FRE6871 R in Finance

Lecture#2, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 12, 2022

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

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

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

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*.

```
> library(microbenchmark)
> # sum() is a compiled primitive function
> sum
> # mean() is a generic function
> mean
> vectorv <- runif(1e6)
> # sum() is much faster than mean()
> all.equal(mean(vectorv), sum(vectorv)/NROW(vectorv))
> summary(microbenchmark(
+   mean_fun = mean(vectorv),
+   sum_fun = sum(vectorv)/NROW(vectorv),
+   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% vectorv, any(vectorv == 1))
> summary(microbenchmark(
+   in_fun = {1 %in% vectorv},
+   any_fun = any(vectorv == 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)
> matrixv <- matrix(1:9, ncol=3, # Create matrix
+   dimnames=list(paste0("row", 1:3),
+           paste0("col", 1:3)))
> # Create specialized function
> matrix_to_dframe <- function(matrixv) {
+   ncols <- ncol(matrixv)
+   dframe <- vector("list", ncols)  # empty vector
+   for (indeks in 1:ncols)  # Populate vector
+     dframe <- matrixv[, indeks]
+   attr(dframe, "row.names") <-  # Add attributes
+     .set_row_names(NROW(matrixv))
+   attr(dframe, "class") <- "data.frame"
+   dframe  # Return data frame
+ }  # end matrix_to_dframe
> # Compare speed of three methods
> summary(microbenchmark(
+   matrix_to_dframe(matrixv),
+   as.data.frame.matrix(matrixv),
+   as.data.frame(matrixv),
+   times=10))[, c(1, 4, 5)]
```

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

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

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

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

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

Increasing Speed of Loops by Pre-allocating Memory

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

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

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

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

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

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

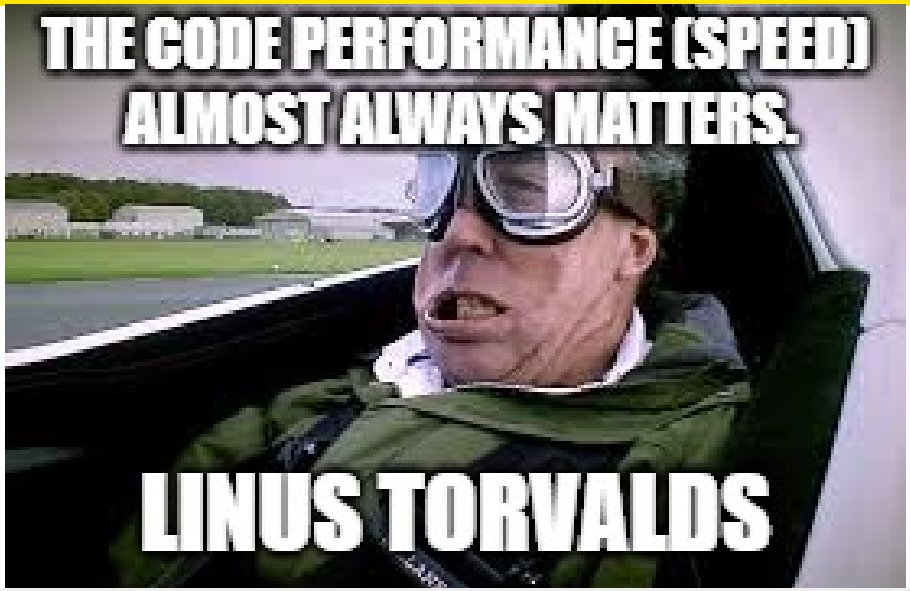
The function `numeric(k)` returns a numeric vector of zeros of length `k`, while `numeric(0)` returns an empty (zero length) numeric vector (not to be confused with a `NULL` object).

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

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

How to write fast R code:

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




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

Benchmarking the Speed of R Code

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

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

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

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

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

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

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

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

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

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

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

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

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

Increasing Speed of Loops by Pre-allocating Memory

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

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

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

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

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

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

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

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

Vectorized Functions for Vector Computations

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

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

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

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

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

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

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

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

Vectorized Functions for Matrix Computations

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

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

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

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

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

Fast R Code for Matrix Computations

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

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

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

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

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

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

Package matrixStats for Fast Matrix Computations

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

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

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

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

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

Package Rfast for Fast Matrix and Numerical Computations

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

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

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

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

Writing Fast R Code Using Vectorized Operations

Writing Fast R Code Using Vectorized Operations

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

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

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

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

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

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

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

Vectorized Functions

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

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

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

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

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

Performing sapply() Loops Over Function Parameters

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

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

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

Creating Vectorized Functions

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

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

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

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

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

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

The mapply() Functional

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

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

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

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

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

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

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

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

Vectorizing Functions Using mapply()

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

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

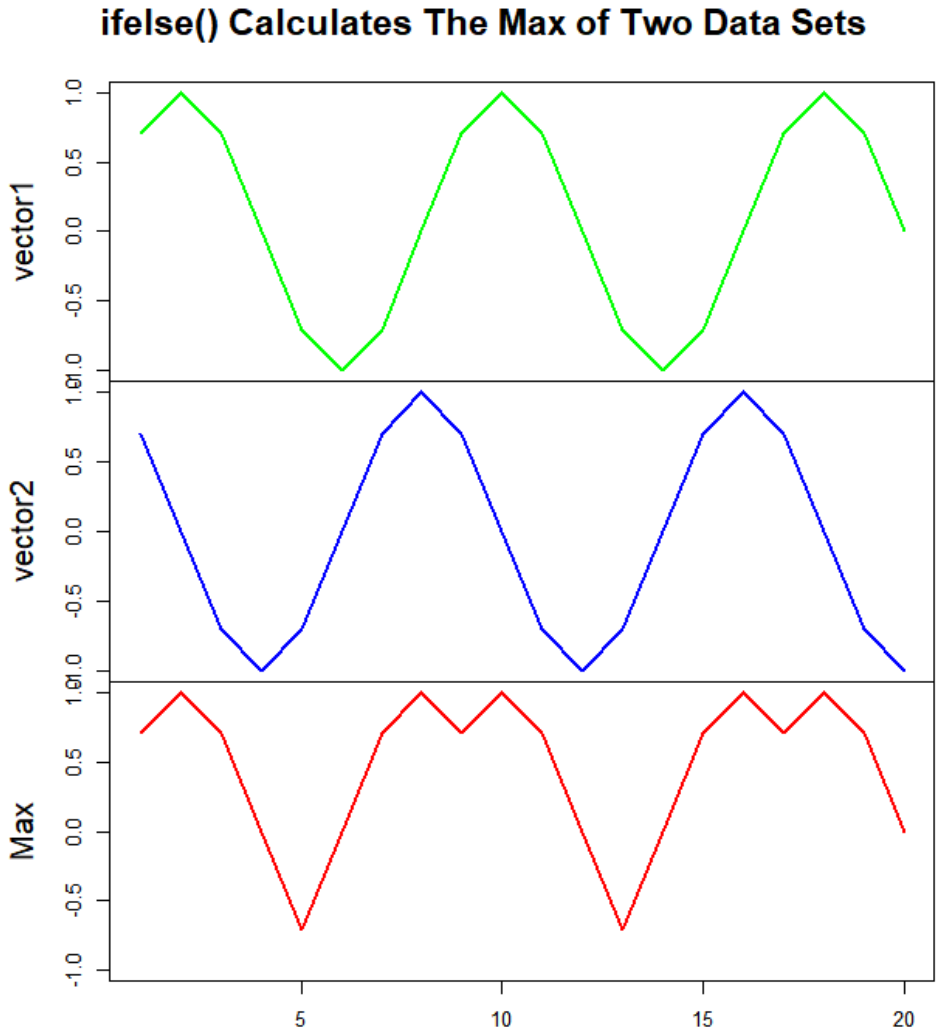
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2,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

Vectorized if-else Statements Using Function ifelse()

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

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

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



Parallel Computing in R

Parallel Computing in R

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

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

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

R Base Package *parallel*

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

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

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

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

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

Parallel Computing Using Package *parallel*

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

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

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

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

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

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

Performing Parallel Loops Using Package *parallel*

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

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

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

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

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

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

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

Computing Advantage of Parallel Computing

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

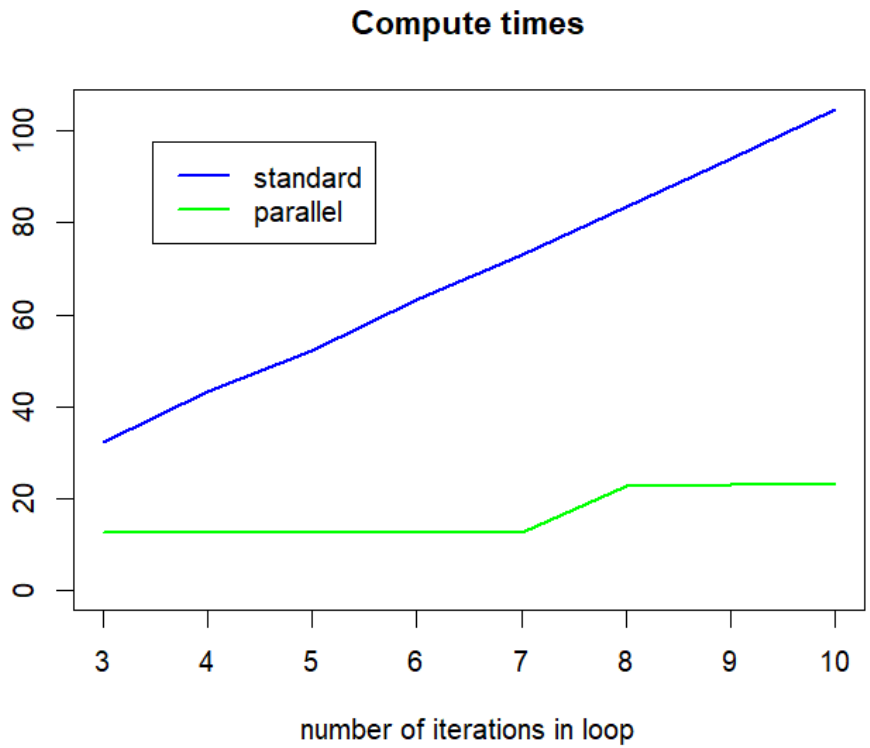
The function `stopCluster()` stops the `R` processes running on several CPU cores.

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

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

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

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




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

Parallel Computing Over Matrices

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

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

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

Initializing Parallel Clusters Under *Windows*

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

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

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

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

Reproducible Parallel Simulations Under *Windows*

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

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

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

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

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

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

Pseudo-Random Numbers

Pseudo-random numbers are deterministic sequences of numbers which have some of the properties of random numbers, but they are not truly random numbers.

Pseudo-random number generators depend on a *seed* value, and produce the same sequence of numbers for a given *seed* value.

The function `set.seed()` initializes the random number generator by specifying the *seed* value.

The choice of *seed* value isn't important, and a given value is just good as any other one.

The function `runif()` produces random numbers from the *uniform* distribution.

The function `rnorm()` produces random numbers from the *normal* distribution.

The function `rt()` produces random numbers from the *t-distribution* with *df* degrees of freedom.

```
> set.seed(1121)  # Reset random number generator
> runif(3)  # three numbers from uniform distribution
> runif(3)  # Simulate another three numbers
> set.seed(1121)  # Reset random number generator
> runif(3)  # Simulate another three numbers
> # Simulate random number from standard normal distribution
> rnorm(1)
> # Simulate five standard normal random numbers
> rnorm(5)
> # Simulate five non-standard normal random numbers
> rnorm(n=5, mean=1, sd=2)  # Match arguments by name
> # Simulate t-distribution with 2 degrees of freedom
> rt(n=5, df=2)
```

The Logistic Map

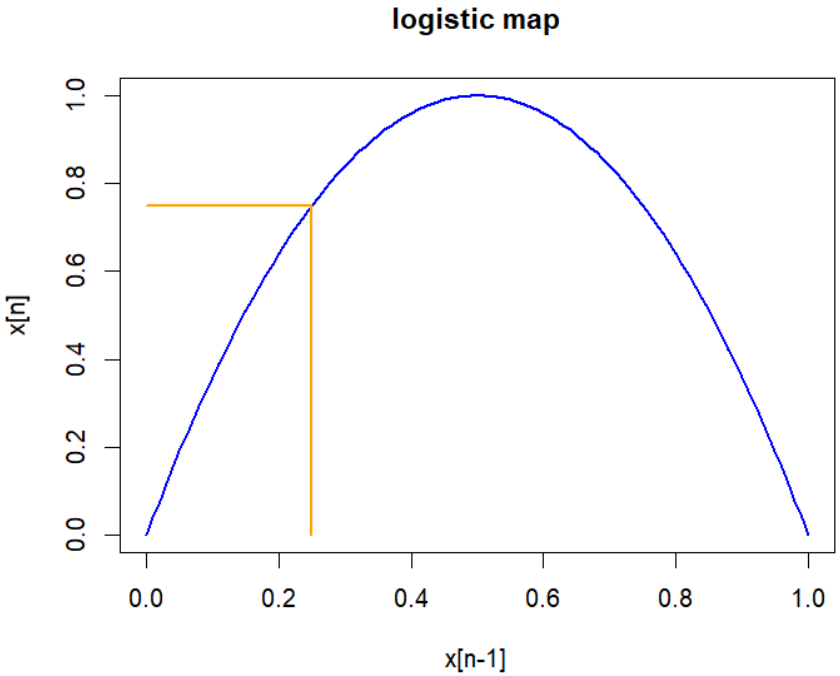
The *logistic map* is a recurrence relation which produces a deterministic sequence of numbers:

$$x_n = r x_{n-1}(1 - x_{n-1})$$

If the *seed* value x_0 is in the interval $(0, 1)$ and if $r = 4$, then the sequence x_n is also contained in the interval $(0, 1)$.

The function `curve()` plots a function defined by its name.

```
> # Define logistic map function
> log_map <- function(x, r=4) r*x*(1-x)
> log_map(0.25, 4)
> # Plot logistic map
> x11(width=6, height=5)
> curve(expr=log_map, type="l", xlim=c(0, 1),
+ xlab="x[n-1]", ylab="x[n]", lwd=2, col="blue",
+ main="logistic map")
> lines(x=c(0, 0.25), y=c(0.75, 0.75), lwd=2, col="orange")
> lines(x=c(0.25, 0.25), y=c(0, 0.75), lwd=2, col="orange")
```



Generating Pseudo-Random Numbers Using Logistic Map

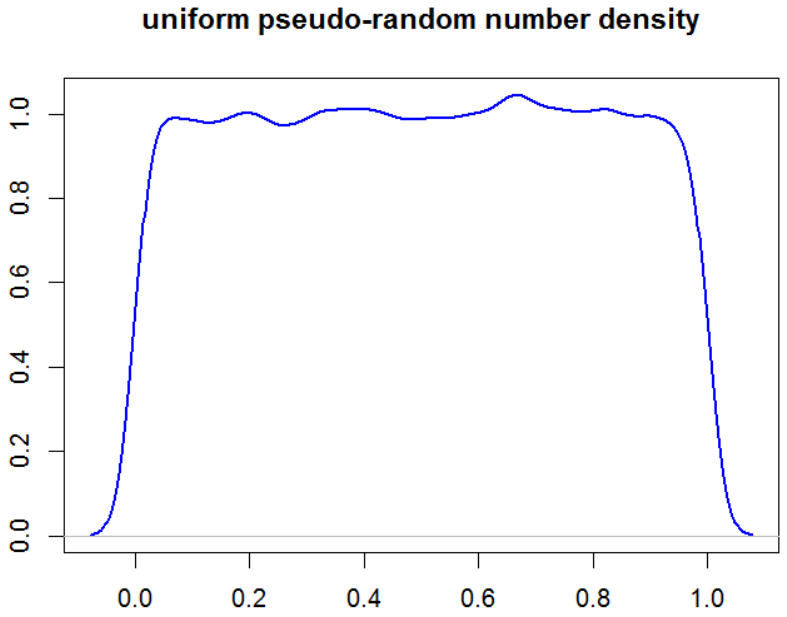
The *logistic map* can be used to calculate sequences of pseudo-random numbers.

For most *seed* values x_0 and $r = 4$, the *logistic map* produces a pseudo-random sequence, but it's not uniformly distributed.

The inverse cosine function `acos()` transforms a *logistic map* sequence into a uniformly distributed sequence,

$$u_n = \arccos(1 - 2x_n)/\pi$$

```
> # Calculate uniformly distributed pseudo-random sequence
> # using logistic map function.
> unifun <- function(seedv, n=10) {
+   # Pre-allocate vector instead of "growing" it
+   output <- numeric(n)
+   # initialize
+   output[1] <- seedv
+   # Perform loop
+   for (i in 2:n) {
+     output[i] <- 4*output[i-1]*(1-output[i-1])
+   }  # end for
+   acos(1-2*output)/pi
+ }  # end unifun
```




```
> unifun(seedv=0.1, n=15)
> plot(
+   density(unifun(seedv=runif(1), n=1e5)),
+   xlab="", ylab="", lwd=2, col="blue",
+   main="uniform pseudo-random number density")
```

Generating Binomial Random Numbers

A *binomial* trial is a coin flip, that results in either a success or failure.

The *binomial* distribution specifies the probability of obtaining a certain number of successes in a sequence of independent *binomial* trials.

Let p be the probability of obtaining a success in a *binomial* trial, and let $(1 - p)$ be the probability of failure.

$p = 0.5$ corresponds to flipping an unbiased coin.

The probability of obtaining k successes in n independent *binomial* trials is equal to:

$$\binom{n}{k} p^k (1-p)^{(n-k)}$$

The function `rbinom()` produces random numbers from the *binomial* distribution.

```
> set.seed(1121)  # Reset random number generator
> # Flip unbiased coin once, 20 times
> rbinom(n=20, size=1, 0.5)
> # Number of heads after flipping twice, 20 times
> rbinom(n=20, size=2, 0.5)
> # Number of heads after flipping thrice, 20 times
> rbinom(n=20, size=3, 0.5)
> # Number of heads after flipping biased coin thrice, 20 times
> rbinom(n=20, size=3, 0.8)
> # Number of heads after flipping biased coin thrice, 20 times
> rbinom(n=20, size=3, 0.2)
> # Flip unbiased coin once, 20 times
> sample(x=0:1, size=20, replace=TRUE)  # Fast
> as.numeric(runif(20) < 0.5)  # Slower
```

Generating Random Samples and Permutations

A *sample* is a subset of elements taken from a set of data elements.

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

By default the *size* of the sample (the `size` argument) is equal to the number of elements in the data vector.

So the call `sample(da_ta)` produces a random permutation of all the elements of `da_ta`.

The function `sample()` with `replace=TRUE` selects samples with replacement (the default is `replace=FALSE`).

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

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

```
> # Permutation of five numbers
> sample(x=5)
> # Permutation of four strings
> sample(x=c("apple", "grape", "orange", "peach"))
> # Sample of size three
> sample(x=5, size=3)
> # Sample with replacement
> sample(x=5, replace=TRUE)
> sample(  # Sample of strings
+   x=c("apple", "grape", "orange", "peach"),
+   size=12,
+   replace=TRUE)
> # Binomial sample: flip coin once, 20 times
> sample(x=0:1, size=20, replace=TRUE)
> # Flip unbiased coin once, 20 times
> as.numeric(runif(20) > 0.5)  # Slower
```

Monte Carlo Simulation

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

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

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

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

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

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

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

Standard Errors of Estimators Using Bootstrap Simulation

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

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

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

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

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

The Distribution of Estimators Using Bootstrap Simulation

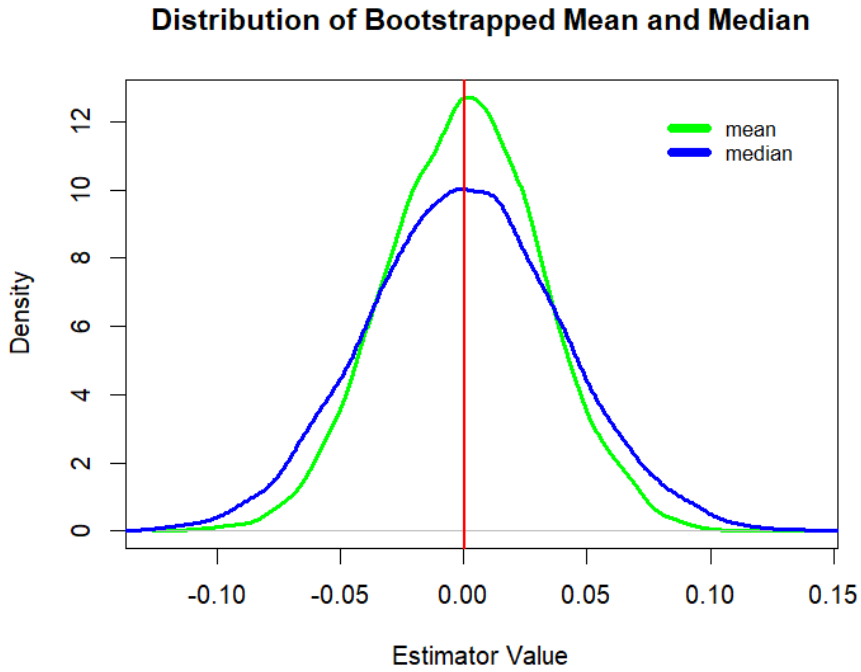
The standard errors of estimators can be calculated using a *bootstrap* simulation.

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

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

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

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




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

Bootstrapping Using Vectorized Operations

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

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

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

Faster code often requires more memory than slower code.

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

Bootstrapping Standard Errors Using Parallel Computing

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

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

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

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

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

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

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

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

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

Parallel Bootstrapping of the *Median Absolute Deviation*

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

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

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

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

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

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

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

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

Resampling From Empirical Datasets

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

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

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

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

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

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

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

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

Bootstrapping From Empirical Datasets

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

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

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

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

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

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

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

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

Standard Errors of Regression Coefficients Using Bootstrap

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

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

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

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

Distribution of Bootstrapped Regression Coefficients

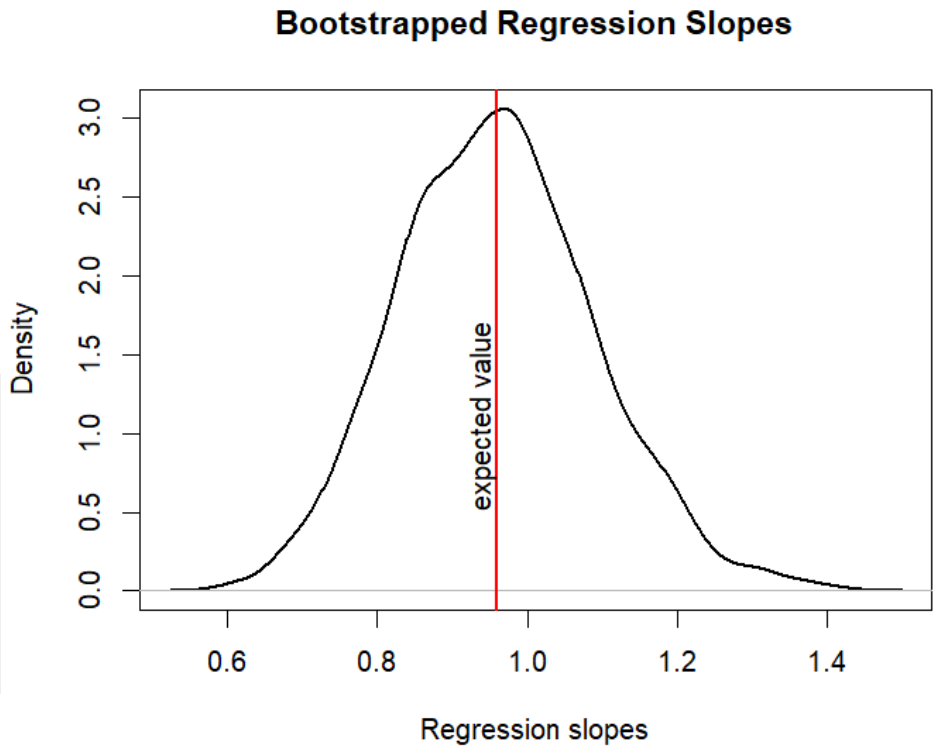
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

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

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

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

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



Bootstrapping Regressions Using Parallel Computing

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

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

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

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

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

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

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

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

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

Analyzing the Bootstrap Data

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

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

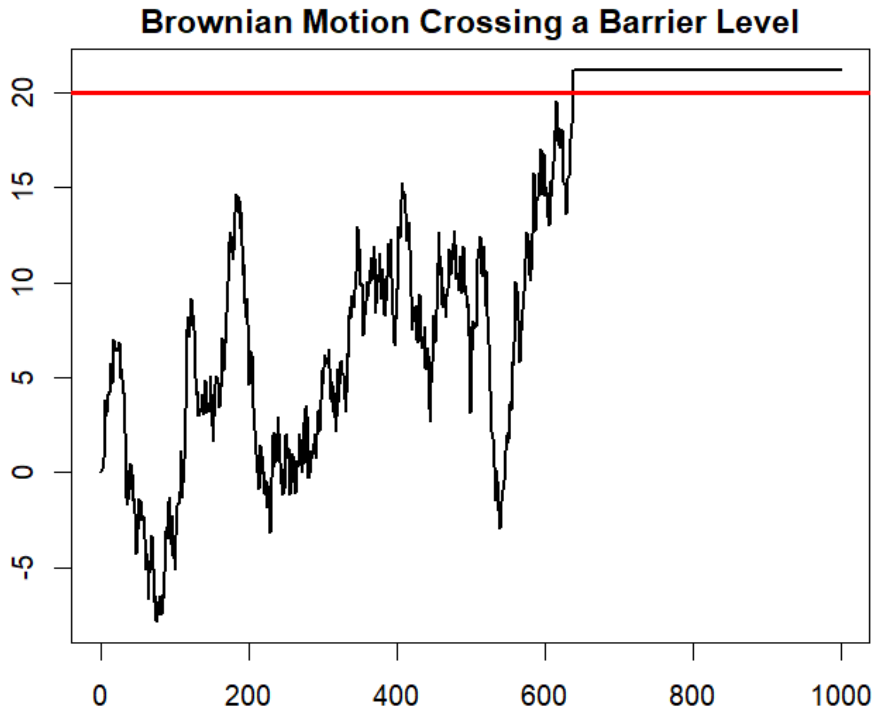
```
> # Collapse the bootstrap list into a vector
> class(bootd)
> bootd <- unlist(bootd)
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd),
+      lwd=2, xlab="Regression slopes",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

Simulating Brownian Motion Using while() Loops

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

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

```
> set.seed(1121)  # Reset random number generator
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> pathv <- numeric(nrows)  # Allocate path vector
> pathv[1] <- 0  # Initialize path
> indeks <- 2  # Initialize simulation index
> while ((indeks <= nrows) && (pathv[indeks - 1] < barl)) {
+ # Simulate next step
+   pathv[indeks] <- pathv[indeks - 1] + rnorm(1)
+   indeks <- indeks + 1  # Advance indeks
+ }  # end while
> # Fill remaining paths after it crosses barl
> if (indeks <= nrows)
+   pathv[indeks:nrows] <- pathv[indeks - 1]
> # Plot the Brownian motion
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(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)
```

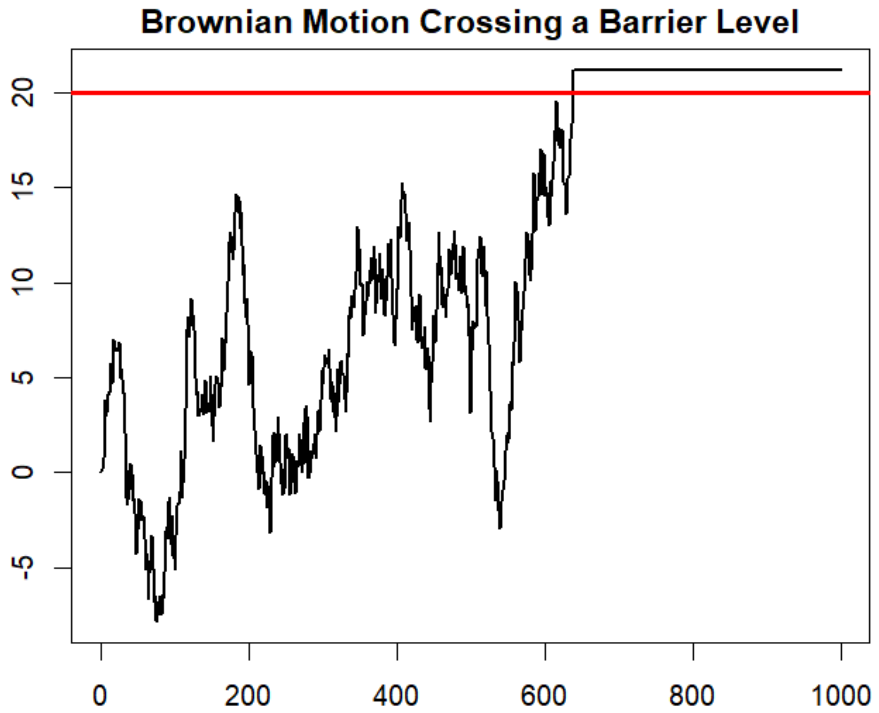


Simulating Brownian Motion Using Vectorized Functions

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.

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



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

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

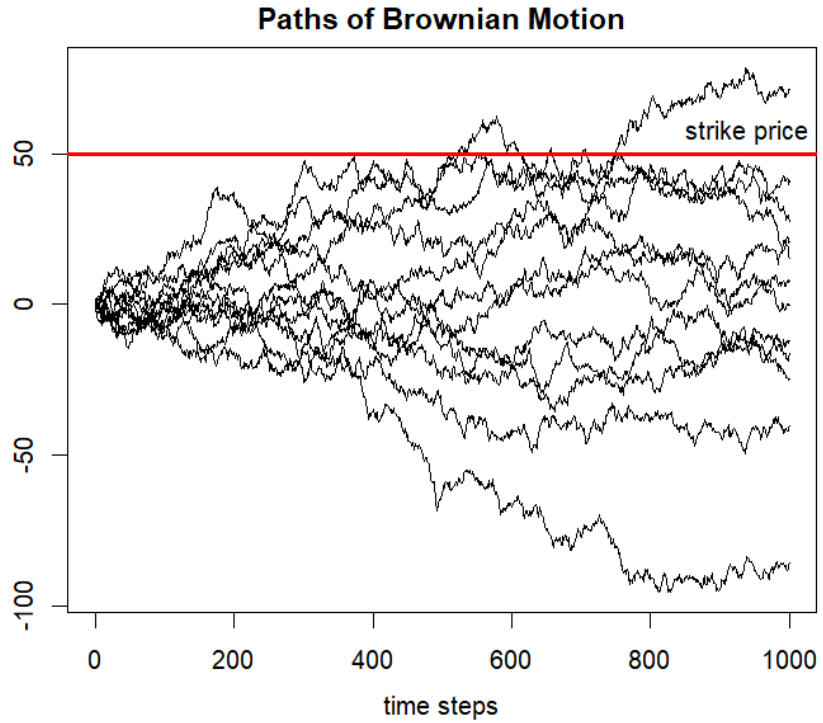
Estimating the Statistics of Brownian Motion

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

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

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

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




```
> # Plot in window
> x11(width=6, height=5)
> par(mar=c(4, 3, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2.5, 1, 0))
> # Select and plot full range of paths
> ordern <- order(pathm[nrows, ])
> pathm[nrows, 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=(nrows-60), y=strikep, labels="strike price", pos=3, cex=1)
```

Bootstrapping From Time Series of Prices

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

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

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

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

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

Variance Reduction Using Antithetic Sampling

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

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

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

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

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

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

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

```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> nrows <- 1000
> datav <- rnorm(nrows)
> # Estimate the 95% quantile
> nboot <- 10000
> bootd <- sapply(1:nboot, function(x) {
+   samplev <- datav[sample.int(nrows, 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(nrows, 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:

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

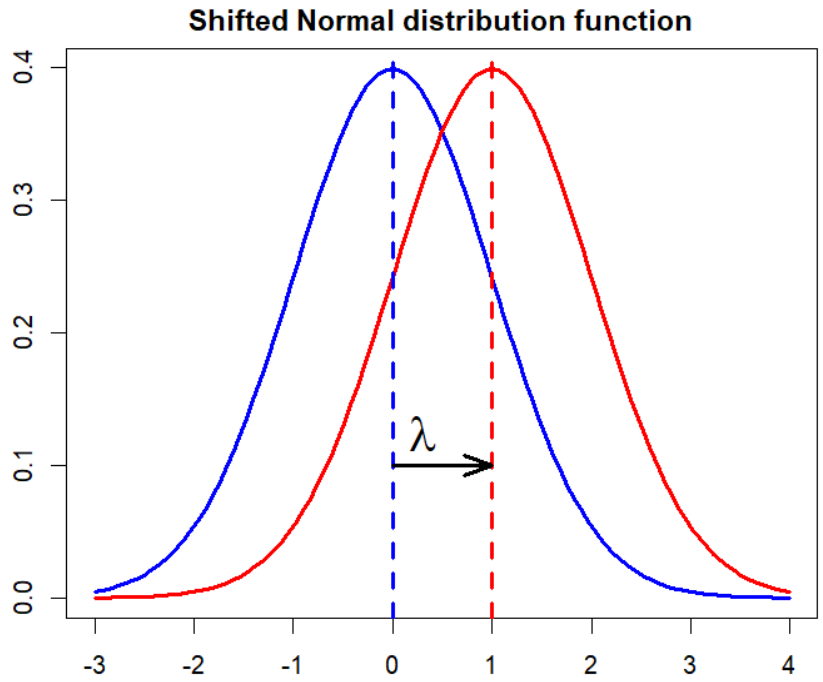
Where $p(x)$ is the probability density, $p(x,\lambda)$ is the tilted density, and λ is the tilt parameter.

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
> nrows <- 1000
> datav <- rnorm(nrows)
> # Cumulative probability from formula
> quantilev <- (-2)
> pnorm(quantilev)
> integrate(dnorm, lower=-Inf, upper=quantilev)
> # Cumulative probability from Naive Monte Carlo
> sum(datav < quantilev)/nrows
> # Generate importance sample
> lambda <- (-1.5)  # Tilt parameter
> data_tilt <- datav + lambda  # Tilt the random numbers
> # Cumulative probability from importance sample
> sum(data_tilt < quantilev)/nrows
> weights <- exp(-lambda*data_tilt + lambda^2/2)
> sum((data_tilt < quantilev)*weights)/nrows
> # Bootstrap of standard errors of cumulative probability
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   datav <- rnorm(nrows)
+   naivemc <- sum(datav < quantilev)/nrows
+   datav <- (datav + lambda)
+   weights <- exp(-lambda*datav + lambda^2/2)
+   isample <- sum((datav < quantilev)*weights)/nrows
+   c(naivemc=naivemc, importmc=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)
> cutoff <- nrows*confl
> datav[cutoff]  # Naive Monte Carlo value
> # Importance sample weights
> data_tilt <- datav + lambda  # Tilt the random numbers
> weights <- exp(-lambda*data_tilt + lambda^2/2)
> # Cumulative probabilities using importance sample
> cumprob <- cumsum(weights)/nrows
> # Quantile from importance sample
> data_tilt[findInterval(confl, cumprob)]
> # Bootstrap of standard errors of quantile
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   datav <- sort(rnorm(nrows))
+   naivemc <- datav[cutoff]
+   data_tilt <- datav + lambda
+   weights <- exp(-lambda*data_tilt + lambda^2/2)
+   cumprob <- cumsum(weights)/nrows
+   isample <- data_tilt[findInterval(confl, cumprob)]
+   c(naivemc=naivemc, importmc=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 <- data_tilt[findInterval(confl, cumprob)]
> sum((data_tilt < varisk)*data_tilt*weights)/sum((data_tilt < varis
> # CVaR from integration
> integrate(function(x) x*dnorm(x), low=-Inf, up=varisk)$value/pnorm
> # Bootstrap of standard errors of expected value
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   datav <- sort(rnorm(nrows))
+   varisk <- datav[cutoff]
+   naivemc <- sum((datav < varisk)*datav)/sum((datav < varisk))
+   data_tilt <- datav + lambda
+   weights <- exp(-lambda*data_tilt + lambda^2/2)
+   cumprob <- cumsum(weights)/nrows
+   varisk <- data_tilt[findInterval(confl, cumprob)]
+   isample <- sum((data_tilt < varisk)*data_tilt*weights)/sum((data
+   c(naivemc=naivemc, importmc=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.

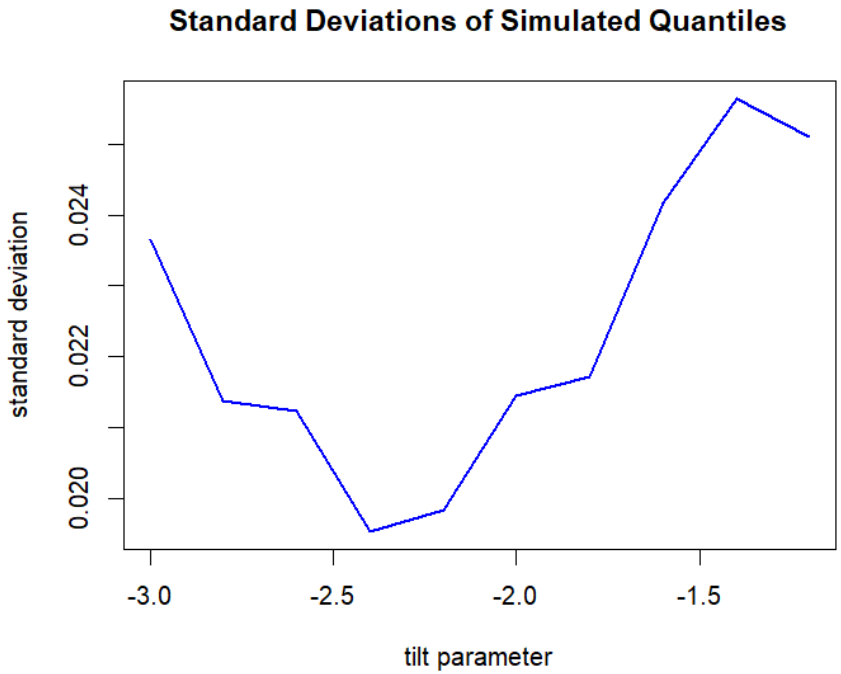
The optimal tilt parameter depends on the estimator and on the required confidence level.

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

```
> # Calculate matrix of random data
> set.seed(1121) # Reset random number generator
> nrows <- 1000; nboot <- 100
> datav <- matrix(rnorm(nboot*nrows), ncol=nboot)
> datav <- Rfast::sort_mat(datav)  # Sort the columns
> # Calculate vector of quantiles for tilt parameter
> confl <- 0.02; cutoff <- confl*nrows
> calc_quant <- function(lambda) {
+   data_tilt <- datav + lambda  # Tilt the random numbers
+   weights <- exp(-lambda*data_tilt + lambda^2/2)
+   # Calculate quantiles for columns
+   sapply(1:nboot, function(boo_t) {
+     cumprob <- cumsum(weights[, boo_t])/nrows
+     data_tilt[findInterval(confl, cumprob), boo_t]
+   })  # end sapply
+ }  # end calc_quant
```




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

Importance Sampling for Binomial Variables

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

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

Where λ is the tilt parameter.

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

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

```
> # Binomial sample
> nrows <- 1000
> probv <- 0.1
> datav <- rbinom(n=nrows, size=1, probv)
> head(datav, 33)
> fre_q <- sum(datav)/nrows
> # Tilted binomial sample
> lambda <- 5
> p_tilted <- lambda*probv/(1 + probv*(lambda - 1))
> weigh_t <- (1 + probv*(lambda - 1))/lambda
> datav <- rbinom(n=nrows, size=1, p_tilted)
> head(datav, 33)
> weigh_t*sum(datav)/nrows
> # Bootstrap of standard errors
> nboot <- 1000
> bootd <- sapply(1:nboot, function(x) {
+   c(naivemc=sum(rbinom(n=nrows, size=1, probv))/nrows,
+     importmc=weigh_t*sum(rbinom(n=nrows, size=1, p_tilted))/nrows)
+ }) # 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
> nrows <- 100  # Number of simulation steps
> nsimu <- 10000  # Number of simulations
> # Calculate matrix of normal variables
> set.seed(1121)
> datav <- rnorm(nsimu*nrows, mean=drift, sd=sigmav)
> datav <- matrix(datav, nc=nsimu)
> # Simulate paths of Brownian motion
> pathm <- matrixStats::colCumsums(datav)
> # Tilt the datav
> lambda <- 0.04  # Tilt parameter
> data_tilt <- datav + lambda  # Tilt the random numbers
> paths_tilt <- matrixStats::colCumsums(data_tilt)
> # Calculate path weights
> weights <- exp(-lambda*data_tilt + lambda^2/2)
> path_weights <- matrixStats::colProds(weights)
> # Or
> path_weights <- exp(-lambda*colSums(data_tilt) + nrows*lambda^2/2)
> # Calculate option payout using standard MC
> strikep <- 10  # Strike price
> payouts <- (pathm[nrows, ] - strikep)
> sum(payouts[payouts > 0])/nsimu
> # Calculate option payout using importance sampling
> payouts <- (paths_tilt[nrows, ] - strikep)
> sum((path_weights*payouts)[payouts > 0])/nsimu
> # Calculate crossing probability using standard MC
> barl <- 10
> crossi <- (colSums(pathm > barl) > 0)
> sum(crossi)/nsimu
> # Calculate crossing probability using importance sampling
> crossi <- colSums(paths_tilt > barl) > 0
> sum(path_weights*crossi)/nsimu
```

Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture2.pdf*, and run all the code in *FRE6871_Lecture2.R*,
- Study *bootstrap simulation* from the files *bootstrap_technique.pdf* and *doBootstrap_primer.pdf*,
- Study the *Vasicek* single factor model from *Vasicek Portfolio Default Distribution.pdf*,
- Study credit portfolio risk models from `BOE Credit Risk Models.pdf` and `BIS Bank Capital Model.pdf`,
- Study CDO models from `Elizalde CDO Vasicek Credit Model.pdf`,
- Study the *CVAR* credit portfolio risk measure from *Danielsson CVAR Estimation Standard Error.pdf*.

Recommended

- Read about plotting from *plot par cheatsheet.pdf* and *ggplot2 cheatsheet.pdf*.