Numerical Analysis

FRE6871 & FRE7241, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

February 7, 2018

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*,

```
> foo <- 0.3/3
> foo  # printed as "0.1"
> foo - 0.1  # foo is not equal to "0.1"
> foo == 0.1  # foo is not equal to "0.1"
> print(foo, digits=10)
> print(foo, digits=16)
> # foo is equal to "0.1" within machine precisi
> all.equal(foo, 0.1)
> foo <- (3-2.9)
> print(foo, digits=20)
> # info machine precision of computer R is runn
> # ?.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,

```
> foo <- sqrt(2)
> foo^2  # printed as "2"
> foo^2 == 2  # foo^2 is not equal to "2"
> print(foo^2, digits=20)
> # foo^2 is equal to "2" within machine precisi
> all.equal(foo^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*,

```
> num_var <- 2
> num_var==2
> identical(num_var, 2)
>
> identical(num_var, NULL)
> # this doesn't work:
> # num_var==NULL
> is.null(num_var)
>
> vec_tor <- c(2, 4, 6)
> vec_tor==2
> identical(vec_tor, 2)
>
> # num_ber is equal to "1.0" within machine pre
> num_ber <- 1.0 + 2*sqrt(.Machine$double.eps)
> all.equal(num_ber, 1.0)
>
> # info machine precision of computer R is runn
> # ?.Machine
> # machine precision
> .Machine$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 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)
```

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
> object.size(runif(1e6))
> format(object.size(runif(1e6)), units="MB")
> # get sizes of objects in workspace
> sort(sapply(ls(),
+   function(ob_ject) {
+     format(object.size(get(ob_ject)), units="K
> # get sizes of objects in workspace
> sort(sapply(mget(ls()), object.size))
> sort(sapply(mget(ls()),
+ function(ob_ject) {
+   format(object.size(ob_ject), units="KB")}
+ ))
> # get sizes of objects in env_etf environment
> sort(sapply(ls(env_etf),
+   function(ob_ject) {
+     object.size(get(ob_ject, env_etf))}))
> # get sizes of objects in env_etf environment
> sort(sapply(mget(ls(env_etf), env_etf),
+       object.size))
> # get total size of all objects in workspace
> print(object.size(x=mget(ls())), units="MB")
> library(gdata)  # load package gdata
> # get names, class, and size of objects in wor
> ob_jects <- ll(unit="bytes")
> # sort by memory size (descending)
> ob_jects[order(ob_jects[, 2], decreasing=TRUE)
> ll()[order(ll()$KB, decreasing=TRUE), ]
```

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()` lists 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,

```
> 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
> v_cells <- gc()["Vcells", "used"]
> # create vector with 1,000,000 cells
> foo_bar <- numeric(1000000)
> # get extra R memory
> gc()["Vcells", "used"] - v_cells
> # 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,

```
> library(microbenchmark)
> foo <- runif(1e6)
> system.time(foo^0.5)
> microbenchmark(sqrt(foo), foo^0.5, 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* Functions

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

This makes *compiled* 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`,

```
> library(microbenchmark)
> # sum() is a compiled primitive function
> sum
> # mean() is a generic function
> mean
> foo <- runif(1e6)
> # sum() is much faster than mean()
> summary(
+   microbenchmark(sum(foo), mean(foo), times=10
+   )[, c(1, 4, 5)]
> # any() is a compiled primitive function
> any
> # any() is much faster than %in% wrapper for m
> summary(
+   microbenchmark(any(foo == 1), {1 %in% foo},
+   )[, 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)
> mat_rix <- matrix(1:9, ncol=3, # create matrix
+   dimnames=list(paste0("row", 1:3),
+           paste0("col", 1:3)))
> # create specialized function
> matrix_to_dframe <- function(mat_rix) {
+   n_col <- ncol(mat_rix)
+   dframe <- vector("list", n_col)  # empty vec
+   for (in_dex in 1:n_col)  # populate vector
+     dframe <- mat_rix[, in_dex]
+   attr(dframe, "row.names") <-  # add attribut
+     .set_row_names(NROW(mat_rix))
+   attr(dframe, "class") <- "data.frame"
+   dframe  # return data frame
+ }  # end matrix_to_dframe
> # compare speed of three methods
> summary(microbenchmark(
+   matrix_to_dframe(mat_rix),
+   as.data.frame.matrix(mat_rix),
+   as.data.frame(mat_rix),
+   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 is therefore preferred obver for() loops,

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

```
> # matrix with 5,000 rows
> big_matrix <- matrix(rnorm(10000), ncol=2)
> # allocate memory for row sums
> row_sums <- numeric(NROW(big_matrix))
> summary(microbenchmark(
+   row_sums=rowSums(big_matrix),  # end row_sums
+   ap_ply=apply(big_matrix, 1, sum),  # end apply
+   l_apply=lapply(1:NROW(big_matrix), function(in_dex)
+     sum(big_matrix[in_dex, ])),  # end lapply
+   v_apply=vapply(1:NROW(big_matrix), function(in_dex)
+     sum(big_matrix[in_dex, ]),
+     FUN.VALUE=c(sum=0)),  # end vapply
+   s_apply=sapply(1:NROW(big_matrix), function(in_dex)
+     sum(big_matrix[in_dex, ])),  # end sapply
+   for_loop=for (i in 1:NROW(big_matrix)) {
+     row_sums[i] <- sum(big_matrix[i,])
+   },  # end for
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Increasing Speed of Loops by Pre-allocating Memory

R doesn't require allocating memory for new vectors or lists, allowing for them to "grow" each time a new element is added,

R allows assigning a value to a vector element that doesn't exist yet (hasn't been allocated),

But when R creates a bigger object from an existing one, it first allocates memory for the new object, and then copies the existing values to the new memory, which is very memory intensive and slow,

Using the functions `c()`, `append()`, `cbind()`, `rbind()`, and `paste()` to append data to objects is even slower than vector assignment,

Adding elements to a vector in a loop is very slow, and therefore not recommended,

Pre-allocating memory for large vectors before performing loops increases their speed,

The function `numeric(k)` returns a numeric vector of zeros of length `k`,

`numeric(0)` returns an empty (zero length) numeric vector (not to be confused with a `NULL`

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

How to Write Fast R Code

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, and call them from R,
- pre-allocate memory for new vectors,
- use `vapply()` and `lapply()` instead of `apply()` and `for()` loops,
- avoid writing too many R function calls (remember that every command in R is a function),

```
> library(microbenchmark)
> foo <- runif(1e6)
> system.time(foo^0.5)
> summary(
+   microbenchmark(sqrt(foo), foo^0.5, times=10)
+   )[, c(1, 4, 5)]
```

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

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,

```
> vec_tor1 <- rnorm(1000000)
> vec_tor2 <- rnorm(1000000)
> big_vector <- numeric(1000000)
> # sum two vectors in two different ways
> summary(microbenchmark(
+   # sum vectors using "for" loop
+   r_loop=(for (i in 1:NROW(vec_tor1)) {
+     big_vector[i] <- vec_tor1[i] + vec_tor2[i]
+   }),
+   # sum vectors using vectorized "+"
+   vec_torized=(vec_tor1 + vec_tor2),
+   times=10))[, c(1, 4, 5)]  # end microbenchma
> # allocate memory for cumulative sum
> cum_sum <- numeric(NROW(big_vector))
> cum_sum[1] <- big_vector[1]
> # calculate cumulative sum in two different wa
> summary(microbenchmark(
+ # cumulative sum using "for" loop
+   r_loop=(for (i in 2:NROW(big_vector)) {
+     cum_sum[i] <- cum_sum[i-1] + big_vector[i]
+   }),
+ # cumulative sum using "cumsum"
+   vec_torized=cumsum(big_vector),
+   times=10))[, c(1, 4, 5)]  # end microbenchma
```

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 row sums two different ways
> summary(microbenchmark(
+   row_sums=rowSums(big_matrix),
+   ap_ply=apply(big_matrix, 1, sum),
+   times=10))[, c(1, 4, 5)]  # end microbenchma
```

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(
+   p_max=
+     do.call(pmax.int,
+ lapply(seq_along(big_matrix[1, ]),
+   function(in_dex) big_matrix[, in_dex])),
+   l_apply=unlist(
+     lapply(seq_along(big_matrix[, 1]),
+   function(in_dex) max(big_matrix[in_dex, ]))),
+   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),

```
> library(matrixStats)  # load package matrixSta
> # calculate row min values three different way
> summary(microbenchmark(
+   row_mins=rowMins(big_matrix),
+   p_min=
+     do.call(pmin.int,
+       lapply(seq_along(big_matrix[1, ]),
+              function(in_dex)
+                big_matrix[, in_dex])),
+   as_data_frame=
+     do.call(pmin.int,
+       as.data.frame.matrix(big_matrix)),
+   times=10))[, c(1, 4, 5)]  # end microbenchma
```

Writing Fast R Code Using Vectorized Operations

Writing Fast R Code Using Vectorized Operations

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

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

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

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

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

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

```
> summary(microbenchmark(  # assign values to ve
+ # fast vectorized assignment loop performed in
+   brack_ets={vec_tor <- numeric(10)
+     vec_tor[] <- 2},
+ # slow because loop is performed in R
+   for_loop={vec_tor <- numeric(10)
+     for (in_dex in seq_along(vec_tor))
+       vec_tor[in_dex] <- 2},
+ # very slow because no memory is pre-allocated
+ # "vec_tor" is "grown" with each new element
+   grow_vec={vec_tor <- numeric(0)
+     for (in_dex in 1:10)
+ # add new element to "vec_tor" ("grow" it)
+       vec_tor[in_dex] <- 2},
+   times=10))[, c(1, 4, 5)]  # end microbenchma
> summary(microbenchmark(  # assign values to ve
+ # fast vectorized assignment loop performed in
+   brack_ets={vec_tor <- numeric(10)
+     vec_tor[4:7] <- rnorm(4)},
+ # slow because loop is performed in R
+   for_loop={vec_tor <- numeric(10)
+     for (in_dex in 4:7)
+       vec_tor[in_dex] <- rnorm(1)},
+   times=10))[, c(1, 4, 5)]  # end microbenchma
```

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(in_put, pa_ram) {
+   pa_ram*in_put
+ }  # end my_fun
> # "in_put" is vectorized
> my_fun(in_put=1:3, pa_ram=2)
> # "pa_ram" is vectorized
> my_fun(in_put=10, pa_ram=2:4)
> # define vectors of parameters of rnorm()
> std_devs <-
+   structure(1:3, names=paste0("sd=", 1:3))
> me_ans <-
+   structure(-1:1, names=paste0("mean=", -1:1))
> # "sd" argument of rnorm() isn't vectorized
> rnorm(1, sd=std_devs)
> # "mean" argument of rnorm() isn't vectorized
> rnorm(1, mean=me_ans)
```

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,

```
> # sapply produces desired vector output
> set.seed(1121)
> sapply(std_devs, function(std_dev) rnorm(n=2,
> set.seed(1121)
> sapply(std_devs, rnorm, n=2, mean=0)
> set.seed(1121)
> sapply(me_ans,
+  function(me_an) rnorm(n=2, mean=me_an))
> set.seed(1121)
> sapply(me_ans, 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 "std_dev"
> 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=std_devs)
> # rnorm() vectorized with respect to "mean" an
> vec_rnorm <- Vectorize(FUN=rnorm,
+         vectorize.args=c("mean", "sd")
+ )  # end Vectorize
> set.seed(1121)
> vec_rnorm(n=2, sd=std_devs)
> set.seed(1121)
> vec_rnorm(n=2, mean=me_ans)
```

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:

$$mapply(FUN = fun, vec_1, vec_2, \ldots) = [fun(vec_{1,1}, vec_{2,1}, \ldots), \ldots, fun(vec_{1,i}, vec_{2,i}, \ldots), \ldots]$$

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
> mapply(rnorm, n=5, mean=me_ans, sd=std_devs)
> mapply(function(in_put, e_xp) in_put^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,

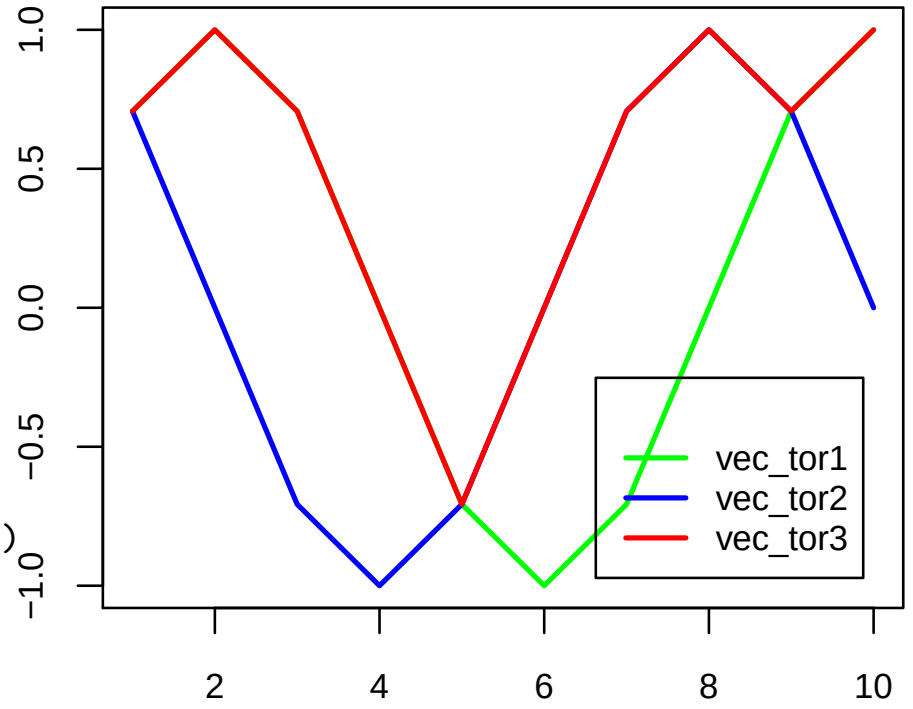
```
> # rnorm() vectorized with respect to "mean" an
> vec_rnorm <- function(n, mean=0, sd=1) {
+   if (NROW(mean)==1 && NROW(sd)==1)
+     rnorm(n=n, mean=mean, sd=sd)
+   else
+     mapply(rnorm, n=n, mean=mean, sd=sd)
+ }  # end vec_rnorm
> # call vec_rnorm() on vector of "sd"
> vec_rnorm(n=2, sd=std_devs)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

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
> vec_tor1 <- sin(0.25*pi*1:10)
> vec_tor2 <- cos(0.25*pi*1:10)
> # create third vector using 'ifelse'
> vec_tor3 <- ifelse(vec_tor1 > vec_tor2,
+           vec_tor1, vec_tor2)
> # cbind all three together
> vec_tor4 <- cbind(vec_tor1, vec_tor2, vec_tor3)
>
> # set plotting parameters
> par(mar=c(7, 2, 1, 2), mgp=c(2, 1, 0),
+     cex.lab=0.8, cex.axis=0.8, cex.main=0.8,
+     cex.sub=0.5)
> # plot matrix
> matplot(vec_tor4, type="l", lty="solid",
+ col=c("green", "blue", "red"),
+ lwd=c(2, 2, 2), xlab="", ylab="")
> # add legend
> legend(x="bottomright", legend=colnames(vec_tor4),
+        title="", inset=0.05, cex=0.8, lwd=2,
+        lty=c(1, 1, 1), col=c("green", "blue", "red"))
```



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,

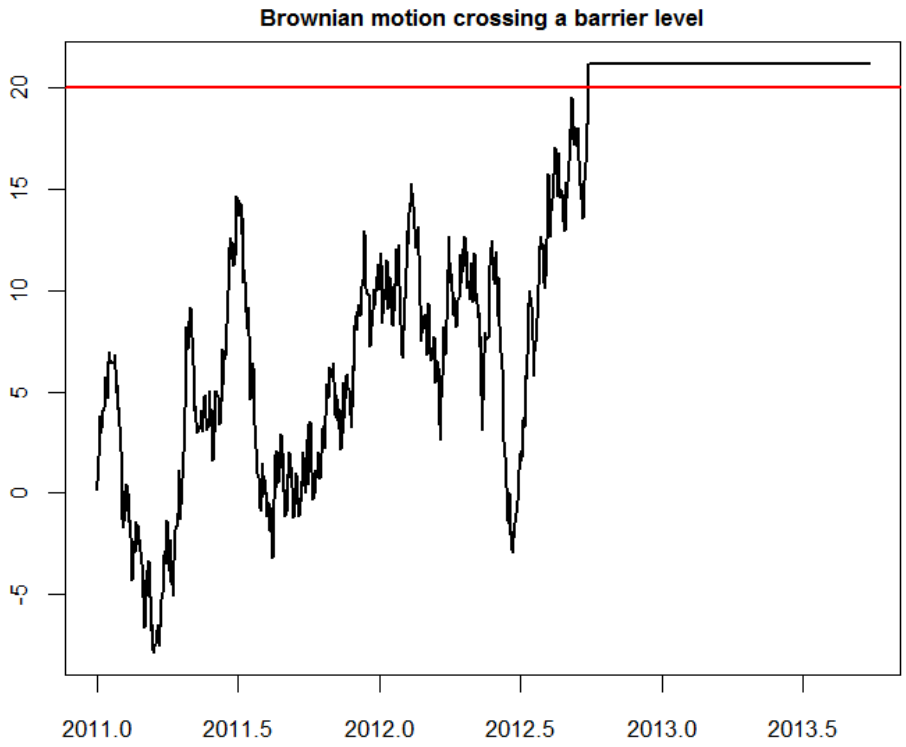
```
> set.seed(1121)  # reset random number generato
> # sample from Standard Normal Distribution
> len_gth <- 1000
> sam_ple <- rnorm(len_gth)
> # sample mean - MC estimate
> mean(sam_ple)
> # sample standard deviation - MC estimate
> sd(sam_ple)
> # MC estimate of cumulative probability
> sam_ple <- sort(sam_ple)
> pnorm(1)
> sum(sam_ple<1)/len_gth
> # MC estimate of quantile
> qnorm(0.75)
> sam_ple[0.75*len_gth]
```

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,

```
> lev_el <- 20  # barrier level
> len_gth <- 1000  # number of simulation steps
> pa_th <- numeric(len_gth)  # allocate path vec
> pa_th[1] <- 0  # initialize path
> in_dex <- 2  # initialize simulation index
> while ((in_dex <= len_gth) &&
+  (pa_th[in_dex - 1] < lev_el)) {
+ # simulate next step
+   pa_th[in_dex] <-
+     pa_th[in_dex - 1] + rnorm(1)
+   in_dex <- in_dex + 1  # advance in_dex
+ }  # end while
> # fill remaining pa_th after it crosses lev_el
> if (in_dex <= len_gth)
+   pa_th[in_dex:len_gth] <- pa_th[in_dex - 1]
> # create daily time series starting 2011
> ts_path <- ts(data=pa_th, frequency=365, start=c(2011, 1))
> plot(ts_path, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=lev_el, lwd=2, col="red")
> title(main="Brownian motion crossing a barrier level",
+       line=0.5)
```

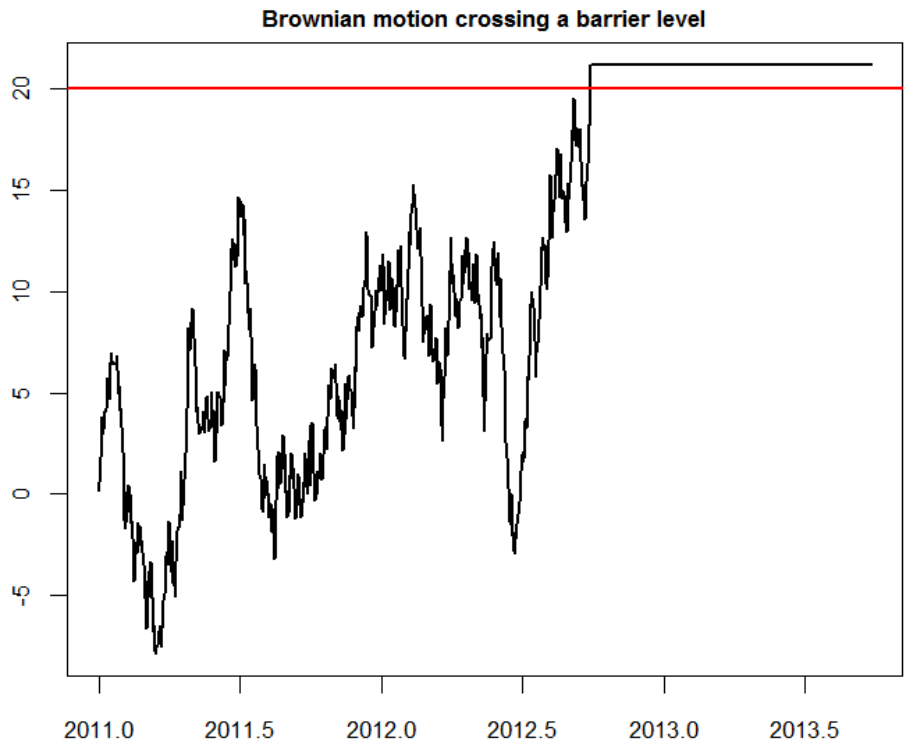


Simulating Brownian Motion Using Vectorized Functions

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,

```
> lev_el <- 20  # barrier level
> len_gth <- 1000  # number of simulation steps
> # simulate path of Brownian motion
> pa_th <- cumsum(rnorm(len_gth))
> # find index when pa_th crosses lev_el
> cro_ss <- which(pa_th > lev_el)
> # fill remaining pa_th after it crosses lev_el
> if (NROW(cro_ss)>0) {
+   pa_th[(cro_ss[1]+1):len_gth] <-
+     pa_th[cro_ss[1]]
+ }  # end if
> # create daily time series starting 2011
> ts_path <- ts(data=pa_th, frequency=365,
+      start=c(2011, 1))
> # create plot with horizontal line
> plot(ts_path, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=lev_el, lwd=2, col="red")
> title(main="Brownian motion crossing a barrier level",
+       line=0.5)
```



The trade-off 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,

Standard Errors 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 data set,

The *bootstrapped* data is then used to re-calculate the estimator many times, producing a vector of values,

The *bootstrapped* estimator values can then be used to calculate the probability distribution of the estimator and its standard error,

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

```
> set.seed(1121)  # reset random number generato
> # sample from Standard Normal Distribution
> len_gth <- 1000
> sam_ple <- rnorm(len_gth)
> # sample mean
> mean(sam_ple)
> # sample standard deviation
> sd(sam_ple)
> # bootstrap of sample mean and median
> boot_strap <- sapply(1:10000, function(x) {
+   boot_sample <- sam_ple[sample.int(len_gth,
+                                     replace=TRUE)]
+   c(mean=mean(boot_sample),
+     median=median(boot_sample))
+ })  # end sapply
> boot_strap[, 1:3]
> # standard error from formula
> sd(sam_ple)/sqrt(len_gth)
> # standard error of mean from bootstrap
> sd(boot_strap["mean", ])
> # standard error of median from bootstrap
> sd(boot_strap["median", ])
```

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 apply 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 apply 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
> num_cores <- detectCores() - 1  # number of co
> clus_ter <- makeCluster(num_cores)  # initiali
> set.seed(1121)  # reset random number generato
> # sample from Standard Normal Distribution
> len_gth <- 1000
> sam_ple <- rnorm(len_gth)
> # bootstrap mean and median under Windows
> boot_strap <- parLapply(clus_ter, 1:10000,
+   function(x, sam_ple, len_gth) {
+   boot_sample <- sam_ple[sample.int(len_gth, r
+   c(mean=mean(boot_sample), median=median(boot
+   }, sam_ple=sam_ple, len_gth=len_gth)  # end
> # bootstrap mean and median under Mac-OSX or L
> boot_strap <- mclapply(1:10000,
+   function(x) {
+   boot_sample <- sam_ple[sample.int(len_gth, r
+   c(mean=mean(boot_sample), median=median(boot
+   }, mc.cores=num_cores)  # end mclapply
> boot_strap <- rutils::do_call(rbind, boot_stra
> # means and standard errors from bootstrap
> apply(boot_strap, MARGIN=2,
+ function(x) c(mean=mean(x), sd=sd(x)))
> # standard error from formula
> sd(sam_ple)/sqrt(len_gth)
> stopCluster(clus_ter)  # stop R processes over
```

Variance Reduction Using Antithetic Sampling

Antithetic Sampling is a *Variance Reduction* technique, in which a new random sample is computed by simply reversing the sign of a Normal random sample ($\phi \to -\phi$), or by complementing a Uniform random sample ($\phi \to 1 - \phi$),

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

```
> set.seed(1121)  # reset random number generato
> # sample from Standard Normal Distribution
> len_gth <- 1000
> sam_ple <- rnorm(len_gth)
> # estimate the 95% quantile
> boot_strap <- sapply(1:10000, function(x) {
+   boot_sample <- sam_ple[sample.int(len_gth,
+                                     replace=TRUE)]
+   quantile(boot_sample, 0.95)
+ })  # end sapply
> sd(boot_strap)
> # estimate the 95% quantile using antithetic s
> boot_strap <- sapply(1:10000, function(x) {
+   boot_sample <- sam_ple[sample.int(len_gth,
+                                     replace=TRUE)]
+   quantile(c(boot_sample, -boot_sample), 0.95)
+ })  # end sapply
> # standard error of quantile from bootstrap
> sd(boot_strap)
> sqrt(2)*sd(boot_strap)
```

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 design matrix,

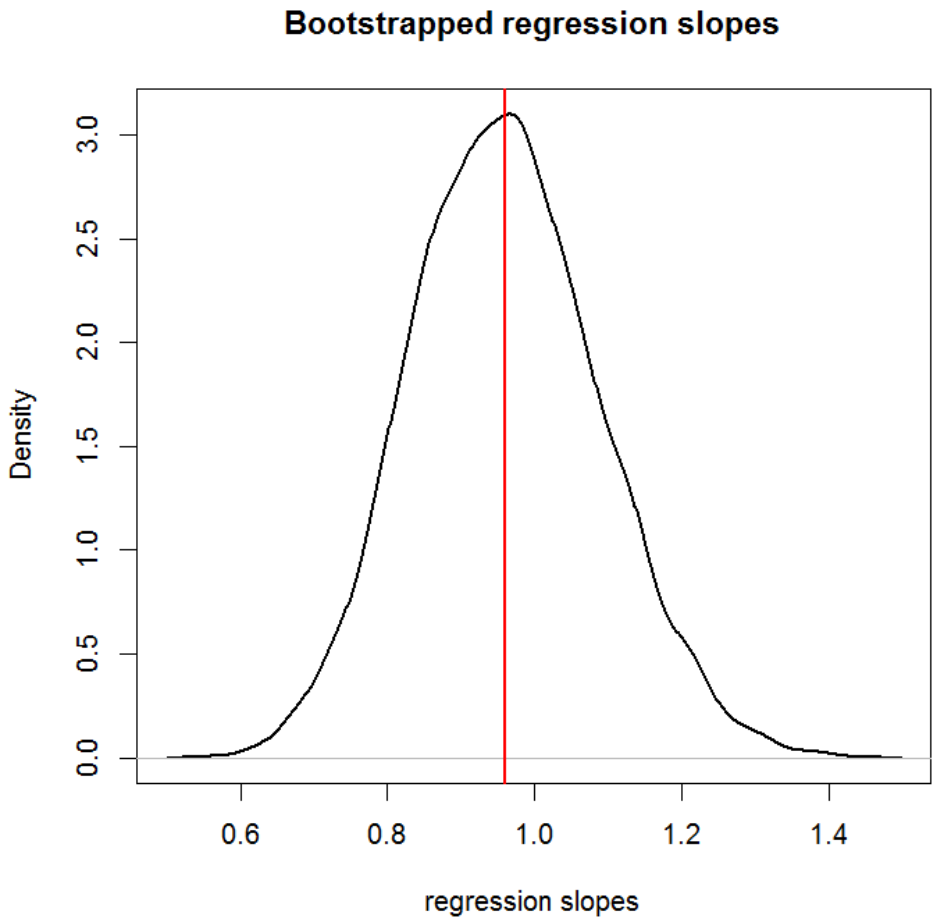
Regressions are performed on the *bootstrapped* design matrices, and the regression coefficients are saved into a matrix of *bootstrapped* coefficients,

```
> set.seed(1121)  # initialize random number gen
> # define explanatory and response variables
> ex_plain <- rnorm(100, mean=2)
> noise <- rnorm(100)
> res_ponse <- -3 + ex_plain + noise
> # define design matrix and regression formula
> de_sign <- data.frame(res_ponse, ex_plain)
> reg_formula <- paste(colnames(de_sign)[1],
+   paste(colnames(de_sign)[-1], collapse="+"),
+   sep=" ~ ")
> # bootstrap the regression
> boot_strap <- sapply(1:100, function(x) {
+   boot_sample <- sample.int(dim(de_sign)[1],
+                             replace=TRUE)
+   reg_model <- lm(reg_formula,
+           data=de_sign[boot_sample, ])
+   reg_model$coefficients
+ })  # 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,

```
> # means and standard errors from bootstrap
> apply(boot_strap, MARGIN=1,
+       function(x) c(mean=mean(x), sd=sd(x)))
> # means and standard errors from regression s
> reg_model <- lm(reg_formula, data=de_sign)
> reg_model$coefficients
> summary(reg_model)$coefficients[, "Std. Erro
> plot(density(boot_strap["ex_plain", ]),
+      lwd=2, xlab="regression slopes",
+      main="Bootstrapped regression slopes")
> abline(v=mean(boot_strap["ex_plain", ]),
+        lwd=2, col="red")
```



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 apply 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 apply 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
> num_cores <- detectCores() - 1  # number of co
> clus_ter <- makeCluster(num_cores)  # initiali
> # bootstrap the regression under Windows
> boot_strap <- parLapply(clus_ter, 1:1000,
+   function(x, reg_formula, de_sign) {
+     boot_sample <-
+ sample.int(dim(de_sign)[1], replace=TRUE)
+     reg_model <- lm(reg_formula,
+ data=de_sign[boot_sample, ])
+     reg_model$coefficients
+   },
+   reg_formula=reg_formula,
+   de_sign=de_sign)  # end parLapply
> # bootstrap the regression under Mac-OSX or Li
> boot_strap <- mclapply(1:1000,
+   function(x) {
+     boot_sample <-
+ sample.int(dim(de_sign)[1], replace=TRUE)
+     lm(reg_formula,
+ data=de_sign[boot_sample, ])$coefficients
+   }, mc.cores=num_cores)  # end mclapply
> stopCluster(clus_ter)  # stop R processes over
> boot_strap <- rutils::do_call(rbind, boot_stra
> # means and standard errors from bootstrap
> apply(boot_strap, MARGIN=2,
+ function(x) c(mean=mean(x), sd=sd(x)))
> x11(width=6, height=6)
```

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*),

```
> 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 apply 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 apply loops under *Windows*, using parallel computing on several CPU cores,

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

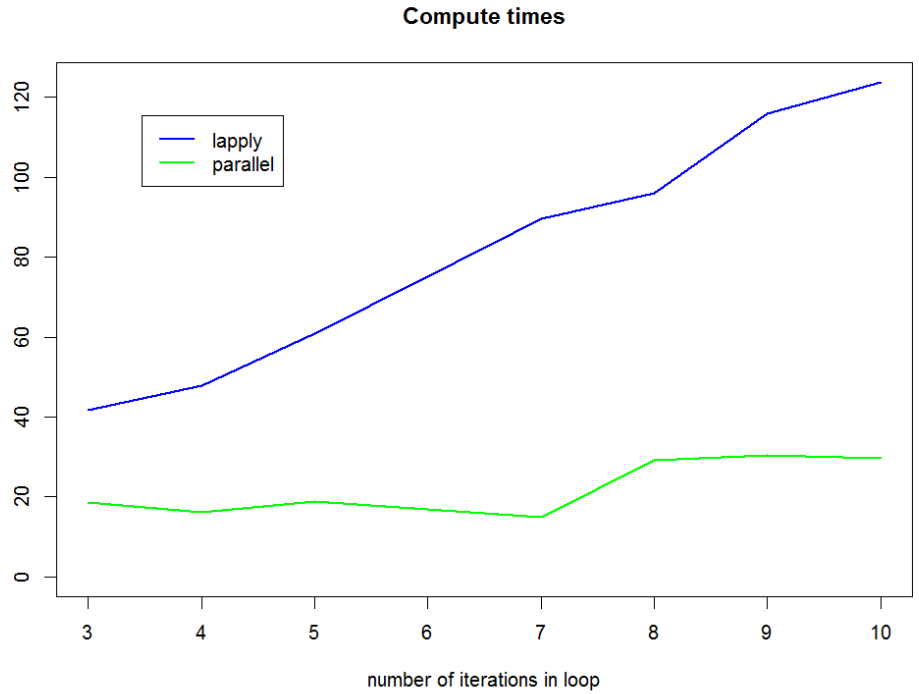
```
> library(parallel)  # load package parallel
> # calculate number of available cores
> num_cores <- detectCores() - 1
> # define function that pauses execution
> paws <- function(x, sleep_time) {
+   Sys.sleep(sleep_time)
+   x
+ }  # end paws
> # perform parallel loop under Mac-OSX or Linux
> paw_s <- mclapply(1:10, paws, mc.cores=num_cor
+           sleep_time=0.01)
> # initialize compute cluster under Windows
> clus_ter <- makeCluster(num_cores)
> # perform parallel loop under Windows
> paw_s <- parLapply(clus_ter, 1:10, paws,
+            sleep_time=0.01)
> library(microbenchmark)  # load package microb
> # compare speed of lapply versus parallel comp
> summary(microbenchmark(
+   l_apply=lapply(1:10, paws, sleep_time=0.01),
+   parl_apply=
+     parLapply(clus_ter, 1:10, paws, sleep_time
+   times=10)
+ )[, c(1, 4, 5)]
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

Computing Overhead of Parallel Computing

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,

```
> # compare speed of lapply with parallel comput
> iter_ations <- 3:10
> compute_times <- sapply(iter_ations,
+   function(max_iterations, sleep_time) {
+     out_put <- summary(microbenchmark(
+ lapply=lapply(1:max_iterations, paws,
+               sleep_time=sleep_time),
+ parallel=parLapply(clus_ter, 1:max_iterations
+         paws, sleep_time=sleep_time),
+ times=10))[, c(1, 4)]
+     structure(out_put[, 2],
+         names=as.vector(out_put[, 1]))
+     }, sleep_time=0.01)
> compute_times <- t(compute_times)
> rownames(compute_times) <- iter_ations
```




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

```
> # define large matrix
> mat_rix <- matrix(rnorm(7*10^5), ncol=7)
> # define aggregation function over column of m
> agg_regate <- function(col_umn) {
+   out_put <- 0
+   for (in_dex in 1:NROW(col_umn))
+     out_put <- out_put + col_umn[in_dex]
+   out_put
+ }  # end agg_regate
> # perform parallel aggregations over columns o
> agg_regations <-
+   parCapply(clus_ter, mat_rix, agg_regate)
> # compare speed of apply with parallel computi
> summary(microbenchmark(
+   ap_ply=apply(mat_rix, MARGIN=2, agg_regate),
+   parl_apply=
+     parCapply(clus_ter, mat_rix, agg_regate),
+   times=10)
+ )[, c(1, 4, 5)]
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

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,

```
> ba_se <- 2
> # fails because child processes don't know ba_
> parLapply(clus_ter, 2:4,
+     function(exponent) ba_se^exponent)
> # ba_se passed to child via dots ... argument:
> parLapply(clus_ter, 2:4,
+     function(exponent, ba_se) ba_se^exponent,
+     ba_se=ba_se)
> # ba_se passed to child via clusterExport:
> clusterExport(clus_ter, "ba_se")
> parLapply(clus_ter, 2:4,
+     function(exponent) ba_se^exponent)
> # fails because child processes don't know zoo
> parSapply(clus_ter, c("VTI", "IEF", "DBC"),
+     function(sym_bol)
+       NROW(index(get(sym_bol, envir=rutils::en
> # zoo function referenced using "::" in child
> parSapply(clus_ter, c("VTI", "IEF", "DBC"),
+     function(sym_bol)
+       NROW(zoo::index(get(sym_bol, envir=rutil
> # package zoo loaded in child process:
> parSapply(clus_ter, c("VTI", "IEF", "DBC"),
+     function(sym_bol) {
+       stopifnot("package:zoo" %in% search() ||
+       NROW(index(get(sym_bol, envir=rutils::en
+     })  # end parSapply
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

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
> num_cores <- detectCores() - 1
> # initialize compute cluster under Windows
> clus_ter <- makeCluster(num_cores)
> # set seed for cluster under Windows
> # doesn't work: set.seed(1121)
> clusterSetRNGStream(clus_ter, 1121)
> # perform parallel loop under Windows
> out_put <- parLapply(clus_ter, 1:70, rnorm, n=
> sum(unlist(out_put))
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
> # perform parallel loop under Mac-OSX or Linux
> out_put <- mclapply(1:10, rnorm, mc.cores=num_
```

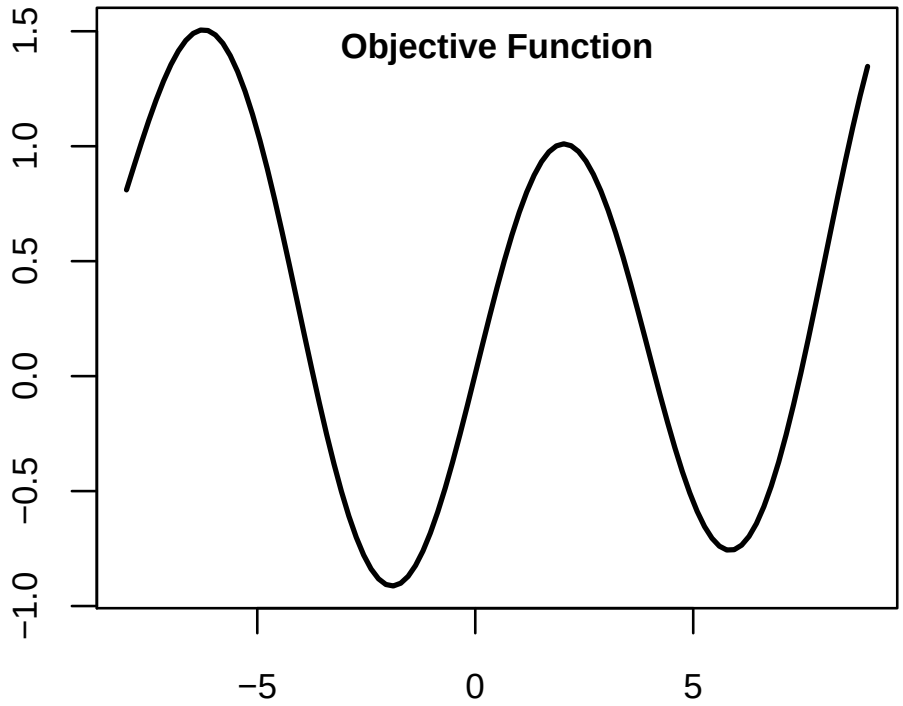
One-dimensional Optimization Using The Functional optimize()

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,

```
> str(optimize)
> # objective function with multiple minima
> object_ive <- function(in_put, param1=0.01) {
+   sin(0.25*pi*in_put) + param1*(in_put-1)^2
+ }  # end object_ive
> unlist(optimize(f=object_ive, interval=c(-4, 2)))
> unlist(optimize(f=object_ive, interval=c(0, 8)))
> options(width=60, dev='pdf')
```



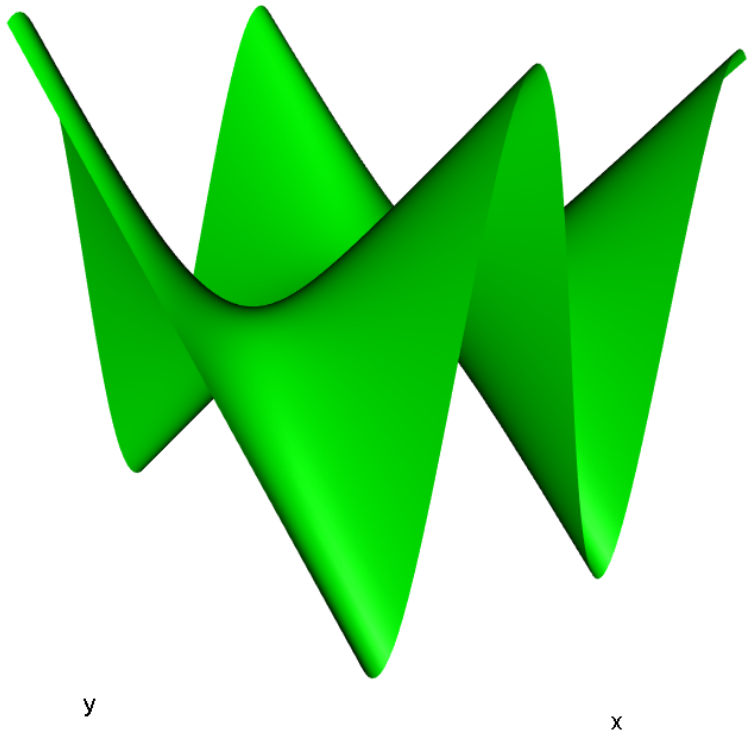

```
> # plot the objective function
> curve(expr=object_ive, type="l", xlim=c(-8, 9)
+ xlab="", ylab="", lwd=2)
> # add title
> title(main="Objective Function", line=-1)
```

Package *rgl* for Interactive 3d Surface Plots

The function `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,

```
> library(rgl)  # load rgl
> # define function of two variables
> sur_face <- function(x, y) y*sin(x)
> # draw 3d surface plot of function
> persp3d(x=sur_face, xlim=c(-5, 5), ylim=c(-5,
+   col="green", axes=FALSE)
> # draw 3d surface plot of matrix
> x_lim <- seq(from=-5, to=5, by=0.1)
> y_lim <- seq(from=-5, to=5, by=0.1)
> persp3d(z=outer(x_lim, y_lim, FUN=sur_face),
+   xlab="x", ylab="y", zlab="sur_face",
+   col="green")
> # save current view to png file
> rgl.snapshot("surface_plot.png")
> # define function of two variables and two par
> sur_face <- function(x, y, par_1=1, par_2=1)
+   sin(par_1*x)*sin(par_2*y)
> # draw 3d surface plot of function
> persp3d(x=sur_face, xlim=c(-5, 5), ylim=c(-5, 5),
+   col="green", axes=FALSE,
+   par_1=1, par_2=2)
```



Multi-dimensional Optimization Using optim()

The functional 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
> rastri_gin <- function(vec_tor, pa_ram=25){
+   sum(vec_tor^2 - pa_ram*cos(vec_tor))
+ }  # end rastri_gin
> vec_tor <- c(pi/6, pi/6)
> rastri_gin(vec_tor=vec_tor)
> # draw 3d surface plot of Rastrigin function
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastri_gin(vec_to
+   xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, zlab="", main="rast
> # optimize with respect to vector argument
> op_tim <- optim(par=vec_tor, fn=rastri_gin,
+         method="L-BFGS-B",
+         upper=c(4*pi, 4*pi),
+         lower=c(pi/2, pi/2),
+         pa_ram=1)
> # optimal parameters and value
> op_tim$par
> op_tim$value
> rastri_gin(op_tim$par, pa_ram=1)
```

The Log-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
> sam_ple <- rnorm(1000, mean=4, sd=2)
> # objective function is log-likelihood
> object_ive <- function(pa_r, sam_ple) {
+   sum(2*log(pa_r[2]) +
+     ((sam_ple - pa_r[1])/pa_r[2])^2)
+ }  # end object_ive
> # vectorize objective function
> vec_objective <- Vectorize(
+   FUN=function(mean, sd, sam_ple)
+     object_ive(c(mean, sd), sam_ple),
+   vectorize.args=c("mean", "sd")
+ )  # end Vectorize
> # objective function on parameter grid
> par_mean <- seq(1, 6, length=50)
> par_sd <- seq(0.5, 3.0, length=50)
> objective_grid <- outer(par_mean, par_sd,
+ vec_objective, sam_ple=sam_ple)
> objective_min <- which(  # grid search
+   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)]
```

Perspective Plot of Likelihood Function

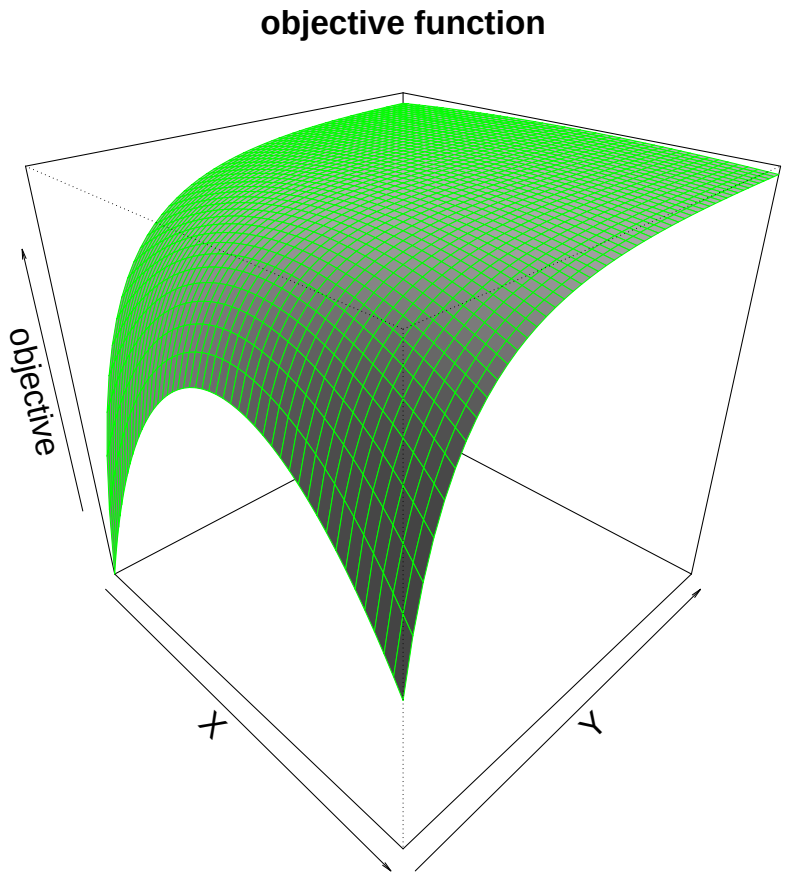
The function `persp()` plots a 3d perspective surface plot of a function specified over a grid of argument values,

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 `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
> par3d(cex=2.0)  # scale text by factor of 2
> persp3d(z=-objective_grid, zlab="objective",
+   col="green", main="objective function")
```

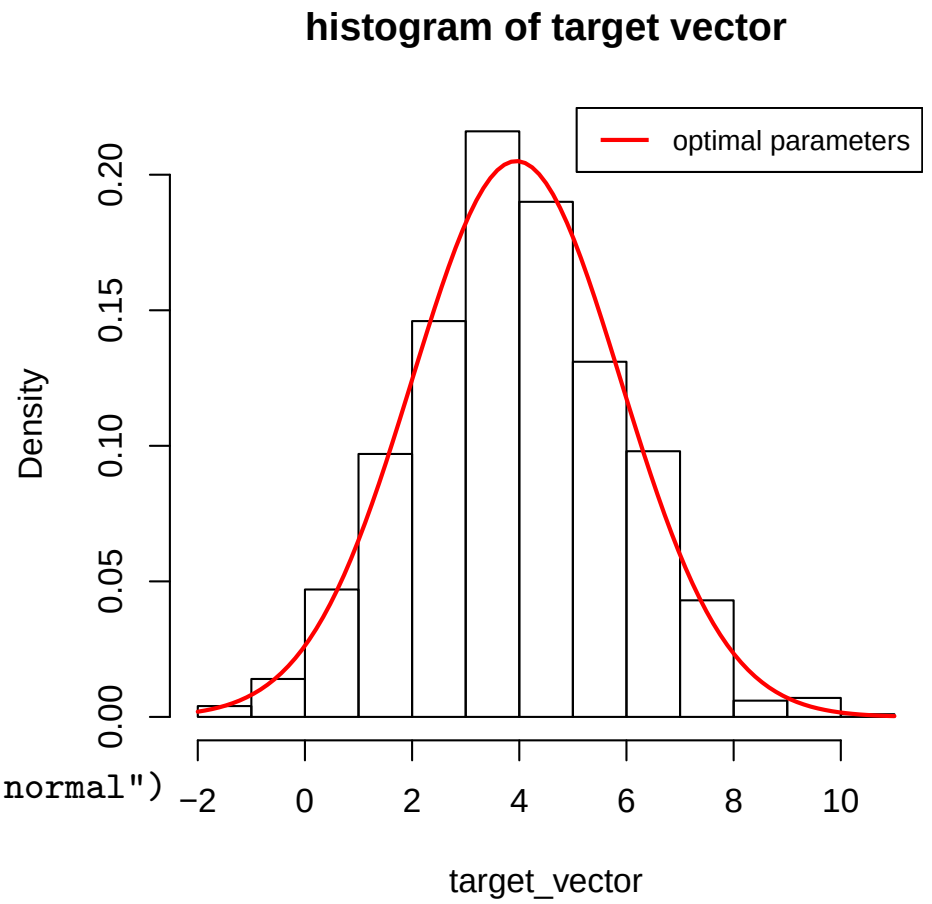
Optimization of Objective Function

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
> par_init <- c(mean=0, sd=1)
> # perform optimization using optim()
> optim_fit <- optim(par=par_init,
+   fn=object_ive, # log-likelihood function
+   sam_ple=sam_ple,
+   method="L-BFGS-B", # quasi-Newton method
+   upper=c(10, 10), # upper constraint
+   lower=c(-10, 0.1)) # lower constraint
> # optimal parameters
> optim_fit$par
> # perform optimization using MASS::fitdistr()
> optim_fit <- MASS::fitdistr(sam_ple, densfun="normal")
> optim_fit$estimate
> optim_fit$sd
```

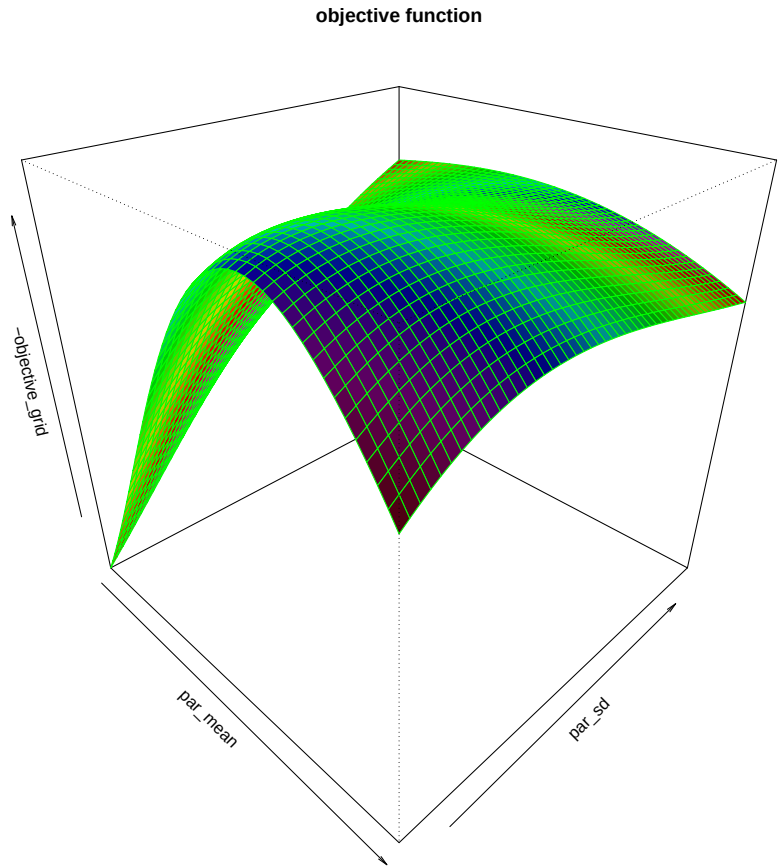
```
> # plot histogram
> histo_gram <- hist(sam_ple, plot=FALSE)
> plot(histo_gram, freq=FALSE,
+      main="histogram of sample")
> curve(expr=dnorm(x, mean=optim_fit$par["mean"],
+           sd=optim_fit$par["sd"]),
+ add=TRUE, type="l", lwd=2, col="red")
```



Mixture Model Likelihood Function

```
> # sample from mixture of normal distributions
> sam_ple <- c(rnorm(100, sd=1.0),
+              rnorm(100, mean=4, sd=1.0))
> # objective function is log-likelihood
> object_ive <- function(pa_r, sam_ple) {
+   likelihood <- pa_r[1]/pa_r[3] *
+   dnorm((sam_ple-pa_r[2])/pa_r[3]) +
+   (1-pa_r[1])/pa_r[5]*dnorm((sam_ple-pa_r[4])/pa_r[5])
+   if (any(likelihood <= 0)) Inf else
+     -sum(log(likelihood))
+ }  # end object_ive
> # vectorize objective function
> vec_objective <- Vectorize(
+   FUN=function(mean, sd, w, m1, s1, sam_ple)
+     object_ive(c(w, m1, s1, mean, sd), sam_ple),
+   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,
+     vec_objective, sam_ple=sam_ple,
+     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
```

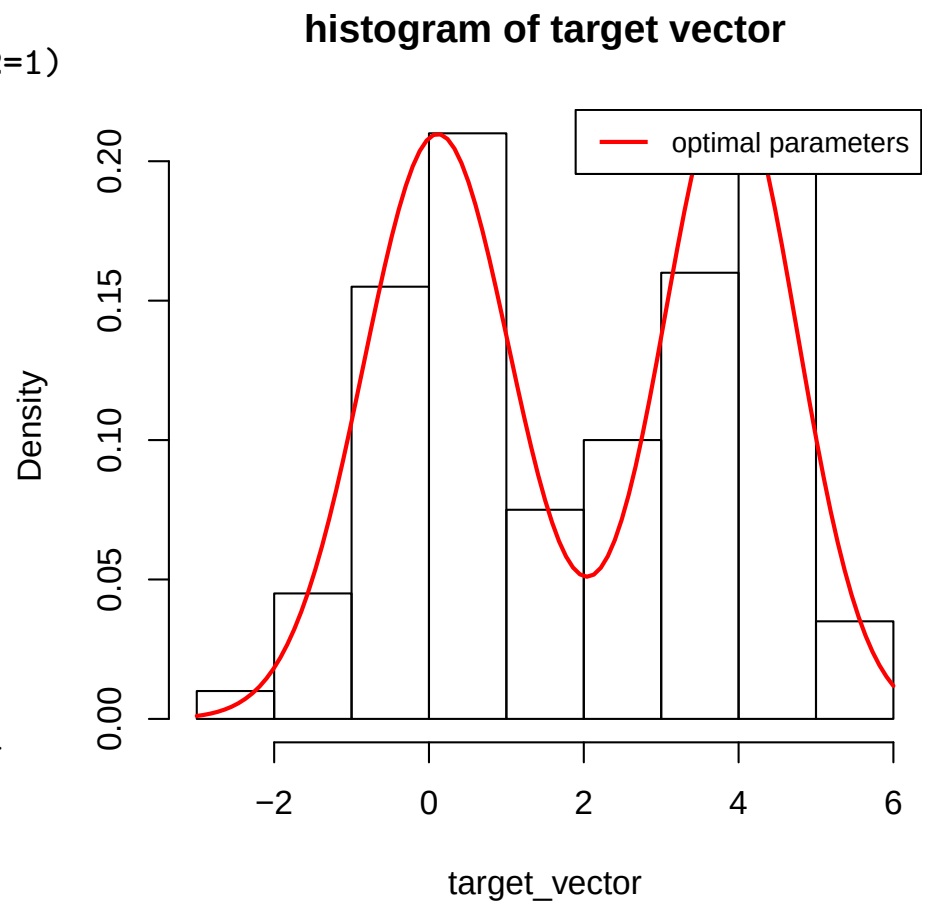
```
> # perspective plot of objective function
> persp(par_mean, par_sd, -objective_grid,
+ theta=45, phi=30,
+ shade=0.5,
+ col=rainbow(50),
+ border="green",
+ main="objective function")
```



Optimization of Mixture Model

```
> # initial parameters
> par_init <- c(weight=0.5, m1=0, s1=1, m2=2, s2=1)
> # perform optimization
> optim_fit <- optim(par=par_init,
+       fn=object_ive,
+       sam_ple=sam_ple,
+       method="L-BFGS-B",
+       upper=c(1,10,10,10,10),
+       lower=c(0,-10,0.2,-10,0.2))
> optim_fit$par
```

```
> # plot histogram
> histo_gram <- hist(sam_ple, plot=FALSE)
> plot(histo_gram, freq=FALSE,
+      main="histogram of sample")
> fit_func <- function(x, pa_r) {
+   pa_r["weight"] *
+     dnorm(x, mean=pa_r["m1"], sd=pa_r["s1"]) +
+   (1-pa_r["weight"]) *
+     dnorm(x, mean=pa_r["m2"], sd=pa_r["s2"])
+ }  # end fit_func
> curve(expr=fit_func(x, pa_r=optim_fit$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* performs *global* optimization using the *Differential Evolution* algorithm,

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

http://www1.icsi.berkeley.edu/~storn/code.html

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,

```
> # Rastrigin function with vector argument for
> rastri_gin <- function(vec_tor, pa_ram=25){
+   sum(vec_tor^2 - pa_ram*cos(vec_tor))
+ }  # end rastri_gin
> vec_tor <- c(pi/6, pi/6)
> rastri_gin(vec_tor=vec_tor)
> library(DEoptim)
> ## optimize rastri_gin using DEoptim
> op_tim <-  DEoptim(rastri_gin,
+   upper=c(6, 6), lower=c(-6, -6),
+   DEoptim.control(trace=FALSE, itermax=50))
> # optimal parameters and value
> op_tim$optim$bestmem
> rastri_gin(op_tim$optim$bestmem)
> summary(op_tim)
> plot(op_tim)
```

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,

http://www1.icsi.berkeley.edu/~storn/code.html

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
> rastri_gin <- function(vec_tor, pa_ram=25){
+   sum(vec_tor^2 - pa_ram*cos(vec_tor))
+ }  # end rastri_gin
> vec_tor <- c(pi/6, pi/6)
> rastri_gin(vec_tor=vec_tor)
> library(DEoptim)
> ## optimize rastri_gin using DEoptim
> op_tim <-  DEoptim(rastri_gin,
+   upper=c(6, 6), lower=c(-6, -6),
+   DEoptim.control(trace=FALSE, itermax=50))
> # optimal parameters and value
> op_tim$optim$bestmem
> rastri_gin(op_tim$optim$bestmem)
> summary(op_tim)
> plot(op_tim)
```

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 are working properly:
> devtools::find_rtools()
> devtools::has_devel()
>
> # 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="C:/Develop/R/lecture_sli
> # multiply two numbers
> rcpp_mult(2, 3)
> rcpp_mult(1:3, 6:4)
> # multiply two vectors
> rcpp_mult_vec(2, 3)
> rcpp_mult_vec(1:3, 6:4)
```

Loops in *Rcpp Sugar*

Loops 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, NumericVec
+ int x_size = x.size();
+ int y_size = y.size();
+ if (x_size != y_size) {
+     return 0;
+   } else {
+     double total = 0;
+     for(int i = 0; i < x_size; ++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_mult_sugar(NumericVector x, NumericVector y) {
+   return sum(x * y);
```

```
> # define R function with loop
> inner_mult_r <- function(x, y) {
+     to_tal <- 0
+     for(i in 1:NROW(x)) {
+ to_tal <- to_tal + x[i] * y[i]
+     }
+     to_tal
+ }  # end inner_mult_r
> # run R function
> inner_mult_r(1:3, 6:4)
> inner_mult_r(1:3, 6:3)
> # compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=inner_mult_r(1:10000, 1:10000),
+   inner_r=1:10000 %*% 1:10000,
+   r_cpp=inner_mult(1:10000, 1:10000),
+   r_cpp_sugar=inner_mult_sugar(1:10000, 1:1000
+   times=10))[, c(1, 4, 5)]
```

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
> ou_proc <- function(len_gth=1000, eq_price=5.(
+               vol_at=0.01, the_ta=0.01) {
+   re_turns <- numeric(len_gth)
+   price_s <- numeric(len_gth)
+   price_s[1] <- eq_price
+   for (i in 2:len_gth) {
+     re_turns[i] <- the_ta*(eq_price - price_s
+     price_s[i] <- price_s[i-1] * exp(re_turns
+   }  # end for
+   price_s
+ }  # end ou_proc
> # simulate Ornstein-Uhlenbeck process
> eq_price <- 5.0; vol_at <- 0.01
> the_ta <- 0.01; len_gth <- 1000
> set.seed(1121)  # reset random numbers
> price_s <- ou_proc(len_gth=len_gth, eq_price=(
```

```
> # define Ornstein-Uhlenbeck function in Rcpp
> Rcpp::cppFunction("
+ NumericVector rcpp_ou_proc(int len_gth, double
+   NumericVector price_s(len_gth);
+   NumericVector re_turns(len_gth);
+   price_s[0] = eq_price;
+   for (int i = 1; i < len_gth; ++i) {
+     re_turns[i] = the_ta*(eq_price - price_s[i
+     price_s[i] = price_s[i-1] * exp(re_turns[i
+   }
+   return price_s;
+ }")  # end cppFunction
> set.seed(1121)  # reset random numbers
> price_s <- rcpp_ou_proc(len_gth=len_gth, eq_pr
> # compare speed of Rcpp and R
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=ou_proc(len_gth=len_gth, eq_price=eq_
+   r_cpp=rcpp_ou_proc(len_gth=len_gth, eq_price
+   times=10))[, c(1, 4, 5)]
```

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-rando
> uni_form <- function(see_d, len_gth=10) {
+   out_put <- numeric(len_gth)
+   out_put[1] <- see_d
+   for (i in 2:len_gth) {
+     out_put[i] <- 4*out_put[i-1]*(1-out_put[i-
+   }  # end for
+   acos(1-2*out_put)/pi
+ }  # end uni_form
>
> # source Rcpp functions from file
> Rcpp::sourceCpp(file="C:/Develop/R/lecture_sli
> # microbenchmark Rcpp code
> library(microbenchmark)
> summary(microbenchmark(
+   pure_r=runif(1e5),
+   r_loop=uni_form(0.3, 1e5),
+   r_cpp=uniform_rcpp(0.3, 1e5),
+   times=10))[, c(1, 4, 5)]
```

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

// 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 uni_form() produces a vector of
// uniformly distributed pseudo-random numbers
// [[Rcpp::export]]
NumericVector uniform_rcpp(double see_d, int len_gth) {
// define pi
static const double pi = 3.14159265;
// allocate output vector
  NumericVector out_put(len_gth);
// initialize output vector
  out_put[0] = see_d;
// perform loop
  for (int i=1; i < len_gth; ++i) {
    out_put[i] = 4*out_put[i-1]*(1-out_put[i-1]);
  }  // end for
// rescale output vector and return it
  return acos(1-2*out_put)/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/RcppArmadillo/index.html
https://github.com/RcppCore/RcppArmadillo

```
> library(RcppArmadillo)
> # source Rcpp functions from file
> Rcpp::sourceCpp(file="C:/Develop/R/lecture_sl
> vec1 <- runif(1e5)
> vec2 <- runif(1e5)
> vec_in(vec1, vec2)
> vec1 %*% vec2
```

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

// Examples of RcppArmadillo functions below

// vec_in() calculates the inner (dot) product of two ve
// It accepts pointers to the two vectors and returns a
//' @export
// [[Rcpp::export]]
double vec_in(const arma::vec& vec1, const arma::vec& ve
  return arma::dot(vec1, vec2);
}  // end vec_in

// mat_2vec_in() calculates the inner (dot) product of a
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
//' @export
// [[Rcpp::export]]
double mat_2vec_in(const arma::vec& vec_tor2, const arma
  return arma::as_scalar(trans(vec_tor2) * (mat_rix * ve
}  // end mat_2vec_in
```

```
> # microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   vec_in=vec_in(vec1, vec2),
+   r_code=(vec1 %*% vec2),
+   times=100))[, c(1, 4, 5)]  # end microbenchm
> # microbenchmark shows:
> # vec_in() is several times faster than %*%, e
> #     expr     mean   median
> # 1 vec_in 110.7067 110.4530
> # 2 r_code 585.5127 591.3575
```

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="C:/Develop/R/lecture_sli
> mat_rix <- matrix(runif(1e5), nc=1e3)
> # de-mean using apply()
> new_mat <- apply(mat_rix, 2,
+   function(x) (x-mean(x)))
> # de-mean using demean_mat()
> demean_mat(mat_rix)
> all.equal(new_mat, mat_rix)
> # microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   demean_mat=demean_mat(mat_rix),
+   apply=(apply(mat_rix, 2, mean)),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # microbenchmark shows:
> # demean_mat() is over 70 times faster than apply()
> #         expr       mean   median
```

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

// Examples of RcppArmadillo functions below

// demean_mat() calculates a matrix with de-meaned colum
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
//' @export
// [[Rcpp::export]]
double demean_mat(arma::mat& mat_rix){
  for (unsigned int i = 0; i < mat_rix.n_cols; i++) {
    mat_rix.col(i) -= arma::mean(mat_rix.col(i));
  }  // end for
  return mat_rix.n_cols;
}  // end demean_mat

// inv_mat() calculates the inverse of symmetric positiv
// 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 RcppArmadillo.
//' @export
// [[Rcpp::export]]
double inv_mat(arma::mat& mat_rix){
  mat_rix = arma::inv_sympd(mat_rix);
  return mat_rix.n_cols;
}  // end inv_mat
```

Vector and Matrix Calculus

Let $\boldsymbol{v}$ and $\boldsymbol{w}$ be vectors, with $\boldsymbol{v} = \{v_i\}_{i=1}^{i=n}$, and let $\mathbb{1}$ be the unit vector, with $\mathbb{1} = \{1\}_{i=1}^{i=n}$,

Then the inner product of $\boldsymbol{v}$ and $\boldsymbol{w}$ can be written as $\boldsymbol{v}^T\boldsymbol{w} = \boldsymbol{w}^T\boldsymbol{v} = \sum_{i=1}^{n} v_i w_i$,

We can then express the sum of the elements of $\boldsymbol{v}$ as the inner product: $\boldsymbol{v}^T\mathbb{1} = \mathbb{1}^T\boldsymbol{v} = \sum_{i=1}^{n} v_i$,

And the sum of squares of $\boldsymbol{v}$ as the inner product: $\boldsymbol{v}^T\boldsymbol{v} = \sum_{i=1}^{n} v_i^2$,

Let $\mathbb{A}$ be a matrix, with $\mathbb{A} = \left\{A_{ij}\right\}_{i,j=1}^{i,j=n}$,

Then the inner product of matrix $\mathbb{A}$ with vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ can be written as:

$$\boldsymbol{v}^T\mathbb{A}\,\boldsymbol{w} = \boldsymbol{w}^T\mathbb{A}^T\boldsymbol{v} = \sum_{i,j=1}^{n} A_{ij} v_i w_j$$

The derivative of a scalar variable with respect to a vector variable is a vector, for example:

$$\frac{d(\boldsymbol{v}^T\mathbb{1})}{d\boldsymbol{v}} = d_v[\boldsymbol{v}^T\mathbb{1}] = d_v[\mathbb{1}^T\boldsymbol{v}] = \mathbb{1}^T$$

$$d_v[\boldsymbol{v}^T\boldsymbol{w}] = d_v[\boldsymbol{w}^T\boldsymbol{v}] = \boldsymbol{w}^T$$

$$d_v[\boldsymbol{v}^T\mathbb{A}\,\boldsymbol{w}] = \boldsymbol{w}^T\mathbb{A}^T$$

$$d_v[\boldsymbol{v}^T\mathbb{A}\,\boldsymbol{v}] = \boldsymbol{v}^T\mathbb{A} + \boldsymbol{v}^T\mathbb{A}^T$$

Eigenvectors and Eigenvalues of Matrices

The vector w is an *eigenvector* of the matrix $\mathbb{A}$, if it satisfies the *eigenvalue* equation:

$$\mathbb{A}\, w = \lambda\, w$$

Where λ is the *eigenvalue* corresponding to the *eigenvector* w,

The number of *eigenvalues* of a matrix is equal to its dimension,

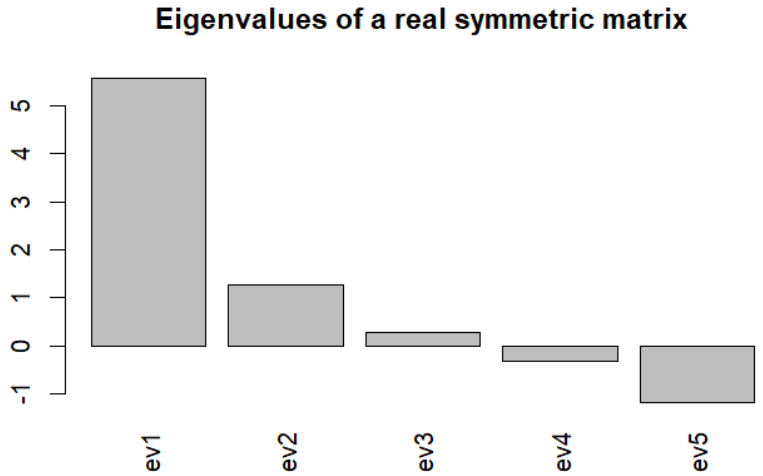
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other,

The *eigenvectors* can be normalized to 1,

The *eigenvectors* form an *orthonormal basis* in which the matrix $\mathbb{A}$ is diagonal,

The function `eigen()` calculates the *eigenvectors* and *eigenvalues* of numeric matrices,

An excellent interactive visualization of *eigenvectors* and *eigenvalues* is available here:
http://setosa.io/ev/eigenvectors-and-eigenvalues/




```
> # create random real symmetric matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- mat_rix + t(mat_rix)
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(mat_rix)
> eigen_vec <- ei_gen$vectors
> dim(eigen_vec)
> # plot eigenvalues
> barplot(ei_gen$values,
+   xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(ei_gen$values)),
+   main="Eigenvalues of a real symmetric matrix")
```

Eigen Decomposition of Matrices

Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other,

The *eigenvectors* form an *orthonormal basis* in which the matrix $\mathbb{A}$ is diagonal:

$$\mathbb{D} = \mathbb{O}^T \mathbb{A} \mathbb{O}$$

Where $\mathbb{D}$ is a *diagonal* matrix containing the *eigenvalues* of matrix $\mathbb{A}$, and $\mathbb{O}$ is an *orthogonal* matrix of its *eigenvectors*,

Any real symmetric matrix $\mathbb{A}$ can be decomposed into a product of its *eigenvalues* and its *eigenvectors* (the *eigen decomposition*):

$$\mathbb{A} = \mathbb{O} \mathbb{D} \mathbb{O}^T$$

The *eigen decomposition* expresses a matrix as the product of a rotation, followed by a scaling, followed by the inverse rotation,

```
> # eigenvectors form an orthonormal basis
> round(t(eigen_vec) %*% eigen_vec,
+   digits=4)
> # diagonalize matrix using eigenvector matrix
> round(t(eigen_vec) %*% (mat_rix %*% eigen_vec)
+   digits=4)
> ei_gen$values
> # eigen decomposition of matrix by rotating th
> eigen_decomp <- eigen_vec %*% (ei_gen$values *
> # create diagonal matrix of eigenvalues
> # diago_nal <- diag(ei_gen$values)
> # eigen_decomp <- eigen_vec %*% (diago_nal %*%
> all.equal(mat_rix, eigen_decomp)
```

Orthogonal matrices represent rotations in *hyperspace*, and their inverse is equal to their transpose: $\mathbb{O}^{-1} = \mathbb{O}^T$,

The *diagonal* matrix $\mathbb{D}$ represents a scaling (stretching) transformation proportional to the *eigenvalues*,

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices,

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so

Positive Definite Matrices

Matrices with positive *eigenvalues* are called *positive definite* matrices,

Matrices with non-negative *eigenvalues* are called *positive semi-definite* matrices (some of their *eigenvalues* may be zero),

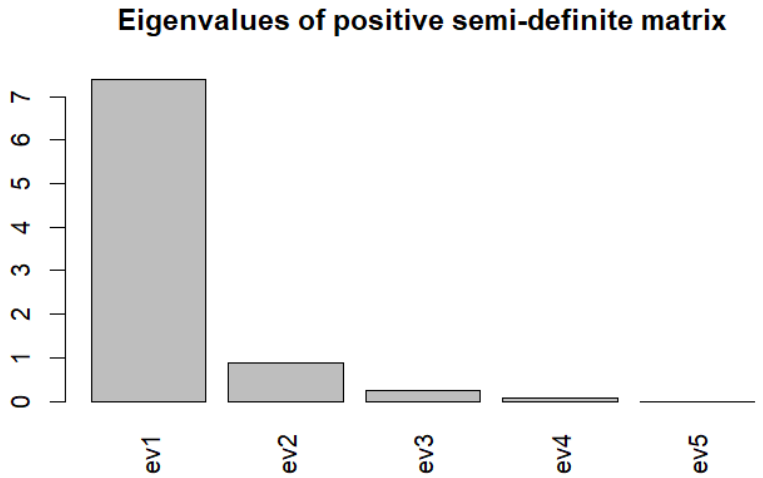
An example of *positive definite* matrices are the covariance matrices of linearly independent variables,

But the covariance matrices of linearly dependent variables have some *eigenvalues* equal to zero, in which case they are *singular*, and only *positive semi-definite*,

All covariance matrices are *positive semi-definite* and all *positive semi-definite* matrices are the covariance matrix of some multivariate distribution,

Matrices which have some *eigenvalues* equal to zero are called *singular* (degenerate) matrices,

For any real matrix $\mathbb{A}$, the matrix $\mathbb{A}^T\mathbb{A}$ is *positive semi-definite*,




```
> # create random positive semi-definite matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(mat_rix)
> ei_gen$values
> # plot eigenvalues
> barplot(ei_gen$values, las=3,
+   xlab="", ylab="",
+   names.arg=paste0("ev", 1:NROW(ei_gen$values)),
+   main="Eigenvalues of positive semi-definite
```

Singular Value Decomposition (*SVD*) of Matrices

The *Singular Value Decomposition* (*SVD*) is a generalization of the *eigen decomposition* of *positive semi-definite* matrices,

The *SVD* of a rectangular matrix $\mathbb{A}$ with dimensions `m` rows and `n` columns is defined as the factorization:

$$\mathbb{A} = \mathbb{U}\mathbf{\Sigma}\mathbb{V}^T$$

If (`m > n`), then $\mathbb{U}$ is an (`m x n`) rectangular matrix, $\mathbf{\Sigma}$ is an (`n x n`) diagonal matrix containing the *singular values*, and $\mathbb{V}$ is an (`n x n`) *orthogonal* matrix, and vice versa if (`m < n`),

The (`m x n`) rectangular matrix $\mathbb{U}$ consists of `n` columns of *orthonormal* left-*singular* vectors, with $\mathbb{U}^T\mathbb{U} = \mathbb{1}$,

The columns of the *orthogonal* matrix $\mathbb{V}$ consist of `n` *orthonormal* right-*singular* vectors, with $\mathbb{V}^T\mathbb{V} = \mathbb{1}$,

The *singular* vectors are only defined up to a reflection (sign), i.e. if `vec` is a *singular* vector, then so is `-vec`,

```
> # dimensions of left and right matrices
> n_left <- 6 ; n_right <- 4
> # create random positive semi-definite matrix
> left_mat <- matrix(runif(n_left^2), nc=n_left)
> left_mat <- crossprod(left_mat)
> # or
> left_mat <- left_mat %*% t(left_mat)
> # calculate left eigenvectors
> ei_gen <- eigen(left_mat)
> left_mat <- ei_gen$vectors[, 1:n_right]
> # create random positive semi-definite matrix
> right_mat <- matrix(runif(n_right^2), nc=n_rig
> right_mat <- crossprod(right_mat)
> # or
> right_mat <- right_mat %*% t(right_mat)
> # calculate right eigenvectors and singular va
> ei_gen <- eigen(right_mat)
> right_mat <- ei_gen$vectors
> sing_values <- ei_gen$values
> # compose rectangular matrix
> mat_rix <-
+   left_mat %*% (sing_values * t(right_mat))
> # mat_rix <- left_mat %*% diag(sing_values) %*
```

In the special case when $\mathbb{A}$ is a *positive semi-definite* matrix, the *SVD* reduces to the *eigen decomposition*, with $\mathbb{U} = \mathbb{V}$,

Singular Value Decomposition Using Function svd()

The *SVD* of a rectangular matrix $\mathbb{A}$ with dimensions m rows and n columns is defined as the factorization:

$$\mathbb{A} = \mathbb{U}\mathbf{\Sigma}\mathbb{V}^T$$

The (m x n) rectangular matrix $\mathbb{U}$ consists of n columns of *orthonormal* left-*singular* vectors, with $\mathbb{U}^T\mathbb{U} = \mathbb{1}$,

The left-*singular* vectors are the *eigenvectors* of the matrix $\mathbb{A}\mathbb{A}^T$,

The columns of the *orthogonal* matrix $\mathbb{V}$ consist of n *orthonormal* right-*singular* vectors, with $\mathbb{V}^T\mathbb{V} = \mathbb{1}$,

The right-*singular* vectors are the *eigenvectors* of the matrix $\mathbb{A}^T\mathbb{A}$,

The left-*singular* matrix $\mathbb{U}$ combined with the right-*singular* matrix $\mathbb{V}$ define a rotation transformation into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

$$\mathbf{\Sigma} = \mathbb{U}^T\mathbb{A}\mathbb{V}$$

```
> # perform singular value decomposition
> s_vd <- svd(mat_rix)
> # compare SVD with inputs
> all.equal(abs(s_vd$u), abs(left_mat))
> all.equal(abs(s_vd$v), abs(right_mat))
> all.equal(s_vd$d, ei_gen$values)
```

The function svd() performs *Singular Value Decomposition* (*SVD*) of a rectangular matrix, and returns a list of three elements: the *singular values*, and the matrices of left-*singular* vectors and the right-*singular* vectors,

Inverse of Square Matrices

The inverse of a square matrix $\mathbb{A}$ is defined as a square matrix $\mathbb{A}^{-1}$ that satisfies the equation:

$$\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$$

Where $\mathbb{1}$ is the identity matrix,

The inverse $\mathbb{A}^{-1}$ matrix can also be expressed as a product of the inverse of its *eigenvalues* ($\mathbb{D}$) and its *eigenvectors* ($\mathbb{O}$):

$$\mathbb{A}^{-1} = \mathbb{O}\,\mathbb{D}^{-1}\,\mathbb{O}^{T}$$

But *singular* (degenerate) matrices (which have some *eigenvalues* equal to zero) don't have an inverse,

The function `solve()` solves systems of linear equations, and also inverts square matrices,

```
> # create random positive semi-definite matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate the inverse of mat_rix
> in_verse <- solve(a=mat_rix)
> # multiply inverse with matrix
> round(in_verse %*% mat_rix, 4)
> round(mat_rix %*% in_verse, 4)
>
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(mat_rix)
> eigen_vec <- ei_gen$vectors
>
> # perform eigen decomposition of inverse
> eigen_inverse <-
+   eigen_vec %*% (t(eigen_vec) / ei_gen$values)
> all.equal(in_verse, eigen_inverse)
> # decompose diagonal matrix with inverse of ei
> # diago_nal <- diag(1/ei_gen$values)
> # eigen_inverse <-
> #   eigen_vec %*% (diago_nal %*% t(eigen_vec))
```

Generalized Inverse of Rectangular Matrices

The generalized inverse of an (`m x n`) rectangular matrix $\mathbb{A}$ is defined as an (`n x m`) matrix $\mathbb{A}^{-1}$ that satisfies the equation:

$$\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$$

The generalized inverse matrix $\mathbb{A}^{-1}$ can be expressed as a product of the inverse of its *singular values* (Σ) and its left and right *singular* matrices ($\mathbb{U}$ and $\mathbb{V}$):

$$\mathbb{A}^{-1} = \mathbb{V}\,\Sigma^{-1}\,\mathbb{U}^T$$

The generalized inverse $\mathbb{A}^{-1}$ can also be expressed as the *Moore-Penrose pseudo-inverse*:

$$\mathbb{A}^{-1} = (\mathbb{A}^T\mathbb{A})^{-1}\mathbb{A}^T$$

In the case when the inverse matrix $\mathbb{A}^{-1}$ exists, then the *pseudo-inverse* matrix simplifies to the inverse: $(\mathbb{A}^T\mathbb{A})^{-1}\mathbb{A}^T = \mathbb{A}^{-1}(\mathbb{A}^T)^{-1}\mathbb{A}^T = \mathbb{A}^{-1}$

The function `MASS::ginv()` calculates the generalized inverse of a matrix,

```
> # create random rectangular matrix
> # case when: n_left > n_right
> n_left <- 6 ; n_right <- 4
> mat_rix <- matrix(runif(n_left*n_right),
+   nc=n_right)
> # calculate generalized inverse of mat_rix
> in_verse <- MASS::ginv(mat_rix)
> round(in_verse %*% mat_rix, 4)
> all.equal(mat_rix,
+     mat_rix %*% in_verse %*% mat_rix)
> # create random rectangular matrix
> # case when: n_left < n_right
> n_left <- 4 ; n_right <- 6
> mat_rix <- matrix(runif(n_left*n_right),
+   nc=n_right)
> # calculate generalized inverse of mat_rix
> in_verse <- MASS::ginv(mat_rix)
> round(mat_rix %*% in_verse, 4)
> # perform singular value decomposition
> s_vd <- svd(mat_rix)
> # calculate generalized inverse from SVD
> svd_inverse <- s_vd$v %*% (t(s_vd$u) / s_vd$d)
> all.equal(svd_inverse, in_verse)
> # calculate Moore-Penrose pseudo-inverse
> mp_inverse <-
+   MASS::ginv(t(mat_rix) %*% mat_rix) %*% t(mat
> all.equal(mp_inverse, in_verse)
```

Generalized Inverse of Singular Matrices

Singular matrices have some *singular values* equal to zero, so they don't have an inverse matrix which satisfies the equation:
$\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$,

But if the *singular values* that are equal to zero are removed, then a generalized inverse for *singular* matrices can be specified by:

$$\mathbb{A}^{-1} = \mathbb{V}_n \, \mathbf{\Sigma}_n^{-1} \, \mathbb{U}_n^T$$

Where $\mathbb{U}_n$, $\mathbb{V}_n$ and $\mathbf{\Sigma}_n$ are the *SVD* matrices with rows and columns corresponding to zero *singular values* removed,

```
> # create random singular matrix
> n_left <- 4 ; n_right <- 6
> mat_rix <- matrix(runif(n_left*n_right), nc=n_right)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate generalized inverse of mat_rix
> in_verse <- MASS::ginv(mat_rix)
> # verify inverse of mat_rix
> all.equal(mat_rix,
+   mat_rix %*% in_verse %*% mat_rix)
```

```
> # perform singular value decomposition
> s_vd <- svd(mat_rix)
> # set tolerance for determining zero singular
> to_l <- sqrt(.Machine$double.eps)
> # check for zero singular values
> s_vd$d
> not_zero <- (s_vd$d > (to_l * s_vd$d[1]))
> # calculate generalized inverse from SVD
> svd_inverse <-
+   s_vd$v[, not_zero] %*%
+   (t(s_vd$u[, not_zero]) / s_vd$d[not_zero])
> all.equal(svd_inverse, in_verse)
> # calculate Moore-Penrose pseudo-inverse
> mp_inverse <-
+   MASS::ginv(t(mat_rix) %*% mat_rix) %*% t(mat
> all.equal(mp_inverse, in_verse)
```

Diagonalizing Generalized Inverse of Singular Matrices

The left-*singular* matrix $\mathbb{U}$ combined with the right-*singular* matrix $\mathbb{V}$ define a rotation transformation into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

$$\Sigma = \mathbb{U}^T \mathbb{A} \mathbb{V}$$

The generalized inverse of *singular* matrices doesn't satisfy the equation: $\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$, but if it's rotated into the same coordinate system where $\mathbb{A}$ is diagonal, then we have:

$$\mathbb{U}^T (\mathbb{A}^{-1}\mathbb{A}) \mathbb{V} = \mathbb{1}_n$$

So that $\mathbb{A}^{-1}\mathbb{A}$ is diagonal in the same coordinate system where $\mathbb{A}$ is diagonal,

```
> # diagonalize the "unit" matrix
> uni_t <- mat_rix %*% in_verse
> round(uni_t, 4)
> round(mat_rix %*% in_verse, 4)
> round(t(s_vd$u) %*% uni_t %*% s_vd$v, 4)
```

Solving Linear Equations Using solve()

A system of linear equations can be defined as:

$$\mathbb{A}\, x = b$$

Where $\mathbb{A}$ is a matrix, b is a vector, and x is the unknown vector,

The solution of the system of linear equations is equal to:

$$x = \mathbb{A}^{-1} b$$

Where $\mathbb{A}^{-1}$ is the *inverse* of the matrix $\mathbb{A}$,

The function solve() solves systems of linear equations, and also inverts square matrices,

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices,

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so that each pair produces a single number:

```
> # define a square matrix
> mat_rix <- matrix(c(1, 2, -1, 2), nc=2)
> vec_tor <- c(2, 1)
> # calculate the inverse of mat_rix
> in_verse <- solve(a=mat_rix)
> in_verse %*% mat_rix
> # calculate solution using inverse of mat_rix
> solu_tion <- in_verse %*% vec_tor
> mat_rix %*% solu_tion
> # calculate solution of linear system
> solu_tion <- solve(a=mat_rix, b=vec_tor)
> mat_rix %*% solu_tion
```

Cholesky Decomposition

The *Cholesky* decomposition of a *positive definite* matrix $\mathbb{A}$ is defined as:

$$\mathbb{A} = \mathbb{L}^T \mathbb{L}$$

Where $\mathbb{L}$ is an upper triangular matrix with positive diagonal elements,

The matrix $\mathbb{L}$ can be considered the square root of $\mathbb{A}$,

The vast majority of random *positive semi-definite* matrices are also *positive definite*,

The function `chol()` calculates the *Cholesky* decomposition of a *positive definite* matrix,

The functions `chol2inv()` and `chol()` calculate the inverse of a *positive definite* matrix two times faster than `solve()`,

```
> # create large random positive semi-definite m
> mat_rix <- matrix(runif(1e4), nc=100)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate eigen decomposition
> ei_gen <- eigen(mat_rix)
> eigen_values <- ei_gen$values
> eigen_vec <- ei_gen$vectors
> # set tolerance for determining zero singular
> to_l <- sqrt(.Machine$double.eps)
> # if needed convert to positive definite matri
> not_zero <- (eigen_values > (to_l * eigen_valu
> if (sum(!not_zero) > 0) {
+   eigen_values[!not_zero] <- 2*to_l
+   mat_rix <- eigen_vec %*%
+     (eigen_values * t(eigen_vec))
+ }  # end if
> # calculate the Cholesky mat_rix
> choles_ky <- chol(mat_rix)
> choles_ky[1:5, 1:5]
> all.equal(mat_rix, t(choles_ky) %*% choles_ky)
> # calculate inverse from Cholesky
> chol_inverse <- chol2inv(choles_ky)
> all.equal(solve(mat_rix), chol_inverse)
> # compare speed of Cholesky inversion
> library(microbenchmark)
> summary(microbenchmark(
+   sol_ve=solve(mat_rix),
+   choles_ky=chol2inv(chol(mat_rix)),
```

Simulating Correlated Returns Using Cholesky Matrix

The *Cholesky* decomposition of a covariance matrix can be used to simulate correlated *Normal* returns following the given covariance matrix: $\mathbb{C} = \mathbb{L}^T\mathbb{L}$

Let $\mathbb{R}$ be a matrix with columns of *uncorrelated* returns following the *Standard Normal* distribution,

The *correlated* returns $\mathbb{R}_c$ can be calculated from the *uncorrelated* returns $\mathbb{R}$ by multilying them by the *Cholesky* matrix $\mathbb{L}$:

$$\mathbb{R}_c = \mathbb{L}^T\mathbb{R}$$

```
> # calculate random covariance matrix
> cov_mat <- matrix(runif(25), nc=5)
> cov_mat <- t(cov_mat) %*% cov_mat
> # calculate the Cholesky mat_rix
> choles_ky <- chol(cov_mat)
> choles_ky
> # simulate random uncorrelated returns
> n_assets <- 5
> n_rows <- 10000
> re_turns <- matrix(rnorm(n_assets*n_rows), nc=
> # calculate correlated returns by applying Cho
> corr_returns <- re_turns %*% choles_ky
> # calculate covariance matrix
> cov_returns <- crossprod(corr_returns) / (n_ro
> all.equal(cov_mat, cov_returns)
```

Eigenvalues of the Covariance Matrix

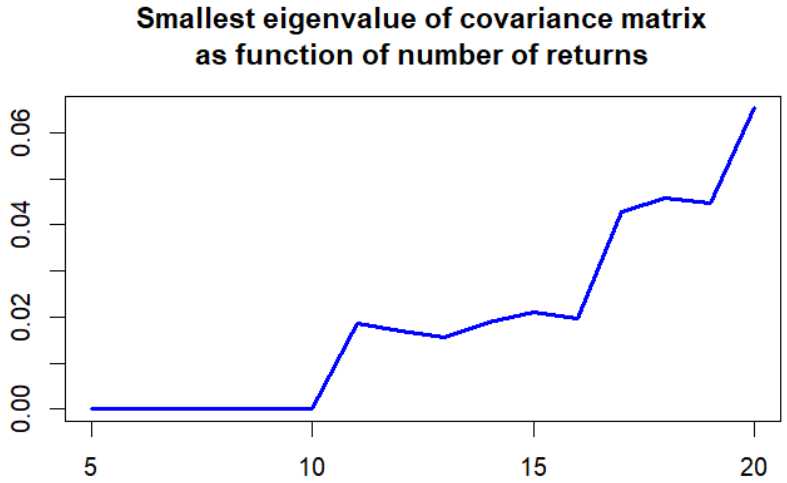
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of k assets (columns), over n time periods (rows), then the sample covariance matrix is equal to:

$$\mathbb{C} = \mathbb{R}^T \mathbb{R}/(n-1)$$

If the number of time periods of returns is less than the number of portfolio assets, then the returns are collinear, and the sample covariance matrix is *singular* (some *eigenvalues* are zero),

The function crossprod() performs *inner* (*scalar*) multiplication, exactly the same as the %*% operator, but it is slightly faster,

```
> # simulate random portfolio returns
> n_assets <- 10
> n_rows <- 100
> re_turns <- matrix(rnorm(n_assets*n_rows), nc
> # de-mean the returns
> re_turns <- apply(re_turns, MARGIN=2, functio
> # calculate covariance matrix
> cov_mat <- crossprod(re_turns) / (n_rows-1)
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(cov_mat)
> ei_gen$values
> barplot(ei_gen$values, # plot eigenvalues
```




```
> # calculate eigenvectors and eigenvalues
> # as function of number of returns
> n_data <- ((n_assets/2):(2*n_assets))
> e_values <- sapply(n_data, function(x) {
+   re_turns <- re_turns[1:x, ]
+   re_turns <- apply(re_turns, MARGIN=2,
+     function(y) (y-mean(y)))
+   cov_mat <- crossprod(re_turns) / (x-1)
+   min(eigen(cov_mat)$values)
+ })  # end sapply
> plot(y=e_values, x=n_data, t="l",
+   xlab="", ylab="", lwd=3, col="blue",
+   main="Smallest eigenvalue of covariance matr
```

draft: Data Science

Data Science is very important to quantitative finance,

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

```
> lev_el <- 20  # barrier level
> len_gth <- 1000  # number of simulation steps
> pa_th <- numeric(len_gth)  # allocate path vec
> pa_th[1] <- 0  # initialize path
> in_dex <- 2  # initialize simulation index
> while ((in_dex <= len_gth) &&
+  (pa_th[in_dex - 1] < lev_el)) {
+ # simulate next step
+   pa_th[in_dex] <-
+     pa_th[in_dex - 1] + rnorm(1)
+   in_dex <- in_dex + 1  # advance in_dex
+ }  # end while
> # fill remaining pa_th after it crosses lev_el
> if (in_dex <= len_gth)
+   pa_th[in_dex:len_gth] <- pa_th[in_dex - 1]
> # create daily time series starting 2011
> ts_path <- ts(data=pa_th, frequency=365, start=c(2011, 1))
> plot(ts_path, type="l", col="black",  # create plot
+      lty="solid", xlab="", ylab="")
> abline(h=lev_el, lwd=2, col="red")  # add horizontal line
> title(main="Brownian motion crossing a barrier level",
+       line=0.5)
```

1. Data is never clean.
2. You will spend most of your time cleaning and preparing data.
3. 95% of tasks do not require deep learning.
4. In 90% of cases generalized linear regression will do the trick.
5. Big Data is just a tool.
6. You should embrace the Bayesian approach.
7. No one cares how you did it.
8. Academia and business are two different worlds.
9. Presentation is key – be a master of Power Point.
10. All models are false, but some are useful.
11. There is no fully automated Data Science. You need to get your hands dirty