FISH / MSL 604

Franz Mueter

Lab 1: Introduction to statistical computing and graphics in R

1. **What is R?** A free version of the powerful, object-oriented S programming language for statistical and graphical analysis. The R philosophy is based on open source code, so everyone can contribute code for specific problems. There is a growing list of user-contributed software at the R-project website (http://www.r-project.org/). The strengths of R include flexibility, thanks to a powerful programming language, and excellent graphics for graphical data analysis.

2. Make sure you have a copy of R installed and running! (www.R-project.org)

3. **Starting\exiting R**: Setting up and saving workspaces, Quitting a session: `q()`
   Managing different projects:
   → Exercise: Setting up a generic workspace for class
   - Create a folder for all your R projects related to this class
   - Find your R icon (e.g. on Start menu, choose 'All Programs' and you should find R)
   - Right click the icon and click 'create a shortcut'
   - Drag the shortcut to the folder that you just set up and rename the shortcut 'MSL 604' or whatever you want to name it.
     (Alternatively, instead of the last three steps, you can right click inside the folder, go to 'New' and click on 'Shortcut', then browse to where your programs are located, most likely on the C drive in "Program files", then find and click on the file: "C:\Program Files\R\R-2.15.1\bin\Rgui.exe" (or whatever the appropriate path for R is) to insert a shortcut (it will prompt you to name the shortcut).
   - Right click the shortcut and click on 'Properties'. Change the path in the 'Start in' field to the directory (=folder) where you want your 'workspace' and related files to be saved. For example, you can use the current directory if you want (i.e. the same directory where the shortcut is located), or a subdirectory for your projects related to this class, or any other directory you choose. Your workspace is a file with extension .RData that will contain all of the R objects that you create during a session.
   - Start an R session by clicking the shortcut. You should get an open Command Window (R console) and you are now ready to start your session!
   - For labs, you should eventually set up a separate directory for each lab. You can then simply copy all lab files to the corresponding lab folder, including the script files that I provide for each lab (which will have extension *.R). If you configured R to recognize files with that extension, you can simply click on a script file to open it in R (or R Studio). The directory from which you start R will automatically be your working directory. I wrote all script files assuming that any required data files will be in your working directory.

4. **Command Window and scripts**
   You can enter commands either directly at the prompt in the Command Window (R console),

or you can open a script window (click on 'New Script' under the File menu).

Working directly from the Command Window is preferable if you are simply exploring, trying new things, or doing something once (all of you commands from the current session are saved in the '.Rhistory' file when you save your workspace).

Working with scripts allows you to set up repetitive analyses or document what you did in more detail by including comments. For example, I saved all of the commands for this introductory session in a script file called "R intro script.R". Go ahead and open the script file from the File menu and feel free to annotate it as you wish for your own purposes. Note that comments are preceded by the '#' symbol, which tells R that what follows is a comment and should be ignored!

To run a particular line of code, simply put the cursor on that line and hit <Ctrl R> or <Ctrl Enter> depending on your settings (Command-Enter on Mac). Note that the line is copied to the Command Window and executed. You can run blocks of code by selecting any section of the script window and hitting <Ctrl R> or <Ctrl Enter>). However, don't blindly run each line of code. Make sure you understand what is going on, experiment on your own, and use `help` (or '`?`') often!!

Let's start with some very basic examples. Type or copy the following commands at the command line prompt in the R console:

```
x <- 1:10                  # creates sequence 1, 2, 3, .., 10
x                          # view result
y <- rnorm(50)             # creates 50 random normal numbers
y
ls()                       # to see objects currently in Data directory
```

Note that most functions in R take "arguments" that are enclosed in parentheses. Even if you don't need to specify any of the arguments because all arguments have default values (such as those for function `ls()`), or if the function does not take any arguments, you still have to use the parentheses to run the function. Otherwise, R will simply print the function definition:

```
ls                         # to see function definition
```

Note: R is case sensitive!

```
Mean(x)                    # results in error message
```

Avoid cluttering, reuse names (overwrite) if original is no longer needed and remove objects after they are no longer needed (see below). NOTE: Be careful! If you delete or overwrite objects in R, you will NOT be able to recover them!

```
rm(x, y)                   # removes x and y
ls()                       # note objects created internally (e.g. .Last.value)
.Last.value                # useful for retrieving results from a long computation if
                             you forgot to assign it
```

```
q()                          # to exit! Alternative: Use File menu
```

(When you exit R it will prompt you whether you want to save your workspace. If you simply click 'Yes', all of the objects you created will be saved in a "nameless" file with extension '*.RData*'. If you want to save your work to a different file, you have to use 'Save Workspace' under the file menu before you quit! I found it easiest and safest to save all of my workspaces in the default nameless files, but in different directories for each project. (That's because the default file for saving your workspace when you exit R is an '.RData' file in the current working directory, even if you start by clicking on a different *.RData file. R will create a new .RData file or overwrite the existing one without warning!).

Hint: Save your workspace frequently. If R crashes, you will lose everything you did since the last save! Luckily, R rarely crashes!

5. **Getting help**: ?command, help menu

```
help(rnorm)                  # to get help file for function 'ls'
?rnorm                       # alternative
args(rnorm)                  # see what arguments it takes
```

Examine the structure of the help files. All help files have the same components. Use help frequently to learn about functions. Look at the examples and at related functions to expand your "vocabulary" of functions (there are 2000+ functions built into the base version of R and many thousands in additional packages)!

6. **S is a function language**. Everything is either an *expression* or an *assignment*

Mathematical and statistical expressions:

```
3 + 4                          # Expression, output directed to screen
sqrt(3/4) * (5 - pi^2)         # Expression, note different operators
x <- rnorm(20)                 # Assignment operator: output assigned to x
x = rnorm(20)                  # Equivalent (only implemented in recent versions
                                 of R, hence I use '<-' throughout)
mean(x)                        # Statistical expression
m <- mean(x)
v <- var(x)
m / sqrt(v)                    # Coefficient of variation
sqrt(x)
rm(m, v)
```

Note: `mean()` and `var()` operate on whole vectors and return one value, `sqrt()` operates element-by-element. Functions generally take zero to many **arguments**, and may be very simple or extremely complex. For example, look at the following definitions:

```
sd                           # simple function
glm                          # very complex function
```

Many R function are transparent and can be modified by the user. Some functions are internal to R and cannot be modified. Many functions call C code or Fortran code for improved speed

```
args(mean)                        # find out what arguments a function takes
```

There are **required** arguments: (e.g. 'x' in `mean`), and **optional** arguments (e.g. ‘trim’ and ‘na.rm’). Arguments can be specified by their name or by their position in order of appearance. Required arguments have no default and result in an error message if not supplied. Optional arguments are associated with some default action that is used if the argument is not explicitly specified.

```
x[4] <- NA                   # replace the 4th element of x by a missing value
x                            # NA for missing value
mean(x)                      # fails due to missing value
mean(x, na.rm =T)            # specify na.rm argument (na.rm=F by default)
mean(x, , T)                 # specify argument by position (without using name)
mean(na.rm=T, x=x, trim = 0.2)    # order is irrelevant if you name arguments
```

Other useful mathematical and statistical functions:

```
stdev, var, min, max, range, abs, sqrt, log, log10 or log(x, base = 10),
exp, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh
```

```
x <- 1:5
sum(x)
prod(x)
cumsum(x)
cumprod(x)
x <- sample(x)
cummax(x)
cummin(x)
pmin(1:20, 20:1)
pmax(1:20, 20:1)
?pmin                        # to learn more about pmin\pmax
```

other functions: `round, trunc, floor, ceiling`

```
round(2.3857)
round(2.3857, 2)
round(23857, -2)
floor(1.8)
ceiling(-1.8)
floor(-1.8)
trunc(-1.8);  trunc(1.8)     # Use semi-colon to separate expressions
```

Note: Names of objects (e.g. x, m, v) cannot start with a number and cannot have certain special characters (\, -, +, etc.). Also, avoid using names of existing functions.

```
mean <- mean(x)              # Can cause problems later
find("mean")                 # To find each occurrence of object mean
search()                     # show search list where objects are located, most contain
                               built-in (and essential) functions
rm(mean)
```

Note that all output from an expression is simply directed to the screen and printed and does not change or create any object, unless you specifically assign output to a named object using an assignment operator (either the `assign()` function, '`<-`', or '`=`').

You can also redirect output to a file:

```
sink("test.txt")                    # direct output to file
```

All of your output that follows will be saved to "*test.txt*" (as a plain text file), e.g.

```
"This is a test"
3 + 4
mean(x)
sink()                              # redirect output to screen
```

→ Exercise: Create a user-defined function to compute coefficient of variation. Type the following at the command prompt or insert it in the accompanying script:

```
CV <- function(x) {
# You can add comments as you wish, e.g. a short description of
# the function, the date it was written, etc.
         sd <- sqrt(var(x))
         m <- mean(x)
         m/sd
}
```

Note this new function has been created in your workspace:

```
ls()
```

Now test your function:

```
data <- rnorm(20)   # Generate some 'data'
data
CV(data)
rm(data)
```

7. **S is an object-oriented programming language**. Everything is stored as an object

Objects: functions, vectors, matrices, data frames, fitted model objects, lists (which can include any or all of the above), and many others

Different objects have different characteristics: modes, classes, and attributes
  modes include: "logical", "numeric", "complex", "character", "null",
        "list", "function", "graphics", "expression", "call"

```
x <- rnorm(20)
mode(x)                       # to get mode of x
x > 0
mode(x>0)
5 * (x > 0)                   # logical vector acts like numerical vector: 0,1
sum(x > 0)                    # useful for getting counts of elements meeting a
                              # specified condition
```

# concatenate elements:

```
y <- c("this", "is", "a", "character", "vector")
y
```

```
plot(0:6, 0:6, type="n")
text(1:5, 5:1, y)
mode(y)
c(1:5, rep(6, 5), seq(7, 20, by=2))      # Another example of concatenating
```

Functions to check for a specific mode: `is.numeric, is.character, is.null`

```
is.character(y)                          # returns TRUE
```

Some <u>classes</u>: lm, glm, tree, arima + lots more + user-defined classes

The "class" attribute of an object is used to decide which **method** of a generic function is to be used in a given instance. This is in part what makes object-oriented programming so powerful and flexible

```
x <- 1:20
y <- rnorm(20)
plot(x, y)                    # Plots y against x because x and y are
                                simple vectors. Uses plot.default method
```

Type of plot depends on **mode** of arguments:

```
x <- c(rep(1,10), rep(2,10))            # create vector with 2 values
plot(x, y)
x <- factor(x)                # convert to a factor (categorical variable)
plot(x,y)
x <- sample(1:20)             # 1:20 in random order
fit <- lm(y~x)                # linear model, regression of y on x
fit                           # prints a short form of the model object
```

Type of plot depends on **class** of argument

```
class(fit)                    # show class
plot(fit)                     # diagnostic plots: uses plot.lm() to generate a
                                 series of diagnostic plots appropriate to the
                                 type of model fit by lm()
methods(plot)                 # list all methods written for plot, which produce
                                 different types of plots for each class of objects
plot                          # print function definition
stats:::plot.lm               # print definition of plot.lm(). Note that this is a
#  "hidden" function, but you can get the definition by specifying the package
args(stats:::plot.lm)         # show arguments that plot.lm can take
plot(fit, 2)                  # plot only one of the graphs
summary(fit)                  # uses summary.lm() to extract appropriate info
resid(fit)                    # uses residuals.lm() function
coef(fit)
fitted(fit)
```

User defined classes are extremely useful in programming repetitive tasks attributes Attributes are length, dimensions of a matrix, names, and just about anything else! Class of an object is also stored as an attribute!

```
iris                                  # built-in data frame
attributes(iris)
names(iris)
```

8. **Data input / output**: using a) menu, b) `read.table()`, `read.csv()`, etc., c) `scan()`

`read.table` and its variants (`read.csv` for comma-delineated files, `read.delim` for tab-delineated files) generate a data.frame object and are designed for rectangular data.

In contrast '`scan`' has much greater flexibility

```
x <- scan()                # Takes input at prompt, one value at a time, and saves it to 'x'
                              useful for creating short vectors of data
```

→ Exercise: Open file GAK1.csv (for example using Excel) to inspect the data. It contains temperature and salinity data for Gulf of Alaska station GAK 1. You can use `read.csv()` to read the data from this comma-delineated file into R and save it as a 'data frame' object, for example '`GAK1.dat`' (The name is arbitrary):

```
GAK1.dat <- read.csv("GAK1.csv")
```

This works if the 'GAK1.csv' file is located in your current working directory (check your working directory by typing '`getwd()`' at the prompt. To change the working directory, use `setwd()` or the menu). Alternatively, you can specify the full path or you can open a dialogue box to look for the file:

```
GAK1.dat <- read.csv(file.choose())
```

Tabular data are typically stored in a "data frame", which allows each column (or variable) to have a different data type (e.g. numeric, character, or factor). All variables in GAK 1 are of mode 'numeric' (If desired, we can convert them to factors or character vectors – see below!).

Show variables in data frame:

```
names(GAK1.dat)
GAK1.dat$depth                      # List the depth variable
plot(GAK1.dat$depth, GAK1.dat$temperature)         # Plot temperature by depth
GAK1.dat$depth <- factor(GAK1.dat$depth)           # Change class to a factor
```

Note how the class affects the way the data are plotted:

```
plot(GAK1.dat$depth, GAK1.dat$temperature)
```

9. **Data structures**: vectors, matrices, arrays, data frames, lists
Subscripting / Indexing

Vectors: x, y above are examples of vectors. Vectors can have mode numeric, factor, character, logical, or complex. Vectors can have names (i.e. a name for each element)

```
x <- 1:20                             # easy way to generate sequence of integers
x <- seq(0,10, length=20)
```

```
names(x)
names(x) <- paste("Element", 1:20, sep="")
x
```

→ Exercise: Using function `rep()`, generate the following sequences (learn about `rep()` first by typing `?rep`):
1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7

→ Exercise: Using function `seq()`, generate:
0, 0.1, 0.2, 0.3, 0.4, …. 1.6

```
y <- runif(20, 1, 3)
y[4] <- NA                       # insert missing value
y                                # note missing value (NA)
sqrt(y)                          # results in missing value in output, true for most
                                 # functions that operate element-by-element
plot(y, type="l")                # ignores missing value, segment of line not drawn
plot(na.omit(y), type="l")
mean(y)                          # results in NA
mean(y, na.rm=T)                 # requires you to specifically remove missing

cbind(x,y)                       # combine two vectors of equal length into matrix
cbind(1:3, 1:20)                 # cbind will happily combine vectors of equal length
                                 # (Shortest vector is simply repeated!)

var(cbind(x,y))                  # again, by default fails because of missing value
var(y, use = "pair")             # another argument for how to handel missing values
?var

names(x) <- letters[1:20]
x[c("a", "h", "m")]
```

<u>Matrices</u>

```
x <- matrix(rnorm(20), nrow=5)
dim(x)
dimnames(x) <- list(NULL, letters[1:4])
x
x > 0                     # matrix of logical values, all elements 0 (is a logical operator)
sum(x>0)                  # total number 0, logicals are 0 (F) and 1(T) in computations
means <- apply(x, 2, mean)            # apply works on matrices (by row or column)
apply(x, 1, function(y) sum(y > 0))         # accepts any other function
sweep(x, 2, means)                    # subtract means

mean(x)                               # scalar result
cor(x)                                # matrix of pairwise corrrelations
var(x)                                # var-cov matrix
```

```
row(x)                              # useful in modeling, matrix computations
col(x)
lower.tri(x)
x[lower.tri(x)]                     # extract elements of lower triangle
diag(x)
diag(5)
diag(diag(x))

X <- runif(30)
dim(X) <- c(10,3)                   # Alternative to matrix for creating matrices
y <- 1:10
t(X) %*% y
crossprod(X, y)                           # more transparent and efficient
x <- 1:10
outer(x,y)                          # outer product
x %o% y                             # alternative to 'outer'
```

Here's a use of 'outer' to compute an arbitrary function over a grid of values.
Example: f(x,y) = cos(y) / (1+x2)

```
f <- function(x, y) cos(y) / (1+x^2)
a <- b <- seq(-4,4,length=50)
z <- outer(a, b, f)
image(a, b, z)
contour(a,b,z, levels=seq(-1, 1, by=.25), 0, add=T)
```

<u>Arrays</u>: Arrays can have any dimension, but you will rarely use more than 3 or 4

```
x <- array(rnorm(20*4*3), c(20,4,3))          # create 3-dimensional array
dimnames(x) <- list(1980:1999, c("Spring", "Summer", "Fall", "Winter"),
   paste("Station", 1:3, sep=""))
x
x[c("1980", "1985"), "Spring", 1:2]

apply(x, c(2,3), mean)                  # long-term means by season and station
apply(x, 3, mean)                       # long-term means by station
```

<u>Lists</u> Lists can contain a collection of very disparate components. Each component of the list may be another list, a single value, a vector or array, or even a function

```
x <- 1:20
y <- 10 + 3*x + rnorm(20, 0, 5)
plot(x,y)
fit <- lm(y~x)
abline(fit)
is.list(fit)
fit                    # uses print.lm() which does NOT print all contents of the list!
names(fit)             # prints names of all components of the list
fit[1]                 # can be subscripted like a vector, extracts a component of the list,
                              # the result is another list of length 1
fit[[1]]               # this is the proper way to only extract the content of one
                          # components of a list! The result is whatever was stored in
                          # that component of the list!
```

```
fit$coef                # alternative to extracting a component for named lists
fit[["coefficients"]]# another alternative using the name (this is useful when
                        # writing functions but requires full name)
coef(fit)               # special function to extract a particular component
```