FRE7241 Algorithmic Portfolio Management

Lecture#1, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 6, 2022

NYU | TANDON SCHOOL OF ENGINEERING

Welcome Students!

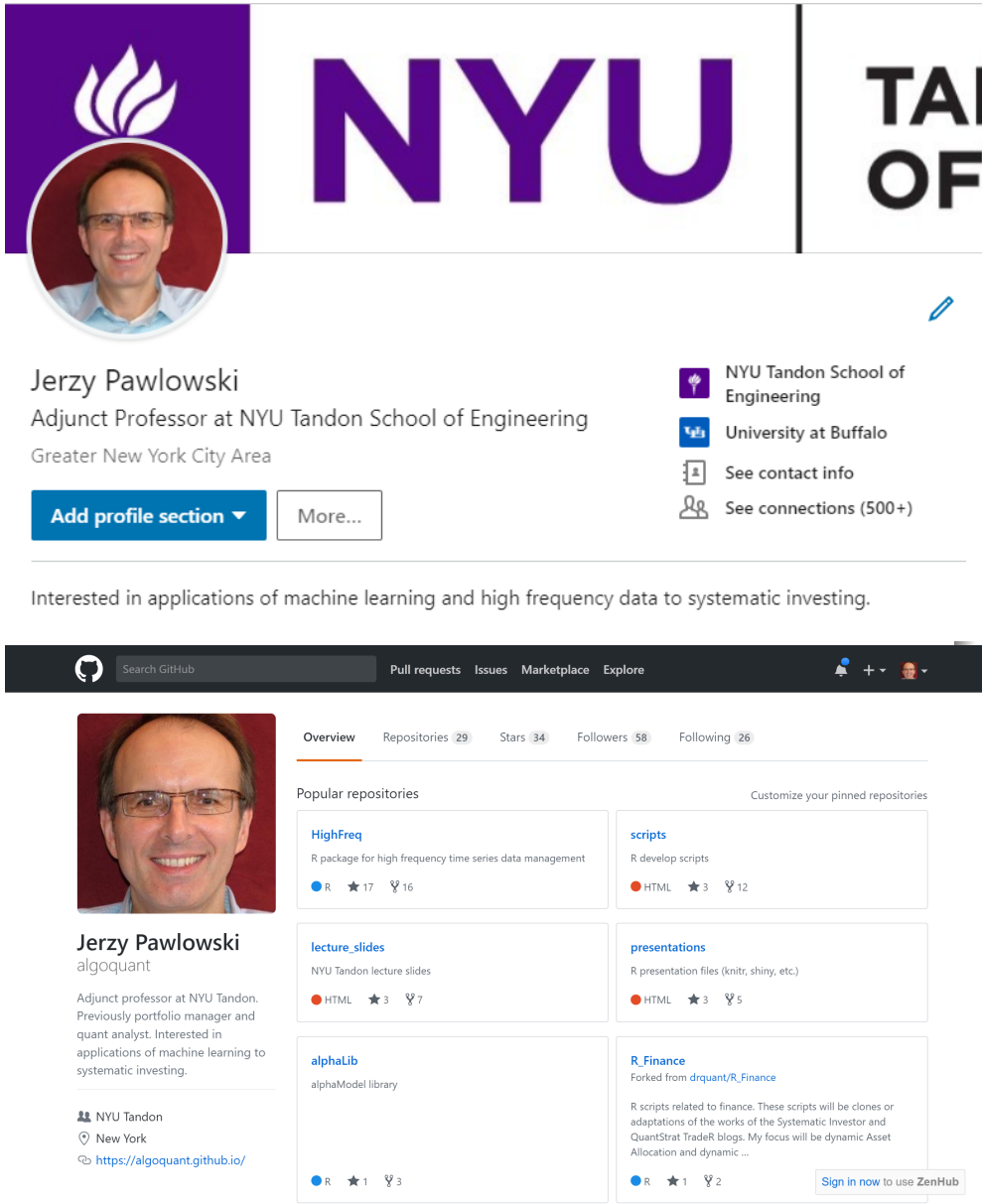
My name is Jerzy Pawlowski jp3900@nyu.edu

I'm an adjunct professor at NYU Tandon because I love teaching and I want to share my professional knowledge with young, enthusiastic students.

I'm interested in applications of *machine learning* to *systematic investing*.

I'm an advocate of *open-source software*, and I share it on GitHub:
My GitHub account

In my finance career, I have worked as a hedge fund *portfolio manager*, *CLO structurer* (banker), and *quant analyst*.
My LinkedIn profile



FRE7241 Course Description and Objectives

Course Description

The course will apply the `R` programming language to *trend following*, *momentum trading*, *statistical arbitrage* (pairs trading), and other active portfolio management strategies. The course will implement volatility and price *forecasting models*, asset pricing and *factor models*, and *portfolio optimization*. The course will apply *machine learning* techniques, such as *parameter regularization* (shrinkage), *bagging* and *backtesting* (cross-validation).

FRE7241 Course Description and Objectives

Course Description

The course will apply the R programming language to *trend following*, *momentum trading*, *statistical arbitrage* (pairs trading), and other active portfolio management strategies. The course will implement volatility and price *forecasting models*, asset pricing and *factor models*, and *portfolio optimization*. The course will apply *machine learning* techniques, such as *parameter regularization* (shrinkage), *bagging* and *backtesting* (cross-validation).

Course Objectives

Students will learn through R coding exercises how to:

- download data from external sources, and to scrub and format it.
- estimate time series parameters, and fit models such as *ARIMA*, *GARCH*, and factor models.
- optimize portfolios under different constraints and risk-return objectives.
- backtest active portfolio management strategies and evaluate their performance.

FRE7241 Course Description and Objectives

Course Description

The course will apply the `R` programming language to *trend following*, *momentum trading*, *statistical arbitrage* (pairs trading), and other active portfolio management strategies. The course will implement volatility and price *forecasting models*, asset pricing and *factor models*, and *portfolio optimization*. The course will apply *machine learning* techniques, such as *parameter regularization* (shrinkage), *bagging* and *backtesting* (cross-validation).

Course Objectives

Students will learn through `R` coding exercises how to:

- download data from external sources, and to scrub and format it.
- estimate time series parameters, and fit models such as *ARIMA*, *GARCH*, and factor models.
- optimize portfolios under different constraints and risk-return objectives.
- backtest active portfolio management strategies and evaluate their performance.

Course Prerequisites

FRE6123 Financial Risk Management and Asset Pricing. The `R` language is considered to be challenging, so this course requires programming experience with other languages such as `C++` or `Python`. Students with less programming experience are encouraged to first take *FRE6871 R in Finance*, and also *FRE6883 Financial Computing* by prof. Song Tang. Students should also have knowledge of basic statistics (random variables, estimators, hypothesis testing, regression, etc.)

Homeworks and Tests

Homeworks and Tests

Grading will be based on homeworks and tests. There will be no final exam.

The tests will require writing code, which should run directly when pasted into an `R` session, and should produce the required output, without any modifications.

Students will be allowed to consult lecture slides, and to copy code from them, and to copy from books or any online sources, but they will be required to provide references to those external sources (such as links or titles and page numbers).

The tests will be closely based on code contained in the lecture slides, so students are encouraged to become very familiar with those slides.

Students will submit their homework and test files only through *Brightspace* (not emails).

Students will be required to bring their laptop computers to class and run the `R` Interpreter, and the `RStudio` Integrated Development Environment (*IDE*), during the lecture.

Homeworks will also include reading assignments designed to help prepare for tests.

Homeworks and Tests

Homeworks and Tests

Grading will be based on homeworks and tests. There will be no final exam.

The tests will require writing code, which should run directly when pasted into an `R` session, and should produce the required output, without any modifications.

Students will be allowed to consult lecture slides, and to copy code from them, and to copy from books or any online sources, but they will be required to provide references to those external sources (such as links or titles and page numbers).

The tests will be closely based on code contained in the lecture slides, so students are encouraged to become very familiar with those slides.

Students will submit their homework and test files only through *Brightspace* (not emails).

Students will be required to bring their laptop computers to class and run the `R` Interpreter, and the `RStudio` Integrated Development Environment (*IDE*), during the lecture.

Homeworks will also include reading assignments designed to help prepare for tests.

Graduate Assistant

The graduate assistant (GA) will be Kaiyun Kang kk3567@nyu.edu.
The GA will answer questions during office hours, or via *Brightspace* forums, not via emails. Please send emails regarding lecture matters from *Brightspace* (not personal emails).

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

The tests will require mostly copying code samples from the lecture slides, making some modifications to them, and combining them with other code samples.

Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

So don't leave test assignments unanswered, and instead copy any code samples from the lecture slides that are related to the solution and make sense.

Contact the GA during office hours via text or phone, and submit questions to the GA or to me via *Brightspace*.

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

The tests will require mostly copying code samples from the lecture slides, making some modifications to them, and combining them with other code samples.

Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

So don't leave test assignments unanswered, and instead copy any code samples from the lecture slides that are related to the solution and make sense.

Contact the GA during office hours via text or phone, and submit questions to the GA or to me via *Brightspace*.

Please Submit *Minimal Working Examples* With Your Questions

When submitting questions, please provide a *minimal working example* that produces the error in `R`, with the following items:

- The *complete* `R` code that produces the error, including the seed value for random numbers,
- The version of `R` (output of command: `sessionInfo()`), and the versions of `R` packages,
- The type and version of your operating system (Windows or OSX),
- The dataset file used by the `R` code,
- The text or screenshots of error messages,

You can read more about producing *minimal working examples* here: http://stackoverflow.com/help/mcve
http://www.jaredknowles.com/journal/2013/5/27/writing-a-minimal-working-example-mwe-in-r

Course Grading Policies

Numerical Scores

Tests will be graded and assigned numerical scores. Each part of the tests will be graded separately and assigned a numerical score.
Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an `R` session, without any modifications. As long as the `R` code uses the required functions and produces the correct output, it will be given full credit.
Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

Course Grading Policies

Numerical Scores

Tests will be graded and assigned numerical scores. Each part of the tests will be graded separately and assigned a numerical score.
Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.
Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

Letter Grades

Letter grades for the course will be derived from the cumulative scores obtained for all the tests. Very high numerical scores close to the maximum won't guarantee an A letter grade, since grading will also depend on the difficulty of the assignments.

Course Grading Policies

Numerical Scores

Tests will be graded and assigned numerical scores. Each part of the tests will be graded separately and assigned a numerical score.
Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an `R` session, without any modifications. As long as the `R` code uses the required functions and produces the correct output, it will be given full credit.
Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

Letter Grades

Letter grades for the course will be derived from the cumulative scores obtained for all the tests. Very high numerical scores close to the maximum won't guarantee an A letter grade, since grading will also depend on the difficulty of the assignments.

Plagiarism

Plagiarism (copying from other students) and cheating will be punished.
But copying code from lecture slides, books, or any online sources is allowed and encouraged.
Students must provide references to any external sources from which they copy code (such as links or titles and page numbers).

FRE7241 Course Materials

Lecture Slides

The course will be mostly self-contained, using detailed lecture slides containing extensive, working `R` code examples.
The course will also utilize data and tutorials which are freely available on the internet.

FRE7241 Course Materials

Lecture Slides

The course will be mostly self-contained, using detailed lecture slides containing extensive, working R code examples.
The course will also utilize data and tutorials which are freely available on the internet.

FRE7241 Recommended Textbooks

- *Financial Data and Models Using R* by Clifford Ang, provides a good introduction to time series, portfolio optimization, and performance measures.
- *Systematic Trading* by Rob Carver, explains practical systematic trading rules.
- *Automated Trading* by Chris Conlan, explains how to implement a practical computer trading system.
- *Statistics and Data Analysis for Financial Engineering* by David Ruppert, introduces regression, cointegration, multivariate time series analysis, *ARIMA*, *GARCH*, *CAPM*, and factor models, with examples in R.
- *Financial Risk Modelling and Portfolio Optimization with R* by Bernhard Pfaff, introduces volatility models, portfolio optimization, and tactical asset allocation, with a great review of R packages and examples in R.

Many textbooks can be downloaded in electronic format from the *NYU Library*.

FRE7241 Supplementary Books

- *Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, introduces machine learning techniques using R, but without deep learning.
- *Quantitative Risk Management* by Alexander J. McNeil, Rudiger Frey, and Paul Embrechts: review of Value at Risk, factor models, ARMA and GARCH, extreme value theory, and credit risk models.
- *Applied Econometrics with R* by Christian Kleiber and Achim Zeileis, introduces advanced statistical models and econometrics.
- *The Art of R Programming* by Norman Matloff, contains a good introduction to R and to statistical models.
- *Advanced R* by Hadley Wickham, is the best book for learning the advanced features of R.
- *Numerical Recipes in C++* by William Press, Saul Teukolsky, William Vetterling, and Brian Flannery, is a great reference for linear algebra and numerical methods, implemented in working C++ code.
- The books *R in Action* by Robert Kabacoff and *R for Everyone* by Jared Lander, are good introductions to R and to statistical models.
- *Quant Finance books* by Jerzy Pawlowski.
- *Quant Trading books* by Jerzy Pawlowski.

FRE7241 Supplementary Materials

Robert Carver's trading blog

Great blog about practical systematic trading and investments, with Python code: http://qoppac.blogspot.com/

Introduction to Computational Finance with R

Good course by prof. Eric Zivot, with lots of R examples:
https://www.datacamp.com/community/open-courses/computational-finance-and-financial-econometrics-with-r

Notepad++ is a free source code editor for MS Windows, that supports several programming languages, including R.

Notepad++ has a very convenient and fast *search and replace* function, that allows *search and replace* in multiple files.
http://notepad-plus-plus.org/

Internal R Help and Documentation

The function `help()` displays documentation on a function or subject.

Preceding the keyword with a single `"?"` is equivalent to calling `help()`.

```
> # Display documentation on function "getwd"
> help(getwd)
> # Equivalent to "help(getwd)"
> ?getwd
```

The function `help.start()` displays a page with links to internal documentation.

R documentation is also available in RGui under the help tab.

The *pdf* files with R documentation are also available directly under:
C:/Program Files/R/R-3.1.2/doc/manual/
(the exact path will depend on the R version.)

```
> # Open the hypertext documentation
> help.start()
```

Introduction to R by Venables and R Core Team.

R Online Help and Documentation

R Cheat Sheets

Cheat Sheets are a fast way to find what you want
https://www.rstudio.com/resources/cheatsheets/

R Programming Wikibook

Wikibooks are crowdsourced textbooks
http://en.wikibooks.org/wiki/R_Programming/

R FAQ

Frequently Asked Questions about R
http://cran.r-project.org/doc/FAQ/R-FAQ.html

R-seek Online Search Tool

R-seek allows online searches specific to the R language
http://www.rseek.org/

R-help Mailing List

R-help is a very comprehensive Q&A mailing list
https://stat.ethz.ch/mailman/listinfo/r-help
R-help has archives of past Q&A - search it before you ask
https://stat.ethz.ch/pipermail/r-help/
GMANE allows searching the R-help archives using a usenet newsgroup style GUI

R Style Guides

DataCamp R style guide

The DataCamp R style guide is very close to what I have adopted:
DataCamp R style guide

Google R style guide

The Google R style guide is similar to DataCamp's:
Google R style guide

Stack Exchange

Stack Overflow

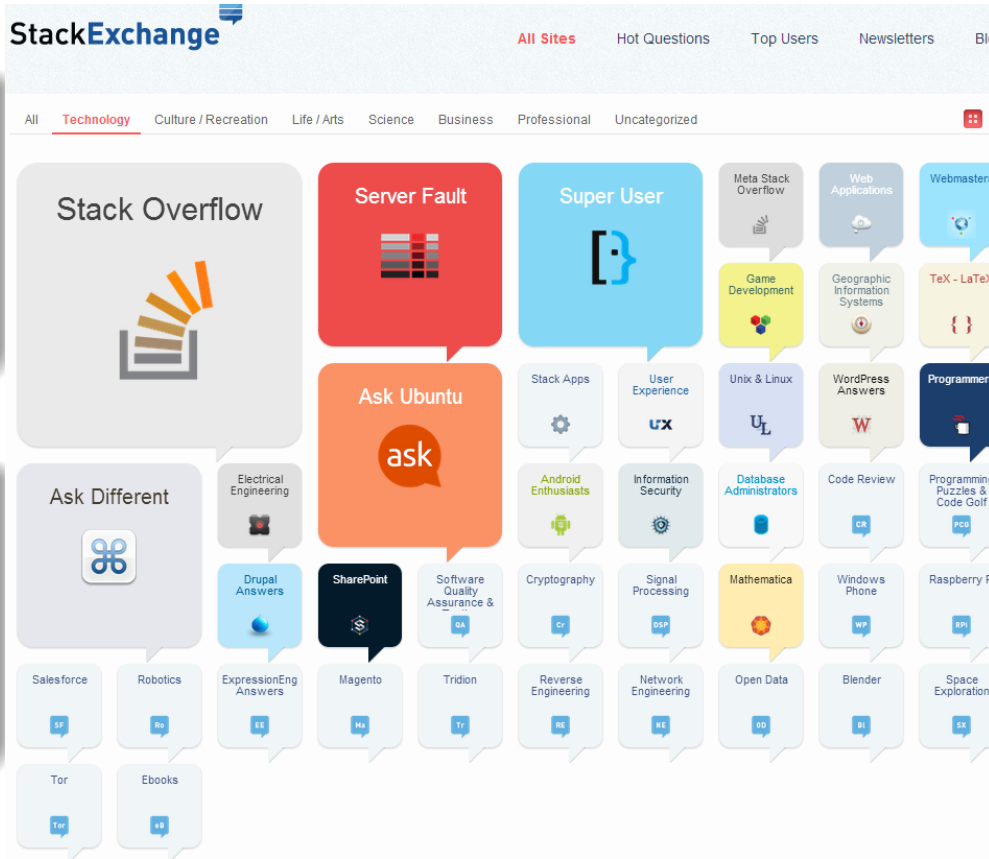
Stack Overflow is a Q&A forum for computer programming, and is part of Stack Exchange

- http://stackoverflow.com
- http://stackoverflow.com/questions/tagged/r
- http://stackoverflow.com/tags/r/info

Stack Exchange

Stack Exchange is a family of Q&A forums in a variety of fields

- http://stackexchange.com/
- http://stackexchange.com/sites#technology
- http://quant.stackexchange.com/



RStudio Support

RStudio has extensive online help, Q&A database, and documentation

https://support.rstudio.com/hc/en-us

https://support.rstudio.com/hc/en-us/sections/200107586-Using-RStudio

https://support.rstudio.com/hc/en-us/sections/200148796-Advanced-Topics

R Online Books and References

Hadley Wickham book *Advanced R*

The best book for learning the advanced features of R: http://adv-r.had.co.nz/

Cookbook for R by Winston Chang from *RStudio*

Good plotting, but not interactive: http://www.cookbook-r.com/

Efficient R programming by Colin Gillespie and Robin Lovelace

Good tips for fast R programming: https://csgillespie.github.io/efficientR/programming.html

Endmemo web book

Good, but not interactive: http://www.endmemo.com/program/R/

Quick-R by Robert Kabacoff

Good, but not interactive: http://www.statmethods.net/

R for Beginners by Emmanuel Paradis

Good, basic introduction to R: http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

R Online Interactive Courses

Datacamp Interactive Courses

Datacamp introduction to R: https://www.datacamp.com/courses/introduction-to-r/

Datacamp list of free courses: https://www.datacamp.com/community/open-courses

Datacamp basic statistics in R: https://www.datacamp.com/community/open-courses/basic-statistics

Datacamp computational finance in R:
https://www.datacamp.com/community/open-courses/computational-finance-and-financial-econometrics-with-r

Datacamp machine learning in R:
https://www.datacamp.com/community/open-courses/kaggle-r-tutorial-on-machine-learning

Try R

Interactive R tutorial, but rather basic: http://tryr.codeschool.com/

R Blogs and Experts

R-Bloggers

R-Bloggers is an aggregator of blogs dedicated to R
http://www.r-bloggers.com/
Tal Galili is the author of R-Bloggers and has his own excellent blog
http://www.r-statistics.com/

Dirk Eddelbuettel

Dirk is a *Top Answerer* for R questions on Stackoverflow, the author of the Rcpp package, and the CRAN Finance View
http://dirk.eddelbuettel.com/
http://dirk.eddelbuettel.com/code/
http://dirk.eddelbuettel.com/blog/
http://www.rinfinance.com/

Romain Frangois

Romain is an R Enthusiast and Rcpp Hero
http://romainfrancois.blog.free.fr/
http://romainfrancois.blog.free.fr/index.php?tag/graphgallery
http://blog.r-enthusiasts.com/

More R Blogs and Experts

Revolution Analytics Blog

R blog by Revolution Analytics software vendor
http://blog.revolutionanalytics.com/

RStudio Blog

R blog by *RStudio*
http://blog.rstudio.org/

GitHub for Hosting Software Projects Online

GitHub is an internet-based online service for hosting repositories of software projects.

GitHub provides version control using *git* (designed by Linus Torvalds).

Most R projects are now hosted on *GitHub*.

Google uses *GitHub* to host its *tensorflow* library for machine learning:

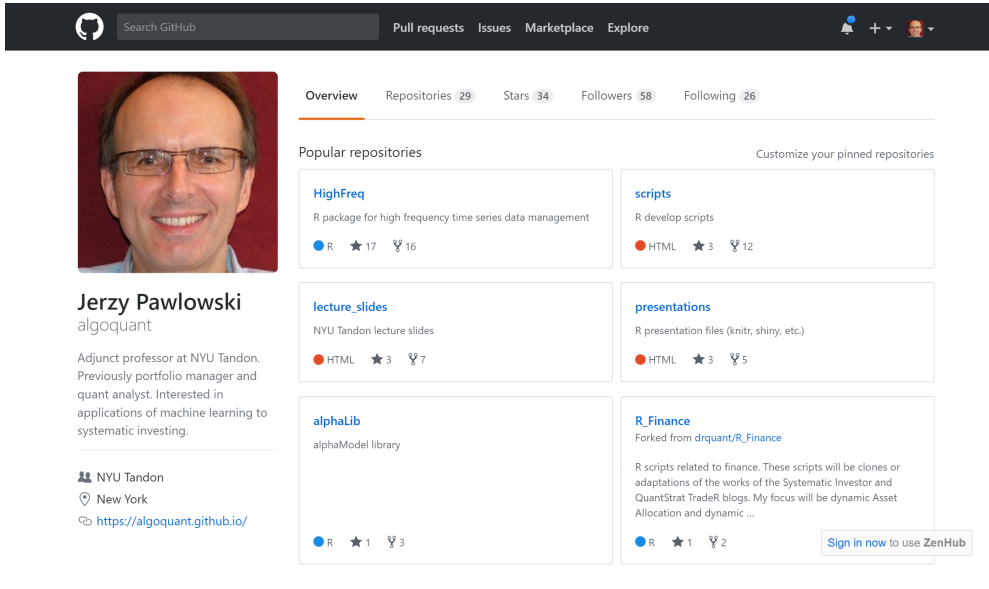
https://github.com/tensorflow/tensorflow

All the *FRE-7241* and *FRE-6871* lectures are hosted on *GitHub*:

https://github.com/algoquant/lecture_slides

https://github.com/algoquant

Hosting projects on *Google* is a great way to advertize your skills and network with experts.



What is R?

- An open-source software environment for statistical computing and graphics.
- An interpreted language, that allows interactive code development.
- A functional language where every operator is an R function.
- A very expressive language that can perform complex operations with very few lines of code.
- A language with metaprogramming facilities that allow programming on the language.
- A language written in C/C++, which can easily call other C/C++ programs.
- Can be easily extended with *packages* (function libraries), providing the latest developments like *Machine Learning*.
- Supports object-oriented programming with *classes* and *methods*.
- Vectorized functions written in C/C++, allow very fast execution of loops over vector elements.

Why is R More Difficult Than Other Languages?

R is more difficult than other languages because:

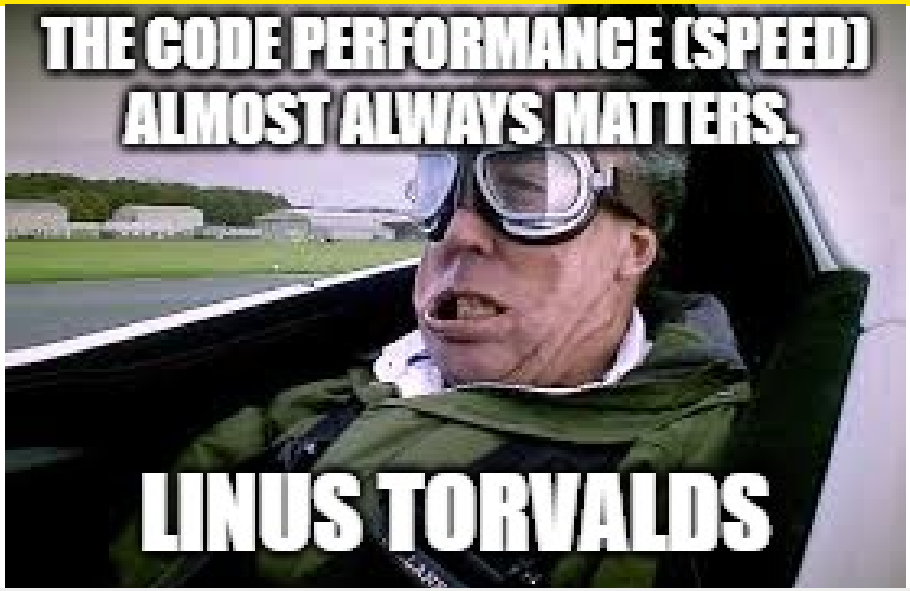
- R is a *functional* language, which makes its syntax unfamiliar to users of procedural languages like C/C++.
- The huge number of user-created *packages* makes it difficult to tell which are the best for particular applications.
- R can produce very cryptic *warning* and *error* messages, because it's a programming environment, so it performs many operations quietly, but those can sometimes fail.
- Fixing errors usually requires analyzing the complex structure of the R programming environment.

This course is designed to teach the most useful elements of R for financial analysis, through case studies and examples,

What are the Best Ways to Use R?

If used properly, R can be fast and interactive:

- Use R as an interface to libraries written in C++, Java, and JavaScript.
- Avoid using too many R function calls (every command in R is a function).
- Avoid using apply() and for() loops for large datasets.
- Use R functions which are *compiled* C++ code, instead of using interpreted R code.
- Use package *data.table* for high performance data management.
- Use package *shiny* for interactive charts of live models running in R.
- Use package *dygraphs* for interactive time series plots.
- Use package *knitr* for *RMarkdown* documents.
- Pre-allocate memory for new objects.
- Write C++ functions in *Rcpp* and *RcppArmadillo*.




```
> # Calculate cumulative sum of a vector
> vectorv <- runif(1e5)
> # Use compiled function
> cumsumv <- cumsum(vectorv)
> # Use for loop
> cumsumv2 <- vectorv
> for (i in 2:NROW(vectorv))
+   cumsumv2[i] <- (vectorv[i] + cumsumv2[i-1])
> # Compare the two methods
> all.equal(cumsumv, cumsumv2)
> # Microbenchmark the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   cumsum=cumsum(vectorv),
+   loop_alloc={
+     cumsumv2 <- vectorv
+     for (i in 2:NROW(vectorv))
+ cumsumv2[i] <- (vectorv[i] + cumsumv2[i-1])
+   },
+   loop_nalloc={
+     # Doesn't allocate memory to cumsumv3
```

The R License

R is open-source software released under the GNU General Public License:

http://www.r-project.org/Licenses

Some other R packages are released under the Creative Commons Attribution-ShareAlike License:

http://creativecommons.org

Installing R and *RStudio*

Students will be required to bring their laptop computers to all the lectures, and to run the R Interpreter and **RStudio** RStudio during the lecture.

Laptop computers will be necessary for following the lectures, and for performing tests.

Students will be required to install and to become proficient with the R Interpreter.
Students can download the R Interpreter from CRAN (Comprehensive R Archive Network):

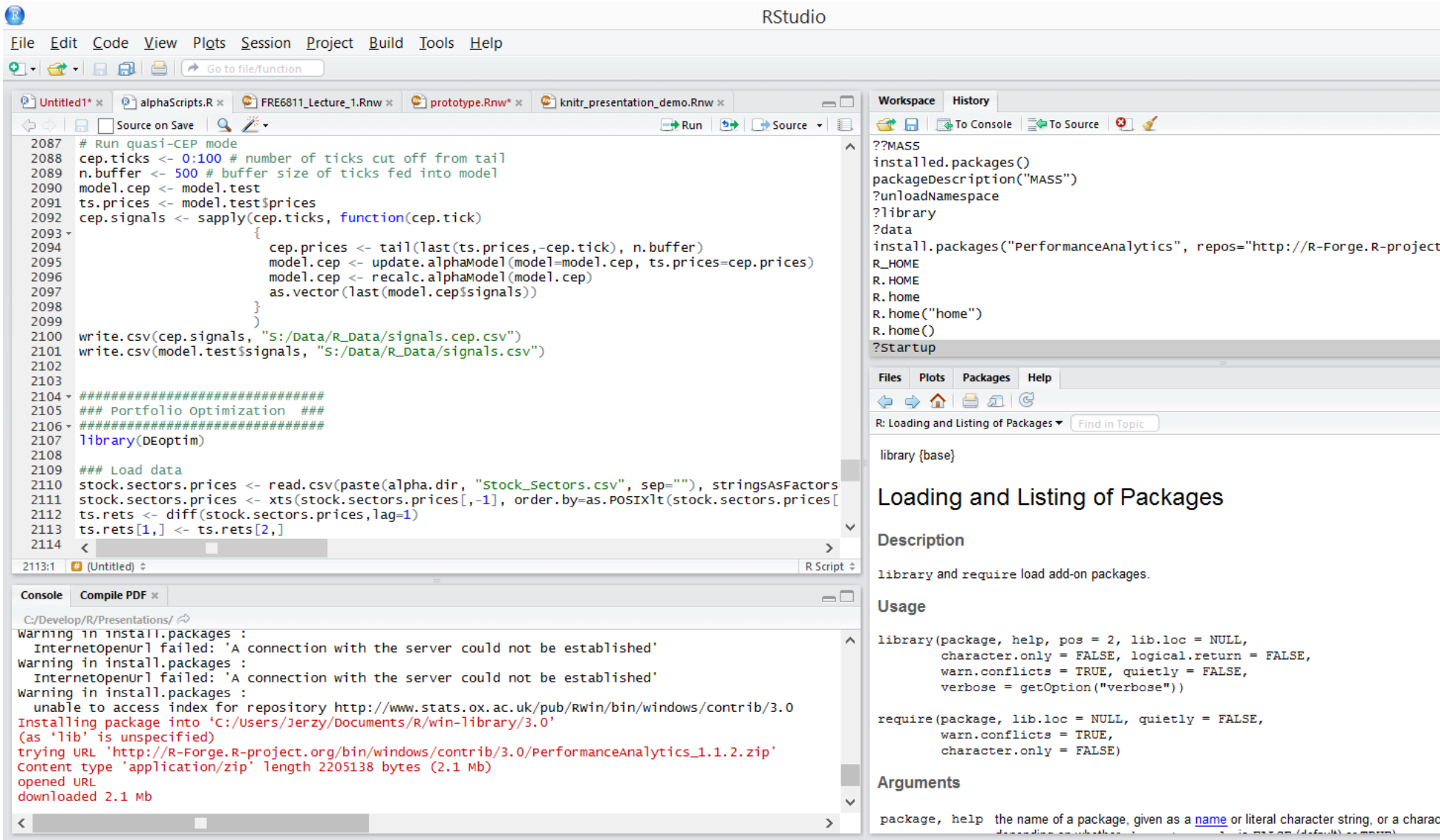
http://cran.r-project.org/

To invoke the RGui interface, click on:
C:/Program Files/R/R-3.1.2/bin/x64/RGui.exe

Students will be required to install and to become proficient with the *RStudio* Integrated Development Environment (*IDE*),

http://www.rstudio.com/products/rstudio/

Using *RStudio*



A First R Session

Variables are created by an assignment operation, and they don't have to be declared.

The standard assignment operator in R is the arrow symbol "<-".

R interretsp text in quotes ("") as character strings.

Text that is not in quotes ("") is interpreted as a *symbol* or *expression*.

Typing a *symbol* or *expression* evaluates it.

R uses the hash "#" sign to mark text as comments.

All text after the hash "#" sign is treated as a comment, and is not executed as code.

```
> # "<-" and "=" are valid assignment operators
> myvar <- 3
>
> # Typing a symbol or expression evaluates it
> myvar
[1] 3
>
> # Text in quotes is interpreted as a string
> myvar <- "Hello World!"
>
> # Typing a symbol or expression evaluates it
> myvar
[1] "Hello World!"
>
> myvar  # Text after hash is treated as comment
[1] "Hello World!"
```

Exploring an R Session

The function `getwd()` returns a vector of length 1, with the first element containing a string with the name of the current working directory (`cwd`).

The function `setwd()` accepts a character string as input (the name of the directory), and sets the working directory to that string.

R is a functional language, and R commands are functions, so they must be followed by parentheses "()".

```
> getwd()  # Get cwd
> setwd("/Users/jerzy/Develop/R")  # Set cwd
> getwd()  # Get cwd
```

Get system date and time

Just the date

```
> Sys.time()  # Get date and time
[1] "2022-09-06 16:26:51 EDT"
>
> Sys.Date()  # Get date only
[1] "2022-09-06"
```

The R Workspace

The workspace is the current R working environment, which includes all user-defined objects and the command history.

The function `ls()` returns names of objects in the R workspace.

The function `rm()` removes objects from the R workspace.

The workspace can be saved into and loaded back from an `.RData` file (compressed binary file format).

The function `save.image()` saves the whole workspace.

The function `save()` saves just the selected objects.

The function `load()` reads data from `.RData` files, and *invisibly* returns a vector of names of objects created in the workspace.

```
> var1 <- 3  # Define new object
> ls()  # List all objects in workspace
> # List objects starting with "v"
> ls(pattern=glob2rx("v*"))
> # Remove all objects starting with "v"
> rm(list=ls(pattern=glob2rx("v*")))
> save.image()  # Save workspace to file .RData in cwd
> rm(var1)  # Remove object
> ls()  # List objects
> load(".RData")
> ls()  # List objects
> var2 <- 5  # Define another object
> save(var1, var2,  # Save selected objects
+      file="/Users/jerzy/Develop/lecture_slides/data/my_data.RData"
> rm(list=ls())  # Remove all objects
> ls()  # List objects
> load_ed <- load(file="/Users/jerzy/Develop/lecture_slides/data/my_
> load_ed
> ls()  # List objects
```

The R Workspace (cont.)

When you quit R you'll be prompted "Save workspace image?"

If you answer *YES* then the workspace will be saved into the `.RData` file in the `cwd`.

When you start R again, the workspace will be automatically loaded from the existing `.RData` file.

```
>   q()  # quit R session
```

The function `history()` displays recent commands.

You can also save and load the command history from a file.

```
> history(5)  # Display last 5 commands
> savehistory(file="myfile")  # Default is ".Rhistory"
> loadhistory(file="myfile")  # Default is ".Rhistory"
```

R Session Info

The function `sessionInfo()` returns information about the current R session.

- `R version`,
- `OS platform`,
- `locale` settings,
- list of packages that are loaded and attached to the search path,
- list of packages that are loaded, but *not* attached to the search path,

```
> sessionInfo()  # Get R version and other session info
R version 4.2.1 (2022-06-23)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS Monterey 12.5.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources
LAPACK: /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] graphics  grDevices utils     datasets  stats     methods   base

other attached packages:
[1] knitr_1.40       HighFreq_0.1     rutils_0.2       dygraphs_1.1
[5] quantmod_0.4.20  TTR_0.24.3       xts_0.12.1       zoo_1.8-10

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.9         lattice_0.20-45   digest_0.6.29     grid_4.2
 [5] magrittr_2.0.3     evaluate_0.16     highr_0.9         stringi_1
 [9] rlang_1.0.5        cli_3.3.0         curl_4.3.2        rstudioap
[13] tools_4.2.1        stringr_1.4.0     htmlwidgets_1.5.4 xfun_0.32
[17] compiler_4.2.1     fastmap_1.1.0     htmltools_0.5.3
```

Environment Variables

R uses environment variables to store information about its environment, such as paths to directories containing files used by R (startup, history, OS).

For example the environment variables:

- R_USER and HOME store the R user Home directory,
- R_HOME stores the root directory of the R installation,

The functions Sys.getenv() and Sys.setenv() display and set the values environment variables.

Sys.getenv("env_var") displays the environment variable "env_var".

Sys.setenv("env_var=value") sets the environment variable "env_var" equal to "value".

```
> Sys.getenv()[5:7]  # List some environment variables
>
> Sys.getenv("HOME")  # Get R user HOME directory
>
> Sys.setenv(Home="/Users/jerzy/Develop/data")  # Set HOME directory
>
> Sys.getenv("HOME")  # Get user HOME directory
>
> Sys.getenv("R_HOME")  # Get R_HOME directory
>
> R.home()  # Get R_HOME directory
>
> R.home("etc")  # Get "etc" sub-directory of R_HOME
```

Global *Options* Settings

R uses a list of global *options* which affect how R computes and displays results.

The function `options()` either sets or displays the values of global *options*.

`options("globop")` displays the current value of option `"globop"`.

`getOption("globop")` displays the current value of option `"globop"`.

`options(globop=value)` sets the option `"globop"` equal to `"value"`.

```
> # ?options  # Long list of global options
> # Interpret strings as characters, not factors
> getOption("stringsAsFactors")  # Display option
> options("stringsAsFactors")  # Display option
> options(stringsAsFactors=FALSE)  # Set option
> # Number of digits printed for numeric values
> options(digits=3)
> # Control exponential scientific notation of print method
> # Positive "scipen" values bias towards fixed notation
> # Negative "scipen" values bias towards scientific notation
> options(scipen=100)
> # Maximum number of items printed to console
> options(max.print=30)
> # Warning levels options
> # Negative - warnings are ignored
> options(warn=-1)
> # zero - warnings are stored and printed after top-confl function
> options(warn=0)
> # One - warnings are printed as they occur
> options(warn=1)
> # 2 or larger - warnings are turned into errors
> options(warn=2)
> # Save all options in variable
> op_tions <- options()
> # Restore all options from variable
> options(op_tions)
```

Environments in R

Environments consist of a *frame* (a set of symbol-value pairs) and an *enclosure* (a pointer to an enclosing environment).

There are three system environments:

- `globalenv()` the user's workspace,
- `baseenv()` the environment of the base package,
- `emptyenv()` the only environment without an enclosure,

Environments form a tree structure of successive enclosures, with the empty environment at its root.

Packages have their own environments.

The enclosure of the base package is the empty environment.

```
> rm(list=ls())
> # Get base environment
> baseenv()
> # Get global environment
> globalenv()
> # Get current environment
> environment()
> # Get environment class
> class(environment())
> # Define variable in current environment
> globv <- 1
> # Get objects in current environment
> ls(environment())
> # Create new environment
> new_env <- new.env()
> # Get calling environment of new environment
> parent.env(new_env)
> # Assign Value to Name
> assign("new_var1", 3, envir=new_env)
> # Create object in new environment
> new_env$new_var2 <- 11
> # Get objects in new environment
> ls(new_env)
> # Get objects in current environment
> ls(environment())
> # Environments are subset like listv
> new_env$new_var1
> # Environments are subset like listv
> new_env[["new_var1"]]
```

The R Search Path

R evaluates variables using the search path, a series of environments:

- global environment,
- package environments,
- base environment,

The function `search()` returns the search path for R objects.

The function `attach()` attaches objects to the search path.

Using `attach()` allows referencing object components by their names alone, rather than as components of objects.

The function `detach()` detaches objects from the search path.

The function `find()` finds where objects are located on the search path.

Rule of Thumb

Be very careful with using `attach()`.

Make sure to `detach()` objects once they're not needed.

```
> search()  # Get search path for R objects
 [1] ".GlobalEnv"        "package:knitr"     "package:graphics"
 [4] "package:grDevices" "package:utils"     "package:datasets"
 [7] "package:HighFreq"  "package:rutils"    "package:dygraphs"
[10] "package:quantmod"  "package:TTR"       "package:xts"
[13] "package:zoo"       "package:stats"     "package:methods"
[16] "Autoloads"         "package:base"
> my_list <- list(flowers=c("rose", "daisy", "tulip"),
+         trees=c("pine", "oak", "maple"))
> my_list$trees
[1] "pine"  "oak"   "maple"
> attach(my_list)
> trees
[1] "pine"  "oak"   "maple"
> search()  # Get search path for R objects
 [1] ".GlobalEnv"        "my_list"           "package:knitr"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:HighFreq"  "package:rutils"
[10] "package:dygraphs"  "package:quantmod"  "package:TTR"
[13] "package:xts"       "package:zoo"       "package:stats"
[16] "package:methods"   "Autoloads"         "package:base"
> detach(my_list)
> head(trees)  # "trees" is in datasets base package
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
```

Extracting Time Series from Environments

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects extracted from an *environment*.

The extractor (accessor) functions from package *quantmod*: `Cl()`, `Vo()`, etc., extract columns from *OHLC* data.

A list of *xts* series can be flattened into a single *xts* series using the function `do.call()`.

The function `do.call()` executes a function call using a function name and a list of arguments.

`do.call()` passes the list elements individually, instead of passing the whole list as one argument.

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list.

Time series can also be extracted from an *environment* by coercing it into a `list`, and then subsetting and merging it into an *xts* series using the function `do.call()`.

```
> library(rutils)  # Load package rutils
> # Define ETF symbols
> symbolv <- c("VTI", "VEU", "IEF", "VNQ")
> # Extract symbolv from rutils::etfenv
> prices <- mget(symbolv, envir=rutils::etfenv)
> # prices is a list of xts series
> class(prices)
> class(prices[[1]])
> # Extract Close prices
> prices <- lapply(prices, quantmod::Cl)
> # Collapse list into time series the hard way
> xts1 <- cbind(prices[[1]], prices[[2]], prices[[3]], prices[[4]])
> class(xts1)
> dim(xts1)
> # Collapse list into time series using do.call()
> prices <- do.call(cbind, prices)
> all.equal(xts1, prices)
> class(prices)
> dim(prices)
> # Extract and cbind in single step
> prices <- do.call(cbind, lapply(
+   mget(symbolv, envir=rutils::etfenv), quantmod::Cl))
> # Or
> # Extract and bind all data, subset by symbolv
> prices <- lapply(symbolv, function(symbol) {
+     quantmod::Cl(get(symbol, envir=rutils::etfenv))
+ })  # end lapply
> # Same, but loop over etfenv without anonymous function
> prices <- do.call(cbind,
+   lapply(as.list(rutils::etfenv)[symbolv], quantmod::Cl))
> # Same, but works only for OHLC series - produces error
> prices <- do.call(cbind,
+   eapply(rutils::etfenv, quantmod::Cl)[symbolv])
```

Managing Time Series

Time series columns can be renamed, and then saved into `.csv` files.

The function `strsplit()` splits the elements of a character vector.

The package *zoo* contains functions `write.zoo()` and `read.zoo()` for writing and reading *zoo* time series from `.txt` and `.csv` files.

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list.

The function `assign()` assigns a value to an object in a specified *environment*, by referencing it using a character string (name).

The function `save()` writes objects to compressed binary `.RData` files.

```
> # Drop ".Close" from column names
> colnames(prices[, 1:4])
> do.call(rbind, strsplit(colnames(prices[, 1:4]), split="[.]"))[,
> colnames(prices) <- do.call(rbind, strsplit(colnames(prices), spli
> # Or
> colnames(prices) <- unname(sapply(colnames(prices),
+     function(colname) strsplit(colname, split="[.]")[[1]][1]))
> tail(prices, 3)
> # Which objects in global environment are class xts?
> unlist(eapply(globalenv(), is.xts))
> # Save xts to csv file
> write.zoo(prices,
+   file="/Users/jerzy/Develop/lecture_slides/data/etf_series.csv",
> # Copy prices into etfenv
> etfenv$etf_list <- etf_list
> # Or
> assign("prices", prices, envir=etfenv)
> # Save to .RData file
> save(etfenv, file="etf_data.RData")
```

Referencing Object Components Using with()

The function with() evaluates an expression in an environment constructed from the data.

with() allows referencing object components by their names alone.

It's often better to use with() instead of attach().

```
> # "trees" is in datasets base package
> head(trees, 3)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
> colnames(trees)
[1] "Girth"  "Height" "Volume"
> mean(Girth)

Error in mean(Girth):  object 'Girth' not found

> mean(trees$Girth)
[1] 13.2
> with(trees,
+      c(mean(Girth), mean(Height), mean(Volume)))
[1] 13.2 76.0 30.2
```

R Packages

Types of R Packages

R can run libraries of functions called packages,

R packages can can also contain data,

Most packages need to be *loaded* into R before they can be used,

R includes a number of base packages that are already installed and loaded,

There's also a special package called the base package, which is responsible for all the basic R functionality,

datasets is a base package containing various datasets, for example EuStockMarkets,

The *base* Packages

R includes a number of packages that are pre-installed (often called *base* packages),
Some *base* packages:

- *base* - basic R functionality,
- *stats* - statistical functions and random number generation,
- *graphics* - basic graphics,
- *utils* - utility functions,
- *datasets* - popular datasets,
- *parallel* - support for parallel computation,

Very popular packages:

- *MASS* - functions and datasets for "Modern Applied Statistics with S",
- *ggplot2* - grammar of graphics plots,
- *shiny* - interactive web graphics from R,
- *slidify* - HTML5 slide shows from R,
- *devtools* - create R packages,
- *roxygen2* - document R packages,
- *Rcpp* - integrate C++ code with R,
- *RcppArmadillo* - interface to Armadillo linear algebra library,
- *forecast* - linear models and forecasting,
- *tseries* - time series analysis and computational finance,
- *zoo* - time series and ordered objects,
- *xts* - advanced time series objects,
- *quantmod* - quantitative financial modeling framework,
- *caTools* - moving window statistics for graphics and time series objects,

CRAN Package Views

CRAN view for package *AER*:
http://cran.r-project.org/web/packages/AER/

Note:

- Authors,
- Version number,
- Reference manual,
- Vignettes,
- Dependencies on other packages.

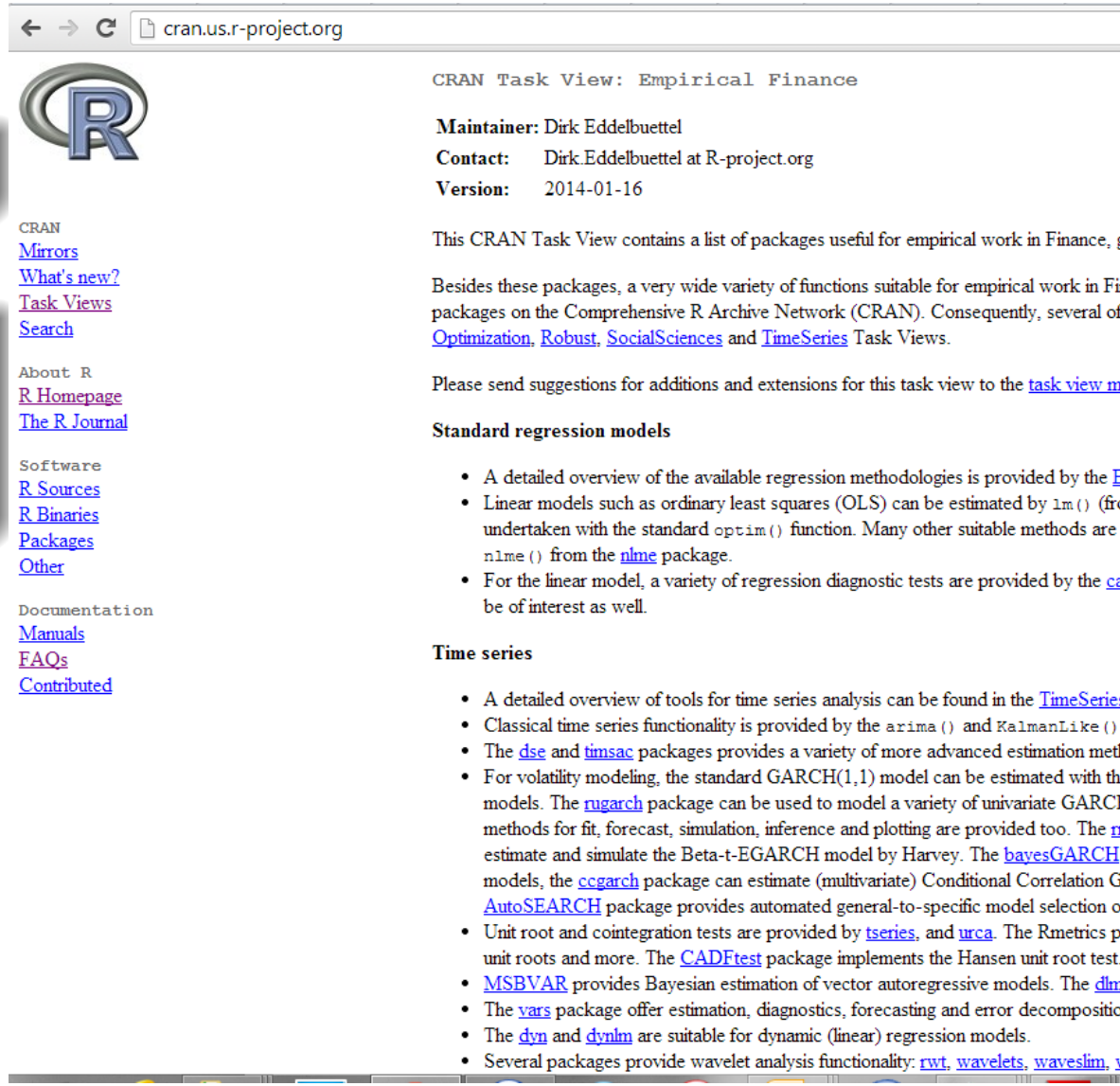
The package source code can be downloaded by clicking on the package source link,

CRAN Task Views

CRAN Finance Task View
http://cran.r-project.org//

Note:

- Maintainer,
- Topics,
- List of packages.



Installing Packages

Most packages need to be *installed* before they can be loaded and used.

Some packages like *MASS* are installed with base R (but not loaded).

Installing a package means downloading and saving its files to a local computer directory (hard disk), so they can be *loaded* by the R system.

The function `install.packages()` installs packages from the R command line.

Most widely used packages are available on the *CRAN* repository:

http://cran.r-project.org/web/packages/

Or on *R-Forge* or *GitHub*:

https://r-forge.r-project.org/

https://github.com/

Packages can also be installed in *RStudio* from the menu (go to Tools and then Install packages),

Packages residing on GitHub can be installed using the devtools packages.

```
> getOption("repos")  # get default package source
> .libPaths()  # get package save directory
> install.packages("AER")  # install "AER" from CRAN
> # install "PerformanceAnalytics" from R-Forge
> install.packages(
+   pkgs="PerformanceAnalytics",  # name
+   lib="C:/Users/Jerzy/Downloads",  # directory
+   repos="http://R-Forge.R-project.org")  # source
> # install devtools from CRAN
> install.packages("devtools")
> # load devtools
> library(devtools)
> # install package "babynamesv" from GitHub
> install_github(repo="hadley/babynamesv")
```

Installing Packages From Source

Sometimes packages aren't available in compiled form, so it's necessary to install them from their source code.

To install a package from source, the user needs to first install compilers and development tools:

For Windows install `Rtools`:

https://cran.r-project.org/bin/windows/Rtools/

For Mac OSX install `XCode` developer tools:

https://developer.apple.com/xcode/downloads/

The function `install.packages()` with argument `type="source"` installs a package from source.

The function `download.packages()` downloads the package's installation files (compressed tar format) to a local directory.

The function `install.packages()` can then be used to install the package from the downloaded files.

```
> # install package "PortfolioAnalytics" from source
> install.packages("PortfolioAnalytics",
+   type="source",
+   repos="http://r-forge.r-project.org")
> # download files for package "PortfolioAnalytics"
> download.packages(pkgs = "PortfolioAnalytics",
+   destdir = ".",  # download to cwd
+   type = "source",
+   repos="http://r-forge.r-project.org")
> # install "PortfolioAnalytics" from local tar source
> install.packages(
+   "C:/Users/Jerzy/Downloads/PortfolioAnalytics_0.9.3598.tar.gz",
+   repos=NULL, type="source")
```

Installed Packages

`defaultPackages` contains a list of packages loaded on startup by default.

The function `installed.packages()` returns a matrix of all packages installed on the system.

```
> getOption("defaultPackages")
> # matrix of installed package information
> pack_info <- installed.packages()
> dim(pack_info)
> # get all installed package names
> sort(unname(pack_info[, "Package"]))
> # get a few package names and their versions
> pack_info[sample(x=1:100, 5), c("Package", "Version")]
> # get info for package "xts"
> t(pack_info["xts", ])
```

Package Files and Directories

Package installation files are organized into multiple directories, including some of the following:

- ~/R containing R source code files,
- ~/src containing C++ and Fortran source code files,
- ~/data containing datasets,
- ~/man containing documentation files,

```
> # list directories in "PortfolioAnalytics" sub-directory
> gsub(
+   "C:/Users/Jerzy/Documents/R/win-library/3.1",
+   "~",
+   list.dirs(
+     file.path(
+       .libPaths()[1],
+       "PortfolioAnalytics")))
 [1] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [2] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [3] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [4] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [5] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [6] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [7] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [8] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
 [9] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
[10] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
[11] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
[12] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
[13] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
[14] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/l
```

Loading Packages

Most packages need to be *loaded* before they can be used in an R session.

Loading a package means attaching the package *namesvpace* to the *search path*, which allows R to call the package functions and data.

The functions `library()` and `require()` load packages, but in slightly different ways.

`library()` produces an *error* (halts execution) if the package can't be loaded.

`require()` returns `TRUE` if the package is loaded successfully, and `FALSE` otherwise.

Therefore `library()` is usually used in script files that might be sourced, while `require()` is used inside functions.

```
> # load package, produce error if can't be loaded
> library(MASS)
> # load package, return TRUE if loaded successfully
> require(MASS)
> # load quietly
> library(MASS, quietly=TRUE)
> # load without any messages
> suppressMessages(library(MASS))
> # remove package from search path
> detach(MASS)
> # install package if it can't be loaded successfully
> if (!require("xts")) install.packages("xts")
```

Referencing Package Objects

After a package is *loaded*, the package functions and data can be accessed by name.

Package objects can also be accessed without *loading* the package, by using the double-colon "::" reference operator.

For example, `TTR::VWAP()` references the function `VWAP()` from the package *TTR*.

This way users don't have to load the package *TTR* (with `library(TTR)`) to use functions from the package *TTR*.

Using the "::" operator displays the source of objects, and makes R code easier to analyze.

```
> # calculate VTI volume-weighted average price
> vwapv <- TTR::VWAP(
+   price=quantmod::Cl(rutils::etfenv$VTI),
+   volume=quantmod::Vo(rutils::etfenv$VTI), n=10)
```

Exploring Packages

The package *Ecdat* contains data sets for econometric analysis.

The data frame `Garch` contains daily currency prices.

The function `data()` loads external data or listv data sets in a package.

Some packages provide *lazy loading* of their data sets, which means they automatically load their data sets when they're needed (when they are called by some operation).

The package's data isn't loaded into `R` memory when the package is *loaded*, so it's not listed using `ls()`, but the package data is available without calling the function `data()`.

The function `data()` isn't required to load data sets that are set up for *lazy loading*.

```
> library()  # list all packages installed on the system
> search()  # list all loaded packages on search path
>
> # get documentation for package "Ecdat"
> packageDescription("Ecdat")  # get short description
> help(package="Ecdat")  # load help page
> library(Ecdat)  # load package "Ecdat"
> data(package="Ecdat")  # list all datasets in "Ecdat"
> ls("package:Ecdat")  # list all objects in "Ecdat"
> browseVignettes("Ecdat")  # view package vignette
> detach("package:Ecdat")  # remove Ecdat from search path
```

```
> library(Ecdat)  # load econometric data sets
> class(Garch)  # Garch is a data frame from "Ecdat"
> dim(Garch)  # daily currency prices
> head(Garch[, -2])  # col 'dm' is Deutsch Mark
> detach("package:Ecdat")  # remove Ecdat from search path
```

Package Namespaces

Package *namesvpaces*:

- Provide a mechanism for calling objects from a package,
- Hide functions and data internal to the package,
- Prevent naming conflicts between user and package names,

When a package is loaded using `library()` or `require()`, its *namesvpace* is attached to the search path.

```
> search()  # get search path for R objects
> library(MASS)  # load package "MASS"
> head(ls("package:MASS"))  # list some objects in "MASS"
> detach("package:MASS")  # remove "MASS" from search path
```

Package Namespaces and the Search Path

Packages may be loaded without their *namesvpace* being attached to the search path.

When packages are loaded, then packages they depend on are also loaded, but their *namesvpaces* aren't necessarily attached to the search path.

The function `loadedNamespaces()` listv all loaded *namesvpaces*, including those that aren't on the search path.

The function `search()` returns the current search path for `R` objects.

`search()` returns many package *namesvpaces*, but not all the loaded *namesvpaces*.

```
> loadedNamespaces()  # get names of loaded namespaces
>
> search()  # get search path for R objects
```

Not Attached Namespaces

The function `sessionInfo()` returns information about the current R session, including packages that are loaded, but *not attached* to the search path.

`sessionInfo()` listv those packages as "loaded via a *namesvpace* (and not attached)"

```
> # get session info,
> # including packages not attached to the search path
> sessionInfo()
```

Non-Visible Objects

Non-visible objects (variables or functions) are either:

- objects from *not attached namesvpaces*,
- objects *not exported* outside a package,

Objects from packages that aren't attached can be accessed using the double-colon "::" reference operator.

Objects that are *not exported* outside a package can be accessed using the triple-colon ":::" reference operator.

Colon operators automatically load the associated package.

Non-visible objects in namespaces often use the ".*" name syntax.

```
> plot.xts  # package xts isn't loaded and attached
> head(xts::plot.xts, 3)
> methods("cbind")  # get all methods for function "cbind"
> stats::cbind.ts  # cbind isn't exported from package stats
> stats:::cbind.ts  # view the non-visible function
> getAnywhere("cbind.ts")
> library(MASS)  # load package 'MASS'
> select  # code of primitive function from package 'MASS'
```

Exploring Namespaces and Non-Visible Objects

The function `getAnywhere()` displays information about `R` objects, including non-visible objects.

Objects referenced *within* packages have different search paths than other objects:
Their search path starts in the package *namesvpace*, then the global environment and then finally the regular search path.

This way references to objects from `within` a package are resolved to the package, and they're not masked by objects of the same name in other environments.

```
> getAnywhere("cbind.ts")
```

Benchmarking the Speed of R Code

The function `system.time()` calculates the execution time (in seconds) used to evaluate a given expression.

`system.time()` returns the *"user time"* (execution time of user instructions), the *"system time"* (execution time of operating system calls), and *"elapsed time"* (total execution time, including system latency waiting).

The function `microbenchmark()` from package `microbenchmark` calculates and compares the execution time of R expressions (in milliseconds), and is more accurate than `system.time()`.

The time it takes to execute an expression is not always the same, since it depends on the state of the processor, caching, etc.

`microbenchmark()` executes the expression many times, and returns the distribution of total execution times.

```
> library(microbenchmark)
> vectorv <- runif(1e6)
> # sqrt() and "^0.5" are the same
> all.equal(sqrt(vectorv), vectorv^0.5)
> # sqrt() is much faster than "^0.5"
> system.time(vectorv^0.5)
> microbenchmark(
+   power = vectorv^0.5,
+   sqrt = sqrt(vectorv),
+   times=10)
```

The "`times`" parameter is the number of times the expression is evaluated.

The choice of the "`times`" parameter is a tradeoff between the time it takes to run `microbenchmark()`, and the desired accuracy,

Using apply() Instead of for() and while() Loops

All the different R loops have similar speed, with apply() the fastest, then vapply(), lapply() and sapply() slightly slower, and for() loops the slowest.

More importantly, the apply() syntax is more readable and concise, and fits the functional language paradigm of R, so it's preferred over for() loops.

Both vapply() and lapply() are *compiled* (*primitive*) functions, and therefore can be faster than other apply() functions.

```
> # Calculate matrix of random data with 5,000 rows
> matrixv <- matrix(rnorm(10000), ncol=2)
> # Allocate memory for row sums
> rowsumv <- numeric(NROW(matrixv))
> summary(microbenchmark(
+   rowsums = rowSums(matrixv),  # end rowsumv
+   apply = apply(matrixv, 1, sum),  # end apply
+   lapply = lapply(1:NROW(matrixv), function(indeks)
+     sum(matrixv[indeks, ])),  # end lapply
+   vapply = vapply(1:NROW(matrixv), function(indeks)
+     sum(matrixv[indeks, ]),
+     FUN.VALUE = c(sum=0)),  # end vapply
+   sapply = sapply(1:NROW(matrixv), function(indeks)
+     sum(matrixv[indeks, ])),  # end sapply
+   forloop = for (i in 1:NROW(matrixv)) {
+     rowsumv[i] <- sum(matrixv[i,])
+   },  # end for
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Increasing Speed of Loops by Pre-allocating Memory

R performs automatic memory management as users assign values to objects.

R doesn't require allocating the full memory for vectors or listv, and allows appending new data to existing objects ("growing" them).

For example, R allows assigning a value to a vector element that doesn't exist yet.

This forces R to allocate additional memory for that element, which carries a small speed penalty.

But when data is appended to an object using the functions `c()`, `append()`, `cbind()`, or `rbind()`, then R allocates memory for the whole new object and copies all the existing values, which is very memory intensive and slow.

It is therefore preferable to pre-allocate memory for large objects before performing loops.

The function `numeric(k)` returns a numeric vector of zeros of length `k`, while `numeric(0)` returns an empty (zero length) numeric vector (not to be confused with a `NULL` object).

```
> vectorv <- rnorm(5000)
> summary(microbenchmark(
+ # Allocate full memory for cumulative sum
+   forloop = {cumsumv <- numeric(NROW(vectorv))
+     cumsumv[1] <- vectorv[1]
+     for (i in 2:NROW(vectorv)) {
+       cumsumv[i] <- cumsumv[i-1] + vectorv[i]
+     }},  # end for
+ # Allocate zero memory for cumulative sum
+   grow_vec = {cumsumv <- numeric(0)
+     cumsumv[1] <- vectorv[1]
+     for (i in 2:NROW(vectorv)) {
+ # Add new element to "cumsumv" ("grow" it)
+       cumsumv[i] <- cumsumv[i-1] + vectorv[i]
+     }},  # end for
+ # Allocate zero memory for cumulative sum
+   com_bine = {cumsumv <- numeric(0)
+     cumsumv[1] <- vectorv[1]
+     for (i in 2:NROW(vectorv)) {
+ # Add new element to "cumsumv" ("grow" it)
+       cumsumv <- c(cumsumv, vectorv[i])
+     }},  # end for
+   times=10))[, c(1, 4, 5)]
```

Vectorized Functions for Vector Computations

Vectorized functions accept `vectors` as their arguments, and return a vector of the same length as their value.

Many *vectorized* functions are also *compiled* (they pass their data to compiled C++ code), which makes them very fast.

The following *vectorized compiled* functions calculate cumulative values over large vectors:

- `cummax()`
- `cummin()`
- `cumsum()`
- `cumprod()`

Standard arithmetic operations (`"+"`, `"-"`, etc.) can be applied to `vectors`, and are implemented as *vectorized compiled* functions.

`ifelse()` and `which()` are *vectorized compiled* functions for logical operations.

But many *vectorized* functions perform their calculations in R code, and are therefore slow, but convenient to use.

```
> vector1 <- rnorm(1000000)
> vector2 <- rnorm(1000000)
> big_vector <- numeric(1000000)
> # Sum two vectors in two different ways
> summary(microbenchmark(
+   # Sum vectors using "for" loop
+   rloop = (for (i in 1:NROW(vector1)) {
+     big_vector[i] <- vector1[i] + vector2[i]
+   }),
+   # Sum vectors using vectorized "+"
+   vectorvized = (vector1 + vector2),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Allocate memory for cumulative sum
> cumsumv <- numeric(NROW(big_vector))
> cumsumv[1] <- big_vector[1]
> # Calculate cumulative sum in two different ways
> summary(microbenchmark(
+ # Cumulative sum using "for" loop
+   rloop = (for (i in 2:NROW(big_vector)) {
+     cumsumv[i] <- cumsumv[i-1] + big_vector[i]
+   }),
+ # Cumulative sum using "cumsum"
+   vectorvized = cumsum(big_vector),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Vectorized Functions for Matrix Computations

`apply()` loops are very inefficient for calculating statistics over rows and columns of very large matrices.

`R` has very fast *vectorized compiled* functions for calculating sums and means of rows and columns:

- `rowSums()`
- `colSums()`
- `rowMeans()`
- `colMeans()`

These *vectorized* functions are also *compiled* functions, so they're very fast because they pass their data to compiled `C++` code, which performs the loop calculations.

```
> # Calculate matrix of random data with 5,000 rows
> matrixv <- matrix(rnorm(10000), ncol=2)
> # Calculate row sums two different ways
> all.equal(rowSums(matrixv),
+   apply(matrixv, 1, sum))
> summary(microbenchmark(
+   rowsumv = rowSums(matrixv),
+   applyloop = apply(matrixv, 1, sum),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Fast R Code for Matrix Computations

The functions `pmax()` and `pmin()` calculate the "parallel" maxima (minima) of multiple vector arguments.

`pmax()` and `pmin()` return a vector, whose *n*-th element is equal to the maximum (minimum) of the *n*-th elements of the arguments, with shorter vectors recycled if necessary.

`pmax.int()` and `pmin.int()` are methods of generic functions `pmax()` and `pmin()`, designed for atomic vectors.

`pmax()` can be used to quickly calculate the maximum values of rows of a matrix, by first converting the matrix columns into a list, and then passing them to `pmax()`.

`pmax.int()` and `pmin.int()` are very fast because they are *compiled* functions (compiled from C++ code).

```
> library(microbenchmark)
> str(pmax)
> # Calculate row maximums two different ways
> summary(microbenchmark(
+   pmax=do.call(pmax.int,
+ lapply(seq_along(matrixv[1, ]),
+   function(indeks) matrixv[, indeks])),
+ applyloop=unlist(lapply(seq_along(matrixv[, 1]),
+   function(indeks) max(matrixv[indeks, ]))),
+   times=10))[, c(1, 4, 5)]
```

Package matrixStats for Fast Matrix Computations

The package *matrixStats* contains functions for calculating aggregations over matrix columns and rows, and other matrix computations, such as:

- estimating location and scale: `rowRanges()`, `colRanges()`, and `rowMaxs()`, `rowMins()`, etc.,
- testing and counting values: `colAnyMissings()`, `colAnys()`, etc.,
- cumulative functions: `colCumsums()`, `colCummins()`, etc.,
- binning and differencing: `binCounts()`, `colDiffs()`, etc.,

A summary of `matrixStats` functions can be found under:
https://cran.r-project.org/web/packages/matrixStats/vignettes/matrixStats-methods.html

The `matrixStats` functions are very fast because they are *compiled* functions (compiled from `C++` code).

```
> install.packages("matrixStats")  # Install package matrixStats
> library(matrixStats)  # Load package matrixStats
> # Calculate row min values three different ways
> summary(microbenchmark(
+   rowmins = rowMins(matrixv),
+   pmin =
+     do.call(pmin.int,
+       lapply(seq_along(matrixv[1, ]),
+              function(indeks)
+                matrixv[, indeks])),
+   as_dframe =
+     do.call(pmin.int,
+       as.data.frame.matrix(matrixv)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Package Rfast for Fast Matrix and Numerical Computations

The package *Rfast* contains functions for fast matrix and numerical computations, such as:

- `colMedians()` and `rowMedians()` for matrix column and row medians,
- `colCumSums()`, `colCumMins()` for cumulative sums and min/max,
- `eigen.sym()` for performing eigenvalue matrix decomposition,

The `Rfast` functions are very fast because they are *compiled* functions (compiled from `C++` code).

```
> install.packages("Rfast")  # Install package Rfast
> library(Rfast)  # Load package Rfast
> # Benchmark speed of calculating ranks
> vectorv <- 1e3
> all.equal(rank(vectorv), Rfast::Rank(vectorv))
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode = rank(vectorv),
+   Rfast = Rfast::Rank(vectorv),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Benchmark speed of calculating column medians
> matrixv <- matrix(1e4, nc=10)
> all.equal(matrixStats::colMedians(matrixv), Rfast::colMedians(matr
> summary(microbenchmark(
+   matrixStats = matrixStats::colMedians(matrixv),
+   Rfast = Rfast::colMedians(matrixv),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Writing Fast R Code Using Vectorized Operations

Writing Fast R Code Using Vectorized Operations

R-style code is code that relies on *vectorized compiled* functions, instead of `for()` loops.

`for()` loops in R are slow because they call functions multiple times, and individual function calls are compute-intensive and slow.

The brackets "[]" operator is a *vectorized compiled* function, and is therefore very fast.

Vectorized assignments using brackets "[]" and `Boolean` or `integer` vectors to subset vectors or matrices are therefore preferable to `for()` loops.

R code that uses *vectorized compiled* functions can be as fast as C++ code.

R-style code is also very *expressive*, i.e. it allows performing complex operations with very few lines of code.

```
> summary(microbenchmark(  # Assign values to vector three different
+ # Fast vectorized assignment loop performed in C using brackets "
+   brackets = {vectorv <- numeric(10)
+     vectorv[] <- 2},
+ # Slow because loop is performed in R
+   forloop = {vectorv <- numeric(10)
+     for (indeks in seq_along(vectorv))
+       vectorv[indeks] <- 2},
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> summary(microbenchmark(  # Assign values to vector two different v
+ # Fast vectorized assignment loop performed in C using brackets "
+   brackets = {vectorv <- numeric(10)
+     vectorv[4:7] <- rnorm(4)},
+ # Slow because loop is performed in R
+   forloop = {vectorv <- numeric(10)
+     for (indeks in 4:7)
+       vectorv[indeks] <- rnorm(1)},
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Vectorized Functions

Functions which use vectorized operations and functions are automatically *vectorized* themselves.

Functions which only call other compiled C++ vectorized functions, are also very fast.

But not all functions are vectorized, or they're not vectorized with respect to their *parameters*.

Some *vectorized* functions perform their calculations in R code, and are therefore slow, but convenient to use.

```
> # Define function vectorized automatically
> my_fun <- function(input, param) {
+   param*input
+ }  # end my_fun
> # "input" is vectorized
> my_fun(input=1:3, param=2)
> # "param" is vectorized
> my_fun(input=10, param=2:4)
> # Define vectors of parameters of rnorm()
> stdevs <- structure(1:3, names=paste0("sd=", 1:3))
> means <- structure(-1:1, names=paste0("mean=", -1:1))
> # "sd" argument of rnorm() isn't vectorized
> rnorm(1, sd=stdevs)
> # "mean" argument of rnorm() isn't vectorized
> rnorm(1, mean=means)
```

Performing sapply() Loops Over Function Parameters

Many functions aren't vectorized with respect to their *parameters*.

Performing sapply() loops over a function's parameters produces vector output.

```
> # Loop over stdevs produces vector output
> set.seed(1121)
> sapply(stdevs, function(stdev) rnorm(n=2, sd=stdev))
> # Same
> set.seed(1121)
> sapply(stdevs, rnorm, n=2, mean=0)
> # Loop over means
> set.seed(1121)
> sapply(means, function(me_an) rnorm(n=2, mean=me_an))
> # Same
> set.seed(1121)
> sapply(means, rnorm, n=2)
```

Creating Vectorized Functions

In order to *vectorize* a function with respect to one of its *parameters*, it's necessary to perform a loop over it.

The function `Vectorize()` performs an `apply()` loop over the arguments of a function, and returns a vectorized version of the function.

`Vectorize()` vectorizes the arguments passed to `"vectorize.args"`.

`Vectorize()` is an example of a *higher order* function: it accepts a function as its argument and returns a function as its value.

Functions that are vectorized using `Vectorize()` or `apply()` loops are just as slow as `apply()` loops, but convenient to use.

```
> # rnorm() vectorized with respect to "stdev"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     sapply(sd, rnorm, n=n, mean=mean)
+ }  # end vec_rnorm
> set.seed(1121)
> vec_rnorm(n=2, sd=stdevs)
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- Vectorize(FUN=rnorm,
+         vectorize.args=c("mean", "sd")
+ )  # end Vectorize
> set.seed(1121)
> vec_rnorm(n=2, sd=stdevs)
> set.seed(1121)
> vec_rnorm(n=2, mean=means)
```

The mapply() Functional

The mapply() functional is a multivariate version of sapply(), that allows calling a non-vectorized function in a vectorized way.

mapply() accepts a multivariate function passed to the "FUN" argument and any number of vector arguments passed to the dots "...".

mapply() calls "FUN" on the vectors passed to the dots "...", one element at a time:

$$mapply(FUN = fun, vec1, vec2, \ldots) = [fun(vec_{1,1}, vec_{2,1}, \ldots), \ldots, fun(vec_{1,i}, vec_{2,i}, \ldots), \ldots]$$

mapply() passes the first vector to the first argument of "FUN", the second vector to the second argument, etc.

The first element of the output vector is equal to "FUN" called on the first elements of the input vectors, the second element is "FUN" called on the second elements, etc.

```
> str(sum)
> # na.rm is bound by name
> mapply(sum, 6:9, c(5, NA, 3), 2:6, na.rm=TRUE)
> str(rnorm)
> # mapply vectorizes both arguments "mean" and "sd"
> mapply(rnorm, n=5, mean=means, sd=stdevs)
> mapply(function(input, e_xp) input^e_xp,
+  1:5, seq(from=1, by=0.2, length.out=5))
```

The output of mapply() is a vector of length equal to the longest vector passed to the dots "..." argument, with the elements of the other vectors recycled if necessary,

Vectorizing Functions Using mapply()

The mapply() functional is a multivariate version of sapply(), that allows calling a non-vectorized function in a vectorized way.

mapply() can be used to vectorize several function arguments simultaneously.

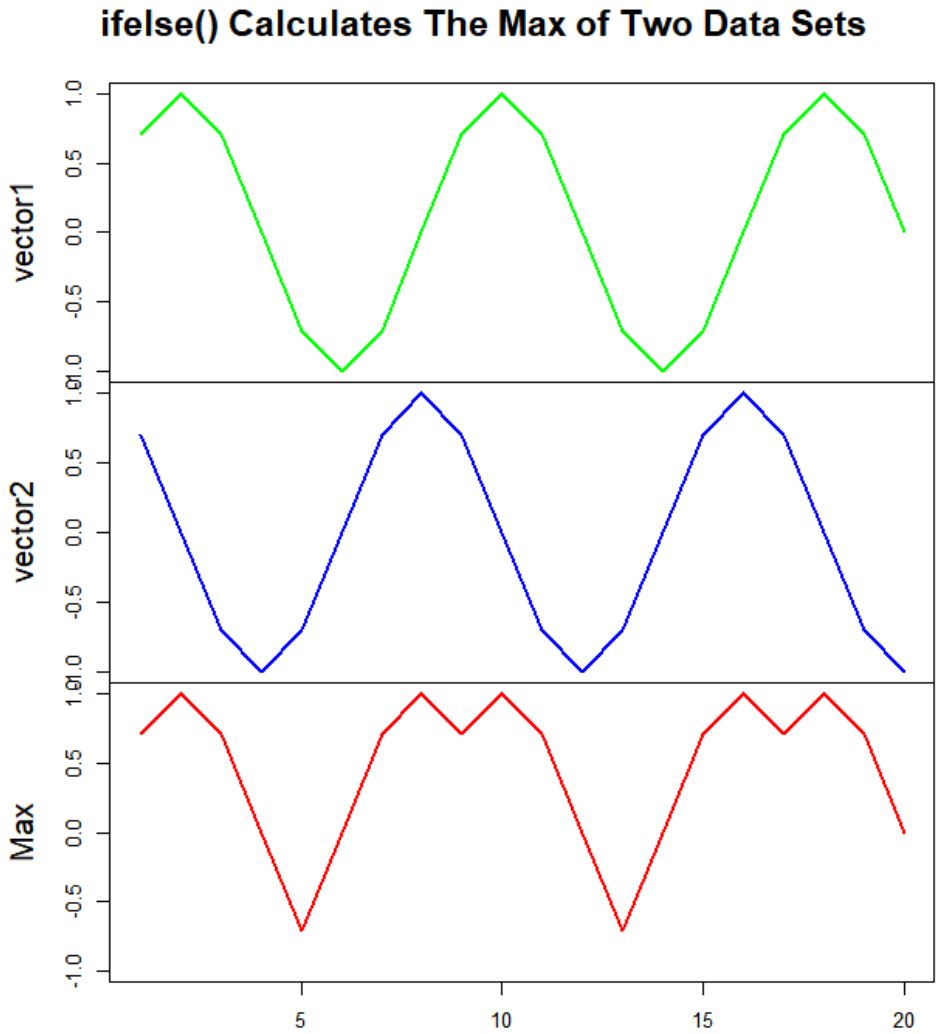
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2, sd=stdevs)
> # Call vec_rnorm() on vector of "mean"
> vec_rnorm(n=2, mean=means)
```

Vectorized if-else Statements Using Function ifelse()

The function `ifelse()` performs *vectorized* `if-else` statements on vectors.

`ifelse()` is much faster than performing an element-wise loop in R.

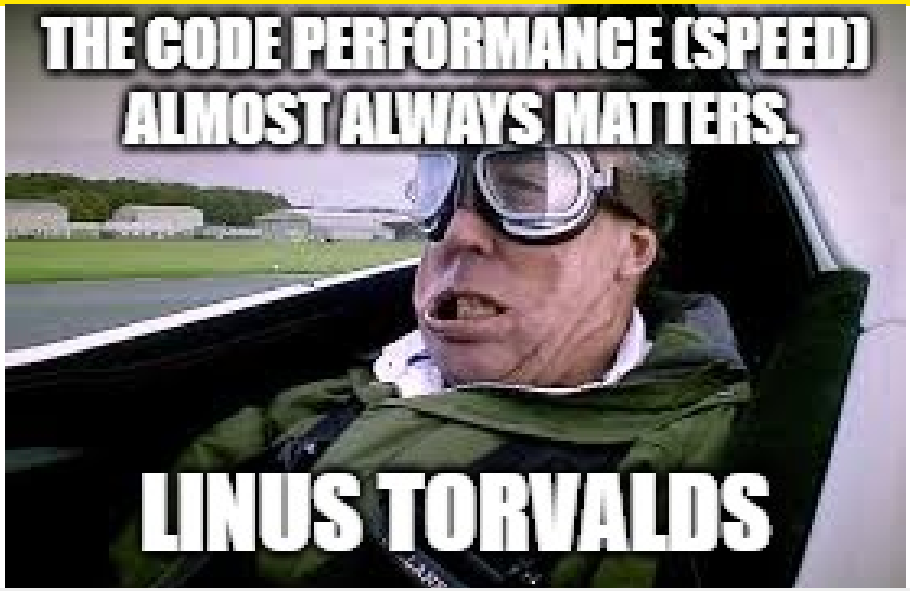
```
> # Create two numeric vectors
> vector1 <- sin(0.25*pi*1:20)
> vector2 <- cos(0.25*pi*1:20)
> # Create third vector using 'ifelse'
> vector3 <- ifelse(vector1 > vector2, vector1, vector2)
> # cbind all three together
> vector3 <- cbind(vector1, vector2, vector3)
> colnames(vector3)[3] <- "Max"
> # Set plotting parameters
> x11(width=6, height=7)
> par(oma=c(0, 1, 1, 1), mar=c(0, 2, 2, 1),
+     mgp=c(2, 1, 0), cex.lab=0.5, cex.axis=1.0, cex.main=1.8, cex.s
> # Plot matrix
> zoo::plot.zoo(vector3, lwd=2, ylim=c(-1, 1),
+   xlab="", col=c("green", "blue", "red"),
+   main="ifelse() Calculates The Max of Two Data Sets")
```



It's *Always* Important to Write Fast R Code

How to write fast R code:

- Avoid using `apply()` and `for()` loops for large datasets.
- Use R functions which are *compiled* C++ code, instead of using interpreted R code.
- Avoid using too many R function calls (every command in R is a function).
- Pre-allocate memory for new objects, instead of appending to them ("growing" them).
- Write C++ functions in *Rcpp* and *RcppArmadillo*.
- Use *function methods* directly instead of using *generic functions*.
- Create specialized functions by extracting only the essential R code from *function methods*.
- *Byte-compile* R functions using the *byte compiler* in package *compiler*.




```
> # Calculate cumulative sum of a vector
> vectorv <- runif(1e5)
> # Use compiled function
> cumsumv <- cumsum(vectorv)
> # Use for loop
> cumsumv2 <- vectorv
> for (i in 2:NROW(cumsumv2))
+   cumsumv2[i] <- (cumsumv2[i] + cumsumv2[i-1])
> # Compare the two methods
> all.equal(cumsumv, cumsumv2)
> # Microbenchmark the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   cumsum=cumsum(vectorv),
+   loop_alloc={
+     cumsumv2 <- vectorv
+     for (i in 2:NROW(cumsumv2))
+ cumsumv2[i] <- (cumsumv2[i] + cumsumv2[i-1])
+   },
+   loop_nalloc={
+     # Doesn't allocate memory to cumsumv3
```

Parallel Computing in R

Parallel Computing in R

Parallel computing means splitting a computing task into separate sub-tasks, and then simultaneously computing the sub-tasks on several computers or CPU cores.

There are many different packages that allow parallel computing in R, most importantly package *parallel*, and packages `foreach`, `doParallel`, and related packages:

http://cran.r-project.org/web/views/HighPerformanceComputing.html
http://blog.revolutionanalytics.com/high-performance-computing/
http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/

R Base Package *parallel*

The package *parallel* provides functions for parallel computing using multiple cores of CPUs,

The package *parallel* is part of the standard R distribution, so it doesn't need to be installed.

http://adv-r.had.co.nz/Profiling.html#parallelise
https://github.com/tobigithub/R-parallel/wiki/R-parallel-package-overview

Packages `foreach`, `doParallel`, and Related Packages

http://blog.revolutionanalytics.com/2015/10/updates-to-the-foreach-package-and-its-friends.html

Parallel Computing Using Package *parallel*

The package *parallel* provides functions for parallel computing using multiple cores of CPUs.

The package *parallel* is part of the standard R distribution, so it doesn't need to be installed.

Different functions from package *parallel* need to be called depending on the operating system (*Windows*, *Mac-OSX*, or *Linux*).

Parallel computing requires additional resources and time for distributing the computing tasks and collecting the output, which produces a computing overhead.

Therefore parallel computing can actually be slower for small computations, or for computations that can't be naturally separated into sub-tasks.

```
> library(parallel)  # Load package parallel
> # Get short description
> packageDescription("parallel")
> # Load help page
> help(package="parallel")
> # List all objects in "parallel"
> ls("package:parallel")
```

Performing Parallel Loops Using Package *parallel*

Some computing tasks naturally lend themselves to parallel computing, like for example performing loops.

Different functions from package *parallel* need to be called depending on the operating system (*Windows*, *Mac-OSX*, or *Linux*).

The function `mclapply()` performs loops (similar to `lapply()`) using parallel computing on several CPU cores under *Mac-OSX* or *Linux*.

Under *Windows*, a cluster of `R` processes (one per each CPU core) need to be started first, by calling the function `makeCluster()`.

Mac-OSX and *Linux* don't require calling the function `makeCluster()`.

The function `parLapply()` is similar to `lapply()`, and performs loops under *Windows* using parallel computing on several CPU cores.

```
> # Define function that pauses execution
> paws <- function(x, sleep_time=0.01) {
+   Sys.sleep(sleep_time)
+   x
+ }  # end paws
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> ncores <- detectCores() - 1
> # Initialize compute cluster under Windows
> cluster <- makeCluster(ncores)
> # Perform parallel loop under Windows
> outv <- parLapply(cluster, 1:10, paws)
> # Perform parallel loop under Mac-OSX or Linux
> outv <- mclapply(1:10, paws, mc.cores=ncores)
> library(microbenchmark)  # Load package microbenchmark
> # Compare speed of lapply versus parallel computing
> summary(microbenchmark(
+   standard = lapply(1:10, paws),
+   parallel = mclapply(1:10, paws, mc.cores=ncores),
+   times=10)
+ )[, c(1, 4, 5)]
```

Computing Advantage of Parallel Computing

Parallel computing provides an increasing advantage for larger number of loop iterations.

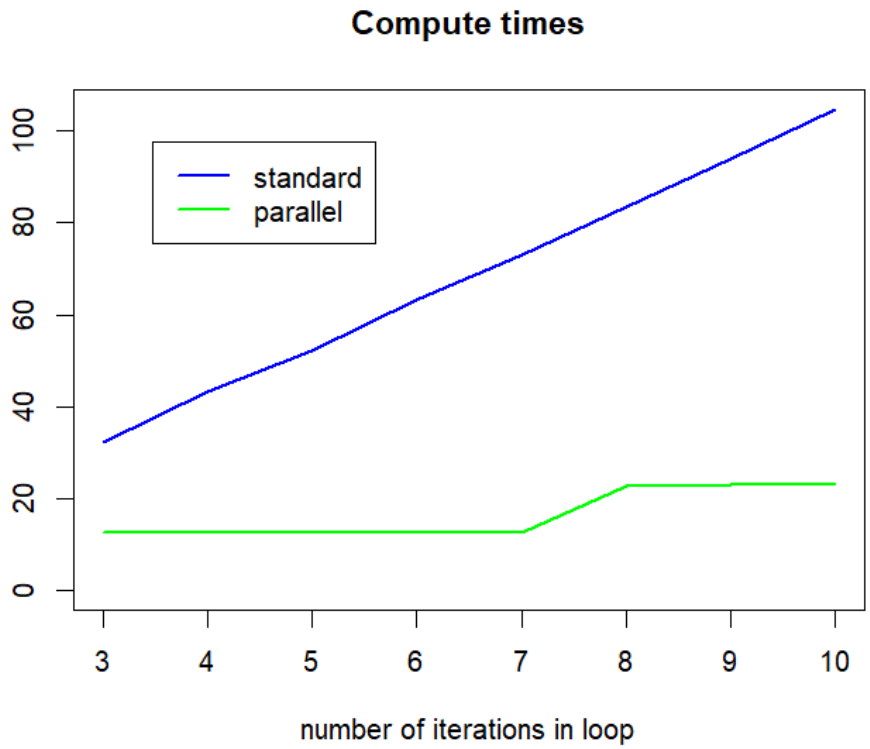
The function `stopCluster()` stops the `R` processes running on several CPU cores.

The function `plot()` by default plots a scatterplot, but can also plot lines using the argument `type="l"`.

The function `lines()` adds lines to a plot.

The function `legend()` adds a legend to a plot.

```
> # Compare speed of lapply with parallel computing
> iterations <- 3:10
> compute_times <- sapply(iterations,
+   function(max_iterations) {
+     summary(microbenchmark(
+ standard = lapply(1:max_iterations, paws),
+ parallel = parLapply(cluster, 1:max_iterations, paws),
+ times=10))[, 4]
+     })  # end sapply
> compute_times <- t(compute_times)
> colnames(compute_times) <- c("standard", "parallel")
> rownames(compute_times) <- iterations
> # Stop R processes over cluster under Windows
> stopCluster(cluster)
```




```
> x11(width=6, height=5)
> plot(x=rownames(compute_times),
+      y=compute_times[, "standard"],
+      type="l", lwd=2, col="blue",
+      main="Compute times",
+      xlab="number of iterations in loop", ylab="",
+      ylim=c(0, max(compute_times[, "standard"])))
> lines(x=rownames(compute_times),
+ y=compute_times[, "parallel"], lwd=2, col="green")
> legend(x="topleft", legend=colnames(compute_times),
+  inset=0.1, cex=1.0, bg="white",
+  lwd=2, lty=1, col=c("blue", "green"))
```

Parallel Computing Over Matrices

Very often we need to perform time consuming calculations over columns of matrices.

The function `parCapply()` performs an apply loop over columns of matrices using parallel computing on several CPU cores.

```
> # Calculate matrix of random data
> matrixv <- matrix(rnorm(1e5), ncol=100)
> # Define aggregation function over column of matrix
> aggfun <- function(column) {
+   output <- 0
+   for (indeks in 1:NROW(column))
+     output <- output + column[indeks]
+   output
+ }  # end aggfun
> # Perform parallel aggregations over columns of matrix
> aggs <- parCapply(cluster, matrixv, aggfun)
> # Compare speed of apply with parallel computing
> summary(microbenchmark(
+   applyloop=apply(matrixv, MARGIN=2, aggfun),
+   parapplyloop=parCapply(cluster, matrixv, aggfun),
+   times=10)
+ )[, c(1, 4, 5)]
> # Stop R processes over cluster under Windows
> stopCluster(cluster)
```

Initializing Parallel Clusters Under *Windows*

Under *Windows* the child processes in the parallel compute cluster don't inherit data and objects from their parent process.

Therefore the required data must be either passed into `parLapply()` via the dots `"..."` argument, or by calling the function `clusterExport()`.

Objects from packages must be either referenced using the double-colon operator `"::"`, or the packages must be loaded in the child processes.

```
> basep <- 2
> # Fails because child processes don't know basep:
> parLapply(cluster, 2:4,
+     function(exponent) basep^exponent)
> # basep passed to child via dots ... argument:
> parLapply(cluster, 2:4,
+     function(exponent, basep) basep^exponent,
+     basep=basep)
> # basep passed to child via clusterExport:
> clusterExport(cluster, "basep")
> parLapply(cluster, 2:4,
+     function(exponent) basep^exponent)
> # Fails because child processes don't know zoo::index():
> parSapply(cluster, c("VTI", "IEF", "DBC"),
+     function(symbol)
+       NROW(zoo::index(get(symbol, envir=rutils::etfenv))))
> # zoo function referenced using "::" in child process:
> parSapply(cluster, c("VTI", "IEF", "DBC"),
+     function(symbol)
+       NROW(zoo::index(get(symbol, envir=rutils::etfenv))))
> # Package zoo loaded in child process:
> parSapply(cluster, c("VTI", "IEF", "DBC"),
+     function(symbol) {
+       stopifnot("package:zoo" %in% search() || require("zoo", quie
+       NROW(zoo::index(get(symbol, envir=rutils::etfenv)))
+     })  # end parSapply
> # Stop R processes over cluster under Windows
> stopCluster(cluster)
```

Reproducible Parallel Simulations Under *Windows*

Simulations use pseudo-random number generators, and in order to perform reproducible results, they must set the *seed* value, so that the number generators produce the same sequence of pseudo-random numbers.

The function `set.seed()` initializes the random number generator by specifying the *seed* value, so that the number generator produces the same sequence of numbers for a given *seed* value.

But under *Windows* `set.seed()` doesn't initialize the random number generators of child processes, and they don't produce the same sequence of numbers.

The function `clusterSetRNGStream()` initializes the random number generators of child processes under *Windows*.

The function `set.seed()` does initialize the random number generators of child processes under *Mac-OSX* and *Linux*.

```
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> ncores <- detectCores() - 1
> # Initialize compute cluster under Windows
> cluster <- makeCluster(ncores)
> # Set seed for cluster under Windows
> # Doesn't work: set.seed(1121)
> clusterSetRNGStream(cluster, 1121)
> # Perform parallel loop under Windows
> output <- parLapply(cluster, 1:70, rnorm, n=100)
> sum(unlist(output))
> # Stop R processes over cluster under Windows
> stopCluster(cluster)
> # Perform parallel loop under Mac-OSX or Linux
> output <- mclapply(1:10, rnorm, mc.cores=ncores, n=100)
```

Monte Carlo Simulation

Monte Carlo simulation consists of generating random samples from a given probability distribution.

The *Monte Carlo* data samples can then used to calculate different parameters of the probability distribution (moments, quantiles, etc.), and its functionals.

The *quantile* of a probability distribution is the value of the *random variable* `x`, such that the probability of values less than `x` is equal to the given *probability* p.

The *quantile* of a data sample can be calculated by first sorting the sample, and then finding the value corresponding closest to the given *probability* p.

The function `quantile()` calculates the sample quantiles. It uses interpolation to improve the accuracy. Information about the different interpolation methods can be found by typing `?quantile`.

The function `sort()` returns a vector sorted into ascending order.

```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> nrows <- 1000
> datav <- rnorm(nrows)
> # Sample mean - MC estimate
> mean(datav)
> # Sample standard deviation - MC estimate
> sd(datav)
> # Monte Carlo estimate of cumulative probability
> pnorm(-2)
> sum(datav < (-2))/nrows
> # Monte Carlo estimate of quantile
> confl <- 0.02
> qnorm(confl)  # Exact value
> cutoff <- confl*nrows
> datav <- sort(datav)
> datav[cutoff]  # Naive Monte Carlo value
> quantile(datav, probs=confl)
> # Analyze the source code of quantile()
> stats:::quantile.default
> # Microbenchmark quantile
> library(microbenchmark)
> summary(microbenchmark(
+   monte_carlo = datav[cutoff],
+   quantilev = quantile(datav, probs=confl),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

Standard Errors of Estimators Using Bootstrap Simulation

The *bootstrap* procedure uses *Monte Carlo* simulation to generate a distribution of estimator values.

The *bootstrap* procedure generates new data by randomly sampling with replacement from the observed (empirical) data set.

If the original data consists of simulated random numbers then we simply simulate another set of these random numbers.

The *bootstrapped* datasets are used to recalculate the estimator many times, to provide a distribution of the estimator and its standard error.

```
> # Sample from Standard Normal Distribution
> nrows <- 1000; datav <- rnorm(nrows)
> # Sample mean and standard deviation
> mean(datav); sd(datav)
> # Bootstrap of sample mean and median
> nboot <- 10000
> bootd <- sapply(1:nboot, function(x) {
+   # Sample from Standard Normal Distribution
+   samplev <- rnorm(nrows)
+   c(mean=mean(samplev), median=median(samplev))
+ })  # end sapply
> bootd[, 1:3]
> bootd <- t(bootd)
> # Standard error from formula
> sd(datav)/sqrt(nrows)
> # Standard error of mean from bootstrap
> sd(bootd[, "mean"])
> # Standard error of median from bootstrap
> sd(bootd[, "median"])
```

The Distribution of Estimators Using Bootstrap Simulation

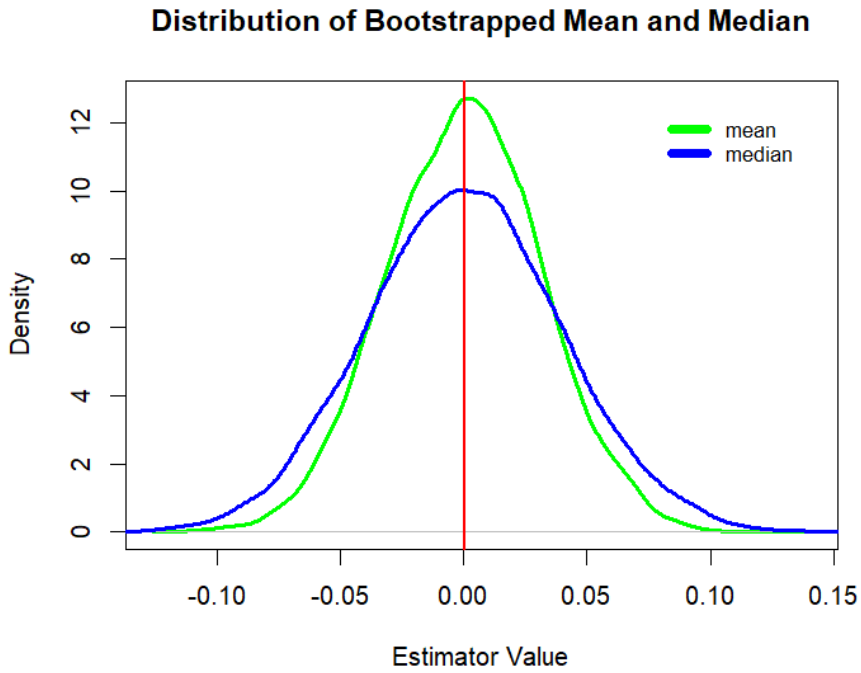
The standard errors of estimators can be calculated using a *bootstrap* simulation.

The *bootstrap* procedure generates new data by randomly sampling with replacement from the observed (empirical) data set.

The *bootstrapped* dataset is used to recalculate the estimator many times.

The *bootstrapped* estimator values are then used to calculate the probability distribution of the estimator and its standard error.

The function `density()` calculates a kernel estimate of the probability density for a sample of data.




```
> # Plot the densities of the bootstrap data
> x11(width=6, height=5)
> plot(density(bootd[, "mean"]), lwd=3, xlab="Estimator Value",
+      main="Distribution of Bootstrapped Mean and Median", col="gre
> lines(density(bootd[, "median"]), lwd=3, col="blue")
> abline(v=mean(bootd[, "mean"]), lwd=2, col="red")
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("mean", "median"), bty="n",
+  lwd=6, bg="white", col=c("green", "blue"))
```

Bootstrapping Using Vectorized Operations

Bootstrap simulations can be accelerated by using vectorized operations instead of R loops.

But using vectorized operations requires calculating a matrix of random data, instead of calculating random vectors in a loop.

This is another example of the tradeoff between speed and memory usage in simulations.

Faster code often requires more memory than slower code.

```
> set.seed(1121)  # Reset random number generator
> nrows <- 1000
> # Bootstrap of sample mean and median
> nboot <- 100
> bootd <- sapply(1:nboot, function(x) median(rnorm(nrows)))
> # Perform vectorized bootstrap
> set.seed(1121)  # Reset random number generator
> # Calculate matrix of random data
> samplev <- matrix(rnorm(nboot*nrows), ncol=nboot)
> boot_vec <- Rfast::colMedians(samplev)
> all.equal(bootd, boot_vec)
> # Compare speed of loops with vectorized R code
> library(microbenchmark)
> summary(microbenchmark(
+   loop = sapply(1:nboot, function(x) median(rnorm(nrows))),
+   cpp = {
+     samplev <- matrix(rnorm(nboot*nrows), ncol=nboot)
+     Rfast::colMedians(samplev)
+     },
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Bootstrapping Standard Errors Using Parallel Computing

The *bootstrap* procedure performs a loop, which naturally lends itself to parallel computing.

Different functions from package *parallel* need to be called depending on the operating system (*Windows*, *Mac-OSX*, or *Linux*).

The function `makeCluster()` starts running `R` processes on several CPU cores under *Windows*.

The function `parLapply()` is similar to `lapply()`, and performs loops under *Windows* using parallel computing on several CPU cores.

The `R` processes started by `makeCluster()` don't inherit any data from the parent `R` process.

Therefore the required data must be either passed into `parLapply()` via the dots `"..."` argument, or by calling the function `clusterExport()`.

The function `mclapply()` performs loops using parallel computing on several CPU cores under *Mac-OSX* or *Linux*.

The function `stopCluster()` stops the `R` processes running on several CPU cores.

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster under
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> nrows <- 1000
> # Bootstrap mean and median under Windows
> nboot <- 10000
> bootd <- parLapply(cluster, 1:nboot,
+   function(x, datav, nrows) {
+   samplev <- rnorm(nrows)
+   c(mean=mean(samplev), median=median(samplev))
+   }, datav=datav, nrows*nrows)  # end parLapply
> # Bootstrap mean and median under Mac-OSX or Linux
> bootd <- mclapply(1:nboot,
+   function(x) {
+   samplev <- rnorm(nrows)
+   c(mean=mean(samplev), median=median(samplev))
+   }, mc.cores=ncores)  # end mclapply
> bootd <- rutils::do_call(rbind, bootd)
> # Means and standard errors from bootstrap
> apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
> # Standard error from formula
> sd(datav)/sqrt(nrows)
> stopCluster(cluster)  # Stop R processes over cluster under Windo
```

Parallel Bootstrapping of the *Median Absolute Deviation*

The *Median Absolute Deviation* (*MAD*) is a robust measure of dispersion (variability), defined using the median instead of the mean:

$$\text{MAD} = \text{median}(\text{abs}(x_i - \text{median}(\mathbf{x})))$$

The advantage of *MAD* is that it's always well defined, even for data that has infinite variance.

For normally distributed data the *MAD* has a larger standard error than the standard deviation.

But for distributions with fat tails (like asset returns), the standard deviation has a larger standard error than the *MAD*.

The *MAD* for normally distributed data is equal to $\Phi^{-1}(0.75) \cdot \hat{\sigma} = 0.6745 \cdot \hat{\sigma}$.

The function `mad()` calculates the *MAD* and divides it by $\Phi^{-1}(0.75)$ to make it comparable to the standard deviation.

```
> nrows <- 1000
> datav <- rnorm(nrows)
> sd(datav); mad(datav)
> median(abs(datav - median(datav)))
> median(abs(datav - median(datav)))/qnorm(0.75)
> # Bootstrap of sd and mad estimators
> nboot <- 10000
> bootd <- sapply(1:nboot, function(x) {
+   samplev <- rnorm(nrows)
+   c(sd=sd(samplev), mad=mad(samplev))
+ })  # end sapply
> bootd <- t(bootd)
> # Analyze bootstrapped variance
> head(bootd)
> sum(is.na(bootd))
> # Means and standard errors from bootstrap
> apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
> # Parallel bootstrap under Windows
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster
> bootd <- parLapply(cluster, 1:nboot,
+   function(x, datav) {
+     samplev <- rnorm(nrows)
+     c(sd=sd(samplev), mad=mad(samplev))
+   }, datav=datav)  # end parLapply
> # Parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+   samplev <- rnorm(nrows)
+   c(sd=sd(samplev), mad=mad(samplev))
+ }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster
> bootd <- rutils::do_call(rbind, bootd)
> # Means and standard errors from bootstrap
> apply(bootd, MARGIN=2, function(x)
+   c(mean=mean(x), stderror=sd(x)))
```

Resampling From Empirical Datasets

Resampling is randomly selecting data from an existing dataset, to create a new dataset with similar properties to the existing dataset.

Resampling is usually performed with replacement, so that each draw is independent from the others.

Resampling is performed when it's not possible or convenient to obtain another set of empirical data, so we simulate a new data set by randomly sampling from the existing data.

The function `sample()` selects a random sample from a vector of data elements.

The function `sample.int()` is a *method* that selects a random sample of *integers*.

The function `sample.int()` with argument `replace=TRUE` selects a sample with replacement (the *integers* can repeat).

The function `sample.int()` is a little faster than `sample()`.

```
> # Calculate time series of VTI returns
> library(rutils)
> returns <- rutils::etfenv$returns$VTI
> returns <- na.omit(returns)
> nrows <- NROW(returns)
> # Sample from VTI returns
> samplev <- returns[sample.int(nrows, replace=TRUE)]
> c(sd=sd(samplev), mad=mad(samplev))
> # sample.int() is a little faster than sample()
> library(microbenchmark)
> summary(microbenchmark(
+   sample.int = sample.int(1e3),
+   sample = sample(1e3),
+   times=10))[, c(1, 4, 5)]
```

Bootstrapping From Empirical Datasets

Bootstrapping is usually performed by resampling from an observed (empirical) dataset.

Resampling consists of randomly selecting data from an existing dataset, with replacement.

Resampling produces a new *bootstrapped* dataset with similar properties to the existing dataset.

The *bootstrapped* dataset is used to recalculate the estimator many times.

The *bootstrapped* estimator values are then used to calculate the probability distribution of the estimator and its standard error.

Bootstrapping shows that for asset returns, the *Median Absolute Deviation* (*MAD*) has a smaller relative standard error than the standard deviation.

Bootstrapping doesn't provide accurate estimates for estimators which are sensitive to the ordering and correlations in the data.

```
> # Sample from time series of VTI returns
> library(rutils)
> returns <- rutils::etfenv$returns$VTI
> returns <- na.omit(returns)
> nrows <- NROW(returns)
> # Bootstrap sd and MAD under Windows
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster under
> clusterSetRNGStream(cluster, 1121)  # Reset random number generato
> nboot <- 10000
> bootd <- parLapply(cluster, 1:nboot,
+   function(x, returns, nrows) {
+     samplev <- returns[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   }, returns=returns, nrows*nrows)  # end parLapply
> # Bootstrap sd and MAD under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+     samplev <- returns[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster under Windo
> bootd <- rutils::do_call(rbind, bootd)
> # Standard error assuming normal distribution of returns
> sd(returns)/sqrt(nboot)
> # Means and standard errors from bootstrap
> stderrors <- apply(bootd, MARGIN=2,
+   function(x) c(mean=mean(x), stderror=sd(x)))
> stderrors
> # Relative standard errors
> stderrors[2, ]/stderrors[1, ]
```

Standard Errors of Regression Coefficients Using Bootstrap

The standard errors of the regression coefficients can be calculated using a *bootstrap* simulation.

The *bootstrap* procedure creates new design matrices by randomly sampling with replacement from the regression design matrix.

Regressions are performed on the *bootstrapped* design matrices, and the regression coefficients are saved into a matrix of *bootstrapped* coefficients.

```
> # Initialize random number generator
> set.seed(1121)
> # Define explanatory and response variables
> predictor <- rnorm(100, mean=2)
> noise <- rnorm(100)
> response <- (-3 + predictor + noise)
> design <- cbind(response, predictor)
> # Calculate alpha and beta regression coefficients
> betav <- cov(design[, 1], design[, 2])/var(design[, 2])
> alpha <- mean(design[, 1]) - betav*mean(design[, 2])
> x11(width=6, height=5)
> plot(response ~ predictor, data=design)
> abline(a=alpha, b=betav, lwd=3, col="blue")
> # Bootstrap of beta regression coefficient
> nboot <- 100
> bootd <- sapply(1:nboot, function(x) {
+   samplev <- sample.int(NROW(design), replace=TRUE)
+   design <- design[samplev, ]
+   cov(design[, 1], design[, 2])/var(design[, 2])
+ })  # end sapply
```

Distribution of Bootstrapped Regression Coefficients

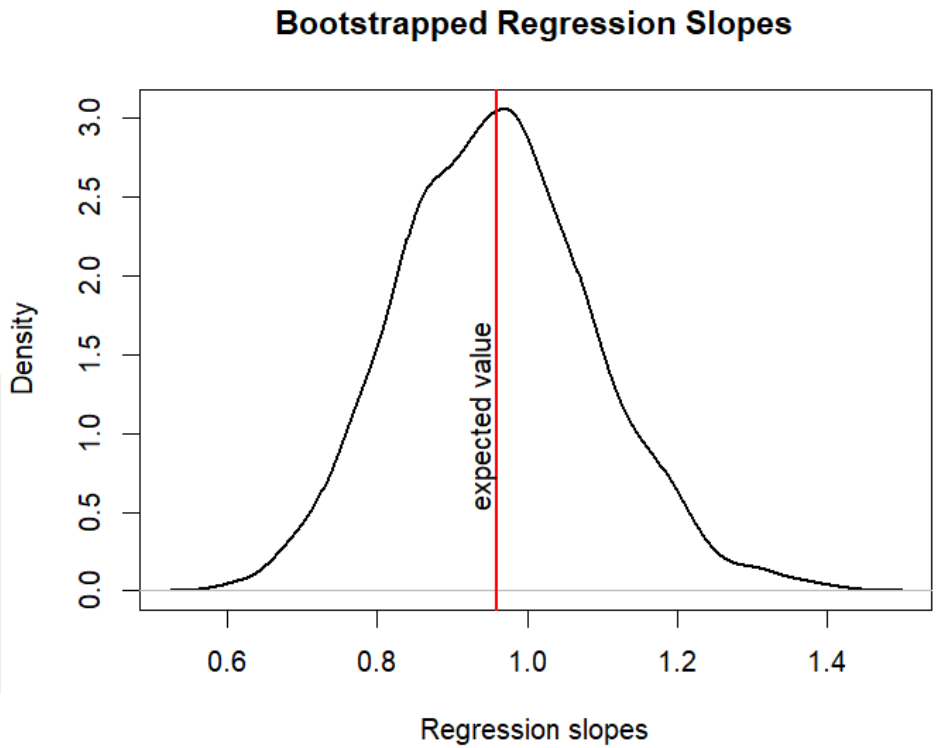
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

The function `density()` calculates a kernel estimate of the probability density for a sample of data.

`abline()` plots a straight line on the existing plot.

The function `text()` draws text on a plot, and can be used to draw plot labels.

```
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd), lwd=2, xlab="Regression slopes",
+      main="Bootstrapped Regression Slopes")
> # Add line for expected value
> abline(v=mean(bootd), lwd=2, col="red")
> text(x=mean(bootd)-0.01, y=1.0, labels="expected value",
+      lwd=2, srt=90, pos=3)
```



Bootstrapping Regressions Using Parallel Computing

The *bootstrap* procedure performs a loop, which naturally lends itself to parallel computing.

Different functions from package *parallel* need to be called depending on the operating system (*Windows*, *Mac-OSX*, or *Linux*).

The function `makeCluster()` starts running `R` processes on several CPU cores under *Windows*.

The function `parLapply()` is similar to `lapply()`, and performs loops under *Windows* using parallel computing on several CPU cores.

The `R` processes started by `makeCluster()` don't inherit any data from the parent `R` process.

Therefore the required data must be passed into `parLapply()` via the dots `"..."` argument.

The function `mclapply()` performs loops using parallel computing on several CPU cores under *Mac-OSX* or *Linux*.

The function `stopCluster()` stops the `R` processes running on several CPU cores.

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster under
> # Bootstrap of regression under Windows
> bootd <- parLapply(cluster, 1:1000,
+   function(x, design) {
+     samplev <- sample.int(NROW(design), replace=TRUE)
+     design <- design[samplev, ]
+     cov(design[, 1], design[, 2])/var(design[, 2])
+   }, design=design)  # end parLapply
> # Bootstrap of regression under Mac-OSX or Linux
> bootd <- mclapply(1:1000,
+   function(x) {
+     samplev <- sample.int(NROW(design), replace=TRUE)
+     design <- design[samplev, ]
+     cov(design[, 1], design[, 2])/var(design[, 2])
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster under Windo
```

Analyzing the Bootstrap Data

The *bootstrap* loop produces a *list* which can be collapsed into a vector.

The function `unlist()` collapses a list with atomic elements into a vector (which can cause type coercion).

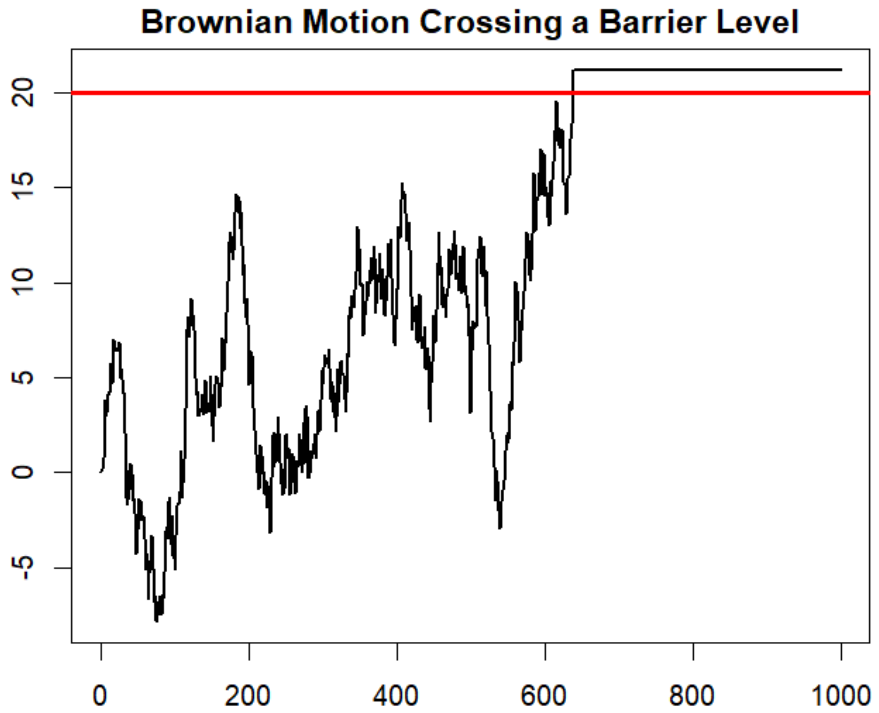
```
> # Collapse the bootstrap list into a vector
> class(bootd)
> bootd <- unlist(bootd)
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd),
+      lwd=2, xlab="Regression slopes",
+      main="Bootstrapped Regression Slopes")
> # Add line for expected value
> abline(v=mean(bootd), lwd=2, col="red")
> text(x=mean(bootd)-0.01, y=1.0, labels="expected value",
+      lwd=2, srt=90, pos=3)
```

Simulating Brownian Motion Using while() Loops

while() loops are often used in simulations, when the number of required loops is unknown in advance.

Below is an example of a simulation of the path of *Brownian Motion* crossing a barrier level.

```
> set.seed(1121)  # Reset random number generator
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> paths <- numeric(nrows)  # Allocate path vector
> paths[1] <- 0  # Initialize path
> indeks <- 2  # Initialize simulation index
> while ((indeks <= nrows) && (paths[indeks - 1] < barl)) {
+ # Simulate next step
+   paths[indeks] <- paths[indeks - 1] + rnorm(1)
+   indeks <- indeks + 1  # Advance indeks
+ }  # end while
> # Fill remaining paths after it crosses barl
> if (indeks <= nrows)
+   paths[indeks*nrows] <- paths[indeks - 1]
> # Plot the Brownian motion
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(paths, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=barl, lwd=3, col="red")
> title(main="Brownian Motion Crossing a Barrier Level", line=0.5)
```

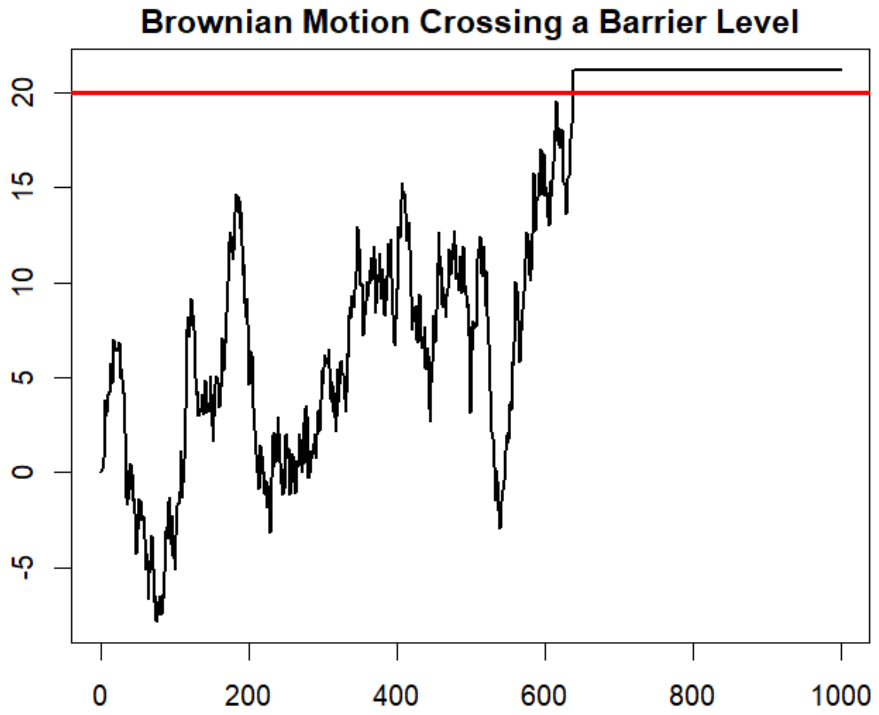


Simulating Brownian Motion Using Vectorized Functions

Simulations in R can be accelerated by pre-computing a vector of random numbers, instead of generatng them one at a time in a loop.

Vectors of random numbers allow using *vectorized* functions, instead of inefficient (slow) `while()` loops.

```
> set.seed(1121)  # Reset random number generator
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> # Simulate path of Brownian motion
> pathv <- cumsum(rnorm(nrows))
> # Find index when pathv crosses barl
> crossp <- which(pathv > barl)
> # Fill remaining pathv after it crosses barl
> if (NROW(crossp)>0) {
+   pathv[(crossp[1]+1):nrows] <- pathv[crossp[1]]
+ }  # end if
> # Plot the Brownian motion
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(pathv, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=barl, lwd=3, col="red")
> title(main="Brownian Motion Crossing a Barrier Level", line=0.5)
```



The tradeoff between speed and memory usage: more memory may be used than necessary, since the simulation may stop before all the pre-computed random numbers are used up.

But the simulation is much faster because the path is simulated using *vectorized* functions,

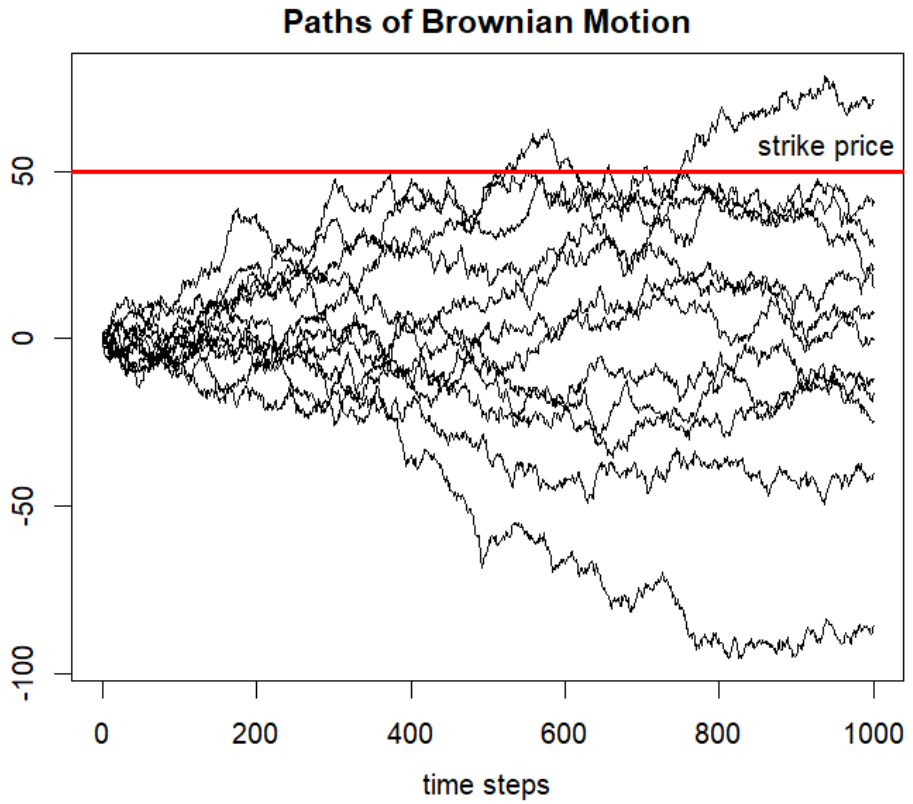
Estimating the Statistics of Brownian Motion

The statistics of Brownian motion can be estimated by simulating multiple paths.

An example of a statistic is the expected value of Brownian motion at a fixed time horizon, which is the option payout for the strike price k: $\mathbb{E}[(p_t - k)_+]$.

Another statistic is the probability of Brownian motion crossing a boundary (barrier) b: $\mathbb{E}[\mathbb{1}(p_t - b)]$.

```
> # Define Brownian motion parameters
> sigmav <- 1.0  # Volatility
> drift <- 0.0  # Drift
> nrows <- 1000  # Number of simulation steps
> nsimu <- 100  # Number of simulations
> # Simulate multiple paths of Brownian motion
> set.seed(1121)
> paths <- rnorm(nsimu*nrows, mean=drift, sd=sigmav)
> paths <- matrix(paths, nc=nsimu)
> paths <- matrixStats::colCumsums(paths)
> # Final distribution of paths
> mean(paths[nrows, ]) ; sd(paths[nrows, ])
> # Calculate option payout at maturity
> strikep <- 50  # Strike price
> payouts <- (paths[nrows, ] - strikep)
> sum(payouts[payouts > 0])/nsimu
> # Calculate probability of crossing the barrier at any point
> barl <- 50
> crossi <- (colSums(paths > barl) > 0)
> sum(crossi)/nsimu
```




```
> # Plot in window
> x11(width=6, height=5)
> par(mar=c(4, 3, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2.5, 1, 0))
> # Select and plot full range of paths
> ordern <- order(paths[nrows, ])
> indeks <- ordern[seq(1, 100, 9)]
> zoo::plot.zoo(paths[, indeks], main="Paths of Brownian Motion",
+   xlab="time steps", ylab=NA, plot.type="single")
> abline(h=strikep, col="red", lwd=3)
> text(x=(nrows-60), y=strikep, labels="strike price", pos=3, cex=1)
```

Bootstrapping From Time Series of Prices

Bootstrapping from a time series of prices requires first converting the prices to *percentage* returns, then bootstrapping the returns, and finally converting them back to prices.

Bootstrapping from *percentage* returns ensures that the bootstrapped prices are not negative.

Below is a simulation of the frequency of bootstrapped prices crossing a barrier level.

```
> # Calculate percentage returns from VTI prices
> library(rutils)
> prices <- quantmod::Cl(rutils::etfenv$VTI)
> startd <- as.numeric(prices[1, ])
> returns <- rutils::diffit(log(prices))
> class(returns); head(returns)
> sum(is.na(returns))
> nrows <- NROW(returns)
> # Define barrier level with respect to prices
> barl <- 1.5*max(prices)
> # Calculate single bootstrap sample
> samplev <- returns[sample.int(nrows, replace=TRUE)]
> # Calculate prices from percentage returns
> samplev <- startd*exp(cumsum(samplev))
> # Calculate if prices crossed barrier
> sum(samplev > barl) > 0
```

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster under
> # Perform parallel bootstrap under Windows
> clusterSetRNGStream(cluster, 1121)  # Reset random number generato
> clusterExport(cluster, c("startd", "barl"))
> nboot <- 10000
> bootd <- parLapply(cluster, 1:nboot,
+   function(x, returns, nrows) {
+     samplev <- returns[sample.int(nrows, replace=TRUE)]
+     # Calculate prices from percentage returns
+     samplev <- startd*exp(cumsum(samplev))
+     # Calculate if prices crossed barrier
+     sum(samplev > barl) > 0
+   }, returns=returns, nrows*nrows)  # end parLapply
> # Perform parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+     samplev <- returns[sample.int(nrows, replace=TRUE)]
+     # Calculate prices from percentage returns
+     samplev <- startd*exp(cumsum(samplev))
+     # Calculate if prices crossed barrier
+     sum(samplev > barl) > 0
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster under Windo
> bootd <- rutils::do_call(rbind, bootd)
> # Calculate frequency of crossing barrier
> sum(bootd)/nboot
```

Bootstrapping From *OHLC* Prices

Bootstrapping from *OHLC* prices requires updating all the price columns, not just the *Close* prices.

The *Close* prices are bootstrapped first, and then the other columns are updated using the differences of the *OHLC* price columns.

Below is a simulation of the frequency of the *High* prices crossing a barrier level.

```
> # Calculate percentage returns from VTI prices
> library(rutils)
> ohlc <- rutils::etfenv$VTI
> prices <- as.numeric(ohlc[, 4])
> startd <- prices[1]
> returns <- rutils::diffit(log(prices))
> nrows <- NROW(returns)
> # Calculate difference of OHLC price columns
> ohlc_diff <- ohlc[, 1:3] - prices
> class(returns); head(returns)
> # Calculate bootstrap prices from percentage returns
> datav <- sample.int(nrows, replace=TRUE)
> boot_prices <- startd*exp(cumsum(returns[datav]))
> boot_ohlc <- ohlc_diff + boot_prices
> boot_ohlc <- cbind(boot_ohlc, boot_prices)
> # Define barrier level with respect to prices
> barl <- 1.5*max(prices)
> # Calculate if High bootstrapped prices crossed barrier level
> sum(boot_ohlc[, 2] > barl) > 0
```

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster under
> # Perform parallel bootstrap under Windows
> clusterSetRNGStream(cluster, 1121)  # Reset random number generato
> clusterExport(cluster, c("startd", "barl", "ohlc_diff"))
> nboot <- 10000
> bootd <- parLapply(cluster, 1:nboot,
+   function(x, returns, nrows) {
+     # Calculate OHLC prices from percentage returns
+     datav <- sample.int(nrows, replace=TRUE)
+     boot_prices <- startd*exp(cumsum(returns[datav]))
+     boot_ohlc <- ohlc_diff + boot_prices
+     boot_ohlc <- cbind(boot_ohlc, boot_prices)
+     # Calculate statistic
+     sum(boot_ohlc[, 2] > barl) > 0
+   }, returns=returns, nrows*nrows)  # end parLapply
> # Perform parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+     # Calculate OHLC prices from percentage returns
+     datav <- sample.int(nrows, replace=TRUE)
+     boot_prices <- startd*exp(cumsum(returns[datav]))
+     boot_ohlc <- ohlc_diff + boot_prices
+     boot_ohlc <- cbind(boot_ohlc, boot_prices)
+     # Calculate statistic
+     sum(boot_ohlc[, 2] > barl) > 0
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster under Windo
> bootd <- rutils::do_call(rbind, bootd)
> # Calculate frequency of crossing barrier
> sum(bootd)/nboot
```

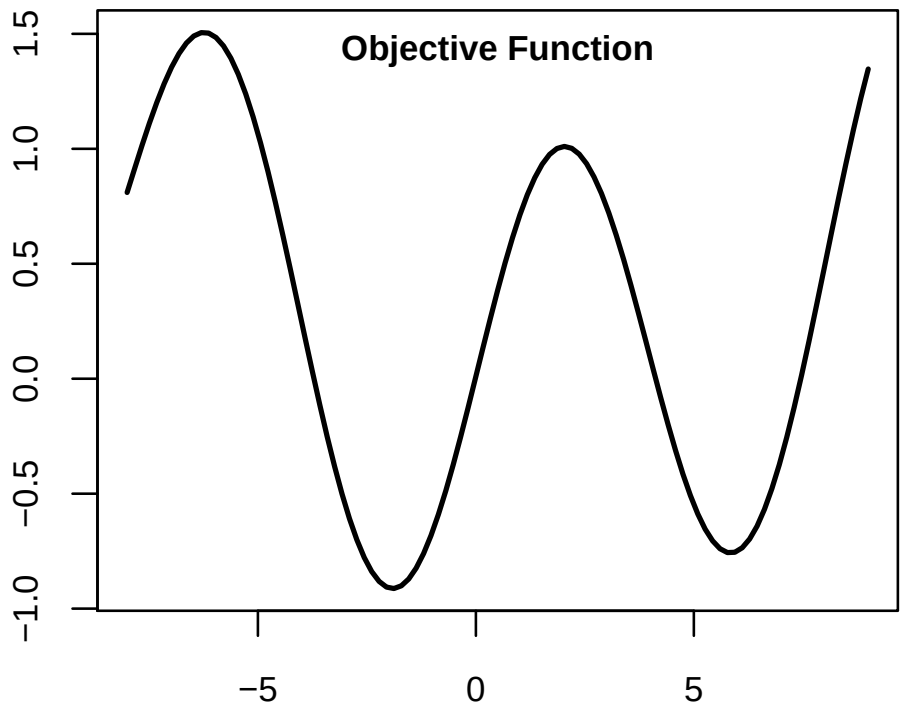
One-dimensional Optimization Using The Functional optimize()

The functional optimize() performs *one-dimensional* optimization over a single independent variable.

optimize() searches for the minimum of the objective function with respect to its first argument, in the specified interval.

optimize() returns a list containing the location of the minimum and the objective function value,

```
> str(optimize)
> # Objective function with multiple minima
> objfun <- function(input, param1=0.01) {
+   sin(0.25*pi*input) + param1*(input-1)^2
+ }  # end objfun
> unlist(optimize(f=objfun, interval=c(-4, 2)))
> unlist(optimize(f=objfun, interval=c(0, 8)))
> options(width=80, dev="pdf")
```



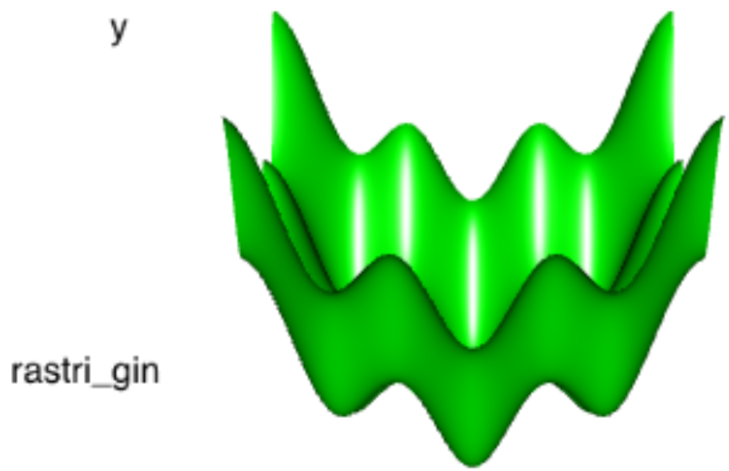

```
> # Plot the objective function
> curve(expr=objfun, type="l", xlim=c(-8, 9),
+ xlab="", ylab="", lwd=2)
> # Add title
> title(main="Objective Function", line=-1)
```

Package *rgl* for Interactive 3d Surface Plots

The package *rgl* creates *interactive* 3d scatter plots and surface plots by calling the *WebGL JavaScript* library.

The function `rgl::persp3d()` plots an *interactive* 3d surface plot of a *vectorized* function or a matrix.

```
> # Rastrigin function
> rastrigin <- function(x, y, param=25) {
+   x^2 + y^2 - param*(cos(x) + cos(y))
+ }  # end rastrigin
> # Rastrigin function is vectorized!
> rastrigin(c(-10, 5), c(-10, 5))
> # Set rgl options and load package rgl
> library(rgl)
> options(rgl.useNULL=TRUE)
> # Draw 3d surface plot of function
> rgl::persp3d(x=rastrigin, xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, param=15)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=400, height=400)
```



Multi-dimensional Optimization Using optim()

The function optim() performs *multi-dimensional* optimization.

The argument fn is the objective function to be minimized.

The argument of fn that is to be optimized, must be a vector argument.

The argument par is the initial vector argument value.

optim() accepts additional parameters bound to the dots "..." argument, and passes them to the fn objective function.

The arguments lower and upper specify the search range for the variables of the objective function fn.

method="L-BFGS-B" specifies the quasi-Newton *gradient* optimization method.

optim() returns a list containing the location of the minimum and the objective function value.

The *gradient* methods used by optim() can only find the local minimum, not the global minimum.

```
> # Rastrigin function with vector argument for optimization
> rastrigin <- function(vectorv, param=25) {
+   sum(vectorv^2 - param*cos(vectorv))
+ }  # end rastrigin
> vectorv <- c(pi/6, pi/6)
> rastrigin(vectorv=vectorv)
> # Draw 3d surface plot of Rastrigin function
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastrigin(vectorv=c(x, y))),
+   xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, zlab="", main="rastrigin")
> # Optimize with respect to vector argument
> optiml <- optim(par=vectorv, fn=rastrigin,
+         method="L-BFGS-B",
+         upper=c(4*pi, 4*pi),
+         lower=c(pi/2, pi/2),
+         param=1)
> # Optimal parameters and value
> optiml$par
> optiml$value
> rastrigin(optiml$par, param=1)
```

The Likelihood Function

The *likelihood* function $\mathcal{L}(\theta|\bar{x})$ is a function of the parameters of a statistical model θ, given a sample of observed values $\bar{x}$, taken under the model's probability distribution $p(x|\theta)$:

$$\mathcal{L}(\theta|x) = \prod_{i=1}^{n} p(x_i|\theta)$$

The *likelihood* function measures how *likely* are the parameters of a statistical model, given a sample of observed values $\bar{x}$.

The *maximum-likelihood* estimate (*MLE*) of the model's parameters are those that maximize the *likelihood* function:

$$\theta_{MLE} = \arg\max_{\theta} \mathcal{L}(\theta|x)$$

In practice the logarithm of the *likelihood* $\log(\mathcal{L})$ is maximized, instead of the *likelihood* itself.

The function `outer()` calculates the *outer* product of two matrices, and by default multiplies the elements of its arguments.

```
> # Sample of normal variables
> datav <- rnorm(1000, mean=4, sd=2)
> # Objective function is log-likelihood
> objfun <- function(parv, datav) {
+   sum(2*log(parv[2]) +
+     ((datav - parv[1])/parv[2])^2)
+ }  # end objfun
> # Objective function on parameter grid
> par_mean <- seq(1, 6, length=50)
> par_sd <- seq(0.5, 3.0, length=50)
> objective_grid <- sapply(par_mean, function(m) {
+   sapply(par_sd, function(sd) {
+     objfun(c(m, sd), datav)
+   })  # end sapply
+ })  # end sapply
> # Perform grid search for minimum
> objective_min <- which(
+   objective_grid==min(objective_grid),
+   arr.ind=TRUE)
> objective_min
> par_mean[objective_min[1]]  # mean
> par_sd[objective_min[2]]  # sd
> objective_grid[objective_min]
> objective_grid[(objective_min[, 1] + -1:1),
+        (objective_min[, 2] + -1:1)]
> # Or create parameter grid using function outer()
> objvecive <- Vectorize(
+   FUN=function(mean, sd, datav)
+     objfun(c(mean, sd), datav),
+   vectorize.args=c("mean", "sd")
+ )  # end Vectorize
> objective_grid <- outer(par_mean, par_sd,
+ objvecive, datav=datav)
```

Perspective Plot of Likelihood Function

The function `persp()` plots a 3d perspective surface plot of a function specified over a grid of argument values.

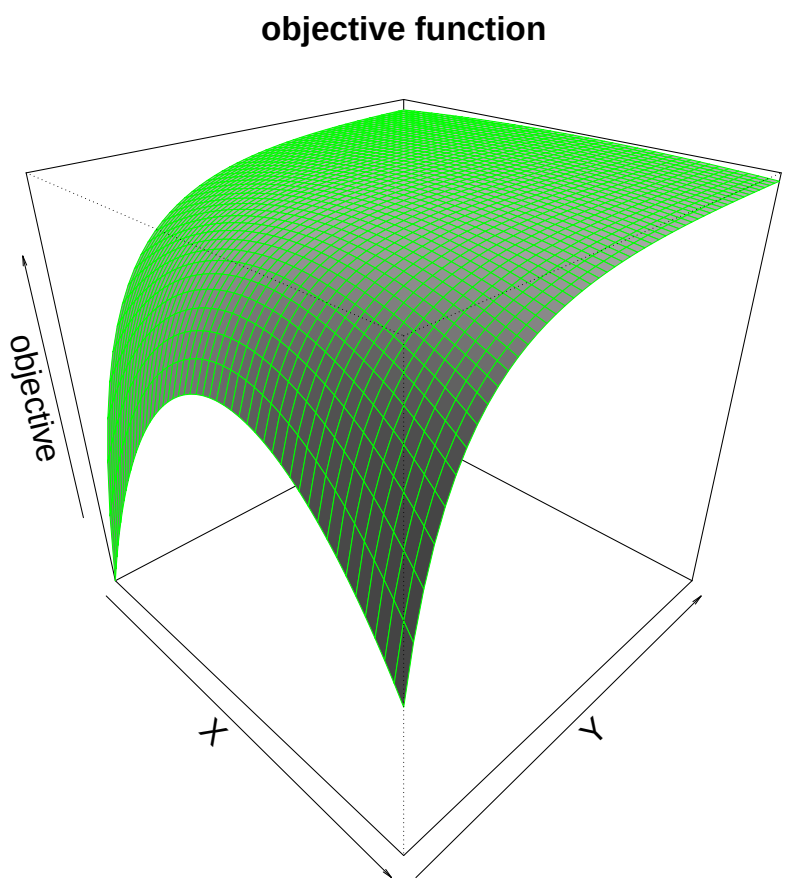
The argument `"z"` accepts a matrix containing the function values.

`persp()` belongs to the base `graphics` package, and doesn't create interactive plots.

The function `rgl::persp3d()` plots an *interactive* 3d surface plot of a function or a matrix.

`rgl` is an `R` package for 3d and perspective plotting, based on the *OpenGL* framework.

```
> # Perspective plot of log-likelihood function
> persp(z=-objective_grid,
+ theta=45, phi=30, shade=0.5,
+ border="green", zlab="objective",
+ main="objective function")
> # Interactive perspective plot of log-likelihood function
> library(rgl)  # Load package rgl
> rgl::par3d(cex=2.0)  # Scale text by factor of 2
> rgl::persp3d(z=-objective_grid, zlab="objective",
+   col="green", main="objective function")
```

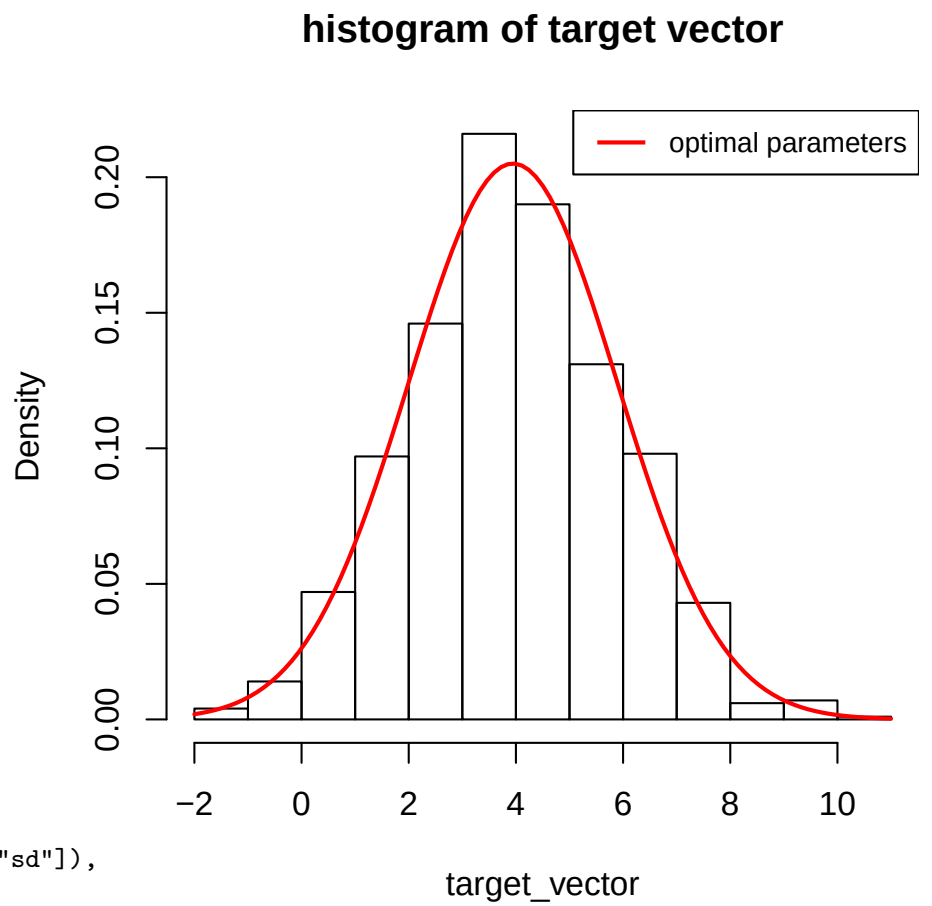


Optimization of Objective Function

The function `optim()` performs optimization of an objective function.

The function `fitdistr()` from package *MASS* fits a univariate distribution to a sample of data, by performing *maximum likelihood* optimization.

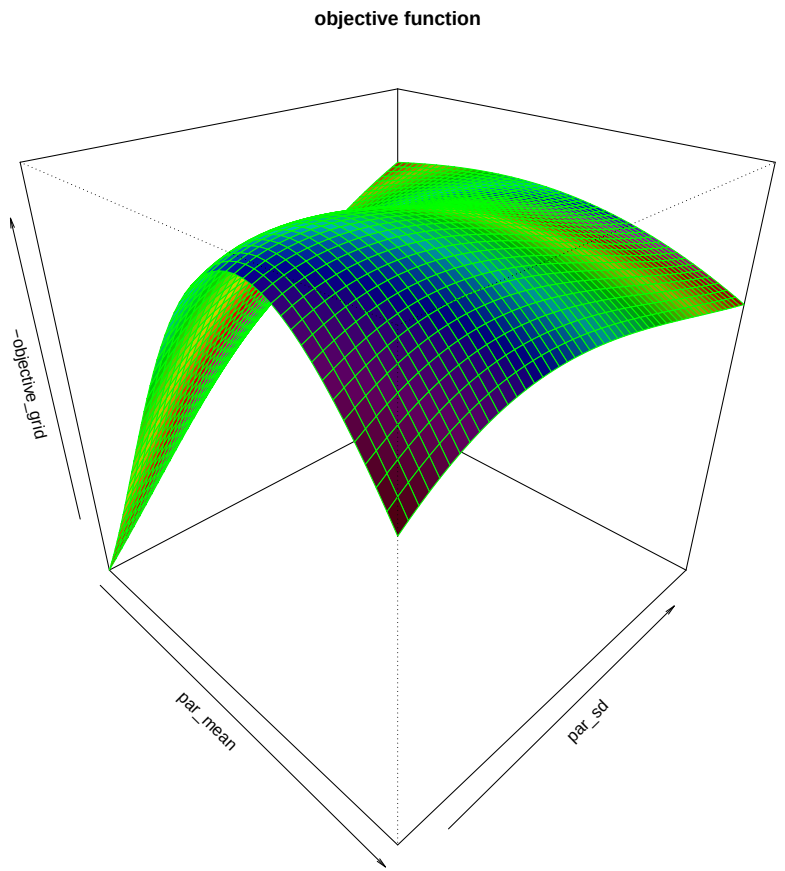
```
> # Initial parameters
> initp <- c(mean=0, sd=1)
> # Perform optimization using optim()
> optim_fit <- optim(par=initp,
+   fn=objfun, # Log-likelihood function
+   datav=datav,
+   method="L-BFGS-B", # Quasi-Newton method
+   upper=c(10, 10), # Upper constraint
+   lower=c(-10, 0.1)) # Lower constraint
> # Optimal parameters
> optim_fit$par
> # Perform optimization using MASS::fitdistr()
> optim_fit <- MASS::fitdistr(datav, densfun="normal")
> optim_fit$estimate
> optim_fit$sd
> # Plot histogram
> histp <- hist(datav, plot=FALSE)
> plot(histp, freq=FALSE, main="histogram of sample")
> curve(expr=dnorm(x, mean=optim_fit$par["mean"], sd=optim_fit$par["sd"]),
+ add=TRUE, type="l", lwd=2, col="red")
> legend("topright", inset=0.0, cex=0.8, title=NULL,
+  leg="optimal parameters", lwd=2, bg="white", col="red")
```



Mixture Model Likelihood Function

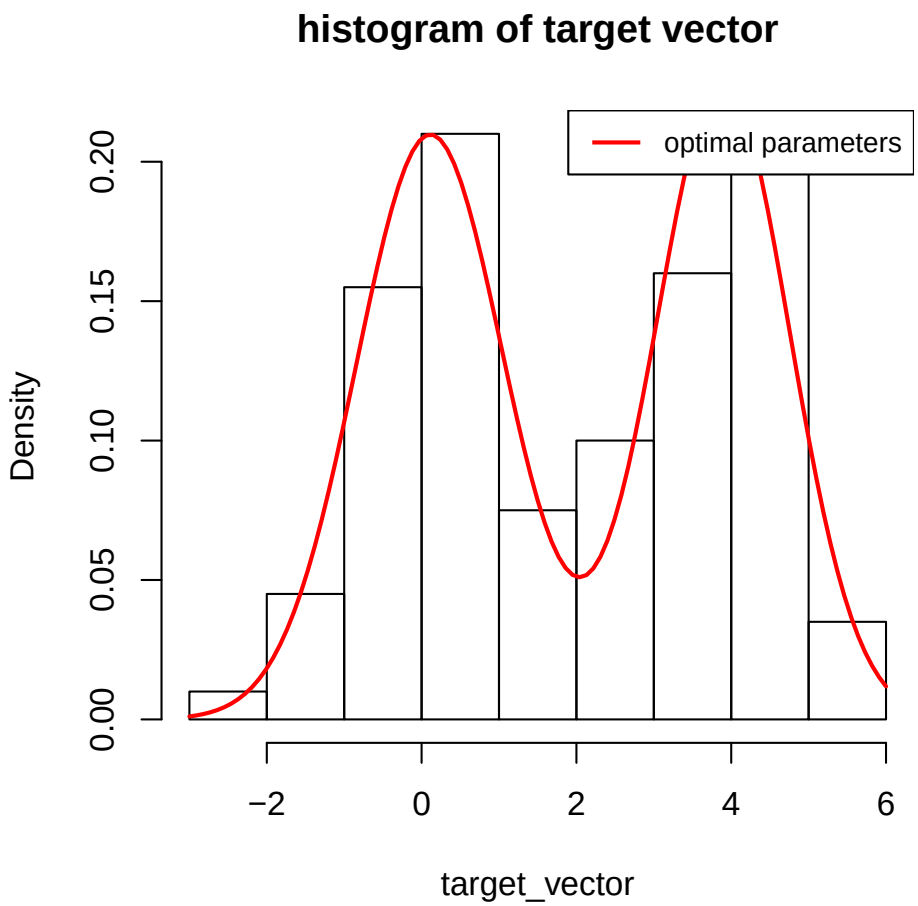
```
> # Sample from mixture of normal distributions
> datav <- c(rnorm(100, sd=1.0),
+       rnorm(100, mean=4, sd=1.0))
> # Objective function is log-likelihood
> objfun <- function(parv, datav) {
+   likev <- parv[1]/parv[3] *
+   dnorm((datav-parv[2])/parv[3]) +
+   (1-parv[1])/parv[5]*dnorm((datav-parv[4])/parv[5])
+   if (any(likev <= 0)) Inf else
+     -sum(log(likev))
+ }  # end objfun
> # Vectorize objective function
> objvecive <- Vectorize(
+   FUN=function(mean, sd, w, m1, s1, datav)
+     objfun(c(w, m1, s1, mean, sd), datav),
+   vectorize.args=c("mean", "sd")
+ )  # end Vectorize
> # Objective function on parameter grid
> par_mean <- seq(3, 5, length=50)
> par_sd <- seq(0.5, 1.5, length=50)
> objective_grid <- outer(par_mean, par_sd,
+     objvecive, datav=datav,
+     w=0.5, m1=2.0, s1=2.0)
> rownames(objective_grid) <- round(par_mean, 2)
> colnames(objective_grid) <- round(par_sd, 2)
> objective_min <- which(objective_grid==
+   min(objective_grid), arr.ind=TRUE)
> objective_min
> objective_grid[objective_min]
> objective_grid[(objective_min[, 1] + -1:1),
+          (objective_min[, 2] + -1:1)]
```

```
> # Perspective plot of objective function
> persp(par_mean, par_sd, -objective_grid,
+ theta=45, phi=30,
+ shade=0.5,
+ col=rainbow(50),
+ border="green",
+ main="objective function")
```



Optimization of Mixture Model

```
> # Initial parameters
> initp <- c(weight=0.5, m1=0, s1=1, m2=2, s2=1)
> # Perform optimization
> optim_fit <- optim(par=initp,
+       fn=objfun,
+       datav=datav,
+       method="L-BFGS-B",
+       upper=c(1,10,10,10,10),
+       lower=c(0,-10,0.2,-10,0.2))
> optim_fit$par
> # Plot histogram
> histp <- hist(datav, plot=FALSE)
> plot(histp, freq=FALSE,
+      main="histogram of sample")
> fitfun <- function(x, parv) {
+   parv["weight"]*dnorm(x, mean=parv["m1"], sd=parv["s1"]) +
+   (1-parv["weight"])*dnorm(x, mean=parv["m2"], sd=parv["s2"])
+ }  # end fitfun
> curve(expr=fitfun(x, parv=optim_fit$par), add=TRUE,
+ type="l", lwd=2, col="red")
> legend("topright", inset=0.0, cex=0.8, title=NULL,
+  leg="optimal parameters",
+  lwd=2, bg="white", col="red")
```



Package *DEoptim* for Global Optimization

The function `DEoptim()` from package *DEoptim* performs *global* optimization using the *Differential Evolution* algorithm.

Differential Evolution is a genetic algorithm which evolves a population of solutions over several generations:

http://www1.icsi.berkeley.edu/~storn/code.html

The first generation of solutions is selected randomly.

Each new generation is obtained by combining the best solutions from the previous generation.

The *Differential Evolution* algorithm is well suited for very large multi-dimensional optimization problems, such as portfolio optimization.

Gradient optimization methods are more efficient than *Differential Evolution* for smooth objective functions with no local minima.

```
> # Rastrigin function with vector argument for optimization
> rastrigin <- function(vectorv, param=25) {
+   sum(vectorv^2 - param*cos(vectorv))
+ }  # end rastrigin
> vectorv <- c(pi/6, pi/6)
> rastrigin(vectorv=vectorv)
> library(DEoptim)
> # Optimize rastrigin using DEoptim
> optiml <-  DEoptim(rastrigin,
+   upper=c(6, 6), lower=c(-6, -6),
+   DEoptim.control(trace=FALSE, itermax=50))
> # Optimal parameters and value
> optiml$optim$bestmem
> rastrigin(optiml$optim$bestmem)
> summary(optiml)
> plot(optiml)
```

Homework Assignment

Required

- Study all the lecture slides in `FRE7241_Lecture_1.pdf`, and run all the code in `FRE7241_Lecture_1.R`,

Recommended

- Read the documentation for packages `rutils.pdf` and `HighFreq.pdf`,