An Introduction to the R Environment

Peter Dalgaard

Center for Statistics
Copenhagen Business School

MPAS Lecture April 2010

Outline

Practicalities

- Short introduction (approx. 90 min)
- Focus on things relevant to your project
- Script of demos on MPAS web page

Plan

- Elementary things about R
  - Data types and some important functions
  - Matrix calculus
  - Working with data sets
  - R as a programming language
- Basic statistics and tests
- Modeling tools
- Elementary Graphics

The R environment

- Built around the programming language R, an Open Source dialect of the S language
- R is Free Software, and runs on a variety of platforms (I'll be using Linux here).
- Command-line execution based on *function calls*
- Extensible with user functions
- Workspace containing data and functions
- Various graphics devices (interactive and non-interactive)

R is a vectorized language

- The basic data type in R is a vector
- Vectors often represent data (e.g. the age for each participant in a study), but also other things like regression coefficients, plot limits, cut points, etc.
- Data types: Numeric (integer/double), character (strings), logical (TRUE/FALSE)
- Factor (really integer + level attribute) for categorical variables
- Lists (generic vectors)

Basic operations

- Standard arithmetic is *vectorized*: `x + y` adds each element of `x` to the corresponding element of `y`
- Recycling: If operating on two vectors of different length, the shorter one is replicated (with warning if it is not an even multiple)
- `c` — concatenate: `c(7, 9, 13)`
- `seq` — sequences: `seq(1, 9, 2)`, short form: `1:5` is the same as `seq(1,5,1)`
- `rep` — replication `rep(1:3, 3:1)` (1 1 1 2 2 3)
- `sum`, `mean`, `range`, ...

Demo 1

```
x <- round(rnorm(10,mean=20,sd=5)) # simulate data
x
mean(x)
m <- mean(x)
x - m # notice recycling
sqrt(sum((x - m)^2)/9)
sd(x)
```

Smart indexing

- `a[5]` single element
- `a[c(5,6,7)]` several elements
- `a[-6]` all except the 6th
- `a[b>200]` index by logical vector
- `a["name"]` by name

Extended data types

- The basic vector types can be combined and extended to form more complex data structures
- *Attributes* extend a basic type with further information. E.g., a vector can have a names attribute, for more readable printing
- *Classes* have two main functions:
  - Hide details
  - Allow *function dispatch* (functions that behave differently depending on the class.

Factors

- Factors are used to describe nominal variables (the term originates from *factorial designs*)
- Internally, they are just integer codes plus a set of names for the *levels*
- They have class `"factor"` making them (a) print nicely and (b) behave consistently
- A factor can also be *ordered* (class `"ordered"`), signifying that there is a natural sort order on the levels
- In model specifications, factors play a fundamental role by indicating that a variable should be treated as a classification rather than as a quantitative variable.

Lists (generic vectors)

- A vector where the elements can have different types
- Functions often return (classed) lists
- Indexing:
  - `lst$A`
  - `lst[[1]]` first element
  - `lst[1]` list containing the first element

Demo 2

(indexing, factors, lists)

Matrix algebra

- R contains a pretty full set of primitives for matrix calculus
- `A %*% B` for matrix multiplication
- `solve(A, b)` for solving linear equations. (`solve(A)` for matrix inverse)
- `t(A)` for transpose of a matrix.

Elementary matrix manipulations

- Matrices are implemented as vectors with a `dim` attribute (of length 2)
- Constructor function: `matrix(1:4,2,2)`
- Indexing in the usual way `M[i,j]`, with all the features of "smart indexing". `M[,j]` is *j*th column, etc.
- Special feature for matrices and arrays: *Matrix indexing*, `M[A]` where `A` has as many columns as `M` has dimensions.

Demo 3

Permutation matrix (*Mx* permutes the elements of *x*)

```
perm <- sample(5) # w/o replacement
n <- length(perm)
M <- matrix(0,n,n)
M[cbind(1:n,perm)] <- 1
M
perm
M %*% 1:n
```

Other matrix techniques

- `diag` has multiple functions: creation of diagonal matrices, extracting, and manipulating the diagonal of a matrix. **Beware:** `diag(v)` is ambiguous if `v` can have length 1.
- `row(X)`, `col(X)` are convenient for generating some forms of matrices.
- `upper.tri` and `lower.tri` generate indexes for accessing the upper/lower triangle of a matrix.
- Matrices can be "glued together" using `cbind` and `rbind`

Row and column matrices

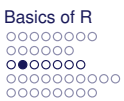
- R usually treats vectors as row or column matrices "as appropriate" (i.e., it guesses)
- E.g., you can left- or right-multiply a vector by a matrix, even though the latter formally requires transposition
- And even do `y %*% x` to get the inner product $y'x$
- If you want to be explicit about it, you can use `rbind` or `cbind` to create the appropriate single-row or single-column matrix.

Using `drop()` and `drop=FALSE`

- Default: If a dimension has length one, it is dropped from results. `M[1,]` is a vector, not $1 \times n$ matrix.
- Often convenient, but source of obscure bugs
- Watch out for extreme cases
- Use `M[1,drop=FALSE]` to prevent this
- Conversely sometimes you get a matrix and want a vector, as in `drop(M %*% 1:n)`

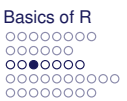
Some Basic Functions

- Constructors of simple objects
- Single-column modifications
- Modifying and subsetting data frames



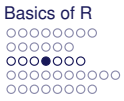
Constructors

- R deals with many kinds of objects besides data sets
- Need to have ways of constructing them from the command line
- We have (briefly) seen the `c` and `list` functions
- Notice the naming forms `c(boys=1.2, girls=1.1)`
- Extracting and setting names with `names(x)`
- For matrices and arrays, use the (surprise) `matrix` and `array` functions. `data.frame` for data frames.
- It is also fairly common to construct a matrix from its columns using `cbind`



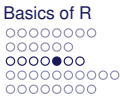
Demo 4

```
x <- c(boys = 1.2, girls = 1.1)
x
names(x)
names(x) <- c("M", "F")
x
matrix(1:4,ncol=2)
cbind(x=0:3,"exp(x)"=exp(0:3))
```

The `factor` Function

- This is typically used when `read.table` gets it wrong
- E.g. group codes read as numeric
- Or read as factors, but with levels in the wrong order (e.g. `c("rare", "medium", "well-done")` sorted alphabetically.)
- Notice the slightly confusing use of `levels` and `labels` arguments.
- `levels` are the value codes *on input*
- `labels` are the value codes *on output* (and become the levels of the resulting factor)



Demo 5

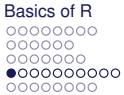
```
aq <- airquality
aq$Month <- factor(aq$Month, levels=5:9,
                   labels=month.name[5:9])
aq$Month
levels(aq$Month) <- month.abb[5:9]
aq$Month
```

The `cut` Function

- The `cut` function converts a numerical variable into k groups according to a set of break points
- The breaks must include **all** $k+1$ interval endpoints
- The intervals are left-open, right-closed by default (`right=FALSE` changes that)
- The lowest endpoint is *not* included by default (`include.lowest=TRUE` to change it.)

Demo 6

```
library(ISwR)
age <- juul$age
age <- age[age >= 10 & age <= 16]
range(age)
agegr <- cut(age, seq(10,16,2), right=FALSE, include.lowest=TR
length(age)
table(agegr)
agegr2 <- cut(age, seq(10,16,2), right=FALSE)
table(agegr2)
```

Data frames

- Like data set in other packages
- Technically: Lists of vectors/factors of same length
- Indexed like matrices (Beware, though: Data frames are *not* matrices) *or* lists
- Generate from read operation or with `data.frame`
- Many sample data frames are available in packages

Attaching data frames

- Normally variables in data frames must be *qualified* by the name of the data frame
- I.e., you have to say `airquality$Month`, in case there is another `Month` in another data frame
- However, if you `attach(airquality)`, then you can access its contents without the qualifier

Demo 7

```
airquality$Month
airquality[airquality$Month==5,]
oz <- airquality[airquality$Month==5,]$Ozone
mean(oz)
mean(oz, na.rm=TRUE)

attach(airquality)
mean(Ozone, na.rm=TRUE)
tapply(Ozone, Month, mean, na.rm=TRUE)
detach()
```


Reading data

- Simple data vectors can be read using `scan()`
- Data frames can be read from most reasonably structured text file formats (space separated columns, tab- and comma-delimited files) using `read.table()` or `read.delim2()`.
- A nice expedient is to read from the clipboard (`read.delim2("clipboard")`)
- The `foreign` package can read files from Stata, SPSS, ...
- Finding the full pathname of a file is tricky. Either change the working directory use `file.choose()` to browse to it.

The workspace

- The *global environment* contains R objects created on the command line.
- There is an additional *search path* of loaded packages and attached data frames.
- When you request an object by name, R looks first in the global environment, and if it doesn't find it there, it continues along the search path.
- The search path is maintained by `library()`, `attach()`, and `detach()`
- Notice that objects in the global environment may mask objects in packages and attached data frames

Demo 8

```
search()
library(ISwR)
data(intake) # From ISwR
ls()
attach(intake)
search()
ls("intake") # show variables in attached data frame
post - pre
rm(intake) # remove data frame
detach()   # remove from search path
```

A Common Mistake

```
attach(juul2)
sex <- factor(sex)
tapply(height, sex, mean, na.rm=TRUE)
detach()
attach(subset(juul2, age > 25))
sex <- factor(sex)
tapply(height, sex, mean, na.rm=TRUE)
```

You get an error saying that `height` and `sex` are of different length. What went wrong?
Second time around, `sex` was found in the global environment *before* the attached data frame.

Subsetting Data Frames

- The syntax for indexing data frames easily gets heavy: `airquality[airquality$Month == 5 & airquality$Ozone > 50,]`
- The `subset` function allows you to say `subset(airquality, Month == 5 & Ozone > 50)`. I.e., it evaluates the second argument within the data frame.

Attaching or not

- Attached data frames can lead to confusion, especially if you modify their variables
- On the other hand you do probably want the simplified notation for convenience (and things like axis labels do depend on it)
- I tend to recommend attaching only **after** making all necessary transformation. Others are more radical and discourage `attach` entirely
- In either case, it is useful to know some functions that allow you to work with the simplified notation, without actually attaching
- `subset` is one example, the others are `transform`, `with`, and `within`

Demo 9

```
juulyoung <- subset(juul, age < 20)

with(juulyoung, plot(age, igf1))

juulnew <- transform(juul,
                     sex=factor(sex, labels=c("M","F")),
                     tanner=factor(tanner))

juulnew2 <- within(juul, {
   sex <- factor(sex, labels=c("M","F"))
   tanner <- factor(tanner)
   rm(testvol)
})
```

Notice that `within` and `transform` are very similar, but `within` allows a bit more flexibility.

R programming

- R is a full programming language
  - Flow control
  - User-written functions
- You will soon be writing your own R functions
- For repeated ad-hoc tasks
- Or, because functions are required input for certain tasks (e.g., optimizers)
- User-written functions are not substantially different from system functions, making R very smoothly extensible.

Argument matching

- A very simple function `logit <- function(p) log(p/(1-p))`
- Usage: `logit(0.5)`
- The return value of a function is the result of the last expression, unless there is an explicit `return()`
- Formal arguments (`p`)
- Actual arguments (`0.5`)
- Positional matching: `plot(x,y)`
- Keyword matching: `t.test(x ~ g, mu=2, alternative="less")`
- Partial matching: `t.test(x ~ g, mu=2, alt="l")`

Scoping

- Objects defined in a function will generally be local to that function and *usually* become unavailable when the function exits
- However, the full story is more complicated
- . . . so we skip most of it.
- One basic point is that a function can “see” objects in the parent in which it was defined (*lexical scoping*).

Flow control

- `if/else`
- `switch()`
- `for` loops
- `repeat`, `while`

Conditional Expressions

```
if (paired)
    xok <- yok <- complete.cases(x, y)
else {
    yok <- !is.na(y)
    xok <- !is.na(x)
}

twopi <- if(clockwise) -2*pi else 2*pi
```

- Notice that the condition is a *scalar*. It doesn't vectorize; only one branch is taken. (Compare the `ifelse()` function)
- Conditions can be combined using `&&` and operators
- An `if` expression does have a result, which can be used as in the 2nd example above

The `for` loop

```
for (i in 1:n) ....
for (i in names) ....
for (ns in list(...)) ...
for(pkg in getOption("defaultPackages")) {....
```

- Notice that the loop is over a vector or a list
- Inside the body of a for loop, the loop variable takes on the value of each element in turn
- Even when it is a numeric sequence, the entire vector is stored in memory. Fortunately, this is rarely a problem in practice
- You can skip to the next element with `next` or exit the loop completely with `break`,

Not Using `for` Loops

- Many applications of for loops have the following structure
  - Allocate a list/vector to hold the results
  - Loop, saving the results of each iteration in turn
- (A common buglet is that people *extend* the vector on every iteration, which can become terribly inefficient)
- This structure can be abstracted into a single function call

  ```
  lapply(lst, fun)
  ```

  which causes `fun` to be applied to each element of `lst`, and returns a list of the results. (Further arguments can be added and will be passed on to `fun`)

Further Apply-functions

- `lapply` – list-apply
- `sapply` – simplifying apply
- `tapply` – tabulating apply
- `apply`, `sweep` – along slices of tables
- `replicate` – repeat expression

Simple Descriptives

- `mean`, `median`, `sd`, etc.
- `quantile(x,p)` where `p` is a vector of proportions
- (actually, there is *nine* different types of quantiles)
- `summary` gives some key quantities or a variable, depending on its type. This also works on entire data frames

Tabulation

- For simple tables of discrete variables, use the `table` function, as in `table(sex,tanner)`, or `xtabs`
- For tables of descriptives the first choice is `tapply`, for example `tapply(age, tanner, mean, na.rm=TRUE`
- Explanation: `age` is split according to groups and `mean` is called on each piece with an *extra argument*, evaluating `mean(age, na.rm=TRUE)` within each group.

Neater Tables

- Some variations over `tapply` is given by the `by` and `aggregate` functions
- Multiway tables are often hard to read and use for presentation purposes. Look into the `ftable` ("flattened" tables) and Martyn Plummer's `stat.table` function in the `Epi` package.

Some standard procedures

- Continuous data by group: `t.test`, `wilcox.test`, `oneway.test`, `kruskal.test`
- Categorical data: `prop.test`, `chisq.test`, `fisher.test`
- Correlations: `cor.test`, with options for nonparametrics

Demo 10

```
library(ISwR)
attach(intake)

t.test(pre, post, paired=TRUE)

detach()
```


Demo 11

```
caesar.shoe

chisq.test(caesar.shoe)
fisher.test(caesar.shoe)

x <- caesar.shoe[1,]
n <- margin.table(caesar.shoe,2)
rbind(x,n)
prop.trend.test(x,n)
```


Modeling Tools: Overview

- Model formulas
- Model objects and summaries
- Comparing models
- Evaluating model fit (plot methods)
- Generalized linear models

Model formulas

- Linear model, $y = X\beta + \varepsilon$
- In practice something like

$$y = \beta_0 + \beta_1 \times \text{height} + \beta_2 \times 1_{(\text{type}=2)} + \beta_3 \times 1_{(\text{type}=3)} + \varepsilon$$

- Wilkinson-Rogers formulas:

$$y = \text{height} + \text{type}$$

(Interpretation depends on whether variables are categorical or continuous)

Model formulas in R

- R representation `y ~ height + type` where `type` is a *factor*
- Interactions `a:b`, `a*b = a + b + a:b`
- Algebra (`a:(b + c) = a:b + a:c` etc.)
- Notice special interpretation of operators

Fitting linear models

```
data(airquality)
aq <- transform(airquality, Month=factor(Month))
fit.aq <- lm(log(Ozone) ~ Solar.R + Wind +
             Temp + Month, data=aq)
```

- `lm` generates a *fitted model object*
- Extract information from model object
- Fit other models based on model object

Inspecting model objects

- Extract information about the fit
- `summary(fit.aq)`
- `fitted(fit.aq)`, `resid(fit.aq)`
- `anova(model1, model2)`
- `plot(fit.aq)` – diagnostics
- `predict(fit.aq, newdata)`

Model search

- `anova(model)` "Type I" sum of squares
- `drop1` ("Type III")
- `step` (AIC/BIC) criteria
- `update`

Demo 12

```
aq <- transform(airquality, Month=factor(Month))
fit.aq <- lm(log(Ozone) ~ Solar.R + Wind +
             Temp + Month, data=aq)
fit.aq2 <- update(fit.aq, ~ . - Month)
summary(fit.aq)
plot(fit.aq)
drop1(fit.aq, test="F")
anova(fit.aq, fit.aq2)
```


R graphics

- The standard interface (i.e., not the `lattice` package)
- Customizing plots
- Graphics parameters
- Math on plots

Standard R graphics

- Ink on paper model; once something is drawn it cannot be erased.
- Sensible default plots
- Arguments can override defaults
- Options to *turn off* various elements of plots (e.g. the axes)
- Functions to *add* elements.

Basic x-y plots

- The `plot` function with one or two numeric arguments
- Scatterplot or line plot (or both) depending on `type` argument: `"l"` for *l*ines, `"p"` for *p*oints (the default), `"b"` for *b*oth, plus quite a few more.
- Functions for adding to a plot: `lines, points, segments, abline, text, mtext, axis`
- Also: formula interface, `plot(y~x)`

Specific plots

- Histograms — `hist(x)`
- Density plots — `plot(density(x))`
- Boxplots — `boxplot(x)`
- Barplots — `barplot(x)` (`x` can be a matrix)
- Pies — `pie()`
- Matrix plots (multiple y columns) — `matplot()`

Graphical parameters

- Arguments to `plot` et al. (67 possibilities!)
- The `par` function can be used to set most of them persistently. Most info is found via `help(par)`
- Look them up! Here are some of the more commonly used:
  - Point and line characteristics: `pch, col, lty, lwd`
  - Multiframe layout: `mfrow, mfcol`
  - Axes: `xlim, ylim, xaxt, yaxt, log`

Demo 13

```
par(mfrow=c(2,2))
matplot(intake)
matplot(t(intake))
matplot(t(intake),type="b")
matplot(t(intake),type="b",pch=1:11,col="black",
        lty="solid", xaxt="n")
axis(1,at=1:2,labels=names(intake))
```


Math on plots

- Sort of like TeX
- Works on unevaluated expressions (quote(alpha), expression(alpha))
- Special conventions: `^`,`[]` sub/superscript, special names `alpha`, `sum`, `int`
- See `help(plotmath)`

Demo 14

```
y <- rnorm(25)
curve(dnorm(x, mean(y), sd(y)), from=-3, to=3)
rug(y)
abline(h=0)
title(main=substitute(paste(mu==m, "    ", sigma==s),
                      list(m=mean(y), s=sd(y)) ))
```