FRE7241 Algorithmic Portfolio Management

Lecture#1, Fall 2021

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 7, 2021

Welcome Students!

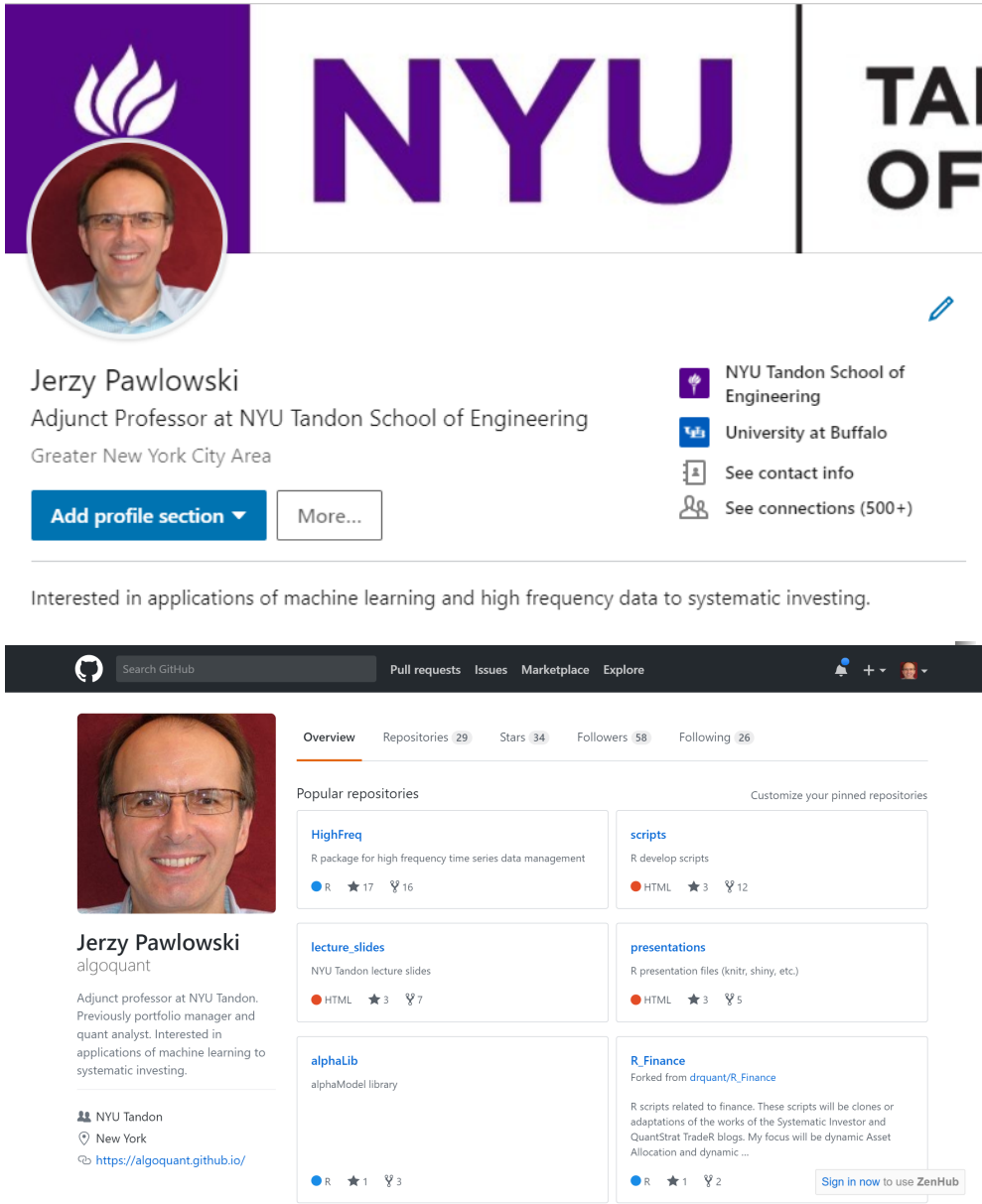
My name is Jerzy Pawlowski jp3900@nyu.edu

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

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

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

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



FRE7241 Course Description and Objectives

Course Description

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

FRE7241 Course Description and Objectives

Course Description

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

Course Objectives

Students will learn through R coding exercises how to:

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

FRE7241 Course Description and Objectives

Course Description

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

Course Objectives

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

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

Course Prerequisites

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

Homeworks and Tests

Homeworks and Tests

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

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

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

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

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

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

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

Homeworks and Tests

Homeworks and Tests

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

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

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

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

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

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

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

Graduate Assistant

The graduate assistant (GA) will be Shardha Koul sk9225@nyu.edu.
The GA will answer questions during office hours, or via *Brightspace* forums, not via emails. Please send emails regarding lecture matters from *Brightspace* (not personal emails).

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

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

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

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

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

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

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

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

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

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

Please Submit *Minimal Working Examples* With Your Questions

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

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

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

Course Grading Policies

Numerical Scores

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

Course Grading Policies

Numerical Scores

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

Letter Grades

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

Course Grading Policies

Numerical Scores

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

Letter Grades

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

Plagiarism

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

FRE7241 Course Materials

Lecture Slides

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

FRE7241 Course Materials

Lecture Slides

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

FRE7241 Recommended Textbooks

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

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

FRE7241 Supplementary Books

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

FRE7241 Supplementary Materials

Robert Carver's trading blog

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

Introduction to Computational Finance with R

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

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

Notepad++ has a very convenient and fast *search and replace* function, that allows *search and replace* in multiple files.

http://notepad-plus-plus.org/

Internal R Help and Documentation

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

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

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

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

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

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

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

Introduction to R by Venables and R Core Team.

R Online Help and Documentation

R Cheat Sheets

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

R Programming Wikibook

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

R FAQ

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

R-seek Online Search Tool

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

R-help Mailing List

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

R Style Guides

DataCamp R style guide

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

Google R style guide

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

Stack Exchange

Stack Overflow

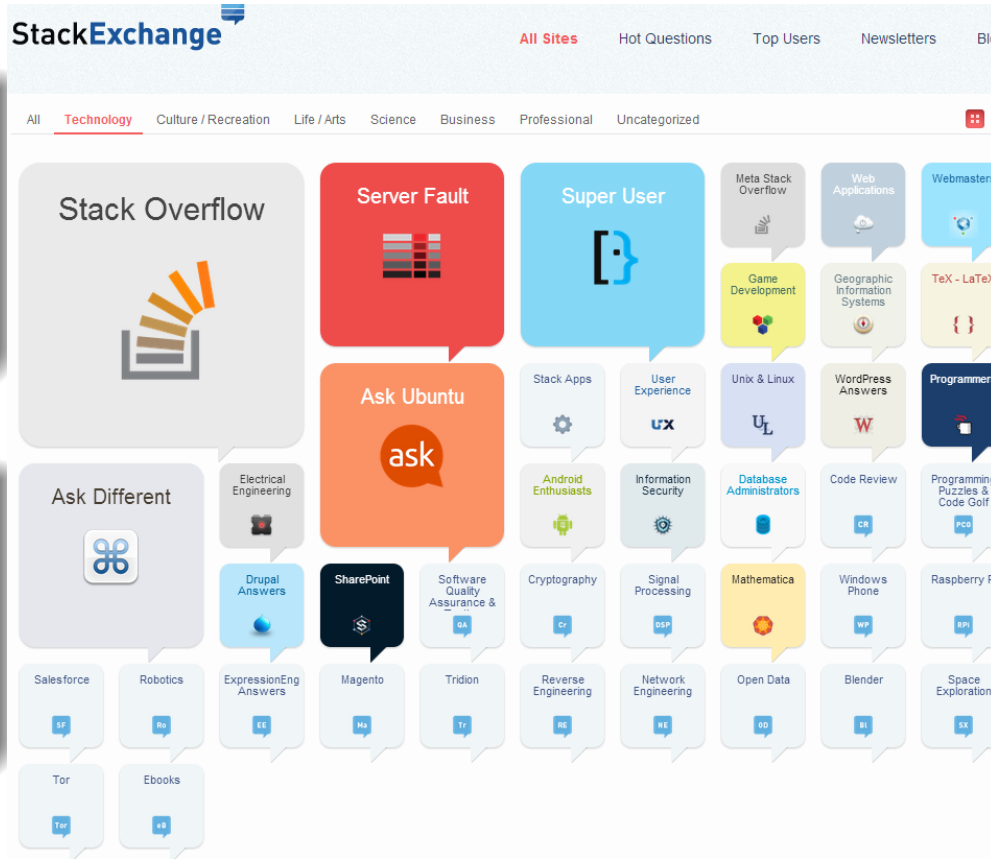
Stack Overflow is a Q&A forum for computer programming, and is part of Stack Exchange

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

Stack Exchange

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

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



RStudio Support

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

- https://support.rstudio.com/hc/en-us
- https://support.rstudio.com/hc/en-us/sections/200107586-Using-RStudio
- https://support.rstudio.com/hc/en-us/sections/200148796-Advanced-Topics

R Online Books and References

Hadley Wickham book *Advanced R*

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

Cookbook for R by Winston Chang from *RStudio*

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

Efficient R programming by Colin Gillespie and Robin Lovelace

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

Endmemo web book

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

Quick-R by Robert Kabacoff

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

R for Beginners by Emmanuel Paradis

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

R Online Interactive Courses

Datacamp Interactive Courses

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

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

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

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

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

Try R

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

R Blogs and Experts

R-Bloggers

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

Dirk Eddelbuettel

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

Romain Frangois

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

More R Blogs and Experts

Revolution Analytics Blog

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

RStudio Blog

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

GitHub for Hosting Software Projects Online

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

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

Most R projects are now hosted on *GitHub*.

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

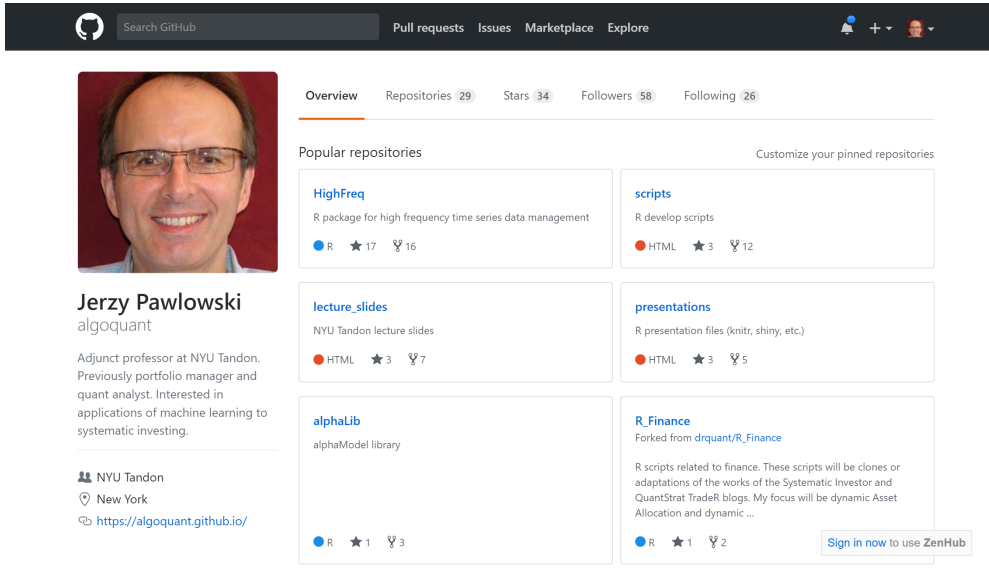
https://github.com/tensorflow/tensorflow

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

https://github.com/algoquant/lecture_slides

https://github.com/algoquant

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



What is R?

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

Why is R More Difficult Than Other Languages?

R is more difficult than other languages because:

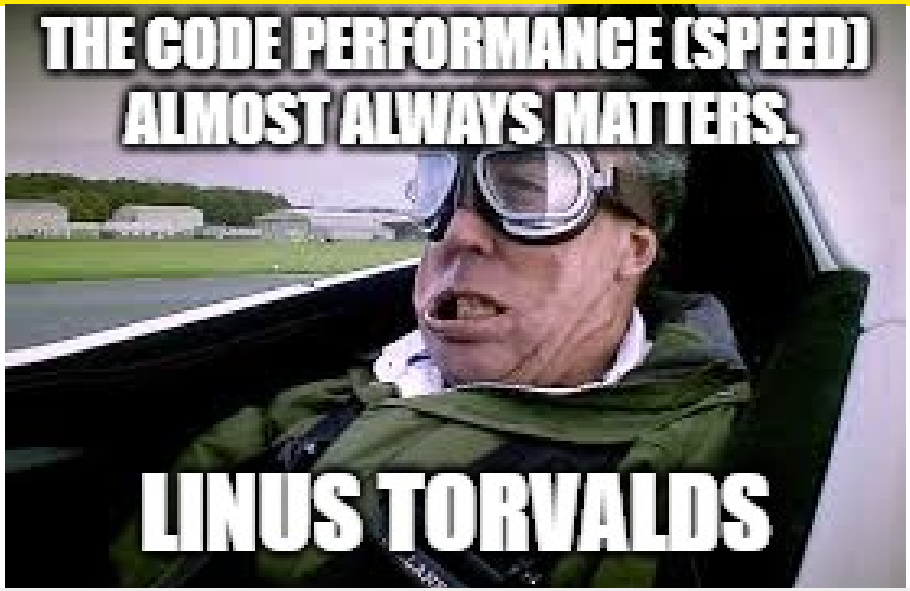
- R is a *functional* language, which makes its syntax unfamiliar to users of procedural languages like C/C++.
- The huge number of user-created *packages* makes it difficult to tell which are the best for particular applications.
- R can produce very cryptic *warning* and *error* messages, because it's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

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

What are the Best Ways to Use R?

If used properly, R can be fast and interactive:

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




```
> # Calculate cumulative sum of a vector
> vec_tor <- runif(1e5)
> # Use compiled function
> cum_sum <- cumsum(vec_tor)
> # Use for loop
> cum_sum2 <- vec_tor
> for (i in 2:NROW(vec_tor))
+   cum_sum2[i] <- (vec_tor[i] + cum_sum2[i-1])
> # Compare the two methods
> all.equal(cum_sum, cum_sum2)
> # Microbenchmark the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   cumsum=cumsum(vec_tor),
+   loop_alloc={
+     cum_sum2 <- vec_tor
+     for (i in 2:NROW(vec_tor))
+ cum_sum2[i] <- (vec_tor[i] + cum_sum2[i-1])
+   },
+   loop_nalloc={
+     # Doesn't allocate memory to cum_sum3
```

The R License

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

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

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

http://creativecommons.org

Installing R and *RStudio*

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

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

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

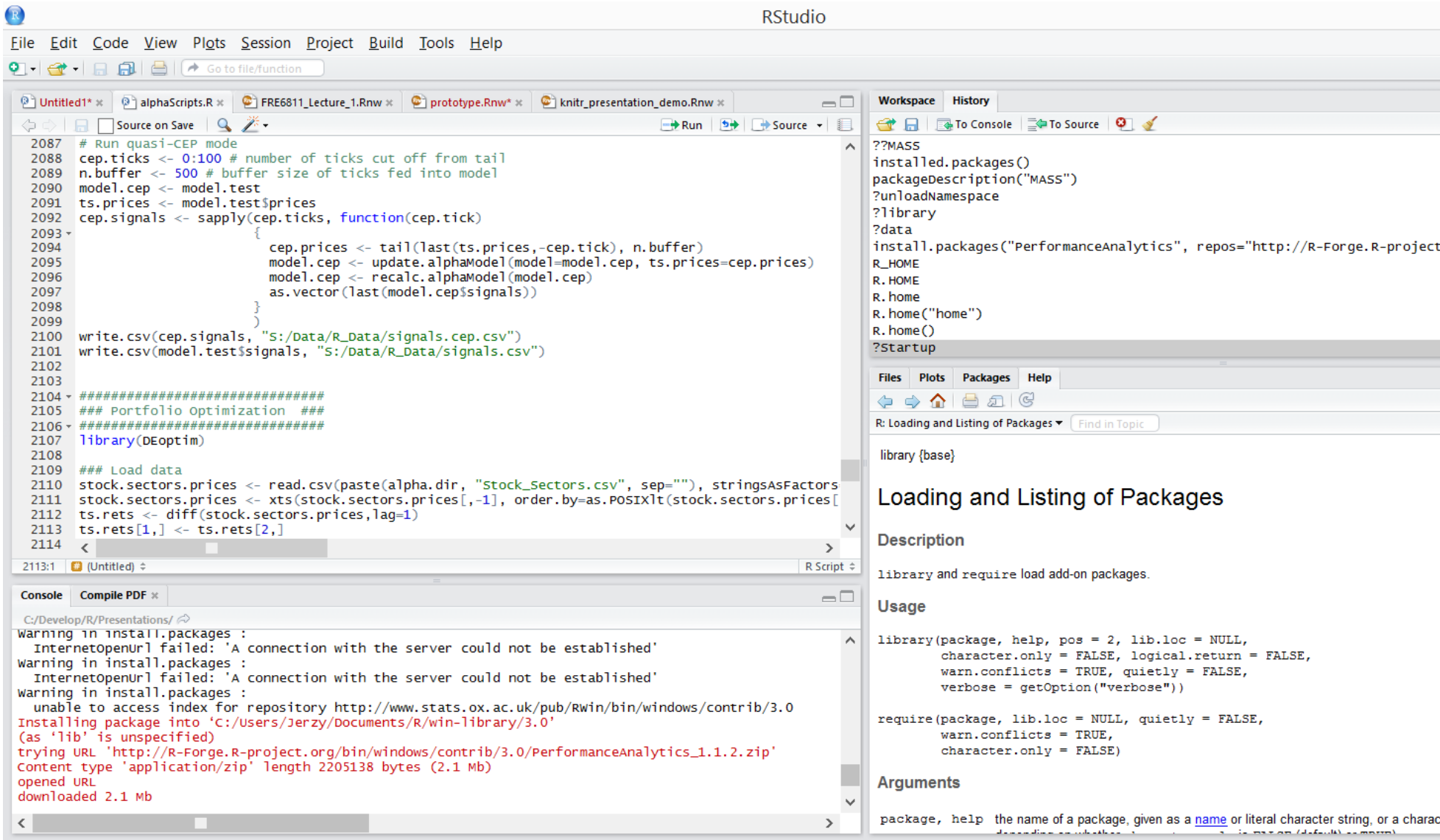
http://cran.r-project.org/

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

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

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

Using *RStudio*



A First R Session

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

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

R 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
> my_var <- 3
>
> # Typing a symbol or expression evaluates it
> my_var
[1] 3
>
> # Text in quotes is interpreted as a string
> my_var <- "Hello World!"
>
> # Typing a symbol or expression evaluates it
> my_var
[1] "Hello World!"
>
> my_var  # 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("C:/Develop/R")  # Set cwd
> getwd()  # Get cwd
```

Get system date and time

Just the date

```
> Sys.time()  # Get date and time
[1] "2021-09-07 17:42:12 EDT"
>
> Sys.Date()  # Get date only
[1] "2021-09-07"
```

The R Workspace

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

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

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

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

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

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

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

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

The R Workspace (cont.)

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

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

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

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

The function `history()` displays recent commands.

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

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

R Session Info

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

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

```
> sessionInfo()  # Get R version and other session info
R version 4.1.1 (2021-08-10)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Big Sur 11.5.2

Matrix products: default
LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/l

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

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

other attached packages:
 [1] tseries_0.10-48           PerformanceAnalytics_2.0.4
 [3] xtable_1.8-4              microbenchmark_1.4-7
 [5] knitr_1.33                HighFreq_0.1
 [7] rutils_0.2                dygraphs_1.1.1.6
 [9] quantmod_0.4.18           TTR_0.24.2
[11] xts_0.12.1                zoo_1.8-9

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.7          magrittr_2.0.1      MASS_7.3-54         RcppR
 [5] lattice_0.20-44     quadprog_1.5-8      rlang_0.4.11        string
 [9] highr_0.9           caTools_1.18.2      tools_4.1.1         parall
[13] Rfast_2.0.3         grid_4.1.1          xfun_0.25           tinyte
[17] matrixStats_0.60.0 htmltools_0.5.1.1 yaml_2.2.1          digest
[21] htmlwidgets_1.5.3  bitops_1.0-7        curl_4.3.2          evalua
[25] stringi_1.7.3      compiler_4.1.1      generics_0.1.0      RcppZ
[29] jsonlite_1.7.2     lubridate_1.7.10
```

Environment Variables

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

For example the environment variables:

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

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

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

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

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

Global *Options* Settings

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

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

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

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

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

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

Environments in R

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

There are three system environments:

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

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

Packages have their own environments.

The enclosure of the base package is the empty environment.

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

The R Search Path

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

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

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

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

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

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

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

Rule of Thumb

Be very careful with using `attach()`.

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

```
> search()  # Get search path for R objects
 [1] ".GlobalEnv"                    "package:tseries"
 [3] "package:PerformanceAnalytics" "package:xtable"
 [5] "package:microbenchmark"        "package:knitr"
 [7] "package:HighFreq"              "package:rutils"
 [9] "package:dygraphs"              "package:quantmod"
[11] "package:TTR"                   "package:xts"
[13] "package:zoo"                   "tools:rstudio"
[15] "package:stats"                 "package:graphics"
[17] "package:grDevices"             "package:utils"
[19] "package:datasets"              "package:methods"
[21] "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"
 [3] "package:tseries"               "package:PerformanceAnalytics"
 [5] "package:xtable"                "package:microbenchmark"
 [7] "package:knitr"                 "package:HighFreq"
 [9] "package:rutils"                "package:dygraphs"
[11] "package:quantmod"              "package:TTR"
[13] "package:xts"                   "package:zoo"
[15] "tools:rstudio"                 "package:stats"
[17] "package:graphics"              "package:grDevices"
[19] "package:utils"                 "package:datasets"
[21] "package:methods"               "Autoloads"
[23] "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
```

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
> sym_bols <- c("VTI", "VEU", "IEF", "VNQ")
> # Extract sym_bols from rutils::etf_env
> price_s <- mget(sym_bols, envir=rutils::etf_env)
> # price_s is a list of xts series
> class(price_s)
> class(price_s[[1]])
> # Extract Close prices
> price_s <- lapply(price_s, quantmod::Cl)
> # Collapse list into time series the hard way
> xts_1 <- cbind(price_s[[1]], price_s[[2]], price_s[[3]], price_s[
> class(xts_1)
> dim(xts_1)
> # Collapse list into time series using do.call()
> price_s <- do.call(cbind, price_s)
> all.equal(xts_1, price_s)
> class(price_s)
> dim(price_s)
> # Extract and cbind in single step
> price_s <- do.call(cbind, lapply(
+   mget(sym_bols, envir=rutils::etf_env), quantmod::Cl))
> # Or
> # Extract and bind all data, subset by sym_bols
> price_s <- lapply(sym_bols, function(sym_bol) {
+     quantmod::Cl(get(sym_bol, envir=rutils::etf_env))
+ })  # end lapply
> # Same, but loop over etf_env without anonymous function
> price_s <- do.call(cbind,
+   lapply(as.list(rutils::etf_env)[sym_bols], quantmod::Cl))
> # Same, but works only for OHLC series - produces error
> price_s <- do.call(cbind,
+   eapply(rutils::etf_env, quantmod::Cl)[sym_bols])
```

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(price_s[, 1:4])
> do.call(rbind, strsplit(colnames(price_s[, 1:4]), split="[.]"))[,
> colnames(price_s) <- do.call(rbind, strsplit(colnames(price_s), sp
> # Or
> colnames(price_s) <- unname(sapply(colnames(price_s),
+     function(col_name) strsplit(col_name, split="[.]")[[1]][1]))
> tail(price_s, 3)
> # Which objects in global environment are class xts?
> unlist(eapply(globalenv(), is.xts))
> # Save xts to csv file
> write.zoo(price_s,
+   file="C:/Develop/lecture_slides/data/etf_series.csv", sep=",")
> # Copy price_s into etf_env
> etf_env$etf_list <- etf_list
> # Or
> assign("price_s", price_s, envir=etf_env)
> # Save to .RData file
> save(etf_env, file="etf_data.RData")
```

Referencing Object Components Using with()

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

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

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

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

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

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

R Packages

Types of R Packages

R can run libraries of functions called packages,

R packages can can also contain data,

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

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

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

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

The *base* Packages

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

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

Very popular packages:

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

CRAN Package Views

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

Note:

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

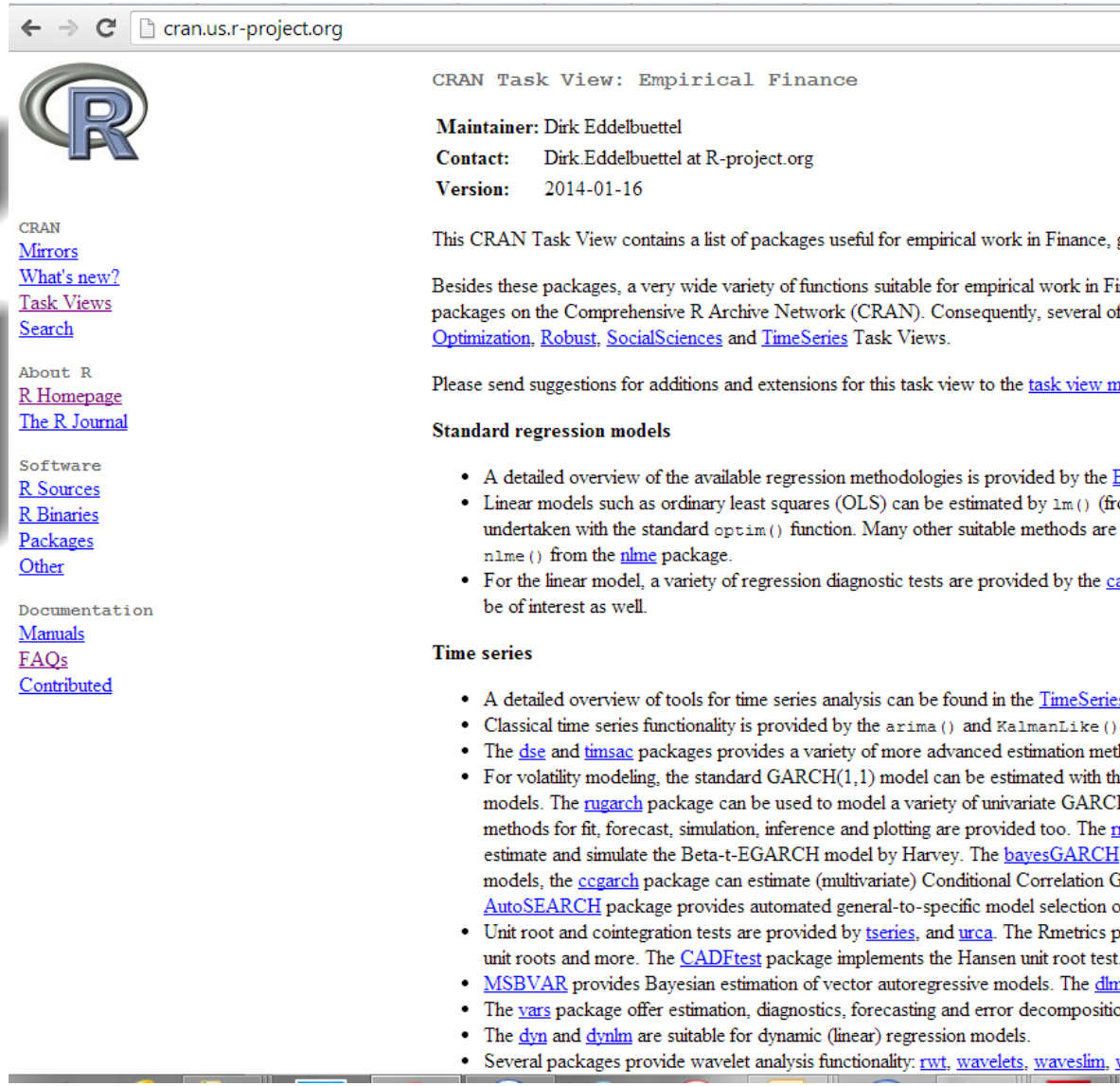
The package source code can be downloaded by clicking on the package source link,

CRAN Task Views

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

Note:

- Maintainer,
- Topics,
- List of packages.



Installing Packages

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

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

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

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

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

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

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

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

https://github.com/

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

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

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

Installing Packages From Source

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

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

For Windows install `Rtools`:

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

For Mac OSX install `XCode` developer tools:

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

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

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

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

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

Installed Packages

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

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

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

Package Files and Directories

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

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

```
> # list directories in "PortfolioAnalytics" sub-directory
> gsub(
+   "C:/Users/Jerzy/Documents/R/win-library/3.1",
+   "~",
+   list.dirs(
+     file.path(
+       .libPaths()[1],
+       "PortfolioAnalytics")))
 [1] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [2] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [3] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [4] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [5] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [6] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [7] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [8] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
 [9] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
[10] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
[11] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
[12] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
[13] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
[14] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library
```

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
> v_wap <- TTR::VWAP(
+   price=quantmod::Cl(rutils::etf_env$VTI),
+   volume=quantmod::Vo(rutils::etf_env$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 lists 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 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)
> vec_tor <- runif(1e6)
> # sqrt() and "^0.5" are the same
> all.equal(sqrt(vec_tor), vec_tor^0.5)
> # sqrt() is much faster than "^0.5"
> system.time(vec_tor^0.5)
> microbenchmark(
+   power = vec_tor^0.5,
+   sqrt = sqrt(vec_tor),
+   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
> mat_rix <- matrix(rnorm(10000), ncol=2)
> # Allocate memory for row sums
> row_sums <- numeric(NROW(mat_rix))
> summary(microbenchmark(
+   row_sums = rowSums(mat_rix),  # end row_sums
+   ap_ply = apply(mat_rix, 1, sum),  # end apply
+   l_apply = lapply(1:NROW(mat_rix), function(in_dex)
+     sum(mat_rix[in_dex, ])),  # end lapply
+   v_apply = vapply(1:NROW(mat_rix), function(in_dex)
+     sum(mat_rix[in_dex, ]),
+     FUN.VALUE = c(sum=0)),  # end vapply
+   s_apply = sapply(1:NROW(mat_rix), function(in_dex)
+     sum(mat_rix[in_dex, ])),  # end sapply
+   for_loop = for (i in 1:NROW(mat_rix)) {
+     row_sums[i] <- sum(mat_rix[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).

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

Vectorized Functions for Vector Computations

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

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

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

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

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

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

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

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

Vectorized Functions for Matrix Computations

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

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

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

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

```
> # Calculate matrix of random data with 5,000 rows
> mat_rix <- matrix(rnorm(10000), ncol=2)
> # Calculate row sums two different ways
> all.equal(rowSums(mat_rix),
+   apply(mat_rix, 1, sum))
> summary(microbenchmark(
+   row_sums = rowSums(mat_rix),
+   ap_ply = apply(mat_rix, 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(
+   p_max=do.call(pmax.int,
+ lapply(seq_along(mat_rix[1, ]),
+   function(in_dex) mat_rix[, in_dex])),
+   l_apply=unlist(lapply(seq_along(mat_rix[, 1]),
+   function(in_dex) max(mat_rix[in_dex, ]))),
+   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(
+   row_mins = rowMins(mat_rix),
+   p_min =
+     do.call(pmin.int,
+       lapply(seq_along(mat_rix[1, ]),
+              function(in_dex)
+                mat_rix[, in_dex])),
+   as_data_frame =
+     do.call(pmin.int,
+       as.data.frame.matrix(mat_rix)),
+   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
> vec_tor <- 1e3
> all.equal(rank(vec_tor), Rfast::Rank(vec_tor))
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode = rank(vec_tor),
+   Rfast = Rfast::Rank(vec_tor),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Benchmark speed of calculating column medians
> mat_rix <- matrix(1e4, nc=10)
> all.equal(matrixStats::colMedians(mat_rix), Rfast::colMedians(mat_
> summary(microbenchmark(
+   matrixStats = matrixStats::colMedians(mat_rix),
+   Rfast = Rfast::colMedians(mat_rix),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Writing Fast R Code Using Vectorized Operations

Writing Fast R Code Using Vectorized Operations

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

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

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

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

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

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

```
> summary(microbenchmark(  # Assign values to vector three different
+ # Fast vectorized assignment loop performed in C using brackets "
+   brack_ets = {vec_tor <- numeric(10)
+     vec_tor[] <- 2},
+ # Slow because loop is performed in R
+   for_loop = {vec_tor <- numeric(10)
+     for (in_dex in seq_along(vec_tor))
+       vec_tor[in_dex] <- 2},
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> summary(microbenchmark(  # Assign values to vector two different
+ # Fast vectorized assignment loop performed in C using brackets "
+   brack_ets = {vec_tor <- numeric(10)
+     vec_tor[4:7] <- rnorm(4)},
+ # Slow because loop is performed in R
+   for_loop = {vec_tor <- numeric(10)
+     for (in_dex in 4:7)
+       vec_tor[in_dex] <- 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(in_put, pa_ram) {
+   pa_ram*in_put
+ }  # end my_fun
> # "in_put" is vectorized
> my_fun(in_put=1:3, pa_ram=2)
> # "pa_ram" is vectorized
> my_fun(in_put=10, pa_ram=2:4)
> # Define vectors of parameters of rnorm()
> std_devs <- structure(1:3, names=paste0("sd=", 1:3))
> me_ans <- structure(-1:1, names=paste0("mean=", -1:1))
> # "sd" argument of rnorm() isn't vectorized
> rnorm(1, sd=std_devs)
> # "mean" argument of rnorm() isn't vectorized
> rnorm(1, mean=me_ans)
```

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 std_devs produces vector output
> set.seed(1121)
> sapply(std_devs, function(std_dev) rnorm(n=2, sd=std_dev))
> # Same
> set.seed(1121)
> sapply(std_devs, rnorm, n=2, mean=0)
> # Loop over me_ans
> set.seed(1121)
> sapply(me_ans, function(me_an) rnorm(n=2, mean=me_an))
> # Same
> set.seed(1121)
> sapply(me_ans, 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 "std_dev"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     sapply(sd, rnorm, n=n, mean=mean)
+ }  # end vec_rnorm
> set.seed(1121)
> vec_rnorm(n=2, sd=std_devs)
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- Vectorize(FUN=rnorm,
+         vectorize.args=c("mean", "sd")
+ )  # end Vectorize
> set.seed(1121)
> vec_rnorm(n=2, sd=std_devs)
> set.seed(1121)
> vec_rnorm(n=2, mean=me_ans)
```

The mapply() Functional

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

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

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

$$mapply(FUN = fun, vec_1, vec_2, \ldots) = [fun(vec_{1,1}, vec_{2,1}, \ldots), \ldots, fun(vec_{1,i}, vec_{2,i}, \ldots), \ldots]$$

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

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

```
> str(sum)
> # na.rm is bound by name
> mapply(sum, 6:9, c(5, NA, 3), 2:6, na.rm=TRUE)
> str(rnorm)
> # mapply vectorizes both arguments "mean" and "sd"
> mapply(rnorm, n=5, mean=me_ans, sd=std_devs)
> mapply(function(in_put, e_xp) in_put^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.

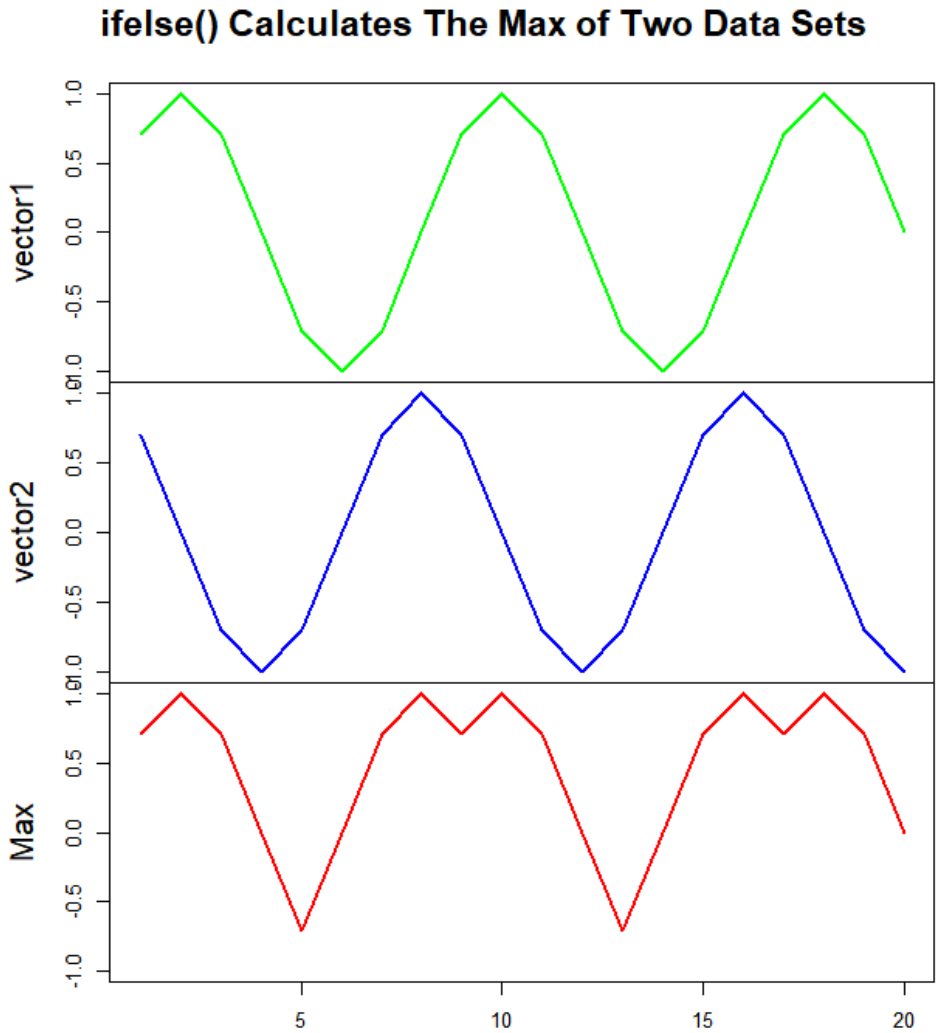
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2,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

Vectorized if-else Statements Using Function ifelse()

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

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

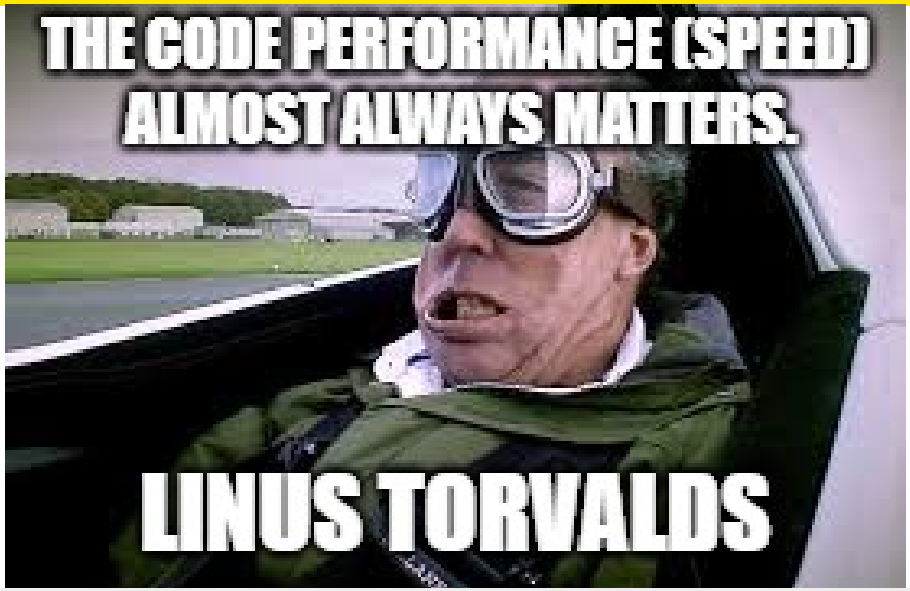
```
> # Create two numeric vectors
> vector1 <- sin(0.25*pi*1:20)
> vector2 <- cos(0.25*pi*1:20)
> # Create third vector using 'ifelse'
> vector3 <- ifelse(vector1 > vector2, vector1, vector2)
> # cbind all three together
> vector3 <- cbind(vector1, vector2, vector3)
> colnames(vector3)[3] <- "Max"
>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



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

How to write fast R code:

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




```
> # Calculate cumulative sum of a vector
> vec_tor <- runif(1e5)
> # Use compiled function
> cum_sum <- cumsum(vec_tor)
> # Use for loop
> cum_sum2 <- vec_tor
> for (i in 2:NROW(cum_sum2))
+   cum_sum2[i] <- (cum_sum2[i] + cum_sum2[i-1])
> # Compare the two methods
> all.equal(cum_sum, cum_sum2)
> # Microbenchmark the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   cumsum=cumsum(vec_tor),
+   loop_alloc={
+     cum_sum2 <- vec_tor
+     for (i in 2:NROW(cum_sum2))
+ cum_sum2[i] <- (cum_sum2[i] + cum_sum2[i-1])
+   },
+   loop_nalloc={
+     # Doesn't allocate memory to cum_sum3
```

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
> n_cores <- detectCores() - 1
> # Initialize compute cluster under Windows
> clus_ter <- makeCluster(n_cores)
> # Perform parallel loop under Windows
> paw_s <- parLapply(clus_ter, 1:10, paws)
> # Perform parallel loop under Mac-OSX or Linux
> paw_s <- mclapply(1:10, paws, mc.cores=n_cores)
> library(microbenchmark)  # Load package microbenchmark
> # Compare speed of lapply versus parallel computing
> summary(microbenchmark(
+   standard = lapply(1:10, paws),
+   parallel = parLapply(clus_ter, 1:10, paws),
+   times=10)
+ )[, c(1, 4, 5)]
```

Computing Advantage of Parallel Computing

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

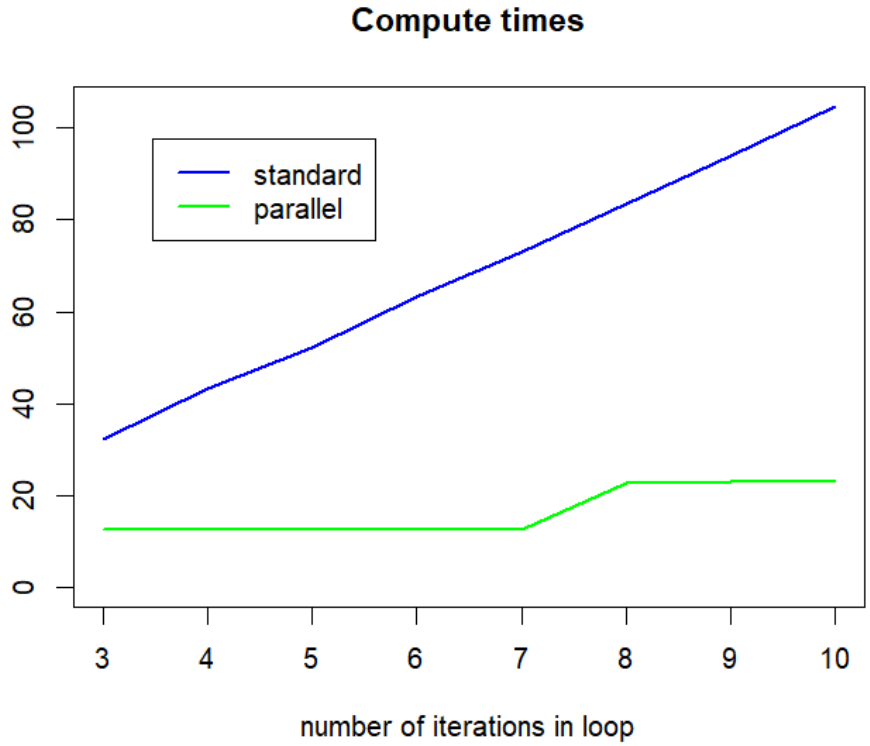
The function `stopCluster()` stops the `R` processes running on several CPU cores.

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




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

Parallel Computing Over Matrices

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

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

```
> # Calculate matrix of random data
> mat_rix <- matrix(rnorm(1e5), ncol=100)
> # Define aggregation function over column of matrix
> agg_regate <- function(col_umn) {
+   out_put <- 0
+   for (in_dex in 1:NROW(col_umn))
+     out_put <- out_put + col_umn[in_dex]
+   out_put
+ }  # end agg_regate
> # Perform parallel aggregations over columns of matrix
> agg_regations <- parCapply(clus_ter, mat_rix, agg_regate)
> # Compare speed of apply with parallel computing
> summary(microbenchmark(
+   ap_ply=apply(mat_rix, MARGIN=2, agg_regate),
+   parl_apply=parCapply(clus_ter, mat_rix, agg_regate),
+   times=10)
+ )[, c(1, 4, 5)]
> # Stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

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.

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

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*.

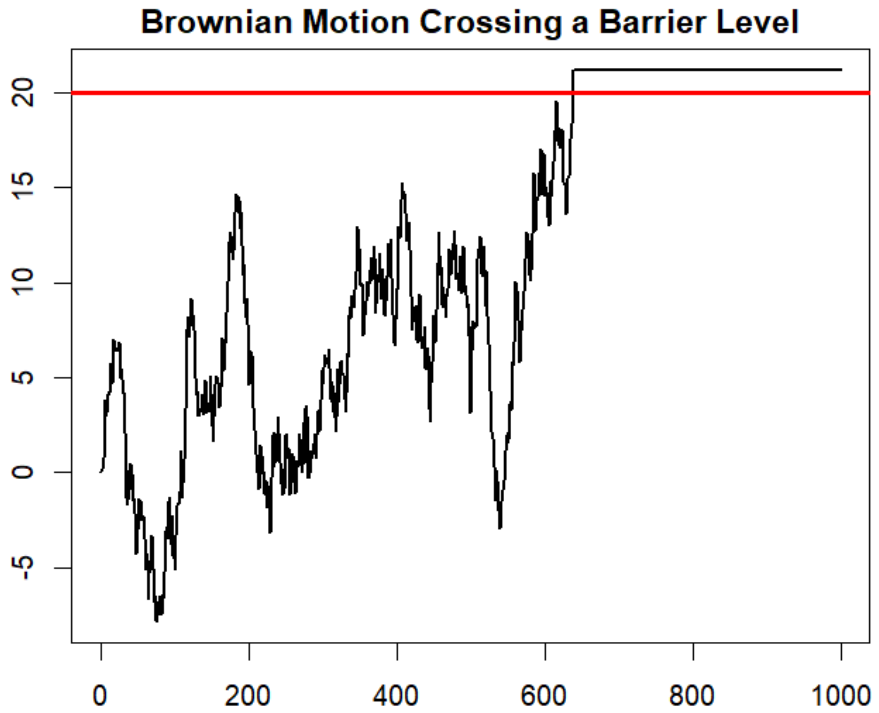
```
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> n_cores <- detectCores() - 1
> # Initialize compute cluster under Windows
> clus_ter <- makeCluster(n_cores)
> # Set seed for cluster under Windows
> # Doesn't work: set.seed(1121)
> clusterSetRNGStream(clus_ter,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

Simulating Brownian Motion Using while() Loops

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

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

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

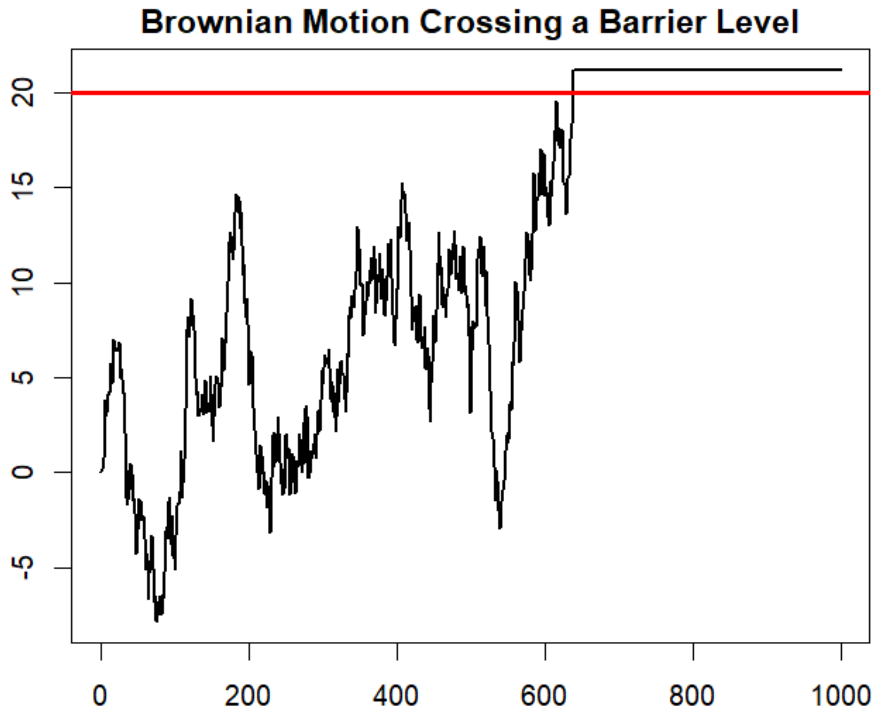


Simulating Brownian Motion Using Vectorized Functions

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

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

```
> set.seed(1121)  # Reset random number generator
> bar_rier <- 20  # Barrier level
> n_rows <- 1000  # Number of simulation steps
> # Simulate path of Brownian motion
> pa_th <- cumsum(rnorm(n_rows))
> # Find index when pa_th crosses bar_rier
> cro_ss <- which(pa_th > bar_rier)
> # Fill remaining pa_th after it crosses bar_rier
> if (NROW(cro_ss)>0) {
+   pa_th[(cro_ss[1]+1):n_rows] <- pa_th[cro_ss[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(pa_th, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=bar_rier, 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,

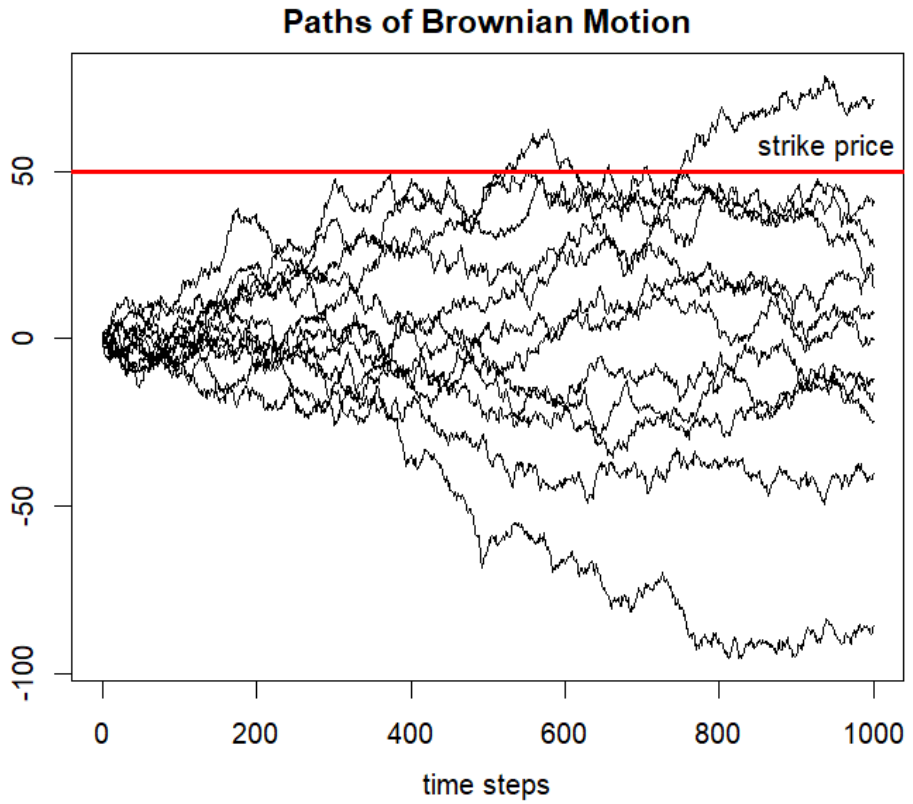
Estimating the Statistics of Brownian Motion

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
> sig_ma <- 1.0  # Volatility
> dri_ft <- 0.0  # Drift
> n_rows <- 1000  # Number of simulation steps
> n_simu <- 100  # Number of simulations
> # Simulate multiple paths of Brownian motion
> set.seed(1121)
> path_s <- rnorm(n_simu*n_rows, mean=dri_ft, sd=sig_ma)
> path_s <- matrix(path_s, nc=n_simu)
> path_s <- matrixStats::colCumsums(path_s)
> # Final distribution of paths
> mean(path_s[n_rows, ]) ; sd(path_s[n_rows, ])
> # Calculate option payout
> strik_e <- 50  # Strike price
> pay_outs <- (path_s[n_rows, ] - strik_e)
> sum(pay_outs[pay_outs > 0])/n_simu
> # Calculate probability of crossing a barrier
> bar_rier <- 50
> cross_ed <- colSums(path_s > bar_rier) > 0
> sum(cross_ed)/n_simu
```




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

Monte Carlo Simulation

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

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

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

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

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

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

```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> # Sample mean - MC estimate
> mean(da_ta)
> # Sample standard deviation - MC estimate
> sd(da_ta)
> # Monte Carlo estimate of cumulative probability
> pnorm(1)
> sum(da_ta < 1)/n_rows
> # Monte Carlo estimate of quantile
> conf_level <- 0.98
> qnorm(conf_level)  # Exact value
> cut_off <- conf_level*n_rows
> da_ta <- sort(da_ta)
> da_ta[cut_off]  # Naive Monte Carlo value
> quantile(da_ta, probs=conf_level)
> # Analyze the source code of quantile()
> stats:::quantile.default
> # Microbenchmark quantile
> library(microbenchmark)
> summary(microbenchmark(
+   monte_carlo = da_ta[cut_off],
+   quan_tile = quantile(da_ta, probs=conf_level),
+   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
> n_rows <- 1000; da_ta <- rnorm(n_rows)
> # Sample mean and standard deviation
> mean(da_ta); sd(da_ta)
> # Bootstrap of sample mean and median
> n_boot <- 10000
> boot_data <- sapply(1:n_boot, function(x) {
+   # Sample from Standard Normal Distribution
+   sampl_e <- rnorm(n_rows)
+   c(mean=mean(sampl_e), median=median(sampl_e))
+ })  # end sapply
> boot_data[, 1:3]
> boot_data <- t(boot_data)
> # Standard error from formula
> sd(da_ta)/sqrt(n_rows)
> # Standard error of mean from bootstrap
> sd(boot_data[, "mean"])
> # Standard error of median from bootstrap
> sd(boot_data[, "median"])
```

The Distribution of Estimators Using Bootstrap Simulation

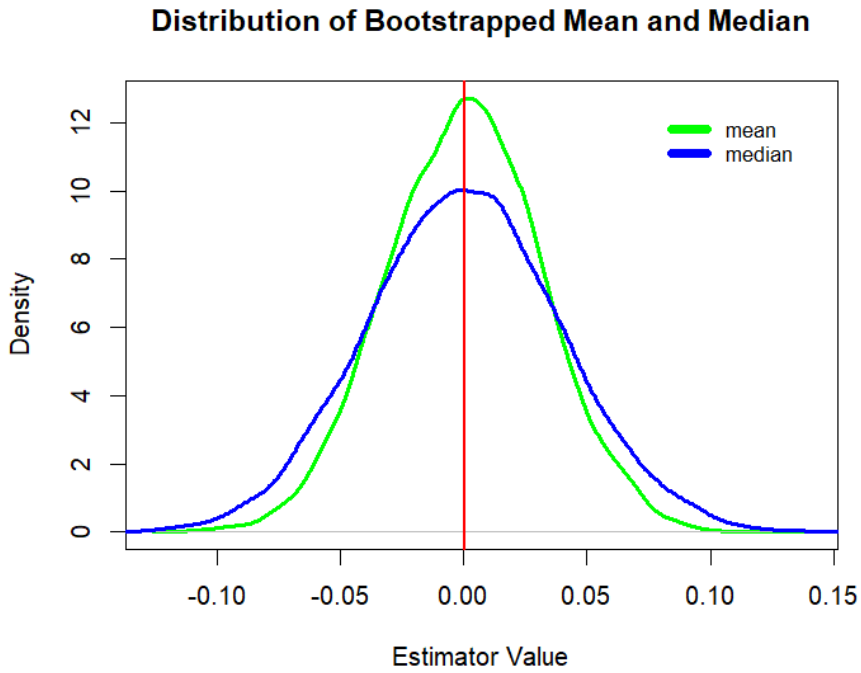
The standard errors of estimators can be calculated using a *bootstrap* simulation.

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(boot_data[, "mean"]), lwd=3, xlab="Estimator Value",
+      main="Distribution of Bootstrapped Mean and Median", col="gre
> lines(density(boot_data[, "median"]), lwd=3, col="blue")
> abline(v=mean(boot_data[, "mean"]), lwd=2, col="red")
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("mean", "median"), bty="n",
+  lwd=6, bg="white", col=c("green", "blue"))
```

Bootstrapping Using Vectorized Operations

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

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

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

Faster code often requires more memory than slower code.

```
> set.seed(1121)  # Reset random number generator
> n_rows <- 1000
> # Bootstrap of sample mean and median
> n_boot <- 100
> boot_data <- sapply(1:n_boot, function(x) median(rnorm(n_rows)))
> # Perform vectorized bootstrap
> set.seed(1121)  # Reset random number generator
> # Calculate matrix of random data
> sampl_e <- matrix(rnorm(n_boot*n_rows), ncol=n_boot)
> boot_vec <- Rfast::colMedians(sampl_e)
> all.equal(boot_data, boot_vec)
> # Compare speed of loops with vectorized R code
> library(microbenchmark)
> summary(microbenchmark(
+   loop = sapply(1:n_boot, function(x) median(rnorm(n_rows))),
+   cpp = {
+     sampl_e <- matrix(rnorm(n_boot*n_rows), ncol=n_boot)
+     Rfast::colMedians(sampl_e)
+     },
+   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
> n_cores <- detectCores() - 1  # Number of cores
> clus_ter <- makeCluster(n_cores)  # Initialize compute cluster und
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> n_rows <- 1000
> # Bootstrap mean and median under Windows
> n_boot <- 10000
> boot_data <- parLapply(clus_ter, 1:n_boot,
+   function(x, da_ta, n_rows) {
+   sampl_e <- rnorm(n_rows)
+   c(mean=mean(sampl_e), median=median(sampl_e))
+   }, da_ta=da_ta, n_rows=n_rows)  # end parLapply
> # Bootstrap mean and median under Mac-OSX or Linux
> boot_data <- mclapply(1:n_boot,
+   function(x) {
+   sampl_e <- rnorm(n_rows)
+   c(mean=mean(sampl_e), median=median(sampl_e))
+   }, mc.cores=n_cores)  # end mclapply
> boot_data <- rutils::do_call(rbind, boot_data)
> # Means and standard errors from bootstrap
> apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=sd(x)))
> # Standard error from formula
> sd(da_ta)/sqrt(n_rows)
> stopCluster(clus_ter)  # Stop R processes over cluster under Windo
```

Parallel Bootstrapping of the *Median Absolute Deviation*

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

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

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

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

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

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

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

```
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> sd(da_ta); mad(da_ta)
> median(abs(da_ta - median(da_ta)))
> median(abs(da_ta - median(da_ta)))/qnorm(0.75)
> # Bootstrap of sd and mad estimators
> n_boot <- 10000
> boot_data <- sapply(1:n_boot, function(x) {
+   sampl_e <- rnorm(n_rows)
+   c(sd=sd(sampl_e), mad=mad(sampl_e))
+ })  # end sapply
> boot_data <- t(boot_data)
> # Analyze bootstrapped variance
> head(boot_data)
> sum(is.na(boot_data))
> # Means and standard errors from bootstrap
> apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=sd(x)))
> # Parallel bootstrap under Windows
> library(parallel)  # Load package parallel
> n_cores <- detectCores() - 1  # Number of cores
> clus_ter <- makeCluster(n_cores)  # Initialize compute cluster
> boot_data <- parLapply(clus_ter, 1:n_boot,
+   function(x, da_ta) {
+     sampl_e <- rnorm(n_rows)
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   }, da_ta=da_ta)  # end parLapply
> # Parallel bootstrap under Mac-OSX or Linux
> boot_data <- mclapply(1:n_boot, function(x) {
+   sampl_e <- rnorm(n_rows)
+   c(sd=sd(sampl_e), mad=mad(sampl_e))
+ }, mc.cores=n_cores)  # end mclapply
> stopCluster(clus_ter)  # Stop R processes over cluster
> boot_data <- rutils::do_call(rbind, boot_data)
> # Means and standard errors from bootstrap
> apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=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)
> re_turns <- rutils::etf_env$re_turns$VTI
> re_turns <- na.omit(re_turns)
> n_rows <- NROW(re_turns)
> # Sample from VTI returns
> sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
> c(sd=sd(sampl_e), mad=mad(sampl_e))
> # 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)
> re_turns <- rutils::etf_env$re_turns$VTI
> re_turns <- na.omit(re_turns)
> n_rows <- NROW(re_turns)
> # Bootstrap sd and MAD under Windows
> library(parallel)  # Load package parallel
> n_cores <- detectCores() - 1  # Number of cores
> clus_ter <- makeCluster(n_cores)  # Initialize compute cluster und
> clusterSetRNGStream(clus_ter, 1121)  # Reset random number generat
> n_boot <- 10000
> boot_data <- parLapply(clus_ter, 1:n_boot,
+   function(x, re_turns, n_rows) {
+     sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   }, re_turns=re_turns, n_rows=n_rows)  # end parLapply
> # Bootstrap sd and MAD under Mac-OSX or Linux
> boot_data <- mclapply(1:n_boot, function(x) {
+     sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   }, mc.cores=n_cores)  # end mclapply
> stopCluster(clus_ter)  # Stop R processes over cluster under Windo
> boot_data <- rutils::do_call(rbind, boot_data)
> # Standard error assuming normal distribution of returns
> sd(re_turns)/sqrt(n_boot)
> # Means and standard errors from bootstrap
> std_errors <- apply(boot_data, MARGIN=2,
+   function(x) c(mean=mean(x), std_error=sd(x)))
> std_errors
> # Relative standard errors
> std_errors[2, ]/std_errors[1, ]
```

Bootstrapping From Time Series of Prices

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

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

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

```
> # Calculate percentage returns from VTI prices
> library(rutils)
> price_s <- quantmod::Cl(rutils::etf_env$VTI)
> star_t <- as.numeric(price_s[1, ])
> re_turns <- rutils::diff_it(log(price_s))
> class(re_turns); head(re_turns)
> sum(is.na(re_turns))
> n_rows <- NROW(re_turns)
> # Define barrier level with respect to price_s
> bar_rier <- 1.5*max(price_s)
> # Calculate single bootstrap sample
> sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
> # Calculate prices from percentage returns
> sampl_e <- star_t*exp(cumsum(sampl_e))
> # Calculate if prices crossed barrier
> sum(sampl_e > bar_rier) > 0
```

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

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)
> oh_lc <- rutils::etf_env$VTI
> price_s <- as.numeric(oh_lc[, 4])
> star_t <- price_s[1]
> re_turns <- rutils::diff_it(log(price_s))
> n_rows <- NROW(re_turns)
> # Calculate difference of OHLC price columns
> ohlc_diff <- oh_lc[, 1:3] - price_s
> class(re_turns); head(re_turns)
> # Calculate bootstrap prices from percentage returns
> da_ta <- sample.int(n_rows, replace=TRUE)
> boot_prices <- star_t*exp(cumsum(re_turns[da_ta]))
> boot_ohlc <- ohlc_diff + boot_prices
> boot_ohlc <- cbind(boot_ohlc, boot_prices)
> # Define barrier level with respect to price_s
> bar_rier <- 1.5*max(price_s)
> # Calculate if High bootstrapped prices crossed barrier level
> sum(boot_ohlc[, 2] > bar_rier) > 0
```

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

Standard Errors of Regression Coefficients Using Bootstrap

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

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

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

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

Distribution of Bootstrapped Regression Coefficients

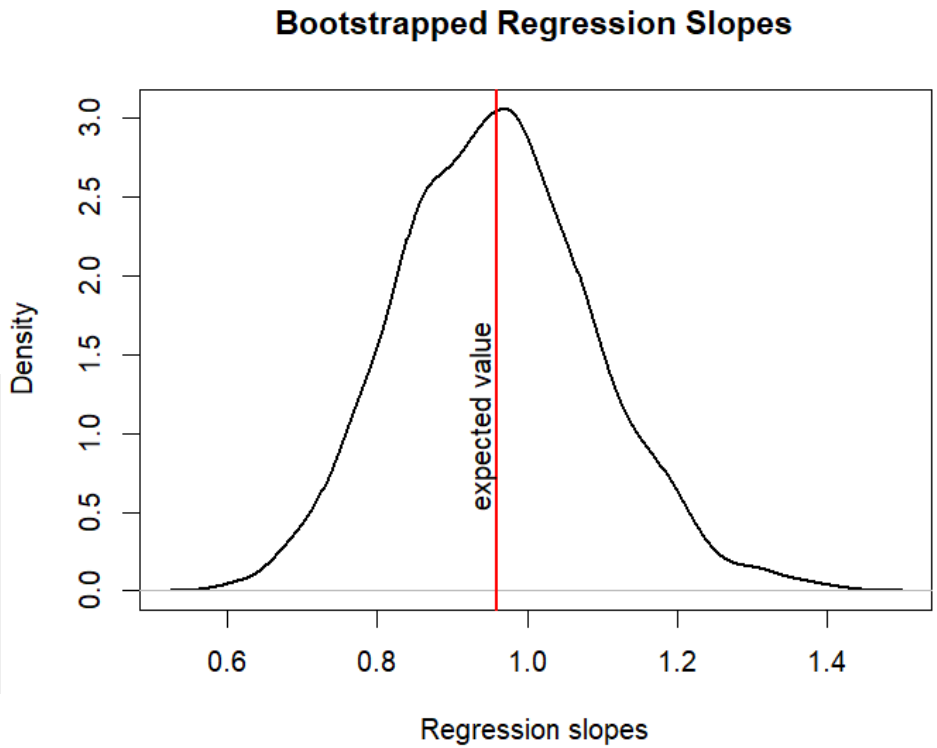
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

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(boot_data), std_error=sd(boot_data))
> # Plot density of bootstrapped beta coefficients
> plot(density(boot_data), lwd=2, xlab="Regression slopes",
+      main="Bootstrapped Regression Slopes")
> # Add line for expected value
> abline(v=mean(boot_data), lwd=2, col="red")
> text(x=mean(boot_data)-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
> n_cores <- detectCores() - 1  # Number of cores
> clus_ter <- makeCluster(n_cores)  # Initialize compute cluster und
> # Bootstrap of regression under Windows
> boot_data <- parLapply(clus_ter, 1:1000,
+   function(x, de_sign) {
+     sampl_e <- sample.int(NROW(de_sign), replace=TRUE)
+     de_sign <- de_sign[sampl_e, ]
+     cov(de_sign[, 1], de_sign[, 2])/var(de_sign[, 2])
+   }, de_sign=de_sign)  # end parLapply
> # Bootstrap of regression under Mac-OSX or Linux
> boot_data <- mclapply(1:1000,
+   function(x) {
+     sampl_e <- sample.int(NROW(de_sign), replace=TRUE)
+     de_sign <- de_sign[sampl_e, ]
+     cov(de_sign[, 1], de_sign[, 2])/var(de_sign[, 2])
+   }, mc.cores=n_cores)  # end mclapply
> stopCluster(clus_ter)  # Stop R processes over cluster under Windo
```

Analyzing the Bootstrap Data

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

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

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

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.

```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> # Estimate the 95% quantile
> n_boot <- 10000
> boot_data <- sapply(1:n_boot, function(x) {
+   sampl_e <- da_ta[sample.int(n_rows, replace=TRUE)]
+   quantile(sampl_e, 0.95)
+ })  # end sapply
> sd(boot_data)
> # Estimate the 95% quantile using antithetic sampling
> boot_data <- sapply(1:n_boot, function(x) {
+   sampl_e <- da_ta[sample.int(n_rows, replace=TRUE)]
+   quantile(c(sampl_e, -sampl_e), 0.95)
+ })  # end sapply
> # Standard error of quantile from bootstrap
> sd(boot_data)
> sqrt(2)*sd(boot_data)
```

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:

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

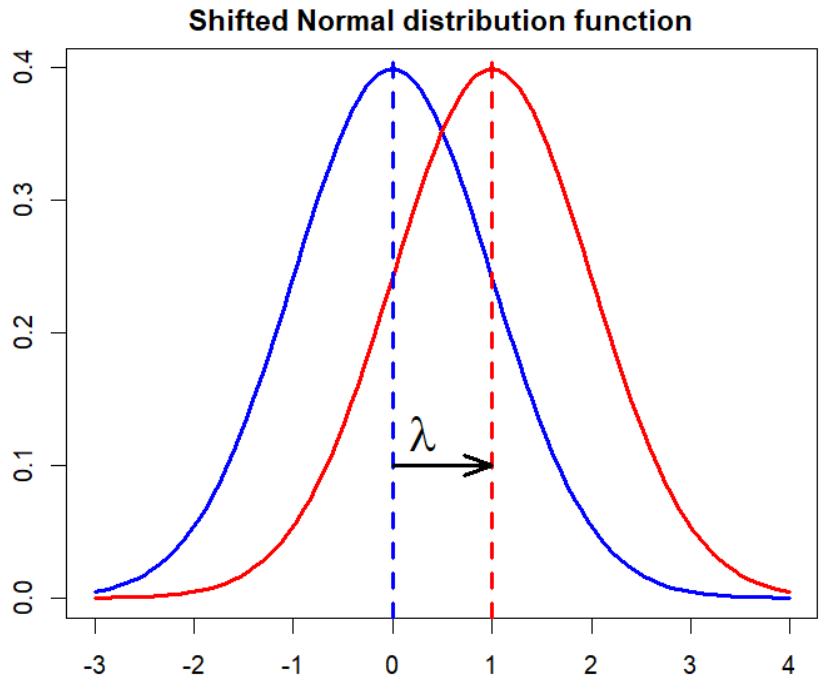
Where $p(x)$ is the probability density, $p(x,\lambda)$ is the tilted density, and λ is the tilt parameter.

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

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

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

Shifting the random variable $x \to x+\lambda$ 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
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> # Cumulative probability from formula
> quan_tile <- (-2)
> pnorm(quan_tile)
> integrate(dnorm, lower=-Inf, upper=quan_tile)
> # Cumulative probability from Naive Monte Carlo
> sum(da_ta < quan_tile)/n_rows
> # Generate importance sample
> lamb_da <- (-1.5)  # Tilt parameter
> data_tilt <- da_ta + lamb_da  # Tilt the random numbers
> # Cumulative probability from importance sample
> sum(data_tilt < quan_tile)/n_rows
> weight_s <- exp(-lamb_da*data_tilt + lamb_da^2/2)
> sum((data_tilt < quan_tile)*weight_s)/n_rows
> # Bootstrap of standard errors of cumulative probability
> n_boot <- 1000
> boot_data <- sapply(1:n_boot, function(x) {
+   da_ta <- rnorm(n_rows)
+   na_ive <- sum(da_ta < quan_tile)/n_rows
+   da_ta <- (da_ta + lamb_da)
+   weight_s <- exp(-lamb_da*da_ta + lamb_da^2/2)
+   im_port <- sum((da_ta < quan_tile)*weight_s)/n_rows
+   c(naive_mc=na_ive, importance=im_port)
+ }) # end sapply
> apply(boot_data, 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
> conf_level <- 0.02
> qnorm(conf_level)  # Exact value
> da_ta <- sort(da_ta)
> cut_off <- n_rows*conf_level
> da_ta[cut_off]  # Naive Monte Carlo value
> # Importance sample weights
> data_tilt <- da_ta + lamb_da  # Tilt the random numbers
> weight_s <- exp(-lamb_da*data_tilt + lamb_da^2/2)
> # Cumulative probabilities using importance sample
> cum_prob <- cumsum(weight_s)/n_rows
> # Quantile from importance sample
> data_tilt[findInterval(conf_level, cum_prob)]
> # Bootstrap of standard errors of quantile
> n_boot <- 1000
> boot_data <- sapply(1:n_boot, function(x) {
+   da_ta <- sort(rnorm(n_rows))
+   na_ive <- da_ta[cut_off]
+   data_tilt <- da_ta + lamb_da
+   weight_s <- exp(-lamb_da*data_tilt + lamb_da^2/2)
+   cum_prob <- cumsum(weight_s)/n_rows
+   im_port <- data_tilt[findInterval(conf_level, cum_prob)]
+   c(naive_mc=na_ive, importance=im_port)
+ }) # end sapply
> apply(boot_data, 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*.

```
> # CVaR from Naive Monte Carlo
> va_r <- da_ta[cut_off]
> sum((da_ta < va_r)*da_ta)/sum((da_ta < va_r))
> # CVaR from importance sample
> va_r <- data_tilt[findInterval(conf_level, cum_prob)]
> sum((data_tilt < va_r)*data_tilt*weight_s)/sum((data_tilt < va_r)*
> # CVaR from integration
> integrate(function(x) x*dnorm(x), low=-Inf, up=va_r)$value/pnorm(v
> # Bootstrap of standard errors of expected value
> n_boot <- 1000
> boot_data <- sapply(1:n_boot, function(x) {
+   da_ta <- sort(rnorm(n_rows))
+   va_r <- da_ta[cut_off]
+   na_ive <- sum((da_ta < va_r)*da_ta)/sum((da_ta < va_r))
+   data_tilt <- da_ta + lamb_da
+   weight_s <- exp(-lamb_da*data_tilt + lamb_da^2/2)
+   cum_prob <- cumsum(weight_s)/n_rows
+   va_r <- data_tilt[findInterval(conf_level, cum_prob)]
+   im_port <- sum((data_tilt < va_r)*data_tilt*weight_s)/sum((data_
+   c(naive_mc=na_ive, importance=im_port)
+ }) # end sapply
> apply(boot_data, 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.

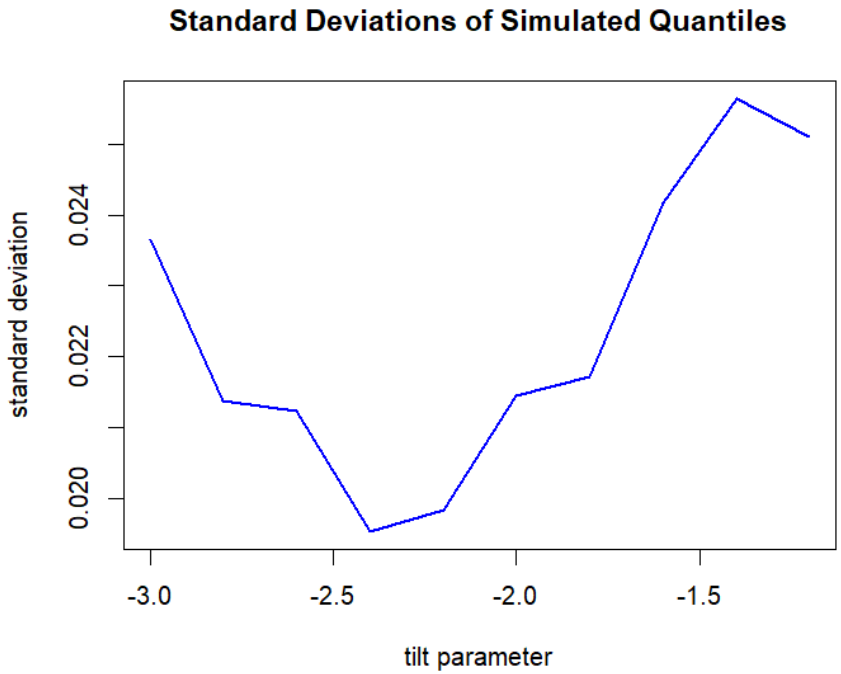
The optimal tilt parameter depends on the estimator and on the required confidence level.

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::sort_mat()` sorts the columns of a matrix using very fast C++ code.

```
> # Calculate matrix of random data
> set.seed(1121) # Reset random number generator
> n_rows <- 1000; n_boot <- 100
> da_ta <- matrix(rnorm(n_boot*n_rows), ncol=n_boot)
> da_ta <- Rfast::sort_mat(da_ta)  # Sort the columns
> # Calculate vector of quantiles for tilt parameter
> conf_level <- 0.02; cut_off <- conf_level*n_rows
> calc_quant <- function(lamb_da) {
+   data_tilt <- da_ta + lamb_da  # Tilt the random numbers
+   weight_s <- exp(-lamb_da*data_tilt + lamb_da^2/2)
+   # Calculate quantiles for columns
+   sapply(1:n_boot, function(boo_t) {
+     cum_prob <- cumsum(weight_s[, boo_t])/n_rows
+     data_tilt[findInterval(conf_level, cum_prob), boo_t]
+   })  # end sapply
+ }  # end calc_quant
```




```
> # Define vector of tilt parameters
> lambda_s <- seq(-3.0, -1.2, by=0.2)
> # Calculate vector of quantiles for tilt parameters
> quantile_s <- sapply(lambda_s, calc_quant)
> # Calculate standard deviations of quantiles for tilt parameters
> std_devs <- apply(quantile_s, MARGIN=2, sd)
> # Calculate the optimal tilt parameter
> lambda_s[which.min(std_devs)]
> # Plot the standard deviations
> x11(width=6, height=5)
> plot(x=lambda_s, y=std_devs,
+      main="Standard Deviations of Simulated Quantiles",
+      xlab="tilt parameter", ylab="standard deviation",
+      type="l", col="blue", lwd=2)
```

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
> n_rows <- 1000
> pro_b <- 0.1
> da_ta <- rbinom(n=n_rows, size=1, pro_b)
> head(da_ta, 33)
> fre_q <- sum(da_ta)/n_rows
> # Tilted binomial sample
> lamb_da <- 5
> p_tilted <- lamb_da*pro_b/(1 + pro_b*(lamb_da - 1))
> weigh_t <- (1 + pro_b*(lamb_da - 1))/lamb_da
> da_ta <- rbinom(n=n_rows, size=1, p_tilted)
> head(da_ta, 33)
> weigh_t*sum(da_ta)/n_rows
> # Bootstrap of standard errors
> n_boot <- 1000
> boot_data <- sapply(1:n_boot, function(x) {
+   c(naive_mc=sum(rbinom(n=n_rows, size=1, pro_b))/n_rows,
+     importance=weigh_t*sum(rbinom(n=n_rows, size=1, p_tilted))/n_r
+ }) # end sapply
> apply(boot_data, 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: $\prod_{i=1}^{t} \exp(-x_i^{tilt}\lambda + \lambda^2/2)$.

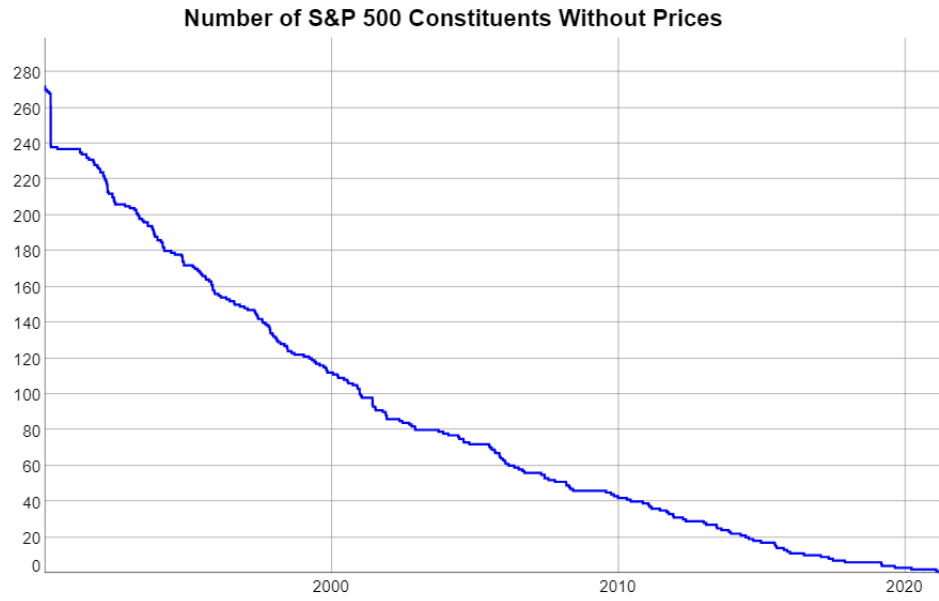
To compensate for the probability tilting, the statistics derived from the tilted Brownian paths must be multiplied by their weight factors.

```
> # Define Brownian motion parameters
> sig_ma <- 1.0  # Volatility
> dri_ft <- 0.0  # Drift
> n_rows <- 100  # Number of simulation steps
> n_simu <- 10000  # Number of simulations
> # Calculate matrix of normal variables
> set.seed(1121)
> da_ta <- rnorm(n_simu*n_rows, mean=dri_ft, sd=sig_ma)
> da_ta <- matrix(da_ta, nc=n_simu)
> # Simulate paths of Brownian motion
> path_s <- matrixStats::colCumsums(da_ta)
> # Tilt the da_ta
> lamb_da <- 0.04  # Tilt parameter
> data_tilt <- da_ta + lamb_da  # Tilt the random numbers
> paths_tilt <- matrixStats::colCumsums(data_tilt)
> # Calculate path weights
> weight_s <- exp(-lamb_da*data_tilt + lamb_da^2/2)
> path_weights <- matrixStats::colProds(weight_s)
> # Or
> path_weights <- exp(-lamb_da*colSums(data_tilt) + n_rows*lamb_da^2
> # Calculate option payout using standard MC
> strik_e <- 10  # Strike price
> pay_outs <- (path_s[n_rows, ] - strik_e)
> sum(pay_outs[pay_outs > 0])/n_simu
> # Calculate option payout using importance sampling
> pay_outs <- (paths_tilt[n_rows, ] - strik_e)
> sum((path_weights*pay_outs)[pay_outs > 0])/n_simu
> # Calculate crossing probability using standard MC
> bar_rier <- 10
> cross_ed <- colSums(path_s > bar_rier) > 0
> sum(cross_ed)/n_simu
> # Calculate crossing probability using importance sampling
> cross_ed <- colSums(paths_tilt > bar_rier) > 0
> sum(path_weights*cross_ed)/n_simu
```

S&P500 Stock Index Constituent Prices

The file sp500.RData contains the *environment* sp500_env with *OHLC* prices and trading volumes of *S&P500* stock index constituents.

The *S&P500* stock index constituent data is of poor quality before 2000, so we'll mostly use the data after the year 2000.



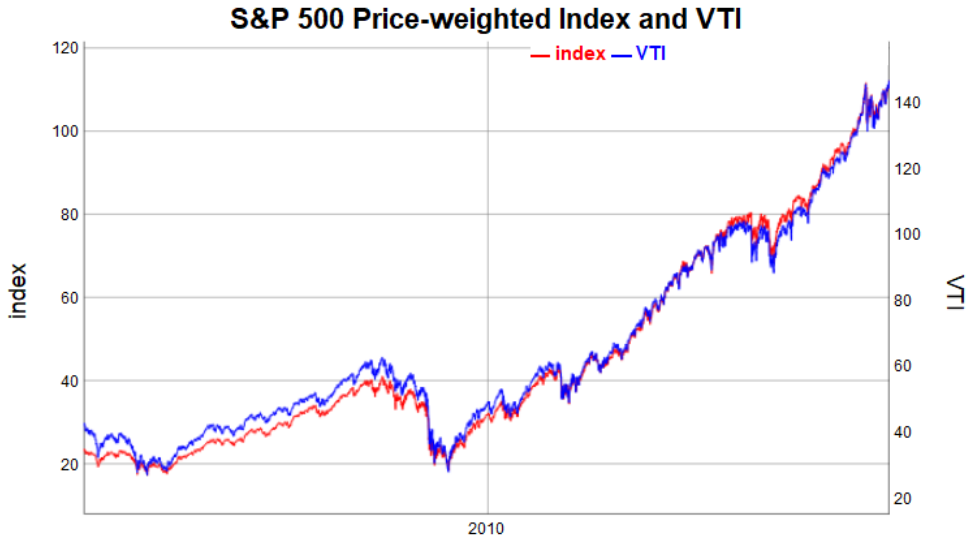

```
> # Load S&P500 constituent stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> price_s <- eapply(sp500_env, quantmod::Cl)
> price_s <- rutils::do_call(cbind, price_s)
> # Carry forward non-NA prices
> price_s <- zoo::na.locf(price_s, na.rm=FALSE)
> # Drop ".Close" from column names
> colnames(price_s[, 1:4])
> colnames(price_s) <- rutils::get_name(colnames(price_s))
> # Or
> # colnames(price_s) <- do.call(rbind,
> #   strsplit(colnames(price_s), split="[.]"))[, 1]
> # Calculate percentage returns of the S&P500 constituent stocks
> # re_turns <- xts::diff.xts(log(price_s))
> re_turns <- xts::diff.xts(price_s)/
+   rutils::lag_it(price_s, pad_zeros=FALSE)
> set.seed(1121)
> sam_ple <- sample(NCOL(re_turns), s=100, replace=FALSE)
> prices_100 <- price_s[, sam_ple]
> returns_100 <- re_turns[, sam_ple]
> save(price_s, prices_100,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_prices.RData")
> save(re_turns, returns_100,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.RData")
```

```
> # Calculate number of constituents without prices
> da_ta <- rowSums(is.na(price_s))
> da_ta <- xts::xts(da_ta, order.by=index(price_s))
> dygraphs::dygraph(da_ta, main="Number of S&P 500 Constituents Wit
+   dyOptions(colors="blue", strokeWidth=2) %>%
+   dyAxis("y", valueRange=c(0, 300))
```

S&P500 Stock Portfolio Index

The price-weighted index of *S&P500* constituents closely follows the VTI *ETF*.

```
> # Calculate price weighted index of constituent
> n_cols <- NCOL(price_s)
> in_dex <- xts(rowSums(price_s)/n_cols, index(price_s))
> colnames(in_dex) <- "index"
> # Combine index with VTI
> da_ta <- cbind(in_dex[index(etf_env$VTI)], etf_env$VTI[, 4])
> col_names <- c("index", "VTI")
> colnames(da_ta) <- col_names
> # Plot index with VTI
> dygraphs::dygraph(da_ta,
+   main="S&P 500 Price-weighted Index and VTI") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red") %>%
+   dySeries(name=col_names[2], axis="y2", col="blue")
```



The *ETF* Database

Exchange-traded Funds (*ETFs*) are funds which invest in portfolios of assets, such as stocks, commodities, or bonds.

ETFs are shares in portfolios of assets, and they are traded just like stocks.

ETFs provide investors with convenient, low cost, and liquid instruments to invest in various portfolios of assets.

The file `etf_list.csv` contains a database of exchange-traded funds (*ETFs*) and exchange traded notes (*ETNs*).

We will select a portfolio of *ETFs* for illustrating various investment strategies.

```
> # Select ETF symbols for asset allocation
> sym_bols <- c("VTI", "VEU", "EEM", "XLY", "XLP", "XLE", "XLF",
+   "XLV", "XLI", "XLB", "XLK", "XLU", "VYM", "IVW", "IWB", "IWD",
+   "IWF", "IEF", "TLT", "VNQ", "DBC", "GLD", "USO", "VXX", "SVXY",
+   "MTUM", "IVE", "VLUE", "QUAL", "VTV", "USMV")
> # Read etf database into data frame
> etf_list <- read.csv(file="/Users/jerzy/Develop/lecture_slides/dat
> rownames(etf_list) <- etf_list$Symbol
> # Select from etf_list only those ETF's in sym_bols
> etf_list <- etf_list[sym_bols, ]
> # Shorten names
> etf_names <- sapply(etf_list$Name, function(name) {
+   name_split <- strsplit(name, split=" ")[[1]]
+   name_split <- name_split[c(-1, -NROW(name_split))]
+   name_match <- match("Select", name_split)
+   if (!is.na(name_match))
+     name_split <- name_split[-name_match]
+   paste(name_split, collapse=" ")
+ })  # end sapply
> etf_list$Name <- etf_names
> etf_list["IEF", "Name"] <- "10 year Treasury Bond Fund"
> etf_list["TLT", "Name"] <- "20 plus year Treasury Bond Fund"
> etf_list["XLY", "Name"] <- "Consumer Discr. Sector Fund"
> etf_list["EEM", "Name"] <- "Emerging Market Stock Fund"
> etf_list["MTUM", "Name"] <- "Momentum Factor Fund"
> etf_list["SVXY", "Name"] <- "Short VIX Futures"
> etf_list["VXX", "Name"] <- "Long VIX Futures"
> etf_list["DBC", "Name"] <- "Commodity Futures Fund"
> etf_list["USO", "Name"] <- "WTI Oil Futures Fund"
> etf_list["GLD", "Name"] <- "Physical Gold Fund"
```

ETF Portfolio for Investment Strategies

The portfolio contains *ETFs* representing different *industry sectors* and *investment styles*.

The *ETFs* with names *X** represent industry *sector funds* (energy, financial, etc.)

The *ETFs* with names *I** represent *style funds* (value, growth, size).

IWB is the Russell 1000 small-cap fund.

MTUM is an *ETF* which owns a stock portfolio representing the *momentum factor*.

DBC is an *ETF* providing the total return on a portfolio of commodity futures.

VXX is an *ETN* providing the total return of *long VIX* futures contracts (specifically the *S&P* VIX Short-Term Futures Index).

VXX is *bearish* because it's *long* VIX futures, and the VIX *rises* when stock prices *drop*.

SVXY is an *ETF* providing the total return of *short VIX* futures contracts.

SVXY is *bullish* because it's *short* VIX futures, and the VIX *drops* when stock prices *rise*.

Symbol	Name	Fund.Type
VTI	Total Stock Market	US Equity ETF
VEU	FTSE All World Ex US	Global Equity ETF
EEM	Emerging Market Stock Fund	Global Equity ETF
XLY	Consumer Discr. Sector Fund	US Equity ETF
XLP	Consumer Staples Sector Fund	US Equity ETF
XLE	Energy Sector Fund	US Equity ETF
XLF	Financial Sector Fund	US Equity ETF
XLV	Health Care Sector Fund	US Equity ETF
XLI	Industrial Sector Fund	US Equity ETF
XLB	Materials Sector Fund	US Equity ETF
XLK	Technology Sector Fund	US Equity ETF
XLU	Utilities Sector Fund	US Equity ETF
VYM	Large-cap Value	US Equity ETF
IVW	S&P 500 Growth Index Fund	US Equity ETF
IWB	Russell 1000	US Equity ETF
IWD	Russell 1000 Value	US Equity ETF
IWF	Russell 1000 Growth	US Equity ETF
IEF	10 year Treasury Bond Fund	US Fixed Income ETF
TLT	20 plus year Treasury Bond Fund	US Fixed Income ETF
VNQ	REIT ETF - DNQ	US Equity ETF
DBC	Commodity Futures Fund	Commodity Based ETF
GLD	Physical Gold Fund	Commodity Based ETF
USO	WTI Oil Futures Fund	Commodity Based ETF
VXX	Long VIX Futures	Commodity Based ETN
SVXY	Short VIX Futures	Commodity Based ETF
MTUM	Momentum Factor Fund	US Equity ETF
IVE	S&P 500 Value Index Fund	US Equity ETF
VLUE	MSCI USA Value Factor	US Equity ETF
QUAL	MSCI USA Quality Factor	US Equity ETF
VTV	Value	US Equity ETF
USMV	MSCI USA Minimum Volatility Fund	US Equity ETF

Exchange Traded Notes (*ETNs*)

ETNs are similar to *ETFs*, with the difference that *ETFs* are shares in a fund which owns the underlying assets, while *ETNs* are notes from issuers which promise payouts according to a formula tied to the underlying asset.

ETFs are similar to mutual funds, while *ETNs* are similar to corporate bonds.

ETNs are technically unsecured corporate debt, but instead of fixed coupons, they promise to provide returns on a market index or futures contract.

The *ETN* issuer promises the payout and is responsible for tracking the index.

The *ETN* investor has counterparty credit risk to the *ETN* issuer.

Kernel Density of Asset Returns

The kernel density is proportional to the number of data points close to a given point.

The kernel density is analogous to a histogram, but it provides more detailed information about the distribution of the data.

The smoothing kernel $K(x)$ is a symmetric function which decreases with the distance x.

The kernel density d_i of a data sample r_i is equal to the sum over the kernel function $K(x)$:

$$d_i = \sum_{j=1}^{n} K(r_i - r_j)$$

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

The parameter *smoothing bandwidth* is the standard deviation of the smoothing kernel $K(x)$.

The function `density()` returns a vector of densities at equally spaced points, not for the original data points.

The function `approx()` interpolates a vector of data into another vector.

```
> library(rutils)  # Load package rutils
> # Calculate VTI percentage returns
> re_turns <- rutils::etf_env$re_turns$VTI
> re_turns <- drop(coredata(na.omit(re_turns)))
> n_rows <- NROW(re_turns)
> # Mean and standard deviation of returns
> c(mean(re_turns), sd(re_turns))
> # Calculate the smoothing bandwidth as the MAD of returns 10 point
> re_turns <- sort(re_turns)
> b_w <- 10*mad(rutils::diff_it(re_turns, lagg=10))
> # Calculate the kernel density
> den_sity <- sapply(1:n_rows, function(i_d) {
+   sum(dnorm(re_turns-re_turns[i_d], sd=b_w))
+ })  # end sapply
> ma_d <- mad(re_turns)
> plot(re_turns, den_sity, xlim=c(-5*ma_d, 5*ma_d),
+      t="l", col="blue", lwd=3,
+      xlab="returns", ylab="density",
+      main="Density of VTI Returns")
> # Calculate the kernel density using density()
> den_sity <- density(re_turns, bw=b_w)
> NROW(den_sity$y)
> x11(width=6, height=5)
> plot(den_sity, xlim=c(-5*ma_d, 5*ma_d),
+      xlab="returns", ylab="density",
+      col="blue", lwd=3, main="Density of VTI Returns")
> # Interpolate the den_sity vector into re_turns
> den_sity <- approx(den_sity$x, den_sity$y, xout=re_turns)
> all.equal(den_sity$x, re_turns)
> plot(den_sity, xlim=c(-5*ma_d, 5*ma_d),
+      xlab="returns", ylab="density",
+      t="l", col="blue", lwd=3,
+      main="Density of VTI Returns")
```

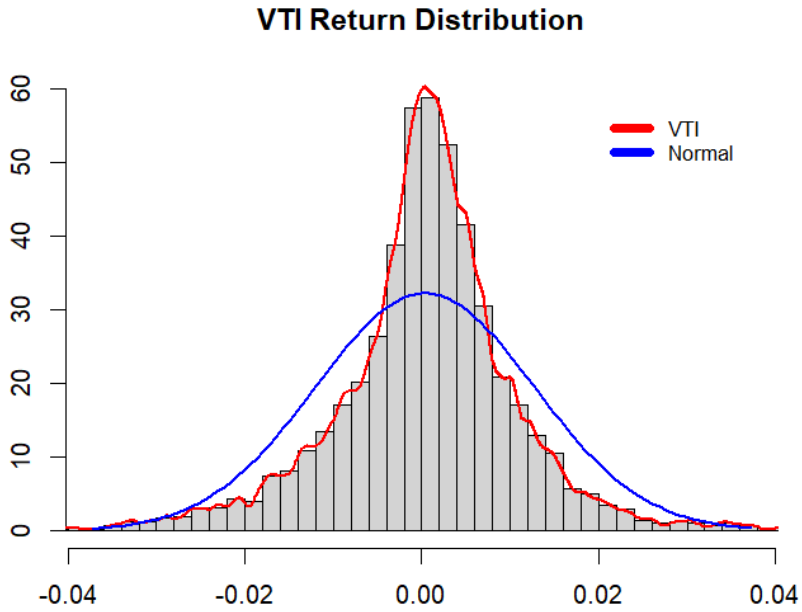
Distribution of Asset Returns

Asset returns are usually not normally distributed and they exhibit *leptokurtosis* (large kurtosis, or fat tails).

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

The parameter `breaks` is the number of cells of the histogram.

The function `lines()` draws a line through specified points.




```
> # Plot histogram
> histo_gram <- hist(re_turns, breaks=100, freq=FALSE,
+   xlim=c(-5*ma_d, 5*ma_d), xlab="", ylab="",
+   main="VTI Return Distribution")
> # Draw kernel density of histogram
> lines(den_sity, col="red", lwd=2)
> # Add density of normal distribution
> curve(expr=dnorm(x, mean=mean(re_turns), sd=sd(re_turns)),
+ add=TRUE, lwd=2, col="blue")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("VTI", "Normal"), bty="n",
+  lwd=6, bg="white", col=c("red", "blue"))
```

The Quantile-Quantile Plot

A *Quantile-Quantile* (*Q-Q*) plot is a plot of points with the same *quantiles*, from two probability distributions.

If the two distributions are similar then all the points in the *Q-Q* plot lie along the diagonal.

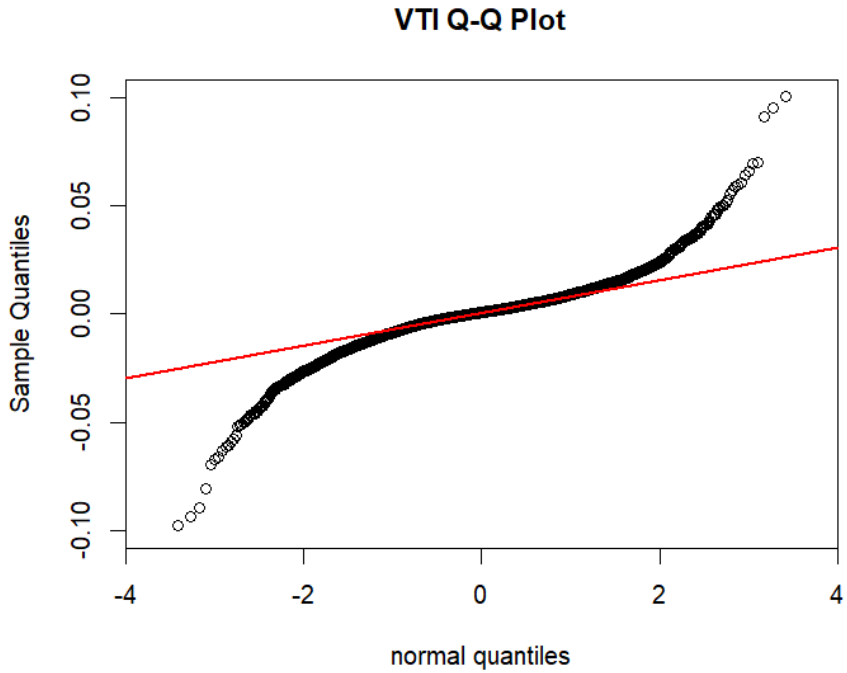
The *VTI Q-Q* plot shows that the *VTI* return distribution has fat tails.

The *p*-value of the *Shapiro-Wilk* test is very close to zero, which shows that the *VTI* returns are very unlikely to be normal.

The function `qqnorm()` produces a normal *Q-Q* plot.

The function `qqline()` fits a line to the normal quantiles.

```
> # Create normal Q-Q plot
> qqnorm(re_turns, ylim=c(-0.1, 0.1), main="VTI Q-Q Plot",
+   xlab="Normal Quantiles")
> # Fit a line to the normal quantiles
> qqline(re_turns, col="red", lwd=2)
> # Perform Shapiro-Wilk test
> shapiro.test(as.numeric(re_turns))
```



Boxplots of Distributions of Values

Box-and-whisker plots (*boxplots*) are graphical representations of a distribution of values.

The bottom and top box edges (*hinges*) are equal to the first and third quartiles, and the *box* width is equal to the interquartile range (*IQR*).

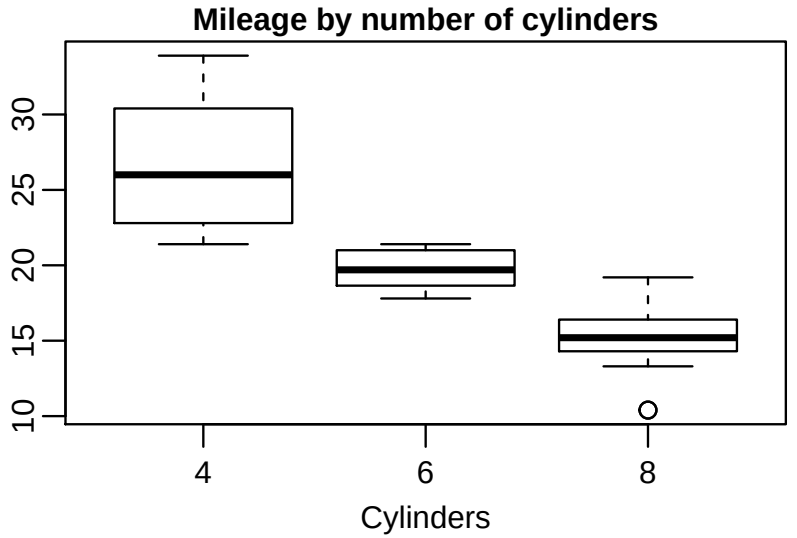
The nominal range is equal to 1.5 times the *IQR* above and below the box *hinges*.

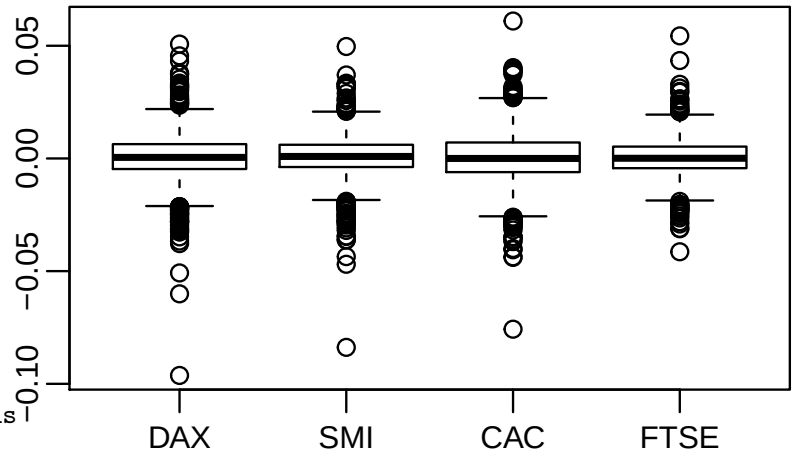
The *whiskers* are dashed vertical lines representing values beyond the first and third quartiles, but within the nominal range.

The *whiskers* end at the last values within the nominal range, while the open circles represent outlier values beyond the nominal range.

The function `boxplot()` has two `methods`: one for `formula` objects (for categorical variables), and another for `data frames`.

```
> # Boxplot method for formula
> boxplot(formula=mpg ~ cyl, data=mtcars,
+   main="Mileage by number of cylinders",
+   xlab="Cylinders", ylab="Miles per gallon")
> # Boxplot method for data frame of EuStockMarkets percentage returns
> boxplot(x=diff(log(EuStockMarkets)))
```





Higher Moments of Asset Returns

The estimators of moments of a probability distribution are given by:

Sample mean: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

Sample variance: $\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$

With their expected values equal to the population mean and standard deviation:
$\mathbb{E}[\bar{x}] = \mu$ and $\mathbb{E}[\hat{\sigma}] = \sigma$

The sample skewness (third moment):

$$\varsigma = \frac{n}{(n-1)(n-2)}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\hat{\sigma}}\right)^3$$

The sample kurtosis (fourth moment):

$$\kappa = \frac{n(n+1)}{(n-1)(n-2)(n-3)}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\hat{\sigma}}\right)^4$$

The normal distribution has skewness equal to 0 and kurtosis equal to 3.

Stock returns typically have negative skewness and kurtosis much greater than 3.

```
> # VTI percentage returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> # Number of observations
> n_rows <- NROW(re_turns)
> # Mean of VTI returns
> mean_rets <- mean(re_turns)
> # Standard deviation of VTI returns
> sd_rets <- sd(re_turns)
> # Skewness of VTI returns
> n_rows/((n_rows-1)*(n_rows-2))*
+   sum(((re_turns - mean_rets)/sd_rets)^3)
> # Kurtosis of VTI returns
> n_rows*(n_rows+1)/((n_rows-1)^3)*
+   sum(((re_turns - mean_rets)/sd_rets)^4)
> # Random normal returns
> re_turns <- rnorm(n_rows, sd=sd_rets)
> # Mean and standard deviation of random normal returns
> mean_rets <- mean(re_turns)
> sd_rets <- sd(re_turns)
> # Skewness of random normal returns
> n_rows/((n_rows-1)*(n_rows-2))*
+   sum(((re_turns - mean_rets)/sd_rets)^3)
> # Kurtosis of random normal returns
> n_rows*(n_rows+1)/((n_rows-1)^3)*
+   sum(((re_turns - mean_rets)/sd_rets)^4)
```

Functions for Calculating Skew and Kurtosis

R provides an easy way for users to write functions.

The function `calc_skew()` calculates the skew of returns, and `calc_kurt()` calculates the kurtosis.

Functions return the value of the last expression that is evaluated.

```
> # calc_skew() calculates skew of returns
> calc_skew <- function(re_turns) {
+   re_turns <- na.omit(re_turns)
+   sum(((re_turns - mean(re_turns))/sd(re_turns))^3)/NROW(re_turns)
+ }  # end calc_skew
> # calc_kurt() calculates kurtosis of returns
> calc_kurt <- function(re_turns) {
+   re_turns <- na.omit(re_turns)
+   sum(((re_turns - mean(re_turns))/sd(re_turns))^4)/NROW(re_turns)
+ }  # end calc_kurt
> # Calculate skew and kurtosis of VTI returns
> calc_skew(re_turns)
> calc_kurt(re_turns)
> # calc_mom() calculates the moments of returns
> calc_mom <- function(re_turns, mo_ment=3) {
+   re_turns <- na.omit(re_turns)
+   sum(((re_turns - mean(re_turns))/sd(re_turns))^mo_ment)/NROW(re_
+ }  # end calc_mom
> # Calculate skew and kurtosis of VTI returns
> calc_mom(re_turns, mo_ment=3)
> calc_mom(re_turns, mo_ment=4)
```

Standard Errors of Estimators

Statistical estimators are functions of samples (which are random variables), and therefore are themselves *random variables*.

The *standard error* (SE) of an estimator is defined as its *standard deviation* (not to be confused with the *population standard deviation* of the underlying random variable).

For example, the *standard error* of the estimator of the mean is equal to:

$$\sigma_\mu = \frac{\sigma}{\sqrt{n}}$$

Where σ is the *population standard deviation* (which is usually unkown).

The *estimator* of this *standard error* is equal to:

$$SE_\mu = \frac{\hat{\sigma}}{\sqrt{n}}$$

where: $\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ is the sample standard deviation (the estimator of the population standard deviation).

```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> # Sample mean
> mean(da_ta)
> # Sample standard deviation
> sd(da_ta)
> # Standard error of sample mean
> sd(da_ta)/sqrt(n_rows)
```

Normal (Gaussian) Probability Distribution

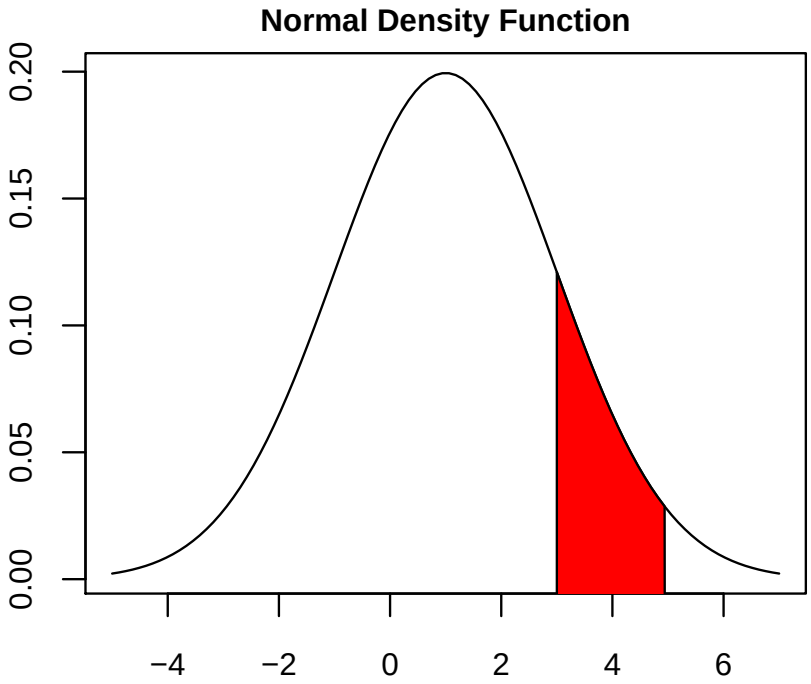
The *Normal (Gaussian)* probability density function is given by:

$$\phi(x, \mu, \sigma) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

The *Standard Normal* distribution $\phi(0, 1)$ is a special case of the *Normal* $\phi(\mu, \sigma)$ with $\mu = 0$ and $\sigma = 1$.

The function `dnorm()` calculates the *Normal* probability density.

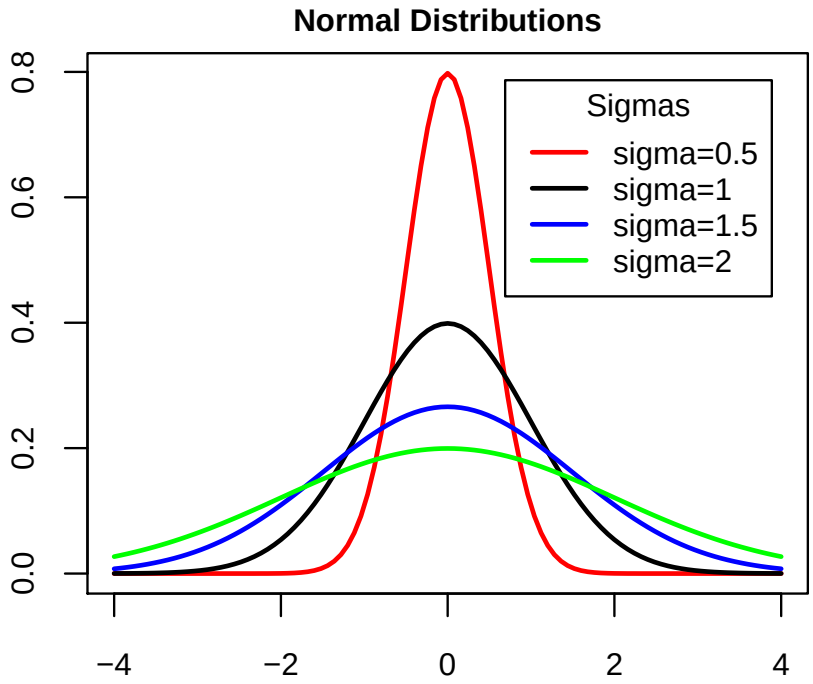
```
> x_var <- seq(-5, 7, length=100)
> y_var <- dnorm(x_var, mean=1.0, sd=2.0)
> plot(x_var, y_var, type="l", lty="solid", xlab="", ylab="")
> title(main="Normal Density Function", line=0.5)
> star_t <- 3; fin_ish <- 5  # Set lower and upper bounds
> # Plot polygon area
> are_a <- ((x_var >= star_t) & (x_var <= fin_ish))
> polygon(c(star_t, x_var[are_a], fin_ish),
+   c(-1, y_var[are_a], -1), col="red")
```



Normal (Gaussian) Probability Distributions

Plots of several *Normal* distributions with different values of σ, using the function `curve()` for plotting functions given by their name.

```
> sig_mas <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> col_ors <- c("red", "black", "blue", "green")
> # Create legend labels
> lab_els <- paste("sigma", sig_mas, sep="=")
> for (in_dex in 1:4) {  # Plot four curves
+   curve(expr=dnorm(x, sd=sig_mas[in_dex]),
+   xlim=c(-4, 4), xlab="", ylab="", lwd=2,
+   col=col_ors[in_dex], add=as.logical(in_dex-1))
+ }  # end for
> # Add title
> title(main="Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, title="Sigmas",
+  lab_els, cex=0.8, lwd=2, lty=1, bty="n", col=col_ors)
```



Student's *t-distribution*

Let $z_1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu}\sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance: $\hat{\sigma}^2 = \frac{1}{\nu-1}\sum_{i=1}^{\nu}(z_i - \bar{z})^2$

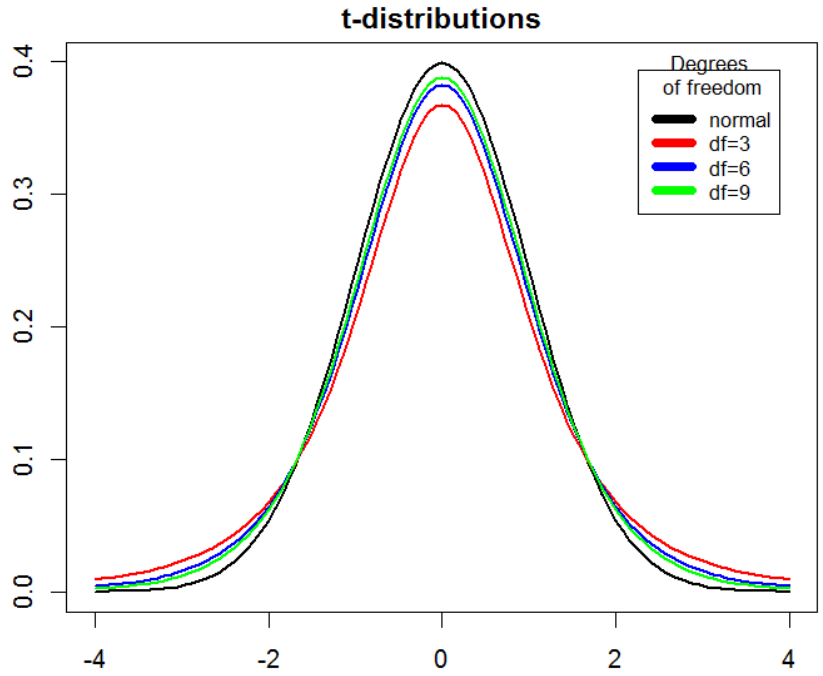
Then the random variable (*t-ratio*):

$$t = \frac{\bar{z} - \mu}{\hat{\sigma}/\sqrt{\nu}}$$

Follows the *t-distribution* with ν degrees of freedom, with the probability density function:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}\left(1 + t^2/\nu\right)^{-(\nu+1)/2}$$

```
> deg_free <- c(3, 6, 9)  # Df values
> col_ors <- c("black", "red", "blue", "green")
> lab_els <- c("normal", paste("df", deg_free, sep="="))
> # Plot a Normal probability distribution
> curve(expr=dnorm, xlim=c(-4, 4), xlab="", ylab="", lwd=2)
> for (in_dex in 1:3) {  # Plot three t-distributions
+   curve(expr=dt(x, df=deg_free[in_dex]), xlab="", ylab="",
+ lwd=2, col=col_ors[in_dex+1], add=TRUE)
+ }  # end for
```




```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title="Degrees\n of freedom", lab_els,
+        cex=0.8, lwd=6, lty=1, col=col_ors)
```

Mixture Models of Returns

Mixture models are produced by randomly sampling data from different distributions.

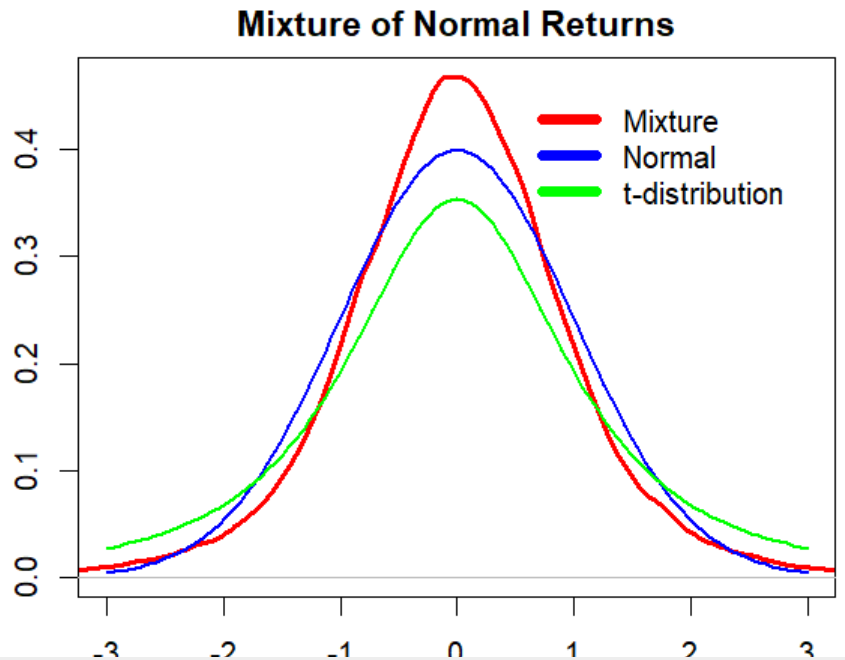
The mixture of two normal distributions with different variances produces a distribution with *leptokurtosis* (large kurtosis, or fat tails).

Student's *t-distribution* has fat tails because the sample variance in the denominator of the *t-ratio* is variable.

The time-dependent volatility of asset returns is referred to as *heteroskedasticity*.

Random processes with *heteroskedasticity* can be considered a type of mixture model.

The *heteroskedasticity* produces *leptokurtosis* (large kurtosis, or fat tails).




```
> # Mixture of two normal distributions with sd=1 and sd=2
> n_rows <- 1e5
> re_turns <- c(rnorm(n_rows/2), 2*rnorm(n_rows/2))
> re_turns <- (re_turns-mean(re_turns))/sd(re_turns)
> # Kurtosis of normal
> calc_kurt(rnorm(n_rows))
> # Kurtosis of mixture
> calc_kurt(re_turns)
> # Or
> n_rows*sum(re_turns^4)/(n_rows-1)^2
```

```
> # Plot the distributions
> plot(density(re_turns), xlab="", ylab="",
+   main="Mixture of Normal Returns",
+   xlim=c(-3, 3), type="l", lwd=3, col="red")
> curve(expr=dnorm, lwd=2, col="blue", add=TRUE)
> curve(expr=dt(x, df=3), lwd=2, col="green", add=TRUE)
> # Add legend
> legend("topright", inset=0.05, lty=1, lwd=6, bty="n",
+   legend=c("Mixture", "Normal", "t-distribution"),
+   col=c("red", "blue", "green"))
```

The Likelihood Function of Student's *t-distribution*

The non-standard Student's *t-distribution* is:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\sigma\,\Gamma(\nu/2)}\,(1+(\frac{t-\mu}{\sigma})^2/\nu)^{-(\nu+1)/2}$$

It has non-zero mean equal to the location parameter μ, and a standard deviation proportional to the scale parameter σ.

The negative logarithm of the probability density is equal to:

$$-\log(f(t)) = -\log(\frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}) + \log(\sigma) + \frac{\nu+1}{2}\,\log(1+(\frac{t-\mu}{\sigma})^2/\nu)$$

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

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

```
> # Objective function is log-likelihood
> likeli_hood <- function(pa_r, free_dom, da_ta) {
+   sum(
+     -log(gamma((free_dom+1)/2) /
+       (sqrt(pi*free_dom) * gamma(free_dom/2))) +
+     log(pa_r[2]) +
+     (free_dom+1)/2 * log(1 + ((da_ta - pa_r[1])/
+                     pa_r[2])^2/free_dom))
+ }  # end likeli_hood
> # Demonstrate equivalence with log(dt())
> likeli_hood(c(1, 0.5), 2, 2:5)
> -sum(log(dt(x=(2:5-1)/0.5, df=2)/0.5))
> # Simpler objective function
> likeli_hood <- function(pa_r, free_dom, da_ta) {
+   -sum(log(dt(x=(da_ta-pa_r[1])/pa_r[2],
+       df=free_dom)/pa_r[2]))
+ }  # end likeli_hood
```

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

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

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

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

Fitting Asset Returns into Student's *t-distribution*

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

The function `fitdistr()` performs a *maximum likelihood* optimization to find the non-standardized Student's *t-distribution* location and scale parameters.

```
> # VTI percentage returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> # Initial parameters
> par_init <- c(mean=0, scale=0.01)
> # Fit distribution using optim()
> optim_fit <- optim(par=par_init,
+   fn=likeli_hood, # Log-likelihood function
+   da_ta=re_turns,
+   free_dom=2, # Degrees of freedom
+   method="L-BFGS-B", # quasi-Newton method
+   upper=c(1, 0.1), # upper constraint
+   lower=c(-1, 1e-7)) # Lower constraint
> # optimal parameters
> lo_cation <- optim_fit$par["mean"]
> scal_e <- optim_fit$par["scale"]
> # Fit VTI returns using MASS::fitdistr()
> optim_fit <- MASS::fitdistr(re_turns,
+   densfun="t", df=2)
> optim_fit$estimate
> optim_fit$sd
> lo_cation <- optim_fit$estimate[1]
> scal_e <- optim_fit$estimate[2]
> summary(optim_fit)
```

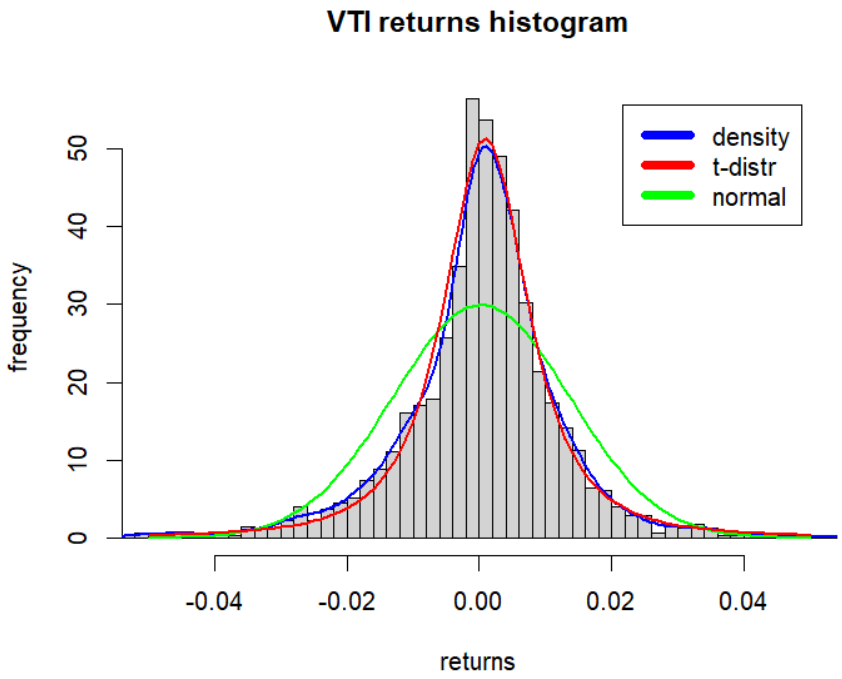
The Student's *t-distribution* Fitted to Asset Returns

Asset returns typically exhibit *negative skewness* and *large kurtosis* (leptokurtosis), or fat tails.

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

The parameter `breaks` is the number of cells of the histogram.

```
> # Plot histogram of VTI returns
> histo_gram <- hist(re_turns, col="lightgrey",
+   xlab="returns", breaks=100, xlim=c(-0.05, 0.05),
+   ylab="frequency", freq=FALSE, main="VTI Returns Histogram")
> lines(density(re_turns, adjust=1.5), lwd=3, col="blue")
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(re_turns),
+   sd=sd(re_turns)), add=TRUE, lwd=3, col="green")
> # Plot t-distribution function
> curve(expr=dt((x-lo_cation)/scal_e, df=2)/scal_e,
+ type="l", lwd=3, col="red", add=TRUE)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("density", "t-distr", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```



Leptokurtosis Fat Tails of Asset Returns

The probability under the *normal* distribution decreases exponentially for large values of x:

$$\phi(x) \propto e^{-x^2/2\sigma^2} \qquad (as\ |x| \to \infty)$$

This is because a normal variable can be thought of as the sum of a large number of independent binomial variables of equal size.

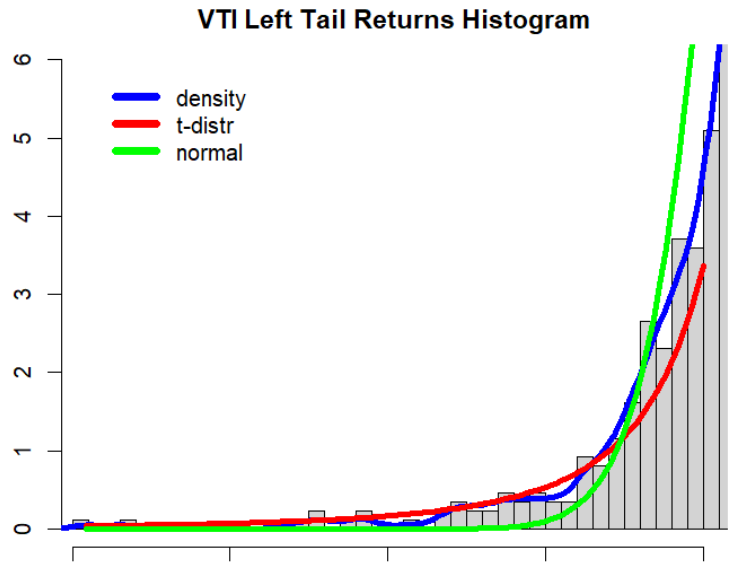
So large values are produced only when all the contributing binomial variables are of the same sign, which is very improbable, so it produces extremely low tail probabilities (thin tails),

But in reality, the probability of large negative asset returns decreases much slower, as the negative power of the returns (fat tails).

The probability under Student's *t-distribution* decreases as a power for large values of x:

$$f(x) \propto |x|^{-(\nu+1)} \qquad (as\ |x| \to \infty)$$

This is because a *t-variable* can be thought of as the sum of normal variables with different volatilities (different sizes).



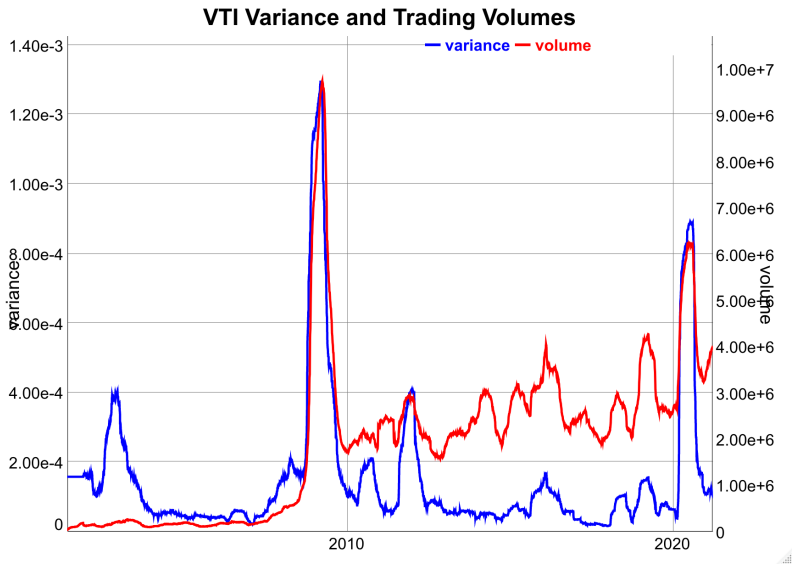

```
> # Plot histogram of VTI returns
> histo_gram <- hist(re_turns, breaks=100, plot=FALSE)
> plot(histo_gram, xlab="returns", ylab="frequency",
+      col="lightgrey", freq=FALSE, main="VTI Left Tail Returns Hist
+      xlim=c(min(re_turns), -0.02),
+      ylim=c(0.0, histo_gram$density[findInterval(-0.02, histo_gram
> lines(density(re_turns, adjust=1.5), lwd=4, col="blue")
> # Plot t-distribution function
> curve(expr=dt((x-lo_cation)/scal_e, df=2)/scal_e, type="l", lwd=4,
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(re_turns), sd=sd(re_turns)), add=TRU
> # Add legend
> legend("topleft", inset=0.05, bty="n",
+   leg=c("density", "t-distr", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Trading Volumes

The rolling average trading volumes have increased significantly since the 2008 crisis, mostly because of high frequency trading (HFT).

Higher levels of volatility coincide with higher *trading volumes*.

The time-dependent volatility of asset returns (*heteroskedasticity*) produces their fat tails (*leptokurtosis*).




```
> # Calculate VTI returns and trading volumes
> oh_lc <- rutils::etf_env$VTI
> clos_e <- drop(coredata(quantmod::Cl(oh_lc)))
> re_turns <- rutils::diff_it(log(clos_e))
> vol_ume <- coredata(quantmod::Vo(oh_lc))
> # Calculate rolling variance
> look_back <- 121
> vari_ance <- HighFreq::roll_var_ohlc(log(oh_lc), method="close", look_back=look_back, scale=FALSE)
> vari_ance[1:look_back, ] <- vari_ance[look_back+1, ]
> # Calculate rolling average volume
> volume_roll <- HighFreq::roll_vec(vol_ume, look_back=look_back)/look_back
> # dygraph plot of VTI variance and trading volumes
> da_ta <- xts::xts(cbind(vari_ance, volume_roll), index(oh_lc))
> col_names <- c("variance", "volume")
> colnames(da_ta) <- col_names
> dygraphs::dygraph(da_ta, main="VTI Variance and Trading Volumes") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], strokeWidth=2, axis="y", col="blue") %>%
+   dySeries(name=col_names[2], strokeWidth=2, axis="y2", col="red")
```

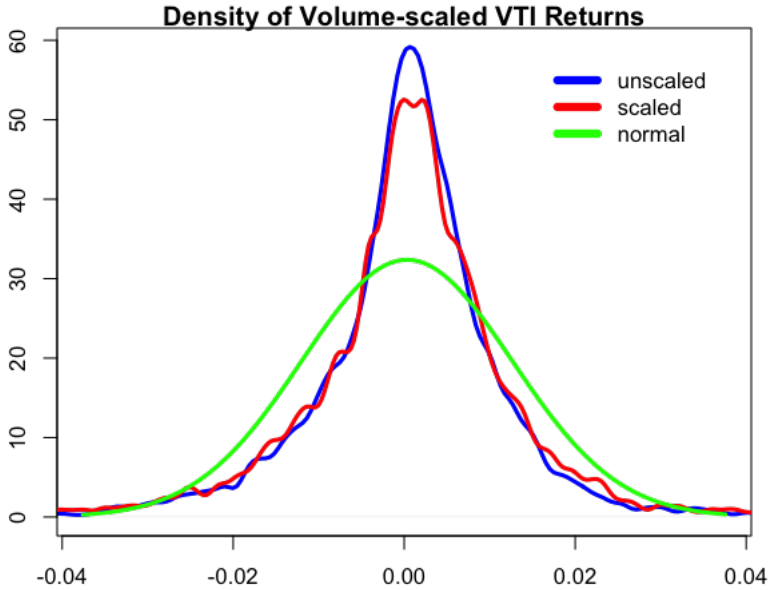
Asset Returns in Trading Time

The time-dependent volatility of asset returns (*heteroskedasticity*) produces their fat tails (*leptokurtosis*).

If asset returns were measured at fixed intervals of *trading volumes* (*trading time* instead of clock time), then the volatility would be lower and less time-dependent.

The asset returns can be adjusted to *trading time* by dividing them by the *square root of the trading volumes*, to obtain scaled returns over equal trading volumes.

The scaled returns have a more positive *skewness* and a smaller *kurtosis* than unscaled returns.




```
> # Scale returns using volume (volume clock)
> rets_scaled <- ifelse(vol_ume > 0, sqrt(volume_roll)*re_turns/sq
> rets_scaled <- sd(re_turns)*rets_scaled/sd(rets_scaled)
> # rets_scaled <- ifelse(vol_ume > 1e4, re_turns/vol_ume, 0)
> # Calculate moments of scaled returns
> n_rows <- NROW(re_turns)
> sapply(list(re_turns=re_turns, rets_scaled=rets_scaled),
+   function(rets) {sapply(c(skew=3, kurt=4),
+     function(x) sum((rets/sd(rets))^x)/n_rows)
+ })  # end sapply
```

```
> # x11(width=6, height=5)
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> # Plot densities of SPY returns
> ma_d <- mad(re_turns)
> # b_w <- mad(rutils::diff_it(re_turns))
> plot(density(re_turns, bw=ma_d/10), xlim=c(-5*ma_d, 5*ma_d),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of Volume-scaled VTI Returns")
> lines(density(rets_scaled, bw=ma_d/10), lwd=3, col="red")
> curve(expr=dnorm(x, mean=mean(re_turns), sd=sd(re_turns)),
+ add=TRUE, lwd=3, col="green")
> # Add legend
> legend("topright", inset=0.05, bty="n",
+   leg=c("unscaled", "scaled", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

The *Shapiro-Wilk* Test of Normality

The *Shapiro-Wilk* test is designed to test the *null hypothesis* that a sample: $\{x_1, \ldots, x_n\}$ is from a normally distributed population.

The test statistic is equal to:

$$W = \frac{(\sum_{i=1}^{n} a_i x_{(i)})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Where the: $\{a_1, \ldots, a_n\}$ are proportional to the *order statistics* of random variables from the normal distribution.

$x_{(k)}$ is the k-th *order statistic*, and is equal to the k-th smallest value in the sample: $\{x_1, \ldots, x_n\}$.

The *Shapiro-Wilk* statistic follows its own distribution, and is less than or equal to 1.

The *Shapiro-Wilk* statistic is close to 1 for samples from normal distributions.

The p-value for *VTI* returns is extremely small, and we conclude that the *null hypothesis* is FALSE, and the *VTI* returns are not from a normally distributed population.

```
> # Calculate VTI percentage returns
> library(rutils)
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> # Reset output digits
> dig_its <- options(digits=5)
> # Shapiro-Wilk test for normal distribution
> shapiro.test(rnorm(NROW(re_turns)))

Shapiro-Wilk normality test

data:  rnorm(NROW(re_turns))
W = 1, p-value = 0.47
> # Shapiro-Wilk test for VTI returns
> shapiro.test(as.numeric(re_turns))

Shapiro-Wilk normality test

data:  as.numeric(re_turns)
W = 0.886, p-value <2e-16
> # Shapiro-Wilk test for uniform distribution
> shapiro.test(runif(NROW(re_turns)))

Shapiro-Wilk normality test

data:  runif(NROW(re_turns))
W = 0.956, p-value <2e-16
> # Restore output digits
> options(digits=dig_its$digits)
```

The *Jarque-Bera* Test of Normality

The *Jarque-Bera* test is designed to test the *null hypothesis* that a sample: $\{x_1, \ldots, x_n\}$ is from a normally distributed population.

The test statistic is equal to:

$$JB = \frac{n}{6}(\varsigma^2 + \frac{1}{4}(\kappa - 3)^2)$$

Where the *skewness* and *kurtosis* are defined as:

$$\varsigma = \frac{1}{n}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{\hat{\sigma}})^3 \qquad \kappa = \frac{1}{n}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{\hat{\sigma}})^4$$

The *Jarque-Bera* statistic asymptotically follows the *chi-squared* distribution with two degrees of freedom.

The *Jarque-Bera* statistic is small for samples from normal distributions.

The *p*-value for *VTI* returns is extremely small, and we conclude that the *null hypothesis* is `FALSE`, and the *VTI* returns are not from a normally distributed population.

```
> library(tseries)  # Load package tseries
> # Jarque-Bera test for normal distribution
> jarque.bera.test(rnorm(NROW(re_turns)))

Jarque Bera Test

data:  rnorm(NROW(re_turns))
X-squared = 0.4, df = 2, p-value = 0.8
> # Jarque-Bera test for VTI returns
> jarque.bera.test(re_turns)

Jarque Bera Test

data:  re_turns
X-squared = 28215, df = 2, p-value <2e-16
> # Jarque-Bera test for uniform distribution
> jarque.bera.test(runif(NROW(re_turns)))

Jarque Bera Test

data:  runif(NROW(re_turns))
X-squared = 319, df = 2, p-value <2e-16
```

The *Kolmogorov-Smirnov* Test for Probability Distributions

The *Kolmogorov-Smirnov* test is designed to test the *null hypothesis* that two samples: $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_n\}$ were obtained from the same probability distribution.

The *Kolmogorov-Smirnov* statistic is the maximum difference between two empirical cumulative distribution functions (cumulative frequencies):

$$D = \sup_i |P(x_i) - P(y_i)|$$

The function `ks.test()` performs the *Kolmogorov-Smirnov* test and returns the statistic and its *p*-value *invisibly*.

The second argument is either a `numeric` vector of data values, or a name of a cumulative distribution function.

The *Kolmogorov-Smirnov* test can be used as a *goodness of fit* test, to test if a set of observations fits a probability distribution.

```
> # KS test for normal distribution
> ks.test(rnorm(100), pnorm)
> # KS test for uniform distribution
> ks.test(runif(100), pnorm)
> # KS test for two similar normal distributions
> ks.test(rnorm(100), rnorm(100, mean=0.1))
> # KS test for two different normal distributions
> ks.test(rnorm(100), rnorm(100, mean=1.0))
> # Fit t-dist into VTI returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> optim_fit <- MASS::fitdistr(re_turns, densfun="t", df=2)
> lo_cation <- optim_fit$estimate[1]
> scal_e <- optim_fit$estimate[2]
> # Perform Kolmogorov-Smirnov test on VTI returns
> da_ta <- lo_cation + scal_e*rt(NROW(re_turns), df=2)
> ks.test(as.numeric(re_turns), da_ta)
```

Chi-squared Distribution

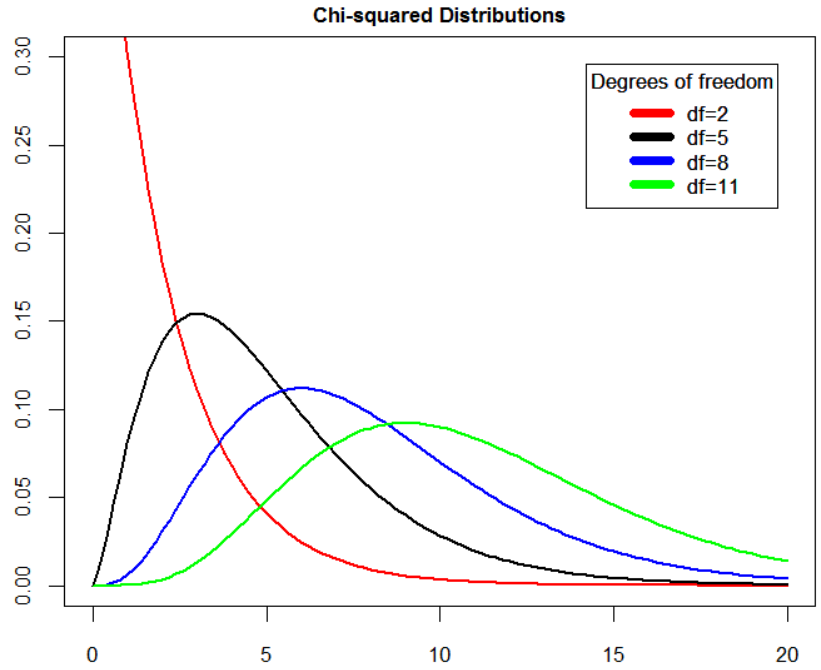
Let $z_1, \ldots, z_k$ be independent standard *Normal* random variables.

Then the random variable $X = \sum_{i=1}^{k} z_i^2$ is distributed according to the *Chi-squared* distribution with k degrees of freedom: $X \sim \chi_k^2$, and its probability density function is given by:

$$f(x) = \frac{x^{k/2-1}\, e^{-x/2}}{2^{k/2}\, \Gamma(k/2)}$$

The *Chi-squared* distribution with k degrees of freedom has mean equal to k and variance equal to $2k$.

```
> # Degrees of freedom
> deg_free <- c(2, 5, 8, 11)
> # Plot four curves in loop
> col_ors <- c("red", "black", "blue", "green")
> for (in_dex in 1:4) {
+   curve(expr=dchisq(x, df=deg_free[in_dex]),
+   xlim=c(0, 20), ylim=c(0, 0.3),
+   xlab="", ylab="", col=col_ors[in_dex],
+   lwd=2, add=as.logical(in_dex-1))
+ }  # end for
```




```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> lab_els <- paste("df", deg_free, sep="=")
> legend("topright", inset=0.05, bty="n",
+        title="Degrees of freedom", lab_els,
+        cex=0.8, lwd=6, lty=1, col=col_ors)
```

The *Chi-squared* Test for the Goodness of Fit

Goodness of Fit tests are designed to test if a set of observations fits an assumed theoretical probability distribution.

The *Chi-squared* test tests if a frequency of counts fits the specified distribution.

The *Chi-squared* statistic is the sum of squared differences between the observed frequencies o_i and the theoretical frequencies p_i:

$$\chi^2 = N \sum_{i=1}^{n} \frac{(o_i - p_i)^2}{p_i}$$

Where N is the total number of observations.

The *null hypothesis* is that the observed frequencies are consistent with the theoretical distribution.

The function `chisq.test()` performs the *Chi-squared* test and returns the statistic and its p-value *invisibly*.

The parameter `breaks` in the function `hist()` should be chosen large enough to capture the shape of the frequency distribution.

```
> # Observed frequencies from random normal data
> histo_gram <- hist(rnorm(1e3, mean=0), breaks=100, plot=FALSE)
> freq_o <- histo_gram$counts
> # Theoretical frequencies
> freq_t <- rutils::diff_it(pnorm(histo_gram$breaks))
> # Perform Chi-squared test for normal data
> chisq.test(x=freq_o, p=freq_t, rescale.p=TRUE, simulate.p.value=TR
> # Return p-value
> chisq_test <- chisq.test(x=freq_o, p=freq_t, rescale.p=TRUE, simul
> chisq_test$p.value
> # Observed frequencies from shifted normal data
> histo_gram <- hist(rnorm(1e3, mean=2), breaks=100, plot=FALSE)
> freq_o <- histo_gram$counts/sum(histo_gram$counts)
> # Theoretical frequencies
> freq_t <- rutils::diff_it(pnorm(histo_gram$breaks))
> # Perform Chi-squared test for shifted normal data
> chisq.test(x=freq_o, p=freq_t, rescale.p=TRUE, simulate.p.value=TR
> # Calculate histogram of VTI returns
> histo_gram <- hist(re_turns, breaks=100, plot=FALSE)
> freq_o <- histo_gram$counts
> # Calculate cumulative probabilities and then difference them
> freq_t <- pt((histo_gram$breaks-lo_cation)/scal_e, df=2)
> freq_t <- rutils::diff_it(freq_t)
> # Perform Chi-squared test for VTI returns
> chisq.test(x=freq_o, p=freq_t, rescale.p=TRUE, simulate.p.value=TR
```

Package *PerformanceAnalytics* for Risk and Performance Analysis

The package *PerformanceAnalytics* contains functions for risk and performance analysis.

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

`managers` is an *xts* time series containing monthly percentage returns of six asset managers (HAM1 through HAM6), the EDHEC Long-Short Equity hedge fund index, the S&P 500, and US Treasury 10-year bond and 3-month bill total returns.

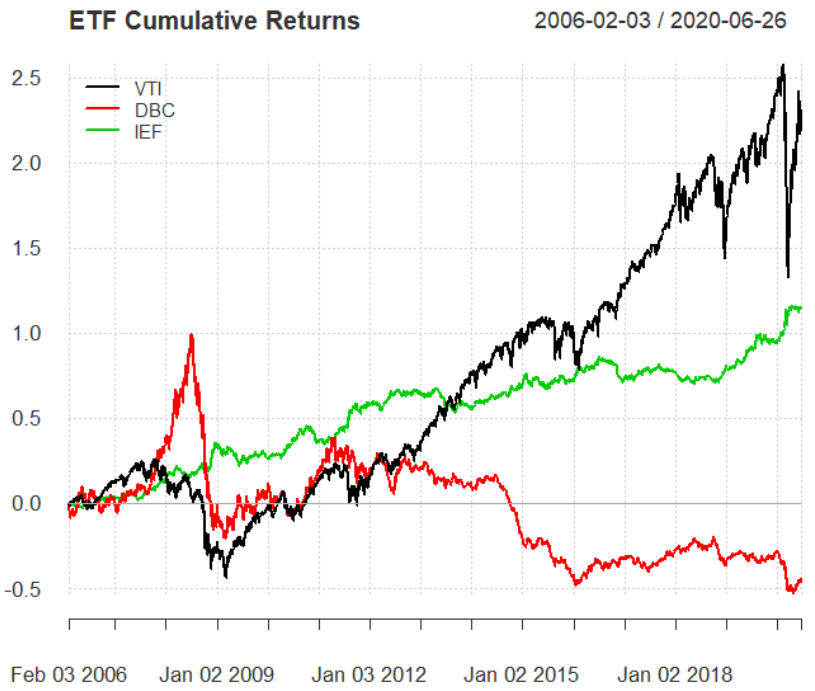
```
> # Load package PerformanceAnalytics
> library(PerformanceAnalytics)
> # Get documentation for package PerformanceAnalytics
> # Get short description
> packageDescription("PerformanceAnalytics")
> # Load help page
> help(package="PerformanceAnalytics")
> # List all objects in PerformanceAnalytics
> ls("package:PerformanceAnalytics")
> # List all datasets in PerformanceAnalytics
> data(package="PerformanceAnalytics")
> # Remove PerformanceAnalytics from search path
> detach("package:PerformanceAnalytics")
```

```
> perf_data <- unclass(data(
+     package="PerformanceAnalytics"))$results[, -(1:2)]
> apply(perf_data, 1, paste, collapse=" - ")
> # Load "managers" data set
> data(managers)
> class(managers)
> dim(managers)
> head(managers, 3)
```

Plots of Cumulative Returns

The function `chart.CumReturns()` from package *PerformanceAnalytics* plots the cumulative returns of a time series of returns.

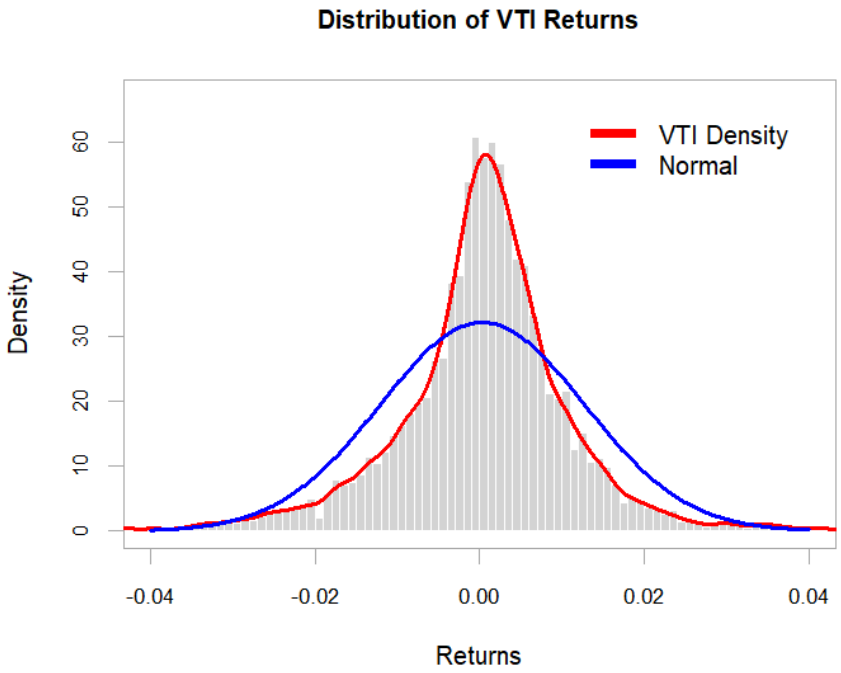
```
> # Load package "PerformanceAnalytics"
> library(PerformanceAnalytics)
> # Calculate ETF returns
> re_turns <- rutils::etf_env$re_turns[, c("VTI", "DBC", "IEF")]
> re_turns <- na.omit(re_turns)
> # Plot cumulative ETF returns
> x11(width=6, height=5)
> chart.CumReturns(re_turns, lwd=2, ylab="",
+   legend.loc="topleft", main="ETF Cumulative Returns")
```



The Distribution of Asset Returns

The function `chart.Histogram()` from package *PerformanceAnalytics* plots the histogram (frequency distribution) and the density of returns.

```
> re_turns <- rutils::etf_env$re_turns$VTI
> re_turns <- na.omit(re_turns)
> x11(width=6, height=5)
> chart.Histogram(re_turns, xlim=c(-0.04, 0.04),
+   colorset = c("lightgray", "red", "blue"), lwd=3,
+   main=paste("Distribution of", colnames(re_turns), "Returns"),
+   methods = c("add.density", "add.normal"))
> legend("topright", inset=0.05, bty="n",
+  leg=c("VTI Density", "Normal"),
+  lwd=6, lty=1, col=c("red", "blue"))
```

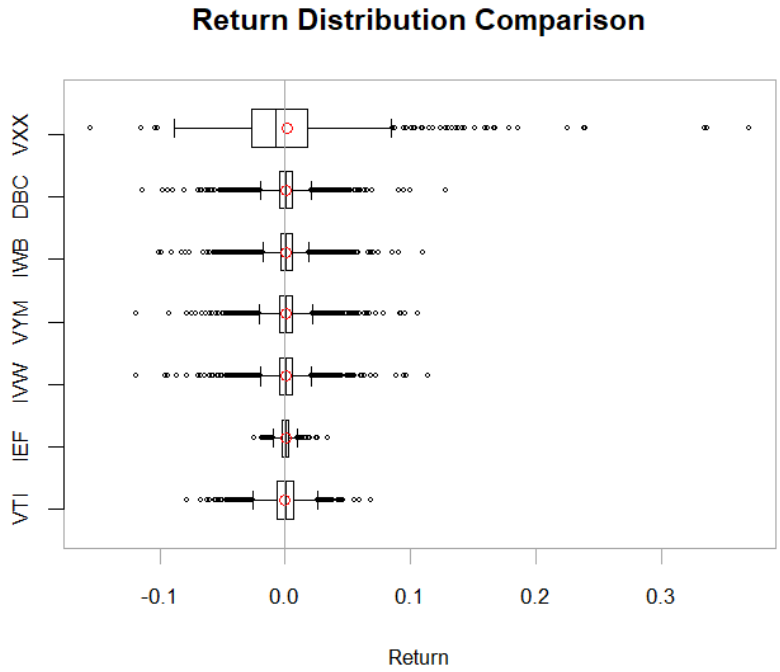


Boxplots of Returns

The function `chart.Boxplot()` from package *PerformanceAnalytics* plots a box-and-whisker plot for a distribution of returns.

The function `chart.Boxplot()` is a wrapper and calls the function `graphics::boxplot()` to plot the box plots.

A *box plot* (box-and-whisker plot) is a graphical display of a distribution of data:
The *box* represents the upper and lower quartiles,
The vertical lines (whiskers) represent values beyond the quartiles,
Open circles represent values beyond the nominal range (outliers).




```
> re_turns <- rutils::etf_env$re_turns[,
+   c("VTI", "IEF", "IVW", "VYM", "IWB", "DBC", "VXX")]
> x11(width=6, height=5)
> chart.Boxplot(names=FALSE, re_turns)
> par(cex.lab=0.8, cex.axis=0.8)
> axis(side=2, at=(1:NCOL(re_turns))/7.5-0.05,labels=colnames(re_turns))
```

The Median Absolute Deviation Estimator of Dispersion

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

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

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

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.

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

```
> # Simulate normally distributed data
> n_rows <- 1000
> da_ta <- rnorm(n_rows)
> sd(da_ta)
> mad(da_ta)
> median(abs(da_ta - median(da_ta)))
> median(abs(da_ta - median(da_ta)))/qnorm(0.75)
> # Bootstrap of sd and mad estimators
> boot_data <- sapply(1:10000, function(x) {
+   sampl_e <- da_ta[sample.int(n_rows, replace=TRUE)]
+   c(sd=sd(sampl_e), mad=mad(sampl_e))
+ })  # end sapply
> boot_data <- t(boot_data)
> # Analyze bootstrapped variance
> head(boot_data)
> sum(is.na(boot_data))
> # Means and standard errors from bootstrap
> apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=sd(x)))
> # Parallel bootstrap under Windows
> library(parallel)  # Load package parallel
> n_cores <- detectCores() - 1  # Number of cores
> clus_ter <- makeCluster(n_cores)  # Initialize compute cluster
> boot_data <- parLapply(clus_ter, 1:10000,
+   function(x, da_ta) {
+     sampl_e <- da_ta[sample.int(n_rows, replace=TRUE)]
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   }, da_ta=da_ta)  # end parLapply
> # Parallel bootstrap under Mac-OSX or Linux
> boot_data <- mclapply(1:10000, function(x) {
+     sampl_e <- da_ta[sample.int(n_rows, replace=TRUE)]
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   }, mc.cores=n_cores)  # end mclapply
> stopCluster(clus_ter)  # Stop R processes over cluster
> boot_data <- rutils::do_call(rbind, boot_data)
> # Means and standard errors from bootstrap
> apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=sd(x)))
```

The Median Absolute Deviation of Asset Returns

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

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.

```
> # VTI returns
> re_turns <- rutils::etf_env$re_turns$VTI
> re_turns <- na.omit(re_turns)
> n_rows <- NROW(re_turns)
> sd(re_turns)
> mad(re_turns)
> # Bootstrap of sd and mad estimators
> boot_data <- sapply(1:10000, function(x) {
+   sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
+   c(sd=sd(sampl_e), mad=mad(sampl_e))
+ })  # end sapply
> boot_data <- t(boot_data)
> # Means and standard errors from bootstrap
> 100*apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=sd(x)))
> # Parallel bootstrap under Windows
> library(parallel)  # Load package parallel
> n_cores <- detectCores() - 1  # Number of cores
> clus_ter <- makeCluster(n_cores)  # Initialize compute cluster
> clusterExport(clus_ter, c("n_rows", "re_turns"))
> boot_data <- parLapply(clus_ter, 1:10000,
+   function(x) {
+     sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   })  # end parLapply
> # Parallel bootstrap under Mac-OSX or Linux
> boot_data <- mclapply(1:10000, function(x) {
+     sampl_e <- re_turns[sample.int(n_rows, replace=TRUE)]
+     c(sd=sd(sampl_e), mad=mad(sampl_e))
+   }, mc.cores=n_cores)  # end mclapply
> stopCluster(clus_ter)  # Stop R processes over cluster
> boot_data <- rutils::do_call(rbind, boot_data)
> # Means and standard errors from bootstrap
> apply(boot_data, MARGIN=2, function(x)
+   c(mean=mean(x), std_error=sd(x)))
```

The Downside Deviation of Asset Returns

Some investors argue that positive returns don't represent risk, only those returns less than the target rate of return r_t.

The *Downside Deviation* (semi-deviation) σ_d is equal to the standard deviation of returns less than the target rate of return r_t:

$$\sigma_d = \sqrt{\frac{1}{n}\sum_{i=1}^{n}([r_i - r_t]_-)^2}$$

The function `DownsideDeviation()` from package *PerformanceAnalytics* calculates the downside deviation, for either the full time series (`method="full"`) or only for the subseries less than the target rate of return r_t (`method="subset"`).

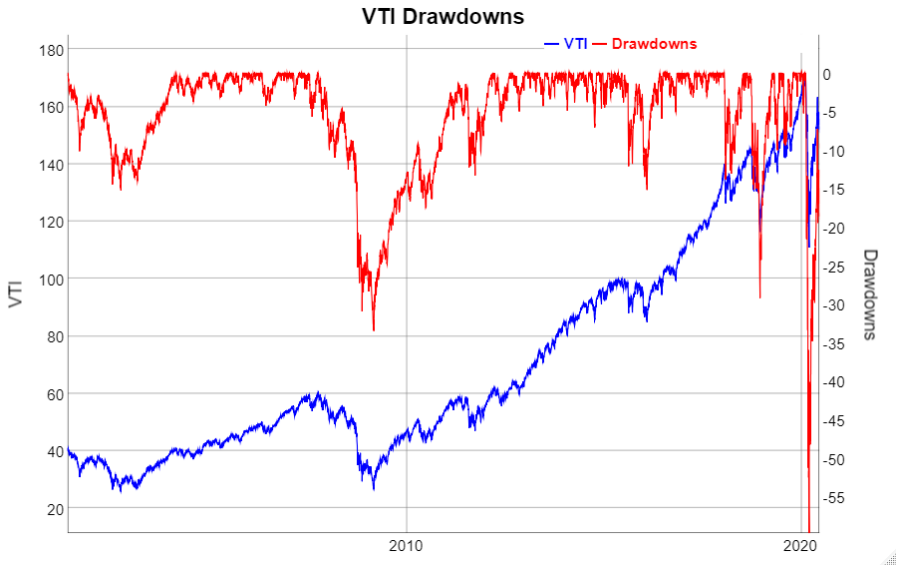
```
> library(PerformanceAnalytics)
> # Define target rate of return of 50 bps
> tar_get <- 0.005
> # Calculate the full downside returns
> returns_sub <- (re_turns - tar_get)
> returns_sub <- ifelse(returns_sub < 0, returns_sub, 0)
> n_rows <- NROW(returns_sub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(returns_sub^2)/n_rows),
+   drop(DownsideDeviation(re_turns, MAR=tar_get, method="full")))
> # Calculate the subset downside returns
> returns_sub <- (re_turns - tar_get)
> returns_sub <- returns_sub[returns_sub < 0]
> n_rows <- NROW(returns_sub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(returns_sub^2)/n_rows),
+   DownsideDeviation(re_turns, MAR=tar_get, method="subset"))
```

Drawdown Risk

The *drawdown* is the drop in prices from their historical peak, and is equal to the difference between the prices minus the cumulative maximum of the prices.

Drawdown risk determines the risk of liquidation due to stop loss limits.

The function `table.Drawdowns()` from package *PerformanceAnalytics* calculates a data frame of drawdowns.



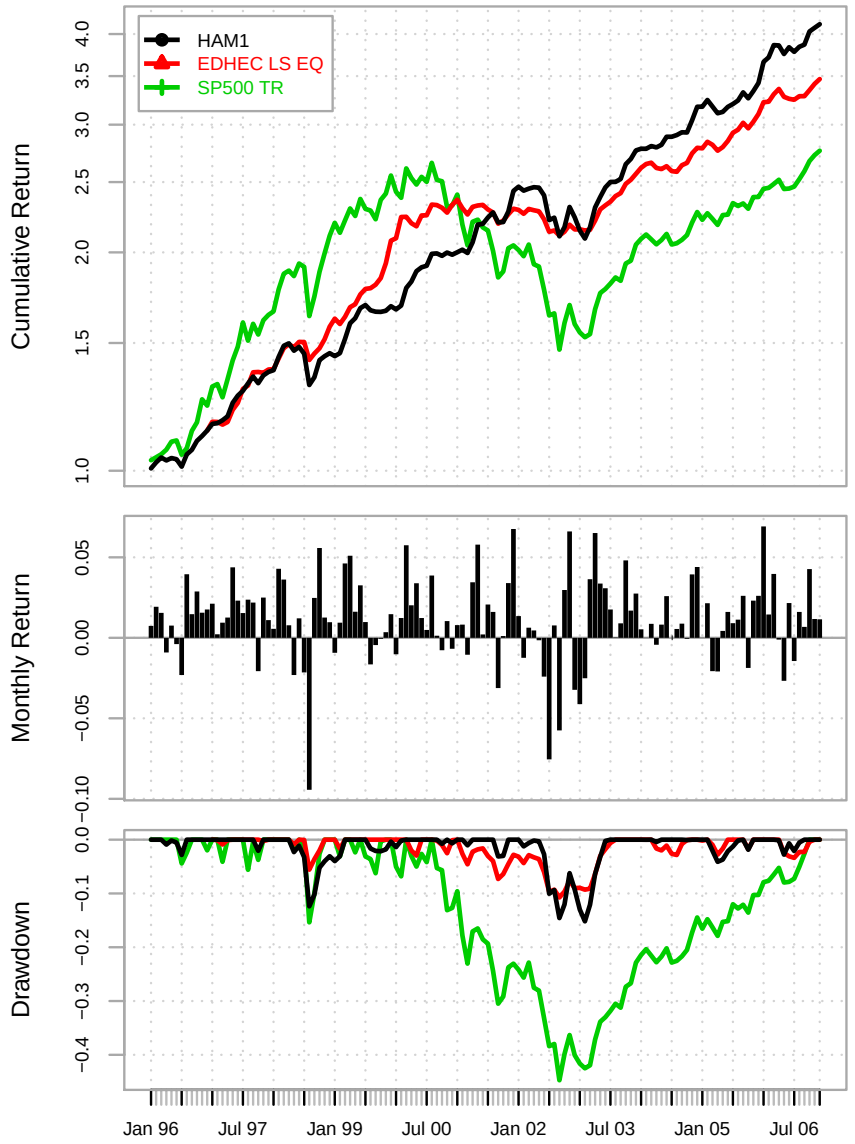
From	Trough	To	Depth	Length	To
2007-10-10	2009-03-09	2012-09-13	-0.20	1243.00	
2001-06-06	2002-10-09	2004-11-12	-0.13	864.00	
2020-02-20	2020-03-23	2020-08-24	-0.09	130.00	
2018-09-21	2018-12-24	2019-04-23	-0.05	146.00	
2015-06-24	2016-02-11	2016-07-08	-0.04	263.00	

```
> # Calculate time series of VTI drawdowns
> clos_e <- log(na.omit(rutils::etf_env$price_s$VTI))
> draw_downs <- (clos_e - cummax(clos_e))
> # PerformanceAnalytics plot of VTI drawdowns
> re_turns <- rutils::diff_it(log(clos_e))
> PerformanceAnalytics::chart.Drawdown(re_turns,
+   ylab="", main="VTI Drawdowns")
> # PerformanceAnalytics table of VTI drawdowns
> PerformanceAnalytics::table.Drawdowns(re_turns)
> # dygraph plot of VTI drawdowns
> da_ta <- cbind(clos_e, draw_downs)
> col_names <- c("VTI", "Drawdowns")
> colnames(da_ta) <- col_names
> dygraphs::dygraph(da_ta, main="VTI Drawdowns") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], valueRange=c(min(da_ta[, "Drawdowns"]), 5), independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue") %>%
+   dySeries(name=col_names[2], axis="y2", col="red")
> # Plot VTI drawdowns
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> x11(width=6, height=5)
> quantmod::chart_Series(x=draw_downs, name="VTI Drawdowns", theme=plot_theme)
```

PerformanceSummary Plots

The function `charts.PerformanceSummary()` from package *PerformanceAnalytics* plots three charts: cumulative returns, return bars, and drawdowns, for time series of returns.

```
> data(managers)
> charts.PerformanceSummary(ham_1,
+   main="", lwd=2, ylog=TRUE)
```



The Loss Distribution of Asset Returns

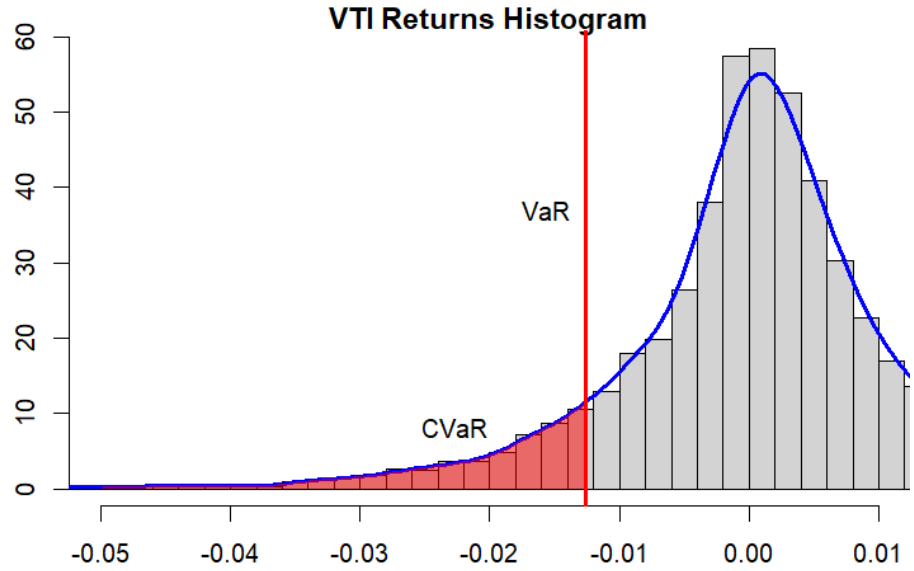
The distribution of returns has a long left tail of negative returns representing the risk of loss.

The *Value at Risk* (VaR) is equal to the quantile of returns corresponding to a given confidence level α.

The *Conditional Value at Risk* (CVaR) is equal to the average of negative returns less than the VaR.

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

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




```
> # VTI percentage returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> conf_level <- 0.1
> va_r <- quantile(re_turns, conf_level)
> c_var <- mean(re_turns[re_turns < va_r])
> # Plot histogram of VTI returns
> histo_gram <- hist(re_turns, col="lightgrey",
+   xlab="returns", ylab="frequency", breaks=100,
+   xlim=c(-0.05, 0.01), freq=FALSE, main="VTI Returns Histogram")
> # Calculate density
> densi_ty <- density(re_turns, adjust=1.5)
```

```
> # Plot density
> lines(densi_ty, lwd=3, col="blue")
> # Plot line for VaR
> abline(v=va_r, col="red", lwd=3)
> text(x=va_r, y=20, labels="VaR", lwd=2, srt=90, pos=2)
> # Plot polygon shading for CVaR
> var_max <- -0.06
> rang_e <- (densi_ty$x < va_r) &  (densi_ty$x > var_max)
> polygon(c(var_max, densi_ty$x[rang_e], va_r),
+   c(0, densi_ty$y[rang_e], 0), col=rgb(1, 0, 0,0.5), border=NA)
> text(x=va_r, y=3, labels="CVaR", lwd=2, pos=2)
```

Value at Risk (*VaR*)

The *Value at Risk* (VaR) is equal to the quantile of returns corresponding to a given confidence level α:

$$\alpha = \int_{-\infty}^{\mathrm{VaR}(\alpha)} \mathrm{f}(r)\,\mathrm{d}r$$

Where $\mathrm{f}(r)$ is the probability density (distribution) of returns.

At a high confidence level, the value of VaR is subject to estimation error, and various numerical methods are used to approximate it.

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 `VaR()` from package *PerformanceAnalytics* calculates the *Value at Risk* using several different methods.

```
> # VTI percentage returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> conf_level <- 0.02
> # Calculate VaR as quantile
> va_r <- quantile(re_turns, conf_level)
> # Or by sorting
> sort_ed <- sort(as.numeric(re_turns))
> in_dex <- round(conf_level*NROW(re_turns))
> va_r <- sort_ed[in_dex]
> # PerformanceAnalytics VaR
> PerformanceAnalytics::VaR(re_turns,
+   p=(1-conf_level), method="historical")
> all.equal(unname(va_r),
+   as.numeric(PerformanceAnalytics::VaR(re_turns,
+   p=(1-conf_level), method="historical")))
```

Conditional Value at Risk (*CVaR*)

The *Conditional Value at Risk* (CVaR) is equal to the average of negative returns less than the VaR:

$$\mathrm{CVaR} = \frac{1}{\alpha} \int_0^{\alpha} \mathrm{VaR}(p)\, \mathrm{d}p$$

The *Conditional Value at Risk* is also called the *Expected Shortfall* (*ES*), or the Expected Tail Loss (*ETL*).

The function `ETL()` from package *PerformanceAnalytics* calculates the *Conditional Value at Risk* using several different methods.

```
> # Calculate VaR as quantile
> va_r <- quantile(re_turns, conf_level)
> # Calculate CVaR as expected loss
> c_var <- mean(re_turns[re_turns < va_r])
> # Or by sorting
> sort_ed <- sort(as.numeric(re_turns))
> in_dex <- round(conf_level*NROW(re_turns))
> va_r <- sort_ed[in_dex]
> c_var <- mean(sort_ed[1:in_dex])
> # PerformanceAnalytics VaR
> PerformanceAnalytics::ETL(re_turns,
+   p=(1-conf_level), method="historical")
> all.equal(c_var,
+   as.numeric(PerformanceAnalytics::ETL(re_turns,
+   p=(1-conf_level), method="historical")))
```

Risk and Return Statistics

The function `table.Stats()` from package *PerformanceAnalytics* calculates a data frame of risk and return statistics of the return distributions.

```
> # Calculate the risk-return statistics
> risk_ret <-
+   PerformanceAnalytics::table.Stats(rutils::etf_env$re_turns)
> class(risk_ret)
> # Transpose the data frame
> risk_ret <- as.data.frame(t(risk_ret))
> # Add Name column
> risk_ret$Name <- rownames(risk_ret)
> # Add Sharpe ratio column
> risk_ret$Sharpe <- risk_ret$"Arithmetic Mean"/risk_ret$Stdev
> # Sort on Sharpe ratio
> risk_ret <- risk_ret[order(risk_ret$Sharpe, decreasing=TRUE), ]
```

	Sharpe	Skewness	Kurtosis
USMV	0.056	-1.018	26.18
MTUM	0.050	-0.827	15.94
IEF	0.049	0.018	3.02
QUAL	0.046	-0.690	18.27
VLUE	0.041	-1.191	21.31
XLP	0.031	-0.082	9.62
XLY	0.028	-0.384	7.86
GLD	0.026	-0.327	6.21
XLV	0.025	0.076	10.81
IWB	0.024	-0.409	10.71
VTI	0.024	-0.399	11.63
IVW	0.024	-0.316	9.50
VYM	0.024	-0.708	14.67
XLU	0.024	0.025	13.22
IWD	0.024	-0.505	12.86
VTV	0.024	-0.685	13.80
IVE	0.023	-0.492	10.16
TLT	0.023	-0.014	4.64
EEM	0.022	0.000	15.09
IWF	0.022	-0.714	34.62
XLI	0.022	-0.400	7.78
XLB	0.019	-0.422	5.61
XLK	0.018	0.088	7.58
VNQ	0.015	-0.555	17.52
VEU	0.013	-0.540	11.47
XLE	0.011	-0.573	13.96
XLF	0.011	-0.140	14.18
SVXY	0.006	-17.179	563.14
DBC	-0.008	-0.414	3.05
USO	-0.029	-1.197	15.75
VXX	-0.073	1.169	5.37

Investor Risk and Return Preferences

Investors typically prefer larger *odd moments* of the return distribution (mean, skewness), and smaller *even moments* (variance, kurtosis).

But positive skewness is often associated with lower returns, which can be observed in the *VIX* volatility ETFs, *VXX* and *SVXY*.

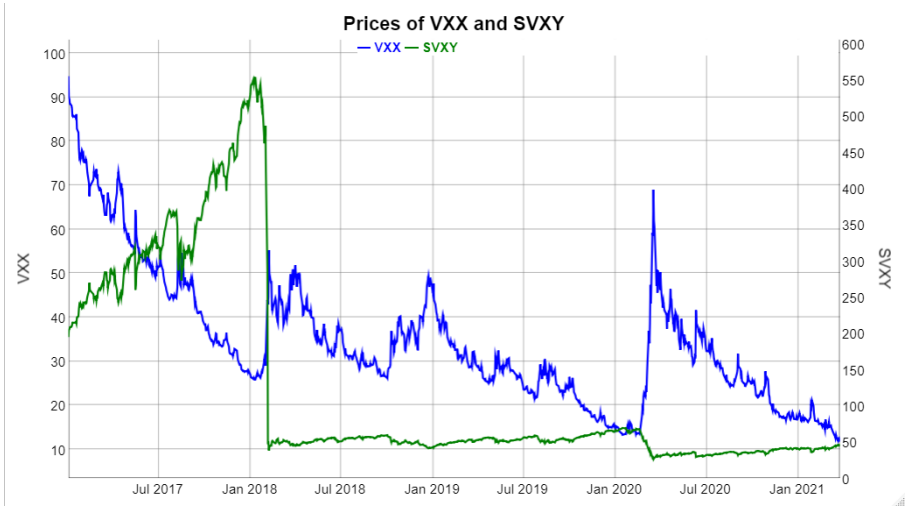
The *VXX* ETF is long the *VIX* index (effectively long an option), so it has positive skewness and small kurtosis, but negative returns (it's short market risk).

Since the *VXX* is effectively long an option, it pays option premiums so it has negative returns most of the time, with isolated periods of positive returns when markets drop.

The *SVXY* ETF is short the *VIX* index, so it has negative skewness and large kurtosis, but positive returns (it's long market risk).

Since the *SVXY* is effectively short an option, it earns option premiums so it has positive returns most of the time, but it suffers sharp losses when markets drop.

	Sharpe	Skewness	Kurtosis
VXX	-0.073	1.17	5.37
SVXY	0.006	-17.18	563.14




```
> # dygraph plot of VTI drawdowns
> price_s <- na.omit(rutils::etf_env$price_s[, c("VXX", "SVXY")])
> price_s <- price_s["2017/"]
> col_names <- c("VXX", "SVXY")
> colnames(price_s) <- col_names
> dygraphs::dygraph(price_s, main="Prices of VXX and SVXY") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", strokeWidth=2, col="blue")
+   dySeries(name=col_names[2], axis="y2", strokeWidth=2, col="green
+   dyLegend(show="always", width=500)
```

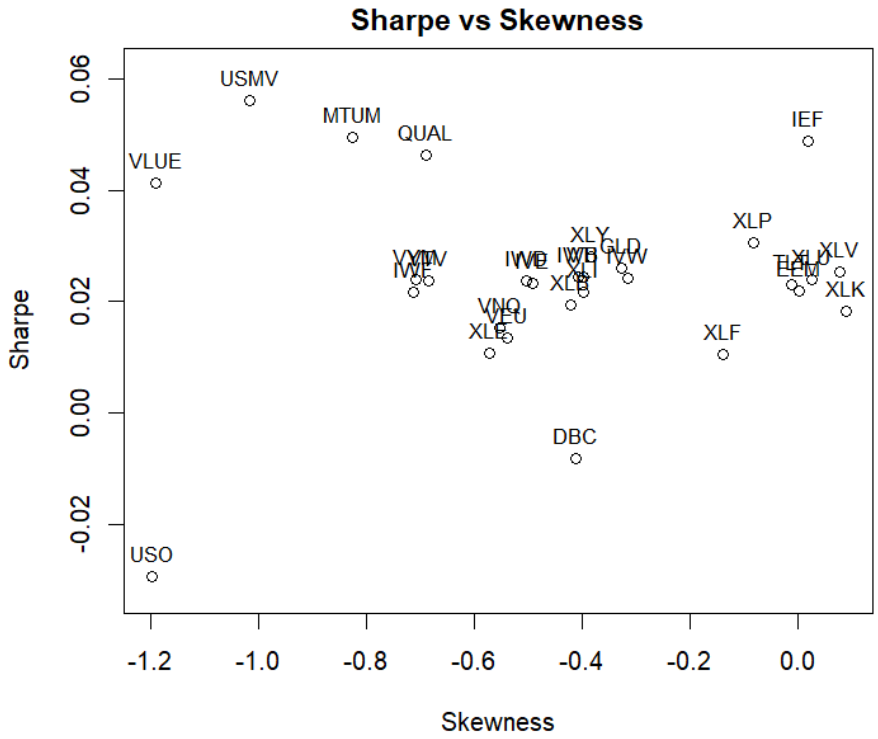
Skewness and Return Tradeoff

Similarly to the *VXX* and *SVXY*, for most other ETFs positive skewness is often associated with lower returns.

Some of the exceptions are bond ETFs (like *IEF*), which have both non-negative skewness and positive returns.

Another exception are commodity ETFs (like *USO* oil), which have both negative skewness and negative returns.

```
> # Remove VIX volatility ETF data
> risk_ret <- risk_ret[-match(c("VXX", "SVXY"), risk_ret$Name), ]
> # Plot scatterplot of Sharpe vs Skewness
> plot(Sharpe ~ Skewness, data=risk_ret,
+      ylim=1.1*range(risk_ret$Sharpe),
+      main="Sharpe vs Skewness")
> # Add labels
> text(x=risk_ret$Skewness, y=risk_ret$Sharpe,
+     labels=risk_ret$Name, pos=3, cex=0.8)
> # Plot scatterplot of Kurtosis vs Skewness
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 1), oma=c(0, 0, 0, 0))
> plot(Kurtosis ~ Skewness, data=risk_ret,
+      ylim=c(1, max(risk_ret$Kurtosis)),
+      main="Kurtosis vs Skewness")
> # Add labels
> text(x=risk_ret$Skewness, y=risk_ret$Kurtosis,
+     labels=risk_ret$Name, pos=1, cex=0.8)
```



Risk-adjusted Return Measures

The *Sharpe* ratio measures the excess returns per unit of risk, and is equal to the excess returns (over a risk-free return r_f) divided by the standard deviation of the returns:

$$S_r = \frac{E[r - r_f]}{\sigma}$$

The *Sortino* ratio is equal to the excess returns divided by the *downside deviation* σ_d (standard deviation of returns that are less than a target rate of return r_t):

$$S_r = \frac{E[r - r_t]}{\sigma_d}$$

The *Calmar* ratio is equal to the excess returns divided by the maximum drawdown of the returns:

$$C_r = \frac{E[r - r_f]}{DD}$$

```
> re_turns <- rutils::etf_env$re_turns[, c("VTI", "IEF")]
> re_turns <- na.omit(re_turns)
> # Calculate the Sharpe ratio
> PerformanceAnalytics::SharpeRatio(re_turns)
> # Calculate the Sortino ratio
> PerformanceAnalytics::SortinoRatio(re_turns)
> # Calculate the Calmar ratio
> PerformanceAnalytics::CalmarRatio(re_turns)
> # Calculate the returns statistics
> tail(PerformanceAnalytics::table.Stats(re_turns), 4)
```

Homework Assignment

Required

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

Recommended

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