FRE7241 Algorithmic Portfolio Management

Lecture#0, Fall 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 3, 2024

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

The R Cheat Sheets are a fast way to find what you want.

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
http://news.gmane.org/gmane.comp.lang.r.general

R Code Style Guidelines

Please follow the R code style from the lecture slides.
Please follow the *Google R Style Guide* to make your R code more readable.

Please also follow these R code style rules:

- Use the left arrow "`<-`" for assignment, not the equals sign "`=`" (to insert "`<-`" into code, use the *Alt-hyphen* shortcut in Windows, or the *Option-hyphen* shortcut on the Mac),
- Use *nouns* for variable names and *verbs* for function names,
- Use a combination of lowercase letters, numbers, and underscores "`_`" for names of variables and functions,
- Add underscores "`_`" to names to avoid conflicts with the names of existing R functions and variables,
- Do not use dots "`.`" in names, except in the names of function *methods* (even though R uses them for variables as well),
- Use underscores "`_`" in file names, instead of spaces,
- Always put a space after a comma, never before it: "`x, y`" not "`x , y`",
- Do not put spaces inside or outside parentheses: "`if (x > 0)`" not "`if ( x > 0 )`",
- Surround infix operators (`==`, `+`, `-`, `<-`, etc.) with spaces: "`x > 0`" not "`x>0`" (even though I don't always follow that rule, to save whitespace),
- Add a comment after the closing curly bracket: "`} # end my_fun`",

You can reformat R code chunks using the *styler* macros in the *RStudio Addins* drop-down menu.
You can also reformat whole files with R code by using the *styler* package.

Stack Exchange

Stack Overflow

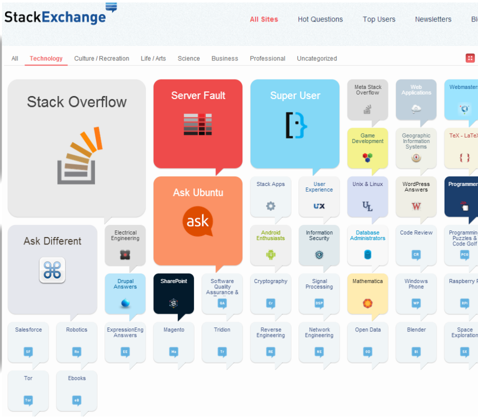
Stack Overflow is a Q&A forum for computer programming, and is part of Stack Exchange

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.

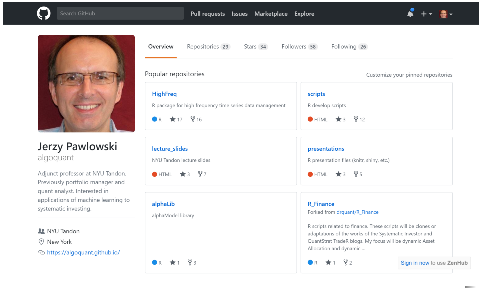
GitHub provides version control using *git* (desved by Linus Torvalds).

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.



Jerzy Pawlowski
algoquant

Adjunct professor at NYU Tandon. Previously portfolio manager and quant analyst. Interested in applications of machine learning to systematic investing.

NYU Tandon
New York
https://algoquant.github.io/

Popular repositories

HighFreq
R package for high frequency time series data management
● R ★ 17 16

scripts
R develop scripts
● HTML ★ 3 12

lecture_slides
NYU Tandon lecture slides
● HTML ★ 3 7

presentations
R presentation files (knitr, shiny, etc.)
● HTML ★ 3 5

alphaLib
alphaModel library
● R ★ 1 3

R_Finance
Forked from drquant/R_Finance
R scripts related to finance. These scripts will be clones or adaptations of the works of the Systematic Investor and QuantStrat TradeR blogs. My focus will be dynamic Asset Allocation and dynamic ...
● R ★ 1 2

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.

Project

Wikipedia

Why is R More Difficult Than Other Languages?

R is more difficult than other languages because:

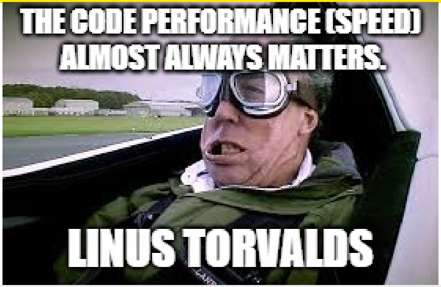
- R is a *functional* language, which makes its syntax unfamiliar to users of procedural languages like C/C++.
- The huge number of user-created *packages* makes it difficult to tell which are the best for particular applications.
- R can produce very cryptic *warning* and *error* messages, because it's 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:

- Avoid using apply() and for() loops for large datasets.
- Pre-allocate memory for new objects.
- Avoid using too many R function calls (every command in R is a function).
- Use R as an interface to libraries written in C++, Java, and JavaScript.
- Use R functions which are *compiled* C++ code, instead of using interpreted R code.
- Write C++ functions in *Rcpp* and *RcppArmadillo*.
- 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.



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

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.

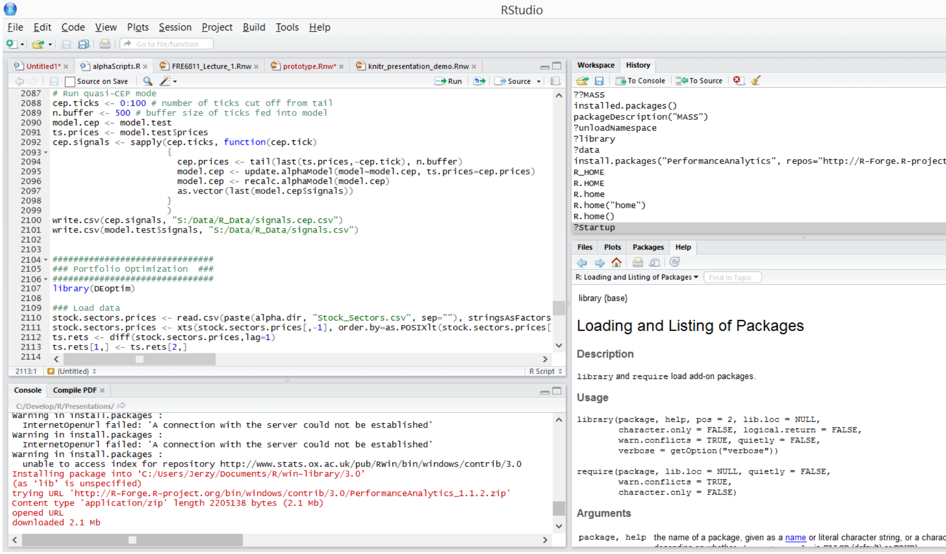
Laptop computers will be necessary for following the lectures, and for performing tests.

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*



```
# Run quasi-CEP mode
cep.ticks <- 0:100 # number of ticks cut off from tail
n.buffer <- 500 # buffer size of ticks fed into model
model.cep <- model.test
ts.prices <- model.test$prices
cep.signals <- sapply(cep.ticks, function(cep.tick)
                      {
                        cep.prices <- tail(last(ts.prices,-cep.tick), n.buffer)
                        model.cep <- update.alphaModel(model=model.cep, ts.prices=cep.prices)
                        model.cep <- recalc.alphaModel(model.cep)
                        as.vector(last(model.cep$signals))
                      }
                      )
write.csv(cep.signals, "S:/Data/R_Data/signals.cep.csv")
write.csv(model.test$signals, "S:/Data/R_Data/signals.csv")


################################
### Portfolio Optimization  ###
################################
library(DEoptim)

### Load data
stock.sectors.prices <- read.csv(paste(alpha.dir, "Stock_Sectors.csv", sep=""), stringsAsFactors
stock.sectors.prices <- xts(stock.sectors.prices[,-1], order.by=as.POSIXlt(stock.sectors.prices[
ts.rets <- diff(stock.sectors.prices,lag=1)
ts.rets[1,] <- ts.rets[2,]
```

```
??MASS
installed.packages()
packageDescription("MASS")
?unloadNamespace
?library
?data
install.packages("PerformanceAnalytics", repos="http://R-Forge.R-project
R_HOME
R.HOME
R.home
R.home("home")
R.home()
?Startup
```

```
Warning in install.packages :
  InternetOpenUrl failed: 'A connection with the server could not be established'
Warning in install.packages :
  InternetOpenUrl failed: 'A connection with the server could not be established'
Warning in install.packages :
  unable to access index for repository http://www.stats.ox.ac.uk/pub/RWin/bin/windows/contrib/3.0
Installing package into ‘C:/Users/Jerzy/Documents/R/win-library/3.0’
(as ‘lib’ is unspecified)
trying URL 'http://R-Forge.R-project.org/bin/windows/contrib/3.0/PerformanceAnalytics_1.1.2.zip'
Content type 'application/zip' length 2205138 bytes (2.1 Mb)
opened URL
downloaded 2.1 Mb
```

library {base}

Loading and Listing of Packages

Description

`library` and `require` load add-on packages.

Usage

```
library(package, help, pos = 2, lib.loc = NULL,
        character.only = FALSE, logical.return = FALSE,
        warn.conflicts = TRUE, quietly = FALSE,
        verbose = getOption("verbose"))

require(package, lib.loc = NULL, quietly = FALSE,
        warn.conflicts = TRUE,
        character.only = FALSE)
```

Arguments

`package, help` the name of a package, given as a name or literal character string, or a charac

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 interprets 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] "2024-09-03 16:34:21 EDT"
>
> Sys.Date()  # Get date only
[1] "2024-09-03"
```

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*"))
> # Delete all objects in workspace 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())  # Delete all objects in workspace
> ls()  # List objects
> loadobj <- load(file="/Users/jerzy/Develop/lecture_slides/data/my_
> loadobj
> 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 4.4.1 (2024-06-14)
Platform: aarch64-apple-darwin20
Running under: macOS Ventura 13.3.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-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

time zone: America/New_York
tzcode source: internal

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

other attached packages:
[1] knitr_1.48       HighFreq_0.1     rutils_0.2       dygraphs_1.1.
[5] quantmod_0.4.26  TTR_0.24.4       xts_0.14.0       zoo_1.8-12

loaded via a namespace (and not attached):
 [1] digest_0.6.36     fastmap_1.2.0     xfun_0.46         lattice_0
 [5] magrittr_2.0.3    htmltools_0.5.8.1 cli_3.6.3         grid_4.4.
 [9] compiler_4.4.1    highr_0.11        tools_4.4.1       rstudioap
[13] curl_5.2.1        evaluate_0.24.0   Rcpp_1.0.13       rlang_1.1
[17] htmlwidgets_1.6.4
```

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
> optionv <- options()
> # Restore all options from variable
> options(optionv)
```

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
> envv <- new.env()
> # Get calling environment of new environment
> parent.env(envv)
> # Assign Value to Name
> assign("new_var1", 3, envir=envv)
> # Create object in new environment
> envv$new_var2 <- 11
> # Get objects in new environment
> ls(envv)
> # Get objects in current environment
> ls(environment())
> # Environments are subset like listv
> envv$new_var1
> # Environments are subset like listv
> envv[["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
> pricev <- mget(symbolv, envir=rutils::etfenv)
> # prices is a list of xts series
> class(pricev)
> class(pricev[[1]])
> # Extract Close prices
> pricev <- lapply(pricev, quantmod::Cl)
> # Collapse list into time series the hard way
> xts1 <- cbind(pricev[[1]], pricev[[2]], pricev[[3]], pricev[[4]])
> class(xts1)
> dim(xts1)
> # Collapse list into time series using do.call()
> pricev <- do.call(cbind, pricev)
> all.equal(xts1, pricev)
> class(pricev)
> dim(pricev)
> # Extract and cbind in single step
> pricev <- do.call(cbind, lapply(
+   mget(symbolv, envir=rutils::etfenv), quantmod::Cl))
> # Or
> # Extract and bind all data, subset by symbolv
> pricev <- lapply(symbolv, function(symbol) {
+     quantmod::Cl(get(symbol, envir=rutils::etfenv))
+ })  # end lapply
> # Same, but loop over etfenv without anonymous function
> pricev <- do.call(cbind,
+   lapply(as.list(rutils::etfenv)[symbolv], quantmod::Cl))
> # Same, but works only for OHLC series - produces error
> pricev <- 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(pricev)
> do.call(rbind, strsplit(colnames(pricev), split="[.]"))[, 1]
> colnames(pricev) <- do.call(rbind, strsplit(colnames(pricev), spli
> # Or
> colnames(pricev) <- unname(sapply(colnames(pricev),
+     function(colname) strsplit(colname, split="[.]")[[1]][1]))
> tail(pricev, 3)
> # Which objects in global environment are class xts?
> unlist(eapply(globalenv(), is.xts))
> # Save xts to csv file
> write.zoo(pricev,
+   file="/Users/jerzy/Develop/lecture_slides/data/etf_series.csv",
```

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,

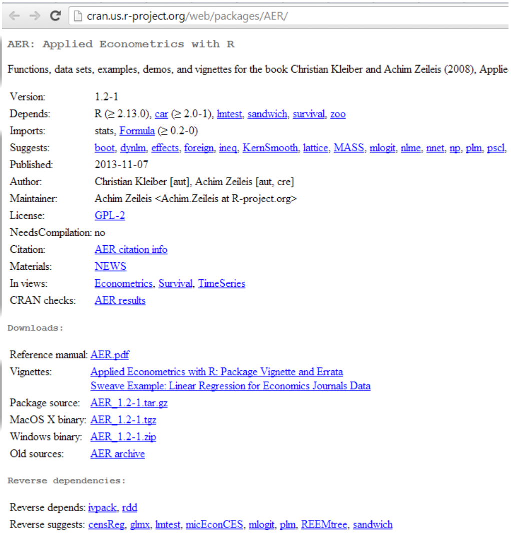
CRAN Package Views

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.us.r-project.org/web/packages/AER/

AER: Applied Econometrics with R

Functions, data sets, examples, demos, and vignettes for the book Christian Kleiber and Achim Zeileis (2008), Applie

Version:	1.2-1
Depends:	R (≥ 2.13.0), car (≥ 2.0-1), lmtest, sandwich, survival, zoo
Imports:	stats, Formula (≥ 0.2-0)
Suggests:	boot, dynlm, effects, foreign, ineq, KernSmooth, lattice, MASS, mlogit, nlme, nnet, np, plm, pscl,
Published:	2013-11-07
Author:	Christian Kleiber [aut], Achim Zeileis [aut, cre]
Maintainer:	Achim Zeileis <Achim.Zeileis at R-project.org>
License:	GPL-2
NeedsCompilation:	no
Citation:	AER citation info
Materials:	NEWS
In views:	Econometrics, Survival, TimeSeries
CRAN checks:	AER results

Downloads:

Reference manual:	AER.pdf
Vignettes:	Applied Econometrics with R: Package Vignette and Errata Sweave Example: Linear Regression for Economics Journals Data
Package source:	AER_1.2-1.tar.gz
MacOS X binary:	AER_1.2-1.tgz
Windows binary:	AER_1.2-1.zip
Old sources:	AER archive

Reverse dependencies:

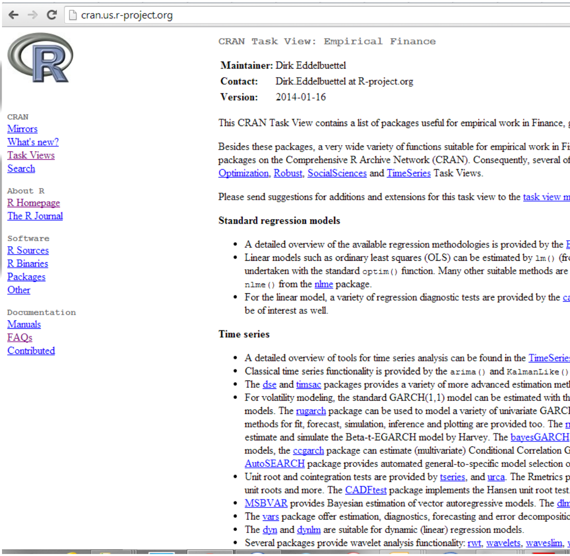
Reverse depends:	ivpack, rdd
Reverse suggests:	censReg, glmx, lmtest, micEconCES, mlogit, plm, REEMtree, sandwich

CRAN Task Views

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

Note:

- Maintainer,
- Topics,
- List of packages.


cran.us.r-project.org

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

CRAN Task View: Empirical Finance

Maintainer: Dirk Eddelbuettel
Contact: Dirk.Eddelbuettel at R-project.org
Version: 2014-01-16

This CRAN Task View contains a list of packages useful for empirical work in Finance, g

Besides these packages, a very wide variety of functions suitable for empirical work in Fi
packages on the Comprehensive R Archive Network (CRAN). Consequently, several of
Optimization, Robust, SocialSciences and TimeSeries Task Views.

Please send suggestions for additions and extensions for this task view to the task view m

Standard regression models

- A detailed overview of the available regression methodologies is provided by the E
- Linear models such as ordinary least squares (OLS) can be estimated by `lm()` (fro
undertaken with the standard `optim()` function. Many other suitable methods are
`nlme()` from the nlme package.
- For the linear model, a variety of regression diagnostic tests are provided by the ca
be of interest as well.

Time series

- A detailed overview of tools for time series analysis can be found in the TimeSeries
- Classical time series functionality is provided by the `arima()` and `KalmanLike()`
- The dse and timsac packages provides a variety of more advanced estimation met
- For volatility modeling, the standard GARCH(1,1) model can be estimated with th
models. The rugarch package can be used to model a variety of univariate GARCH
methods for fit, forecast, simulation, inference and plotting are provided too. The r
estimate and simulate the Beta-t-EGARCH model by Harvey. The bayesGARCH
models, the ccgarch package can estimate (multivariate) Conditional Correlation G
AutoSEARCH package provides automated general-to-specific model selection o
- Unit root and cointegration tests are provided by tseries, and urca. The Rmetrics p
unit roots and more. The CADFtest package implements the Hansen unit root test
- MSBVAR provides Bayesian estimation of vector autoregressive models. The dlm
- The vars package offer estimation, diagnostics, forecasting and error decompositio
- The dyn and dynlm are suitable for dynamic (linear) regression models.
- Several packages provide wavelet analysis functionality: rwt, wavelets, waveslim, v

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 "babynamev" from GitHub
> install_github(repo="hadley/babynamev")
```

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
> packinfo <- installed.packages()
> dim(packinfo)
> # get all installed package names
> sort(unname(packinfo[, "Package"]))
> # get a few package names and their versions
> packinfo[sample(x=1:100, 5), c("Package", "Version")]
> # get info for package "xts"
> t(packinfo["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")))
character(0)
```

Loading Packages

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

Loading a package means attaching the package *namespace* 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 *namespaces*:

- 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 *namespace* 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 *namespace* being attached to the search path.

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

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

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

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

```
> 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()` lists those packages as "loaded via a *namespace* (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 namespaces*,
- 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 *namespace*, 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)
> vecv <- runif(1e6)
> # sqrt() and "^0.5" are the same
> all.equal(sqrt(vecv), vecv^0.5)
> # sqrt() is much faster than "^0.5"
> system.time(vecv^0.5)
> microbenchmark(
+   power = vecv^0.5,
+   sqrt = sqrt(vecv),
+   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
> matv <- matrix(rnorm(10000), ncol=2)
> # Allocate memory for row sums
> rowsumv <- numeric(NROW(matv))
> summary(microbenchmark(
+   rowsumv = rowSums(matv),  # end rowsumv
+   applyloop = apply(matv, 1, sum),  # end apply
+   lapply = lapply(1:NROW(matv), function(indeks)
+     sum(matv[indeks, ])),  # end lapply
+   vapply = vapply(1:NROW(matv), function(indeks)
+     sum(matv[indeks, ]),
+     FUN.VALUE = c(sum=0)),  # end vapply
+   sapply = sapply(1:NROW(matv), function(indeks)
+     sum(matv[indeks, ])),  # end sapply
+   forloop = for (i in 1:NROW(matv)) {
+     rowsumv[i] <- sum(matv[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 lists, 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).

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

```
> vec1 <- rnorm(1000000)
> vec2 <- rnorm(1000000)
> vecbig <- numeric(1000000)
> # Sum two vectors in two different ways
> summary(microbenchmark(
+   # Sum vectors using "for" loop
+   rloop = (for (i in 1:NROW(vec1)) {
+     vecbig[i] <- vec1[i] + vec2[i]
+   }),
+   # Sum vectors using vectorized "+"
+   vectorized = (vec1 + vec2),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Allocate memory for cumulative sum
> cumsumv <- numeric(NROW(vecbig))
> cumsumv[1] <- vecbig[1]
> # Calculate cumulative sum in two different ways
> summary(microbenchmark(
+ # Cumulative sum using "for" loop
+   rloop = (for (i in 2:NROW(vecbig)) {
+     cumsumv[i] <- cumsumv[i-1] + vecbig[i]
+   }),
+ # Cumulative sum using "cumsum"
+   vectorized = cumsum(vecbig),
+   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
> matv <- matrix(rnorm(10000), ncol=2)
> # Calculate row sums two different ways
> all.equal(rowSums(matv), apply(matv, 1, sum))
> summary(microbenchmark(
+   rowsumv = rowSums(matv),
+   applyloop = apply(matv, 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(matv[1, ]),
+   function(indeks) matv[, indeks])),
+   applyloop=unlist(lapply(seq_along(matv[, 1]),
+   function(indeks) max(matv[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(matv),
+   pmin =
+     do.call(pmin.int,
+       lapply(seq_along(matv[1, ]),
+              function(indeks)
+                matv[, indeks])),
+   as_dframe =
+     do.call(pmin.int,
+       as.data.frame.matrix(matv)),
+   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
> vecv <- 1e3
> all.equal(rank(vecv), Rfast::Rank(vecv))
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = rank(vecv),
+   Rfast = Rfast::Rank(vecv),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Benchmark speed of calculating column medians
> matv <- matrix(1e4, nc=10)
> all.equal(matrixStats::colMedians(matv), Rfast::colMedians(matv))
> summary(microbenchmark(
+   matrixStats = matrixStats::colMedians(matv),
+   Rfast = Rfast::colMedians(matv),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

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 = {vecv <- numeric(10)
+     vecv[] <- 2},
+ # Slow because loop is performed in R
+   forloop = {vecv <- numeric(10)
+     for (indeks in seq_along(vecv))
+       vecv[indeks] <- 2},
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> summary(microbenchmark(  # Assign values to vector two different w
+ # Fast vectorized assignment loop performed in C using brackets "
+   brackets = {vecv <- numeric(10)
+     vecv[4:7] <- rnorm(4)},
+ # Slow because loop is performed in R
+   forloop = {vecv <- numeric(10)
+     for (indeks in 4:7)
+       vecv[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, "Mersenne-Twister", sample.kind="Rejection")
> sapply(stdevs, function(stdev) rnorm(n=2, sd=stdev))
> # Same
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> sapply(stdevs, rnorm, n=2, mean=0)
> # Loop over means
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> sapply(means, function(meanv) rnorm(n=2, mean=meanv))
> # Same
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> 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, "Mersenne-Twister", sample.kind="Rejection")
> 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, "Mersenne-Twister", sample.kind="Rejection")
> vec_rnorm(n=2, sd=stdevs)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> 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:

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

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.

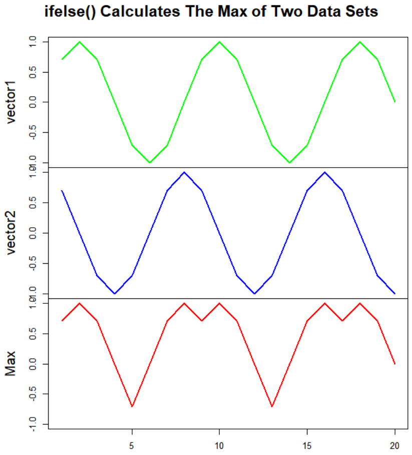
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2,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

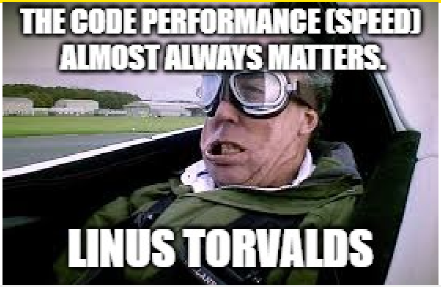
```
> # Create two numeric vectors
> vec1 <- sin(0.25*pi*1:20)
> vec2 <- cos(0.25*pi*1:20)
> # Create third vector using 'ifelse'
> vec3 <- ifelse(vec1 > vec2, vec1, vec2)
> # cbind all three together
> vec3 <- cbind(vec1, vec2, vec3)
> colnames(vec3)[3] <- "Max"
> # Set plotting parameters
> x11(width=6, 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(vec3, 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
> vecv <- runif(1e5)
> # Use compiled function
> cumsumv <- cumsum(vecv)
> # Use for loop
> cumsumv2 <- vecv
> for (i in 2:NROW(vecv))
+   cumsumv2[i] <- (vecv[i] + cumsumv2[i-1])
> # Compare the outputs of the two methods
> all.equal(cumsumv, cumsumv2)
> # Microbenchmark the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   cumsum=cumsum(vecv), # Vectorized
+   loop_alloc={cumsumv2 <- vecv # Allocate memory to cumsumv3
+     for (i in 2:NROW(vecv))
+ cumsumv2[i] <- (vecv[i] + cumsumv2[i-1])
+   },
+   loop_nalloc={cumsumv3 <- vecv[1] # Doesn't allocate memory to cu
+     for (i in 2:NROW(vecv))
+ cumsumv3[i] <- (vecv[i] + cumsumv3[i-1])
```

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
> compclust <- makeCluster(ncores)
> # Perform parallel loop under Windows
> outv <- parLapply(compclust, 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 = parLapply(compclust, 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.

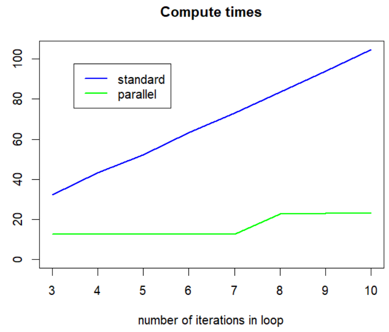
The function `stopCluster()` stops the R processes running on several CPU cores.

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
> runv <- 3:10
> timev <- sapply(runv, function(nruns) {
+     summary(microbenchmark(
+ standard = lapply(1:nruns, paws),
+ # parallel = parLapply(compclust, 1:nruns, paws),
+ parallel = mclapply(1:nruns, paws, mc.cores=ncores),
+ times=10))[, 4]
+     })  # end sapply
> timev <- t(timev)
> colnames(timev) <- c("standard", "parallel")
> rownames(timev) <- runv
> # Stop R processes over cluster under Windows
> stopCluster(compclust)
```



```
> x11(width=6, height=5)
> plot(x=rownames(timev),
+      y=timev[, "standard"],
+      type="l", lwd=2, col="blue",
+      main="Compute times",
+      xlab="Number of iterations in loop", ylab="",
+      ylim=c(0, max(timev[, "standard"])))
> lines(x=rownames(timev),
+ y=timev[, "parallel"], lwd=2, col="green")
> legend(x="topleft", legend=colnames(timev),
+  inset=0.1, cex=1.0, bty="n", bg="white",
+  y.intersp=0.3, 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
> matv <- matrix(rnorm(1e5), ncol=100)
> # Define aggregation function over column of matrix
> aggfun <- function(column) {
+   datav <- 0
+   for (indeks in 1:NROW(column))
+     datav <- datav + column[indeks]
+   datav
+ }  # end aggfun
> # Perform parallel aggregations over columns of matrix
> aggs <- parCapply(compclust, matv, aggfun)
> # Compare speed of apply with parallel computing
> summary(microbenchmark(
+   apply=apply(matv, MARGIN=2, aggfun),
+   parapply=parCapply(compclust, matv, aggfun),
+   times=10)
+ )[, c(1, 4, 5)]
> # Stop R processes over cluster under Windows
> stopCluster(compclust)
```

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

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
> compclust <- makeCluster(ncores)
> # Set seed for cluster under Windows
> # Doesn't work: set.seed(1121, "Mersenne-Twister", sample.kind="Re
> clusterSetRNGStream(compclust, 1121)
> # Perform parallel loop under Windows
> datav <- parLapply(compclust, 1:10, rnorm, n=100)
> sum(unlist(datav))
> # Stop R processes over cluster under Windows
> stopCluster(compclust)
> # Perform parallel loop under Mac-OSX or Linux
> datav <- 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.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Sample from Standard Normal Distribution
> nsimu <- 1000
> datav <- rnorm(nsimu)
> # Sample mean - MC estimate
> mean(datav)
> # Sample standard deviation - MC estimate
> sd(datav)
> # Monte Carlo estimate of cumulative probability
> pnorm(-2)
> sum(datav < (-2))/nsimu
> # Monte Carlo estimate of quantile
> confl <- 0.02
> qnorm(confl)  # Exact value
> cutoff <- confl*nsimu
> 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],
+   quantv = 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
> nsimu <- 1000; datav <- rnorm(nsimu)
> # 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(nsimu)
+   c(mean=mean(samplev), median=median(samplev))
+ })  # end sapply
> bootd[, 1:3]
> bootd <- t(bootd)
> # Standard error from formula
> sd(datav)/sqrt(nsimu)
> # 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

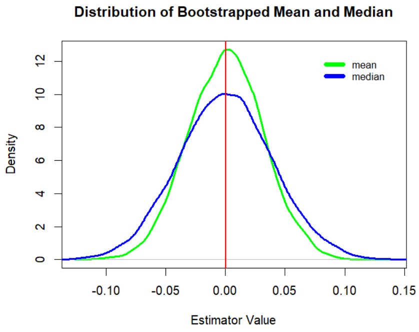
The standard errors of estimators can be calculated using a *bootstrap* simulation.

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", y.intersp=0.4,
+  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.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nsimu <- 1000
> # Bootstrap of sample mean and median
> nboot <- 100
> bootd <- sapply(1:nboot, function(x) median(rnorm(nsimu)))
> # Perform vectorized bootstrap
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Calculate matrix of random data
> samplev <- matrix(rnorm(nboot*nsimu), ncol=nboot)
> bootv <- matrixStats::colMedians(samplev)
> all.equal(bootd, bootv)
> # Compare speed of loops with vectorized R code
> library(microbenchmark)
> summary(microbenchmark(
+   loop = sapply(1:nboot, function(x) median(rnorm(nsimu))),
+   cpp = {
+     samplev <- matrix(rnorm(nboot*nsimu), ncol=nboot)
+     matrixStats::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
> compclust <- makeCluster(ncores)  # Initialize compute cluster und
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Sample from Standard Normal Distribution
> nsimu <- 1000
> # Bootstrap mean and median under Windows
> nboot <- 10000
> bootd <- parLapply(compclust, 1:nboot, function(x, datav, nsimu) {
+   samplev <- rnorm(nsimu)
+   c(mean=mean(samplev), median=median(samplev))
+ }, datav=datav, nsimu=nsimu)  # end parLapply
> # Bootstrap mean and median under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+   samplev <- rnorm(nsimu)
+   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(nsimu)
> stopCluster(compclust)  # Stop R processes over cluster under Win
```

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:

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

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.

```
> nsimu <- 1000
> datav <- rnorm(nsimu)
> 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(nsimu)
+   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
> compclust <- makeCluster(ncores)  # Initialize compute cluster
> bootd <- parLapply(compclust, 1:nboot, function(x, datav) {
+   samplev <- rnorm(nsimu)
+   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(nsimu)
+   c(sd=sd(samplev), mad=mad(samplev))
+ }, mc.cores=ncores)  # end mclapply
> stopCluster(compclust)  # 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)
> retp <- rutils::etfenv$returns$VTI
> retp <- na.omit(retp)
> nrows <- NROW(retp)
> # Sample from VTI returns
> samplev <- retp[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)
> retp <- rutils::etfenv$returns$VTI
> retp <- na.omit(retp)
> nrows <- NROW(retp)
> # Bootstrap sd and MAD under Windows
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster und
> clusterSetRNGStream(compclust, 1121)  # Reset random number genera
> nboot <- 10000
> bootd <- parLapply(compclust, 1:nboot, function(x, retp, nsimu) {
+   samplev <- retp[sample.int(nsimu, replace=TRUE)]
+   c(sd=sd(samplev), mad=mad(samplev))
+ }, retp=retp, nsimu=nsimu)  # end parLapply
> # Bootstrap sd and MAD under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+   samplev <- retp[sample.int(nsimu, replace=TRUE)]
+   c(sd=sd(samplev), mad=mad(samplev))
+ }, mc.cores=ncores)  # end mclapply
> stopCluster(compclust)  # Stop R processes over cluster under Wind
> bootd <- rutils::do_call(rbind, bootd)
> # Standard error of standard deviation assuming normal distributio
> sd(retp)/sqrt(nsimu)
> # 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, "Mersenne-Twister", sample.kind="Rejection")
> # Define predictor and response variables
> nsimu <- 100
> predm <- rnorm(nsimu, mean=2)
> noisev <- rnorm(nsimu)
> respv <- (-3 + 2*predm + noisev)
> desm <- cbind(respv, predm)
> # Calculate alpha and beta regression coefficients
> betac <- cov(desm[, 1], desm[, 2])/var(desm[, 2])
> alphac <- mean(desm[, 1]) - betac*mean(desm[, 2])
> x11(width=6, height=5)
> plot(respv ~ predm, data=desm)
> abline(a=alphac, b=betac, lwd=3, col="blue")
> # Bootstrap of beta regression coefficient
> nboot <- 100
> bootd <- sapply(1:nboot, function(x) {
+   samplev <- sample.int(nsimu, replace=TRUE)
+   desm <- desm[samplev, ]
+   cov(desm[, 1], desm[, 2])/var(desm[, 2])
+ })  # end sapply
```

Distribution of Bootstrapped Regression Coefficients

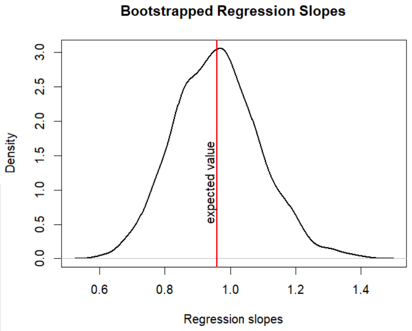
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

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
> compclust <- makeCluster(ncores)  # Initialize compute cluster und
> # Bootstrap of regression under Windows
> bootd <- parLapply(compclust, 1:1000, function(x, desm) {
+   samplev <- sample.int(nsimu, replace=TRUE)
+   desm <- desm[samplev, ]
+   cov(desm[, 1], desm[, 2])/var(desm[, 2])
+ }, desm=desm)  # end parLapply
> # Bootstrap of regression under Mac-OSX or Linux
> bootd <- mclapply(1:1000, function(x) {
+   samplev <- sample.int(nsimu, replace=TRUE)
+   desm <- desm[samplev, ]
+   cov(desm[, 1], desm[, 2])/var(desm[, 2])
+ }, mc.cores=ncores)  # end mclapply
> stopCluster(compclust)  # Stop R processes over cluster under Win
```

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).

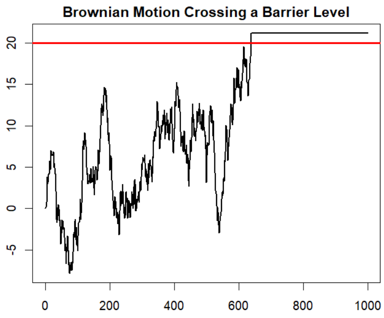
```
> # Collapse the bootstrap list into a vector
> class(bootd)
> bootd <- unlist(bootd)
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd),
+      lwd=2, xlab="Regression slopes",
+      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.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> barl <- 20  # Barrier level
> nsteps <- 1000  # Number of simulation steps
> pathv <- numeric(nsteps)  # Allocate path vector
> pathv[1] <- rnorm(1)  # Initialize path
> it <- 2  # Initialize simulation index
> while ((it <= nsteps) && (pathv[it - 1] < barl)) {
+ # Simulate next step
+   pathv[it] <- pathv[it - 1] + rnorm(1)
+   it <- it + 1  # Advance index
+ }  # end while
> # Fill remaining path after it crosses barl
> if (it <= nsteps)
+   pathv[it:nsteps] <- pathv[it - 1]
> # 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)
```

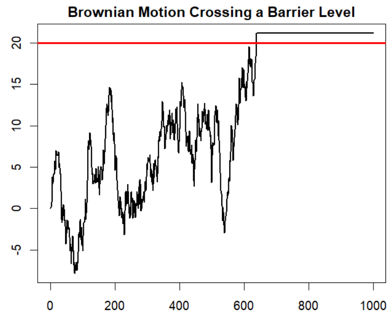


Simulating Brownian Motion Using Vectorized Functions

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.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> barl <- 20  # Barrier level
> nsteps <- 1000  # Number of simulation steps
> # Simulate path of Brownian motion
> pathv <- cumsum(rnorm(nsteps))
> # Find index when path crosses barl
> crossp <- which(pathv > barl)
> # Fill remaining path after it crosses barl
> if (NROW(crossp) > 0) {
+   pathv[(crossp[1]+1):nsteps] <- 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,

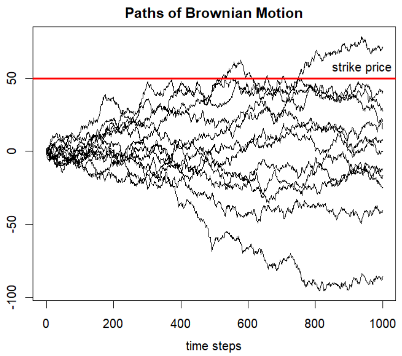
Estimating the Statistics of Brownian Motion

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
> nsteps <- 1000  # Number of simulation steps
> npaths <- 100  # Number of simulation paths
> # Simulate multiple paths of Brownian motion
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> pathm <- rnorm(npaths*nsteps, mean=drift, sd=sigmav)
> pathm <- matrix(pathm, nc=npaths)
> pathm <- matrixStats::colCumsums(pathm)
> # Final distribution of paths
> mean(pathm[nsteps, ]) ; sd(pathm[nsteps, ])
> # Calculate option payout at maturity
> strikep <- 50  # Strike price
> payouts <- (pathm[nsteps, ] - strikep)
> sum(payouts[payouts > 0])/npaths
> # Calculate probability of crossing the barrier at any point
> barl <- 50
> crossi <- (colSums(pathm > barl) > 0)
> sum(crossi)/npaths
```



```
> # 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(pathm[nsteps, ])
> pathm[nsteps, ordern]
> indeks <- ordern[seq(1, 100, 9)]
> zoo::plot.zoo(pathm[, indeks], main="Paths of Brownian Motion",
+   xlab="time steps", ylab=NA, plot.type="single")
> abline(h=strikep, col="red", lwd=3)
> text(x=(nsteps-60), y=strikep, labels="strike price", pos=3, cex=
```

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

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster und
> # Perform parallel bootstrap under Windows
> clusterSetRNGStream(compclust, 1121)  # Reset random number genera
> clusterExport(compclust, c("prici", "barl"))
> nboot <- 10000
> bootd <- parLapply(compclust, 1:nboot, function(x, retp, nrows) {
+   samplev <- retp[sample.int(nrows, replace=TRUE)]
+   # Calculate prices from percentage returns
+   samplev <- prici*exp(cumsum(samplev))
+   # Calculate if prices crossed barrier
+   sum(samplev > barl) > 0
+ }, retp=retp, nrows=nrows)  # end parLapply
> stopCluster(compclust)  # Stop R processes over cluster under Winc
> # Perform parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+   samplev <- retp[sample.int(nrows, replace=TRUE)]
+   # Calculate prices from percentage returns
+   samplev <- prici*exp(cumsum(samplev))
+   # Calculate if prices crossed barrier
+   sum(samplev > barl) > 0
+ }, mc.cores=ncores)  # end mclapply
> bootd <- rutils::do_call(c, 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
> pricev <- as.numeric(ohlc[, 4])
> prici <- pricev[1]
> retp <- rutils::diffit(log(pricev))
> nrows <- NROW(retp)
> # Calculate difference of OHLC price columns
> pricediff <- ohlc[, 1:3] - pricev
> class(retp); head(retp)
> # Calculate bootstrap prices from percentage returns
> datav <- sample.int(nrows, replace=TRUE)
> priceboot <- prici*exp(cumsum(retp[datav]))
> ohlcboot <- pricediff + priceboot
> ohlcboot <- cbind(ohlcboot, priceboot)
> # Define barrier level with respect to prices
> barl <- 1.5*max(pricev)
> # Calculate if High bootstrapped prices crossed barrier level
> sum(ohlcboot[, 2] > barl) > 0
```

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

Variance Reduction Using Antithetic Sampling

Variance reduction are techniques for increasing the precision of Monte Carlo simulations.

Naive Monte Carlo refers to *Monte Carlo* simulation without using *variance reduction* techniques.

Antithetic Sampling is a *variance reduction* technique in which a new random sample is computed from an existing sample, without generating new random numbers.

In the case of a *Normal* random sample ϕ, the new *antithetic* sample is equal to minus the existing sample: $\phi_{new} = -\phi$.

In the case of a *Uniform* random sample ϕ, the new *antithetic* sample is equal to 1 minus the existing sample: $\phi_{new} = 1 - \phi$.

Antithetic Sampling doubles the number of independent samples, so it reduces the standard error by $\sqrt{2}$.

Antithetic Sampling doesn't change any other parameters of the simulation.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Sample from Standard Normal Distribution
> nsimu <- 1000
> datav <- rnorm(nsimu)
> # Estimate the 95% quantile
> nboot <- 10000
> bootd <- sapply(1:nboot, function(x) {
+   samplev <- datav[sample.int(nsimu, replace=TRUE)]
+   quantile(samplev, 0.95)
+ })  # end sapply
> sd(bootd)
> # Estimate the 95% quantile using antithetic sampling
> bootd <- sapply(1:nboot, function(x) {
+   samplev <- datav[sample.int(nsimu, replace=TRUE)]
+   quantile(c(samplev, -samplev), 0.95)
+ })  # end sapply
> # Standard error of quantile from bootstrap
> sd(bootd)
> sqrt(2)*sd(bootd)
```

Simulating Rare Events Using Probability Tilting

Rare events can be simulated more accurately by *tilting* (deforming) their probability distribution, so that rare events occur more frequently.

A popular probability *tilting* method is exponential (Esscher) tilting:

$$p(x, \lambda) = \frac{\exp(\lambda x) p(x)}{\int_{-\infty}^{\infty} \exp(\lambda x) p(x) dx}$$

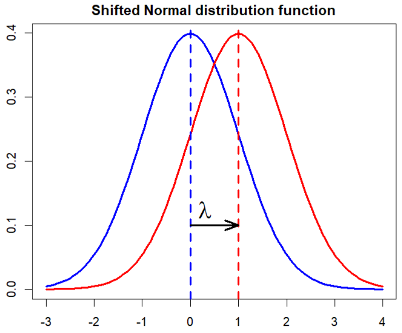
Where $p(x)$ is the probability density, $p(x, \lambda)$ is the tilted density, and λ is the tilt parameter.

For the *Normal* distribution $\phi(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}$, exponential tilting is equivalent to shifting the distribution by λ: $x \rightarrow x + \lambda$.

$$\phi(x, \lambda) = \frac{\exp(\lambda x) \exp(-x^2/2)}{\int_{-\infty}^{\infty} \exp(\lambda x) \exp(-x^2/2) dx} =$$

$$\frac{\exp(-(x - \lambda)^2/2)}{\sqrt{2\pi}} = \exp(x\lambda - \lambda^2/2) \cdot \phi(x, \lambda = 0)$$

Shifting the random variable $x \rightarrow x + \lambda$ is equivalent to multiplying the distribution by the weight factor: $\exp(x\lambda - \lambda^2/2)$.



```
> # Plot a Normal probability distribution
> curve(expr=dnorm, xlim=c(-3, 4),
+ main="Shifted Normal distribution function",
+ xlab="", ylab="", lwd=3, col="blue")
> # Add shifted Normal probability distribution
> curve(expr=dnorm(x, mean=1), add=TRUE, lwd=3, col="red")
> # Add vertical dashed lines
> abline(v=0, lwd=3, col="blue", lty="dashed")
> abline(v=1, lwd=3, col="red", lty="dashed")
> arrows(x0=0, y0=0.1, x1=1, y1=0.1, lwd=3,
+  code=2, angle=20, length=grid::unit(0.2, "cm"))
> text(x=0.3, 0.1, labels=bquote(lambda), pos=3, cex=2)
```

Variance Reduction Using Importance Sampling

Importance sampling is a *variance reduction* technique for simulating rare events more accurately.

The *variance* of an estimate produced by simulation decreases with the number of events which contribute to the estimate: $\sigma^2 \propto \frac{1}{n}$.

Importance sampling simulates rare events more frequently by *tilting* the probability distribution, so that more events contribute to the estimate.

In standard Monte Carlo simulation, the simulated data points have equal probabilities.

But in *importance sampling*, the simulated data must be weighted (multiplied) to compensate for the tilting of the probability.

The tilt weights are equal to the ratio of the base probability distribution divided by the tilted distribution, which for the *Normal* distribution are equal to:

$$w_x = \frac{\phi(x, \lambda = 0)}{\phi(x, \lambda)} = \exp(-x\lambda + \lambda^2/2)$$

```
> # Sample from Standard Normal Distribution
> nsimu <- 1000
> datav <- rnorm(nsimu)
> # Cumulative probability from formula
> quantv <- (-2)
> pnorm(quantv)
> integrate(dnorm, lower=-Inf, upper=quantv)
> # Cumulative probability from Naive Monte Carlo
> sum(datav < quantv)/nsimu
> # Generate importance sample
> lambdaf <- (-1.5)  # Tilt parameter
> datat <- datav + lambdaf  # Tilt the random numbers
> # Cumulative probability from importance sample - wrong!
> sum(datat < quantv)/nsimu
> # Cumulative probability from importance sample - correct
> weightv <- exp(-lambdaf*datat + lambdaf^2/2)
> sum((datat < quantv)*weightv)/nsimu
> # Bootstrap of standard errors of cumulative probability
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   datav <- rnorm(nsimu)
+   naivemc <- sum(datav < quantv)/nsimu
+   datav <- (datav + lambdaf)
+   weightv <- exp(-lambdaf*datav + lambdaf^2/2)
+   isample <- sum((datav < quantv)*weightv)/nsimu
+   c(naivemc=naivemc, impsample=isample)
+ }) # end sapply
> apply(bootd, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
```

Calculating Quantiles Using Importance Sampling

The quantiles can be calculated from the cumulative probabilities of the importance sample data.

The importance sample data points must be weighted to compensate for the tilting of the probability.

Importance sampling can be used to estimate the *VaR* (*quantile*) corresponding to a given *confidence level*.

The standard error of the *VaR* estimate using importance sampling can be several times smaller than that of *naive Monte Carlo*.

The reduction of standard error is greater for higher *confidence levels*.

Naive Monte Carlo refers to *Monte Carlo* simulation without using *variance reduction* techniques.

The function `findInterval()` returns the indices of the intervals specified by "vec" that contain the elements of "x".

```
> # Quantile from Naive Monte Carlo
> confl <- 0.02
> qnorm(confl)  # Exact value
> datav <- sort(datav)  # Must be sorted for importance sampling
> cutoff <- nsimu*confl
> datav[cutoff]  # Naive Monte Carlo value
> # Importance sample weights
> datat <- datav + lambdaf  # Tilt the random numbers
> weightv <- exp(-lambdaf*datat + lambdaf^2/2)
> # Cumulative probabilities using importance sample
> cumprob <- cumsum(weightv)/nsimu
> # Quantile from importance sample
> datat[findInterval(confl, cumprob)]
> # Bootstrap of standard errors of quantile
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   datav <- sort(rnorm(nsimu))
+   naivemc <- datav[cutoff]
+   datat <- datav + lambdaf
+   weightv <- exp(-lambdaf*datat + lambdaf^2/2)
+   cumprob <- cumsum(weightv)/nsimu
+   isample <- datat[findInterval(confl, cumprob)]
+   c(naivemc=naivemc, impsample=isample)
+ }) # end sapply
> apply(bootd, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
```

Calculating *CVaR* Using Importance Sampling

Importance sampling can be used to estimate the Conditional Value at Risk (*CVaR*) corresponding to a given *confidence level*.

First the *VaR* (*quantile*) is estimated, and then the *expected value* (*CVaR*) is estimated using it.

The standard error of the *CVaR* estimate using importance sampling can be several times smaller than that of *naive Monte Carlo*.

The reduction of standard error is greater for higher *confidence levels*.

```
> # VaR and CVaR from Naive Monte Carlo
> varisk <- datav[cutoff]
> sum((datav <= varisk)*datav)/sum((datav <= varisk))
> # CVaR from importance sample
> varisk <- datat[findInterval(confl, cumprob)]
> sum((datat <= varisk)*datat*weightv)/sum((datat <= varisk)*weightv
> # CVaR from integration
> integrate(function(x) x*dnorm(x), low=-Inf, up=varisk)$value/pnorm
> # Bootstrap of standard errors of CVaR
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   datav <- sort(rnorm(nsimu))
+   varisk <- datav[cutoff]
+   naivemc <- sum((datav <= varisk)*datav)/sum((datav <= varisk))
+   datat <- datav + lambdaf
+   weightv <- exp(-lambdaf*datat + lambdaf^2/2)
+   cumprob <- cumsum(weightv)/nsimu
+   varisk <- datat[findInterval(confl, cumprob)]
+   isample <- sum((datat <= varisk)*datat*weightv)/sum((datat <= va
+   c(naivemc=naivemc, impsample=isample)
+ }) # end sapply
> apply(bootd, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
```

The Optimal Tilt Parameter for Importance Sampling

The tilt parameter λ should be chosen to minimize the standard error of the estimator.

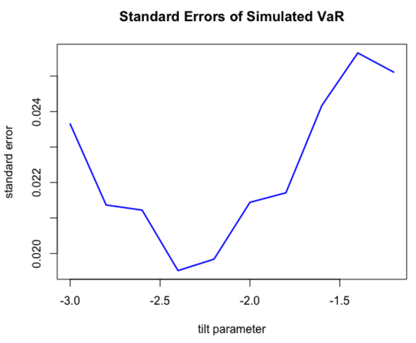
The optimal tilt parameter depends on the estimator and on the required confidence level.

More tilting is needed at higher confidence levels, to provide enough significant data points.

When performing a loop over the tilt parameters, the same matrix of random data can be used for different tilt parameters.

The function `Rfast::colSort()` sorts the columns of a matrix using very fast C++ code.

```
> # Calculate matrix of random data
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection") # Rese
> nsimu <- 1000; nboot <- 100
> datav <- matrix(rnorm(nboot*nsimu), ncol=nboot)
> datav <- Rfast::colSort(datav)  # Sort the columns
> # Bootstrap function for VaR (quantile) for a single tilt parame
> calc_vars <- function(lambdaf, confl=0.05) {
+   datat <- datav + lambdaf  # Tilt the random numbers
+   weightv <- exp(-lambdaf*datat + lambdaf^2/2)
+   # Calculate quantiles for columns
+   sapply(1:nboot, function(it) {
+     cumprob <- cumsum(weightv[, it])/nsimu
+     datat[findInterval(confl, cumprob), it]
+   })  # end sapply
+ }  # end calc_vars
> # Bootstrap vector of VaR for a single tilt parameter
> bootd <- calc_vars(-1.5)
```



```
> # Define vector of tilt parameters
> lambdav <- seq(-3.0, -1.2, by=0.2)
> # Calculate vector of VaR for vector of tilt parameters
> varisk <- sapply(lambdav, calc_vars, confl=0.02)
> # Calculate standard deviations of VaR for tilt parameters
> stdevs <- apply(varisk, MARGIN=2, sd)
> # Calculate the optimal tilt parameter
> lambdav[which.min(stdevs)]
> # Plot the standard deviations
> x11(width=6, height=5)
> plot(x=lambdav, y=stdevs,
+      main="Standard Errors of Simulated VaR",
+      xlab="tilt parameter", ylab="standard error",
+      type="l", col="blue", lwd=2)
```

draft: Importance Sampling For Empirical Datasets

The quantiles can be calculated from the cumulative probabilities of the importance sample data.

The importance sample data points must be weighted to compensate for the tilting of the probability.

Importance sampling can be used to estimate the *VaR* (*quantile*) corresponding to a given *confidence level*.

The standard error of the *VaR* estimate using importance sampling can be several times smaller than that of *naive Monte Carlo*.

The reduction of standard error is greater for higher *confidence levels*.

Naive Monte Carlo refers to *Monte Carlo* simulation without using *variance reduction* techniques.

Importance Sampling for Binomial Variables

The probability p of a binomial variable can be tilted to $p(\lambda)$ as follows:

$$p(\lambda) = \frac{\lambda p}{1 + p(\lambda - 1)}$$

Where λ is the tilt parameter.

The weight is equal to the ratio of the base probability divided by the tilted probability:

$$w = \frac{1 + p(\lambda - 1)}{\lambda}$$

```
> # Binomial sample
> nsimu <- 1000
> probv <- 0.1
> datav <- rbinom(n=nsimu, size=1, probv)
> head(datav, 33)
> # Tilted binomial sample
> lambdaf <- 5
> probt <- lambdaf*probv/(1 + probv*(lambdaf - 1))
> weightv <- (1 + probv*(lambdaf - 1))/lambdaf
> datav <- rbinom(n=nsimu, size=1, probt)
> head(datav, 33)
> weightv*sum(datav)/nsimu
> # Bootstrap of standard errors
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   c(naivemc=sum(rbinom(n=nsimu, size=1, probv))/nsimu,
+     impsample=weightv*sum(rbinom(n=nsimu, size=1, probt))/nsimu)
+ }) # end sapply
> apply(bootd, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
```

Importance Sampling of Brownian Motion

The statistics that depend on extreme paths of Brownian motion can be simulated more accurately using *importance sampling*.

The normally distributed variables x_i are shifted by the tilt parameter λ to obtain the importance sample variables x_i^{tilt}: $x_i^{tilt} = x_i + \lambda$.

The Brownian paths p_t are equal to the cumulative sums of the tilted variables x_i^{tilt}: $p_t = \sum_{i=1}^{t} x_i^{tilt}$.

Each tilted Brownian path has an associated weight factor equal to the product: $\prod_{i=1}^{t} \exp(-x_i^{tilt} \lambda + \lambda^2/2)$.

To compensate for the probability tilting, the statistics derived from the tilted Brownian paths must be multiplied by their weight factors.

```
> # Define Brownian motion parameters
> sigmav <- 1.0  # Volatility
> drift <- 0.0  # Drift
> nsteps <- 100  # Number of simulation steps
> nsimu <- 10000  # Number of simulation paths
> # Calculate matrix of normal variables
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> datav <- rnorm(nsimu*nsteps, mean=drift, sd=sigmav)
> datav <- matrix(datav, nc=nsimu)
> # Simulate paths of Brownian motion
> pathm <- matrixStats::colCumsums(datav)
> # Tilt the datav
> lambdaf <- 0.04  # Tilt parameter
> datat <- datav + lambdaf  # Tilt the random numbers
> patht <- matrixStats::colCumsums(datat)
> # Calculate path weights
> weightm <- exp(-lambdaf*datat + lambdaf^2/2)
> weightm <- matrixStats::colProds(weightm)
> # Or
> weightm <- exp(-lambdaf*colSums(datat) + nsteps*lambdaf^2/2)
> # Calculate option payout using naive MC
> strikep <- 10  # Strike price
> payouts <- (pathm[nsteps, ] - strikep)
> sum(payouts[payouts > 0])/nsimu
> # Calculate option payout using importance sampling
> payouts <- (patht[nsteps, ] - strikep)
> sum((weightm*payouts)[payouts > 0])/nsimu
> # Calculate crossing probability using naive MC
> barl <- 10
> crossi <- (colSums(pathm > barl) > 0)
> sum(crossi)/nsimu
> # Calculate crossing probability using importance sampling
> crossi <- colSums(patht > barl) > 0
> sum(weightm*crossi)/nsimu
```