R Basis—Part 2

R. Montgomerie, Queen's University

This is the second part of my tutorial/workshop for learning the basics of R. I assume that you have mastered everything in part 1. I focus here on some extensions of that material with respect to data handling, writing functions, exploring output and plotting graphs.

2.1 Handling Data

Against my own principle of using only one technique to perform each task in R, I am going to describe three different methods to refer to your data, as these will be useful to know about. In 1.8, I introduced and used the **with()** function to refer to data frames. I like this method because it's always clear exactly which data frame you are referring to in an analysis, but it can sometimes be cumbersome to specify this every time, as you need to wrap each analysis in the **with()** function and keeping track of the extra brackets can be a pain. Here are some examples using **with()** to refer to built-in datasets (except for the first example, which uses an imported dataset):

```
> with(dat, aov(bill.length~body.mass*sex)) #full anova including interaction term, imported dataset stored in object 'dat'

> with(iris, glm(Sepal.Length~Species)) *general linear model using the 'iris' dataset

> with(sleep, t.test(extra~group)) *t-test to compare the amount of extra sleep between 2 groups in 'sleep' dataset

> with(CO2, boxplot(uptake~Treatment)) #boxplots of CO2 uptake data for chilled and unchilled treatments in the 'CO2' dataset
```

For each of these, we could have instead just specified **data =** inside the function's brackets, like this:

```
> aov(bill.length~body.mass*sex, data = dat)

> glm(Sepal.Length~Species, data=iris)

> t.test(extra~group, data=sleep)

> boxplot(uptake~Treatment,data=CO2)
```

and in each case this is more efficient and uses fewer brackets. This method is preferred

whenever it is available—see the R documentation for each function—so I suggest you use **with ()** only when you cannot specify the data frame within a function. Check the R Help for information about a function; if the syntax for the function has the word data in it, then you can use the **data=** command as above.

Overall, I suggest you use **data=** when you can, and **with()** the rest of the time.

Finally, almost all textbooks suggest you use **attach(xxx)** to loads the specified data frame (xxx) from the workspace into memory in such a way that R considers this to be the default data frame. R then assumes that all variables subsequently referred to are included in this data frame:

```
> attach(iris)

> glm(Sepal.Length~Species) #these variables are in the data.frame iris
> boxplot(Petal.Length~Species) #these variables are in the data.frame iris
```

Unfortunately, if you want to refer to a different data frame in the same session, you need to **detach()** the current frame, as in **detach(iris)**, then **attach()** or otherwise refer to the new data frame. Failing to **detach()** an attached data frame, can lead to all kinds of trouble and that is why many authors recommend against using it. But if you fail to **detach()** at the end of a session and save your workspace, that same data frame will be attached when you reopen R and can lead to lots of confusion. I would just avoid it and use either **with()** or **data =**

2.2 Functions

Each R package has many built-in functions that are followed by **()** within which you put a variety of commands and information. These functions have a suite of underlying instructions that make them work, written in the R programming language. While there are literally thousands of functions available in R packages, you may want to write your own functions to do tasks specific to your own analyses. To do that you simply specify the underlying code and store it in an new object, as shown below

To see the instructions underlying one of the built-in functions in an R package, simply enter the function name without the brackets:

```
> sd #this is the function to calculate the standard deviation of a vector (variable)
function (x, na.rm = FALSE)
{
    if (is.matrix(x))
        apply(x, 2, sd, na.rm = na.rm)
    else if (is.vector(x))
        sqrt(var(x, na.rm = na.rm))
    else if (is.data.frame(x))
        sapply(x, sd, na.rm = na.rm)
    else sqrt(var(as.vector(x), na.rm = na.rm))
}
<environment: namespace:stats>
```

Briefly, we will look at how a simple function is written and operates. I notice, for example, that there seems to be no function for calculating the coefficient of variation (CV) in R. I need this a lot so I wrote my own function, CV, which is a name I gave it—you can use any valid object name:

```
> CV = function(x) {sd(x)/mean(x)}
```

where **CV** is the new function, and it is defined as a function by **function(x)** followed by the expression to be evaluated, enclosed in curly brackets. Use the R editor to create such functions, then you can copy and paste them into the R console whenever you need them. Use the function by specifying that it acts on an object as defined in the function:

```
> CV = function(x) {sd(x)/mean(x)}
> CV(iris$Sepal.Length)
[1] 0.1417113
```

To write more complex functions, it is best to spread them out on several lines, to make for easier reading, as follows, with the function name on the first line, ending with {, and } by itself on the last line (this example from the Quick-R website):

```
# function example - get measures of central tendency and spread for a numeric vector
   x. The user has a choice of measures and whether the results are printed.
mysummary <- function(x,npar=TRUE,print=TRUE) {
  if (!npar) {
    center <- mean(x); spread <- sd(x)
  } else {
    center <- median(x); spread <- mad(x)
  }
  if (print & !npar) {
    cat("Mean=", center, "\n", "SD=", spread, "\n")
  } else if (print & npar) {
    cat("Median=", center, "\n", "MAD=", spread, "\n")
  }
  result <- list(center=center,spread=spread)
  return(result)
}
```

Once you have created a function it remains in the Workspace until you clear that workspace, so you can use it time and again without having to respecify it.

2.3 Internal Structure of R Output

When you run an R command (function), like **glm()**, the output is all stored in an object with an internal structure that you can assess, and this is extremely useful when doing randomization tests. Here is an example using **str()** to see what that structure is:

```
> modfull=glm(iris)
> str(modfull)
List of 30
 $ coefficients     : Named num [1:6] 2.171 0.496 0.829 -0.315 -0.724 ...
  ..- attr(*, "names")= chr [1:6] "(Intercept)" "Sepal.Width" "Petal.Length"
    "Petal.Width" ...
 $ residuals        : Named num [1:150] 0.0952 0.1432 -0.0731 -0.2894 -0.0544 ...
  ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
 $ fitted.values    : Named num [1:150] 5 4.76 4.77 4.89 5.05 ...
  ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...

[and another 45 lines of output]
```

To see, and use, any of this output just precede the desired output with **modfull$**:

```
> modfull$residuals
          1           2           3           4           5           6           7           8           9
 0.095211981  0.143156450 -0.073096946 -0.289356835 -0.054376913
 0.011114267 -0.323683608 -0.038123516 -0.307254656 -0.020872352
[and so on]
```

This means you can also analyze this material, as in testing the residuals for normality:

```
> shapiro.test(modfull$residuals)
        Shapiro-Wilk normality test
data:  modfull$residuals
W = 0.9954, p-value = 0.9202
```

Note that you can also enter **str(summary(modfull))** to see the underlying structure of the summary of modfull. In the above example, the summary contains the AIC value, for example, and you can access this using **summary(modfull)$aic**.

2.5 Randomization Tests (Resampling Statistics)

The basic idea is to take a random sample from a set of data many times, and use those random samples to test a statistical hypothesis. This is also an excellent way to calculate confidence intervals based upon the actual distribution of your data, rather than on the normal distribution:

```
>
# first create a function to obtain R-Squared from the data
> rsq <- function(formula, data, indices) {
    d <- data[indices,] # allows boot to select sample
    fit <- lm(formula, data=d)
    return(summary(fit)$r.square)
    }
# bootstrapping with 1000 replications
results <- boot(data=mtcars, statistic=rsq, R=1000, formula=mpg~wt+disp)
results # view results
plot(results)
boot.ci(results, type="bca") # get 95% bias-corrected confidence interval
```

Here is a more complicated example that selects one of the 1-3 samples taken from specific individuals, then runs a mixed model on the new subsample and calculates R^2 and saves the R^2 into a vector. This procedure is repeated 1000 time then the average R^2 is calculated:

```
> #resampling the mixed model
> count=0 # the variable count is set to zero
> N=1000 # the variable N is set to 1000
> rsq=0 # the variable rsq is set to zero
> for (i in 1:N){ #set up a loop so that 'i' runs from 1 to N;
        rowno=c(1:19,sample(20:21,1),22,sample(23:24,1),sample(25:27,1), sample
(28:30,1), sample(31:32,1),sample(33:34,1),35,sample(36:37,1),38,sample(39:40,1),
41,sample(42:44,1),45,sample(46:47,1),48,sample(49:50,1)) #this is a vector of row
numbers to choose from the dataset
        modfull = lm(Std4rtCops[rowno] ~ 1 +SpBGBG6030[rowno]     +SpBRBG60
[rowno]+BGQlcc45[rowno], data = dat1) # this is the mixed model run on the vector of
rows specified above
        modsum=summary(modfull) #this puts the summary from the model into the
object modsum
        rsq=rsq+modsum$r.squared #adds the R2 from that model into the object rsq
        }
> rsqmean=rsq/1000
```

2.6 Advanced Graph Plotting

The basic plot commands in the default 'graphics' package have a lot of flexibility, but there are even more manipulations you can do to these graphs by using the **par()** command before you begin plotting, and by adding additional graph commands to suit your needs. The whole process of producing publication quality graphs in a specific style can be quite complex, so I start with some basic graphs, the show you how to use the par() command, and finish with some additional graphing commands that allow you to tweak every feature of a graph.

For each of the major plot types (line, scatter, bar, histogram, box), I set up a text file using the editor with all of the options that I like to use for each graph, so that I can just copy and paste the code into R to produce graphs in my preferred format and style. To make that easy, I first put the variables I want to plot into x and y objects and I make my standard code just refer to x and y:

```
> y = iris$Sepal.Length

> x = iris$Petal.Length

>  plot(x,y)
```

Note that I use the default datasets in all plots so you can reproduce them for yourself.

Simple plots

I use these plots just for visualizing my data and exploratory data analysis, as they have minimal formatting. The codes for the different graphs are shown first, with some brief explanations, then on the following pages are the resulting graphs labelled A, B etc [Not done yet—we will plot these in our workshop]. Each of the green boxes below is a separate R session.

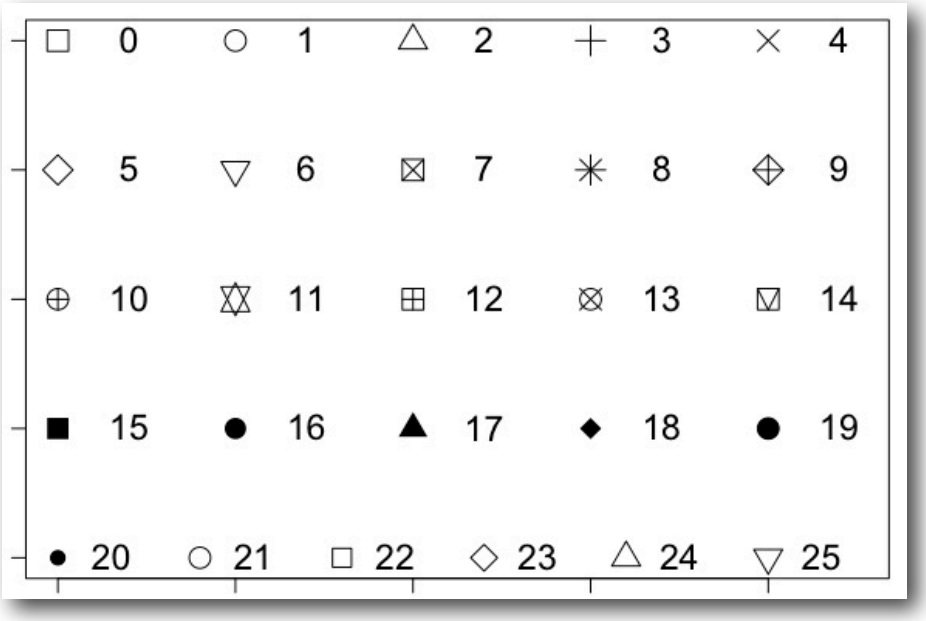
• using **plot()**

```
> y = iris$Sepal.Length
> x = iris$Petal.Length
> plot(x,y) #this is the simplest scatterplot of the variable y on the variable x, with default symbol (open circle) and axes labeled x and y
> plot(x,y, log= "y") #plots y on a log scale (or use x to plot x on a log scale, or xy for both axes)
> plot(x,y, xlab="Petal length (mm)",ylab="Sepal length (mm)",main="Iris data", sub="plotted 27 October 2009",pch=0,cex=2.0) #same, but with axes labeled, a main and a subtitle, the symbol changed to type 8 which is *, and 'cex=' to specify symbol size as a proportion of the default size

> plot(iris) #plots a scatterplot matrix of the whole data frame showing each variable on both x and y axes

> x=iris$Species
> plot(x,y) #when x is a categorical variable, this makes a Tukey boxplot for each level
> plot(y,x) #makes a dot plot for those same variables
> plot(AirPassengers) #when data frame is in this form, a line plot is made by default
```

The first 25 symbol types are:



- using **boxplot()**; note that these are Tukey boxplots where the whiskers extend to 1.5 x IQR and all data points are shown outside this range—you can change this if you wish (see documentation)

```
> x = iris$Petal.Length
> y = iris$Sepal.Length
> z = iris$Species
> boxplot(x) #boxplot of the variable x
> boxplot(x, y) #boxplots of the variables x and y side by side
> boxplot(x~z) #boxplots of the variable x at each z (categorical), with categories
  labeled as in the variable
> boxplot(x~z, notch=T,horizontal=T,varwidth=T) # notch=T shows 95%CL of the
  median, horizontal=T makes the plots horizontal, varwidth=T makes the width of
  the boxes equal to the square root of the sample size
> boxplot(x~z, boxwex=0.1) #boxwex makes the width of the boxes that proportion of
  the default width
```

- using **barplot()**

```
> barplot(euro) #plots one bar for each column in a matrix
> barplot(euro) #plots one bar for each column in a matrix
```

Note that this hardly begins to scratch the surface for barplots. In my experience each new plot needs additional tweaking.

• using **hist()**

```
> x=iris$Petal.Width
> hist(x) #default histogram where width of bins is chosen by R
> hist(x,breaks=5) #'breaks= ` suggests the number of bins (might not be possible so
  closest possible is chosen)
> hist(x,breaks=5,col=1, labels=T) #`col=1' makes the bars black instead of open;
  `albels=T' puts the sample size above each bar
```

Setting Graph Parameters

R comes with default graphing parameters stored internally. Before you change these it's often wise to store them in an object so you can restore the defaults later. You can always restore the defaults simply by restarting R, but otherwise use:

```
> default.par=par(no.readonly=T)
```

then restore by:

```
> par(default.par)
```

It's often useful to set new graphing parameters so that these will apply to all subsequent graphs. Here are some useful ones:

```
> par(oma=c(5,5,5,5)) #sets the outer margin at 5 units; this is the margin outside
  the graph and labels, and they are listed in the order (bottom,left,top, right)
> par(cex=1.5) #sets the amount by which all symbols will be magnified relative to the
  default
> par(cex.lab=1.5) #sets the amount by which x and y labels will be magnified relative
  to the default
> par(family="sans") #specifies the font family for text, "sans" is Helvetica
> par(lwd=1.3333) #sets the width of lines in 0.75 pt units, so 1 pt here
> par(tcl=-1) #sets length of tic marks, negative for outside tics
> par(pch=19) #sets the symbol type
> par(las=1) #makes the numbers on axes horizontal
```

and they can be combined into a single **par()** command that can specify your standard settings for any graph:

```
> par(oma=c(5,5,5,5), cex=1.5, cex.lab=1.5, family="sans", las=1, lwd=1.5,
  tcl=-1,pch=19)
```

Note that the size and aspect ratio of the window do not have to be set, as you can simply change window size by dragging on the lower right corner and the graph will change accordingly. Most parameter values can be set with the plot() function, but not oma or omi, and I prefer to write them within the plot functions when I can, as shown below.

Graph Construction

For complete control over the style of your graph, it is sometimes useful to build a graph bit-by-bit, setting the main **par()** values first, then plotting the symbols, and finally adding axes, tic marks, numbers, and labels. I like to start by specifying the x and y variables and doing a simple default plot so that I can have a look at the data and the axes:

```
> x = iris$Petal.Length
> y = iris$Sepal.Length
> plot(x,y) #default plot to examine the range of data etc
```

Based on this default plot, I decide where I want each axis to start and end, how big to make the symbols, and where I want the labels and tic marks. Then I proceed to build up a graph from scratch, in a new Plot pane:

```
> default.par=par(no.readonly=T) #store default params
> plot(x,y,xlim=c(0,7),ylim=c(4,8),las=1,xlab="Raoul", ylab="", pch=1,
cex=2,lwd=1.4,axes=F) #redo plot making axes the size I want them, no xy labels,
symbols at 1.5 size, and plain axes on all 4 sides
> box(lwd=1.333)#makes a box around the axes with line width = 1 pt where 1 unit =
0.75 pt
> axis(1, tcl=-0.75,at=seq(0,7,1),lwd=0,lwd.ticks=1.4,labels=T) #bottom axis with
major tics at intervals of 1 between 0 and 9, with labels
> axis(1, tcl=-0.5,at=seq(0.5,7.5,1),lwd=0,lwd.ticks=1.4,labels=F) #bottom axis with
minor tics at intervals of 1 between 0.5 and 8.5, no labels
> axis(3, tcl=-0.75,at=seq(0,7,1),lwd=0,lwd.ticks=1.4,labels=F) #top axis with major
tics at intervals of 1 between 0 and 9, with no labels
> axis(3, tcl=-0.5,at=seq(0.5,7.5,1),lwd=0,lwd.ticks=1.4,labels=F) #top axis with
minor tics at intervals of 1 between 0.5 and 8, no labels
> axis(2, tcl=-0.75,at=seq(4, 8,1),lwd=0,lwd.ticks=1.4,labels = T) #left axis with
major tics at intervals of 1 between 4 and 9, with labels
> axis(2, tcl=-0.5,at=seq(4.5, 7.5,1),labels = F) #left axis with minor tics at intervals
of 1 between 4.5 and 8.5, no labels
> axis(4, tcl=-0.75,at=seq(4,8,1),labels = F) #right axis with major tics at intervals of
1 between 4 and 9, with nolabels
> axis(4, tcl=-0.5,at=seq(4.5, 7.5,1),labels = F) #right axis with minor tics at intervals
of 1 between 4.5 and 8.5, no labels
> par(default.par) #restores the default pars
```

Once you have decided how you like each of your graph types to look, simply save all this code in the editor so that it is ready to re-use as needed, with changes to the numbers, as needed.

2.7 General and Generalized Linear Mixed Models

This is not the place to learn about GLMMs, if you are a novice. Have a look at the textbooks by Zuur and Faraway listed at the end of the first handout, for both an introduction and details of running GLMMs with R. Here are the most useful packages, in my opinion:

- **nlme**: this uses a traditional approach to general (but not generalized) with P-values etc
- **lme4**: does both general and generalized mixed models, but does not provide P-values etc; this is probably the most widely used package in the ecology literature
- **MCMCglmm**: most sophisticated approach, using Markov Chain Monte Carlo simulations for statistical testing etc; special emphasis on correlated random effects arising from pedigrees and phylogenies

And here is some basic syntax, with Y1 as the response variable, X1 and X2 as fixed effects, and X9 as a random effect, fit by maximizing the restricted log-likelihood (REML) in nlme and lme4:

```
> mod1 = lme(Y1 ~ 1 + X1 +  X2 , data = dat, random = ~1 | X9, method = "REML")
    #general mixed model using nlme

> mod2 = lmer(Y1 ~ 1 + X1 + X2 + (1 | X9), data = dat, REML = T, family = normal)
#generalized mixed model using lme4, and normal errors

> mod2a = lmer(Y1 ~ 1 + X1 + X2 + (1 | X9), data = dat, REML = T, family =
binomial) #generalized mixed model using lme4, and binomial errors with logit link
function

> mod3 = MCMCglmm(Y1 ~ 1 + X1 +  X2 , data = dat, random = ~ X9, family =
"zibinomial") #generalized mixed model using MCMCglmm, and zero-inflated binomial
errors with logit link function
```

2.8 MultiModel Inference

Use **MuMIn** for both model evaluation and model averaging, as follows:

```
> mod2 = lmer(Y1 ~ 1 + X1 + X2 + (1 | X9), data = dat, REML = T, family = normal)
    #generalized mixed model using lme4, and normal errors

> dd = dredge(mod2)

> print (dd, abbrev.names = F) #ensures that the full names of variables are shown on
the output


> aa=model.avg(get.models(dd, subset = delta < 2),method="NA") # get averaged
coefficients, for all models with AICc ≤ 2
```

2.9 More Resources

Books

Chambers JM. 2008. *Software for Data Analysis: Programming with R*. Springer, NY. ISBN 978-0-387-75935-7.

Haddock SHD, Dunn CW. 2011. Practical Computing for Biologists. Sinauer Associates (Sunderland, MA). An excellent introduction to programming and database management

Kabacoff R. 2010. *R in Action*. Manning. [http://www.manning.com/kabacoff]

Wickham H. 2009. *ggplot: Elegant Graphics for Data Analysis*. Use R. Springer. ISBN: 978-0-98140-6

Spector P. 2008. *Data Manipulation with R*. Springer, NY. ISBN 978-0-387-74730-9.

Downloadable pdfs

The R Guide [http://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf] a pretty good introductory manual

Using R [http://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf] lots of advanced stuff here, particularly on graphics

R-ref card http://cran.r-project.org/doc/contrib/Short-refcard.pdf] an excellent quick reference list

Websites

http://www.r-bloggers.com/ this is a blog that aggregates information from dozens of other blogs that have information about R. It's an excellent source of up-to-date information, tutorials, etc

http://psy-ed.wikidot.com/glmm this is a wiki maintained by the Psychology Group at Univ Edinburgh, with this page especially about mixed effects models

http://manuals.bioinformatics.ucr.edu/home/programming-in-r all about programming in R

http://www.zoology.ubc.ca/~schluter/bio548/index.html Dolph Schluter's stats course at UBC, using R, with lots of biological examples and useful information

http://www.oga-lab.net/RGM2/images.php?show=all&pageID=345 and R manual for graphics and other things

http://www.harding.edu/fmccown/r/ info and scripts for producing simple graphs in R