FRE7241 Algorithmic Portfolio Management

Lecture#1, Spring 2025

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

March 18, 2025

Welcome Students!

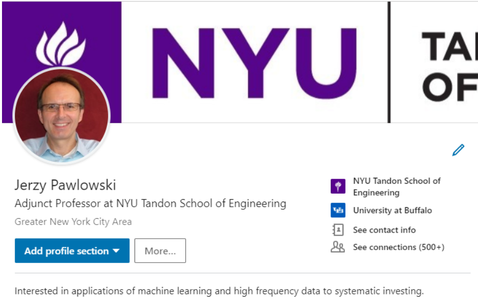
My name is Jerzy Pawlowski jp3900@nyu.edu

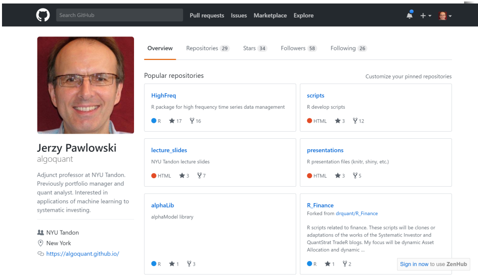
I'm an adjunct professor at NYU Tandon because I love teaching and I want to share my professional knowledge with young, enthusiastic students.

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 banker* (structurer), and *quant risk 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).
This course is challenging, so it requires devoting a significant amount of time!

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).
This course is challenging, so it requires devoting a significant amount of time!

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).
This course is challenging, so it requires devoting a significant amount of time!

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 Recommendations

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

Homeworks and Tests

Homeworks and Tests

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

The tests will be announced several days in advance.

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

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

Students will be allowed to copy code from the lecture slides, 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).

Students are encouraged to use AI applications, such as ChatGPT, *GitHub Copilot*, *Copilot for RStudio*, etc. But you must include the name of the AI application in your solution.

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

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

Homeworks and Tests

Homeworks and Tests

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

The tests will be announced several days in advance.

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

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

Students will be allowed to copy code from the lecture slides, 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).

Students are encouraged to use AI applications, such as ChatGPT, *GitHub Copilot*, *Copilot for RStudio*, etc. But you must include the name of the AI application in your solution.

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 Lakshay Dua ld3074@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 assignments will mostly require 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 assignments will mostly require 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 the 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

Homeworks and tests will be graded and assigned numerical scores. Each part of homeworks and 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

Homeworks and tests will be graded and assigned numerical scores. Each part of homeworks and 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 percentage scores obtained for all the homeworks and tests. The percentage scores will be calculated by adding together the scores of all the homeworks and tests, and dividing them by the sum of the maximum scores. The percentage scores are usually very high - above 90%. So a very high percentage score will not guarantee an A letter grade, since grading will also depend on the difficulty of the assignments.

Course Grading Policies

Numerical Scores

Homeworks and tests will be graded and assigned numerical scores. Each part of homeworks and 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 percentage scores obtained for all the homeworks and tests. The percentage scores will be calculated by adding together the scores of all the homeworks and tests, and dividing them by the sum of the maximum scores. The percentage scores are usually very high - above 90%. So a very high percentage score will not 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

- *Advances in Financial Machine Learning* by Marcos Lopez de Prado - Machine learning techniques applied to trading and portfolio management.
- *Systematic Trading* by Robert Carver - Practical trading knowledge by an experienced portfolio manager.
- *Systematic Investing* by Robert Carver - Practical investment knowledge by a successful investor.
- *Quantitative Trading* by Xin Guo, Tze Leung Lai, Howard Shek, Samuel Po-Shing Wong - Advanced topics in quantitative trading by academic experts.
- *Financial Data and Models Using R* by Clifford Ang - Good introduction to time series, portfolio optimization, and performance measures.
- *Automated Trading* by Chris Conlan - How to implement practical computer trading systems.
- *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 Textbooks

Supplementary Textbooks

- *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/

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
> symbolv <- c("SPY", "VTI", "QQQ", "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", "AIEC
> # Read etf database into data frame
> etflist <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data
> rownames(etflist) <- etflist$Symbol
> # Select from etflist only those ETF's in symbolv
> etflist <- etflist[symbolv, ]
> # Shorten names
> etfnames <- sapply(etflist$Name, function(name) {
+   namesplit <- strsplit(name, split=" ")[[1]]
+   namesplit <- namesplit[c(-1, -NROW(namesplit))]
+   name_match <- match("Select", namesplit)
+   if (!is.na(name_match))
+     namesplit <- namesplit[-name_match]
+   paste(namesplit, collapse=" ")
+ })  # end sapply
> etflist$Name <- etfnames
> etflist["IEF", "Name"] <- "10 year Treasury Bond Fund"
> etflist["TLT", "Name"] <- "20 plus year Treasury Bond Fund"
> etflist["XLY", "Name"] <- "Consumer Discr. Sector Fund"
> etflist["EEM", "Name"] <- "Emerging Market Stock Fund"
> etflist["MTUM", "Name"] <- "Momentum Factor Fund"
> etflist["SVXY", "Name"] <- "Short VIX Futures"
> etflist["VXX", "Name"] <- "Long VIX Futures"
> etflist["DBC", "Name"] <- "Commodity Futures Fund"
> etflist["USO", "Name"] <- "WTI Oil Futures Fund"
> etflist["GLD", "Name"] <- "Physical Gold Fund"
```

ETF Database for Investment Strategies

The database 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.

The *SPY ETF* owns the *S&P500* index constituents. *SPY* is the biggest, the most liquid, and the oldest ETF. SPY has over $400 billion of shares outstanding, and trades over $20 billion per day, at a bid-ask spread of only one tick (cent=$0.01, or about 0.0022%).

The *QQQ ETF* owns the *Nasdaq-100* index constituents.

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.

Symbol	Name	Fund.Type
SPY	S&P 500	US Equity ETF
VTI	Total Stock Market	US Equity ETF
QQQ	QQQ Trust	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
AIEQ	AI Powered Equity	US Asset Allocation ET

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.

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*.

Downloading ETF Prices Using Package *quantmod*

The function `getSymbols()` downloads time series data into the specified *environment*.

`getSymbols()` downloads the daily *OHLC* prices and trading volume (Open, High, Low, Close, Adjusted, Volume).

`getSymbols()` creates objects in the specified *environment* from the input strings (names), and assigns the data to those objects, without returning them as a function value, as a *side effect*.

If the argument `"auto.assign"` is set to `FALSE`, then `getSymbols()` returns the data, instead of assigning it silently.

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Tiingo* and *Alpha Vantage* as the only major providers of free daily *OHLC* stock prices.

But Quandl doesn't provide free *ETF* prices, leaving *Alpha Vantage* as the best provider of free daily *ETF* prices.

```
> # Select ETF symbols for asset allocation
> symbolv <- c("SPY", "VTI", "QQQ", "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", "AIEQ
> library(rutils)  # Load package rutils
> etfenv <- new.env()  # New environment for data
> # Boolean vector of symbols already downloaded
> isdown <- symbolv %in% ls(etfenv)
> # Download data for symbolv using single command - creates pacing
> getSymbols.av(symbolv, adjust=TRUE, env=etfenv,
+   output.size="full", api.key="T7JPW54ES8G75310")
> # Download data from Alpha Vantage using while loop
> nattempts <- 0  # number of download attempts
> while ((sum(!isdown) > 0) & (nattempts < 10)) {
+   # Download data and copy it into environment
+   nattempts <- nattempts + 1
+   cat("Download attempt = ", nattempts, "\n")
+   for (symboln in na.omit(symbolv[!isdown][1:5])) {
+     cat("Processing: ", symboln, "\n")
+     tryCatch(  # With error handler
+ quantmod::getSymbols.av(symboln, adjust=TRUE, env=etfenv, auto.ass
+ # Error handler captures error condition
+ error=function(msg) {
+   print(paste0("Error handler: ", msg))
+ },  # end error handler
+ finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdown <- symbolv %in% ls(etfenv)
+   cat("Pausing 1 minute to avoid pacing...\n")
+   Sys.sleep(65)
+ }  # end while
> # Download all symbolv using single command - creates pacing error
> # quantmod::getSymbols.av(symbolv, env=etfenv, adjust=TRUE, from="
```

Inspecting ETF Prices in an Environment

The function `get()` retrieves objects that are referenced using character strings, instead of their names.

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

```
> ls(etfenv)  # List files in etfenv
> # Get class of object in etfenv
> class(get(x=symbolv[1], envir=etfenv))
> # Another way
> class(etfenv$VTI)
> colnames(etfenv$VTI)
> # Get first 3 rows of data
> head(etfenv$VTI, 3)
> # Get last 11 rows of data
> tail(etfenv$VTI, 11)
> # Get class of all objects in etfenv
> eapply(etfenv, class)
> # Get class of all objects in R workspace
> lapply(ls(), function(namev) class(get(namev)))
> # Get end dates of all objects in etfenv
> as.Date(sapply(etfenv, end))
```

Adjusting Stock Prices Using Package *quantmod*

Traded stock and bond prices experience jumps after splits and dividends, and must be adjusted to account for them.

The function `adjustOHLC()` adjusts *OHLC* prices.

The function `get()` retrieves objects that are referenced using character strings, instead of their names.

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

The functions `get()` and `assign()` allow retrieving and assigning values to objects that are referenced using character strings.

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

If the argument `"adjust"` in function `getSymbols()` is set to `TRUE`, then `getSymbols()` returns adjusted data.

```
> # Check of object is an OHLC time series
> is.OHLC(etfenv$VTI)
> # Adjust single OHLC object using its name
> etfenv$VTI <- adjustOHLC(etfenv$VTI, use.Adjusted=TRUE)
>
> # Adjust OHLC object using string as name
> assign(symbolv[1], adjustOHLC(
+     get(x=symbolv[1], envir=etfenv), use.Adjusted=TRUE),
+   envir=etfenv)
>
> # Adjust objects in environment using vector of strings
> for (symboln in ls(etfenv)) {
+   assign(symboln,
+    adjustOHLC(get(symboln, envir=etfenv), use.Adjusted=TRUE),
+    envir=etfenv)
+ }  # end for
```

Extracting Time Series from Environments

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

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

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

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

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

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

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

```
> library(rutils)  # Load package rutils
> # Define ETF symbols
> symbolv <- c("VTI", "VEU", "IEF", "VNQ")
> # Extract symbolv from rutils::etfenv
> pricev <- mget(symbolv, envir=rutils::etfenv)
> # pricev is a list of xts series
> class(pricev)
> class(pricev[[1]])
> tail(pricev[[1]])
> # Extract close prices
> pricev <- lapply(pricev, quantmod::Cl)
> # Collapse list into time series the hard way
> prices2 <- cbind(pricev[[1]], pricev[[2]], pricev[[3]], pricev[[4
> class(price2)
> dim(price2)
> # Collapse list into time series using do.call()
> pricev <- do.call(cbind, pricev)
> all.equal(price2, pricev)
> class(pricev)
> dim(pricev)
> # Or extract and cbind in single step
> pricev <- do.call(cbind, lapply(
+   mget(symbolv, envir=rutils::etfenv), quantmod::Cl))
> # Or extract and bind all data, subset by symbolv
> pricev <- lapply(symbolv, function(symboln) {
+     quantmod::Cl(get(symboln, envir=rutils::etfenv))
+ })  # end lapply
> # Or loop over etfenv without anonymous function
> pricev <- do.call(cbind,
+   lapply(as.list(rutils::etfenv)[symbolv], quantmod::Cl))
> # Same, but works only for OHLC series - produces error
> pricev <- do.call(cbind,
+   eapply(rutils::etfenv, quantmod::Cl)[symbolv])
```

Managing Time Series

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

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

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

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

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

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

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

Calculating Percentage Returns from Close Prices

The function `quantmod::dailyReturn()` calculates the percentage daily returns from the *Close* prices.

The `lapply()` and `sapply()` functionals perform a loop over the columns of *zoo* and *xts* series.

```
> # Extract VTI prices
> pricev <- etfenv$prices[ ,"VTI"]
> pricev <- na.omit(pricev)
> # Calculate percentage returns "by hand"
> pricel <- as.numeric(pricev)
> pricel <- c(pricel[1], pricel[-NROW(pricel)])
> pricel <- xts(pricel, zoo::index(pricev))
> retp <- (pricev-pricel)/pricel
> # Calculate percentage returns using dailyReturn()
> retd <- quantmod::dailyReturn(pricev)
> head(cbind(retd, retp))
> all.equal(retd, retp, check.attributes=FALSE)
> # Calculate returns for all prices in etfenv$prices
> retp <- lapply(etfenv$prices, function(xtsv) {
+   retd <- quantmod::dailyReturn(na.omit(xtsv))
+   colnames(retd) <- names(xtsv)
+   retd
+ })  # end lapply
> # "retp" is a list of xts
> class(retp)
> class(retp[[1]])
> # Flatten list of xts into a single xts
> retp <- do.call(cbind, retp)
> class(retp)
> dim(retp)
> # Copy retp into etfenv and save to .RData file
> # assign("retp", retp, envir=etfenv)
> etfenv$retp <- retp
> save(etfenv, file="/Users/jerzy/Develop/lecture_slides/data/etf_da
```

Managing Data Inside Environments

The function `as.environment()` coerces objects (listv) into an environment.

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

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

```
> library(rutils)
> startd <- "2012-05-10"; endd <- "2013-11-20"
> # Select all objects in environment and return as environment
> newenv <- as.environment(eapply(etfenv, "[",
+             paste(startd, endd, sep="/")))
> # Select only symbolv in environment and return as environment
> newenv <- as.environment(
+   lapply(as.list(etfenv)[symbolv], "[",
+    paste(startd, endd, sep="/")))
> # Extract and cbind Close prices and return to environment
> assign("prices", rutils::do_call(cbind,
+   lapply(ls(etfenv), function(symboln) {
+     xtsv <- quantmod::Cl(get(symboln, etfenv))
+     colnames(xtsv) <- symboln
+     xtsv
+   })), envir=newenv)
> # Get sizes of OHLC xts series in etfenv
> sapply(mget(symbolv, envir=etfenv), object.size)
> # Extract and cbind adjusted prices and return to environment
> colname <- function(xtsv)
+   strsplit(colnames(xtsv), split="[.]")[[1]][1]
> assign("prices", rutils::do_call(cbind,
+          lapply(mget(etfenv$symbolv, envir=etfenv),
+                 function(xtsv) {
+                   xtsv <- Ad(xtsv)
+                   colnames(xtsv) <- colname(xtsv)
+                   xtsv
+          })), envir=newenv)
```

Stock Databases And Survivorship Bias

The file `sp500_constituents.csv` contains a *data frame* of over 700 present (and also some past) *S&P500* index constituents.

The file `sp500_constituents.csv` is updated with stocks recently added to the *S&P500* index by downloading the *SPY ETF Holdings*.

But the file `sp500_constituents.csv` doesn't include companies that have gone bankrupt. For example, it doesn't include Enron, which was in the *S&P500* index before it went bankrupt in 2001.

Most databases of stock prices don't include companies that have gone bankrupt or have been liquidated.

This introduces a *survivorship bias* to the data, which can skew portfolio simulations and strategy backtests.

Accurate strategy simulations require starting with a portfolio of companies at a "point in time" in the past, and tracking them over time.

Research databases like the *WRDS* database provide stock prices of companies that are no longer traded.

The stock tickers are stored in the column `"Ticker"` of the `sp500` *data frame*.

Some tickers (like "BRK.B" and "BF.B") are not valid symbols in *Tiingo*, so they must be renamed.

```
> # Load data frame of S&P500 constituents from CSV file
> sp500 <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/s
> # Inspect data frame of S&P500 constituents
> dim(sp500)
> colnames(sp500)
> # Extract tickers from the column Ticker
> symbolv <- sp500$Ticker
> # Get duplicate tickers
> tablev <- table(symbolv)
> duplicatv <- tablev[tablev > 1]
> duplicatv <- names(duplicatv)
> # Get duplicate records (rows) of sp500
> sp500[symbolv %in% duplicatv, ]
> # Get unique tickers
> symbolv <- unique(symbolv)
> # Find index of ticker "BRK.B"
> which(symbolv=="BRK.B")
> # Rename "BRK.B" to "BRK-B" and "BF.B" to "BF-B"
> symbolv[which(symbolv=="BRK.B")] <- "BRK-B"
> symbolv[which(symbolv=="BF.B")] <- "BF-B"
```

Downloading Stock Time Series From *Tiingo*

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Tiingo*, *Alpha Vantage*, and Quandl as the only major providers of free daily *OHLC* stock prices.

But Quandl doesn't provide free *ETF* prices, while *Tiingo* does.

The function `getSymbols()` has a *method* for downloading time series data from *Tiingo*, called `getSymbols.tiingo()`.

Users must first obtain a *Tiingo API key*, and then pass it in `getSymbols.tiingo()` calls:
https://www.tiingo.com/

Note that the data are downloaded as `xts` time series, with a date-time index of class `POSIXct` (not `Date`).

```
> # Load package rutils
> library(rutils)
> # Create new environment for data
> sp500env <- new.env()
> # Boolean vector of symbols already downloaded
> isdown <- symbolv %in% ls(sp500env)
> # Download in while loop from Tiingo and copy into environment
> nattempts <- 0  # Number of download attempts
> while ((sum(!isdown) > 0) & (nattempts<3)) {
+   # Download data and copy it into environment
+   nattempts <- nattempts + 1
+   cat("Download attempt = ", nattempts, "\n")
+   for (symboln in symbolv[!isdown]) {
+     cat("processing: ", symboln, "\n")
+     tryCatch(  # With error handler
+ quantmod::getSymbols(symboln, src="tiingo", adjust=TRUE, auto.assi
+            from="1990-01-01", env=sp500env, api.key="j84ac2b9c5bde
+ # Error handler captures error condition
+ error=function(msg) {
+   print(paste0("Error handler: ", msg))
+ },  # end error handler
+ finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdown <- symbolv %in% ls(sp500env)
+   Sys.sleep(2)  # Wait 2 seconds until next attempt
+ }  # end while
> class(sp500env$AAPL)
> class(zoo::index(sp500env$AAPL))
> tail(sp500env$AAPL)
> symbolv[!isdown]
```

Coercing Date-time Indices

The date-time indices of the *OHLC* stock prices are in the `POSIXct` format suitable for intraday prices, not daily prices.

The function `as.Date()` coerces `POSIXct` objects into `Date` objects.

The function `get()` retrieves objects that are referenced using character strings, instead of their names.

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

The functions `get()` and `assign()` allow retrieving and assigning values to objects that are referenced using character strings.

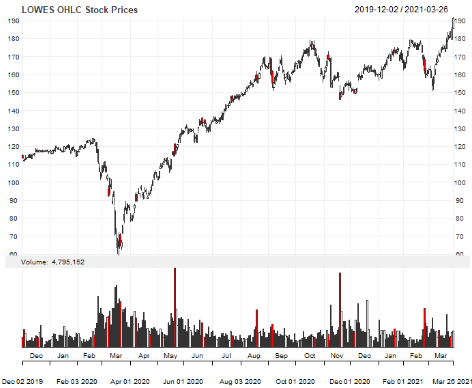
```
> # The date-time index of AAPL is POSIXct
> class(zoo::index(sp500env$AAPL))
> # Coerce the date-time index of AAPL to Date
> zoo::index(sp500env$AAPL) <- as.Date(zoo::index(sp500env$AAPL))
> # Coerce all the date-time indices to Date
> for (symboln in ls(sp500env)) {
+   ohlc <- get(symboln, envir=sp500env)
+   zoo::index(ohlc) <- as.Date(zoo::index(ohlc))
+   assign(symboln, ohlc, envir=sp500env)
+ }  # end for
```

Managing Exceptions in Stock Symbols

The column names for symbol "LOW" (Lowe's company) must be renamed for the extractor function quantmod::Lo() to work properly.

Tickers which contain a dot in their name (like "BRK.B") are not valid symbols in R, so they must be downloaded separately and renamed.

```
> # "LOW.Low" is a bad column name
> colnames(sp500env$LOW)
> strsplit(colnames(sp500env$LOW), split="[.]")
> do.call(cbind, strsplit(colnames(sp500env$LOW), split="[.]"))
> do.call(cbind, strsplit(colnames(sp500env$LOW), split="[.]"))[2, ]
> # Extract proper names from column names
> namev <- rutils::get_name(colnames(sp500env$LOW), field=2)
> # Or
> # namev <- do.call(rbind, strsplit(colnames(sp500env$LOW),
> #                                    split="[.]"))[, 2]
> # Rename "LOW" colnames to "LOWES"
> colnames(sp500env$LOW) <- paste("LOVES", namev, sep=".")
> sp500env$LOWES <- sp500env$LOW
> rm(LOW, envir=sp500env)
> # Rename BF-B colnames to "BFB"
> colnames(sp500env$"BF-B") <- paste("BFB", namev, sep=".")
> sp500env$BFB <- sp500env$"BF-B"
> rm("BF-B", envir=sp500env)
> # Rename BRK-B colnames
> sp500env$BRKB <- sp500env$`BRK-B`
> rm(`BRK-B`, envir=sp500env)
> colnames(sp500env$BRKB) <- gsub("BRK-B", "BRKB", colnames(sp500e
> # Save OHLC prices to .RData file
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> # Download "BRK.B" separately with auto.assign=FALSE
> # BRKB <- quantmod::getSymbols("BRK-B", auto.assign=FALSE, src="tiingo", adjust=TRUE, from="1990-01-01", api.key="j84ac2b9c5bde2d68e3
> # colnames(BRKB) <- paste("BRKB", namev, sep=".")
> # sp500env$BRKB <- BRKB
```



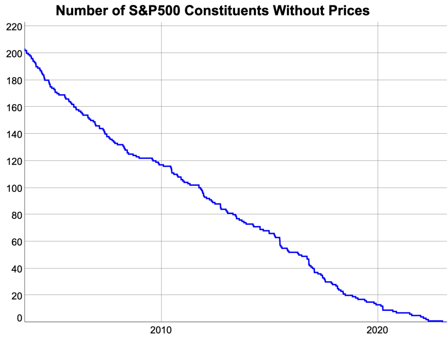

```
> # Plot OHLC candlestick chart for LOWES
> chart_Series(x=sp500env$LOWES["2019-12/"],
+   TA="add_Vo()", name="LOWES OHLC Stock Prices")
> # Plot dygraph
> dygraphs::dygraph(sp500env$LOWES["2019-12/", -5], main="LOWES OHL
+   dyCandlestick()
```

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")
> pricev <- eapply(sp500env, quantmod::Cl)
> pricev <- rutils::do_call(cbind, pricev)
> # Carry forward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> # Drop ".Close" from column names
> colnames(pricev)
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> # Calculate percentage returns of the S&P500 constituent stocks
> # retp <- xts::diff.xts(log(pricev))
> retp <- xts::diff.xts(pricev)/
+   rutils::lagit(pricev, pad_zeros=FALSE)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> samplev <- sample(NCOL(retp), s=100, replace=FALSE)
> prices100 <- pricev[, samplev]
> returns100 <- retp[, samplev]
> save(pricev, prices100,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_prices.RData")
> save(retp, returns100,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.RData")
```

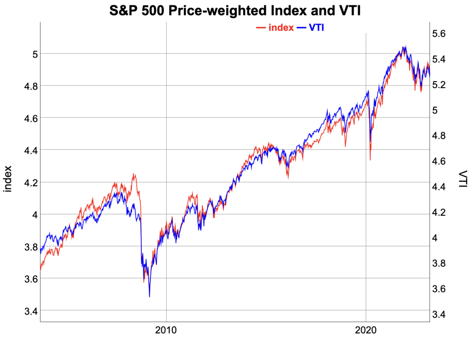



```
> # Calculate number of constituents without prices
> datav <- rowSums(is.na(pricev))
> datav <- xts::xts(datav, order.by=zoo::index(pricev))
> dygraphs::dygraph(datav, main="Number of S&P500 Constituents Witho
+   dyOptions(colors="blue", strokeWidth=2)
```

S&P500 Stock Portfolio Index

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

```
> # Calculate price weighted index of constituent
> ncols <- NCOL(pricev)
> pricev <- zoo::na.locf(pricev, fromLast=TRUE)
> indeks <- xts(rowSums(pricev)/ncols, zoo::index(pricev))
> colnames(indeks) <- "index"
> # Combine index with VTI
> datav <- cbind(indeks[zoo::index(etfenv$VTI)], etfenv$VTI[, 4])
> colv <- c("index", "VTI")
> colnames(datav) <- colv
> # Plot index with VTI
> endd <- rutils::calc_endpoints(datav, interval="weeks")
> dygraphs::dygraph(log(datav)[endd],
+   main="S&P 500 Price-weighted Index and VTI") %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", col="red") %>%
+   dySeries(name=colv[2], axis="y2", col="blue")
```



Writing Time Series To Files

The data from *Tiingo* is downloaded as xts time series, with a date-time index of class POSIXct (not Date).

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

The easiest way to share data between R and Excel is through .csv files.

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

The function data.table::fread() reads from .csv files over 6 times faster than the function read.csv()!

The function data.table::fwrite() writes to .csv files over 12 times faster than the function write.csv(), and 278 times faster than function cat()!

```
> # Save the environment to compressed .RData file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> save(sp500env, file=paste0(dirn, "sp500.RData"))
> # Save the ETF prices into CSV files
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/SP500/"
> for (symboln in ls(sp500env)) {
+   zoo::write.zoo(sp500env$symbol, file=paste0(dirn, symboln, ".csv
+ }  # end for
> # Or using lapply()
> filens <- lapply(ls(sp500env), function(symboln) {
+   xtsv <- get(symboln, envir=sp500env)
+   zoo::write.zoo(xtsv, file=paste0(dirn, symboln, ".csv"))
+   symboln
+ })  # end lapply
> unlist(filens)
> # Or using eapply() and data.table::fwrite()
> filens <- eapply(sp500env , function(xtsv) {
+   filen <- rutils::get_name(colnames(xtsv)[1])
+   data.table::fwrite(data.table::as.data.table(xtsv), file=paste0(
+   filen
+ })  # end eapply
> unlist(filens)
```

Reading Time Series from Files

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

The function `Sys.glob()` listv files matching names obtained from wildcard expansion.

The easiest way to share data between `R` and `Excel` is through `.csv` files.

The function `as.Date()` parses `character` strings, and coerces `numeric` and `POSIXct` objects into `Date` objects.

The function `data.table::setDF()` coerces a *data table* object into a *data frame* using a *side effect*, without making copies of data.

The function `data.table::fread()` reads from `.csv` files over 6 times faster than the function `read.csv()`!

```
> # Load the environment from compressed .RData file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> load(file=paste0(dirn, "sp500.RData"))
> # Get all the .csv file names in the directory
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/SP500/"
> filens <- Sys.glob(paste0(dirn, "*.csv"))
> # Create new environment for data
> sp500env <- new.env()
> for (filen in filens) {
+   xtsv <- xts::as.xts(zoo::read.csv.zoo(filen))
+   symboln <- rutils::get_name(colnames(xtsv)[1])
+   # symboln <- strsplit(colnames(xtsv), split="[.]")[[1]][1]
+   assign(symboln, xtsv, envir=sp500env)
+ }  # end for
> # Or using fread()
> for (filen in filens) {
+   xtsv <- data.table::fread(filen)
+   data.table::setDF(xtsv)
+   xtsv <- xts::xts(xtsv[, -1], as.Date(xtsv[, 1]))
+   symboln <- rutils::get_name(colnames(xtsv)[1])
+   assign(symboln, xtsv, envir=sp500env)
+ }  # end for
```

Downloading Stock Time Series From *Alpha Vantage*

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Tiingo*, *Alpha Vantage*, and Quandl as the only major providers of free daily *OHLC* stock prices.

But Quandl doesn't provide free *ETF* prices, while *Alpha Vantage* does.

The function `getSymbols()` has a *method* for downloading time series data from *Alpha Vantage*, called `getSymbols.av()`.

Users must first obtain an *Alpha Vantage API key*, and then pass it in `getSymbols.av()` calls: https://www.alphavantage.co/

The function `adjustOHLC()` with argument `use.Adjusted=TRUE`, adjusts all the *OHLC* price columns, using the *Adjusted* price column.

```
> # Remove all files from environment(if necessary)
> rm(list=ls(sp500env), envir=sp500env)
> # Download in while loop from Alpha Vantage and copy into environm
> isdown <- symbolv %in% ls(sp500env)
> nattempts <- 0
> while ((sum(!isdown) > 0) & (nattempts < 10)) {
+   # Download data and copy it into environment
+   nattempts <- nattempts + 1
+   for (symboln in symbolv[!isdown]) {
+     cat("processing: ", symboln, "\n")
+     tryCatch(  # With error handler
+ quantmod::getSymbols(symboln, src="av", adjust=TRUE, auto.assign=T
+            output.size="full", api.key="T7JPW54ES8G75310"),
+ # error handler captures error condition
+ error=function(msg) {
+   print(paste0("Error handler: ", msg))
+ },  # end error handler
+ finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdown <- symbolv %in% ls(sp500env)
+   Sys.sleep(2)  # Wait 2 seconds until next attempt
+ }  # end while
> # Adjust all OHLC prices in environment
> for (symboln in ls(sp500env)) {
+   assign(symboln,
+     adjustOHLC(get(x=symboln, envir=sp500env), use.Adjusted=TRUE),
+     envir=sp500env)
+ }  # end for
```

Downloading The *S&P500* Index Time Series From *Yahoo*

The *S&P500* stock market index is a capitalization-weighted average of the 500 largest U.S. companies, and covers about 80% of the U.S. stock market capitalization.

Notice: *Yahoo* no longer provides a public API for data.

There are workarounds but they're tedious.

Yahoo provides daily *OHLC* prices for the *S&P500* index (symbol *^GSPC*), and for the *S&P500* total return index (symbol *^SP500TR*).

But special characters in some stock symbols, like "-" or "^" are not allowed in R names.

For example, the symbol *^GSPC* for the *S&P500* stock market index isn't a valid name in R.

The function `setSymbolLookup()` creates valid names corresponding to stock symbols, which are then used by the function `getSymbols()` to create objects with the valid names.

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Alpha Vantage* and Quandl as the only major providers of free daily *OHLC* stock prices.

```
> # Assign name SP500 to ^GSPC symbol
> quantmod::setSymbolLookup(SP500=list(name="^GSPC", src="yahoo"))
> quantmod::getSymbolLookup()
> # View and clear options
> options("getSymbols.sources")
> options(getSymbols.sources=NULL)
> # Download S&P500 prices into etfenv
> quantmod::getSymbols("SP500", env=etfenv,
+     adjust=TRUE, auto.assign=TRUE, from="1990-01-01")
>
> chart_Series(x=etfenv$SP500["2016/"],
+        TA="add_Vo()", name="S&P500 index")
```

Downloading The *DJIA* Index Time Series From *Yahoo*

The Dow Jones Industrial Average (*DJIA*) stock market index is a price-weighted average of the 30 largest U.S. companies (same number of shares per company).

Yahoo provides daily *OHLC* prices for the *DJIA* index (symbol *^DJI*), and for the *DJITR* total return index (symbol *DJITR*).

But special characters in some stock symbols, like `"-"` or `"^"` are not allowed in `R` names.

For example, the symbol *^DJI* for the *DJIA* stock market index isn't a valid name in `R`.

The function `setSymbolLookup()` creates valid names corresponding to stock symbols, which are then used by the function `getSymbols()` to create objects with the valid names.

```
> # Assign name DJIA to ^DJI symbol
> setSymbolLookup(DJIA=list(name="^DJI", src="yahoo"))
> getSymbolLookup()
> # view and clear options
> options("getSymbols.sources")
> options(getSymbols.sources=NULL)
> # Download DJIA prices into etfenv
> quantmod::getSymbols("DJIA", env=etfenv,
+     adjust=TRUE, auto.assign=TRUE, from="1990-01-01")
> chart_Series(x=etfenv$DJIA["2016/"],
+        TA="add_Vo()", name="DJIA index")
```

Calculating Prices and Returns From *OHLC* Data

The function `na.locf()` from package *zoo* replaces `NA` values with the most recent non-`NA` values prior to it.

The function `na.locf()` with argument `fromLast=TRUE` replaces `NA` values with non-`NA` values in reverse order, starting from the end.

The function `rutils::get_name()` extracts symbol names (tickers) from a vector of character strings.

```
> pricev <- eapply(sp500env, quantmod::Cl)
> pricev <- rutils::do_call(cbind, pricev)
> # Carry forward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> # Get first column name
> colnames(pricev[, 1])
> rutils::get_name(colnames(pricev[, 1]))
> # Modify column names
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> # Calculate percentage returns
> retp <- xts::diff.xts(pricev)/
+   rutils::lagit(pricev, pad_zeros=FALSE)
> # Select a random sample of 100 prices and returns
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> samplev <- sample(NCOL(retp), s=100, replace=FALSE)
> prices100 <- pricev[, samplev]
> returns100 <- retp[, samplev]
> # Save the data into binary files
> save(pricev, prices100,
+      file="/Users/jerzy/Develop/lecture_slides/data/sp500_prices.R
> save(retp, returns100,
+      file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.
```

Downloading Stock Prices From Polygon

Polygon is a premium provider of live and historical stock price data, both daily and intraday (minutes).

Polygon provides 2 years of daily historical stock prices for free. But users must first obtain a Polygon *API key*.

Polygon provides the historical *OHLC* stock prices in *JSON* format.

JSON (JavaScript Object Notation) is a data format consisting of symbol-value pairs.

The package *jsonlite* contains functions for managing data in *JSON* format.

The functions `fromJSON()` and `toJSON()` convert data from *JSON* format to `R` objects, and vice versa.

The functions `read_json()` and `write_json()` read and write *JSON* format data in files.

The function `download.file()` downloads data from an internet website `URL` and writes it to a file.

```
> # Setup code
> symboln <- "SPY"
> startd <- as.Date("1990-01-01")
> todayd <- Sys.Date()
> tspan <- "day"
> # Replace below your own Polygon API key
> apikey <- "SEpnsBpiRyONMJdl48r6dOo0_pjmCu5r"
> # Create url for download
> urll <- paste0("https://api.polygon.io/v2/aggs/ticker/", symboln,
> # Download SPY OHLC prices in JSON format from Polygon
> ohlc <- jsonlite::read_json(urll)
> class(ohlc)
> NROW(ohlc)
> names(ohlc)
> # Extract list of prices from json object
> ohlc <- ohlc$results
> # Coerce from list to matrix
> ohlc <- lapply(ohlc, unlist)
> ohlc <- do.call(rbind, ohlc)
> # Coerce time from milliseconds to dates
> datev <- ohlc[, "t"]/1e3
> datev <- as.POSIXct(datev, origin="1970-01-01")
> datev <- as.Date(datev)
> tail(datev)
> # Coerce from matrix to xts
> ohlc <- ohlc[, c("o","h","l","c","v","vw")]
> colnames(ohlc) <- c("Open", "High", "Low", "Close", "Volume", "VWA
> ohlc <- xts::xts(ohlc, order.by=datev)
> tail(ohlc)
> # Save the xts time series to compressed RData file
> save(ohlc, file="/Users/jerzy/Data/spy_daily.RData")
> # Candlestick plot of SPY OHLC prices
> dygraphs::dygraph(ohlc[, 1:4], main=paste("Candlestick Plot of", s
+   dygraphs::dyCandlestick()
```

Downloading Multiple Stock Prices From Polygon

The stock prices for multiple stocks can be downloaded in a `while()` loop.

```
> # Select ETF symbols for asset allocation
> symbolv <- c("SPY", "VTI", "QQQ", "VEU", "EEM", "XL
+ "XLE", "XLF", "XLV", "XLI", "XLB", "XLK", "XLU", "V
+ "IWB", "IWD", "IWF", "IEF", "TLT", "VNQ", "DBC", "G
+ "VXX", "SVXY", "MTUM", "IVE", "VLUE", "QUAL", "VTV"
> # Setup code
> etfenv <- new.env()  # New environment for data
> # Boolean vector of symbols already downloaded
> isdown <- symbolv %in% ls(etfenv)
```

```
> # Download data from Polygon using while loop
> while (sum(!isdown) > 0) {
+   for (symboln in symbolv[!isdown]) {
+     cat("Processing:", symboln, "\n")
+     tryCatch({  # With error handler
+ # Download OHLC bars from Polygon into JSON format file
+ urll <- paste0("https://api.polygon.io/v2/aggs/ticker/", symboln, "/range/1/",
+ ohlc <- jsonlite::read_json(urll)
+ # Extract list of prices from json object
+ ohlc <- ohlc$results
+ # Coerce from list to matrix
+ ohlc <- lapply(ohlc, unlist)
+ ohlc <- do.call(rbind, ohlc)
+ # Coerce time from milliseconds to dates
+ datev <- ohlc[, "t"]/1e3
+ datev <- as.POSIXct(datev, origin="1970-01-01")
+ datev <- as.Date(datev)
+ # Coerce from matrix to xts
+ ohlc <- ohlc[, c("o","h","l","c","v","vw")]
+ colnames(ohlc) <- paste0(symboln, ".", c("Open", "High", "Low", "Close", "Volum
+ ohlc <- xts::xts(ohlc, order.by=datev)
+ # Save to environment
+ assign(symboln, ohlc, envir=etfenv)
+ Sys.sleep(1)
+ },
+     error={function(msg) print(paste0("Error handler: ", msg))},
+     finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdown <- symbolv %in% ls(etfenv)
+ }  # end while
> save(etfenv, file="/Users/jerzy/Develop/lecture_slides/data/etf_data.RData")
```

Calculating the Stock Alphas, Betas, and Other Performance Statistics

The package *PerformanceAnalytics* contains functions for calculating risk and performance statistics, such as the *variance*, *skewness*, *kurtosis*, *beta*, *alpha*, etc.

The function PerformanceAnalytics::table.CAPM() calculates the *beta* β and *alpha* α values, the *Treynor* ratio, and other performance statistics.

The function PerformanceAnalytics::table.Stats() calculates a data frame of risk and return statistics of the return distributions.

```
> prices <- eapply(etfenv, quantmod::Cl)
> prices <- do.call(cbind, prices)
> # Drop ".Close" from colnames
> colnames(prices) <- do.call(rbind, strsplit(colnames(prices), spli
> # Calculate the log returns
> retp <- xts::diff.xts(log(prices))
> # Copy prices and returns into etfenv
> etfenv$prices <- prices
> etfenv$retp <- retp
> # Copy symbolv into etfenv
> etfenv$symbolv <- symbolv
> # Calculate the risk-return statistics
> riskstats <- PerformanceAnalytics::table.Stats(retp)
> # Transpose the data frame
> riskstats <- as.data.frame(t(riskstats))
> # Add Name column
> riskstats$Name <- rownames(riskstats)
> # Copy riskstats into etfenv
> etfenv$riskstats <- riskstats
> # Calculate the beta, alpha, Treynor ratio, and other performance
> capmstats <- PerformanceAnalytics::table.CAPM(Ra=retp[, symbolv],
+                                          Rb=retp[, "VTI"], scale=2
> colv <- strsplit(colnames(capmstats), split=" ")
> colv <- do.call(cbind, colv)[1, ]
> colnames(capmstats) <- colv
> capmstats <- t(capmstats)
> capmstats <- capmstats[, -1]
> colv <- colnames(capmstats)
> whichv <- match(c("Annualized Alpha", "Information Ratio", "Treyno
> colv[whichv] <- c("Alpha", "Information", "Treynor")
> colnames(capmstats) <- colv
> capmstats <- capmstats[order(capmstats[, "Alpha"], decreasing=TRUE
> # Copy capmstats into etfenv
> etfenv$capmstats <- capmstats
> save(etfenv, file="/Users/jerzy/Develop/lecture_slides/data/etf_da
```

Scraping *S&P500* Stock Index Constituents From Websites

The *S&P500* index constituents change over time, and *Standard & Poor's* replaces companies that have decreased in capitalization with ones that have increased.

The *S&P500* index may contain more than 500 stocks because some companies have several share classes of stock.

The *S&P500* index constituents may be scraped from websites like Wikipedia, using dedicated packages.

The function `getURL()` from package *RCurl* downloads the *html* text data from an internet website URL.

The function `readHTMLTable()` from package *XML* extracts tables from *html* text data or from a remote URL, and returns them as a list of *data frames* or matrices.

`readHTMLTable()` can't parse secure URLs, so they must first be downloaded using function `getURL()`, and then parsed using `readHTMLTable()`.

```
> library(RCurl)  # Load package RCurl
> library(XML)  # Load package XML
> # Download text data from URL
> sp500 <- getURL(
+   "https://en.wikipedia.org/wiki/List_of_S%26P500_companies")
> # Extract tables from the text data
> sp500 <- readHTMLTable(sp500)
> str(sp500)
> # Extract colnames of data frames
> lapply(sp500, colnames)
> # Extract S&P500 constituents
> sp500 <- sp500[[1]]
> head(sp500)
> # Create valid R names from symbols containing "-" or "."characte
> sp500$namev <- gsub("-", "_", sp500$Ticker)
> sp500$namev <- gsub("[.]", "_", sp500$names)
> # Write data frame of S&P500 constituents to CSV file
> write.csv(sp500,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_Yahoo.csv",
+   row.names=FALSE)
```

Downloading *S&P500* Time Series Data From *Yahoo*

Before time series data for the *S&P500* index constituents can be downloaded from *Yahoo*, it's necessary to create valid names corresponding to symbols containing special characters like "-".

The function `setSymbolLookup()` creates a lookup table for *Yahoo* symbols, using valid names in R.

For example *Yahoo* uses the symbol "BRK-B", which isn't a valid name in R, but can be mapped to "BRK_B", using the function `setSymbolLookup()`.

```
> library(rutils)  # Load package rutils
> # Load data frame of S&P500 constituents from CSV file
> sp500 <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/s
> # Register symbols corresponding to R names
> for (indeks in 1:NROW(sp500)) {
+   cat("processing: ", sp500$Ticker[indeks], "\n")
+   setSymbolLookup(structure(
+     list(list(name=sp500$Ticker[indeks])),
+     names=sp500$names[indeks]))
+ }  # end for
> sp500env <- new.env()  # new environment for data
> # Remove all files (if necessary)
> rm(list=ls(sp500env), envir=sp500env)
> # Download data and copy it into environment
> rutils::get_data(sp500$names,
+    env_out=sp500env, startd="1990-01-01")
> # Or download in loop
> for (symboln in sp500$names) {
+   cat("processing: ", symboln, "\n")
+   rutils::get_data(symboln,
+    env_out=sp500env, startd="1990-01-01")
+ }  # end for
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp50
> chart_Series(x=sp500env$BRKB["2016/"],
+        TA="add_Vo()", name="BRK-B stock")
```

Downloading *FRED* Time Series Data

FRED is a database of economic time series maintained by the Federal Reserve Bank of St. Louis:
http://research.stlouisfed.org/fred2/

The function `getSymbols()` downloads time series data into the specified *environment*.

`getSymbols()` can download *FRED* data with the argument `"src"` set to `FRED`.

If the argument `"auto.assign"` is set to `FALSE`, then `getSymbols()` returns the data, instead of assigning it silently.


```
> # Download U.S. unemployment rate data
> unrate <- quantmod::getSymbols("UNRATE",
+    auto.assign=FALSE, src="FRED")
> # Plot U.S. unemployment rate data
> dygraphs::dygraph(unrate["1990/"], main="U.S. Unemployment Rate")
+   dyOptions(colors="blue", strokeWidth=2)
> # Or
> quantmod::chart_Series(unrate["1990/"], name="U.S. Unemployment Ra
```

The *Quandl* Database

Quandl is a distributor of third party data, and offers several million financial, economic, and social datasets.

Much of the Quandl data is free, while premium data can be obtained under a temporary license.

Quandl provides online help and a guide to its datasets:
https://www.quandl.com/help/r
https://www.quandl.com/browse
https://www.quandl.com/blog/getting-started-with-the-quandl-api
https://www.quandl.com/blog/stock-market-data-guide

Quandl provides stock prices, stock fundamentals, financial ratios, zoo::indexes, options and volatility, earnings estimates, analyst ratings, etc.:
https://www.quandl.com/blog/api-for-stock-data

```
> install.packages("devtools")
> library(devtools)
> # Install package Quandl from github
> install_github("quandl/R-package")
> library(Quandl)  # Load package Quandl
> # Register Quandl API key
> Quandl.api_key("pVJi9Nv3V8CD3Js5s7Qx")
> # Get short description
> packageDescription("Quandl")
> # Load help page
> help(package="Quandl")
> # Remove Quandl from search path
> detach("package:Quandl")
```

Quandl has developed an R package called *Quandl* that allows downloading data from Quandl directly into R.

To make more than 50 downloads a day, you need to register your *Quandl API key* using the function `Quandl.api_key()`,

Downloading Time Series Data from *Quandl*

Quandl data can be downloaded directly into R using the function Quandl().

The dots "..." argument of the Quandl() function accepts additional parameters to the *Quandl API*,

Quandl datasets have a unique *Quandl code* in the format "database/ticker", which can be found on the Quandl website for that dataset:
https://www.quandl.com/data/WIKI?keyword=aapl

WIKI is a user maintained free database of daily prices for 3,000 U.S. stocks,
https://www.quandl.com/data/WIKI

SEC is a free database of stock fundamentals extracted from *SEC 10Q* and *10K* filings (but not harmonized),
https://www.quandl.com/data/SEC

RAYMOND is a free database of harmonized stock fundamentals, based on the *SEC* database,
https://www.quandl.com/data/RAYMOND-Raymond https://www.quandl.com/data/RAYMOND-Raymond?keyword=aapl

```
> library(rutils)  # Load package rutils
> # Download EOD AAPL prices from WIKI free database
> pricev <- Quandl(code="WIKI/AAPL",
+   type="xts", startd="1990-01-01")
> x11(width=14, height=7)
> chart_Series(pricev["2016", 1:4], name="AAPL OHLC prices")
> # Add trade volume in extra panel
> add_TA(pricev["2016", 5])
> # Download euro currency rates
> pricev <- Quandl(code="BNP/USDEUR",
+     startd="2013-01-01",
+     endd="2013-12-01", type="xts")
> # Download multiple time series
> pricev <- Quandl(code=c("NSE/OIL", "WIKI/AAPL"),
+     startd="2013-01-01", type="xts")
> # Download AAPL gross profits
> prof_it <- Quandl("RAYMOND/AAPL_GROSS_PROFIT_Q", type="xts")
> chart_Series(prof_it, name="AAPL gross profits")
> # Download Hurst time series
> pricev <- Quandl(code="PE/AAPL_HURST",
+     startd="2013-01-01", type="xts")
> chart_Series(pricev["2016/", 1], name="AAPL Hurst")
```

Stock Index and Instrument Metadata on *Quandl*

Instrument metadata specifies properties of instruments, like its currency, contract size, tick value, delievery months, start date, etc.

Quandl provides instrument metadata for stock indices, futures, and currencies:

https://www.quandl.com/blog/useful-listv

Quandl also provides constituents for stock indices, for example the *S&P500*, *Dow Jones Industrial Average*, *NASDAQ Composite*, *FTSE 100*, etc.

```
> # Load S&P500 stock Quandl codes
> sp500 <- read.csv(
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_quandl.csv'
> # Replace "-" with "_" in symbols
> sp500$free_code <- gsub("-", "_", sp500$free_code)
> head(sp500)
> # vector of symbols in sp500 frame
> tickers <- gsub("-", "_", sp500$ticker)
> # Or
> tickers <- matrix(unlist(
+   strsplit(sp500$free_code, split="/"),
+   use.names=FALSE), ncol=2, byrow=TRUE)[, 2]
> # Or
> tickers <- do_call_rbind(
+   strsplit(sp500$free_code, split="/"))[, 2]
```

Downloading Multiple Time Series from *Quandl*

Time series data for a portfolio of stocks can be downloaded by performing a loop over the function `Quandl()` from package Quandl.

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

```
> sp500env <- new.env()  # new environment for data
> # Remove all files (if necessary)
> rm(list=ls(sp500env), envir=sp500env)
> # Boolean vector of symbols already downloaded
> isdown <- tickers %in% ls(sp500env)
> # Download data and copy it into environment
> for (ticker in tickers[!isdown]) {
+   cat("processing: ", ticker, "\n")
+   datav <- Quandl(code=paste0("WIKI/", ticker),
+             startd="1990-01-01", type="xts")[, -(1:7)]
+   colnames(datav) <- paste(ticker,
+     c("Open", "High", "Low", "Close", "Volume"), sep=".")
+   assign(ticker, datav, envir=sp500env)
+ }  # end for
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp5
> chart_Series(x=sp500env$XOM["2016/"], TA="add_Vo()", name="XOM sto
```

Downloading Futures Time Series from *Quandl*

Quandl provides the Wiki CHRIS Database of time series of prices for 600 different futures contracts.

The Wiki CHRIS Database contains daily *OHLC* prices for continuous futures contracts.

A continuous futures contract is a time series of prices obtained by chaining together prices from consecutive futures contracts.

The data is curated by the Quandl community from data provided by the *CME*, *ICE*, *LIFFE*, and other exchanges.

The *Quandl codes* are specified as `CHRIS/{EXCHANGE}_{CODE}{DEPTH}`, where `{DEPTH}` is the depth of the chained contract.

The chained front month contracts have depth 1, the back month contracts have depth 2, etc.

The continuous front and back month contracts allow building continuous futures curves.

Quandl data can be downloaded directly into `R` using the function `Quandl()`.

```
> library(Quandl)
> # Register Quandl API key
> Quandl.api_key("pVJi9Nv3V8CD3Js5s7Qx")
> # Download E-mini S&P500 futures prices
> pricev <- Quandl(code="CHRIS/CME_ES1",
+   type="xts", startd="1990-01-01")
> pricev <- pricev[, c("Open", "High", "Low", "Last", "Volume")]
> colnames(pricev)[4] <- "Close"
> # Plot the prices
> x11(width=5, height=4)  # Open x11 for plotting
> chart_Series(x=pricev["2008-06/2009-06"],
+        TA="add_Vo()", name="S&P500 Futures")
> # Plot dygraph
> dygraphs::dygraph(pricev["2008-06/2009-06", -5],
+   main="S&P500 Futures") %>%
+   dyCandlestick()
```

For example, the *Quandl code* for the continuous *E-mini S&P500* front month futures is `CHRIS/CME_ES1`, while for the back month it's `CHRIS/CME_ES2`, for the second back month it's `CHRIS/CME_ES3`, etc.

The *Quandl code* for the *E-mini Oil* futures is `CHRIS/CME_QM1`, for the *E-mini euro FX* futures is `CHRIS/CME_E71`, etc.

Downloading *VIX* Futures Files from CBOE

The CFE (CBOE Futures Exchange) provides daily CBOE Historical Data for Volatility Futures, including the *VIX* futures.

The CBOE data incudes *OHLC* prices and also the *settlement* price (in column "Settle").

The *settlement* price is usually defined as the weighted average price (*WAP*) or the midpoint price, and is different from the *Close* price.

The *settlement* price is used for calculating the daily *mark to market* (value) of the futures contract.

Futures exchanges require that counterparties exchange (settle) the *mark to market* value of the futures contract daily, to reduce counterparty default risk.

The function `download.file()` downloads files from the internet.

The function `tryCatch()` executes functions and expressions, and handles any *exception conditions* produced when they are evaluated.

```
> # Read CBOE futures expiration dates
> datev <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/f
+   row.names=1)
> dirn <- "/Users/jerzy/Develop/data/vix_data"
> dir.create(dirn)
> symbolv <- rownames(datev)
> filens <- file.path(dirn, paste0(symbolv, ".csv"))
> log_file <- file.path(dirn, "log_file.txt")
> cboe_url <- "https://markets.cboe.com/us/futures/market_statistics
> urls <- paste0(cboe_url, datev[, 1])
> # Download files in loop
> for (it in seq_along(urls)) {
+     tryCatch(  # Warning and error handler
+   download.file(urls[it],
+           destfile=filens[it], quiet=TRUE),
+ # Warning handler captures warning condition
+ warning=function(msg) {
+   cat(paste0("Warning handler: ", msg, "\n"), file=log_file, apper
+ },  # end warning handler
+ # Error handler captures error condition
+ error=function(msg) {
+   cat(paste0("Error handler: ", msg, "\n"), append=TRUE)
+ },  # end error handler
+ finally=cat(paste0("Processing file name = ", filens[it], "\n"), a
+     )  # end tryCatch
+ }  # end for
```

Downloading *VIX* Futures Data Into an Environment

The function `quantmod::getSymbols()` with the parameter `src="cfe"` downloads CFE data into the specified *environment*. (But this requires first loading the package *qmao*.)

Currently `quantmod::getSymbols()` doesn't download the most recent data.

```
> # Create new environment for data
> vix_env <- new.env()
> # Download VIX data for the months 6, 7, and 8 in 2018
> library(qmao)
> quantmod::getSymbols("VX", Months=1:12,
+   Years=2018, src="cfe", auto.assign=TRUE, env=vix_env)
> # Or
> qmao::getSymbols.cfe(Symbols="VX",
+   Months=6:8, Years=2018, env=vix_env,
+   verbose=FALSE, auto.assign=TRUE)
> # Calculate the classes of all the objects
> # In the environment vix_env
> unlist(eapply(vix_env, function(x) {class(x)[1]}))
> class(vix_env$VX_M18)
> colnames(vix_env$VX_M18)
> # Save the data to a binary file called "vix_cboe.RData".
> save(vix_env,
+   file="/Users/jerzy/Develop/data/vix_data/vix_cboe.RData")
```

Kernel Density of Asset Returns

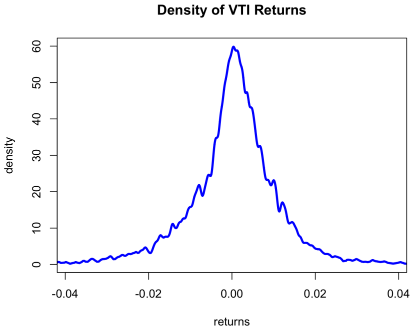
The kernel density is proportional to the number of data points close to a given point.

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_r at a point r is equal to the sum over the kernel function $K(x)$:

$$d_r = \sum_{j=1}^{n} K(r - r_j)$$




```
> library(rutils)  # Load package rutils
> # Calculate VTI percentage returns
> retp <- rutils::etfenv$returns$VTI
> retp <- drop(coredata(na.omit(retp)))
> nrows <- NROW(retp)
> # Mean and standard deviation of returns
> c(mean(retp), sd(retp))
> # Calculate the smoothing bandwidth as the MAD of returns 10 poi
> retp <- sort(retp)
> bwidth <- 10*mad(rutils::diffit(retp, lagg=10))
> # Calculate the kernel density using a loop
> dens1 <- sapply(1:nrows, function(it) {
+   sum(dnorm(retp-retp[it], sd=bwidth))
+ })/nrows  # end sapply
```

```
> # Plot the kernel density
> madv <- mad(retp)
> plot(retp, dens1, xlim=c(-5*madv, 5*madv),
+      t="l", col="blue", lwd=3,
+      xlab="returns", ylab="density",
+      main="Density of VTI Returns")
```

Kernel Density Using the Function density()

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

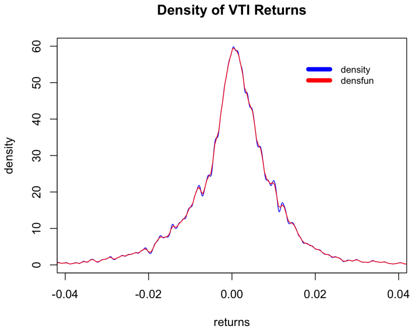
The parameter *smoothing bandwidth* is the standard deviation of the smoothing kernel $K(x)$.

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.

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

```
> # Calculate the kernel density using density()
> densv <- density(retp, bw=bwidth)
> NROW(densv$y)
> plot(densv, xlim=c(-5*madv, 5*madv),
+      xlab="returns", ylab="density",
+      col="blue", lwd=3, main="Density of VTI Returns")
> # Interpolate the densv vector into returns
> densv <- approx(densv$x, densv$y, xout=retp)
> all.equal(densv$x, retp)
```




```
> # Plot the two density estimates
> plot(retp, dens1, xlim=c(-5*madv, 5*madv),
+      xlab="returns", ylab="density",
+      t="l", col="blue", lwd=1,
+      main="Density of VTI Returns")
> lines(retp, densv$y, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("density", "densfun"), bty="n", y.intersp=0.4,
+  lwd=6, bg="white", col=c("blue", "red"))
```

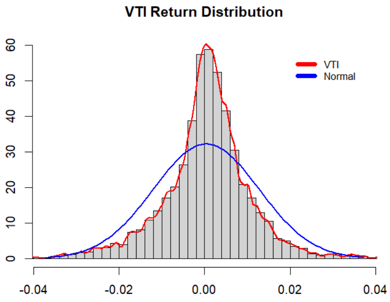
Distribution of Asset Returns

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
> histp <- hist(retp, breaks=100, freq=FALSE,
+   xlim=c(-5*madv, 5*madv), xlab="", ylab="",
+   main="VTI Return Distribution")
> # Draw kernel density of histogram
> lines(densv, col="red", lwd=2)
> # Add density of normal distribution
> curve(expr=dnorm(x, mean=mean(retp), sd=sd(retp)),
+ add=TRUE, lwd=2, col="blue")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("VTI", "Normal"), bty="n", y.intersp=0.4,
+  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.

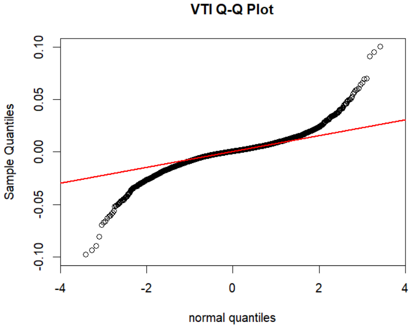
The *p*-value of the *Shapiro-Wilk* test is very close to zero, which shows that the *VTI* returns are very unlikely to be normal.

The function shapiro.test() performs the *Shapiro-Wilk* test of normality.

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



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*).

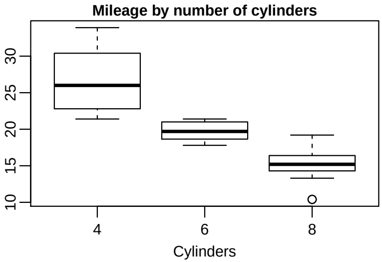
The nominal range is equal to 1.5 times the *IQR* above and below the box *hinges*.

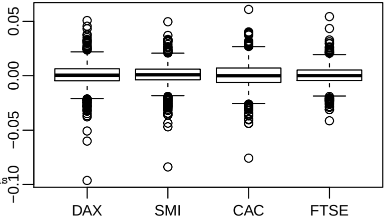
The *whiskers* are dashed vertical lines representing values beyond the first and third quartiles, but within the nominal range.

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):

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

The sample kurtosis (fourth moment):

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

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.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> # Number of observations
> nrows <- NROW(retp)
> # Mean of VTI returns
> retm <- mean(retp)
> # Standard deviation of VTI returns
> stdev <- sd(retp)
> # Skewness of VTI returns
> nrows/((nrows-1)*(nrows-2))*sum(((retp - retm)/stdev)^3)
> # Kurtosis of VTI returns
> nrows*(nrows+1)/((nrows-1)^3)*sum(((retp - retm)/stdev)^4)
> # Random normal returns
> retp <- rnorm(nrows, sd=stdev)
> # Mean and standard deviation of random normal returns
> retm <- mean(retp)
> stdev <- sd(retp)
> # Skewness of random normal returns
> nrows/((nrows-1)*(nrows-2))*sum(((retp - retm)/stdev)^3)
> # Kurtosis of random normal returns
> nrows*(nrows+1)/((nrows-1)^3)*sum(((retp - retm)/stdev)^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(retp) {
+   retp <- na.omit(retp)
+   sum(((retp - mean(retp))/sd(retp))^3)/NROW(retp)
+ }  # end calc_skew
> # calc_kurt() calculates kurtosis of returns
> calc_kurt <- function(retp) {
+   retp <- na.omit(retp)
+   sum(((retp - mean(retp))/sd(retp))^4)/NROW(retp)
+ }  # end calc_kurt
> # Calculate skew and kurtosis of VTI returns
> calc_skew(retp)
> calc_kurt(retp)
> # calc_mom() calculates the moments of returns
> calc_mom <- function(retp, moment=3) {
+   retp <- na.omit(retp)
+   sum(((retp - mean(retp))/sd(retp))^moment)/NROW(retp)
+ }  # end calc_mom
> # Calculate skew and kurtosis of VTI returns
> calc_mom(retp, moment=3)
> calc_mom(retp, moment=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).

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Sample from Standard Normal Distribution
> nrows <- 1000
> datav <- rnorm(nrows)
> # Sample mean
> mean(datav)
> # Sample standard deviation
> sd(datav)
> # Standard error of sample mean
> sd(datav)/sqrt(nrows)
```

Normal (Gaussian) Probability Distribution

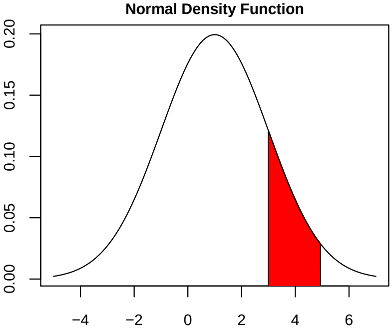
The *Normal (Gaussian)* probability density function is given by:

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

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.

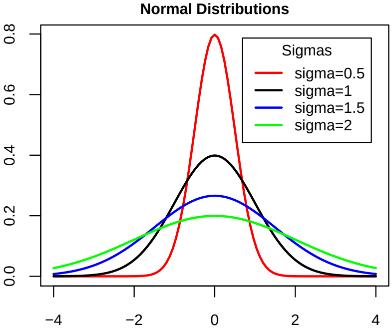
```
> xvar <- seq(-5, 7, length=100)
> yvar <- dnorm(xvar, mean=1.0, sd=2.0)
> plot(xvar, yvar, type="l", lty="solid", xlab="", ylab="")
> title(main="Normal Density Function", line=0.5)
> startp <- 3; endd <- 5  # Set lower and upper bounds
> # Set polygon base
> subv <- ((xvar >= startp) & (xvar <= endd))
> polygon(c(startp, xvar[subv], endd),  # Draw polygon
+   c(-1, yvar[subv], -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.

```
> sigmavs <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> colorv <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("sigma", sigmavs, sep="=")
> for (it in 1:4) {  # Plot four curves
+   curve(expr=dnorm(x, sd=sigmavs[it]),
+   xlim=c(-4, 4), xlab="", ylab="", lwd=2,
+   col=colorv[it], add=as.logical(it-1))
+ }  # end for
> # Add title
> title(main="Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, title="Sigmas", y.intersp=0.4,
+  labelv, cex=0.8, lwd=2, lty=1, bty="n", col=colorv)
```



Student's *t-distribution*

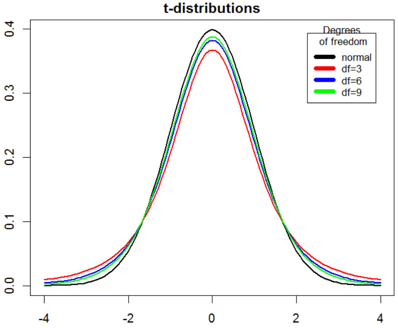
Let $z_1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu} \sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance: $\hat{\sigma}^2 = \frac{1}{\nu - 1} \sum_{i=1}^{\nu} (z_i - \bar{z})^2$

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:

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




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

```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+        title="Degrees\n of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colorv)
```

Mixture Models of Returns

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

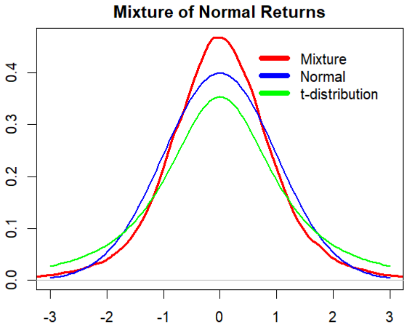
The mixture of two normal distributions with different variances produces a distribution with *leptokurtosis* (large kurtosis, or fat tails).

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
> nrows <- 1e5
> retp <- c(rnorm(nrows/2), 2*rnorm(nrows/2))
> retp <- (retp-mean(retp))/sd(retp)
> # Kurtosis of normal
> calc_kurt(rnorm(nrows))
> # Kurtosis of mixture
> calc_kurt(retp)
> # Or
> nrows*sum(retp^4)/(nrows-1)^2
```

```
> # Plot the distributions
> plot(density(retp), 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"), y.intersp=0.4,
+   col=c("red", "blue", "green"))
```

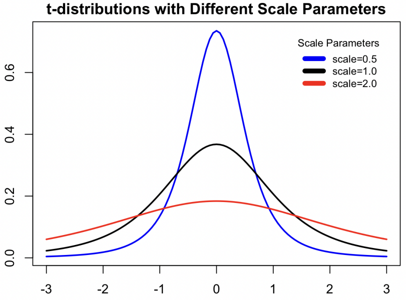
Non-standard Student's *t-distribution*

The non-standard Student's *t-distribution* has the probability density function:

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

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

```
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # x11(width=6, height=5)
> # Define density of non-standard t-distribution
> tdistr <- function(x, dfree, locv=0, scalev=1) {
+   dt((x-locv)/scalev, df=dfree)/scalev
+ }  # end tdistr
> # Or
> tdistr <- function(x, dfree, locv=0, scalev=1) {
+   gamma((dfree+1)/2)/(sqrt(pi*dfree)*gamma(dfree/2)*scalev)*
+     (1+((x-locv)/scalev)^2/dfree)^(-(dfree+1)/2)
+ }  # end tdistr
> # Calculate vector of scale values
> scalev <- c(0.5, 1.0, 2.0)
> colorv <- c("blue", "black", "red")
> labelv <- paste("scale", format(scalev, digits=2), sep="=")
> # Plot three t-distributions
> for (it in 1:3) {
+   curve(expr=tdistr(x, dfree=3, scalev=scalev[it]), xlim=c(-3, 3),
+ xlab="", ylab="", lwd=2, col=colorv[it], add=(it>1))
+ }  # end for
```




```
> # Add title
> title(main="t-distributions with Different Scale Parameters", line
> # Add legend
> legend("topright", inset=0.05, bty="n", title="Scale Parameters",
+         cex=0.8, lwd=6, lty=1, col=colorv, y.intersp=0.4)
```

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.

The *Shapiro-Wilk* test is not reliable for large sample sizes, so it's limited to less than 5000 sample size.

```
> # Calculate VTI percentage returns
> library(rutils)
> retp <- as.numeric(na.omit(rutils::etfenv$returns$VTI))[1:499]
> # Reduce number of output digits
> ndigits <- options(digits=5)
> # Shapiro-Wilk test for normal distribution
> nrows <- NROW(retp)
> shapiro.test(rnorm(nrows))

 Shapiro-Wilk normality test

data:  rnorm(nrows)
W = 0.997, p-value = 0.47
> # Shapiro-Wilk test for VTI returns
> shapiro.test(retp)

 Shapiro-Wilk normality test

data:  retp
W = 0.991, p-value = 0.0029
> # Shapiro-Wilk test for uniform distribution
> shapiro.test(runif(nrows))

 Shapiro-Wilk normality test

data:  runif(nrows)
W = 0.957, p-value = 7.2e-11
> # Restore output digits
> options(digits=ndigits$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 2 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(nrows))

 Jarque Bera Test

data:  rnorm(nrows)
X-squared = 1, df = 2, p-value = 0.5
> # Jarque-Bera test for VTI returns
> jarque.bera.test(retp)

 Jarque Bera Test

data:  retp
X-squared = 22, df = 2, p-value = 2e-05
> # Jarque-Bera test for uniform distribution
> jarque.bera.test(runif(NROW(retp)))

 Jarque Bera Test

data:  runif(NROW(retp))
X-squared = 29, df = 2, p-value = 5e-07
```

The *Kolmogorov-Smirnov* Test for Probability Distributions

The *Kolmogorov-Smirnov* test *null hypothesis* is 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 depends on 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 to `ks.test()` can be 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 <- ks.test(rnorm(100), pnorm)
> ks_test$p.value
> # KS test for uniform distribution
> ks.test(runif(100), pnorm)
> # KS test for two shifted normal distributions
> ks.test(rnorm(100), rnorm(100, mean=0.1))
> ks.test(rnorm(100), rnorm(100, mean=1.0))
> # KS test for two different normal distributions
> ks.test(rnorm(100), rnorm(100, sd=2.0))
> # KS test for VTI returns vs normal distribution
> retp <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> retp <- (retp - mean(retp))/sd(retp)
> ks.test(retp, pnorm)
```

Chi-squared Distribution

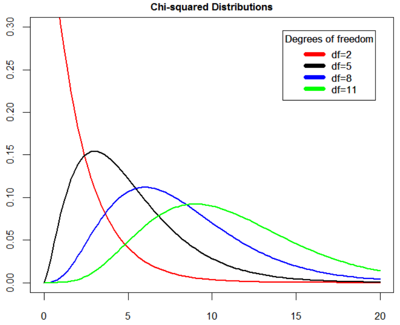
Let $z_1, \ldots, z_k$ be independent standard *Normal* random variables.

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




```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> labelv <- paste("df", degf, sep="=")
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+        title="Degrees of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colorv)
```

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
> histp <- hist(rnorm(1e3, mean=0), breaks=100, plot=FALSE)
> countsn <- histp$counts
> # Theoretical frequencies
> countst <- rutils::diffit(pnorm(histp$breaks))
> # Perform Chi-squared test for normal data
> chisq.test(x=countsn, p=countst, rescale.p=TRUE, simulate.p.value=
> # Return p-value
> chisqtest <- chisq.test(x=countsn, p=countst, rescale.p=TRUE, simu
> chisqtest$p.value
> # Observed frequencies from shifted normal data
> histp <- hist(rnorm(1e3, mean=2), breaks=100, plot=FALSE)
> countsn <- histp$counts/sum(histp$counts)
> # Theoretical frequencies
> countst <- rutils::diffit(pnorm(histp$breaks))
> # Perform Chi-squared test for shifted normal data
> chisq.test(x=countsn, p=countst, rescale.p=TRUE, simulate.p.value=
> # Calculate histogram of VTI returns
> histp <- hist(retp, breaks=100, plot=FALSE)
> countsn <- histp$counts
> # Calculate cumulative probabilities and then difference them
> countst <- pt((histp$breaks-locv)/scalev, df=2)
> countst <- rutils::diffit(countst)
> # Perform Chi-squared test for VTI returns
> chisq.test(x=countsn, p=countst, rescale.p=TRUE, simulate.p.value=
```

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 from function dt()
> likefun <- function(par, dfree, datav) {
+   -sum(log(dt(x=(datav-par[1])/par[2], df=dfree)/par[2]))
+ }  # end likefun
> # Demonstrate equivalence with log(dt())
> likefun(c(1, 0.5), 2, 2:5)
> -sum(log(dt(x=(2:5-1)/0.5, df=2)/0.5))
> # Objective function is negative log-likelihood
> likefun <- function(par, dfree, datav) {
+   sum(-log(gamma((dfree+1)/2)/(sqrt(pi*dfree)*gamma(dfree/2))) +
+     log(par[2]) + (dfree+1)/2*log(1+((datav-par[1])/par[2])^2/dfre
+ }  # end likefun
```

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.

```
> # Calculate VTI percentage returns
> retp <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> # Fit VTI returns using MASS::fitdistr()
> fitobj <- MASS::fitdistr(retp, densfun="t", df=3)
> summary(fitobj)
> # Fitted parameters
> fitobj$estimate
> locv <- fitobj$estimate[1]
> scalev <- fitobj$estimate[2]
> locv; scalev
> # Standard errors of parameters
> fitobj$sd
> # Log-likelihood value
> fitobj$value
> # Fit distribution using optim()
> initp <- c(mean=0, scale=0.01)  # Initial parameters
> fitobj <- optim(par=initp,
+   fn=likefun, # Log-likelihood function
+   datav=retp,
+   dfree=3, # Degrees of freedom
+   method="L-BFGS-B", # Quasi-Newton method
+   upper=c(1, 0.1), # Upper constraint
+   lower=c(-1, 1e-7)) # Lower constraint
> # Optimal parameters
> locv <- fitobj$par["mean"]
> scalev <- fitobj$par["scale"]
> locv; scalev
```

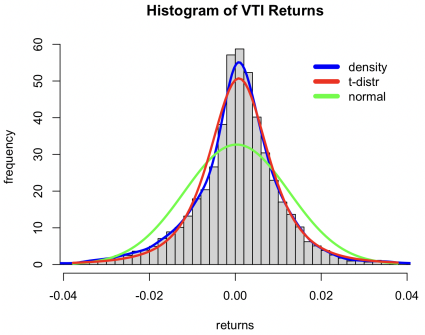
The Student's *t-distribution* Fitted to Asset Returns

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

Stock returns fit the non-standard *t-distribution* with 3 degrees of freedom quite well.

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

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




```
> dev.new(width=6, height=5, noRStudioGD=TRUE)
> # x11(width=6, height=5)
> # Plot histogram of VTI returns
> madv <- mad(retp)
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", breaks=100, xlim=c(-5*madv, 5*madv),
+   ylab="frequency", freq=FALSE, main="Histogram of VTI Returns")
> lines(density(retp, adjust=1.5), lwd=3, col="blue")
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(retp),
+   sd=sd(retp)), add=TRUE, lwd=3, col="green")
> # Define non-standard t-distribution
> tdistr <- function(x, dfree, locv=0, scalev=1) {
+   dt((x-locv)/scalev, df=dfree)/scalev
+ }  # end tdistr
> # Plot t-distribution function
> curve(expr=tdistr(x, dfree=3, locv=locv, scalev=scalev), col="red", lwd=3, add=TRUE)
> # Add legend
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+   leg=c("density", "t-distr", "normal"),
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Goodness of Fit of Student's *t-distribution* Fitted to Asset Returns

The *Q-Q* plot illustrates the relative distributions of two samples of data.

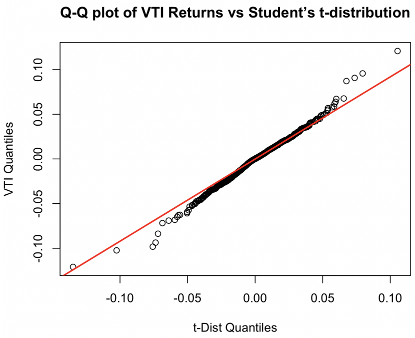
The *Q-Q* plot shows that stock returns fit the non-standard *t-distribution* with 3 degrees of freedom quite well.

The function qqplot() produces a *Q-Q* plot for two samples of data.

The function ks.test() performs the *Kolmogorov-Smirnov* test for the similarity of two distributions.

The *null hypothesis* of the *Kolmogorov-Smirnov* test is that the two samples were obtained from the same probability distribution.

The *Kolmogorov-Smirnov* test rejects the *null hypothesis* that stock returns follow closely the non-standard *t-distribution* with 3 degrees of freedom.




```
> # Calculate sample from non-standard t-distribution with df=3
> datat <- locv + scalev*rt(NROW(retp), df=3)
> # Q-Q plot of VTI Returns vs non-standard t-distribution
> qqplot(datat, retp, xlab="t-Dist Quantiles", ylab="VTI Quantiles"
+        main="Q-Q plot of VTI Returns vs Student's t-distribution"
> # Calculate quartiles of the distributions
> probs <- c(0.25, 0.75)
> qrets <- quantile(retp, probs)
> qtdata <- quantile(datat, probs)
> # Calculate slope and plot line connecting quartiles
> slope <- diff(qrets)/diff(qtdata)
> intercept <- qrets[1]-slope*qtdata[1]
> abline(intercept, slope, lwd=2, col="red")
```

```
> # KS test for VTI returns vs t-distribution data
> ks.test(retp, datat)
> # Define cumulative distribution of non-standard t-distribution
> ptdistr <- function(x, dfree, locv=0, scalev=1) {
+   pt((x-locv)/scalev, df=dfree)
+ }  # end ptdistr
> # KS test for VTI returns vs cumulative t-distribution
> ks.test(sample(retp, replace=TRUE), ptdistr, dfree=3, locv=locv, s
```

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.

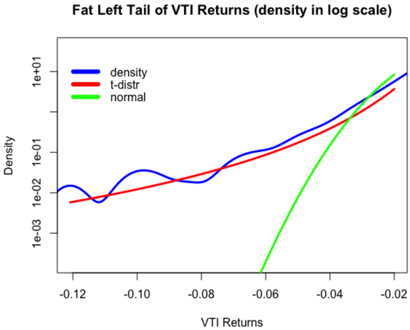
So large values are produced only when all the contributing binomial variables are of the same sign, which is very improbable, so it produces extremely low tail probabilities (thin tails),

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).



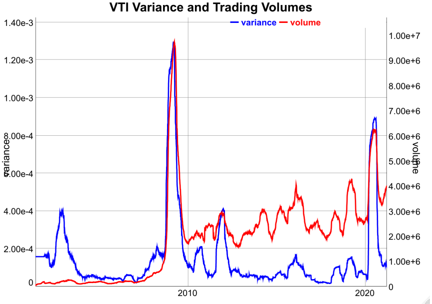

```
> # Plot log density of VTI returns
> plot(density(retp, adjust=4), xlab="VTI Returns", ylab="Density",
+      main="Fat Left Tail of VTI Returns (density in log scale)",
+      type="l", lwd=3, col="blue", xlim=c(min(retp), -0.02), log="y
> # Plot t-distribution function
> curve(expr=dt((x-locv)/scalev, df=3)/scalev, lwd=3, col="red", add
> # Plot the Normal probability distribution
> curve(expr=dnorm(x, mean=mean(retp), sd=sd(retp)), lwd=3, col="gre
> # Add legend
> legend("topleft", inset=0.01, bty="n", y.intersp=c(0.25, 0.25, 0.2
+   legend=c("density", "t-distr", "normal"), y.intersp=0.4,
+   lwd=6, lty=1, col=c("blue", "red", "green"))
```

Trading Volumes

The 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
> ohlc <- rutils::etfenv$VTI
> closep <- drop(coredata(quantmod::Cl(ohlc)))
> retp <- rutils::diffit(log(closep))
> volumv <- coredata(quantmod::Vo(ohlc))
> # Calculate trailing variance
> lookb <- 121
> varv <- HighFreq::roll_var_ohlc(log(ohlc), method="close", lookb=lookb, scale=FALSE)
> varv[1:lookb, ] <- varv[lookb+1, ]
> # Calculate trailing average volume
> volumr <- HighFreq::roll_sum(volumv, lookb=lookb)/lookb
> # dygraph plot of VTI variance and trading volumes
> datav <- xts::xts(cbind(varv, volumr), zoo::index(ohlc))
> colv <- c("variance", "volume")
> colnames(datav) <- colv
> dygraphs::dygraph(datav, main="VTI Variance and Trading Volumes") %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], strokeWidth=2, axis="y", col="blue") %>%
+   dySeries(name=colv[2], strokeWidth=2, axis="y2", col="red") %>%
+   dyLegend(show="always", width=500)
```

Asset Returns in Trading Time

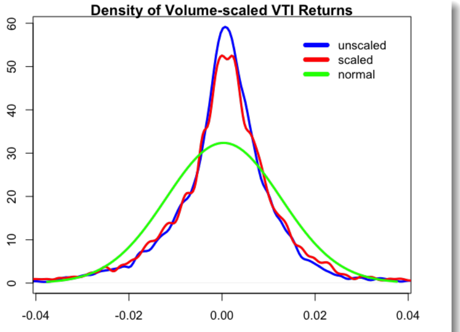
The time-dependent volatility of asset returns (*heteroskedasticity*) produces their fat tails (*leptokurtosis*).

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 the returns using volume clock to trading time
> retsc <- ifelse(volumv > 0, sqrt(volumr)*retp/sqrt(volumv), 0)
> retsc <- sd(retp)*retsc/sd(retsc)
> # retsc <- ifelse(volumv > 1e4, retp/volumv, 0)
> # Calculate moments of scaled returns
> nrows <- NROW(retp)
> sapply(list(retp=retp, retsc=retsc),
+   function(rets) {sapply(c(skew=3, kurt=4),
+      function(x) sum((rets/sd(rets))^x)/nrows)
+ })  # 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
> madv <- mad(retp)
> # bwidth <- mad(rutils::diffit(retp))
> plot(density(retp, bw=madv/10), xlim=c(-5*madv, 5*madv),
+      lwd=3, mgp=c(2, 1, 0), col="blue",
+      xlab="returns (standardized)", ylab="frequency",
+      main="Density of Volume-scaled VTI Returns")
> lines(density(retsc, bw=madv/10), lwd=3, col="red")
> curve(expr=dnorm(x, mean=mean(retp), sd=sd(retp)),
+ add=TRUE, lwd=3, col="green")
> # Add legend
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
```

Package *PerformanceAnalytics* for Risk and Performance Analysis

The package *PerformanceAnalytics* contains functions for calculating risk and performance statistics, such as the variance, skewness, kurtosis, beta, alpha, etc.

The function `data()` loads external data or listv 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.

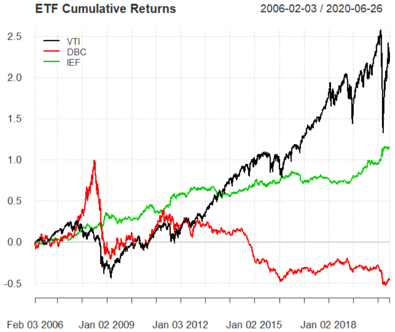
```
> # Load package PerformanceAnalytics
> library(PerformanceAnalytics)
> # Get documentation for package PerformanceAnalytics
> # Get short description
> packageDescription("PerformanceAnalytics")
> # Load help page
> help(package="PerformanceAnalytics")
> # List all objects in PerformanceAnalytics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

```
> perfstats <- unclass(data(
+     package="PerformanceAnalytics"))$results[, -(1:2)]
> apply(perfstats, 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.

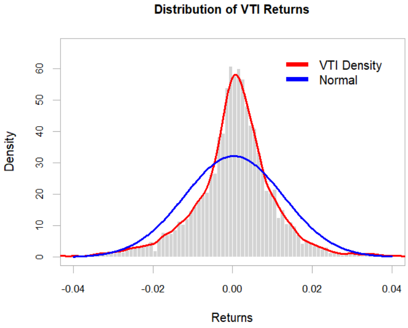
```
> # Load package "PerformanceAnalytics"
> library(PerformanceAnalytics)
> # Calculate ETF returns
> retp <- rutils::etfenv$returns[, c("VTI", "DBC", "IEF")]
> retp <- na.omit(retp)
> # Plot cumulative ETF returns
> x11(width=6, height=5)
> chart.CumReturns(retp, lwd=2, ylab="",
+   legend.loc="topleft",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

```
> retp <- na.omit(rutils::etfenv$returns$VTI)
> chart.Histogram(retp, xlim=c(-0.04, 0.04),
+   colorset = c("lightgray", "red", "blue"), lwd=3,
+   main=paste("Distribution of", colnames(retp), "Returns"),
+   methods = c("add.density", "add.normal"))
> legend("topright", inset=0.05, bty="n", y.intersp=0.4,
+  leg=c("VTI Density", "Normal"),
+  lwd=6, lty=1, col=c("red", "blue"))
```



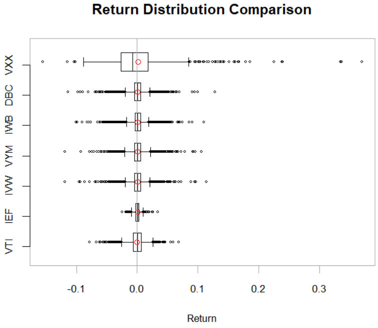
Boxplots of Returns

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).

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



The Median Absolute Deviation Estimator of Dispersion

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

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

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

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

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.

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

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
> targetr <- 0.005
> # Calculate the full downside returns
> retsub <- (retp - targetr)
> retsub <- ifelse(retsub < 0, retsub, 0)
> nrows <- NROW(retsub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(retsub^2)/nrows),
+   drop(DownsideDeviation(retp, MAR=targetr, method="full")))
> # Calculate the subset downside returns
> retsub <- (retp - targetr)
> retsub <- retsub[retsub < 0]
> nrows <- NROW(retsub)
> # Calculate the downside deviation
> all.equal(sqrt(sum(retsub^2)/nrows),
+   drop(DownsideDeviation(retp, MAR=targetr, method="subset")))
```

Drawdown Risk

A *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.

```
> # Calculate time series of VTI drawdowns
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> drawdns <- (closep - cummax(closep))
> # Extract the date index from the time series closep
> datev <- zoo::index(closep)
> # Calculate the maximum drawdown date and depth
> indexmin <- which.min(drawdns)
> datemin <- datev[indexmin]
> maxdd <- drawdns[datemin]
> # Calculate the drawdown start and end dates
> startd <- max(datev[(datev < datemin) & (drawdns == 0)])
> endd <- min(datev[(datev > datemin) & (drawdns == 0)])
> # dygraph plot of VTI drawdowns
> datav <- cbind(closep, drawdns)
> colv <- c("VTI", "Drawdowns")
> colnames(datav) <- colv
> dygraphs::dygraph(datav, main="VTI Drawdowns") %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2],
+    valueRange=(1.2*range(drawdns)+0.1), independentTicks=TRUE) %
+   dySeries(name=colv[1], axis="y", col="blue") %>%
+   dySeries(name=colv[2], axis="y2", col="red") %>%
+   dyEvent(startd, "start drawdown", col="blue") %>%
+   dyEvent(datemin, "max drawdown", col="red") %>%
+   dyEvent(endd, "end drawdown", col="green")
```


```
> # Plot VTI drawdowns using package quantmod
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("blue")
> x11(width=6, height=5)
> quantmod::chart_Series(x=closep, name="VTI Drawdowns", theme=plot_
> xval <- match(startd, datev)
> yval <- max(closep)
> abline(v=xval, col="blue")
> text(x=xval, y=0.95*yval, "start drawdown", col="blue", cex=0.9)
> xval <- match(datemin, datev)
> abline(v=xval, col="red")
> text(x=xval, y=0.9*yval, "max drawdown", col="red", cex=0.9)
> xval <- match(endd, datev)
> abline(v=xval, col="green")
> text(x=xval, y=0.85*yval, "end drawdown", col="green", cex=0.9)
```

Drawdown Risk Using `PerformanceAnalytics::table.Drawdowns()`

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

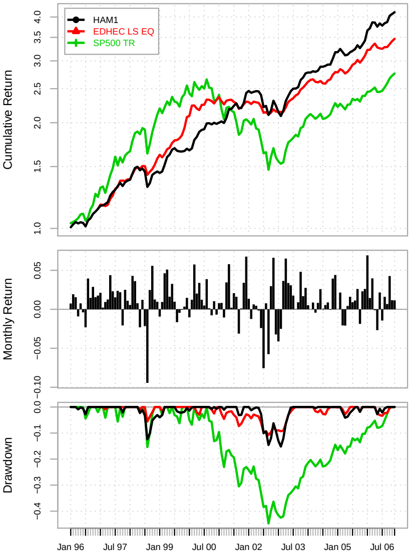
```
> library(xtable)
> library(PerformanceAnalytics)
> closep <- log(quantmod::Cl(rutils::etfenv$VTI))
> retp <- rutils::diffit(closep)
> # Calculate table of VTI drawdowns
> tablev <- PerformanceAnalytics::table.Drawdowns(retp, geometric=FALSE)
> # Convert dates to strings
> tablev <- cbind(sapply(tablev[, 1:3], as.character), tablev[, 4:7])
> # Print table of VTI drawdowns
> print(xtable(tablev), comment=FALSE, size="tiny", include.rownames=FALSE)
```

From	Trough	To	Depth	Length	To Trough	Recovery
2007-10-10	2009-03-09	2012-03-13	-0.57	1115.00	355.00	760.00
2001-06-06	2002-10-09	2004-11-04	-0.45	858.00	336.00	522.00
2020-02-20	2020-03-23	2020-08-12	-0.18	122.00	23.00	99.00
2022-01-04	2022-10-12	2023-12-18	-0.10	492.00	195.00	297.00
2018-09-21	2018-12-24	2019-04-23	-0.10	146.00	65.00	81.00

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(ham1,
+   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.

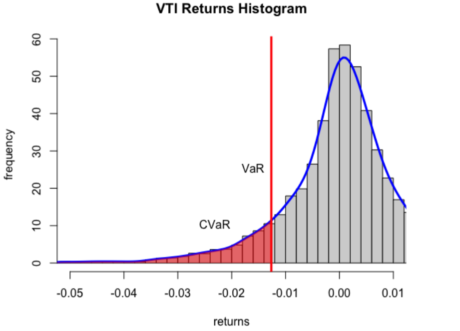
The *Value at Risk* (VaR) is equal to the quantile of returns corresponding to a given confidence level α.

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.

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> confl <- 0.1
> varisk <- quantile(retp, confl)
> cvar <- mean(retp[retp <= varisk])
> # Plot histogram of VTI returns
> x11(width=6, height=5)
> par(mar=c(3, 2, 1, 0), oma=c(0, 0, 0, 0))
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", ylab="frequency", breaks=100,
+   xlim=c(-0.05, 0.01), freq=FALSE, main="VTI Returns Histogram")
> # Calculate density
> densv <- density(retp, adjust=1.5)
```




```
> # Plot density
> lines(densv, lwd=3, col="blue")
> # Plot line for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk, y=25, labels="VaR", lwd=2, pos=2)
> # Plot polygon shading for CVaR
> text(x=1.5*varisk, y=10, labels="CVaR", lwd=2, pos=2)
> varmax <- -0.06
> rangev <- (densv$x < varisk) &  (densv$x > varmax)
> polygon(c(varmax, densv$x[rangev], varisk),
+   c(0, densv$y[rangev], 0), col=rgb(1, 0, 0,0.5), border=NA)
```

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`.

A simpler but less accurate way of calculating the quantile is by sorting and selecting the data closest to the quantile.

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

```
> # Calculate VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> confl <- 0.05
> # Calculate VaR approximately by sorting
> sortv <- sort(as.numeric(retp))
> cutoff <- round(confl*nrows)
> varisk <- sortv[cutoff]
> # Calculate VaR as quantile
> varisk <- quantile(retp, probs=confl)
> # PerformanceAnalytics VaR
> PerformanceAnalytics::VaR(retp, p=(1-confl), method="historical")
> all.equal(unname(varisk),
+   as.numeric(PerformanceAnalytics::VaR(retp,
+   p=(1-confl), 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
> varisk <- quantile(retp, confl)
> # Calculate CVaR as expected loss
> cvar <- mean(retp[retp <= varisk])
> # PerformanceAnalytics VaR
> PerformanceAnalytics::ETL(retp, p=(1-confl), method="historical")
> all.equal(unname(cvar),
+   as.numeric(PerformanceAnalytics::ETL(retp,
+     p=(1-confl), 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
> riskstats <-
+   PerformanceAnalytics::table.Stats(rutils::etfenv$returns)
> class(riskstats)
> # Transpose the data frame
> riskstats <- as.data.frame(t(riskstats))
> # Add Name column
> riskstats$Name <- rownames(riskstats)
> # Add Sharpe ratio column
> riskstats$"Arithmetic Mean" <-
+   sapply(rutils::etfenv$returns, mean, na.rm=TRUE)
> riskstats$Sharpe <-
+   sqrt(252)*riskstats$"Arithmetic Mean"/riskstats$Stdev
> # Sort on Sharpe ratio
> riskstats <- riskstats[order(riskstats$Sharpe, decreasing=TRUE), ]
```

	Sharpe	Skewness	Kurtosis
USMV	0.838	-0.864	21.73
AIEQ	0.823	-0.482	1.35
QUAL	0.732	-0.514	12.83
MTUM	0.689	-0.641	11.03
SPY	0.536	-0.295	11.06
VLUE	0.485	-0.941	17.19
GLD	0.484	-0.317	6.04
IEF	0.475	0.050	2.50
VTI	0.452	-0.385	10.79
VTV	0.449	-0.668	14.10
XLV	0.443	0.067	10.25
VYM	0.440	-0.681	14.91
XLP	0.423	-0.124	8.77
XLY	0.419	-0.363	6.53
IWB	0.400	-0.395	10.05
XLI	0.399	-0.376	7.61
IVW	0.395	-0.305	8.20
IWD	0.369	-0.488	12.83
XLU	0.365	-0.004	11.71
IVE	0.357	-0.483	10.31
IWF	0.356	-0.655	30.25
QQQ	0.353	-0.033	6.49
XLK	0.340	0.060	6.54
XLB	0.323	-0.369	5.43
EEM	0.292	0.025	15.91
XLE	0.263	-0.531	12.61
TLT	0.252	-0.011	3.42
VNQ	0.247	-0.538	18.38
XLF	0.195	-0.125	14.47
SVXY	0.163	-18.273	680.40
VEU	0.157	-0.509	12.00
DBC	0.021	-0.487	3.32
USO	-0.285	-1.127	14.12
VXX	-1.143	1.170	6.05

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*.

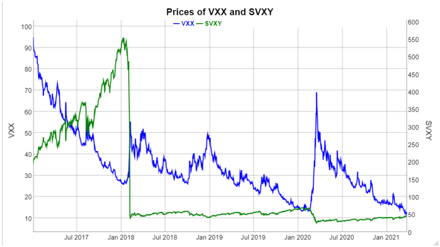
The *VXX* ETF is long the *VIX* index (effectively long an option), so it has positive skewness and small kurtosis, but negative returns (it's short market risk).

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	-1.143	1.17	6.05
SVXY	0.163	-18.27	680.40




```
> # dygraph plot of VXX versus SVXY
> pricev <- na.omit(rutils::etfenv$prices[, c("VXX", "SVXY")])
> pricev <- pricev["2017/"]
> colv <- c("VXX", "SVXY")
> colnames(pricev) <- colv
> dygraphs::dygraph(pricev, main="Prices of VXX and SVXY") %>%
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", strokeWidth=2, col="blue") %>%
+   dySeries(name=colv[2], axis="y2", strokeWidth=2, col="green") %
+   dyLegend(show="always", width=300) %>% dyLegend(show="always", w
+   dyLegend(show="always", width=300)
```

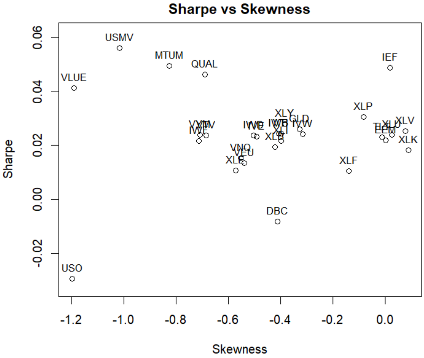
Skewness and Return Tradeoff

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
> riskstats <- riskstats[-match(c("VXX", "SVXY"), riskstats$Name), ]
> # Plot scatterplot of Sharpe vs Skewness
> plot(Sharpe ~ Skewness, data=riskstats,
+      ylim=1.1*range(riskstats$Sharpe),
+      main="Sharpe vs Skewness")
> # Add labels
> text(x=riskstats$Skewness, y=riskstats$Sharpe,
+     labels=riskstats$Name, pos=3, cex=0.8)
> # Plot scatterplot of Kurtosis vs Skewness
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 1), oma=c(0, 0, 0, 0))
> plot(Kurtosis ~ Skewness, data=riskstats,
+      ylim=c(1, max(riskstats$Kurtosis)),
+      main="Kurtosis vs Skewness")
> # Add labels
> text(x=riskstats$Skewness, y=riskstats$Kurtosis,
+     labels=riskstats$Name, pos=1, cex=0.5)
```



Risk-adjusted Return Measures

The *Sharpe ratio* $\mathrm{S_r}$ is equal to the excess returns (in excess of the risk-free rate r_f) divided by the standard deviation σ of the returns:

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

The *Sortino ratio* $\mathrm{So_r}$ 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):

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

The *Calmar ratio* $\mathrm{C_r}$ is equal to the excess returns divided by the *maximum drawdown* DD of the returns:

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

The *Dowd ratio* $\mathrm{D_r}$ is equal to the excess returns divided by the *Value at Risk* (VaR) of the returns:

$$\mathrm{D_r} = \frac{E[r - r_f]}{\mathrm{VaR}}$$

The *Conditional Dowd ratio* $\mathrm{Dc_r}$ is equal to the excess returns divided by the *Conditional Value at Risk* (CVaR) of the returns:

$$\mathrm{Dc_r} = \frac{E[r - r_f]}{\mathrm{CVaR}}$$

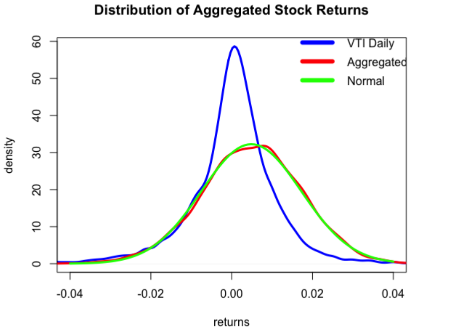
```
> library(PerformanceAnalytics)
> retp <- rutils::etfenv$returns[, c("VTI", "IEF")]
> retp <- na.omit(retp)
> # Calculate the Sharpe ratio
> confl <- 0.05
> PerformanceAnalytics::SharpeRatio(retp, p=(1-confl),
+   method="historical")
> # Calculate the Sortino ratio
> PerformanceAnalytics::SortinoRatio(retp)
> # Calculate the Calmar ratio
> PerformanceAnalytics::CalmarRatio(retp)
> # Calculate the Dowd ratio
> PerformanceAnalytics::SharpeRatio(retp, FUN="VaR",
+   p=(1-confl), method="historical")
> # Calculate the Dowd ratio from scratch
> varisk <- sapply(retp, quantile, probs=confl)
> -sapply(retp, mean)/varisk
> # Calculate the Conditional Dowd ratio
> PerformanceAnalytics::SharpeRatio(retp, FUN="ES",
+   p=(1-confl), method="historical")
> # Calculate the Conditional Dowd ratio from scratch
> cvar <- sapply(retp, function(x) {
+   mean(x[x < quantile(x, confl)])
+ })
> -sapply(retp, mean)/cvar
```

Risk of Aggregated Stock Returns

Stock returns aggregated over longer holding periods are closer to normally distributed, and their skewness, kurtosis, and tail risks are significantly lower than for daily returns.

Stocks become less risky over longer holding periods, so investors may choose to own a higher percentage of stocks, provided they hold them for a longer period of time.

```
> # Calculate VTI daily percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> nrows <- NROW(retp)
> # Bootstrap aggregated monthly VTI returns
> holdp <- 22
> reta <- sqrt(holdp)*sapply(1:nrows, function(x) {
+     mean(retp[sample.int(nrows, size=holdp, replace=TRUE)])
+ })  # end sapply
> # Calculate mean, standard deviation, skewness, and kurtosis
> datav <- cbind(retp, reta)
> colnames(datav) <- c("VTI", "Agg")
> sapply(datav, function(x) {
+   # Standardize the returns
+   meanv <- mean(x); stdev <- sd(x); x <- (x - meanv)/stdev
+   c(mean=meanv, stdev=stdev, skew=mean(x^3), kurt=mean(x^4))
+ })  # end sapply
> # Calculate the Sharpe and Dowd ratios
> confl <- 0.02
> ratiom <- sapply(datav, function(x) {
+   stdev <- sd(x)
+   varisk <- unname(quantile(x, probs=confl))
+   cvar <- mean(x[x < varisk])
+   mean(x)/c(Sharpe=stdev, Dowd=-varisk, DowdC=-cvar)
+ })  # end sapply
> # Annualize the daily risk
> ratiom[, 1] <- sqrt(22)*ratiom[, 1]
```




```
> # Plot the densities of returns
> plot(density(retp), t="l", lwd=3, col="blue",
+      xlab="returns", ylab="density", xlim=c(-0.04, 0.04),
+      main="Distribution of Aggregated Stock Returns")
> lines(density(reta), t="l", col="red", lwd=3)
> curve(expr=dnorm(x, mean=mean(reta), sd=sd(reta)), col="green", lw
> legend("topright", legend=c("VTI Daily", "Aggregated", "Normal"),
+  inset=-0.1, bg="white", lty=1, lwd=6, col=c("blue", "red", "greer
```

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`,