FRE7241 Algorithmic Portfolio Management

Lecture#4, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

February 20, 2018

Geometric Brownian Motion

If the percentage asset returns $\mathrm{d}\log P$ follow *Brownian motion* (GBM):

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

Then asset prices follow *Geometric Brownian motion*:

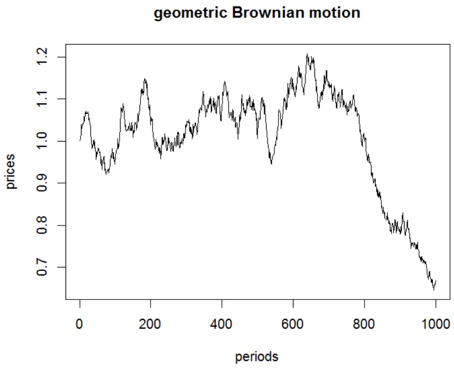
$$\mathrm{d}P_t = \mu P_t \mathrm{d}t + \sigma P_t \mathrm{d}W_t$$

Where σ is the volatility, and $\mathrm{d}W_t$ follows the standard normal distribution $N(0, \sqrt{\mathrm{d}t})$,

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, (in accordance with Ito's lemma),

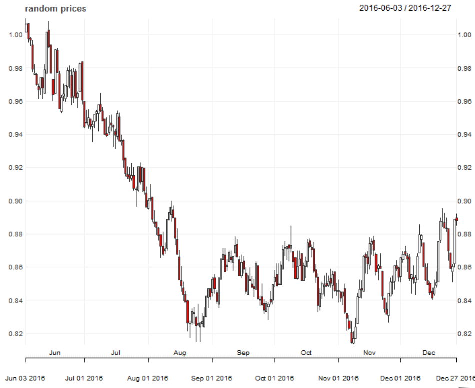


```
> # define daily volatility and growth rate
> vol_at <- 0.01; dri_ft <- 0.0; len_gth <- 1000
> # simulate geometric Brownian motion
> re_turns <- vol_at*rnorm(len_gth) +
+   dri_ft - vol_at^2/2
> price_s <- exp(cumsum(re_turns))
> plot(price_s, type="l",
+      xlab="periods", ylab="prices",
+      main="geometric Brownian motion")
```

Simulating Random *OHLC* Prices

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

```
> # simulate geometric Brownian motion
> vol_at <- 0.01/sqrt(48)
> dri_ft <- 0.0
> len_gth <- 10000
> in_dex <- seq(from=as.POSIXct(paste(Sys.Date()
+   length.out=len_gth, by="30 min")
> price_s <- xts(exp(cumsum(vol_at*rnorm(len_gth
+   order.by=in_dex)
> price_s <- cbind(price_s,
+   volume=sample(x=10*(2:18), size=len_gth, rep
> # aggregate to daily OHLC data
> oh_lc <- xts::to.daily(price_s)
> quantmod::chart_Series(oh_lc, name="random pri
> # dygraphs candlestick plot using pipes syntax
> library(dygraphs)
> dygraphs::dygraph(oh_lc[, 1:4]) %>%
+   dyCandlestick()
> # dygraphs candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dygraph(oh_lc[, 1:4]))
```

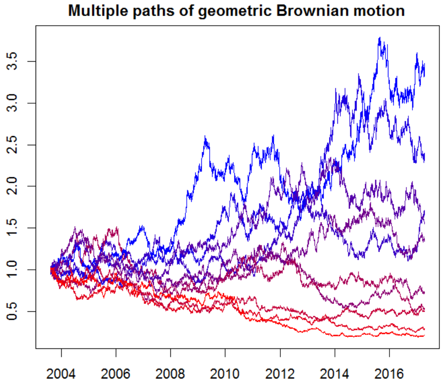


Paths of Geometric Brownian Motion

If asset prices follow *Geometric Brownian motion*, then at any point in time, they are distributed according to the *Log-normal* distribution,

The volatility increases with time as the square root of time: $\sigma \propto \sqrt{t}$

The skewness of the price distribution increases exponentially with the volatility and time: $\mathbb{E}[(x - \mathbb{E}[x])^3] \propto e^{1.5\sigma^2} \propto e^{1.5t}$

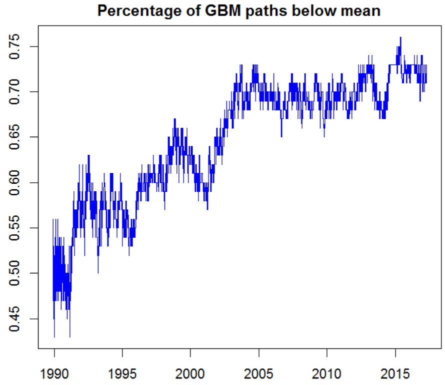


```
> # define daily volatility and growth rate
> vol_at <- 0.01; dri_ft <- 0.0; len_gth <- 5000
> path_s <- 10
> # simulate multiple paths of geometric Brownia
> price_s <- matrix(vol_at*rnorm(path_s*len_gth)
+     dri_ft - vol_at^2/2, nc=path_s)
> price_s <- exp(matrixStats::colCumsums(price_s,,
> # create zoo time series
> price_s <- zoo(price_s, order.by=seq.Date(Sys.Date()-NROW(price_s)+1, Sys.Date(), by=1))
> # plot zoo time series
> col_ors <- colorRampPalette(c("red", "blue"))(NCOL(price_s))
> col_ors <- col_ors[order(order(price_s[NROW(price_s), ]))]
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(price_s, main="Multiple paths of geometric Brownian motion",
+    xlab=NA, ylab=NA, plot.type="single", col=col_ors)
```

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,

The skewness of the price distribution increases exponentially with the volatility and time:
$\mathbb{E}[(x - \mathbb{E}[x])^3] \propto e^{1.5\sigma^2} \propto e^{1.5t}$



```
> # define daily volatility and growth rate
> vol_at <- 0.01; dri_ft <- 0.0; len_gth <- 1000
> path_s <- 100
> # simulate multiple paths of geometric Brownia
> price_s <- matrix(vol_at*rnorm(path_s*len_gth)
+     dri_ft - vol_at^2/2, nc=path_s)
> price_s <- exp(matrixStats::colCumsums(price_s
> # calculate percentage of paths below the expected value
> per_centage <- rowSums(price_s < 1.0) / path_s
> # create zoo time series of percentage of paths below the expected value
> per_centage <- zoo(per_centage, order.by=seq.Date(Sys.Date()-NROW(per_centage)+1, Sys.Date(),
> # plot zoo 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(per_centage, main="Percentage of GBM paths below mean",
+    xlab=NA, ylab=NA, col="blue")
```

Log-normal Probability Distribution

Let x be a random variable which follows the *Normal* distribution $N(x, \mu, \sigma)$,

Then the exponential of x: $y = e^x$ follows the *Log-normal* distribution:,

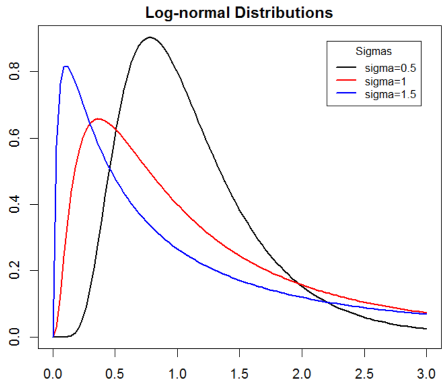
$$logN(y, \mu, \sigma) = \frac{\exp(-(\log y - \mu)^2/2\sigma^2)}{y\sigma\sqrt{2\pi}}$$

The mean of the *Log-normal* distribution is equal to: $\mathbb{E}[x] = \exp(\mu + \sigma^2/2)$

The *Log-normal* distribution has a positive skewness (third moment) equal to:
$\mathbb{E}[(x - \mathbb{E}[x])^3] = (e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$

If asset returns follow the *Normal* probability distribution, then asset prices follow the *Log-normal* distribution,

```
> # sigma values
> sig_mas <- c(0.5, 1, 1.5)
> # create plot colors
> col_ors <- c("black", "red", "blue")
> # create legend labels
> lab_els <- paste("sigma", sig_mas, sep="=")
```

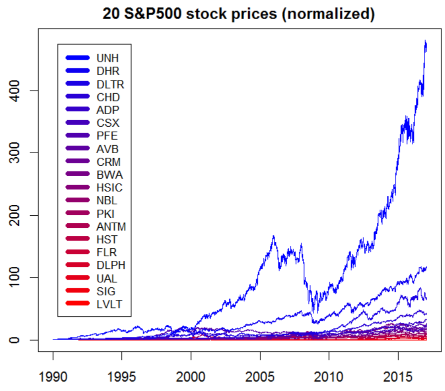


```
> # plot all curves
> for (in_dex in 1:NROW(sig_mas)) {
+   curve(expr=dlnorm(x, sdlog=sig_mas[in_dex]),
+   type="l", xlim=c(0, 3),
+   xlab="", ylab="", lwd=2,
+   col=col_ors[in_dex],
+   add=as.logical(in_dex-1))
+ }  # end for
> # add title
> title(main="Log-normal Distributions", line=0
```

Time Evolution of Stock Prices

Stock prices evolve in time similarly to *Geometric Brownian motion*, and they also exhibit a very skewed distribution of prices,

```
> # load S&P500 stock prices
> load("C:/Develop/R/lecture_slides/data/sp500.R
> ls(env_sp500)
> # extract closing prices
> price_s <- eapply(env_sp500, quantmod::Cl)
> # flatten price_s into a single xts series
> price_s <- rutils::do_call(cbind, price_s)
> # carry forward and backward non-NA prices
> price_s <- zoo::na.locf(price_s)
> price_s <- zoo::na.locf(price_s, fromLast=TRUE
> sum(is.na(price_s))
> # rename and normalize columns
> colnames(price_s) <- sapply(colnames(price_s),
+   function(col_name) strsplit(col_name, split=
> price_s <- xts(t(t(price_s) / as.numeric(pric
+          order.by=index(price_s))
> # calculate permution index for sorting the l
> or_der <- order(price_s[NROW(price_s), ])
> # select a few symbols
> sym_bols <- colnames(price_s)[or_der]
> sym_bols <- sym_bols[seq.int(from=1, to=(NROW
```



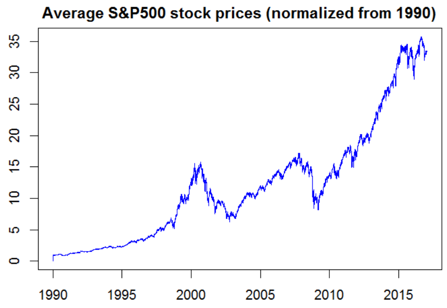
```
> # plot xts time series of price_s
> col_ors <- colorRampPalette(c("red", "blue"))(
> col_ors <- col_ors[order(order(price_s[NROW(pr
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(price_s[, sym_bols], main="20 S&P500
+    xlab=NA, ylab=NA, plot.type="single", col=c
> legend(x="topleft", inset=0.05, cex=0.8,
+  legend=rev(sym_bols), col=rev(col_ors), lwd=6
```

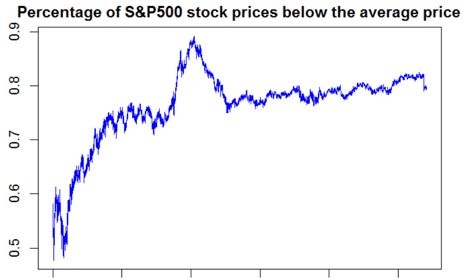
Distribution of Stock Prices

In most stock indices, a small number of stocks reach very high prices, while the prices of the majority of the other stocks remain below the average index price,

For example, for a recent cohort of S&P500 stocks (but with prices starting from 1990), the current prices of almost 80% of the stocks are now below the average price of the cohort,

```
> # calculate average of valid stock prices
> val_id <- (price_s != 1)  # valid stocks
> num_stocks <- rowSums(val_id)
> num_stocks[1] <- NCOL(price_s)
> in_dex <- rowSums(price_s * val_id) / num_stoc
> # calculate percentage of stock prices below t
> per_centage <- rowSums((price_s < in_dex) & va
> # create zoo time series of average stock pric
> in_dex <- zoo(in_dex, order.by=index(price_s))
> # plot zoo time series of average stock prices
> x11(width=6, height=4)
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot.zoo(in_dex, main="Average S&P500 stock pr
+    xlab=NA, ylab=NA, col="blue")
> # create xts time series of percentage of stoc
> per_centage <- xts(per_centage, order.by=index
> # plot percentage of stock prices below the av
> plot.zoo(per_centage[-(1:2),],
```





Autocorrelation Function

The *Autocorrelation Function* is the correlation coefficient of a time series with its lagged values:

$$\rho_k = \frac{1}{(n-k)\sigma^2} \sum_{i=k+1}^{n} (x_i - \bar{x})(x_{i-k} - \bar{x})$$

The function acf() from the base package *stats* calculates and plots the autocorrelation function for a univariate time series,

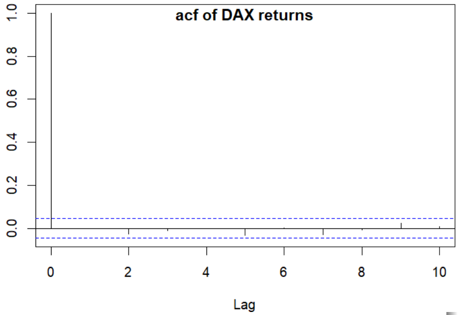
acf() returns the acf data invisibly - the return value isn't automatically printed to the console,

The acf() return data can be assigned to a variable, and then printed,

```
> library(zoo)
> re_turns <- diff(log(EuStockMarkets[, 1]))
> # acf() autocorrelation from package stats
> acf(zoo::coredata(re_turns), lag=10, main="")
> title(main="acf of DAX returns", line=-1)
```

The package *zoo* is designed for managing *time series* and ordered objects,

The function coredata extracts the core underlying data from a complex object,



The horizontal dashed lines are confidence intervals of the autocorrelation estimator (at 95% significance level),

The DAX time series of returns does not appear to have statistically significant autocorrelations,

The function acf() has the drawback that it plots the lag-zero autocorrelation (which is simply 1),

Ljung-Box Test of Autocorrelation

The *Ljung-Box* test *null hypothesis* is that autocorrelations are equal to zero,

The test statistic is:

$$Q = n(n+2) \sum_{k=1}^{maxlag} \frac{\hat{\rho}_k^2}{n-k}$$

Where `n` is the sample size, and the $\hat{\rho}_k$ are sample autocorrelations,

The *Ljung-Box* statistic follows the *chi-squared* distribution with *maxlag* degrees of freedom,

The *Ljung-Box* statistic is small for time series that are *not* autocorrelated,

The *p*-value for DAX returns is large, and we conclude that the *null hypothesis* is `TRUE`, and that DAX returns are *not* autocorrelated,

The *p*-value for changes in econometric data is extremely small, and we conclude that the *null hypothesis* is `FALSE`, and that econometric data *are* autocorrelated,

```
> # Ljung-Box test for DAX returns
> # 'lag' is the number of autocorrelation coeff
> Box.test(re_turns, lag=10, type="Ljung")
>
> # changes in 3 month T-bill rate are autocorre
> Box.test(macro_diff[, "3mTbill"],
+    lag=10, type="Ljung")
>
> # changes in unemployment rate are autocorrela
> Box.test(macro_diff[, "unemprate"],
+    lag=10, type="Ljung")
```

Improved Autocorrelation Function

Inspection of the data returned by acf() shows how to omit the lag-zero autocorrelation,

```
> dax_acf <- acf(coredata(re_turns), plot=FALSE
> summary(dax_acf)  # get the structure of the '
> # print(dax_acf)  # print acf data
> dim(dax_acf$acf)
> dim(dax_acf$lag)
> head(dax_acf$acf)
```

The below wrapper function for acf() omits the lag-zero autocorrelation,

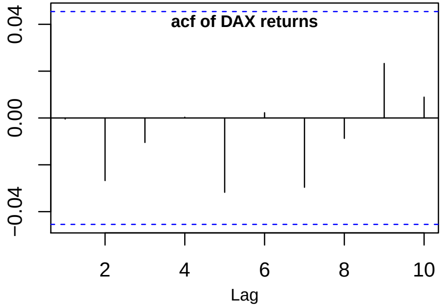
```
> acf_plus <- function (ts_data, plot=TRUE,
+                 xlab="Lag", ylab="",
+                 main="", ...) {
+   acf_data <- acf(x=ts_data, plot=FALSE, ...)
+ # remove first element of acf data
+   acf_data$acf <-  array(data=acf_data$acf[-1]
+     dim=c((dim(acf_data$acf)[1]-1), 1, 1))
+   acf_data$lag <-  array(data=acf_data$lag[-1]
+     dim=c((dim(acf_data$lag)[1]-1), 1, 1))
+   if (plot) {
+     ci <- qnorm((1+0.95)/2)*sqrt(1/length(ts_d
+     ylim <- c(min(-ci, range(acf_data$acf[-1])
+         max(ci, range(acf_data$acf[-1])))
+     plot(acf_data, xlab=xlab, ylab=ylab,
+    ylim=ylim, main=main, ci=0)
+     abline(h=c(-ci, ci), col="blue", lty=2)
+   }
+   invisible(acf_data)  # return invisibly
+ }  # end acf_plus
```

Autocorrelation of DAX Returns

The DAX time series of returns does not appear to have statistically significant autocorrelations,

But the acf plot alone is not enough to test whether autocorrelations are statistically significant or not,

```
> # improved autocorrelation function
> acf_plus(coredata(re_turns), lag=10, main="")
> title(main="acf of DAX returns", line=-1)
> # Ljung-Box test for DAX returns
> Box.test(re_turns, lag=10, type="Ljung")
```

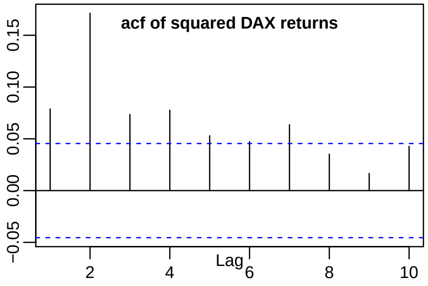


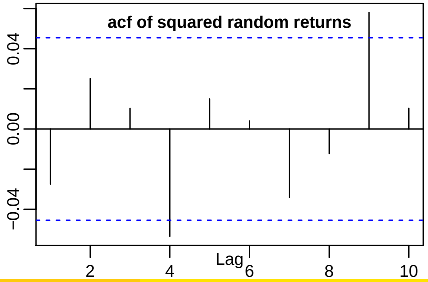
Autocorrelation of Squared DAX Returns

Squared DAX returns do have statistically significant autocorrelations,

But squared random returns are not autocorrelated,

```
> # autocorrelation of squared DAX returns
> acf_plus(coredata(re_turns)^2,
+    lag=10, main="")
> title(main="acf of squared DAX returns",
+ line=-1)
> # autocorrelation of squared random returns
> acf_plus(rnorm(length(re_turns))^2,
+    lag=10, main="")
> title(main="acf of squared random returns",
+ line=-1)
> # Ljung-Box test for squared DAX returns
> Box.test(re_turns^2, lag=10, type="Ljung")
```





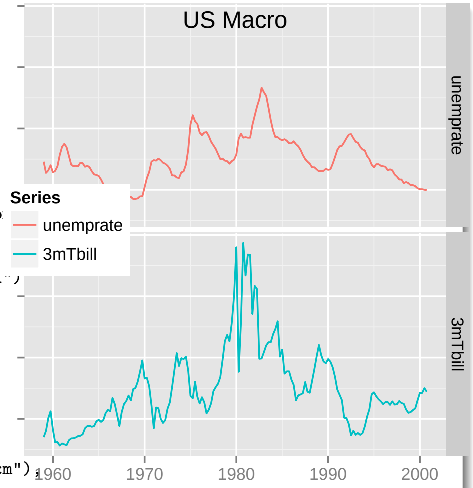
U.S. Macroeconomic Data

The package *Ecdat* contains the Macrodat U.S. macroeconomic data,

"lhur" is the unemployment rate (average of months in quarter),

"fygm3" 3 month treasury bill interest rate (last month in quarter)

```
> library(Ecdat)  # load Ecdat
> colnames(Macrodat)  # United States Macroecono
> macro_zoo <- as.zoo(  # coerce to "zoo"
+     Macrodat[, c("lhur", "fygm3")])
> colnames(macro_zoo) <- c("unemprate", "3mTbill")
> # ggplot2 in multiple panes
> autoplot(  # generic ggplot2 for "zoo"
+   object=macro_zoo, main="US Macro",
+   facets=Series ~ .) + # end autoplot
+   xlab("") +
+ theme(  # modify plot theme
+   legend.position=c(0.1, 0.5),
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+   plot.background=element_blank(),
+   axis.text.y=element_blank()
+ )  # end theme
```

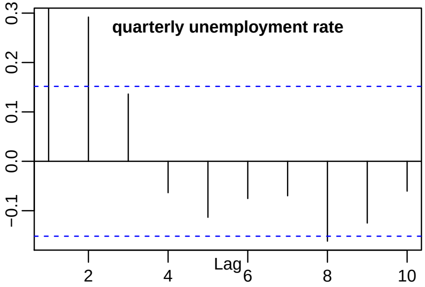


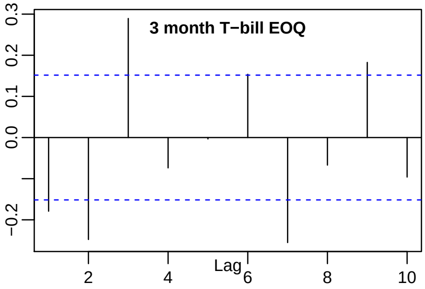
Autocorrelation of Econometric Data

Most econometric data displays a high degree of autocorrelation,

But time series of tradeable prices display very low autocorrelation,

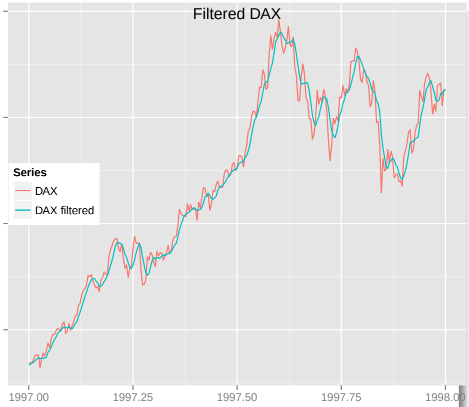
```
> macro_diff <- na.omit(diff(macro_zoo))
>
> acf_plus(coredata(macro_diff[, "unemprate"]),
+     lag=10)
> title(main="quarterly unemployment rate",
+ line=-1)
>
> acf_plus(coredata(macro_diff[, "3mTbill"]),
+     lag=10)
> title(main="3 month T-bill EOQ", line=-1)
```





Filtering Time Series

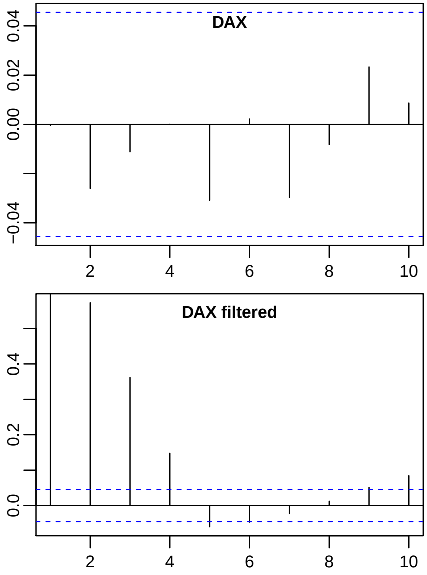
```
> library(zoo)  # load zoo
> library(ggplot2)  # load ggplot2
> library(gridExtra)  # load gridExtra
> # extract DAX time series
> dax_ts <- EuStockMarkets[, 1]
> # filter past values only (sides=1)
> dax_filt <- filter(dax_ts,
+     filter=rep(1/5,5), sides=1)
> # coerce to zoo and merge the time series
> dax_filt <- cbind(as.zoo(dax_ts),
+                 as.zoo(dax_filt))
> colnames(dax_filt) <- c("DAX", "DAX filtered")
> dax_data <- window(dax_filt,
+                  start=1997, end=1998)
> autoplot(  # plot ggplot2
+     dax_data, main="Filtered DAX",
+     facets=NULL) +  # end autoplot
+ xlab("") + ylab("") +
+ theme(  # modify plot theme
+     legend.position=c(0.1, 0.5),
+     plot.title=element_text(vjust=-2.0),
+     plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"),
+     plot.background=element_blank(),
+     axis.text.y=element_blank()
+     )  # end theme
> # end ggplot2
```



Autocorrelation Function of Filtered Time Series

Filtering a time series creates autocorrelations,

```
> re_turns <- na.omit(diff(log(dax_filt)))
> par(mfrow=c(2,1))  # set plot panels
>
> acf_plus(coredata(re_turns[, 1]), lag=10,
+    xlab="")
> title(main="DAX", line=-1)
>
> acf_plus(coredata(re_turns[, 2]), lag=10,
+    xlab="")
> title(main="DAX filtered", line=-1)
```



Partial Autocorrelation Function

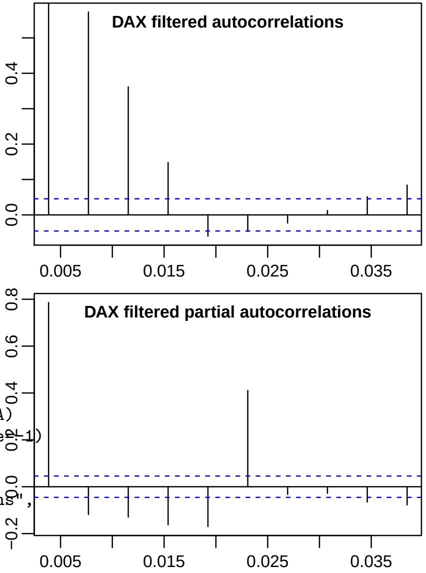
An autocorrelation of lag 1 creates autocorrelations of lag 2, 3, ..., which may obscure higher order autocorrelations,

A linear combination of a time series and its lag can be created, such that its lag 1 autocorrelation is zero,

The lag 2 autocorrelation of this new series is called the *partial autocorrelation* of lag 2, and represents the true second order autocorrelation,

The *partial autocorrelation* of lag k is the autocorrelation lag k, after all the autocorrelations of lag 1,..., k-1 have been removed,

```
> par(mfrow=c(2,1))  # set plot panels
> # autocorrelation from "stats"
> acf_plus(re_turns[, 2], lag=10, xlab=NA, ylab=NA)
> title(main="DAX filtered autocorrelations", line=-1)
> # partial autocorrelation
> pacf(re_turns[, 2], lag=10, xlab=NA, ylab=NA)
> title(main="DAX filtered partial autocorrelations",
+       line=-1)
```



Simulating Autoregressive Processes

An *autoregressive* time series process *AR(p)* of order p is defined as:

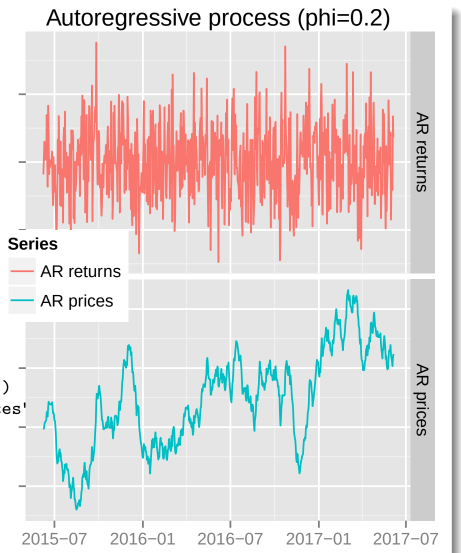
$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_p r_{i-p} + \varepsilon_i$$

Where the ε_i are independent random variables with zero mean and constant variance,

The *AR(p)* process is a special case of an *ARIMA* process,

The function `arima.sim()` simulates *ARIMA* processes,

```
> in_dex <- Sys.Date() + 0:728  # two year daily
> set.seed(1121)  # reset random numbers
> zoo_arima <- zoo(  # AR time series of returns
+   x=arima.sim(n=729, model=list(ar=0.2)),
+   order.by=in_dex)  # zoo_arima
> zoo_arima <- cbind(zoo_arima, cumsum(zoo_arima))
> colnames(zoo_arima) <- c("AR returns", "AR prices'
> autoplot(object=zoo_arima, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(
+   legend.position=c(0.1, 0.5),
+   plot.background=element_blank(),
```

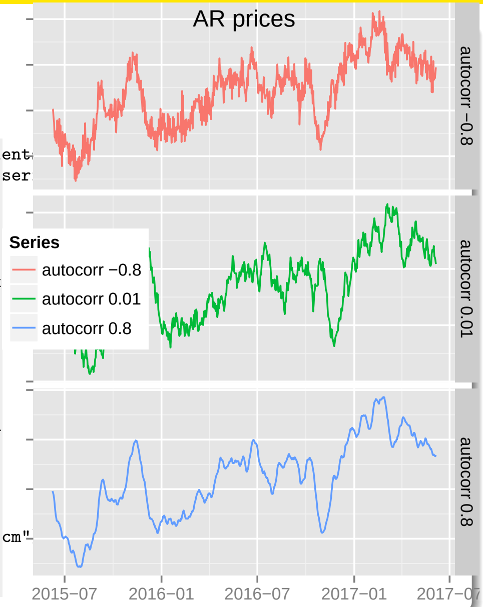


Examples of Autoregressive Processes

The "model" argument contains a list of *ARIMA* coefficients $\{\varphi_i\}$,

Positive coefficient values cause positive *autocorrelation*, and vice cersa,

```
> ar_coeff <- c(-0.8, 0.01, 0.8)  # AR coefficient
> zoo_arima <- sapply(  # create three AR time ser
+   ar_coeff, function(phi) {
+     set.seed(1121)  # reset random numbers
+     arima.sim(n=729, model=list(ar=phi))
+   } )
> zoo_arima <- zoo(x=zoo_arima, order.by=in_dex
> # convert returns to prices
> zoo_arima <- cumsum(zoo_arima)
> colnames(zoo_arima) <-
+   paste("autocorr", ar_coeff)
> autoplot(zoo_arima, main="AR prices",
+    facets=Series ~ .) +
+     facet_grid(Series ~ ., scales="free_y") +
+ xlab("") +
+ theme(
+   legend.position=c(0.1, 0.5),
+   plot.title=element_text(vjust=-2.0),
+   plot.margin=unit(c(-0.5, 0.0, -0.5, 0.0), "cm"
+   plot.background=element_blank(),
+   axis.text.y=element_blank())
```



Autocorrelation of Autoregressive Processes

An *autoregressive* process of order *one AR(1)* is defined by the formula: $r_i = \varphi_1 r_{i-1} + \varepsilon_i$

An *AR(1)* process can be simulated recursively as follows:

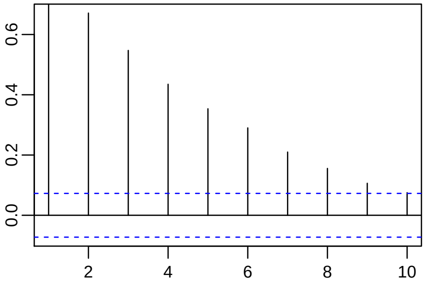
$$r_1 = \varepsilon_1$$
$$r_2 = \varphi_1 r_1 + \varepsilon_2 = \varepsilon_2 + \varphi_1 \varepsilon_1$$
$$r_3 = \varepsilon_3 + \varphi_1 \varepsilon_2 + \varphi_1^2 \varepsilon_1$$
$$r_4 = \varepsilon_4 + \varphi_1 \varepsilon_3 + \varphi_1^2 \varepsilon_2 + \varphi_1^3 \varepsilon_1$$

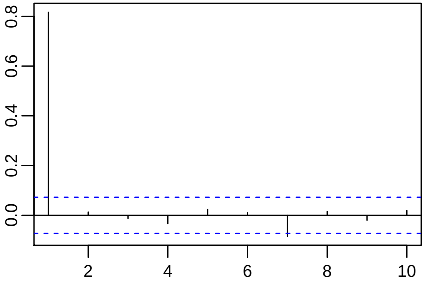
If $\varphi_1 < 1.0$ then the influence of any single shock ε_i decays exponentially,

If $\varphi_1 = 1.0$ then the influence of any single shock ε_i persists forever, and the variance of r_i increases linearly with time,

An *AR(1)* process has an exponentially declining ACF and a non-zero PACF at lag one,

```
> # simulate AR(1) process
> ari_ma <- arima.sim(n=729, model=list(ar=0.8))
> # ACF of AR(1) process
> acf_plus(ari_ma, lag=10, xlab="", ylab="",
+    main="ACF of AR(1) process")
> # PACF of AR(1) process
> pacf(ari_ma, lag=10, xlab="", ylab="",
+      main="PACF of AR(1) process")
```





Stationary Processes and Their Characteristic Equations

A process is *stationary* if its probability distribution does not change with time,

Stationary processes have a constant mean and variance,

The *autoregressive* process *AR(p)*:
$p_i = \varphi_1 p_{i-1} + \varphi_2 p_{i-2} + \ldots + \varphi_p p_{i-p} + \varepsilon_i$

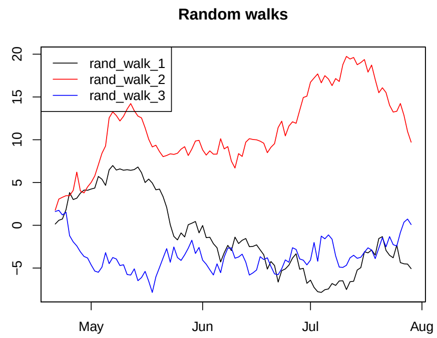
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_p z^p = 0$

An autoregressive process is stationary only if the absolute values of all the roots of its characteristic equation are greater than 1,

An *AR(1)* process: $p_i = \varphi_1 p_{i-1} + \varepsilon_i$ has the following characteristic equation: $1 - \varphi_1 z = 0$, with a root equal to: $z = 1/\varphi_1$,

If $\varphi_1 = 1$, then the characteristic equation has a *unit root*: $z = 1/\varphi_1$, and therefore isn't stationary, and the process follows:
$p_i = p_{i-1} + \varepsilon_i$,

The above is called a *Wiener* process (Brownian motion, random walk), and it's an example of a *unit-root* process,



```
> rand_walk <- cumsum(zoo(matrix(rnorm(3*100), n
+               order.by=(Sys.Date()+0:99)))
> colnames(rand_walk) <-
+   paste("rand_walk", 1:3, sep="_")
> plot(rand_walk, main="Random walks",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue"))
> # add legend
> legend(x="topleft",
+  legend=colnames(rand_walk),
+  col=c("black", "red", "blue"), lty=1)
```

Integrated and Unit-root Processes

The variance of the *AR(1)* process $p_i = \varphi_1 p_{i-1} + \varepsilon$ is equal to:

$$\sigma^2 = \mathbb{E}[p_i^2] = \frac{\sigma_\varepsilon^2}{(1 - \varphi_1^2)}$$

If $\varphi_1 = 1$, then the process becomes a *Wiener* process, which has a *unit-root*, with its *variance* growing infinite over time, so the process isn't stationary,

The variance of the *Wiener* process $p_i = p_{i-1} + \varepsilon$ is proportional to time: $\sigma_i^2 = \mathbb{E}[p_i^2] = i\sigma_\varepsilon^2$,

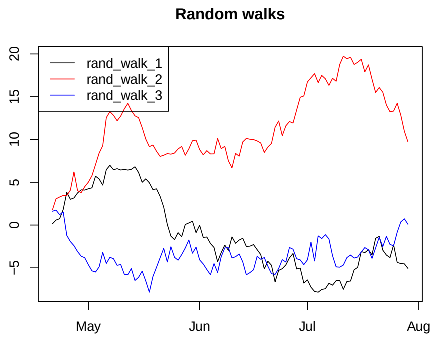
Asset prices follow an *integrated* process with respect to asset returns:

$$p_n = \sum_{i=1}^{n} r_i$$

If returns follow an *AR(1)* process:

$$r_i = \varphi_1 r_{i-1} + \varepsilon_i$$

Then asset prices follow the process:



```
> rand_walk <- cumsum(zoo(matrix(rnorm(3*100), n
+               order.by=(Sys.Date()+0:99)))
> colnames(rand_walk) <-
+   paste("rand_walk", 1:3, sep="_")
> plot(rand_walk, main="Random walks",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue"))
> # add legend
> legend(x="topleft",
+  legend=colnames(rand_walk),
+  col=c("black", "red", "blue"), lty=1)
```

Dickey-Fuller Test for Unit-roots

The *Dickey-Fuller* and *Augmented Dickey-Fuller* tests are designed to test the *null hypothesis* that a time series process has a *unit root*,

The *Augmented Dickey-Fuller* (*ADF*) test fits the following regression model, designed to determine if the time series exhibits mean reversion:

$$r_i = \gamma p_{i-1} + \varphi_2 r_{i-1} + \ldots + \varphi_p r_{i-p} + \varepsilon_i$$

where $p_i = p_{i-1} + r_i$, so that:

$$p_i = (1 + \gamma) p_{i-1} + \varphi_2 r_{i-1} + \ldots + \varphi_p r_{i-p} + \varepsilon_i$$

If the mean reversion parameter is negative: $\gamma < 0$, then the time series of prices has no *unit root*,

The *null hypothesis* is that the price process has a unit root ($\gamma = 0$, no mean reversion), while the alternative hypothesis is that the price process is stationary ($\gamma < 0$, mean reversion),

```
> # simulate AR(1) process
> set.seed(1121)
> ari_ma <- arima.sim(n=729, model=list(ar=0.8))
> tseries::adf.test(ari_ma)
> set.seed(1121)
> ari_ma <- arima.sim(n=10000, model=list(ar=0.8
> tseries::adf.test(ari_ma)
> # simulate Brownian motion
>
> rand_walk <- cumsum(rnorm(729))
> tseries::adf.test(rand_walk)
> set.seed(1121)
> rand_walk <- cumsum(rnorm(10000))
> tseries::adf.test(rand_walk)
```

The *ADF* test statistic is equal to the *t*-value of the γ parameter: $t_\gamma = \hat{\gamma} / SE_\gamma$ (which follows its own distribution, different from the t-distribution),

The *ADF* test is weak in the sense that it requires a lot of data to identify a *unit root* process,

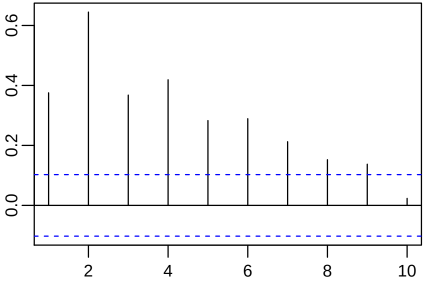
Identification of Autoregressive Processes

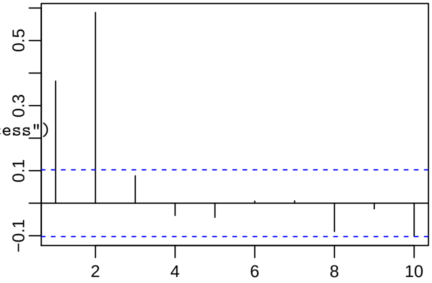
An $AR(3)$ process of order *three* is defined by the formula:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \varphi_3 r_{i-3} + \varepsilon_i$$

Autoregressive processes *AR(p)* of order p have an exponentially declining ACF and a non-zero PACF up to lag p,

```
> ar3_zoo <- zoo(  # AR(3) time series of returns
+   x=arima.sim(n=365,
+     model=list(ar=c(0.1, 0.5, 0.1))),
+   order.by=in_dex)  # zoo_arima
> # ACF of AR(3) process
> acf_plus(ar3_zoo, lag=10,
+  xlab="", ylab="", main="ACF of AR(3) process")
>
> # PACF of AR(3) process
> pacf(ar3_zoo, lag=10,
+      xlab="", ylab="", main="PACF of AR(3) process")
```





Fitting Autoregressive Models

The function `arima()` from the base package *stats* fits a specified ARIMA model to a univariate time series,

The function `auto.arima()` from the package *forecast* automatically fits an ARIMA model to a univariate time series,

```
> ar3_zoo <- arima.sim(n=1000,
+       model=list(ar=c(0.1, 0.3, 0.1)))
> arima(ar3_zoo, order = c(5,0,0))  # fit AR(5) model
> library(forecast)  # load forecast
> auto.arima(ar3_zoo)  # fit ARIMA model
```

Ornstein-Uhlenbeck Process

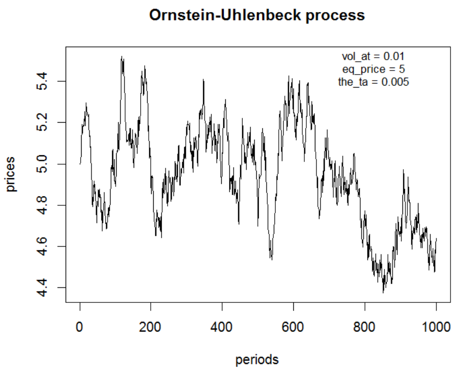
Under the *Ornstein-Uhlenbeck* process, the percentage returns $\mathrm{d}\log P$ are proportional to the difference between the equilibrium price μ minus the current price P_t:

$$\mathrm{d}\log P_t = \theta(\mu - P_t)\mathrm{d}t + \sigma \mathrm{d}W_t$$

Where θ is the strength of mean reversion, and σ is the volatility,

The *Ornstein-Uhlenbeck* process must be simulated using a `for()` loop, since it is path-dependent,

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

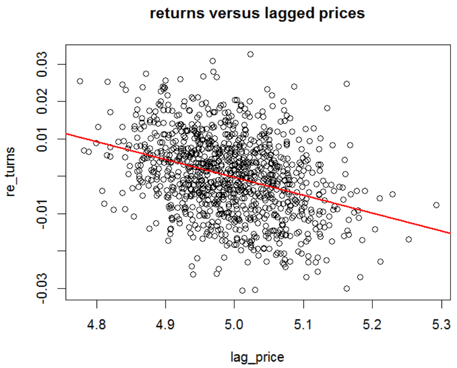


```
> plot(price_s, type="l",
+      xlab="periods", ylab="prices",
+      main="Ornstein-Uhlenbeck process")
> legend("topright",
+  title=paste(c(paste0("vol_at = ", vol_at),
+               paste0("eq_price = ", eq_price)
+               paste0("the_ta = ", the_ta)),
+             collapse="\n"),
+  legend="", cex=0.8,
+  inset=0.1, bg="white", bty="n")
```

Ornstein-Uhlenbeck Process Mean Reversion

Under the *Ornstein-Uhlenbeck* process, the returns are negatively correlated to the lagged prices,

```
> # define Ornstein-Uhlenbeck parameters
> eq_price <- 5.0; the_ta <- 0.05
> len_gth <- 1000
> # simulate Ornstein-Uhlenbeck process
> re_turns <- numeric(len_gth)
> price_s <- numeric(len_gth)
> price_s[1] <- 5.0
> set.seed(1121)  # reset random numbers
> for (i in 2:len_gth) {
+   re_turns[i] <- the_ta*(eq_price - price_s[i-
+     vol_at*rnorm(1)
+   price_s[i] <- price_s[i-1] * exp(re_turns[i]
+ }  # end for
> re_turns <- rutils::diff_it(log(price_s))
> lag_price <- rutils::lag_it(price_s)
> lag_price[1] <- lag_price[2]
> for_mula <- re_turns ~ lag_price
> l_m <- lm(for_mula)
> summary(l_m)
> # plot regression
> plot(for_mula, main="returns versus lagged prices")
> abline(l_m, lwd=2, col="red")
```



Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

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

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

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

Linear Regression of Returns

The returns of *XLP* and *VTI* are highly correlated because they are driven by common market factors of returns,

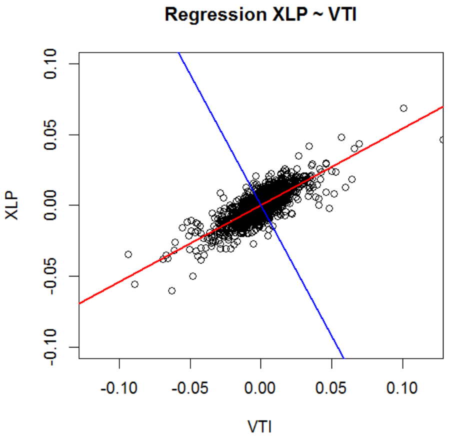
The *t*-statistic (*t*-value) is the ratio of the estimated value divided by its standard error,

The *p*-value is the probability of obtaining the observed value of the *t*-statistic, or more extreme values,

```
> # specify formula and perform regression
> reg_formula <- XLP ~ VTI
> reg_model <- lm(reg_formula,
+              data=rutils::env_etf$re_turns)
> # get regression coefficients
> coef(summary(reg_model))
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.000127   8.92e-05    1.42    0.156
VTI         0.552105   7.53e-03   73.33    0.000
> # Durbin-Watson test of autocorrelation of res
> lmtest::dwtest(reg_model)

Durbin-Watson test

data:  reg_model
DW = 2, p-value = 1
alternative hypothesis: true autocorrelation is
```



```
> # plot scatterplot of returns with aspect rati
> plot(reg_formula, data=rutils::env_etf$re_turn
+      xlim=c(-0.1, 0.1), ylim=c(-0.1, 0.1),
+      asp=1, main="Regression XLP ~ VTI")
> # add regression line and perpendicular line
> abline(reg_model, lwd=2, col="red")
> abline(a=0, b=-1/coef(summary(reg_model))[2, 1
```

Linear Regression Summary Statistics

```
> re_turns <- na.omit(rutils::env_etf$re_turns)
> # perform regressions and collect statistics
> etf_reg_stats <- sapply(colnames(re_turns)[-1],
+                  function(etf_name) {
+ # specify regression formula
+   reg_formula <- as.formula(
+     paste(etf_name, "~ VTI"))
+ # perform regression
+   reg_model <- lm(reg_formula, data=re_turns)
+ # get regression summary
+   reg_model_sum <- summary(reg_model)
+ # collect regression statistics
+   etf_reg_stats <- with(reg_model_sum,
+     c(alpha=coefficients[1, 1],
+ p_alpha=coefficients[1, 4],
+ beta=coefficients[2, 1],
+ p_beta=coefficients[2, 4]))
+   etf_reg_stats <- c(etf_reg_stats,
+          p_dw=lmtest::dwtest(reg_model)$p.value)
+   etf_reg_stats
+ })  # end sapply
> etf_reg_stats <- t(etf_reg_stats)
> # sort by p_alpha
> etf_reg_stats <- etf_reg_stats[
+   order(etf_reg_stats[, "p_alpha"]), ]
```

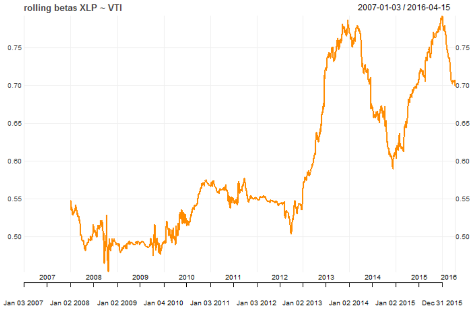
```
> etf_reg_stats[, 1:3]
```

Rolling Beta Regressions Over Time

The function `rollapply()` allows performing regressions over a rolling window,

The function `roll_lm()` from package *roll* performs rolling regressions in C++, in parallel, and is therefore much faster than function `rollapply()`,

```
> library(HighFreq)
> # specify regression formula
> reg_formula <- XLP ~ VTI
> # perform rolling beta regressions every month
> beta_s <- rollapply(rutils::env_etf$re_turns,
+   FUN=function(de_sign)
+   coef(lm(reg_formula, data=de_sign))[2],
+   by=22, by.column=FALSE, align="right")
> beta_s <- na.omit(beta_s)
> # plot beta_s in x11() window
> x11(width=(wid_th <- 6), height=(hei_ght <- 4)
> chart_Series(x=beta_s[, "VTI"],
+   name=paste("rolling betas", format(reg_formu
> # perform daily rolling beta regressions in pa
> library(roll)
> beta_s <- roll_lm(x=rutils::env_etf$re_turns[
+             y=rutils::env_etf$re_turns[, "XLP"
+             width=252)$coefficients
```



```
> # compare speed of rollapply() versus roll_lm(
> library(microbenchmark)
> da_ta <- rutils::env_etf$re_turns["2012", c("V
> summary(microbenchmark(
+   rollapply=rollapply(da_ta, width=22,
+ FUN=function(de_sign)
+ coef(lm(reg_formula, data=de_sign))[2],
+   by.column=FALSE, align="right"),
+   roll_lm=roll_lm(x=da_ta[, "VTI"],
+              y=da_ta[, "XLP"],
+              width=22)$coefficients,
+   times=10))[, c(1, 4, 5)]  # end microbenchma
```

Capital Asset Pricing Model (*CAPM*)

The *Capital Asset Pricing Model* decomposes asset returns into *systematic* returns (proportional to the market returns) and *idiosyncratic* returns (uncorrelated to market returns):

$$R - R_f = \alpha + \beta(R_m - R_f) + \varepsilon$$

Where R_m are the market returns, and R_f are the risk-free returns,

The *systematic* risk and returns are proportional to β,

β can be obtained from linear regression, and is proportional to the correlation of returns between the asset and the market:

$$\beta = \frac{\sum_{i=1}^{n} (R_i - \bar{R})(R_{i,m} - \bar{R_m})}{\sum_{i=1}^{n} (R_{i,m} - \bar{R_m})^2}$$

The *CAPM* model states that if an asset has higher β risk, then it should earn higher *systematic* returns,

```
> library(PerformanceAnalytics)
> CAPM.beta(Ra=re_turns[, "XLP"],
+     Rb=re_turns[, "VTI"])
> CAPM.beta.bull(Ra=re_turns[, "XLP"],
+   Rb=re_turns[, "VTI"])
> CAPM.beta.bear(Ra=re_turns[, "XLP"],
+   Rb=re_turns[, "VTI"])
> CAPM.alpha(Ra=re_turns[, "XLP"],
+      Rb=re_turns[, "VTI"])
```

The *idiosyncratic* returns are equal to the sum of α plus ε,

Alpha (α) are the returns in excess of *systematic* returns, that can be attributed to portfolio selection or active manager performance,

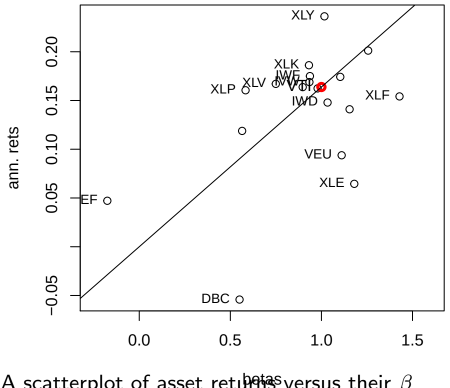
The *idiosyncratic* risk (equal to ε) is uncorrelated to the *systematic* risk, and can be reduced through portfolio diversification,

The Security Market Line

According to the *CAPM* model, assets should earn a *systematic* return proportional to their *systematic* risk (β),

The *Security Market Line* (SML) represents the linear relationship between *systematic* risk (β) and return, for different stocks,

```
> etf_betas <- sapply(
+   re_turns[, colnames(re_turns)!="VXX"],
+   CAPM.beta, Rb=re_turns[, "VTI"])
> etf_annrets <- sapply(
+   re_turns[, colnames(re_turns)!="VXX"],
+   Return.annualized)
> # plot scatterplot
> plot(etf_annrets ~ etf_betas, xlab="betas",
+        ylab="ann. rets", xlim=c(-0.25, 1.6))
> points(x=1, y=etf_annrets["VTI"], col="red",
+  lwd=3, pch=21)
> abline(a=0, b=etf_annrets["VTI"])
> label_names <- rownames(etf_reg_stats)[1:13]
> # add labels
> text(x=1, y=etf_annrets["VTI"], labels="VTI",
+      pos=2)
> text(x=etf_betas[label_names],
+      y=etf_annrets[label_names],
+      labels=label_names, pos=2, cex=0.8)
```



A scatterplot of asset returns versus their β shows which assets earn a positive α, and which don't,

If an asset lies on the *SML*, then its returns are mostly *systematic*, and its α is equal to zero,

Assets above the *SML* have a positive α), and those below have a negative α),

Risk-adjusted Performance Measurement

The *Treynor* ratio measures the excess returns per unit of *systematic* risk (β), and is equal to the excess returns (over a risk-free return) divided by the β:

$$T_r = \frac{E[R - R_f]}{\beta}$$

The *Treynor* ratio is similar to the *Sharpe* ratio, with the difference that its denominator represents only *systematic* risk, not total risk,

The *Information* ratio is equal to the excess returns (over a benchmark) divided by the *tracking error* (standard deviation of excess returns):

$$I_r = \frac{E[R - R_b]}{\sqrt{\sum_{i=1}^{n} (R_i - R_{i,b})^2}}$$

The *Information* ratio measures the amount of outperformance versus the benchmark, and the consistency of outperformance,

```
> library(PerformanceAnalytics)
> TreynorRatio(Ra=re_turns[, "XLP"],
+      Rb=re_turns[, "VTI"])
>
> InformationRatio(Ra=re_turns[, "XLP"],
+      Rb=re_turns[, "VTI"])
```

CAPM Summary Statistics

```
> table.CAPM(Ra=re_turns[, c("XLP", "XLF")],
+       Rb=re_turns[, "VTI"], scale=252)
```

```
> capm_stats <- table.CAPM(Ra=re_turns[, colnames(re_turns)!="VTI"],
+         Rb=re_turns[, "VTI"], scale=252)
> colnames(capm_stats) <-
+   sapply(colnames(capm_stats),
+   function (str) {strsplit(str, split=" ")[[1]][1]})
> capm_stats <- as.matrix(capm_stats)
> capm_stats <- t(capm_stats)
> capm_stats <- capm_stats[
+   order(capm_stats[, "Annualized Alpha"],
+   decreasing=TRUE), ]
> # copy capm_stats into env_etf and save to .RData file
> assign("capm_stats", capm_stats, envir=env_etf)
> save(env_etf, file='etf_data.RData')
```

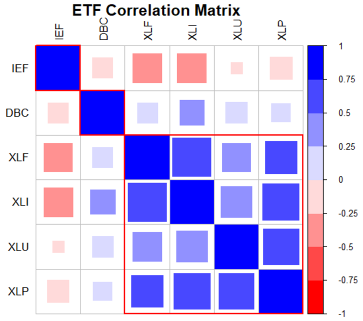
```
> capm_stats[, c("Information Ratio", "Annua
```

Covariance Matrix of ETF Returns

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

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

```
> # Select ETF symbols
> sym_bols <- c("IEF", "DBC", "XLU", "XLF", "XLP
> # calculate ETF prices and simple returns (not
> price_s <- rutils::env_etf$price_s[, sym_bols]
> price_s <- zoo::na.locf(price_s)
> price_s <- zoo::na.locf(price_s, fromLast=TRUE
> in_dex <- index(price_s)
> re_turns <- rutils::diff_it(price_s)
> # de-mean and scale the returns
> re_turns <- t(t(re_turns) - colMeans(re_turns),
> re_turns <- t(t(re_turns) / sqrt(colSums(re_turns^2)/(NROW(re_turns)-1)))
> re_turns <- xts(re_turns, in_dex)
> # alternative de-mean and scale the returns
> # re_turns <- rutils::diff_it(price_s)
> # re_turns <- scale(re_turns, center=TRUE, sca
> # re_turns <- xts(re_turns, in_dex)
> # or
> # re_turns <- lapply(re_turns, function(x) {x
> # re_turns <- rutils::do_call(cbind, re_turns)
> # re_turns <- apply(re_turns, 2, scale)
> # covariance matrix and variance vector of ret
> cov_mat <- cov(re_turns)
> [illegible]
```



```
> # plot the correlation matrix
> col_ors <- colorRampPalette(c("red", "white",
> corrplot(cor_mat, title="Correlation Matrix",
+     tl.col="black", tl.cex=0.8, mar = c(0,0,1,
+     method="square", col=col_ors(NCOL(cor_mat)
+     cl.offset=0.75, cl.cex=0.7,
+     cl.align.text="l", cl.ratio=0.25)
> # draw rectangles on the correlation matrix pl
> corrRect.hclust(cor_mat, k=NCOL(cor_mat) %/% 2
+     method="complete", col="red")
```

Principal Component Vectors

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

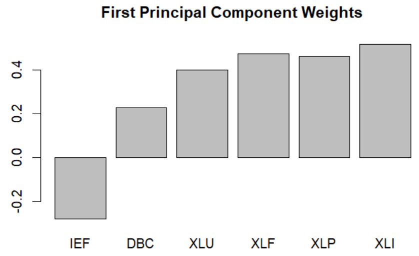
$$\mathbf{pc}_j = \sum_{i=1}^{k} w_{ij} \, \mathbf{r}_i$$

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

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

$$\mathbf{w}_j = \arg\max \left\{ \mathbf{pc}_j^T \, \mathbf{pc}_j \right\}$$
$$\mathbf{pc}_i^T \, \mathbf{pc}_j = 0 \, (i \neq j)$$

```
> # create initial vector of portfolio weights
> n_weights <- NROW(sym_bols)
> weight_s <- rep(1/sqrt(n_weights), n_weights)
> names(weight_s) <- sym_bols
> # objective function equal to minus portfolio
> object_ive <- function(weight_s, re_turns) {
+   portf_rets <- re_turns %*% weight_s
+   -sum(portf_rets^2) +
+     1e7*(1 - sum(weight_s^2))^2
+ } # end object_ive
```



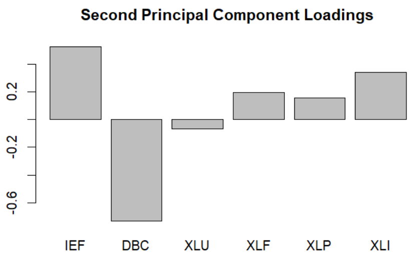
```
> # find weights with maximum variance
> optim_run <- optim(par=weight_s,
+                    fn=object_ive,
+                    re_turns=re_turns,
+                    method="L-BFGS-B",
+                    upper=rep(10.0, n_weights),
+                    lower=rep(-10.0, n_weights))
> # optimal weights and maximum variance
> weight_s <- optim_run$par
> -object_ive(weight_s, re_turns)
> # plot first principal component weights
> barplot(weight_s, names.arg=names(weight_s),
+   xlab="", ylab="",
+   main="First Principal Component Weights")
```

Higher Order Principal Components

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

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

```
> # pc1 weights and returns
> weights_1 <- weight_s
> pc_1 <- re_turns %*% weights_1
> # redefine objective function
> object_ive <- function(weight_s, re_turns) {
+   portf_rets <- re_turns %*% weight_s
+   -sum(portf_rets^2) +
+     1e7*(1 - sum(weight_s^2))^2 +
+     1e9*(sum(pc_1*portf_rets))^2
+ }  # end object_ive
> # find second principal component weights
> optim_run <- optim(par=weight_s,
+                    fn=object_ive,
+                    re_turns=re_turns,
+                    method="L-BFGS-B",
+                    upper=rep(10.0, n_weights),
+                    lower=rep(-10.0, n_weights))
```



```
> # pc2 weights and returns
> weights_2 <- optim_run$par
> pc_2 <- re_turns %*% weights_2
> sum(pc_1*pc_2)
> # plot second principal component loadings
> barplot(weights_2, names.arg=names(weights_2),
+   xlab="", ylab="",
+   main="Second Principal Component Loadings")
```

Eigenvalues of the Covariance Matrix

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

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

Where λ is a *Lagrange multiplier*,

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

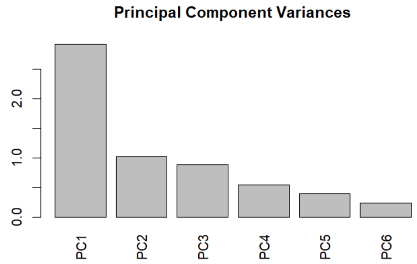
$$\mathbb{C} \, w = \lambda \, w$$

The above is the *eigenvalue* equation of the covariance matrix $\mathbb{C}$,

The optimal weights w form an *eigenvector*, and λ is the *eigenvalue* corresponding to the *eigenvector* w,

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

$$\sum_{i=1}^{k} \lambda_i = \sum_{i=1}^{k} r_i^T \, r_i$$



```
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(cov_mat)
> ei_gen$vectors
> weights_1
> weights_2
> ei_gen$values[1]
> var(pc_1)
> (cov_mat %*% weights_1) / weights_1
> ei_gen$values[2]
> var(pc_2)
> (cov_mat %*% weights_2) / weights_2
> sum(vari_ance)
> sum(ei_gen$values)
> barplot(ei_gen$values, # plot eigenvalues
```

Principal Component Analysis of ETF Returns

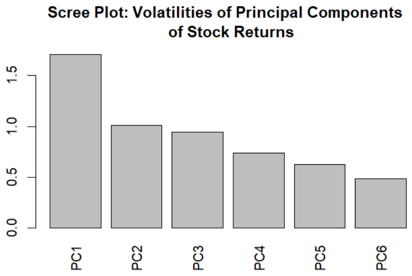
Principal Component Analysis (*PCA*) is a *dimensionality reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series,

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

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

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

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



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

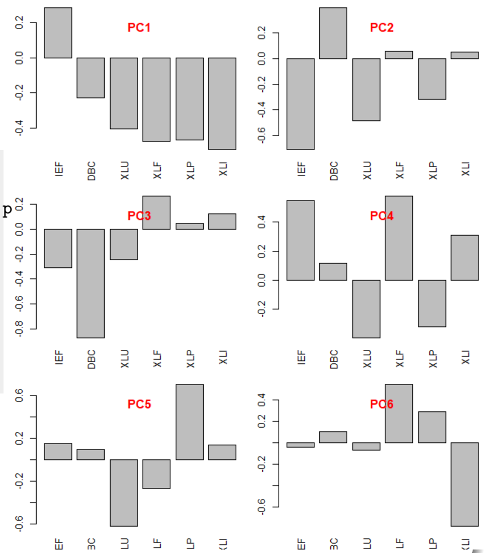
```
> # perform principal component analysis PCA
> pc_a <- prcomp(re_turns, scale=TRUE)
> # plot standard deviations of principal compon
> barplot(pc_a$sdev,
+   names.arg=colnames(pc_a$rotation),
+   las=3, xlab="", ylab="",
+   main="Scree Plot: Volatilities of Principal
+   of Stock Returns")
```

Principal Component Loadings (Weights)

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

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

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

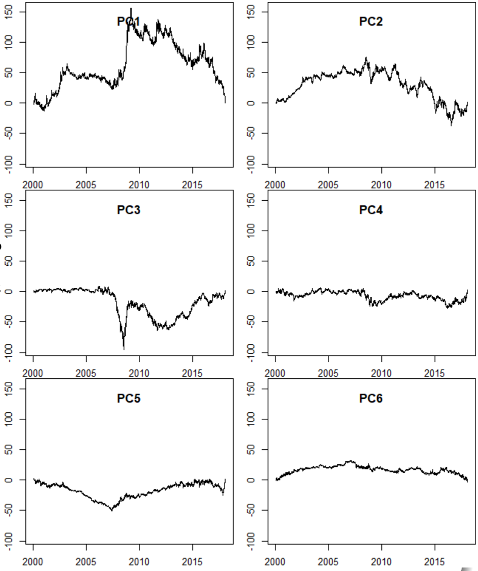


Principal Component Time Series

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

Higher order *principal components* are gradually less volatile,

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



Time Series from the *Principal Components*

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

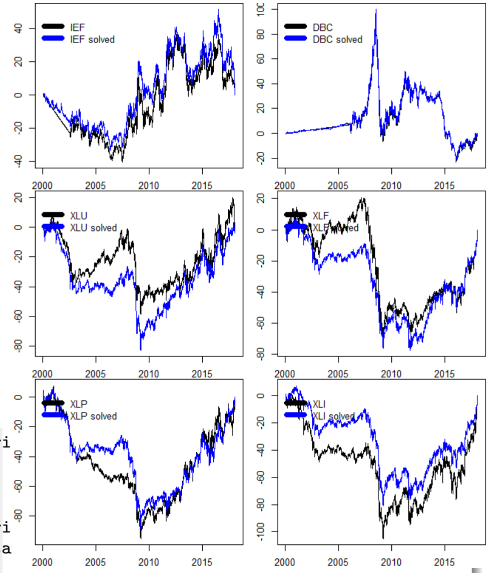
The original time series of returns can be calculated approximately from just the first few *principal components*, which demonstrates that *PCA* is a form of *dimensionality reduction*,

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

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

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

```
> # invert all the principal component time seri
> pca_rets <- re_turns %*% pc_a$rotation
> sol_ved <- pca_rets %*% solve(pc_a$rotation)
> all.equal(re_turns, sol_ved)
> # invert first 3 principal component time seri
> sol_ved <- pca_rets[, 1:3] %*% solve(pc_a$rota
> sol_ved <- xts::xts(sol_ved, in_dex)
> sol_ved <- xts:::cumsum.xts(sol_ved)
```



Principal Component Analysis Versus *Eigen Decomposition*

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

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

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

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

```
> # eigen decomposition of covariance matrix
> re_turns <- rutils::diff_it(price_s)
> cov_mat <- cov(re_turns)
> ei_gen <- eigen(cov_mat)
> # perform PCA without scaling
> pc_a <- prcomp(re_turns, scale=FALSE)
> # compare outputs
> all.equal(ei_gen$values, pc_a$sdev^2)
> all.equal(abs(unname(ei_gen$vectors)),
+     abs(unname(pc_a$rotation)))
> # eigen decomposition of correlation matrix
> cor_mat <- cor(re_turns)
> ei_gen <- eigen(cor_mat)
> # perform PCA with scaling
> pc_a <- prcomp(re_turns, scale=TRUE)
> # compare outputs
> all.equal(ei_gen$values, pc_a$sdev^2)
> all.equal(abs(unname(ei_gen$vectors)),
+     abs(unname(pc_a$rotation)))
```

Homework Assignment

Required

- Read all the lecture slides in *FRE7241_Lecture_4.pdf*, and run all the code in *FRE7241_Lecture_4.R*

Recommended

- TBA