MAS115 R programming
Lab Class 6
Graphics[1]

1 Graphics in R

1.1 Introduction to Plotting

One of the great strengths of **R** is the huge range of plotting commands that it provides. These allow the user to create extremely informative and customisable graphs which can then be saved for inclusion in reports. There are two sets of graphical functions in **R**: *base* graphics and *lattice* graphics. In this course we will only give an introduction to the base graphics suite.[2]

The usual method of constructing plots in **R** is to initially use the `plot()` command to create the basic image. This can then be built upon using lots of other commands to add the elements you want such as labels, legends, new overlaid graphs,... In this worksheet, we will introduce plotting following this scheme.

Getting help:

Like all the **R** functions, every plotting command has a help file that can be accessed via e.g. `?plot`. However, because of the huge flexibility in plotting, some of these help files are not easy to read or understand. If you are having difficulty understanding the help, it is normally best to actually run the extensive examples from the `Examples` section contained in the help file. Hopefully these will provide sample code which does something similar to what you need that can then be modified for your specific wishes.

1.2 The `plot()` command

The simplest command to create your graphs is the `plot()` command. This is a so-called 'generic' function which can recognise the type of object that you are passing it and behave accordingly, in this case creating the plot differently depending on the type of argument supplied. For example, it creates a scatter plot of a numerical variable `y` against either another variable `x` or an index; or a function of a single variable between two limits. Try the following code and look at the output. Notice that you can enter the data in lots of forms (it doesn't just have to be two separate vectors).

```
x <- 1:10
y <- 3*x^2 - 2*x
plot(y) # Plots against 1:length(y)
plot(x,y)
plot(list(x=x, y=y)) # Remember what a $ sign means for a list when looking at the axes
plot(cbind(x, y))
plot(cos, -2*pi, 2*pi)
```

[1] Thanks to R. Ripley for her help in designing this lab sheet

[2] For information on the lattice functions, see `http://portal.stats.ox.ac.uk/userdata/ruth/APTS2012/APTS.html`

```
xy <- data.frame(a=x, b=y)
plot(b ~ a, data=xy)
```

Changing the type of plot The default option for a plot is a scatter plot but you can change this using the `type` argument with the `plot()` command. This determines how the points are joined up. The options provided are:

- `'p'` — the default which just plots the points and doesn't join them up
- `'l'` — plots a line (note that the order of the data matters) - **this is the letter 'l' for lines, not the digit 1**
- `'b'` — plots both points and lines but the lines miss the points
- `'o'` — plots points and overlaid lines joining them
- `'h'` — plots histogram type vertical lines joining the points to the horizontal axis
- `'s'` — plots a step function joining the points
- `'n'` — plots no points, just the axes for later use (more useful than you might think)

Try these out on our trivial example:

```
# Types of plots
plot(x, y, type='p')
plot(x, y, type='l')
plot(x, y, type='b')
plot(x, y, type='o')
plot(x, y, type='h')
plot(x, y, type='s')
```

Changing the plotting character or line type You can also change the type of point that is plotted (the default is the open circle) by using the `pch` argument. I often prefer to represent points using a solid bullet or a plus sign:

```
# Changing the plotting character
plot(x, y, pch = 19)
# or
plot(x, y, pch = "+")
```

but there are lots of other options (have a look at `?points` if interested). You can also alter the type of line used (the default is a solid line) with the `lty` argument (look at `?lines` for info). For example:

```
# Changing the line type
plot(x, y, type ='l', lty = 2)
```

Changing the axes You will often want to add useful labels to the axes, or change the limits of the plotting window i.e. the range of both `x` and `y`. This can also be done by specifying arguments in the `plot()` command. For example try:

```
> plot(x, y, type = 'l', xlab = "My x-axis", ylab = "A function", xlim = c(0,8),
+ ylim = c(-30,120), main = "A useless title")
```

Adding new lines and points to plots Every time that you call `plot()` it will create a new plot. If we want to add a new curve to our current plot we have to use a different command. There are lots of functions which allow you to add new lines/points to your plot. These include:

- `points()` — allows you to add points to a plot. The points are specified in the same way as `plot()`. For more information look at the help file `?points` and `example(points)`.
- `lines()` — allows you to add lines to the plot. Again the points to be joined are specified in the same way as for `plot()`.
- `text(x, y, labels, ...)` — allows you to add text (labels) to your plot at the specified (x, y) coordinates. You can add multiple pieces of text at the same time if the arguments are vectors. See the help file for more information.
- `abline()` — another way of adding a straight line to a plot. Normally, you specify the intercept and slope but you can also add horizontal/vertical lines or pass other objects it can recognise (such as a linear model as we will see later).

To see an example of these in action try adding the following line to your code:

```
z <- -2*x^2 + 4*x + 80
lines(x, z, col = "green", lty = 4)
```

Adding a legend to a plot We might want to add a key to our plot so that viewers will know what the different lines (or points) correspond to. **R** allows us to do this using the `legend()` function. The syntax for this function is as follows:

```
legend(x, y, legend,...)
```

Here `x,y` specify the coordinates of the upper left corner of the box that will contain the legend. Alternatively you can specify the position with a keyword e.g. `"topright"`. The argument `legend` is a character vector of labels for the items you want to identify. You then have the option of specifying the features which identify the items e.g. `lty=`, `col=`, `pch=`. These should be the same length as `legend`. For example, suppose we want to add a legend to distinguish between the two lines currently on our plot. Try running:

```
legend(0, 115, legend = c("Function 1", "Function 2"),
       col = c("black", "green"), lty = c(1,4))
```

Creating windows with a layout for multiple plots Sometimes we will want to lay out multiple plots (i.e. totally separate graphs with separate axes) in one individual plotting window. To do this we need to set the *graphics parameters* within the `par()` function. For example `par(mfrow=c(2, 4))` will give a 2 by 4 array of figures which will be filled by rows each time you call a new `plot()`. Think of `mfrow` as short for "multiple figures, by row"; note that `mfcol` also exists. For example try:

```
par(mfrow = c(2,1))
plot(x, y, type = "l")
plot(x, z, type = "l", col = "green")
```

This mechanism is fine for simple cases. For much more flexibility, see `split.screen` or `layout`. **Note: R will store your multiple plot layout set via par so, if you want to change back to e.g. a single plot, then you will need to reset the graphics parameters with e.g.** `par(mfrow = c(1,1))`.

You can also set a huge amount of other parameters in `par` which has a very extensive help file. Many of these parameters can also be added directly to the `plot()` command itself e.g. `cex` which specifies the size of the plotting characters or `col` which specifies the colour in which the item is to be drawn.

Typesetting mathematics in labels You can also use a semi-LaTeX type mark-up when writing any text label on your plot (e.g. axis label, legend, text, label). For information look at `?plotmath` or try `demo(plotmath)`. The help file for `text` also has some simple examples. Try the following:

```
par(mfrow = c(1,1))
plot(x, y, type = "l", xlab = expression(theta),
     ylab =  expression(3*theta^2 - 2*theta))
```

2 Saving your plot to include in a report

After having created your beautiful plot you will generally want to save it so that you can include it in a report (e.g. Word or LaTeX). So far you may have done this by saving the plot from the graphics window — **this is not generally a good idea**. Saving the graphics window itself does not allow you control over the text size and when you rescale it into your external document the labelling may end up incredibly small and illegible. Also typically you will not want to have completely square plots (the default plotting window) but instead decide upon a layout which fits best the data you want to present.

The much better saving option is to use the `pdf()` built-in device **R** provides. The way this works is to open the special `pdf()` device, run the commands to create the plot, and then close the device using `dev.off()`. For example we can save the above plot in our working directory with

```
pdf("Myfirstplot.pdf", width = 6, height = 4)
plot(x, y, type = "l", xlab = expression(theta),
     ylab =  expression(3*theta^2 - 2*theta))
dev.off()
```

In order to select the best size of graph you can actually measure the size of the space in your document if it were printed on A4 that you wish to fill with your plot. It's normally best to then choose a slightly larger plot in **R** - I recommend 1.5 times as large as the size you want in LaTeX. You can then rescale it in the LaTeX documents. The example above would create a good plot of width about 4 inches in your LaTeX.

Note that when you run the command (if you run all the lines together) then nothing is actually produced on screen but the pdf file will be created in your working directory (which is identified at the top left of the RStudio console window).

2.1 Building up a plot to save it

To create a plot you will want to build up and store all the commands in your script window so that they can be altered easily if need be. You can then just rerun all these command together and make the plot.

It is best to first of all create the plot you want without using the `pdf()` device and then simply add the `pdf()` and `dev.off()` wrapper only once you are happy with the output. Note that in the above example we created a plot that was not square and this probably looked better than the default version. In order to test out and view the graph with this plotting size before saving it as a pdf, you can adjust the plotting window in RStudio, or open an `x11()` window and specify the width and height.

3 An example to work with

We have now covered some of the very basic ideas of plotting with a really boring example. For the rest of the practical we are going to try a more realistic (and interesting) example using the `Animals` dataframe built into the `MASS` package. This is a dataset recording the brain and body Weights for 28 Species. Start by loading the package and looking at the data.

```
> library(MASS)
> ?Animals
> Animals
```

We are going to be interested in finding out if there a relationship of the form

$$y_i = \alpha x_i^{\beta} + \epsilon_i$$

where y_i is the brain weight and x_i is the body weight of the animals considered. We will start by plotting the data against each other.

```
> plot(brain ~ body, data = Animals)
```

Unfortunately this plot is pretty uninformative as most of the data are squashed into one corner. We'll try a transformation instead.

Tasks:

- Create a plot of log(*body*) against log(*brain*) weights using solid bullets as a plotting character. Make the axis labels "Log of brainweight in g" and "Log of bodyweight in kg". Change the range of the y-axis to run from 0 to 12. Include an informative title for the plot.
- If the model above holds what would you expect the scatterplot of this data to look like?

Note that if we only wanted to display the data on a logarithmic scale, we could simply do that as part of the plot function, using the optional argument `log`. However, as we are going to do a little modelling too, it makes sense to actually obtain the transformed data.

4 Adding things to your base plot

To test our model we are going to fit what is known as a linear regression to the transformed data. Basically this tries to find the best straight line fit to the data i.e. the straight line which goes closest to the data.

4.1 Plotting multiple curves on the same graph

We can fit a linear model to the data using the `lm()` function in **R**. We can then draw the line itself by using the `abline` command as follows:

```
> abline(lm(log(brain) ~ log(body), data = Animals))
```

4.2 Identifying points

If you look at the plot we have created there are three points on the right hand side that lie well below the curve. These look like they could be outliers which do not fit with the rest of the data and we might be concerned that they are throwing off our best line fit significantly. In order to try and identify these points then we can use the `identify()` command which allows us to click on the plot to identify points (and label them) on a plot. The syntax for this function is

```
identify(x, y, labels)
```

where `x, y` are the locations of the points on the plot, from which you want to select and identify particular points. You actually need to go to the plot window and use your mouse to click on points. The label is placed at the point of the plot nearest the mouse click. To finish selecting points you can either

- Click the middle mouse button
- Right click and select stop
- Select stop from the graphics menu

Returned in the workspace will be the indices of identified points in the vector `x`.
For total control over where labels are placed in the plotting window it is better to use `identify` *without the* `labels` *argument. The labels can then be added in using* `text` *with the co-ordinates of the point returned.*

Tasks

- Identify the three unusual points using the command

```
> identify(log(Animals$body), log(Animals$brain), labels = row.names(Animals))
```

- Do you notice anything in common amongst these data points as compared to the rest of the data set? Should these data points be included in our analysis or treated as outliers and excluded from the model fitting?
- Create a new data frame called AnimalsNew which has removed the three identified points
- Fit a new model (using `lm`) to AnimalsNew and add this line to your plot. Your new line should be **red** and **dashed** — have a look at `?par` to figure out how to change the colour and what line type you need.

4.3 Adding a legend

We now want to add a key to our plot so that viewers will know what the different lines (points) correspond to.

Tasks

- Create a suitable legend that identifies the two regression lines (using their different colours and line types). Make sure that the legend labels are informative.

4.4 Saving your plot

You should have saved all the commands used to produce the above in a script file so that they can all be run again easily without any typing. Currently it still requires user interaction however, since you added the `identify()` points by hand. This means that at the moment we cannot create a pdf output without input from the user.

Tasks

- Replace the labels that were added by `identify()` with labels added via the `text` command - you will want to use the fact that you know which rows correspond to the points identified along with their rownames. You will also need to select the correct positioning of the labels so as to make the labels readable (look at the help file for `text` to figure this out).
- Rerun your modified code to make sure that the plot is produced without any interaction from the user required.
- Add a `pdf()` wrapper around your code and save the graph as `Bodybrainweightreression.pdf` with a width of 8 inches and height of 6 inches.
- Check that you can see your plot from *outside* R.