimality condition* for finding a correct solution to the non-linear least squares fit (that is: the squared residual is at its minimum when its 1st derivative with respect to ***u*** is 0). Fullfillment of the *2nd order optimality condition* (all 2nd order derivatives of the squared residual with respect to ***u*** are positive) is not necessary in this case, because the *Newton-Raphson* iteration assures we are finding a minimum instead of a maximum.

At the end of each iteration we update the ***u*** vector by adding to it the calculated ***Δu*** vector. At the beginning of each iteration we update the ***g*** vector using the updated ***u***. We iterate until the difference in the sum of squared residual between two consecutive cycles is less than a predefined *tolerance* value, or until **,** whereis the *error tolerance*.

Now we are ready to outline the complete *Newton-Raphson* algorithm:

Start with a 'guess' or 'improved' vector

**⇓**

Evaluate

**⇓**

Calculate the *Jacobian* matrix

**⇓**

Solve for

**⇓**

Typically the partial derivatives that go in the *Jacobian* are calculated by one of the *finite difference approximations* as:

where *h* is a small increment applied to , and represents a unit vector in the direction of . Notice that the equation is a standard *linear* matrix equation of the type, and is solved the usual way by Gaussian elimination, or linear least squares if *i > j*.

The following is the complete application of the *Newton-Raphson* method to the *regression* problem of finding the coefficients of a decay model () that best fits the observed data () by minimizing the *squared length* of the residual:

We start by setting up the time vector:

t = [0 5 10 15 20 25 30 35 40 45 50 55 60]';

nt = length(t)

xvec = t;

and the vector of observed data points:

yvec = [0.9669 0.5627 0.4608 0.2979 0.3493 0.4414 0.2387 0.2586 ...

0.0988 0.0896 0.1247 0.0378 0.03031]';

nobs = length(yvec);

Here we plot the observed data:

lin\_nonlin\_lsq = figure;

set(gcf,'Unit','Normalized','Position',[0 0.2 0.5 0.8])

plot(t,yvec,'sb','MarkerSize',30)

xlim([-1 1.02\*max(t)])

xlabel('Time (minutes)')

ylabel('[compound X] (microM)')

hold on

and the linear fit:

A = [ones(nt,1) xvec];

u\_l = A\log(yvec);

u\_l(1) = exp(u\_l(1));

yvec\_l = u\_l(1)\*exp(u\_l(2)\*t);

plot(t,yvec\_l,'--r');

sse\_l = (yvec\_l-yvec)'\*(yvec\_l-yvec);

mse\_l = sse\_l/(nobs-length(u\_l));

sigma\_l = sqrt(mse\_l);

c\_yvec = yvec - mean(yvec);

sst = c\_yvec'\*c\_yvec;

rsquare\_l = 1-sse\_l/sst;

u\_cov\_l = mse\_l\*inv(A'\*A);

[Corr\_l,sigma\_u\_l] = corrcov(u\_cov\_l);

Conf\_95\_l = [u\_l-1.96\*sigma\_u\_l u\_l+1.96\*sigma\_u\_l];

legend('Exper. data','linear fit')

Next we carry out a non-linear least squares fit. We take an initial guess for the ***u0*** value from the highest observed point, and for the decay constant from the estimated half-time (~10 min), as we did previously:

u = [yvec(1); log(1/2)/xvec(3)];

nvars = size(u,1);

u1 = u(1);

u2 = u(2);

We set a very small *tolerance* value such that if the difference in the *sum of squared* residuals (*sos*) between two consecutive iterations becomes less than this value the iteration will stop. We also set the initial difference in the *sos* to some multiple of the tolerance so that the iteration does not stop immediately:

tolerance = 1E-8;

delta\_sos = tolerance\*10;

We start by calculating the value of for our initial guess ***u0***:

g = u1\*exp(u2\*xvec) - yvec;

niter = 0;

For housekeeping we store the *Jacobian* (all the gradients), the norm of the steepest descent vector , the norm of the residual , the norm of the ***Δu*** step, and the value of the error:

that is minimized in the *linear* *least square* step of each iteration:

grad = zeros(nobs,nvars,1000);

grad\_norm = zeros(1000,1);

res\_sos = zeros(1000,1);

res\_norm = zeros(1000,1);

delta\_u\_norm = zeros(1000,1);

iter\_error = zeros(1000,1);

Here we start a while loop:

while delta\_sos > tolerance

We keep track of the number of iterations required to converge.

niter = niter + 1;

We store the value from the previous cycle as :

g\_old = g;

We calculate the partial derivatives of ***g*** at ***u0*** with respect to *u1*,*u2*. In order to do this we calculate a 'centered difference' by increasing and decreasing *u1*,*u2* by a very small amount: a standard choice for the calculation of derivative is *eps*(1/3) = 6e-6, where *eps* = 2.2204E-16 is the distance from 1 to the next larger double precision number. Smaller step sizes can give origin to numerical errors. Since the elements of ***u*** can differ by order of magnitudes, we determine the proper step size for each variable by multiplying the current variable value by *eps*(1/3):

du1 = u1\*eps^(1/3);

du2 = u2\*eps^(1/3);

g1 = @(u1) u1\*exp(u2\*xvec) - yvec\_n;

g2 = @(u2) u1\*exp(u2\*xvec) - yvec\_n;

J1 = (g1(u1+du1)-g1(u1-du1))/(2\*du1);

J2 = (g2(u2+du2)-g2(u2-du2))/(2\*du2);

J = [J1 J2];

where each difference ***J1*** and ***J2*** becomes a column of the *Jacobian*.

The 'approximately' linear system is now described by the equation: ***JΔu*** = ***-g***, in which ***Δu*** is the vector of unknowns. We solve for ***Δu*** by solving the normal equation: ***JTJΔu = -JTg***. This is equivalent to minimizing ||***JΔu*** + ***g***||2.

We recall here that if we have a matrix of weights ***W***, the normal equation becomes: ***JTWJΔu = -JTWg(u)***, which is amenable to further modifications leading to the *Levemberg-Marquard* algorithm:

***(JTWJ + λI)Δu = -JTWg(u)***

in which ***-JTWg(u)*** is the new *steepest descent vector*. When ***λ*** is very large:

***Δu*** ≈ ***-(λI)-1JTWg(u)***

and ***Δu*** becomes a small fraction of the steepest descent vector: ***λ*** can be progressively reduced during the optimization until, when set to 0, it converts the process to a standard Newton method.

We rename –***g*** and we use *QR* factorization

h = -g;

[Q,R] = qr(J,0);

delta\_u = R\Q'\*h;

u = u + delta\_u;

u1 = u(1);

u2 = u(2);

Here we calculate the new residual function with the updated ***u*** vector:

g = u1\*exp(u2\*xvec) - yvec\_n;

Here we calculate the sum of squared residual using the old and new residuals, and the difference between the two values, which will be compared to the *tolerance* value.

sos\_old = g\_old'\*g\_old;

sos = g'\*g;

delta\_sos = abs(sos-sos\_old);

Table:

Jacobian

grad(:,:,niter) = J;

Norm of the steepest descent vector

grad\_norm(niter) = norm(J'\*g);

Norm of the residual

res\_norm(niter) = sqrt(sos);

Norm of the step

delta\_u\_norm(niter) = norm(delta\_u);

Function to be minimized:

1. squared error on ***–g*** at each *linear least squares* step

error\_iter(niter) = (J\*delta\_u + g\_old)'\*(J\*delta\_u + g\_old);

1. this squared error is the same as the squared residual ***g*** for the next iteration

res\_sos(niter) = sos;

end % end of the while loop

Here we plot the non-linear least squares solution:

yvec\_nl = u1\*exp(u2\*xvec);

plot(xvec, yvec\_nl,'-b')

Here we calculate the Mean Squared Error and the Standard Deviation.

mse = sos/(nobs-nvars);

sigma = sqrt(mse)

Here we calculate the covariance matrix of the solution from the last value of the Jacobian: ***JTJ*** is the *precision* matrix (the *information matrix* in linear least squares), its inverse, weighted by the mean square error, *mse* = ***σ*2**, is the *covariance* matrix of the solution ***u*** (CHAPTER 6).

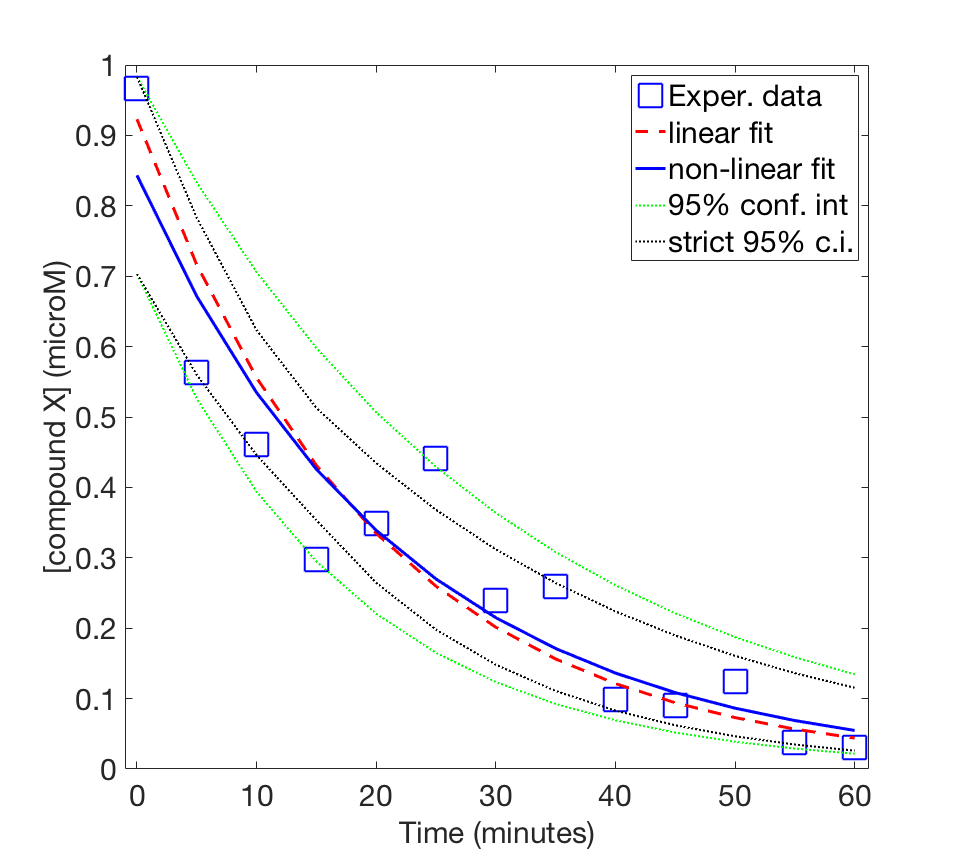
u\_cov = mse\*inv(J'\*J);

The covariance matrix can be converted into a *correlation* matrix; also the diagonal of the covariance matrix gives the variances and therefore the σ's (or *standard errors*) of the solution ***u***.

[Corr,sigma\_u] = corrcov(u\_cov);

The 95% confidence intervals is obtained by subtracting or adding 1.96\*σ to the 'expected' values of ***u***.

Conf\_95 = [u-1.96\*sigma\_u u+1.96\*sigma\_u]

Plot confidence intervals

u\_minus = u-1.96\*sigma\_u;

u\_plus = u+1.96\*sigma\_u;

yvec\_nl\_lci = u\_minus(1)\*exp(u\_minus(2)\*xvec);

yvec\_nl\_uci = u\_plus(1)\*exp(u\_plus(2)\*xvec);

Plot strict interpretation of confidence intervals

yvec\_nl\_lci1m = u(1)\*exp(u\_minus(2)\*xvec);

yvec\_nl\_lci2m = u(1)\*exp(u\_plus(2)\*xvec);

yvec\_nl\_uci1m = u\_minus(1)\*exp(u(2)\*xvec);

yvec\_nl\_uci2m = u\_plus(1)\*exp(u(2)\*xvec);

yvec\_high = max([yvec\_nl\_lci2m';yvec\_nl\_uci2m']);

yvec\_low = min([yvec\_nl\_lci1m';yvec\_nl\_uci1m']);

plot(xvec,yvec\_nl\_lci,':g','LineWidth',2)

plot(xvec,yvec\_low,':k','LineWidth',2)

plot(xvec,yvec\_nl\_uci,':g','LineWidth',2)

plot(xvec,yvec\_high,':k','LineWidth',2)

legend({'Exper. data','linear fit','non-linear fit',...

'95% conf. int','strict 95% c.i.'},'FontSize',30)

Rsquared:

rsquare = 1-sos/sst

Number of iterations

niter

We can tabulate the outcome of the NLLS iterations:

cycles = [1 : niter]

cnames = {'Iterations' 'Residual\_sos' 'Step\_norm' 'Gradient\_norm'};

NLLS\_T1 = table((cycles)',res\_sos(cycles),delta\_u\_norm(cycles),grad\_norm(cycles),...

'VariableNames',cnames)

and also compare the final result obtained with LLS and NLLS:

cnames = {'Parameter' 'LLS' 'NLLS' };

NLLS\_T2 = table(categorical({'U1';'U2';'SSE';'MSE';'SIGMA';'RSQUARE'}),...

[u\_l(1);u\_l(2);sse\_l;mse\_l;sigma\_l;rsquare\_l],...

[u(1);u(2);sos;mse;sigma;rsquare],'VariableNames',cnames)

The *non-linear least squares* (blue continuous line in the plot) shows slightly better values of **σ** and ***R2*** than the *linear least squares* solution (red dashed line in the plot) found previously. The better fit is due to the fact that in this case we are fitting the experimental points 'directly' rather than converting them to their corresponding 'linearized' values.

Now that we understand the 'principle' behind *non-linear* least-squares, it is convenient to learn how to use the routines already available in MATLAB without having to re-write an *ad hoc* minimization code each time. Several built-in functions available in different Toolboxes produce essentially the same result:

u\_zero = [yvec(1);log(1/2)/xvec(3)];

Optimization Toolbox:

functions **lsqnonlin** or **lsqcurvefit**:

options = ...

optimoptions('lsqnonlin','Display','iter','FinDiffType','central','TolFun',1e-8,'TolX',1e-8);

[u,sos,res,flag,output,lambda,J] = ...

lsqnonlin(@(u)u(1)\*exp(u(2)\*t)-yvec\_n u\_zero,[],[],options)

options = ...

optimoptions('lsqcurvefit','Display','iter','FinDiffType','central','TolFun',1e-8,'TolX',1e-8);

[u,sos,res,~,~,~,J] = ...

lsqcurvefit(@(u,t)u(1)\*exp(u(2)\*t), u\_zero,t,yvec\_n,[],[],options);

In which the empty matrices define the *lower* and *upper bounds* for the solution ***u***.

Statistics Toolbox:

functions **nlinfit, nlparci, nlpredci, nlintool,** :

modelfun = @(u,t)u(1)\*exp(u(2)\*t);

options = statset('Display','iter','TolTypeFun','abs','TolFun',1e-8,'TolTypeX','abs','TolX',1e-8);

[u,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(t,yvec,modelfun, u\_zero,options);

ci = nlparci(u,R,'jacobian',J,'alpha',0.05); % Confidence interval

[Ypred,delta] = nlpredci(modelfun,t,u,R,'Jacobian',J, 'alpha',0.05);

nlintool(t,yvec,modelfun, u\_zero,0.05,'time','[Y]');

Curve Fitting Toolbox:

function **fit**:

fo = fitoptions('Method','NonlinearLeastSquares','StartPoint', u\_zero, ...

'Algorithm','Trust-Region','Display','iter','TolFun',1e-8,'TolX',1e-8);

f = fittype('a\*exp(b\*x)','options',fo);

[decay\_model,GOF,output] = fit(t,yvec,f);

u = coeffvalues(decay\_model)'

Generic MATLAB:

function **fminsearch:** it uses a *Downhill Simplex* search (CHAPTER 17) rather than derivatives, and therefore there is no Jacobian. Since it minimizes the scalar output of a function, the model function includes the calculation of the sum of squared residuals. In this case we are converting the search for the *zero* of the residual function ***g*** into a search for the *minimum* of the squared length ***gTg*** of the residual function.

modelfun = @(u) sum((u(1)\*exp(u(2)\*xvec)-yvec).^2);

options = optimset('Display','iter', 'TolFun',1e-8, 'TolX',1e-8);

[u,fval,exitflag,output] = fminsearch(modelfun,u\_zero,options);

**PRACTICE**

**1:** We have carried out an experiment in which we followed the fluorescence changes produced by the binding of ligand L to 0.1 μM receptor R. The X-ray structure of the complex L:R in the presence of 200 μM X shows clearly that there are 2 distinct binding sites both apparently equally occupied. We have collected 30 experimental points with L concentrations logarithmically spaced between 1 μM and 200 μM. We have also collected an additional point with 3 mM L, at which concentration we are completely confident all the binding sites of R were completely saturated. Finally we have expressed the fluorescence values obtained in the concentration range xvec = 1-200 μM as a percentage of the fluorescence value at 3 mM. Our final experimental data is thus expressed as 'fractional saturation', yvec = [LR]/[Rtot]. The following are our binding data:

[xvec = [1.0000 1.2005 1.4411 1.7300 2.0767 2.4930 2.9928 3.5927 4.3129 5.1774 ...

6.2153 7.4611 8.9567 10.7522 12.9075 15.4949 18.6009 22.3295 26.8056 32.1789 ...

38.6293 46.3728 55.6684 66.8274 80.2233 96.3045 115.6092 138.7836 ...

166.6035 200.0000]';

yvec = [0.2121 0.2411 0.2724 0.3056 0.3407 0.3772 0.4149 0.4534 0.4923 0.5312 ...

0.5699 0.6080 0.6450 0.6806 0.7147 0.7467 0.7766 0.8042 0.8294 0.8522 0.8726 ...

0.8906 0.9065 0.9204 0.9324 0.9428 0.9517 0.9593 0.9658 0.9712]';

Knowing that the fractional saturation of a receptor by a ligand can be expressed as the *hyperbolic binding function*:

yvec = [LR]/[Rtot] = [L]/([L]+*Kd*)

a. Determine the *Kd* of both binding sites of R for L, assuming that both binding sites contribute equally to the fluorescence of the complex if they are equally saturated. Plot your global fit and the individual components of the fit corresponding to the two binding sites.

b. Can you distinguish your model from one in which there is only 1 binding site?

c. Compare your results with the results you would obtain with a traditional Scatchard plot. A Scatchard plot is a plot of Bound/Unbound vs Bound. With this plot the slope is -1/Kd and the intercept on the X axis is the concentration of the receptor. If there are two binding sites we should be able to recognize a break in the line going through the points of the plot. Since slope = -1/Kd, if the Kd is small (high affinity) the slope is going to be steeper: this is the left-hand side of the Scatchard plot. The site with lower affinity corresponds to the right-hand side of the Scatchard plot. Use *linear* least-squares (either writing your own function or using a MATLAB buil-in function) to fit the points in the Scatchard plot with either a single line (one site) or two separate lines (low and high affinity sites).

**2.** Calculate the 1st derivative of vector **b** (representing an absorbance spectrum) to 2nd order and 4th order accuracy using the ***finite difference method***.

b = importfile('DATABASE/Absorbance\_spectrum.txt');

b = Absorbance\_spectrum;

plot(b(:,1),b(:,2),'-r')

hold on

In the internal loop of the program we wrote for *nonlinear* least-squares we calculated the 1st derivatives using the *centered difference* method. Also all the built-in MATLAB functions for *nonlinear* least-squares use the same method, which is a subset of the more general *finite difference method*. The centered difference formula for a 1st derivative of 2nd order accuracy, , is :

where *h* is the *stepsize* between points.

This formula derives from taking an average between a *forward difference*:

and a *backward difference*:

The corresponding ***finite difference matrix*** looks like:

We use the following MATLAB syntax to generate the matrix:

A = toeplitz([0 1 zeros(1,99)]);

openvar A

A = triu(A);

A = A-A';

A\_O2 = A/2;

openvar A\_O2

% Here we calculate the 1st derivative of the spectrum with 2nd Order accuracy:

b2 = b(:,2);

b\_1st\_der\_O2 = A\_O2\*b2;

plot(b(2:end-1,1),b\_1st\_der\_O2(2:end-1,1)\*30,'-b')

% Alternatively we could have used a direct expression:

dx = 1;

dfdx = (b2([2:100]+dx)-b2([2:100]-dx))/(2\*dx)

plot(b(2:end-1,1),dfdx\*30,'-c')

% If we use the MATLAB 'gradient' function we don't lose the 1st and last point:

dfdx = gradient(b2);

plot(b(:,1),dfdx\*30,'-c')

The centered difference formula for a 1st derivative of 4th order accuracy, **O(h4)**, is:

The ***finite difference matrix*** looks like:

This matrix is pentadiagonal (its *bandwidth* is 5).

Here we calculate the 1st derivative of the spectrum with (O4):

A = toeplitz([0 8 -1 zeros(1,98)]);

openvar A

A = triu(A);

A = A-A';

A\_O4 = A/12;

openvar A\_O4

b\_1st\_der\_O4 = A\_O4\*b2;

plot(b(3:end-2,1),b\_1st\_der\_O4(3:end-2,1)\*30,'-g')

Do you recognize what is the problem in calculating directly the derivative of an experimental spectrum?

The data is very 'noisy': it is important to first get a 'smooth' model of the data and then to calculate the derivative. For higher order derivatives it is best to first smooth the lower order derivative and then to calculate the higher order one using the same finite difference matrix.

a. Do a search inside MATLAB help library or on the web to find functions that can be used to smooth your spectrum.

b. Calculate both 1st and 2nd derivative of the spectrum doing a smoothing step before each derivative calculation.

One typical interpretation of our analysis would be that in the spectrum there are several chemical species whose absorption peaks corresponds to the negative peaks in the 2nd derivative of the spectrum.

**3.** Calculate the 1st and 2nd derivative of the function *F(x)* shown below using anonymous functions and complex step differentiation (CSD):

F = @(x) exp(x)./((cos(x)).^3 + (sin(x)).^3)

The following are the *complex step differentiation* algorithms for 1st and 2nd derivative:

**1st derivative:**

a. Compare this algorithm to the *centered difference*: . Since in CSD there are no differences, the terms with order h2 or higher can be ignored and the interval h can be chosen up to machine precision (i.e. ). Therefore all that is needed is O(h2) accuracy with extremely small steps.

**2nd derivative:**

The 2nd derivative by CSD is faster than that by *central finite differences*:

but a difference is again required, which makes the algorithm unstable for h smaller than 10-6.

h0 = ezplot(F,[-pi/4,pi/2])

set(h0, 'Color','b');

axis([-pi/4,pi/2,0,6])

set(gca,'xtick',[-pi/4,0,pi/4,pi/2])

line([pi/4,pi/4],[F(pi/4),F(pi/4)],'marker','.','markersize',18)

b. Recalling how we derived the central finite difference matrix for the 1st derivative:

derive the central finite difference matrix for the 2nd derivative: