Advanced Econometric Methods EMET3011/8014

Lecture 3

John Stachurski

Semester 1, 2011

Announcements

- Get yourself a fresh copy of the course notes PDF
- Comp lab solutions posted
- Assignment dates on the course homepage

Today's Lecture

More R

- Input and Output
- Conditions
- Repetition
- User-Defined Functions
- General Programming Tips

I/O

I have a file on my computer called `"test.txt"` containing text

```
10 9 8 7 6 5 4 3 2 1
```

I try to read data in `"test.txt"` into vector:

```
> x <- scan("test.txt")
```

Error message tells me the file cannot be found...

Reason: `"test.txt"` is not in the **current working directory**

```
> getwd()
[1] "/home/john"
```

Full path to the file:

`/home/john/emet_project/test.txt`

Solution:

```
> setwd("/home/john/emet_project/")
```

- Note for Windows users: Replace `\` with `/` in paths

The call to `scan()` now succeeds:

```
> x <- scan("test.txt")
Read 10 items
> x
 [1] 10  9  8  7  6  5  4  3  2  1
```

Can see contents of current working directory with `dir()` function:

```
> dir()
[1] "test.txt"
> pdf("foo.pdf")  # Will write to CW directory
> plot(x)
> dev.off()
null device
          1
> dir()
[1] "foo.pdf"  "test.txt"
```

Common task: Reading data sets into data frame

File: `"testdf.txt"`

```
X  Y
1 10
2 20
3 30
```

Can read into a data frame using `read.table()` function

```
> df <- read.table("testdf.txt", header=TRUE)
> df
  X  Y
1 1 10
2 2 20
3 3 30
> class(df)
[1] "data.frame"
```

- `header=TRUE` because first line contains column names

Options

- `read.table("somefile.txt", sep=";")`
- `read.table("somefile.txt", skip=8)`
- `read.csv("somefile.csv")`

R can also handle foreign data formats (Excel, STATA, SPSS, etc.)

Reading data directly from Internet:

```
> read.table("http://johnstachurski.net/emet/testdf.
    txt", header=TRUE)
```

Problems? Perhaps you are behind a firewall. Try

```
Sys.setenv(http_proxy="http://user:pass@proxy:8080")
```

- `user` is username
- `pass` is password
- `proxy` is proxy server

Still doesn't work? Save text file to hard disk and proceed as before

Other Options for Data Input

We mentioned `scan()` before:

```
> x <- scan("file.txt")  # Read row/col into vector
```

If no file name is given, then `scan()` reads from the screen:

```
> x <- scan()
1: 1 12 3 55 128    # Me typing numbers in
6:                  # And hitting return a second time
Read 5 items
> x
[1]   1  12   3  55 128
```

Information from users, returned as string:

```
> x <- readline("Enter the value of x: ")
```

Output

To write a whole data frame:

```
> write.table(summers, "summers.txt")
```

Low-level alternative is `cat()`

Can substitute for the `print()` function (with more control):

```
> x <- 3
> cat("The value of x is", x, "\n")  # \n for new line
The value of x is 3
>
```

Can also write this information to the hard disk:

```
> cat("The value of x is", x, "\n", file="test.txt")
```

Scripting

A **script**/**program** is a

- sequence of commands
- stored in a text file

Consider, for example, the two commands

```
> plot(x <- sort(rnorm(50)), type = "s")
> points(x, cex = .5, col = "dark red")
```

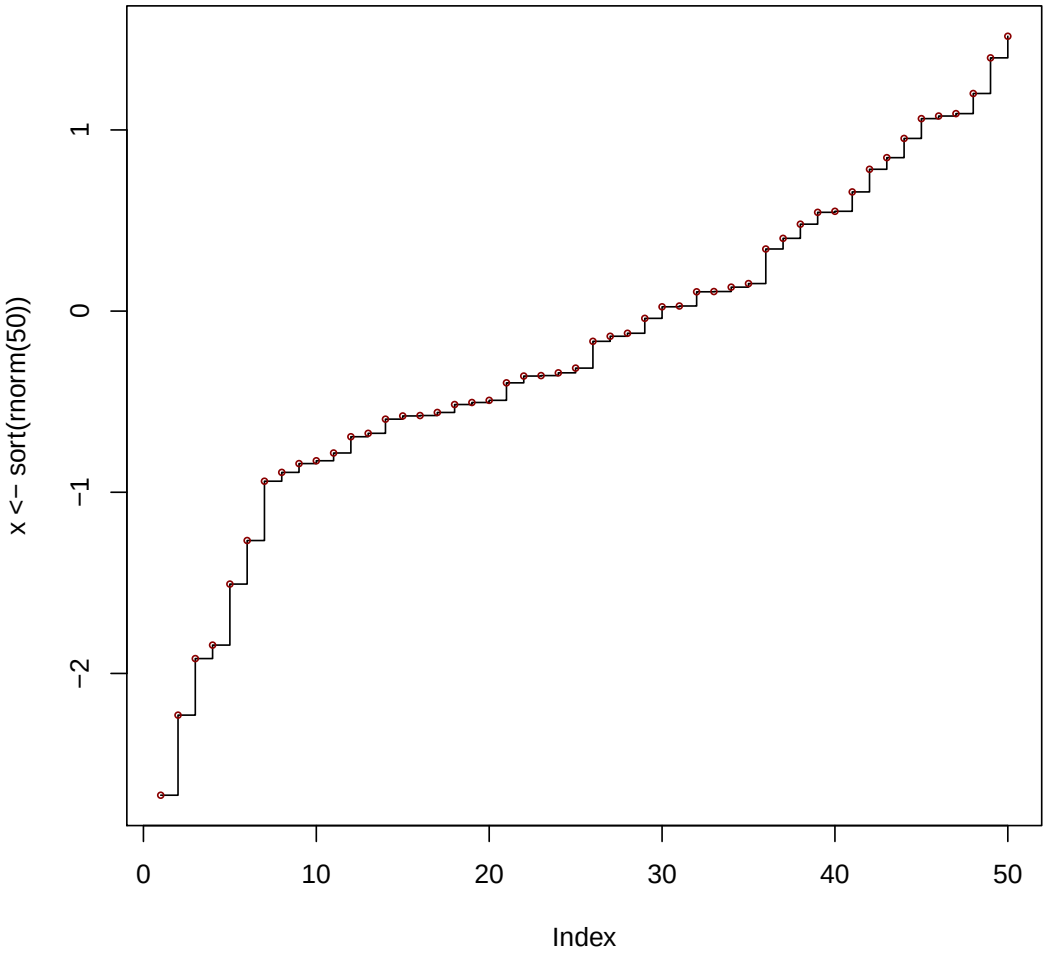

Figure: Step function

To reproduce next session, save commands in a text file

Other reasons to use scripts:

- Run long sequence of commands
- Make incremental improvements
- Isolate problems
- Share with colleagues, etc.

Remember: Scripts are saved in text files

Don't confuse text files with word processor documents!

R will not be able to read a program written in MS Word

- Unless you specifically save it as a text file

But please don't

- Word processors aren't designed for the job

Text files are manipulated with **text editors**

Basic text editor supplied by most environments:

- Notepad (Windows), TextEdit (Mac), Vi, etc (Linux)

Alternatively, if R has GUI, then "File" menu $\rightarrow$ create script

Serious text editors for coding: features such as

- syntax highlighting, automatic indentation, code completion

Google around to find out more

How to run a script through the R interpreter?

Simplest way:

1. open text file with a text editor
2. copy and paste at prompt

A better way: use `source("filename.R")`

- .R is the standard extension
- see discussion of working directory and paths

Scripting Summary

To run commands through R, two options:

- type at prompt, or
- into text file (script) and then run

Typically, a mix is best

Common pattern:

- Experiment at prompt (how do I plot again?)
- Type commands into text file (script)
- Run them through interpreter
- Interact with results at the prompt

Final comment: Generally, running commands via a script equivalent to typing commands at the prompt one by one

Exception: type `x` at prompt and hit return gives value of `x`

In a script run via `source()`, need explicit `cat`/`print` command to get information to screen

Conditions

We have already met the logical values `TRUE` and `FALSE`:

```
> x <- TRUE
> x
[1] TRUE
> x <- T
> x
[1] TRUE
> x <- F
> x
[1] FALSE
```

Some expressions in R evaluate to either `TRUE` or `FALSE`.

For example:

```
> 2 > 3
[1] FALSE
> y <- 3 >= 2
> y
[1] TRUE
```

Here “>” and “>=” called **relational operators**

Testing for equality:

```
> 2 == 3
[1] FALSE
```

Note double equal sign!

```
> x == 2  # Testing equality
> x = 2  # Assignment, equivalent to x <- 2
```

Inequality:

```
> 2 != 3
[1] TRUE
> !(2 == 3)  # Equivalent to 2 != 3
[1] TRUE
```

Aside: Inf

`Inf` behaves much as the symbol ∞ does in comparisons

```
> 10^100 > Inf
[1] FALSE
> -Inf < 0
[1] TRUE
```

and arithmetic operations:

```
> Inf + Inf
[1] Inf
> Inf - Inf    #  Result is NaN (Not a Number)
[1] NaN
> 1 + Inf
[1] Inf
> 10 / Inf
[1] 0
```

Relational operators can be applied to vectors

Produce element by element comparisons

For example:

```
> x <- c(1, 2)
> y <- c(0, 10)
> x > y
[1]  TRUE FALSE
```

Here

- `x[1]` is compared against `y[1]`
- `x[2]` is compared against `y[2]`

Often, want to compare a whole vector against a single value

Example: Which elements of `x` are greater than 1?

```
> x <- c(1, 4, 5, 9, 0)
> x > rep(1, length(x))     # The long way
[1] FALSE  TRUE  TRUE  TRUE FALSE
> x > 1                     # The easy way
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

Combining logical expressions with **Boolean operators** AND, OR

Consider two statements P and Q

- E.g., P = the statement "x is greater than 3,"
- E.g., Q = the statement "I come from Mars"

Given statements P and Q, can also consider statements

- P AND Q — true if both true, false otherwise
- P OR Q — false if both false, true otherwise

In R, AND and OR represented by "&" and "|" respectively

Examples:

```
> 1 < 2 & 2 < 3  # AND: Both true, so true
[1] TRUE
> 1 < 2 | 2 < 1  # OR: One true, so true
[1] TRUE
> 1 < 2 & 2 < 1  # AND: One false, so false
[1] FALSE
```

Note: `a < x & x < b`, not `a < x < b`

Boolean operators can be applied to logical vectors

As usual, the action is element by element:

```
> x <- c(1, 4, 5, 9, 0)
> 5 <= x
[1] FALSE FALSE  TRUE  TRUE FALSE
> x <= 7
[1]  TRUE  TRUE  TRUE FALSE  TRUE
> 5 <= x & x <= 7
[1] FALSE FALSE  TRUE FALSE FALSE
```

Note: `5 <= x <= 7` does not work

Boolean Arithmetic

Logical values can be used in algebraic expressions

`TRUE` evaluates to one and `FALSE` evaluates to zero:

```
> FALSE + TRUE
[1] 1
> FALSE * TRUE
[1] 0
> sum(c(TRUE, TRUE, FALSE))
[1] 2
```

Boolean arithmetic is handy!

For example, to find out how many elements of `y` exceed 3:

```
> sum(y > 3)
```

To find out the fraction of elements of `y` that exceed 3:

```
> mean(y > 3)
```

Can you see how this works?

Conditional Extraction

Vectors can be indexed by logical vectors:

```
> y <- seq(2, 4, length=5)
> y
[1] 2.0 2.5 3.0 3.5 4.0
> y[c(TRUE, FALSE, FALSE, FALSE, TRUE)]
[1] 2 4
```

Hence, if `y` is any vector, then

```
> y[y > 3]
```

returns all elements of `y` that exceed 3

Example: data frame `wages`, first few lines of are as follows:

```
    sex salary
1    F  11.21
2    F  10.79
3    M   8.92
4    F  10.42
5    M   9.75
6    F   9.90
```

To compute average wage for females:

```
> mean(wages$salary[wages$sex=="F"])
[1] 10.03096
```

If-Then-Else

Example:

```
password <- readline("Enter your password: ")
if (password == "foobar") {
    print("Welcome")
    # Do something
} else {
    print("Access denied.")
    # Do something else
}
```

This kind of stuff won't come up much for us

Can often replaced by convenient function `ifelse()`

Example of `ifelse()`

```
> ifelse(1 > -1, "foo", "bar")    # Returns "foo"
[1] "foo"
> ifelse(-1 > 1, "foo", "bar")    # Returns "bar"
[1] "bar"
```

First statement inside the brackets is evaluated

If true, the second value is returned

If false, the third value is returned

Note: First statement can also be vector comparison

Example: Have data on years of schooling, including university:

```
> ys
[1] 10 12 15 12 16 17 11
```

Want to create a dummy variable in new vector `tertiary`

- `tertiary` $= 1/0$ if more/less than 12 years of schooling

Can be accomplished as follows:

```
> tertiary <- ifelse(ys > 12, 1, 0)
> tertiary
[1] 0 0 1 0 1 1 0
```

For Loops

The most common kind of loop

Example: sum the integers from 1 to 100

```
> x <- 0
> for (i in 1:100) {x <- x + i}
> x
[1] 5050
```

First `x` is set to zero

Next, `i` is stepped through each element of `1:100`

Calculation `x <- x + i` is performed at each step

We could replace the line

```
> for (i in 1:100) {x <- x + i}
```

with the lines

```
> i <- 1
> x <- x + i
> i <- 2
> x <- x + i
. . .           # Many lines omitted
> i <- 100
> x <- x + i
```

Equivalent, but clearly the former is preferred

Last loop is just for illustration

A simpler way of performing the same calculation:

```
> sum(1:100)
[1] 5050
```

In R, the latter is more efficient than the `for` loop

We'll talk about why that's the case later on

Another example of a `for` loop

Simulate rv counting heads in 1,000 flips of fair coin

```
num.heads <- 0
for (i in 1:1000) {
    b <- runif(1)
    if (b < 0.5) num.heads <- num.heads + 1
}
print(num.heads)  # Explicit print within scripts
```

Commands inside curly brackets are performed at each step

The coin flip is simulated by drawing $b \sim U[0,1]$.

If $b < 1/2$, the outcome is regarded as "heads"

Again, there's are easier way to do this in R

For example,

```
> sum(runif(1000) < 0.5)
```

will also do the job

So will

```
> rbinom(1, size=1000, prob=0.5)
```

Why?

For loops often used to step through indices of a vector

Example: Vector `fs` has data on firm sizes, e.g.,

```
fs <- c(1023, 87, 12, 2, 145, 324)  # Num of employees
```

Want to create vector `ftype`, replacing numbers with categories

- small: less that 50 employees
- medium: btw 50 and 1,000 employees
- large: more than 1,000 employees

```
ftype <- c("L", "M", "S", "S", "M", "M")
```

```
ftype <- character(0)  # Empty character vector

for (i in 1:length(fs)) {
    if (fs[i] < 50) ftype[i] <- "S"
    if (fs[i] >= 50 & fs[i] <= 1000) ftype[i] <- "M"
    if (fs[i] > 1000) ftype[i] <- "L"
}
```

Sometimes, vectorized functions can be used to avoid loops

For longer programs, explicit loops pretty much essential

Example task:

- Regress inflation on all subsets of 10 different regressors
- See which has the highest adjusted R squared

There are over 1,000 different possible subsets

A `for` loop would be a natural choice to step through subsets

While Loops

Simulate rv counting number of coin flips until first head

```
flip.num <- 0
coin.face <- 0

while (coin.face == 0) {
    flip.num <- flip.num + 1
    b <- runif(1)
    if (b < 0.5) coin.face <- 1   # heads occured
}

print(flip.num)
```

Loop continues until `coin.face == 0` evaluates to `FALSE`

Why is a `for` loop not appropriate?

Let's repeat simulation 1000 times to approximate expectation

```
num.repetitions <- 1000
outcomes <- rep(0, num.repetitions)  # Empty vector

for (i in 1:num.repetitions) {
    flip.num <- 0
    coin.face <- 0
    while (coin.face == 0) {
        flip.num <- flip.num + 1
        b <- runif(1)
        if (b < 0.5) coin.face <- 1
    }
    outcomes[i] <- flip.num
}

print(mean(outcomes))
```

Functions

R provides numerous **functions** as part of the R environment

For example, `sum()` is a function

Takes a vector as its **argument** and **returns** sum of elements

```
> y <- sum(x)  # x is argument, y gets return value
```

A more complicated function is `plot()`:

```
> plot(a, b, col="blue")  # a, b are vectors
```

The first two arguments are **positional arguments**

Meaning is determined from their order in the function call

If `a` and `b` differ, then so will output of

```
> plot(b, a, col="blue")  # Order of a, b reversed
```

Argument `"blue"` is a **named argument**

- Useful when many arguments
- Optional—holds sensible default value

Example 3. `integrate()` performs numerical integration:

```
> integrate(cos, -1, 5)
```

Computes the integral $\int_{-1}^{5} \cos(x)dx$ numerically

On my computer, it returns the output

```
-0.1174533 with absolute error < 4.3e-14
```

Observe that the first argument is a function

User-Defined Functions

Functions `sum()`, `plot()` and `integrate()` are **built-in functions**

Part of the environment supplied when R starts

Often convenient to define our own **user-defined functions**

When writing longer programs, user-defined functions almost indispensable

Good programmers create their own functions!

Example: Recall the built-in function `integrate()`

```
> integrate(cos, -1, 5)
```

How to compute $\int_{-1}^{5} x^2 \cos(x) dx$?

Solution:

- create **user-defined function** representing $y = x^2 \cos(x)$
- then pass to `integrate()`

```
> f <- function(x) return(x * x * cos(x))
```

- `f` is just a name, chosen arbitrarily
- `return()` determines return value

Let's test and then integrate:

```
> f(3) ==  3 * 3 * cos(3)
[1] TRUE
> integrate(f, -1, 5)
-18.97950 with absolute error < 2.5e-13
```

A function with no arguments:

```
> f <- function() print("foo")
> f()
[1] "foo"
```

A function with two arguments:

```
> g <- function(x, y) return(x^2 + 2 * x * y + y^2)
> g(2, 3)
[1] 25
```

A function with named arguments:

```
> h <- function(x, intrcpt=0) return(intrcpt + 2 * x)
> h(1)
[1] 2
> h(1, intrcpt=3)
[1] 5
```

Rules for Functions

- Any number of arguments (positional or named)
- Arguments can be any R object
- Return value can be any R object
- Functions return only one object

To send back multiple pieces of data, bundle into vector or list

Example of a longer function, in a script:

```
f <- function(q) {    #  q = probability of heads
    flip.num <- 0
    coin.face <- 0
    while (coin.face == 0) {
        flip.num <- flip.num + 1
        b <- runif(1)
        if (b < q) coin.face <- 1  # with prob q
    }
    return(flip.num)
}
```

Curly brackets delimit function body

Use of indentation helps for human readers

Local Variables

Variables defined inside a function are **local variables**.

Consider following code:

```
> x <- 1
> f <- function() {x <- 2; print(x)}
> f()
[1] 2
> x
[1] 1
```

Why is `x` still bound to 1?

(See course notes for full explanation)

Local/global variable distinction common to almost all languages

Reason is data encapsulation

Example: Complex functions declare lots of variables

Don't want those variable names to conflict with the variable names that you are using in your workspace

Why Functions?

Functions break programs down into smaller logical components

Each component can be designed individually

Fits with natural problem solving design process

Because used to solve specific tasks, encourage code reuse

Saves duplication of effort for common tasks

Now make one implementation as good as possible

Efficiency

Sooner or later you will bump up against limits of what your computer can do

Computers may be getting faster, but

- data sets are getting larger
- econometric problems getting more complex

Buying new hardware often makes little difference

Need to look at the efficiency of your code

Programs converted into **machine code** before execution by CPU

In a compiled languages, done in one pass prior to execution

In an interpreted language, individual commands converted to machine code on the fly

Once off compilation prior to execution is efficient:

- Compiler can see program as a whole, optimize
- Less overhead in machine code translation

Hence C, Fortran 100x faster than R in certain operations

Why don't we all just program in C then?

Because programming statistics in C is painful

R optimized for humans rather than computers

Human time is far more valuable than computer time

In fact, R is written mainly in C

The Good:

- A friendly interface to C code for stats programming

The Bad:

- Less low-level control relative to hand-coding C

The Ugly:

- No ugly

What can we learn about R from this discussion?

Pack big batches of operations into individual commands

- Achieved mainly through vectorized operations
- Allows R to pass the whole operation out to optimized machine code, pre-compiled from purpose-built C routines

Example: Naive function to square all elements in vector:

```
f <- function(x) {
    y <- numeric(length(x))
    for (i in 1:length(x)) { y[i] <- x[i]^2 }
    return(y)
}
```

100x slower than vectorized operation:

```
> x <- rnorm(10^6)
> system.time(x^2)
   user  system elapsed
  0.024   0.004   0.046
> system.time(f(x))
   user  system elapsed
  3.412   0.012   3.423
```

Clarity

In fact, very little of your code needs to be optimized

Often, 99% of CPU time used by tiny subset of your program

Only this code needs to be optimized

And only if run-time justifies extra work

- Human time is expensive, computer time is cheap

For the vast majority of your code, clarity is the priority.

To aid clarity:

- Add comments
- Use indentation
- Use functions to break the problem up

Bug Hunts

Errors come in two main classes:

- **syntax errors**: flaws in syntax causing R to issue error message
- **semantic errors**: logical mistakes causing program to operate incorrectly

Latter can be hard to track down (no error message)

Debugging 101: Track your variables during execution to locate source of error

```
cat("x =", x, "and y =", y, "\n")
```