FRE6871 R in Finance

Lecture#1, Spring 2026

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

January 26, 2026

NYU | TANDON SCHOOL OF ENGINEERING

Welcome Students!

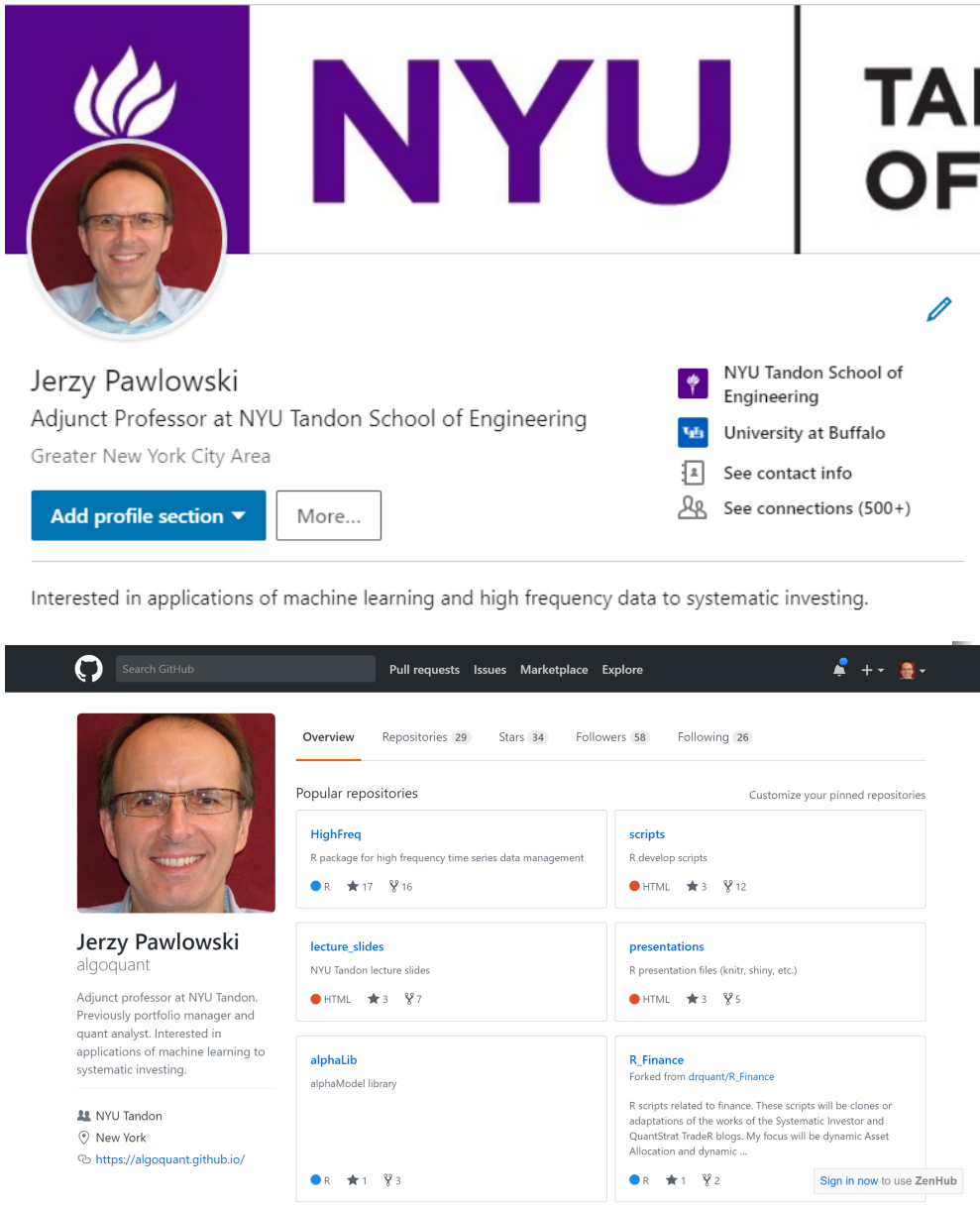
My name is Jerzy Pawlowski jp3900@nyu.edu

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

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

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

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



FRE6871 Course Description and Objectives

Course Description

The course will study the applications of the R statistical language to financial data analysis and modeling. The applications will include *classification* for credit scoring, *Monte Carlo simulation* for option pricing and credit portfolio modeling, and *Principal Component Analysis* (*PCA*) for interest rate yield curve modeling. The course will apply statistical techniques, such as *hypothesis testing*. *linear regression*, *logistic regression*, and *bootstrap simulation*.

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

FRE6871 Course Description and Objectives

Course Description

The course will study the applications of the R statistical language to financial data analysis and modeling. The applications will include *classification* for credit scoring, *Monte Carlo simulation* for option pricing and credit portfolio modeling, and *Principal Component Analysis* (*PCA*) for interest rate yield curve modeling. The course will apply statistical techniques, such as *hypothesis testing*. *linear regression*, *logistic regression*, and *bootstrap simulation*.

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

Course Objectives

Students will learn through R coding exercises how to:

- Manipulate data structures (vectors, data frames, dates, and time series).
- Download data from external sources, and to scrub and format it.
- Create interactive plots and visualizations.
- Build financial models.
- Perform exception and error handling, and debugging.

FRE6871 Course Description and Objectives

Course Description

The course will study the applications of the `R` statistical language to financial data analysis and modeling. The applications will include *classification* for credit scoring, *Monte Carlo simulation* for option pricing and credit portfolio modeling, and *Principal Component Analysis* (*PCA*) for interest rate yield curve modeling. The course will apply statistical techniques, such as *hypothesis testing*. *linear regression*, *logistic regression*, and *bootstrap simulation*.

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

Course Objectives

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

- Manipulate data structures (vectors, data frames, dates, and time series).
- Download data from external sources, and to scrub and format it.
- Create interactive plots and visualizations.
- Build financial models.
- Perform exception and error handling, and debugging.

Course Prerequisites

The `R` language is considered to be challenging, so this course requires some programming experience with other languages such as `C++` or `Python`. Students should also have knowledge of basic statistics (random variables, estimators, hypothesis testing, regression, etc.) The course *FRE7241 Algorithmic Portfolio Management* is designed as a followup course to *FRE6871*.

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 Preyansh Agrawal pa2753@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 of the 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).

FRE6871 Course Materials

Introductory Slides

The course will be mostly self-contained, using detailed lecture slides containing extensive, working `R` code examples.
Please download the introductory slides:
`R_environment.pdf`, `data_management.pdf`, `data_structures.pdf`, `expressions.pdf`, `packages.pdf`, `functions.pdf`, `plotting.pdf`
From the Share Drive
Use the slides as a reference.
The course will also utilize data and tutorials which are freely available on the internet.

FRE6871 Course Materials

Introductory Slides

The course will be mostly self-contained, using detailed lecture slides containing extensive, working R code examples.
Please download the introductory slides:
`R_environment.pdf`, `data_management.pdf`, `data_structures.pdf`, `expressions.pdf`, `packages.pdf`, `functions.pdf`, `plotting.pdf`
From the Share Drive
Use the slides as a reference.
The course will also utilize data and tutorials which are freely available on the internet.

FRE6871 Recommended Textbooks

- *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.
- *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.
- *Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, introduces machine learning techniques using R, but without deep learning.
- *Advanced R* by Hadley Wickham, is the best book for learning the advanced features of R.
- *The Art of R Programming* by Norman Matloff, contains a good introduction to R and to some statistical models.

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

FRE6871 Supplementary Textbooks

Supplementary Textbooks

- The books *R in Action* by Robert Kabacoff and *R for Everyone* by Jared Lander, are good introductions to R and to statistical models.
- *Applied Econometrics with R* by Christian Kleiber and Achim Zeileis, introduces advanced statistical models and econometrics.
- *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.
- *Quant Finance books* by Jerzy Pawlowski.
- *Quant Trading books* by Jerzy Pawlowski.

FRE6871 Supplementary Materials

`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/

Benchmarking the Speed of R Code

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

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

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

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

`microbenchmark()` executes the expression many times, and returns the distribution of total execution times in a *data frame*.

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

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

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

Writing Fast R Code Using *Compiled* C++ Functions

Compiled C++ functions directly call compiled C++ or Fortran code, which performs the calculations and returns the result back to R.

This makes *compiled* C++ functions much faster than *interpreted* functions, which have to be parsed by R.

`sum()` is much faster than `mean()`, because `sum()` is a *compiled* function, while `mean()` is an *interpreted* function.

Given a single argument, `any()` is equivalent to `%in%`, but is much faster because it's a *compiled* function.

`%in%` is a wrapper for `match()` defined as follows: `"%in%" <- function(x, table) match(x, table, nomatch=0) > 0`.

The function `all.equal()` tests the equality of two objects to within the square root of the *machine precision*.

```
> # sum() is a compiled primitive function
> sum
> # mean() is a generic function
> mean
> vecv <- runif(1e6)
> # sum() is much faster than mean()
> all.equal(mean(vecv), sum(vecv)/NROW(vecv))
> library(microbenchmark)
> summary(microbenchmark(
+   mean = mean(vecv),
+   sum = sum(vecv)/NROW(vecv),
+   times=10))[, c(1, 4, 5)]
> # any() is a compiled primitive function
> any
> # any() is much faster than %in% wrapper for match()
> all.equal(1 %in% vecv, any(vecv == 1))
> summary(microbenchmark(
+   inop = {1 %in% vecv},
+   anyfun = any(vecv == 1),
+   times=10))[, c(1, 4, 5)]
```

Writing Fast R Code Without Method Dispatch

As a general rule, calling generic functions is slower than directly calling individual methods, because generic functions must execute extra R code for method dispatch.

The generic function `as.data.frame()` coerces matrices and other objects into data frames.

The method `as.data.frame.matrix()` coerces only matrices into data frames.

`as.data.frame.matrix()` is about 50% faster than `as.data.frame()`, because it skips extra R code in `as.data.frame()` needed for argument validation, error checking, and method dispatch.

Users can create even faster functions of their own by extracting only the essential R code into their own specialized functions, ignoring R code needed to handle different types of data.

Such specialized functions are faster but less flexible, so they may fail with different types of data.

```
> library(microbenchmark)
> matv <- matrix(1:9, ncol=3, ##  Create matrix
+   dimnames=list(paste0("row", 1:3),
+            paste0("col", 1:3)))
> # Create specialized function
> matrix_to_dframe <- function(matv) {
+   ncols <- ncol(matv)
+   dframe <- vector("list", ncols)  ##  empty vector
+   for (indeks in 1:ncols)  ##  Populate vector
+     dframe <- matv[, indeks]
+   attr(dframe, "row.names") <-  ##  Add attributes
+     .set_row_names(NROW(matv))
+   attr(dframe, "class") <- "data.frame"
+   dframe  ##  Return data frame
+ }  ##  end matrix_to_dframe
> # Compare speed of three methods
> summary(microbenchmark(
+   matrix_to_dframe(matv),
+   as.data.frame.matrix(matv),
+   as.data.frame(matv),
+   times=10))[, c(1, 4, 5)]
```

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

All the different R loops have similar speed, but apply(), sapply(), and lapply() are sometimes faster than for() loops.

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

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

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

Increasing Speed of Loops by Pre-allocating Memory

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

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

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

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

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

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

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

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

Byte Compilation of R Functions

The *byte code compiler* translates R expressions into a simpler set of commands called *bytecode*, which can be interpreted much faster by a *byte code interpreter*.

Byte-compilation eliminates many routine interpreter operations, and typically speeds up processing by about 2 to 5 times.

The package `compiler` (included in R) contains functions for *byte-compilation*.

The function `compiler::cmpfun()` performs *byte-compilation* of a function.

When a function is passed into some functionals (like `microbenchmark()`) it is automatically *byte-compiled just-in-time* (JIT), so that when it's run the second time it runs faster.

The function `compiler::enableJIT()` enables or disables automatic *JIT byte-compilation*.

JIT is disabled if the `level` argument is equal to 0, with greater `level` values forcing more extensive compilation.

The default *JIT* `level` is 3.

```
> # Disable JIT
> jit_level <- compiler::enableJIT(0)
> # Create inefficient function
> meanfun <- function(x) {
+   datav <- 0; nrows <- NROW(x)
+   for(it in 1:nrows)
+     datav <- datav + x[it]/nrows
+   datav
+ }  ##  end meanfun
> # Byte-compile function and inspect it
> meanbyte <- compiler::cmpfun(meanfun)
> meanbyte
> # Test function
> vecv <- runif(1e3)
> all.equal(mean(vecv), meanbyte(vecv), meanfun(vecv))
> # microbenchmark byte-compile function
> summary(microbenchmark(
+   mean(vecv),
+   meanbyte(vecv),
+   meanfun(vecv),
+   times=10))[, c(1, 4, 5)]
> # Create another inefficient function
> sapply2 <- function(x, FUN, ...) {
+   datav <- vector(length=NROW(x))
+   for (it in seq_along(x))
+     datav[it] <- FUN(x[it], ...)
+   datav
+ }  ##  end sapply2
> sapply2_comp <- compiler::cmpfun(sapply2)
> all.equal(sqrt(vecv),
+   sapply2(vecv, sqrt),
+   sapply2_comp(vecv, sqrt))
> summary(microbenchmark(
+   sqrt(vecv),
+   sapply2_comp(vecv, sqrt),
+   sapply2(vecv, sqrt),
+   times=10))[, c(1, 4, 5)]
> # enable JIT
```

Profiling the Performance of R Expressions

Profiling of a computer program means measuring the amount of memory and time used for the execution of its different components.

Profiling can be implemented by polling a computer program in fixed time intervals, and writing the information (like the call stack) to a file.

The command `Rprof(filename)` turns on the profiling of R expressions, and saves the profiling data into the file `filename`.

If an R expression is executed after profiling is enabled, then its profiling data is written to the file `filename`.

The command `Rprof(NULL)` turns off profiling.

The function `summaryRprof()` compiles a summary of the profiling data from a file.

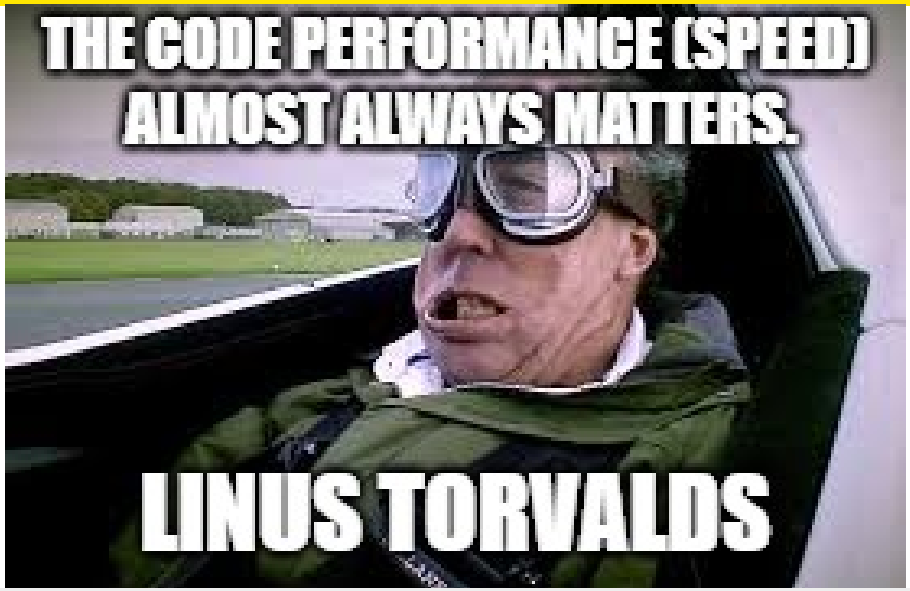
```
> # Define functions for profiling
> profun <- function() {fastfun(); slowfun()}
> fastfun <- function() Sys.sleep(0.1)
> slowfun <- function() Sys.sleep(0.2)
> # Turn on profiling
> Rprof(filename="/Users/jerzy/Develop/data_def/profile.out")
> # Run code for profiling
> replicate(n=10, profun())
> # Turn off profiling
> Rprof(NULL)
> # Compile summary of profiling from file
> summaryRprof("/Users/jerzy/Develop/data_def/profile.out")
```

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

AI systems don't always produce fast R code, so it's important to know how to write fast R code yourself.

How to write fast R code:

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




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

Increasing Speed of Loops by Pre-allocating Memory

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

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

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

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

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

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

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

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

Vectorized Functions for Vector Computations

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

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

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

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

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

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

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

```
> vec1 <- rnorm(1000000)
> vec2 <- rnorm(1000000)
> vecbig <- numeric(1000000)
> # Sum two vectors in two different ways
> summary(microbenchmark(
+   ##  Sum vectors using "for" loop
+   rloop = (for (i in 1:NROW(vec1)) {
+     vecbig[i] <- vec1[i] + vec2[i]
+   }),
+   ##  Sum vectors using vectorized "+"
+   vectorized = (vec1 + vec2),
+   times=10))[, c(1, 4, 5)]  ##  end microbenchmark summary
> # Allocate memory for cumulative sum
> cumsumv <- numeric(NROW(vecbig))
> cumsumv[1] <- vecbig[1]
> # Calculate cumulative sum in two different ways
> summary(microbenchmark(
+ # Cumulative sum using "for" loop
+   rloop = (for (i in 2:NROW(vecbig)) {
+     cumsumv[i] <- cumsumv[i-1] + vecbig[i]
+   }),
+ # Cumulative sum using "cumsum"
+   vectorized = cumsum(vecbig),
+   times=10))[, c(1, 4, 5)]  ##  end microbenchmark summary
```

Vectorized Functions for Matrix Computations

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

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

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

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

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

Fast R Code for Matrix Computations

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

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

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

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

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

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

Package matrixStats for Fast Matrix Computations

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

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

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

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

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

Package Rfast for Fast Matrix and Numerical Computations

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

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

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

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

Writing Fast R Code Using Vectorized Operations

Writing Fast R Code Using Vectorized Operations

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

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

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

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

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

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

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

Vectorized Functions

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

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

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

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

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

Performing sapply() Loops Over Function Parameters

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

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

```
> # Loop over stdevs produces vector output
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> sapply(stdevs, function(stdev) rnorm(n=2, sd=stdev))
> # Same
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> sapply(stdevs, rnorm, n=2, mean=0)
> # Loop over means
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> sapply(means, function(meanv) rnorm(n=2, mean=meanv))
> # Same
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> sapply(means, rnorm, n=2)
```

Creating Vectorized Functions

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

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

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

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

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

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

The mapply() Functional

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

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

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

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

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

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

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

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

Vectorizing Functions Using mapply()

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

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

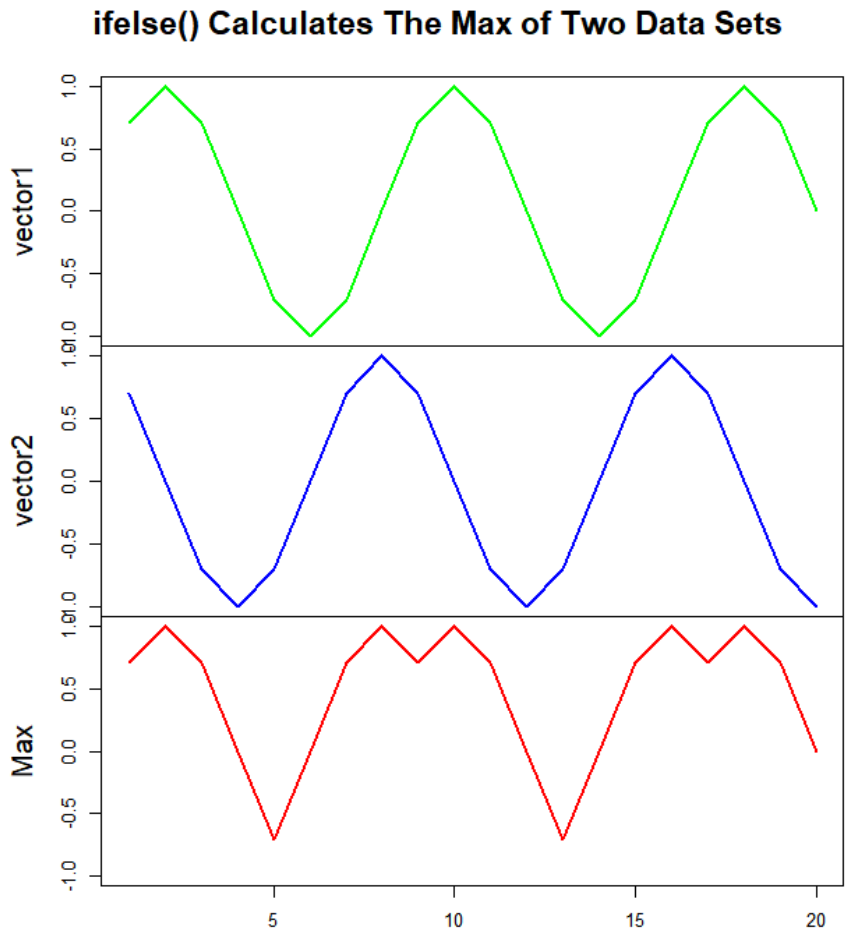
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  ##  end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2,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

Vectorized if-else Statements Using Function ifelse()

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

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

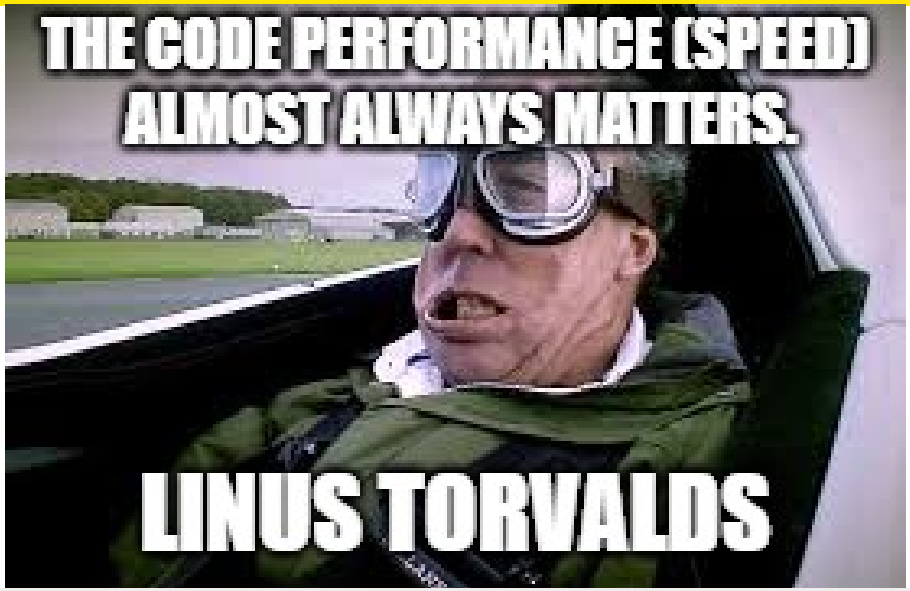
```
> # Create two numeric vectors
> vec1 <- sin(0.25*pi*1:20)
> vec2 <- cos(0.25*pi*1:20)
> # Create third vector using 'ifelse'
> vec3 <- ifelse(vec1 > vec2, vec1, vec2)
> # cbind all three together
> vec3 <- cbind(vec1, vec2, vec3)
> colnames(vec3)[3] <- "Max"
> # Set plotting parameters
> x11(width=6, height=7)
> par(oma=c(0, 1, 1, 1), mar=c(0, 2, 2, 1),
+     mgp=c(2, 1, 0), cex.lab=0.5, cex.axis=1.0, cex.main=1.8, cex.s
> # Plot matrix
> zoo::plot.zoo(vec3, lwd=2, ylim=c(-1, 1),
+   xlab="", col=c("green", "blue", "red"),
+   main="ifelse() Calculates The Max of Two Data Sets")
```



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

How to write fast R code:

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




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

Parallel Computing in R

Parallel Computing in R

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

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

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

R Base Package *parallel*

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

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

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

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

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

Parallel Computing Using Package *parallel*

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

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

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

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

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

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

Performing Parallel Loops Using Package *parallel*

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

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

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

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

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

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

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

Computing Advantage of Parallel Computing

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

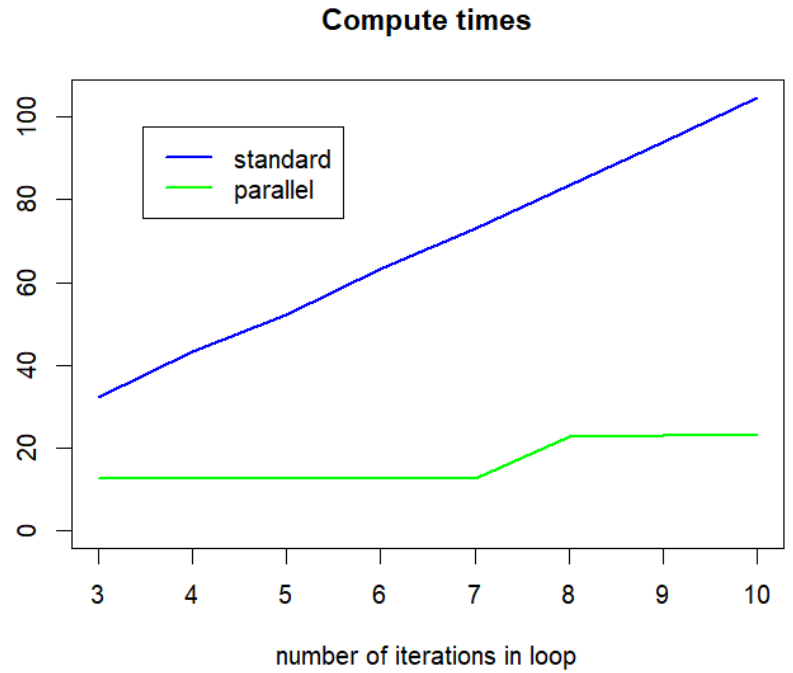
The function `stopCluster()` stops the `R` processes running on several CPU cores.

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

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

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

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




```
> x11(width=6, height=5)
> plot(x=rownames(timev),
+      y=timev[, "standard"],
+      type="l", lwd=2, col="blue",
+      main="Compute times",
+      xlab="Number of iterations in loop", ylab="",
+      ylim=c(0, max(timev[, "standard"])))
> lines(x=rownames(timev),
+ y=timev[, "parallel"], lwd=2, col="green")
> legend(x="topleft", legend=colnames(timev),
+  inset=0.1, cex=1.0, bty="n", bg="white",
+  y.intersp=0.3, lwd=2, lty=1, col=c("blue", "green"))
```

Parallel Computing Over Matrices

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

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

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

Initializing Parallel Clusters Under *Windows*

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

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

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

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

Reproducible Parallel Simulations Under *Windows*

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

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

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

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

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

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

Monte Carlo Simulation

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

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

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

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

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

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

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

Standard Errors of Estimators Using Bootstrap Simulation

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

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

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

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

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

The Distribution of Estimators Using Bootstrap Simulation

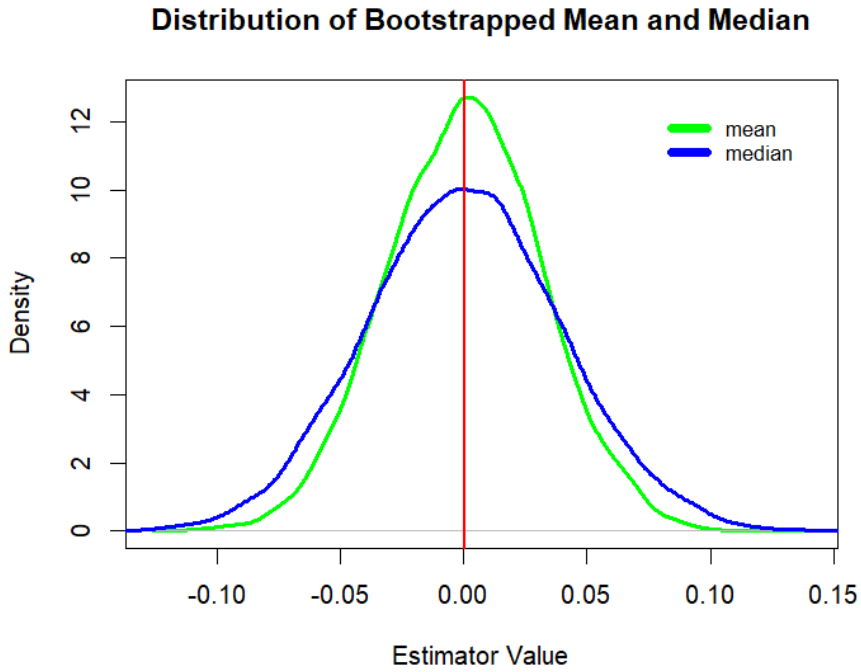
The standard errors of estimators can be calculated using a *bootstrap* simulation.

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

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

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

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




```
> # Plot the densities of the bootstrap data
> x11(width=6, height=5)
> plot(density(bootd[, "mean"]), lwd=3, xlab="Estimator Value",
+      main="Distribution of Bootstrapped Mean and Median", col="gre
> lines(density(bootd[, "median"]), lwd=3, col="blue")
> abline(v=mean(bootd[, "mean"]), lwd=2, col="red")
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("mean", "median"), bty="n", y.intersp=0.4,
+  lwd=6, bg="white", col=c("green", "blue"))
```

Bootstrapping Using Vectorized Operations

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

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

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

Faster code often requires more memory than slower code.

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

Bootstrapping Standard Errors Using Parallel Computing

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

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

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

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

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

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

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

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

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

Parallel Bootstrap of the *Median Absolute Deviation*

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

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

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

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

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

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

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

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

Standard Errors of Regression Coefficients Using Bootstrap

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

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

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

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

Distribution of Bootstrapped Regression Coefficients

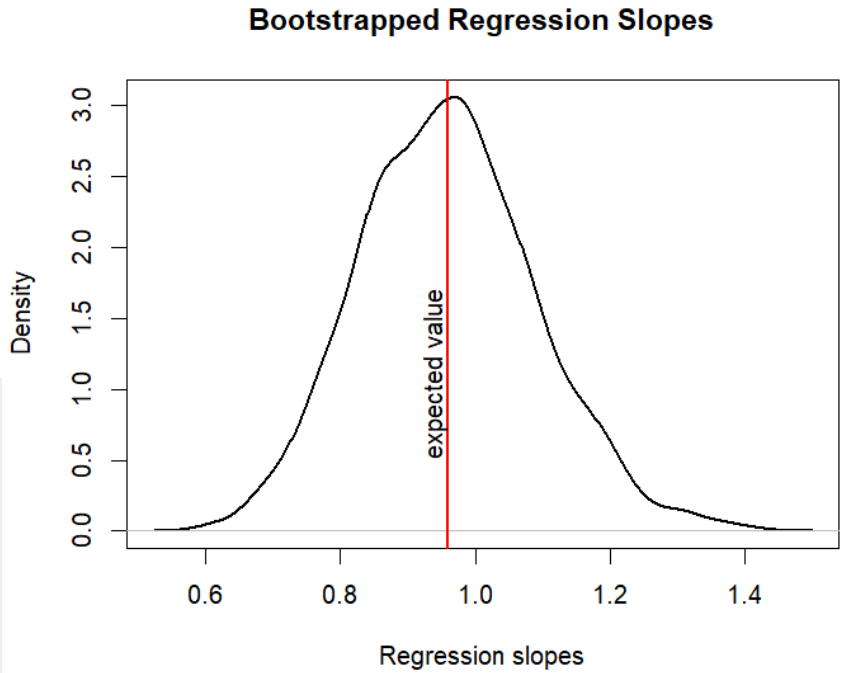
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

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

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

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

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



Bootstrapping Regressions Using Parallel Computing

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

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

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

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

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

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

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

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

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

Analyzing the Bootstrap Data

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

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

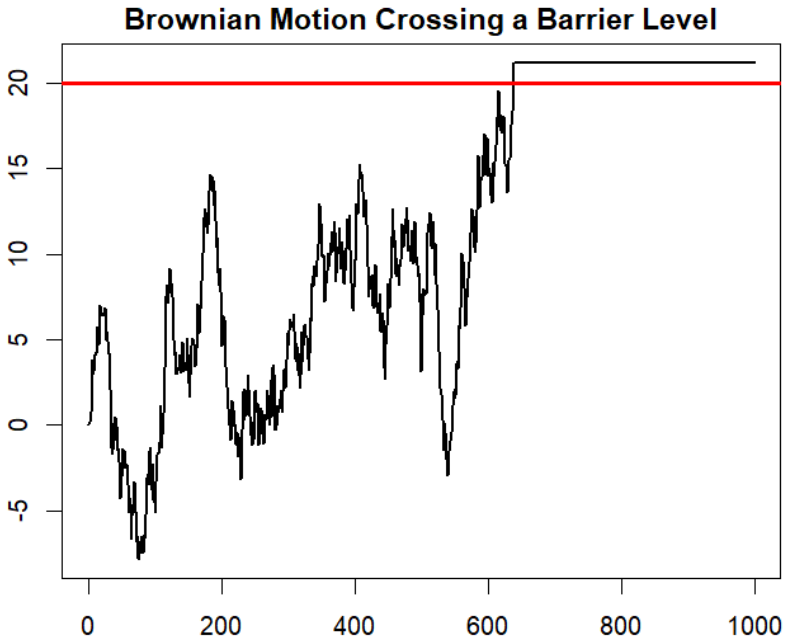
```
> # Collapse the bootstrap list into a vector
> class(bootd)
> bootd <- unlist(bootd)
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd),
+      lwd=2, xlab="Regression slopes",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

Simulating Brownian Motion Using while() Loops

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

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

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

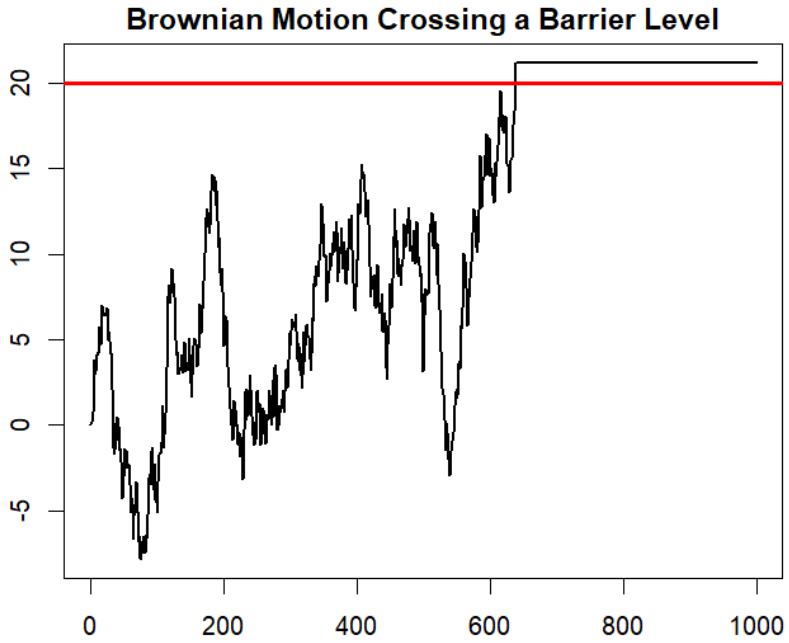


Simulating Brownian Motion Using Vectorized Functions

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

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

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



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

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

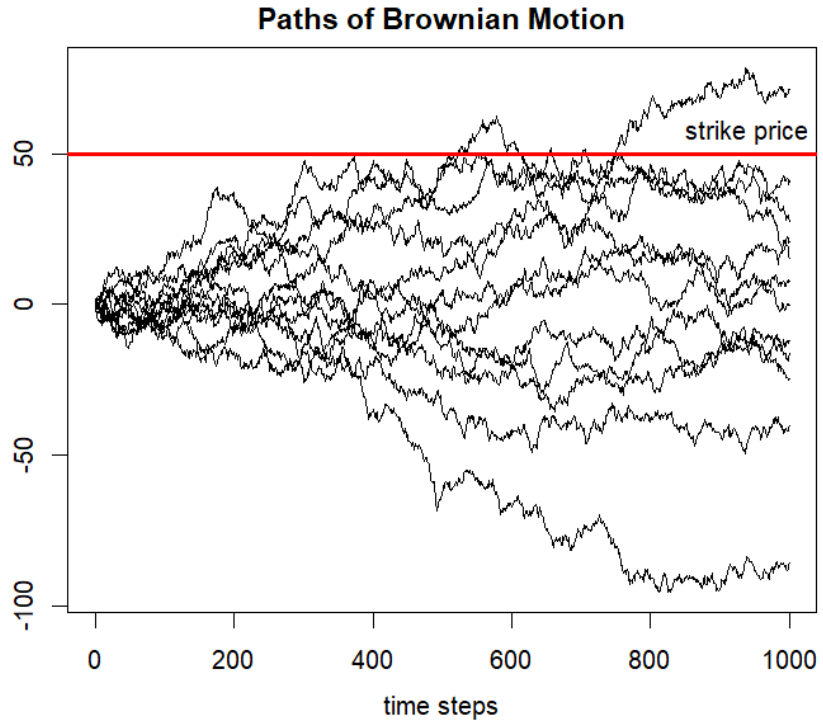
Estimating the Statistics of Brownian Motion

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

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

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

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




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

Resampling From Empirical Datasets

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

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

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

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

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

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

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

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

Bootstrapping From Empirical Datasets

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

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

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

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

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

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

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

```
> # Bootstrap sd and MAD under Windows
> library(parallel)  ##  Load package parallel
> ncores <- detectCores() - 1  ##  Number of cores
> compclust <- makeCluster(ncores)  ##  Initialize compute cluster u
> clusterSetRNGStream(compclust, 1121)  ##  Reset random number gene
> nboot <- 10000
> bootd <- parLapply(compclust, 1:nboot, function(x, retp, nsimu) {
+   samplev <- retp[sample.int(nsimu, replace=TRUE)]
+   c(sd=sd(samplev), mad=mad(samplev))
+ }, retp=retp, nsimu=nrows)  ##  end parLapply
> # Bootstrap sd and MAD under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, 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 under Wi
> bootd <- rutils::do_call(rbind, bootd)
> # Standard error of standard deviation assuming normal distributio
> sd(retp)/sqrt(nrows)
> # Means and standard errors from bootstrap
> stderr <- apply(bootd, MARGIN=2,
+   function(x) c(mean=mean(x), stderror=sd(x)))
> stderr
> # Relative standard errors
> stderr[2, ]/stderr[1, ]
```

Bootstrap of Time Series of Prices

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

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

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

```
> # Calculate log returns from VTI prices
> library(rutils)
> pricev <- quantmod::Cl(rutils::etfenv$VTI)
> pricev <- log(as.numeric(pricev))
> nrows <- NROW(pricev)
> prici <- pricev[1]
> retp <- rutils::diffit(pricev)
> class(retp); head(retp)
> sum(is.na(retp))
> # Define barrier level with respect to prices
> barl <- 2*max(pricev)
> # Calculate single bootstrap sample
> samplev <- retp[sample.int(nrows, replace=TRUE)]
> # Calculate prices from percentage returns
> samplev <- prici*exp(cumsum(samplev))
> # Calculate if prices crossed barrier
> sum(samplev > barl) > 0
```

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

Block Bootstrap of Time Series of Prices

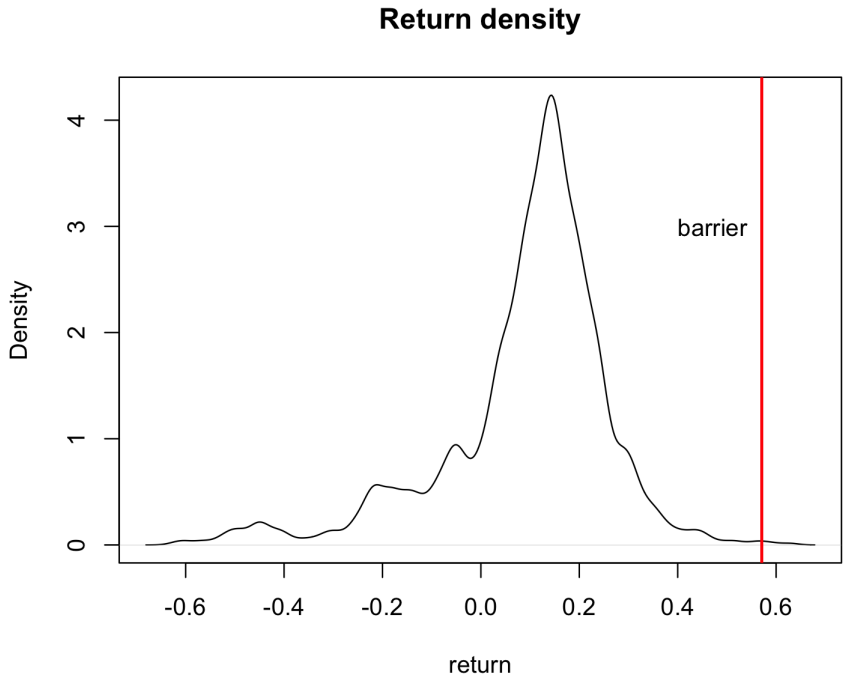
Bootstrapping from the empirical returns doesn't preserve the correlation structure of the returns.

In block bootstrap, multiple rows of the time series data are sampled to generate new data.

Block bootstrap requires a fixed time horizon, short enough to sample from the whole time series data.

For sampling from rows of data, it's better to convert the time series to a matrix.

```
> # Define barrier level with respect to the prices
> barl <- 0.1*max(pricev)
> # Define time horizon of 1 year in days
> holdp <- 252
> # Sample the start dates for the bootstrap
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> startd <- sample.int(nrows-holdp, nboot, replace=TRUE)
> # Bootstrap the cumulative returns
> samplev <- sapply(startd, function(x) {
+   pricev[x+holdp-1] - pricev[x]
+ })  ##  end sapply
> # Faster way to calculate the cumulative returns
> samplev <- pricev[startd+holdp-1] - pricev[startd]
> # Calculate how many cumulative returns exceed the barrier
> sum(samplev > barl) > 0
> # Plot the density of cumulative returns
> densv <- density(samplev)
> plot(densv, xlab="return", main="Return density")
> abline(v=barl, col="red", lwd=2)
> text(x=barl, y=0.7*max(densv$y), pos=2, "barrier")
```




```
> # Bootstrap the whole paths of cumulative returns
> samplev <- sapply(startd, function(x) {
+   pricev[x:(x+holdp-1)] - pricev[x]
+ })  ##  end sapply
> dim(samplev)
> samplev[1:5, 1:5]
> # Calculate which of the paths crossed the barrier at any point
> crossd <- apply(samplev, 2, function(x) {sum(x > barl) > 0})
> sum(crossd)
> which(crossd)
> plot(samplev[, which(crossd)[1]], t="l")
```

Bootstrapping From *OHLC* Prices

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

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

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

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

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

Variance Reduction Using Antithetic Sampling

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

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

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

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

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

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

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

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

Simulating Rare Events Using Probability Tilting

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

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

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

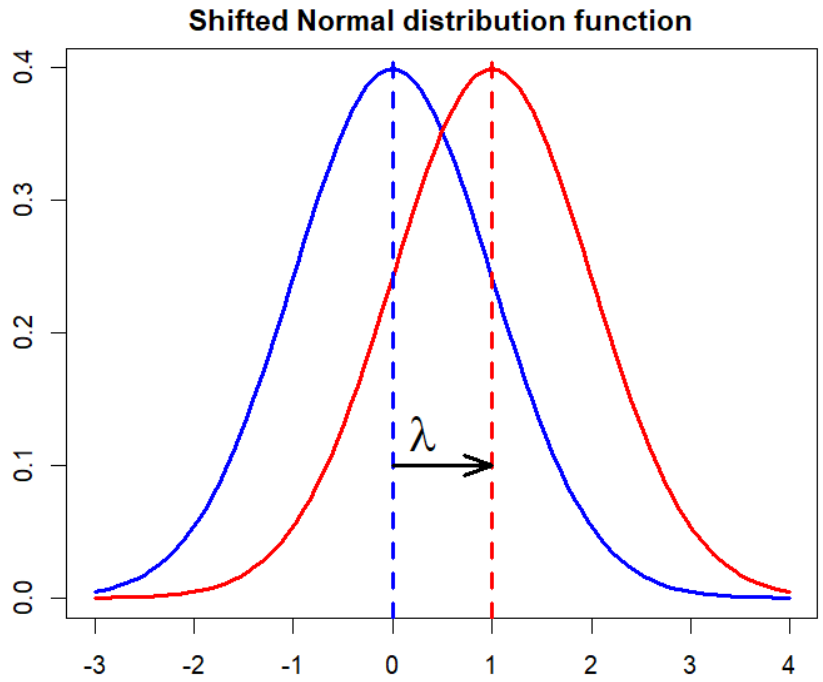
Where $p(x)$ is the probability density, $p(x,\lambda)$ is the tilted density, and λ is the tilt parameter.

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

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

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

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




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

Variance Reduction Using Importance Sampling

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

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

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

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

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

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

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

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

Calculating Quantiles Using Importance Sampling

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

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

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

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

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

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

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

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

Calculating *CVaR* Using Importance Sampling

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

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

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

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

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

The Optimal Tilt Parameter for Importance Sampling

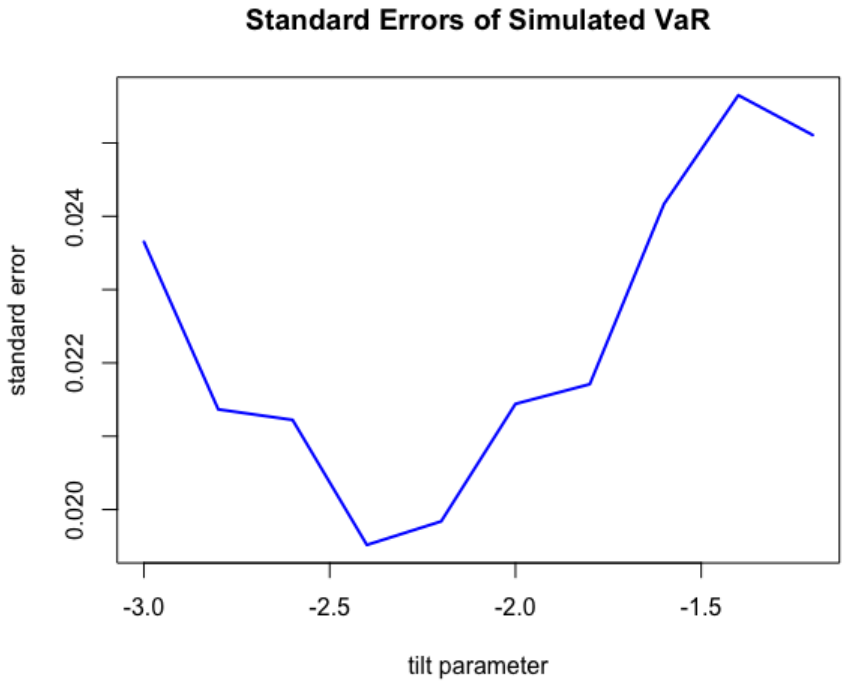
The tilt parameter λ should be chosen to minimize the standard error of the estimator.

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

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

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

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




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

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

Importance Sampling for Binomial Variables

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

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

Where λ is the tilt parameter.

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

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

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

Importance Sampling of Brownian Motion

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

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

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

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

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

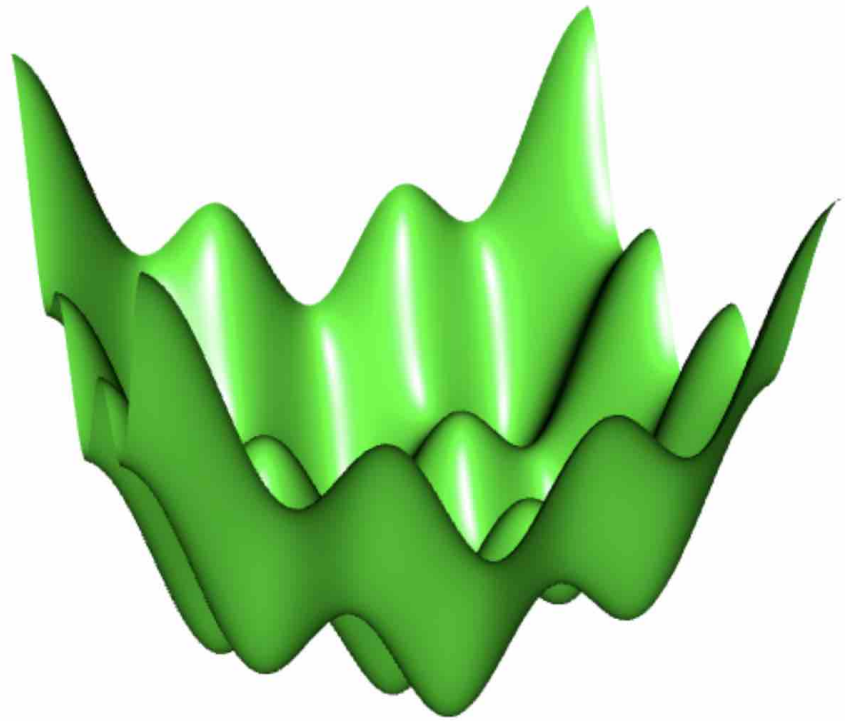
```
> # Define Brownian motion parameters
> sigmav <- 1.0  ##  Volatility
> drift <- 0.0  ##  Drift
> nsteps <- 100  ##  Number of simulation steps
> nsimu <- 1000  ##  Number of simulation paths
> # Calculate matrix of normal variables
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> datav <- rnorm(nsimu*nsteps, mean=drift, sd=sigmav)
> datav <- matrix(datav, nc=nsimu)
> # Simulate paths of Brownian motion
> pathm <- matrixStats::colCumsums(datav)
> # Tilt the datav
> lambdaf <- 0.1  ##  Tilt parameter
> datat <- datav + lambdaf  ##  Tilt the random numbers
> patht <- matrixStats::colCumsums(datat)
> zoo::plot.zoo(patht[, sample(nsimu, 20)], main="Paths of Brownian
> # Calculate path weights
> weightm <- exp(-lambdaf*datat + lambdaf^2/2)
> weightm <- matrixStats::colProds(weightm)
> # Or
> weightm <- exp(-lambdaf*colSums(datat) + nsteps*lambdaf^2/2)
> # Calculate option payout using naive MC
> strikep <- 10  ##  Strike price
> payouts <- (pathm[nsteps, ] - strikep)
> sum(payouts[payouts > 0])/nsimu
> # Calculate option payout using importance sampling
> payouts <- (patht[nsteps, ] - strikep)
> sum((weightm*payouts)[payouts > 0])/nsimu
> # Calculate crossing probability using naive MC
> barl <- 10
> crossi <- (colSums(pathm > barl) > 0)
> sum(crossi)/nsimu
> # Calculate crossing probability using importance sampling
> crossi <- colSums(patht > barl) > 0
> sum(weightm*crossi)/nsimu
```

Package *rgl* for Interactive 3d Surface Plots

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

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

```
> # Rastrigin function
> rastrigin <- function(x, y, param=25) {
+   x^2 + y^2 - param*(cos(x) + cos(y))
+ }  ##  end rastrigin
> # Rastrigin function is vectorized!
> rastrigin(c(-10, 5), c(-10, 5))
> # Set rgl options and load package rgl
> library(rgl)
> options(rgl.useNULL=TRUE)
> # Draw 3d surface plot of function
> rgl::persp3d(x=rastrigin, xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, param=15)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=800, height=800)
```



Newton-Raphson Optimization

The *Newton-Raphson* method finds the minimum of the objective function by finding the root of its first derivative, using a recursive formula:

$$x_{n+1} = x_n - \frac{f'(x)}{f''(x)}$$

It uses the first and second derivatives of the objective function to find the coordinate x of its minimum. If the derivatives are not given analytically, they can be approximated using finite differences.

Note that the coordinate step is larger when the second derivative $f''(x)$ is small, and vice versa. Because when the second derivative is small then the function is flat, so a larger step is needed to reach the minimum value quicker. If the second derivative is large then the function is more convex, so a smaller step is better to avoid overshooting the minimum.

```
> # Perform one-dimensional optimization using Newton-Raphson metho
> optim_newton <- function(f, x0, h = 1e-5, maxiter = 50, tol = 1e-
+   # Initialize the variables
+   x <- x0
+   histv <- numeric(maxiter + 1)
+   histv[1] <- x
+   # Iterate using Newton-Raphson formula
+   for (k in 1:maxiter) {
+     # Calculate the first derivative
+     fp <- (f(x + h) - f(x - h)) / (2 * h)
+     # Calculate the second derivative
+     fpp <- (f(x + h) - 2 * f(x) + f(x - h)) / (h^2)
+     # Check for convergence
+     if (abs(fp) < tol) {
+       return(list(root = x, history = histv[1:k]))
+     } # end if
+     # Update the coordinate
+     x <- x - fp / fpp
+     histv[k + 1] <- x
+   } # end for k
+   return(list(par = x, history = histv))
+ } # end optim_newton
> # Calculate the minimum using quasi-Newton method
> funx <- function(x) x^4 - 3*x^3 + 2
> optiml <- rutils::optim_newton(funx, x0 = 3)
> optiml$par
> optiml$history
> plot(optiml$history, type="b", main="Newton-Raphson Optimization
+      xlab="iteration", ylab="x value")
```

Newton-Raphson Multivariate Optimization

If the objective function is multivariate then the first derivative is a vector of gradients $f' = \nabla f = \frac{\partial f}{\partial x}$, and the second derivative is a matrix of partial derivatives called the Hessian $f'' = H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

The *Newton-Raphson* iterative formula for the minimum of a multivariate function, states that the update of the solution is equal to the inverse Hessian matrix times the function gradient:

$$x_{n+1} = x_n - H^{-1} \nabla f$$

The formula requires the calculation of the inverse of the Hessian matrix H^{-1} at each step.

But in practice, the objective function's derivatives and the inverse of its Hessian matrix are often not given analytically, so they must be approximated using finite differences.

Quasi-Newton BFGS Optimization

Quasi-Newton methods calculate the derivatives and the inverse Hessian matrix approximately at each iteration step.

The *BFGS* method updates the solution of the function minimum x_n by multiplying the inverse Hessian matrix H_n^{-1} times the function gradient ∇f_n:

$$x_{n+1} = x_n - \alpha\delta_n$$

Where $\delta_n = -H_n^{-1}\nabla f_n$ is the increment of x_n, and α is a factor to satisfy the Armijo condition.

The Armijo condition ensures that the objective function decreases with each iteration and that it doesn't overshoot the minimum:

$$f(x_n + \alpha\delta_n) \leq f(x_n) + \alpha\nabla f_n^T \delta_n$$

If the Armijo condition is not satisfied, then the factor α is decreased until the condition is satisfied.

The inverse of the Hessian matrix H_n^{-1} is updated at each step of the iteration using the *BFGS* formula:

$$H_{n+1}^{-1} = (I - \rho sy^T)H_n^{-1}(I - \rho ys^T) + \rho ss^T$$

Where $s = \alpha\delta_n$ is the change in the solution, $y = \nabla f_{n+1} - \nabla f_n$ is the change in the gradient, $\rho = \frac{1}{y^T s}$, and I is the identity matrix.

The updated inverse Hessian satisfies the secant condition: $H_{n+1}^{-1}(\nabla f_{n+1} - \nabla f_n) = x_{n+1} - x_n$.

```
> # BFGS loop to find the minimum
> for (n in 1:maxiter) {
+   dn <- - H %*% grad  # Increment of x
+   # Stop if increment is too small
+   if (sum(dn^2) < tol^2) break
+   # Perform loop to satisfy the Armijo condition
+   alpha <- 1
+   unchflag <- FALSE # Flag if x is unchanged
+   while (alpha > 1e-12) {
+     xn <- as.vector(x + alpha * dn) # New solution
+     fxn <- f(xn)                    # New function value
+     # Check for convergence based on step size
+     if (sum((xn - x)^2) < tol^2) {
+       unchflag <- TRUE # x is unchanged - stop loop
+       break
+     } # end if
+     # Armijo condition for sufficient decrease in function value
+     if (fxn <= fx + c1 * alpha * sum(grad * dn)) break
+     # Decrease the step size
+     alpha <- alpha * rho
+   } # end while
+   # If unchanged condition is detected, accept the current values
+   if (unchflag) {
+     x <- xn; fx <- fxn
+     history[itern, ] <- x; itern <- itern + 1
+     break
+   } # end if
+   # If the Armijo condition failed then stop
+   if (alpha <= 1e-12) break
+   # BFGS update of the gradient and inverse Hessian
+   dx <- xn - x; gradn <- calc_grad(xn)
+   dg <- gradn - grad; yp <- sum(dg * dx)
+   if (yp > 1e-12) {
+     I <- diag(narg)
+     V <- I - outer(dx, dg) / yp
+     H <- V %*% H %*% t(V) + outer(dx, dx) / yp
+   } # end if
+   # Copy variables for next iteration
```

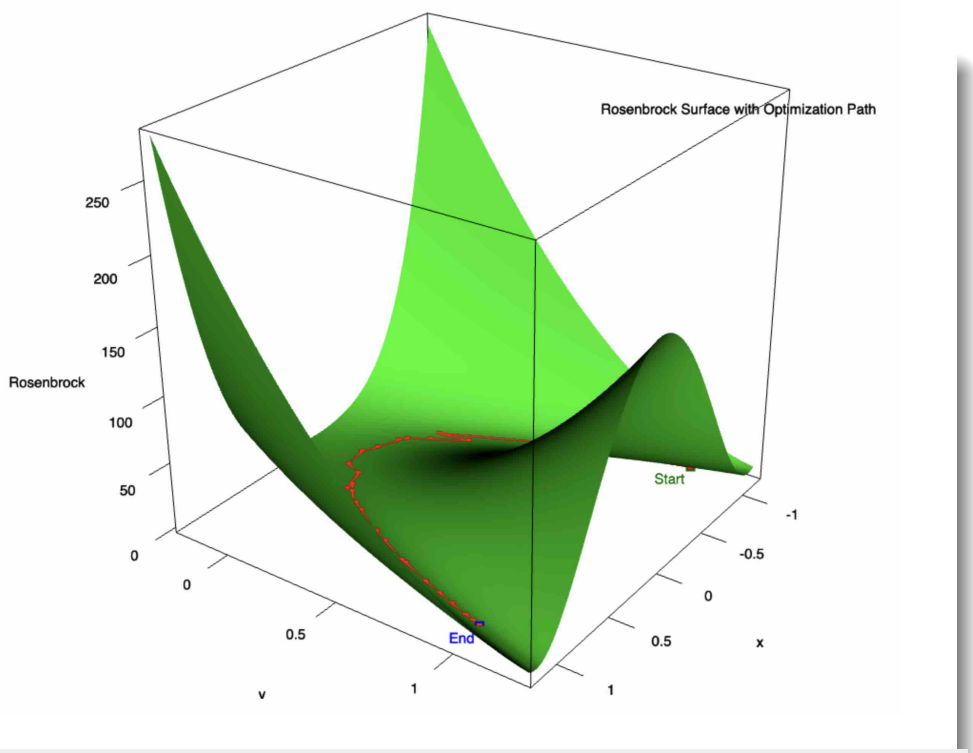
BFGS Optimization Path

The *BFGS* method is a popular quasi-Newton method, but it requires a large amount of memory for large multivariate optimization problems.

The *L-BFGS-B* method is a more efficient version of the *BFGS* method, which uses less memory.

The *Rosenbrock* function is used to test optimization algorithms. It has a global minimum at the point (1, 1) where the function value is 0.

```
> # Define the Rosenbrock function
> rosefun <- function(x) {
+   (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
+ } # end rosefun
> # Calculate the minimum using BFGS method
> optiml <- rutils::optim_bfgs(rosefun, x0 = c(-1.2, 1))
> optiml$par      # Minimum coordinates
> optiml$value    # Function value at the minimum
> optiml$history  # Optimization path
> optiml$iter     # Number of iterations
> # Solve using optim() for comparison
> optim(c(-1.2, 1), rosefun, method = "L-BFGS-B")
> ## Prepare data for 3D surface plot
> # Define the Rosenbrock function with two arguments
> rosefun2 <- function(x, y) {
+   (1 - x)^2 + 100 * (y - x^2)^2
+ } # end rosefun
> # Extract optimization path
> optimh <- optiml$history
> xp <- optimh[, 1]
> yp <- optimh[, 2]
> zp <- rosefun2(xp, yp)
> # Create grid for Rosenbrock surface
> xg <- seq(min(xp) - 0.5, max(xp) + 0.5, length.out = 100)
> yg <- seq(min(yp) - 0.5, max(yp) + 0.5, length.out = 100)
```




```
> # Draw 3d surface plot of function
> rgl::persp3d(x=rosefun2,
+       main = "Rosenbrock Surface with Optimization Path",
+       xlim=c(min(xp) - 0.2, max(xp) + 0.2),
+       ylim=c(min(yp) - 0.2, max(yp) + 0.2),
+       xlab = "x", ylab = "y", zlab = "Rosenbrock",
+       specular = "black",
+       col="green", axes=TRUE, param=15)
> # Draw optimization path
> rgl::lines3d(xp, yp, zp, col = "red", lwd = 3)
> rgl::points3d(xp, yp, zp, col = "red", size = 6)
> # Draw start and end points
> rgl::points3d(xp[1], yp[1], zp[1], col = "darkgreen", size = 10)
> rgl::points3d(xp[length(xp)], yp[length(yp)], zp[length(zp)],
+       col = "blue", size = 10)
> # Add labels
```

Coordinate Descent Optimization

Coordinate descent minimizes multivariate functions along each coordinate in a loop.

Coordinate descent can apply the quasi-Newton method to optimize along each coordinate separately.

The advantage of coordinate descent is that it doesn't need to calculate the inverse of the whole Hessian matrix. It only calculates the Hessian along each coordinate.

The inverse Hessian $H_{j,n}^{-1}$ for coordinate j at step $n+1$ is updated using the one-dimensional *BFGS* step:

$$H_{j,n+1}^{-1} = H_{j,n}^{-1} - \frac{\delta_x^2}{\delta_g}$$

Where $\delta_x = x_{j,n+1} - x_{j,n}$ is the change of the coordinate j, and $\delta_g = g_{j,n+1} - g_{j,n}$ is the change of the gradient along the coordinate j.

Coordinate descent performs well for high-dimensional problems like portfolio optimization. It also doesn't need to multiply and store large matrices in memory like the *BFGS* method does. But in many cases it's not as fast as the *BFGS* method.

```
> # Coordinate descent loop combined with BFGS quasi-Newton method
> for (n in 1:maxiter) {
+   xold <- x # Previous value of x
+   # Loop over the coordinates
+   for (j in 1:narg) {
+     gj <- grad[j]
+     if (abs(gj) < tol2) next # Skip this coordinate if gradient is
+     # The increment of the coordinate j
+     dj <- - H[j] * gj
+     if (abs(dj) < tol2) next # Skip this coordinate if step is too
+     # Perform Armijo condition loop along coordinate j
+     alpha <- 1
+     repeat {
+       xtr <- x # Trial value of x
+       xtr[j] <- x[j] + alpha * dj
+       fxtr <- f(xtr)  # Trial function value
+       # Break if Armijo condition is satisfied
+       if ((fxtr <= fx + c1 * alpha * gj * dj) || (alpha < 1e-12))
+ break
+       # Reduce the step size and repeat test
+       alpha <- alpha * rho
+     } # end repeat
+     # If the Armijo condition failed then skip this coordinate
+     dx <- alpha * dj
+     if (abs(dx) < tol2) next
+     # Accept the step
+     x[j] <- x[j] + dx
+     fx <- fxtr
+     # Update gradient component using central difference
+     e <- rep(0, narg); e[j] <- 1
+     fp <- f(x + h * e)[1]
+     fm <- f(x - h * e)[1]
+     gn <- (fp - fm) / (2 * h) # New gradient component
+     dg <- (gn - gj) # Change in gradient
+     # Hessian secant update
+     if (dg * dx > 1e-12) {
+       H[j] <- max(1e-12, dx / dg)
+     } else {
```

One-dimensional Optimization Using The Functional optimize()

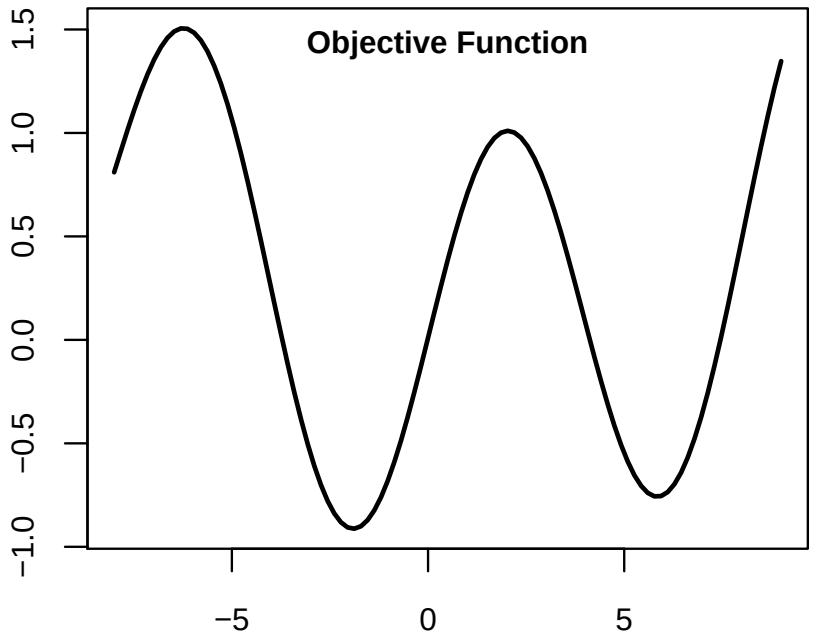
The functional optimize() performs *one-dimensional* optimization over a single independent variable.

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

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

The argument tol specifies the numerical accuracy, with smaller values of tol requiring more computations.

```
> # Display the structure of optimize()
> str(optimize)
> # Objective function with multiple minima
> objfun <- function(input, param1=0.01) {
+   sin(0.25*pi*input) + param1*(input-1)^2
+ }  ##  end objfun
> optiml <- optimize(f=objfun, interval=c(-4, 2))
> class(optiml)
> unlist(optiml)
> # Find minimum in different interval
> unlist(optimize(f=objfun, interval=c(0, 8)))
> # Find minimum with less accuracy
> accl <- 1e4*.Machine$double.eps^0.25
> unlist(optimize(f=objfun, interval=c(0, 8), tol=accl))
> # Microbenchmark optimize() with less accuracy
> library(microbenchmark)
> summary(microbenchmark(
+   more_accurate = optimize(f=objfun, interval=c(0, 8)),
+   less_accurate = optimize(f=objfun, interval=c(0, 8), tol=accl),
+   times=100))[, c(1, 4, 5)]  ##  end microbenchmark summary
```




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

Multi-dimensional Optimization Using optim()

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

The argument fn is the objective function to be minimized.

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

The argument par is the initial vector argument value.

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

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

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

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

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

```
> # Rastrigin function with vector argument for optimization
> rastrigin <- function(vecv, param=25) {
+   sum(vecv^2 - param*cos(vecv))
+ }  ##  end rastrigin
> vecv <- c(pi, pi/4)
> rastrigin(vecv=vecv)
> # Draw 3d surface plot of Rastrigin function
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastrigin(vecv=c(x, y))),
+   xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, zlab="", main="rastrigin")
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=800, height=800)
> # Optimize with respect to vector argument
> optiml <- optim(par=vecv, fn=rastrigin,
+         method="L-BFGS-B",
+         upper=c(14*pi, 14*pi),
+         lower=c(pi/2, pi/2),
+         param=1)
> # Optimal parameters and value
> optiml$par
> optiml$value
> rastrigin(optiml$par, param=1)
```

Optimization Algorithms Using optim()

Table: Comparison of optimization methods for portfolio optimization

Method	Type	Strengths	Weaknesses	Suitability for Portfolio Optimization
BFGS	Quasi-Newton	Fast convergence, uses gradient + Hessian approximation	Memory-heavy for large problems	Strong choice for medium-sized portfolios
L-BFGS-B	Limited-memory BFGS with box constraints	Handles bounds (e.g., weights ≥ 0), memory-efficient	Requires smooth objective	Best overall for portfolio optimization with constraints
CG (Conjugate Gradient)	Gradient-based	Efficient for large-scale problems	Slower convergence than BFGS	Good for very large portfolios
Nelder-Mead	Derivative-free simplex	Works without gradients, robust for small problems	Slow, scales poorly in high dimensions	Limited use; not ideal for large portfolios
Brent	1D derivative-free	Very efficient in one dimension	Only works for single-variable problems	Not applicable to portfolios
SANN (Simulated Annealing)	Stochastic global search	Escapes local minima, works on non-smooth functions	Very slow, no guarantee of optimality	Useful for highly non-convex problems

The Likelihood Function

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

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

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

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

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

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

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

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

Perspective Plot of Likelihood Function

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

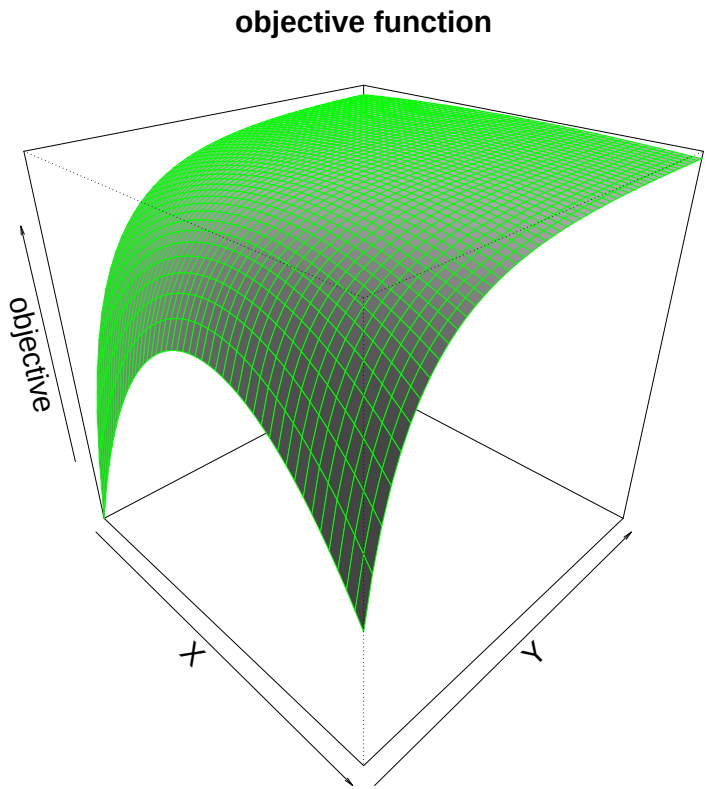
The argument `"z"` accepts a matrix containing the function values.

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

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

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

```
> # Perspective plot of log-likelihood function
> persp(z=-objgrid,
+       theta=45, phi=30, shade=0.5,
+       border="green", zlab="objective",
+       main="objective function")
> # Interactive perspective plot of log-likelihood function
> library(rgl)  ##  Load package rgl
> rgl::par3d(cex=2.0)  ##  Scale text by factor of 2
> rgl::persp3d(z=-objgrid, zlab="objective",
+ col="green", main="objective function")
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=800, height=800)
```

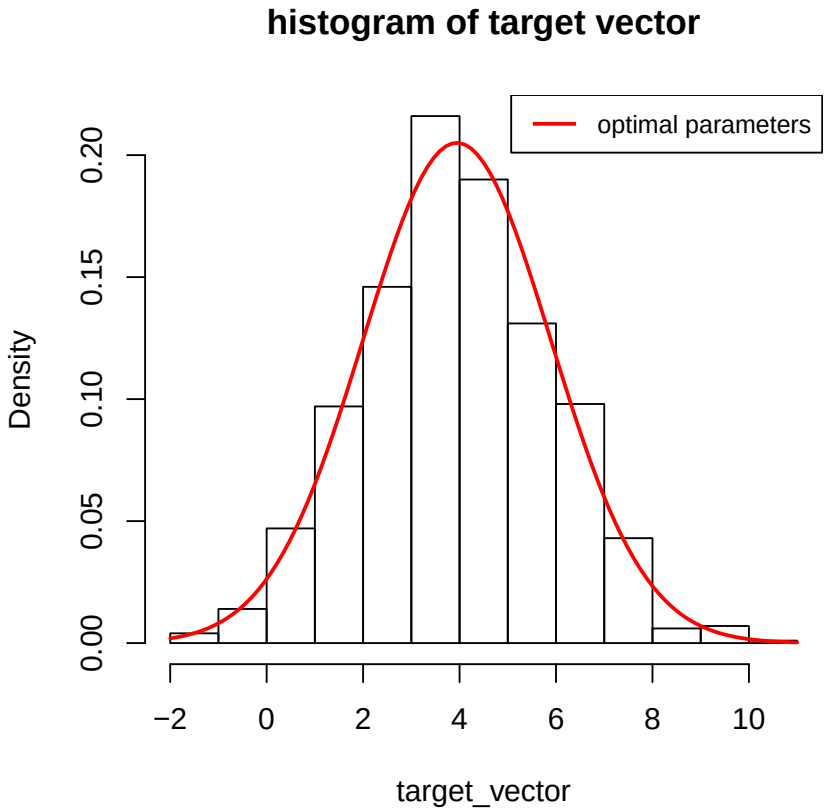


Optimization of Objective Function

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

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

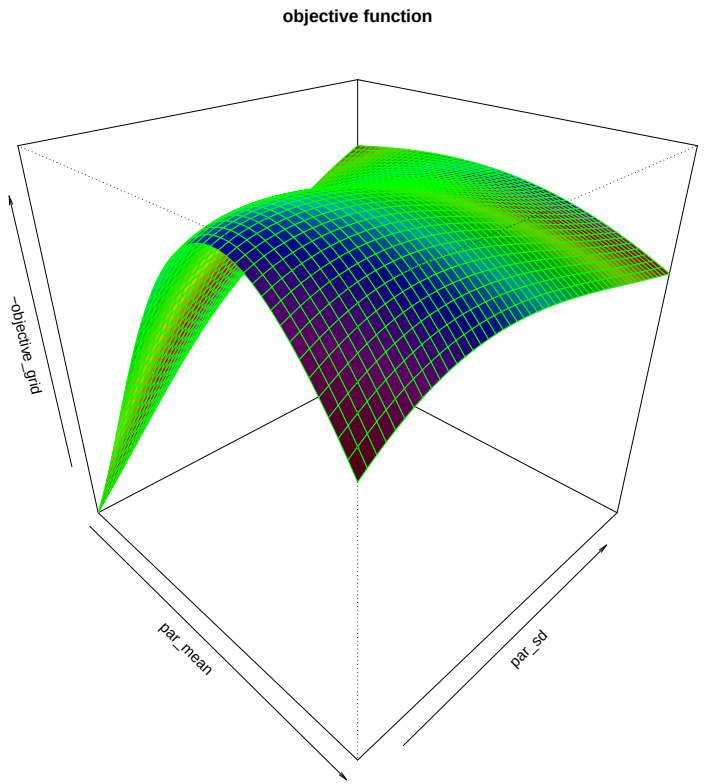
```
> # Initial parameters
> initp <- c(mean=0, sd=1)
> # Perform optimization using optim()
> optiml <- optim(par=initp,
+   fn=objfun, ##  Log-likelihood function
+   datav=datav,
+   method="L-BFGS-B", ##  Quasi-Newton method
+   upper=c(10, 10), ##  Upper constraint
+   lower=c(-10, 0.1)) ##  Lower constraint
> # Optimal parameters
> optiml$par
> # Perform optimization using MASS::fitdistr()
> optiml <- MASS::fitdistr(datav, densfun="normal")
> optiml$estimate
> optiml$sd
> # Plot histogram
> histp <- hist(datav, plot=FALSE)
> plot(histp, freq=FALSE, main="histogram of sample")
> curve(expr=dnorm(x, mean=optiml$par["mean"], sd=optiml$par["sd"]),
+ add=TRUE, type="l", lwd=2, col="red")
> legend("topright", leg="optimal parameters",
+    inset=0.0, cex=0.8, title=NULL, y.intersp=0.4,
+    bty="n", lwd=2, bg="white", col="red")
```



Mixture Model Likelihood Function

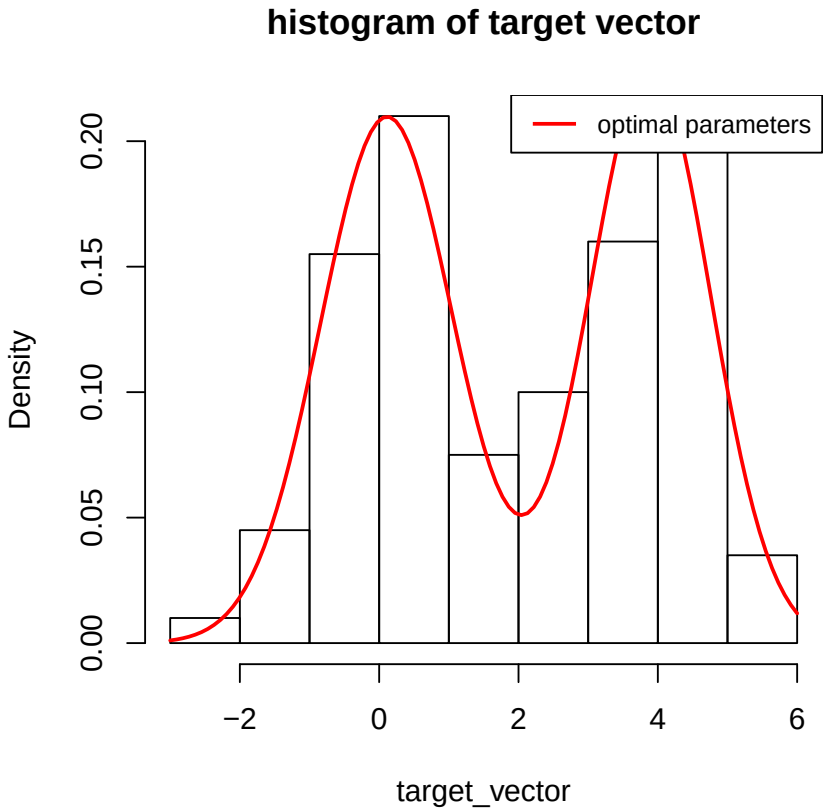
```
> # Sample from mixture of normal distributions
> datav <- c(rnorm(100, sd=1.0),
+       rnorm(100, mean=4, sd=1.0))
> # Objective function is log-likelihood
> objfun <- function(parv, datav) {
+   likev <- parv[1]/parv[3] *
+   dnorm((datav-parv[2])/parv[3]) +
+   (1-parv[1])/parv[5]*dnorm((datav-parv[4])/parv[5])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

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



Optimization of Mixture Model

```
> # Initial parameters
> initp <- c(weight=0.5, m1=0, s1=1, m2=2, s2=1)
> # Perform optimization
> optiml <- optim(par=initp,
+       fn=objfun,
+       datav=datav,
+       method="L-BFGS-B",
+       upper=c(1,10,10,10,10),
+       lower=c(0,-10,0.2,-10,0.2))
> optiml$par
> # Plot histogram
> histp <- hist(datav, plot=FALSE)
> plot(histp, freq=FALSE,
+      main="histogram of sample")
> fitfun <- function(x, parv) {
+   parv["weight"]*dnorm(x, mean=parv["m1"], sd=parv["s1"]) +
+   (1-parv["weight"])*dnorm(x, mean=parv["m2"], sd=parv["s2"])
+ }  ##  end fitfun
> curve(expr=fitfun(x, parv=optiml$par), add=TRUE,
+ type="l", lwd=2, col="red")
> legend("topright", leg="optimal parameters", inset=0.0,
+  cex=0.8, title=NULL, y.intersp=0.4, bty="n",
+  lwd=2, bg="white", col="red")
```



Package *DEoptim* for Global Optimization

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

Differential Evolution is a genetic algorithm which evolves a population of solutions over several generations:
https://link.springer.com/content/pdf/10.1023/A:1008202821328.pdf

The first generation of solutions is selected randomly.

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

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

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

A limitation of *Differential Evolution* is that it doesn't estimate the standard errors of the parameters.

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

Homework Assignment

Required

- Download the introductory slides from the Share Drive, and use them as references: `R_environment.pdf`, `data_management.pdf`, `data_structures.pdf`, `expressions.pdf`, `packages.pdf`, `functions.pdf`, `plotting.pdf`
- Study all the lecture slides in `FRE6871_Lecture_1.pdf`, and run all the code in `FRE6871_Lecture_1.R`,
- Read about *bootstrap simulation* from the files `bootstrap_technique.pdf` and `doBootstrap_primer.pdf`,
- Read about *optimization methods*:
  Bolker Optimization Methods.pdf, *Yollin Optimization.pdf*, *Boudt DEoptim Large Portfolio Optimization.pdf*.

Recommended

- Download and read the `R` Cheat Sheets from here.
- Read about plotting from *plot_par_cheatsheet.pdf* and *ggplot2_cheatsheet.pdf*.
- Study the *RStudio Style Guide*.