FRE7241 Algorithmic Portfolio Management

Lecture#7, Spring 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 30, 2024

Geometric Brownian Motion

If the percentage asset returns $r_t \mathrm{d}t = \mathrm{d} \log p_t$ follow *Brownian motion*:

$$r_t \mathrm{d}t = \mathrm{d} \log p_t = (\mu - \frac{\sigma^2}{2})\mathrm{d}t + \sigma \, \mathrm{d}W_t$$

Then asset prices p_t follow *Geometric Brownian motion* (GBM):

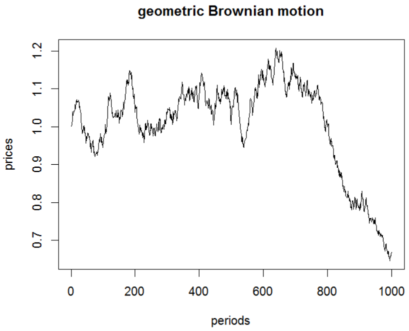
$$\mathrm{d}p_t = \mu p_t \mathrm{d}t + \sigma \, p_t \mathrm{d}W_t$$

Where σ is the volatility of asset returns, and W_t is a *Brownian Motion*, with $\mathrm{d}W_t$ following the standard normal distribution $\phi(0, \sqrt{\mathrm{d}t})$.

The solution of *Geometric Brownian motion* is equal to:

$$p_t = p_0 \exp[(\mu - \frac{\sigma^2}{2})t + \sigma \, W_t]$$

The convexity correction: $-\frac{\sigma^2}{2}$ ensures that the growth rate of prices is equal to μ, (according to Ito's lemma).



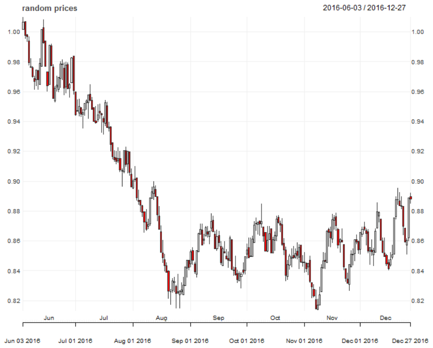
```
> # Define daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 1000
> # Simulate geometric Brownian motion
> retp <- sigmav*rnorm(nrows) + drift - sigmav^2/2
> pricev <- exp(cumsum(retp))
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="geometric Brownian motion")
```

Simulating Random *OHLC* Prices

Random *OHLC* prices are useful for testing financial models.

The function `sample()` selects a random sample from a vector of data elements.
The function `sample()` with `replace=TRUE` selects samples with replacement (the default is `replace=FALSE`).

```
> # Simulate geometric Brownian motion
> sigmav <- 0.01/sqrt(48)
> drift <- 0.0
> nrows <- 1e4
> datev <- seq(from=as.POSIXct(paste(Sys.Date()-250, "09:30:00")),
+   length.out=nrows, by="30 min")
> pricev <- exp(cumsum(sigmav*rnorm(nrows) + drift - sigmav^2/2))
> pricev <- xts(pricev, order.by=datev)
> pricev <- cbind(pricev,
+   volume=sample(x=10*(2:18), size=nrows, replace=TRUE))
> # Aggregate to daily OHLC data
> ohlc <- xts::to.daily(pricev)
> quantmod::chart_Series(ohlc, name="random prices")
> # dygraphs candlestick plot using pipes syntax
> library(dygraphs)
> dygraphs::dygraph(ohlc[, 1:4]) %>% dyCandlestick()
> # dygraphs candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dygraph(ohlc[, 1:4]))
```



The *Log-normal* Probability Distribution

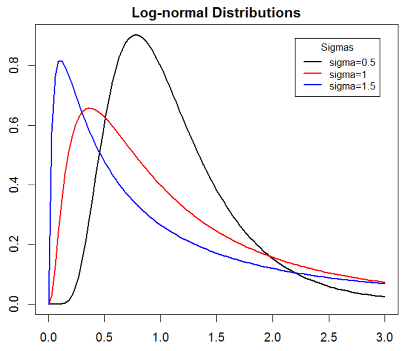
If x follows the *Normal* distribution $\phi(x, \mu, \sigma)$, then the exponential of x: $y = e^x$ follows the *Log-normal* distribution $\log \phi()$:

$$\log \phi(y, \mu, \sigma) = \frac{\exp(-(\log y - \mu)^2/2\sigma^2)}{y\sigma\sqrt{2\pi}}$$

With mean equal to: $\bar{y} = \mathbb{E}[y] = e^{(\mu+\sigma^2/2)}$, and median equal to: $\tilde{y} = e^{\mu}$

With variance equal to: $\sigma_y^2 = (e^{\sigma^2} - 1)e^{(2\mu+\sigma^2)}$, and skewness (third moment) equal to:
$\varsigma = \mathbb{E}[(y - \mathbb{E}[y])^3] = (e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$

```
> # Standard deviations of log-normal distribution
> sigmavs <- c(0.5, 1, 1.5)
> # Create plot colors
> colorv <- c("black", "red", "blue")
> # Plot all curves
> for (indeks in 1:NROW(sigmavs)) {
+   curve(expr=dlnorm(x, sdlog=sigmavs[indeks]),
+   type="l", lwd=2, xlim=c(0, 3),
+   xlab="", ylab="", col=colorv[indeks],
+   add=as.logical(indeks-1))
+ }  # end for
```



```
> # Add title and legend
> title(main="Log-normal Distributions", line=0.5)
> legend("topright", inset=0.05, title="Sigmas",
+  paste("sigma", sigmavs, sep="="),
+  cex=0.8, lwd=2, lty=rep(1, NROW(sigmavs)),
+  col=colorv)
```

The Standard Deviation of *Log-normal* Prices

If percentage asset returns are *normally* distributed and follow *Brownian motion*, then asset prices follow *Geometric Brownian motion*, and they are *Log-normally* distributed at every point in time.

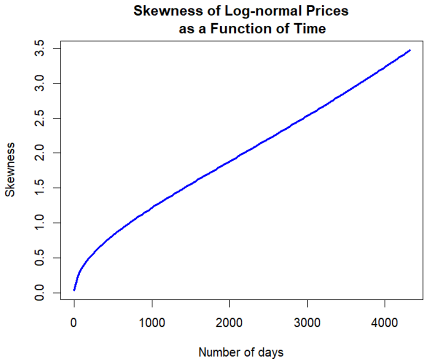
The standard deviation of *log-normal* prices is equal to the return volatility σ_r times the square root of time: $\sigma = \sigma_r \sqrt{t}$.

The *Log-normal* distribution has a strong positive skewness (third moment) equal to:

$$\varsigma = \mathbb{E}[(y - \mathbb{E}[y])^3] = (e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$$

For large standard deviation, the skewness increases exponentially with the standard deviation and with time: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

```
> # Return volatility of VTI etf
> sigmav <- sd(rutils::diffit(log(rutils::etfenv$VTI[, 4])))
> sigma2 <- sigmav^2
> nrows <- NROW(rutils::etfenv$VTI)
> # Standard deviation of log-normal prices
> sqrt(nrows)*sigmav
```



```
> # Skewness of log-normal prices
> calcskew <- function(t) {
+   expv <- exp(t*sigma2)
+   (expv + 2)*sqrt(expv - 1)
+ }  # end calcskew
> curve(expr=calcskew, xlim=c(1, nrows), lwd=3,
+ xlab="Number of days", ylab="Skewness", col="blue",
+ main="Skewness of Log-normal Prices
+ as a Function of Time")
```

The Mean and Median of *Log-normal* Prices

The mean of the *Log-normal* distribution:
$\bar{y} = \mathbb{E}[y] = \exp(\mu + \sigma^2/2)$ is greater than its median, which is equal to: $\tilde{y} = \exp(\mu)$.

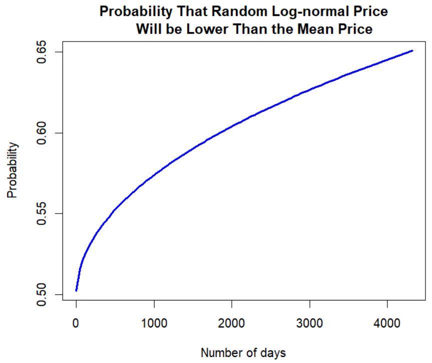
So if stock prices follow *Geometric Brownian motion* and are distributed *log-normally*, then a stock selected at random will have a high probability of havng a lower price than the mean expected price.

The cumulative *Log-normal* probability distribution is equal to $\mathrm{F}(x) = \Phi(\frac{\log y - \mu}{\sigma})$, where $\Phi()$ is the cumulative standard normal distribution.

So the probability that the price of a randomly selected stock will be lower than the mean price is equal to $\mathrm{F}(\bar{y}) = \Phi(\sigma/2)$.

Therefore an investor without skill, who selects stocks at random, has a high probability of underperforming the index.

Performing as well as the index requires *significant* investment skill, while outperforming the index requires *exceptional* investment skill.



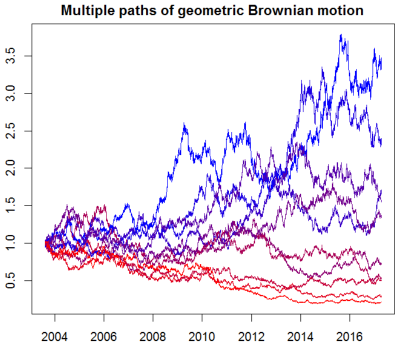
```
> # Probability that random log-normal price will be lower than the
> curve(expr=pnorm(sigmav*sqrt(x)/2),
+ xlim=c(1, nrows), lwd=3,
+ xlab="Number of days", ylab="Probability", col="blue",
+ main="Probability That Random Log-normal Price
+ Will be Lower Than the Mean Price")
```

Paths of Geometric Brownian Motion

The standard deviation of *log-normal* prices σ is equal to the volatility of returns σ_r times the square root of time: $\sigma = \sigma_r \sqrt{t}$.

For large standard deviation, the skewness ς increases exponentially with the standard deviation and with time: $\varsigma \propto e^{1.5 \sigma^2} = e^{1.5 t \sigma_r^2}$

```
> # Define daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 5000
> npaths <- 10
> # Simulate multiple paths of geometric Brownian motion
> pricev <- rnorm(npaths*nrows, sd=sigmav) + drift - sigmav^2/2
> pricev <- matrix(pricev, nc=npaths)
> pricev <- exp(matrixStats::colCumsums(pricev))
> # Create xts time series
> pricev <- xts(pricev, order.by=seq.Date(Sys.Date()-nrows+1, Sys.Da
> # Sort the columns according to largest terminal values
> pricev <- pricev[, order(pricev[nrows, ])]
> # Plot xts time series
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(pricev))
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(pricev, main="Multiple paths of geometric Brownian motion",
+    xlab=NA, ylab=NA, plot.type="single", col=colorv)
```

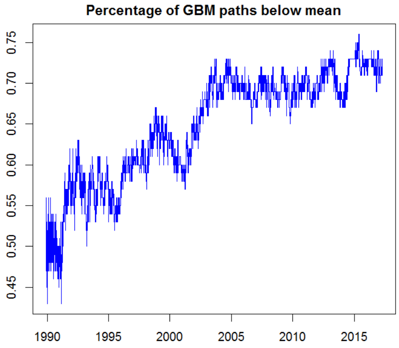


Distribution of Paths of Geometric Brownian Motion

Prices following *Geometric Brownian motion* have a large positive skewness, so that the expected value of prices is skewed by a few paths with very high prices, while the prices of the majority of paths are below their expected value.

For large standard deviation, the skewness ς increases exponentially with the standard deviation and with time: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

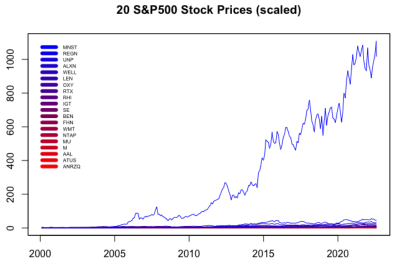
```
> # Define daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 10000
> npaths <- 100
> # Simulate multiple paths of geometric Brownian motion
> pricev <- rnorm(npaths*nrows, sd=sigmav) + drift - sigmav^2/2
> pricev <- matrix(pricev, nc=npaths)
> pricev <- exp(matrixStats::colCumsums(pricev))
> # Calculate fraction of paths below the expected value
> fractv <- rowSums(pricev < 1.0) / npaths
> # Create xts time series of percentage of paths below the expecte
> fractv <- xts(fractv, order.by=seq.Date(Sys.Date()-NROW(fractv)+1, Sys.Date(), by=1))
> # Plot xts time series of percentage of paths below the expected value
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(fractv, main="Percentage of GBM paths below mean",
+    xlab=NA, ylab=NA, col="blue")
```



Time Evolution of Stock Prices

Stock prices evolve over time similar to *Geometric Brownian motion*, and they also exhibit a very skewed distribution of prices.

```
> # Load S&P500 stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> ls(sp500env)
> # Extract the closing prices
> pricev <- eapply(sp500env, quantmod::Cl)
> # Flatten the prices into a single xts series
> pricev <- rutils::do_call(cbind, pricev)
> # Carry forward and backward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> pricev <- zoo::na.locf(pricev, fromLast=TRUE)
> sum(is.na(pricev))
> # Drop ".Close" from column names
> colnames(pricev)
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> # Select prices after the year 2000
> pricev <- pricev["2000/", ]
> # Scale the columns so that prices start at 1
> pricev <- lapply(pricev, function(x) x/as.numeric(x[1]))
> pricev <- rutils::do_call(cbind, pricev)
> # Sort the columns according to the final prices
> nrows <- NROW(pricev)
> ordern <- order(pricev[nrows, ])
> pricev <- pricev[, ordern]
> # Select 20 symbols
> symbolv <- colnames(pricev)
> symbolv <- symbolv[round(seq.int(from=1, to=NROW(symbolv), length.out=20))]
```



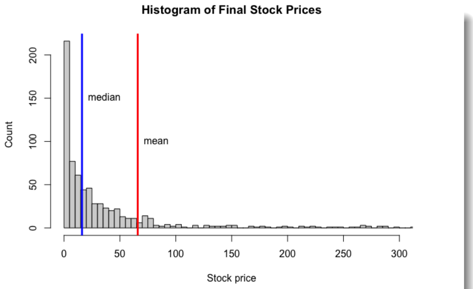
```
> # Plot xts time series of prices
> colorv <- colorRampPalette(c("red", "blue"))(NROW(symbolv))
> endd <- rutils::calc_endpoints(pricev, interval="weeks")
> plot.zoo(pricev[endd, symbolv], main="20 S&P500 Stock Prices (sca
+    xlab=NA, ylab=NA, plot.type="single", col=colorv)
> legend(x="topleft", inset=0.02, cex=0.5, bty="n", y.intersp=0.5,
+  legend=rev(symbolv), col=rev(colorv), lwd=6, lty=1)
```

Distribution of Final Stock Prices

The distribution of the final stock prices is extremely skewed, with over 80% of the *S&P500* constituent stocks from 1990 now below the average price of that portfolio.

The *mean* of the final stock prices is much greater than the *median*.

```
> # Calculate the final stock prices
> pricef <- drop(zoo::coredata(pricev[nrows, ]))
> # Calculate the mean and median stock prices
> max(pricef); min(pricef)
> which.max(pricef)
> which.min(pricef)
> mean(pricef)
> median(pricef)
> # Calculate the percentage of stock prices below the mean
> sum(pricef < mean(pricef))/NROW(pricef)
```



```
> # Plot a histogram of final stock prices
> hist(pricef, breaks=1e3, xlim=c(0, 300),
+      xlab="Stock price", ylab="Count",
+      main="Histogram of Final Stock Prices")
> # Plot a histogram of final stock prices
> abline(v=median(pricef), lwd=3, col="blue")
> text(x=median(pricef), y=150, lab="median", pos=4)
> abline(v=mean(pricef), lwd=3, col="red")
> text(x=mean(pricef), y=100, lab="mean", pos=4)
```

Distribution of Stock Prices Over Time

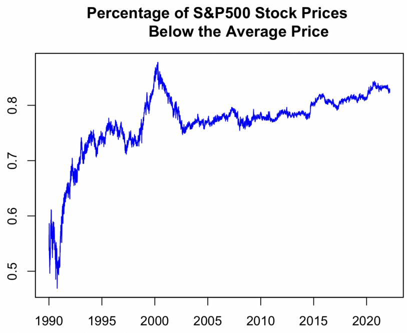
Usually, a small number of stocks in an index reach very high prices, while the prices of the majority of stocks remain below the index price (the average price of the index portfolio).

For example, the current prices of over 80% of the *S&P500* constituent stocks from 1990 are now below the average price of that portfolio.

Therefore an investor without skill, who selects stocks at random, has a high probability of underperforming the index, because they will most likely miss selecting the best performing stocks.

Performing as well as the index requires *significant* investment skill, while outperforming the index requires *exceptional* investment skill.

```
> # Calculate average of valid stock prices
> validp <- (pricev != 1)  # Valid stocks
> nstocks <- rowSums(validp)
> nstocks[1] <- NCOL(pricev)
> indeks <- rowSums(pricev*validp)/nstocks
> # Calculate fraction of stock prices below the average price
> fractv <- rowSums((pricev < indeks) & validp)/nstocks
> # Create xts time series of average stock prices
> indeks <- xts(indeks, order.by=zoo::index(pricev))
```



```
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # x11(width=6, height=4)
> # Plot xts time series of average stock prices
> plot.zoo(indeks, main="Average S&P500 Stock Prices (normalized fro
+    xlab=NA, ylab=NA, col="blue")
> # Create xts time series of percentage of stock prices below the a
> fractv <- xts(fractv, order.by=zoo::index(pricev))
> # Plot percentage of stock prices below the average price
> plot.zoo(fractv[-(1:2),],
+    main="Percentage of S&P500 Stock Prices
+    Below the Average Price",
+    xlab=NA, ylab=NA, col="blue")
```

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:

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

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

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

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

The *Minimum Variance* Portfolio

The portfolio variance is equal to: $\mathbf{w}^T \mathbb{C}\, \mathbf{w}$, where $\mathbb{C}$ is the covariance matrix of returns.

If the portfolio weights $\mathbf{w}$ are subject to *linear* constraints: $\mathbf{w}^T \mathbb{1} = \sum_{i=1}^n w_i = 1$, then the weights that minimize the portfolio variance can be found by minimizing the *Lagrangian*:

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

Where λ is a *Lagrange multiplier*.

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

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

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

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

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

By setting the derivative to zero we find $\mathbf{w}$ equal to:

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

By multiplying the above from the left by $\mathbb{1}^T$, and using $\mathbf{w}^T \mathbb{1} = 1$, 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:

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

If the portfolio weights are subject to *quadratic* constraints: $\mathbf{w}^T \mathbf{w} = 1$ then the minimum variance weights are equal to the highest order *principal component* (with the smallest eigenvalue) of the covariance matrix $\mathbb{C}$.

Returns and Variance of the *Minimum Variance* Portfolio

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

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

The daily returns of the *minimum variance* portfolio are equal to:

$$\mathbf{r}_{mv} = \frac{\mathbf{r}^T \mathbb{C}^{-1} \mathbb{1}}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}} = \frac{\mathbf{r}^T \mathbb{C}^{-1} \mathbb{1}}{c_{11}}$$

Where $\mathbf{r}$ are the daily stock returns, and $c_{11} = \mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}$.

The variance of the *minimum variance* portfolio is equal to:

$$\sigma^2_{mv} = \mathbf{w}^T \mathbb{C} \, \mathbf{w} = \frac{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{C} \, \mathbb{C}^{-1} \mathbb{1}}{(\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1})^2} = \frac{1}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}} = \frac{1}{c_{11}}$$

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.

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

```
> # Calculate daily stock returns
> symbolv <- c("VTI", "IEF", "DBC")
> nstocks <- NROW(symbolv)
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> # Calculate covariance matrix of returns and its inverse
> covmat <- cov(retp)
> covinv <- solve(a=covmat)
> unitv <- rep(1, nstocks)
> # Calculate the minimum variance weights
> c11 <- drop(t(unitv) %*% covinv %*% unitv)
> weightmv <- drop(covinv %*% unitv/c11)
> # Calculate the daily minvar portfolio returns in two ways
> retmv <- (retp %*% weightmv)
> all.equal(retmv, (retp %*% covinv %*% unitv)/c11)
> # Calculate the minimum variance in three ways
> all.equal(var(retmv),
+   t(weightmv) %*% covmat %*% weightmv,
+   1/(t(unitv) %*% covinv %*% unitv))
```

The *Efficient Portfolios*

A portfolio which has the smallest variance, given a target return, is an *efficient portfolio*.

The efficient portfolio weights have two constraints: the sum of portfolio weights $\mathbf{w}$ is equal to 1: $\mathbf{w}^T \mathbb{1} = \sum_{i=1}^n w_i = 1$, and the mean portfolio return is equal to the target return r_t: $\mathbf{w}^T \bar{\mathbf{r}} = \sum_{i=1}^n w_i \bar{r}_i = r_t$.

Where $\bar{\mathbf{r}}$ are the mean stock returns.

The stock weights that minimize the portfolio variance under these constraints can be found by minimizing the *Lagrangian*:

$$\mathcal{L} = \mathbf{w}^T \mathbb{C}\, \mathbf{w} - \lambda_1 (\mathbf{w}^T \mathbb{1} - 1) - \lambda_2 (\mathbf{w}^T \mathbf{r} - r_t)$$

Where λ_1 and λ_2 are the *Lagrange multipliers*.

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

$$d_w \mathcal{L} = 2\mathbf{w}^T \mathbb{C} - \lambda_1 \mathbb{1}^T - \lambda_2 \bar{\mathbf{r}}^T$$

By setting the derivative to zero we obtain the efficient portfolio weights $\mathbf{w}$:

$$\mathbf{w} = \frac{1}{2}(\lambda_1 \mathbb{C}^{-1} \mathbb{1} + \lambda_2 \mathbb{C}^{-1} \bar{\mathbf{r}})$$

By multiplying the above from the left first by $\mathbb{1}^T$, and then by $\bar{\mathbf{r}}^T$, we obtain a system of two equations for λ_1 and λ_2:

$$2\mathbb{1}^T \mathbf{w} = \lambda_1 \mathbb{1}^T \mathbb{C}^{-1} \mathbb{1} + \lambda_2 \mathbb{1}^T \mathbb{C}^{-1} \bar{\mathbf{r}} = 2$$

$$2\bar{\mathbf{r}}^T \mathbf{w} = \lambda_1 \bar{\mathbf{r}}^T \mathbb{C}^{-1} \mathbb{1} + \lambda_2 \bar{\mathbf{r}}^T \mathbb{C}^{-1} \bar{\mathbf{r}} = 2r_t$$

The above can be written in matrix notation as:

$$\begin{bmatrix} \mathbb{1}^T \mathbb{C}^{-1} \mathbb{1} & \mathbb{1}^T \mathbb{C}^{-1} \bar{\mathbf{r}} \\ \bar{\mathbf{r}}^T \mathbb{C}^{-1} \mathbb{1} & \bar{\mathbf{r}}^T \mathbb{C}^{-1} \bar{\mathbf{r}} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2r_t \end{bmatrix}$$

Or:

$$\begin{bmatrix} c_{11} & c_{r1} \\ c_{r1} & c_{rr} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \mathbb{F}\lambda = 2 \begin{bmatrix} 1 \\ r_t \end{bmatrix} = 2u$$

With $c_{11} = \mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}$, $c_{r1} = \mathbb{1}^T \mathbb{C}^{-1} \bar{\mathbf{r}}$, $c_{rr} = \bar{\mathbf{r}}^T \mathbb{C}^{-1} \bar{\mathbf{r}}$, $\lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}$, $u = \begin{bmatrix} 1 \\ r_t \end{bmatrix}$, and $\mathbb{F} = u^T \mathbb{C}^{-1} u = \begin{bmatrix} c_{11} & c_{r1} \\ c_{r1} & c_{rr} \end{bmatrix}$.

The *Lagrange multipliers* can be solved as:

$$\lambda = 2\mathbb{F}^{-1} u$$

The *Efficient Portfolio* Weights

The efficient portfolio weights $\mathbf{w}$ can now be solved as:

$$\mathbf{w} = \frac{1}{2}(\lambda_1 \, \mathbb{C}^{-1}\mathbb{1} + \lambda_2 \, \mathbb{C}^{-1}\bar{\mathbf{r}}) =$$

$$\frac{1}{2}\begin{bmatrix} \mathbb{C}^{-1}\mathbb{1} \\ \mathbb{C}^{-1}\bar{\mathbf{r}} \end{bmatrix}^T \lambda = \begin{bmatrix} \mathbb{C}^{-1}\mathbb{1} \\ \mathbb{C}^{-1}\bar{\mathbf{r}} \end{bmatrix}^T \mathbb{F}^{-1} \, u =$$

$$\frac{1}{\det \mathbb{F}}\begin{bmatrix} \mathbb{C}^{-1}\mathbb{1} \\ \mathbb{C}^{-1}\bar{\mathbf{r}} \end{bmatrix}^T \begin{bmatrix} c_{rr} & -c_{r1} \\ -c_{r1} & c_{11} \end{bmatrix} \begin{bmatrix} 1 \\ r_t \end{bmatrix} =$$

$$\frac{(c_{rr} - c_{r1} \, r_t) \, \mathbb{C}^{-1}\mathbb{1} + (c_{11} \, r_t - c_{r1}) \, \mathbb{C}^{-1}\bar{\mathbf{r}}}{\det \mathbb{F}}$$

With $c_{11} = \mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}$, $c_{r1} = \mathbb{1}^T \mathbb{C}^{-1} \bar{\mathbf{r}}$, $c_{rr} = \bar{\mathbf{r}}^T \mathbb{C}^{-1} \bar{\mathbf{r}}$.
And $\det \mathbb{F} = c_{11} c_{rr} - c_{r1}^2$ is the determinant of the matrix $\mathbb{F}$.

The above formula shows that the efficient portfolio weights are a linear function of the target return.

Therefore a convex sum of two efficient portfolio weights: $w = \alpha w_1 + (1 - \alpha) w_2$, are also the weights of an *efficient portfolio*, with target return equal to: $r_t = \alpha r_1 + (1 - \alpha) r_2$

```
> # Calculate vector of mean returns
> retm <- colMeans(retp)
> # Specify the target return
> retarget <- 1.5*mean(retp)
> # Products of inverse with mean returns and unit vector
> c11 <- drop(t(unitv) %*% covinv %*% unitv)
> cr1 <- drop(t(unitv) %*% covinv %*% retm)
> crr <- drop(t(retm) %*% covinv %*% retm)
> fmat <- matrix(c(c11, cr1, cr1, crr), nc=2)
> # Solve for the Lagrange multipliers
> lagm <- solve(a=fmat, b=c(2, 2*retarget))
> # Calculate the efficient portfolio weights
> weightv <- 0.5*drop(covinv %*% cbind(unitv, retm) %*% lagm)
> # Calculate constraints
> all.equal(1, sum(weightv))
> all.equal(retarget, sum(retm*weightv))
```

Variance of the *Efficient Portfolios*

The *efficient portfolio* variance is equal to:

$$\sigma^2 = \mathbf{w}^T \mathbb{C} \, \mathbf{w} = \frac{1}{4} \lambda^T \mathbb{F} \, \lambda = u^T \mathbb{F}^{-1} \, u = \frac{1}{\det \mathbb{F}} \begin{bmatrix} 1 \\ r_t \end{bmatrix}^T \begin{bmatrix} c_{rr} & -c_{r1} \\ -c_{r1} & c_{11} \end{bmatrix} \begin{bmatrix} 1 \\ r_t \end{bmatrix} = \frac{c_{11} r_t^2 - 2 c_{r1} r_t + c_{rr}}{\det \mathbb{F}}$$

The above formula shows that the variance of the *efficient portfolios* is a *parabola* with respect to the target return r_t.

The vertex of the *parabola* is the minimum variance portfolio: $r_{mv} = c_{r1}/c_{11} = \mathbb{1}^T \mathbb{C}^{-1} \bar{\mathbf{r}} / \mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}$ and $\sigma^2_{mv} = 1/c_{11} = 1/\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}$.

The *efficient portfolio* variance can be expressed in terms of the difference $\Delta_r = r_t - r_{mv}$ as:

$$\sigma^2 = \frac{\Delta_r^2 + \det \mathbb{F}}{c_{11} \det \mathbb{F}}$$

So that if $\Delta_r = 0$ then $\sigma^2 = 1/c_{11}$.

Where $\det \mathbb{F} = c_{11} c_{rr} - c_{r1}^2$ is the determinant of the matrix $\mathbb{F}$.

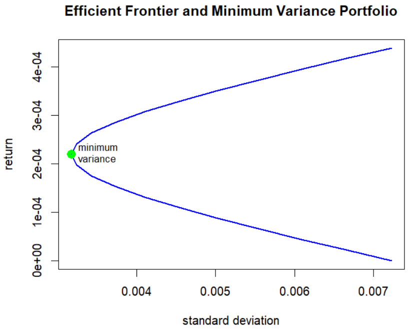
```
> # Calculate the efficient portfolio returns
> reteff <- drop(retp %*% weightv)
> reteffm <- mean(reteff)
> all.equal(reteffm, retarget)
> # Calculate the efficient portfolio variance in three ways
> uu <- c(1, retarget)
> finv <- solve(fmat)
> detf <- (c11*crr-cr1^2)  # det(fmat)
> all.equal(var(reteff),
+   drop(t(uu) %*% finv %*% uu),
+   (c11*reteffm^2-2*cr1*reteffm+crr)/detf)
```

The *Efficient Frontier*

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

The *efficient frontier* is the plot of the target returns r_t and the standard deviations of the *efficient portfolios*, which is a *hyperbola*.

```
> # Calculate the daily and mean minvar portfolio returns
> c11 <- drop(t(unitv) %*% covinv %*% unitv)
> weightv <- drop(covinv %*% unitv/c11)
> retmv <- (retp %*% weightv)
> retmvm <- sum(weightv*retm)
> # Calculate the minimum variance
> varmv <- 1/c11
> stdevmv <- sqrt(varmv)
> # Calculate efficient frontier from target returns
> targetv <- retmvm*(1+seq(from=(-1), to=1, by=0.1))
> stdevs <- sapply(targetv, function(rett) {
+   uu <- c(1, rett)
+   sqrt(drop(t(uu) %*% finv %*% uu))
+ })  # end sapply
```



```
> # Plot the efficient frontier
> plot(x=stdevs, y=targetv, t="l", col="blue", lwd=2,
+      main="Efficient Frontier and Minimum Variance Portfolio",
+      xlab="standard deviation", ylab="return")
> points(x=stdevmv, y=retmvm, col="green", lwd=6)
> text(x=stdevmv, y=retmvm, labels="minimum \nvariance",
+      pos=4, cex=0.8)
```

The *Tangent Line* and the *Risk-free* Rate

The *tangent* line connects the risk-free point ($\sigma = 0, r = r_f$) with a single tangent point on the *efficient frontier*.

A *tangent* line can be drawn at every point on the *efficient frontier*.

The slope β of the *tangent* line can be calculated by differentiating the efficient portfolio variance σ^2 by the target return r_t:

$$\frac{d\sigma^2}{dr_t} = 2\sigma \frac{d\sigma}{dr_t} = \frac{2c_{11} r_t - 2c_{r1}}{\det \mathbb{F}}$$

$$\frac{d\sigma}{dr_t} = \frac{c_{11} r_t - c_{r1}}{\sigma \det \mathbb{F}}$$

$$\beta = \frac{\sigma \det \mathbb{F}}{c_{11} r_t - c_{r1}}$$

The *tangent* line connects the *tangent* point on the *efficient frontier* with a *risk-free* rate r_f.

The *risk-free* rate r_f can be calculated as the intercept of the tangent line:

$$r_f = r_t - \sigma\,\beta = r_t - \frac{\sigma^2 \det \mathbb{F}}{c_{11} r_t - c_{r1}} =$$

$$r_t - \frac{c_{11} r_t^2 - 2c_{r1} r_t + c_{rr}}{\det \mathbb{F}} \frac{\det \mathbb{F}}{c_{11} r_t - c_{r1}} =$$

$$r_t - \frac{c_{11} r_t^2 - 2c_{r1} r_t + c_{rr}}{c_{11} r_t - c_{r1}} = \frac{c_{r1} r_t - c_{rr}}{c_{11} r_t - c_{r1}}$$

```
> # Calculate standard deviation of efficient portfolio
> uu <- c(1, retarget)
> stdeveff <- sqrt(drop(t(uu) %*% finv %*% uu))
> # Calculate the slope of the tangent line
> detf <- (c11*crr-cr1^2)  # det(fmat)
> sharper <- (stdeveff*detf)/(c11*retarget-cr1)
> # Calculate the risk-free rate as intercept of the tangent line
> riskf <- retarget - sharper*stdeveff
> # Calculate the risk-free rate from target return
> all.equal(riskf,
+   (retarget*cr1-crr)/(retarget*c11-cr1))
```

The *Capital Market Line*

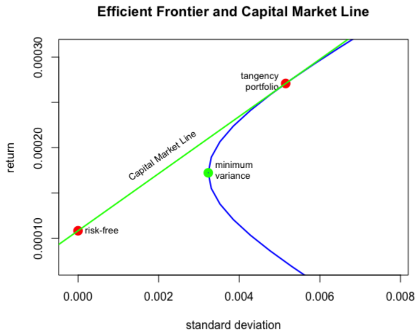
The *Capital Market Line* (CML) is the tangent line connecting the risk-free point ($\sigma = 0, r = r_f$) with a single tangent point on the *efficient frontier*.

The *tangency portfolio* is the *efficient portfolio* at the tangent point corresponding to the given *risk-free* rate.

Each value of the *risk-free* rate r_f corresponds to a unique *tangency portfolio*.

For a given *risk-free* rate r_f, the *tangency portfolio* has the highest *Sharpe ratio* among all the *efficient portfolios*.

```
> # Plot efficient frontier
> aspratio <- 1.0*max(stdevs)/diff(range(targetv))
> plot(x=stdevs, y=targetv, t="l", col="blue", lwd=2, asp=aspratio,
+      xlim=c(0.4, 0.6)*max(stdevs), ylim=c(0.2, 0.9)*max(targetv),
+      main="Efficient Frontier and Capital Market Line",
+      xlab="standard deviation", ylab="return")
> # Plot the minimum variance portfolio
> points(x=stdevmv, y=retmvm, col="green", lwd=6)
> text(x=stdevmv, y=retmvm, labels="minimum \nvariance",
+      pos=4, cex=0.8)
```



```
> # Plot the tangent portfolio
> points(x=stdeveff, y=retarget, col="red", lwd=6)
> text(x=stdeveff, y=retarget, labels="tangency\nportfolio", pos=2,
> # Plot the risk-free point
> points(x=0, y=riskf, col="red", lwd=6)
> text(x=0, y=riskf, labels="risk-free", pos=4, cex=0.8)
> # Plot the tangent line
> abline(a=riskf, b=sharper, lwd=2, col="green")
> text(x=0.6*stdev, y=0.8*retarget,
+      labels="Capital Market Line", pos=2, cex=0.8,
+      srt=180/pi*atan(aspratio*sharper))
```

The *Capital Market Line* Portfolios

The points on the *Capital Market Line* represent portfolios consisting of the *tangency portfolio* and the *risk-free* asset (bond).

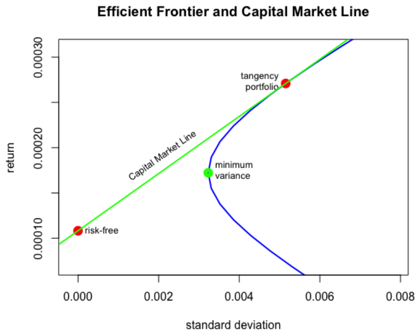
The *Capital Market Line* represents delevered and levered portfolios, consisting of the *tangency portfolio* combined with the *risk-free* asset (bond).

The *CML* portfolios have weights proportional to the tangency portfolio weights.

The *CML* portfolios above the tangent point are levered with respect to the *tangency portfolio* through borrowing at the *risk-free* rate r_f. Their weights are equal to the tangency portfolio weights multiplied by a factor greater than 1.

The *CML* portfolios below the tangent point are delevered with respect to the *tangency portfolio* through investing at the *risk-free* rate r_f. Their weights are equal to the tangency portfolio weights multiplied by a factor less than 1.

All the *CML* portfolios have the same *Sharpe ratio*.



Maximum Sharpe Portfolio Weights

The *Sharpe* ratio is equal to the ratio of excess returns divided by the portfolio standard deviation:

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

Where $\mu = \bar{\mathbf{r}} - r_f$ is the vector of mean excess returns (in excess of the risk-free rate r_f), $\mathbf{w}$ is the vector of portfolio weights, and $\sigma = \sqrt{\mathbf{w}^T \mathbb{C}\, \mathbf{w}}$, 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:

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

We then get:

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

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

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

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

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

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

The *maximum Sharpe* portfolio is an *efficient portfolio*, and so its mean return is equal to some target return r_t: $\bar{\mathbf{r}}^T \mathbf{w} = \sum_{i=1}^{n} w_i r_i = r_t$.

The mean return of the *maximum Sharpe* portfolio is equal to:

$$r_t = \bar{\mathbf{r}}^T \mathbf{w} = \frac{\bar{\mathbf{r}}^T \mathbb{C}^{-1} \mu}{\mathbb{1}^T \mathbb{C}^{-1} \mu} = \frac{\bar{\mathbf{r}}^T \mathbb{C}^{-1} (\bar{\mathbf{r}} - r_f)}{\mathbb{1}^T \mathbb{C}^{-1} (\bar{\mathbf{r}} - r_f)} =$$

$$\frac{\bar{\mathbf{r}}^T \mathbb{C}^{-1} \mathbb{1}\, r_f - \bar{\mathbf{r}}^T \mathbb{C}^{-1} \mathbb{1}\, \bar{\mathbf{r}}}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}\, r_f - \bar{\mathbf{r}}^T \mathbb{C}^{-1} \mathbb{1}} = \frac{c_{r1}\, r_f - c_{rr}}{c_{11}\, r_f - c_{r1}}$$

The above formula calculates the target return r_t from the risk-free rate r_f.

Returns and Variance of the *Maximum Sharpe* Portfolio

The *maximum Sharpe* portfolio weights depend on the value of the risk-free rate r_f:

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

The mean return of the *maximum Sharpe* portfolio is equal to:

$$r_t = \bar{\mathbf{r}}^T\mathbf{w} = \frac{\bar{\mathbf{r}}^T\mathbb{C}^{-1}\mu}{\mathbb{1}^T\mathbb{C}^{-1}\mu} = \frac{c_{r1}\, r_f - c_{rr}}{c_{11}\, r_f - c_{r1}}$$

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

$$\sigma^2 = \mathbf{w}^T\mathbb{C}\,\mathbf{w} = \frac{\mu^T\mathbb{C}^{-1}\mathbb{C}\,\mathbb{C}^{-1}\mu}{(\mathbb{1}^T\mathbb{C}^{-1}\mu)^2} = \frac{\mu^T\mathbb{C}^{-1}\mu}{(\mathbb{1}^T\mathbb{C}^{-1}\mu)^2} =$$
$$\frac{(\bar{\mathbf{r}} - r_f)^T\mathbb{C}^{-1}(\bar{\mathbf{r}} - r_f)}{(\mathbb{1}^T\mathbb{C}^{-1}(\bar{\mathbf{r}} - r_f))^2} = \frac{c_{11}r_t^2 - 2c_{r1}r_t + c_{rr}}{\det \mathbb{F}}$$

The above formula expresses the *maximum Sharpe* portfolio variance as a function of its mean return r_t.

The *maximum Sharpe* ratio is equal to:

$$SR = \frac{\mathbf{w}^T\mu}{\sigma} = \frac{\mu^T\mathbb{C}^{-1}\mu}{\mathbb{1}^T\mathbb{C}^{-1}\mu} \Big/ \frac{\sqrt{\mu^T\mathbb{C}^{-1}\mu}}{\mathbb{1}^T\mathbb{C}^{-1}\mu} =$$
$$\sqrt{\mu^T\mathbb{C}^{-1}\mu} = \sqrt{(\bar{\mathbf{r}} - r_f)^T\mathbb{C}^{-1}(\bar{\mathbf{r}} - r_f)}$$

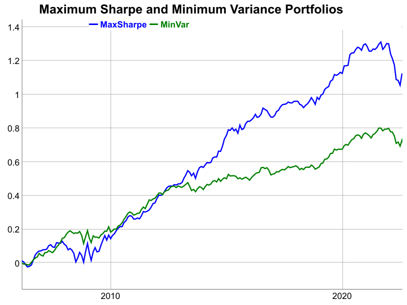
```
> # Calculate the mean excess returns
> riskf <- retarget - sharper*stdeveff
> retx <- (retm - riskf)
> # Calculate the efficient portfolio weights
> weightv <- 0.5*drop(covinv %*% cbind(unitv, retm) %*% lagm)
> # Calculate the maximum Sharpe weights
> weightms <- drop(covinv %*% retx)/sum(covinv %*% retx)
> all.equal(weightv, weightms)
> # Calculate the maximum Sharpe mean return in two ways
> all.equal(sum(retm*weightv),
+   (cr1*riskf-crr)/(c11*riskf-cr1))
> # Calculate the maximum Sharpe daily returns
> retd <- (retp %*% weightms)
> # Calculate the maximum Sharpe variance in four ways
> detf <- (c11*crr-cr1^2)  # det(fmat)
> all.equal(var(retd),
+   t(weightv) %*% covmat %*% weightv,
+   (t(retx) %*% covinv %*% retx)/sum(covinv %*% retx)^2,
+   (c11*retarget^2-2*cr1*retarget+crr)/detf)
> # Calculate the maximum Sharpe ratio
> sqrt(252)*sum(weightv*retx)/
+   sqrt(drop(t(weightv) %*% covmat %*% weightv))
> # Calculate the stock Sharpe ratios
> sqrt(252)*sapply((retp - riskf), function(x) mean(x)/sd(x))
```

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.

The *maximum Sharpe* portfolio has both a higher Sharpe ratio and higher absolute returns.

```
> # Calculate optimal portfolio returns
> wealthv <- cbind(retp %*% weightms, retp %*% weightmv)
> wealthv <- xts::xts(wealthv, zoo::index(retp))
> colnames(wealthv) <- c("MaxSharpe", "MinVar")
> # Calculate the Sharpe and Sortino ratios
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=(mean(x)-riskf)/sd(x), Sortino=(mean(x)-ris
> # Plot the log wealth
> endd <- rutils::calc_endpoints(retp, interval="weeks")
> dygraphs::dygraph(cumsum(wealthv)[endd],
+   main="Maximum Sharpe and Minimum Variance Portfolios") %>%
+   dyOptions(colors=c("blue", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```



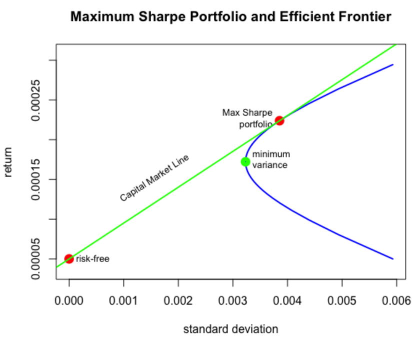
The *Maximum Sharpe* Portfolios and the *Efficient Frontier*

The *maximum Sharpe* portfolios are *efficient portfolios*, so they form the *efficient frontier*.

A *market portfolio* is the portfolio of all the available assets, with weights proportional to their market capitalizations.

The *maximum Sharpe* portfolio is sometimes considered to be the *market portfolio*, because it's the optimal portfolio for the given value of the risk-free rate r_f.

```
> # Calculate the maximum Sharpe portfolios for different risk-free
> detf <- (c11*crr-cr1^2)  # det(fmat)
> riskfv <- retmvm*seq(from=1.3, to=20, by=0.1)
> riskfv <- c(riskfv, retmvm*seq(from=(-20), to=0.7, by=0.1))
> effront <- sapply(riskfv, function(riskf) {
+   # Calculate the maximum Sharpe mean return
+   reteffm <- (cr1*riskf-crr)/(c11*riskf-cr1)
+   # Calculate the maximum Sharpe standard deviation
+   stdev <- sqrt((c11*reteffm^2-2*cr1*reteffm+crr)/detf)
+   c(return=reteffm, stdev=stdev)
+ })  # end sapply
> effront <- effront[, order(effront["return", ])]
> # Plot the efficient frontier
> reteffv <- effront["return", ]
> stdevs <- effront["stdev", ]
> aspratio <- 0.6*max(stdevs)/diff(range(reteffv))
> plot(x=stdevs, y=reteffv, t="l", col="blue", lwd=2, asp=aspratio
+   main="Maximum Sharpe Portfolio and Efficient Frontier",
+   xlim=c(0.0, max(stdevs)), xlab="standard deviation", ylab="retu
> # Plot the minimum variance portfolio
> points(x=stdevmv, y=retmvm, col="green", lwd=6)
> text(x=stdevmv, y=retmvm, labels="minimum \nvariance", pos=4, ce
```



```
> # Calculate the maximum Sharpe return and standard deviation
> riskf <- min(reteffv)
> retmax <- (cr1*riskf-crr)/(c11*riskf-cr1)
> stdevmax <- sqrt((c11*retmax^2-2*cr1*retmax+crr)/detf)
> # Plot the maximum Sharpe portfolio
> points(x=stdevmax, y=retmax, col="red", lwd=6)
> text(x=stdevmax, y=retmax, labels="Max Sharpe\nportfolio", pos=2,
> # Plot the risk-free point
> points(x=0, y=riskf, col="red", lwd=6)
> text(x=0, y=riskf, labels="risk-free", pos=4, cex=0.8)
> # Plot the tangent line
> sharper <- (stdevmax*detf)/(c11*retmax-cr1)
> abline(a=riskf, b=sharper, lwd=2, col="green")
> text(x=0.6*stdevmax, y=0.8*retmax, labels="Capital Market Line",
+      pos=2, cex=0.8, srt=180/pi*atan(aspratio*sharper))
```

Efficient Portfolios and Their Tangent Lines

The *efficient frontier* consists of all the *maximum Sharpe* portfolios corresponding to different values of the risk-free rate.

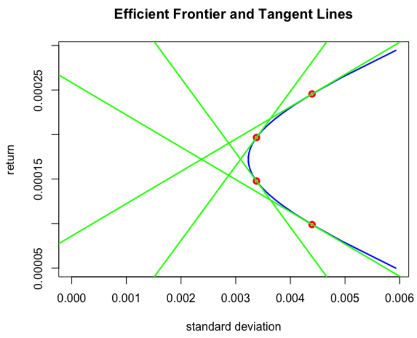
The target return can be expressed as a function of the risk-free rate as:

$$r_t = \frac{c_{r1} \, r_f - c_{rr}}{c_{11} \, r_f - c_{r1}}$$

If $r_f \to \pm\infty$ then $r_t \to r_{mv} = c_{r1} / c_{11}$.

But if the risk-free rate tends to the mean returns of the minimum variance portfolio: $r_f \to r_{mv} = c_{r1} / c_{11}$, then $r_t \to \pm\infty$, which means that there is no efficient portfolio corresponding to the risk-free rate equal to the mean returns of the minimum variance portfolio: $r_f = r_{mv} = c_{r1} / c_{11}$.

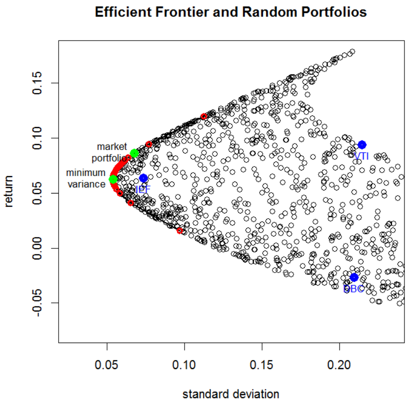
```
> # Plot the efficient frontier
> reteffv <- effront["return", ]
> stdevs <- effront["stdev", ]
> plot(x=stdevs, y=reteffv, t="l", col="blue", lwd=2,
+   xlim=c(0.0, max(stdevs)),
+   main="Efficient Frontier and Tangent Lines",
+   xlab="standard deviation", ylab="return")
```



```
> # Calculate vector of mean returns
> reteffv <- min(reteffv) + diff(range(reteffv))*c(0.2, 0.4, 0.6, 0
> # Plot the tangent lines
> for (reteffm in reteffv) {
+   # Calculate the maximum Sharpe standard deviation
+   stdev <- sqrt((c11*reteffm^2-2*cr1*reteffm+crr)/detf)
+   # Calculate the slope of the tangent line
+   sharper <- (stdev*detf)/(c11*reteffm-cr1)
+   # Calculate the risk-free rate as intercept of the tangent line
+   riskf <- reteffm - sharper*stdev
+   # Plot the tangent portfolio
+   points(x=stdev, y=reteffm, col="red", lwd=3)
+   # Plot the tangent line
+   abline(a=riskf, b=sharper, lwd=2, col="green")
+ } # end for
```

Random Portfolios

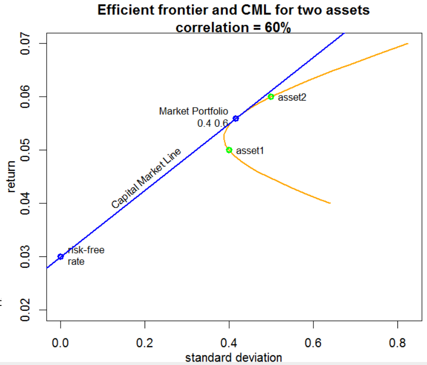
```
> # Calculate random portfolios
> nportf <- 1000
> randportf <- sapply(1:nportf, function(it) {
+   weightv <- runif(nstocks-1, min=-0.25, max=1.0)
+   weightv <- c(weightv, 1-sum(weightv))
+   # Portfolio returns and standard deviation
+   c(return=252*sum(weightv*retm),
+     stdev=sqrt(252*drop(weightv %*% covmat %*% weightv)))
+ })  # end sapply
> # Plot scatterplot of random portfolios
> x11(widthp <- 6, heightp <- 6)
> plot(x=randportf["stdev", ], y=randportf["return", ],
+      main="Efficient Frontier and Random Portfolios",
+      xlim=c(0.5*stdev, 0.8*max(randportf["stdev", ])),
+      xlab="standard deviation", ylab="return")
> # Plot maximum Sharpe portfolios
> lines(x=effront[, "stdev"], y=effront[, "return"], lwd=2)
> points(x=effront[, "stdev"], y=effront[, "return"],
+  col="red", lwd=3)
> # Plot the minimum variance portfolio
> points(x=stdev, y=retp, col="green", lwd=6)
> text(stdev, retp, labels="minimum\nvariance", pos=2, cex=0.8)
> # Plot efficient portfolio
> points(x=effront[marketp, "stdev"],
+  y=effront[marketp, "return"], col="green", lwd=6)
> text(x=effront[marketp, "stdev"], y=effront[marketp, "return"],
+      labels="market\nportfolio", pos=2, cex=0.8)
```



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

Plotting Efficient Frontier for Two-asset Portfolios

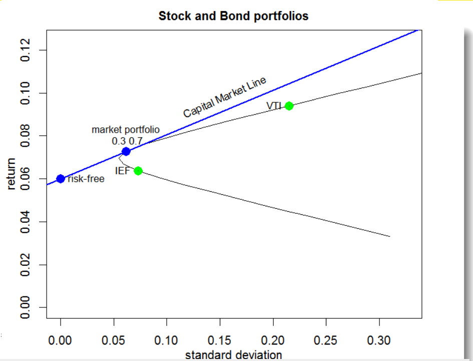
```
> riskf <- 0.03
> retp <- c(asset1=0.05, asset2=0.06)
> stdevs <- c(asset1=0.4, asset2=0.5)
> corrp <- 0.6
> covmat <- matrix(c(1, corrp, corrp, 1), nc=2)
> covmat <- t(t(stdevs*covmat)*stdevs)
> weightv <- seq(from=(-1), to=2, length.out=31)
> weightv <- cbind(weightv, 1-weightv)
> retp <- weightv %*% retp
> portfsd <- sqrt(rowSums(weightv*(weightv %*% covmat)))
> sharper <- (retp-riskf)/portfsd
> whichmax <- which.max(sharper)
> sharpem <- max(sharper)
> # Plot efficient frontier
> x11(widthp <- 6, heightp <- 5)
> par(mar=c(3,3,2,1)+0.1, oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> plot(portfsd, retp, t="l",
+  main=paste0("Efficient frontier and CML for two assets\ncorrelat:
+  xlab="standard deviation", ylab="return",
+  lwd=2, col="orange",
+  xlim=c(0, max(portfsd)),
+  ylim=c(0.02, max(retp)))
> # Add efficient portfolio (maximum Sharpe ratio portfolio)
> points(portfsd[whichmax], retp[whichmax],
+  col="blue", lwd=3)
> text(x=portfsd[whichmax], y=retp[whichmax],
+      labels=paste(c("efficient portfolio\n",
+  structure(c(weightv[whichmax], 1-weightv[whichmax]),
+          names=names(retp))), collapse=" "),
+      pos=2, cex=0.8)
```



```
> # Plot individual assets
> points(stdevs, retp, col="green", lwd=3)
> text(stdevs, retp, labels=names(retp), pos=4, cex=0.8)
> # Add point at risk-free rate and draw Capital Market Line
> points(x=0, y=riskf, col="blue", lwd=3)
> text(0, riskf, labels="risk-free\nrate", pos=4, cex=0.8)
> abline(a=riskf, b=sharpem, lwd=2, col="blue")
> rangev <- par("usr")
> text(portfsd[whichmax]/2, (retp[whichmax]+riskf)/2,
+      labels="Capital Market Line", cex=0.8, , pos=3,
+      srt=45*atan(sharpem*(rangev[2]-rangev[1])/
+              (rangev[4]-rangev[3])*
+              heightp/widthp)/(0.25*pi))
```

Efficient Frontier of Stock and Bond Portfolios

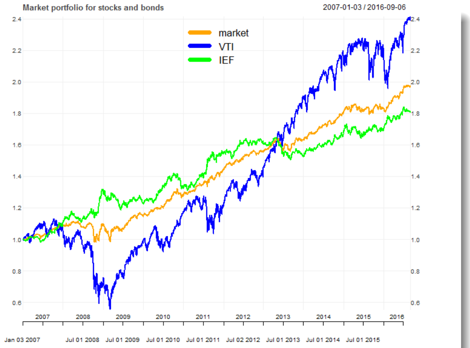
```
> # Vector of symbol names
> symbolv <- c("VTI", "IEF")
> # Matrix of portfolio weights
> weightv <- seq(from=(-1), to=2, length.out=31)
> weightv <- cbind(weightv, 1-weightv)
> # Calculate portfolio returns and volatilities
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> retp <- retp %*% t(weightv)
> portfv <- cbind(252*colMeans(retp),
+   sqrt(252)*matrixStats::colSds(retp))
> colnames(portfv) <- c("returns", "stdev")
> riskf <- 0.06
> portfv <- cbind(portfv,
+   (portfv[, "returns"]-riskf)/portfv[, "stdev"])
> colnames(portfv)[3] <- "Sharpe"
> whichmax <- which.max(portfv[, "Sharpe"])
> sharpem <- portfv[whichmax, "Sharpe"]
> plot(x=portfv[, "stdev"], y=portfv[, "returns"],
+      main="Stock and Bond portfolios", t="l",
+      xlim=c(0, 0.7*max(portfv[, "stdev"])), ylim=c(0, max(portfv[
+      xlab="standard deviation", ylab="return")
> # Add blue point for efficient portfolio
> points(x=portfv[whichmax, "stdev"], y=portfv[whichmax, "returns"]
> text(x=portfv[whichmax, "stdev"], y=portfv[whichmax, "returns"],
+      labels=paste(c("efficient portfolio\n",
+   structure(c(weightv[whichmax, 1], weightv[whichmax, 2]), names=
+      pos=3, cex=0.8)
```



```
> # Plot individual assets
> retm <- 252*sapply(retp, mean)
> stdevs <- sqrt(252)*sapply(retp, sd)
> points(stdevs, retm, col="green", lwd=6)
> text(stdevs, retm, labels=names(retp), pos=2, cex=0.8)
> # Add point at risk-free rate and draw Capital Market Line
> points(x=0, y=riskf, col="blue", lwd=6)
> text(0, riskf, labels="risk-free", pos=4, cex=0.8)
> abline(a=riskf, b=sharpem, col="blue", lwd=2)
> rangev <- par("usr")
> text(max(portfv[, "stdev"])/3, 0.75*max(portfv[, "returns"]),
+      labels="Capital Market Line", cex=0.8, , pos=3,
+      srt=45*atan(sharpem*(rangev[2]-rangev[1])/
+             (rangev[4]-rangev[3])*
+             heightp/widthp)/(0.25*pi))
```

Performance of Efficient Portfolio for Stocks and Bonds

```
> # Calculate cumulative returns of VTI and IEF
> retsoptim <- lapply(retp,
+   function(retp) exp(cumsum(retp)))
> retsoptim <- rutils::do_call(cbind, retsoptim)
> # Calculate the efficient portfolio returns
> retsoptim <- cbind(exp(cumsum(retp %*%
+     c(weightv[whichmax], 1-weightv[whichmax]))),
+   retsoptim)
> colnames(retsoptim)[1] <- "efficient"
> # Plot efficient portfolio with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue", "green")
> chart_Series(retsoptim, theme=plot_theme,
+    name="Efficient Portfolio for Stocks and Bonds")
> legend("top", legend=colnames(retsoptim),
+    cex=0.8, inset=0.1, bg="white", lty=1,
+    lwd=6, col=plot_theme$col$line.col, bty="n")
```



Maximum Return Portfolio Using Linear Programming

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

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

Where **r** is the vector of returns, and **w** is the vector of portfolio weights, with a linear constraint:

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

And a box constraint:

$$0 \leq w_i \leq 1$$

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.

```
> library(rutils)
> library(Rglpk)
> # Vector of symbol names
> symbolv <- c("VTI", "IEF", "DBC")
> nstocks <- NROW(symbolv)
> # Calculate the objective vector - the mean returns
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> objvec <- colMeans(retp)
> # Specify matrix of linear constraint coefficients
> coeffm <- matrix(c(rep(1, nstocks), 1, 1, 0),
+            nc=nstocks, byrow=TRUE)
> # Specify the logical constraint operators
> logop <- c("==", "<=")
> # Specify the vector of constraints
> consv <- c(1, 0)
> # Specify box constraints (-1, 1) (default is c(0, Inf))
> boxc <- list(lower=list(ind=1:nstocks, val=rep(-1, nstocks)),
+        upper=list(ind=1:nstocks, val=rep(1, nstocks)))
> # Perform optimization
> optiml <- Rglpk::Rglpk_solve_LP(
+   obj=objvec,
+   mat=coeffm,
+   dir=logop,
+   rhs=consv,
+   bounds=boxc,
+   max=TRUE)
> all.equal(optiml$optimum, sum(objvec*optiml$solution))
> optiml$solution
> coeffm %*% optiml$solution
```

Conditional Value at Risk (*CVaR*)

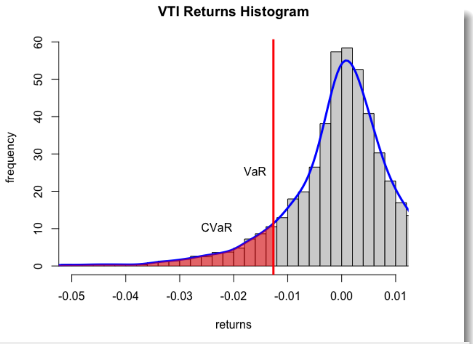
The *Conditional Value at Risk* (*CVaR*) is equal to the average of the *VaR* for confidence levels less than a given confidence level α:

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

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

The function `density()` calculates a kernel estimate of the probability density for a sample of data, and returns a list with a vector of loss values and a vector of corresponding densities.

```
> # Calculate the VTI percentage returns
> retp <- na.omit(rutils::etfenv$returns$VTI)
> confl <- 0.1
> varisk <- quantile(retp, confl)
> cvar <- mean(retp[retp < varisk])
> # Or
> sortv <- sort(as.numeric(retp))
> varind <- round(confl*NROW(retp))
> varisk <- sortv[varind]
> cvar <- mean(sortv[1:varind])
> # Plot histogram of VTI returns
> varmin <- (-0.05)
> histp <- hist(retp, col="lightgrey",
+   xlab="returns", breaks=100, xlim=c(varmin, 0.01),
+   ylab="frequency", freq=FALSE, main="VTI Returns Histogram")
```



```
> # Plot density of losses
> densv <- density(retp, adjust=1.5)
> lines(densv, lwd=3, col="blue")
> # Add line for VaR
> abline(v=varisk, col="red", lwd=3)
> ymax <- max(densv$y)
> text(x=varisk, y=2*ymax/3, labels="VaR", lwd=2, pos=2)
> # Add shading for CVaR
> rangev <- (densv$x < varisk) & (densv$x > varmin)
> polygon(
+   c(varmin, densv$x[rangev], varisk),
+   c(0, densv$y[rangev], 0),
+   col=rgb(1, 0, 0,0.5), border=NA)
> text(x=1.5*varisk, y=ymax/7, labels="CVaR", lwd=2, pos=2)
```

CVaR Portfolio Weights Using Linear Programming

The stock 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,

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

Where b_i is the negative objective vector, and $\mathbf{w}$ is the vector of portfolio weights, with a linear constraint:

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

And a box constraint:

$$0 \leq w_i \leq 1$$

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

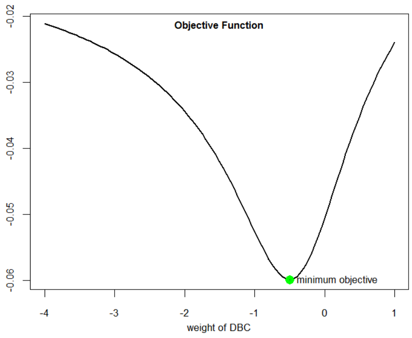
```
> library(rutils)  # Load rutils
> library(Rglpk)
> # Vector of symbol names and returns
> symbolv <- c("VTI", "IEF", "DBC")
> nstocks <- NROW(symbolv)
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> retm <- colMeans(retp)
> confl <- 0.05
> rmin <- 0 ; wmin <- 0 ; wmax <- 1
> weightsum <- 1
> ncols <- NCOL(retp) # number of assets
> nrows <- NROW(retp) # number of rows
> # Create objective vector
> objvec <- c(numeric(ncols), rep(-1/(confl/nrows), nrows), -1)
> # Specify matrix of linear constraint coefficients
> coeffm <- rbind(cbind(rbind(1, retm),
+                 matrix(data=0, nrow=2, ncol=(nrows+1))),
+           cbind(coredata(retp), diag(nrows), 1))
> # Specify the logical constraint operators
> logop <- c("==", ">=", rep(">=", nrows))
> # Specify the vector of constraints
> consv <- c(weightvum, rmin, rep(0, nrows))
> # Specify box constraints (wmin, wmax) (default is c(0, Inf))
> boxc <- list(lower=list(ind=1:ncols, val=rep(wmin, ncols)),
+        upper=list(ind=1:ncols, val=rep(wmax, ncols)))
> # Perform optimization
> optiml <- Rglpk_solve_LP(obj=objvec, mat=coeffm, dir=logop, rhs=co
> all.equal(optiml$optimum, sum(objvec*optiml$solution))
> coeffm %*% optiml$solution
> as.numeric(optiml$solution[1:ncols])
```

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.

```
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> # Create initial vector of portfolio weights
> weightv <- rep(1, NROW(symbolv))
> names(weightv) <- symbolv
> # Objective equal to minus Sharpe ratio
> objfun <- function(weightv, retp) {
+   retp <- retp %*% weightv
+   if (sd(retp) == 0)
+     return(0)
+   else
+     -return(mean(retp)/sd(retp))
+ }  # end objfun
> # Objective for equal weight portfolio
> objfun(weightv, retp=retp)
> optiml <- unlist(optimize(
+   f=function(weight)
+     objfun(c(1, 1, weight), retp=retp),
+   interval=c(-4, 1)))
> # Vectorize objective function with respect to third weight
> objvec <- function(weightv) sapply(weightv,
+   function(weight) objfun(c(1, 1, weight),
+     retp=retp))
> # Or
> objvec <- Vectorize(FUN=function(weight)
+     objfun(c(1, 1, weight), retp=retp),
+   vectorize.args="weight")  # end Vectorize
> objvec(1)
> objvec(1:3)
```



```
> # Plot objective function with respect to third weight
> curve(expr=objvec,
+       type="l", xlim=c(-4.0, 1.0),
+       xlab=paste("weight of", names(weightv[3])),
+       ylab="", lwd=2)
> title(main="Objective Function", line=(-1))  # Add title
> points(x=optiml[1], y=optiml[2], col="green", lwd=6)
> text(x=optiml[1], y=optiml[2],
+      labels="minimum objective", pos=4, cex=0.8)
>
> ### below is simplified code for plotting objective function
> # Create vector of DBC weights
> weightv <- seq(from=-4, to=1, by=0.1)
> obj_val <- sapply(weightv,
+   function(weight) objfun(c(1, 1, weight)))
> plot(x=weightv, y=obj_val, t="l",
```

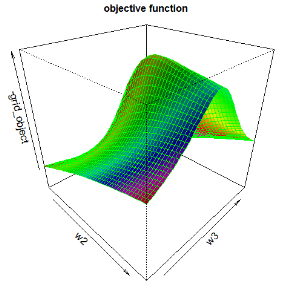
Perspective Plot of Portfolio Objective Function

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 weights
> objvec <- Vectorize(
+   FUN=function(w1, w2, w3) objfun(c(w1, w2, w3)),
+   vectorize.args=c("w2", "w3"))  # end Vectorize
> # Calculate objective on 2-d (w2 x w3) parameter grid
> w2 <- seq(-3, 7, length=50)
> w3 <- seq(-5, 5, length=50)
> grid_object <- outer(w2, w3, FUN=objvec, w1=1)
> 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 function
> library(rgl)
> rgl::persp3d(z=-grid_object, zlab="objective",
+   col="green", main="objective function")
> rgl::persp3d(
+   x=function(w2, w3) {-objvec(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.

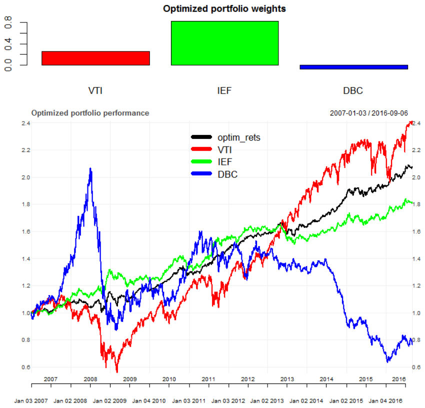
`optim()` returns a list containing the location of the minimum and the objective function value.

```
> # Optimization to find weights with maximum Sharpe ratio
> optiml <- optim(par=weightv,
+              fn=objfun,
+              retp=retp,
+              method="L-BFGS-B",
+              upper=c(1.1, 10, 10),
+              lower=c(0.9, -10, -10))
> # Optimal parameters
> optiml$par
> optiml$par <- optiml$par/sum(optiml$par)
> # Optimal Sharpe ratio
> -objfun(optiml$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(optiml$par, col=c("red", "green", "blue"),
+   main="Optimized portfolio weights")
> # Calculate cumulative returns of VTI, IEF, DBC
> retc <- lapply(retp,
+   function(retp) exp(cumsum(retp)))
> retc <- rutils::do_call(cbind, retc)
> # Calculate optimal portfolio returns with VTI, IEF, DBC
> retsoptim <- cbind(
+   exp(cumsum(retp %*% optiml$par)),
+   retc)
> colnames(retsoptim)[1] <- "retsoptim"
> # Plot optimal returns with VTI, IEF, DBC
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "green", "blue")
> chart_Series(retsoptim, theme=plot_theme,
+        name="Optimized portfolio performance")
> legend("top", legend=colnames(retsoptim), cex=1.0,
+    inset=0.1, bg="white", lty=1, lwd=6,
+    col=plot_theme$col$line.col, bty="n")
> # Or plot non-compounded (simple) cumulative returns
> PerformanceAnalytics::chart.CumReturns(
+   cbind(retp %*% optiml$par, retp),
+   lwd=2, ylab="", legend.loc="topleft", main="")
```



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
> optiml <- solve.QP(Dmat=2*covmat,
+             dvec=rep(0, 2),
+             Amat=matrix(0, nr=2, nc=1),
+             bvec=0)
> # Minimum variance weights sum equal to 1
> optiml <- solve.QP(Dmat=2*covmat,
+             dvec=rep(0, 2),
+             Amat=matrix(1, nr=2, nc=1),
+             bvec=1)
> # Optimal value of objective function
> t(optiml$solution) %*% covmat %*% optiml$solution
> ## Perform simple optimization for reference
> # Objective function for simple optimization
> objfun <- function(x) {
+   x <- c(x, 1-x)
+   t(x) %*% covmat %*% x
+ }  # end objfun
> unlist(optimize(f=objfun, interval=c(-1, 2)))
```

Portfolio Optimization Using Package *quadprog*

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

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

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

The portfolio weights $\mathbf{w}$ are constrained as:

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

$$0 \leq w_i \leq 1$$

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 returns
> symbolv <- c("VTI", "IEF", "DBC")
> nstocks <- NROW(symbolv)
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> # Calculate the covariance matrix
> covmat <- cov(retp)
> # Minimum variance weights, with sum equal to 1
> optiml <- quadprog::solve.QP(Dmat=2*covmat,
+             dvec=numeric(3),
+             Amat=matrix(1, nr=3, nc=1),
+             bvec=1)
> # Minimum variance, maximum returns
> optiml <- quadprog::solve.QP(Dmat=2*covmat,
+             dvec=apply(0.1*retp, 2, mean),
+             Amat=matrix(1, nr=3, nc=1),
+             bvec=1)
> # Minimum variance positive weights, sum equal to 1
> a_mat <- cbind(matrix(1, nr=3, nc=1),
+        diag(3), -diag(3))
> b_vec <- c(1, rep(0, 3), rep(-1, 3))
> optiml <- quadprog::solve.QP(Dmat=2*covmat,
+             dvec=numeric(3),
+             Amat=a_mat,
+             bvec=b_vec,
+             meq=1)
```

Package *DEoptim* for Global Optimization

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

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

The first generation of solutions is selected randomly.

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

The best solutions are selected for creating the next generation.

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

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

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

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 returns
> symbolv <- c("VTI", "IEF", "DBC")
> nstocks <- NROW(symbolv)
> retp <- na.omit(rutils::etfenv$returns[, symbolv])
> # Objective equal to minus Sharpe ratio
> objfun <- function(weightv, retp) {
+   retp <- retp %*% weightv
+   if (sd(retp) == 0)
+     return(0)
+   else
+     -return(mean(retp)/sd(retp))
+ }  # end objfun
> # Perform optimization using DEoptim
> optiml <- DEoptim::DEoptim(fn=objfun,
+   upper=rep(10, NCOL(retp)),
+   lower=rep(-10, NCOL(retp)),
+   retp=retp,
+   control=list(trace=FALSE, itermax=100, parallelType=1))
> weightv <- optiml$optim$bestmem/sum(abs(optiml$optim$bestmem))
> names(weightv) <- colnames(retp)
```

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:

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

The portfolio weights **w** are shrunk to zero as the parameters λ and α increase.

```
> # Objective with shrinkage penalty
> objfun <- function(weightv, retp, lambdaf, alpha) {
+   retp <- retp %*% weightv
+   if (sd(retp) == 0)
+     return(0)
+   else {
+     penaltyv <- lambdaf*((1-alpha)*sum(weightv^2) +
+ alpha*sum(abs(weightv)))
+     -return(mean(retp)/sd(retp) + penaltyv)
+   }
+ }  # end objfun
> # Objective for equal weight portfolio
> weightv <- rep(1, NROW(symbolv))
> names(weightv) <- symbolv
> lambdaf <- 0.5 ; alpha <- 0.5
> objfun(weightv, retp=retp, lambdaf=lambdaf, alpha=alpha)
> # Perform optimization using DEoptim
> optiml <- DEoptim::DEoptim(fn=objfun,
+   upper=rep(10, NCOL(retp)),
+   lower=rep(-10, NCOL(retp)),
+   retp=retp,
+   lambdaf=lambdaf,
+   alpha=alpha,
+   control=list(trace=FALSE, itermax=100, parallelType=1))
> weightv <- optiml$optim$bestmem/sum(abs(optiml$optim$bestmem))
> names(weightv) <- colnames(retp)
```

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 the excess returns divided by the individual variances:

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

The portfolio weights are proportional to the *Kelly ratios* - the excess returns divided by the variances:

$$w_i \propto \frac{\mu_i}{\sigma_i^2}$$

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

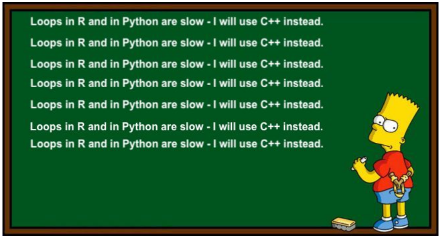
The package *Rcpp* allows calling C++ functions from R, by compiling the C++ code and creating R functions.

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

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

The package *Rcpp* relies on *Rtools* for compiling the C++ code:
https://cran.r-project.org/bin/windows/Rtools/

You can learn more about the package *Rcpp* here:
http://adv-r.had.co.nz/Rcpp.html
http://www.rcpp.org/
http://gallery.rcpp.org/


Loops in R and in Python are slow - I will use C++ instead.
Loops in R and in Python are slow - I will use C++ instead.
Loops in R and in Python are slow - I will use C++ instead.
Loops in R and in Python are slow - I will use C++ instead.
Loops in R and in Python are slow - I will use C++ instead.
Loops in R and in Python are slow - I will use C++ instead.
Loops in R and in Python are slow - I will use C++ instead.

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

Function cppFunction() for Compiling C++ code

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

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

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

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

Performing Loops in *Rcpp Sugar*

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

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

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

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

Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

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

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

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

Rcpp Attributes

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

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

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

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

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

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

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

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

Generating Random Numbers Using Logistic Map in *Rcpp*

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

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

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

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

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

Package *RcppArmadillo* for Fast Linear Algebra

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

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

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

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

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

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

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

// The function inner_mat() calculates the inner (dot) p
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_mat(const arma::vec& vecv2, const arma::mat
  return arma::as_scalar(trans(vecv2) * (matv * vecv1));
}  // end inner_mat
```

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

Simulating *ARIMA* Processes Using *RcppArmadillo*

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

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

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

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

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

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

  return arimav;
}  // end sim_arima
```

Fast Matrix Algebra Using *RcppArmadillo*

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

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

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

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

// Examples of \emph{RcppArmadillo} functions below

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

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

Fast Correlation Matrix Inverse Using *RcppArmadillo*

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

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

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

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

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

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

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

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

Portfolio Optimization Using *RcppArmadillo*

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

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

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

Strategy Backtesting Using *RcppArmadillo*

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

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

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

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

  // Return the strategy pnls
  return pnls;

}  // end back_test
```

Package *reticulate* for Running Python from RStudio

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

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

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

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

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

Running Python Under *reticulate*

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

Homework Assignment

No homework!