Student Workshop

Computational Tools for Economics Using R

Matteo Sostero
m.sostero@sssup.it

International Doctoral Program in Economics
Sant'Anna School of Advanced Studies, Pisa

October 2013

Outline

1. About the workshop
2. Getting to know R
3. Array Manipulation and Linear Algebra
4. Functions, Plotting and Optimisation
5. Computational Statistics
6. Data Analysis
7. Graphics with `ggplot2`
8. Econometrics
9. Procedural programming

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

About this Workshop

Computational Tools

- Part of the ‘toolkit’ of computational and empirical research.
- Direct applications to many subjects (Maths, Stats, Econometrics…).
- Develop a problem-solving mindset.

The R language

- Free, open source, cross-platform, with nice IDE.
- Very wide range of applications (Matlab + Stata + SPSS + …).
- Easy coding and plenty of (free) packages suited to particular needs.
- Large (and growing) user base and online community.
- Steep learning curve (occasionally cryptic syntax).
- Scattered references.

Workshop material

The online resource for this workshop is:

http://bit.ly/IDPEctw

It will contain the following material:

- The syllabus for the workshop.
- These slides (work in progress, they will be updated if necessary!).
- The scripts (R code) for the examples.
- Datasets used for examples, if not already included in R.

References and online resources

- UCLA Resources for R http://www.ats.ucla.edu/stat/r/
- Chang, 'Cookbook for R' http://www.cookbook-r.com/
- Paradis, E. (2002). 'R for Beginners.' http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf
- Crawley, M. J. (2012). *The R Book* (2nd ed., p. 1076). Wiley.
- Matloff, N. (2011). *The Art of R programming* (p. 400). No Starch Press.
- Wikibook 'R programming' http://en.wikibooks.org/wiki/R_Programming

Were to ask questions (search before asking!):

- R-help Google group https://groups.google.com/forum/#!forum/r-help-archive
- http://stackoverflow.com/

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with ggplot2

8 Econometrics

9 Procedural programming

Installing R and Rstudio

- Windows and Mac OS: download and install the `R` 'base' version for your operating system from http://cran.r-project.org/.
- In Ubuntu, from a shell:

  ```
  $ sudo gedit /etc/apt/sources.list
  ```

  add the line

  ```
  deb http://cran.mirror.garr.it/mirrors/CRAN/bin/linux/ubuntu/raring/
  ```

  save, close, then then run from the shell:

  ```
  $ sudo apt-get update
  $ sudo apt-get install r-base
  ```

- In all OSs: Download and install the `Rstudio` integrated development environment from http://www.rstudio.com/.

On a 64-bit machine, both 32-bit and 64-bit versions are installed, but we will rely on the 32-bit one for compatibility reasons.

Getting to know `Rstudio`

`Rstudio` is an *integrated development environment*, as it brings together most components needed for programming:

Script	Workspace History Files
Console	Plots Packages Help

The most important are:

- A *console* that interprets commands, to interact with the `R` 'engine'.
- A *script* editor, to develop and save sequences of commands, to be interpreted by the console.

`R` is an *interpreted* language, meaning (roughly) that commands can be executed line-by-line, without explicit compilation.

A first session in R using Rstudio

Open the file `Workshop1.R` from the shared folder (`bit.ly/IDPEctw`).

- This opens an R *script* in `Rstudio`: a complete sequence of instructions to perform a series of commands.
- As a good practice, scripts can be saved for future use, to ensure reproducibility of analysis.
- Commands can be 'run' (i.e., sent to the console) by selecting one or more lines of code from the script and pressing Ctrl + R.
- *Comments*—parts of code that should not be interpreted by R—are preceded by #.
- The console display can be 'cleaned' with Ctrl + L. This does not affect the contents of R's memory (the *workspace*).
- Previously run command can be found in the *history*.

A first session in R using Rstudio

At its most basic, R can be used as a calculator, using standard syntax:

```
> 3*2*(5^2+1)
[1] 156
> sqrt(2)  # square root function f(x) = √x.
[1] 1.414214
> exp(1)   # exponential function f(x) = e^x
[1] 2.718282
```

Notice that both *function* arguments and *expressions* are enclosed in `( )`. Other—more or less obvious—mathematical functions include:

- `log(x)` : $f(x) = \ln(x)$ `log(x, base=a)` : $f(x) = \log_a(x)$.
- `factorial(n)` : $f(n) = \begin{cases} n! & \text{if} \quad n \in \mathbb{N} \\ \Gamma(n+1) = \int_0^\infty t^n e^{-t}\,\mathrm{d}t & \text{if} \quad n \in \mathbb{R} \end{cases}$
- `abs(x)` : $|\,x\,|$

The assignment operator

In R, *variables* are assigned values using the assignment operator `<-` .
Values of variables are displayed by *invoking* (naming them in the console):

```
> x <- 3
> x
[1] 3
```

- Assignment is *silent*: no need to end lines with ‘ ; ’—although these can be used to separate different expressions in the same line of code.

  ```
  > a <- 1; b <- 2; a + b
  [1] 3
  ```

- *Declaration* and assignment occur simultaneously.
- R allows easy, warning-free *overwriting* of variables (use with care!)

```
> x <- 3
> x <- 5
```

```
> x
[1] 5
```

- There are similar assignment operators: `=` , `<<-` .
  `<-` is recommended over `=` because it’s less ambiguous.

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

Vectors

Vectors of elements are defined with `c( )` (as in 'concatenate') and separated by commas:

```
c(1,2,3)
[1] 1 2 3
```

Vectors of integer sequences from a to b can be created with the `:` operator:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

Notice:

- *vectors* are defined in a looser sense than in linear algebra: in R they are simply one-dimensional *arrays* of (possibly non-numerical) elements.
- For all practical purposes, numerical vectors in R created with `c( )` are *column vectors* in linear algebra, though they're displayed as row vectors for convenience.

Functions and vectors

R is an *array* language: it generalises to arrays operations defined for scalars. For instance, to apply $f(x) = \sqrt{x}$ to *every element* of a vector:

```
> sqrt(c(4,9,16))
[1] 2 3 4
```

Some functions are designed specifically to deal with vectors:

- `sum( )` sums every element.

```
> sum(c(4,9,16))
[1] 29
```

- `length( )` gives the *number of elements* in the vector (not the *norm*).

```
> length(c(4,9,16))
[1] 3
```

- `mean( )` averages the elements.

```
> mean(c(2,3,4))
[1] 3
```

Matrices

Matrices are created using the function

`matrix(data=x, nrow=m, ncol=n)`

where

- `x` is a vector of elements.
- `m` is the (integer) number of rows.
- `n` is the (integer) number of columns.

For instance:

```
> M <- matrix(data=c(1,2,3,4), nrow=2, ncol=2)
> M
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Notice:

- The names of function arguments can be omitted: `matrix(x,m,n)`.
- Matrices are filled column-by-column by default; this can be changed by adding the option `byrow=TRUE` : `matrix(x,m,n,byrow=TRUE)`.

Working with arrays

Vectors and matrices are examples of *arrays* (1- and 2-dimensional, respectively), whose elements are positioned according to *indices*.

We can apply the array *subsetting* operator `[ ]` to a vector `v` or a matrix `M`:

- `v[i]` returns the ith element of `v` ($i \in \{1, 2, \ldots, n\},\ v \in \mathbb{R}^n$).
- `v[c(i,j)]` returns the ith *and* j^{th} elements of `v` ($i, j \in \{1, 2, \ldots, n\}$).
- `v[c(i:j)]` returns the ith *to* j^{th} elements of `v` ($i, j \in \{1, 2, \ldots, n\}$).
- `v[-i]` returns `v`, *except* for the ith element.
- `M[i,j]` returns the element $M_{i,j}$ of `M`.
- `M[i, ]` returns the ith *row* of `M`.
- `M[ ,j]` returns the jth *column* of `M`.

Warning

The assignment operator `[ ]` should not be confused with `( )`, which enclose function arguments!

Working with arrays: array functions

We already know that R generalises functions defined for scalars to arrays—that is, when a vector or a matrix is the argument of some mathematical function, the function will be applied element-wise:

Consider for instance the following vector u and matrix M:

```
> u
[1] 2 7
```

```
> M
     [,1] [,2]
[1,]    5    6
[2,]    8    1
```

Applying an operations to the vectors (e.g, raising them to the third power) results in the operation being applied to *every element* of the vectors:

```
> u^3
[1]   8 343
```

```
> M^3
     [,1] [,2]
[1,]  125  216
[2,]  512    1
```

Working with arrays: element-wise operations

General rule

In R functions applied to arrays element-wise, unless otherwise specified.

For instance, multiplication by * is element-wise (and in vertical order!):

```
> u
[1] 2 7
> M
     [,1] [,2]
[1,]    5    6
[2,]    8    1
```

```
> u*M
     [,1] [,2]
[1,]   10   12
[2,]   56    7
```

```
> M*u
     [,1] [,2]
[1,]   10   12
[2,]   56    7
```

This feature can be useful in some instances, likely to lead to mistakes in others.

Working with arrays: vector and matrix multiplication

The matrix/vector multiplication used in linear algebra is given by `%*%`

```
> M %*% u
     [,1]
[1,]   52
[2,]   23
```

$$(Mu)_i = \sum_{k=1}^{n} M_{ik} u_k$$

R may also automatically (and silently) transpose vectors to make them conformable in a matrix multiplication. The transpose command is `t( )`

```
> u %*% M
     [,1] [,2]
[1,]   66   19
```

```
> t(u) %*% M
     [,1] [,2]
[1,]   66   19
```

We can use these commands to compute inner and outer products of vectors:

```
> u%*%u  # t(u)%*%u
     [,1]
[1,]   53
```

inner product
$u \cdot u = u'u$

```
> u %*% t(u)
> u %o% u
```

outer product
$u \otimes u = uu'$

`u%*%u` gives 1×1 matrix: `drop(u%*%u)` returns a scalar

Working with arrays: inversion of matrices

Non-singular square matrices A ($\det(A) \neq 0$) are inverted with `solve(A)`
We find A^{-1} such that $AA^{-1} = A^{-1}A = \mathbb{1}_n$.

```
> A <- matrix(c(4,3,3,2),2,2)
> det(A)
[1] -1
```

```
> solve(A)
     [,1] [,2]
[1,]   -2    3
[2,]    3   -4
```

```
> solve(A) %*% A
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

The function `solve(A,b)` also finds the vector x solving $Ax = b$.

```
> b <- c(2,7)
> solve(A,b)
[1]  17 -22
```

```
> A %*% solve(A,b)
     [,1]
[1,]    2
[2,]    7
```

Working with arrays: matrix operators

The function `diag( )` works in three ways:

1. Given a *matrix* `A`, `diag(A)` returns the *vector* of its diagonal elements

```
> diag(matrix(c(4,3,3,2),2,2))
[1] 4 2
```

2. Given a *vector* `u`, `diag(u)` returns a diagonal *matrix* with `u` as diagonal elements.

```
> diag(c(2,7))
     [,1] [,2]
[1,]    2    0
[2,]    0    7
```

3. Given a *scalar* `i`, `diag(i)` returns an *identity matrix* $\mathbb{1}_{i \times i}$.

```
> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Working with matrices: diagonalisation

Reminder

A matrix $A_{(n \times n)}$ is said to be *diagonalisable* if $A = P\Lambda P^{-1}$, where:

- Λ is an $(n \times n)$ diagonal matrix, such that $\Lambda_{i,i} = \lambda_i$, eigenvalue of A (i.e., solutions to $(A - \lambda I) = 0$).
- P is an $(n \times n)$ matrix, whose columns are the eigenvectors v_i associated to λ_i (i.e., solving $Av = \lambda v$).

Working with matrices: diagonalisation

The function `eigen(A)` returns a *vector of eigenvalues* and a *matrix of eigenvectors* of a given square matrix `A`.

```
> A <- matrix(c(1,1,0,3),2,2)
> eigen(A)
$values
[1] 3 1

$vectors
     [,1]       [,2]
[1,]    0  0.8944272
[2,]    1 -0.4472136
```

we can select either with `eigen(A)$values` or `eigen(A)$vectors`

```
> lambda  <- eigen(A)$values
> P       <- eigen(A)$vectors
> P %*% diag(lambda) %*% solve(P)
     [,1] [,2]
[1,]    1    0
[2,]    1    3
```

Working with matrices: additional functions

There are a number of functions to manipulate matrices:

- `sum(diag(A))` finds the trace of `A` $\left(\mathrm{tr}(A) = \sum_{i=1}^{n} a_{i,i} \equiv \sum_{i=1}^{n} \lambda_i\right)$

```
> sum(diag(A))
[1] 4
```

```
> sum(eigen(A)$values)
[1] 4
```

- `chol(S)` computes the Choleski factorization of a real symmetric positive-definite square matrix `S`.

The additional library `Matrix` (part of the `R-base` distribution, but not loaded by default; use `library("Matrix")`) has some additional matrix methods.

- `rankMatrix(A)` computes the rank of a square matrix `A` (beware: numerical method!)
- matrix methods to deal efficiently with sparse matrices (consisting mostly of zeros).
- functions to compute fancy matrix decompositions.

Floating-point arithmetic, or 'the Devil is in the decimals'

Because of the way computers represent and deal with real numbers (the so-called *binary representation* and *floating point arithmetic*) operations like

- matrix inversion—where many numbers interact additively
- eigendecomposition—involving finding the roots of a polynomial
- finding the rank of a matrix—also polynomial root-finding

may give 'unexpected' results:

$$B = \begin{bmatrix} 12 & 6 \\ 8 & 1 \end{bmatrix}; \quad B^{-1} = \begin{bmatrix} -1/36 & 1/6 \\ 2/9 & -1/3 \end{bmatrix}$$

$$BB^{-1} = B^{-1}B = \mathbb{1}_2$$

```
> B <- matrix(c(12,8,6,1),2,2)
> B %*% solve(B)
             [,1]         [,2]
[1,] 1.000000e+00 1.110223e-16
[2,] 2.775558e-17 1.000000e+00
```

Results can be rounded to (at most) `n` digits using the function `round(x,n)`:

```
> round(B%*%solve(B), 5)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

By the way: `1.110223e-16`

$= 1.110223 \times 10^{-16}$

$= 0.0000000000000001110223$

≈ 0

Application: the Ordinary Least Squares estimator

OLS in linear algebra

Given a linear system $y = X\beta$, where:

y is an $(n \times 1)$ vector;

X is is an $(n \times k)$ matrix;

β is a $(k \times 1)$ vector of (unknown) coefficients;

- if $n > k$, the system has no solutions (it's *overdetermined*).
- we want $\hat{\beta}$ for which $y = X\hat{\beta}$ holds 'as closely as possible'.
- by 'as closely as possible' we mean

$$\hat{\beta} = \arg\min_{\beta} \|y - X\beta\| = \arg\min_{\beta} (y - X\beta)'(y - X\beta)$$

- the solution can be shown to be $\hat{\beta}_{ols} = (X'X)^{-1} X'y$

Application: the Ordinary Least Squares estimator

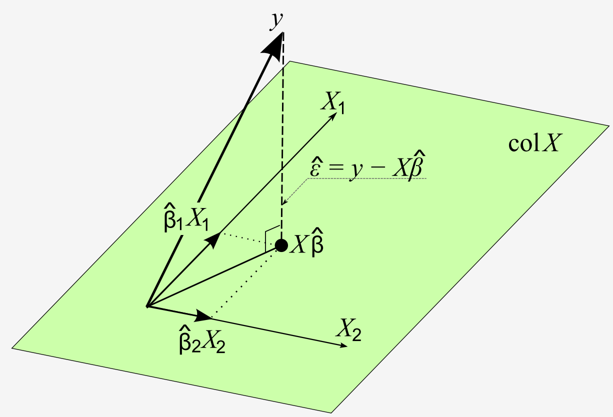


Figure : Interpretation of OLS as projection y on the space spanned by X_1, X_2. (From Wikipedia)

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

Defining functions

It's easy to write function—defined in a mathematical sense—in R.

For instance, consider the function

$$f : \mathbb{R} \to \mathbb{R}$$
$$x \mapsto 3x^2 + 2x$$

This can be created with

```
f <- function(x) 3*x^2+2*x
```

once we have defined a function, by supplying it with an argument we get the image. Naturally, also user-defined functions $\mathbb{R} \to \mathbb{R}$ work element-wise when they are fed a vector of arguments:

```
> f(3)
[1] 33
```

```
> f(c(1,2,3))
[1]  5 16 33
```

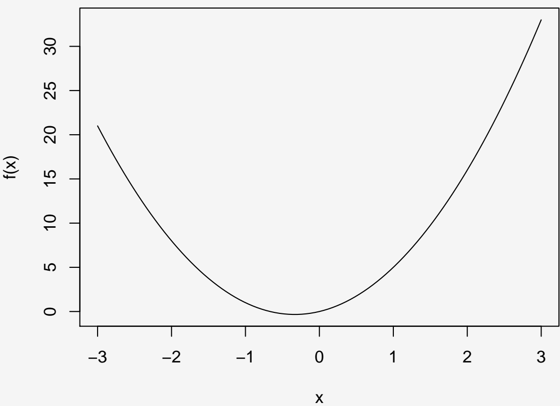
Plotting functions

The element-wise property allows to plot the function easily:

```
curve(f(x), from=-3, to=3)
```

(where `from` and `to` define the subset of the domain of $f(x)$ to be plotted.)

This gives



Defining functions of several variables, with parameters

We can generalise this to functions of more than one variable $\mathbb{R}^n \to \mathbb{R}$. For instance $z(x, y) = x^a y^b$ where $\{a, b\} \in \mathbb{R}_0^+$ are parameters.

Again, we define the function as follows.

```
z <- function(x,y) x^a*y^b
```

Notice that the function depends also on the *parameters* a and b. These have not yet been assigned in R, but we can define a function involving them anyway (this feature is called *lazy evaluation*).

```
> z(x=1,y=2) # a and b are not defined
Error in z(1, 2) : object 'a' not found
```

```
> a <- 0.3; b <- 0.6
> z(x=1,y=2)
[1] 1.515717
```

If we define a and b, even outside the scope of the function (so-called *assignment to the global environment*), we are able to find a value of the image $z(x, y)$.

Plotting functions $\mathbb{R}^2 \to \mathbb{R}$

In order to plot the function $z(x, y) = x^a y^b$, we first evaluate it on a grid of points.

```
x.val <- seq(1, 10, length=50)
y.val <- seq(1, 10, length=50)
z.val <- outer(x.val, y.val, z)
```

where

- `seq(1, 10, length=50)` creates a vector of 50 equally distant numbers $[x_1, x_2, \ldots, x_{50}] \in [1, 10]$
- `outer(x.val,y.val,z)` evaluates the function on the grid:

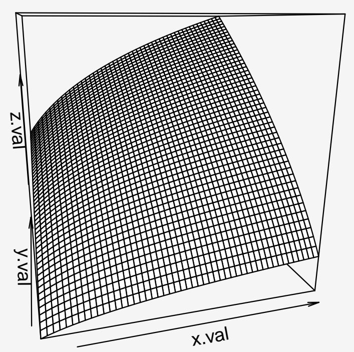
$$\begin{bmatrix} z(x_1, y_1) & \ldots & z(x_1, y_n) \\ \vdots & \ddots & \vdots \\ z(x_n, y_1) & \ldots & z(x_n, y_n) \end{bmatrix}$$

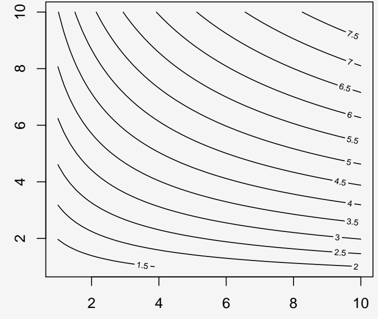
Plotting functions $\mathbb{R}^2 \to \mathbb{R}$

We use the grid defined by `x.val,y.val,z.val` to plot the function $z(x, y)$ over $(x, y) \in [1, 10] \times [1, 10]$.

```
persp(x.val,y.val, z.val)
```

```
contour(x.val,y.val, z.val)
```





Optimising functions $\mathbb{R} \to \mathbb{R}$

R has different numerical optimisation methods to find local minima or maxima: `optimize(f,interval)` finds the *minimum* of a generic function $\mathbb{R} \to \mathbb{R}$ `f` over a specified interval. To find the *maximum*, add the option `maximum=TRUE`.

```
> optimize(f=f(2), interval=c(-20,20))
$minimum
[1] -0.3333333

$objective
[1] -0.3333333
```

it returns the $\text{argmin}_{x\subset I} f(x)$ and $\min_{x\subset I} f(x)$.

Optimising functions $\mathbb{R}^n \to \mathbb{R}$

The equivalent optimiser for functions $\mathbb{R}^n \to \mathbb{R}$, `optim(par, fn)`, also minimises functions by default but works a bit differently:

- it requires the objective function to have just one formal argument, supplied in vector form.
- The first argument of `optim( )` is the vector of initial 'guesses' of function arguments initialise the optimisation.
- There is no `maximise=TRUE` option: just minimise over $-f(x)$ instead.

```
> g <- function(x) (x[1]-2)^2 + (x[2] -1)^2
> g(c(1,2)) # this is a two-variable function, entered as a single formal argument in vector form.
[1] 2
> optim(par=c(1,2),g)
$par
[1] 2.0000203 0.9999614

$value
[1] 1.901513e-09
# other output, omitted
```

An introduction to the apply family of functions

- As we have seen, many functions, including user-defined ones, are evaluated element-wise when fed array (vector and matrix) arguments.
- In some cases, however, it's not possible to give array arguments to functions—especially in the case of non-mathematical ones.
- The powerful and versatile `apply( )` function addresses this issue, by allowing to… *apply* a function to one or more 'margins' (i.e., elements, rows, columns, etc.) of an array.
- Consider for instance a case in which we would like to sum the squares *of each row* of a matrix:

```
> M
     [,1] [,2]
[1,]    5    6
[2,]    8    1
```

```
> apply(X=M, MARGIN=1, FUN=function(x) sum(x^2))
[1] 61 65
```

where `X=M` is the array, `MARGIN=1` means 'to every row' and `function(x) sum(x^2)` defines a function that squares every element of a vector and sums them, naturally, the result is a vector.

An introduction to the `apply` family of functions

- A similar reasoning can be used for summing *every column*: the option is now `MARGIN=2`.

```
> apply(M, 2, function(x) sum(x^2))
[1] 89 37
```

- If we want to apply the function to *every element*, the option is `MARGIN=c(1,2)`.

This is all very nice, but:

- Why not call the ‘margins’ `rows` and `columns` directly?
- Because `apply` also works for n-dimensional arrays!

Using apply on 3-dimensional arrays

Three-dimensional arrays are a generalisation of matrices, with a ‘depth’ dimension. Indexing with [ ] as for vectors and matrices.

```
> A <- array(c(1:8), dim=c(2,2,2))
> A
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

```
> A[1,,] # first row of both matrices
     [,1] [,2]
[1,]    1    5
[2,]    3    7
> A[,1,] # first column of both matrices
     [,1] [,2]
[1,]    1    5
[2,]    2    6
> A[,,1] # first matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> ssq <- function(x) sum(x^2)
> apply(A, c(1,2), ssq)
     [,1] [,2]
[1,]   26   58
[2,]   40   80
```

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

Computational Statistics with R

Some vector-oriented commands provide basic tools for statistical analysis:

- `sum( )`, `mean( )` and `median( )` sum, average and find the median of the elements in a numeric vector, respectively.
- `var( )` and `sd( )` compute the (unbiased) estimates of variance and standard deviations averages the elements of a numeric vector.
- `cov( )` and `cor( )` compute the covariance and correlation of two numeric vector of the same length, or that among the columns of a matrix.

Working with probability distributions

Functions to handle common probability distributions in various ways are easily implemented.

The syntax works by combining a prefix denoting the required feature of the distribution with the name of the distribution itself:

probability density	$f_X(x)$	`d`	×		`unif`	uniform
cumulative density	$F_X(q)$	`p`			`norm`	normal
quantile function	$F^{-1}(p)$	`q`			`t`	Student's t
(pseudo-)random draws	x i.i.d. $\sim X$	`r`			`binom`	binomial
					`chisq`	χ^2
					`logis`	logistic
					`gamma`	Γ
					…	…

Working with probability distributions

Obviously, both the features and the distribution themselves require to provide parameters. For instance:

- $\mathcal{U}(a, b)$ the uniform distribution is parametrised by it bounds `min` and `max`.
- $\mathcal{N}(\mu, \sigma)$ the normal distribution is parametrised by the *mean* `mu` and the *standard deviation* `sd`
- t_{df} the Student's t distribution is parametrised by the number of *degrees of freedom* `df`

For other distributions (there are a lot of them!), the parameters are indicated on the help page of the distribution; see also Paradis (2002).

Working with probability distribution: examples

Consider the normal distribution $\mathcal{N}(\mu = 1, \sigma = 2)$:

- the probability density at a generic point x is given by:

```
> dnorm(x=2, mean=2, sd=1)
[1] 0.3989423
```

- the cumulative density at a generic point q is given by:

```
> pnorm(q=3, mean=2, sd=1)
[1] 0.8413447
```

- the quantile corresponding to a generic probability p is given by:

```
> qnorm(p=0.2, mean=2, sd=1)
[1] 1.158379
```

- a sequence of n pseudo-random realisations is given by:

```
> rnorm(n=5, mean=2, sd=1)
[1]  2.145997  2.071448 -1.259927  2.246776  1.844732
```

Pseudo-random number generation

We can think of random number generation as producing a sequence of variables:

$$\{x_1, x_2, x_3, \ldots, x_n\} \quad x_i \in [0, 1] \; \forall i \in [1, n]$$

where any couple of distinct realisations x_i and x_j $i, j \in [1, n], \; i \neq j$ are 'sufficiently independent' from one another.

In fact, given the deterministic nature of computers, the sequence is actually the result of a deterministic process, completely determined by an initial *seed*:

- in normal applications, the seed is determined 'randomly' (e.g., from the computer clock). In this case, every generated sequence will appear different.
- however, we can specify the seed manually with `set.seed( )`, so as to be able to reproduce identical sequences, if necessary.
- The sequence taken as such, however, appears to contain elements sufficiently unrelated to each other as to appear random.

Generating pseudo-random numbers in R

Consider what happens when we generate a sequence of 2 numbers $\sim \mathcal{U}[0,1]$ with the command `runif(2)`:

Simply generating the sequences results in different numbers every time (and on different computers).

```
> runif(2)
[1] 0.4659625 0.2659726
> runif(2)
[1] 0.85782772 0.04583117
```

Setting an arbitrary *seed* (e.g., 123) and—generating the sequence immediately after—always yields the same results.

```
> set.seed(123)
> runif(2)
[1] 0.2875775 0.7883051
> set.seed(123)
> runif(2)
[1] 0.2875775 0.7883051
```

This is useful to guarantee exact reproducibility of simulation results when pseudo-random number are used (e.g., random sampling, bootstrap, Monte Carlo, etc.).

Random number generation and the inverse cdf

We can generate random numbers based on an arbitrary distribution F, provided that we know the inverse cumulative distribution function F^{-1} .

Proposition

Let U be a r.v. uniformly distributed on $[0, 1]$ and let $X = F^{-1}(U)$. Then the cumulative density function of X is F.

Proof.

$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x))$$

□

Adapted from J. A. Rice (1995) *Mathematical Statistics and Data Analysis.*

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

Data Analysis in R

We are going to consider these steps to perform data analysis in R:

1. Understand the structure of the dataset.
2. Computing summary statistics.
3. Subsetting: selecting variables and values.
4. Inspecting graphically the distribution and correlation of variables.
5. Performing statistical tests.

We are going to consider (more or less) 'tidy' datasets (i.e., properly formatted and encoded, no missing values, etc.) as well as the ways to deal with some of the more common complications, bearing in mind the following:

> *Happy families are all alike; every unhappy family is unhappy in its own way.*
>
> Leo Tolstoy

> *Like families, tidy datasets are all alike, but every messy dataset is messy in its own way.*
>
> Hadley Wickham

Dataset handling and descriptive statistics

Some of the main functions to handle datasets in R:

- `str( )` gives the *structure* of a dataset: the number, name and type of variables, the number of observations.
- `summary( )` gives *summary statistics* of a dataset: means, quantiles, frequencies, etc.
- `head( )` returns the first 5 observation of a dataset (useful if the dataset is large); `tail( )` returns the last five.
- `names( )` returns the *variable names* (i.e., column names) of the dataset; `row.names( )` does the same for row names. Both can also be used to assign and overwrite variable and row names into datasets.

Application: the Swiss dataset

R contains some example datasets, for illustrative purposes.

Entering

```
data(swiss)
```

loads a dataset on Swiss fertility and socioeconomic indicators, for 47 French-speaking provinces in 1888. Additional details on the data can be found with `help(swiss)`.

Let's use the commands described in the previous slide to describe and summarize the dataset.

Subsetting a `data.frame`

Datasets in R are stored in `data.frames` objects: these are similar to matrices where rows frequently denote different observations and columns denote variables (possibly of different types).

- The operator `$` allows to select all observations of a given variables in a dataset, as in `swiss$Fertility`.
- The subsetting operator `[ ]` also works as for matrix operations: as presented earlier for matrices, entries of a `data.frame` are identified by a *row index* (typically denoting a given observation) and a *column index* (typically denoting a given variable).
- The function `subset(x, subset, select)` allows an intuitive subsetting of a `data.frame` `x`, based on a logical expression, returning one or more variables. For instance:

```
subset(x=swiss, subset=Fertility > 80, select=Examination)
```

returns the value of the variable 'Education' for the observations whose values of 'Fertility' are *greater than 80* in the `swiss` dataset.

Subsetting a `data.frame`, continued

The combination of the operators `$` and `[ ]` allow to implements sophisticated and compact subsetting operations.

For instance, the following couple of commands each select the same observations between them.

- to select the values of the variable 'Examination' for those observations whose 'Fertility' is *greater than 80*:

```
> subset(swiss, Fertility > 80, Examination)
> swiss$Examination[swiss$Fertility > 80]
```

- to select the values of the variables 'Examination' *and* 'Catholic' for those observations whose 'Fertility' is *greater than 80*:

```
> subset(swiss, Fertility > 80, c(Examination,Catholic))
> swiss[swiss$Fertility > 80, c("Examination","Catholic")]
```

Subsetting and logical operations

At the heart of subsetting operations lay *logical values* associated to *array indices* (we encountered the latter when we discussed arrays).

- *logical values* are `TRUE` and `FALSE` (abbreviated as `T` and `F`, respectively).
- subsetting by indices returns array values whose indices are set to `TRUE`:

```
> v <- c(1,2,3)
```

```
> v[c(T,F,T)]
[1] 1 3
```

Logical values are also the result of *logical operations*, involving one or more of the following operators, using standard *Boolean algebra*, which evaluates the truth or falsehood of propositions (possibly involving arrays):

- `==` to test for *strict equality* and `!=` to test for *difference*.
- `>`, `>=`, `<`, `<=` to test for $>$, $\geq$, $<$, $\leq$, respectively.
- `%in%` to test for belonging ($\in$).
- propositions can make use of the logical operators: `&` (logical AND); `|` (logical OR); `xor( )` (logical XOR or *exclusive* OR); `!` (logical NOT).
- `all( )`, `any( )` test if a proposition hold for *all* or *any* array elements.

Subsetting and logical operations, continued

Therefore, an expression like

```
> swiss$Fertility > 80
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE #...
```

returns a *logical vector* (i.e., a vectors whose values are TRUE or FALSE).

When the expression is used *within* the subsetting operator [ ], it will return the values of the array corresponding to the TRUE entries of the expression:

```
> swiss$Examination[swiss$Fertility > 80]
 [1] 15  6  5 12 16 14 12 16 14  3
```

in this particular case, it returns the values of the variable 'Examination' corresponding to the units whose 'Fertility' is larger than 80.

Warning about strict equality ==

The logical operator ==, used to test for *strict equality*, should be used with caution with numerical values, because of the *double-precision representation* of real numbers in computers (not just in R).

Consider the following cases:

- testing equality in (operations involving) integers works as expected:

```
> 1+1==2
[1] TRUE
```

- testing equality between *two* non-integers also works as expected:

```
> 0.2==0.2
[1] TRUE
```

- testing an operation involving two non-integers, however, we may get an unexpected result:

```
> 0.2+0.1==0.3
[1] FALSE
```

Warning about strict equality ==, continued

The reason for the apparent puzzle has to do with the errors that may accumulate in addition between double-precision floating-point numbers.

We can display 22 decimal digits of `0.2+0.1` using the function `print( )`

```
> print(0.2+0.1, digits=22)
[1] 0.3000000000000000444089
```

Roughly speaking, this is because computers have limitations in terms of representing real numbers: they represent them in base 2 and have a finite memory to store decimal digits. Thus, storing non-integer numbers involves a small rounding error (initially too small to see), which becomes relevant when different such numbers are added up.

This also means that some equalities that *should not* hold return `TRUE`

$$1 + 1 \times 10^{-32} \neq 1$$

```
> 1+1e-32 == 1
[1] TRUE
```

Good to know. . . but don't worry too much: in many cases just use `round( )`

Basic statistical plots

We can do any number of statistical plots with basic `R` commands to support a preliminary statistical analysis. These include

- `plot(x,y)` produces a *scatterplot* of two vectors of numeric variables `x`, `y` of the same length; `plot` is also a generic plotting function that tries to infer the appropriate plot from context.
- `pairs( )` produces a *matrix of paired scatterplots* of a `data.frame`.
- `hist(x)` produces a density *histogram* of a numeric vector `x`.
- `plot(density(x))` produces a plot of the *kernel density estimate* of a numeric vector `x`, with adjustable bandwidth.
- `boxplot( )` creates one or more *box-plot* (or *box-and-whiskers plot*), depending on the argument being a numeric vector or `data.frame`.
- `qqplot( )`, `qqnorm( )` and `qqline( )` produce (empirical vs theoretical) *quantile-quantile plots.*

Later, we will introduce the library `ggplot2` to make more advanced (and nicer-looking!) plots.

Performing statistical tests

We can easily implement different statistical tests using R:

- `t.test( )` implements a t-test on one or two vectors of numeric values. By default the null hypothesis is `mu=0` and the confidence level is 95%.
- `ks.test(x, y)` implements Kolmogorov-Smirnov Tests (one- or two-sample) of a vector of numeric values `x` against another vector of numerical values `y` or against theoretical cumulative distribution function.

Loading data from external sources

We can import many data formats, stored on the computer or the internet. As a general (though not exhaustive) guideline, consider the following:

First locate the dataset on your system, using the `Rstudio` tab Files » ... and then Files » More » Set As Working Directory.
Alternatively, type the path directly into the script with `setwd("./path/")` (you may also copy/paste the path from your system file browser).

Then, depending how the dataset is encoded:

- text file: `.txt`,`.csv`,`.tab`, etc. : these can be imported with `Rstudio` from Workspace » Import Dataset or with basic R functions: `read.table()`, `read.csv()`, which allow fine control of the settings.
- Excel spreadsheet: `.xls .xlsx`: either save the content to a text file (e.g., `.csv`) and use the previous procedure, or use the `xlsx` library (32-bit R only) and its `read.xlsx( )` function.
- Binary data from STATA SAS, etc. can be imported using the library `foreign` and its functions (see the package documentation for details).

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with ggplot2

8 Econometrics

9 Procedural programming

ggplot2 and the 'Grammar of Graphics'

`ggplot2` is a plotting system based on the concept of 'Grammar of Graphics' developed by Leland Wilkinson (2005).

- It envisages a statistical graphic as a mapping from data to *aesthetic attributes* (colour, shape, size) of *geometric objects* (points, lines, bars).
- The plot may also contain *statistical transformations* of the data.
- The plot is drawn on a specific *coordinate system*.
- *Faceting* can generate the same plot for different subsets of the dataset.

The combination of these independent components makes up a graphic.

In addition, `ggplot2`:

- implements the grammar by *layers* (H. Wickham, 2009)
- incorporates concepts related to statistical graphs and data analysis developed by John W. Tukey (1977), Edward R. Tufte (1990, 1997, 2001)
- allows to account data perception, as documented by William S. Cleveland and Robert McGill.

Wide and Long data and the `reshape2` package

In order to plot data with `ggplot2`, we have to provide it in (or transform it into) a ‘long’ format, using the package `reshape2` (by also by H. Wickham).

‘Wide’ table

Person	Age	Weight
Bob	32	128
Alice	24	86
Steve	64	95

`melt` →

← `cast`

‘Long’ table

Person	variable	value
Bob	Age	32
Alice	Age	24
Steve	Age	64
Bob	Weight	128
Alice	Weight	86
Steve	Weight	95

Transforming ‘wide to long’ is achieved with the command `melt( )` and, if necessary, by specifying one (or more) `id.vars`: variable(s) in our original `data.frame`, which (taken together) uniquely identify observations.

Constructing a graphic with `ggplot2`

The syntax of `ggplot2` implement the 'Grammar of Graphics', by layers:

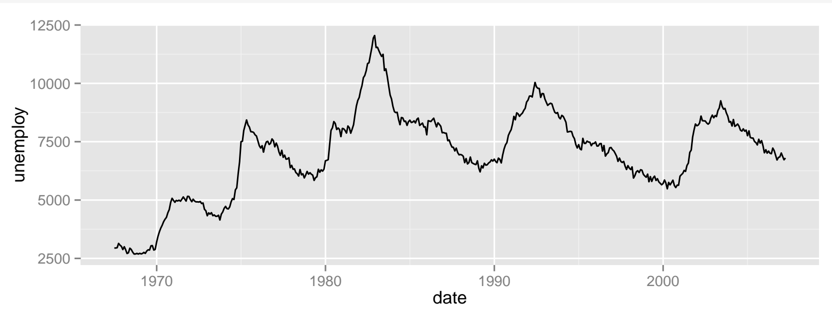
1. first, use the command `ggplot` to specify the long `data.frame` and the *aesthetic attributes* to be interpreted: variables, colours, shapes, etc.
2. specify which *geometric object* should interpret the aesthetic attributes (points, lines, surfaces, etc.).
3. If necessary:
   - modify axis type (scaling, inversion, etc.)
   - specify to *facet* the plot by some variable.
   - modify axes scales and labels, grids, title, etc.
4. evaluate and display the graphic.

This is an abstract specification of graphics, which may be confusing at first, but is *very* powerful and versatile.

Simple ggplot2 plots with qplot

In simple cases, this specification by layer is automatically taken care of by ggplot2 using the command qplot( ). Consider what happens when using the economics dataset provided as an example in ggplot2:

```
library("ggplot2")  # load the library (and embedded datasets with it)
head(economics)  # display the first few lines to see that the data.frame is already in 'long' format
qplot(x=date, y=unemploy, data=economics, geom="line")
```



Creating graphics by layers

We can obtain the same result with the full-fledged syntax:

1. The `economics` dataset is already in 'long' format for a single time series: every row uniquely identifies a time period, to which correspond observations of different variables.
2. We first select the relevant *aesthetic attributes* from the `economics` dataset: `date` (variable on the *x-axis*) and `unemploy` (on the *y-axis*).
3. The *geometric object* to be plotted is a line, to represent the time series.

```
ggplot(economics, aes(x=date, y=unemploy)) + geom_line()
```

This layered feature, which allows more control for details, is useful to define a plot over multiple lines, adding details and evaluating the plot at the end:

```
g <- ggplot(economics, aes(x=date, y=unemploy)) + geom_line()
g + labs(y="Unemployed, thousands", title="Evolution of population")
```

Faceted graphics

This layered specification allows easy *faceting*: representing different subsets of data side-by-side, which may have one or more axis in common.

1. To represent *more than one* variable on the vertical axis, we first reshape the `data.frame`, to group all the variables in a single column

```
library("reshape2") # load the reshape2 library to melt data.frames
data.melted <- melt(economics, id.vars="date")
head(data.melted); str(data.melted) # show the structure of the melted data.frame
```

2. building the plot: the aesthetic attributes are `date` and `value`

```
g <- ggplot(data.melted, aes(x=date,y=value))
```

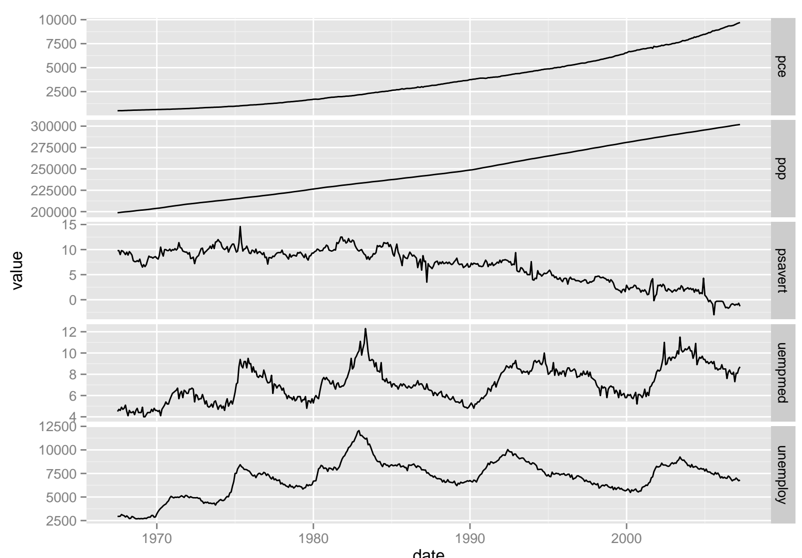
3. we *facet* the plot by `variable` so it has the time axis scale in common, and different vertical axis scales; `variable~.` stacks them vertically

```
g <- g  + facet_grid(variable ~ ., scales="free_y")
```

4. Finally, we specify lines as geometric feature and evaluate the plot.

```
g + geom_line()
```

Faceted graphics



Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

Econometrics in R: the OLS estimator

We are going to refer primarily to multiple linear regression examples (Ordinary Least Squares estimator or OLS; remember the formula?).

The multiple linear regression estimator is implemented by the command `lm(y ~ x1 + x2 + ...)` where `y` is the *dependent variable* and `x1`, `x2` are *independent variables*; the intercept is included by default.[1] The function returns a *fitted model* object, whose attributes can be manipulated as needed.

For instance, using the `swiss` dataset, we estimate the model

$$\text{Fertility}_i = \beta_0 + \beta_1\,\text{Agriculture}_i + \beta_2\,\text{Education}_i + \beta_3\,\text{Catholic}_i + \varepsilon_i$$

and save the estimate in a variable `my.model`

```
my.model <- lm(Fertility ~ Agriculture + Education + Catholic, data=swiss)
```

[1]The ~ symbol can be entered by copying it from the help page of the `lm` command.

Attributes of fitted models

Once we have estimated a linear model with `lm( )`, we can extract its the attributes and compute diagnostics:

- `my.model$coefficients` returns the vector of coefficient estimates of the model.
- `my.model$residuals` returns the vector of residuals of the model.
- `summary(my.model)` produces a table of regression results.
- `plot(my.model)` produces four diagnostic plot for the residuals.

Exporting regression results

We can export the regression results to LaTeX using the stargazer package (to be installed):

```
library("stargazer")
stargazer(my.model,single.row=T)
```

Compiled in LaTeX, it produces nice-looking tables such as the one on the opposite side.

	Dependent variable:
	Fertility
Agriculture	-0.203^{***}
	(0.071)
Education	-1.072^{***}
	(0.156)
Catholic	0.145^{***}
	(0.030)
Constant	86.225^{***}
	(4.735)
Observations	47
R^2	0.642
Adjusted R^2	0.617
Residual Std. Error	7.728 (df = 43)
F Statistic	25.732^{***} (df = 3; 43)
Note:	$^{*}p<0.1$; $^{**}p<0.05$; $^{***}p<0.01$

Other econometric estimators

Many more regression estimators can be used, sometimes relying on external packages:

- Binary-response (probit, logit) and count data (poisson) models are implement with the command `glm( )` (as in 'generalised linear models').
- Censored and truncated models (Tobit) are implemented with the package `censReg`.
- Linear panel data models are (pooled OLS, WG, FE, RE) are implemented with the package `plm`.
- Basic time series functionality is achieved with the command `ts`. Time series objects (especially irregularly spaced ones) are better handled with the `xts` package. The command `arima( )` estimates AR-MA and integrated models (and combinations thereof).

See also the guide by Farnsworth for applications and examples

Outline

1 About the workshop

2 Getting to know R

3 Array Manipulation and Linear Algebra

4 Functions, Plotting and Optimisation

5 Computational Statistics

6 Data Analysis

7 Graphics with `ggplot2`

8 Econometrics

9 Procedural programming

Procedural programming

R, besides being a statistical environment, is also a programming language, which allows to easily extend existing capabilities to perform complex operations. This is implemented with *Procedural programming*—intended as *imperative programming*—a programming paradigm consisting in sequences of commands for the computer to perform.[2]

[2] See http://en.wikipedia.org/wiki/Imperative_programming

Loops

A versatile feature in procedural programming is the ability to run *loops*: repetitions or iterations of a sequence of commands for a specified number of times or elements. These are implemented in R as:

- `for` loops, which *iterate* a sequence of commands, enclosed by `{ }` for each element of a vector. For instance, we may take every $i \in \{1, 2, \ldots, 10\}$ and square it:

  ```
  for (i in 1:10) { print(i^2) }
  ```

  the function `print( )` displays the result in every iteration.
- `while` loops, which perform a sequence of commands *while a certain condition is* `TRUE`. We can perform the same operations as above with:

  ```
  x <- 1
  while(x<=10){
    print(x^2)
    x <- x + 1
  }
  ```

for loops and vector indexing

In most practical applications, we can obtain the same type of behaviour both with a `for` and a `while` loop. We use the former, because it's generally less cumbersome.

We can easily to implement operations over *elements of arrays*: for instance, we may want to square every element of a vector $x = [1, 3, 5, 7, 9]$.

- We can run the loop directly on *every element* `e` of `x`:

```
x <- c(1,3,5,7,9)
for(e in x) { print( e^2 ) }
```

- Alternatively, we can treat the iteration counter `i` as an array index to get `x[i]` (the ith element of `x`) and squaring it:

```
for(i in 1:length(x)) { print( x[i]^2 ) }
```

  Notice that we iterate `i` over `1:length(x)` (i.e., $[1, 2, 3, 4]$).

Using `for` loops and indexing is more general, as the next example shows.

Application: for loops, indexing and the random walk

Consider the following example of a *random walk*, a simple stochastic process:

$$x_t = \begin{cases} x_1 = 0 \\ x_t = x_{t-1} + \varepsilon_t \quad t \in 2, 3, \ldots \quad \varepsilon_t \overset{i.i.d.}{\sim} \mathcal{N}(0,1) \end{cases}$$

We may try to simulate the behaviour of the process, for 100 time steps using a loop:

```
x <- rep(0,100)  # initialise the vector that will contain x_t for all time steps

for(t in 2:100){   # vector indices start at 1 and x_1 = 0.
  x[t] <- x[t-1] + rnorm(1)
}
```

Try `plot(x, type="l")` to get a *line plot* of this realised *time series*.

Sampling an ensemble of time series

We can sample from the ensemble of time series: for instance, by creating 50 different realisations of same process, given different realisations of ε_t.

First, we initialise a matrix with 50 columns (one for each series) and 100 rows (one for each time period). It's filled with `0`, which are going to be overwritten during the loop.

```
X <- matrix(0,nrow=100, ncol=50)
```

Then, we fill the matrix `X` *row by row*, using `rnorm(50)` to add a vector of independent realisations of ε_t.

```
for(t in 2:100){
  X[t,] <- X[t-1,] + rnorm(50)
}
```

We can plot the sample of the ensemble (every column representing a distinct series) with `matplot( )`

```
matplot(X, type="l", lty=1)
```

Flow control: `if` and `else`

`if` statement performs a given command when a certain condition is met.

In our example, we may want to display $i^2,\ i \in [1, 2, \ldots, 10]$ *only if* $i^2 > 8$:

```
for(i in 1:10){
  if (i^2 > 8) { print(i^2) }
}
```

The command `else` performs a given operation when the condition set by `if` is *not met*. In our (increasingly silly) example of squaring integers, we may wish to print i^2 only if $i^2 > 8$ and print another message otherwise:

```
for(i in 1:10){
  if (i^2 > 8) { cat("The square of", i, "is", i^2, "\n" ) }
  else { cat("The square of", i, "is not larger than 8","\n" ) }
}
```

the command `cat( )` allows to print together strings of text and variable values, separated by commas. The text `"\n"` forces a line break

Application: root-finding algorithm

Consider the following algorithm that, given $a \in \mathbb{R}$, finds $\sqrt{a}$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

From an initial 'guess' x_0, we recursively compute x_n, until the result no longer changes (i.e., we found a *fixed point* in the mapping). Then $x_n = \sqrt{a}$

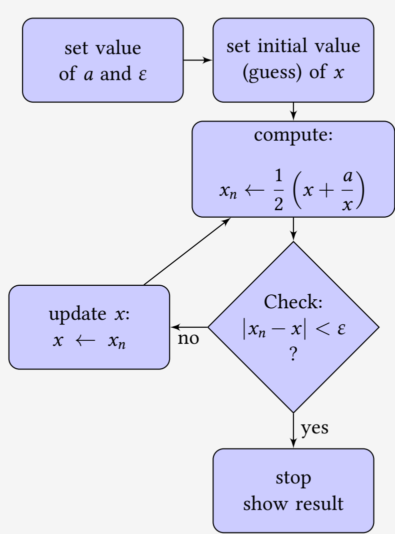
In practice, 'no longer changes' amounts to making $|x_{n+1} - x_n| < \varepsilon$, for a small ε of our choice, since requiring for strict equality is too restrictive, for computational reasons.

How can we implement this algorithm using loops?

The root-finding algorithm, flow chart

One way to implement the algorithm is to realise that, for given a and ε, the relevant variables are just the 'current' value of x and 'next' one, which we call x_n.

Then, the algorithm can be represented with the flow chart on the opposite side.



The root-finding algorithm using a `for` loop

We can now write the code in R. First, we initialise the values of a, x, ε:

```
a <- 5; x <- 10; eps <- 1e-5
```

Then, we implement the algorithm with a `for` loop: we allow for up to 100 iterations, but we interrupt the iterations with `break` when the result has been found. If the loop has been run for the maximum number of iterations (`if(i == 100)`), an error message is shown.

```
for(i in 1:100){
  x.next <- 0.5*(x+a/x)
  if( abs(x.next-x) < eps) {
    cat("the root of",a,"is",x.next," found in",i,"iterations")
    break
  }
  else( x <- x.next)
  if(i == 100) print("did not converge")
}
```

The root-finding algorithm using a while loop

We can implement the algorithm using a `while` loop:

```
a <- 5; x <- 10; x.next <- x + a; eps <- 1e-5
```

Notice that, unlike the `for` loop, here we need to provide an initial value also for x_n: this is because the `while` loop evaluates the logical expression `abs(x-x.next) > eps` *at the beginning of each iteration.* Thus, we need to provide some initial value of x_n for the first iteration: we assign $x_n \leftarrow x + a$ as a stand-in value, but the algorithm only requires $x_n \in \mathbb{R},\ |x_n - x| > \varepsilon$.

```
while(abs(x-x.next) > eps){
  x <- x.next
  x.next <- 0.5*(x+a/x)
  k <- k + 1
  if(k == 100) cat("Could not find the root in", k,"iterations")
}
```

We use $k \in \mathbb{N}$ as a *counter* to be incremented at every iteration, in order to stop the iterations if the algorithm does not converge in a 'reasonable' number of iterations.