Numerical Analysis

FRE6871 & FRE7241, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

December 8, 2022

Floating Point Numbers

R prints floating point numbers without showing their full internal representation, which can cause confusion about their true value.

Real numbers which have an infinite number of significant digits can only be represented approximately inside a computer.

Floating point numbers are approximate representations of *real* numbers inside a computer.

Machine precision is a number that specifies the accuracy of floating point numbers in a computer.

The representation of floating point numbers in R depends on the *machine precision* of the computer operating system.

The variable `.Machine` contains information about the numerical characteristics of the computer R is running on, such as the largest `double` and `integer` numbers, and the *machine precision*.

```
> numv <- 0.3/3
> numv  # Printed as "0.1"
> numv - 0.1  # numv is not equal to "0.1"
> numv == 0.1  # numv is not equal to "0.1"
> print(numv, digits=10)
> print(numv, digits=16)
> # numv is equal to "0.1" within machine precision
> all.equal(numv, 0.1)
> numv <- (3-2.9)
> print(numv, digits=20)
> # Info machine precision of computer R is running on
> # ?.Machine
> # Machine precision
> .Machine$double.eps
```

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

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

The generic function `print()` prints its argument and returns it *invisibly*,

Floating Point Calculations

Calculations with floating point numbers are subject to *numerical error* (they're not perfectly accurate).

Rounding a number means replacing it with the closest number of a given precision.

The *IEC 60559* convention is to round to the nearest even number (1.5 to 2, and also 2.5 to 2), which preserves the mean of a sequence.

The function `round()` rounds a number to the specified number of decimal places.

Truncating a number means replacing it with the largest integer which is less than the given number.

The function `trunc()` truncates a number.

The function `ceiling()` returns the smallest integer which is greater than the given number.

```
> numv <- sqrt(2)
> numv^2  # Printed as "2"
> numv^2 == 2  # numv^2 is not equal to "2"
> print(numv^2, digits=20)
> # numv^2 is equal to "2" within machine precision
> all.equal(numv^2, 2)
> # Numbers with precision 0.1
> 0.1*(1:10)
> # Round to precision 0.1
> round(3.675, 1)
> # Round to precision 1.0
> round(3.675)
> # Round to nearest even number
> c(round(2.5), round(3.5), round(4.5))
> round(4:20/2)  # Round to nearest even number
> trunc(3.675)  # Truncate
```

Comparing Objects With identical() and all.equal()

The function `identical()` tests if two objects are exactly the same, and always returns a single logical `TRUE` or `FALSE` (never `NA` or logical `vectors`).

For atomic arguments `identical()` often gives the same result as the `"=="` operator, but it's not synonymous with it in general.

The `"=="` operator applies the *recycling rule* to vector arguments and returns logical `vectors`, but `identical()` doesn't and returns a single logical value.

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

The variable `.Machine` contains information about the numerical characteristics of the computer `R` is running on, such as the largest `double` and `integer` numbers, and the *machine precision*.

```
> numv <- 2
> numv==2
> identical(numv, 2)
>
> identical(numv, NULL)
> # This doesn't work:
> # numv==NULL
> is.null(numv)
>
> vectorv <- c(2, 4, 6)
> vectorv == 2
> identical(vectorv, 2)
>
> # numv is equal to "1.0" within machine precision
> numv <- 1.0 + 2*sqrt(.Machine$double.eps)
> all.equal(numv, 1.0)
>
> # Info machine precision of computer R is running on
> # ?.Machine
> # Machine precision
> .Machine$double.eps
```

Modular Arithmetic Operators

R has two modular arithmetic *operators*:

- "%/%" performs *modulo* division,
- "%%" calculates remainder of *modulo* division,

Modulo division of floating point (non-integer) numbers sometimes produces incorrect results because of limited *machine precision* of floating point numbers.

For example, the number 0.2 is stored as a binary number slightly larger than 0.2, so the result of calculating `0.6 %/% 0.2` is 2 instead of 3.

See also the discussion in: http://stackoverflow.com/questions/13614749/modulus-bug-in-r

```
> 4.7 %/% 0.5  # Modulo division
> 4.7 %% 0.5  # Remainder of modulo division
> # Reversing modulo division usually
> # returns the original number
> (4.7 %% 0.5) + 0.5 * (4.7 %/% 0.5)
> # Modulo division of non-integer numbers can
> # produce incorrect results
> 0.6 %/% 0.2  # Produces 2 instead of 3
> 6 %/% 2  # Use integers to get correct result
> # 0.2 stored as binary number
> # Slightly larger than 0.2
> print(0.2, digits=22)
```

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)
```

Kernel Density of Asset Returns

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

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

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

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

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

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

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

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

```
> library(rutils)  # Load package rutils
> # Calculate VTI percentage returns
> retsp <- rutils::etfenv$returns$VTI
> retsp <- drop(coredata(na.omit(retsp)))
> nrows <- NROW(retsp)
> # Mean and standard deviation of returns
> c(mean(retsp), sd(retsp))
> # Calculate the MAD of returns 10 points apart
> retsp <- sort(retsp)
> bwidth <- 10*mad(rutils::diffit(retsp, lagg=10))
> # Calculate the kernel density
> densityv <- sapply(1:nrows, function(it) {
+   sum(dnorm(retsp-retsp[it], sd=bwidth))
+ })  # end sapply
> madv <- mad(retsp)
> plot(retsp, densityv, xlim=c(-5*madv, 5*madv),
+      t="l", col="blue", lwd=3,
+      xlab="returns", ylab="density",
+      main="Density of VTI Returns")
> # Calculate the kernel density using density()
> densityv <- density(retsp, bw=bwidth)
> NROW(densityv$y)
> x11(width=6, height=5)
> plot(densityv, xlim=c(-5*madv, 5*madv),
+      xlab="returns", ylab="density",
+      col="blue", lwd=3, main="Density of VTI Returns")
> # Interpolate the densityv vector into returns
> densityv <- approx(densityv$x, densityv$y, xout=retsp)
> all.equal(densityv$x, retsp)
> plot(densityv, xlim=c(-5*madv, 5*madv),
+      xlab="returns", ylab="density",
+      t="l", col="blue", lwd=3,
+      main="Density of VTI Returns")
```

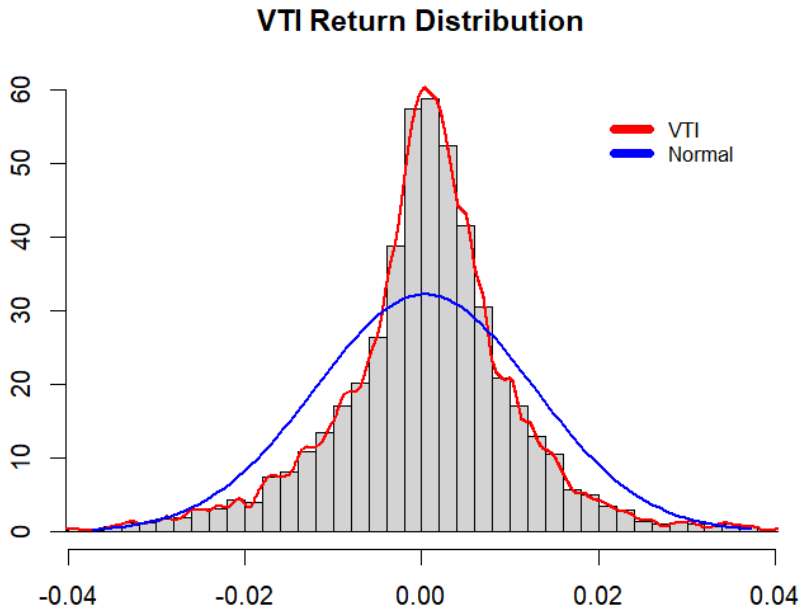
Distribution of Asset Returns

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

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

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

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




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

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
> vectorv <- runif(1e6)
> object.size(vectorv)
> format(object.size(vectorv), units="MB")
> # Get sizes of objects in workspace
> sort(sapply(ls(), function(ob_ject) {
+   format(object.size(get(ob_ject)), units="KB")}))
> # Get sizes of all objects in workspace
> sort(sapply(mget(ls()), object.size))
> sort(sapply(mget(ls()), function(ob_ject) {
+   format(object.size(ob_ject), 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(ob_ject) {
+   object.size(get(ob_ject, 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 namesv, class, and size of objects in workspace
> ob_jects <- gdata::ll(unit="bytes")
> # Sort by memory size (descending)
> ob_jects[order(ob_jects[, 2], decreasing=TRUE), ]
> gdata::ll()[order(ll()$KB, decreasing=TRUE), ]
> # Get sizes of objects in etfenv environment
> gdata::ll(unit="bytes", etfenv)
```

Managing Very Large Datasets Using Package *SOAR*

The package *SOAR* allows performing calculations with multiple, very large datasets, without loading them all at once into R memory.

Package *SOAR* uses *delayed assignment* of objects (*lazy loading*), which means that they don't reside in R memory, but they're silently loaded from the hard drive when they're needed.

The function `Store()` removes objects from memory, stores them in an *object cache*, and places the *object cache* on the search path.

The *object cache* is a sub-directory of the *cwd* called `.R_Cache`, and contains `.RData` files with the stored objects.

The stored objects aren't listed in the R workspace, but they are visible on the search path as *promises*.

The function `Ls()` listv the objects stored in the *object cache*, and attaches the *cache* to the search path.

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

The function `data()` isn't required to load data sets that are set up for *lazy loading*.

```
> library(SOAR)  # Load package SOAR
> # Get sizes of objects in workspace
> sort(sapply(mget(ls()), object.size))
> Store(etf_list)  # Store in object cache
> # Get sizes of objects in workspace
> sort(sapply(mget(ls()), object.size))
> search()  # Get search path for R objects
> Ls()  # List object cache
> find("etf_list")  # Find object on search path
```

Memory Usage and Garbage Collection in R

Garbage collection is the process of releasing memory occupied by objects no longer in use by a computer program.

The function gc() performs garbage collection and reports the memory used by R in units of *Vcells* (vector cells, which are 8 *bytes* each).

R performs garbage collection automatically, so calling gc() is designed mostly to report the memory used by R.

The memory used by R is usually greater than the total size of all objects in the workspace, because R requires additional memory.

```
> # Get R memory
> vcells <- gc()["Vcells", "used"]
> # Create vector with 1,000,000 elements
> numv <- numeric(1000000)
> # Get extra R memory
> gc()["Vcells", "used"] - vcells
> # Get total size of all objects in workspace
> print(object.size(x=mget(ls())), units="MB")
```

Benchmarking the Speed of R Code

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Writing Fast R Code Without Method Dispatch

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

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

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

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

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

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

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

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

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

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

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

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

Increasing Speed of Loops by Pre-allocating Memory

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

R doesn't require allocating the full memory for vectors or 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).

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

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

Profiling the Performance of R Expressions

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

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

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

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

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

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

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

Package *profvis* for Interactive Visualizations of *Profiling*

The package *profvis* creates interactive visualizations of *profiling* data produced by function `Rprof()`:
https://rstudio.github.io/profvis/

The function `profvis::profvis()` profiles an R expression and creates an interactive *flame graph* visualization:
https://rstudio.github.io/profvis/examples.html

Profiling of different types of loops over the columns of *matrices* and *data frames* shows that `colMeans()` is the fastest for *matrices*, while `lapply()` is the fastest for *data frames*.

`profvis::profvis()` can also profile *shiny apps*.

Profiling can also be launched using the *Profile* menu in *RStudio*.

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

```
> # Profile plotting of regression
> profvis::profvis({
+   plot(price ~ carat, data=ggplot2::diamonds)
+   model <- lm(price ~ carat, data=ggplot2::diamonds)
+   abline(model, col="red")
+ })  # end profvis
> # Four methods of calculating column means of matrix
> matrixv <- matrix(rnorm(1e5), ncol=5e4)
> profvis::profvis({
+   mean_s <- apply(matrixv, 2, mean)
+   mean_s <- colMeans(matrixv)
+   mean_s <- lapply(matrixv, mean)
+   mean_s <- vapply(matrixv, mean, numeric(1))
+ })  # end profvis
> # Four methods of calculating data frame column means
> dframe <- as.data.frame(matrixv)
> profvis::profvis({
+   mean_s <- apply(dframe, 2, mean)
+   mean_s <- colMeans(dframe)
+   mean_s <- lapply(dframe, mean)
+   mean_s <- vapply(dframe, mean, numeric(1))
+ })  # end profvis
> # Profile a shiny app
> profvis::profvis(
+   shiny::runExample(example="06_tabsets",
+             display.mode="normal")
+ )  # end profvis
```

draft: How to Write Fast R Code

Bullet points are duplicates of next slide.

R code can be very fast, provided that the user understands the best ways of writing fast R code:

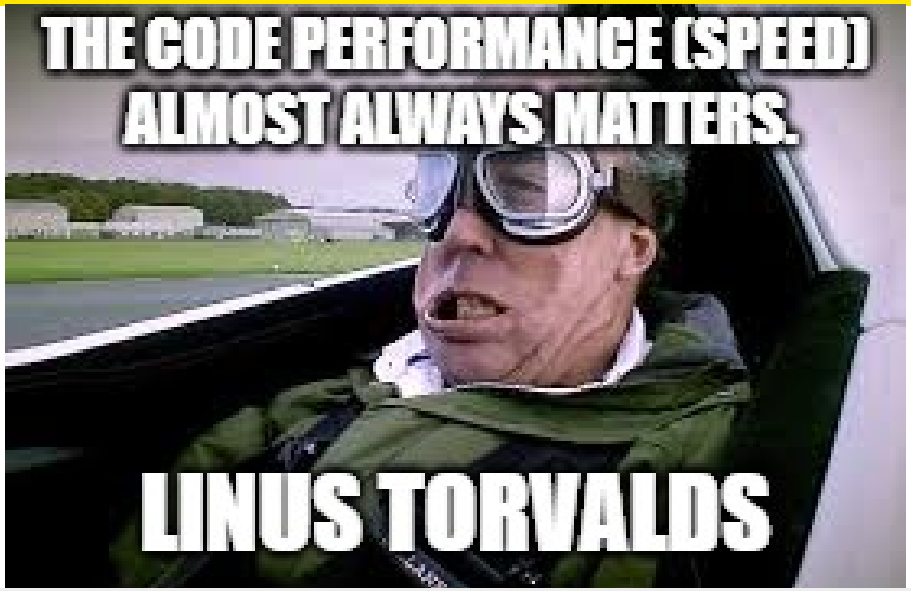
- Call *compiled* functions instead of writing R code for the same task,
- Call function methods directly instead of calling generic functions,
- Create specialized functions by extracting only the essential R code from function methods,
- Write your own C++ functions, compile them using *RcppArmadillo*, and call them from R,
- Pre-allocate memory for new vectors and matrices,
- Avoid writing too many R function calls (remember that every command in R is a function call).

Task	Low-performance R	High-performance R
Loops	for() or apply() loops	C-compiled and vectorized functions
Memory	Automatic R memory allocation	User memory allocation
Dispatch	Generic functions	Class methods
Code	Verbose R code	Rcpp code

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

How to write fast R code:

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




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

Benchmarking the Speed of R Code

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

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

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

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

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

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

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

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

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

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

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

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

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

Increasing Speed of Loops by Pre-allocating Memory

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

R doesn't require allocating the full memory for vectors or 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).

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

Vectorized Functions for Vector Computations

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

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

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

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

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

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

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

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

Vectorized Functions for Matrix Computations

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

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

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

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

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

Fast R Code for Matrix Computations

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

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

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

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

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

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

Package matrixStats for Fast Matrix Computations

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

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

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

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

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

Package Rfast for Fast Matrix and Numerical Computations

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

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

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

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

Writing Fast R Code Using Vectorized Operations

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

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

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

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

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

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

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

Vectorized Functions

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

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

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

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

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

Performing sapply() Loops Over Function Parameters

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

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

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

Creating Vectorized Functions

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

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

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

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

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

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

The mapply() Functional

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

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

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

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

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

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

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

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

Vectorizing Functions Using mapply()

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

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

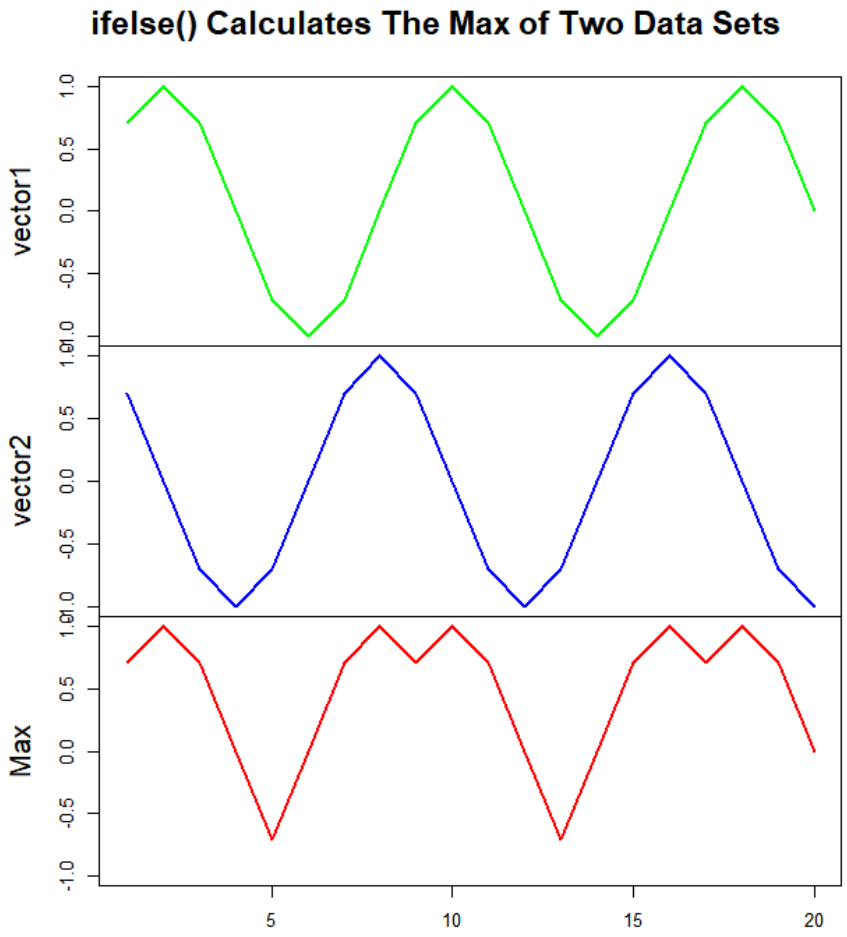
```
> # rnorm() vectorized with respect to "mean" and "sd"
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # Call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2,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

Vectorized if-else Statements Using Function ifelse()

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

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

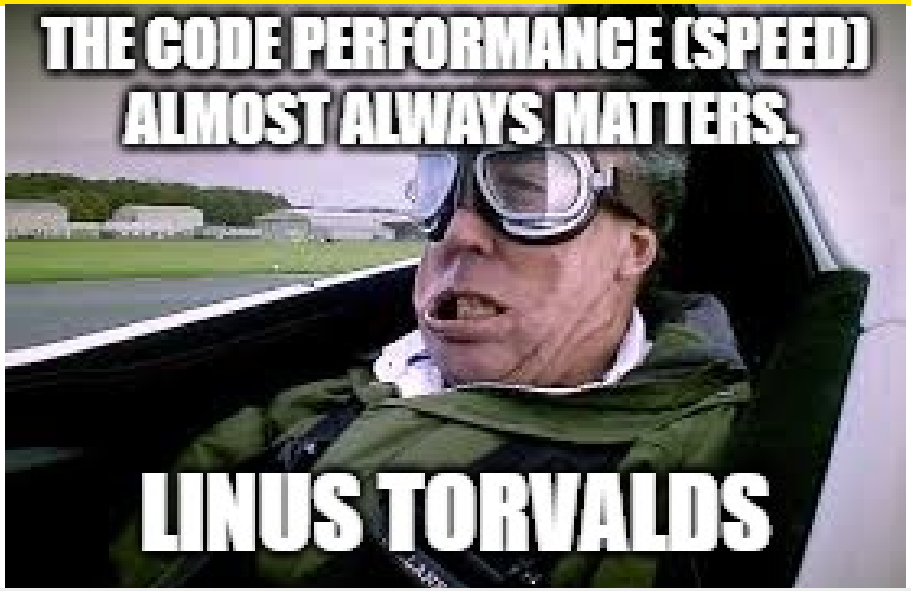
```
> # Create two numeric vectors
> vector1 <- sin(0.25*pi*1:20)
> vector2 <- cos(0.25*pi*1:20)
> # Create third vector using 'ifelse'
> vector3 <- ifelse(vector1 > vector2, vector1, vector2)
> # cbind all three together
> vector3 <- cbind(vector1, vector2, vector3)
> colnames(vector3)[3] <- "Max"
>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



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

How to write fast R code:

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




```
> # Calculate cumulative sum of a vector
> vectorv <- runif(1e5)
> # Use compiled function
> cumsumv <- cumsum(vectorv)
> # Use for loop
> cumsumv2 <- vectorv
> for (i in 2:NROW(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(vectorv),
+   loop_alloc={
+     cumsumv2 <- vectorv
+     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
> cluster <- makeCluster(ncores)
> # Perform parallel loop under Windows
> outv <- parLapply(cluster, 1:10, paws)
> # Perform parallel loop under Mac-OSX or Linux
> outv <- mclapply(1:10, paws, mc.cores=ncores)
> library(microbenchmark)  # Load package microbenchmark
> # Compare speed of lapply versus parallel computing
> summary(microbenchmark(
+   standard = lapply(1:10, paws),
+   parallel = parLapply(cluster, 1:10, paws),
+   times=10)
+ )[, c(1, 4, 5)]
```

Computing Advantage of Parallel Computing

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

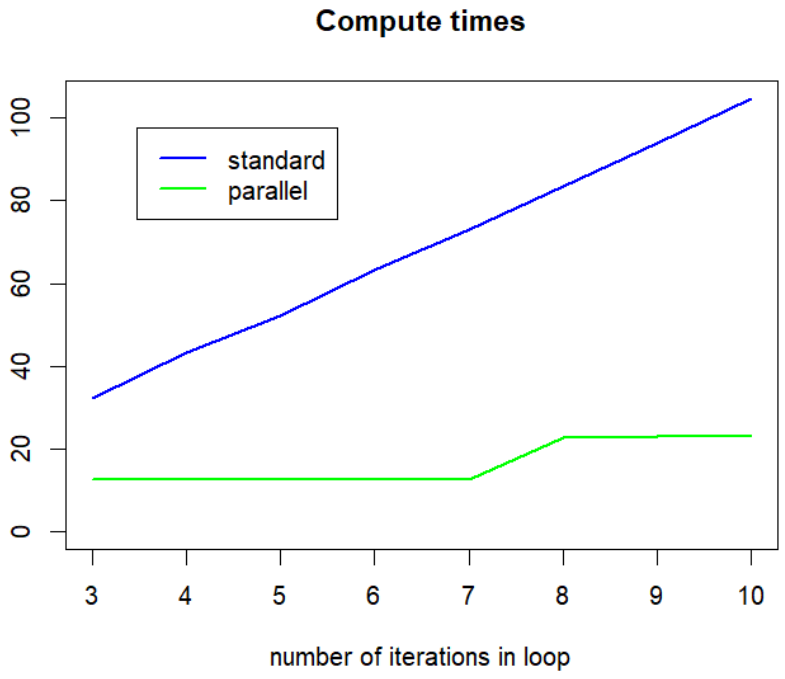
The function `stopCluster()` stops the R processes running on several CPU cores.

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

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

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

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




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

Parallel Computing Over Matrices

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

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

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

Initializing Parallel Clusters Under *Windows*

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

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

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

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

Reproducible Parallel Simulations Under *Windows*

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

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

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

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

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

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

Monte Carlo Simulation

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

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

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

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

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

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

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

Standard Errors of Estimators Using Bootstrap Simulation

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

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

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

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

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

The Distribution of Estimators Using Bootstrap Simulation

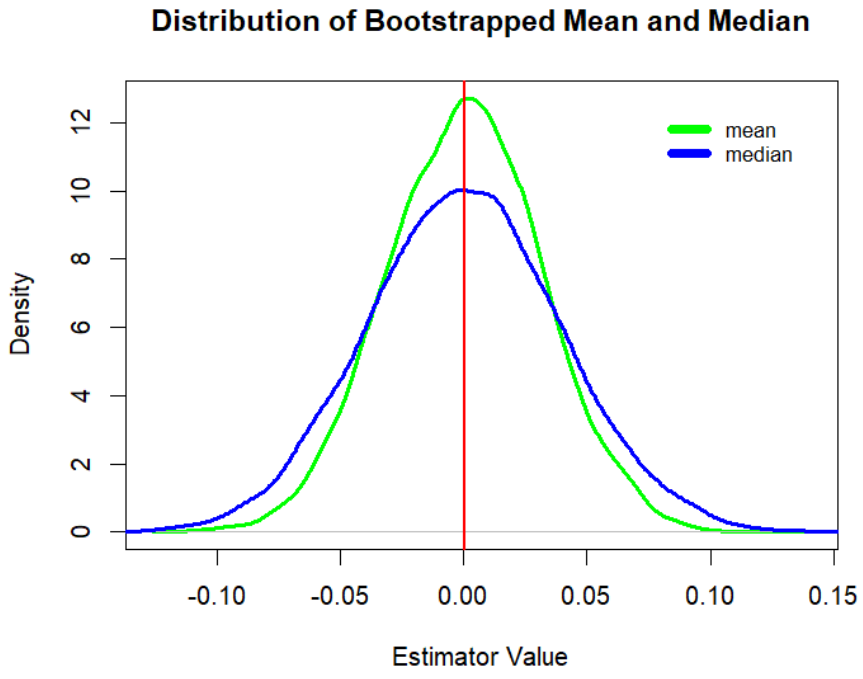
The standard errors of estimators can be calculated using a *bootstrap* simulation.

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

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

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

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




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

Bootstrapping Using Vectorized Operations

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

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

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

Faster code often requires more memory than slower code.

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

Bootstrapping Standard Errors Using Parallel Computing

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

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

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

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

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

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

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

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

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

Parallel Bootstrapping of the *Median Absolute Deviation*

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

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

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

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

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

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

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

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

Resampling From Empirical Datasets

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

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

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

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

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

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

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

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

Standard Errors of Regression Coefficients Using Bootstrap

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

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

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

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

Distribution of Bootstrapped Regression Coefficients

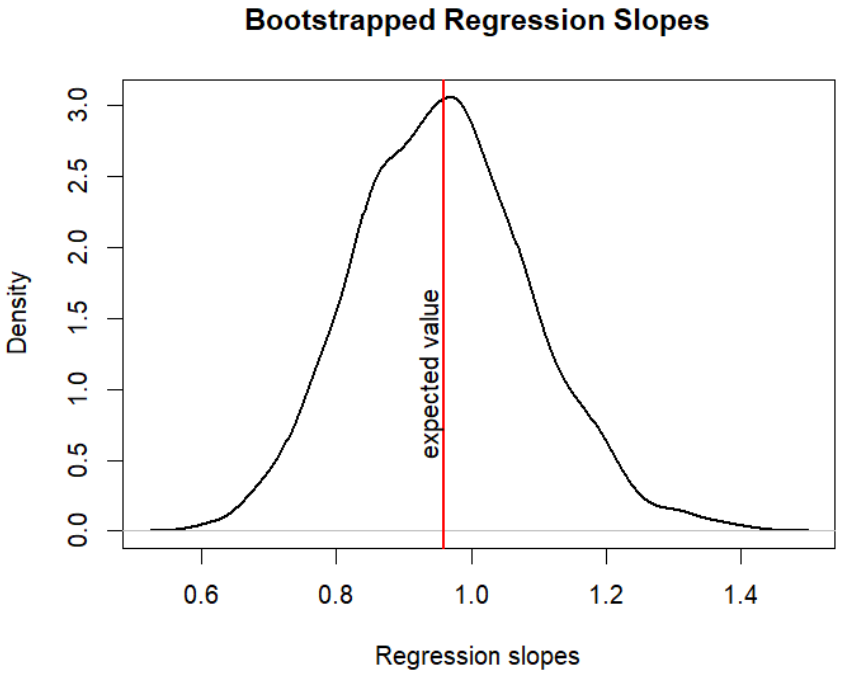
The *bootstrapped* coefficient values can be used to calculate the probability distribution of the coefficients and their standard errors,

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

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

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

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



Bootstrapping Regressions Using Parallel Computing

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

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

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

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

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

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

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

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

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

Analyzing the Bootstrap Data

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

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

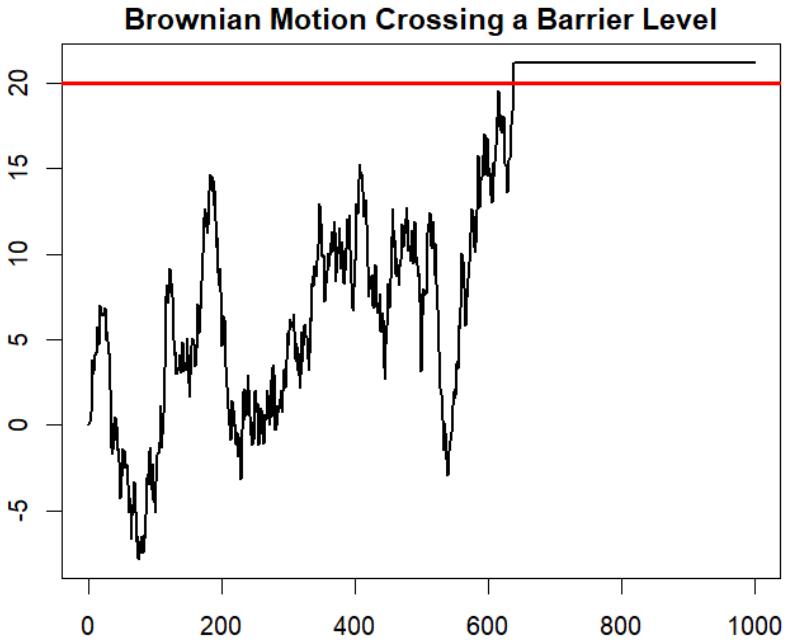
```
> # Collapse the bootstrap list into a vector
> class(bootd)
> bootd <- unlist(bootd)
> # Mean and standard error of beta regression coefficient
> c(mean=mean(bootd), stderror=sd(bootd))
> # Plot density of bootstrapped beta coefficients
> plot(density(bootd),
+      lwd=2, xlab="Regression slopes",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

Simulating Brownian Motion Using while() Loops

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

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

```
> set.seed(1121)  # Reset random number generator
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> pathv <- numeric(nrows)  # Allocate path vector
> pathv[1] <- 0  # Initialize path
> it <- 2  # Initialize simulation index
> while ((it <= nrows) && (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 <= nrows)
+   pathv[it:nrows] <- 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)
```

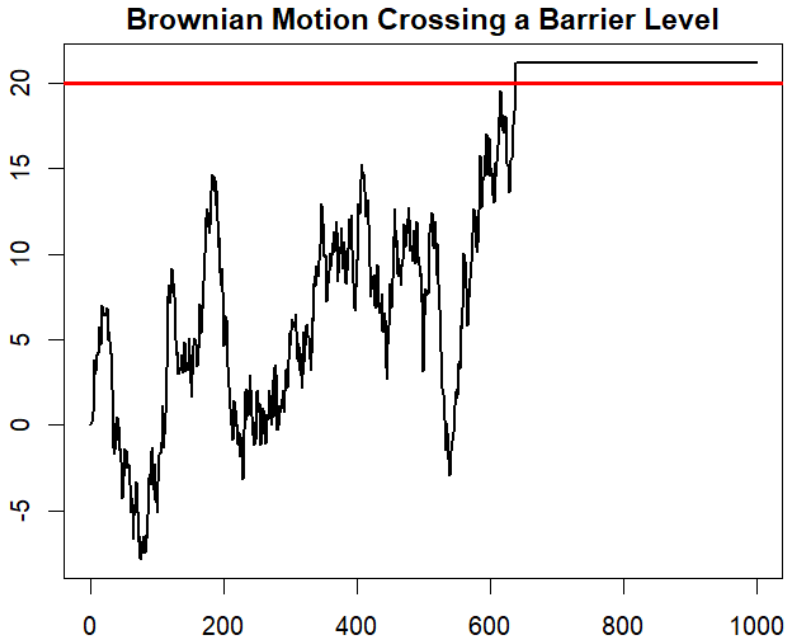


Simulating Brownian Motion Using Vectorized Functions

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.

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



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

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

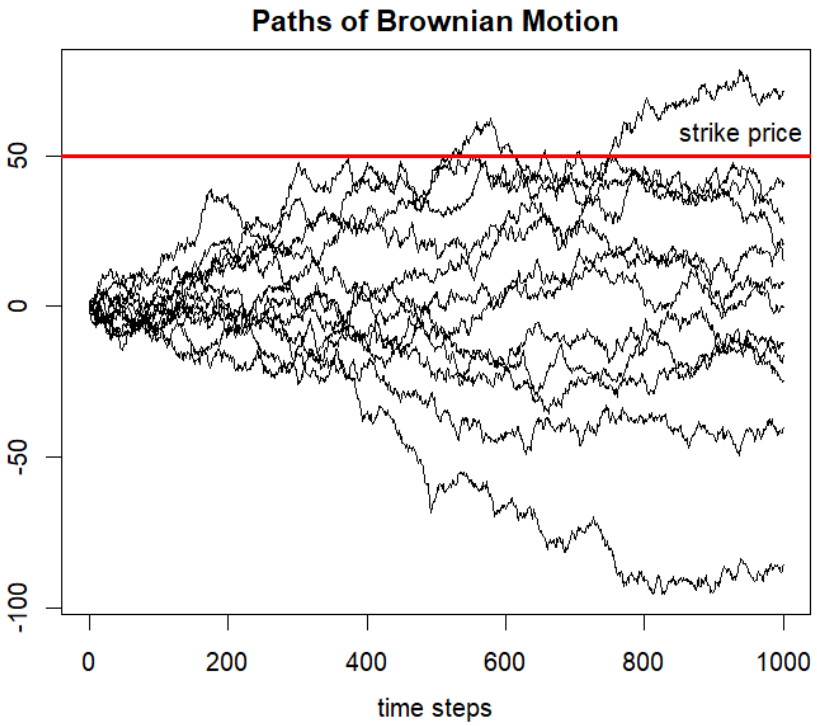
Estimating the Statistics of Brownian Motion

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

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

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

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




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

Bootstrapping From Time Series of Prices

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

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

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

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

```
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1  # Number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster under
> # Perform parallel bootstrap under Windows
> clusterSetRNGStream(cluster, 1121)  # Reset random number generato
> clusterExport(cluster, c("startd", "barl"))
> nboot <- 10000
> bootd <- parLapply(cluster, 1:nboot,
+   function(x, retsp, nrows) {
+     samplev <- retsp[sample.int(nrows, replace=TRUE)]
+     # Calculate prices from percentage returns
+     samplev <- startd*exp(cumsum(samplev))
+     # Calculate if prices crossed barrier
+     sum(samplev > barl) > 0
+   }, retsp=retsp, nrows*nrows)  # end parLapply
> # Perform parallel bootstrap under Mac-OSX or Linux
> bootd <- mclapply(1:nboot, function(x) {
+     samplev <- retsp[sample.int(nrows, replace=TRUE)]
+     # Calculate prices from percentage returns
+     samplev <- startd*exp(cumsum(samplev))
+     # Calculate if prices crossed barrier
+     sum(samplev > barl) > 0
+   }, mc.cores=ncores)  # end mclapply
> stopCluster(cluster)  # Stop R processes over cluster under Window
> bootd <- rutils::do_call(rbind, 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]
> retsp <- rutils::diffit(log(pricev))
> nrows <- NROW(retsp)
> # Calculate difference of OHLC price columns
> ohlc_diff <- ohlc[, 1:3] - pricev
> class(retsp); head(retsp)
> # Calculate bootstrap prices from percentage returns
> datav <- sample.int(nrows, replace=TRUE)
> boot_prices <- startd*exp(cumsum(retsp[datav]))
> boot_ohlc <- ohlc_diff + boot_prices
> boot_ohlc <- cbind(boot_ohlc, boot_prices)
> # Define barrier level with respect to prices
> barl <- 1.5*max(pricev)
> # Calculate if High bootstrapped prices crossed barrier level
> sum(boot_ohlc[, 2] > barl) > 0
```

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

Variance Reduction Using Antithetic Sampling

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

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

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

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

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

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

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

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

Simulating Rare Events Using Probability Tilting

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

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

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

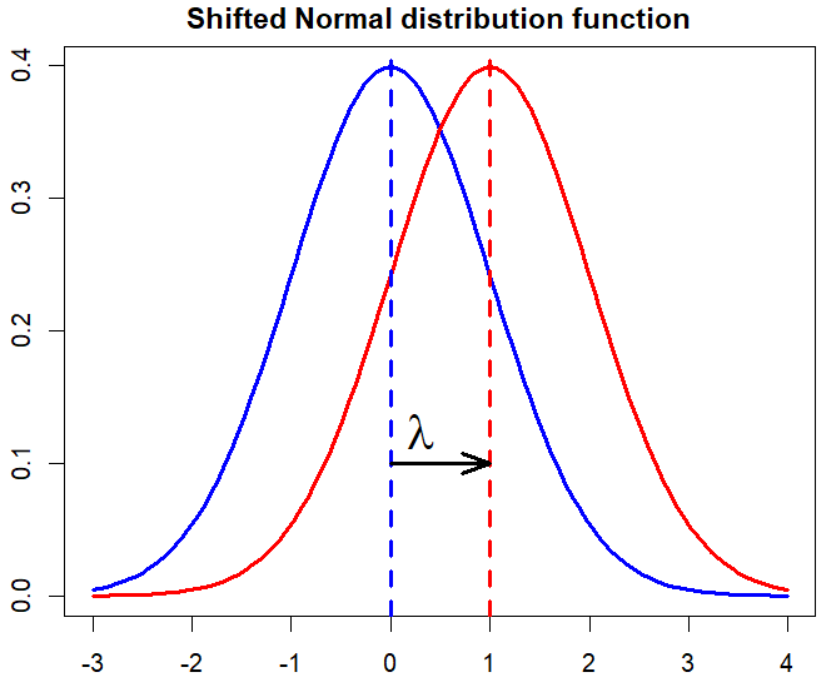
Where $p(x)$ is the probability density, $p(x,\lambda)$ is the tilted density, and λ is the tilt parameter.

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

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

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

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




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

Variance Reduction Using Importance Sampling

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

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

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

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

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

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

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

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

Calculating Quantiles Using Importance Sampling

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

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

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

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

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

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

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

```
> # Quantile from Naive Monte Carlo
> confl <- 0.02
> qnorm(confl)  # Exact value
> datav <- sort(datav)
> cutoff <- nrows*confl
> datav[cutoff]  # Naive Monte Carlo value
> # Importance sample weights
> datat <- datav + lambda  # Tilt the random numbers
> weights <- exp(-lambda*datat + lambda^2/2)
> # Cumulative probabilities using importance sample
> cumprob <- cumsum(weights)/nrows
> # 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(nrows))
+   naivemc <- datav[cutoff]
+   datat <- datav + lambda
+   weights <- exp(-lambda*datat + lambda^2/2)
+   cumprob <- cumsum(weights)/nrows
+   isample <- datat[findInterval(confl, cumprob)]
+   c(naivemc=naivemc, importmc=isample)
+ }) # end sapply
> apply(bootd, MARGIN=1,
+   function(x) c(mean=mean(x), sd=sd(x)))
```

Calculating *CVaR* Using Importance Sampling

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

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

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

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

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

The Optimal Tilt Parameter for Importance Sampling

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

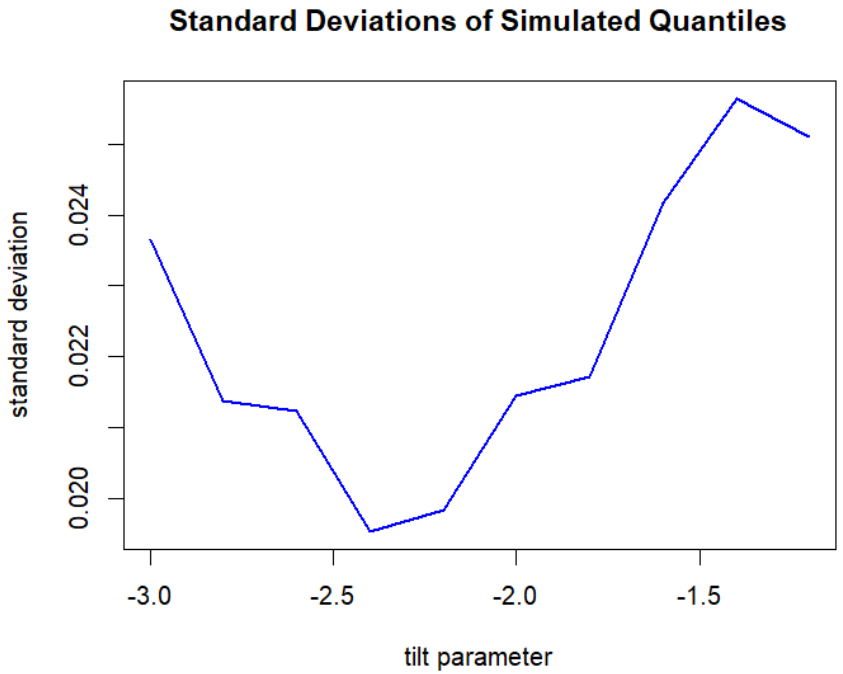
The optimal tilt parameter depends on the estimator and on the required confidence level.

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

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

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

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




```
> # Define vector of tilt parameters
> lambdav <- seq(-3.0, -1.2, by=0.2)
> # Calculate vector of quantiles for tilt parameters
> quantiles <- sapply(lambdav, calc_quant)
> # Calculate standard deviations of quantiles for tilt parameters
> stdevs <- apply(quantiles, 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 Deviations of Simulated Quantiles",
+      xlab="tilt parameter", ylab="standard deviation",
+      type="l", col="blue", lwd=2)
```

draft: Importance Sampling For Empirical Datasets

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.

Importance Sampling for Binomial Variables

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

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

Where λ is the tilt parameter.

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

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

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

Importance Sampling of Brownian Motion

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

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

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

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

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

```
> # Define Brownian motion parameters
> sigmav <- 1.0  # Volatility
> drift <- 0.0  # Drift
> nrows <- 100  # Number of simulation steps
> nsimu <- 10000  # Number of simulations
> # Calculate matrix of normal variables
> set.seed(1121)
> datav <- rnorm(nsimu*nrows, mean=drift, sd=sigmav)
> datav <- matrix(datav, nc=nsimu)
> # Simulate paths of Brownian motion
> pathm <- matrixStats::colCumsums(datav)
> # Tilt the datav
> lambda <- 0.04  # Tilt parameter
> 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) + nrows*lambda^2/2)
> # Calculate option payout using standard MC
> strikep <- 10  # Strike price
> payouts <- (pathm[nrows, ] - strikep)
> sum(payouts[payouts > 0])/nsimu
> # Calculate option payout using importance sampling
> payouts <- (patht[nrows, ] - strikep)
> sum((weightm*payouts)[payouts > 0])/nsimu
> # Calculate crossing probability using standard MC
> barl <- 10
> crossi <- (colSums(pathm > barl) > 0)
> sum(crossi)/nsimu
> # Calculate crossing probability using importance sampling
> crossi <- colSums(patht > barl) > 0
> sum(weightm*crossi)/nsimu
```

One-dimensional Optimization Using The Functional optimize()

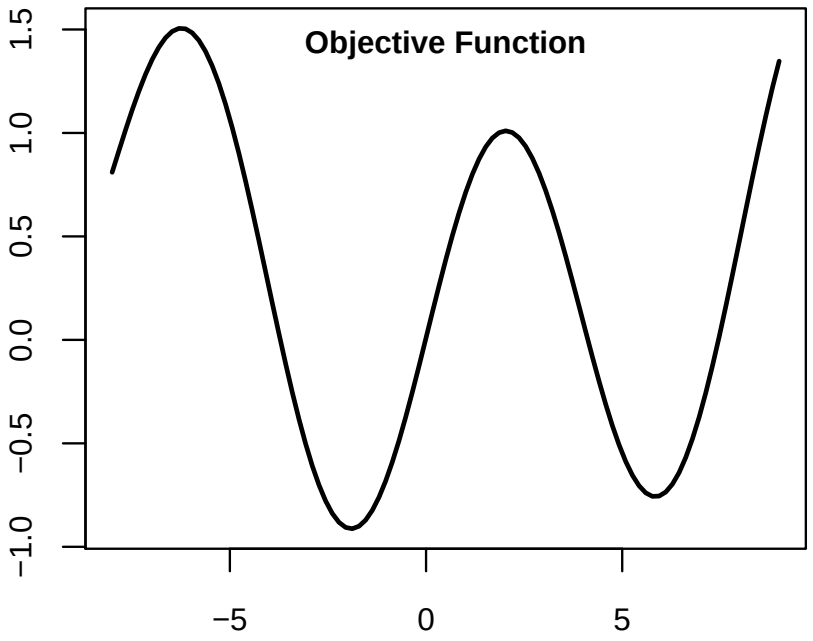
The functional `optimize()` performs *one-dimensional* optimization over a single independent variable.

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

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

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

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



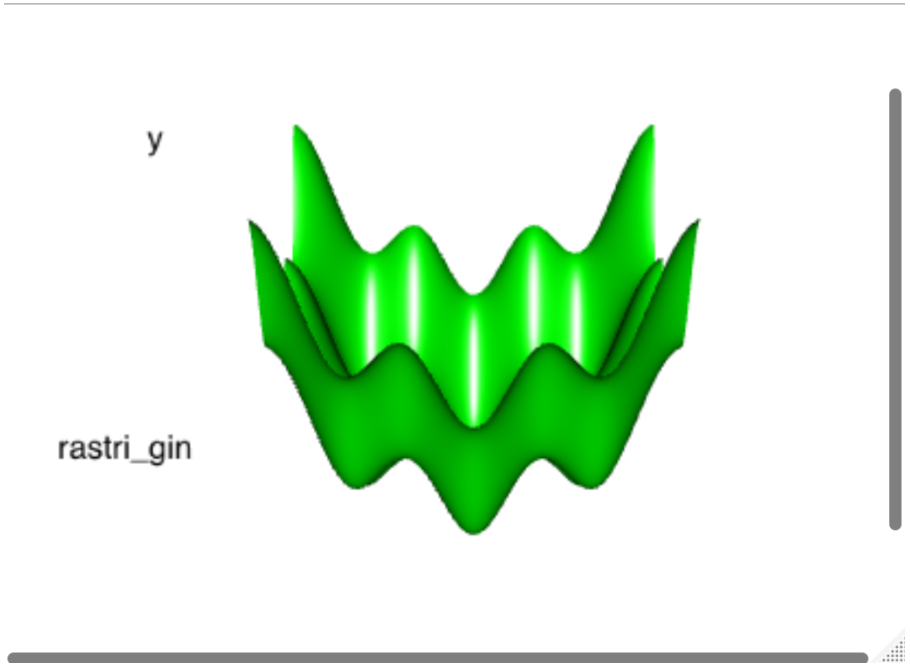

```
> # Plot the objective function
> curve(expr=objfun, type="l", xlim=c(-8, 9),
+ xlab="", ylab="", lwd=2)
> # Add title
> title(main="Objective Function", line=-1)
```

Package *rgl* for Interactive 3d Surface Plots

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

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

```
> # Rastrigin function
> rastrigin <- function(x, y, param=25) {
+   x^2 + y^2 - param*(cos(x) + cos(y))
+ }  # end rastrigin
> # Rastrigin function is vectorized!
> rastrigin(c(-10, 5), c(-10, 5))
> # Set rgl options and load package rgl
> library(rgl)
> options(rgl.useNULL=TRUE)
> # Draw 3d surface plot of function
> rgl::persp3d(x=rastrigin, xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, param=15)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=400, height=400)
```



Multi-dimensional Optimization Using optim()

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

The argument `fn` is the objective function to be minimized.

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

The argument `par` is the initial vector argument value.

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

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

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

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

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

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

The Likelihood Function

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

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

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

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

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

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

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

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

Perspective Plot of Likelihood Function

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

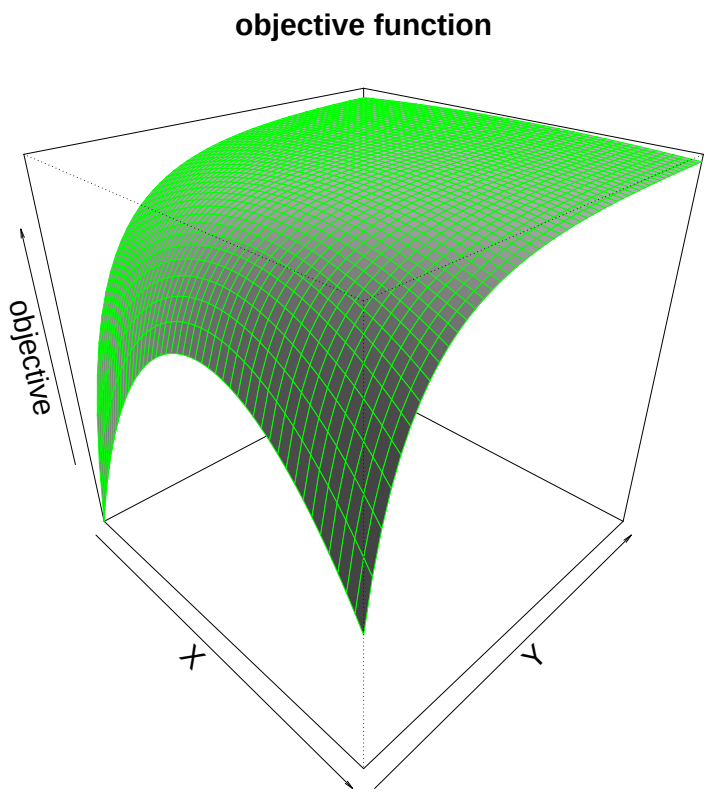
The argument `"z"` accepts a matrix containing the function values.

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

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

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

```
> # Perspective plot of log-likelihood function
> persp(z=-objective_grid,
+ theta=45, phi=30, shade=0.5,
+ border="green", zlab="objective",
+ main="objective function")
> # Interactive perspective plot of log-likelihood function
> library(rgl)  # Load package rgl
> rgl::par3d(cex=2.0)  # Scale text by factor of 2
> rgl::persp3d(z=-objective_grid, zlab="objective",
+   col="green", main="objective function")
```

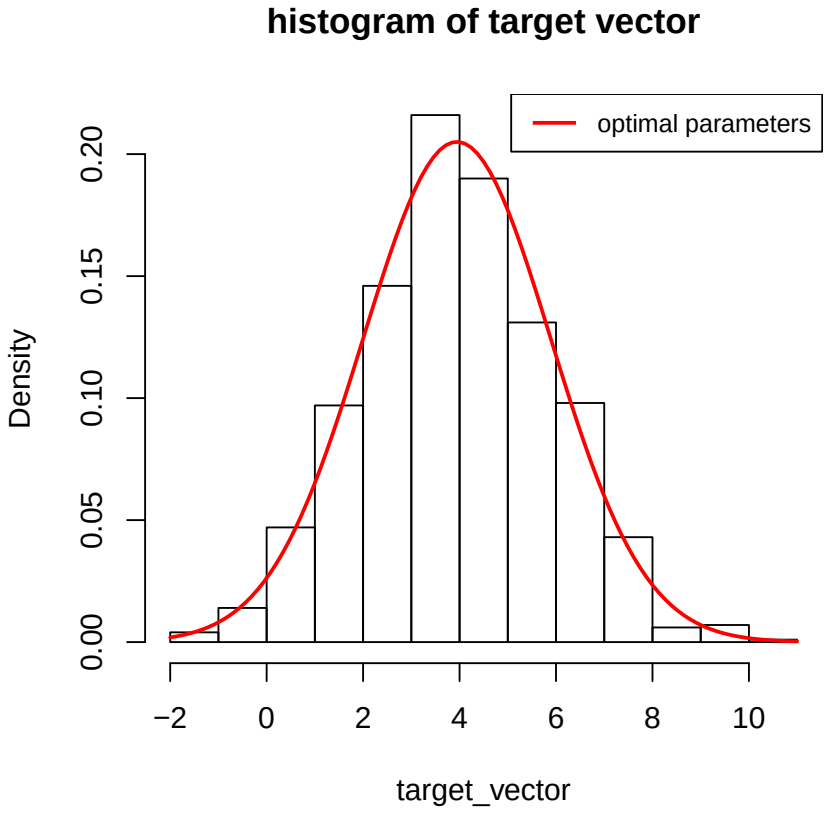


Optimization of Objective Function

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

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

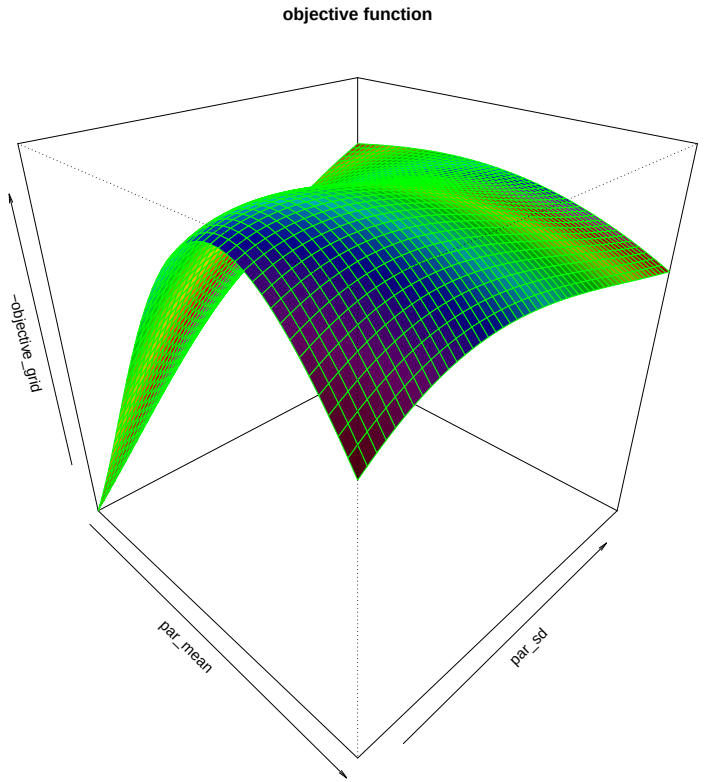
```
> # Initial parameters
> initp <- c(mean=0, sd=1)
> # Perform optimization using optim()
> optiml <- optim(par=initp,
+   fn=objfun, # Log-likelihood function
+   datav=datav,
+   method="L-BFGS-B", # Quasi-Newton method
+   upper=c(10, 10), # Upper constraint
+   lower=c(-10, 0.1)) # Lower constraint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



Mixture Model Likelihood Function

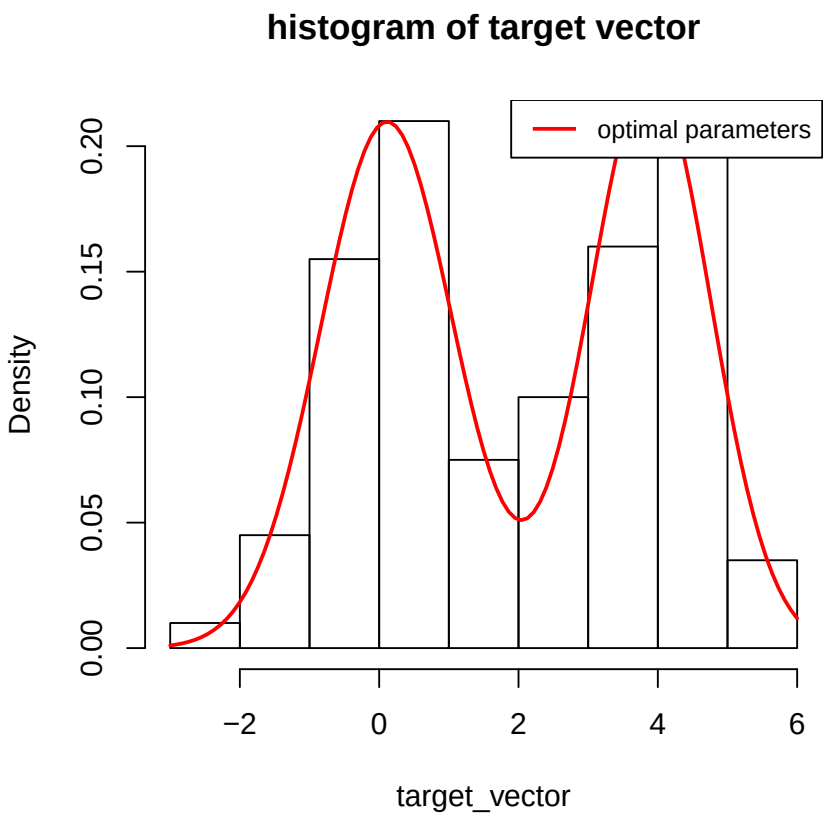
```
> # Sample from mixture of normal distributions
> datav <- c(rnorm(100, sd=1.0),
+       rnorm(100, mean=4, sd=1.0))
> # Objective function is log-likelihood
> objfun <- function(parv, datav) {
+   likev <- parv[1]/parv[3] *
+   dnorm((datav-parv[2])/parv[3]) +
+   (1-parv[1])/parv[5]*dnorm((datav-parv[4])/parv[5])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

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



Optimization of Mixture Model

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



draft: Package *ROI* Optimization Framework

The package *ROI* provides a framework for defining optimization problems and their associated constraints, and an interface to fast optimization functions.

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

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

The first generation of solutions is selected randomly.

Each new generation is obtained by combining solutions from the previous generation,

The best solutions are selected for creating the next generation.

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

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

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

Package *DEoptim* for Global Optimization

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

Differential Evolution is a genetic algorithm which evolves a population of solutions over several generations:

https://link.springer.com/content/pdf/10.1023/A:1008202821328.pdf

The first generation of solutions is selected randomly.

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

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

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

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

Package *Rcpp* for Calling C++ Programs from R

The package *Rcpp* allows calling C++ programs from R, by compiling the C++ code and creating R functions.

Rcpp functions are R functions that were compiled from C++ code using package *Rcpp*.

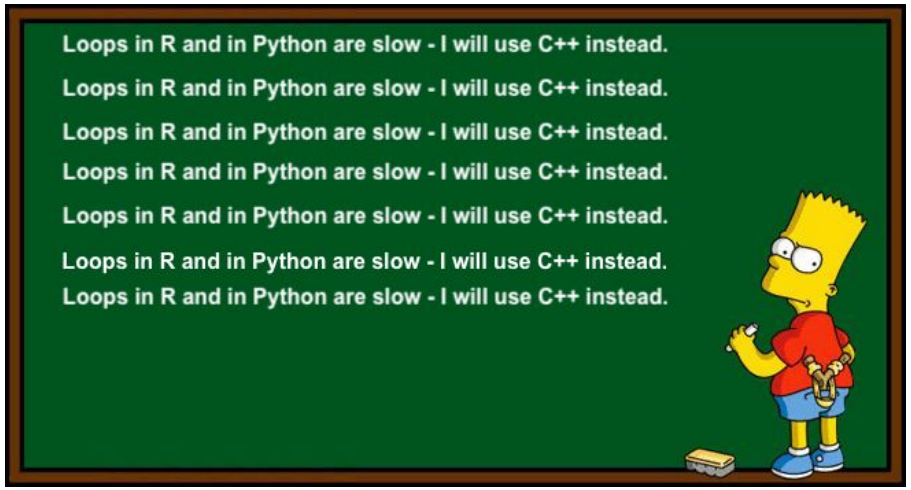
Rcpp functions are much faster than code written in R, so they're suitable for large numerical calculations.

The package *Rcpp* relies on *Rtools* for compiling the C++ code:

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

You can learn more about the package *Rcpp* here:

http://adv-r.had.co.nz/Rcpp.html
http://www.rcpp.org/
http://gallery.rcpp.org/




```
> # Verify that Rtools or XCode are working properly:
> devtools::find_rtools()  # Under Windows
> devtools::has_devel()
> # Install the packages Rcpp and RcppArmadillo
> install.packages(c("Rcpp", "RcppArmadillo"))
> # Load package Rcpp
> library(Rcpp)
> # Get documentation for package Rcpp
> # Get short description
> packageDescription("Rcpp")
> # Load help page
> help(package="Rcpp")
> # List all datasets in "Rcpp"
> data(package="Rcpp")
> # List all objects in "Rcpp"
> ls("package:Rcpp")
> # Remove Rcpp from search path
> detach("package:Rcpp")
```

Function cppFunction() for Compiling C++ code

The function cppFunction() compiles C++ code into an R function.

The function cppFunction() creates an R function only for the current R session, and it must be recompiled for every new R session.

The function sourceCpp() compiles C++ code contained in a file into R functions.

```
> # Define Rcpp function
> Rcpp::cppFunction("
+   int times_two(int x)
+     { return 2 * x;}
+   ")  # end cppFunction
> # Run Rcpp function
> times_two(3)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Multiply two numbers
> mult_rcpp(2, 3)
> mult_rcpp(1:3, 6:4)
> # Multiply two vectors
> mult_vec_rcpp(2, 3)
> mult_vec_rcpp(1:3, 6:4)
```

Performing Loops in *Rcpp Sugar*

Loops written in *Rcpp* can be two orders of magnitude faster than loops in R!

Rcpp Sugar allows using R-style vectorized syntax in *Rcpp* code.

```
> # Define Rcpp function with loop
> Rcpp::cppFunction("
+ double inner_mult(NumericVector x, NumericVector y) {
+ int xsize = x.size();
+ int ysize = y.size();
+ if (xsize != ysize) {
+     return 0;
+   } else {
+     double total = 0;
+     for(int i = 0; i < xsize; ++i) {
+ total += x[i] * y[i];
+   }
+   return total;
+   }
+ }")  # end cppFunction
> # Run Rcpp function
> inner_mult(1:3, 6:4)
> inner_mult(1:3, 6:3)
> # Define Rcpp Sugar function with loop
> Rcpp::cppFunction("
+ double inner_sugar(NumericVector x, NumericVector y) {
+   return sum(x * y);
+ }")  # end cppFunction
> # Run Rcpp Sugar function
> inner_sugar(1:3, 6:4)
> inner_sugar(1:3, 6:3)
```

```
> # Define R function with loop
> inner_multr <- function(x, y) {
+     sumv <- 0
+     for(i in 1:NROW(x)) {
+ sumv <- sumv + x[i] * y[i]
+     }
+     sumv
+ }  # end inner_multr
> # Run R function
> inner_multr(1:3, 6:4)
> inner_multr(1:3, 6:3)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=inner_multr(1:10000, 1:10000),
+   innerp=1:10000 %*% 1:10000,
+   Rcpp=inner_mult(1:10000, 1:10000),
+   sugar=inner_sugar(1:10000, 1:10000),
+   times=10))[, c(1, 4, 5)]
```

draft: *Rcpp* Examples

Adapt from:
RcppExamples.pdf

Simulating the Ornstein-Uhlenbeck Process in *Rcpp* is about 30 times faster than in R!

```
> # Define Ornstein-Uhlenbeck function in R
> nboot <- 1000
> bootd <- function(datav, nboot=nboot) {
+   bootd <- sapply(1:nboot, function(x) {
+     samplev <- datav[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   })  # end sapply
+   bootd <- t(bootd)
+   # Analyze bootstrapped variance
+   head(bootd)
+   sum(is.na(bootd))
+   # Means and standard errors from bootstrap
+   apply(bootd, MARGIN=2,
+     function(x) c(mean=mean(x), stderror=sd(x)))
+
+   retsp <- numeric(nrows)
+   pricev <- numeric(nrows)
+   pricev[1] <- eq_price
+   for (i in 2:nrows) {
+     retsp[i] <- thetav*(eq_price - pricev[i-1]) + volat*rnorm(1)
+     pricev[i] <- pricev[i-1] + retsp[i]
+   }  # end for
+   pricev
+ }  # end bootd
```

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namesvpace

// This is a simple example of exporting a C++ function
// You can source this function into an R session using
// function Rcpp::sourceCpp()
// (or via the Source button on the editor toolbar).
// Learn more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/

// function unifun() produces a vector of
// uniformly distributed pseudo-random numbers
// [[Rcpp::export]]
NumericVector unifuncpp(double seedv, int(nrows) {
// define pi
static const double pi = 3.14159265;
// allocate output vector
  NumericVector output(nrows);
// initialize output vector
  output[0] = seedv;
// perform loop
  for (int i=1; i < nrows; ++i) {
    output[i] = 4*output[i-1]*(1-output[i-1]);
  }  // end for
// rescale output vector and return it
  return acos(1-2*output)/pi;
}
```

Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

Simulating the Ornstein-Uhlenbeck Process in *Rcpp* is about 30 times faster than in R!

```
> # Define Ornstein-Uhlenbeck function in R
> sim_our <- function(nrows=1000, eq_price=5.0,
+               volat=0.01, theta=0.01) {
+   retsp <- numeric(nrows)
+   pricev <- numeric(nrows)
+   pricev[1] <- eq_price
+   for (i in 2:nrows) {
+     retsp[i] <- theta*(eq_price - pricev[i-1]) + volat*rnorm(1)
+     pricev[i] <- pricev[i-1] + retsp[i]
+   }  # end for
+   pricev
+ }  # end sim_our
> # Simulate Ornstein-Uhlenbeck process in R
> eq_price <- 5.0; sigmav <- 0.01
> thetav <- 0.01; nrows <- 1000
> set.seed(1121)  # Reset random numbers
> ousim <- sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=tl
```

```
> # Define Ornstein-Uhlenbeck function in Rcpp
> Rcpp::cppFunction("
+ NumericVector sim_oucpp(double eq_price,
+                     double volat,
+                     double thetav,
+                     NumericVector innov) {
+   int nrows = innov.size();
+   NumericVector prices(nrows);
+   NumericVector returns(nrows);
+   prices[0] = eq_price;
+   for (int it = 1; it < nrows; it++) {
+     returns[it] = thetav*(eq_price - prices[it-1]) + volat*innov[i
+     prices[it] = prices[it-1] + returns[it];
+   }  // end for
+   return prices;
+ }")  # end cppFunction
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121)  # Reset random numbers
> oucpp <- sim_oucpp(eq_price=eq_price,
+   volat=sigmav, theta=thetav, innov=rnorm(nrows))
> all.equal(ousim, oucpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=thet
+   Rcpp=sim_oucpp(eq_price=eq_price, volat=sigmav, theta=thetav, in
+   times=10))[, c(1, 4, 5)]
```

Rcpp Attributes

Rcpp attributes are instructions for the C++ compiler, embedded in the *Rcpp* code as C++ comments, and preceded by the "//" symbol.

The Rcpp::depends attribute specifies additional C++ library dependencies.

The Rcpp::export attribute specifies that a function should be exported to R, where it can be called as an R function.

Only functions which are preceded by the Rcpp::export attribute are exported to R.

The function sourceCpp() compiles C++ code contained in a file into R functions.

```
> # Source Rcpp function for Ornstein-Uhlenbeck process from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> set.seed(1121)  # Reset random numbers
> oucpp <- sim_oucpp(eq_price=eq_price,
+   volat=sigmav,
+   theta=thetav,
+   innov=rnorm(nrows))
> all.equal(ousim, oucpp)
> # Compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=thetav),
+   Rcpp=sim_oucpp(eq_price=eq_price, volat=sigmav, theta=thetav, innov=rnorm(nrows)),
+   times=10))[, c(1, 4, 5)]
```

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namesvpace

// The function sim_oucpp() simulates an Ornstein-Uhlenb
// export the function roll_maxmin() to R
// [[Rcpp::export]]
NumericVector sim_oucpp(double eq_price,
                        double volat,
                        double thetav,
                        NumericVector innov) {
  int(nrows = innov.size();
  NumericVector pricev*nrows);
  NumericVector retsp*nrows);
  pricev[0] = eq_price;
  for (int it = 1; it < nrows; it++) {
    retsp[it] = thetav*(eq_price - pricev[it-1]) + volat
    pricev[it] = pricev[it-1] + retsp[it];
  }  // end for
  return pricev;
}  // end sim_oucpp
```

Generating Random Numbers Using Logistic Map in *Rcpp*

The *logistic map* in *Rcpp* is about seven times faster than the loop in R, and even slightly faster than the standard `runif()` function in R!

```
> # Calculate uniformly distributed pseudo-random sequence
> unifun <- function(seedv, nrows=10) {
+   output <- numeric(nrows)
+   output[1] <- seedv
+   for (i in 2:nrows) {
+     output[i] <- 4*output[i-1]*(1-output[i-1])
+   }  # end for
+   acos(1-2*output)/pi
+ }  # end unifun
>
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Microbenchmark Rcpp code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=runif(1e5),
+   rloop=unifun(0.3, 1e5),
+   Rcpp=unifuncpp(0.3, 1e5),
+   times=10))[, c(1, 4, 5)]
```

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namesvpace

// This is a simple example of exporting a C++ function
// You can source this function into an R session using
// function Rcpp::sourceCpp()
// (or via the Source button on the editor toolbar).
// Learn more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/

// function unifun() produces a vector of
// uniformly distributed pseudo-random numbers
// [[Rcpp::export]]
NumericVector unifuncpp(double seedv, int(nrows) {
// define pi
static const double pi = 3.14159265;
// allocate output vector
  NumericVector output(nrows);
// initialize output vector
  output[0] = seedv;
// perform loop
  for (int i=1; i < nrows; ++i) {
    output[i] = 4*output[i-1]*(1-output[i-1]);
  }  // end for
// rescale output vector and return it
  return acos(1-2*output)/pi;
}
```

draft: Bootstrap Simulation Using *Rcpp*

Simulating the Ornstein-Uhlenbeck Process in *Rcpp* is about 30 times faster than in R!

```
> # Define Ornstein-Uhlenbeck function in R
> bootd <- function(datav, nboot=1000) {
+   bootd <- sapply(1:nboot, function(x) {
+     samplev <- datav[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   })  # end sapply
+   bootd <- t(bootd)
+   # Analyze bootstrapped variance
+   head(bootd)
+   sum(is.na(bootd))
+   # Means and standard errors from bootstrap
+   apply(bootd, MARGIN=2,
+     function(x) c(mean=mean(x), stderror=sd(x)))
+
+   retsp <- numeric(nrows)
+   pricev <- numeric(nrows)
+   pricev[1] <- eq_price
+   for (i in 2:nrows) {
+     retsp[i] <- thetav*(eq_price - pricev[i-1]) + volat*rnorm(1)
+     pricev[i] <- pricev[i-1] + retsp[i]
+   }  # end for
+   pricev
+ }  # end bootd
> # Simulate Ornstein-Uhlenbeck process in R
> eq_price <- 5.0; sigmav <- 0.01
> thetav <- 0.01; nrows <- 1000
> set.seed(1121)  # Reset random numbers
> ousim <- sim_our(nrows, eq_price=eq_price, volat=sigmav, theta=thetav)
```

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namesvpace

// This is a simple example of exporting a C++ function
// You can source this function into an R session using
// function Rcpp::sourceCpp()
// (or via the Source button on the editor toolbar).
// Learn more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/

// function unifun() produces a vector of
// uniformly distributed pseudo-random numbers
// [[Rcpp::export]]
NumericVector unifuncpp(double seedv, int(nrows) {
// define pi
static const double pi = 3.14159265;
// allocate output vector
  NumericVector output(nrows);
// initialize output vector
  output[0] = seedv;
// perform loop
  for (int i=1; i < nrows; ++i) {
    output[i] = 4*output[i-1]*(1-output[i-1]);
  }  // end for
// rescale output vector and return it
  return acos(1-2*output)/pi;
}
```

draft: Converting Functions and Objects Between C++ and R

C++ functions and objects need to be converted to R objects, and vice versa.

The function Rcpp::wrap() converts C++ functions and objects to R objects.

The syntax as<T>() converts R objects to C++ objects.

Adapt from:
Rcpp-modules.pdf
https://gallery.rcpp.org/articles/custom-templated-wrap-and-as-for-seamingless-interfaces/
http://dirk.eddelbuettel.com/code/rcpp.html

Simulating the Ornstein-Uhlenbeck Process in *Rcpp* is about 30 times faster than in R!

```
> # Define Ornstein-Uhlenbeck function in R
> bootd <- function(datav, nboot=1000) {
+   bootd <- sapply(1:nboot, function(x) {
+     samplev <- datav[sample.int(nrows, replace=TRUE)]
+     c(sd=sd(samplev), mad=mad(samplev))
+   })  # end sapply
+   bootd <- t(bootd)
+   # Analyze bootstrapped variance
+   head(bootd)
+   sum(is.na(bootd))
+   # Means and standard errors from bootstrap
+   apply(bootd, MARGIN=2,
+   function(x) c(mean=mean(x), stderror=sd(x)))
+
+   retsp <- numeric(nrows)
+   pricev <- numeric(nrows)
+   pricev[1] <- eq_price
+   for (i in 2:nrows) {
+     retsp[i] <- thetav*(eq_price - pricev[i-1]) + volat*rnorm(1)
```

```
// Rcpp header with information for C++ compiler
#include <Rcpp.h> // include Rcpp C++ header files
using namespace Rcpp; // use Rcpp C++ namesvpace

// This is a simple example of exporting a C++ function
// You can source this function into an R session using
// function Rcpp::sourceCpp()
// (or via the Source button on the editor toolbar).
// Learn more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/

// function unifun() produces a vector of
// uniformly distributed pseudo-random numbers
// [[Rcpp::export]]
NumericVector unifuncpp(double seedv, int(nrows) {
// define pi
static const double pi = 3.14159265;
// allocate output vector
  NumericVector output(nrows);
// initialize output vector
  output[0] = seedv;
// perform loop
  for (int i=1; i < nrows; ++i) {
    output[i] = 4*output[i-1]*(1-output[i-1]);
  }  // end for
// rescale output vector and return it
  return acos(1-2*output)/pi;
}
```

Package *RcppArmadillo* for Fast Linear Algebra

The package *RcppArmadillo* allows calling the high-level *Armadillo* C++ linear algebra library.

Armadillo provides ease of use and speed, with syntax similar to *Matlab*.

RcppArmadillo functions are often faster than even compiled R functions, because they use better optimized C++ code:
http://arma.sourceforge.net/speed.html

You can learn more about *RcppArmadillo*:
http://arma.sourceforge.net/
http://dirk.eddelbuettel.com/code/rcpp.armadillo.html
https://cran.r-project.org/web/packages/\emph{RcppArmadillo}/index.html
https://github.com/RcppCore/\emph{RcppArmadillo}

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts
> vec1 <- runif(1e5)
> vec2 <- runif(1e5)
> inner_vec(vec1, vec2)
> vec1 %*% vec2
```

```
// Rcpp header with information for C++ compiler
#include <\emph{RcppArmadillo}.h>
using namespace Rcpp;
using namespace arma;
// [[Rcpp::depends(\emph{RcppArmadillo})]]

// The function inner_vec() calculates the inner (dot) p
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_vec(arma::vec vec1, arma::vec vec2) {
  return arma::dot(vec1, vec2);
}  // end inner_vec

// The function inner_mat() calculates the inner (dot) p
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_mat(const arma::vec& vectorv2, const arma::
  return arma::as_scalar(trans(vectorv2) * (matrixv * ve
}  // end inner_mat
```

```
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcpp = inner_vec(vec1, vec2),
+   rcode = (vec1 %*% vec2),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Microbenchmark shows:
> # inner_vec() is several times faster than %*%, especially for lo
> #     expr     mean   median
> # 1 inner_vec 110.7067 110.4530
> # 2 rcode 585.5127 591.3575
```

Simulating *ARIMA* Processes Using *RcppArmadillo*

ARIMA processes can be simulated using *RcppArmadillo* even faster than by using the function `filter()`.

```
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> # Define AR(2) coefficients
> coeff <- c(0.9, 0.09)
> nrows <- 1e4
> set.seed(1121)
> innov <- rnorm(nrows)
> # Simulate ARIMA using filter()
> arimar <- filter(x=innov, filter=coeff, method="recursive")
> # Simulate ARIMA using sim_ar()
> innov <- matrix(innov)
> coeff <- matrix(coeff)
> arimav <- sim_ar(coeff, innov)
> all.equal(drop(arimav), as.numeric(arimar))
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcpp = sim_ar(coeff, innov),
+   filter = filter(x=innov, filter=coeff, method="recursive"),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
// Rcpp header with information for C++ compiler
#include <\emph{RcppArmadillo}.h> // include C++ header
using namespace arma; // use C++ namesvpace from Armadil
// declare dependency on \emph{RcppArmadillo}
// [[Rcpp::depends(\emph{RcppArmadillo})]]

//' @export
// [[Rcpp::export]]
arma::vec sim_ar(const arma::vec& innov, const arma::vec
  uword nrows = innov.n_elem;
  uword look_back = coeff.n_elem;
  arma::vec arimav[nrows);

  // startup period
  arimav(0) = innov(0);
  arimav(1) = innov(1) + coeff(look_back-1) * arimav(0);
  for (uword it = 2; it < look_back-1; it++) {
    arimav(it) = innov(it) + arma::dot(coeff.subvec(look
  }  // end for

  // remaining periods
  for (uword it = look_back; it < nrows; it++) {
    arimav(it) = innov(it) + arma::dot(coeff, arimav.sub
  }  // end for

  return arimav;
}  // end sim_arima
```

Fast Matrix Algebra Using *RcppArmadillo*

RcppArmadillo functions can be made even faster by operating on pointers to matrices and performing calculations in place, without copying large matrices.

RcppArmadillo functions can be compiled using the same *Rtools* as those for *Rcpp* functions:
https://cran.r-project.org/bin/windows/Rtools/

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/lecture_slides/scripts/
> matrixv <- matrix(runif(1e5), nc=1e3)
> # De-mean using apply()
> matr <- apply(matrixv, 2, function(x) (x-mean(x)))
> # De-mean in place using Rcpp demean_mat()
> demean_mat(matrixv)
> all.equal(matr, matrixv)
> # Microbenchmark \emph{RcppArmadillo} code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = (apply(matrixv, 2, mean)),
+   rcpp = demean_mat(matrixv),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # Perform matrix inversion
> # Create random positive semi-definite matrix
> matrixv <- matrix(runif(25), nc=5)
> matrixv <- t(matrixv) %*% matrixv
> # Invert the matrix
> matrixinv <- solve(matrixv)
> inv_mat(matrixv)
> all.equal(matrixinv, matrixv)
> # Microbenchmark \emph{RcppArmadillo} code
> summary(microbenchmark(
+   rcode = solve(matrixv),
+   rcpp = inv_mat(matrixv),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
// Rcpp header with information for C++ compiler
#include <\emph{RcppArmadillo}.h> // include C++ header
using namespace arma; // use C++ namesvpace from Armadil
// declare dependency on \emph{RcppArmadillo}
// [[Rcpp::depends(\emph{RcppArmadillo})]]

// Examples of \emph{RcppArmadillo} functions below

// The function demean_mat() calculates a matrix with de
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
int demean_mat(arma::mat& matrixv) {
  for (uword i = 0; i < matrixv.n_cols; i++) {
    matrixv.col(i) -= arma::mean(matrixv.col(i));
  }  // end for
  return matrixv.n_cols;
}  // end demean_mat

// The function inv_mat() calculates the inverse of symm
// definite matrix.
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inv_mat(arma::mat& matrixv) {
  matrixv = arma::inv_sympd(matrixv);
  return matrixv.n_cols;
}  // end inv_mat
```

Fast Correlation Matrix Inverse Using *RcppArmadillo*

RcppArmadillo can be used to quickly calculate the regularized inverse of correlation matrices.

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp("/Users/jerzy/Develop/lecture_slides/scripts/High
> # Calculate matrix of random returns
> matrixv <- matrix(rnorm(300), nc=5)
> # Regularized inverse of correlation matrix
> dimax <- 4
> cormat <- cor(matrixv)
> eigend <- eigen(cormat)
> invmat <- eigend$vectors[, 1:dimax] %*%
+   (t(eigend$vectors[, 1:dimax]) / eigend$values[1:dimax])
> # Regularized inverse using \emph{RcppArmadillo}
> invarma <- calc_inv(cormat, dimax=dimax)
> all.equal(invmat, invarma)
> # Microbenchmark \emph{RcppArmadillo} code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode = {eigend <- eigen(cormat)
+ eigend$vectors[, 1:dimax] %*% (t(eigend$vectors[, 1:dimax]) / eige
+   rcpp = calc_inv(cormat, dimax=dimax),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

```
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(\emph{RcppArmadillo})]]
#include <\emph{RcppArmadillo}.h>
// include Rcpp C++ header files
using namespace std;
using namespace Rcpp; // use Rcpp C++ namesvpace
using namespace arma;

//' @export
// [[Rcpp::export]]
arma::mat calc_inv(const arma::mat& tseries,
                   double eigen_thresh = 0.01,
                   arma::uword dimax = 0) {

  // Allocate SVD variables
  arma::vec svdval;  // Singular values
  arma::mat svdu, svdv;  // Singular matrices
  // Calculate the SVD
  arma::svd(svdu, svdval, svdv, tseries);
  // Calculate the number of non-small singular values
  arma::uword svdnum = arma::sum(svdval > eigen_thresh*a

  // If no regularization then set dimax to (svdnum - 1)
  if (dimax == 0) {
    // Set dimax
    dimax = svdnum - 1;
  } else {
    // Adjust dimax
    dimax = std::min(dimax - 1, svdnum - 1);
  }  // end if

  // Remove all small singular values
  svdval = svdval.subvec(0, dimax);
  svdu = svdu.cols(0, dimax);
  svdv = svdv.cols(0, dimax);

  // Calculate the regularized inverse from the SVD deco
  return svdv*arma::diagmat(1/svdval)*svdu.t();
```

Portfolio Optimization Using *RcppArmadillo*

Fast portfolio optimization using matrix algebra can be implemented using *RcppArmadillo*.

```
// Fast portfolio optimization using matrix algebra and \emph{RcppArmadillo}
arma::vec calc_weights(const arma::mat& returns, // Asset returns
                       Rcpp::List controlv) { // List of portfolio optimization parameters

  // Apply different calculation methods for weights
  switch(calc_method(method)) {
  case methodenum::maxsharpe: {
    // Mean returns of columns
    arma::vec colmeans = arma::trans(arma::mean(returns, 0));
    // Shrink colmeans to the mean of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::mean(colmeans));
    // Calculate weights using regularized inverse
    weights = calc_inv(covmat, eigen_thresh, dimax)*colmeans;
    break;
  }  // end maxsharpe
  case methodenum::maxsharpemed: {
    // Median returns of columns
    arma::vec colmeans = arma::trans(arma::median(returns, 0));
    // Shrink colmeans to the median of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::median(colmeans));
    // Calculate weights using regularized inverse
    weights = calc_inv(covmat, eigen_thresh, dimax)*colmeans;
    break;
  }  // end maxsharpemed
  case methodenum::minvarlin: {
    // Minimum variance weights under linear constraint
    // Multiply regularized inverse times unit vector
    weights = calc_inv(covmat, eigen_thresh, dimax)*arma::ones(ncols);
    break;
  }  // end minvarlin
  case methodenum::minvarquad: {
    // Minimum variance weights under quadratic constraint
    // Calculate highest order principal component
    arma::vec eigenval;
    arma::mat eigenvec;
```

Strategy Backtesting Using *RcppArmadillo*

Fast backtesting of strategies can be implemented using *RcppArmadillo*.

```
arma::mat back_test(const arma::mat& excess, // Asset excess returns
                    const arma::mat& returns, // Asset returns
                    Rcpp::List controlv, // List of portfolio optimization model parameters
                    arma::uvec startp, // Start points
                    arma::uvec endd, // End points
                    double lambda = 0.0, // Decay factor for averaging the portfolio weights
                    double coeff = 1.0, // Multiplier of strategy returns
                    double bid_offer = 0.0) { // The bid-offer spread

  double lambda1 = 1-lambda;
  arma::uword nweights = returns.n_cols;
  arma::vec weights(nweights, fill::zeros);
  arma::vec weights_past = ones(nweights)/std::sqrt(nweights);
  arma::mat pnls = zeros(returns.n_rows, 1);

  // Perform loop over the end points
  for (arma::uword it = 1; it < endd.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate the portfolio weights
    weights = coeff*calc_weights(excess.rows(startp(it-1), endd(it-1)), controlv);
    // Calculate the weights as the weighted sum with past weights
    weights = lambda1*weights + lambda*weights_past;
    // Calculate out-of-sample returns
    pnls.rows(endd(it-1)+1, endd(it)) = returns.rows(endd(it-1)+1, endd(it))*weights;
    // Add transaction costs
    pnls.row(endd(it-1)+1) -= bid_offer*sum(abs(weights - weights_past))/2;
    // Copy the weights
    weights_past = weights;
  }  // end for

  // Return the strategy pnls
  return pnls;

}  // end back_test
```

Package *reticulate* for Running Python from RStudio

The package *reticulate* allows running Python functions and scripts from RStudio.

The package *reticulate* relies on Python for interpreting the Python code.

```
> # Install package reticulate
> install.packages("reticulate")
> # Start Python session
> reticulate::repl_python()
> # Exit Python session
> exit
```

You must set your Global Options in RStudio to your Python executable, for example:
/Library/Frameworks/Python.framework/Versions/3.10/bin/python3.10

You can learn more about the package *reticulate* here:
https://rstudio.github.io/reticulate/

Running Python Under *reticulate*

```
"""
Script for loading OHLC data from a CSV file and plotting a candlestick plot.
"""
# Import packages
import pandas as pd
import numpy as np
import plotly.graph_objects as go
# Load OHLC data from csv file - the time index is formatted inside read_csv()
symbol = "SPY"
range = "day"
filename = "/Users/jerzy/Develop/data/" + symbol + "_" + range + ".csv"
ohlc = pd.read_csv(filename)
datev = ohlc.Date
# Calculate log stock prices
ohlc[["Open", "High", "Low", "Close"]] = np.log(ohlc[["Open", "High", "Low", "Close"]])
# Calculate moving average
lookback = 55
closep = ohlc.Close
pricema = closep.ewm(span=lookback, adjust=False).mean()
# Plotly simple candlestick with moving average
# Create empty graph object
plotfig = go.Figure()
# Add trace for candlesticks
plotfig = plotfig.add_trace(go.Candlestick(x=datev,
  open=ohlc.Open, high=ohlc.High, low=ohlc.Low, close=ohlc.Close,
  name=symbol+" Log OHLC Prices", showlegend=False))
# Add trace for moving average
plotfig = plotfig.add_trace(go.Scatter(x=datev, y=pricema,
  name="Moving Average", line=dict(color="blue")))
# Customize plot
plotfig = plotfig.update_layout(title=symbol + " Log OHLC Prices",
  title_font_size=24, title_font_color="blue", yaxis_title="Price",
  font_color="black", font_size=18, xaxis_rangeslider_visible=False)
# Customize legend
plotfig = plotfig.update_layout(legend=dict(x=0.2, y=0.9, traceorder="normal",
  itemsizing="constant", font=dict(family="sans-serif", size=18, color="blue")))
# Render the plot
plotfig.show()
```