Introduction to R for econometricians

It is easy to lie with statistics, but easier to lie without them. FRED MOSTELLER

Why R

- Statisticians use R. Some econometricians are starting to use it.
- It's a flexible statistical programming environment.
- It has lots of user-written packages, functions which are relatively easy to modify.
- It's free.

Introducing R

"S is a programming language and environment for all kinds of computing involving data. It has a simple goal: to turn ideas into software, quickly and faithfully." JOHN M. CHAMBERS

- S is a language for "programming with data".
  JOHN CHAMBERS of Bell Labs has been its main developer for more than two decades.
- R is an Open Source system originally written by ROSS IHAKA and ROBERT GENTLEMAN at the University of Auckland in about 1994.
- R is not unlike S (actually they are very similar!)
- R is now developed by a small core team, for all details see:
  `www.r-project.org`.

Introducing R . . . continued

Commands to `R` are expressions or assignments.

expression

```
4/3 * pi * (27)^3                      : [1] 82447.96
```

assignment

```
a <- 27
```

Everything within the R language is an object.
Normally `R` objects are accessed by their name, which is made up from *letters*, *digits* $0-9$ in non-initial position, or a *period, "."*, that acts like a letter. `R` is case sensitive.

Every object has a class.

Introducing R . . . help & comment

getting help

All functions and data sets in `R` have a documentation! For information on a function or data set,

```
?function-name,
```

which is equivalent to

```
help(function-name).
```

To search all help pages for a specific term

```
help.search("term").
```

Help pages can also be displayed in a HTML version, therefore

```
help.start().
```

writing comments

A line starting with `#` is treated as a comment and not processed.

Introducing R ... your workspace

Objects are normally stored in a workspace.

`ls()` lists all objects currently in your workspace

`rm(`*`object`*`)` removes *object* from your workspace

`save(`*`object`*`, file=`*`path/file`*`)` saves an *object* to a *file*

`load(`*`path/file`*`)` loads an *object* from a *file*

`save.image()` saves your workspace to a file called `.RData` in your working directory.
Happens also if you type `q("yes")`.

`getwd()` shows the path of your current working directory

`setwd(`*`path`*`)` allows you to set a new `path` for your current working directory

Introducing R . . . additional packages

The functionality of an `R` installation can be extended by packages. Additional packages provide you functions, data sets, and the corresponding documentation. A growing number of packages is available from CRAN

`CRAN.r-project.org`

`library()` shows all packages installed on your system

`library(asuR)` loads an already installed package (here asuR) (asuR is the package that accompanies this course)

`library(help=asuR)` displays all functions, data sets, and vignettes in a package (here asuR)

`data()` shows the data sets of all installed packages

`data(package="asuR")` shows data set(s) from a package, here asuR

`data(pea)` loads the data set "pea" to your workspace (therefore the package asuR has to be loaded)

Introducing R . . . **vector**

`c()` creates a vector of the specified elements (`c` for concatenate)

```
> genus <- c("Daphnia", "Boletus", "Hippopotamus", "Salmo", "Linaria",
+     "Ixodes", "Apis")
> species <- c("magna", "edulis", "amphibius", "trutta", "alpina",
+     "ricinus", "mellifera")
> weight <- c(0.001, 100, 3200000, 1000, 2.56, 0.001, 0.01)
> legs <- as.integer(c(0, 0, 4, 0, 0, 8, 6))
> animal <- c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE)
```

`length()` returns the length of a vector

```
length(genus)                                   : [1] 7
```

`paste()` takes two vectors and concatenates them as characters

```
name <- paste(genus, species)
```

Introducing R ... vector ... continued

`seq()` to generate sequences of numbers

```
seq(from=4, to=7)                        : [1] 4 5 6 7
4:7                                      # short form of the previous
seq(from=4, to=7, by=0.5)                : [1] 4.0 4.5 5.0 5.5 6.0 6.5 7.0
```

`rep()` to replicate elements of a vector

```
rep(c(2,4,6), times=3)                   : [1] 2 4 6 2 4 6 2 4 6
rep(c(2,4,6), each=3)                    : [1] 2 2 2 4 4 4 6 6 6
rep(c(2,4,6), times=3, each=2)           : [1] 2 2 4 4 6 6 2 2 4 4 6 6 2 2 4 4 6
```

Introducing R . . . factor

A special type of a vector that usually stores categorical variables.

`factor()` makes a factor out of a vector

```
> kingdom <- factor(c("animal", "fungi", "animal", "animal", "plant",
+     "animal", "animal"))
```

`levels()` provides a character vector with the levels of a factor

```
levels(kingdom)                              : [1] "animal" "fungi"  "plant"
```

Internally a factor is stored as a set of codes and an attribute giving the corresponding levels.

```
unclass(kingdom)                         : [1] 1 2 1 1 3 1 1
                                         : attr(,"levels")
                                         : [1] "animal" "fungi"  "plant"
```

Introducing R . . . data.frame

Used to store data. It is a list of variables, all of the same length, possibly of different types.

`data.frame()` a function to generate data frames

```
> bio <- data.frame(name = I(paste(genus, species)), weight_g = weight,
+     leg_no = legs, animal = animal, kingdom = kingdom)
```

`names()` displays the names of the variables in a data frame

`row.names()` displays the row names

`str()` a useful summary of the structure of a data frame

`summary()` provides a summary of all variables in a data frame

`attach()` makes the variables of a data frame accessible by their name

`detach()` the inverse

`write.table()` to write a data frame to a text file

```
> write.table(bio, file = "~/temp/bio.txt", row.names = FALSE,
+     sep = "\t")
```

`read.table()` to read a data frame from a text file

```
> bio.new <- read.table(file = "~/temp/bio.txt", header = TRUE,
+     sep = "\t", row.names = "name", colClasses = c("character",
+         "numeric", "integer", "logical", "factor"))
```

Introducing R . . . matrix & array

A matrix has all its arguments of the same type and always two dimensions. An array is like a matrix but with a flexible number of dimensions.

`matrix()` a function to create a matrix from a vector

`rbind()` takes vectors and binds them as rows together

`cbind()` takes vectors and binds them as columns together

```
> mat <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
> mat1 <- matrix(c(1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12), nrow = 3,
+     ncol = 4)
> mat2 <- rbind(c(1:4), c(5:8), c(9:12))
> mat3 <- cbind(c(1, 5, 9), c(2, 6, 10), c(3, 7, 11), c(4, 8, 12))
```

`dim()` returns the dimensions

`dimnames()` to give a name to the columns and rows

```
> dimnames(mat) <- list(c("first row", "second row", "last row"),
+     paste("c_", 1:4, sep = ""))
```

Introducing R . . . list

A list is a collection of *components* that can be from different classes and of different length.

```
> organisms <- list(animals = list(genera = c("Daphnia", "Hippopotamus",
+     "Salmo", "Ixodes", "Apis"), publications = 110), plants = list(genera = "Linaria"
+     publications = 50), fungi = "Boletus")
```

`$` to extract components by their name

```
organisms$animals$genera            : [1] "Daphnia" "Hippopotamus" "Salmo" "Ixodes'
                                    # a character vector of length five
```

`[[` to extract a component by its position

```
organisms[[1]][[1]]                 : [1] "Daphnia" "Hippopotamus" "Salmo" "Ixodes'
                                    # a character vector of length five
```

`[` to extract a sub-vector

```
organisms[[1]][1]                   : $genera
                                    : [1] "Daphnia" "Hippopotamus" "Salmo" "Ixodes'
                                    # a list of length one
```

Introducing R ... function

```
function-name  <- function(argument1, argument2, ...){

      function.body

}
```

An example for calculating the t value according to

$$\frac{(\bar{X}_1 - \bar{X}_2) - (\mu_1 - \mu_2)}{S_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \sim t_{n_1+n_2-2} \qquad \text{with,} \quad S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}$$

```
> my.t <- function(mean1, mean2, s1, s2, n1, n2, mu.diff = 0) {
+     s.pooled <- ((n1 - 1) * s1^2 + (n2 - 1) * s2^2)/(n1 + n2 -
+         2)
+     t <- ((mean1 - mean2) - (mu.diff))/(sqrt(s.pooled * (1/n1 +
+         1/n2)))
+     cat(round(t, 2))
+     cat(paste(": compare to a t dist. with", n1 + n2 - 2, "df \n"))
+ }
```

The function can then be called with, e.g.,

```
my.t(mean1=24, mean2=18, s1=3, s2=4, n1=34, n2=50)   : 7.43; compare to t-dist. 82 df
```

Reading in data

R is able to read data from many formats.

`read.table()` to read a data frame from a text file

```
> bio.new <- read.table(file = "~/temp/bio.txt", header = TRUE,
+     sep = "\t", row.names = "name", colClasses = c("character",
+         "numeric", "integer", "logical", "factor"))
```

`read.dta()` to read a data frame from a Stata file, Stata files automatically have headers.

```
> library(foreign)
> mydata <- read.dta("C:/WINDOWS/Desktop/blah.dta")
```

`read.csv()` to read a data frame from a csv file

indexing

Many data manipulations in `R` rely on indexing. Indexing is used to address a subsets of vectors, matrices, or data frame. The general form for using an index is

```
object[index-vector] .
```

Indexing is used to select a subset of an object,

```
new.object <- object[index-vector] ,
```

to replace a subset of an object,

```
object[index-vector] <- new.element ,
```

or to sort an object (see below for an example).

On the following pages we will see what types the ***index-vector*** can take (for vectors and data frames & matrices respectively).

The examples below always apply to the following vector object

```
> x <- c(11, 44, 33, NA, 22)
```

indexing . . . vectors

logical vector

- must be of the same length as the vector
- values corresponding to TRUE are included; corresponding to FALSE are omitted (NA inserts an NA at the corresponding position)

```
x[c(TRUE, FALSE, FALSE, FALSE, TRUE)]    : [1] 11 22
x[!is.na(x)]                             : [1] 11 44 33 22
x[x>=33]                                 : [1] 44 33 NA
x[(x==33 | x==44) & !is.na(x)]           : [1] 44 33
```

vector of positive integers (factors)

- the values of the integers must be smaller or equal to the length of the vector
- the corresponding elements are selected and concatenated in the order they were selected
- for factors as index vector (works like: x[unclass(factor)])

```
x[c(1,2,5)]                              : [1] 11 44 22
x[1:3]                                   : [1] 11 44 33
x[order(x)]                              : [1] 11 22 33 44 NA
```

vector of negative integers

- the absolute values of the integers must be smaller or equal to the length of the vector
- the corresponding elements are excluded

```
x[c(-1,-2,-5)]                                : [1] 33 NA
```

vector of character strings

- only applies if object has names
- the corresponding elements are selected and concatenated in the order they were selected

```
names(x) <- c("first", "largest", "middle", "non.available", "second")
x[c("largest", "first")]
                                              : largest   first
                                              :      44      11
```

indexing . . . data frames and matrices

Columns in a data frame are often selected with the `$` operator

```
bio$names                                        # equivalent to bio[,"names"]
```

Data frames and matrices can be indexed by giving two indices ([*`rows,columns`*]).

```
bio[bio$animal, ]                                # all rows where animal is TRUE
bio[bio$weight_g>1, "name"]                      # name of all organisms heavier >1g
```

indexing ... data frames and matrices ... continued

An array (and therefore also a matrix) can be indexed by a $m \times k$ matrix. Each of the m rows of this matrix is used to select one element.

```
> mat <- matrix(1:9, ncol=3)              :      [,1] [,2] [,3]
                                          : [1,]    1    4    7
                                          : [2,]    2    5    8
                                          : [3,]    3    6    9
select <- rbind(c(2,1),c(3,1),c(3,2))
mat[select]                               : [1] 2 3 6
```

If you extract elements from a data frame or a matrix, the result is coerced to the lowest possible dimension. This default behavior can be changed by adding `drop=FALSE`,

```
bio[,"kingdom"]                               # returns a vector
bio[,"kingdom", drop=FALSE]                   # returns a data frame
```

Regressions

Let's consider the simplest case. Suppose we have a data frame called `byu` containing columns for `age`, `salary`, and `exper`. We want to regress various forms of `age` and `exper` on `salary`. A simple linear regression might be

```
> lm(byu$salary ~ byu$age + byu$exper)
```

or alternately:

```
> lm(salary ~ age + exper,data=byu)
```

as a third alternative, we could "attach" the dataframe, which makes its columns available as regular variables

```
> attach(byu)
> lm(salary ~ age + exper)
```

Regressions ... More

Using `lm()` results in an abbreviated summary being sent to the screen, giving only the β coefficient estimates. For more exhaustive analysis, we can save the results in as a data member or "fitted model"

```
> result <- lm(salary ~ age + exper + age*exper,data=byu)
> summary(result)
> myresid <- result$resid
> vcov(result)
```

To do a different model as an example:

```
> salary$agesq <- (salary$age)^2
> result <- lm(salary ~ age + agesq + log(exper) + age*log(exper),data=byu)
```

Regressions ... Models With Factors/Groups

When a variable included in a regression is of type factor, the requisite dummy variables are automatically created. For example, if we wanted to regress the adoption of personal computers (pc) on the number of employees in the firm (emple) and include a dummy for each state (where `state` is a vector of two letter abbreviations), we could simply run the regression

```
> summary(lm(pc~emple+state))

Call:
lm(formula = pc ~ emple + state)

Residuals:
    Min      1Q  Median      3Q     Max
-1.7543 -0.5505  0.3512  0.4272  0.5904

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.572e-01  6.769e-02   8.232   <2e-16 ***
emple        1.459e-04  1.083e-05  13.475   <2e-16 ***
stateAL     -4.774e-03  7.382e-02  -0.065    0.948
stateAR      2.249e-02  8.004e-02   0.281    0.779

 ...
```

```
stateWY      1.200e-01  1.041e-01   1.153    0.249
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
```

Quantile Regression with Sparse Matrix

Ordinary least squares regression methods produce an estimate of the expectation of the dependent variable conditional on the independent. Fitted values, then, are an estimate of the conditional mean. If instead of the conditional mean we want an estimate of the expected conditional median or some other quantile, we use the `rq()` command from the *quantreg* package. The syntax is essentially the same as `lm()` except that we can specify the parameter `tau`, which is the quantile we want (it is between 0 and 1). By default, `tau`=.5, which corresponds to a median regression—another name for least absolute deviation regression.

The advantage of R with quantile regression is that it allows sparse matrix. This makes some infeasible quantile regressions doable now.

Example: To run a quantile regression of firm sizes on some dummies variables, on a data set with about 12 million observations.

- Load the libraries:

  ```
  library(SparseM)
  library(quantreg)
  library(MASS)
  ```

Quantile Regression with Sparse Matrix - continued

- Compress the design matrix into sparse matrix:

```
x <- cbind(othe,priv,forei,collect,prov,yea,q3a,q3b,q4b,q4a2,priv.q3b,fore.q3b,col
sparsex <- as.matrix.csr(x)
```

- Compress the design matrix into sparse matrix:

```
rq1.int005 <- rq.fit.sfn(as.matrix.csr(sparsexxx), emp,tau=0.05)
```

Time Series

Suppose we are interested in estimating an ARIMA model for store sales, and we are not sure what ARIMA structure it has. One way to do it is to compare the model fitness statistics (AIC or BIC) to select model.

```
library(nlme)
attach(sales_store1)
store1.cat.sales.arma00.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma01.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma02.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma10.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma11.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma12.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma20.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma21.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
store1.cat.sales.arma22.gls <- gls(cat_sales_t ~ cat_sales_c + open + pre_open + open_tre
```

Time Series . . . Ljung-Box test

To test whether the residual is serial-correlated, Ljung-Box test can be run as:

```
Box.test(residuals(store1.cat.sales.arma00.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma01.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma02.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma10.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma11.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma12.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma20.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma21.gls),  type="Ljung")
Box.test(residuals(store1.cat.sales.arma22.gls),  type="Ljung")
```

Unit Root test and cointegration test

To test whether the depedent variable is nonstationary, Phillips-Perron test can be run as:

```
library(tseries)
attach(sales_store1)
pp.test(cat_sales_t)
```

To test whether the depedent variable and one of the independent variable are cointegrated, Phillips-Ouliaris test can be run as:

```
library(tseries)
attach(sales_store1)
store1.dir.sales.co <- po.test(cbind(total_directsales_t, total_directsales_c), demean=TR
```

graphics

The graphics system of `R` is very powerful. You can get a sample gallery with

```
demo(graphics).
```

A very large gallery with many complex examples is at
`http://addictedtor.free.fr/graphiques/`.