FRE6871 R in Finance

Lecture#3, Spring 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 8, 2024

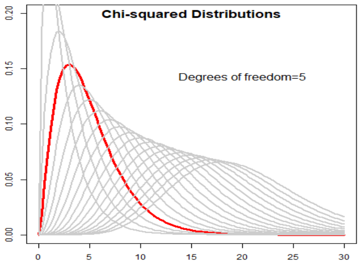
Plotting Using Expression Objects

It's sometimes convenient to create an *expression* object containing plotting commands, to be able to later create plots using it.

The function quote() produces an *expression* object without evaluating it.

The function eval() evaluates an *expression* in a specified *environment*.

```
> # Create a plotting expression
> expv <- quote({
+   degf <- 2:20
+   rangev <- (1:NROW(degf))
+   indeks <- 4
+   # Plot a curve
+   curve(expr=dchisq(x, df=degf[indeks]),
+ xlim=c(0, 30), ylim=c(0, 0.2),
+ xlab="", ylab="", lwd=3, col="red")
+   # Add grey lines to plot
+   for (it in rangev[-indeks]) {
+     curve(expr=dchisq(x, df=degf[it]),
+   xlim=c(0, 30), ylim=c(0, 0.2),
+   xlab="", ylab="", lwd=2, col="grey80", add=TRUE)
+   }  # end for
+   # Add title
+   title(main="Chi-squared Distributions", line=-1.5, cex.main=1.5)
+   # Add legend
+   text(x=20, y=0.15, labels=paste0("Degrees of freedom=",
+       degf[indeks]), pos=1, cex=1.3)
+ })  # end quote
```



```
> # View the plotting expression
> expv
> # Create plot by evaluating the plotting expression
> x11(width=6, height=4)
> eval(expv)
```

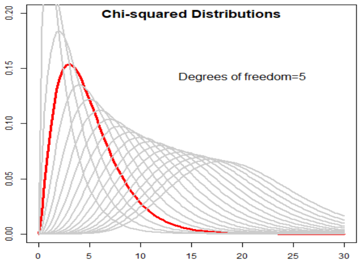
Animated Plots Using Package *animation*

The package *animation* allows creating animated plots in the form of *gif* and *html* documents.

The function saveGIF() produces a *gif* image with an animated plot.

The function saveHTML() produces an *html* document with an animated plot.

```
> library(animation)
> # Create an expression for creating multiple plots
> expv <- quote({
+   degf <- 2:20
+   rangev <- (1:NROW(degf))
+   # Set image refesh interval
+   animation::ani.options(interval=0.5)
+   # Create multiple plots with curves
+   for (indeks in rangev) {
+     curve(expr=dchisq(x, df=degf[indeks]),
+   xlim=c(0, 30), ylim=c(0, 0.2),
+   xlab="", ylab="", lwd=3, col="red")
+     # Add grey lines to plot
+     for (it in rangev[-indeks]) {
+       curve(expr=dchisq(x, df=degf[it]),
+     xlim=c(0, 30), ylim=c(0, 0.2),
+     xlab="", ylab="", lwd=2, col="grey80", add=TRUE)
+     }  # end for
+     # Add title
+     title(main="Chi-squared Distributions", line=-1.5, cex.main=
+     # Add legend
+     text(x=20, y=0.15, labels=paste0("Degrees of freedom=",
+       degf[indeks]), pos=1, cex=1.3)
+   }  # end for
+ })  # end quote
```



```
> # Create plot by evaluating the plotting expression
> x11(width=6, height=4)
> eval(expv)
> # Create gif with animated plot
> animation::saveGIF(expr=eval(expv),
+   movie.name="chi_squared.gif",
+   img.name="chi_squared")
> # Create html with animated plot
> animation::saveHTML(expr=eval(expv),
+   img.name="chi_squared",
+   htmlfile="chi_squared.html",
+   description="Chi-squared Distributions")  # end saveHTML
```

Dynamic Documents Using *R markdown*

markdown is a simple markup language designed for creating documents in different formats, including *pdf* and *html*.

R Markdown is a modified version of *markdown*, which allows creating documents containing *math formulas* and R code embedded in them.

An *R Markdown* document (with extension .Rmd) contains:

- A *YAML* header,
- Text in *R Markdown* code format,
- Math formulas (equations), delimited using either single "$" symbols (for inline formulas), or double "$$" symbols (for display formulas),
- R code chunks, delimited using either single "`" backtick symbols (for inline code), or triple "```" backtick symbols (for display code).

The packages *rmarkdown* and *knitr* compile R documents into either *pdf*, *html*, or *MS Word* documents.

~~~~
---
title: "My First R Markdown Document"
author: Jerzy Pawlowski
date: '`r format(Sys.time(), "%m/%d/%Y")`'
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)

# install package quantmod if it can't be loaded success
if (!require("quantmod"))
  install.packages("quantmod")
```

### R Markdown
This is an *R Markdown* document. Markdown is a simple f

One of the advantages of writing documents *R Markdown*

You can read more about publishing documents using *R* h
https://algoquant.github.io/r,/markdown/2016/07/02/Publi

You can read more about using *R* to create *HTML* docum
https://algoquant.github.io/2016/07/05/Interactive-Plots

Clicking the **Knit** button in *RStudio*, compiles the

Example of an *R* code chunk:
```{r cars}
summary(cars)
```

### Plots in *R Markdown* documents

Plots can also be embedded, for example:
```{r pressure, echo=FALSE}
plot(pressure)
~~~~

Package *shiny* for Creating Interactive Applications

The package *shiny* creates interactive applications running in R, with their outputs presented as live visualizations.

Shiny allows changing the model parameters, recalculating the model, and displaying the resulting outputs as plots and charts.

A *shiny app* is a file with *shiny* commands and R code.

The *shiny* code consists of a *shiny interface* and a *shiny server*.

The *shiny interface* contains widgets for data input and an area for plotting.

The *shiny server* contains the R model code and the plotting code.

The function `shiny::fluidPage()` creates a GUI layout for the user inputs of model parameters and an area for plots and charts.

The function `shiny::renderPlot()` renders a plot from the outputs of a live model.

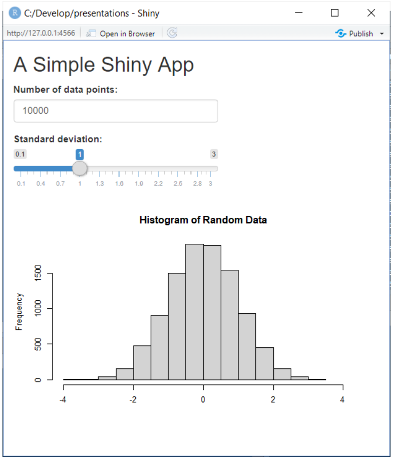
The function `shiny::shinyApp()` creates a shiny app from a *shiny interface* and a *shiny server*.

```
> ## App setup code that runs only once at startup.
> ndata <- 1e4
> stdev <- 1.0
>
> ## Define the user interface
> uiface <- shiny::fluidPage(
+   # Create numeric input for the number of data points.
+   numericInput("ndata", "Number of data points:", value=ndata),
+   # Create slider input for the standard deviation parameter.
+   sliderInput("stdev", label="Standard deviation:",
+         min=0.1, max=3.0, value=stdev, step=0.1),
+   # Render plot in a panel.
+   plotOutput("plotobj", height=300, width=500)
+ )  # end user interface
>
> ## Define the server function
> servfun <- function(input, output) {
+   output$plotobj <- shiny::renderPlot({
+     # Simulate the data
+     datav <- rnorm(input$ndata, sd=input$stdev)
+     # Plot the data
+     par(mar=c(2, 4, 4, 0), oma=c(0, 0, 0, 0))
+     hist(datav, xlim=c(-4, 4), main="Histogram of Random Data")
+   })  # end renderPlot
+ }  # end servfun
>
> # Return a Shiny app object
> shiny::shinyApp(ui=uiface, server=servfun)
```

Running Shiny Apps in *RStudio*

A *shiny app* can be run by pressing the "Run App" button in *RStudio*.

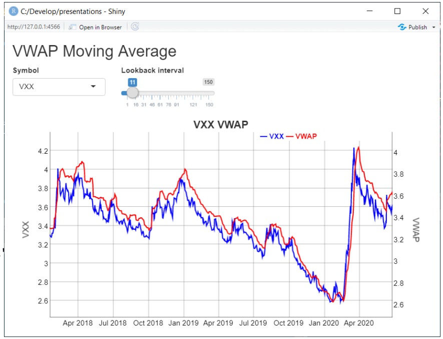
When the *shiny app* is run, the *shiny* commands are translated into *JavaScript* code, which creates a graphical user interface (GUI) with buttons, sliders, and boxes for data input, and also with the output plots and charts.



Positioning and Sizing Widgets Within the Shiny GUI

The functions `shiny::fluidRow()` and `shiny::column()` allow positioning and sizing widgets within the *shiny* GUI.

```
> ## Create elements of the user interface
> uiface <- shiny::fluidPage(
+   titlePanel("VWAP Moving Average"),
+   # Create single row of widgets with two slider inputs
+   fluidRow(
+     # Input stock symbol
+     column(width=3, selectInput("symbol", label="Symbol",
+                            choices=symbolv, selected=symbol)),
+     # Input look-back interval
+     column(width=3, sliderInput("lookb", label="Lookback interval"
+                            min=1, max=150, value=11, step=1))
+   ),  # end fluidRow
+   # Create output plot panel
+   mainPanel(dygraphs::dygraphOutput("dyplot"), width=12)
+ )  # end fluidPage interface
```



Shiny Apps With Reactive Expressions

The package *shiny* allows specifying reactive expressions which are evaluated only when their input data is updated.

Reactive expressions avoid performing unnecessary calculations.

If the reactive expression is invalidated (recalculated), then other expressions that depend on its output are also recalculated.

This way calculations cascade through the expressions that depend on each other.

The function `shiny::reactive()` transforms an expression into a reactive expression.

```
> ## Define the server function
> servfun <- shiny::shinyServer(function(input, output) {
+   # Get the close and volume data in a reactive environment
+   closep <- shiny::reactive({
+     # Get the data
+     ohlc <- get(input$symbol, data_env)
+     closep <- log(quantmod::Cl(ohlc))
+     volum <- quantmod::Vo(ohlc)
+     # Return the data
+     cbind(closep, volum)
+   })  # end reactive code
+
+   # Calculate the VWAP indicator in a reactive environment
+   vwapv <- shiny::reactive({
+     # Get model parameters from input argument
+     lookb <- input$lookb
+     # Calculate the VWAP indicator
+     closep <- closep()[, 1]
+     volum <- closep()[, 2]
+     vwapv <- HighFreq::roll_sum(tseries=closep*volum, lookb=lookb)
+     volumroll <- HighFreq::roll_sum(tseries=volum, lookb=lookb)
+     vwapv <- vwapv/volumroll
+     vwapv[is.na(vwapv)] <- 0
+     # Return the plot data
+     datav <- cbind(closep, vwapv)
+     colnames(datav) <- c(input$symbol, "VWAP")
+     datav
+   })  # end reactive code
+
+   # Return the dygraph plot to output argument
+   output$dyplot <- dygraphs::renderDygraph({
+     colnamev <- colnames(vwapv())
+     dygraphs::dygraph(vwapv(), main=paste(colnamev[1], "VWAP")) %>
+ dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+ dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+ dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWidt
+ dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeWic
+   })  # end output plot
+ })  # end server code
```

Reactive Event Handlers

Event handlers are functions which evaluate expressions when an event occurs (like a button press).

The functions `shiny::observeEvent()` and `shiny::eventReactive()` are event handlers.

The function `shiny::eventReactive()` returns a value, while `shiny::observeEvent()` produces a side-effect, without returning a value.

The function `shiny::reactiveValues()` creates a list for storing reactive values, which can be updated by event handlers.

```
> ## Define the server function
> servfun <- shiny::shinyServer(function(input, output) {
+
+   # Create an empty list of reactive values.
+   value_s <- reactiveValues()
+
+   # Get input parameters from the user interface.
+   nrows <- reactive({
+     # Add nrows to list of reactive values.
+     value_s*nrows <- input$nrows
+     input$nrows
+   })  # end reactive code
+
+   # Broadcast a message to the console when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     cat("Input button pressed\n")
+   })  # end observeEvent
+
+   # Send the data when the button is pressed.
+   datav <- eventReactive(eventExpr=input$button, valueExpr={
+     # eventReactive() executes on input$button, but not on nrows(
+     cat("Sending", nrows(), "rows of data\n")
+     datav <- head(mtcars, input$nrows)
+     value_s$mpg <- mean(datav$mpg)
+     datav
+   })  # end eventReactive
+   #   datav
+
+   # Draw table of the data when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     datav <- datav()
+     cat("Received", value_s*nrows, "rows of data\n")
+     cat("Average mpg = ", value_s$mpg, "\n")
+     cat("Drawing table\n")
+     output$tablev <- renderTable(datav)
+   })  # end observeEvent
+
+ })  # end server code
>
```

Vector and Matrix Calculus

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

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

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

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

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

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

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

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

$$\begin{aligned}\frac{d(\mathbf{v}^T\mathbb{1})}{d\mathbf{v}} = d_v[\mathbf{v}^T\mathbb{1}] = d_v[\mathbb{1}^T\mathbf{v}] &= \mathbb{1}^T \\ d_v[\mathbf{v}^T\mathbf{w}] = d_v[\mathbf{w}^T\mathbf{v}] &= \mathbf{w}^T \\ d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{w}] &= \mathbf{w}^T\mathbb{A}^T \\ d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{v}] &= \mathbf{v}^T\mathbb{A} + \mathbf{v}^T\mathbb{A}^T\end{aligned}$$

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.

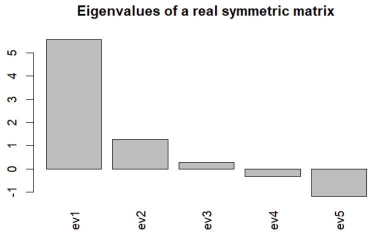
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

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 a random real symmetric matrix
> matv <- matrix(runif(25), nc=5)
> matv <- matv + t(matv)
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(matv)
> eigenvec <- eigend$vectors
> dim(eigenvec)
> # Plot eigenvalues
> barplot(eigend$values, xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(eigend$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:

$$\Sigma = \mathbb{O}^T \mathbb{A} \, \mathbb{O}$$

Where Σ is a *diagonal* matrix containing the *eigenvalues* of matrix $\mathbb{A}$, and $\mathbb{O}$ is an *orthogonal* matrix of its *eigenvectors*, with $\mathbb{O}^T \mathbb{O} = \mathbb{1}$.

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

$$\mathbb{A} = \mathbb{O} \, \Sigma \, \mathbb{O}^T$$

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(eigenvec) %*% eigenvec, digits=4)
> # Diagonalize matrix using eigenvector matrix
> round(t(eigenvec) %*% (matv %*% eigenvec), digits=4)
> eigend$values
> # Eigen decomposition of matrix by rotating the diagonal matrix
> matrixe <- eigenvec %*% (eigend$values * t(eigenvec))
> # Create diagonal matrix of eigenvalues
> # diagmat <- diag(eigend$values)
> # matrixe <- eigenvec %*% (diagmat %*% t(eigenvec))
> all.equal(matv, matrixe)
```

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

The *diagonal* matrix Σ 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 that each pair produces a single number.

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).

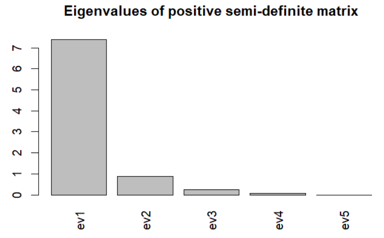
An example of *positive definite* matrices are the covariance matrices of linearly independent variables.

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

Singular Value Decomposition (*SVD*) of Matrices

The *Singular Value Decomposition* (*SVD*) is a generalization of the *eigen decomposition* of square matrices.

The *SVD* of a rectangular matrix $\mathbb{A}$ is defined as the factorization:

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

Where $\mathbb{U}$ and $\mathbb{V}$ are the left and right *singular matrices*, and Σ is a diagonal matrix of *singular values*.

If $\mathbb{A}$ has `m` rows and `n` columns and if (`m > n`), then $\mathbb{U}$ is an (`m x n`) *rectangular* matrix, Σ is an (`n x n`) *diagonal* matrix, and $\mathbb{V}$ is an (`n x n`) *orthogonal* matrix, and if (`m < n`) then the dimensions are: (`m x m`), (`m x m`), and (`m x n`).

The left $\mathbb{U}$ and right $\mathbb{V}$ singular matrices consist of columns of *orthonormal* vectors, so that $\mathbb{U}^T \mathbb{U} = \mathbb{V}^T \mathbb{V} = \mathbb{1}$.

In the special case when $\mathbb{A}$ is a square matrix, then $\mathbb{U} = \mathbb{V}$, and the *SVD* reduces to the *eigen decomposition*.

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.

```
> # Perform singular value decomposition
> matv <- matrix(rnorm(50), nc=5)
> svdec <- svd(matv)
> # Recompose matv from SVD mat_rices
> all.equal(matv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Columns of U and V are orthonormal
> round(t(svdec$u) %*% svdec$u, 4)
> round(t(svdec$v) %*% svdec$v, 4)
```

The Left and Right Singular Matrices

The left $\mathbb{U}$ and right $\mathbb{V}$ singular matrices define rotation transformations into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

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

The columns of $\mathbb{U}$ and $\mathbb{V}$ are called the *singular* vectors, and they are only defined up to a reflection (change in sign), i.e. if vec is a singular vector, then so is -vec.

The left singular matrix $\mathbb{U}$ forms the *eigenvectors* of the matrix $\mathbb{A}\mathbb{A}^T$.

The right singular matrix $\mathbb{V}$ forms the *eigenvectors* of the matrix $\mathbb{A}^T\mathbb{A}$.

```
> # Dimensions of left and right matrices
> nrows <- 6 ; ncols <- 4
> # Calculate the left matrix
> leftmat <- matrix(runif(nrows^2), nc=nrows)
> eigend <- eigen(crossprod(leftmat))
> leftmat <- eigend$vectors[, 1:ncols]
> # Calculate the right matrix and singular values
> rightmat <- matrix(runif(ncols^2), nc=ncols)
> eigend <- eigen(crossprod(rightmat))
> rightmat <- eigend$vectors
> singval <- sort(runif(ncols, min=1, max=5), decreasing=TRUE)
> # Compose rectangular matrix
> matv <- leftmat %*% (singval * t(rightmat))
> # Perform singular value decomposition
> svdec <- svd(matv)
> # Recompose matv from SVD
> all.equal(matv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Compare SVD with matv components
> all.equal(abs(svdec$u), abs(leftmat))
> all.equal(abs(svdec$v), abs(rightmat))
> all.equal(svdec$d, singval)
> # Eigen decomposition of matv squared
> retsq <- matv %*% t(matv)
> eigend <- eigen(retsq)
> all.equal(eigend$values[1:ncols], singval^2)
> all.equal(abs(eigend$vectors[, 1:ncols]), abs(leftmat))
> # Eigen decomposition of matv squared
> retsq <- t(matv) %*% matv
> eigend <- eigen(retsq)
> all.equal(eigend$values, singval^2)
> all.equal(abs(eigend$vectors), abs(rightmat))
```

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

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

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

The inverse of *non-symmetric* matrices can be calculated using *Singular Value Decomposition* (*SVD*).

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

```
> # Create a random positive semi-definite matrix
> matv <- matrix(runif(25), nc=5)
> matv <- t(matv) %*% matv
> # Calculate the inverse of matv
> invmat <- solve(a=matv)
> # Multiply inverse with matrix
> round(invmat %*% matv, 4)
> round(matv %*% invmat, 4)
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(matv)
> eigenvec <- eigend$vectors
> # Calculate the inverse from eigen decomposition
> inveigen <- eigenvec %*% (t(eigenvec) / eigend$values)
> all.equal(invmat, inveigen)
> # Decompose diagonal matrix with inverse of eigenvalues
> # diagmat <- diag(1/eigend$values)
> # inveigen <- eigenvec %*% (diagmat %*% t(eigenvec))
```

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 $\mathbb{V}$):

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

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.

```
> # Random rectangular matrix: nrows > ncols
> nrows <- 6 ; ncols <- 4
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> # Calculate the generalized inverse of matv
> invmat <- MASS::ginv(matv)
> round(invmat %*% matv, 4)
> all.equal(matv, matv %*% invmat %*% matv)
> # Random rectangular matrix: nrows < ncols
> nrows <- 4 ; ncols <- 6
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> # Calculate the generalized inverse of matv
> invmat <- MASS::ginv(matv)
> all.equal(matv, matv %*% invmat %*% matv)
> round(matv %*% invmat, 4)
> round(invmat %*% matv, 4)
> # Perform singular value decomposition
> svdec <- svd(matv)
> # Calculate the generalized inverse from SVD
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> all.equal(invsvd, invmat)
> # Calculate the Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matv) %*% matv) %*% t(matv)
> all.equal(invmp, invmat)
```

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

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

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

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

```
> # Create a random singular matrix
> # More columns than rows: ncols > nrows
> nrows <- 4 ; ncols <- 6
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> matv <- t(matv) %*% matv
> # Perform singular value decomposition
> svdec <- svd(matv)
> # Incorrect inverse from SVD because of zero singular values
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> # Inverse property doesn't hold
> all.equal(matv, matv %*% invsvd %*% matv)
```

```
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Check for zero singular values
> round(svdec$d, 12)
> notzero <- (svdec$d > (precv*svdec$d[1]))
> # Calculate the regularized inverse from SVD
> invsvd <- svdec$v[, notzero] %*%
+   (t(svdec$u[, notzero]) / svdec$d[notzero])
> # Verify inverse property of matv
> all.equal(matv, matv %*% invsvd %*% matv)
> # Calculate the regularized inverse using MASS::ginv()
> invmat <- MASS::ginv(matv)
> all.equal(invsvd, invmat)
> # Calculate the Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matv) %*% matv) %*% t(matv)
> all.equal(invmp, invmat)
```

Diagonalizing the 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
> unitmat <- matv %*% invmat
> round(unitmat, 4)
> round(matv %*% invmat, 4)
> round(t(svdec$u) %*% unitmat %*% svdec$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
> matv <- matrix(c(1, 2, -1, 2), nc=2)
> vecv <- c(2, 1)
> # Calculate the inverse of matv
> invmat <- solve(a=matv)
> invmat %*% matv
> # Calculate the solution using inverse of matv
> solutionv <- invmat %*% vecv
> matv %*% solutionv
> # Calculate the solution of linear system
> solutionv <- solve(a=matv, b=vecv)
> matv %*% solutionv
```

Fast Matrix Inverse Using C++

The *Armadillo* C++ functions can be several times faster than R functions - even those that are compiled from C++ code.

That's because the *Armadillo* C++ library calls routines optimized for fast numerical calculations.

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

The C++ *Armadillo* function `arma::inv()` calculates the matrix inverse several times faster than the function `solve()`.

The function `solve()` calculates the matrix inverse several times faster than the function `MASS::ginv()`.

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

// [[Rcpp::export]]
arma::mat calc_invmat(arma::mat& matv) {

  return arma::inv(matv);

}  // end calc_invmat
```

```
> # Create a random matrix
> matv <- matrix(rnorm(100), nc=10)
> # Calculate the matrix inverse using solve()
> invmatr <- solve(a=matv)
> round(invmatr %*% matv, 4)
> # Compile the C++ file using Rcpp
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/Rcpp/test_fun.cpp")
> # Calculate the matrix inverse using C++
> invmat <- calc_invmat(matv)
> all.equal(invmat, invmatr)
> # Compare the speed of RcppArmadillo with R code
> library(microbenchmark)
> summary(microbenchmark(
+   ginv=MASS::ginv(matv),
+   solve=solve(matv),
+   cpp=calc_invmat(matv),
+   times=10))[, c(1, 4, 5)]
```

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 matrix
> matv <- matrix(runif(1e4), nc=100)
> matv <- t(matv) %*% matv
> # Calculate the eigen decomposition
> eigend <- eigen(matv)
> eigenval <- eigend$values
> eigenvec <- eigend$vectors
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # If needed convert to positive definite matrix
> notzero <- (eigenval > (precv*eigenval[1]))
> if (sum(!notzero) > 0) {
+   eigenval[!notzero] <- 2*precv
+   matv <- eigenvec %*% (eigenval * t(eigenvec))
+ }  # end if
> # Calculate the Cholesky matv
> cholmat <- chol(matv)
> cholmat[1:5, 1:5]
> all.equal(matv, t(cholmat) %*% cholmat)
> # Calculate the inverse from Cholesky
> invchol <- chol2inv(cholmat)
> all.equal(solve(matv), invchol)
> # Compare speed of Cholesky inversion
> library(microbenchmark)
> summary(microbenchmark(
+   solve=solve(matv),
+   cholmat=chol2inv(chol(matv)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

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 multiplying them by the *Cholesky* matrix $\mathbb{L}$:

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

```
> # Calculate the random covariance matrix
> covmat <- matrix(runif(25), nc=5)
> covmat <- t(covmat) %*% covmat
> # Calculate the Cholesky matrix
> cholmat <- chol(covmat)
> cholmat
> # Simulate random uncorrelated returns
> nassets <- 5
> nrows <- 10000
> retp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate the correlated returns by applying Cholesky
> retscorr <- retp %*% cholmat
> # Calculate the covariance matrix
> covmat2 <- crossprod(retscorr) /(nrows-1)
> all.equal(covmat, covmat2)
```

Eigenvalues of Singular Covariance Matrices

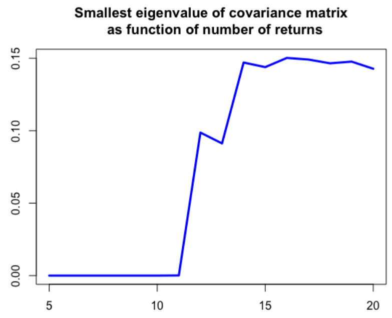
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of k stocks (columns), over n time periods (rows), then the sample covariance matrix is equal to:

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

If the number of rows is less than the number of stocks, then the returns are *collinear*, and the sample covariance matrix is *singular*, with some *eigenvalues* equal to zero.

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

```
> # Simulate random stock returns
> nassets <- 10
> nrows <- 100
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> retp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate the centered (de-meaned) returns matrix
> retp <- t(t(retp) - colMeans(retp))
> # Or
> retp <- apply(retp, MARGIN=2, function(x) (x-mean(x)))
> # Calculate the covariance matrix
> covmat <- crossprod(retp) /(nrows-1)
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(covmat)
> eigend$values
> barplot(eigend$values, # Plot eigenvalues
+   xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(eigend$values)),
+   main="Eigenvalues of Covariance Matrix")
```



```
> # Calculate the eigenvalues and eigenvectors
> # as function of number of returns
> ndata <- ((nassets/2):(2*nassets))
> eigenval <- sapply(ndata, function(x) {
+   retp <- retp[1:x, ]
+   retp <- apply(retp, MARGIN=2, function(y) (y - mean(y)))
+   covmat <- crossprod(retp) / (x-1)
+   min(eigen(covmat)$values)
+ })  # end sapply
> plot(y=eigenval, x=ndata, t="l", xlab="", ylab="", lwd=3, col="blu
+   main="Smallest eigenvalue of covariance matrix
+   as function of number of returns")
```

Regularized Inverse of Singular Covariance Matrices

The *regularization* technique allows calculating the inverse of *singular* covariance matrices while reducing the effects of statistical noise.

If the number of time periods of returns is less than the number of assets (columns), then the covariance matrix of returns is *singular*, and some of its *eigenvalues* are zero, so it doesn't have an inverse.

The *regularized* inverse $\mathbb{C}_n^{-1}$ is calculated by removing the higher order eigenvalues that are almost zero, and keeping only the first *n eigenvalues*:

$$\mathbb{C}_n^{-1} = \mathbb{O}_n \, \Sigma_n^{-1} \, \mathbb{O}_n^T$$

Where Σ_n and $\mathbb{O}_n$ are matrices with the higher order eigenvalues and eigenvectors removed.

The function `MASS::ginv()` calculates the *regularized* inverse of a matrix.

```
> # Create rectangular matrix with collinear columns
> matv <- matrix(rnorm(10*8), nc=10)
> # Calculate the covariance matrix
> covmat <- cov(matv)
> # Calculate the inverse of covmat - error
> invmat <- solve(covmat)
> # Calculate the regularized inverse of covmat
> invmat <- MASS::ginv(covmat)
> # Verify inverse property of matv
> all.equal(covmat, covmat %*% invmat %*% covmat)
> # Perform eigen decomposition
> eigend <- eigen(covmat)
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Calculate the regularized inverse matrix
> notzero <- (eigenval > (precv * eigenval[1]))
> invreg <- eigenvec[, notzero] %*%
+   (t(eigenvec[, notzero]) / eigenval[notzero])
> # Verify that invmat is same as invreg
> all.equal(invmat, invreg)
```

The Bias-Variance Tradeoff of the Regularized Inverse

Removing the very small higher order eigenvalues can also be used to reduce the propagation of statistical noise and improve the signal-to-noise ratio.

Removing a larger number of eigenvalues further reduces the noise, but it increases the bias of the covariance matrix.

This is an example of the *bias-variance tradeoff*.

Even though the *regularized* inverse $\mathbb{C}_n^{-1}$ does not satisfy the matrix inverse property, its out-of-sample forecasts may be more accurate than those using the actual inverse matrix.

The parameter `dimax` specifies the number of eigenvalues used for calculating the *regularized* inverse of the covariance matrix of returns.

The optimal value of the parameter `dimax` can be determined using *backtesting* (*cross-validation*).

```
> # Calculate the regularized inverse matrix using cutoff
> dimax <- 3
> invmat <- eigenvec[, 1:dimax] %*%
+   (t(eigenvec[, 1:dimax]) / eigend$values[1:dimax])
> # Verify that invmat is same as invreg
> all.equal(invmat, invreg)
```

Shrinkage Estimator of Covariance Matrices

The estimates of the covariance matrix suffer from statistical noise, and those noise are magnified when the covariance matrix is inverted.

In the *shrinkage* technique the covariance matrix $\mathbb{C}_s$ is estimated as a weighted sum of the sample covariance estimator $\mathbb{C}$ plus a target matrix $\mathbb{T}$:

$$\mathbb{C}_s = (1 - \alpha)\,\mathbb{C} + \alpha\,\mathbb{T}$$

The target matrix $\mathbb{T}$ represents an estimate of the covariance matrix subject to some constraint, such as that all the correlations are equal to each other.

The shrinkage intensity α determines the amount of shrinkage that is applied, with $\alpha = 1$ representing a complete shrinkage towards the target matrix.

The *shrinkage* estimator reduces the estimate variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).

```
> # Create a random covariance matrix
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> matv <- matrix(rnorm(5e2), nc=5)
> covmat <- cov(matv)
> cormat <- cor(matv)
> stdev <- sqrt(diag(covmat))
> # Calculate the target matrix
> cormean <- mean(cormat[upper.tri(cormat)])
> targetmat <- matrix(cormean, nr=NROW(covmat), nc=NCOL(covmat))
> diag(targetmat) <- 1
> targetmat <- t(t(targetmat * stdev) * stdev)
> # Calculate the shrinkage covariance matrix
> alphac <- 0.5
> covshrink <- (1-alphac)*covmat + alphac*targetmat
> # Calculate the inverse matrix
> invmat <- solve(covshrink)
```

Recursive Matrix Inverse

The inverse of a square matrix $\mathbb{A}$ can be calculated approximately using the recursive *Schulz formula*:

$$\mathbb{A}_{i+1}^{-1} \leftarrow 2\mathbb{A}_i^{-1} - \mathbb{A}_i^{-1}\mathbb{A}\mathbb{A}_i^{-1}$$

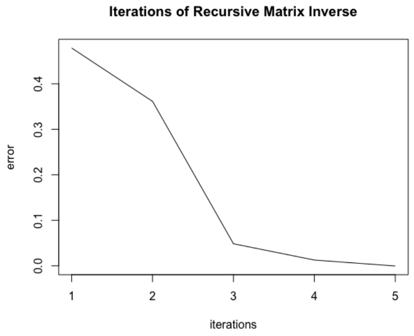
The *Schulz formula* requires a good initial value for the inverse matrix $\mathbb{A}_1^{-1}$ or else the recursion diverges.

If the initial inverse matrix $\mathbb{A}_1^{-1}$ is very close to the actual inverse $\mathbb{A}^{-1}$, then the *Schulz formula* produces a very good approximation with just a few iterations.

The *Schulz formula* is useful for updating the inverse when the matrix $\mathbb{A}$ changes only slightly. For example, for updating the inverse of the covariance matrix as it changes slowly over time.

The super-assignment operator "<<-" modifies variables in the *enclosing* environment in which the function was *defined* (*lexical* scoping).

```
> # Create a random matrix
> matv <- matrix(rnorm(100), nc=10)
> # Calculate the inverse of matv
> invmat <- solve(a=matv)
> # Multiply inverse with matrix
> round(invmat %*% matv, 4)
> # Calculate the initial inverse
> invmatr <- invmat + matrix(rnorm(100, sd=0.1), nc=10)
> # Calculate the approximate recursive inverse of matv
> invmatr <- (2*invmatr - invmatr %*% matv %*% invmatr)
> # Calculate the sum of the off-diagonal elements
> sum((invmatr %*% matv)[upper.tri(matv)])
```



```
> # Calculate the recursive inverse of matv in a loop
> invmatr <- invmat + matrix(rnorm(100, sd=0.1), nc=10)
> iterv <- sapply(1:5, function(x) {
+ # Calculate the recursive inverse of matv
+   invmatr <<- (2*invmatr - invmatr %*% matv %*% invmatr)
+ # Calculate the sum of the off-diagonal elements
+   sum((invmatr %*% matv)[upper.tri(matv)])
+ })  # end sapply
> # Plot the iterations
> plot(x=1:5, y=iterv, t="l", xlab="iterations", ylab="error",
+      main="Iterations of Recursive Matrix Inverse")
```

Downloading Treasury Bond Rates from *FRED*

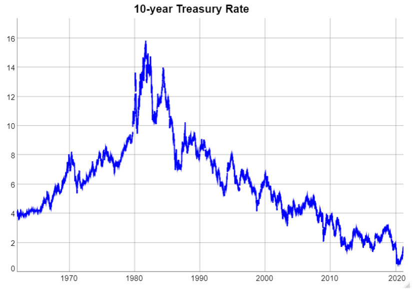
The constant maturity Treasury rates are yields of hypothetical fixed-maturity bonds, interpolated from the market yields of actual Treasury bonds.

The *FRED* database contains current and historical constant maturity Treasury rates,
https://fred.stlouisfed.org/series/DGS5

quantmod::getSymbols() creates objects in the specified *environment* from the input strings (names).

It then assigns the data to those objects, without returning them as a function value, as a *side effect*.

```
> # Symbols for constant maturity Treasury rates
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20", "DGS30")
> # Create new environment for time series
> ratesenv <- new.env()
> # Download time series for symbolv into ratesenv
> quantmod::getSymbols(symbolv, env=ratesenv, src="FRED")
> # List files in ratesenv
> ls(ratesenv)
> # Get class of all objects in ratesenv
> sapply(ratesenv, class)
> # Get class of all objects in R workspace
> sapply(ls(), function(name) class(get(name)))
> # Save the time series environment into a binary .RData file
> save(ratesenv, file="/Users/jerzy/Develop/lecture_slides/data/ra
```



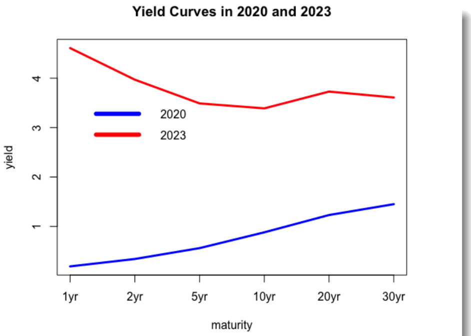
```
> # Get class of time series object DGS10
> class(get(x="DGS10", envir=ratesenv))
> # Another way
> class(ratesenv$DGS10)
> # Get first 6 rows of time series
> head(ratesenv$DGS10)
> # Plot dygraphs of 10-year Treasury rate
> dygraphs::dygraph(ratesenv$DGS10, main="10-year Treasury Rate") %
+   dyOptions(colors="blue", strokeWidth=2)
> # Plot 10-year constant maturity Treasury rate
> x11(width=6, height=5)
> par(mar=c(2, 2, 0, 0), oma=c(0, 0, 0, 0))
> chart_Series(ratesenv$DGS10["1990/"], name="10-year Treasury Rate
```

Treasury Yield Curve

The *yield curve* is a vector of interest rates at different maturities, on a given date.

The *yield curve* shape changes depending on the economic conditions: in recessions rates drop and the curve flattens, while in expansions rates rise and the curve steepens.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get most recent yield curve
> ycnow <- eapply(ratesenv, xts::last)
> class(ycnow)
> ycnow <- do.call(cbind, ycnow)
> # Check if 2020-03-25 is not a holiday
> date2020 <- as.Date("2020-03-25")
> weekdays(date2020)
> # Get yield curve from 2020-03-25
> yc2020 <- eapply(ratesenv, function(x) x[date2020])
> yc2020 <- do.call(cbind, yc2020)
> # Combine the yield curves
> ycurves <- c(yc2020, ycnow)
> # Rename columns and rows, sort columns, and transpose into matr
> colnames(ycurves) <- substr(colnames(ycurves), start=4, stop=11)
> ycurves <- ycurves[, order(as.numeric(colnames(ycurves)))]
> colnames(ycurves) <- paste0(colnames(ycurves), "yr")
> ycurves <- t(ycurves)
> colnames(ycurves) <- substr(colnames(ycurves), start=1, stop=4)
```

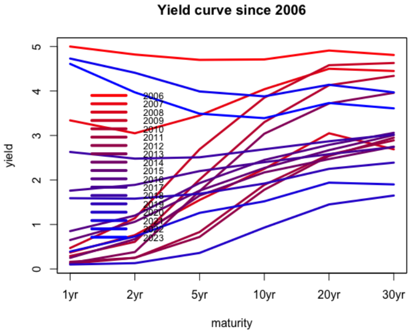


```
> # Plot using matplot()
> colorv <- c("blue", "red")
> matplot(ycurves, main="Yield Curves in 2020 and 2023", xaxt="n",
+   type="l", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(ycurves)), rownames(ycurves))
> # Add legend
> legend("topleft", legend=colnames(ycurves), y.intersp=0.1,
+  bty="n", col=colorv, lty=1, lwd=6, inset=0.05, cex=1.0)
```

Treasury Yield Curve Over Time

The *yield curve* has changed shape dramatically depending on the economic conditions: in recessions rates drop and the curve flattens, while in expansions rates rise and the curve steepens.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get end-of-year dates since 2006
> datev <- xts::endpoints(ratesenv$DGS1["2006/"], on="years")
> datev <- zoo::index(ratesenv$DGS1["2006/"][datev])
> # Create time series of end-of-year rates
> ycurves <- eapply(ratesenv, function(ratev) ratev[datev])
> ycurves <- rutils::do_call(cbind, ycurves)
> # Rename columns and rows, sort columns, and transpose into matrix
> colnames(ycurves) <- substr(colnames(ycurves), start=4, stop=11)
> ycurves <- ycurves[, order(as.numeric(colnames(ycurves)))]
> colnames(ycurves) <- paste0(colnames(ycurves), "yr")
> ycurves <- t(ycurves)
> colnames(ycurves) <- substr(colnames(ycurves), start=1, stop=4)
> # Plot matrix using plot.zoo()
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(ycurves))
> plot.zoo(ycurves, main="Yield Curve Since 2006", lwd=3, xaxt="n"
+    plot.type="single", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(ycurves)), rownames(ycurves))
> # Add legend
> legend("topleft", legend=colnames(ycurves), y.intersp=0.1,
+  bty="n", col=colorv, lty=1, lwd=4, inset=0.05, cex=0.8)
```



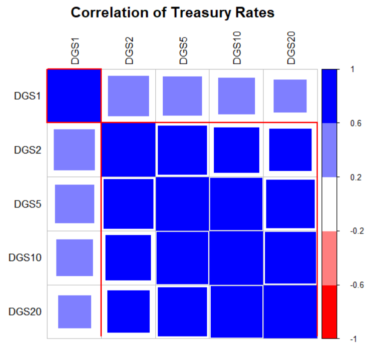
```
> # Alternative plot using matplot()
> matplot(ycurves, main="Yield curve since 2006", xaxt="n", lwd=3, l
+   type="l", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(ycurves)), rownames(ycurves))
> # Add legend
> legend("topleft", legend=colnames(ycurves), y.intersp=0.1,
+  bty="n", col=colorv, lty=1, lwd=4, inset=0.05, cex=0.8)
```

Covariance Matrix of Interest Rates

The covariance matrix $\mathbb{C}$, of the interest rate matrix $\mathbf{r}$ is given by:

$$\mathbb{C} = \frac{(\mathbf{r} - \bar{\mathbf{r}})^T (\mathbf{r} - \bar{\mathbf{r}})}{n - 1}$$

```
> # Extract rates from ratesenv
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20")
> ratem <- mget(symbolv, envir=ratesenv)
> ratem <- rutils::do_call(cbind, ratem)
> ratem <- zoo::na.locf(ratem, na.rm=FALSE)
> ratem <- zoo::na.locf(ratem, fromLast=TRUE)
> # Calculate daily percentage rates changes
> retp <- rutils::diffit(log(ratem))
> # Center (de-mean) the returns
> retp <- lapply(retp, function(x) {x - mean(x)})
> retp <- rutils::do_call(cbind, retp)
> sapply(retp, mean)
> # Covariance and Correlation matrices of Treasury rates
> covmat <- cov(retp)
> cormat <- cor(retp)
> # Reorder correlation matrix based on clusters
> library(corrplot)
> ordern <- corrMatOrder(cormat, order="hclust",
+   hclust.method="complete")
> cormat <- cormat[ordern, ordern]
```



```
> # Plot the correlation matrix
> x11(width=6, height=6)
> colorv <- colorRampPalette(c("red", "white", "blue"))
> corrplot(cormat, title=NA, tl.col="black",
+     method="square", col=colorv(NCOL(cormat)), tl.cex=0.8,
+     cl.offset=0.75, cl.cex=0.7, cl.align.text="l", cl.ratio=0.25)
> title("Correlation of Treasury Rates", line=1)
> # Draw rectangles on the correlation matrix plot
> corrRect.hclust(cormat, k=NROW(cormat) %/% 2,
+   method="complete", col="red")
```

Principal Component Vectors

Principal components are linear combinations of the `k` return vectors $\mathbf{r}_i$:

$$\mathbf{pc}_j = \sum_{i=1}^{k} w_{ij} \, \mathbf{r}_i$$

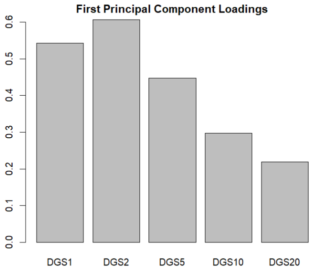
Where $\mathbf{w}_j$ is a vector of weights (loadings) of the *principal component* `j`, with $\mathbf{w}_j^T \mathbf{w}_j = 1$.

The weights $\mathbf{w}_j$ are chosen to maximize the variance of the *principal components*, under the condition that they are orthogonal:

$$\mathbf{w}_j = \arg\max \left\{ \mathbf{pc}_j^T \, \mathbf{pc}_j \right\}$$

$$\mathbf{pc}_i^T \, \mathbf{pc}_j = 0 \; (i \neq j)$$

```
> # Create initial vector of portfolio weights
> nweights <- NROW(symbolv)
> weightv <- rep(1/sqrt(nweights), nweights)
> names(weightv) <- symbolv
> # Objective function equal to minus portfolio variance
> objfun <- function(weightv, retp) {
+   retp <- retp %*% weightv
+   -1e7*var(retp) + 1e7*(1 - sum(weightv*weightv))^2
+ }  # end objfun
> # Objective function for equal weight portfolio
> objfun(weightv, retp)
> # Compare speed of vector multiplication methods
> library(microbenchmark)
> summary(microbenchmark(
+   transp=t(retp) %*% retp,
+   sumv=sum(retp*retp),
```



```
> # Find weights with maximum variance
> optiml <- optim(par=weightv,
+   fn=objfun,
+   retp=retp,
+   method="L-BFGS-B",
+   upper=rep(5.0, nweights),
+   lower=rep(-5.0, nweights))
> # Optimal weights and maximum variance
> weights1 <- optiml$par
> objfun(weights1, retp)
> # Plot first principal component loadings
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> barplot(weights1, names.arg=names(weights1),
+   xlab="", ylab="", main="First Principal Component Loadings")
```

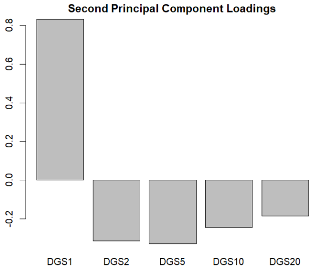
Higher Order Principal Components

The *second principal component* can be calculated by maximizing its variance, under the constraint that it must be orthogonal to the *first principal component*.

Similarly, higher order *principal components* can be calculated by maximizing their variances, under the constraint that they must be orthogonal to all the previous *principal components*.

The number of principal components is equal to the dimension of the covariance matrix.

```
> # pc1 weights and returns
> pc1 <- drop(retp %*% weights1)
> # Redefine objective function
> objfun <- function(weightv, retp) {
+   retp <- retp %*% weightv
+   -1e7*var(retp) + 1e7*(1 - sum(weightv^2))^2 +
+     1e7*sum(weights1*weightv)^2
+ }  # end objfun
> # Find second principal component weights
> optiml <- optim(par=weightv,
+               fn=objfun,
+               retp=retp,
+               method="L-BFGS-B",
+               upper=rep(5.0, nweights),
+               lower=rep(-5.0, nweights))
```



```
> # pc2 weights and returns
> weights2 <- optiml$par
> pc2 <- drop(retp %*% weights2)
> sum(pc1*pc2)
> # Plot second principal component loadings
> barplot(weights2, names.arg=names(weights2),
+   xlab="", ylab="", main="Second Principal Component Loadings")
```

Eigenvalues of the Covariance Matrix

The portfolio variance: $\mathbf{w}^T \mathbb{C} \mathbf{w}$ can be maximized under the *quadratic* weights constraint $\mathbf{w}^T \mathbf{w} = 1$, by maximizing the *Lagrangian* $\mathcal{L}$:

$$\mathcal{L} = \mathbf{w}^T \mathbb{C} \mathbf{w} \, - \, \lambda \, (\mathbf{w}^T \mathbf{w} - 1)$$

Where λ is a *Lagrange multiplier*.

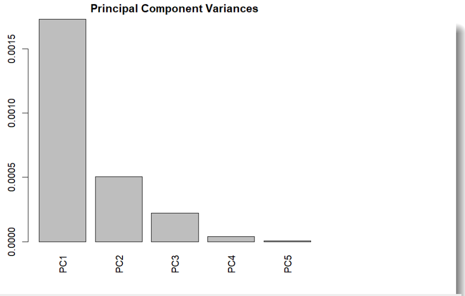
The maximum variance portfolio weights can be found by differentiating $\mathcal{L}$ with respect to $\mathbf{w}$ and setting it to zero:

$$\mathbb{C} \, \mathbf{w} = \lambda \, \mathbf{w}$$

The above is the *eigenvalue* equation of the covariance matrix $\mathbb{C}$, with the optimal weights $\mathbf{w}$ forming an *eigenvector*, and λ is the *eigenvalue* corresponding to the *eigenvector* $\mathbf{w}$.

The *eigenvalues* are the variances of the *eigenvectors*, and their sum is equal to the sum of the return variances:

$$\sum_{i=1}^{k} \lambda_i = \frac{1}{1-k} \sum_{i=1}^{k} \mathbf{r}_i^T \mathbf{r}_i$$



```
> eigend <- eigen(covmat)
> eigend$vectors
> # Compare with optimization
> all.equal(sum(diag(covmat)), sum(eigend$values))
> all.equal(abs(eigend$vectors[, 1]), abs(weights1), check.attribute
> all.equal(abs(eigend$vectors[, 2]), abs(weights2), check.attribute
> all.equal(eigend$values[1], var(pc1), check.attributes=FALSE)
> all.equal(eigend$values[2], var(pc2), check.attributes=FALSE)
> # Eigenvalue equations are satisfied approximately
> (covmat %*% weights1) / weights1 / var(pc1)
> (covmat %*% weights2) / weights2 / var(pc2)
> # Plot eigenvalues
> barplot(eigend$values, names.arg=paste0("PC", 1:nweights),
+   las=3, xlab="", ylab="", main="Principal Component Variances")
```

Principal Component Analysis Versus *Eigen Decomposition*

Principal Component Analysis (*PCA*) is equivalent to the *eigen decomposition* of either the correlation or the covariance matrix.

If the input time series *are* scaled, then *PCA* is equivalent to the eigen decomposition of the *correlation matrix*.

If the input time series *are not* scaled, then *PCA* is equivalent to the eigen decomposition of the *covariance matrix*.

Scaling the input time series improves the accuracy of the *PCA dimension reduction*, allowing a smaller number of *principal components* to more accurately capture the data contained in the input time series.

The function `prcomp()` performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class `prcomp`.

The `prcomp()` argument `scale=TRUE` specifies that the input time series should be scaled by their standard deviations.

```
> # Eigen decomposition of correlation matrix
> eigend <- eigen(cormat)
> # Perform PCA with scaling
> pcad <- prcomp(retp, scale=TRUE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
> # Eigen decomposition of covariance matrix
> eigend <- eigen(covmat)
> # Perform PCA without scaling
> pcad <- prcomp(retp, scale=FALSE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
```

Principal Component Analysis of the Yield Curve

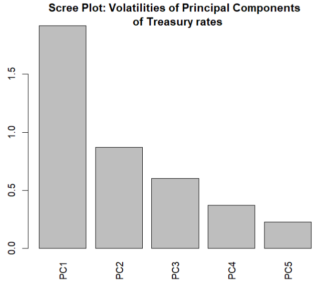
Principal Component Analysis (*PCA*) is a *dimension reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series.

The input time series are often scaled by their standard deviations, to improve the accuracy of *PCA dimension reduction*, so that more information is retained by the first few *principal component* time series.

If the input time series are not scaled, then *PCA* analysis is equivalent to the *eigen decomposition* of the covariance matrix, and if they are scaled, then *PCA* analysis is equivalent to the *eigen decomposition* of the correlation matrix.

The function prcomp() performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class prcomp.

The prcomp() argument scale=TRUE specifies that the input time series should be scaled by their standard deviations.



A *scree plot* is a bar plot of the volatilities of the *principal components*.

```
> # Perform principal component analysis PCA
> pcad <- prcomp(retp, scale=TRUE)
> # Plot standard deviations
> barplot(pcad$sdev, names.arg=colnames(pcad$rotation),
+   las=3, xlab="", ylab="",
+   main="Scree Plot: Volatilities of Principal Components
+   of Treasury rates")
```

Yield Curve Principal Component Loadings (Weights)

Principal component loadings are the weights of portfolios which have mutually orthogonal returns.

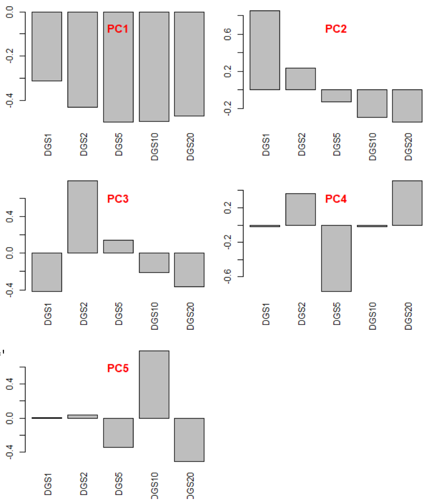
The *principal component* portfolios represent the different orthogonal modes of the data variance.

The first *principal component* of the *yield curve* is the correlated movement of all rates up and down.

The second *principal component* is *yield curve* steepening and flattening.

The third *principal component* is the *yield curve* butterfly movement.

```
> # Calculate principal component loadings (weights)
> pcad$rotation
> # Plot loading barplots in multiple panels
> par(mfrow=c(3,2))
> par(mar=c(3.5, 2, 2, 1), oma=c(0, 0, 0, 0))
> for (ordern in 1:NCOL(pcad$rotation)) {
+   barplot(pcad$rotation[, ordern], las=3, xlab="", ylab="", main='
+   title(paste0("PC", ordern), line=-2.0, col.main="red")
+ }  # end for
```



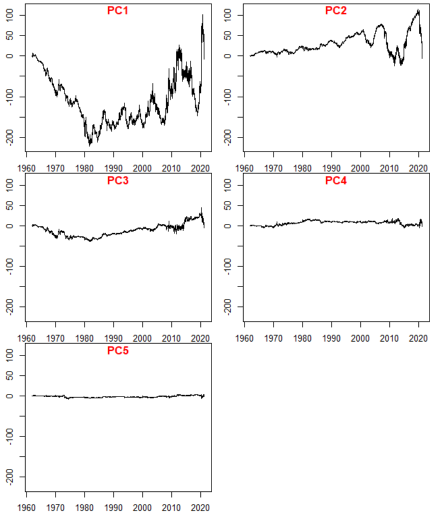
Yield Curve Principal Component Time Series

The time series of the *principal components* can be calculated by multiplying the loadings (weights) times the original data.

The *principal component* time series have mutually orthogonal returns.

Higher order *principal components* are gradually less volatile.

```
> # Standardize (center and scale) the returns
> retp <- lapply(retp, function(x) {(x - mean(x))/sd(x)})
> retp <- rutils::do_call(cbind, retp)
> sapply(retp, mean)
> sapply(retp, sd)
> # Calculate principal component time series
> retpcac <- retp %*% pcad$rotation
> all.equal(pcad$x, retpcac, check.attributes=FALSE)
> # Calculate products of principal component time series
> round(t(retpcac) %*% retpcac, 2)
> # Coerce to xts time series
> retpcac <- xts(retpcac, order.by=zoo::index(retp))
> retpcac <- cumsum(retpcac)
> # Plot principal component time series in multiple panels
> par(mfrow=c(3,2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> rangev <- range(retpcac)
> for (ordern in 1:NCOL(retpcac)) {
+   plot.zoo(retpcac[, ordern], ylim=rangev, xlab="", ylab="")
+   title(paste0("PC", ordern), line=-1, col.main="red")
+ }  # end for
```



Inverting Principal Component Analysis

The original time series can be calculated *exactly* from the time series of all the *principal components*, by inverting the loadings matrix.

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

```
> # Invert all the principal component time series
> retpca <- retp %*% pcad$rotation
> solved <- retpca %*% solve(pcad$rotation)
> all.equal(coredata(retp), solved)
```

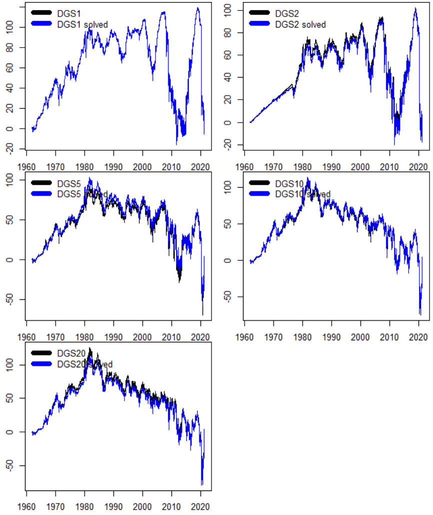
Dimension Reduction Using Principal Component Analysis

The original time series can be calculated *approximately* from just the first few *principal components*, which demonstrates that *PCA* is a form of *dimension reduction*.

A popular rule of thumb is to use the *principal components* with the largest variances, which sum up to 80% of the total variance of returns.

The *Kaiser-Guttman* rule uses only *principal components* with variance greater than 1.

```
> # Invert first 3 principal component time series
> solved <- retpca[, 1:3] %*% solve(pcad$rotation)[1:3, ]
> solved <- xts::xts(solved, zoo::index(retp))
> solved <- cumsum(solved)
> retc <- cumsum(retp)
> # Plot the solved returns
> par(mfrow=c(3,2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> for (symbol in symbolv) {
+   plot.zoo(cbind(retc[, symbol], solved[, symbol]),
+     plot.type="single", col=c("black", "blue"), xlab="", ylab="")
+   legend(x="topleft", bty="n", y.intersp=0.1,
+    legend=paste0(symboln, c("", " solved")),
+    title=NULL, inset=0.0, cex=1.0, lwd=6,
+    lty=1, col=c("black", "blue"))
+ }  # end for
```



Calibrating Yield Curve Using Package *RQuantLib*

The package *RQuantLib* is an interface to the *QuantLib* open source C/C++ library for quantitative finance, mostly designed for pricing fixed-income instruments and options.

The function `DiscountCurve()` calibrates a *zero coupon yield curve* from *money market* rates, *Eurodollar* futures, and *swap* rates.

The function `DiscountCurve()` interpolates the *zero coupon* rates into a vector of dates specified by the `times` argument.

```
> library(RQuantLib)  # Load RQuantLib
> # Specify curve parameters
> curvep <- list(tradeDate=as.Date("2018-01-17"),
+          settleDate=as.Date("2018-01-19"),
+          dt=0.25,
+          interpWhat="discount",
+          interpHow="loglinear")
> # Specify market data: prices of FI instruments
> pricev <- list(d3m=0.0363,
+          fut1=96.2875,
+          fut2=96.7875,
+          fut3=96.9875,
+          fut4=96.6875,
+          s5y=0.0443,
+          s10y=0.05165,
+          s15y=0.055175)
> # Specify dates for calculating the zero rates
> datev <- seq(0, 10, 0.25)
> # Specify the evaluation (as of) date
> setEvaluationDate(as.Date("2018-01-17"))
> # Calculate the zero rates
> ratev <- DiscountCurve(params=curvep, tsQuotes=pricev, times=datev
> # Plot the zero rates
> x11()
> plot(x=ratev$zerorates, t="l", main="zerorates")
```

Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture_3.pdf*, and run all the code in *FRE6871_Lecture_3.R*
- Read about the *bootstrap technique* in:
  bootstrap_technique.pdf and *doBootstrap_primer.pdf*
- Read about applying the *importance sampling technique* for calculating *CVaR*:
  Muller CVAR Importance Sampling.pdf

Recommended

- Read about why *CVaR* is a coherent risk measure:
  https://en.wikipedia.org/wiki/Expected_shortfall
  https://en.wikipedia.org/wiki/Coherent_risk_measure#Value_at_risk
- Read about why *CVaR* has very large standard errors:
  Danielsson CVAR Estimation Standard Error.pdf
  http://www.bloomberg.com/view/articles/2016-05-23/big-banks-risk-does-not-compute