Part A Simulation and Statistical Programming HT15

Lecturer: Geoff Nicholls

University of Oxford

Lecture 8: Modularity; Vectorisation; Handling Data

Notes and Problem sheets are available at

www.stats.ox.ac.uk\ ∼nicholls\PartASSP

Overview

1. Using functions to write code that works
2. For-loops and `apply()`

3. Reading and writing data.
4. Attaching data frames to the search path.
5. Further operations on data: subsetting.
6. Plotting data
   (a) Histograms and overlays.
   (b) Boxplots and the R Formula notation.
   (c) Point plots
7. Debugging*, Saving a workspace*

More functions

In the last practical I asked you to write some code to simulate a normal by rejection from a double exponential. Recall the algorithm

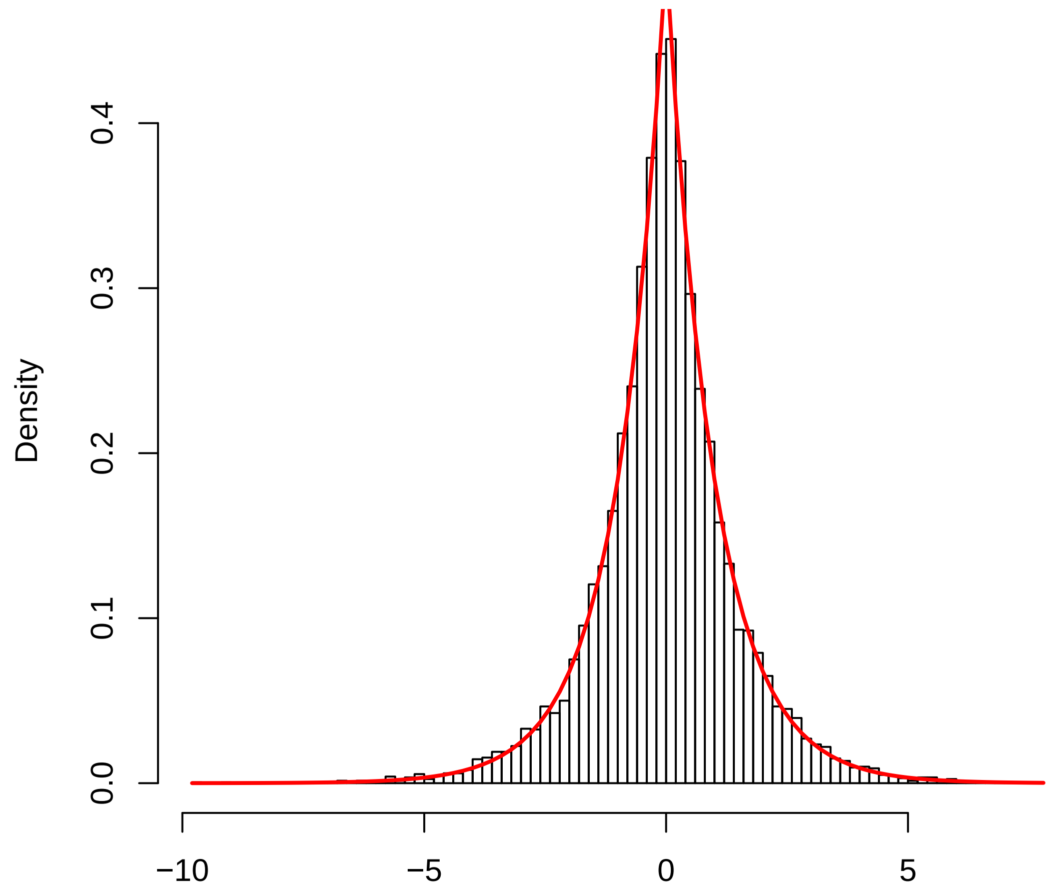
[1] simulate $y \sim \exp(-|x|)$ and $u \sim U(0,1)$
[2] if $u < \exp(-y^2/2 + |y| - 1/2)$ accept $X = y$ and stop. Otherwise repeat [1].

We break this job down into manageable pieces.

First write a function to sample $y \sim \exp(-|x|)$ - and test it!

```
q<-function(n=1) {
  #simulate exp(-|x|)

  X<-log(runif(n))

  Y<-sample(c(-1,1),n,replace=T)

  return(X*Y)

}

> q(n=4)
[1] -0.2589713443  0.0043643318  0.0007715165 -1.4679607563
```

```
> y<-q(1e4)

> a<-hist(y,100,freq=F)

> lines(a$breaks,0.5*exp(-abs(a$breaks)),col=2,lwd=2)
```



Then write a function implementing the rejection sampler.

[1] simulate $y \sim \exp(-|x|)$ and $u \sim U(0,1)$

[2] if $u < \exp(-y^2/2 + |y| - 1/2)$ accept $X = y$ and stop. Otherwise repeat [1].

```
my_rnorm<-function() {
   #simulate X
   finished<-FALSE;
   while (!finished) {
      y<-q(n=1);
      u<-runif(n=1)
      p.over.Mq<-exp(-y^2/2+abs(y)-0.5)
      finished<-(u<p.over.Mq)
   }
   return(y)
}
```

I illustrated how to test this in the practical solutions.

Try to write your code so it is easy to check and maintain. We mentioned modularity (breaking code elements up into functions). Information hiding (dont let a function have information it doesnt need) makes code easier to read and correct. Careful planning and commenting help. Give variables meaningful names.

For-loops and `apply()`

We try to avoid for-loops when we code. For-loops tend to run slower and are harder to read.

```
#cumulative sum
n<-10; X<-rep(1,n); for (i in 2:10) X[i]<-i+X[i-1]; X
X<-cumsum(1:n); X

#sum x^2 x=1...n
tot<-0; for (x in 1:10) tot<-tot+x^2
sum((1:10)^2)
```

`apply(object,row/col=1/2,function)` is often useful in this context. It applies the function to all the entries in each row or column of the object.

```
>  X<-matrix(c(1,2,3,4),2,2); X
     [,1] [,2]
[1,]    1    3
[2,]    2    4
>  apply(X,1,sum) #sum rows
[1] 4 6
>  apply(X,2,max) #sum columns
[1] 2 4
>  how many entries in each column less than 2.5?
>  apply(X,2,function(x) {sum(x<2.5)})
[1] 2 0
```

We can often use `apply()` to replace for-loops

```
#simulate n Gamma(a,b)
a<-3; b<-4.1; n<-10; X<-numeric(n)
for (i in 1:n) {
   tot<-0
   for (j in (1:a)) {
        tot<-tot-log(runif(1))/b
   }
   X[i]<-tot
}
X

#or
M<- -log(matrix(runif(a*n),a,n));
X<-apply(M,2,sum); X
```

Getting Data into R

`hellung.txt` is text file of data from an experiment on the growth of Tetrahymena cells.

The cell concentration (**`conc`**) was set at the beginning of the experiment and the average cell diameter (**`diameter`**) was measured for two groups of cell cultures where glucose was either added (**`glucose=1`**) or not added (**`glucose=2`**) to the growth medium. A theoretical model predicts diameter proportional to the log of concentration. The effect of glucose on the cell diameter is of interest.

The data are a table, with one row for each observation and a column for each variable. The first row gives variable names.

We can use the **`read.table()`** function to read it. We can read over the net or from a local file. If we use a local file we have to give the path, or set the working directory os the directory containing the file.

This table has a header line, so we tell R to expect a header.

```
hd=read.table('hellung.txt',header=TRUE)
```

creates a **`data.frame`** with cases corresponding to rows and variables to columns in the file.

If we have a data frame and want to save it we have a few options. To save it as an array of numbers with variable names as column headers

```
write.table(hd,'filename.txt',header=TRUE)
```

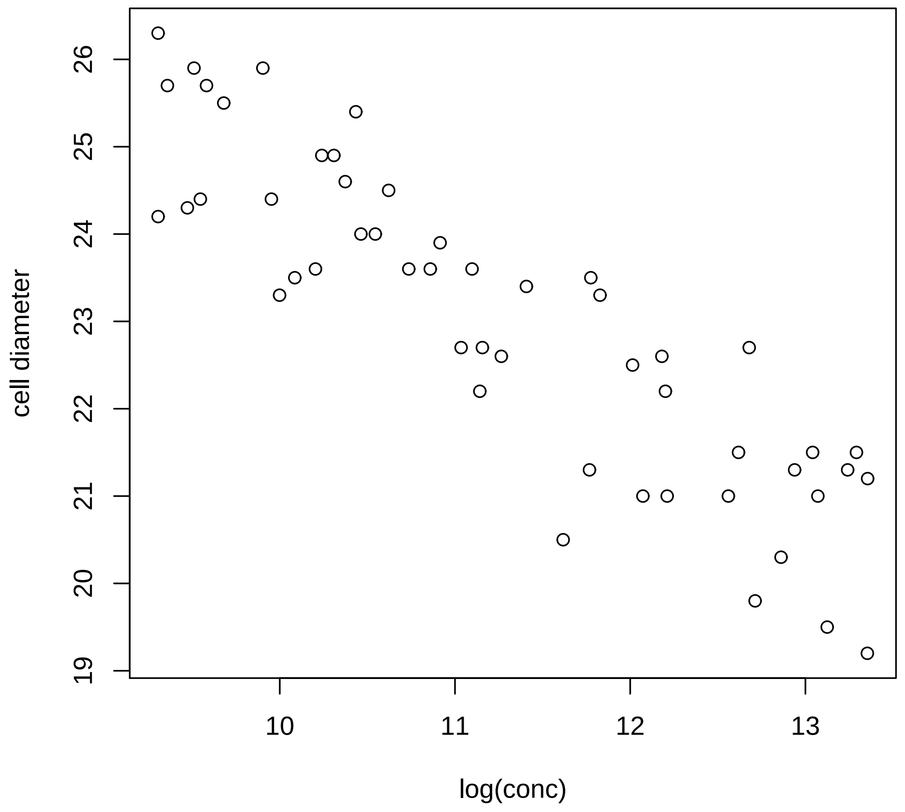
How many observations are there, and what are the variable names? `dim(hd), names(hd)`

Show me the first few rows, and basic properties of the variables! `head(hd),str(hd), summary(hd)`

Show me the data!

```
#basic plotting
x=log(hd$conc); y=hd$diameter;
plot(x,y,xlab='log(conc)',ylab='cell diameter')
```

There are R-packages (of which more anon) for reading specialized formats such as xls and image formats.



The function and variable search path

When you type a function or variable name, R searches for the object in a list called the search path.

`search()` shows you the databases in which R searches. The first, ".GlobalEnv", is your R console workspace. The last, "package:base" has all the basic R functions.

If you **`attach()`** a data frame to the search path, the variables inside the data frame can be found.

For example, you must type **hd$glucose** to get the **glucose** numbers. Following **attach(hd)**, you can access it with just **glucose**. Now **search()** shows **hd** in the path.

```
plot(log(conc),diameter) #an error
attach(hd)
plot(log(conc),diameter)
```

You can **detach('hd')** an object to remove it from the path. You might do this if you wanted avoid a conflict with something else with the same name. In many commands you can specify where to look for the variables, using the **data=** option. This is often easier read, and shows explicitly which data are being used.

Data summaries Here are some useful functions.

```
mean(), median()
sd(), var(), cov(),cor()
range(), quantile(), summary()
min(), max(), pmin(), pmax()
which.max(), which.min()
sum(),cumsum(),cumprod()
```

Many of these functions have an argument na.rm which needs to be set to TRUE in order to remove NAs from the data.

There are no NA's in these data so here is a simple example: If `eg=c(1,2,NA,3,4)` then `mean(eg)` gives an error but `mean(eg,na.rm=T)` gives 2.5, the mean of $(1, 2, 3, 4)$. R is forcing you to be explicit about the NA-handling.

Applying functions over data, and subsetting

The apply() command is also very useful for applying functions to all rows or all columns of an array or data frame. For example, `apply(hd,2,max)` finds the maximum of each variable.

Sometimes we want to work on subsets of a data frame. Boolean variables are converted to array indices when they appear as the index to a vector.

For example, to compare the mean cell diameters for cells with and without glucose we might calculate `mean(diameter[glucos` and `mean(diameter[glucose==2])`

We can create a new data frame from the subset of observations of interest: for example, `hd.g1=hd[glucose==1,]` pulls out the rows of `hd` that satisfy our condition, and takes all columns.

Plotting Data: Histograms

Graphical data displays provide insight to data. Think carefully about how to display data to highlight features of interest. We begin with simple exploratory plots.

`hist(diameter,freq=FALSE)` gives a histogram of the numbers in the variable `diameter`. The y-axis shows counts.

`hist(diameter,freq=TRUE)` gives a histogram with area scaled to one, like a probability density.

`par(mfrow = c(1,2)); hist(diameter); hist(conc)` puts two entire plots in one window, in 1 row and 2 columns.

You may need to adjust the number of histogram bins. The default 'Sturges' rule gives too few on large data sets. Try the 'Scott' rule, fix the number yourself, or give a vector of bin locations.

```
n = 100000; x = rnorm(n)
par(mfrow = c(2,2));
hist(x, main = "Sturges")
hist(x, breaks = "Scott", main = "Scott")
hist(x, breaks = sqrt(n),main='sqrt(n)')
hist(x, breaks = seq(from=-5,to=5,length.out=200),
     main='my breaks')
```

Overlays

The `lines()`, `points()` and `text()` commands can be used to overlay lines, points and text on an existing plot (already created with `plot()` or `hist()`).

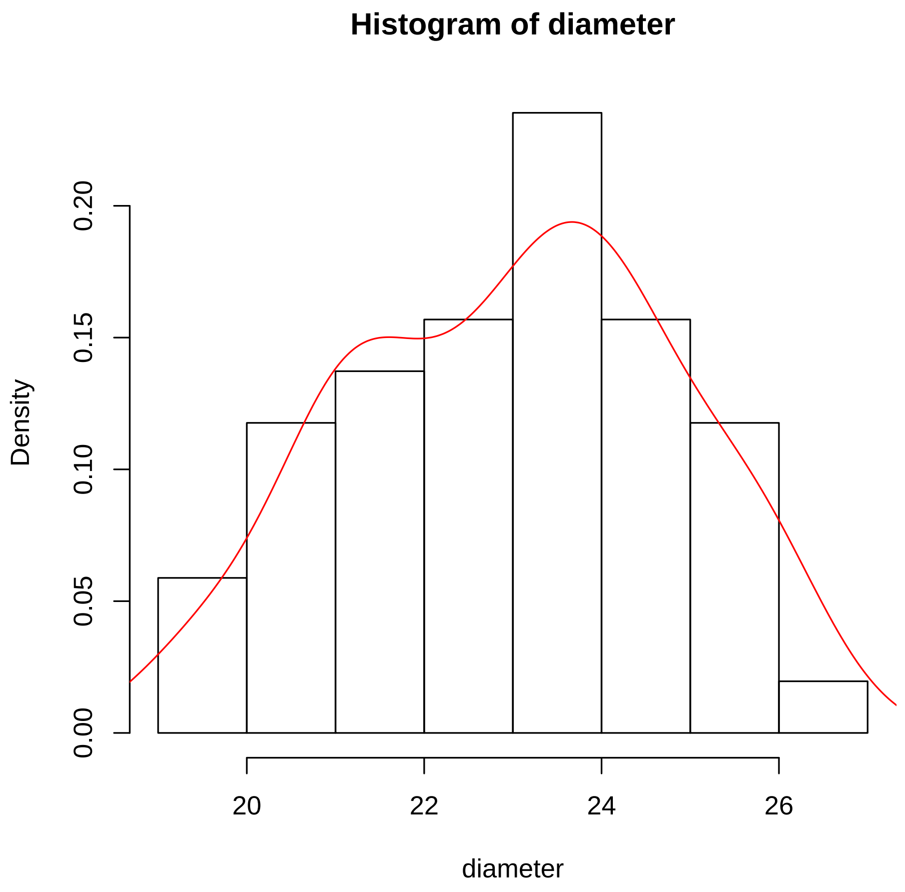
For example, we can overlay a density estimate (computed using the `density()` function) on a histogram.

```
hist(diameter,freq=FALSE);
lines(density(diameter),col=2)
```

Here `lines` will recognize `density()` output and done something sensible.

```
hist(diameter,freq=FALSE);
d=density(diameter)
lines(d$x,d$y,col=2)
```

has the same effect.



Boxplots and the R formula notation

These are useful for plotting a continuous variable against one or more discrete variables.

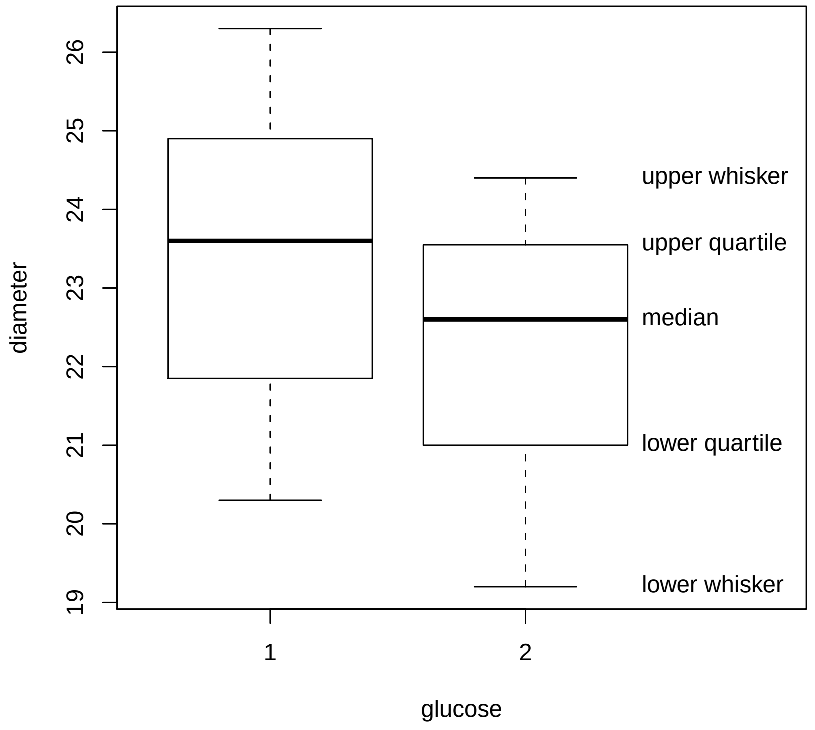
The R formula notation is useful here. The formula

```
variable1~variable2
```

means variable1 is a response variable and depends on variable2.

For example, we expect the distribution of cell diameters to be different depending on whether or not glucose was used.

```
boxplot(diameter~glucose,xlab='glucose',ylab='diameter')
```



Points beyond the outer whiskers are possible outliers.

Scatter plots

We started with an elementary plot. By varying color (`col`) and point character `pch` from point to point, you can highlight different subsets of data within the plot. For example

```
plot(log(conc), diameter, col=glucose, pch=glucose)
```

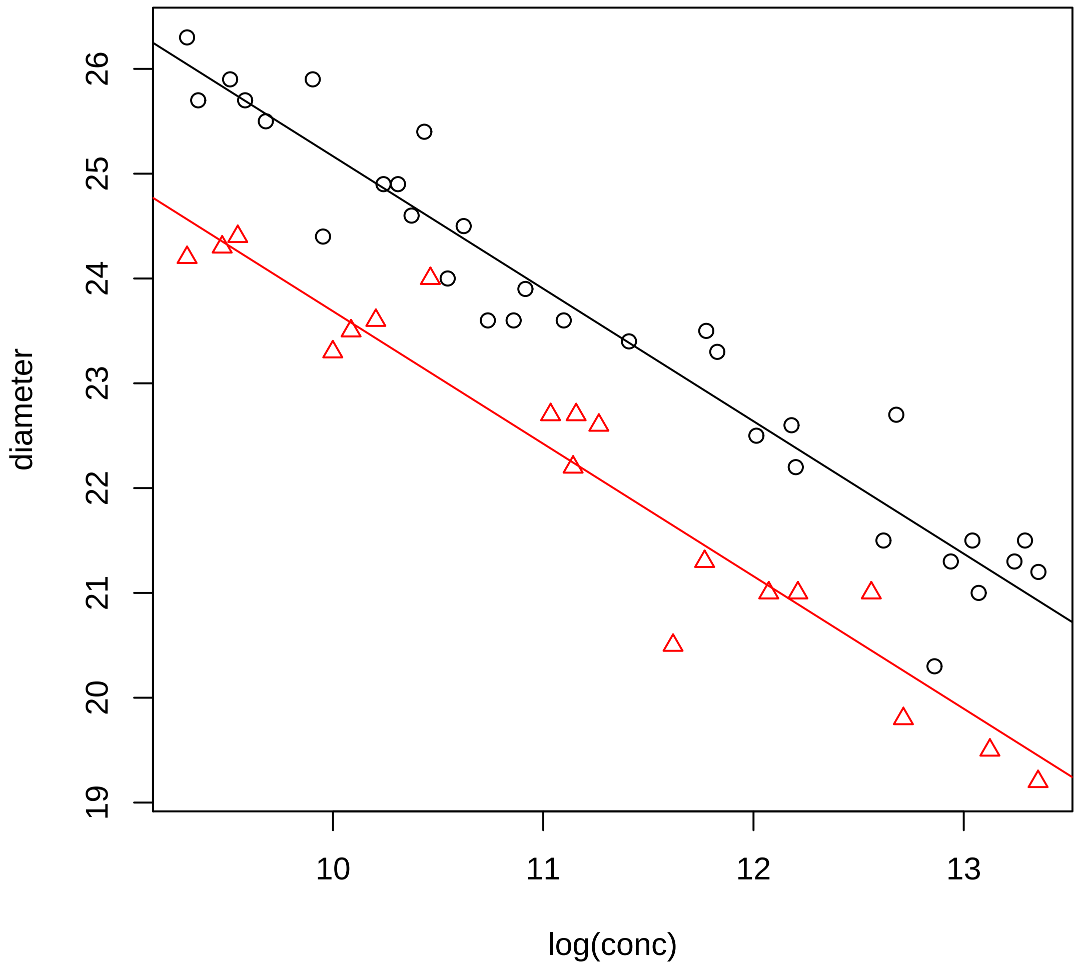
The color and shape of each point is decided by its glucose value.

We can overlay a line using the `abline(intercept,slope)` command, which simply takes the intercept and slope.

```
abline(37.806,1.264)
abline(36.327,1.264,col=2)
```

We can plot or overlay a function using `curve()`

```
curve(37.806-1.264*x, from=8 , to=15, add=T)
curve(36.327-1.264*x, from=8 , to=15, add=T, col=2)
```



Debugging

Your have just implemented an algorithm in R. It ran without reporting errors on the first input you tried. What do you do? Answer: assume it contains errors. Test it thoroughly using input for which you know the correct output.

You are implementing an algorithm in R. It crashes*. What do you do?

When an error occurs R saves the list of active functions. traceback() prints that list. See example in `L5.R`.

*terminates with an error message

You cant see the values of variables local to a function you called. You could use `print()` or `cat(sprintf())` commands to see what values variables take inside the function.

`debug()` lets you see what's going on inside a function interactively. You can examine (or change!) the value of variables, or execute any other R command, inside the function. You can also execute a debugger command

- n - next : execute the next line of code
- c - continue : let the function continue running
- Q - quit the debugger

undebug() can be used to turn off debugging on the function. The **browser()** command can also be put inside functions to start the debugger.

Try **debug(my_rnorm)** and then **my_rnorm()**.

This (ie **debug()**) is like desk-checking a programme. We pretend we are the computer and carry out the programme instructions by hand on a piece of paper. I usually use this as a last resort to fix a programme.

Saving a session

Data come to us in many different formats. R has functions to read them.

It sometime useful to save everything in the workspace (so we can quit, and continue where we left off later).

```
save.image(file = ”MyPracSession.Rdata”)
```

does that. To bring the session back up,

```
load(file = ”MyPracSession.Rdata”)
```

R saves in a compressed machine readable format, not easily read and edited.