

REVIEW OF R

Leopoldo Catania

Aarhus University and CREATES

`leopoldo.catania@econ.au.dk`

Outline

This set of slides includes a review of R. It is divided in three parts:

- Introduction to R.
- Getting Started with R.
- Basic programming.

INTRODUCTION TO R

Leopoldo Catania

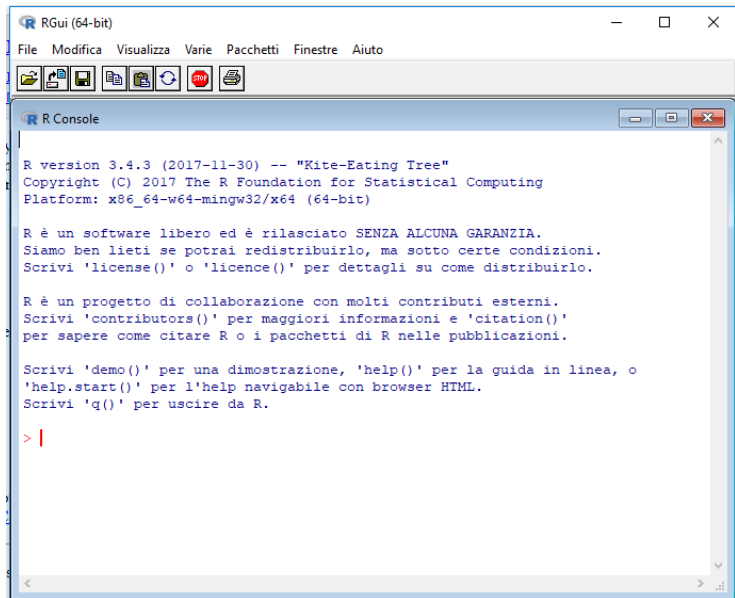
Aarhus University and CREATES

`leopoldo.catania@econ.au.dk`

The R programming language

- R is a free software environment for statistical computing and graphics.
- It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS.
- R is maintained by its community! You can contribute to it's development.
- R has a modular structure: external libraries are freely (also) available from the R repository CRAN.
- Download: <https://cloud.r-project.org/>

Plain R



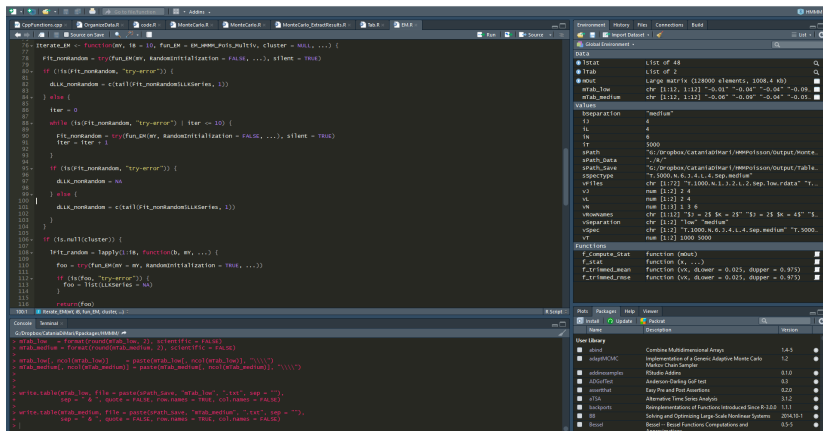
The screenshot shows the RGui (64-bit) application window. The title bar reads "RGui (64-bit)". The menu bar includes "File", "Modifica", "Visualizza", "Varie", "Pacchetti", "Finestre", and "Aiuto". The toolbar contains icons for file operations (open, save, print, etc.). The R Console window is open, displaying the following text:

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"  
Copyright (C) 2017 The R Foundation for Statistical Computing  
Platform: x86_64-w64-mingw32/x64 (64-bit)  
  
R è un software libero ed è rilasciato SENZA ALCUNA GARANZIA.  
Siamo ben lieti se potrai redistribuirlo, ma sotto certe condizioni.  
Scrivi 'license()' o 'licence()' per dettagli su come distribuirlo.  
  
R è un progetto di collaborazione con molti contributi esterni.  
Scrivi 'contributors()' per maggiori informazioni e 'citation()' per sapere come citare R o i pacchetti di R nelle pubblicazioni.  
  
Scrivi 'demo()' per una dimostrazione, 'help()' per la guida in linea, o 'help.start()' per l'help navigabile con browser HTML.  
Scrivi 'q()' per uscire da R.  
  
> |
```

RStudio

- The plain R interface is quite minimalist...
- We will use the RStudio graphical interface. This is freely available from:
<https://www.rstudio.com/products/rstudio/download/>
- First install R, then install RStudio!

RStudio



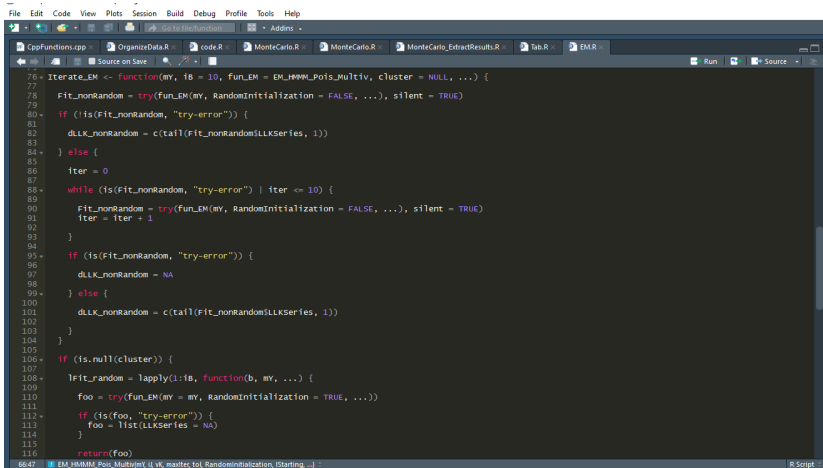
Much nicer...

RStudio panels

The screenshot displays the RStudio interface with four main panels:

- Source Panel (Top Left):** Contains R code for a function `iterate_M`. The code includes a `while` loop for random initialization and a `try-error` block for handling errors. The function returns a list of results. **1** is marked next to the `try-error` block.
- Environment Panel (Top Right):** Shows the global environment with variables like `data`, `bseparation`, `v1`, `v2`, `v3`, `v4`, `v5`, `v6`, `v7`, `v8`, `v9`, `v10`, `v11`, `v12`, `v13`, `v14`, `v15`, `v16`, `v17`, `v18`, `v19`, `v20`, `v21`, `v22`, `v23`, `v24`, `v25`, `v26`, `v27`, `v28`, `v29`, `v30`, `v31`, `v32`, `v33`, `v34`, `v35`, `v36`, `v37`, `v38`, `v39`, `v40`, `v41`, `v42`, `v43`, `v44`, `v45`, `v46`, `v47`, `v48`, `v49`, `v50`, `v51`, `v52`, `v53`, `v54`, `v55`, `v56`, `v57`, `v58`, `v59`, `v60`, `v61`, `v62`, `v63`, `v64`, `v65`, `v66`, `v67`, `v68`, `v69`, `v70`, `v71`, `v72`, `v73`, `v74`, `v75`, `v76`, `v77`, `v78`, `v79`, `v80`, `v81`, `v82`, `v83`, `v84`, `v85`, `v86`, `v87`, `v88`, `v89`, `v90`, `v91`, `v92`, `v93`, `v94`, `v95`, `v96`, `v97`, `v98`, `v99`, `v100`, `v101`, `v102`, `v103`, `v104`, `v105`, `v106`, `v107`, `v108`, `v109`, `v110`, `v111`, `v112`, `v113`, `v114`, `v115`, `v116`, `v117`, `v118`, `v119`, `v120`, `v121`, `v122`, `v123`, `v124`, `v125`, `v126`, `v127`, `v128`, `v129`, `v130`, `v131`, `v132`, `v133`, `v134`, `v135`, `v136`, `v137`, `v138`, `v139`, `v140`, `v141`, `v142`, `v143`, `v144`, `v145`, `v146`, `v147`, `v148`, `v149`, `v150`, `v151`, `v152`, `v153`, `v154`, `v155`, `v156`, `v157`, `v158`, `v159`, `v160`, `v161`, `v162`, `v163`, `v164`, `v165`, `v166`, `v167`, `v168`, `v169`, `v170`, `v171`, `v172`, `v173`, `v174`, `v175`, `v176`, `v177`, `v178`, `v179`, `v180`, `v181`, `v182`, `v183`, `v184`, `v185`, `v186`, `v187`, `v188`, `v189`, `v190`, `v191`, `v192`, `v193`, `v194`, `v195`, `v196`, `v197`, `v198`, `v199`, `v200`, `v201`, `v202`, `v203`, `v204`, `v205`, `v206`, `v207`, `v208`, `v209`, `v210`, `v211`, `v212`, `v213`, `v214`, `v215`, `v216`, `v217`, `v218`, `v219`, `v220`, `v221`, `v222`, `v223`, `v224`, `v225`, `v226`, `v227`, `v228`, `v229`, `v230`, `v231`, `v232`, `v233`, `v234`, `v235`, `v236`, `v237`, `v238`, `v239`, `v240`, `v241`, `v242`, `v243`, `v244`, `v245`, `v246`, `v247`, `v248`, `v249`, `v250`, `v251`, `v252`, `v253`, `v254`, `v255`, `v256`, `v257`, `v258`, `v259`, `v260`, `v261`, `v262`, `v263`, `v264`, `v265`, `v266`, `v267`, `v268`, `v269`, `v270`, `v271`, `v272`, `v273`, `v274`, `v275`, `v276`, `v277`, `v278`, `v279`, `v280`, `v281`, `v282`, `v283`, `v284`, `v285`, `v286`, `v287`, `v288`, `v289`, `v290`, `v291`, `v292`, `v293`, `v294`, `v295`, `v296`, `v297`, `v298`, `v299`, `v300`, `v301`, `v302`, `v303`, `v304`, `v305`, `v306`, `v307`, `v308`, `v309`, `v310`, `v311`, `v312`, `v313`, `v314`, `v315`, `v316`, `v317`, `v318`, `v319`, `v320`, `v321`, `v322`, `v323`, `v324`, `v325`, `v326`, `v327`, `v328`, `v329`, `v330`, `v331`, `v332`, `v333`, `v334`, `v335`, `v336`, `v337`, `v338`, `v339`, `v340`, `v341`, `v342`, `v343`, `v344`, `v345`, `v346`, `v347`, `v348`, `v349`, `v350`, `v351`, `v352`, `v353`, `v354`, `v355`, `v356`, `v357`, `v358`, `v359`, `v360`, `v361`, `v362`, `v363`, `v364`, `v365`, `v366`, `v367`, `v368`, `v369`, `v370`, `v371`, `v372`, `v373`, `v374`, `v375`, `v376`, `v377`, `v378`, `v379`, `v380`, `v381`, `v382`, `v383`, `v384`, `v385`, `v386`, `v387`, `v388`, `v389`, `v390`, `v391`, `v392`, `v393`, `v394`, `v395`, `v396`, `v397`, `v398`, `v399`, `v400`, `v401`, `v402`, `v403`, `v404`, `v405`, `v406`, `v407`, `v408`, `v409`, `v410`, `v411`, `v412`, `v413`, `v414`, `v415`, `v416`, `v417`, `v418`, `v419`, `v420`, `v421`, `v422`, `v423`, `v424`, `v425`, `v426`, `v427`, `v428`, `v429`, `v430`, `v431`, `v432`, `v433`, `v434`, `v435`, `v436`, `v437`, `v438`, `v439`, `v440`, `v441`, `v442`, `v443`, `v444`, `v445`, `v446`, `v447`, `v448`, `v449`, `v450`, `v451`, `v452`, `v453`, `v454`, `v455`, `v456`, `v457`, `v458`, `v459`, `v460`, `v461`, `v462`, `v463`, `v464`, `v465`, `v466`, `v467`, `v468`, `v469`, `v470`, `v471`, `v472`, `v473`, `v474`, `v475`, `v476`, `v477`, `v478`, `v479`, `v480`, `v481`, `v482`, `v483`, `v484`, `v485`, `v486`, `v487`, `v488`, `v489`, `v490`, `v491`, `v492`, `v493`, `v494`, `v495`, `v496`, `v497`, `v498`, `v499`, `v500`, `v501`, `v502`, `v503`, `v504`, `v505`, `v506`, `v507`, `v508`, `v509`, `v510`, `v511`, `v512`, `v513`, `v514`, `v515`, `v516`, `v517`, `v518`, `v519`, `v520`, `v521`, `v522`, `v523`, `v524`, `v525`, `v526`, `v527`, `v528`, `v529`, `v530`, `v531`, `v532`, `v533`, `v534`, `v535`, `v536`, `v537`, `v538`, `v539`, `v540`, `v541`, `v542`, `v543`, `v544`, `v545`, `v546`, `v547`, `v548`, `v549`, `v550`, `v551`, `v552`, `v553`, `v554`, `v555`, `v556`, `v557`, `v558`, `v559`, `v560`, `v561`, `v562`, `v563`, `v564`, `v565`, `v566`, `v567`, `v568`, `v569`, `v570`, `v571`, `v572`, `v573`, `v574`, `v575`, `v576`, `v577`, `v578`, `v579`, `v580`, `v581`, `v582`, `v583`, `v584`, `v585`, `v586`, `v587`, `v588`, `v589`, `v590`, `v591`, `v592`, `v593`, `v594`, `v595`, `v596`, `v597`, `v598`, `v599`, `v600`, `v601`, `v602`, `v603`, `v604`, `v605`, `v606`, `v607`, `v608`, `v609`, `v610`, `v611`, `v612`, `v613`, `v614`, `v615`, `v616`, `v617`, `v618`, `v619`, `v620`, `v621`, `v622`, `v623`, `v624`, `v625`, `v626`, `v627`, `v628`, `v629`, `v630`, `v631`, `v632`, `v633`, `v634`, `v635`, `v636`, `v637`, `v638`, `v639`, `v640`, `v641`, `v642`, `v643`, `v644`, `v645`, `v646`, `v647`, `v648`, `v649`, `v650`, `v651`, `v652`, `v653`, `v654`, `v655`, `v656`, `v657`, `v658`, `v659`, `v660`, `v661`, `v662`, `v663`, `v664`, `v665`, `v666`, `v667`, `v668`, `v669`, `v670`, `v671`, `v672`, `v673`, `v674`, `v675`, `v676`, `v677`, `v678`, `v679`, `v680`, `v681`, `v682`, `v683`, `v684`, `v685`, `v686`, `v687`, `v688`, `v689`, `v690`, `v691`, `v692`, `v693`, `v694`, `v695`, `v696`, `v697`, `v698`, `v699`, `v700`, `v701`, `v702`, `v703`, `v704`, `v705`, `v706`, `v707`, `v708`, `v709`, `v710`, `v711`, `v712`, `v713`, `v714`, `v715`, `v716`, `v717`, `v718`, `v719`, `v720`, `v721`, `v722`, `v723`, `v724`, `v725`, `v726`, `v727`, `v728`, `v729`, `v730`, `v731`, `v732`, `v733`, `v734`, `v735`, `v736`, `v737`, `v738`, `v739`, `v740`, `v741`, `v742`, `v743`, `v744`, `v745`, `v746`, `v747`, `v748`, `v749`, `v750`, `v751`, `v752`, `v753`, `v754`, `v755`, `v756`, `v757`, `v758`, `v759`, `v760`, `v761`, `v762`, `v763`, `v764`, `v765`, `v766`, `v767`, `v768`, `v769`, `v770`, `v771`, `v772`, `v773`, `v774`, `v775`, `v776`, `v777`, `v778`, `v779`, `v780`, `v781`, `v782`, `v783`, `v784`, `v785`, `v786`, `v787`, `v788`, `v789`, `v790`, `v791`, `v792`, `v793`, `v794`, `v795`, `v796`, `v797`, `v798`, `v799`, `v800`, `v801`, `v802`, `v803`, `v804`, `v805`, `v806`, `v807`, `v808`, `v809`, `v810`, `v811`, `v812`, `v813`, `v814`, `v815`, `v816`, `v817`, `v818`, `v819`, `v820`, `v821`, `v822`, `v823`, `v824`, `v825`, `v826`, `v827`, `v828`, `v829`, `v830`, `v831`, `v832`, `v833`, `v834`, `v835`, `v836`, `v837`, `v838`, `v839`, `v840`, `v841`, `v842`, `v843`, `v844`, `v845`, `v846`, `v847`, `v848`, `v849`, `v850`, `v851`, `v852`, `v853`, `v854`, `v855`, `v856`, `v857`, `v858`, `v859`, `v860`, `v861`, `v862`, `v863`, `v864`, `v865`, `v866`, `v867`, `v868`, `v869`, `v870`, `v871`, `v872`, `v873`, `v874`, `v875`, `v876`, `v877`, `v878`, `v879`, `v880`, `v881`, `v882`, `v883`, `v884`, `v885`, `v886`, `v887`, `v888`, `v889`, `v890`, `v891`, `v892`, `v893`, `v894`, `v895`, `v896`, `v897`, `v898`, `v899`, `v900`, `v901`, `v902`, `v903`, `v904`, `v905`, `v906`, `v907`, `v908`, `v909`, `v910`, `v911`, `v912`, `v913`, `v914`, `v915`, `v916`, `v917`, `v918`, `v919`, `v920`, `v921`, `v922`, `v923`, `v924`, `v925`, `v926`, `v927`, `v928`, `v929`, `v930`, `v931`, `v932`, `v933`, `v934`, `v935`, `v936`, `v937`, `v938`, `v939`, `v940`, `v941`, `v942`, `v943`, `v944`, `v945`, `v946`, `v947`, `v948`, `v949`, `v950`, `v951`, `v952`, `v953`, `v954`, `v955`, `v956`, `v957`, `v958`, `v959`, `v960`, `v961`, `v962`, `v963`, `v964`, `v965`, `v966`, `v967`, `v968`, `v969`, `v970`, `v971`, `v972`, `v973`, `v974`, `v975`, `v976`, `v977`, `v978`, `v979`, `v980`, `v981`, `v982`, `v983`, `v984`, `v985`, `v986`, `v987`, `v988`, `v989`, `v990`, `v991`, `v992`, `v993`, `v994`, `v995`, `v996`, `v997`, `v998`, `v999`, `v1000`. **3** is marked next to the `data` variable.
- Console Panel (Bottom Left):** Shows the execution of the function `iterate_M`. The output is a list of results. **2** is marked next to the `iterate_M` function call.
- Files Panel (Bottom Right):** Shows the project file structure. **4** is marked next to the `data` file.

RStudio panels: source



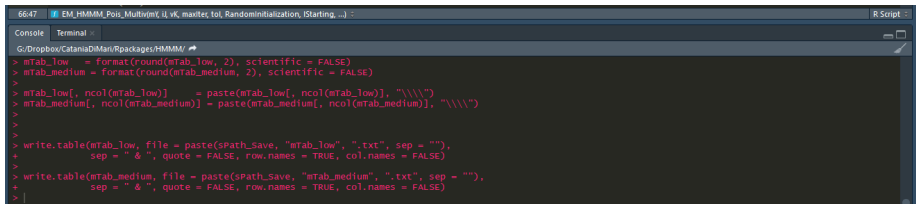
```

File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function Addins
CppFunctions.cpp OrganizeData.R code.R MonteCarlo.R MonteCarlo.R MonteCarlo_ExtractResults.R Tab.R EM.R
Source on Save Run Source
76 Iterate_EM <- function(my, ib = 10, fun_EM = EM_HMM_Pois_Multiv, cluster = NULL, ...) {
77
78   Fit_nonRandom = try(fun_EM(my, RandomInitialization = FALSE, ...), silent = TRUE)
79
80   if (!is(Fit_nonRandom, "try-error")) {
81     dLLK_nonRandom = c(tail(Fit_nonRandom$LLKSeries, 1))
82   } else {
83     iter = 0
84     while (is(Fit_nonRandom, "try-error") | iter <= 10) {
85       Fit_nonRandom = try(fun_EM(my, RandomInitialization = FALSE, ...), silent = TRUE)
86       iter = iter + 1
87     }
88     if (is(Fit_nonRandom, "try-error")) {
89       dLLK_nonRandom = NA
90     } else {
91       dLLK_nonRandom = c(tail(Fit_nonRandom$LLKSeries, 1))
92     }
93   }
94   if (is.null(cluster)) {
95     lFit_random = lapply(1:ib, function(b, my, ...) {
96       foo = try(fun_EM(my = my, RandomInitialization = TRUE, ...))
97       if (is(foo, "try-error")) {
98         foo = list(LLKSeries = NA)
99       }
100       return(foo)
101     })
102   }
103 }
104
105
106
107
108
109
110
111
112
113
114
115
116
66:47 EM_HMM_Pois_Multiv(int iL, vL, mvariter, tol, RandomInitialization, iStarting, ...) R Script

```

Where you write your code.

RStudio panels: Console



```
66:47 EM_HMMM_Pois_Multiv(mv, il, vK, maxiter, tol, RandomInitialization, lStarting, ...) : R Script
Console Terminal
G:/Dropbox/CataniaDiMari/Rpackages/HMMM/
> mTab_low = format(round(mTab_low, 2), scientific = FALSE)
> mTab_medium = format(round(mTab_medium, 2), scientific = FALSE)
>
> mTab_low[, ncol(mTab_low)] = paste(mTab_low[, ncol(mTab_low)], "\\")
> mTab_medium[, ncol(mTab_medium)] = paste(mTab_medium[, ncol(mTab_medium)], "\\")
>
>
> write.table(mTab_low, file = paste(sPath_Save, "mTab_low", ".txt", sep = ""),
+             sep = " & ", quote = FALSE, row.names = TRUE, col.names = FALSE)
>
> write.table(mTab_medium, file = paste(sPath_Save, "mTab_medium", ".txt", sep = ""),
+             sep = " & ", quote = FALSE, row.names = TRUE, col.names = FALSE)
>
```

Where you run your code.

RStudio panels: Environment

The screenshot shows the RStudio Environment panel. At the top, there are tabs for Environment, History, Files, Connections, and Build. Below these is a toolbar with icons for Import Dataset and a search icon. The main area is titled 'Global Environment' and contains a search bar. The panel is divided into three sections: Data, Values, and Functions.

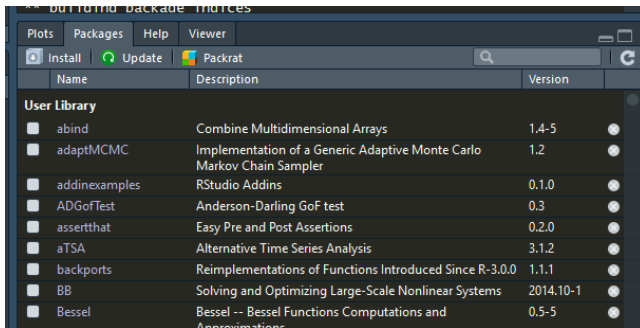
Data		
lstat	List of 48	
ltab	List of 2	
mOut	Large matrix (128000 elements, 1008.4 Kb)	
mTab_low	chr [1:12, 1:12] "-0.01" "-0.04" "-0.04" "-0.09..."	
mTab_medium	chr [1:12, 1:12] "-0.06" "-0.09" "-0.04" "-0.05..."	

Values		
vFiles	chr [1:72] "T.1000.N.1.J.2.L.2.Sep.low.rdata" "T..."	
vJ	num [1:2] 2 4	
vL	num [1:2] 2 4	
vN	num [1:3] 1 3 6	
vRowNames	chr [1:12] "\$J = 2\$ \$K = 2\$" "\$J = 2\$ \$K = 4\$" "\$..."	
vseparation	chr [1:2] "low" "medium"	
vspec	chr [1:2] "T.1000.N.6.J.4.L.4.Sep.medium" "T.5000..."	
VT	num [1:2] 1000 5000	

Functions		
f_Compute_Stat	function (mOut)	
f_stat	function (x, ...)	
f_trimmed_mean	function (vx, dLower = 0.025, dupper = 0.975)	
f_trimmed_rmse	function (vx, dLower = 0.025, dupper = 0.975)	

Where you see the object stored in memory

RStudio panels: Plots, Packages, Help



- Figures are displayed
- List of available packages
- Help

Standard functions in R

- R comes with a lot of standard functions
 - Printing stuff to the screen, inverting matrices, performing OLS, generating random numbers... no need to reinvent the wheel!
- These functions are also available in R packages. However these are 'special' packages which are included by default in R and don't need to be loaded. Examples are: 'stats', 'base', 'graphics'....

Standard functions in R

- These functions generally have sensible names:
 - `print` to print something to the screen
 - `plot` to plot figures
 - `solve` to invert a matrix
 - `lm` to perform OLS ('linear model' \rightarrow 'lm')
 - `sum`, `mean`, `median`...
- Obviously, too many to list here. R manuals are freely available online!
<https://cran.r-project.org/>

External packages in R

More than 12'000 external packages are available on CRAN:
<https://cran.r-project.org/web/packages/>.

Knowing the name of a package, say ("rugarch"), this can be installed in R with:
`instal.packages("rugarch")`

After a package has been installed, in order to load it into the R environment we need to run:

```
library("rugarch").
```

Types of variables

The function `class()` allows you to find the class of an R object. For instance:

```
> cfoo = "sun"
> class(cfoo)
[1] "character"
>
> bfoo = TRUE
> class(bfoo)
[1] "logical"
>
> dfoo = 1.785632458
> class(dfoo)
[1] "numeric"
```

The "foo" terminology stands for a variable which is not important, i.e. a variable where intermediate results are stored. We will later see that "c", "b", and "d" before the name "foo" does actually have a reason!

Vectors, Matrices, Arrays, Data frames

Data can be organized in different ways in R depending on the particular needs.

If we plan to do linear algebra we want to use:

```
> vY = c(1, 2, 5.48652) # vectors
```

```
> mY = matrix(c(1, 2, 3,  
                4, 5, 6,  
                7, 8, 9.485), ncol = 3) # matrices
```

```
> aY = array(1:27, dim = c(3, 3, 3))# arrays
```

If we want to organize data (numeric/character or mixed) we use:

```
d = data.frame(x = 1, y = 1:10, fac = LETTERS[1:10]) #data frames
```

vectors, matrices and arrays can be: "numeric" or "character" i.e., they can contain only two types or variables (we cannot have a vector with some elements "numeric" and others "character"). data.frames can contain both.

Access elements

To access elements of vectors, matrices and arrays we use the square brackets:

```
> vY[1]
[1] 1
> vY[1:2]
[1] 1 2
> vY[c(3, 1)]
[1] 5.48652 1.00000
> mY[1, 1]
[1] 1
> mY[1, ]
[1] 1 4 7
> mY[1, c(1, 3)]
[1] 1 7
> aY[1, 1, 1]
[1] 1
> aY[1, 1, ]
[1] 1 10 19
```

HELP!

When you are in troubles the `help()` function is your friend! Whenever you want to understand the functioning of any function, say `list`, you type:

```
> help(list)
```

This is equivalent to:

```
> ?list
```

If you don't remember the function name but part of it, or something related to it, you can use the `"?"` operator, for example:

```
> ??download
```

GETTING STARTED WITH R

Leopoldo Catania

Aarhus University and CREATES

`leopoldo.catania@econ.au.dk`

Outline

- Arithmetic
- Variables, Functions
- Vectors
- Matrices
- Logical expressions
- Missing data: NA
- Lists
- Factors
- Dataframes

Working Directory

When you run R, it nominates one of the directories on your hard drive as a working directory, which is where it looks for user-written programs and data files.

You can determine the current working directory using the command `getwd`.

You can do this using the command `setwd("dir")`, where `dir` is the directory address.

```
> getwd()
[1] "C:/Users/leopo/Documents"
>
> dir = "G:/foo" #this path needs to exist
>
> setwd(dir)
>
> getwd()
[1] "G:/foo"
```

Arithmetic

R uses the usual symbols for addition $+$, subtraction $-$, multiplication $*$, division $/$, and exponentiation $^$.

Parentheses () can be used to specify the order of operations.

```
> (1 + 1/100)^100  
[1] 2.704814
```

Notice that by default R prints 7 significant digits. You can change the display to x digits using `options(digits = x)`.

See `help(options)` for other options.

Build in Functions

R has a number of built-in functions, for example `sin(x)`, `cos(x)`, `tan(x)`, (all in radians), `exp(x)`, `log(x)`, and `sqrt(x)`. Some special constants such as `pi` (π) are also predefined.

```
> exp(1)
[1] 2.718282
> options(digits = 16)
> exp(1)
[1] 2.718281828459045
> pi
[1] 3.141592653589793
> sin(pi/6)
[1] 0.49999999999999999
```

The functions `floor(x)` and `ceiling(x)` round down and up, respectively, to the nearest integer.

Defining variables

To assign a value to a variable we use (almost equivalently) the assignment commands `=` and `<-`. For example:

```
> iX <- 100
```

and

```
> iX = 100
```

are equivalent. Later we see when this is not the case. We can perform operations with `iX`:

```
> iX  
[1] 100  
> (1 + 1/iX)^iX  
[1] 2.704814
```

Notice that:

```
> iX = iX + 1  is allowed.  
> iX  
[1] 101
```

Functions

In mathematics a function takes one or more arguments (or inputs) and produces one or more outputs (or return values). Functions in R work in an analogous way.

Consider the `seq` function. This function allows you to create sequence of numbers:

```
> seq(from = 1, to = 9, by = 2)
[1] 1 3 5 7 9
```

Notice that by default (see `help(seq)`), the `by` argument is equal to 1, such that:

```
> seq(from = 1, to = 3)
[1] 1 2 3
```

Arguments of functions

Every function has a default order for the arguments. For `seq` this is: `from`, `to`, `by`.

If you provide arguments in this order, then they do not need to be named:

```
> seq(1, 9, 2)
[1] 1 3 5 7 9
```

But you can choose to give the arguments out of order provided you give them names in the format `argument_name = expression`.

```
> seq(by = 2, to = 9, from = 1)
[1] 1 3 5 7 9
```

Which is of course different from:

```
> seq(2, 9, 1)
[1] 2 3 4 5 6 7 8 9
```

Vectors

We have already seen the `c` function. However, vectors can be created by any function that returns vectors as output. For instance `seq` and `rep`. The `rep(vX, n)` function replicates the vector `vX`, `n` times:

```
> vX = c(1, 3, 4)
> rep(vX, 3)
[1] 1 3 4 1 3 4 1 3 4
```

The function `c` also accepts vectors as inputs:

```
> vX <- seq(1, 20, by = 2)
[1] 1 3 5 7 9 11 13 15 17 19

> vY <- rep(3, 4)
[1] 3 3 3 3

> vZ <- c(vY, vX)
[1] 3 3 3 3 1 3 5 7 9 11 13 15 17 19
```

Vector operations

All algebraic operations are defined for vectors and act on each element separately, that is, elementwise:

```
> vX <- c(1, 2, 3)
> vY <- c(4, 5, 6)
> vX * vY
[1] 4 10 18
> vX + vY
[1] 5 7 9
> vY ^ vX
[1] 4 25 216
```

However care should be taken when vectors of unequal length are used:

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

Warning message:

In `c(1, 2, 3) + c(1, 2)` :

longer object length is not a multiple of shorter object length

Vector operations

When you apply an algebraic expression to two vectors of unequal length, R automatically repeats the shorter vector until it has something the same length as the longer vector.

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
> (1:10) ^ c(1, 2)
[1] 1 4 3 16 5 36 7 64 9 100
```

This happens even when the shorter vector is of length 1, allowing the shorthand notation:

```
> 2 + c(1, 2, 3)
[1] 3 4 5
> 2 * c(1, 2, 3)
[1] 2 4 6
```

For example using the modulus operator: $a \bmod b$ (%%):

```
1:20 %% 3
[1] 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

Missing Data: NA

In real experiments it is often the case, for one reason or another, that certain observations are missing.

Depending on the statistical analysis involved, missing data can be ignored or invented (a process called imputation).

R represents missing observations through the data value NA. They can be mixed in with all other kinds of data:

```
> vA <- c(11, NA, 13)
> vA
[1] 11 NA 13
```

Performing analysis with NAs can be problematic:

```
> mean(vA) # NAs can propagate
[1] NA
```

The logical argument `na.rm` is often specified to deal with NAs.

```
> mean(vA, na.rm = TRUE) # NAs can be removed
[1] 12
```

Searching for NAs

Sometimes you want to search for NAs in your dataset to prevent misleading results at the end of your analysis. The function `is.na` searches for NAs inside vectors (also matrices and arrays) and returns a logical output of the same size of the input provided:

```
> vA <- c(11, NA, 13)
> is.na(vA) # identify missing elements
[1] FALSE TRUE FALSE
```

When `length(vA)` is very large we might want to write:

```
> any(is.na(vA)) # are any missing?
[1] TRUE
```

In order to remove NAs we can use:

```
> na.omit(vA)
[1] 11 13
attr(,"na.action")
[1] 2
attr(,"class")
[1] "omit"
```


Logical Expressions

A logical expression is formed using the comparison operators:

- $<$ lower than
- $>$ bigger than
- \leq lower or equal than
- \geq bigger or equal than
- $==$ equal to
- $!=$ not equal to
- $\&$ and
- $|$ or
- $!$ not

These operators return a logical output:

```
> vX = c(1, 2, 3, 4)
> vX == 2
[1] FALSE TRUE FALSE FALSE
> vX != 2
[1] TRUE FALSE TRUE TRUE
```

Logical Expressions

Vectors (but also matrices and arrays) can be accessed also using logical indicators:

```
> vX <- c(1, 3, 4, 18)
```

```
> vX > 2
```

```
[1] FALSE  TRUE  TRUE  TRUE
```

```
> vX[vX > 2 & vX < 10]
```

```
[1] 3 4
```

The subset function can be used for a similar scope:

```
> subset(vX, subset = vX > 2)
```

```
[1] 3 4 18
```

Logical Expressions: & and |

Take two logical objects:

```
> vX = 4  
> vY = vX > 2  
> vY  
[1] TRUE  
>  
> vZ = vX < 3  
> vZ  
[1] FALSE
```

The & operator returns TRUE if both vY and vZ are TRUE, FALSE otherwise:

```
> vY & vZ  
[1] FALSE
```

The | operator returns TRUE if y or z are TRUE, FALSE otherwise:

```
> vY | vZ  
[1] TRUE
```

Sequential `&&` and `||`

The logical operators `&&` and `||` are sequentially evaluated versions of `&` and `|`, respectively.

To evaluate `vX & vY`, R first evaluates `vX` and `vY`, then returns `TRUE` if `vX` and `vY` are both `TRUE`, `FALSE` otherwise.

To evaluate `vX && vY`, R first evaluates `vX`. If `vX` is `FALSE` then R returns `FALSE` without evaluating `vY`. If `vX` is `TRUE` then R evaluates `vY` and returns `TRUE` if `vY` is `TRUE`, `FALSE` otherwise.

Sequential evaluation of `vX` and `vY` is useful when `vY` is not always well defined, or when `vY` takes a long time to compute.

Sequential && and ||

As an example of the first instance, suppose we wish to know if $vX * \sin(1/vX) = 0$.

```
> vX <- 0
> vX * sin(1/vX) == 0
[1] NA
Warning message:
In sin(1/vX) : NaNs produced

> (vX == 0) | (sin(1/vX) == 0)
[1] TRUE
Warning message:
In sin(1/vX) : NaNs produced

> (vX == 0) || (sin(1/vX) == 0)
[1] TRUE
```

Note that && and || only work on scalars, whereas & and | work on vectors on an element-by-element basis.

Matrices

We have already seen how to build matrices in R:

```
> mA <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
```

To retrieve the dimension of a matrix use `dim`:

```
> dim(mA)
[1] 2 3
```

Useful functions for matrices:

- `diag`, extract the diagonal elements and return a vector or create a diagonal matrix, see `help(diag)`!
- `rbind`, join matrices with rows of the same length (stacking vertically)
- `cbind`, join matrices with columns of the same length (stacking horizontally)
- `solve`, invert a matrix
- `eigen`, extract eigenvalues and associated eigenvectors of a matrix
- `t`, transpose of matrix

Multiplication with matrices

Define:

```
> mA <- matrix(c(3, 5, 2, 3), nrow = 2, ncol = 2)
> mB <- matrix(c(9, 4, 1, 2), nrow = 2, ncol = 2)
>
> mA
      [,1] [,2]
[1,]    3    2
[2,]    5    3
> mB
      [,1] [,2]
[1,]    9    1
[2,]    4    2
```

Multiplication with matrices

Element-wise multiplication, $A \circ B$ (Hadamard product):

```
> mA * mB
      [,1] [,2]
[1,]    27    2
[2,]    20    6
```

Matrix multiplication, AB :

```
> mA %*% mB
      [,1] [,2]
[1,]    35    7
[2,]    57   11
```

Kronecker product, $A \otimes B$:

```
> kronecker(mA, mB)
      [,1] [,2] [,3] [,4]
[1,]    27    3   18    2
[2,]    12    6    8    4
[3,]    45    5   27    3
[4,]    20   10   12    6
```


Lists

We have seen that all the elements of a vector have to be of the same type: `numeric`, `character`, or `logical`. The type of the vector is called: mode

A list is an indexed set of objects (and so has a length) whose elements can be of different types, including other lists! The mode of a list is `list`.

A list is just a generic container for other objects and the power and utility of lists comes from this generality.

In R lists are often used for collecting and storing complicated function output.

For example, the first element of a list can be a vector, the second can be another list and the third can be a matrix.

Lists

A list is created using the `list(...)` command, with comma-separated arguments:

```
> my.list <- list("one", TRUE, 3, c("f", "o", "u", "r"))
```

Double square brackets are used to extract a single element:

```
> my.list[[1]]  
[1] "one"  
> mode(my.list[[1]])  
[1] "character"
```

Single square brackets are used to select a sublist:

```
> my.list[1]  
[[1]]  
[1] "one"  
> mode(my.list[1])  
[1] "list"
```

Lists

When displaying a list, R uses double square brackets `[[1]]`, `[[2]]`, etc., to indicate list elements, then single square brackets `[1]`, `[2]`, etc., to indicate vector elements within the list.

The elements of a list can be named when the list is created, using arguments of the form `name1 = x1`, `name2 = x2`, etc.:

```
my.list <- list(first = "one", second = TRUE, third = 3,  
  fourth = c("f","o","u","r"))
```

```
> names(my.list)  
[1] "first" "second" "third" "fourth"
```

```
> my.list$second  
[1] TRUE
```

Note the use of the `$` operator to access named lists.

Lists

Alternatively, a list elements can be named later by assigning a value to the `names` attribute:

```
> my.list <- list( "one", TRUE, 3, c("f", "o", "u", "r"))  
  
> names(my.list) <- c("first", "second", "third", "fourth")
```

To flatten a list `x`, that is convert it to a vector, we use `unlist(x)`:

```
> x <- list(1, c(2, 3), c(4, 5, 6))  
> unlist(x)  
[1] 1 2 3 4 5 6
```

If the list object itself comprises lists, then these lists are also flattened, unless the argument `recursive = FALSE` is set in `unlist`.

Dataframes

We have already seen how to work in R with numbers, strings, and logical values.

We have also worked with homogeneous collections of such objects, grouped into numeric, character, or logical vectors.

The defining characteristic of the vector data structure in R is that all components must be of the same mode.

Obviously to work with datasets from real experiments we need a way to group data of differing modes!

Dataframes

Suppose to have the following dataset representing a forestry experiment in which we randomly selected a number of plots and then from each plot selected a number of trees.

Plot	Tree	Species	Diameter (cm)	Height (m)
2	1	DF	39	20.5
2	2	WL	48	33.0
3	2	GF	52	30.0
3	5	WC	36	20.7
3	8	WC	38	22.5
⋮	⋮	⋮	⋮	⋮

For each tree we measured its height and diameter (which are `numeric`), and also the species of tree (which is a `character` string).

Dataframes

As experimental data collated in a table look like an array, you may be tempted to represent it in R as a matrix.

But in R matrices cannot contain heterogeneous data (data of different modes, like numeric and character).

Lists and dataframes are able to store much more complicated data structures than matrices.

A dataframe is a list of vectors restricted to be of equal length. Each vector (column of the dataframe) can be of any of the basic modes of object. To create a dataframe we write:

```
mData <- data.frame(Plot = c(1, 2, 2, 5, 8, 8),  
                    Tree = c(2, 2, 3, 3, 3, 2),  
                    Species = c("DF", "WL", "GF", "WC", "WC", "GF"),  
                    Diameter = c(39, 48, 52, 35, 37, 30),  
                    Height = c(20.5, 33.0, 30.0, 20.7, 22.5, 20.1))
```

Dataframes: extract

Each column, or variable, in a dataframe has a unique name. We can extract that variable by means of the dataframe name, the column name, the a dollar sign, or as we do for a matrix:

```
> mData$Diameter  
[1] 39 48 52 35 37 30
```

```
> mData[["Diameter"]]  
[1] 39 48 52 35 37 30
```

```
> mData[[4]]  
[1] 39 48 52 35 37 30
```

```
> mData[, 4]  
[1] 39 48 52 35 37 30
```

```
> mData[, "Diameter"]  
[1] 39 48 52 35 37 30
```


Dataframes: assign

To assign a new variable to a dataframe we write:

```
mData$newdata <- c(1, 2, 3, 4, 5, 6)
```

If the new variable is the same across all the observations we can write:

```
mData$newdata2 <- TRUE
```

This also works as long as the number of rows of the dataframe is a multiple of the length of the new variable (if it is not a multiple we get an error):

```
mData$newdata3 <- c("one", "two")
```

```
> mData
```

	Plot	Tree	Species	Diameter	Height	newdata1	newdata2	newdata3
1	1	2	DF	39	20.5	1	TRUE	one
2	2	2	WL	48	33.0	2	TRUE	two
3	2	3	GF	52	30.0	3	TRUE	one
4	5	3	WC	35	20.7	4	TRUE	two
5	8	3	WC	37	22.5	5	TRUE	one
6	8	2	GF	30	20.1	6	TRUE	two

BASIC PROGRAMMING

Leopoldo Catania

Aarhus University and CREATES

`leopoldo.catania@econ.au.dk`

Outline

- The `if` statement
- The `for` loop
- The `while` loop
- Vector-based programming
- Load data in R
- Write data from R
- Plotting

Programming: Intro

This lecture introduces a set of basic programming constructs, which are the building blocks of most programs.

Some of these tools are used by practically all programming languages, for example, conditional execution by `if` statements, and looped execution by `for` and `while` statements.

Notice that, code that seems to be efficient in another language may not be efficient in R. For example, R is known to be quite slow in computing `for` and `while` loops. On the other hand, R is fast in doing vector-based operations.

Programming: Intro

A program or script is just a list of commands, which are executed one after the other.

A program is usually composed by three parts:

- input
- computations
- output

Recall that documentation also plays an important role!!!

Usually, we write the list of commands in separate files, called a scripts, which we can save in our Hard Drive.

Suppose we have a program saved as `prog.r` in the working directory (recall the `getwd` and `setwd` functions). In order to run or execute the program we use the command:

```
source("prog.r")
```

The if statement

It is often useful to choose the execution of some or other part of a program to depend on a condition. The if function has the form

```
if (logical_expression) {  
  expression_1  
  ...  
}
```

A natural extension of the if command includes an else part:

```
if (logical_expression) {  
  expression_1  
  ...  
} else {  
  expression_2  
  ...  
}
```

The if statement

When an if expression is evaluated, if `logical_expression` is TRUE then the first group of expressions is executed and the second group of expressions is not executed. Conversely if `logical_expression` is FALSE then only the second group of expressions is executed.

```
dX = rnorm(1)

if (dX > 0) {

print("x is positive")

} else {

print("x is nonpositive")

}
```

Nested if

Nested if statements are constructed using the `else if` command:

```
if (logical_expression_1) {  
  expression_1  
  ...  
} else if (logical_expression_2) {  
  expression_2  
  ...  
} else if (logical_expression_3) {  
  expression_3  
  ...  
} else {  
  expression_4  
  ...  
}
```


The for loop

The `for` command has the following form, where `x` is a simple variable and `vector` is a vector.

```
for (x in vector) {  
  expression_1  
  ...  
}
```

When executed, the `for` command executes the group of expressions within the braces `{}` once for each element of `vector`:

```
for (iX in 1:50) {  
  print(iX)  
}
```

Note that `vector` can be a `list`, which we cover later.

for loop example

Consider the following for loop example:

```
> vX <- seq(1, 9, by = 2)
[1] 1 3 5 7 9
> dSum_x <- 0
> for (iX in vX) {
  dSum_x <- dSum_x + iX
  cat("The current loop element is", iX, "\n")
  cat("The cumulative total is", dSum_x, "\n")
}
The current loop element is 1
The cumulative total is 1
...
The current loop element is 9
The cumulative total is 25
> sum(vX)
[1] 25
> cumsum(vX)
[1] 1 4 9 16 25
```

The while loop

Often we do not know beforehand how many times we need to go around a loop. That is, each time we go around the loop, we check some condition to see if we are done yet.

In this situation we use a `while` loop, which has the form:

```
while (logical_expression) {  
# This should have an impact on logical_expression  
expression_1  
...  
}
```

When a `while` command is executed, `logical_expression` is evaluated first. If it is `TRUE` then the group of expressions in braces `{}` is executed.

Until `logical_expression` becomes `FALSE`, the `while` command executes the group of expressions in braces `{}`.

Warning: `while` loop can run forever!

Difference between for and while loops

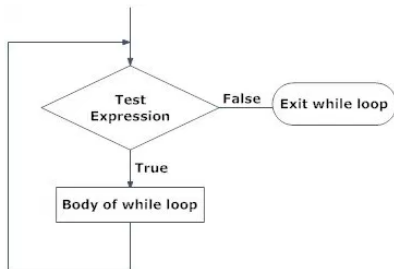
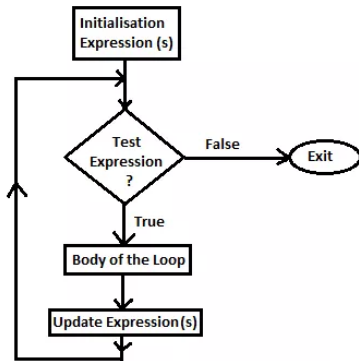


Figure: Flowchart of while loop

Vector-based programming

It is often necessary to perform an operation upon each of the elements of a vector.

R is set up so that such programming tasks can be accomplished using vector operations rather than looping.

Using vector operations is more efficient computationally, as well as more concise literally. For example, find the sum of the first n squares:

```
> iN <- 100
> dS <- 0
> for (i in 1:iN) {
  dS <- dS + i^2
}
> dS
[1] 338350

> sum((1:iN)^2)
[1] 338350
```

Computational time: loop vs vector-operation

```

> iN <- 1e8
> system.time({
  dS <- 0
  for (i in 1:iN) {
    dS <- dS + i^2
  }
  dS
})
   user  system elapsed 
  3.41    0.00    3.43 
> system.time({sum((1:iN)^2)})
   user  system elapsed 
  0.44    0.09    0.53 

```

In both approaches the vector $1:N$ needs to be created. This occupies approximately 400MB of RAM. Can you think of an alternative procedure which does not require the evaluation of $1:iN$?

Load data in R

R implements functionalities to read from text files like: `.txt` and `.csv`. In order to read excel files external libraries are required such as: **excel**, **gdata**, **rodbc**, **XLConnect**, **xls**, **xlsReadWrite**, **xlsx** are needed.

R provides a number of ways to read data from a file: `scan`, `read.table`, `read.csv`. In this course we will mostly use `read.table`.

```
read.table(file, header = FALSE, sep = "", dec = ".",  
           row.names, col.names, na.strings = "NA", skip = 0, ...)
```

... means that other arguments are available. See `help(read.table)`.

- **file**: the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an absolute path, the file name is relative to the current working directory, `getwd()`.
- **header**: a logical value indicating whether the file contains the names of the variables as its first line.

The `read.table` function

- `sep`: the field separator character. Values on each line of the file are separated by this character. If `sep = ""` (the default) the separator is 'white space'.
- `dec`: the character used in the file for decimal points.
- `row.names`: a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row name.
- `col.names`: a vector of optional names for the variables. The default is to use "V" followed by the column number.
- `na.strings`: a character vector of strings which are to be interpreted as NA values.
- `skip`: the number of lines of the data file to skip before beginning to read data.

Load MAERSK historical prices

Load the MAERSK historical prices contained in the MAERSK-B.CO.csv file located in your working directory `getwd()`.

```
mData = read.table(file = "MAERSK-B.CO.csv",
                    sep = ",", dec = ".",
                    header = TRUE, row.names = 1,
                    na.strings = "null")
```

```
> head(mData)
```

	Open	High	Low	Close	Adj.Close	Volume
2000-02-01	7640.00	7666.67	7333.33	7466.67	2267.703	4590
2000-02-02	7466.67	7666.67	7400.00	7566.67	2298.074	5640
2000-02-03	7741.07	7800.00	7659.00	7800.00	2368.939	3090
2000-02-04	7800.00	7800.00	7525.80	7600.00	2308.197	1185
2000-02-07	7660.00	7666.67	7566.67	7600.00	2308.197	1035
2000-02-08	7600.00	8200.00	7600.00	8110.80	2463.332	12990

```
> dim(mData)
```

```
[1] 4571    6
```

Descriptive Statistics of the MAERSK log-returns

First search for NAs:

```
> any(is.na(mData[, "Adj.Close"]))
[1] TRUE
```

how many ?

```
> length(which(is.na(mData[, "Adj.Close"])))
[1] 59
```

Compute financial log-returns omitting the NAs:

```
vY = diff(log(na.omit(mData[, "Adj.Close"])))
```

Compute descriptive statistics:

```
> c("mean" = mean(vY), "sd" = sd(vY), "median" = median(vY))
      mean      sd      median
0.0003439318 0.0220065696 0.0000000000
```

Output a file

R provides a number of commands for writing output to a file.

We will generally use `write.table` for writing numeric values and `cat` for writing text, or a combination of numeric and character values.

The command `write.table` has the form:

```
write.table(x, file = "", quote = TRUE, sep = " ",  
            na = "NA", dec = ".", row.names = TRUE,  
            col.names = TRUE, ...)
```

- `x`: the object to be written, preferably a `matrix` or `data.frame`.
- `file`: character string naming a file.

The `write.table` function

- `quote`: a logical value (TRUE or FALSE). If TRUE, any character columns will be surrounded by double quotes ("").
- `sep` and `dec` work as in `read.table`.
- `na`: the string to use for missing values in the data.
- `row.names`: either a logical value indicating whether the row names of `x` are to be written along with `x`, or a character vector of row names to be written.
- `col.names`: either a logical value indicating whether the column names of `x` are to be written along with `x`, or a character vector of column names to be written.

Example:

```
write.table(vY, file = "MAERSK_returns.txt", row.names = FALSE,  
           col.names = FALSE, dec = ".")
```

Are you able to set `row.names` equal to the log-returns dates?

Plotting

R provides a huge number of plotting routines. The generic function for plotting of R objects is `plot`. See `help(plot)` to discover how many options are available.

External packages can be used for plotting. One of the most used is **ggplot2**, see <http://ggplot2.org/>.

`plot` can be used to represent a pair of data points `x` and `y`:

```
vX = seq(-10, 10, 0.1)
vY = sin(vX)
```

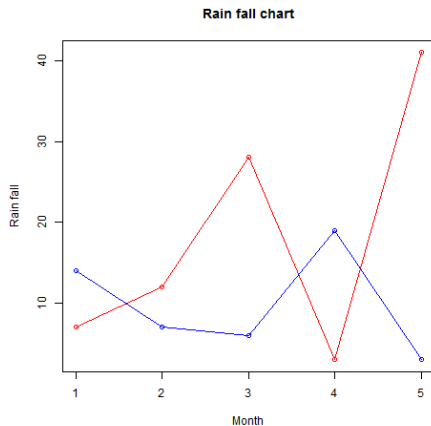
```
plot(vX, vY)
```

of a single vector of points:

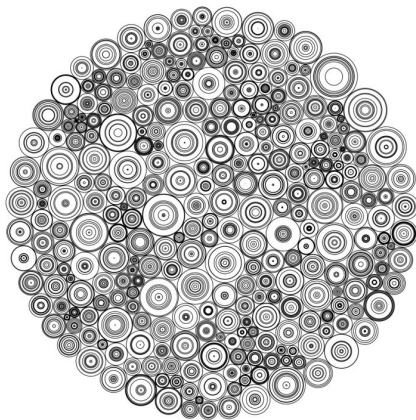
```
vZ = runif(10)
plot(vZ)
```

in this second case the `x`-axis is simply replaced by `1:length(vZ)`.

Plotting: ugly plot



Plotting: nice plot



see <https://www.r-graph-gallery.com/> for more nice plots!

Basic arguments for plot

- `type`, the type of plot: `type = "p"` for points, `type = "l"` for lines... see `help(plot)`.
- `main`, title of the plot.
- `xlab`, `ylab`: labels for the x- and y-axes, respectively.
- `col`, colour e.g. `col = "red"`.
- `lty`, if `type = "l"` determines the line type, e.g., 0=blank, 1=solid (default), 2=dashed, 3=dotted...
- `lwd`, if `type = "l"` determines line width, e.g., `lwd = 1` normal width, `lwd = 2` more width etc..

To add points `(x[1], y[1])`, `(x[2], y[2])`, ... to the current plot, use `points(x, y)`. To add lines instead use `lines(x, y)`. To add text use `text(x, y, "text")`

Vertical or horizontal lines can be drawn using `abline(v = xpos)` and `abline(h = ypos)`

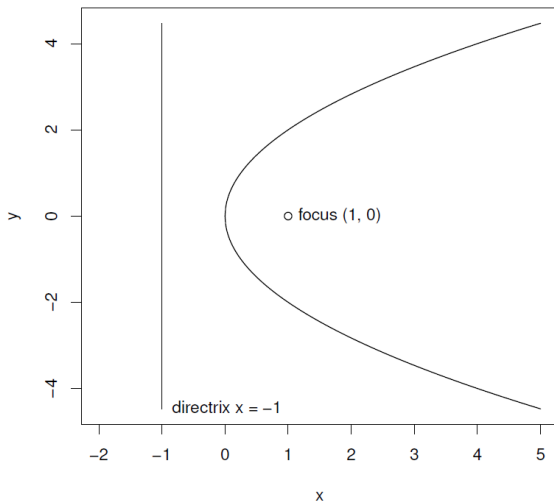
The functions `points`, `lines`, `abline` also accept the `col`, `lty`, `lwd` arguments.

Plot of a parabola

```
x <- seq(0, 5, by = 0.01)
y.upper <- 2*sqrt(x)
y.lower <- -2*sqrt(x)
y.max <- max(y.upper)
y.min <- min(y.lower)
plot(c(-2, 5), c(y.min, y.max), type = "n", xlab = "x", ylab = "y")
lines(x, y.upper)
lines(x, y.lower)
abline(v=-1)
points(1, 0)
text(1, 0, "focus (1, 0)", pos=4)
text(-1, y.min, "directrix x = -1", pos = 4)
title("The parabola  $y^2 = 4*x$ ")
```

Plot of a parabola

The parabola $y^2 = 4x$



PROGRAMMING WITH FUNCTIONS

Leopoldo Catania

Aarhus University and CREATES

`leopoldo.catania@econ.au.dk`

John Chambers

"To understand computations in R, two slogans are helpful:

- *Everything that exists is an object.*
- *Everything that happens is a function call"*

– John Chambers

Outline

- Definitions of functions in R
- Function's variables
- Arguments of a function
- Ellipsis
- Vector-based programming using functions
- The `apply` family
- Recursive programming

Intro

In this lecture we cover the creation of functions, the rules that they must follow, and how they relate to and communicate with the environments from which they are called.

User-defined functions are now one of the main building blocks for developing sophisticated software. R packages are nothing more than a collection of user-defined functions.

Recall that all R packages available on CRAN are freely downloadable, i.e., you can use/extend/include their code for your analysis.

Functions

A function has the general form:

```
name <- function(argument_1, argument_2, ...) {  
  expression_1  
  expression_2  
  <some other expressions>  
  return(output)  
}
```

- `argument_1`, `argument_2`, etc., are the names of variables.
- `expression_1`, `expression_2`, and `output` are all regular R expressions.
- `name` is the name of the function.

Note that some functions have no arguments: e.g.: `getwd()`.

Example: roots of a quadratic function

Assume we want to find the roots of:

$$a_2x^2 + a_1x + a_0 = 0.$$

The analytic solution is:

$$(x_1, x_2) = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2a_2}.$$

The associated code is:

```
rootfinder <- function(dA0, dA1, dA2) {
  vOut = numeric(2)
  vOut[1] = (-dA1 + sqrt(dA2^2 - 4 * dA2 * dA0))/(2 * dA2)
  vOut[2] = (-dA1 - sqrt(dA2^2 - 4 * dA2 * dA0))/(2 * dA2)
  return(vOut)
}
```

Do you think this to be a good strategy?

Example: roots of a quadratic function

Problems related to the efficiency of the code:

- We are computing algebraic operations too many times.
- We are not exploiting vector-operations

Problems related to the user usage. We are assuming that a_2 , a_1 and a_0 always imply the coded solution.

- What if $a_2 = a_1 = a_0 = 0$?
- What if $a_2 = a_1 = 0$?
- What if $a_2 = 0$?
- What if $a_2^2 - 4a_2a_0 = 0$?
- What if $a_2^2 - 4a_2a_0 < 0$?

We need to modify our code!

Example: roots of a quadratic function

Reformulate the analytical solution as:

$$(x_1, x_2) = \begin{cases} \mathbb{R}, & \text{if } a_2 = a_1 = a_0 = 0 \\ \emptyset, & \text{if } a_2 = a_1 = 0 \wedge a_0 \neq 0 \\ -a_0/a_1 & \text{if } a_2 = 0 \wedge a_1 \neq 0 \wedge a_0 \neq 0 \\ \frac{-a_1 \pm \sqrt{|\Delta|} \sqrt{\text{sgn}(\Delta)}}{2a_2}, & \text{otherwise.} \end{cases}$$

where $\Delta = a_2^2 - 4a_2a_0$.

Note that:

$$\sqrt{\text{sgn}(\Delta)} = \begin{cases} 1, & \text{if } \Delta \geq 0 \\ i, & \text{if } \Delta < 0. \end{cases}$$

```

rootfinder <- function(dA0, dA1, dA2) {
  if (dA2 == 0 && dA1 == 0 && dA0 == 0) {
    vRoots <- Inf
  } else if (dA2 == 0 && dA1 == 0) {
    vRoots <- NULL
  } else if (dA2 == 0) {
    vRoots <- -dA0/dA1
  } else {
    # calculate the discriminant
    dDelta <- dA1^2 - 4 * dA2 * dA0
    if (dDelta > 0) {
      vRoots <- (-dA1 + c(1, -1) * sqrt(dDelta))/(2*dA2)
    } else if (dDelta == 0) {
      vRoots <- rep(-dA1 / (2 * dA2), 2)
    } else {
      di <- complex(1, 0, 1)
      vRoots <- (-dA1 + c(1,-1) * sqrt(-dDelta) * di)/(2 * dA2)
    }
  }
  return(vRoots)
}

```

Example: roots of a quadratic function

Suppose we have saved the function `rootfinder` in the script `rootfinder.R` which is located in the `script` folder inside our working directory.

To use the function we first load it (using `source` or by copying and pasting into R), then call it, supplying suitable arguments.

```
> source("../scripts/rootfinder.r")
> rootfinder(1, 1, 0)
[1] -1 -1
>
> rootfinder(1, 0, -1)
[1] -1 1
>
> rootfinder(1, -2, 1)
[1] 1
>
> rootfinder(1, 1, 1)
[1] -0.5+0.8660254i -0.5-0.8660254i
```

Advantages of coding with functions

Once a function is loaded, it can be used again and again without having to reload it.

User-defined functions can be used in the same way as predefined functions are used in R. In particular they can be used within other functions.

The use of functions allows you to break down a programming task into smaller logical units.

Large programs are typically made up of a number of smaller functions, each of which does a simple well-defined task.

The mechanic of a function in R

When a function is executed the computer performs a series of operations:

- Set aside space in memory for the function.
- Make a copy of the function code.
- Transfer controls to the function.
- Run the function.
- Pass the output of the function back to the main program.
- Delete the copy of the function and all its variables.

Variables inside a function

Last point implies that arguments and variables that are defined within a function exist only within that function.

If you define and use a variable `x` inside a function, it does not exist outside the function.

```
> test <- function(x) {  
+ y <- x + 1  
+ return(y)  
+ }  
> test(1)  
[1] 2
```

```
> x  
Error: Object "x" not found  
> y  
Error: Object "y" not found
```

The scope of a variable

If variables with the same name exist inside and outside a function, then they are separate and do not interact at all.

You can think of a function as a separate environment that communicates with the outside world only through the values of its arguments and its output expression.

That part of a program in which a variable is defined is called its scope.

Restricting the scope of variables within a function provides an assurance that calling the function will not modify variables outside the function, except by assigning the returned value.

Beware, however, the scope of a variable is not symmetric!

Variables defined inside a function cannot be seen outside, but variables defined outside the function can be seen inside the function, **provided there is not a variable with the same name defined inside.**

The scope of a variable

Consider for example:

```
> test2 <- function(x) {  
+ y <- x + z  
+ return(y)  
+ }  
  
> z <- 1  
> test2(1)  
[1] 2  
  
> z <- 2  
> test2(1)  
[1] 3
```

Best practice is to always pass variables to be used inside the function as additional arguments:

```
> test2 <- function(x, z) {  
+ y <- x + z  
+ return(y)  
+ }
```

Arguments of a function

The argument of a function can be mandatory or optional depending on the function specification.

The arguments of an existing function can be obtained by calling the `formals` function.

```
> formals(matrix)
$data
[1] NA
$nrow
[1] 1
$ncol
[1] 1
$byrow
[1] FALSE
$dimnames
NULL
```

Default arguments

In order to simplify calling functions, some arguments may be assigned default values.

Default values are used in case the argument is not provided in the call to the function:

```
> test3 <- function(x = 1) {  
+   return(x)  
+ }  
> test3(2)  
[1] 2  
> test3()  
[1] 1
```

Default arguments: partial matching

Sometimes you will want to define arguments so that they can take only a small number of different values, and the function will stop informatively if an inappropriate value is passed.

When writing the function, we include a vector of the permissible values for any such argument, and then check them using the `match.arg` function. For example:

```
> funk <- function(vibe = c("Do", "Be", "Dooby", "Dooo")) {  
+   vibe <- match.arg(vibe)  
+   return(vibe)  
+ }
```

```
> funk("Dooby")  
[1] "Dooby"
```

```
> funk("Dum")
```

```
Error in match.arg(vibe) (from #2) :
```

```
'arg' should be one of "Do", "Be", "Dooby", "Dooo"
```

Ellipsis ...

R provides a very useful means of passing arguments, unaltered, from the function that is being called to the functions that are called within it.

These arguments do not need to be named explicitly in the outer function, hence providing great flexibility.

To use this facility you need to include ... in your argument list.

These three dots (an ellipsis) act as a placeholder for any extra arguments given to the function.

Ellipsis ...

Consider for example the function 'square of the mean':

```
> SquareMean <- function(vX, ...) {  
+   dSM <- mean(vX, ...) ^ 2  
+   return(dSM)  
+ }  
>  
> SquareMean(c(1, 3, 5, NA))  
[1] NA  
>  
> SquareMean(c(1, 3, 5, NA), na.rm = TRUE)  
[1] 9
```

The use of ellipsis dramatically increases the flexibility of the R programming language.

Vector-based programming using functions

R provides a family of powerful and flexible functions that make it easier for user-defined functions to handle vector inputs these belong to the `apply` family of functions.

Here we cover: `apply`, `sapply`, `lapply`, and `mapply`.

Other functions belonging to this family are: `tapply`, `vapply`, and `eapply`, which require a bit of advanced R programming knowledge.

sapply

The effect of `sapply(vX, FUN)` is to apply the function `FUN` to every element of vector `vX`. Consider the function, `f`, "sum of all integers lower than `X`":

```
f <- function(dX) {  
  if (dX < 0) {  
    stop("dX need to be positive.")  
  }  
  dSum = 0.0  
  iC   = 0  
  while (iC <= dX) {  
    dSum = dSum + iC  
    iC = iC + 1  
  }  
  return(dSum)  
}
```


sapply and lapply

Suppose we want to apply the function `f` to the following vector of observations `vX = c(1, 4, 9.478, 6, 75, 0.48)`. We can of course write a for loop:

```
> vX = c(1, 4, 9.478, 6, 75, 0.48)
> vSum = numeric(length(vX))
> for (i in 1:length(vSum)) {
+   vSum[i] = f(vX[i])
+ }
> vSum
[1]      1      10      45      21 2850       0
```

Or we can use the `sapply` function:

```
> vSum = sapply(vX, f)
> vSum
[1]      1      10      45      21 2850       0
```

`lapply` does the same of `sapply` but always returns a list.

mapply

`mapply` does the same of `sapply` but allows you to iterate over multiple arguments. Consider for example the function `rep` which takes arguments: `x` and `times`. `mapply` allows you to write:

```
> mapply(rep, times = 1:4, x = 4:1)
```

```
[[1]]
```

```
[1] 4
```

```
[[2]]
```

```
[1] 3 3
```

```
[[3]]
```

```
[1] 2 2 2
```

```
[[4]]
```

```
[1] 1 1 1 1
```

apply

`apply` allows you to evaluate a function, `FUN`, over the margins of an array. It's formula is:

```
apply(X, MARGIN, FUN, ...)
```

where:

- `X` is an array
- `MARGIN` is the index/indices of the array to which apply `FUN`
- `FUN` is the function to apply
- `...` are extra arguments for `FUN`

Note that `MARGIN` can be a vector of indices. If `X` is a *matrix* (an array of 2 dimensions), then `MARGINS = 1` indicates rows, while `MARGINS = 2` indicates columns.

apply example

To compute the cumulative sum over the columns of a matrix `mX` we write:

```
> mX = matrix(1:16, 4, 4)
> mX
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	5	9	13
[2,]	2	6	10	14
[3,]	3	7	11	15
[4,]	4	8	12	16

```
>
> apply(mX, 2, cumsum) # cumulative sum over columns
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	5	9	13
[2,]	3	11	19	27
[3,]	6	18	30	42
[4,]	10	26	42	58

Recursive programming

Many algorithms are recursive in nature. Consider for example the evaluation of $n!$, i.e., the factorial of the nonnegative integer n .

Obviously $n! = n(n-1)!$, such that we can write a function like:

```
> myfactorial <- function(iN) {  
+   if ((iN == 0) | (iN == 1)) {  
+     return(1)  
+   } else {  
+     return(iN * myfactorial(iN - 1))  
+   }  
+ }
```

```
> myfactorial(5)  
[1] 120
```

```
> factorial(5)  
[1] 120
```

Multiple Outputs

- A function can generate multiple output values
- Example: parameter estimates, their standard errors, and the optimized log-likelihood value
- A strategy in R is to create a `list` with all your outputs and return it as a single output of a function

For example:

```
l0ut = list(Par = vParam,  
            SE  = vStdErr,  
            LLK = dLogLik)  
  
return(l0ut)
```

Comments

- Remember, you often want to re-use a code
- It is often hard to read code that someone else has written, or code that you have written yourself months ago. Using sensible function and variable names, and Hungarian notation helps. . . somewhat
- Solution: **comment** your code!
- In R, `#` signals a comment to the end of the line:
`vBeta = c(0, 1, -5) # Initial values`

Comments

- It is best practice to declare what a script does at its begin.
- Furthermore, at the start of every function, put something like

```
##
## FunctionName(Inputs)
##
## Purpose:
##   Description of what the function does
##
## Input:
##   List of inputs, describing what they represent
##
## Output:
##   List of outputs, describing what they represent
##
## Return value:
##   List of return values, describing what they represent
##
```

In Rstudio to comment a selected portion of code we have the `Ctrl + alt + C` shortcut.

Comments

- In addition, add a general description of the program at the top of the .R file. This should include your name and the date, for future reference
- May seem like a lot of work. . . but it will pay off in the end
- So: comment a lot!