Computational Methods in Astrophysics

Dr Rob Thacker (AT319E)
thacker@ap.smu.ca

More on R

- Useful things for scripting:
  - Data frames
  - I/O
  - Looping
  - Graphics
  - Models
- Note: R is fussy about having the right quotes, copy and paste often fails because of this

Data frames

- Commonly used in R for multi-variable data
  - Consider people data: heights, masses, hair colour etc
  - Non-numeric values are called factors e.g. “blue” “brown” for eye colour
- Logically it’s a matrix of columns/vectors of equal length but potentially different types
  - Once set-up you can address variables using $ sign
- Could use built-in data, but let’s see how to construct one first

Data frames

- Start the console, create a script and enter

```
height <- c(1.7,1.65,1.34,1.5,1.8)
name <- c(“Izzy”,”Chris”,”Mel”,”Viv”,”Alex”)
mass <- c(70,55,50,62,80)
eyes <- c(“brown”,”green”,”brown”,”blue”,”brown”)
hair <- c(“brown”,”blonde”,”blonde”,”blonde”,”brown”)
ourpop <-data.frame(name,eyes,hair,height,mass)
```

Interacting with data frames

- The full table can be printed using `ourpop`
- Try `ourpop$height`
- Can also use `ourpop[,1]` to get column
- `ourpop[1,]` to get first row
- Note if you define something in R using a variable with a value e.g. `mylist <- list(a=2,b=1)`
  - `[2]` will report the variable name & value
  - `[[2]]` will report just the value (try them!)

Subsetting & sampling data frames

- The `subset()` function allows you to quickly select data that matches criteria, e.g. try

```
mypop <- subset(ourpop,height>1.5)
mypop2 <- subset(ourpop,height>1.5 & height < 1.79)
```

  - `mypop,mypop2` will be a data frames as well
- You can randomly sample any data using `sample()`, e.g. try

```
mysamp <-
ourpop$height[sample(1:nrow(ourpop),10,replace=TRUE)]
```

  - In this case you'll produce a vector of samples

Simple Output

- `print()` – this is generic output function that is specified for different datatypes
  - Depending on what you pass, you'll get different results
- Try `print(ourpop); print("Hello World")`
  - Try removing quotes – it fails why?
- Essentially same as `ourpop` in console
- `methods(print)` will tell you what it is defined for (in this case a lot!)

Concatenated Output

- `cat()` is much simpler than `print()` – can't handle a data frame for example
  - But it does handle newline
  - Allows you to write to a file as well
  - Separate lines of output still need a loop
- Try `cat("Hi ",ourpop$height,"\n")`
- You can restrict the number of pieces too:

```
cat("Hi ",ourpop$height[1:3],"\n")
```

- Plus add a file argument and separator

```
cat("Hi ",ourpop$height[1:3],"\n",file="myfile",sep=" , ")
```

Redirecting

- `sink("myfile.txt")` will redirect the console (strictly the R output) to myfile.txt
  - `sink()` restores output
  - Can also check how many are being used with `sink.number()`
  - Try, `sink("list.txt"); 1:10 ; sink()`
- For graphics there are specific devices, e.g.
  - pdf("myplot.pdf")
  - jpeg("myplot.jpeg")
- More on this later when we look at graphics

Reading from files

- R understands directory handling & paths:
  - To get the current working directory `getwd()`
  - To set the current working directory setwd e.g.
    - setwd("C:/Users/Rob/Documents")
  - `dir()` lists current directory contents
- Already noted: `source("myfile.R")` will execute script
- Simplest way to read a table of separate vals:
  - `mytab <- read.table("list.txt")`
  - Check help – can specify separator

Reading from files: specialist

- R can read other stats-related formats too
  - Excel – `read.xls()`
  - SPSS – `read.spss()`
  - Minitab – `read.mtp()`
- Comma separated variable files too:
  - `read.csv()`
  - Normally expects variables names in first line, e.g

```
Height, mass, name
1.7, 60, John
```

Reading from files: the web!

- Instead of a directory name, you can give an http address! Try this:
  - `mytab <-`

    `read.table("http://ap.smu.ca/~thacker/list.txt")`
- Note if you need passwords then there are options, including using the Rcurl package
  - Obvious point – passwords in scripts are a bad idea!
  - You can easily forget and send someone a script with your passwords!

Quick thoughts on "data manipulation"

- Selecting, inserting, deleting are all supported in R, but not always in a simple way
  - Strictly speaking a data manipulation language like SQL is needed – see the RSQLite package
- So never a bad idea to preprocess data first
- If your data is small enough you could always use a spreadsheet
  - Excel is surprisingly powerful in terms of the manipulations it can do
    - Even with a few tens of thousands of data elements

Looping & timing

- R is "vector language", and you should try and think that way
  - Of course it isn't always possible to vectorize
- To time how long operation takes use `Sys.time()`

```
start.time <- Sys.time()

end.time <- Sys.time()
time.taken = start.time - end.time
print(time.taken)
```

Looping – what can you do?

- R supports three types of loops
  - for
  - while
  - repeat

For loops

- Think about a vector of loop values controlling loop

```
for (i in 1:10000) {
        An operation
}
```

- For strides: `steps <- seq(1,10000,by=2)` then

```
for (i in steps) {
        An operation
}
```

- Like a loop with a loop index array

for loop: Exercise

- Create a vector `mylist` with values 1:100000
- Create a vector `mylist.sq = NULL`
- Now write a loop from 1:100000 that sets each element of `mylist.sq` to the square of `mylist`
- Time this using `Sys.time()`
- Nxt instead declare `mylist.sq = rep(0,100000)`
- `rm(mylist,mylist.sq)` rerun
- What time difference do you get?

while loop

- While loops are the next step up in complexity
  - Consider the condition at the beginning of each iteration
- General format:

```
while (condition){
      An operation
}
```

- Example, `for` loop as `while` loop (try it):

```
j = 1
while (j<=10) {
cat("j=",j," \n")
j=j+1
}
```

if constructs

- R supports if (condition) control structures

```
x = 1
if (x>0) {
    cat("x is positive")
} else if (x < 0) {
cat("x is negative")
} else {
cat("x is zero")
}
```

- Can use if's to break out of loops

Using breaks

- Flow control like "break out goto"
- Can use in any kind of loop structure – even for

```
x <- 1:10
for (j in x) {
    if (j == 7) {
        break
    }
    cat("j=",j,"\n")
    }
```

- Remember – position of break condition will determine whether following code is executed
    - Easy to trip yourself up on this…

repeat loop

- Repeat loops differ from while loops two ways
  - 1) There's no explicit condition following repeat
  - 2) you must break using a condition to leave the loop
- Example for loop in repeat format:

```
j = 1
repeat {
    if (j == 7) {
        break
    }
    cat("j=",j,"\n")
    j=j+1
    }
```

Important to watch where you put the break point – easy to get your loop logic wrong. When in doubt, print out…

Tips for better performance of “for loops”

- Ensure the list/vector you are writing to is the right length before you start
  - Growing the list/vector on each iteration is expensive
  - Even if you don’t know exact length, you probably have an upper bound
- Get as many operations outside the loop as you can

Why is vectorization so much better?

- It goes beyond just compiled vs interpreted
- Each call to a function requires R to determine what type data is being passed and then send the correct data type to a compiled function
  - For vectors this is straightforward – all same datatype
  - Doing this *once* rather than repeated calls is obviously better!
  - These issues are sorted out at compile time in compiled languages

When do you have to use loops?

- If one iteration depends on the previous one (recall data dependence issues)
- If a function doesn’t take a vector input
- Sometimes recursive situations require it too

Plots!

- As for most packages, simple plots are easy, more detailed ones need more qualifiers
- Try this: `plot(ourpop$height,ourpop$mass)`
- To create a line-point plot try

```
plot(ourpop$height,ourpop$mass,type="o")
```

- Highlights that R plots in data order when creating line graphs
- So need to create an ordering array – that's not difficult

```
op.sort = order(ourpop$height)
```

- Now try

```
plot(ourpop$height[op.sort],ourpop$mass[op.sort],type="o")
```

Labels and ranges

- Axis labels:

```
plot(ourpop$height[op.sort],ourpop$mass[op.sort],type="o",
ylab="Mass/kg",xlab="Height/m")
```

- Setting ranges

```
plot(ourpop$height[op.sort],ourpop$mass[op.sort],typ
e="o",ylab="Mass/kg",xlab="Height/m",ylim=c(40,90),x
lim=c(1,2))
```

- Tip: make sure you don't use a colon e.g. ylim=c(1:2) – that will fail
- Colour: try

```
plot(ourpop$height[op.sort],ourpop$mass[op.sort],typ
e="o",ylab="Mass/kg",xlab="Height/m",ylim=c(40,90),x
lim=c(1,2),col="blue")
```

- To add a title, use the title function: `title(main="Mass vs weight")`

Creating hardcopy

- Need to pipe to file and appropriate device
- pdf example:

```
pdf("myfile.pdf")
plot(ourpop$height[op.sort],ourpop$mass[op.sort],type="o",ylab="Mass/kg",xlab="Height/m",ylim=c(40,90),xlim=c(1,2),col="blue")
dev.off() #flush to file
```

- **Always remember to close with dev.off()**
- `help("device")` will tell which graphical devices are available (typically, pdf, ps, xfig, bitmap+…)

Annotation & legends

- You can add text using the text command e.g.

```
text(1.2,80,"Hi there")
```

x,y position are the first two values

Note you can also use text to label points:

```
text(ourpop$height[op.sort],ourpop$mass[op.sort],our
pop$name[op.sort],cex=0.6,pos=4, col="red")
```

- Legends:

```
legend("topleft",lty=1,col="blue",pch=21,"H
eights")
```

- A bit messy, but it works! Try help("legend") for more info

Simple fitting

- Linear fitting can be done with `lm(y~x)`

    Try: `lm(ourpop$mass~ourpop$height)`

- Should get

```
Call:
lm(formula = ourpop$mass ~ ourpop$height)
Coefficients:
      (Intercept)  ourpop$height
         -24.85            55.23
```

- Better to put into a "fit" object, e.g.

```
      fit = lm(ourpop$height~ourpop$mass)
      summary(fit)
```

- You can plot the residuals etc using `plot(fit),` & plot the fit with `abline(fit)`

Summary

- Data frames are a powerful way of storing data that can be easily subsetted
- Avoid loops when you can – vectorization is much faster
- Basic I/O is much like a terminal, but be aware there are more sophisticated packages out there
- Plotting is tricky, but amazingly powerful, we've only just touched on things today