Financial Simulation using R

Majeed Simaan[1]

[1]Lally School of Management at RPI

Feb 2, 2018

Agenda

- Crash Course in R
  - Basics
  - Loops and the `apply` functions
  - Data Manipulation
  - Plotting
- Simulations
  - Distributions in R
  - Generating Random Numbers
  - Density Functions and Moments
- Option Pricing
  - Simulating Stock Prices
  - Pricing Options

Agenda

- Crash Course in R
    - Basics
    - Loops and the `apply` functions
    - Data Manipulation
    - Plotting
- Simulations
    - Distributions in R
    - Generating Random Numbers
    - Density Functions and Moments
- Option Pricing
    - Simulating Stock Prices
    - Pricing Options
- If time allows...
    - Open Source Data (`quantmod` package)
    - Financial Time Series (`xts` package)

Suggested Readings and Resources

- Intro
  1. Lally School January 2015 R Bootcamp (slides)
  2. The R manuals (see link)
  3. DataCamp
  4. The Art of R Programming by Matloff (2011) (see link)

Suggested Readings and Resources

- Intro
  1. Lally School January 2015 R Bootcamp (slides)
  2. The R manuals (see link)
  3. DataCamp
  4. The Art of R Programming by Matloff (2011) (see link)
- Advanced
  1. The R Inferno by Burns (2011) (see link)
  2. Advanced R by Wickham (2014) (see link)

Suggested Readings and Resources

- Intro
  1. Lally School January 2015 R Bootcamp (slides)
  2. The R manuals (see link)
  3. DataCamp
  4. The Art of R Programming by Matloff (2011) (see link)
- Advanced
  1. The R Inferno by Burns (2011) (see link)
  2. Advanced R by Wickham (2014) (see link)
- Applications
  1. R for Data Science by Grolemund and Wickham (2016) (see link)
  2. Statistical Learning by James et al., 2014
  3. Financial Modeling by Ang, 2015
  4. Analysis of Financial Time Series by Tsay (2010) (see link)

Introduction

Types of Data

- R has five basic classes
  1. Logical
  2. Integer
  3. Numeric
  4. Character
  5. Complex
- The most basic object is a **vector**
  - Can only contain objects of the same class
  - The same applies to matrices
- A **list** can have objects of different types
- **In R, not everything is a matrix!**

Basics

- Can use either <- or = to assign new objects

```
> x <- 1:5
> x
```

```
[1] 1 2 3 4 5
```

Basics

- Can use either <- or = to assign new objects

```
> x <- 1:5
> x

[1] 1 2 3 4 5
```

- Can also assign labels to objects

```
> names(x) <- letters[1:5] # assign names to x
> x

a b c d e
1 2 3 4 5
```

- Note that R has a number of built-in objects, e.g. `letters`

Basics

- Can use either <- or = to assign new objects

```
> x <- 1:5
> x

[1] 1 2 3 4 5
```

- Can also assign labels to objects

```
> names(x) <- letters[1:5] # assign names to x
> x

a b c d e
1 2 3 4 5
```

- Note that R has a number of built-in objects, e.g. `letters`
- Can pull data using indices or labels, e.g.

```
> x[1:2] # pulls the first two elements

a b
1 2

> x["c"] # pulls the element labeled c

c
3
```

- Returns NA if label/index does not exist, e.g. x["z"]

- Note that x is a vector but not a matrix

```
> dim(x)

NULL
```

- If we define x as a matrix

```
> X <- as.matrix(x)
> dim(X)

[1] 5 1
```

- Note that x is a vector but not a matrix

```
> dim(x)
```

```
NULL
```

- If we define x as a matrix

```
> X <- as.matrix(x)
> dim(X)
```

```
[1] 5 1
```

- R uses the function c to combine vectors or lists

```
> x2 <- c(x,x,x,x,x)
> x2
```

```
a b c d e a b c d e a b c d e a b c d e a b c d e
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

- Note that `x` is a vector but not a matrix

```
> dim(x)

NULL
```

- If we define `x` as a matrix

```
> X <- as.matrix(x)
> dim(X)

[1] 5 1
```

- R uses the function `c` to combine vectors or lists

```
> x2 <- c(x,x,x,x,x)
> x2

a b c d e a b c d e a b c d e a b c d e a b c d e
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

- You can also stack vectors in a matrix using the `matrix` function

```
> X2 <- matrix(x2,length(x))
> X2

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    2    2    2    2    2
[3,]    3    3    3    3    3
[4,]    4    4    4    4    4
[5,]    5    5    5    5    5
```

- R has a repetitive nature

  > x + 1:10

  ```
   [1]  2  4  6  8 10  7  9 11 13 15
  ```

- whereas

  > x + 1:9

  ```
  [1]  2  4  6  8 10  7  9 11 13
  Warning message:
  In x + 1:9 :
    longer object length is not a multiple of shorter object length
  ```

- R has a repetitive nature

```
> x + 1:10

 [1]  2  4  6  8 10  7  9 11 13 15
```

- whereas

```
> x + 1:9

[1]  2  4  6  8 10  7  9 11 13
Warning message:
In x + 1:9 :
  longer object length is not a multiple of shorter object length
```

- One benefit of repetitions is creating larger dimensions using smaller one

```
# create matrix using the vector x
> X3 <- matrix(x,length(x),length(x))


> identical(X3,X2) # returns TRUE if two objects are identical


[1] TRUE
```

- Similar to vectors, rows and columns of matrices can be assigned labels

```
> rownames(X3) <- letters[1:nrow(X3)]
> colnames(X3) <- LETTERS[1:ncol(X3)]
> X3.subset <- X3[c("a","c"),c("D","E")]
> X3.subset
```

```
  D E
a 1 1
c 3 3
```

- Note that `.` is not different than any other letter

- Similar to vectors, rows and columns of matrices can be assigned labels

```
> rownames(X3) <- letters[1:nrow(X3)]
> colnames(X3) <- LETTERS[1:ncol(X3)]
> X3.subset <- X3[c("a","c"),c("D","E")]
> X3.subset
```

```
  D E
a 1 1
c 3 3
```

- Note that . is not different than any other letter

- Alternatively, one can use indices to pull items from a matrix

```
> X3.subset2 <- X3[c(1,3),c(4,5)]
> all(X3.subset == X3.subset2)
```

```
[1] TRUE
```

- Also, to refer to a row/column, it suffices to call either dimension, e.g.

```
> X3[1,]
```

```
A B C D E
1 1 1 1 1
```

```
> X3[,1]
```

```
a b c d e
1 2 3 4 5
```

Practice

1. Write a one line command that stacks all numbers from 1 to 100 into a 10×10 squared matrix named `M`, such that

```
> M

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1    2    3    4    5    6    7    8    9    10
 [2,]   11   12   13   14   15   16   17   18   19    20
 [3,]   21   22   23   24   25   26   27   28   29    30
 [4,]   31   32   33   34   35   36   37   38   39    40
 [5,]   41   42   43   44   45   46   47   48   49    50
 [6,]   51   52   53   54   55   56   57   58   59    60
 [7,]   61   62   63   64   65   66   67   68   69    70
 [8,]   71   72   73   74   75   76   77   78   79    80
 [9,]   81   82   83   84   85   86   87   88   89    90
[10,]   91   92   93   94   95   96   97   98   99   100
```

   Hint: *note that by default*
   `matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)`

2. Use common summary functions you already from MATLAB to summarize the `M` matrix, e.g. min, max, mean, etc...
   Hint: *note that common functions have the same name in MATLAB and R (see the R/MATLAB manual for a reference `https://cran.r-project.org/doc/contrib/Hiebeler-matlabR.pdf`)*

Functions

- Unlike MATLAB, R does not require functions to be stored in separate files
- Multiple functions can be defined in the main script

Functions

- Unlike MATLAB, R does not require functions to be stored in separate files
- Multiple functions can be defined in the main script

- To define a function f that, for instance, returns the first and last elements of a given vector x, one can write

```
> f1 <- function(x) {
+   x1 <- x[1]
+   x2 <- x[length(x)]
+   return(c(x1,x2)) # functions must return something
+ }
>
> f(x)

a e
1 5
```

- Note that the braces { } are needed to set the start and end of a function

Functions

- Unlike MATLAB, R does not require functions to be stored in separate files
- Multiple functions can be defined in the main script

- To define a function `f` that, for instance, returns the first and last elements of a given vector `x`, one can write

```
> f1 <- function(x) {
+   x1 <- x[1]
+   x2 <- x[length(x)]
+   return(c(x1,x2)) # functions must return something
+ }
>
> f(x)

a e
1 5
```

- Note that the braces { } are needed to set the start and end of a function
- Similar to the @(x) command in MATLAB, one can write

```
> f2 <- function(x) c(x[1],x[length(x)])
> f2(x)

a e
1 5
```

- Functions in R can load objects defined outside the function, for instance

```
> f3 <- function(x) mean((x - mean(x)) > a*sd(x))
> a <- 0.5
> f3(x)

[1] 0.4

> a <- 1
> f3(x)

[1] 0.2
```

- However, this also increases the chances to commit an error in the code

- Functions in R can load objects defined outside the function, for instance

```
> f3 <- function(x) mean((x - mean(x)) > a*sd(x))
> a <- 0.5
> f3(x)

[1] 0.4

> a <- 1
> f3(x)

[1] 0.2
```

- However, this also increases the chances to commit an error in the code

Practice

Write a function named `M.f` that takes two arguments: matrix `M` and constant a. The function should return the proportion of elements that are larger than a. For instance, the function should return

```
> M.f(M,-Inf)

[1] 1

> M.f(M,Inf)

[1] 0
```

where R defines `Inf` as an infinite number

Loops

- Similar to MATLAB, R has `for` and `while` loops
- The `for` loop

```
> M.mean2 <- numeric() # define an empty numeric object
> for(i in 1:ncol(M) ) {
+   mean.i <- mean(M[,i])
+   M.mean2 <- c(M.mean2, mean.i)
+   }
> M.mean2
```

```
 [1] 46 47 48 49 50 51 52 53 54 55
```

Loops

- Similar to MATLAB, R has for and while loops
- The for loop

```
> M.mean2 <- numeric() # define an empty numeric object
> for(i in 1:ncol(M) ) {
+   mean.i <- mean(M[,i])
+   M.mean2 <- c(M.mean2, mean.i)
+   }
> M.mean2
```

```
 [1] 46 47 48 49 50 51 52 53 54 55
```

- The while loop

```
> i <- 1
> M.mean2 <- numeric() # define an empty numeric object
> while (i <= ncol(M)) {
+   mean.i <- mean(M[,i])
+   M.mean2 <- c(M.mean2, mean.i)
+   i <- i + 1
+   }
> M.mean2
```

```
 [1] 46 47 48 49 50 51 52 53 54 55
```

The apply functions

- It is recommended to avoid loops when one can
- Fortunately, R provides a number of good alternatives

The apply functions

- It is recommended to avoid loops when one can
- Fortunately, R provides a number of good alternatives
    1. `apply`: applies a function over rows/columns of a given matrix, e.g.

```
> apply(M, 1, mean) # for rows
 [1]  5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
> apply(M, 2, mean) # for columns
 [1] 46 47 48 49 50 51 52 53 54 55
```

The apply functions

- It is recommended to avoid loops when one can
- Fortunately, R provides a number of good alternatives
  1. `apply`: applies a function over rows/columns of a given matrix, e.g.

     ```
     > apply(M, 1, mean) # for rows
      [1]  5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
     > apply(M, 2, mean) # for columns
      [1] 46 47 48 49 50 51 52 53 54 55
     ```

  2. `sapply`: applies a function over a given set, e.g.

     ```
     > sapply(1:ncol(M), function(i) mean(M[,i]))
      [1] 46 47 48 49 50 51 52 53 54 55
     ```

The apply functions

- It is recommended to avoid loops when one can
- Fortunately, R provides a number of good alternatives
  1. apply: applies a function over rows/columns of a given matrix, e.g.

     ```
     > apply(M, 1, mean) # for rows
      [1]  5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
     > apply(M, 2, mean) # for columns
      [1] 46 47 48 49 50 51 52 53 54 55
     ```
  2. sapply: applies a function over a given set, e.g.

     ```
     > sapply(1:ncol(M), function(i) mean(M[,i]))
      [1] 46 47 48 49 50 51 52 53 54 55
     ```
- Using either, one can define a temporary function to be applied:

  ```
  > apply(M, 2, function(x)  mean(((x - mean(x))/sd(x))^2)     )
   [1] 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9
  ```

Practice

The `EuStockMarkets` object is a built-in dataset that contains data on 4 major European stock indices. You will need to

1. store the data into an object named `ds`
2. use a loop to compute the min and max of each index in the data
   - report the results in a 2×4 matrix, where columns are named correspondingly
3. repeat the above using the `apply` function
   - can you do it in one line?
4. finally, how much time did you save by using the `apply` function?
   - **hint** use the `Sys.time()` command to capture the system's time

Data Frames

- The problem with matrices is that all columns must have the same format

```
> i <- 1:26
> l <- letters
> L <- cbind(i,l) # combines columns
> sapply(1:ncol(L),function(i) class(L[,i])  )

          i           l
"character" "character"
```

- A `list` object can combine objects of different formats

Data Frames

- The problem with matrices is that all columns must have the same format

```
> i <- 1:26
> l <- letters
> L <- cbind(i,l) # combines columns
> sapply(1:ncol(L),function(i) class(L[,i])  )

          i           l
"character" "character"
```

- A `list` object can combine objects of different formats

- Nevertheless, `data.frame` objects provide a better solution

```
> L2 <- data.frame(i,l, stringsAsFactors = F) # avoid factors
> sapply(1:ncol(L2),function(i) class(L2[,i])  )

[1] "integer"   "character"

> summary(L2)

       i              l            
 Min.   : 1.00   Length:26         
 1st Qu.: 7.25   Class :character  
 Median :13.50   Mode  :character  
 Mean   :13.50                     
 3rd Qu.:19.75                     
 Max.   :26.00                     
```

- The summary is a “cheap” way to look at the data

- The plyr package (Wickham, 2011) is very useful to manipulate data frames (df)
- Let's take a look at the iris built-in data frame

```
> summary(iris,digits = 2)
```

```
  Sepal.Length   Sepal.Width   Petal.Length   Petal.Width          Species
 Min.   :4.3   Min.   :2.0   Min.   :1.0   Min.   :0.1   setosa    :50
 1st Qu.:5.1   1st Qu.:2.8   1st Qu.:1.6   1st Qu.:0.3   versicolor:50
 Median :5.8   Median :3.0   Median :4.3   Median :1.3   virginica :50
 Mean   :5.8   Mean   :3.1   Mean   :3.8   Mean   :1.2
 3rd Qu.:6.4   3rd Qu.:3.3   3rd Qu.:5.1   3rd Qu.:1.8
 Max.   :7.9   Max.   :4.4   Max.   :6.9   Max.   :2.5
```

- The plyr package (Wickham, 2011) is very useful to manipulate data frames (df)
- Let's take a look at the iris built-in data frame
  > summary(iris,digits = 2)

```
  Sepal.Length  Sepal.Width   Petal.Length  Petal.Width        Species
 Min.   :4.3   Min.   :2.0   Min.   :1.0   Min.   :0.1   setosa    :50
 1st Qu.:5.1   1st Qu.:2.8   1st Qu.:1.6   1st Qu.:0.3   versicolor:50
 Median :5.8   Median :3.0   Median :4.3   Median :1.3   virginica :50
 Mean   :5.8   Mean   :3.1   Mean   :3.8   Mean   :1.2
 3rd Qu.:6.4   3rd Qu.:3.3   3rd Qu.:5.1   3rd Qu.:1.8
 Max.   :7.9   Max.   :4.4   Max.   :6.9   Max.   :2.5
```

- For instance, the ddply is applied on df and returns df, hence the dd
- Whereas, the dlply is applied on df and returns a list, hence the dl

- The plyr package (Wickham, 2011) is very useful to manipulate data frames (df)
- Let's take a look at the iris built-in data frame

```
> summary(iris,digits = 2)
```

```
  Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species  
 Min.   :4.3   Min.   :2.0   Min.   :1.0   Min.   :0.1   setosa    :50  
 1st Qu.:5.1   1st Qu.:2.8   1st Qu.:1.6   1st Qu.:0.3   versicolor:50  
 Median :5.8   Median :3.0   Median :4.3   Median :1.3   virginica :50  
 Mean   :5.8   Mean   :3.1   Mean   :3.8   Mean   :1.2                  
 3rd Qu.:6.4   3rd Qu.:3.3   3rd Qu.:5.1   3rd Qu.:1.8                  
 Max.   :7.9   Max.   :4.4   Max.   :6.9   Max.   :2.5                  
```

- For instance, the ddply is applied on df and returns df, hence the dd
- Whereas, the dlply is applied on df and returns a list, hence the dl
- To see this, we can look at the average sepal length for species using the following command

```
> library(plyr)
> ddply(iris,"Species", function(ds)  mean(ds[,"Sepal.Length"]) )
```

```
     Species    V1
1     setosa 5.006
2 versicolor 5.936
3  virginica 6.588
```

- The plyr package (Wickham, 2011) is very useful to manipulate data frames (df)
- Let's take a look at the iris built-in data frame

```
> summary(iris,digits = 2)
```

```
  Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species
 Min.   :4.3   Min.   :2.0   Min.   :1.0   Min.   :0.1   setosa    :50
 1st Qu.:5.1   1st Qu.:2.8   1st Qu.:1.6   1st Qu.:0.3   versicolor:50
 Median :5.8   Median :3.0   Median :4.3   Median :1.3   virginica :50
 Mean   :5.8   Mean   :3.1   Mean   :3.8   Mean   :1.2
 3rd Qu.:6.4   3rd Qu.:3.3   3rd Qu.:5.1   3rd Qu.:1.8
 Max.   :7.9   Max.   :4.4   Max.   :6.9   Max.   :2.5
```

- For instance, the ddply is applied on df and returns df, hence the dd
- Whereas, the dlply is applied on df and returns a list, hence the dl
- To see this, we can look at the average sepal length for species using the following command

```
> library(plyr)
> ddply(iris,"Species", function(ds)  mean(ds[,"Sepal.Length"]) )
```

```
     Species    V1
1     setosa 5.006
2 versicolor 5.936
3  virginica 6.588
```

- Otherwise, one can split the data based on species and work on each data separately

```
> iris.list <- dlply(iris,"Species",data.frame)
> length(iris.list)
```

```
[1] 3
```

```
> sapply(iris.list, nrow)
```

```
    setosa versicolor  virginica
        50         50         50
```

Practice

Using the `iris` dataset, which species has the highest maximum petal length-to-width ratio?

Hints

- one can easily create a new variable called `XYZ` to dataset `ds` by assigning the corresponding values to a new called variable `ds[,"XYZ"] <- ....`
- use the `ddply` function

Plots

- The main plot function in R is called `plot`
- The function is powerful and can be ran on different objects, for instance
  - use it on a dataset to get a perspective
  - use it on a regression for diagnostics
  - use it directly on time series objects, e.g. `xts`

Plots

- The main plot function in R is called `plot`
- The function is powerful and can be ran on different objects, for instance
  - use it on a dataset to get a perspective
  - use it on a regression for diagnostics
  - use it directly on time series objects, e.g. `xts`
- For instance the plot can be ran on different objects, matrices, vectors, or data frames:

```
> y <- iris[,"Petal.Length"]
> x <- iris[,"Petal.Width"]
> plot(y~x)
```

- Alternatively, the same plot can be produced in one line command

```
> plot(Petal.Length ~ Petal.Width,data = iris)
```

Plots

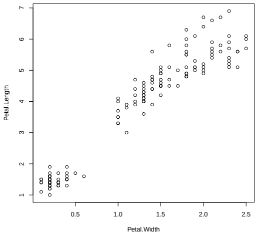
- The main plot function in R is called `plot`
- The function is powerful and can be ran on different objects, for instance
  - use it on a dataset to get a perspective
  - use it on a regression for diagnostics
  - use it directly on time series objects, e.g. `xts`
- For instance the plot can be ran on different objects, matrices, vectors, or data frames:

```
> y <- iris[,"Petal.Length"]
> x <- iris[,"Petal.Width"]
> plot(y~x)
```

- Alternatively, the same plot can be produced in one line command

```
> plot(Petal.Length ~ Petal.Width,data = iris)
```

- In either case, we get

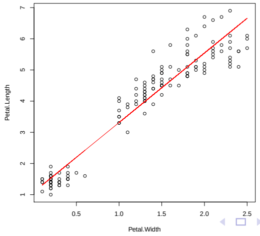


- In addition to `y` and `x`, one can specify a number of arguments
  1. `col`: setting color using an integer or character, e.g. `col = 2` or `col = "red"`
  2. `type`: type of plot. e.g. `type = "l"` returns a line
  3. `lwd`: width of line
  4. `lty`: type of line, e.g. dashed or solid
  5. `pch`: shape of dots
  6. `xlab`: x-axis labels
  7. `ylab`: y-axis labels
  8. `xlim`: range of x-axis (useful for multiple lines)
  9. `ylim`: range of y-axis (useful for multiple lines)

- In addition to `y` and `x`, one can specify a number of arguments
  1. `col`: setting color using an integer or character, e.g. `col = 2` or `col = "red"`
  2. `type`: type of plot. e.g. `type = "l"` returns a line
  3. `lwd`: width of line
  4. `lty`: type of line, e.g. dashed or solid
  5. `pch`: shape of dots
  6. `xlab`: x-axis labels
  7. `ylab`: y-axis labels
  8. `xlim`: range of x-axis (useful for multiple lines)
  9. `ylim`: range of y-axis (useful for multiple lines)
- After calling the plot, one can also add points, lines, or legend
  1. `lines`: adds line for a given `y` and `x`
  2. `points`: adds points for a given `y` and `x`
  3. `abline`: is useful to add horizontal, vertical, or linear lines

- In addition to y and x, one can specify a number of arguments
  1. col: setting color using an integer or character, e.g. col = 2 or col = "red"
  2. type: type of plot. e.g. type = "l" returns a line
  3. lwd: width of line
  4. lty: type of line, e.g. dashed or solid
  5. pch: shape of dots
  6. xlab: x-axis labels
  7. ylab: y-axis labels
  8. xlim: range of x-axis (useful for multiple lines)
  9. ylim: range of y-axis (useful for multiple lines)
- After calling the plot, one can also add points, lines, or legend
  1. lines: adds line for a given y and x
  2. points: adds points for a given y and x
  3. abline: is useful to add horizontal, vertical, or linear lines
- For instance, one can add a fitted linear regression line as follows

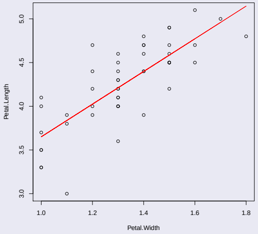
```
> iris[,"Petal.Length.hat"] <- fitted(lm(Petal.Length ~ Petal.Width,data = iris)
> lines(Petal.Length.hat~Petal.Width, data = iris, col = 2)
```



Practice

Write a function that plots the relationship between the petal length and width (in line with the figure from the previous slide), for a given species. The function should be named `lm.species` and should take a character as its main and only argument. For instance, the following command should yield

```
> lm.species("versicolor")
```



Hints:

1. recall the `iris.list`
2. each item in `iris.list` corresponds to a sub-dataset
3. in lists, one need to use double brackets to retrieve a certain item
   - `iris.list[[1]]` returns the first dataset, which is the setosa species
   - alternatively, one can achieve so by `iris.list[["setosa"]]`

Simulations

RNG

- The basic sampling method in R is the `sample` command
  - allowing user to randomly sample a subset from a given set

```
> sample(1:10,5)
[1] 10  8  5  9  2
> sample(1:10,5)
[1] 9 1 3 6 4
```

RNG

- The basic sampling method in R is the `sample` command
    - allowing user to randomly sample a subset from a given set

```
> sample(1:10,5)

[1] 10  8  5  9  2

> sample(1:10,5)

[1] 9 1 3 6 4
```

- One needs to specify whether the sampling is done with replacement

```
> sample(1:10,20)

Error in sample.int(length(x), size, replace, prob) :
  cannot take a sample larger than the population when 'replace = FALSE'

> sample(1:10,20,replace = T)

 [1]  3  7  9  9  9  8  8  6  1  9  6  5 10  8  7  9  8 10  3  3
```

RNG

- The basic sampling method in R is the `sample` command
    - allowing user to randomly sample a subset from a given set

```
> sample(1:10,5)

[1] 10  8  5  9  2

> sample(1:10,5)

[1] 9 1 3 6 4
```

- One needs to specify whether the sampling is done with replacement

```
> sample(1:10,20)

Error in sample.int(length(x), size, replace, prob) :
  cannot take a sample larger than the population when 'replace = FALSE'

> sample(1:10,20,replace = T)

 [1]  3  7  9  9  9  8  8  6  1  9  6  5 10  8  7  9  8 10  3  3
```

- To set a seed, one can use `set.seed` function

```
> sapply(1:5, function(i) { set.seed(13); sample(1:10,5)    }     )

     [,1] [,2] [,3] [,4] [,5]
[1,]    8    8    8    8    8
[2,]    3    3    3    3    3
[3,]    4    4    4    4    4
[4,]    1    1    1    1    1
[5,]    6    6    6    6    6
```

Distributions

- Most distributions in R follow similar commands
- Assume we are interested in the normal distribution, then
  1. `dnorm` is the probability density function
  2. `pnorm` is the cumulative distribution function
  3. `qnorm` is the quantile for a given normal
  4. `rnorm` is a RNG for a given `n` sample

Distributions

- Most distributions in R follow similar commands
- Assume we are interested in the normal distribution, then
    1. `dnorm` is the probability density function
    2. `pnorm` is the cumulative distribution function
    3. `qnorm` is the quantile for a given normal
    4. `rnorm` is a RNG for a given `n` sample

- In fact, our focus will be mainly on the RNG functions for normal distribution
- By default, the `rnorm` is a standard normal, i.e.

```
> rnorm
  function (n, mean = 0, sd = 1)
```

Distributions

- Most distributions in R follow similar commands
- Assume we are interested in the normal distribution, then
  1. `dnorm` is the probability density function
  2. `pnorm` is the cumulative distribution function
  3. `qnorm` is the quantile for a given normal
  4. `rnorm` is a RNG for a given `n` sample

- In fact, our focus will be mainly on the RNG functions for normal distribution
- By default, the `rnorm` is a standard normal, i.e.

```
> rnorm
  function (n, mean = 0, sd = 1)
```

- Let $X_i \sim N(i, 1)$ iid, for $i = 1, .., 5$,
- To simulate $n = 10^3$ observations for each variable, one can run

```
> norm.list <- lapply(1:5, function(i) rnorm(10^3,mean = i, sd = 1)  )
> sapply(norm.list, mean)
[1] 1.039613 2.012433 3.068028 4.015763 4.812596
```

Practice

- Let X_1 and X_2 be two independent random Gaussian variables, with $\mu_1 = 10$, $\mu_2 = 15$, $\sigma_1 = 3$, and $\sigma_2 = 5$, i.e.

$$X_1 \sim N(10, 3^2) \tag{1}$$

and

$$X_2 \sim N(15, 5^2) \tag{2}$$

- Using a simulation, find the probability that

$$\mathbb{P}(X_1 < X_2) \tag{3}$$

- In addition, compare your result with the exact solution.
  - **Hint**: since X_1 and X_2 are independent, then

$$X_1 - X_2 \sim N(\mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2) \tag{4}$$

Multivariate Normal

- We know when $X_i \sim N(\mu_i, \sigma_i^2)$, it follows that

$$X = \begin{bmatrix} X_1 \\ \vdots \\ X_d \end{bmatrix} \sim N_d\left(\mu, \Sigma\right) \tag{5}$$

with

- μ denoting a $d \times 1$ vector of mean returns and
- Σ is a non-singular $d \times d$ covariance matrix

Multivariate Normal

- We know when $X_i \sim N(\mu_i, \sigma_i^2)$, it follows that

$$X = \begin{bmatrix} X_1 \\ \vdots \\ X_d \end{bmatrix} \sim N_d\left(\mu, \Sigma\right) \tag{5}$$

  with
  - μ denoting a $d \times 1$ vector of mean returns and
  - Σ is a non-singular $d \times d$ covariance matrix

- One can simulate the vector X using the `mvrnorm` function from the `MASS` package (Venables & Ripley, 2002)
- Nevertheless, one needs to specify a vector of means, μ, and a covariance matrix, Σ

Multivariate Normal

- We know when $X_i \sim N(\mu_i, \sigma_i^2)$, it follows that

$$X = \begin{bmatrix} X_1 \\ \vdots \\ X_d \end{bmatrix} \sim N_d\left(\mu, \Sigma\right) \tag{5}$$

  with
  - μ denoting a $d \times 1$ vector of mean returns and
  - Σ is a non-singular $d \times d$ covariance matrix

- One can simulate the vector X using the `mvrnorm` function from the `MASS` package (Venables & Ripley, 2002)
- Nevertheless, one needs to specify a vector of means, μ, and a covariance matrix, Σ

- Let's get back to the `EuStockMarkets` dataset

```
> ds <- EuStockMarkets
> R <- (ds[-1,]/ds[-nrow(ds),] - 1)*100 # computes returns
> Mu <- apply(R, 2, mean) # mean vector
> Sigma <- var(R) # covariance matrix
```

- For a given μ and Σ, one can generate n vectors of X

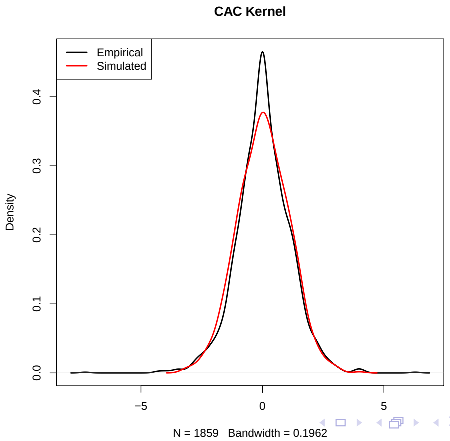
```
> n <- 10^3
> X <- mvrnorm(N,Mu,Sigma)
> dim(X)

[1] 1000    4
```

- Let's compare between the simulated distribution and the empirical one
- The `density` function allows the user to estimate the kernel density of variable x

- Let's compare between the simulated distribution and the empirical one
- The `density` function allows the user to estimate the kernel density of variable x

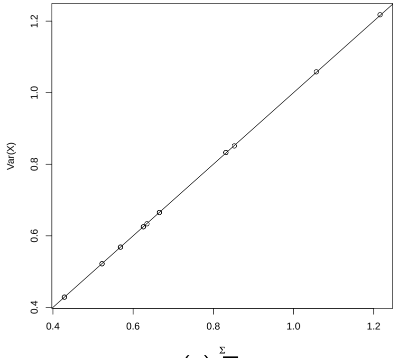
- Looking at the CAC index, we have

```
> kernel_emp <- density(R[,"CAC"])
> kernel_sim <- density(X[,"CAC"])
> plot(kernel_emp, main  = "CAC Kernel", lwd = 2)
> lines(kernel_sim, col = 2, lwd = 2)
> legend("topleft", c("Empirical","Simulated"), col = 1:2,lwd = 2)
```

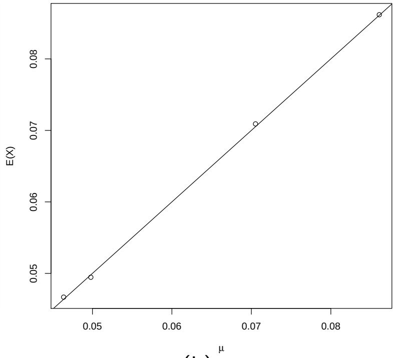


- Also, note since X is generated using μ and Σ, the mean and covariance of the simulated data should be consistent

```
> n <- 10^6
> X <- mvrnorm(n,Mu,Sigma)
> plot(var(X) ~ Sigma, xlab = expression(Sigma), ylab = "Var(X)")
> abline(a = 0, b = 1)
> plot(apply(X,2,mean) ~ Mu, xlab = expression(mu), ylab = "E(X)")
> abline(a = 0, b = 1)
```



(a) Σ



(b) μ

Practice

- Consider a portfolio strategy that equally allocates among the 4 indices from `EuStock-Markets`
  - assume daily re-balancing
- Simulate the portfolio return over 250 days
- Repeat the above step 1000 times and draw the density of the portfolio Sharpe-ratio (SR)
- Make some observations about the performance

Practice

- Consider a portfolio strategy that equally allocates among the 4 indices from `EuStock-Markets`
  - assume daily re-balancing
- Simulate the portfolio return over 250 days
- Repeat the above step 1000 times and draw the density of the portfolio Sharpe-ratio (SR)
- Make some observations about the performance

Hint

Create a function `SR_f` that simulates the portfolio return over 250 days and returns. Given this, compute the SR using the average to standard deviation ratio.

```
> SR_f <- function(i) {
+   R.250 <-  mvrnorm(250,Mu,Sigma)
+   Rp <- R.250%*%W
+   SR <- (mean(Rp)/sd(Rp))
+   return(SR)
+   }
```

Practice

- Consider a portfolio strategy that equally allocates among the 4 indices from `EuStockMarkets`
  - assume daily re-balancing
- Simulate the portfolio return over 250 days
- Repeat the above step 1000 times and draw the density of the portfolio Sharpe-ratio (SR)
- Make some observations about the performance

Hint

Create a function `SR_f` that simulates the portfolio return over 250 days and returns. Given this, compute the SR using the average to standard deviation ratio.

```
> SR_f <- function(i) {
+   R.250 <-  mvrnorm(250,Mu,Sigma)
+   Rp <- R.250%*%W
+   SR <- (mean(Rp)/sd(Rp))
+   return(SR)
+   }
```

Note

If W denotes a $d \times 1$ vector of portfolio weights while R is a $n \times d$ denoting the returns of d assets over n periods, then the portfolio return is given by a $n \times 1$ vector R_p, such that

$$R_p = RW \tag{6}$$

For matrix multiplication, use the %*% operator

Cholesky Decomposition

- For the sake of argument, assume we cannot use the `mvrnorm` function
- How can we, then, simulate a multivariate Gaussian data?
- One answer is the Cholesky Decomposition (CD)

- The idea behind CD is the following
  1. Simulate d univariate standard normally distributed random variables and stack it in a $n \times d$ matrix called Z
  2. For a given Σ, use the CD to find matrix A, such that

$$\Sigma = AA' \tag{7}$$

  3. Finally, map the Z matrix with respect to A and μ

$$X = M + ZA \tag{8}$$

  where M is a $n \times d$ matrix with each row is equal to μ'

CD in Action

- Let's consider again the `EuStockMarkets` data
- Since we have 4 indices, then $d = 4$
- Simulate 4 standard normal random variables over $n = 10^6$ sample, named Z
- Since we know Σ, use `chol` function to find A
- Finally, map Z using μ and A

```
> n <- 10^6
> A <- chol(Sigma)
> M <- matrix(Mu,n,length(Mu),byrow = T)
> Z <- sapply( 1:length(Mu),function(i) rnorm(n) )
> X_CD <- M + Z%*%A
```

CD in Action

- Let's consider again the EuStockMarkets data
- Since we have 4 indices, then $d = 4$
- Simulate 4 standard normal random variables over $n = 10^6$ sample, named Z
- Since we know Σ, use chol function to find A
- Finally, map Z using μ and A

```
> n <- 10^6
> A <- chol(Sigma)
> M <- matrix(Mu,n,length(Mu),byrow = T)
> Z <- sapply( 1:length(Mu),function(i) rnorm(n) )
> X_CD <- M + Z%*%A
```

Reflect

- Compare between X_CD and X
- Recall that X was produced using the mvrnorm function
- Both should yield similar results
  - in terms of moments and kernel

Option Pricing

Simulating Stock Prices

- If the stock price obeys to the following dynamics

$$\frac{dS_t}{S_t} = \mu t + \sigma dW_t \tag{9}$$

then for an initial price S_0, the solution is given by

$$\log\left(\frac{S_t}{S_0}\right) = \left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t \tag{10}$$

with

$$W_t \sim N(0, t) \tag{11}$$

Simulating Stock Prices

- If the stock price obeys to the following dynamics

$$\frac{dS_t}{S_t} = \mu t + \sigma dW_t \tag{9}$$

then for an initial price S_0, the solution is given by

$$\log\left(\frac{S_t}{S_0}\right) = \left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t \tag{10}$$

with

$$W_t \sim N(0, t) \tag{11}$$

- The log of the ratio S_t/S_0 denotes the change in price over time t, i.e. return

Simulating Stock Prices

- If the stock price obeys to the following dynamics

$$\frac{dS_t}{S_t} = \mu t + \sigma dW_t \tag{9}$$

then for an initial price S_0, the solution is given by

$$\log\left(\frac{S_t}{S_0}\right) = \left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t \tag{10}$$

with

$$W_t \sim N(0, t) \tag{11}$$

- The log of the ratio S_t/S_0 denotes the change in price over time t, i.e. return
- Let r_t denote the return on the stock price over t, such that

$$r_t \sim N\left(t\left(\mu - \frac{\sigma^2}{2}\right), t\sigma^2\right) \tag{12}$$

Simulating Stock Prices

- If the stock price obeys to the following dynamics

$$\frac{dS_t}{S_t} = \mu t + \sigma dW_t \tag{9}$$

then for an initial price S_0, the solution is given by

$$\log\left(\frac{S_t}{S_0}\right) = \left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t \tag{10}$$

with

$$W_t \sim N(0, t) \tag{11}$$

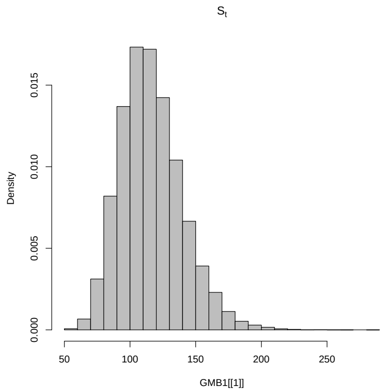
- The log of the ratio S_t/S_0 denotes the change in price over time t, i.e. return
- Let r_t denote the return on the stock price over t, such that

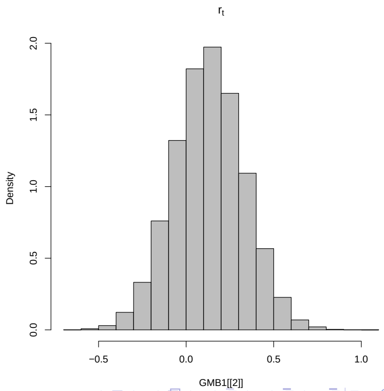
$$r_t \sim N\left(t\left(\mu - \frac{\sigma^2}{2}\right), t\sigma^2\right) \tag{12}$$

- Since we worked with normal distributions, we know how to simulate W_t
- From Equation (10), if we know how to simulate W_t, then we can simulate either r_t or S_t

- The case for the univariate case should be straightforward

```
> GMB_f <- function(n,S0,Mu,Sigma,Time) {
+   EX <- Time*(Mu - (Sigma^2)/2)
+   VX <- Time*(Sigma^2)
+   r_t <- rnorm(n,EX,sqrt(VX))
+   S_t <- S0*exp(r_t)
+   list(S_t,r_t)
+   }
> GMB1 <- GMB_f(10^5,100,0.15,0.2,1)
> hist(GMB1[[1]], main = expression(S[t]), col = "gray", freq = F)
> hist(GMB1[[2]], main = expression(r[t]), col = "gray", freq = F)
```





Simulating Multiple Stock Prices

- Simulating multiple stock prices follow suit with the procedure from (5)
- Equation (12) takes the following multivariate form

$$r_t = \begin{bmatrix} r_{t,1} \\ \vdots \\ r_{t,d} \end{bmatrix} \sim N_d\left(t\boldsymbol{\mu}, t\Sigma\right) \tag{13}$$

- Note that the ith of the mean vector μ_t is given by

$$\boldsymbol{\mu}_i = \left(\mu_i - \frac{\sigma_i^2}{2}\right), \forall i \in \{1, .., d\} \tag{14}$$

and the ith row and jth column of Σ_t is

$$\Sigma_{t,ij} = \rho_{ij}\sigma_i\sigma_j \tag{15}$$

where ρ_{ij} is the correlation coefficient between i and j, $\forall i, j \in \{1, ..., d\}$ and $\rho_{ij} = 1$ $\forall i = j$.

Ultimately, if we know $\boldsymbol{\mu}$ and Σ, we can simulate terminal prices for d stocks

- Let's consider the following case with three stocks
  - $S_{0,1} = S_{0,2} = S_{0,3} = 100$
  - $\sigma_1 = 0.2$, $\sigma_2 = 0.3$, and $\sigma_3 = 0.25$
  - For $i = 1, 2, 3$, we have $\mu_i = 0.06$
  - $\rho_{12} = 0.5$, $\rho_{13} = 0.25$, and $\rho_{23} = -0.25$

Practice

- Simulate the prices of the three stocks for $t = 1$ year
- Price the following spread option between stock 1 and 2 (i.e. Q9 from HW2)

- Let's consider the following case with three stocks
  - $S_{0,1} = S_{0,2} = S_{0,3} = 100$
  - $\sigma_1 = 0.2$, $\sigma_2 = 0.3$, and $\sigma_3 = 0.25$
  - For $i = 1, 2, 3$, we have $\mu_i = 0.06$
  - $\rho_{12} = 0.5$, $\rho_{13} = 0.25$, and $\rho_{23} = -0.25$

Practice

- Simulate the prices of the three stocks for $t = 1$ year
- Price the following spread option between stock 1 and 2 (i.e. Q9 from HW2)

Hint

Recall that the covariance matrix can decomposed as

$$\Sigma = \mathbf{DRD} \qquad (16)$$

where

- $\mathbf{D}$ is the diagonal matrix of standard deviations, i.e. $\mathbf{D}_{ii} = \sigma_i \; \forall i = 1, .., d$ and $\mathbf{D}_{ij} = 0$ $\forall i \neq j$
- $\mathbf{R}$ is the correlation matrix, where $\mathbf{R}_{ij} = \rho_{ij}$ and $\mathbf{R}_{ii} = 1 \; \forall i = 1, .., d$

- The main issue is identifying the parameters from (13)
- If we can, then simulating the prices is straightforward
  - with or without the `mvrnorm` function

- The main issue is identifying the parameters from (13)
- If we can, then simulating the prices is straightforward
  - with or without the `mvrnorm` function

```
> Time <- 1
> r <- 0.06
> K <- 1
> Mu <- rep(r,3)
> D <- diag(c(0.2,0.3,0.5)) # diagonal of sigmas
> R <- rbind(c(1,0.5,0.25), c(0.5,1,-0.25), c(0.25,-0.25,1))
> Sigma <- D%*%R%*%D
> S <- rep(100,3)
> n <- 10^4
> GMB_MV_f <- function(n,S,Mu,Sigma,Time) {
+   EX <- Time*(Mu - (diag(D)^2)/2)
+   VX <- Time*(Sigma)
+   r_t <- mvrnorm(n,EX,VX)
+   S_t <- sapply(1:ncol(r_t), function(i) exp(r_t[,i])*S[i])
+   list(S_t,r_t)
+ }
> GMB2 <- GMB_MV_f(n,S,Mu,Sigma,Time)
> S_T <- GMB2[[1]]
> Spread <- S_T[,1] - S_T[,2] - K
> Spread[Spread < 0] <- 0
> mean(Spread)*exp(-r*Time)

[1] 10.06153
```

Open Source Data

References I

[]Ang, C. S. 2015. *Analyzing financial data and implementing financial models using r*. Springer.

[]James, G., Witten, D., Hastie, T., & Tibshirani, R. 2014. An introduction to statistical learning: with applications in r.

[]Venables, W. N., & Ripley, B. D. 2002. *Modern applied statistics with s* (Fourth ed.). New York: Springer. Retrieved from `http://www.stats.ox.ac.uk/pub/MASS4` (ISBN 0-387-95457-0)

[]Wickham, H. 2011. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, *40*(1), 1–29. Retrieved from `http://www.jstatsoft.org/v40/i01/`