FRE6871 R in Finance

Lecture#2, Spring 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 1, 2024

Determining the Memory Usage of R Objects

The function `object.size()` displays the amount of memory (in *bytes*) allocated to R objects.

The generic function `format()` formats R objects for printing and display.

The method `format.object_size()` defines a *megabyte* as `1,048,576` *bytes* (2^{20}), not `1,000,000` *bytes*.

The function `get()` accepts a character string and returns the value of the corresponding object in a specified *environment*.

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

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects.

The function `ll()` from package `gdata` displays the amount of memory (in *bytes*) allocated to R objects.

```
> # Get size of an object
> vecv <- runif(1e6)
> object.size(vecv)
> format(object.size(vecv), units="MB")
> # Get sizes of objects in workspace
> sort(sapply(ls(), function(namev) {
+   format(object.size(get(namev)), units="KB")}))
> # Get sizes of all objects in workspace
> sort(sapply(mget(ls()), object.size))
> sort(sapply(mget(ls()), function(objectv) {
+   format(object.size(objectv), units="KB")}
+ ))
> # Get total size of all objects in workspace
> format(object.size(x=mget(ls())), units="MB")
> # Get sizes of objects in rutils::etfenv environment
> sort(sapply(ls(rutils::etfenv), function(namev) {
+   object.size(get(namev, rutils::etfenv))}))
> sort(sapply(mget(ls(rutils::etfenv), rutils::etfenv),
+       object.size))
> library(gdata)  # Load package gdata
> # Get size of data frame columns
> gdata::ll(unit="bytes", mtcars)
> # Get namev, class, and size of objects in workspace
> objframe <- gdata::ll(unit="bytes")
> # Sort by memory size (descending)
> objframe[order(objframe[, 2], decreasing=TRUE), ]
> gdata::ll()[order(ll()$KB, decreasing=TRUE), ]
> # Get sizes of objects in etfenv environment
> gdata::ll(unit="bytes", etfenv)
```

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, with `apply()` the fastest, then `vapply()`, `lapply()` and `sapply()` slightly slower, and `for()` loops the slowest.

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

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

```
> # Calculate matrix of random data with 5,000 rows
> matv <- matrix(rnorm(10000), ncol=2)
> # Allocate memory for row sums
> rowsumv <- numeric(NROW(matv))
> summary(microbenchmark(
+   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
```

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

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

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

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

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

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

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

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

Vectorized Functions

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

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

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

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

```
> # Define function vectorized automatically
> 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:

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

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

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

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

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

Vectorizing Functions Using mapply()

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

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

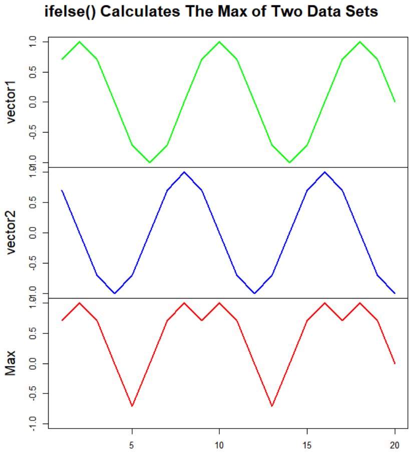
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2,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

Vectorized if-else Statements Using Function ifelse()

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

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

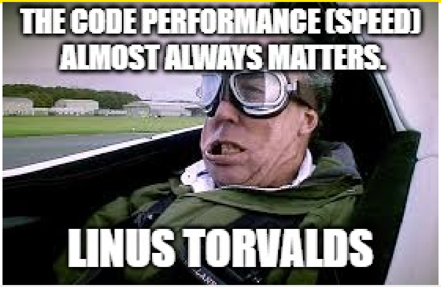
```
> # Create two numeric vectors
> vec1 <- sin(0.25*pi*1:20)
> vec2 <- cos(0.25*pi*1:20)
> # Create third vector using 'ifelse'
> vec3 <- ifelse(vec1 > vec2, vec1, vec2)
> # cbind all three together
> vec3 <- cbind(vec1, vec2, vec3)
> colnames(vec3)[3] <- "Max"
> # Set plotting parameters
> x11(width=6,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



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

How to write fast R code:

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



```
> # Calculate cumulative sum of a vector
> vecv <- runif(1e5)
> # Use compiled function
> cumsumv <- cumsum(vecv)
> # Use for loop
> cumsumv2 <- vecv
> for (i in 2:NROW(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.

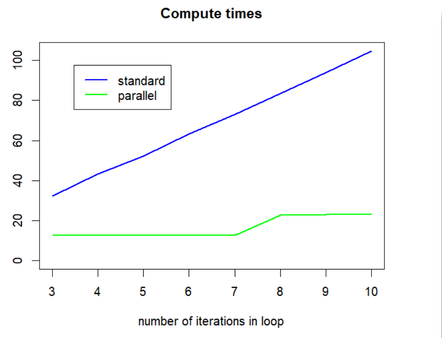
The function stopCluster() stops the R processes running on several CPU cores.

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 used to calculate different parameters of the probability distribution (moments, quantiles, etc.), and its functionals.

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

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

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

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

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

Standard Errors of Estimators Using Bootstrap Simulation

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

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

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

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

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

The Distribution of Estimators Using Bootstrap Simulation

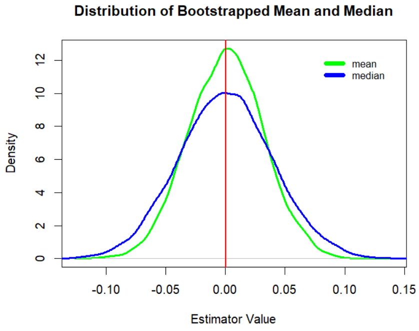
The standard errors of estimators can be calculated using a *bootstrap* simulation.

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

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

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

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



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

Bootstrapping Using Vectorized Operations

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

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

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

Faster code often requires more memory than slower code.

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

Bootstrapping Standard Errors Using Parallel Computing

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

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

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

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

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

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

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

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

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

Parallel Bootstrapping of the *Median Absolute Deviation*

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

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

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)
```

Resampling From Empirical Datasets

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

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

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

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

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

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

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

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

Bootstrapping From Empirical Datasets

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

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

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

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

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

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

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

```
> # Sample from time series of VTI returns
> library(rutils)
> retp <- rutils::etfenv$returns$VTI
> retp <- na.omit(retp)
> nrows <- NROW(retp)
> # Bootstrap sd and MAD under Windows
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster und
> clusterSetRNGStream(compclust, 1121)  # Reset random number genera
> nboot <- 10000
> bootd <- parLapply(compclust, 1:nboot, function(x, retp, nsimu) {
+   samplev <- retp[sample.int(nsimu, replace=TRUE)]
+   c(sd=sd(samplev), mad=mad(samplev))
+ }, retp=retp, nsimu=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 Win
> bootd <- rutils::do_call(rbind, bootd)
> # Standard error of standard deviation assuming normal distributio
> sd(retp)/sqrt(nrows)
> # Means and standard errors from bootstrap
> stderrors <- apply(bootd, MARGIN=2,
+   function(x) c(mean=mean(x), stderror=sd(x)))
> stderrors
> # Relative standard errors
> stderrors[2, ]/stderrors[1, ]
```

Standard Errors of Regression Coefficients Using Bootstrap

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

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

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

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

Distribution of Bootstrapped Regression Coefficients

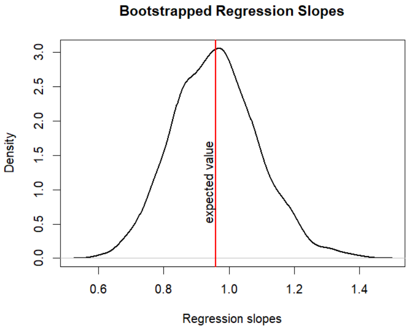
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

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

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

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

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



Bootstrapping Regressions Using Parallel Computing

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

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

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

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

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

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

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

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

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster und
> # Bootstrap of regression under Windows
> bootd <- parLapply(compclust, 1:1000, function(x, desm) {
+   samplev <- sample.int(nsimu, replace=TRUE)
+   desm <- desm[samplev, ]
+   cov(desm[, 1], desm[, 2])/var(desm[, 2])
+ }, design=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 Wind
```

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).

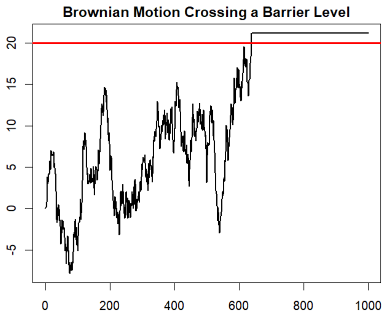
```
> # Collapse the bootstrap list into a vector
> class(bootd)
> bootd <- unlist(bootd)
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd),
+      lwd=2, xlab="Regression slopes",
+      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
> nsimu <- 1000  # Number of simulation steps
> pathv <- numeric(nsimu)  # Allocate path vector
> pathv[1] <- rnorm(1)  # Initialize path
> it <- 2  # Initialize simulation index
> while ((it <= nsimu) && (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 <= nsimu)
+   pathv[it:nsimu] <- 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)
```

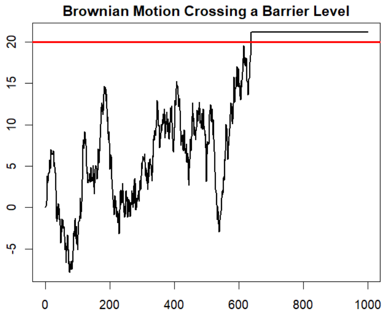


Simulating Brownian Motion Using Vectorized Functions

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
> nsimu <- 1000  # Number of simulation steps
> # Simulate path of Brownian motion
> pathv <- cumsum(rnorm(nsimu))
> # 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):nsimu] <- 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,

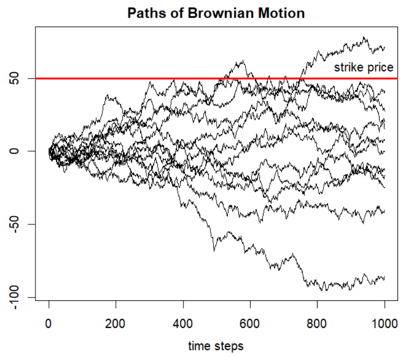
Estimating the Statistics of Brownian Motion

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
> nsimu <- 100  # Number of simulation paths
> # Simulate multiple paths of Brownian motion
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> pathm <- rnorm(nsimu*nsteps, mean=drift, sd=sigmav)
> pathm <- matrix(pathm, nc=nsimu)
> 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])/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[nsteps, ])
> pathm[nsteps, ordern]
> indeks <- ordern[seq(1, 100, 9)]
> zoo::plot.zoo(pathm[, indeks], main="Paths of Brownian Motion",
+   xlab="time steps", ylab=NA, plot.type="single")
> abline(h=strikep, col="red", lwd=3)
> text(x=(nsteps-60), y=strikep, labels="strike price", pos=3, cex=
```

Bootstrapping From Time Series of Prices

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

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

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

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

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

Bootstrapping From *OHLC* Prices

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

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

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

```
> # Calculate percentage returns from VTI prices
> library(rutils)
> ohlc <- rutils::etfenv$VTI
> pricev <- as.numeric(ohlc[, 4])
> startd <- 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 <- startd*exp(cumsum(retp[datav]))
> ohlcboot <- pricediff + priceboot
> ohlcboot <- cbind(ohlcboot, priceboot)
> # Define barrier level with respect to prices
> barl <- 1.5*max(pricev)
> # Calculate if High bootstrapped prices crossed barrier level
> sum(ohlcboot[, 2] > barl) > 0
```

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

Variance Reduction Using Antithetic Sampling

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

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

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

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

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

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

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

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

Simulating Rare Events Using Probability Tilting

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

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

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

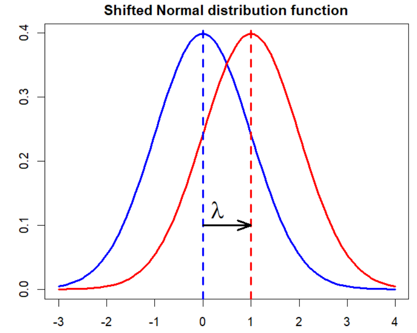
Where $p(x)$ is the probability density, $p(x, \lambda)$ is the tilted density, and λ is the tilt parameter.

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

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

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

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
> lambda <- (-1.5)  # Tilt parameter
> datat <- datav + lambda  # Tilt the random numbers
> # Cumulative probability from importance sample - wrong!
> sum(datat < quantv)/nsimu
> # Cumulative probability from importance sample - correct
> weightv <- exp(-lambda*datat + lambda^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 + lambda)
+   weightv <- exp(-lambda*datav + lambda^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 + lambda  # Tilt the random numbers
> weightv <- exp(-lambda*datat + lambda^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 + lambda
+   weightv <- exp(-lambda*datat + lambda^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 + lambda
+   weightv <- exp(-lambda*datat + lambda^2/2)
+   cumprob <- cumsum(weightv)/nsimu
+   varisk <- datat[findInterval(confl, cumprob)]
+   isample <- sum((datat <= varisk)*datat*weightv)/sum((datat <= va
+   c(naivemc=naivemc, impsample=isample)
+ }) # end sapply
> apply(bootd, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
```

The Optimal Tilt Parameter for Importance Sampling

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

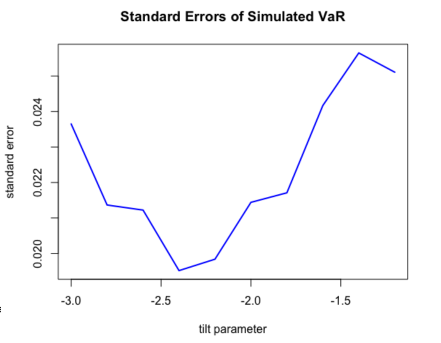
The optimal tilt parameter depends on the estimator and on the required confidence level.

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

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

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

```
> # Calculate matrix of random data
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection") # Rese
> nsimu <- 1000; nboot <- 100
> datav <- matrix(rnorm(nboot*nsimu), ncol=nboot)
> datav <- Rfast::colSort(datav)  # Sort the columns
> # Bootstrap function for VaR (quantile) for a single tilt parame
> calc_vars <- function(lambda, confl=0.05) {
+   datat <- datav + lambda  # Tilt the random numbers
+   weightv <- exp(-lambda*datat + lambda^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 of Brownian Motion

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

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

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

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

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

```
> # Define Brownian motion parameters
> sigmav <- 1.0  # Volatility
> drift <- 0.0  # Drift
> nsteps <- 100  # Number of simulation steps
> nsimu <- 10000  # Number of simulation paths
> # Calculate matrix of normal variables
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> datav <- rnorm(nsimu*nsteps, mean=drift, sd=sigmav)
> datav <- matrix(datav, nc=nsimu)
> # Simulate paths of Brownian motion
> pathm <- matrixStats::colCumsums(datav)
> # Tilt the datav
> lambda <- 0.04  # Tilt parameter
> datat <- datav + lambda  # Tilt the random numbers
> patht <- matrixStats::colCumsums(datat)
> # Calculate path weights
> weightm <- exp(-lambda*datat + lambda^2/2)
> weightm <- matrixStats::colProds(weightm)
> # Or
> weightm <- exp(-lambda*colSums(datat) + nsteps*lambda^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
```

Simulating Single-period Defaults

Consider a portfolio of credit assets (bonds or loans) over a single period of time.

At the end of the period, some of the assets default, while the rest don't.

The default probabilities are equal to p_i.

Individual defaults can be simulated by comparing the probabilities p_i with the uniform random numbers u_i.

Default occurs if u_i is less than the default probability p_i:

$$u_i < p_i$$

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) `for()` loops.

```
> # Calculate random default probabilities
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nbonds <- 100
> probv <- runif(nbonds, max=0.2)
> mean(probv)
> # Simulate number of defaults
> unifv <- runif(nbonds)
> sum(unifv < probv)
> # Simulate average number of defaults using for() loop (inefficie
> nsimu <- 1000
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> defaultv <- numeric(nsimu)
> for (i in 1:nsimu) {  # Perform loop
+   unifv <- runif(nbonds)
+   defaultv[i] <- sum(unifv < probv)
+ }  # end for
> # Calculate average number of defaults
> mean(defaultv)
> # Simulate using vectorized functions (efficient way)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> unifm <- matrix(runif(nsimu*nbonds), ncol=nsimu)
> defaultv <- colSums(unifm < probv)
> mean(defaultv)
> # Plot the distribution of defaults
> x11(width=6, height=5)
> plot(density(defaultv), main="Distribution of Defaults",
+      xlab="number of defaults", ylab="frequency")
> abline(v=mean(defaultv), lwd=3, col="red")
```

Asset Values and Default Thresholds

Defaults can also be simulated using normally distributed variables a_i called *asset values*, instead of the uniformly distributed variables u_i.

The asset values a_i are the *quantiles* corresponding to the uniform variables u_i: $a_i = \Phi^{-1}(u_i)$ (where $\Phi()$ is the cumulative *Standard Normal* distribution).

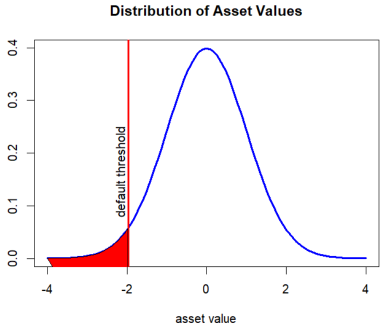
Similarly, the default probabilities p_i are also transformed into *default thresholds* t_i, which are the *quantiles*: $t_i = \Phi^{-1}(p_i)$.

Before, default occurred if u_i was less than the default probability p_i: $u_i < p_i$.

Now, default occurs if the *asset value* a_i is less than the *default threshold* t_i: $a_i < t_i$.

The asset values a_i are mathematical variables which can be negative, so they are not actual company asset values.

```
> # Calculate default thresholds and asset values
> threshv <- qnorm(probv)
> assetm <-qnorm(unifm)
> # Simulate defaults
> defaultv <- colSums(assetm < threshv)
> mean(defaultv)
```



```
> # Plot Standard Normal distribution
> x11(width=6, height=5)
> xlim <- 4; threshv <- qnorm(0.025)
> curve(expr=dnorm(x), type="l", xlim=c(-xlim, xlim),
+ xlab="asset value", ylab="", lwd=3,
+ col="blue", main="Distribution of Asset Values")
> abline(v=threshv, col="red", lwd=3)
> text(x=threshv-0.1, y=0.15, labels="default threshold",
+  lwd=2, srt=90, pos=3)
> # Plot polygon area
> xvar <- seq(-xlim, xlim, length=100)
> yvar <- dnorm(xvar)
> intail <- ((xvar >= (-xlim)) & (xvar <= threshv))
> polygon(c(xlim, xvar[intail], threshv),
+   c(-1, yvar[intail], -1), col="red")
```

Vasicek Model of Correlated Asset Values

So far, the asset values are independent from each other, but in reality default events are correlated.

The *Vasicek* model introduces correlation between the asset values a_i.

Under the *Vasicek* single factor model, the asset value a_i is equal to the sum of a *systematic* factor s, plus an *idiosyncratic* factor z_i:

$$a_i = \sqrt{\rho}\, s + \sqrt{1 - \rho}\, z_i$$

Where ρ is the correlation between asset values.

The variables s, z_i, and a_i all follow the *Standard Normal* distribution $\phi(0, 1)$.

The matrix of asset values is arranged with columns corresponding to simulations and rows corresponding to bonds or loans.

The *Vasicek* model resembles the *CAPM* model, with the asset value equal to the sum of a *systematic* factor plus an *idiosyncratic* factor.

The Bank for International Settlements (BIS) uses the *Vasicek* model as part of its regulatory capital requirements for bank credit risk:
http://bis2information.org/content/Vasicek_model
https://www.bis.org/bcbs/basel3.htm
https://www.bis.org/bcbs/irbriskweight.pdf

```
> # Define correlation parameters
> rho <- 0.2
> rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> nbonds <- 5 ; nsimu <- 10000
> # Calculate vector of systematic and idiosyncratic factors
> sysv <- rnorm(nsimu)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> isync <- rnorm(nsimu*nbonds)
> dim(isync) <- c(nbonds, nsimu)
> # Simulate asset values using vectorized functions (efficient way)
> assetm <- t(rhos*sysv + t(rhosm*isync))
> # Asset values are standard normally distributed
> apply(assetm, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
> # Calculate correlations between asset values
> cor(t(assetm))
> # Simulate asset values using for() loop (inefficient way)
> # Allocate matrix of assets
> assetn <- matrix(nrow=nbonds, ncol=nsimu)
> # Simulate asset values using for() loop
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> for (i in 1:nsimu) {  # Perform loop
+   assetn[, i] <- rhos*sysv[i] + rhosm*rnorm(nbonds)
+ }  # end for
> all.equal(assetn, assetm)
> # benchmark the speed of the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   forloop={for (i in 1:nsimu) {
+     rhos*sysv[i] + rhosm*rnorm(nbonds)}},
+   vectorized={t(rhos*sysv + t(rhosm*isync))},
+   times=10))[, c(1, 4, 5)]
```

Vasicek Model of Correlated Defaults

Under the *Vasicek* model, default occurs if the *asset value* a_i is less than the *default threshold* t_i:

$$a_i = \sqrt{\rho} s + \sqrt{1 - \rho} z_i$$
$$a_i < t_i$$

The *systematic* factor s may be considered to represent the state of the macro economy, with positive values representing an economic expansion, and negative values representing an economic recession.

When the value of the *systematic* factor s is positive, then the asset values will all tend to be bigger as well, which will produce fewer defaults.

But when the *systematic* factor is negative, then the asset values will tend to be smaller, which will produce more defaults.

This way the *Vasicek* model introduces a correlation among defaults.

```
> # Calculate random default probabilities
> nbonds <- 5
> probv <- runif(nbonds, max=0.2)
> mean(probv)
> # Calculate default thresholds
> threshv <- qnorm(probv)
> # Calculate number of defaults using vectorized functions (efficie
> # Calculate vector of number of defaults
> rowMeans(assetm < threshv)
> probv
> # Calculate number of defaults using for() loop (inefficient way)
> # Allocate matrix of defaultm
> defaultm <- matrix(nrow=nbonds, ncol=nsimu)
> # Simulate asset values using for() loop
> for (i in 1:nsimu) {  # Perform loop
+   defaultm[, i] <- (assetm[, i] < threshv)
+ }  # end for
> rowMeans(defaultm)
> rowMeans(assetm < threshv)
> # Calculate correlations between defaults
> cor(t(defaultm))
```

Asset Correlation and Default Correlation

Default correlation is defined as the correlation between the Boolean vectors of default events.

The *Vasicek* model introduces correlation among default events, through the correlation of *asset values*.

If *asset values* have a positive correlation, then the defaults among credits are clustered together, and if one credit defaults then the other credits are more likely to default as well.

Empirical studies have found that the asset correlation ρ can vary between 5% to 20%, depending on the default risk.

Credits with higher default risk tend to also have higher asset correlation, since they are more sensitive to the economic conditions.

Default correlations are usually much lower than the corresponding asset correlations.

```
> # Define default probabilities
> nbonds <- 2
> defprob <- 0.2
> threshv <- qnorm(defprob)
> # Define correlation parameters
> rho <- 0.2
> rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> # Calculate vector of systematic factors
> nsimu <- 1000
> sysv <- rnorm(nsimu)
> isync <- rnorm(nsimu*nbonds)
> dim(isync) <- c(nbonds, nsimu)
> # Simulate asset values using vectorized functions
> assetm <- t(rhos*sysv + t(rhosm*isync))
> # Calculate number of defaults using vectorized functions
> defaultm <- (assetm < threshv)
> # Calculate average number of defaults and compare to defprob
> rowMeans(defaultm)
> defprob
> # Calculate correlations between assets
> cor(t(assetm))
> # Calculate correlations between defaults
> cor(t(defaultm))
```

Cumulative Defaults Under the Vasicek Model

A formula for the default distribution under the Vasicek Model can be derived under the assumption that the number of assets is very large and that they all have the same default probabilities $p_i = p$.

In that case the single default threshold is equal to $t = \Phi^{-1}(p)$.

If the systematic factor s is fixed, then the *asset value* a_i follows the *Normal* distribution with mean equal to $\sqrt{\rho} s$ and standard deviation equal to $\sqrt{1-\rho}$:

$$a_i = \sqrt{\rho} s + \sqrt{1-\rho} z_i$$

The conditional default probability $p(s)$, given the systematic factor s, is equal to:

$$p(s) = \Phi\left(\frac{t - \sqrt{\rho} s}{\sqrt{1-\rho}}\right)$$

Since the systematic factor s is fixed, then the defaults are all independent with the same default probability $p(s)$.

Because the number of assets is very large, the percentage x of the portfolio that defaults, is equal to the conditional default probability $x = p(s)$.

We can invert the formula $x = \Phi(\frac{t - \sqrt{\rho} s}{\sqrt{1-\rho}})$ to obtain the systematic factor s:

$$s = \frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}$$

Since the systematic factor s follows the *Standard Normal* distribution, then the portfolio cumulative default probability $P(x)$ is equal to:

$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}\right)$$

Cumulative Default Distribution And Correlation

The cumulative portfolio default probability $P(x)$:

$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}\right)$$

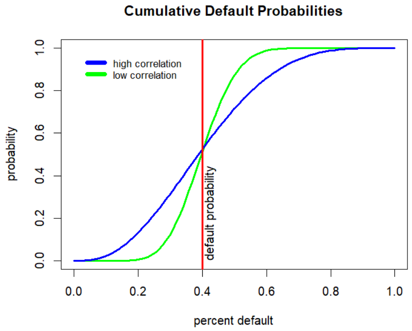
Depends on the correlation parameter ρ.

If the correlation ρ is very low (close to 0) then the percentage x of the portfolio defaults is always very close to the default probability p, and the cumulative default probability curve is steep close to the expected value of p.

If the correlation ρ is very high (close to 1) then the percentage x of the portfolio defaults has a very wide dispersion around the default probability p, and the cumulative default probability curve is flat close to the expected value of p.

This is because with high correlation, the assets will tend to all default together or not default.

```
> # Define cumulative default distribution function
> cumdefdistr <- function(x, threshv=(-2), rho=0.2)
+   pnorm((sqrt(1-rho)*qnorm(x) - threshv)/sqrt(rho))
> defprob <- 0.4; threshv <- qnorm(defprob)
> cumdefdistr(x=0.2, threshv=qnorm(defprob), rho=rho)
> # Plot cumulative default distribution function
> curve(expr=cumdefdistr(x, threshv=threshv, rho=0.05),
+ xlim=c(0, 0.999), lwd=3, xlab="percent default", ylab="probabili
+ col="green", main="Cumulative Default Probabilities")
```



```
> # Plot default distribution with higher correlation
> curve(expr=cumdefdistr(x, threshv=threshv, rho=0.2),
+     xlim=c(0, 0.999), add=TRUE, lwd=3, col="blue", main="")
> # Add legend
> legend(x="topleft",
+    legend=c("high correlation", "low correlation"),
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    bty="n", lwd=6, lty=1, col=c("blue", "green"))
> # Add unconditional default probability
> abline(v=defprob, col="red", lwd=3)
> text(x=defprob, y=0.0, labels="default probability",
+  lwd=2, srt=90, pos=4)
```

Distribution of Defaults Under the Vasicek Model

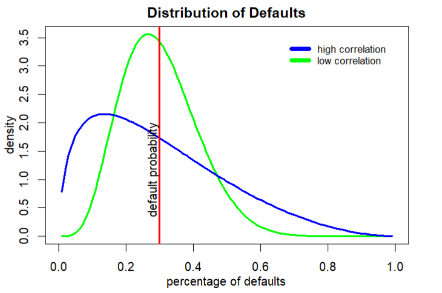
The probability density $f(x)$ of portfolio defaults is equal to the derivative of the cumulative default distribution $P(x)$:

$$f(x) = \frac{\sqrt{1-\rho}}{\sqrt{\rho}} \exp(-\frac{1}{2\rho}(\sqrt{1-\rho}\,\Phi^{-1}(x) - t)^2 + \frac{1}{2}\Phi^{-1}(x)^2)$$

If the correlation ρ is very low (close to 0) then the probability density $f(x)$ is centered around the default probability p.

If the correlation ρ is very high (close to 1) then the probability density $f(x)$ is wide, with significant probability of large portfolio defaults and also small portfolio defaults.

```
> # Define default probability density function
> defdistr <- function(x, threshv=(-2), rho=0.2)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnorm(x) -
+   threshv)^2/(2*rho) + qnorm(x)^2/2)
> # Define parameters
> rho <- 0.2 ; rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> defprob <- 0.3; threshv <- qnorm(defprob)
> defdistr(0.03, threshv=threshv, rho=rho)
> # Plot probability distribution of defaults
> curve(expr=defdistr(x, threshv=threshv, rho=0.1),
+ xlim=c(0, 1.0), lwd=3,
+ xlab="Default percentage", ylab="Density",
+ col="green", main="Distribution of Defaults")
```



```
> # Plot default distribution with higher correlation
> curve(expr=defdistr(x, threshv=threshv, rho=0.3),
+ xlab="default percentage", ylab="",
+ add=TRUE, lwd=3, col="blue", main="")
> # Add legend
> legend(x="topright",
+    legend=c("high correlation", "low correlation"),
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    bty="n", lwd=6, lty=1, col=c("blue", "green"))
> # Add unconditional default probability
> abline(v=defprob, col="red", lwd=3)
> text(x=defprob, y=2, labels="default probability",
+  lwd=2, srt=90, pos=2)
```

Distribution of Defaults Under Extreme Correlations

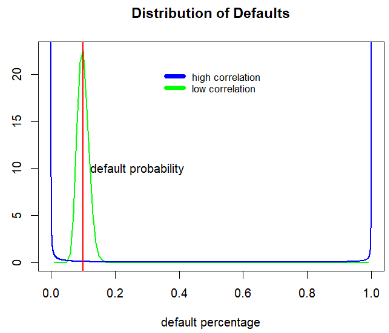
If the correlation ρ is close to 0, then the asset values a_i are independent from each other, and defaults are also independent, so that the percentage of portfolio defaults is very close to the default probability p.

In that case, the probability density of portfolio defaults is very narrow and is centered on the default probability p.

If the correlation ρ is close to 1, then the asset values a_i are almost the same, and defaults occur at the same time, so that the percentage of portfolio defaults is either 0 or 1.

In that case, the probability density of portfolio defaults becomes *bimodal*, with two peaks around *zero* and 1.

```
> # Plot default distribution with low correlation
> curve(expr=defdistr(x, threshv=threshv, rho=0.01),
+   xlab="default percentage", ylab="", lwd=2,
+   col="green", main="Distribution of Defaults")
> # Plot default distribution with high correlation
> curve(expr=defdistr(x, threshv=threshv, rho=0.99),
+   xlab="percentage of defaults", ylab="density",
+   add=TRUE, lwd=2, n=10001, col="blue", main="")
```



```
> # Add legend
> legend(x="top", legend=c("high correlation", "low correlation"),
+    title=NULL, inset=0.1, cex=1.0, bg="white",
+    bty="n", lwd=6, lty=1, col=c("blue", "green"))
> # Add unconditional default probability
> abline(v=0.1, col="red", lwd=2)
> text(x=0.1, y=10, lwd=2, pos=4, labels="default probability")
```

Numerical Integration of Functions

The function `integrate()` performs numerical integration of a function of a single variable, i.e. it calculates a definite integral over an integration interval.

Additional parameters can be passed to the integrated function through the dots `"..."` argument of the function `integrate()`.

The function `integrate()` accepts the integration limits `-Inf` and `Inf` equal to minus and plus infinity.

```
> # Get help for integrate()
> ?integrate
> # Calculate slowly converging integral
> func <- function(x) {1/((x+1)*sqrt(x))}
> integrate(func, lower=0, upper=10)
> integrate(func, lower=0, upper=Inf)
> # Integrate function with parameter lambda
> func <- function(x, lambda=1) {
+   exp(-x*lambda)
+ }  # end func
> integrate(func, lower=0, upper=Inf)
> integrate(func, lower=0, upper=Inf, lambda=2)
> # Cumulative probability over normal distribution
> pnorm(-2)
> integrate(dnorm, low=2, up=Inf)
> str(dnorm)
> pnorm(-1)
> integrate(dnorm, low=2, up=Inf, mean=1)
> # Expected value over normal distribution
> integrate(function(x) x*dnorm(x), low=2, up=Inf)
```

Portfolio Loss Distribution

The expected loss (EL) of a credit portfolio is equal to the sum of the default probabilities p_i multiplied by the loss given default LGD (aka the *loss severity* - equal to 1 minus the *recovery rate*):

$$EL = \sum_{i=1}^{n} p_i LGD_i$$

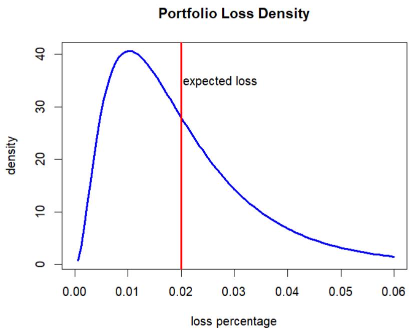
Then the *cumulative loss distribution* is equal to:

$$P(x) = \Phi(\frac{\sqrt{1-\rho}\,\Phi^{-1}(\frac{x}{LGD}) - t}{\sqrt{\rho}})$$

And the *default distribution* is the derivative, and is equal to:

$$f(x) = \frac{\sqrt{1-\rho}}{LGD\sqrt{\rho}} \exp(-\frac{1}{2\rho}(\sqrt{1-\rho}\Phi^{-1}(\frac{x}{LGD}) - t)^2 + \frac{1}{2}\Phi^{-1}(\frac{x}{LGD}))^2$$

```
> # Vasicek model parameters
> rho <- 0.1; lgd <- 0.4
> defprob <- 0.05; threshv <- qnorm(defprob)
> # Define Vasicek cumulative loss distribution
> cumlossdistr <- function(x, threshv=(-2), rho=0.2, lgd=0.4)
+   pnorm((sqrt(1-rho)*qnorm(x/lgd) - threshv)/sqrt(rho))
> # Define Vasicek loss distribution function
> lossdistr <- function(x, threshv=(-2), rho=0.2, lgd=0.4)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnorm(x/lgd) - threshv)^2/(2*rho) + qnorm(x/lgd)^2/2)/lgd
```



```
> # Plot probability distribution of losses
> x11(width=6, height=5)
> curve(expr=lossdistr(x, threshv=threshv, rho=rho),
+ cex.main=1.8, cex.lab=1.8, cex.axis=1.5,
+ type="l", xlim=c(0, 0.06),
+ xlab="loss percentage", ylab="density", lwd=3,
+ col="blue", main="Portfolio Loss Density")
> # Add line for expected loss
> abline(v=lgd*defprob, col="red", lwd=3)
> text(x=lgd*defprob-0.001, y=35, labels="expected loss", lwd=3, pos
```

Collateralized Debt Obligations (*CDOs*)

Collateralized Debt Obligations (cash *CDOs*) are securities (bonds) collateralized by other debt assets.

The *CDO* assets can be debt instruments like bonds, loans, and mortgages.

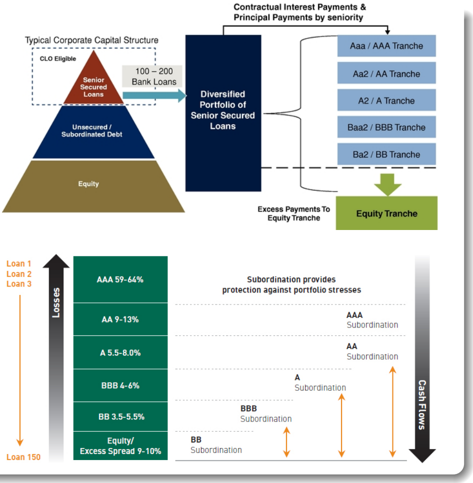
The *CDO* liabilities are *CDO* tranches, which receive cashflows from the *CDO* assets, and are exposed to their defaults.

CDO tranches have an attachment point (subordination, i.e. the percentage of asset default losses at which the tranche starts absorbing those losses), and a detachment point when the tranche is wiped out (suffers 100% losses).

The *equity tranche* is the most junior tranche, and is the first to absorb default losses.

The *mezzanine tranches* are senior to the *equity tranche* and absorb losses ony after the *equity tranche* is wiped out.

The *senior tranche* is the most senior tranche, and is the last to absorb losses.



CDO Tranche Losses

Single-tranche (synthetic) *CDOs* are credit default swaps which reference credit portfolios.

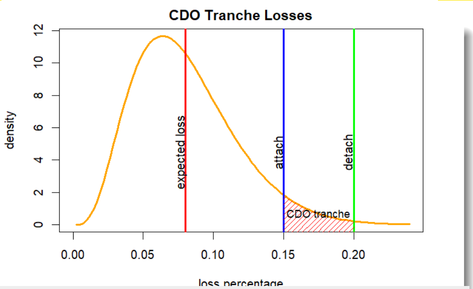
The expected loss *EL* on a *CDO* tranche is:

$$EL = \frac{1}{d-a} \int_{a}^{d} (x-a)\, f(x)\, \mathrm{d}x + \int_{d}^{LGD} f(x)\, \mathrm{d}x$$

Where $f(x)$ is the density of portfolio losses, and a and d are the tranche attachment (subordination) and detachment points.

The difference $(d-a)$ is the tranche *thickness*, so that *EL* is the expected loss as a percentage of the tranche notional.

A single-tranche *CDO* can be thought of as a short option spread on the asset defaults, struck at the attachment and detachment points.



```
> # Define Vasicek cumulative loss distribution
> # (with error handling for x)
> cumlossdistr <- function(x, threshv=(-2), rho=0.2, lgd=0.4) {
+   qnormv <- ifelse(x/lgd < 0.999, qnorm(x/lgd), 3.1)
+   pnorm((sqrt(1-rho)*qnormv - threshv)/sqrt(rho))
+ }  # end cumlossdistr
> # Define Vasicek loss distribution function
> # (vectorized version with error handling for x)
> lossdistr <- function(x, threshv=(-2), rho=0.1, lgd=0.4) {
+   qnormv <- ifelse(x/lgd < 0.999, qnorm(x/lgd), 3.1)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnormv - threshv)^2/(2*rho
+ }  # end lossdistr
```

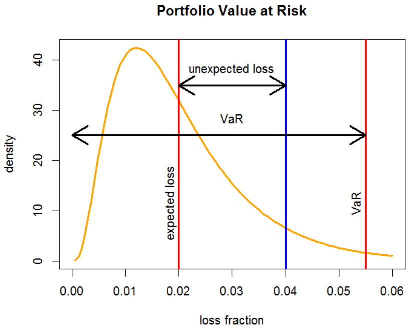
```
> defprob <- 0.2; threshv <- qnorm(defprob)
> rho <- 0.1; lgd <- 0.4
> attachp <- 0.15; detachp <- 0.2
> # Expected tranche loss is sum of two terms
> tranchel <-
+   # Loss between attachp and detachp
+   integrate(function(x, attachp) (x-attachp)*lossdistr(x,
+ threshv=threshv, rho=rho, lgd=lgd),
+ low=attachp, up=detachp, attachp=attachp)$value/(detachp-attachp)
+   # Loss in excess of detachp
+   (1-cumlossdistr(x=detachp, threshv=threshv, rho=rho, lgd=lgd))
> # Plot probability distribution of losses
> curve(expr=lossdistr(x, threshv=threshv, rho=rho),
+ cex.main=1.8, cex.lab=1.8, cex.axis=1.5,
+ type="l", xlim=c(0, 3*lgd*defprob),
+ xlab="loss percentage", ylab="density", lwd=3,
+ col="orange", main="CDO Tranche Losses")
> # Add line for expected loss
> abline(v=lgd*defprob, col="red", lwd=3)
> text(x=lgd*defprob-0.001, y=4, labels="expected loss",
+ lwd=2, srt=90, pos=3, cex=1.8)
```

Portfolio Value at Risk

Value at Risk (*VaR*) measures extreme portfolio loss (but not the worst possible loss), defined as the *quantile* of the loss distribution, corresponding to a given confidence level α.

A loss exceeding the *EL* is called the Unexpected Loss (*UL*), and can be calculated from the *portfolio loss distribution*.

```
> # Add lines for unexpected loss
> abline(v=0.04, col="blue", lwd=3)
> arrows(x0=0.02, y0=35, x1=0.04, y1=35, code=3, lwd=3, cex=0.5)
> text(x=0.03, y=36, labels="unexpected loss", lwd=2, pos=3)
> # Add lines for VaR
> abline(v=0.055, col="red", lwd=3)
> arrows(x0=0.0, y0=25, x1=0.055, y1=25, code=3, lwd=3, cex=0.5)
> text(x=0.03, y=26, labels="VaR", lwd=2, pos=3)
> text(x=0.055-0.001, y=10, labels="VaR", lwd=2, srt=90, pos=3)
```



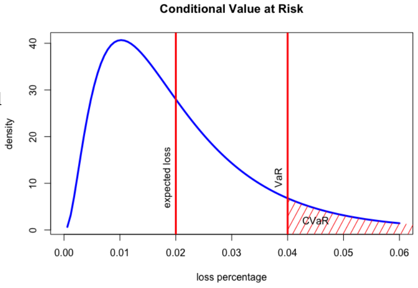
Portfolio Conditional Value at Risk

The *Conditional Value at Risk* (*CVaR*) is equal to the average of the *VaR* values for all the confidence levels greater than the given confidence level α:

$$\mathrm{CVaR} = \frac{1}{1-\alpha} \int_{\alpha}^{1} \mathrm{VaR}(p)\,\mathrm{d}p = \frac{1}{1-\alpha} \int_{\mathrm{VaR}}^{LGD} x\,f(x)\,\mathrm{d}$$

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

```
> varisk <- 0.04; varmax <- 4*lgd*defprob
> # Calculate CVaR
> cvar <- integrate(function(x) x*lossdistr(x, threshv=threshv,
+   rho=rho, lgd=lgd), low=varisk, up=lgd)$value
> cvar <- cvar/integrate(lossdistr, low=varisk, up=lgd,
+    threshv=threshv, rho=rho, lgd=lgd)$value
> # Plot probability distribution of losses
> curve(expr=lossdistr(x, threshv=threshv, rho=rho),
+ type="l", xlim=c(0, 0.06),
+ xlab="loss percentage", ylab="density", lwd=3,
+ col="blue", main="Conditional Value at Risk")
> # Add line for expected loss
> abline(v=lgd*defprob, col="red", lwd=3)
> text(x=lgd*defprob-0.001, y=10, labels="expected loss", lwd=2, s
```



```
> # Add lines for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk-0.001, y=10, labels="VaR",
+  lwd=2, srt=90, pos=3)
> # Add shading for CVaR
> vars <- seq(varisk, varmax, length=100)
> densv <- sapply(vars, lossdistr,
+   threshv=threshv, rho=rho)
> # Draw shaded polygon
> polygon(c(varisk, vars, varmax), density=20,
+   c(-1, densv, -1), col="red", border=NA)
> text(x=varisk+0.005, y=0, labels="CVaR", lwd=2, pos=3)
```

Value at Risk Under the Vasicek Model

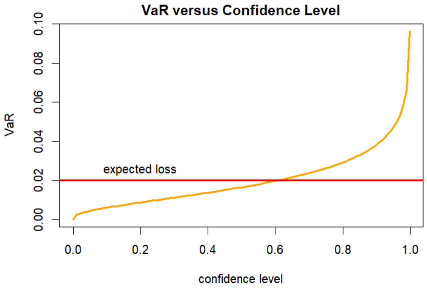
Value at Risk (*VaR*) measures extreme portfolio loss (but not the worst possible loss), defined as the *quantile* of the loss distribution, corresponding to a given confidence level α.

The *cumulative loss distribution* is equal to:

$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}(\frac{x}{LGD}) - t}{\sqrt{\rho}}\right)$$

Then the *quantile* of the loss distribution ($x = VaR$), for a given a confidence level $\alpha = P(x)$, is given by the inverse of the *cumulative loss distribution*:

$$VaR(\alpha) = LGD \cdot \Phi\left(\frac{\sqrt{\rho}\Phi^{-1}(\alpha) + t}{\sqrt{1-\rho}}\right)$$

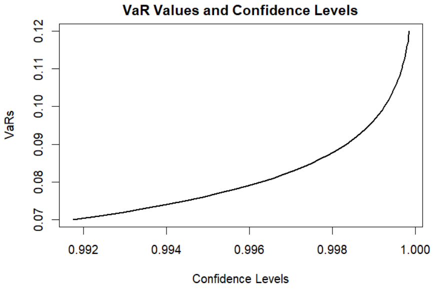


```
> # VaR (quantile of the loss distribution)
> varfun <- function(x, threshv=qnorm(0.1), rho=0.1, lgd=0.4)
+   lgd*pnorm((sqrt(rho)*qnorm(x) + threshv)/sqrt(1-rho))
> varfun(x=0.99, threshv=threshv, rho=rho, lgd=lgd)
> # Plot VaR
> curve(expr=varfun(x, threshv=threshv, rho=rho, lgd=lgd),
+ type="l", xlim=c(0, 0.999), xlab="confidence level", ylab="VaR", lwd=3,
+ col="orange", main="VaR versus Confidence Level")
> # Add line for expected loss
> abline(h=lgd*defprob, col="red", lwd=3)
> text(x=0.2, y=lgd*defprob, labels="expected loss", lwd=2, pos=3)
```

Value at Risk and Confidence Levels

The confidence levels of *VaR* values can also be calculated by integrating over the tail of the loss density function.

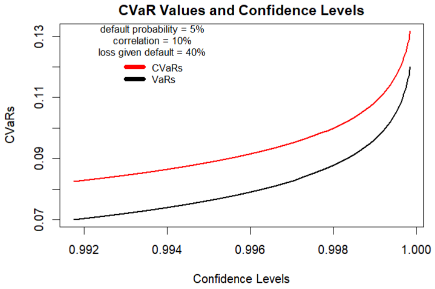
```
> # Integrate lossdistr() over full range
> integrate(lossdistr, low=0.0, up=lgd,
+     threshv=threshv, rho=rho, lgd=lgd)
> # Calculate expected losses using lossdistr()
> integrate(function(x) x*lossdistr(x, threshv=threshv,
+   rho=rho, lgd=lgd), low=0.0, up=lgd)
> # Calculate confidence levels corresponding to VaR values
> vars <- seq(0.07, 0.12, 0.001)
> confls <- sapply(vars, function(varisk) {
+   integrate(lossdistr, low=varisk, up=lgd,
+       threshv=threshv, rho=rho, lgd=lgd)
+ })  # end sapply
> confls <- cbind(as.numeric(t(confls)[, 1]), vars)
> colnames(confls) <- c("levels", "VaRs")
> # Calculate 95% confidence level VaR value
> confls[match(TRUE, confls[, "levels"] < 0.05), "VaRs"]
> plot(x=1-confls[, "levels"],
+      y=confls[, "VaRs"], lwd=2,
+      xlab="confidence level", ylab="VaRs",
+      t="l", main="VaR Values and Confidence Levels")
```



Conditional Value at Risk Under the Vasicek Model

The *CVaR* values can be calculated by integrating over the tail of the loss density function.

```
> # Calculate CVaR values
> cvars <- sapply(vars, function(varisk) {
+   integrate(function(x) x*lossdistr(x, threshv=threshv,
+ rho=rho, lgd=lgd), low=varisk, up=lgd)})  # end sapply
> confls <- cbind(confls, as.numeric(t(cvars)[, 1]))
> colnames(confls)[3] <- "CVaRs"
> # Divide CVaR by confidence level
> confls[, "CVaRs"] <- confls[, "CVaRs"]/confls[, "levels"]
> # Calculate 95% confidence level CVaR value
> confls[match(TRUE, confls[, "levels"] < 0.05), "CVaRs"]
> # Plot CVaRs
> plot(x=1-confls[, "levels"], y=confls[, "CVaRs"],
+      t="l", col="red", lwd=2,
+      ylim=range(confls[, c("VaRs", "CVaRs")]),
+      xlab="confidence level", ylab="CVaRs",
+      main="CVaR Values and Confidence Levels")
```



```
> # Add VaRs
> lines(x=1-confls[, "levels"], y=confls[, "VaRs"], lwd=2)
> # Add legend
> legend(x="topleft", legend=c("CVaRs", "VaRs"),
+    title="default probability = 5%
+ correlation = 10%
+ loss given default = 40%",
+    inset=0.1, cex=1.0, bg="white", bty="n",
+    lwd=6, lty=1, col=c("red", "black"))
```

Simulating Portfolio Losses Under the Vasicek Model

If the default probabilities p_i are not all the same, then there's no simple formula for the *portfolio loss distribution* under the Vasicek Model.

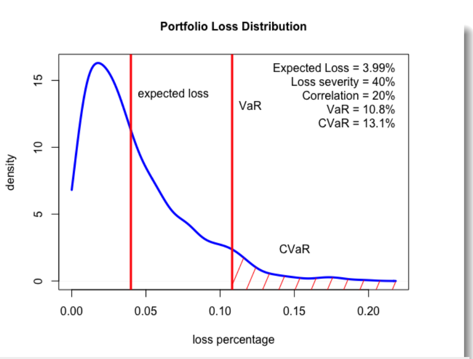
In that case the portfolio losses and *VaR* must be simulated.

```
> # Define model parameters
> nbonds <- 300; nsimu <- 1000; lgd <- 0.4
> # Define correlation parameters
> rho <- 0.2; rhos <- sqrt(rho); rhosm <- sqrt(1-rho)
> # Calculate default probabilities and thresholds
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> probv <- runif(nbonds, max=0.2)
> threshv <- qnorm(probv)
> # Simulate losses under the Vasicek model
> sysv <- rnorm(nsimu)
> assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
> assetm <- t(rhos*sysv + t(rhosm*assetm))
> lossm <- lgd*colSums(assetm < threshv)/nbonds
```

VaR and *CVaR* Under the Vasicek Model

The function `density()` calculates a kernel estimate of the probability density for a sample of data, and returns a list with a vector of loss values and a vector of corresponding densities.

```
> # Calculate VaR from confidence level
> confl <- 0.95
> varisk <- quantile(lossm, confl)
> # Calculate the CVaR as the mean losses in excess of VaR
> cvar <- mean(lossm[lossm > varisk])
> # Plot the density of portfolio losses
> densv <- density(lossm, from=0)
> plot(densv, xlab="loss percentage", ylab="density",
+    cex.main=1.0, cex.lab=1.0, cex.axis=1.0,
+    lwd=3, col="blue", main="Portfolio Loss Distribution")
> # Add vertical line for expected loss
> exploss <- lgd*mean(probv)
> abline(v=exploss, col="red", lwd=3)
> xmax <- max(densv$x); ymax <- max(densv$y)
> text(x=exploss, y=(6*ymax/7), labels="expected loss",
+      lwd=2, pos=4, cex=1.0)
> # Add vertical line for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk, y=4*ymax/5, labels="VaR", lwd=2, pos=4, cex=1.0)
```

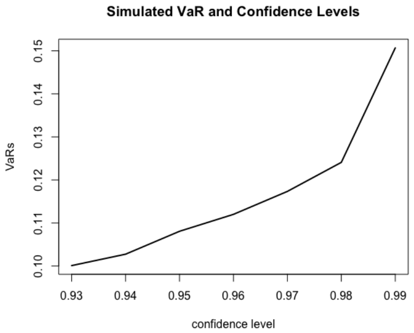


```
> # Draw shaded polygon for CVaR
> intail <- (densv$x > varisk)
> xvar <- c(min(densv$x[intail]), densv$x[intail], max(densv$x))
> polygon(xvar, c(-1, densv$y[intail], -1), col="red", border=NA, de
> # Add text for CVaR
> text(x=5*varisk/4, y=(ymax/7), labels="CVaR", lwd=2, pos=4, cex=1.
> # Add text with data
> text(xmax, ymax, labels=paste0(
+    "Expected Loss = ", format(100*exploss, digits=3), "%", "\n",
+    "Loss severity = ", format(100*lgd, digits=3), "%", "\n",
+    "Correlation = ", format(100*rho, digits=3), "%", "\n",
+    "VaR = ", format(100*varisk, digits=3), "%", "\n",
+    "CVaR = ", format(100*cvar, digits=3), "%"),
+    adj=c(1, 1), cex=1.0, lwd=2)
```

Simulating *VaR* Under the Vasicek Model

The *VaR* can be calculated from the simulated portfolio losses using the function `quantile()`.

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`.



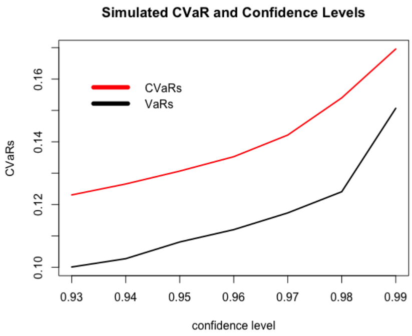
```
> # Calculate VaRs from confidence levels
> confls <- seq(0.93, 0.99, 0.01)
> vars <- quantile(lossm, probs=confls)
> plot(x=confls, y=vars, t="l", lwd=2,
+    xlab="confidence level", ylab="VaRs",
+    main="Simulated VaR and Confidence Levels")
```

Simulating *CVaR* Under the Vasicek Model

The *CVaR* can be calculated from the frequency of tail losses in excess of the *VaR*.

The function `table()` calculates the frequency distribution of categorical data.

```
> # Calculate CVaRs
> cvars <- sapply(vars, function(varisk) {
+   mean(lossm[lossm >= varisk])
+ })  # end sapply
> cvars <- cbind(cvars, vars)
> # Alternative CVaR calculation using frequency table
> # first calculate frequency table of losses
> # tablev <- table(lossm)/nsimu
> # Calculate CVaRs from frequency table
> # cvars <- sapply(vars, function(varisk) {
> #   tailrisk <- tablev[names(tablev) > varisk]
> #   tailrisk %*% as.numeric(names(tailrisk)) / sum(tailrisk)
> # })  # end sapply
```



```
> # Plot CVaRs
> plot(x=confls, y=cvars[, "cvars"],
+    t="l", col="red", lwd=2, ylim=range(cvars),
+    xlab="confidence level", ylab="CVaRs",
+    main="Simulated CVaR and Confidence Levels")
> # Add VaRs
> lines(x=confls, y=cvars[, "vars"], lwd=2)
> # Add legend
> legend(x="topleft", legend=c("CVaRs", "VaRs"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.1, lwd=6, lty=1, col=c("red", "black"))
```

Function for Simulating *VaR* Under the Vasicek Model

The function calc_var() simulates default losses under the *Vasicek* model, for a vector of confidence levels, and calculates a vector of *VaR* and *CVaR* values.

```
> calc_var <- function(threshv, # Default thresholds
+    lgd=0.6, # loss given default
+    rhos, rhosm, # asset correlation
+    nsimu=1000, # number of simulations
+    confls=seq(0.93, 0.99, 0.01) # Confidence levels
+    ) {
+   # Define model parameters
+   nbonds <- NROW(threshv)
+   # Simulate losses under the Vasicek model
+   sysv <- rnorm(nsimu)
+   assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
+   assetm <- t(rhos*sysv + t(rhosm*assetm))
+   lossm <- lgd*colSums(assetm < threshv)/nbonds
+   # Calculate VaRs and CVaRs
+   vars <- quantile(lossm, probs=confls)
+   cvars <- sapply(vars, function(varisk) {
+     mean(lossm[lossm >= varisk])
+   })  # end sapply
+   names(vars) <- confls
+   names(cvars) <- confls
+   c(vars, cvars)
+ }  # end calc_var
```

Standard Errors of *VaR* Using Bootstrap Simulation

The values of *VaR* and *CVaR* produced by the function calc_var() are subject to uncertainty because they're calculated from a simulation.

We can calculate the standard errors of *VaR* and *CVaR* by running the function calc_var() many times and repeating the simulation in a loop.

This bootstrap will only capture the uncertainty due to the finite number of trials in the simulation, but not due to the uncertainty of model parameters.

```
> # Define model parameters
> nbonds <- 300; nsimu <- 1000; lgd <- 0.4
> rho <- 0.2; rhos <- sqrt(rho); rhosm <- sqrt(1-rho)
> # Calculate default probabilities and thresholds
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> probv <- runif(nbonds, max=0.2)
> threshv <- qnorm(probv)
> confls <- seq(0.93, 0.99, 0.01)
> # Define number of bootstrap simulations
> nboot <- 500
> # Perform bootstrap of calc_var
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> bootd <- sapply(rep(lgd, nboot), calc_var,
+   threshv=threshv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end sapply
> bootd <- t(bootd)
> # Calculate standard errors of VaR and CVaR from bootd data
> varsd <- apply(bootd[, 1:7], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> cvarsd <- apply(bootd[, 8:14], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> # Scale the standard errors of VaRs and CVaRs
> varsds <- varsd[2, ]/varsd[1, ]
> cvarsds <- cvarsd[2, ]/cvarsd[1, ]
```

Standard Errors of *VaR* at High Confidence Levels

The standard errors of *VaR* and *CVaR* are inversely proportional to square root of the number of loss events in the simulation, that exceed the *VaR*.

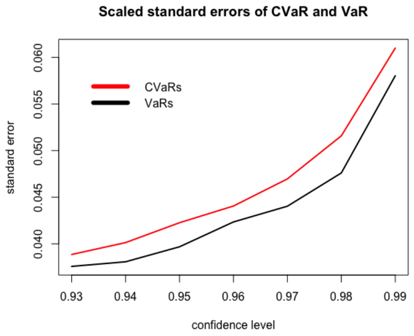
So the greater the number of loss events, the smaller the standard errors, and vice versa.

But as the confidence level increases, the *VaR* also increases, and the number of loss events decreases, causing larger standard errors.

So the as the confidence level increases, the standard errors of *VaR* and *CVaR* also increase.

The *scaled* (relative) standard errors of *VaR* and *CVaR* also increase with the confidence level, making them much less reliable at very high confidence levels.

The standard error of *CVaR* is even greater than that of *VaR*.

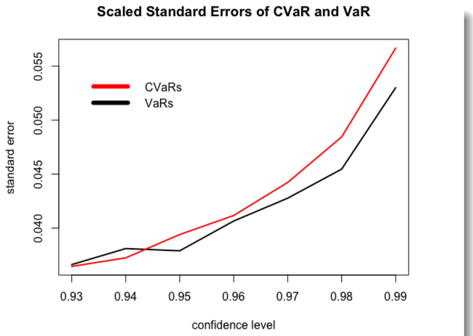


```
> # Plot the scaled standard errors of VaRs and CVaRs
> plot(x=names(varsds), y=varsds,
+   t="l", lwd=2, ylim=range(c(varsds, cvarsds)),
+   xlab="confidence level", ylab="standard error",
+   main="Scaled Standard Errors of CVaR and VaR")
> lines(x=names(cvarsds), y=cvarsds, lwd=2, col="red")
> legend(x="topleft", legend=c("CVaRs", "VaRs"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.1, lwd=6, lty=1, col=c("red", "black"))
```

Standard Errors of *VaR* Using Parallel Bootstrap

The *scaled* standard errors of *VaR* and *CVaR* increase with the confidence level, making them much less reliable at very high confidence levels.

```
> library(parallel)  # load package parallel
> ncores <- detectCores() - 1  # number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster
> # Perform bootstrap of calc_var for Windows
> clusterSetRNGStream(compclust, 1121)
> bootd <- parLapply(compclust, rep(lgd, nboot),
+   fun=calc_var, threshv=threshv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end parLapply
> stopCluster(compclust)  # Stop R processes over cluster
> # Bootstrap under Mac-OSX or Linux
> bootd <- mclapply(rep(lgd, nboot),
+   FUN=calc_var, threshv=threshv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end mclapply
> bootd <- rutils::do_call(rbind, bootd)
> # Calculate standard errors of VaR and CVaR from bootd data
> varsd <- apply(bootd[, 1:7], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> cvarsd <- apply(bootd[, 8:14], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> # Scale the standard errors of VaRs and CVaRs
> varsds <- varsd[2, ]/varsd[1, ]
> cvarsds <- cvarsd[2, ]/cvarsd[1, ]
```



```
> # Plot the standard errors of VaRs and CVaRs
> plot(x=names(varsds), y=varsds, t="l", lwd=2,
+   ylim=range(c(varsds, cvarsds)),
+   xlab="confidence level", ylab="standard error",
+   main="Scaled Standard Errors of CVaR and VaR")
> lines(x=names(cvarsds), y=cvarsds, lwd=2, col="red")
> legend(x="topleft", legend=c("CVaRs", "VaRs"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.1, lwd=6, lty=1, col=c("red", "black"))
```

Vasicek Model With Uncertain Default Probabilities

The previous bootstrap only captured the uncertainty due to the finite simulation trials, but not due to the uncertainty of model parameters, such as the default probabilities and correlations.

The below function calc_var() can simulate the *Vasicek* model with uncertain default probabilities.

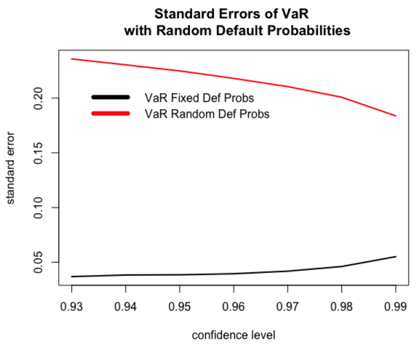
```
> calc_var <- function(probv, # Default probabilities
+    lgd=0.6, # loss given default
+    rhos, rhosm, # asset correlation
+    nsimu=1000, # number of simulations
+    confls=seq(0.93, 0.99, 0.01) # Confidence levels
+    ) {
+   # Calculate random default thresholds
+   threshv <- qnorm(runif(1, min=0.5, max=1.5)*probv)
+   # Simulate losses under the Vasicek model
+   nbonds <- NROW(probv)
+   sysv <- rnorm(nsimu)
+   assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
+   assetm <- t(rhos*sysv + t(rhosm*assetm))
+   lossm <- lgd*colSums(assetm < threshv)/nbonds
+   # Calculate VaRs and CVaRs
+   vars <- quantile(lossm, probs=confls)
+   cvars <- sapply(vars, function(varisk) {
+     mean(lossm[lossm >= varisk])
+   })  # end sapply
+   names(vars) <- confls
+   names(cvars) <- confls
+   c(vars, cvars)
+ }  # end calc_var
```

Standard Errors Due to Uncertain Default Probabilities

The greatest contribution to the standard errors of *VaR* and *CVaR* is from the uncertainty of model parameters, such as the default probabilities, correlations, and loss severities.

For example, a 50% uncertainty in the default probabilities can produce a 20% uncertainty of the *VaR*.

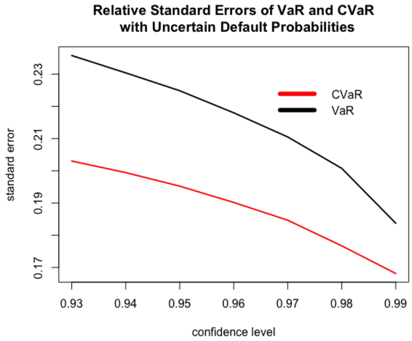
```
> library(parallel)  # load package parallel
> ncores <- detectCores() - 1  # number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster
> # Perform bootstrap of calc_var for Windows
> clusterSetRNGStream(compclust, 1121)
> bootd <- parLapply(compclust, rep(lgd, nboot),
+   fun=calc_var, probv=probv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end parLapply
> stopCluster(compclust)  # Stop R processes over cluster
> # Bootstrap under Mac-OSX or Linux
> bootd <- mclapply(rep(lgd, nboot),
+   FUN=calc_var, probv=probv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end mclapply
> bootd <- rutils::do_call(rbind, bootd)
> # Calculate standard errors of VaR and CVaR from bootd data
> varsd <- apply(bootd[, 1:7], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> cvarsd <- apply(bootd[, 8:14], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> # Scale the standard errors of VaRs and CVaRs
> varsdsu <- varsd[2, ]/varsd[1, ]
> cvarsdsu <- cvarsd[2, ]/cvarsd[1, ]
```



```
> # Plot the standard errors of VaRs under uncertain default probabi
> plot(x=colnames(varsd), y=varsds, t="l",
+  col="black", lwd=2, ylim=range(c(varsds, varsdsu)),
+   xlab="confidence level", ylab="standard error",
+   main="Standard Errors of VaR
+   with Random Default Probabilities")
> lines(x=colnames(varsd), y=varsdsu, lwd=2, col="red")
> legend(x="topleft",
+    legend=c("VaR Fixed Def Probs", "VaR Random Def Probs"),
+    bty="n", title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.1, lwd=6, lty=1, col=c("black", "red"))
```

Relative Errors Due to Uncertain Default Probabilities

The *scaled* (relative) standard errors of *VaR* and *CVaR* under uncertain default probabilities decrease with higher confidence level, because the standard errors are less dependent on the confidence level and don't increase as fast as the *VaR* does.



```
> # Plot the standard errors of VaRs and CVaRs
> plot(x=colnames(varsd), y=varsdsu, t="l", lwd=2,
+   ylim=range(c(varsdsu, cvarsdsu)),
+   xlab="confidence level", ylab="standard error",
+   main="Relative Standard Errors of VaR and CVaR
+   with Uncertain Default Probabilities")
> lines(x=colnames(varsd), y=cvarsdsu, lwd=2, col="red")
> legend(x="topright", legend=c("CVaR", "VaR"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.1, lwd=6, lty=1, col=c("red", "black"))
```

Model Risk of Credit Portfolio Models

Credit portfolio models are subject to very significant *model risk* due to the uncertainties of model parameters, such as the default probabilities, correlations, and loss severities.

Model risk is the risk of incorrect model predictions due to incorrect model specification, and due to incorrect model parameters.

Jon Danielsson at the London School of Economics (LSE) has studied the model risk of *VaR* and *CVaR* in: Why Risk is So Hard to Measure, and in Model Risk of Risk Models.

Jon Danielsson has pointed out that there's not enough historical data to be able to accurately calculate the credit model parameters.

Jon Danielsson and Chen Zhou have demonstrated that accurately estimating *CVaR* at 5% confidence would require decades of price history, something that simply doesn't exist for many assets.

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*.
  You can download `R` Cheat Sheets here.