FRE7241 Algorithmic Portfolio Management

Lecture#5, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

February 27, 2018

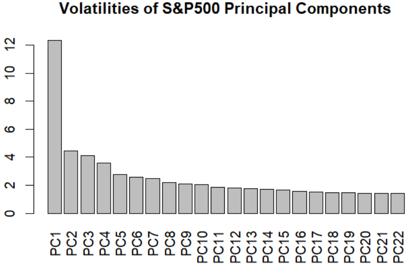
Principal Components of *S&P500* Stock Constituents

The *PCA* standard deviations are the volatilities of the *principal component* time series,

The original time series of returns can be calculated approximately from the first few *principal components* with the largest standard deviations,

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

Another rule of thumb is to use the *principal components* with the largest standard deviations which sum up to 80% of the total variance of returns,



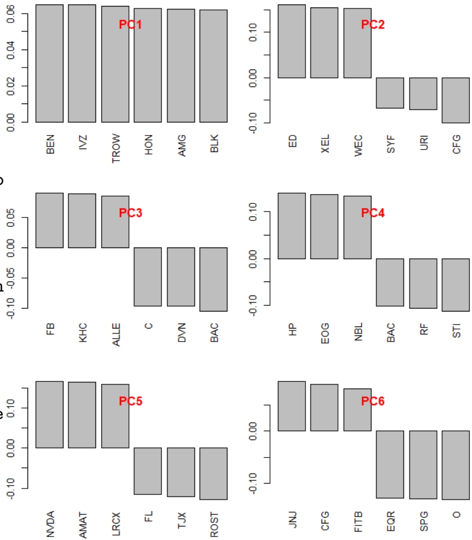

```
> # load S&P500 constituent stock prices
> load("C:/Develop/R/lecture_slides/data/sp500_prices.RData")
> date_s <- index(price_s)
> # calculate simple returns (not percentage)
> re_turns <- rutils::diff_it(price_s)
> # de-mean (center) and scale the returns
> re_turns <- t(t(re_turns) - colMeans(re_turns))
> re_turns <- t(t(re_turns) / sqrt(colSums(re_turns^2)/(NROW(re_turns)-1)))
> re_turns <- xts(re_turns, date_s)
> # perform principal component analysis PCA
> pc_a <- prcomp(re_turns, scale=TRUE)
> # find number of components with variance greater than 2
```

S&P500 Principal Component Loadings (Weights)

Principal component loadings are the weights of *principal component* portfolios,

The *principal component* portfolios have mutually orthogonal returns represent the different orthogonal modes of the return variance,

```
> # Principal component loadings (weights)
> # Plot barplots with PCA weights in multiple p
> n_comps <- 6
> par(mfrow=c(n_comps/2, 2))
> par(mar=c(4, 2, 2, 1), oma=c(0, 0, 0, 0))
> # First principal component weights
> weight_s <- sort(pc_a$rotation[, 1], decreasin
> barplot(weight_s[1:6],
+   las=3, xlab="", ylab="", main="")
> title(paste0("PC", 1), line=-2.0,
+ col.main="red")
> for (or_der in 2:n_comps) {
+   weight_s <- sort(pc_a$rotation[, or_der], de
+   barplot(weight_s[c(1:3, 498:500)],
+   las=3, xlab="", ylab="", main="")
+   title(paste0("PC", or_der), line=-2.0,
+   col.main="red")
+ }  # end for
```

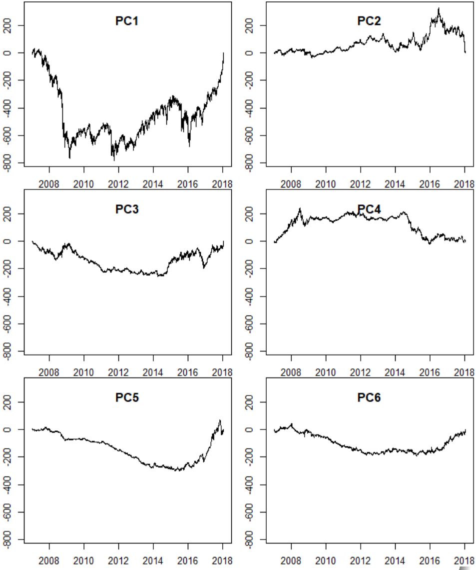


S&P500 Principal Component Time Series

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

Higher order *principal components* are gradually less volatile,

```
> # principal component time series
> pca_rets <- xts(re_turns %*% pc_a$rotation[, 1
+             order.by=date_s)
> round(cov(pca_rets), 3)
> pca_ts <- xts:::cumsum.xts(pca_rets)
> # plot principal component time series in mult
> par(mfrow=c(n_comps/2, 2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> ra_nge <- range(pca_ts)
> for (or_der in 1:n_comps) {
+   plot.zoo(pca_ts[, or_der],
+      ylim=ra_nge,
+      xlab="", ylab="")
+   title(paste0("PC", or_der), line=-2.0)
+ }  # end for
```



S&P500 Factor Model From *Principal Components*

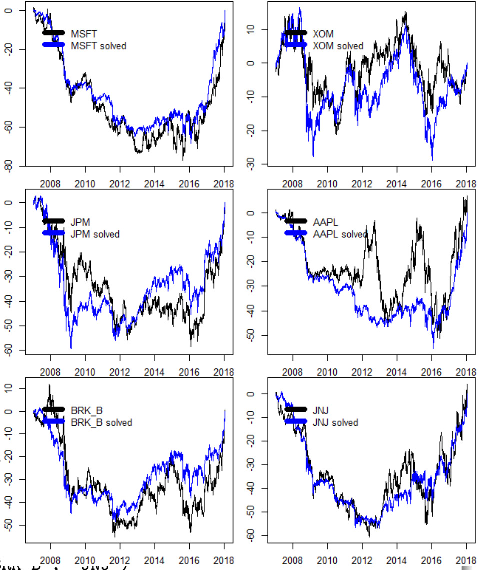
By inverting the *PCA* analysis, the *S&P500* constituent returns can be calculated from the first k *principal components* under a *factor model*:

$$\mathbf{r}_i = \alpha_i + \sum_{j=1}^{k} \beta_{ji} \, \mathbf{F}_j + \varepsilon_i$$

The *principal components* are interpreted as *market factors*: $\mathbf{F}_j = \mathbf{pc}_j$,

The *market betas* are the inverse of the *principal component loadings*: $\beta_{ji} = w_{ij}$,

The ε_i are the *idiosyncratic* returns, which should be mutually independent and uncorrelated to the *market factor* returns,




```
> # invert principal component time series
> inv_rotation <- solve(pc_a$rotation)
> all.equal(inv_rotation, t(pc_a$rotation))
> sol_ved <- pca_rets %*% inv_rotation[1:n_comps
> sol_ved <- xts::xts(sol_ved, date_s)
> sol_ved <- xts:::cumsum.xts(sol_ved)
> cum_returns <- xts:::cumsum.xts(re_turns)
> # plot the solved returns
> sym_bols <- c("MSFT", "XOM", "JPM", "AAPL", "B
> for (sym_bol in sym_bols) {
```

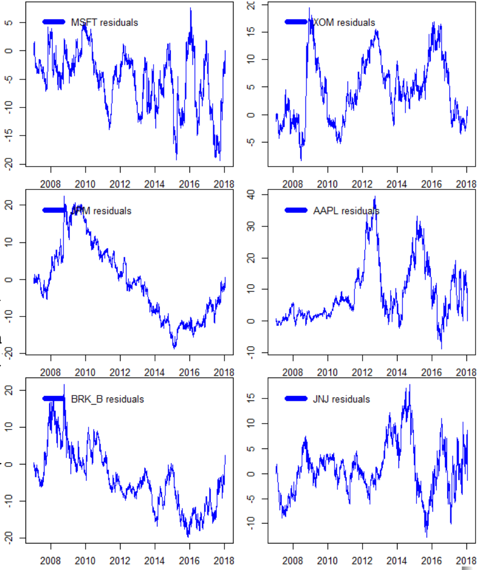
S&P500 Factor Model Residuals

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

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

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

```
> # perform ADF unit-root tests on original seri
> sapply(sym_bols, function(sym_bol) {
+   c(series=tseries::adf.test(cum_returns[, sym
+     resid=tseries::adf.test(cum_returns[, sym_
+ })  # end sapply
> # plot the residuals
> for (sym_bol in sym_bols) {
+   plot.zoo(cum_returns[, sym_bol] - sol_ved[,
+     plot.type="single", col="blue", xlab="", y
+   legend(x="topleft", bty="n",
+    legend=paste0(sym_bol, " residuals"),
+    title=NULL, inset=0.05, cex=1.0, lwd=6,
+    lty=1, col="blue")
+ }  # end for
> # perform ADF unit-root test on principal component time series
```



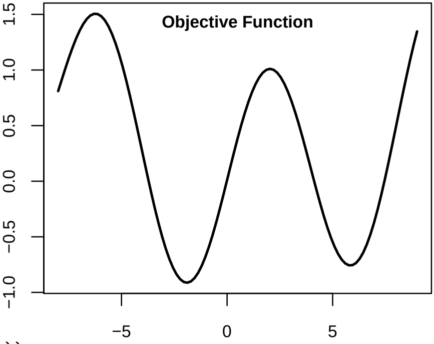
One-dimensional Optimization Using The Functional `optimize()`

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

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

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

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



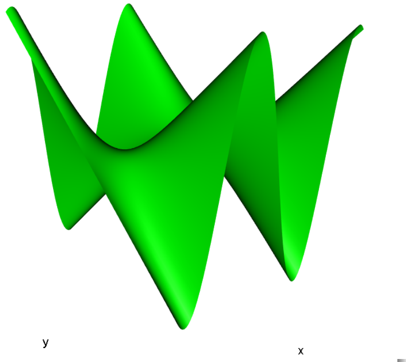

```
> # plot the objective function
> curve(expr=object_ive, type="l", xlim=c(-8, 9),
+ xlab="", ylab="", lwd=2)
> # add title
> title(main="Objective Function", line=-1)
```

Package *rgl* for Interactive 3d Surface Plots

The function persp3d() plots an *interactive* 3d surface plot of a function or a matrix,

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

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



Multi-dimensional Optimization Using optim()

The functional optim() performs *multi-dimensional* optimization,

The argument fn is the objective function to be minimized,

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

The argument par is the initial vector argument value,

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

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

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

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

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

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

The Log-likelihood Function

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

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

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

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

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

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

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

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

Perspective Plot of Likelihood Function

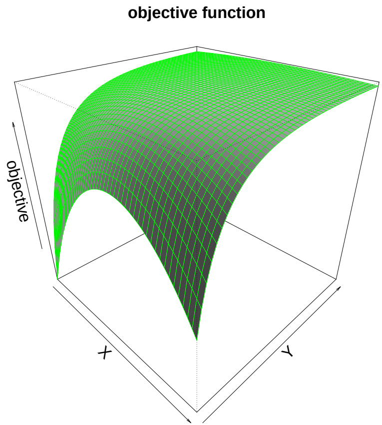
The function persp() plots a 3d perspective surface plot of a function specified over a grid of argument values,

The argument "z" accepts a matrix containing the function values,

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

The function persp3d() plots an *interactive* 3d surface plot of a function or a matrix,

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



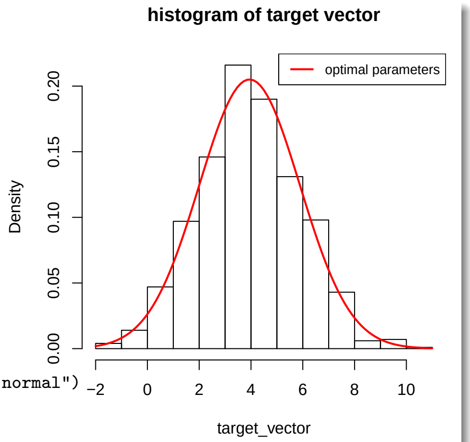

```
> # perspective plot of log-likelihood function
> persp(z=-objective_grid,
+ theta=45, phi=30, shade=0.5,
+ border="green", zlab="objective",
+ main="objective function")
> # interactive perspective plot of log-likelihood function
> library(rgl)  # load package rgl
> par3d(cex=2.0)  # scale text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

Optimization of Objective Function

The function optim() performs optimization of an objective function,

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

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

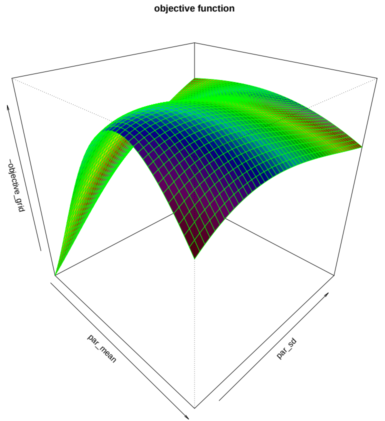



```
> # plot histogram
> histo_gram <- hist(sam_ple, plot=FALSE)
> plot(histo_gram, freq=FALSE,
+      main="histogram of sample")
> curve(expr=dnorm(x, mean=optim_fit$par["mean"],
+            sd=optim_fit$par["sd"]),
+ add=TRUE, type="l", lwd=2, col="red")
```

Mixture Model Likelihood Function

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

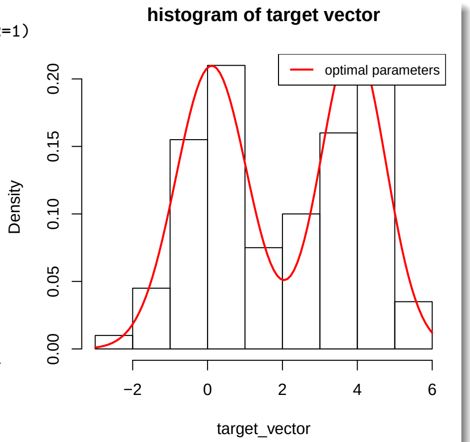
```
> # perspective plot of objective function
> persp(par_mean, par_sd, -objective_grid,
+ theta=45, phi=30,
+ shade=0.5,
+ col=rainbow(50),
+ border="green",
+ main="objective function")
```



Optimization of Mixture Model

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

```
> # plot histogram
> histo_gram <- hist(sam_ple, plot=FALSE)
> plot(histo_gram, freq=FALSE,
+      main="histogram of sample")
> fit_func <- function(x, pa_r) {
+   pa_r["weight"] *
+     dnorm(x, mean=pa_r["m1"], sd=pa_r["s1"]) +
+   (1-pa_r["weight"]) *
+     dnorm(x, mean=pa_r["m2"], sd=pa_r["s2"])
+ }  # end fit_func
> curve(expr=fit_func(x, pa_r=optim_fit$par), add=TRUE,
+ type="l", lwd=2, col="red")
> legend("topright", inset=0.0, cex=0.8, title=NULL,
+  leg="optimal parameters",
+  lwd=2, bg="white", col="red")
```



Package *DEoptim* for Global Optimization

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

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

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

The first generation of solutions is selected randomly,

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

The best solutions are selected for creating the next generation,

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

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

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

Vector and Matrix Calculus

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

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

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

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

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

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

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

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

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

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

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

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

Maximum Return Portfolio Using Linear Programming

The weights of the maximum return portfolio are obtained by maximizing the portfolio returns:

$$w_{max} = \underset{w}{\arg\max}[\, r^T w \,] = \underset{w}{\arg\max}[\sum_{i=1}^{n} w_i r_i \,]$$

Where r is the vector of returns, and w is the vector of portfolio weights, constrained by:

$$w^T \mathbb{1} = \sum_{i=1}^{n} w_i = 1$$

$$0 \leq w_i \leq 1$$

The weights of the maximum return portfolio can be calculated using linear programming (*LP*), which is the optimization of linear objective functions subject to linear constraints,

The function `Rglpk_solve_LP()` from package *Rglpk* solves linear programming problems by calling the *GNU Linear Programming Kit* library,

```
> # vector of symbol names
> sym_bols <- c("VTI", "IEF", "DBC")
> n_weights <- NROW(sym_bols)
> # calculate mean returns
> re_turns <- rutils::env_etf$re_turns[, sym_bol
> mean_rets <- sapply(re_turns, mean)
> # specify weight constraints
> constraint_s <- matrix(c(rep(1, n_weights),
+                 1, 1, 0),
+                 nc=n_weights, byrow=TRUE)
> direction_s <- c("==", "<=")
> rh_s <- c(1, 0)
> # specify weight bounds (-1, 1) (default is c(
> bound_s <-
+   list(lower=list(ind=1:n_weights, val=rep(-1,
+  upper=list(ind=1:n_weights, val=rep(1, n_weig
> # perform optimization
> op_tim <- Rglpk::Rglpk_solve_LP(
+   obj=mean_rets,
+   mat=constraint_s,
+   dir=direction_s,
+   rhs=rh_s,
+   bounds=bound_s,
+   max=TRUE)
> unlist(op_tim[1:2])
```

Minimum Variance Portfolio Weights

If $\mathbb{C}$ is equal to the covariance matrix of returns, then the portfolio variance is equal to:

$$w^T \mathbb{C} \, w$$

Where the sum of portfolio weights w_i is constrained to equal 1: $w^T \mathbb{1} = \sum_{i=1}^n w_i = 1$,

The weights that minimize the portfolio variance can be found by minimizing the *Lagrangian*:

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

Where λ is a *Lagrange multiplier*,

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

$$d_w[w^T \mathbb{1}] = d_w[\mathbb{1}^T w] = \mathbb{1}^T$$
$$d_w[w^T r] = d_w[r^T w] = r^T$$
$$d_w[w^T \mathbb{C} \, w] = w^T \mathbb{C} + w^T \mathbb{C}^T$$

Where $\mathbb{1}$ is the unit vector, and $w^T \mathbb{1} = \mathbb{1}^T w = \sum_{i=1}^n x_i$

The derivative of the *Lagrangian* $\mathcal{L}$ with respect to w is given by:

$$d_w \mathcal{L} = 2 w^T \mathbb{C} - \lambda \mathbb{1}^T$$

By setting the derivative to zero we find w equal to:

$$w = \frac{1}{2} \lambda \, \mathbb{C}^{-1} \mathbb{1}$$

By multiplying the above from the left by $\mathbb{1}^T$, and using $w^T \mathbb{1} = 1$, we find λ to be equal to:

$$\lambda = \frac{2}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}}$$

And finally the portfolio weights are then equal to:

$$w = \frac{\mathbb{C}^{-1} \mathbb{1}}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}}$$

Variance of Minimum Variance Portfolio

The weights of the minimum variance portfolio under the constraint $w^T \mathbb{1} = 1$ can be calculated using the inverse of the covariance matrix:

$$w = \frac{\mathbb{C}^{-1} \mathbb{1}}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}}$$

The variance of the minimum variance portfolio is equal to:

$$VAR = \frac{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{C} \, \mathbb{C}^{-1} \mathbb{1}}{(\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1})^2} = \frac{1}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}}$$

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:

The function `drop()` removes any dimensions of length *one*,

```
> # define a covariance matrix
> std_devs <- c(asset1=0.3, asset2=0.6)
> cor_rel <- 0.8
> co_var <- matrix(c(1, cor_rel, cor_rel, 1),
+                 nc=2)
> co_var <- t(t(std_devs*co_var)*std_devs)
> # calculate inverse of covariance mat_rix
> in_verse <- solve(a=co_var)
> u_nit <- rep(1, NCOL(co_var))
> # minimum variance weights with constraint
> # weight_s <- solve(a=co_var, b=u_nit)
> weight_s <- in_verse %*% u_nit
> weight_s <- weight_s / drop(t(u_nit) %*% weigh
> # minimum variance
> weight_s %*% co_var %*% weight_s
> 1/(t(u_nit) %*% in_verse %*% u_nit)
```

Maximum *Sharpe* Portfolio Weights

The *Sharpe* ratio is defined as the ratio of excess returns divided by the portfolio standard deviation:

$$SR = \frac{w^T \mu}{\sigma}$$

Where $\mu = r - r_{rf}$ is the vector of excess returns (returns in excess of the risk-free rate), w is the vector of portfolio weights, and $\sigma = \sqrt{w^T \mathbb{C}\, w}$, where $\mathbb{C}$ is the covariance matrix of returns,

We can calculate the maximum *Sharpe* portfolio weights by setting the derivative of the *Sharpe* ratio with respect to the weights, to zero:

$$d_w SR = \frac{1}{\sigma}(\mu^T - \frac{(w^T \mu)(w^T \mathbb{C})}{\sigma^2}) = 0$$

We then get:

$$(w^T \mathbb{C}\, w)\, \mu = (w^T \mu)\, \mathbb{C} w$$

We can multiply the above equation by $\mathbb{C}^{-1}$ to get:

$$w = \frac{w^T \mathbb{C}\, w}{w^T \mu}\, \mathbb{C}^{-1} \mu$$

We can finally rescale the weights so that they satisfy the constraint $w^T \mathbb{1} = 1$:

$$w = \frac{\mathbb{C}^{-1} \mu}{\mathbb{1}^T \mathbb{C}^{-1} \mu}$$

These are the weights of the maximum *Sharpe* portfolio, with the vector of excess returns equal to μ, and the covariance matrix equal to $\mathbb{C}$,

Returns and Variance of Maximum *Sharpe* Portfolio

The weights of the maximum *Sharpe* portfolio are equal to:

$$w = \frac{\mathbb{C}^{-1}\mu}{\mathbb{1}^T\mathbb{C}^{-1}\mu}$$

Where μ is the vector of excess returns, and $\mathbb{C}$ is the covariance matrix,

The excess returns of the maximum *Sharpe* portfolio are equal to:

$$R = w^T\mu = \frac{\mu^T\mathbb{C}^{-1}\mu}{\mathbb{1}^T\mathbb{C}^{-1}\mu}$$

The variance of the maximum *Sharpe* portfolio is equal to:

$$VAR = \frac{\mu^T\mathbb{C}^{-1}\mathbb{C}\,\mathbb{C}^{-1}\mu}{(\mathbb{1}^T\mathbb{C}^{-1}\mu)^2} = \frac{\mu^T\mathbb{C}^{-1}\mu}{(\mathbb{1}^T\mathbb{C}^{-1}\mu)^2}$$

The *Sharpe* ratio is equal to:

$$SR = \sqrt{\mu^T\mathbb{C}^{-1}\mu}$$

```
> # calculate excess re_turns
> risk_free <- 0.03/252
> ex_cess <- re_turns - risk_free
> # calculate covariance and inverse matrix
> co_var <- cov(re_turns)
> u_nit <- rep(1, NCOL(co_var))
> in_verse <- solve(a=co_var)
> # calculate mean excess returns
> ex_cess <- sapply(ex_cess, mean)
> # weights of maximum Sharpe portfolio
> # weight_s <- solve(a=co_var, b=re_turns)
> weight_s <- in_verse %*% ex_cess
> weight_s <- weight_s/drop(t(u_nit) %*% weight_
> # Sharpe ratios
> sqrt(252)*sum(weight_s * ex_cess) /
+   sqrt(drop(weight_s %*% co_var %*% weight_s))
> sapply(re_turns - risk_free,
+   function(x) sqrt(252)*mean(x)/sd(x))
> weights_maxsharpe <- weight_s
```

Optimal Portfolios Under Zero Correlation

If the correlations of returns are equal to zero, then the covariance matrix is diagonal:

$$\mathbb{C} = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^2 \end{pmatrix}$$

Where σ_i^2 is the variance of returns of asset `i`,

The inverse of $\mathbb{C}$ is then simply:

$$\mathbb{C}^{-1} = \begin{pmatrix} \sigma_1^{-2} & 0 & \cdots & 0 \\ 0 & \sigma_2^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^{-2} \end{pmatrix}$$

The minimum variance portfolio weights are proportional to the inverse of the individual variances:

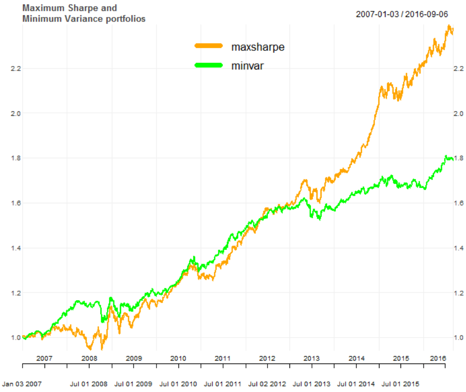
$$w_i = \frac{1}{\sigma_i^2 \sum_{i=1}^n \sigma_i^{-2}}$$

The maximum *Sharpe* portfolio weights are proportional to the ratio of excess returns divided by the individual variances:

$$w_i = \frac{\mu_i}{\sigma_i^2 \sum_{i=1}^n \mu_i \sigma_i^{-2}}$$

Maximum *Sharpe* and *Minimum Variance* Performance

The maximum *Sharpe* and *Minimum Variance* portfolios are both *efficient portfolios*, with the lowest risk (standard deviation) for the given level of return,




```
> library(quantmod)
> # calculate minimum variance weights
> weight_s <- in_verse %*% u_nit
> weights_minvar <-
+   weight_s / drop(t(u_nit) %*% weight_s)
> # calculate optimal portfolio returns
> optim_rets <- xts(
+   x=cbind(exp(cumsum(re_turns %*% weights_maxs
+     exp(cumsum(re_turns %*% weights_minvar))),
+   order.by=index(re_turns))
> colnames(optim_rets) <- c("maxsharpe", "minvar
> # plot optimal portfolio returns, with custom
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "green"
> x11(width=6, height=5)
> chart_Series(optim_rets, theme=plot_theme,
+        name="Maximum Sharpe and \nMinimum Variance portfolios")
> legend("top", legend=colnames(optim_rets), cex=0.8,
+  inset=0.1, bg="white", lty=c(1, 1), lwd=c(6, 6),
+  col=plot_theme$col$line.col, bty="n")
```

The *Efficient Frontier* and *Capital Market Line*

The maximum *Sharpe* portfolio weights depend on the value of the risk-free rate r_{rf},

$$w = \frac{\mathbb{C}^{-1}(r - r_{rf})}{\mathbb{1}^T \mathbb{C}^{-1}(r - r_{rf})}$$

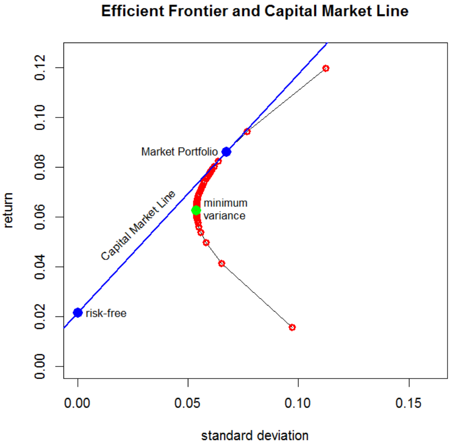
The *Efficient Frontier* is the set of *efficient portfolios*, that have the lowest risk (standard deviation) for the given level of return,

The maximum *Sharpe* portfolios are *efficient portfolios*, and they lie on the *Efficient Frontier*, forming a tangent line from the risk-free rate to the *Efficient Frontier*, known as the *Capital Market Line* (CML),

The maximum *Sharpe* portfolios are considered to be the *Market* portfolios, corresponding to different values of the risk-free rate r_{rf},

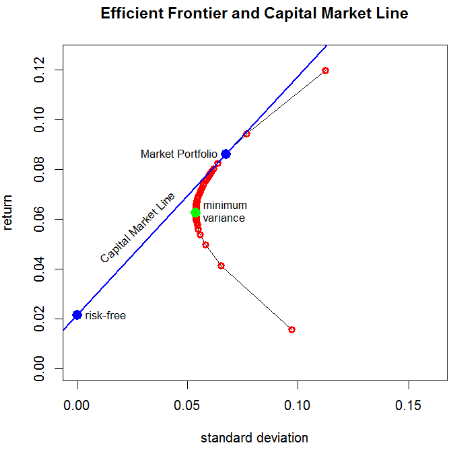
The maximum *Sharpe* portfolios are also called *tangency* portfolios, since they are the tangency point on the *Efficient Frontier*,

The *Capital Market Line* is the line drawn from the *risk-free* rate to the *market* portfolio on the *Efficient Frontier*.



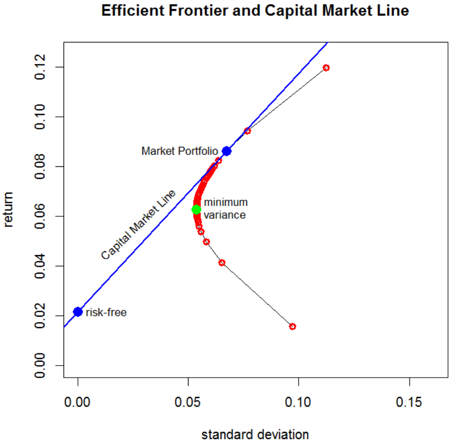
Plotting *Efficient Frontier* and Maximum *Sharpe* Portfolios

```
> # calculate minimum variance weights
> weight_s <- in_verse %*% u_nit
> weight_s <- weight_s / drop(t(u_nit) %*% weigh
> # minimum standard deviation and return
> std_dev <- sqrt(252*drop(weight_s %*% co_var %
> min_ret <- 252*sum(weight_s * mean_rets)
> # calculate maximum Sharpe portfolios
> risk_free <- (min_ret * seq(-10, 10, by=0.1)^3
> eff_front <- sapply(risk_free, function(risk_f
+   weight_s <- in_verse %*% (mean_rets - risk_f
+   weight_s <- weight_s/drop(t(u_nit) %*% weigh
+   # portfolio return and standard deviation
+   c(return=252*sum(weight_s * mean_rets),
+     stddev=sqrt(252*drop(weight_s %*% co_var %
+ })  # end sapply
> eff_front <- cbind(252*risk_free, t(eff_front)
> colnames(eff_front)[1] <- "risk-free"
> eff_front <- eff_front[is.finite(eff_front[, "
> eff_front <- eff_front[order(eff_front[, "retu
> # plot maximum Sharpe portfolios
> plot(x=eff_front[, "stddev"],
+      y=eff_front[, "return"], t="l",
+      xlim=c(0.0*std_dev, 3.0*std_dev),
+      ylim=c(0.0*min_ret, 2.0*min_ret),
+      main="Efficient Frontier and Capital Market Line",
+      xlab="standard deviation", ylab="return")
> points(x=eff_front[, "stddev"], y=eff_front[, "return"],
+  col="red", lwd=3)
```



Plotting the *Capital Market Line*

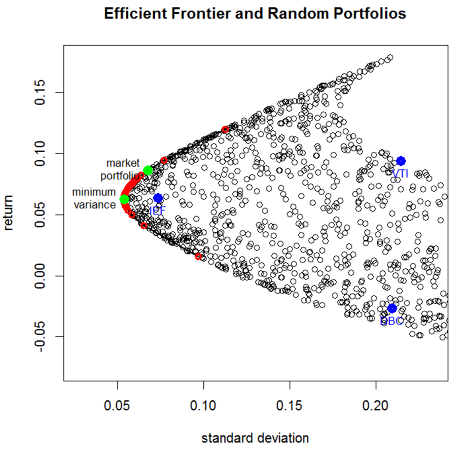
```
> # plot minimum variance portfolio
> points(x=std_dev, y=min_ret, col="green", lwd=
> text(std_dev, min_ret, labels="minimum \nvaria
+      pos=4, cex=0.8)
> # draw Capital Market Line
> sor_ted <- sort(eff_front[, 1])
> risk_free <-
+   sor_ted[findInterval(x=0.5*min_ret, vec=sor_
> points(x=0, y=risk_free, col="blue", lwd=6)
> text(x=0, y=risk_free, labels="risk-free",
+      pos=4, cex=0.8)
> in_dex <- match(risk_free, eff_front[, 1])
> points(x=eff_front[in_dex, "stddev"],
+  y=eff_front[in_dex, "return"],
+  col="blue", lwd=6)
> text(x=eff_front[in_dex, "stddev"],
+      y=eff_front[in_dex, "return"],
+      labels="market portfolio",
+      pos=2, cex=0.8)
> sharp_e <- (eff_front[in_dex, "return"]-risk_f
+   eff_front[in_dex, "stddev"]
> abline(a=risk_free, b=sharp_e, col="blue", lwd
> text(x=0.7*eff_front[in_dex, "stddev"],
+      y=0.7*eff_front[in_dex, "return"]+0.01,
+      labels="Capital Market Line", pos=2, cex=0.8,
+      srt=45*atan(sharp_e*hei_ght/wid_th)/(0.25*pi))
```



The *Capital Market Line* represents delevered and levered portfolios, consisting of the *market* portfolio combined with the *risk-free* rate,

Plotting Random Portfolios

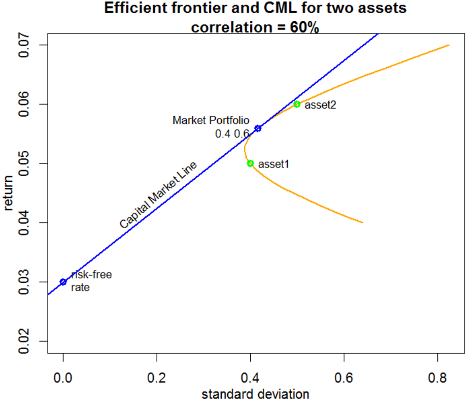
```
> # calculate random portfolios
> n_portf <- 1000
> ret_sd <- sapply(1:n_portf, function(in_dex) {
+   weight_s <- runif(n_weights-1, min=-0.25, ma
+   weight_s <- c(weight_s, 1-sum(weight_s))
+   # portfolio return and standard deviation
+   c(return=252*sum(weight_s * mean_rets),
+     stddev=sqrt(252*drop(weight_s %*% co_var %
+ })  # end sapply
> # plot scatterplot of random portfolios
> x11(wid_th <- 6, hei_ght <- 6)
> plot(x=ret_sd["stddev", ], y=ret_sd["return", ],
+      main="Efficient Frontier and Random Portf
+      xlim=c(0.5*std_dev, 0.8*max(ret_sd["stdde
+      xlab="standard deviation", ylab="return")
> # plot maximum Sharpe portfolios
> lines(x=eff_front[, "stddev"],
+      y=eff_front[, "return"], lwd=2)
> points(x=eff_front[, "stddev"], y=eff_front[,
+  col="red", lwd=3)
> # plot minimum variance portfolio
> points(x=std_dev, y=min_ret, col="green", lwd=
> text(std_dev, min_ret, labels="minimum\nvarian
+      pos=2, cex=0.8)
> # plot market portfolio
> points(x=eff_front[in_dex, "stddev"],
+  y=eff_front[in_dex, "return"], col="green",
> text(x=eff_front[in_dex, "stddev"],
+      y=eff_front[in_dex, "return"],
```




```
> # plot individual assets
> points(x=sqrt(252*diag(co_var)),
+  y=252*mean_rets, col="blue", lwd=6)
> text(x=sqrt(252*diag(co_var)), y=252*mean_rets
+      labels=names(mean_rets),
+      col="blue", pos=1, cex=0.8)
```

Plotting Efficient Frontier for Two-asset Portfolios

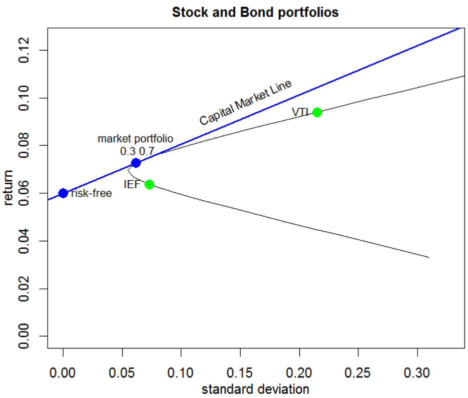
```
> risk_free <- 0.03
> re_turns <- c(asset1=0.05, asset2=0.06)
> std_devs <- c(asset1=0.4, asset2=0.5)
> cor_rel <- 0.6
> co_var <- matrix(c(1, cor_rel, cor_rel, 1), nc
> co_var <- t(t(std_devs*co_var)*std_devs)
> weight_s <- seq(from=-1, to=2, length.out=31)
> weight_s <- cbind(weight_s, 1-weight_s)
> portf_rets <- weight_s %*% re_turns
> portf_sd <-
+   sqrt(rowSums(weight_s * (weight_s %*% co_var
> sharpe_ratios <- (portf_rets-risk_free)/portf_
> in_dex <- which.max(sharpe_ratios)
> max_Sharpe <- max(sharpe_ratios)
> # plot efficient frontier
> x11(wid_th <- 6, hei_ght <- 5)
> par(mar=c(3,3,2,1)+0.1, oma=c(0, 0, 0, 0), mgp
> plot(portf_sd, portf_rets, t="l",
+  main=paste0("Efficient frontier and CML for t
+  xlab="standard deviation", ylab="return",
+  lwd=2, col="orange",
+  xlim=c(0, max(portf_sd)),
+  ylim=c(0.02, max(portf_rets)))
> # add Market Portfolio (maximum Sharpe ratio
> points(portf_sd[in_dex], portf_rets[in_dex],
+  col="blue", lwd=3)
> text(x=portf_sd[in_dex], y=portf_rets[in_dex]
+       labels=paste(c("market portfolio\n",
+  structure(c(weight_s[in_dex], 1-weight_s[in_
```




```
> # plot individual assets
> points(std_devs, re_turns, col="green", lwd=3)
> text(std_devs, re_turns, labels=names(re_turns
> # add point at risk-free rate and draw Capital
> points(x=0, y=risk_free, col="blue", lwd=3)
> text(0, risk_free, labels="risk-free\nrate", p
> abline(a=risk_free, b=max_Sharpe, lwd=2, col="
> range_s <- par("usr")
> text(portf_sd[in_dex]/2, (portf_rets[in_dex]+r
+       labels="Capital Market Line", cex=0.8,
```

Efficient Frontier of Stock and Bond Portfolios

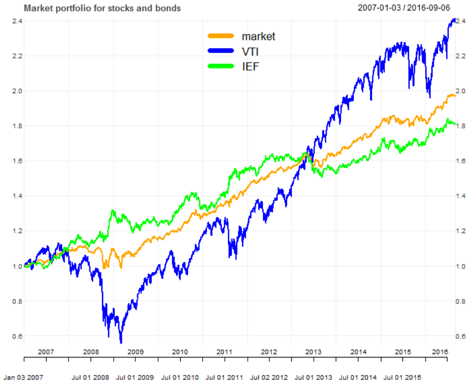
```
> # vector of symbol names
> sym_bols <- c("VTI", "IEF")
> # matrix of portfolio weights
> weight_s <- seq(from=-1, to=2, length.out=31)
> weight_s <- cbind(weight_s, 1-weight_s)
> # calculate portfolio returns and volatilities
> re_turns <- rutils::env_etf$re_turns[, sym_bol
> ret_sd <- re_turns %*% t(weight_s)
> ret_sd <- cbind(252*colMeans(ret_sd),
+   sqrt(252)*matrixStats::colSds(ret_sd))
> colnames(ret_sd) <- c("returns", "stddev")
> risk_free <- 0.06
> ret_sd <- cbind(ret_sd,
+   (ret_sd[, "returns"]-risk_free)/ret_sd[, "st
> colnames(ret_sd)[3] <- "Sharpe"
> in_dex <- which.max(ret_sd[, "Sharpe"])
> max_Sharpe <- ret_sd[in_dex, "Sharpe"]
> plot(x=ret_sd[, "stddev"], y=ret_sd[, "returns
+      main="Stock and Bond portfolios", t="l",
+      xlim=c(0, 0.7*max(ret_sd[, "stddev"])), y
+      xlab="standard deviation", ylab="return")
> # add blue point for market portfolio
> points(x=ret_sd[in_dex, "stddev"], y=ret_sd[in
> text(x=ret_sd[in_dex, "stddev"], y=ret_sd[in_d
+      labels=paste(c("market portfolio\n", stru
+      pos=3, cex=0.8)
```




```
> # plot individual assets
> mean_rets <- 252*sapply(re_turns, mean)
> std_devs <- sqrt(252)*sapply(re_turns, sd)
> points(std_devs, mean_rets, col="green", lwd=6
> text(std_devs, mean_rets, labels=names(re_turn
> # add point at risk-free rate and draw Capital
> points(x=0, y=risk_free, col="blue", lwd=6)
> text(0, risk_free, labels="risk-free", pos=4,
> abline(a=risk_free, b=max_Sharpe, col="blue",
> range_s <- par("usr")
```

Performance of Market Portfolio for Stocks and Bonds

```
> # calculate cumulative returns of VTI and IEF
> optim_rets <- lapply(re_turns,
+   function(re_turns) exp(cumsum(re_turns)))
> optim_rets <- rutils::do_call(cbind, optim_ret
> # calculate market portfolio returns
> optim_rets <- cbind(
+   exp(cumsum(re_turns %*%
+     c(weight_s[in_dex], 1-weight_s[in_dex]))),
+   optim_rets)
> colnames(optim_rets)[1] <- "market"
> # plot market portfolio with custom line color
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue",
> chart_Series(optim_rets, theme=plot_theme,
+        name="Market portfolio for stocks and b
> legend("top", legend=colnames(optim_rets),
+  cex=0.8, inset=0.1, bg="white", lty=c(1, 1),
+  lwd=c(6, 6), col=plot_theme$col$line.col, bty
```

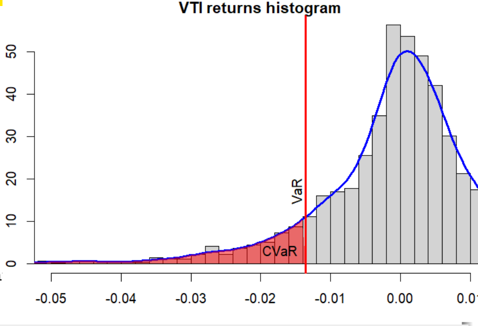


Conditional Value at Risk (*CVaR*)

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

$$\mathrm{CVaR} = \frac{1}{\alpha} \int_0^\alpha \mathrm{VaR}(p) \, \mathrm{d}p$$

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




```
> # VTI percentage returns
> re_turns <- rutils::diff_xts(log(Ad(rutils::en
> conf_level <- 0.1
> va_r <- quantile(re_turns, conf_level)
> c_var <- mean(re_turns[re_turns < va_r])
> # or
> sort_ed <- sort(as.numeric(re_turns))
> in_dex <- round(conf_level*NROW(re_turns))
> va_r <- sort_ed[in_dex]
> c_var <- mean(sort_ed[1:in_dex])
> # plot histogram of VTI returns
> histo_gram <- hist(re_turns, col="lightgrey",
+   xlab="returns", breaks=100, xlim=c(-0.05, 0
+   ylab="frequency", freq=FALSE,
+   main="VTI returns histogram")
> dens_ity <- density(re_turns, adjust=1.5)
> lines(dens_ity, lwd=3, col="blue")
```

```
> # add line for VaR
> abline(v=va_r, col="red", lwd=3)
> text(x=va_r, y=20, labels="VaR",
+      lwd=2, srt=90, pos=2)
> # add shading for CVaR
> var_max <- -0.06
> rang_e <- (dens_ity$x < va_r) & (dens_ity$x >
> polygon(
+   c(var_max, dens_ity$x[rang_e], va_r),
+   c(0, dens_ity$y[rang_e], 0),
+   col=rgb(1, 0, 0,0.5), border=NA)
> text(x=va_r, y=3, labels="CVaR", lwd=2, pos=2)
```

CVaR Portfolio Weights Using Linear Programming

The weights of the minimum *CVaR* portfolio can be calculated using linear programming (*LP*), which is the optimization of linear objective functions subject to linear constraints,

$$w_{min} = \arg\max_{w} [\sum_{i=1}^{n} w_i b_i]$$

Where b_i is the negative objective vector, and w is the vector of returns weights, constrained by:

$$w^T \mathbb{1} = \sum_{i=1}^{n} w_i = 1$$

$$0 \leq w_i \leq 1$$

The function `Rglpk_solve_LP()` from package *Rglpk* solves linear programming problems by calling the *GNU Linear Programming Kit* library,

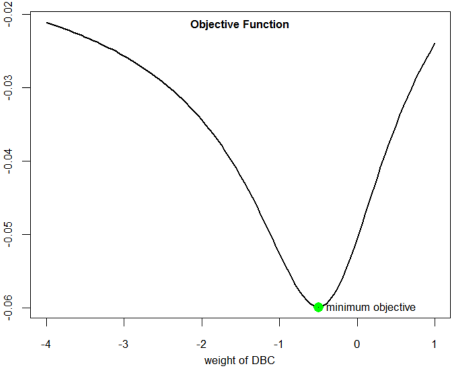
```
> # vector of symbol names and returns
> sym_bols <- c("VTI", "IEF", "DBC")
> n_weights <- NROW(sym_bols)
> re_turns <- rutils::env_etf$re_turns[((NROW(re
> mean_rets <- colMeans(re_turns)
> conf_level <- 0.05
> r_min <- 0 ; w_min <- 0 ; w_max <- 1
> weight_sum <- 1
> n_col <- NCOL(re_turns) # number of assets
> n_row <- NROW(re_turns) # number of rows
> # creat objective vector
> obj_vector <- c(numeric(n_col), rep(-1/(conf_l
> # specify weight constraints
> constraint_s <- rbind(
+   cbind(rbind(1, mean_rets),
+   matrix(data=0, nrow=2, ncol=(n_row+1))),
+   cbind(coredata(re_turns), diag(n_row), 1))
> rh_s <- c(weight_sum, r_min, rep(0, n_row))
> direction_s <- c("==", ">=", rep(">=", n_row))
> # specify weight bounds
> bound_s <- list(
+   lower=list(ind=1:n_col, val=rep(w_min, n_col
+   upper=list(ind=1:n_col, val=rep(w_max, n_col
> # perform optimization
> op_tim <- Rglpk_solve_LP(obj=obj_vector, mat=c
> op_tim$solution
> constraint_s %*% op_tim$solution
> obj_vector %*% op_tim$solution
```

Sharpe Ratio Objective Function

The function optimize() performs *one-dimensional* optimization over a single independent variable,

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

```
> # create initial vector of portfolio weights
> weight_s <- rep(1, NROW(sym_bols))
> names(weight_s) <- sym_bols
> # objective equal to minus Sharpe ratio
> object_ive <- function(weight_s, re_turns) {
+   portf_rets <- re_turns %*% weight_s
+   if (sd(portf_rets) == 0)
+     return(0)
+   else
+     return(-mean(portf_rets)/sd(portf_rets))
+ }  # end object_ive
> # objective for equal weight portfolio
> object_ive(weight_s, re_turns=re_turns)
> op_tim <- unlist(optimize(
+   f=function(weight)
+     object_ive(c(1, 1, weight), re_turns=re_tu
+   interval=c(-4, 1)))
> # vectorize objective function with respect to
> vec_object <- function(weights) sapply(weights
+   function(weight) object_ive(c(1, 1, weight)
+     re_turns=re_turns))
```




```
> # plot objective function with respect to thir
> curve(expr=vec_object,
+       type="l", xlim=c(-4.0, 1.0),
+       xlab=paste("weight of", names(weight_s[3
+       ylab="", lwd=2)
> title(main="Objective Function", line=-1)  # a
> points(x=op_tim[1], y=op_tim[2], col="green",
> text(x=op_tim[1], y=op_tim[2],
+      labels="minimum objective", pos=4, cex=0.
>
```

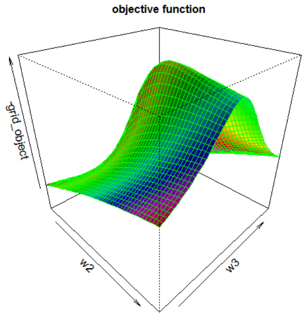
Perspective Plot of Portfolio Objective Function

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

The function outer() calculates the values of a function over a grid spanned by two variables, and returns a matrix of function values,

The package *rgl* allows creating *interactive* 3d scatterplots and surface plots including perspective plots, based on the *OpenGL* framework,

```
> # vectorize function with respect to all weigh
> vec_object <- Vectorize(
+   FUN=function(w1, w2, w3)
+     object_ive(c(w1, w2, w3)),
+   vectorize.args=c("w2", "w3"))  # end Vectori
> # calculate objective on 2-d (w2 x w3) paramet
> w2 <- seq(-3, 7, length=50)
> w3 <- seq(-5, 5, length=50)
> grid_object <- outer(w2, w3, FUN=vec_object, w
> rownames(grid_object) <- round(w2, 2)
> colnames(grid_object) <- round(w3, 2)
> # perspective plot of objective function
> persp(w2, w3, -grid_object,
+ theta=45, phi=30, shade=0.5,
+ col=rainbow(50), border="green",
+ main="objective function")
```




```
> # interactive perspective plot of objective fu
> library(rgl)
> rgl::persp3d(z=-grid_object, zlab="objective",
+   col="green", main="objective function")
> rgl::persp3d(
+   x=function(w2, w3)
+     -vec_object(w1=1, w2, w3),
+   xlim=c(-3, 7), ylim=c(-5, 5),
+   col="green", axes=FALSE)
```

Multi-dimensional Portfolio Optimization

The functional `optim()` performs *multi-dimensional* optimization,

The argument `par` are the initial parameter values,

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

The argument of the objective function which is to be optimized, must be a vector argument,

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

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

`method="L-BFGS-B"` specifies the quasi-Newton optimization method,

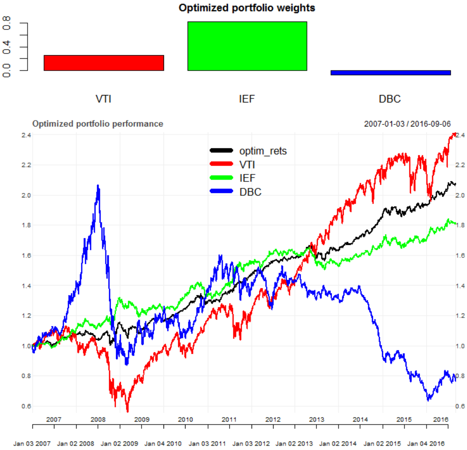
`optim()` returns a list containing the location of the minimum and the objective function value,

```
> # optimization to find weights with maximum Sh
> op_tim <- optim(par=weight_s,
+                 fn=object_ive,
+                 method="L-BFGS-B",
+                 upper=c(1.1, 10, 10),
+                 lower=c(0.9, -10, -10))
> # optimal parameters
> op_tim$par
> op_tim$par <- op_tim$par/sum(op_tim$par)
> # optimal Sharpe ratio
> -object_ive(op_tim$par)
```

Optimized Portfolio Performance

The optimized portfolio has both long and short positions, and outperforms its individual component assets,

```
> # plot in two vertical panels
> layout(matrix(c(1,2), 2),
+  widths=c(1,1), heights=c(1,3))
> # barplot of optimal portfolio weights
> barplot(op_tim$par, col=c("red", "green", "blu
+   main="Optimized portfolio weights")
> # calculate cumulative returns of VTI, IEF, DB
> cum_rets <- lapply(re_turns,
+   function(re_turns) exp(cumsum(re_turns)))
> cum_rets <- rutils::do_call(cbind, cum_rets)
> # calculate optimal portfolio returns with VTI
> optim_rets <- cbind(
+   exp(cumsum(re_turns %*% op_tim$par)),
+   cum_rets)
> colnames(optim_rets)[1] <- "optim_rets"
> # plot optimal returns with VTI, IEF, DBC
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "
> chart_Series(optim_rets, theme=plot_theme,
+         name="Optimized portfolio performance")
> legend("top", legend=colnames(optim_rets), cex=0.8,
+  inset=0.1, bg="white", lty=c(1, 1), lwd=c(6, 6),
+  col=plot_theme$col$line.col, bty="n")
> # or plot non-compounded (simple) cumulative returns
```



Package *quadprog* for Quadratic Programming

Quadratic programming (*QP*) is the optimization of quadratic objective functions subject to linear constraints,

Let $O(x)$ be an objective function that is quadratic with respect to a vector variable x:

$$O(x) = \frac{1}{2} x^T \mathbb{Q} x - d^T x$$

Where $\mathbb{Q}$ is a *positive definite* matrix ($x^T \mathbb{Q} x > 0$), and d is a vector,

An example of a *positive definite* matrix is the covariance matrix of linearly independent variables,

Let the linear constraints on the variable x be specified as:

$$\mathbb{A} x \geq b$$

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

The function `solve.QP()` from package *quadprog* performs optimization of quadratic objective functions subject to linear constraints,

```
> library(quadprog)
> # minimum variance weights without constraints
> op_tim <- solve.QP(Dmat=2*co_var,
+               dvec=rep(0, 2),
+               Amat=matrix(0, nr=2, nc=1),
+               bvec=0)
> # minimum variance weights sum equal to 1
> op_tim <- solve.QP(Dmat=2*co_var,
+               dvec=rep(0, 2),
+               Amat=matrix(1, nr=2, nc=1),
+               bvec=1)
> # optimal value of objective function
> t(op_tim$solution) %*% co_var %*% op_tim$solut
> ## perform simple optimization for reference
> # objective function for simple optimization
> object_ive <- function(x) {
+   x <- c(x, 1-x)
+   t(x) %*% co_var %*% x
+ }  # end object_ive
> unlist(optimize(f=object_ive, interval=c(-1, 2
```

Portfolio Optimization Using Package *quadprog*

The objective function is designed to minimize portfolio variance and maximize its returns:

$$O(x) = w^T \mathbb{C}\, w - w^T r$$

Where $\mathbb{C}$ is the covariance matrix of returns, r is the vector of returns, and w is the vector of portfolio weights,

The portfolio weights w_i are constrained as:

$$w^T \mathbb{1} = \sum_{i=1}^{n} w_i = 1$$

$$0 \leq w_i \leq 1$$

The function `solve.QP()` has the arguments:

`Dmat` and `dvec` are the matrix and vector defining the quadratic objective function,

`Amat` and `bvec` are the matrix and vector defining the constraints,

`meq` specifies the number of equality constraints (the first `meq` constraints are equalities, and the rest are inequalities),

```
> # calculate daily percentage re_turns
> sym_bols <- c("VTI", "IEF", "DBC")
> re_turns <- rutils::env_etf$re_turns[, sym_bol
> # calculate the covariance matrix
> co_var <- cov(re_turns)
> # minimum variance weights, with sum equal to
> op_tim <- quadprog::solve.QP(Dmat=2*co_var,
+               dvec=numeric(3),
+               Amat=matrix(1, nr=3, nc=1),
+               bvec=1)
> # minimum variance, maximum returns
> op_tim <- quadprog::solve.QP(Dmat=2*co_var,
+               dvec=apply(0.1*re_turns, 2, mean),
+               Amat=matrix(1, nr=3, nc=1),
+               bvec=1)
> # minimum variance positive weights, sum equal
> a_mat <- cbind(matrix(1, nr=3, nc=1),
+           diag(3), -diag(3))
> b_vec <- c(1, rep(0, 3), rep(-1, 3))
> op_tim <- quadprog::solve.QP(Dmat=2*co_var,
+               dvec=numeric(3),
+               Amat=a_mat,
+               bvec=b_vec,
+               meq=1)
```

Portfolio Optimization Using Package *Deoptim*

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

```
> # calculate daily percentage re_turns
> re_turns <- rutils::env_etf$re_turns[, sym_bol
> # objective equal to minus Sharpe ratio
> object_ive <- function(weight_s, re_turns) {
+   portf_rets <- re_turns %*% weight_s
+   if (sd(portf_rets) == 0)
+     return(0)
+   else
+     return(-mean(portf_rets)/sd(portf_rets))
+ }  # end object_ive
> # perform optimization using DEoptim
> op_tim <- DEoptim::DEoptim(fn=object_ive,
+   upper=rep(10, NCOL(re_turns)),
+   lower=rep(-10, NCOL(re_turns)),
+   re_turns=re_turns,
+   control=list(trace=FALSE, itermax=100, paral
> weight_s <- op_tim$optim$bestmem/sum(abs(op_ti
> names(weight_s) <- colnames(re_turns)
```

Portfolio Optimization Using *Shrinkage*

The technique of *shrinkage* (*regularization*) is designed to reduce the number of parameters in a model, for example in portfolio optimization,

The *shrinkage* technique adds a penalty term to the objective function,

The *elastic net* regularization is a combination of *ridge* regularization and *Lasso* regularization:

$$w_{max} = \arg\max_{w} [\, w^T r - \lambda((1-\alpha)\sum_{i=1}^{n} w_i^2 + \alpha \sum_{i=1}^{n} |w_i|) \,]$$

The portfolio weights w_i are shrunk to zero as the parameters λ and α increase,

```
> # objective with shrinkage penalty
> object_ive <- function(weight_s, re_turns, lam
+   portf_rets <- re_turns %*% weight_s
+   if (sd(portf_rets) == 0)
+     return(0)
+   else {
+     penal_ty <- lamb_da*((1-al_pha)*sum(weight
+ al_pha*sum(abs(weight_s)))
+     return(-mean(portf_rets)/sd(portf_rets) + 
+   }
+ }  # end object_ive
> # objective for equal weight portfolio
> weight_s <- rep(1, NROW(sym_bols))
> names(weight_s) <- sym_bols
> lamb_da <- 0.5 ; al_pha <- 0.5
> object_ive(weight_s, re_turns=re_turns,
+   lamb_da=lamb_da, al_pha=al_pha)
> # perform optimization using DEoptim
> op_tim <- DEoptim::DEoptim(fn=object_ive,
+   upper=rep(10, NCOL(re_turns)),
+   lower=rep(-10, NCOL(re_turns)),
+   re_turns=re_turns,
+   lamb_da=lamb_da,
+   al_pha=al_pha,
+   control=list(trace=FALSE, itermax=100, paral
> weight_s <-
+   op_tim$optim$bestmem/sum(abs(op_tim$optim$be
> names(weight_s) <- colnames(re_turns)
```

Homework Assignment

Required

Read all the lecture slides in `FRE7241_Lecture_5.pdf`, and run all the code in `FRE7241_Lecture_5.R`

Recommended

- Read about *optimization methods*:
  Bolker Optimization Methods.pdf
  Yollin Optimization.pdf
  Boudt DEoptim Large Portfolio Optimization.pdf
- Read about *PCA* in:
  pca-handout.pdf
  pcaTutorial.pdf