STAT0023 Workshop 4: Optimisation, maximum likelihood and nonlinear least squares

This week we'll look at the basic Newton-Raphson algorithm for finding the maximum or minimum value of a function, as well as the `nlm()` routine in R that can be used to minimise fairly complicated functions in applications such as maximum likelihood or least squares estimation. Most of the background and theory is summarised in the lecture slides / handouts, available from the 'Course overview and useful materials' tab on the course Moodle page.

1 Setting up

For this workshop, in addition to the lecture slides / handouts you will need the following, all of which can be downloaded from the 'Week 4' tab of the Moodle page:

- These instructions.
- The R scripts `QuarticExample.r` and `tploglik.r`, and the data file `nls2.dat`. You should save these to the appropriate place on your `N:` drive, as you created it during Workshop 1. Refer to Section 1 of the notes for Workshop 1 if necessary — and remember to make sure that your web browser doesn't change the filenames when downloading them (Section 1.3 of the Workshop 1 notes).

Having downloaded the files, start up RStudio and use the `Session` menu to set the working directory appropriately, as usual.

2 Numerical Function Minimisation

2.1 Newton-Raphson Method

In the lecture, you saw that the Newton-Raphson algorithm is the basis of many optimisation routines in modern software. To maximise or minimise a function $h(x)$ of a single variable x, starting with an initial guess x_0 at the location of the optimum, the algorithm works by approximating $h(x)$ with a quadratic function $H(x)$ in the neighbourhood of x_0; finding the minimum of the approximating quadratic (which is easy); and taking this as an improved estimate of the optimum of the original function $h(x)$. See the lecture slide 'Newton-Raphson for functions of one variable' for the mathematical details, along with some diagrams illustrating the intuition behind the method.

To optimise a function $h(x)$, the algorithm is as follows:

Step 1: Find expressions for the first and second derivatives $h'(x)$ and $h''(x)$.

Step 2: Choose an initial guess x_0 at the location of the optimum, and set $i = 0$. The quantity i will be used to keep track of how many iterations are performed.

Step 3: Calculate $x_{i+1} = x_i - h'(x_i)/h''(x_i)$, and increment i by 1 (i.e. add 1 to i, to record the fact that another iteration has taken place).

Step 4: If some stopping criterion is satisfied then stop; otherwise return to step 3.

When this algorithm completes, the value of i will tell you how many iterations were performed and the value of x_i will be your estimate of the optimum. In Step 4, some commonly used stopping criteria are:

Rule 1: Stop when the relative change in h becomes small: $|\frac{h(x_{i+1})-h(x_i)}{h(x_i)}| < \epsilon$ where ϵ is a small number like 10^{-6};

Rule 2: Stop when the relative change from x_i to x_{i+1} becomes small: $|\frac{x_{i+1}-x_i}{x_i}| < \eta$ where η is another (or perhaps the same) small number;

Rule 3: Stop when the number of iterations i reaches a fixed number N, which is usually chosen to be large enough that the algorithm has a reasonable chance of **converging** (i.e. stopping because one of the other criteria is met) first.

In the third case — where the algorithm stops because it exceeds the maximum number of iterations N — there is no guarantee that x_N is an approximate optimum of h. In this case we say that **the algorithm has not converged**.

In the first and second cases, the process will stop when a local minimum or maximum is reached, but there are functions for which it may converge at a *false optimum* (e.g. an inflection point) or alternatively functions for which it does not converge at all (e.g. $H(x) = e^x$ has no finite minimum / maximum). There are different ways of detecting when such problems occur: the simplest is to repeat the optimisation with different starting values x_0 and see if you get the same answer each time.

It is usual to use both Rule 3 with a relatively large value of N to prevent the process from "running away" and, either 1 or 2 to stop when we have a suitable approximation. Notice, however, that Rules 1 or 2 will be problematic if either x_i or $h(x_i)$ is close to zero: modern software packages will therefore either modify the rules slightly to overcome this problem, or supplement the rules with other criteria based on absolute, rather than relative changes.

Notes on the Newton-Raphson algorithm

- The algorithm is designed to solve the equation $h'(x) = 0$: thus, when it converges it may be to either a maximum or a minimum.
- If you look at the structure of the algorithm, you'll see that it can be implemented using a 'while' loop (see Workshop 3): the iterations proceed 'while' none of the conditions in Rule 1, Rule 2 or Rule 3 are satisfied.
- The algorithm can be generalised to functions of several variables, using the multivariate version of Taylor's theorem to obtain the approximating quadratic function.

2.2 Example: minimisation of a simple function

Consider a simple quartic function

$$h(x) = x^4 - 2x^3 + 3x^2 - 4x + 5\ . \tag{1}$$

This has derivative

$$h'(x) = 4x^3 - 6x^2 + 6x - 4 = (x-1)(4x^2 - 2x + 4) \tag{2}$$

and second derivative

$$h''(x) = 12x^2 - 12x + 6. \tag{3}$$

A plot of these three functions is given on the lecture slide 'Newton-Raphson example: $h(x) = x^4 - 2x^3 + 3x^2 - 4x + 5$'.

From equation **(2)**, you can see that the equation $h'(x) = 0$ has one (real) root at $x = 1$ (the quadratic factor has complex roots). Putting $x = 1$ into **(3)** yields $h''(1) = 3 > 0$, so $h(x)$ has a minimum at $x = 1$. Let's pretend we don't know this, and that we want to use the Newton-Raphson algorithm to find the minimum.

The Newton-Raphson update formula is $x_{i+1} = x_i - h'(x_i)/h''(x_i)$ which, for this example, yields

$$x_{i+1} = x_i - \frac{4x_i^3 - 6x_i^2 + 6x_i - 4}{12x_i^2 - 12x_i + 6}\ ,$$

from equations **(2)** and **(3)**. Let's take an initial value of $x_0 = 0$, then

$$x_1 = 0 - \frac{-4}{6} = \frac{2}{3}.$$

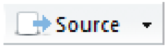
So our first "approximation" is that the minimum of h is at $\frac{2}{3}$. This is closer to the true minimum than the initial value, but it's still quite a big error. Let's do another iteration therefore:

$$x_2 = x_1 - \frac{4x_1^3 - 6x_1^2 + 6x_1 - 4}{12x_1^2 - 12x_1 + 6} = \frac{2}{3} + \frac{4}{9} = 1\frac{1}{9}.$$

Hence the second "approximation" is at $1\frac{1}{9}$ which is closer still to the true value.

Exercise 1

The R script `QuarticExample.r`, which you downloaded earlier, contains a function called `QuarticMin()` which implements the Newton-Raphson algorithm for this particular quartic function. Proceed as follows:

1. Open the script by clicking on it in the 'Files' tab in the bottom right-hand pane in RStudio; and then click the  button to run it. You will see the `QuarticMin` object in your workspace.

2. Before looking to see how the function works, type

   ```
   > QuarticMin(0)
   ```

   at the Console prompt. The '0' here is the initial value $x_0 = 0$. When the function has finished its work, you may need to scroll up a bit to see all of the output. You will see the 'iteration history' on the screen: check the results for the first two iterations against the manual calculations above. After this, you will see the **output** of the function:

   ```
   $x
   [1] 1

   $hx
   [1] 3

   $gradient
   [1] 9.865989e-08

   $N.iter
   [1] 5
   ```

   This says that the function minimum is at $x = 1$, where the value of the function is $h(x) = 3$ and the gradient is 9.9×10^{-8}; and that this minimum was found in 5 iterations.

3. Now it's time for you to look and see how the function works. Read through its definition and the comments. Note the following:

   - The arguments to the function are `x0` (the initial value); `tol` (this corresponds to the value for both ϵ and η in Rules 1 and 2 above); `MaxIter` (this is the value of N in Rule 3); and `Verbose` (which controls whether or not the algorithm writes detailed progress to screen as it works). There are default values for all of the arguments except `x0`: thus, the command `QuarticMin(0)` above works with `tol=1e-6`, `MaxIter=100` and `Verbose=TRUE`.
   - The function returns a **list** result, with components `x`, `hx`, `gradient` and `N.iter`. You can see these in the output above. This list is defined in the final line of the function (remember that the value of any R function is defined by the last statement executed — see the slides from Lecture 3 if you've forgotten this). The advantage of returning the results as a list is that you can access them separately if you assign the value of the function to an object. For example, if you type

     ```
     > fish <- QuarticMin(0)
     ```

     then you can access the different components using expressions like `fish$x`, `fish$gradient` and so on.

- The main work of the function is done using a `while()` loop: the expression `abs(dh.dx) > tol & abs(RelChange) > tol & Iter < MaxIter` will be `TRUE` if the gradient is bigger than `tol` **and** the relative change in the estimate is bigger than `tol` **and** the iteration count has not yet reached `MaxIter`. Some preliminary initialisation is needed, to ensure that all of the quantities in this expression are defined on entry to the loop.
- The `if (Verbose) { ... }` sections ensure that progress is written to screen only if `Verbose` is `TRUE`.

If you don't understand anything about how the function works, ask!

4. When you've finished reading through the function and you have a reasonable idea how it works, try running it again with a different starting value. You might not want to see the details of all the iterations from now on, so try something like

```
> QuarticMin(1, Verbose=FALSE)
```

How many iterations are needed here? Why? Then try starting values like 10, 100, $-1000, \ldots$. Does the algorithm always converge? What happens to the iteration count as the starting value gets further from the true minimum?

2.3 Minimisation in R

In the lecture you saw that minimisation in R can be done using the `nlm()` command, which stands for "non-linear minimisation". According to the R documentation, `nlm()` implements a "Newton-type algorithm." It's not **exact** Newton-Raphson — some differences compared with the algorithm as outlined above are:

- By default, `nlm()` **estimates** the gradient (first derivative) and **Hessian** (second derivative) at each iteration. This saves you from having to code the derivatives h' and h'', at the cost of (slightly) slower convergence.
- The true Newton-Raphson method will converge to either a local maximum or a local minimum, but we don't know which one we'll get in advance. `nlm()`, however, always converges to a (local) **minimum** if it can find one.
- `nlm()` contains some built-in safety checks to prevent the algorithm from 'running away', for example if it starts to take bigger and bigger steps instead of smaller and smaller ones (this can happen if you try to minimise a function that doesn't have a finite minimum).

To use `nlm()` in R you should follow three steps, as follows:

Step 1: Define the function to be minimised. This should be defined as an R function.

Step 2: Call `nlm()` with an appropriate starting value.

Step 3: Check that `nlm()` has converged to something sensible.

Exercise 2

We will test `nlm` with the quartic function **(1)** from the previous exercise. Here are your instructions:

1. Open a new R script in your RStudio session (on the `File` menu, select `New File` and then `R Script`). In this script, define a new R function:

   ```
   quartic <- function(x) {
     x^4 - (2*x^3) + (3*x^2) - (4*x) + 5
   }
   ```

   Save the file, giving it a sensible name (don't overwrite any other files!); and then click the Source button to run it. You will see the `quartic` object in your workspace. This is 'Step 1' above.

2. Call `nlm()` with an appropriate starting value ('Step 2'). For comparison with the previous exercise, we'll choose the starting value $x_0 = 0$:

   ```
   > nlm(quartic,0)
   $minimum
   [1] 3

   $estimate
   [1] 0.9999996

   $gradient
   [1] 5.968559e-07

   $code
   [1] 1

   $iterations
   [1] 6
   ```

That's all! Of course, you need to know how to interpret the results. Just like our own function in Exercise 1, `nlm()` returns a **list** result, with five elements:

- `minimum`, the value of `quartic` at the estimated minimum;
- `estimate`, the estimated value of `x` which minimises `quartic`;
- `gradient`, the estimated gradient of `quartic` at the estimated minimum;
- `code`, a value giving the reason why the iterative process has stopped (see below); and

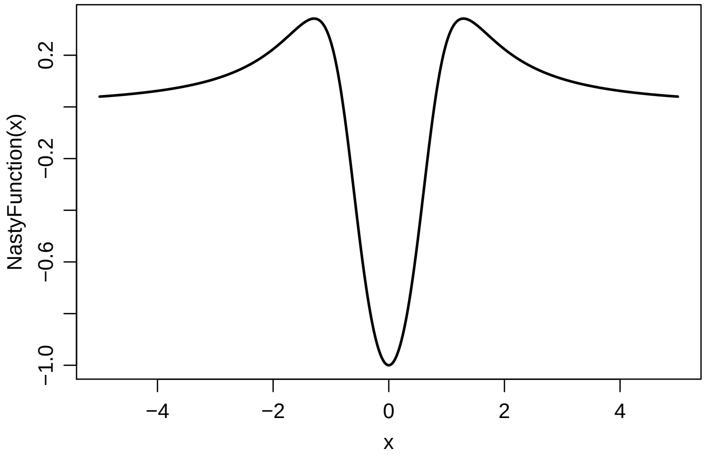


Figure 1: Plot of the function considered in Exercise 3.

.

- `iterations`, the number of iterations made.

The value for `code` indicates the reason that the minimisation routine stopped. These include the reasons given in Rules 1–3 (Section 2.1), along with two others. The bottom line is that if `code` is 1 or 2 then the result is **probably** reliable; if `code` is 3 then the result **may** be reliable; and if `code` is 4 or 5 then the result is unreliable. See the lecture slides ('Checking convergence with `nlm()`'), as well as the `nlm()` help page, for more details.

As well as checking that `code` is 1 or 2, it is good practice to check that the same result is obtained by using different starting points. Try `nlm(quartic,10)`, `nlm(quartic,-10)` etc. to reassure yourself that `nlm()` does indeed find the correct minimum regardless of where you start from (within reason!).

Exercise 3: a nasty function

You have now seen how optimisation works for a nicely behaved quartic function. Now let's see what can go wrong, by considering the function

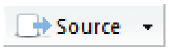
$$h(x) = \left(x^2 - \frac{1}{1+x^2}\right)\left(1+x^4\right)^{-1} \ .$$

A plot of this function is given in Figure 1. It has a global minimum at $x = 0$, with $h(0) = -1$. But: can you see the potential problem for a numerical optimisation routine? Let's see how `nlm()` copes with it:

- Open a new script and define the function to R. The code is:

```
NastyFunction <- function(x) {
  (x^2 - (1/(1+x^2)) ) / (1 + x^4)
}
```

Be careful with the brackets! Spacing the code out can help to protect against mistakes.

- Save the file, giving it a sensible name; and then click the  button to run it. If you get any errors, correct them and try again. If there are no errors, you'll see the `NastyFunction` object in your workspace.

- To check that you've defined the function correctly, plot it — it should look like Figure 1. A really easy way to do this is to go `curve(NastyFunction,-5,5)`: this plots the function from $x = -5$ to $x = +5$. You might have noticed the vertical axis label in Figure 1: now you know where it came from! If your plot **doesn't** look like Figure 1, go back and correct your function definition.

- Now call `nlm()` with an initial value $x_0 = 1$. Does it converge to the correct value? What value of `code` do you get? How many iterations were there?

- Call `nlm()` with an initial value $x_0 = 2$. What happens this time? Notice in particular the iteration count, and the value of `code`.

The purpose of this example is to demonstrate that even if the `code` from `nlm()` is 1 or 2, this doesn't **guarantee** that the result is reliable (can you understand why `nlm()` **thought** that it had found a minimum?). Plotting the function, and trying the optimisation from different starting points, are helpful ways to protect against this kind of thing.

3 Maximum Likelihood

In the lecture, you learned how to use maximum likelihood estimation for a truncated Poisson distribution. Now we will repeat this exercise, but using a different dataset. You might want to open a new R script for this exercise so that you can type your commands, correct any errors as you go and then save your work when you've finished. Here are your instructions:

1. Define the data to R using

   ```
   > TPdata <- rep(1:11, c(17,48,68,71,42,19,14,7,1,0,1))
   ```

   This command takes each element of the vector `1:11` and repeats it according to the frequencies given in the vector `c(17,48,68,71,42,19,14,7,1,0,1)`: thus, the result contains 17 ones, 48 twos, 68 threes and so on.

2. Obtain a frequency table of these data using:

   ```
   > table(TPdata)
   ```

3. Obtain the sample mean of the data. You should know how to do this, so fill in the ???????? for yourself:

```
> ybar <- ????????
> cat(paste("Sample mean is",round(ybar,3),"\n"))
```

4. Define a function to evaluate the negative log-likelihood of a truncated Poisson distribution: the mathematical form of this is given on the lecture slide 'Truncated Poisson example: MLE in R', and the R code is on the subsequent slide. To make your lives easier however, the code is provided to you here in file `tploglik.r` that you downloaded earlier. Open this and run it, to make the function available to your R session. You should know how to do this . . .

5. Output the function to the screen using

   ```
   > tploglik
   ```

6. Produce a plot of the negative log-likelihood against theta.

   ```
   > theta <- seq(from=0.1,to=7,by=0.1)
   > plot(theta, tploglik(theta,TPdata), type="l",ylab=expression(-log*L(theta)))
   ```

   (note: it's an '`l`' for 'lines', not a '`1`' for 'one'!)

7. Now run `nlm` using a starting value of 2.

   ```
   > nlm(tploglik,2,y=TPdata)
   ```

   Check that the convergence code has a value of 1 or 2. What is the estimate of θ? Is the answer close to the value you were expecting?

8. Re-run `nlm` using starting values of 7 and then 0.1, again checking the convergence code. Do you get the same answer? How many iterations does it take for the program to converge now?

In the final step above, notice that you get some warning messages. Can you suggest why these warnings occur? Do you have any idea how to prevent them (apart from turning off the warning system in R, which is cheating!).[1]

3.1 Standard errors of maximum likelihood estimates

Maximum likelihood theory tells us that in sufficiently large samples the **maximum likelihood estimator** (MLE) has approximately a normal distribution:

$$\hat{\theta} \xrightarrow{d} N(\theta, I(\theta)^{-1}),$$

[1] `options(warn=-1)` will do it, but don't! If you've just done it, then type `options(warn=0)` to turn the warning system on again.

where $I(\theta)$ is the (*expected*) Fisher information

$$I(\theta) = \mathsf{E}\left[-\frac{\partial^2}{\partial\theta^2}\ell(\theta;\mathbf{Y})\right] \tag{4}$$

and $\ell(\theta;\mathbf{Y})$ is the log-likelihood function. The quantity inside the square brackets [ ] in (4) is the second derivative of the negative log-likelihood. In large samples therefore, the variance of the MLE is Var$(\hat{\theta}) \approx I(\theta)^{-1}$ and the standard error is approximately $I(\theta)^{-1/2}$. You have covered this in previous classes (STAT0005 or equivalent). From this, we can calculate an approximate 95% confidence interval for θ as $\hat{\theta} \pm 1.96I(\theta)^{-1/2}$.

If we want to calculate a confidence interval in practice however, we are faced with two problems. First, we don't know the value of θ; and second, it may not be easy to calculate the expectation in (4).

In practice, the first of these problems can be solved by using the MLE $\hat{\theta}$ instead of θ; and the second can be solved by using the **observed** instead of expected information (i.e. omit the expectation from equation **(4)**). It can be shown that this is legitimate in large enough samples, in the sense that the approximations are negligible with high probability. Therefore, instead of using $I(\theta)$ to compute standard errors, we can instead use

$$J(\hat{\theta}) = -\left\{\frac{\partial^2}{\partial\theta^2}\ell(\theta;\mathbf{Y})\right\}\Big|_{\theta=\hat{\theta}} . \tag{5}$$

This is called the (*observed*) Fisher information $J(\theta)$, evaluated at $\theta = \hat{\theta}$. The standard error of $\hat{\theta}$ is then estimated as $\left[J\left(\hat{\theta}\right)\right]^{-1/2}$.

This sounds complicated. However, $J(\hat{\theta})$ is nothing more than the second derivative — or **Hessian** — of the negative log-likelihood, evaluated at the minimum. If you look back to the `QuarticMin()` function from Exercise 1 in Section 2, you'll see that the Newton-Raphson algorithm computes the second derivatives as it goes — so if we can access the final calculated second derivative, we should be able to compute the standard errors almost 'for free'. So: just call `nlm()` with an additional argument `hessian=TRUE`. If you do this, the result will contain an additional element called `hessian`, which is exactly $J(\hat{\theta})$.

To implement this for the truncated Poisson example, here are your instructions:

1. Type

   ```
   > TP.fitted <- nlm(tploglik, 3.7, y=TPdata, hessian=TRUE)
   ```

   The result is a list object, as before. Type `TP.fitted` to see the entire object, or `TP.fitted$estimate`, `TP.fitted$hessian` and so forth to see different components of it.

2. To obtain estimates of the variance and standard error of $\hat{\theta}$, you can now use `1/TP.fitted$hessian` and `1/sqrt(TP.fitted$hessian)` respectively.

3. Use the standard error of $\hat{\theta}$ to compute an approximate 95% confidence interval for θ (the formula is given above — if you didn't notice it, you're not reading carefully enough!).

4 Non-Linear Least Squares

The final example in this week's workshop is on nonlinear least squares. We'll look at an example similar to the one in the lecture: the data are in file `nls2.dat` that you downloaded earlier. As in the lecture example, there are two variables x and Y, and the model to be fitted is

$$Y_i = \beta_0 e^{\beta_1 x_i} + \epsilon_i \qquad (i = 1, \ldots, n)$$

with $\epsilon_i \sim N(0, \sigma^2)$ independently for each observation. The model can be fitted using nonlinear least-squares: the estimates of β_0 and β_1 are chosen to minimise

$$h(\beta_0, \beta_1; \mathbf{x}, \mathbf{Y}) = \sum_i (Y_i - \beta_0 e^{\beta_1 x_i})^2 \ ,$$

where $\mathbf{x}$ and $\mathbf{Y}$ represent vectors containing **all** of the data values for the respective variables.

As with the previous exercise, you might want to open a new R script for this exercise, in order that you can save your commands once everything is working. Here are your instructions for reading the data, plotting them and fitting the model using nonlinear least squares:

1. Read the data into R using

   ```
   > NLSdata <- read.table(file="nls2.dat", header=TRUE)
   ```

   Use the `str()` command (Workshop 1) to obtain some information on the `NLSdata` object. How many observations are there in total?

2. Plot the data:

   ```
   > plot(NLSdata$x, NLSdata$Y)
   ```

3. Plot the data again, but this time using blue filled circles as plotting symbols; with the x- and y-axes labelled as 'x' and 'Y' respectively; and with a title 'Data for nonlinear regression example'. You may need to refer back to previous workshops to help with this, and to the help page for the `points()` command to find out how to specify filled circles as plotting symbols.

4. Ignoring the error term, the model says that $\mathbb{E}\left(Y_i\right) = \beta_0 e^{\beta_1 x_i}$. If you want to avoid nonlinear least squares, you might think that you could linearise the relationship by taking logs: $\log \mathbb{E}\left(Y_i\right) = \log \beta_0 + \beta_1 x_i$. This might suggest that you could analyse the data by regressing $\log(Y)$ on x. To see if this is sensible, produce a plot:

   ```
   > plot(NLSdata$x, log(NLSdata$Y))
   ```

   Would it make sense to fit a linear regression model of $\log(Y)$ upon x?

5. The answer to the last question is ‘no’. Nonetheless, to give yourselves some practice at using material that you have learned already, go ahead and fit a linear regression model of $\log(Y)$ upon x using the `lm()` command. Make a note of the estimated coefficients: they will be useful later on. Also, produce some residual plots. Are the model assumptions satisfied? (if you don’t know, you need to do more of the **Get your eye in** Moodle quizzes from Week 2).

6. Having decided that nonlinear least-squares estimation is unavoidable, we need to follow Steps 1–3 from Section 2.3. Step 1 is to define an R function that evaluates the sum of squares:

   ```
   sumsqerr <- function(theta,x,Y) {
     beta0 <- theta[1]
     beta1 <- theta[2]
     mu <- beta0*exp(beta1*x)
     sum((Y-mu)^2)
   }
   ```

   We’re not giving you this one to download: you have to type it yourselves! Notice that the parameters β_0 and β_1 are supplied to this function in a **vector** called `theta`, containing two elements: the first two lines of the function simply extract these two elements and assign them to objects called `beta0` and `beta1`, because this improves the readability of the subsequent code. The only reason for bundling the coefficients into a single vector is that `nlm()` insists on it (see below).

7. Step 2 of the `nlm` recipe is to identify a suitable starting value and then to call `nlm()`. So far, we haven’t said much about starting values. In general however, it can’t be bad to choose something that is fairly close to the actual minimum. Often, ‘simple but not quite right’ methods can help to identify useful starting values. This is where your earlier `lm()` fit comes in useful. If you did it correctly, you should find that the estimated intercept and slope are 0.3 and -0.3 respectively, to one decimal place. Referring back to item 4 in these instructions, you find that the intercept is a rough estimate of $\log \beta_0$ so that β_0 itself is around $\exp(0.3) \approx 1.3$; and that the slope is a rough estimate of β_1. **We have some defensible starting values!** Now call `nlm()`:

   ```
   > NLS.fit <- nlm(sumsqerr, c(1.3,-0.3), x=NLSdata$x, Y=NLSdata$Y, hessian=TRUE)
   > NLS.fit
   ```

   What is the convergence code? Does it look OK? How close is the estimate to the ‘naïve’ value that you obtained from `lm()` previously (i.e. the starting value)?

   Notice that the second argument to the `nlm()` call above is a **vector** containing two values. This is because there are two coefficients to be estimated: `nlm()` will always try to minimise with respect to the values provided in its second argument, so if we want to minimise with respect to two or more quantities then we must supply them as a single vector.

8. In "simple" situations like this, another way to check convergence is just to plot the fitted function. Here is some code that will do this for you:

```
> beta0 <- NLS.fit$estimate[1]
> beta1 <- NLS.fit$estimate[2]
> x.grid <- seq(0,10,0.1)
> mu <- beta0*exp(beta1*x.grid)
> plot(NLSdata$x, NLSdata$Y, xlab="x", ylab="Y",col="blue",pch=16)
> lines(x.grid,mu,col="red")
```

   Don't just type the code and run it: make sure that you understand it. Does the fit look reasonable?

You might wonder about how to calculate standard errors and confidence intervals for nonlinear least-squares parameter estimates such as those above. This is beyond the scope of STAT0023: however, it is **relatively** straightforward because it can be shown that under the model assumptions, the negative log-likelihood is proportional to the sum of squared errors. This means that we can apply the previous theory involving the Fisher information, with small modifications, to find the approximate standard errors of nonlinear least-squares estimators. One additional complication in the present example is that there is more than one parameter. This means that the observed information **(5)** is now a **matrix** of second derivatives, with (i, j)th element $-\partial^2\ell/\partial\theta_i\partial\theta_j$; and that the approximate standard errors of the parameter estimates are the **square roots** of the **diagonal elements** of the **inverse** of this matrix (if you understand the mathematics, this will make sense ...). To obtain the matrix of second derivatives you need the `hessian=TRUE` argument in the `nlm()` call (see example below); the `solve()` command inverts a matrix, and the `diag()` command extracts the diagonal elements.

5 Moodle quiz

Again, there's just one Moodle quiz this week. You know what to do.