Time Series Univariate

FRE6871 & FRE7241, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 30, 2022

NYU | TANDON SCHOOL OF ENGINEERING

Package *tseries* for Time Series Analysis

The package *tseries* contains functions for time series analysis and computational finance, such as:

- downloading historical data,
- plotting time series,
- calculating risk and performance measures,
- statistical *hypothesis testing*,
- calibrating models to time series,
- portfolio optimization,

Package *tseries* accepts time series of class `"ts"` and `"zoo"`, and also has its own class `"irts"` for irregular spaced time-series objects.

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

The function `zoo::coredata()` extracts the underlying numeric data from a complex data object.

```
> # Get documentation for package tseries
> packageDescription("tseries")  # Get short description
>
> help(package="tseries")  # Load help page
>
> library(tseries)  # Load package tseries
>
> data(package="tseries")  # List all datasets in "tseries"
>
> ls("package:tseries")  # List all objects in "tseries"
>
> detach("package:tseries")  # Remove tseries from search path
```

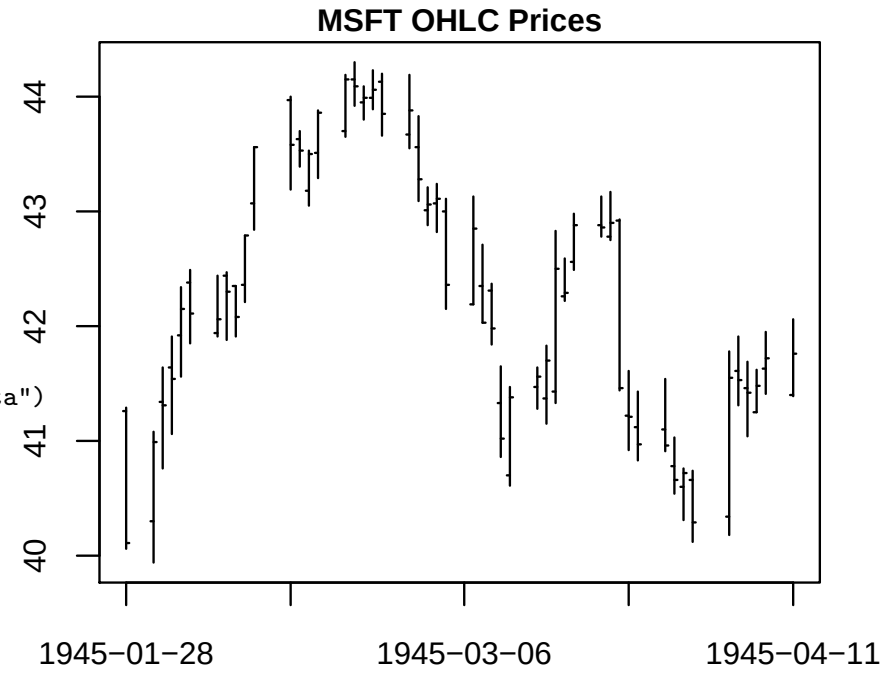
Plotting *OHLC* Time Series Using Package *tseries*

The package *tseries* contains functions for plotting time series:

- `seqplot.ts()` for plotting two time series in same panel.
- `plotOHLC()` for plotting *OHLC* time series.

The function `plotOHLC()` from package *tseries* plots *OHLC* time series.

```
> load(file="/Users/jerzy/Develop/lecture_slides/data/zoo_data.RData")
> # Get start and end dates
> dates <- time(stxts_adj)
> endd <- dates[NROW(dates)]
> startd <- round((4*endd + dates[1])/5)
> # Plot using plotOHLC
> plotOHLC(window(stxts_adj,
+           start=startd,
+           end=endd)[, 1:4],
+    xlab="", ylab="")
> title(main="MSFT OHLC Prices")
```



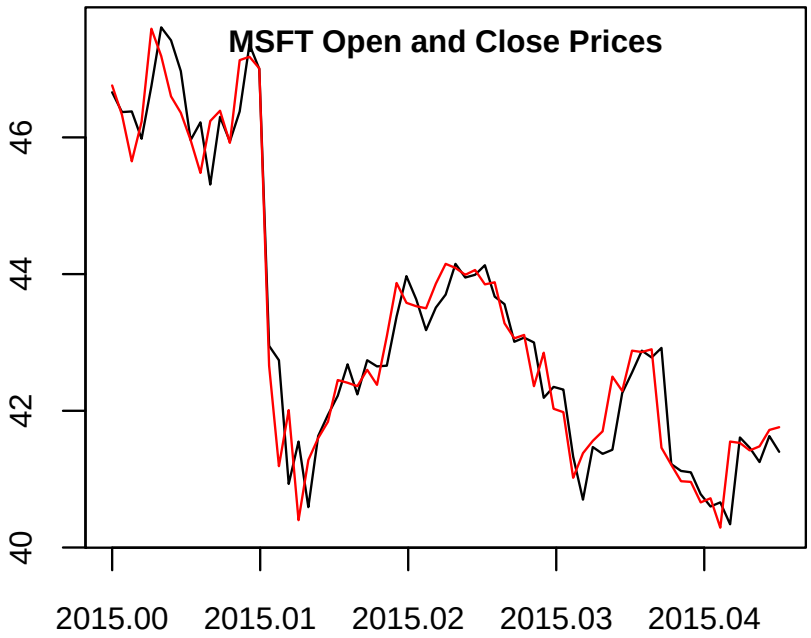
Plotting Two Time Series Using *tseries*

The function `seqplot.ts()` from package *tseries* plots two time series in same panel.

A *ts* time series can be created from a *zoo* time series using the function *ts()*, after extracting the data and date attributes from the *zoo* time series.

The function `decimal_date()` from package *lubridate* converts `POSIXct` objects into `numeric` *year-fraction* dates.

```
> library(lubridate)  # Load lubridate
> # Get start and end dates of msft
> startd <- lubridate::decimal_date(start(msft))
> endd <- lubridate::decimal_date(end(msft))
> # Calculate frequency of msft
> tstep <- NROW(msft)/(endd-startd)
> # Extract data from msft
> datav <- zoo::coredata(
+   window(msft, start=as.Date("2015-01-01"),
+    end=end(msft)))
> # Create ts object using ts()
> stxts <- ts(data=datav,
+   start=lubridate::decimal_date(as.Date("2015-01-01")),
+   frequency=tstep)
> seqplot.ts(x=stxts[, 1], y=stxts[, 4], xlab="", ylab="")
> title(main="MSFT Open and Close Prices", line=-1)
```



The function `zoo::coredata()` extracts the underlying numeric data from a complex data object.

Risk and Performance Estimation Using *tseries*

The package *tseries* contains functions for calculating risk and performance:

- `maxdrawdown()` for calculating the maximum drawdown,
- `sharpe()` for calculating the *Sharpe* ratio (defined as the excess return divided by the standard deviation),
- `sterling()` for calculating the *Sterling* ratio (defined as the return divided by the maximum drawdown),

```
> library(tseries)  # Load package tseries
> # Calculate maximum drawdown
> maxdrawdown(msft_adj[, "AdjClose"])
> max_drawd <- maxdrawdown(msft_adj[, "AdjClose"])
> zoo::index(msft_adj)[max_drawd$from]
> zoo::index(msft_adj)[max_drawd$to]
> # Calculate Sharpe ratio
> sharpe(msft_adj[, "AdjClose"])
> # Calculate Sterling ratio
> sterling(as.numeric(msft_adj[, "AdjClose"]))
```

Hypothesis Testing Using *tseries*

The package *tseries* contains functions for testing statistical hypothesis on time series:

- `jarque.bera.test()` *Jarque-Bera* test for normality of distribution of returns,
- `adf.test()` *Augmented Dickey-Fuller* test for existence of unit roots,
- `pp.test()` *Phillips-Perron* test for existence of unit roots,
- `kpss.test()` *KPSS* test for stationarity,
- `po.test()` *Phillips-Ouliaris* test for cointegration,
- `bds.test()` *BDS* test for randomness,

```
> msft <- suppressWarnings(  # Load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(msft)
> dim(msft)
> tail(msft, 4)
>
> # Calculate Sharpe ratio
> sharpe(msft[, "Close"], r=0.01)
> # Add title
> plot(msft[, "Close"], xlab="", ylab="")
> title(main="MSFT Close Prices", line=-1)
```

Calibrating Time Series Models Using *tseries*

The package *tseries* contains functions for calibrating models to time series:

- `garch()` for calibrating *GARCH* volatility models,
- `arma()` for calibrating `ARMA` models,

```
> library(tseries)  # Load package tseries
> msft <- suppressWarnings(  # Load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(msft)
> dim(msft)
> tail(msft, 4)
>
> # Calculate Sharpe ratio
> sharpe(msft[, "Close"], r=0.01)
> # Add title
> plot(msft[, "Close"], xlab="", ylab="")
> title(main="MSFT Close Prices", line=-1)
```

Portfolio Optimization Using *tseries*

The package *tseries* contains functions for miscellaneous functions:

`portfolio.optim()` for calculating mean-variance efficient portfolios.

```
> library(tseries)  # Load package tseries
> msft <- suppressWarnings(  # Load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(msft)
> dim(msft)
> tail(msft, 4)
>
> # Calculate Sharpe ratio
> sharpe(msft[, "Close"], r=0.01)
> # Add title
> plot(msft[, "Close"], xlab="", ylab="")
> title(main="MSFT Close Prices", line=-1)
```

Package *quantmod* for Quantitative Financial Modeling

Package *quantmod* for Quantitative Financial Modeling

The package *quantmod* is designed for downloading, manipulating, and visualizing *OHLC* time series data.

quantmod operates on time series of class `"xts"`, and provides many useful functions for building quantitative financial models:

- `getSymbols()` for downloading data from external sources (*Yahoo*, *FRED*, etc.),
- `getFinancials()` for downloading financial statements,
- `adjustOHLC()` for adjusting *OHLC* data,
- `Op()`, `Cl()`, `Vo()`, etc. for extracting *OHLC* data columns,
- `periodReturn()`, `dailyReturn()`, etc. for calculating periodic returns,
- `chartSeries()` for candlestick plots of *OHLC* data,
- `addBBands()`, `addMA()`, `addVo()`, etc. for adding technical indicators (Moving Averages, Bollinger Bands) and volume data to a plot,

```
> # Load package quantmod
> library(quantmod)
> # Get documentation for package quantmod
> # Get short description
> packageDescription("quantmod")
> # Load help page
> help(package="quantmod")
> # List all datasets in "quantmod"
> data(package="quantmod")
> # List all objects in "quantmod"
> ls("package:quantmod")
> # Remove quantmod from search path
> detach("package:quantmod")
```

Plotting *OHLC* Time Series Using `chartSeries()`

The function `chartSeries()` from package *quantmod* can produce a variety of plots for *OHLC* time series, including candlestick plots, bar plots, and line plots.

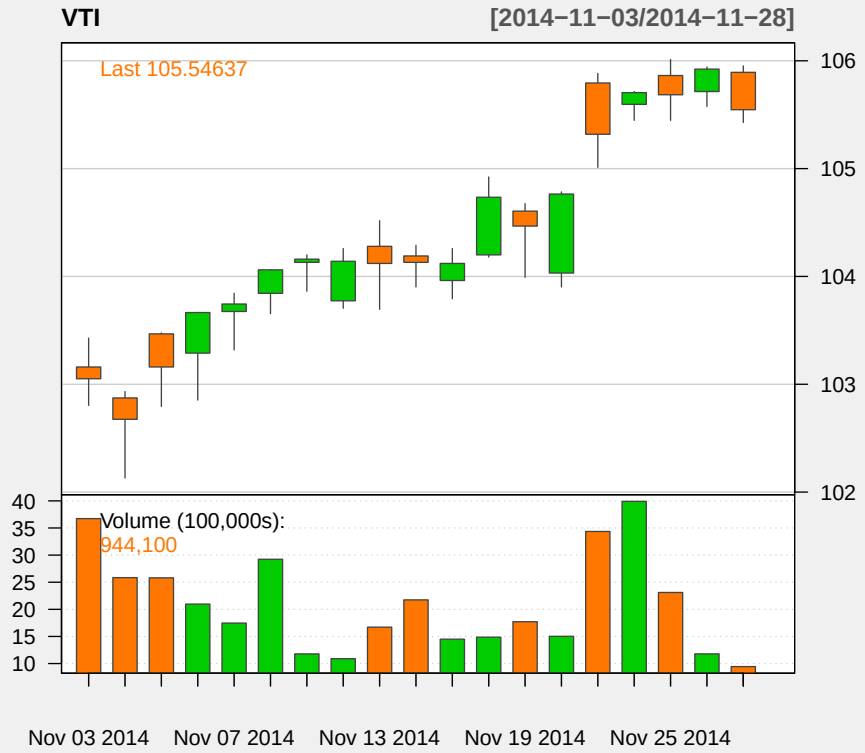
The argument `"type"` determines the type of plot (candlesticks, bars, or lines).

The argument `"theme"` accepts a `"chart.theme"` object, containing parameters that determine the plot appearance (colors, size, fonts).

`chartSeries()` automatically plots the volume data in a separate panel.

Candlestick plots are designed to visualize *OHLC* time series.

```
> # Plot OHLC candlechart with volume
> chartSeries(etfenv$VTI["2014-11"],
+       name="VTI",
+       theme=chartTheme("white"))
> # Plot OHLC bar chart with volume
> chartSeries(etfenv$VTI["2014-11"],
+       type="bars",
+       name="VTI",
+       theme=chartTheme("white"))
```



Each *candlestick* displays one period of data, and consists of a box representing the *Open* and *Close* prices, and a vertical line representing the *High* and *Low* prices.

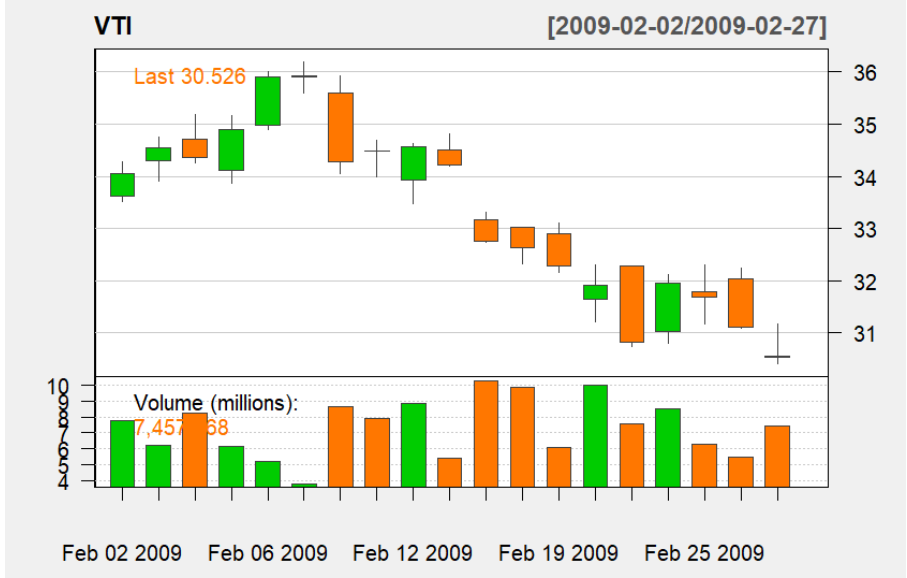
The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

Redrawing Plots Using reChart()

The function reChart() redraws plots using the same data set, but using additional parameters that control the plot appearance.

The argument "subset" allows subsetting the data to a smaller range of dates.

```
> # Plot OHLC candlechart with volume
> chartSeries(etfenv$VTI["2008-11/2009-04"], name="VTI")
> # Redraw plot only for Feb-2009, with white theme
> reChart(subset="2009-02", theme=chartTheme("white"))
```



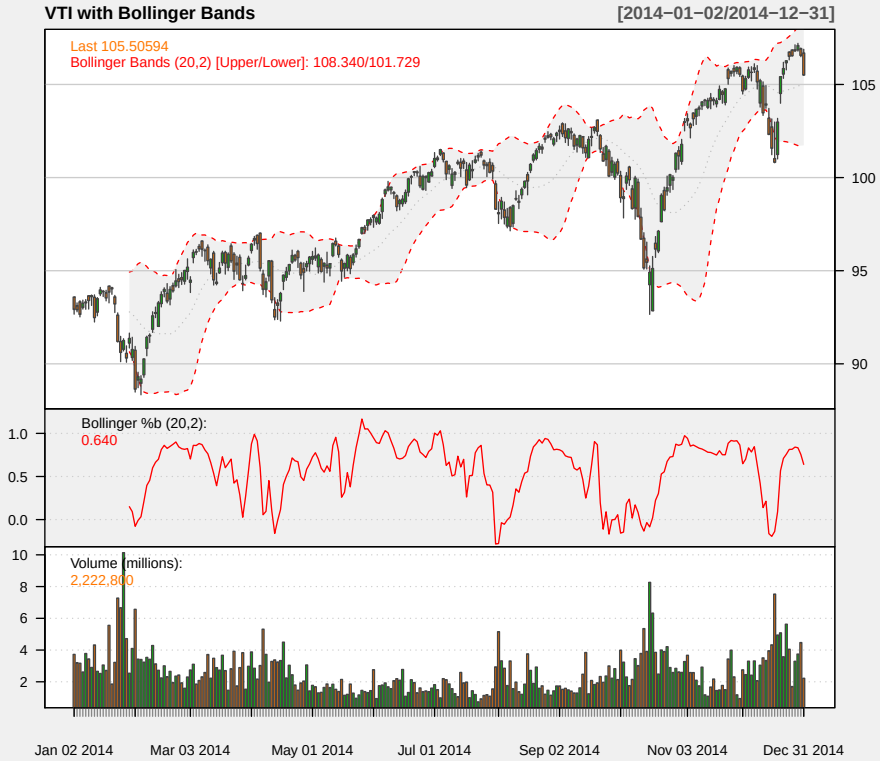
Plotting Technical Indicators Using chartSeries()

The argument "TA" allows adding technical indicators to the plot.

The technical indicators are functions provided by the package *TTR*.

The function newTA() allows defining new technical indicators.

```
> # Candlechart with Bollinger Bands
> chartSeries(etfenv$VTI["2014"],
+       TA="addBBands(): addBBands(draw='percent'): addVo()",
+       name="VTI with Bollinger Bands",
+       theme=chartTheme("white"))
> # Candlechart with two Moving Averages
> chartSeries(etfenv$VTI["2014"],
+       TA="addVo(): addEMA(10): addEMA(30)",
+       name="VTI with Moving Averages",
+       theme=chartTheme("white"))
> # Candlechart with Commodity Channel Index
> chartSeries(etfenv$VTI["2014"],
+       TA="addVo(): addBBands(): addCCI()",
+       name="VTI with Technical Indicators",
+       theme=chartTheme("white"))
```



Adding Indicators and Lines Using addTA()

The function addTA() adds indicators and lines to plots, and allows plotting lines representing a single vector of data.

The addTA() function argument "on" determines on which plot panel (subplot) the indicator is drawn.

"on=NA" is the default, and draws in a new plot panel below the existing plot.

"on=1" draws in the foreground of the main plot panel, and "on=-1" draws in the background.

```
> ohlc <- rutils::etfenv$VTI["2009-02/2009-03"]
> VTI_close <- quantmod::Cl(ohlc)
> VTI_vol <- quantmod::Vo(ohlc)
> # Calculate volume-weighted average price
> vwapv <- TTR::VWAP(price=VTI_close, volume=VTI_vol, n=10)
> # Plot OHLC candlechart with volume
> chartSeries(ohlc, name="VTI plus VWAP", theme=chartTheme("white"))
> # Add VWAP to main plot
> addTA(ta=vwapv, on=1, col='red')
> # Add price minus VWAP in extra panel
> addTA(ta=(VTI_close-vwapv), col='red')
```

The function VWAP() from package *TTR* calculates the Volume Weighted Average Price as the average of past prices multiplied by their trading volumes, divided by the total volume.

The argument "n" represents the number of look-back intervals used for averaging,

Shading Plots Using addTA()

addTA() accepts Boolean vectors for shading of plots.

The function addLines() draws vertical or horizontal lines in plots.

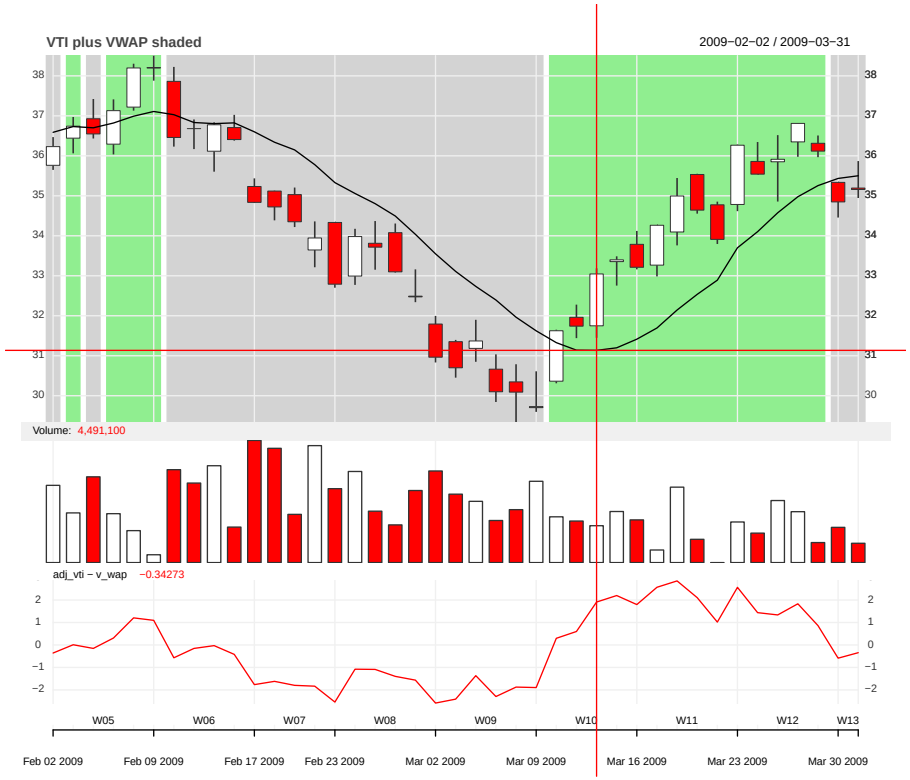
```
> # Plot OHLC candlechart with volume
> chartSeries(ohlc, name="VTI plus VWAP shaded",
+       theme=chartTheme("white"))
> # Add VWAP to main plot
> addTA(ta=vwapv, on=1, col='red')
> # Add price minus VWAP in extra panel
> addTA(ta=(VTI_close-vwapv), col='red')
> # Add background shading of areas
> addTA((VTI_close-vwapv) > 0, on=-1,
+ col="lightgreen", border="lightgreen")
> addTA((VTI_adj-vwapv) < 0, on=-1,
+ col="lightgrey", border="lightgrey")
> # Add vertical and horizontal lines at vwapv minimum
> addLines(v=which.min(vwapv), col='red')
> addLines(h=min(vwapv), col='red')
```

Plotting Time Series Using chart_Series()

The function chart_Series() from package *quantmod* is an improved version of chartSeries(), with better aesthetics.

chart_Series() plots are compatible with the base graphics package in R, so that standard plotting functions can be used in conjunction with chart_Series().

```
> # OHLC candlechart VWAP in main plot,
> chart_Series(x=ohlc, # Volume in extra panel
+         TA="add_Vo(); add_TA(vwapv, on=1)",
+         name="VTI plus VWAP shaded")
> # Add price minus VWAP in extra panel
> add_TA(VTI_adj-vwapv, col='red')
> # Add background shading of areas
> add_TA((VTI_adj-vwapv) > 0, on=-1,
+ col="lightgreen", border="lightgreen")
> add_TA((VTI_adj-vwapv) < 0, on=-1,
+ col="lightgrey", border="lightgrey")
> # Add vertical and horizontal lines
> abline(v=which.min(vwapv), col='red')
> abline(h=min(vwapv), col='red')
```



chart_Series() also has its own functions for adding indicators: add_TA(), add_BBands(), etc.

Note that functions associated with chart_Series() contain an underscore in their name,

Plot and Theme Objects of chart_Series()

The function chart_Series() creates a *plot object* and returns it *invisibly*.

A *plot object* is an environment of class *replot*, containing parameters specifying a plot.

A plot can be rendered by calling, plotting, or printing the *plot object*.

A plot *theme object* is a list containing parameters that determine the plot appearance (colors, size, fonts).

The function chart_theme() returns the *theme object*.

chart_Series() plots can be modified by modifying *plot objects* or *theme objects*.

Plot and theme objects can be modified directly, or by using accessor and setter functions.

The parameter "plot=FALSE" suppresses plotting and allows modifying *plot objects*.

```
> # Extract plot object
> chobj <- chart_Series(x=ohlc, plot=FALSE)
> class(chobj)
> ls(chobj)
> class(chobj$get_ylim)
> class(chobj$set_ylim)
> # ls(chobj$Env)
> class(chobj$Env$actions)
> plot_theme <- chart_theme()
> class(plot_theme)
> ls(plot_theme)
```

Customizing chart_Series() Plots

chart_Series() plots can be customized by modifying the plot and theme objects.

Plot and theme objects can be modified directly, or by using accessor and setter functions.

A plot is rendered by calling, plotting, or printing the plot object.

The parameter "plot=FALSE" suppresses plotting and allows modifying *plot objects*.

```
> ohlc <- rutils::etfenv$VTI["2010-04/2010-05"]
> # Extract, modify theme, format tick marks "%b %d"
> plot_theme <- chart_theme()
> plot_theme$format.labels <- "%b %d"
> # Create plot object
> chobj <- chart_Series(x=ohlc, theme=plot_theme, plot=FALSE)
> # Extract ylim using accessor function
> ylim <- chobj$get_ylim()
> ylim[[2]] <- structure(range(quantmod::Cl(ohlc)) + c(-1, 1),
+   fixed=TRUE)
> # Modify plot object to reduce y-axis range
> chobj$set_ylim(ylim)  # use setter function
> # Render the plot
> plot(chobj)
```

Plotting chart_Series() in Multiple Panels

chart_Series() plots are compatible with the base graphics package, allowing easy plotting in multiple panels.

The parameter "plot=FALSE" suppresses plotting and allows adding extra plot elements.

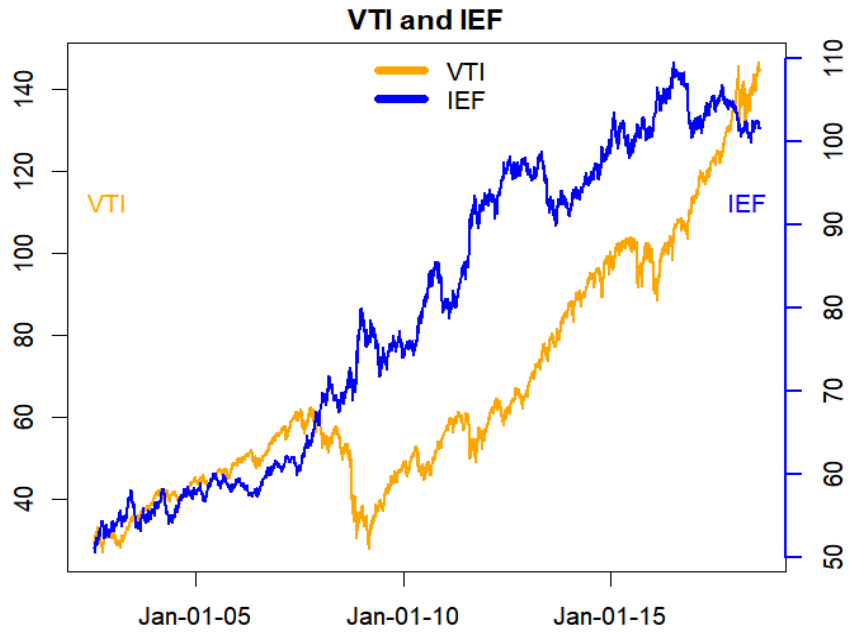
```
> # Calculate VTI and XLF volume-weighted average price
> vwapv <- TTR::VWAP(price=quantmod::Cl(rutils::etfenv$VTI),
+       volume=quantmod::Vo(rutils::etfenv$VTI), n=10)
> XLF_vwap <- TTR::VWAP(price=quantmod::Cl(rutils::etfenv$XLF),
+       volume=quantmod::Vo(rutils::etfenv$XLF), n=10)
> # Open graphics device, and define
> # Plot area with two horizontal panels
> x11(); par(mfrow=c(2, 1))
> chobj <- chart_Series(  # Plot in top panel
+   x=etfenv$VTI["2009-02/2009-04"],
+   name="VTI", plot=FALSE)
> add_TA(vwapv["2009-02/2009-04"], lwd=2, on=1, col='blue')
> # Plot in bottom panel
> chobj <- chart_Series(x=etfenv$XLF["2009-02/2009-04"],
+   name="XLF", plot=FALSE)
> add_TA(XLF_vwap["2009-02/2009-04"], lwd=2, on=1, col='blue')
```

zoo Plots With Two "y" Axes

The function `plot.zoo()` plots time series.

The function `axis()` plots customized axes in an existing plot.

```
> # Open plot window and set plot margins
> x11(width=6, height=4)
> par(mar=c(2, 2, 2, 2), oma=c(1, 1, 1, 1))
> # Plot first time series without x-axis
> zoo::plot.zoo(pricev[, 1], lwd=2, col="orange",
+         xlab=NA, ylab=NA, xaxt="n")
> # Create X-axis date labels and add X-axis
> dates <- pretty(zoo::index(pricev))
> axis(side=1, at=dates, labels=format(dates, "%b-%d-%y"))
> # Plot second time series without y-axis
> par(new=TRUE)  # Allow new line on same plot
> zoo::plot.zoo(pricev[, 2], xlab=NA, ylab=NA,
+         lwd=2, yaxt="n", col="blue", xaxt="n")
> # Plot second y-axis on right
> axis(side=4, lwd=2, col="blue")
> # Add axis labels
> mtext(colnamev[1], cex=1.2, lwd=3, side=2, las=2, adj=(-0.5), padj=(-5), col="orange")
> mtext(colnamev[2], cex=1.2, lwd=3, side=4, las=2, adj=1.5, padj=(-5), col="blue")
> # Add title and legend
> title(main=paste(colnamev, collapse=" and "), line=0.5)
> legend("top", legend=colnamev, cex=1.0, bg="white",
+  lty=1, lwd=6, col=c("orange", "blue"), bty="n")
```



Plotting *OHLC* Time Series Using Package *dygraphs*

The function `dygraph()` from package *dygraphs* creates interactive plots for *xts* time series.

The function `dyCandlestick()` creates a *candlestick* plot object for *OHLC* data, and uses the first four columns to plot *candlesticks*, and it plots any additional columns as lines.

The function `dyOptions()` adds options (like colors, etc.) to a *dygraph* plot.

```
> library(dygraphs)
> # Calculate volume-weighted average price
> ohlc <- rutils::etfenv$VTI
> vwapv <- TTR::VWAP(price=quantmod::Cl(ohlc),
+     volume=quantmod::Vo(ohlc), n=20)
> # Add VWAP to OHLC data
> datav <- cbind(ohlc[, 1:4], vwapv)["2009-01/2009-04"]
> # Create dygraphs object
> dyplot <- dygraphs::dygraph(datav)
> # Increase line width and color
> dyplot <- dygraphs::dyOptions(dyplot,
+   colors="red", strokeWidth=3)
> # Convert dygraphs object to candlestick plot
> dyplot <- dygraphs::dyCandlestick(dyplot)
> # Render candlestick plot
> dyplot
> # Candlestick plot using pipes syntax
> dygraphs::dygraph(datav) %>% dyCandlestick() %>%
+   dyOptions(colors="red", strokeWidth=3)
> # Candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dyOptions(dygraphs::dygraph(datav),
+   colors="red", strokeWidth=3))
```

Each *candlestick* displays one period of data, and consists of a box representing the *Open* and *Close* prices, and a vertical line representing the *High* and *Low* prices.

The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

dygraphs OHLC Plots With Background Shading

The function `dyShading()` adds shading to a *dygraphs* plot object.

The function `dyShading()` requires a vector of dates for shading.

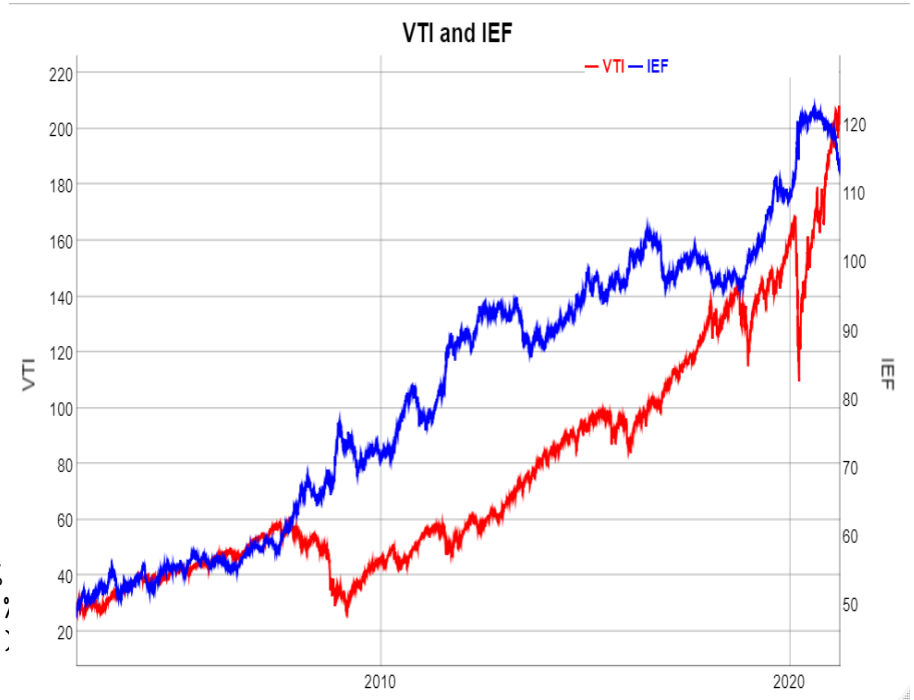
```
> # Create candlestick plot with background shading
> indic <- (Cl(datav) > datav[, "VWAP"])
> whichv <- which(rutils::diffit(indic) != 0)
> indic <- rbind(first(indic), indic[whichv, ], last(indic))
> dates <- zoo::index(indic)
> indic <- ifelse(drop(coredata(indic)), "lightgreen", "antiquewhite
> # Create dygraph object without rendering it
> dyplot <- dygraphs::dygraph(datav) %>% dyCandlestick() %>%
+   dyOptions(colors="red", strokeWidth=3)
> # Add shading
> for (i in 1:(NROW(indic)-1)) {
+     dyplot <- dyplot %>%
+ dyShading(from=dates[i], to=dates[i+1], color=indic[i])
+ }  # end for
> # Render the dygraph object
> dyplot
```

dygraphs Plots With Two "y" Axes

The function `dyAxis()` from package *dygraphs* plots customized axes to a *dygraphs* plot object.

The function `dySeries()` adds a time series to a *dygraphs* plot object.

```
> library(dygraphs)
> # Prepare VTI and IEF prices
> pricev <- cbind(Cl(rutils::etfenv$VTI), Cl(rutils::etfenv$IEF))
> pricev <- na.omit(pricev)
> colnamev <- rutils::get_name(colnames(pricev))
> colnames(pricev) <- colnamev
> # dygraphs plot with two y-axes
> library(dygraphs)
> dygraphs::dygraph(pricev, main=paste(colnamev, collapse=" and "))
+   dyAxis(name="y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis(name="y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", strokeWidth=2, col="red") %
+   dySeries(name=colnamev[2], axis="y2", strokeWidth=2, col="blue")
```



draft: Package *qmao* for Quantitative Financial Modeling

The package *qmao* is designed for downloading, manipulating, and visualizing *OHLC* time series data, package *quantmod*

qmao uses time series of class `"xts"`, and provides many useful functions for building quantitative financial models:

- `getSymbols()` for downloading data from external sources (*Yahoo*, *FRED*, etc.),
- `getFinancials()` for downloading financial statements,
- `adjustOHLC()` for adjusting *OHLC* data,
- `Op()`, `Cl()`, `Vo()`, etc. for extracting *OHLC* data columns,
- `periodReturn()`, `dailyReturn()`, etc. for calculating periodic returns,
- `chartSeries()` for candlestick plots of *OHLC* data,
- `addBBands()`, `addMA()`, `addVo()`, etc. for adding technical indicators (Moving Averages, Bollinger Bands) and volume data to a plot,

```
> # Load package qmao
> library(qmao)
> # Get documentation for package qmao
> # Get short description
> packageDescription("qmao")
> # Load help page
> help(package="qmao")
> # List all datasets in "qmao"
> data(package="qmao")
> # List all objects in "qmao"
> ls("package:qmao")
> # Remove qmao from search path
> detach("package:qmao")
```

Monte Carlo Simulation

Monte Carlo simulation consists of generating random samples from a given probability distribution.

The *Monte Carlo* data samples can then used to calculate different parameters of the probability distribution (moments, quantiles, etc.), and its functionals.

The *quantile* of a probability distribution is the value of the *random variable* `x`, such that the probability of values less than `x` is equal to the given *probability* p.

The *quantile* of a data sample can be calculated by first sorting the sample, and then finding the value corresponding closest to the given *probability* p.

The function `quantile()` calculates the sample quantiles. It uses interpolation to improve the accuracy. Information about the different interpolation methods can be found by typing `?quantile`.

The function `sort()` returns a vector sorted into ascending order.

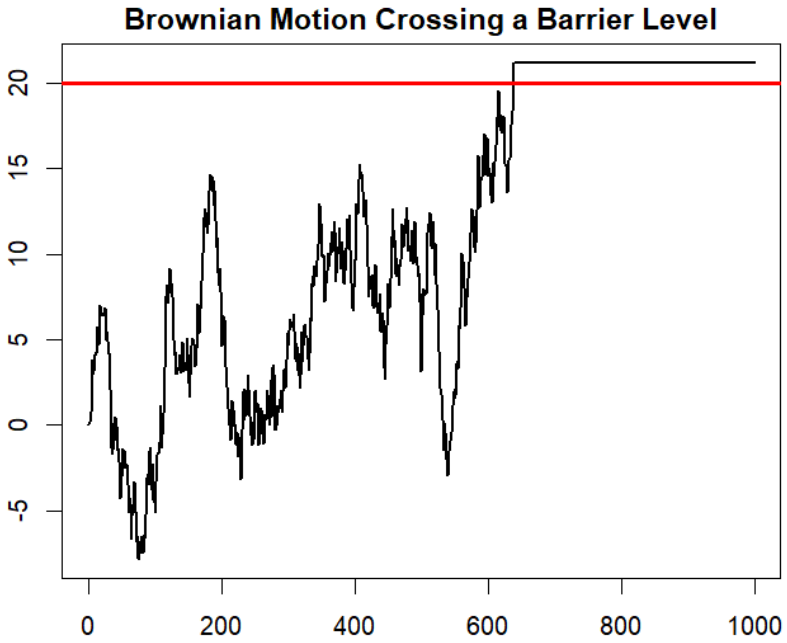
```
> set.seed(1121)  # Reset random number generator
> # Sample from Standard Normal Distribution
> nrows <- 1000
> datav <- rnorm(nrows)
> # Sample mean - MC estimate
> mean(datav)
> # Sample standard deviation - MC estimate
> sd(datav)
> # Monte Carlo estimate of cumulative probability
> pnorm(1)
> sum(datav < 1)/nrows
> # Monte Carlo estimate of quantile
> confl <- 0.98
> qnorm(confl)  # Exact value
> cutoff <- confl*nrows
> datav <- sort(datav)
> datav[cutoff]  # Naive Monte Carlo value
> quantile(datav, probs=confl)
> # Analyze the source code of quantile()
> stats:::quantile.default
> # Microbenchmark quantile
> library(microbenchmark)
> summary(microbenchmark(
+   monte_carlo = datav[cutoff],
+   quantilev = quantile(datav, probs=confl),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

Simulating Brownian Motion Using while() Loops

while() loops are often used in simulations, when the number of required loops is unknown in advance.

Below is an example of a simulation of the path of *Brownian Motion* crossing a barrier level.

```
> set.seed(1121)  # Reset random number generator
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> pathv <- numeric(nrows)  # Allocate path vector
> pathv[1] <- 0  # Initialize path
> it <- 2  # Initialize simulation index
> while ((it <= nrows) && (pathv[it - 1] < barl)) {
+ # Simulate next step
+   pathv[it] <- pathv[it - 1] + rnorm(1)
+   it <- it + 1  # Advance index
+ }  # end while
> # Fill remaining path after it crosses barl
> if (it <= nrows)
+   pathv[it:nrows] <- pathv[it - 1]
> # Plot the Brownian motion
> x11(width=6, height=4)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(pathv, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=barl, lwd=3, col="red")
> title(main="Brownian Motion Crossing a Barrier Level", line=0.5)
```

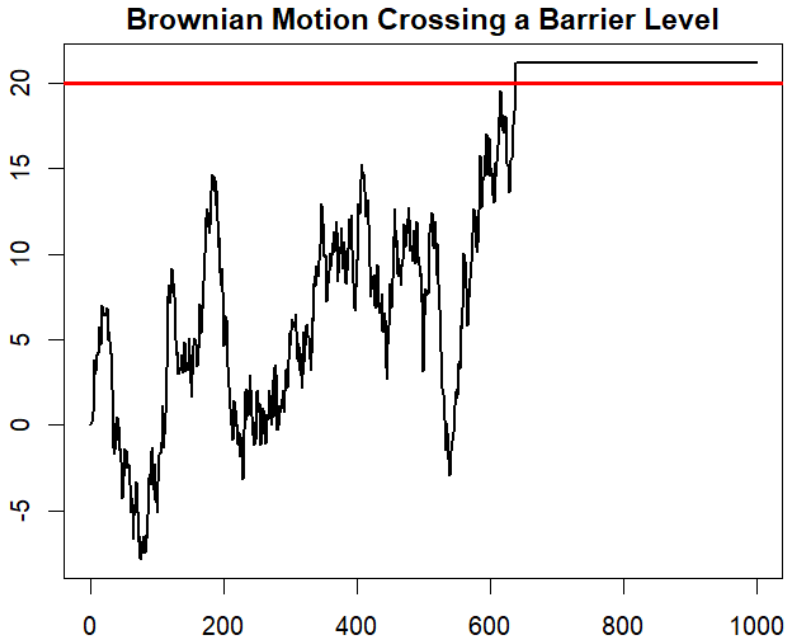


Simulating Brownian Motion Using Vectorized Functions

Simulations in R can be accelerated by pre-computing a vector of random numbers, instead of generatng them one at a time in a loop.

Vectors of random numbers allow using *vectorized* functions, instead of inefficient (slow) `while()` loops.

```
> set.seed(1121)  # Reset random number generator
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> # Simulate path of Brownian motion
> pathv <- cumsum(rnorm(nrows))
> # Find index when path crosses barl
> crossp <- which(pathv > barl)
> # Fill remaining path after it crosses barl
> if (NROW(crossp)>0) {
+   pathv[(crossp[1]+1):nrows] <- pathv[crossp[1]]
+ }  # end if
> # Plot the Brownian motion
> x11(width=6, height=4)
> par(mar=c(3, 3, 2, 1), oma=c(1, 1, 1, 1))
> plot(pathv, type="l", col="black",
+      lty="solid", lwd=2, xlab="", ylab="")
> abline(h=barl, lwd=3, col="red")
> title(main="Brownian Motion Crossing a Barrier Level", line=0.5)
```



The tradeoff between speed and memory usage: more memory may be used than necessary, since the simulation may stop before all the pre-computed random numbers are used up.

But the simulation is much faster because the path is simulated using *vectorized* functions,

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):

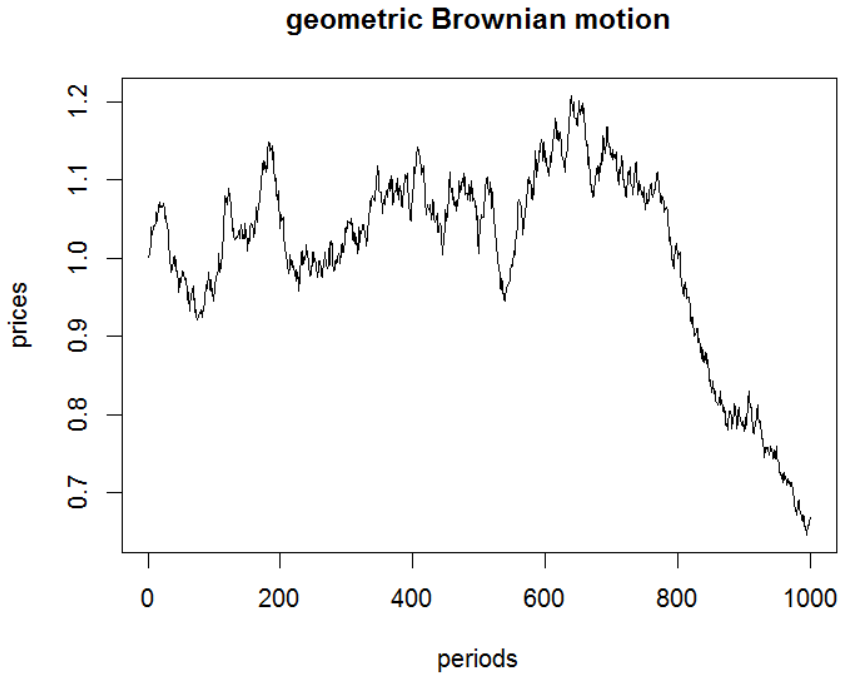
$$\mathrm{d}p_t = \mu p_t \mathrm{d}t + \sigma \, p_t \mathrm{d}W_t$$

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
> retsp <- sigmav*rnorm(nrows) + drift - sigmav^2/2
> pricev <- exp(cumsum(retsp))
> 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
> dates <- 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=dates)
> 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

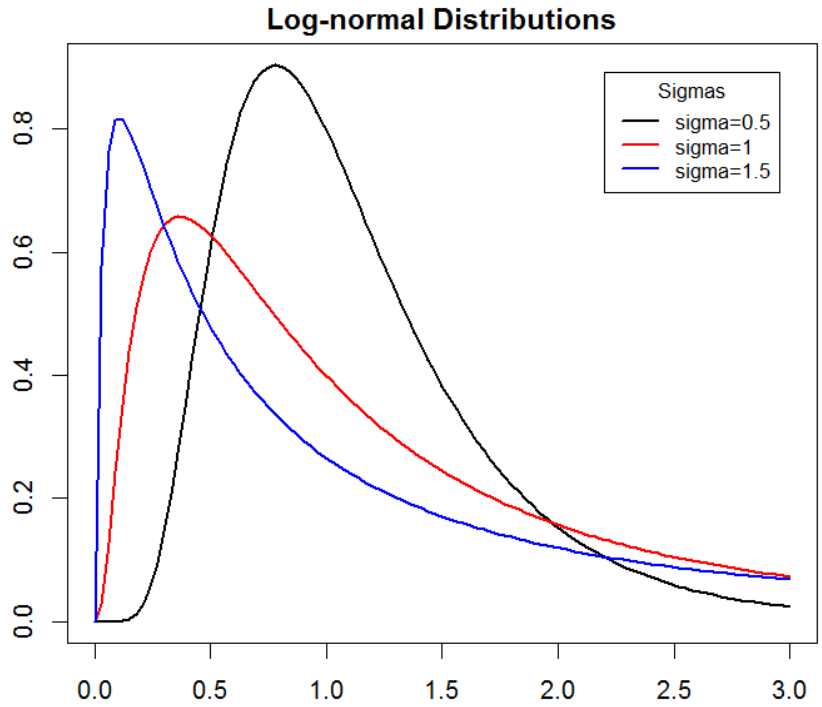
If x follows the *Normal* distribution $\phi(x, \mu, \sigma)$, then the exponential of x: $y = e^x$ follows the *Log-normal* distribution $\log \phi()$:

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

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.

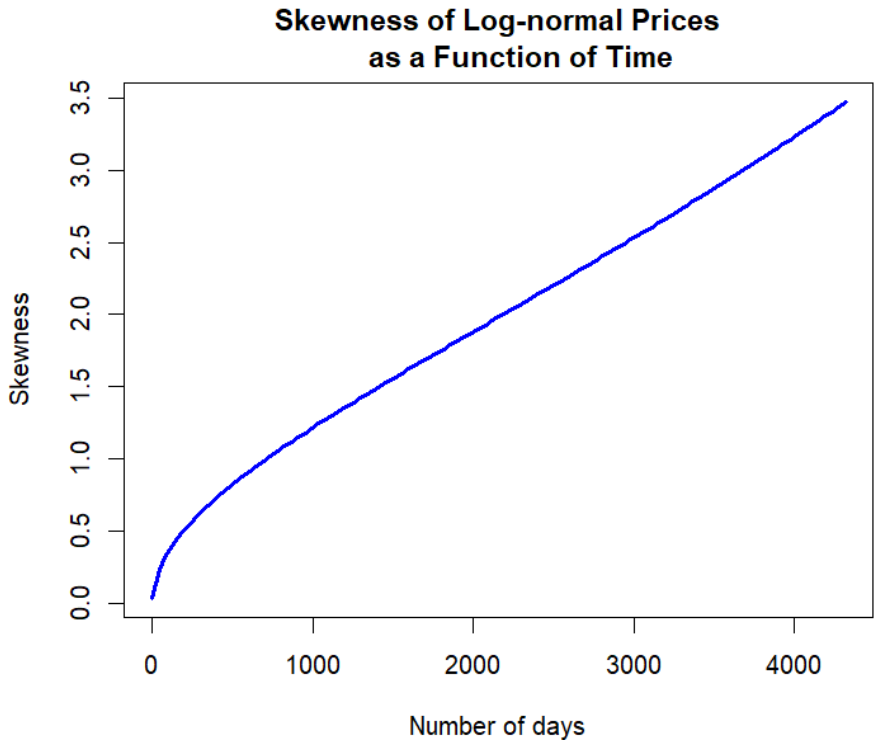
The standard deviation of *log-normal* prices is equal to the return volatility σ_r times the square root of time: $\sigma = \sigma_r \sqrt{t}$.

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: $\tilde{y} = \exp(\mu)$.

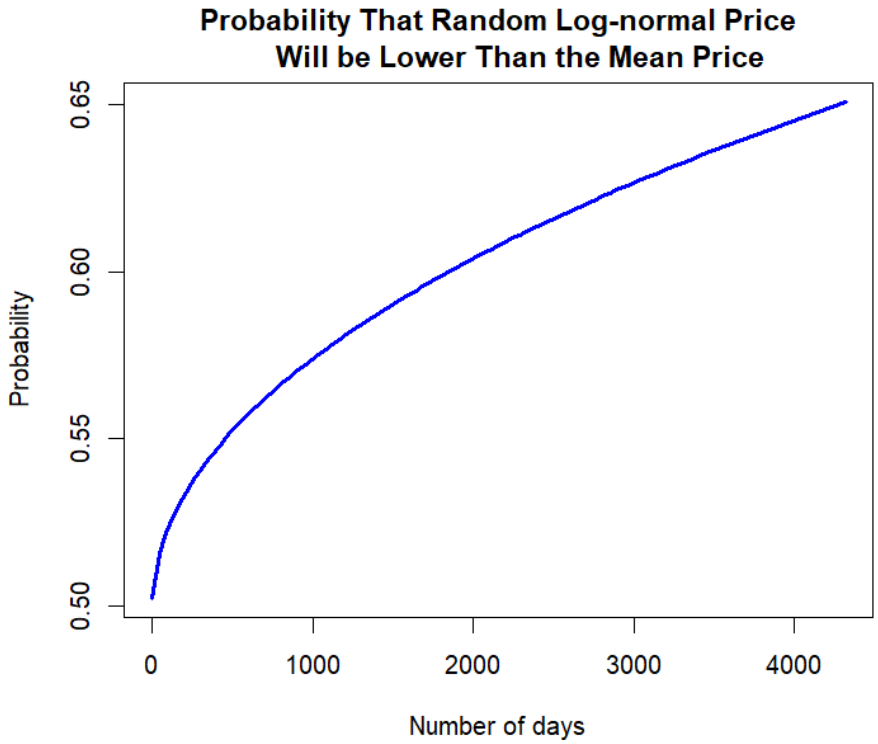
So if stock prices follow *Geometric Brownian motion* and are distributed *log-normally*, then a stock selected at random will have a high probability of havng a lower price than the mean expected price.

The cumulative *Log-normal* probability distribution is equal to $F(x) = \Phi(\frac{\log y - \mu}{\sigma})$, 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 $F(\bar{y}) = \Phi(\sigma/2)$.

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.



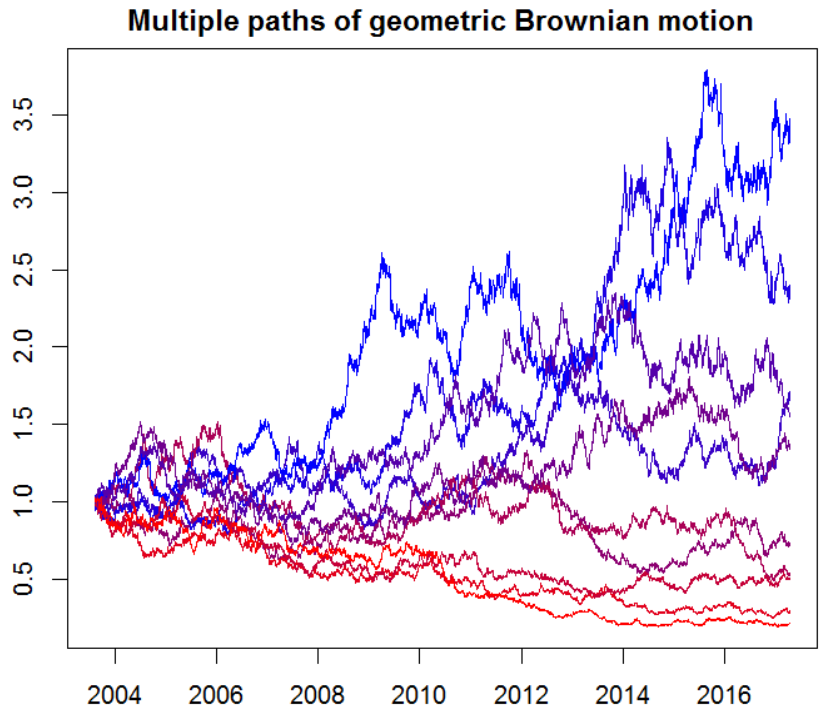

```
> # Probability that random log-normal price will be lower than the
> curve(expr=pnorm(sigmav*sqrt(x)/2),
+ xlim=c(1, nrows), lwd=3,
+ xlab="Number of days", ylab="Probability", col="blue",
+ main="Probability That Random Log-normal Price
+ Will be Lower Than the Mean Price")
```

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: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

```
> # 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)
```

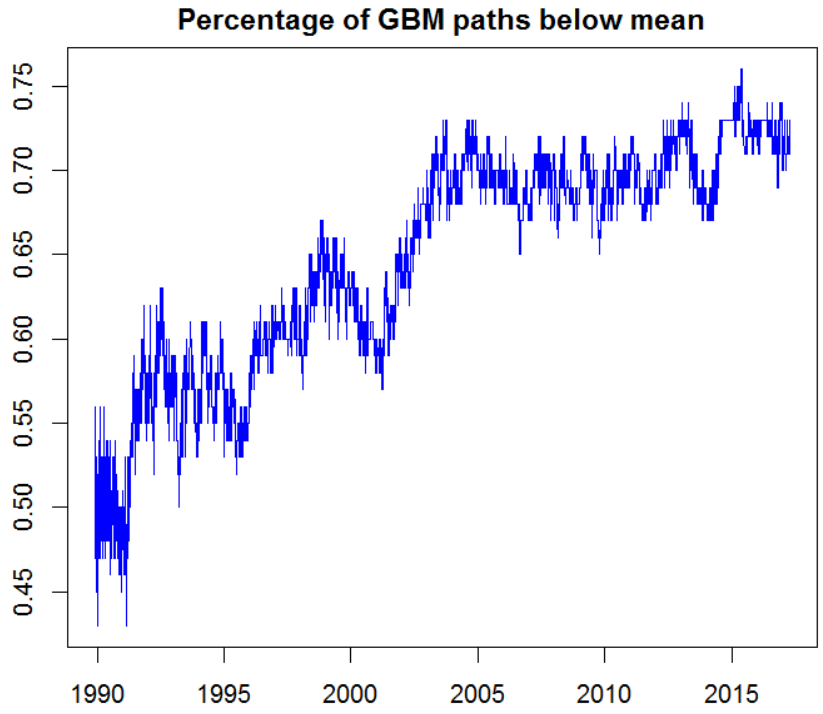


Distribution of Paths of Geometric Brownian Motion

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: $\varsigma \propto e^{1.5\sigma^2} = e^{1.5t\sigma_r^2}$

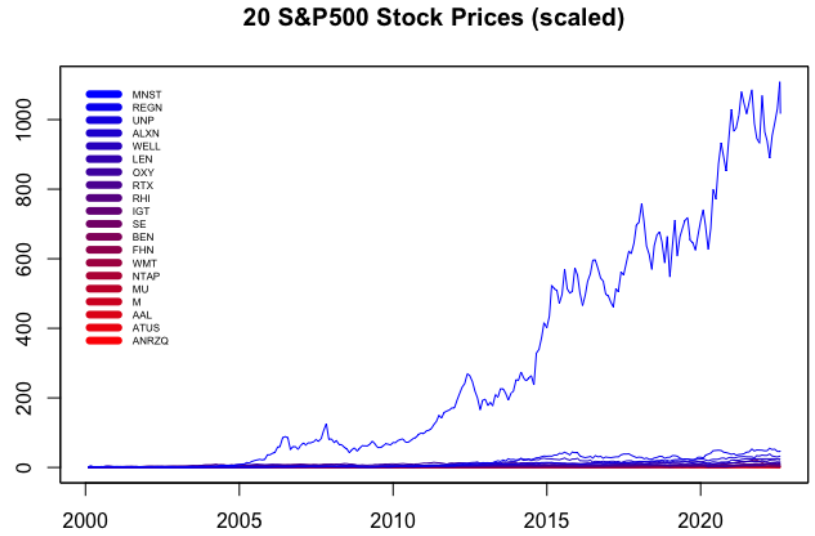
```
> # Define daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 10000
> npaths <- 100
> # Simulate multiple paths of geometric Brownian motion
> pricev <- rnorm(npaths*nrows, sd=sigmav) + drift - sigmav^2/2
> pricev <- matrix(pricev, nc=npaths)
> pricev <- exp(matrixStats::colCumsums(pricev))
> # 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[, 1:4])
> 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))]
```



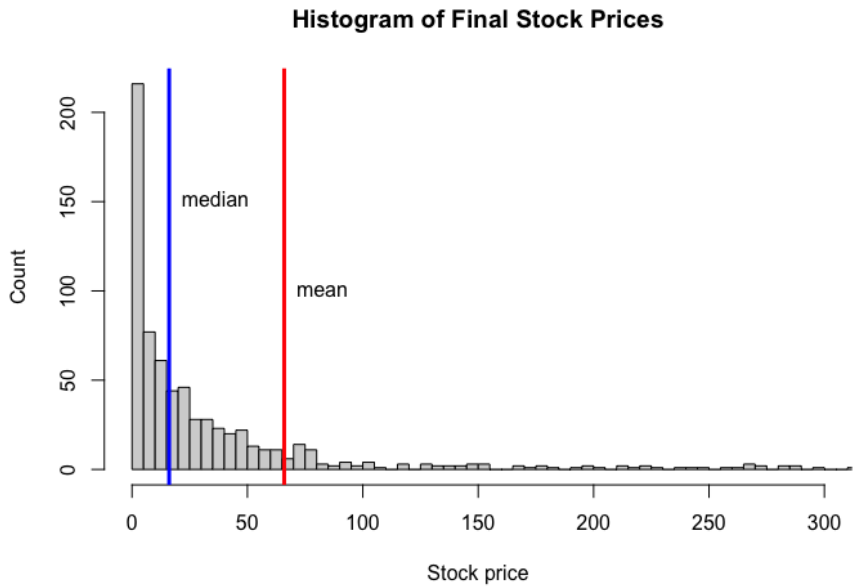

```
> # Plot xts time series of prices
> colorv <- colorRampPalette(c("red", "blue"))(NROW(symbolv))
> endp <- rutils::calc_endpoints(pricev, interval="months")
> plot.zoo(pricev[endp, symbolv], main="20 S&P500 Stock Prices (scal
+    xlab=NA, ylab=NA, plot.type="single", col=colorv)
> legend(x="topleft", 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)
```

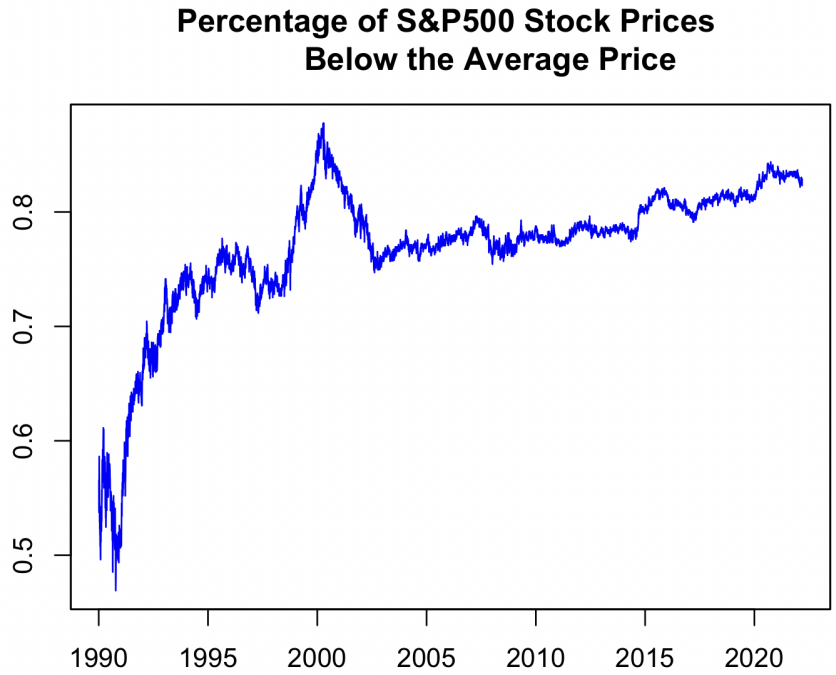
Distribution of Stock Prices Over Time

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 fr
+    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")
```

Autocorrelation Function of Time Series

The estimator of *autocorrelation* of a time series of returns r_i is equal to:

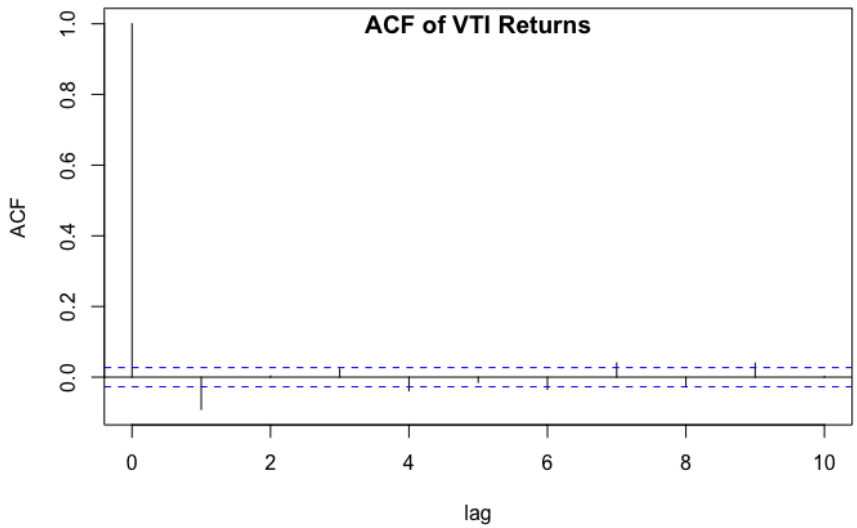
$$\rho_k = \frac{\sum_{i=k+1}^{n}(r_i - \bar{r})(r_{i-k} - \bar{r})}{(n-k)\,\sigma^2}$$

The *autocorrelation function* (*ACF*) is the vector of autocorrelation coefficients ρ_k.

The function `stats::acf()` calculates and plots the autocorrelation function of a time series.

The function `stats::acf()` has the drawback that it plots the lag zero autocorrelation (which is trivially equal to 1).

```
> # Open plot window under MS Windows
> x11(width=6, height=4)
> par(mar=c(3, 2, 1, 1), oma=c(1, 0, 0, 0))
> # Calculate VTI percentage returns
> retsp <- na.omit(rutils::etfenv$returns$VTI)
> # Plot autocorrelations of VTI returns using stats::acf()
> stats::acf(retsp, lag=10, xlab="lag", main="")
> title(main="ACF of VTI Returns", line=-1)
> # Calculate two-tailed 95% confidence interval
> qnorm(0.975)/sqrt(NROW(retsp))
```



The *VTI* time series of returns does not appear to have statistically significant autocorrelations.

The horizontal dashed lines are two-tailed confidence intervals of the autocorrelation estimator at 95% significance level: $\frac{\Phi^{-1}(0.975)}{\sqrt{n}}$.

But the visual inspection of the *ACF* plot alone is not enough to test whether autocorrelations are statistically significant or not.

Improved Autocorrelation Function

The function `acf()` has the drawback that it plots the lag zero autocorrelation (which is simply equal to 1).

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

The function `acf()` returns the *ACF* data invisibly, i.e. the return value can be assigned to a variable, but otherwise it isn't automatically printed to the console.

The function `rutils::plot_acf()` from package *rutils* is a wrapper for `acf()`, and it omits the lag zero autocorrelation.

```
> # Get the ACF data returned invisibly
> acfv <- acf(retsp, plot=FALSE)
> summary(acfv)
> # Print the ACF data
> print(acfv)
> dim(acfv$acf)
> dim(acfv$lag)
> head(acfv$acf)
```

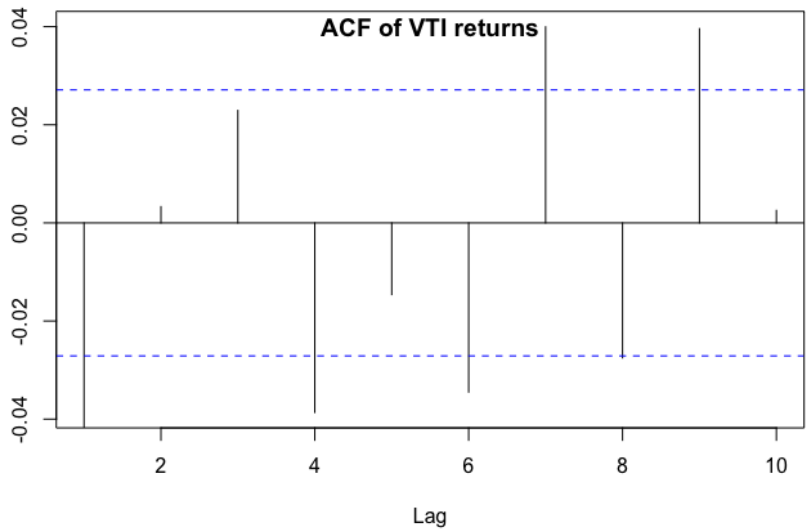
```
> plot_acf <- function(xtsv, lagg=10, plotobj=TRUE,
+                xlab="Lag", ylab="", main="", ...) {
+   # Calculate the ACF without a plot
+   acfv <- acf(x=xtsv, lag.max=lagg, plot=FALSE, ...)
+   # Remove first element of ACF data
+   acfv$acf <- array(data=acfv$acf[-1],
+     dim=c((dim(acfv$acf)[1]-1), 1, 1))
+   acfv$lag <- array(data=acfv$lag[-1],
+     dim=c((dim(acfv$lag)[1]-1), 1, 1))
+   # Plot ACF
+   if (plotobj) {
+     ci <- qnorm((1+0.95)/2)/sqrt(NROW(xtsv))
+     ylim <- c(min(-ci, range(acfv$acf[-1])),
+         max(ci, range(acfv$acf[-1])))
+     plot(acfv, xlab=xlab, ylab=ylab,
+    ylim=ylim, main="", ci=0)
+     title(main=main, line=0.5)
+     abline(h=c(-ci, ci), col="blue", lty=2)
+   }  # end if
+   # Return the ACF data invisibly
+   invisible(acfv)
+ }  # end plot_acf
```

Autocorrelations of *VTI* Returns

The *VTI* returns appear to have some small, yet significant negative autocorrelations at `lag=1`.

But the visual inspection of the *ACF* plot alone is not enough to test whether autocorrelations are statistically significant or not.

```
> # Improved autocorrelation function
> x11(width=6, height=4)
> rutils::plot_acf(retsp, lag=10, main="")
> title(main="ACF of VTI returns", line=-1)
> # Ljung-Box test for VTI returns
> Box.test(retsp, lag=10, type="Ljung")
```



Ljung-Box Test for Autocorrelations of Time Series

The *Ljung-Box* test, tests if the autocorrelations of a time series are *statistically significant*.

The *null hypothesis* of the *Ljung-Box* test is that the 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 have *statistically insignificant* autocorrelations.

The function `Box.test()` calculates the *Ljung-Box* test and returns the test statistic and its p-value.

```
> # Ljung-Box test for VTI returns
> # 'lag' is the number of autocorrelation coefficients
> Box.test(retsp, lag=10, type="Ljung")
> # Ljung-Box test for random returns
> Box.test(rnorm(NROW(retsp)), lag=10, type="Ljung")
> library(Ecdat)  # Load Ecdat
> macrodata <- as.zoo(Macrodat[, c("lhur", "fygm3")])
> colnames(macrodata) <- c("unemprate", "3mTbill")
> macrodiff <- na.omit(diff(macrodata))
> # Changes in 3 month T-bill rate are autocorrelated
> Box.test(macrodiff[, "3mTbill"], lag=10, type="Ljung")
> # Changes in unemployment rate are autocorrelated
> Box.test(macrodiff[, "unemprate"], lag=10, type="Ljung")
```

The *p*-value for *VTI* returns is small, and we conclude that the *null hypothesis* is `FALSE`, and that *VTI* returns do have some small autocorrelations.

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.

draft: Standard Errors of Autocorrelations

Under the *null hypothesis* of zero autocorrelation, the standard error of the autocorrelation estimator is equal to: $\frac{1}{\sqrt{n-2}}$, and slowly decreases as the square root of n - the length of the time series.

The function `cor()` calculates the correlation between two numeric vectors.

The function `cor.test()` performs a test of the statistical significance of the correlation coefficient.

The horizontal dashed lines are two-tailed confidence intervals of the autocorrelation estimator at 95% significance level: .

The *Ljung-Box* test, tests if the autocorrelations of a time series are *statistically significant*.

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 have *statistically insignificant* autocorrelations.

The function `Box.test()` calculates the *Ljung-Box* test

```
> # Calculate VTI and XLF percentage returns
> retsp <- rutils::etfenv$returns[, c("VTI", "XLF")]
> retsp <- na.omit(retsp)
> nrows <- NROW(retsp)
> # De-mean (center) and scale the returns
> retsp <- apply(retsp, MARGIN=2, function(x) (x-mean(x))/sd(x))
> apply(retsp, MARGIN=2, sd)
> # Calculate the correlation
> drop(retsp[, "VTI"] %*% retsp[, "XLF"])/(nrows-1)
> corv <- cor(retsp[, "VTI"], retsp[, "XLF"])
> # Test statistical significance of correlation
> cortest <- cor.test(retsp[, "VTI"], retsp[, "XLF"])
> confl <- qnorm((1+0.95)/2)/sqrt(nrows)
> corv*c(1-confl, 1+confl)
>
> # Get source code
> stats:::cor.test.default
>
> # Ljung-Box test for VTI returns
> # 'lag' is the number of autocorrelation coefficients
> Box.test(retsp, lag=10, type="Ljung")
> library(Ecdat)  # Load Ecdat
> macrodata <- as.zoo(Macrodat[, c("lhur", "fygm3")])
> colnames(macrodata) <- c("unemprate", "3mTbill")
> macrodiff <- na.omit(diff(macrodata))
> # Changes in 3 month T-bill rate are autocorrelated
> Box.test(macrodiff[, "3mTbill"], lag=10, type="Ljung")
> # Changes in unemployment rate are autocorrelated
> Box.test(macrodiff[, "unemprate"], lag=10, type="Ljung")
```

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

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

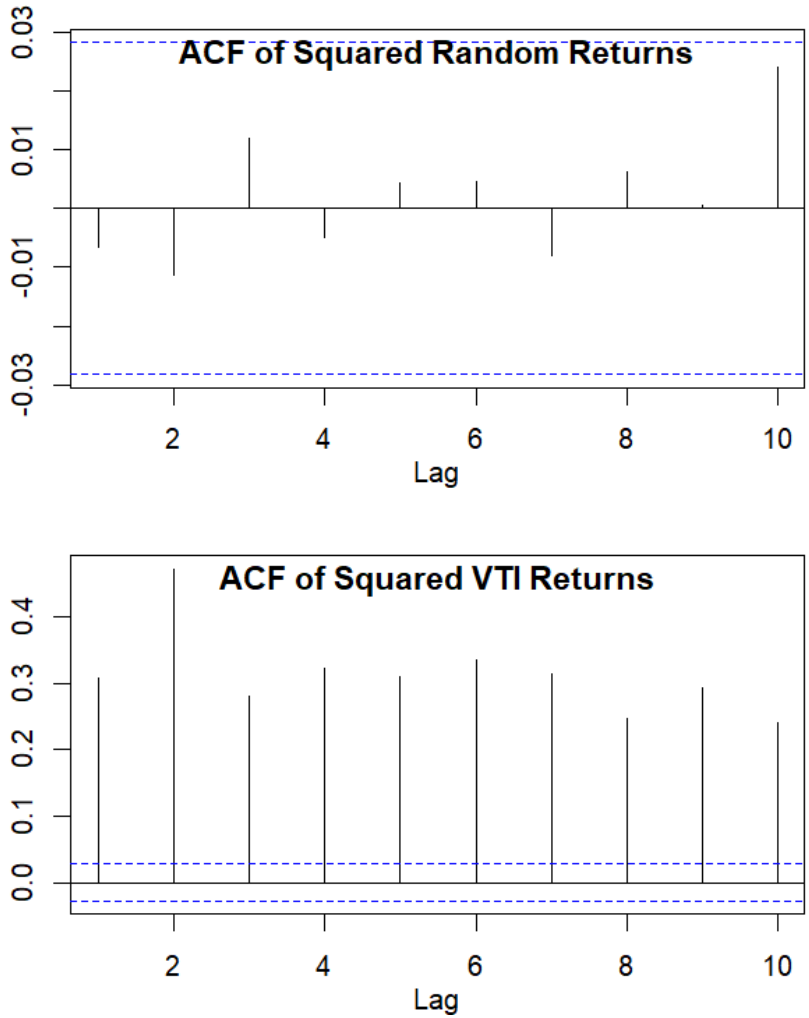
Autocorrelations of Squared *VTI* Returns

Squared random returns are not autocorrelated.

But squared *VTI* returns do have statistically significant autocorrelations.

The autocorrelations of squared asset returns are a very important feature.

```
> # Open plot window under MS Windows
> x11(width=6, height=7)
> # Set two vertical plot panels
> par(mfrow=c(2,1))
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> # Plot autocorrelations of squared random returns
> rutils::plot_acf(rnorm(NROW(retsp))^2, lag=10, main="")
> title(main="ACF of Squared Random Returns", line=-1)
> # Plot autocorrelations of squared VTI returns
> rutils::plot_acf(retsp^2, lag=10, main="")
> title(main="ACF of Squared VTI Returns", line=-1)
> # Ljung-Box test for squared VTI returns
> Box.test(retsp^2, lag=10, type="Ljung")
```



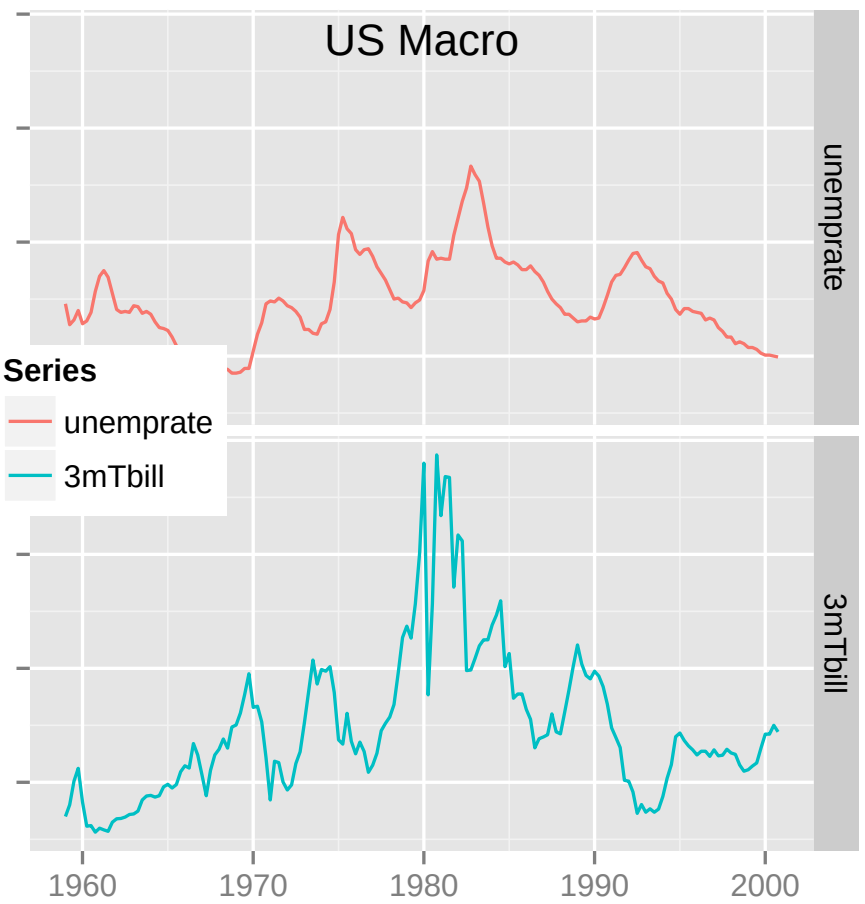
U.S. Macroeconomic Data

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 Macroeconomic Time Series
> # Coerce to "zoo"
> macrodata <- as.zoo(Macrodat[, c("lhur", "fygm3")])
> colnames(macrodata) <- c("unemprate", "3mTbill")
> # ggplot2 in multiple panes
> autoplot(  # Generic ggplot2 for "zoo"
+   object=macrodata, 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
```



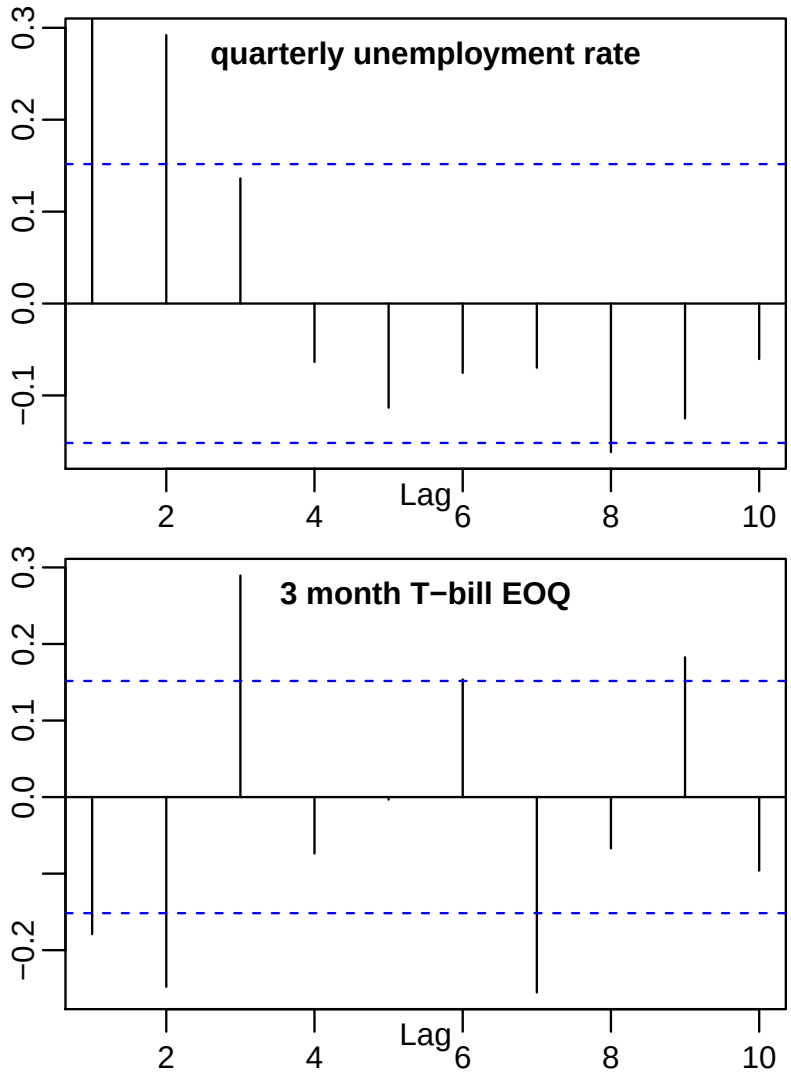
Autocorrelations of Econometric Data

Most econometric data displays a high degree of autocorrelation.

But the time series of asset returns display very low autocorrelations.

The function `zoo::coredata()` extracts the underlying numeric data from a complex data object.

```
> # Open plot window under MS Windows
> par(oma=c(15, 1, 1, 1), mgp=c(0, 0.5, 0), mar=c(1, 1, 1, 1),
+     cex.lab=0.8, cex.axis=0.8, cex.main=0.8, cex.sub=0.5)
> # Set two vertical plot panels
> par(mfrow=c(2,1))
> macrodiff <- na.omit(diff(macrodata))
> # Plot the autocorrelations
> rutils::plot_acf(coredata(macrodiff[, "unemprate"]),
+   lag=10, main="quarterly unemployment rate")
> rutils::plot_acf(coredata(macrodiff[, "3mTbill"]),
+   lag=10, main="3 month T-bill EOQ")
```

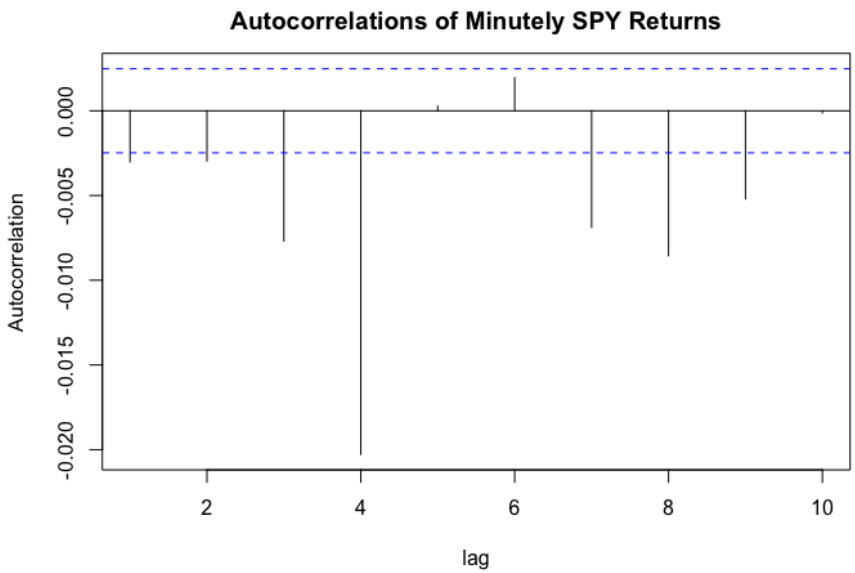


Autocorrelations of High Frequency Returns

The package *HighFreq* contains three time series of intraday 1-minute *OHLC* price bars, called `SPY`, `TLT`, and `VXX`, for the *SPY*, *TLT*, and *VXX* ETFs.

Minutely *SPY* returns have statistically significant negative autocorrelations.

```
> # Calculate SPY log prices and percentage returns
> ohlc <- HighFreq::SPY
> ohlc[, 1:4] <- log(ohlc[, 1:4])
> nrows <- NROW(ohlc)
> closep <- quantmod::Cl(ohlc)
> retsp <- rutils::diffit(closep)
> colnames(retsp) <- "SPY"
> # Open plot window under MS Windows
> x11(width=6, height=4)
> # Open plot window on Mac
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # Plot the autocorrelations of minutely SPY returns
> acfv <- rutils::plot_acf(as.numeric(retsp), lag=10,
+       xlab="lag", ylab="Autocorrelation", main="")
> title("Autocorrelations of Minutely SPY Returns", line=1)
> # Calculate the sum of autocorrelations
> sum(acfv$acf)
```



Autocorrelation as Function of Aggregation Interval

For *minutely SPY* returns, the *Ljung-Box* statistic is large and its *p*-value is very small, so we can conclude that *minutely SPY* returns have statistically significant autocorrelations.

The level of the autocorrelations depends on the sampling frequency, with higher frequency returns having more significant negative autocorrelations.

SPY returns aggregated to longer time intervals are less autocorrelated.

As the returns are aggregated to a lower periodicity, they become less autocorrelated, with daily returns having almost insignificant autocorrelations.

The function `rutils::to_period()` aggregates an *OHLC* time series to a lower periodicity.

```
> # Ljung-Box test for minutely SPY returns
> Box.test(retsp, lag=10, type="Ljung")
> # Calculate hourly SPY percentage returns
> closeh <- Cl(xts::to.period(x=ohlc, period="hours"))
> retsh <- rutils::diffit(closeh)
> # Ljung-Box test for hourly SPY returns
> Box.test(retsh, lag=10, type="Ljung")
> # Calculate daily SPY percentage returns
> closed <- Cl(xts::to.period(x=ohlc, period="days"))
> retsd <- rutils::diffit(closed)
> # Ljung-Box test for daily SPY returns
> Box.test(retsd, lag=10, type="Ljung")
> # Ljung-Box test statistics for aggregated SPY returns
> sapply(list(minutely=retsp, hourly=retsh, daily=retsd),
+   function(rets) {
+     Box.test(rets, lag=10, type="Ljung")$statistic
+ })  # end sapply
```

Volatility as a Function of the Aggregation Interval

The estimated volatility σ scales as the *power* of the length of the aggregation time interval Δt:

$$\frac{\sigma_t}{\sigma} = \Delta t^H$$

Where `H` is the *Hurst* exponent, σ is the return volatility, and σ_t is the volatility of the aggregated returns.

If returns follow *Brownian motion* then the volatility scales as the *square root* of the length of the aggregation interval (`H = 0.5`).

If returns are *mean reverting* then the volatility scales slower than the *square root* (`H < 0.5`).

If returns are *trending* then the volatility scales faster than the *square root* (`H > 0.5`).

The length of the daily time interval is often approximated to be equal to `390 = 6.5*60` minutes, since the exchange trading session is equal to `6.5` hours, and daily volatility is dominated by the trading session.

The daily volatility is exaggerated by price jumps over the weekends and holidays, so it should be scaled.

The minutely volatility is exaggerated by overnight price jumps.

```
> # Daily SPY volatility from daily returns
> sd(retsd)
> # Minutely SPY volatility scaled to daily interval
> sqrt(6.5*60)*sd(retsp)
> # Minutely SPY returns without overnight price jumps (unit per se
> retsp <- retsp/rutils::diffit(xts::.index(retsp))
> retsp[1] <- 0
> # Daily SPY volatility from minutely returns
> sqrt(6.5*60)*60*sd(retsp)
> # Daily SPY returns without weekend and holiday price jumps (unit
> retsd <- retsd/rutils::diffit(xts::.index(retsd))
> retsd[1] <- 0
> # Daily SPY volatility without weekend and holiday price jumps
> 24*60*60*sd(retsd)
```

The package *HighFreq* contains three time series of intraday 1-minute *OHLC* price bars, called `SPY`, `TLT`, and `VXX`, for the *SPY*, *TLT*, and *VXX* ETFs.

The function `rutils::to_period()` aggregates an *OHLC* time series to a lower periodicity.

The function `zoo::index()` extracts the time index of a time series.

The function `xts::.index()` extracts the time index expressed in the number of seconds.

Hurst Exponent From Volatility

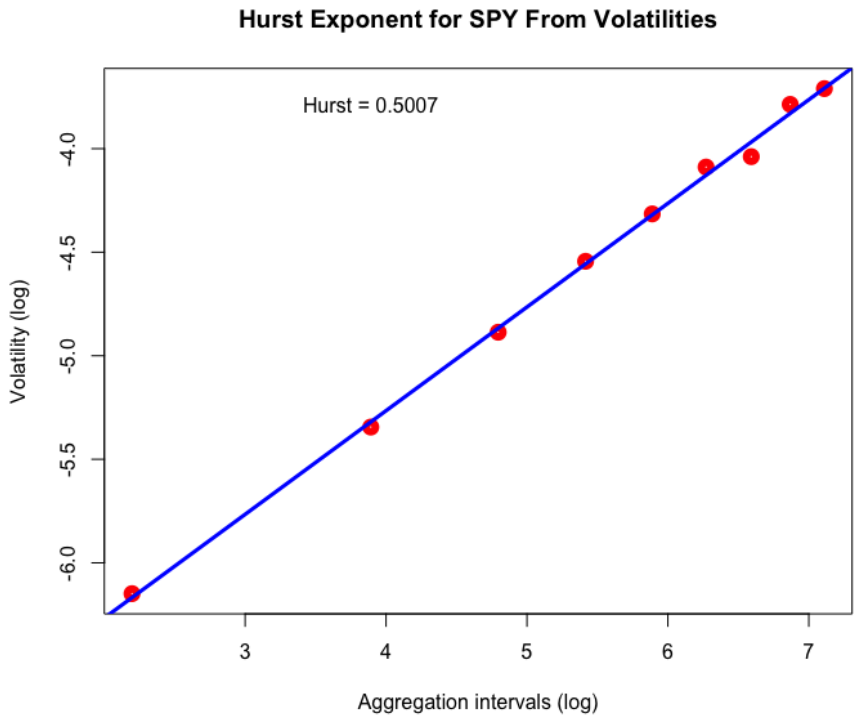
For a single aggregation interval, the *Hurst exponent* `H` is equal to:

$$H = \frac{\log \sigma_t - \log \sigma}{\log \Delta t}$$

For a vector of aggregation intervals Δt_i, the *Hurst exponent* `H` is equal to the regression slope between the logarithms of the aggregated volatilities σ_i versus the logarithms of the aggregation intervals Δt_i:

$$H = \frac{\text{cov}(\log \sigma_i, \log \Delta t_i)}{\text{var}(\log \Delta t_i)}$$

```
> # Calculate volatilities for vector of aggregation intervals
> aggv <- seq.int(from=3, to=35, length.out=9)^2
> volv <- sapply(aggv, function(aggint) {
+   naggs <- nrows %/% aggint
+   endp <- c(0, nrows - naggs*aggint + (0:naggs)*aggint)
+   # endp <- rutils::calc_endpoints(closep, interval=aggint)
+   sd(rutils::diffit(closep[endp]))
+ })  # end sapply
> # Calculate the Hurst from single data point
> volog <- log(volv)
> agglog <- log(aggv)
> (last(volog) - first(volog))/(last(agglog) - first(agglog))
> # Calculate the Hurst from regression slope using formula
> hurstexp <- cov(volog, agglog)/var(agglog)
> # Or using function lm()
> model <- lm(volog ~ agglog)
> coef(model)[2]
```




```
> # Plot the volatilities
> x11(width=6, height=4)
> par(mar=c(4, 4, 2, 1), oma=c(1, 1, 1, 1))
> plot(volog ~ agglog, lwd=6, col="red",
+      xlab="Aggregation intervals (log)", ylab="Volatility (log)",
+      main="Hurst Exponent for SPY From Volatilities")
> abline(model, lwd=3, col="blue")
> text(agglog[2], volog[NROW(volog)-1],
+      paste0("Hurst = ", round(hurstexp, 4)))
```

Rescaled Range Analysis

The range $R_{\Delta t}$ of prices p_t over an interval Δt, is the difference between the highest attained price minus the lowest:

$$R_t = \max_{\Delta t}[p_\tau] - \min_{\Delta t}[p_\tau]$$

The *Rescaled Range* $RS_{\Delta t}$ is equal to the range $R_{\Delta t}$ divided by the standard deviation of the price differences σ_t: $RS_{\Delta t} = R_t/\sigma_t$.

The *Rescaled Range* $RS_{\Delta t}$ for a time series of prices is calculated by:

- Dividing the time series into non-overlapping intervals of length Δt,
- Calculating the *rescaled range* $RS_{\Delta t}$ for each interval,
- Calculating the average of the *rescaled ranges* $RS_{\Delta t}$ for all the intervals.

Rescaled Range Analysis (R/S) consists of calculating the average *rescaled range* $RS_{\Delta t}$ as a function of the length of the aggregation interval Δt.

```
> # Calculate cumulative SPY returns
> closep <- cumsum(retsp)
> nrows <- NROW(closep)
> # Calculate the rescaled range
> aggint <- 500
> naggs <- nrows %/% aggint
> endp <- c(0, nrows - naggs*aggint + (0:naggs)*aggint)
> # Or
> # endp <- rutils::calc_endpoints(closep, interval=aggint)
> rrange <- sapply(2:NROW(endp), function(np) {
+   indeks <- (endp[np-1]+1):endp[np]
+   diff(range(closep[indeks]))/sd(retsp[indeks])
+ })  # end sapply
> mean(rrange)
> # Calculate the Hurst from single data point
> log(mean(rrange))/log(aggint)
```

Hurst Exponent From Rescaled Range

The average *Rescaled Range* $RS_{\Delta t}$ is proportional to the length of the aggregation interval Δt raised to the power of the *Hurst exponent* `H`:

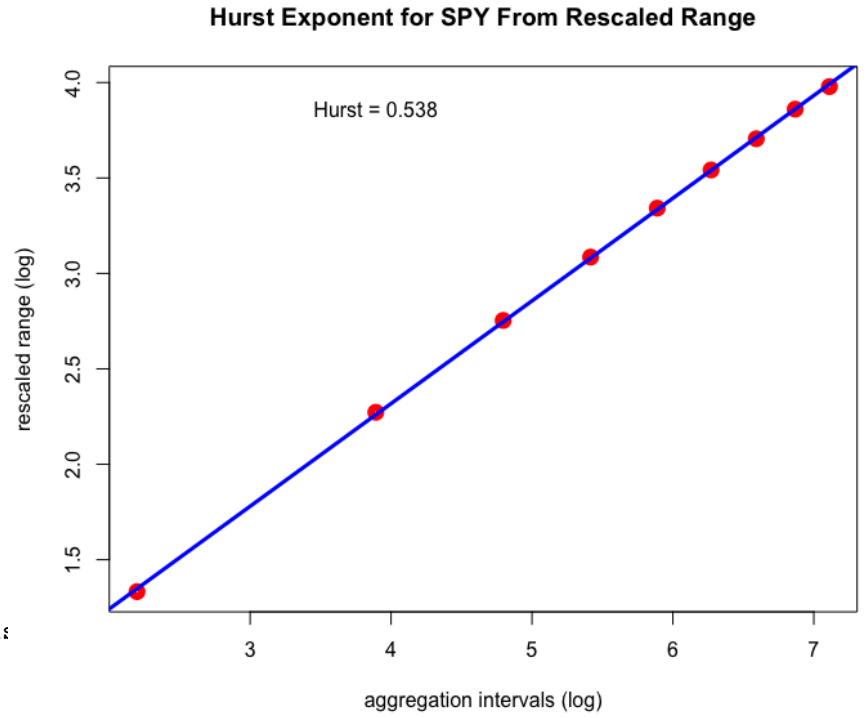
$$RS_{\Delta t} \propto \Delta t^{H}$$

So the *Hurst exponent* `H` is equal to:

$$H = \frac{\log RS_{\Delta t}}{\log \Delta t}$$

The Hurst exponents calculated from the *rescaled range* and the *volatility* are similar but not exactly equal because they use different methods to estimate price dispersion.

```
> # Calculate the rescaled range for vector of aggregation intervals
> rrange <- sapply(aggv, function(aggint) {
+ # Calculate the end points
+   naggs <- nrows %/% aggint
+   endp <- c(0, nrows - naggs*aggint + (0:naggs)*aggint)
+ # Calculate the rescaled ranges
+   rrange <- sapply(2:NROW(endp), function(np) {
+     indeks <- (endp[np-1]+1):endp[np]
+     diff(range(closep[indeks]))/sd(retsp[indeks])
+   })  # end sapply
+   mean(na.omit(rrange))
+ })  # end sapply
> # Calculate the Hurst as regression slope using formula
> rangelog <- log(rrange)
> agglog <- log(aggv)
> hurstexp <- cov(rangelog, agglog)/var(agglog)
> # Or using function lm()
> model <- lm(rangelog ~ agglog)
> coef(model)[2]
```



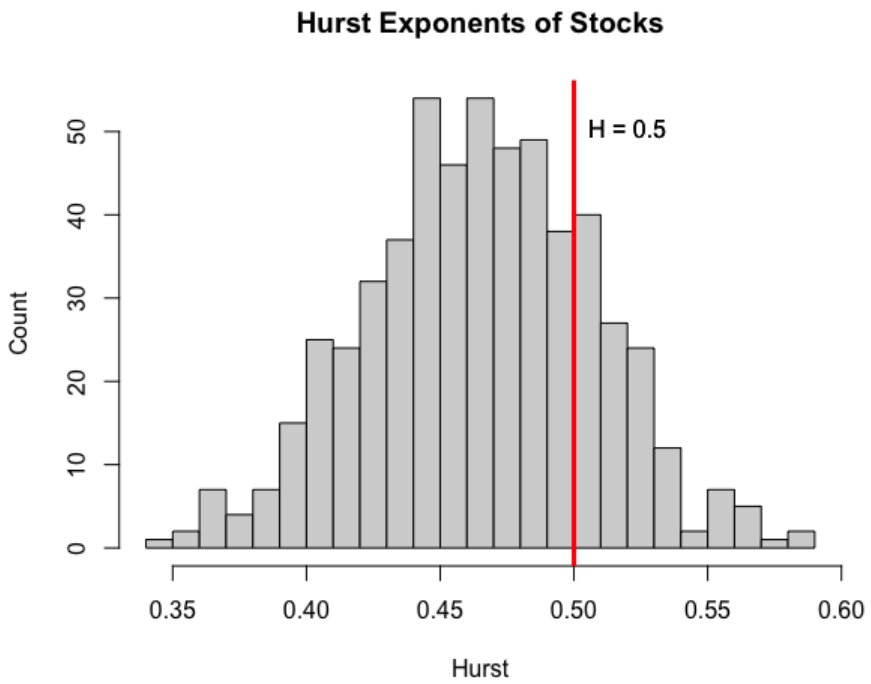

```
> plot(rangelog ~ agglog, lwd=6, col="red",
+      xlab="aggregation intervals (log)",
+      ylab="rescaled range (log)",
+      main="Hurst Exponent for SPY From Rescaled Range")
> abline(model, lwd=3, col="blue")
> text(agglog[2], rangelog[NROW(rangelog)-1],
+      paste0("Hurst = ", round(hurstexp, 4)))
```

Hurst Exponents of Stocks

The Hurst exponents of stocks are typically slightly less than `0.5`, because their idiosyncratic risk components are mean-reverting.

The function `HighFreq::calc_hurst()` calculates the Hurst exponent in `C++` using volatility ratios.

```
> # Load S&P500 constituent OHLC stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> class(sp500env$AAPL)
> head(sp500env$AAPL)
> # Calculate log stock prices after the year 2000
> pricev <- eapply(sp500env, function(ohlc) {
+   closep <- log(quantmod::Cl(ohlc)["2000/"])
+ # Ignore short lived and penny stocks (less than $1)
+   if ((NROW(closep) > 4000) & (last(closep) > 0))
+     return(closep)
+ })  # end eapply
> # Calculate the number of NULL prices
> sum(sapply(pricev, is.null))
> # Calculate the names of the stocks (remove NULL pricev)
> namev <- sapply(pricev, is.null)
> namev <- namev[!namev]
> namev <- names(namev)
> pricev <- pricev[namev]
> # Calculate the Hurst exponents of stocks
> aggv <- trunc(seq.int(from=3, to=10, length.out=5)^2)
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> # Dygraph of stock with largest Hurst exponent
> namev <- names(which.max(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Dygraph of stock with smallest Hurst exponent
> namev <- names(which.min(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
```



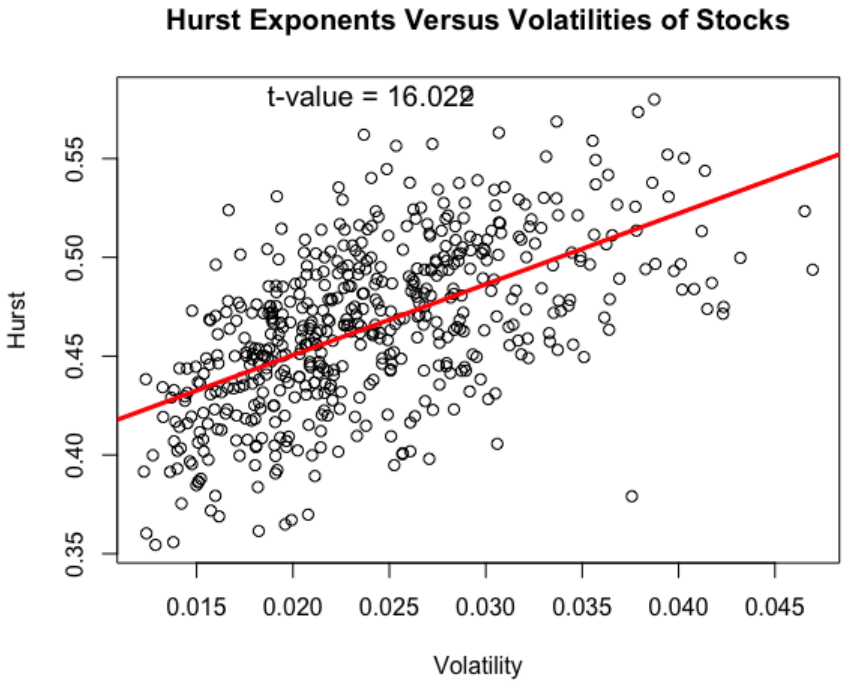

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=20, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of Stocks")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

Stock Volatility and Hurst Exponents

There is a strong relationship between stock volatility and hurst exponents.

Highly volatile stocks tend to have large Hurst exponents.

```
> # Calculate the volatility of stocks
> volat <- sapply(pricev, function(closep) {
+     sqrt(HighFreq::calc_var(HighFreq::diffit(closep)))
+ })  # end sapply
> # Dygraph of stock with highest volatility
> namev <- names(which.max(volat))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Dygraph of stock with lowest volatility
> namev <- names(which.min(volat))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Calculate the regression of the Hurst exponents versus volatilit
> model <- lm(hurstv ~ volat)
> summary(model)
```



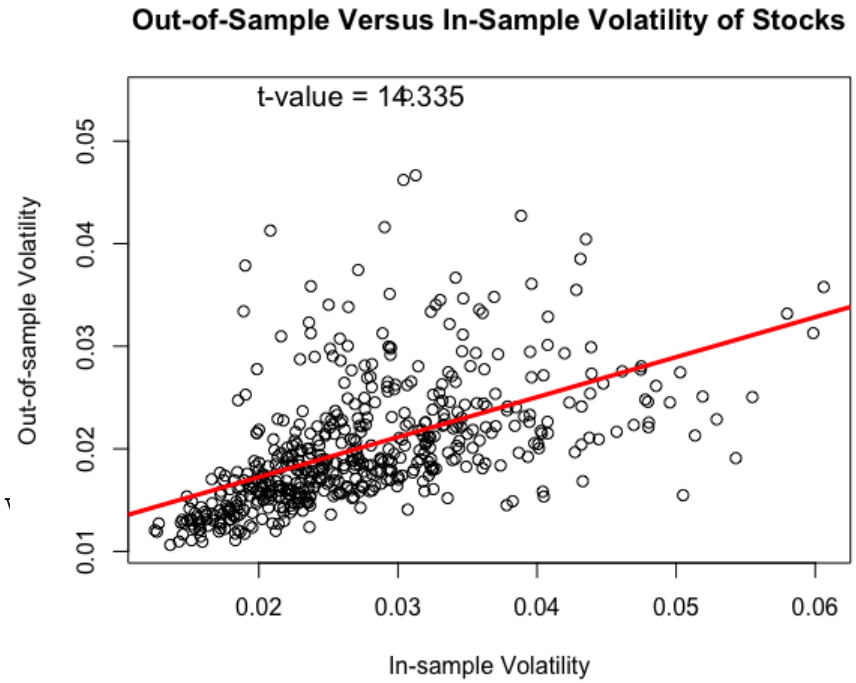

```
> # Plot scatterplot of the Hurst exponents versus volatilities
> plot(hurstv ~ volat, xlab="Volatility", ylab="Hurst",
+      main="Hurst Exponents Versus Volatilities of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(volat), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Out-of-Sample Volatility of Stocks

There is a strong relationship between *out-of-sample* and *in-sample* stock volatility.

Highly volatile stocks *in-sample* also tend to have high volatility *out-of-sample*.

```
> # Calculate the in-sample volatility of stocks
> volatis <- sapply(pricev, function(closep) {
+     sqrt(HighFreq::calc_var(HighFreq::diffit(closep["/2010"])))
+ })  # end sapply
> # Calculate the out-of-sample volatility of stocks
> volatos <- sapply(pricev, function(closep) {
+     sqrt(HighFreq::calc_var(HighFreq::diffit(closep["2010/"])))
+ })  # end sapply
> # Calculate the regression of the out-of-sample versus in-sample v
> model <- lm(volatos ~ volatis)
> summary(model)
```



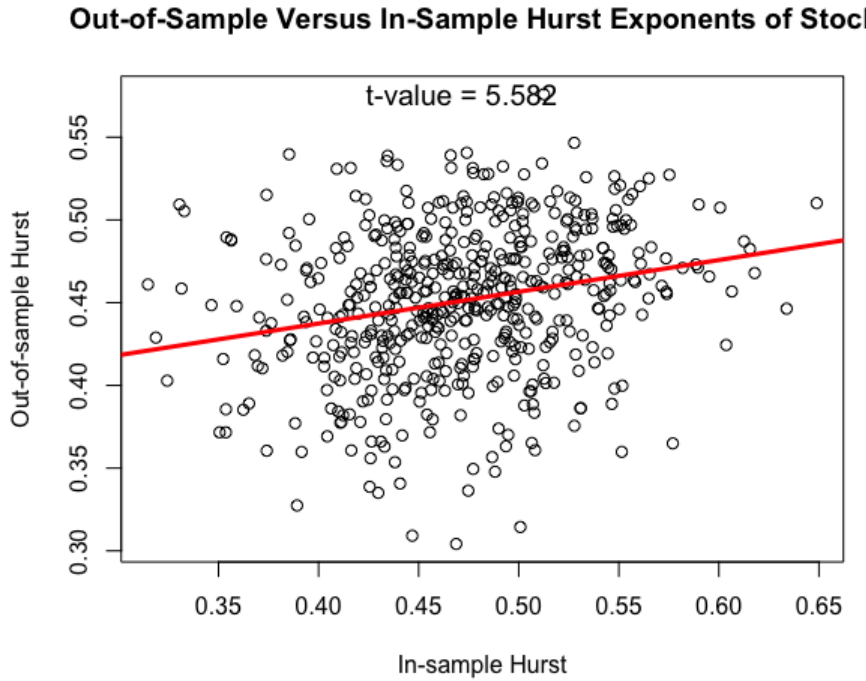

```
> # Plot scatterplot of the out-of-sample versus in-sample volatilit
> plot(volatos ~ volatis, xlab="In-sample Volatility", ylab="Out-of-
+      main="Out-of-Sample Versus In-Sample Volatility of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(volatis), y=max(volatos),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Out-of-Sample Hurst Exponents of Stocks

The *out-of-sample* Hurst exponents of stocks have a significant positive correlation to the *in-sample* Hurst exponents.

That means that stocks with larger *in-sample* Hurst exponents tend to also have larger *out-of-sample* Hurst exponents (but not always).

This is because stock volatility persists *out-of-sample*, and Hurst exponents are larger for higher volatility stocks.




```
> # Calculate the in-sample Hurst exponents of stocks
> hurstis <- sapply(pricev, function(closep) {
+   HighFreq::calc_hurst(closep["/2010"], aggv=aggv)
+ })  # end sapply
> # Calculate the out-of-sample Hurst exponents of stocks
> hurstos <- sapply(pricev, function(closep) {
+   HighFreq::calc_hurst(closep["2010/"], aggv=aggv)
+ })  # end sapply
> # Calculate the regression of the out-of-sample versus in-sample Hurst exponents
> model <- lm(hurstos ~ hurstis)
> summary(model)
```

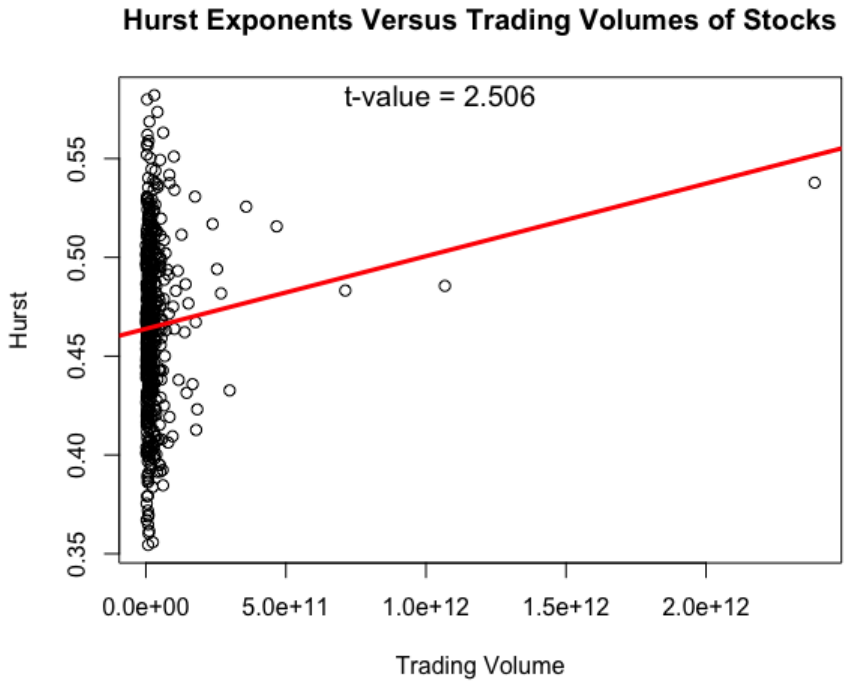
```
> # Plot scatterplot of the out-of-sample versus in-sample Hurst exp
> plot(hurstos ~ hurstis, xlab="In-sample Hurst", ylab="Out-of-sampl
+      main="Out-of-Sample Versus In-Sample Hurst Exponents of Stock
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(hurstis), y=max(hurstos),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Stock Trading Volumes and Hurst Exponents

The relationship between stock trading volumes and Hurst exponents is not very significant.

The relationship is dominated by a few stocks with very large trading volumes, like *AAPL*, which also tend to be more volatile and therefore have larger Hurst exponents.

```
> # Calculate the stock trading volumes after the year 2000
> volumev <- eapply(sp500env, function(ohlc) {
+     sum(quantmod::Vo(ohlc)["2000/"])
+ })  # end eapply
> # Remove NULL values
> volumev <- volumev[names(pricev)]
> volumev <- unlist(volumev)
> which.max(volumev)
> # Calculate the number of NULL prices
> sum(is.null(volumev))
> # Calculate the Hurst exponents of stocks
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> # Calculate the regression of the Hurst exponents versus trading volumes
> model <- lm(hurstv ~ volumev)
> summary(model)
```



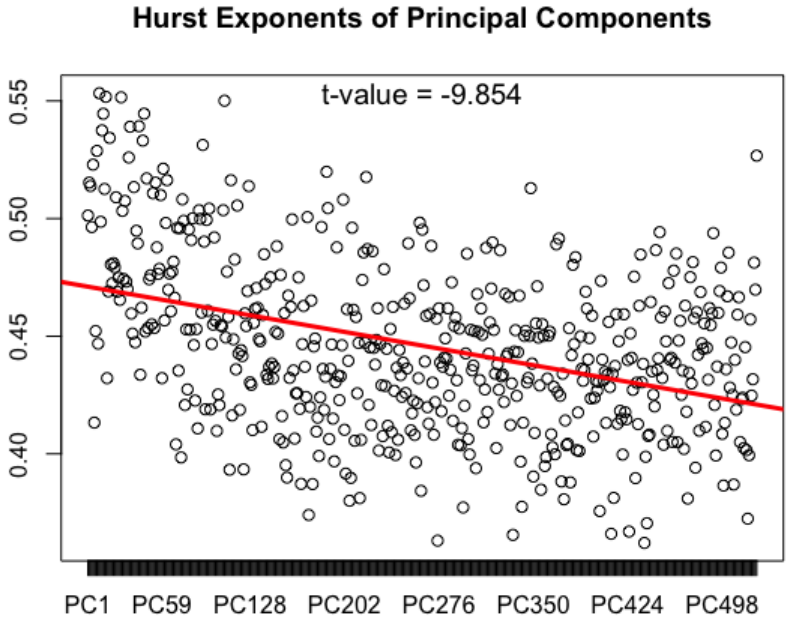

```
> # Plot scatterplot of the Hurst exponents versus trading volumes
> plot(hurstv ~ volumev, xlab="Trading Volume", ylab="Hurst",
+      main="Hurst Exponents Versus Trading Volumes of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=quantile(volumev, 0.998), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Hurst Exponents of Stock Principal Components

The Hurst exponents of the lower order principal components are typically larger than of the higher order principal components.

This is because the lower order principal components represent systematic risk factors, while the higher order principal components represent idiosyncratic risk factors, which are mean-reverting.

The Hurst exponents of most higher order principal components are less than 0.5, so they can potentially be traded in mean-reverting strategies.




```
> # Calculate log stock returns
> retsp <- lapply(pricev, rutils::diffit)
> retsp <- rutils::do_call(cbind, retsp)
> retsp[is.na(retsp)] <- 0
> sum(is.na(retsp))
> # Drop ".Close" from column names
> colnames(retsp[, 1:4])
> colnames(retsp) <- rutils::get_name(colnames(retsp))
> # Calculate PCA prices using matrix algebra
> eigend <- eigen(cor(retsp))
> retspca <- retsp %*% eigend$vectors
> pricepca <- xts::xts(matrixStats::colCumsums(retspca),
+                      order.by=index(retsp))
> colnames(pricepca) <- paste0("PC", 1:NCOL(retsp))
> # Calculate the Hurst exponents of PCAs
> hurstv <- sapply(pricepca, HighFreq::calc_hurst, aggv=aggv)
> # Dygraph of PCA with largest Hurst exponent
> namev <- names(which.max(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
> # Dygraph of PCA with smallest Hurst exponent
> namev <- names(which.min(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
```

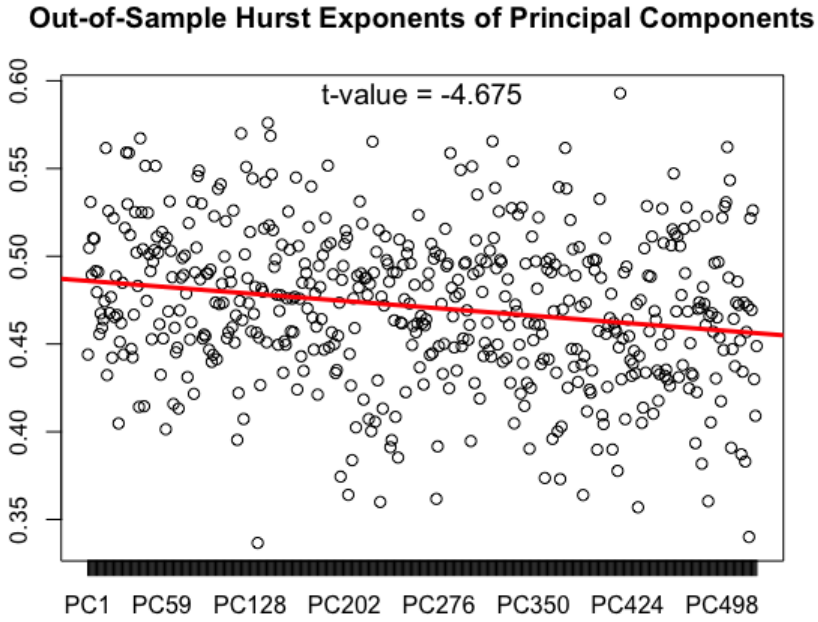
```
> # Plot the Hurst exponents of principal components without x-axis
> plot(hurstv, xlab=NA, ylab=NA, xaxt="n",
+      main="Hurst Exponents of Principal Components")
> # Add X-axis with PCA labels
> axis(side=1, at=(1:NROW(hurstv)), labels=names(hurstv))
> # Calculate the regression of the PCA Hurst exponents versus thei
> orderv <- 1:NROW(hurstv)
> model <- lm(hurstv ~ orderv)
> summary(model)
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(orderv), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Out-of-Sample Hurst Exponents of Stock Principal Components

The *out-of-sample* Hurst exponents of principal components also decrease with the increasing PCA order, the statistical significance is much lower.

That's because the PCA weights are not persistent *out-of-sample* - the PCA weights in the *out-of-sample* interval are often quite different from the *in-sample* weights.

```
> # Calculate in-sample eigen decomposition using matrix algebra
> eigend <- eigen(cor(retsp["/2010"]))
> # Calculate out-of-sample PCA prices
> retspca <- retsp["2010/"] %*% eigend$vectors
> pricepca <- xts::xts(matrixStats::colCumsums(retspca),
+                      order.by=index(retsp["2010/"]))
> colnames(pricepca) <- paste0("PC", 1:NCOL(retsp))
> # Calculate the Hurst exponents of PCAs
> hurstv <- sapply(pricepca, HighFreq::calc_hurst, aggv=aggv)
> # Dygraph of PCA with largest Hurst exponent
> namev <- names(which.max(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
> # Dygraph of PCA with smallest Hurst exponent
> namev <- names(which.min(hurstv))
> dygraphs::dygraph(get(namev, pricepca), main=namev)
```



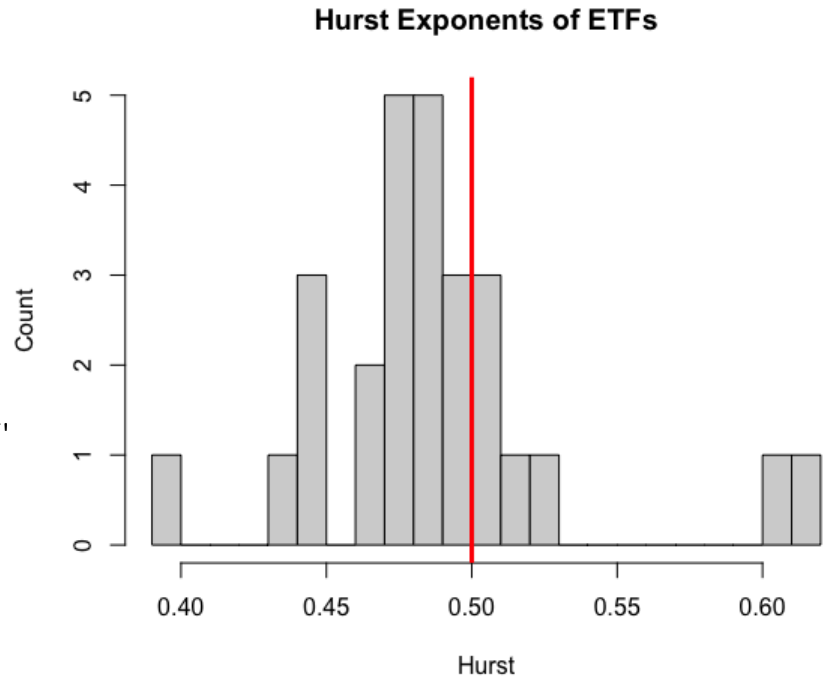

```
> # Plot the Hurst exponents of principal components without x-axis
> plot(hurstv, xlab=NA, ylab=NA, xaxt="n",
+      main="Out-of-Sample Hurst Exponents of Principal Components")
> # Add X-axis with PCA labels
> axis(side=1, at=(1:NROW(hurstv)), labels=names(hurstv))
> # Calculate the regression of the PCA Hurst exponents versus their
> model <- lm(hurstv ~ orderv)
> summary(model)
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(model)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(orderv), y=max(hurstv),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Hurst Exponents of ETFs

The Hurst exponents of ETFs are also typically slightly less than 0.5, but they're closer to 0.5 than stocks, because they're portfolios stocks, so they have less idiosyncratic risk.

For this data sample, the commodity ETFs have the largest Hurst exponents while stock sector ETFs have the smallest Hurst exponents.

```
> # Get ETF log prices
> symbolv <- rutils::etfenv$symbolv
> symbolv <- symbolv[!(symbolv %in% c("MTUM", "QUAL", "VLUE", "USMV'
> pricev <- lapply(mget(symbolv, rutils::etfenv), function(x) {
+   log(na.omit(quantmod::Cl(x)))
+ })  # end lapply
> # Calculate the Hurst exponents of ETFs
> aggv <- trunc(seq.int(from=3, to=10, length.out=5)^2)
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> hurstv <- sort(unlist(hurstv))
> # Dygraph of ETF with smallest Hurst exponent
> namev <- names(first(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
> # Dygraph of ETF with largest Hurst exponent
> namev <- names(last(hurstv))
> dygraphs::dygraph(get(namev, pricev), main=namev)
```



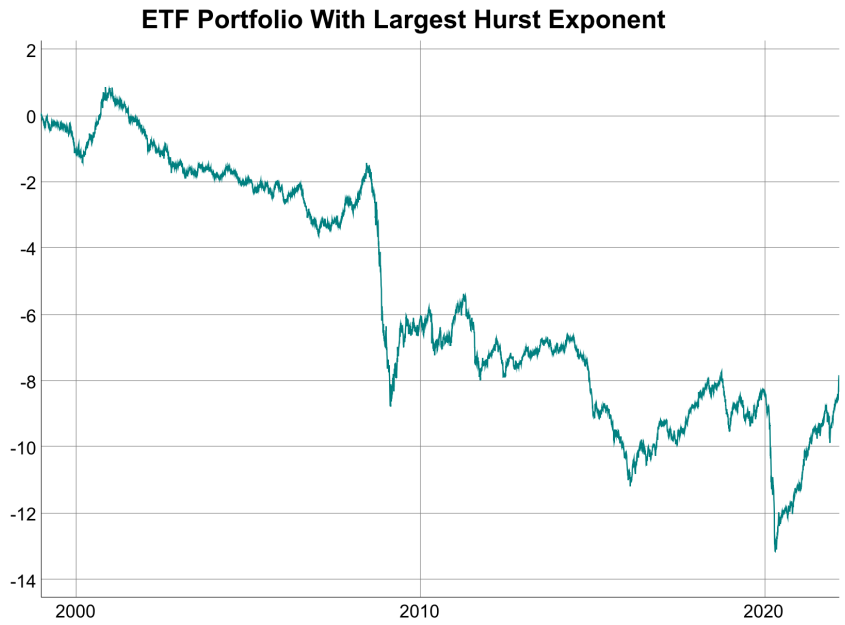

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=2e1, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of ETFs")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

ETF Portfolio With Largest Hurst Exponent

The portfolio weights can be optimized to maximize the portfolio's Hurst exponent.

The optimized portfolio exhibits very strong trending of returns, especially in periods of high volatility.

```
> # Calculate log ETF returns
> symbolv <- rutils::etfenv$symbolv
> symbolv <- symbolv[!(symbolv %in% c("MTUM", "QUAL", "VLUE", "USMV'
> retsp <- rutils::etfenv$returns[, symbolv]
> retsp[is.na(retsp)] <- 0
> sum(is.na(retsp))
> # Calculate the Hurst exponent of an ETF portfolio
> calc_phurst <- function(weightv, retsp) {
+   -HighFreq::calc_hurst(matrix(cumsum(retsp %*% weightv)), aggv=ag
+ }  # end calc_phurst
> # Calculate the portfolio weights with maximum Hurst
> nweights <- NCOL(retsp)
> weightv <- rep(1/sqrt(nweights), nweights)
> calc_phurst(weightv, retsp=retsp)
> optiml <- optim(par=weightv, fn=calc_phurst, retsp=retsp,
+           method="L-BFGS-B",
+           upper=rep(10.0, nweights),
+           lower=rep(-10.0, nweights))
> # Optimal weights and maximum Hurst
> weightv <- optiml$par
> names(weightv) <- colnames(retsp)
> -calc_phurst(weightv, retsp=retsp)
```




```
> # Dygraph of ETF portfolio with largest Hurst exponent
> wealthv <- xts::xts(cumsum(retsp %*% weightv), zoo::index(retsp))
> dygraphs::dygraph(wealthv, main="ETF Portfolio With Largest Hurst
```

Out-of-Sample ETF Portfolio With Largest Hurst Exponent

The portfolio weights can be optimized *in-sample* to maximize the portfolio's Hurst exponent.

But the *out-of-sample* Hurst exponent is close to `H = 0.5`, which means it's close to a random Brownian motion process.

```
> # Calculate the in-sample maximum Hurst portfolio weights
> optiml <- optim(par=weightv, fn=calc_phurst, retsp=retsp["/2010"],
+           method="L-BFGS-B",
+           upper=rep(10.0, nweights),
+           lower=rep(-10.0, nweights))
> # Optimal weights and maximum Hurst
> weightv <- optiml$par
> names(weightv) <- colnames(retsp)
> # Calculate the in-sample Hurst exponent
> -calc_phurst(weightv, retsp=retsp["/2010"])
> # Calculate the out-of-sample Hurst exponent
> -calc_phurst(weightv, retsp=retsp["2010/"])
```

Autoregressive Processes

An *autoregressive* process *AR(n)* of order n for a time series r_i is defined as:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

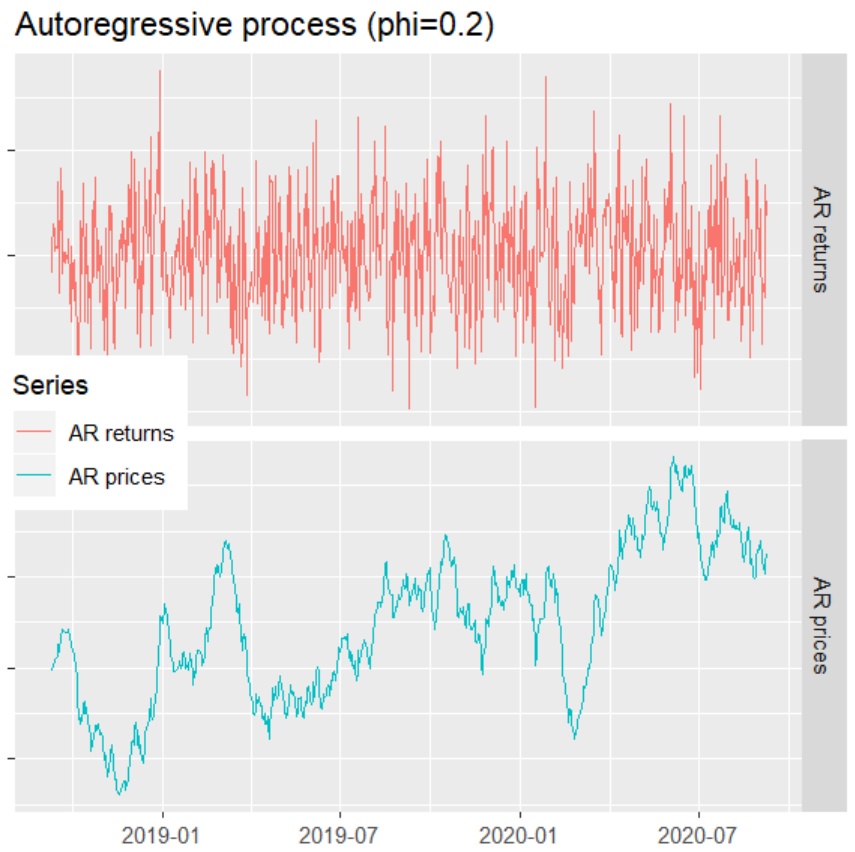
Where φ_i are the *AR(n)* coefficients, and ξ_i are standard normal *innovations*.

The *AR(n)* process is a special case of an *ARIMA* process, and is simply called an *AR(n)* process.

If the *AR(n)* process is *stationary* then the time series r_i is mean reverting to zero.

The function `arima.sim()` simulates *ARIMA* processes, with the `"model"` argument accepting a `list` of *AR(n)* coefficients φ_i.

```
> # Simulate AR processes
> set.seed(1121)  # Reset random numbers
> dates <- Sys.Date() + 0:728  # Two year daily series
> # AR time series of returns
> arimav <- xts(x=arima.sim(n=NROW(dates), model=list(ar=0.2)),
+           order.by=dates)
> arimav <- cbind(arimav, cumsum(arimav))
> colnames(arimav) <- c("AR returns", "AR prices")
```



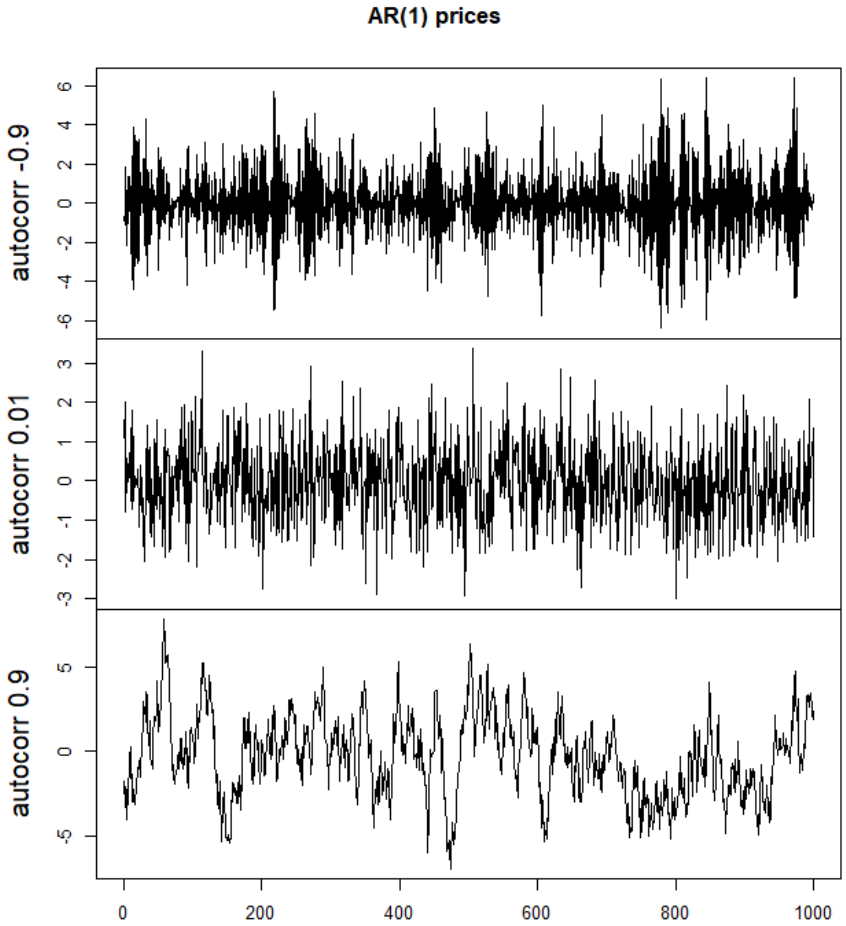

```
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> autoplot(object=arimav, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(legend.position=c(0.1, 0.5),
+   plot.background=element_blank(),
+   axis.text.y=element_blank())
```

Examples of Autoregressive Processes

The speed of mean reversion of an *AR(1)* process depends on the *AR(n)* coefficient φ_1, with a negative coefficient producing faster mean reversion, and a positive coefficient producing stronger diversion.

A positive coefficient φ_1 produces a diversion away from the mean, so that the time series r_i wanders away from the mean for longer periods of time.

```
> coeff <- c(-0.9, 0.01, 0.9)  # AR coefficients
> # Create three AR time series
> arimav <- sapply(coeff, function(phi) {
+   set.seed(1121)  # Reset random numbers
+   arima.sim(n=NROW(dates), model=list(ar=phi))
+ })  # end sapply
> colnames(arimav) <- paste("autocorr", coeff)
> plot.zoo(arimav, main="AR(1) prices", xlab=NA)
> # Or plot using ggplot
> arimav <- xts(x=arimav, order.by=dates)
> library(ggplot)
> autoplot(arimav, main="AR(1) 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())
```



Simulating Autoregressive Processes

An *autoregressive* process *AR(n)*:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

Can be simulated by using an explicit recursive loop in R.

AR(n) processes can also be simulated by using the function `filter()` directly, with the argument `method="recursive"`.

The function `filter()` applies a linear filter to a vector, and returns a time series of class `"ts"`.

```
> # Define AR(3) coefficients and innovations
> coeff <- c(0.1, 0.39, 0.5)
> nrows <- 1e2
> set.seed(1121); innov <- rnorm(nrows)
> # Simulate AR process using recursive loop in R
> arimav <- numeric(nrows)
> arimav[1] <- innov[1]
> arimav[2] <- coeff[1]*arimav[1] + innov[2]
> arimav[3] <- coeff[1]*arimav[2] + coeff[2]*arimav[1] + innov[3]
> for (it in 4:NROW(arimav)) {
+   arimav[it] <- arimav[(it-1):(it-3)] %*% coeff + innov[it]
+ }  # end for
> # Simulate AR process using filter()
> arimafaster <- filter(x=innov, filter=coeff, method="recursive")
> class(arimafaster)
> all.equal(arimav, as.numeric(arimafaster))
> # Fast simulation of AR process using C_rfilter()
> arimacpp <- .Call(stats:::C_rfilter, innov, coeff,
+                   double(NROW(coeff) + NROW(innov)))[-(1:3)]
> all.equal(arimav, arimacpp)
```

Simulating Autoregressive Processes Using arima.sim()

The function `arima.sim()` simulates *ARIMA* processes by calling the function `filter()`.

ARIMA processes can also be simulated by using the function `filter()` directly, with the argument `method="recursive"`.

Simulating stationary *autoregressive* processes requires a *warmup period*, to allow the process to reach its stationary state.

The required length of the *warmup period* depends on the smallest root of the characteristic equation, with a longer *warmup period* needed for smaller roots, that are closer to 1.

The *rule of thumb* (heuristic rule, guideline) is for the *warmup period* to be equal to 6 divided by the logarithm of the smallest characteristic root plus the number of *AR(n)* coefficients: $\frac{6}{\log(minroot)} + \text{numcoeff}$

```
> # Calculate modulus of roots of characteristic equation
> rootv <- Mod(polyroot(c(1, -coeff)))
> # Calculate warmup period
> warmup <- NROW(coeff) + ceiling(6/log(min(rootv)))
> set.seed(1121)
> nrows <- 1e4
> innov <- rnorm(nrows + warmup)
> # Simulate AR process using arima.sim()
> arimav <- arima.sim(n=nrows,
+   model=list(ar=coeff),
+   start.innov=innov[1:warmup],
+   innov=innov[(warmup+1):NROW(innov)])
> # Simulate AR process using filter()
> arimafast <- filter(x=innov, filter=coeff, method="recursive")
> all.equal(arimafast[-(1:warmup)], as.numeric(arimav))
> # Benchmark the speed of the three methods of simulating AR proces
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(x=innov, filter=coeff, method="recursive"),
+   arima_sim=arima.sim(n=nrows,
+                       model=list(ar=coeff),
+                       start.innov=innov[1:warmup],
+                       innov=innov[(warmup+1):NROW(innov)]),
+   arima_loop={for (it in 4:NROW(arimav)) {
+   arimav[it] <- arimav[(it-1):(it-3)] %*% coeff + innov[it]}}
+   ), times=10)[, c(1, 4, 5)]
```

Autocorrelations of Autoregressive Processes

The autocorrelation ρ_i of an *AR(1)* process (defined as $r_i = \varphi r_{i-1} + \xi_i$), satisfies the recursive equation: $\rho_i = \varphi \rho_{i-1}$, with $\rho_1 = \varphi$.

Therefore *AR(1)* processes have exponentially decaying autocorrelations: $\rho_i = \varphi^i$.

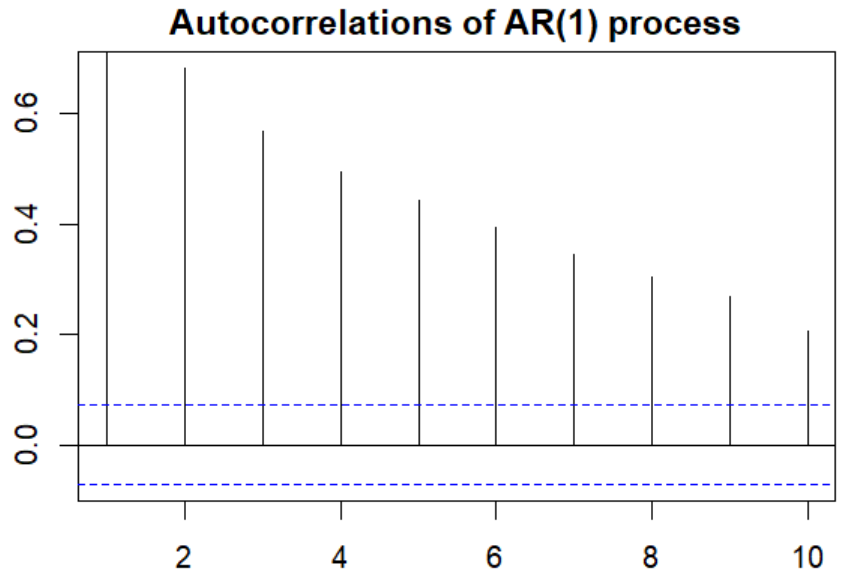
The *AR(1)* process can be simulated recursively:

$$r_1 = \xi_1$$
$$r_2 = \varphi r_1 + \xi_2 = \xi_2 + \varphi \xi_1$$
$$r_3 = \xi_3 + \varphi \xi_2 + \varphi^2 \xi_1$$
$$r_4 = \xi_4 + \varphi \xi_3 + \varphi^2 \xi_2 + \varphi^3 \xi_1$$

Therefore the *AR(1)* process can be expressed as a *moving average* (*MA*) of the *innovations* ξ_i: $r_i = \sum_{i=1}^{n} \varphi^{i-1} \xi_i$.

If $\varphi < 1.0$ then the influence of the innovation ξ_i decays exponentially.

If $\varphi = 1.0$ then the influence of the random innovations ξ_i persists indefinitely, so that the variance of r_i increases linearly with time.



An *AR(1)* process has an exponentially decaying *ACF*.

```
> x11(width=6, height=4)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> # Simulate AR(1) process
> arimav <- arima.sim(n=1e3, model=list(ar=0.8))
> # ACF of AR(1) process
> acfv <- rutils::plot_acf(arimav, lag=10, xlab="", ylab="",
+   main="Autocorrelations of AR(1) process")
> acfv$acf[1:5]
```

Partial Autocorrelations

An autocorrelation of lag 1 induces higher order autocorrelations of lag `2, 3, ...`, which may obscure the direct higher order autocorrelations.

If two random variables are both correlated to a third variable, then they are indirectly correlated with each other.

The indirect correlation can be removed by defining new variables with no correlation to the third variable.

The *partial correlation* is the correlation after the correlations to the common variables are removed.

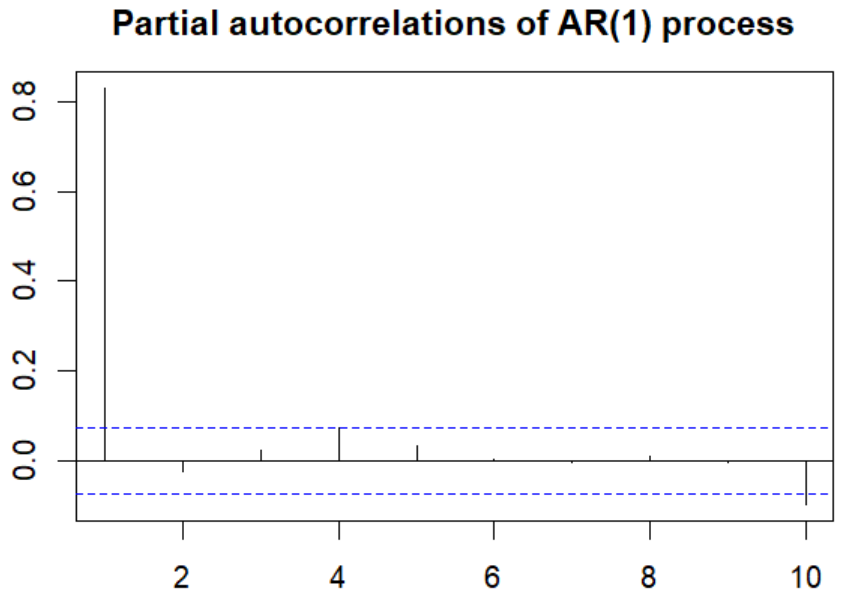
A linear combination of the time series and its own lag can be created, such that its lag 1 autocorrelation is zero.

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 of lag `k`, after all the autocorrelations of lag `1, ..., k-1` have been removed.

The *partial autocorrelations* ϱ_i are the estimators of the coefficients ϕ_i of the *AR(n)* process.

The function `pacf()` calculates and plots the *partial autocorrelations* using the Durbin-Levinson algorithm.



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

```
> # PACF of AR(1) process
> pacfv <- pacf(arimav, lag=10, xlab="", ylab="", main="")
> title("Partial autocorrelations of AR(1) process", line=1)
> pacfv <- as.numeric(pacfv$acf)
> pacfv[1:5]
```

draft: Higher Order Autocorrelations

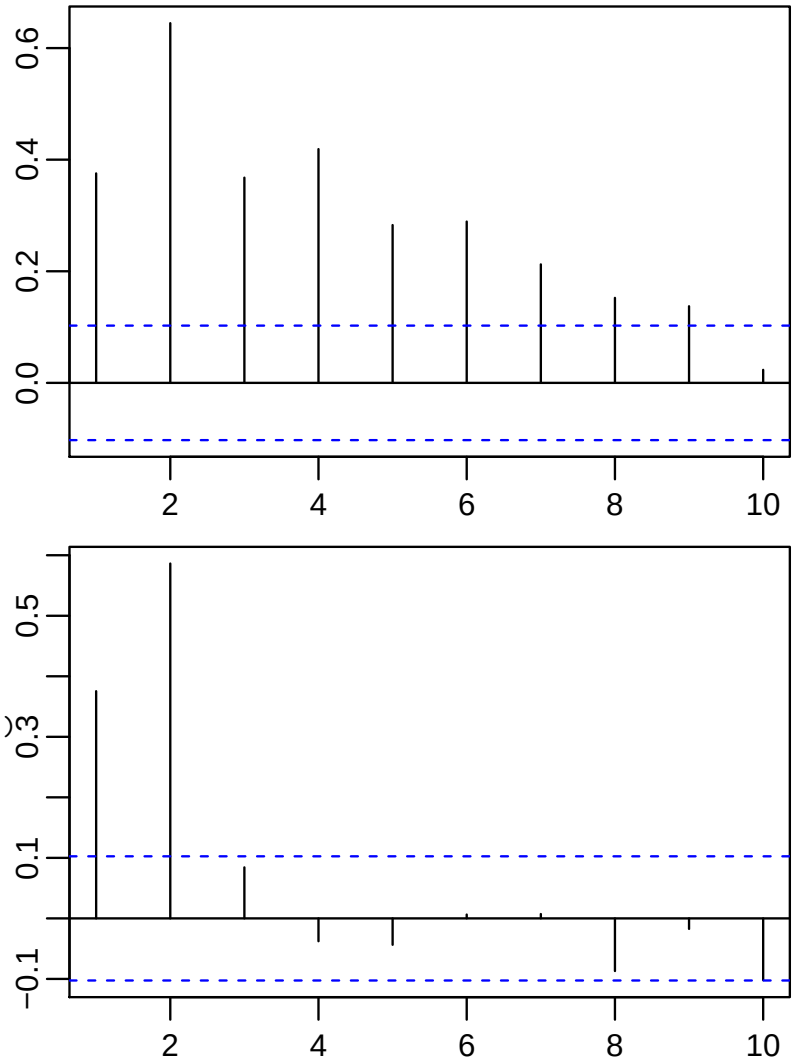
An *AR(3)* process of order *three* is defined by the formula:

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

Autoregressive processes *AR(n)* of order *n* have an exponentially decaying *ACF* and a non-zero *PACF* up to lag *n*.

The number of non-zero *partial autocorrelations* is equal to the *order* parameter *n* of the *AR(n)* process.

```
> # Set two vertical plot panels
> par(mfrow=c(2,1))
> # Simulate AR process of returns
> arimav <- arima.sim(n=1e5, model=list(ar=c(0.0, 0.5, 0.1)))
> # ACF of AR(3) process
> rutils::plot_acf(arimav, lag=10, xlab="", ylab="",
+    main="ACF of AR(3) process")
> # PACF of AR(3) process
> pacf(arimav, lag=10, xlab="", ylab="", main="PACF of AR(3) process")
```



Stationary Processes and Unit Root Processes

A process is *stationary* if its probability distribution does not change with time, which means that it has constant mean and variance.

The *autoregressive* process *AR(n)*:
$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$

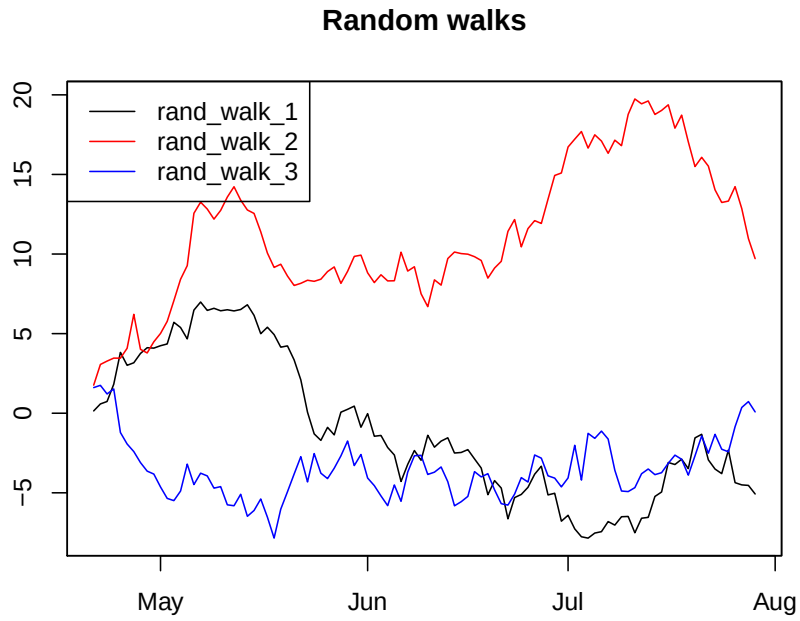
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_n z^n = 0$

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

If the sum of the autoregressive coefficients is equal to 1: $\sum_{i=1}^{n} \varphi_i = 1$, then the process has a root equal to 1 (it has a *unit root*), so it's not *stationary*.

Non-stationary processes with unit roots are called *unit root* processes.

A simple example of a *unit root* process is the *Brownian Motion*: $p_i = p_{i-1} + \xi_i$




```
> rand_walk <- cumsum(zoo(matrix(rnorm(3*100), ncol=3),
+             order.by=(Sys.Date()+0:99)))
> colnames(rand_walk) <- paste("rand_walk", 1:3, sep="_")
> plot.zoo(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 cumulative sum of a given process is called its *integrated* process.

For example, asset prices follow an *integrated* process with respect to asset returns: $p_n = \sum_{i=1}^{n} r_i$.

If returns follow an *AR(n)* process:
$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$

Then asset prices follow the process:
$p_i = (1 + \varphi_1) p_{i-1} + (\varphi_2 - \varphi_1) p_{i-2} + \ldots + (\varphi_n - \varphi_{n-1}) p_{i-n} - \varphi_n p_{i-n-1} + \xi_i$

The sum of the coefficients of the price process is equal to 1, so it has a *unit root* for all values of the φ_i coefficients.

The *integrated* process of an *AR(n)* process is always a *unit root* process.

For example, if returns follow an *AR(1)* process:
$r_i = \varphi r_{i-1} + \xi_i$.

Then asset prices follow the process:
$p_i = (1 + \varphi) p_{i-1} - \varphi p_{i-2} + \xi_i$

Which is a *unit root* process for all values of φ, because the sum of its coefficients is equal to 1.

If $\varphi = 0$ then the above process is a *Brownian Motion* (random walk).

```
> # Simulate arima with large AR coefficient
> set.seed(1121)
> nrows <- 1e4
> arimav <- arima.sim(n=nrows, model=list(ar=0.99))
> tseries::adf.test(arimav)
> # Integrated series has unit root
> tseries::adf.test(cumsum(arimav))
> # Simulate arima with negative AR coefficient
> set.seed(1121)
> arimav <- arima.sim(n=nrows, model=list(ar=-0.99))
> tseries::adf.test(arimav)
> # Integrated series has unit root
> tseries::adf.test(cumsum(arimav))
```

The Variance of Unit Root Processes

An *AR(1)* process: $r_i = \varphi r_{i-1} + \xi_i$ has the following characteristic equation: $1 - \varphi z = 0$, with a root equal to: $z = 1/\varphi$

If $\varphi = 1$, then the characteristic equation has a *unit root* (and therefore it isn't *stationary*), and the process follows: $r_i = r_{i-1} + \xi_i$

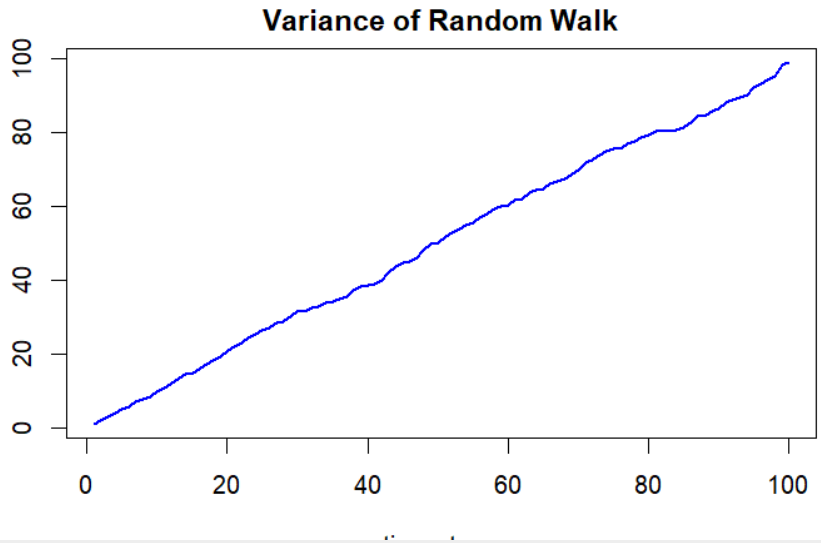
The above is called a *Brownian Motion*, and it's an example of a *unit root* process.

The expected value of the *AR(1)* process $r_i = \varphi r_{i-1} + \xi_i$ is equal to zero: $\mathbb{E}[r_i] = \frac{\mathbb{E}[\xi_i]}{1-\varphi} = 0$.

And its variance is equal to: $\sigma^2 = \mathbb{E}[r_i^2] = \frac{\sigma_\xi^2}{1-\varphi^2}$.

If $\varphi = 1$, then the *variance* grows over time and becomes infinite over time, so the process isn't *stationary*.

The variance of the *Brownian Motion* $r_i = r_{i-1} + \xi$ is proportional to time: $\sigma_i^2 = \mathbb{E}[r_i^2] = i\sigma_\xi^2$




```
> # Simulate random walks using apply() loops
> set.seed(1121)  # Initialize random number generator
> rand_walks <- matrix(rnorm(1000*100), ncol=1000)
> rand_walks <- apply(rand_walks, 2, cumsum)
> variance <- apply(rand_walks, 1, var)
> # Simulate random walks using vectorized functions
> set.seed(1121)  # Initialize random number generator
> rand_walks <- matrixStats::colCumsums(matrix(rnorm(1000*100), ncol
> variance <- matrixStats::rowVars(rand_walks)
> par(mar=c(5, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot(variance, xlab="time steps", ylab="",
+      t="l", col="blue", lwd=2,
+      main="Variance of Random Walk")
```

The Brownian Motion Process

In the *Brownian Motion* process, the returns r_i are equal to the random *innovations*:

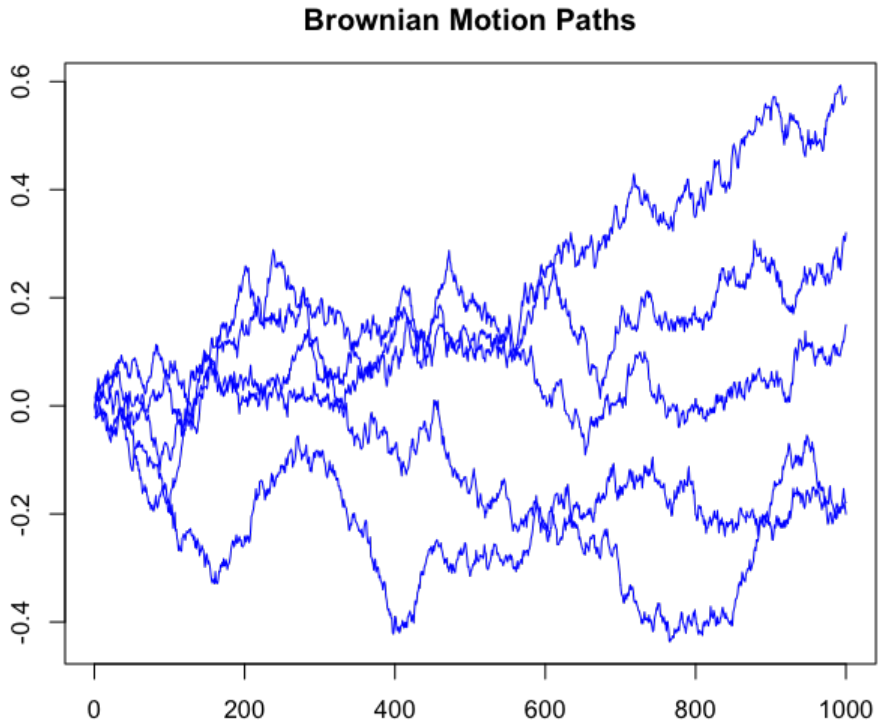
$$r_i = p_i - p_{i-1} = \sigma \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where σ is the volatility of returns, and ξ_i are random normal *innovations* with zero mean and unit variance.

The *Brownian Motion* process for prices can be written as an *AR(1)* autoregressive process with coefficient $\varphi = 1$:

$$p_i = \varphi p_{i-1} + \sigma \xi_i$$




```
> # Define Brownian Motion parameters
> nrows <- 1000; sigmav <- 0.01
> # Simulate 5 paths of Brownian motion
> pricev <- matrix(rnorm(5*nrows, sd=sigmav), nc=5)
> pricev <- matrixStats::colCumsums(pricev)
> # Open plot window on Mac
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(2, 2, 3, 1), oma=c(0, 0, 0, 0))
> # Plot 5 paths of Brownian motion
> matplot(y=prices, main="Brownian Motion Paths",
+   xlab="", ylab="", type="l", lty="solid", lwd=1, col="blue")
> # Save plot to png file on Mac
> quartz.save("figure/brown_paths.png", type="png", width=6, height=4)
```

The Ornstein-Uhlenbeck Process

In the *Ornstein-Uhlenbeck* process, the returns r_i are equal to the difference between the equilibrium price μ minus the latest price p_{i-1}, times the mean reversion parameter θ, plus random *innovations*:

$$r_i = p_i - p_{i-1} = \theta\,(\mu - p_{i-1}) + \sigma\,\xi_i$$

$$p_i = p_{i-1} + r_i$$

Where σ is the volatility of returns, and ξ_i are random normal *innovations* with zero mean and unit variance.

The *Ornstein-Uhlenbeck* process for prices can be written as an *AR(1)* process plus a drift:

$$p_i = \theta\,\mu + (1 - \theta)\,p_{i-1} + \sigma\,\xi_i$$

The *Ornstein-Uhlenbeck* process cannot be simulated using the function `filter()` because of the drift term, so it must be simulated using explicit loops, either in `R` or in `C++`.

The compiled *Rcpp* `C++` code can be over 100 times faster than loops in `R`!

```
> # Define Ornstein-Uhlenbeck parameters
> init_price <- 0.0; priceq <- 1.0;
> sigmav <- 0.02; thetav <- 0.01; nrows <- 1000
> # Initialize the data
> innov <- rnorm(nrows)
> retsp <- numeric(nrows)
> pricev <- numeric(nrows)
> retsp[1] <- sigmav*innov[1]
> pricev[1] <- init_price
> # Simulate Ornstein-Uhlenbeck process in R
> for (i in 2:nrows) {
+   retsp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+   pricev[i] <- pricev[i-1] + retsp[i]
+ }  # end for
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> prices_cpp <- HighFreq::sim_ou(init_price=init_price, eq_price=pri
+   theta=thetav, innov=matrix(innov))
> all.equal(pricev, drop(pricev_cpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (i in 2:nrows) {
+     retsp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+     pricev[i] <- pricev[i-1] + retsp[i]}},
+   Rcpp=HighFreq::sim_ou(init_price=init_price, eq_price=priceq,
+     theta=thetav, innov=matrix(innov)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

The Solution of the Ornstein-Uhlenbeck Process

The *Ornstein-Uhlenbeck* process in continuous time is:

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

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

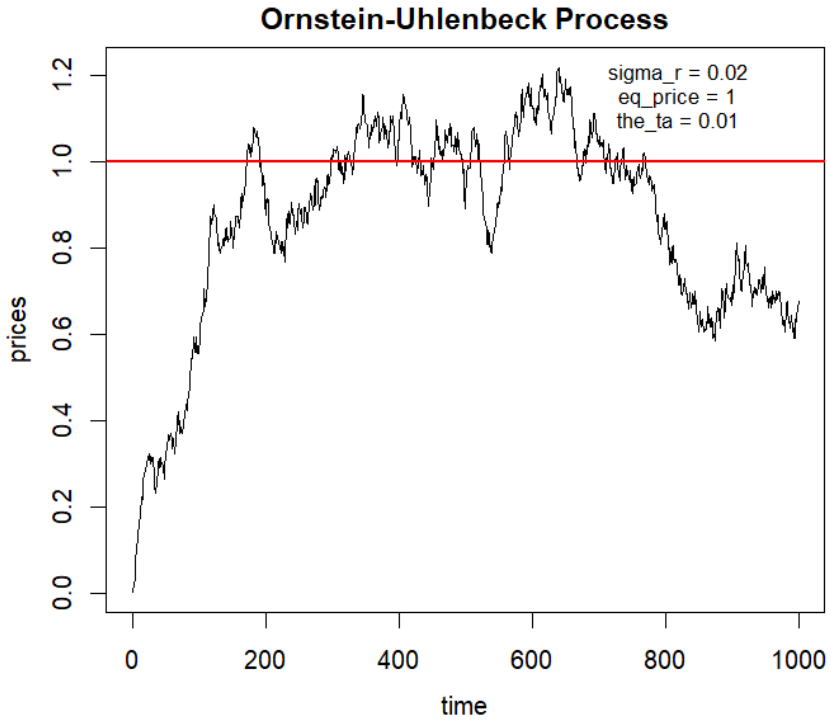
The solution of the *Ornstein-Uhlenbeck* process is given by:

$$p_t = p_0 e^{-\theta t} + \mu(1 - e^{-\theta t}) + \sigma \int_0^t e^{\theta(s-t)} \mathrm{d}W_s$$

The mean and variance are given by:
$\mathbb{E}[p_t] = p_0 e^{-\theta t} + \mu(1 - e^{-\theta t}) \to \mu$
$\mathbb{E}[(p_t - \mathbb{E}[p_t])^2] = \frac{\sigma^2}{2\theta}(1 - e^{-\theta t}) \to \frac{\sigma^2}{2\theta}$

The *Ornstein-Uhlenbeck* process is mean reverting to a non-zero equilibrium price μ.

The *Ornstein-Uhlenbeck* process needs a *warmup period* before it reaches equilibrium.

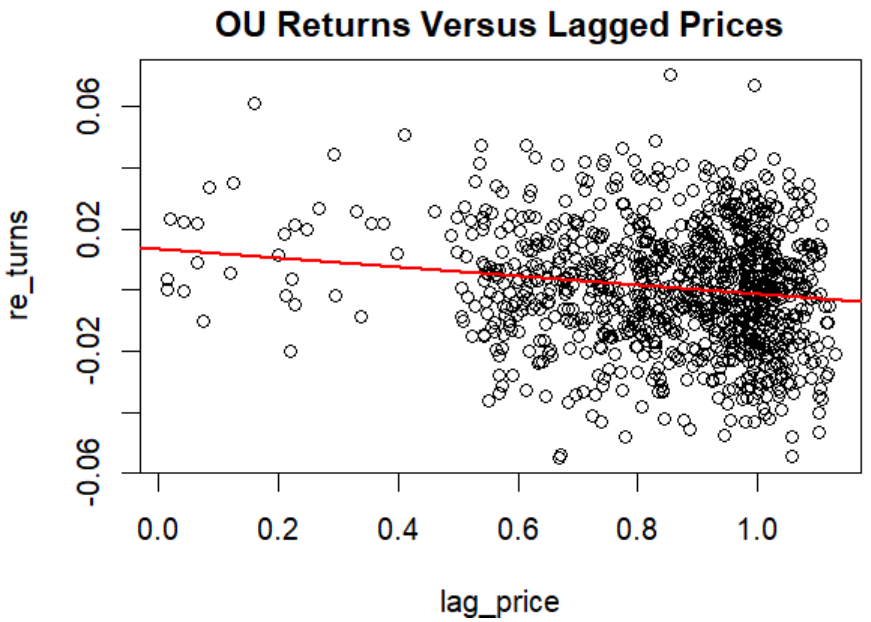



```
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Ornstein-Uhlenbeck Process")
> legend("topright", title=paste(c(paste0("sigmav = ", sigmav),
+                paste0("eq_price = ", ),
+                paste0("thetav = ", thetav)),
+              collapse="\n"),
+  legend="", cex=0.8, inset=0.1, bg="white", bty="n")
> abline(h=, col='red', lwd=2)
```

Ornstein-Uhlenbeck Process Returns Correlation

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

```
> retsp <- rutils::diffit(pricev)
> pricelag <- rutils::lagit(pricev)
> formulav <- retsp ~ pricelag
> regmod <- lm(formulav)
> summary(regmod)
> # Plot regression
> plot(formulav, main="OU Returns Versus Lagged Prices")
> abline(regmod, lwd=2, col="red")
```



Calibrating the Ornstein-Uhlenbeck Parameters

The volatility parameter of the Ornstein-Uhlenbeck process can be estimated directly from the standard deviation of the returns.

The θ and μ parameters can be estimated from the linear regression of the returns versus the lagged prices.

Calculating regression parameters directly from formulas has the advantage of much faster calculations.

```
> # Calculate volatility parameter
> c(volatility=sigmav, estimate=sd(retsp))
> # Extract OU parameters from regression
> coeff <- summary(regmod)$coefficients
> # Calculate regression alpha and beta directly
> betav <- cov(retsp, pricelag)/var(pricelag)
> alpha <- (mean(retsp) - betav*mean(pricelag))
> cbind(direct=c(alpha=alpha, beta=betav), lm=coeff[, 1])
> all.equal(c(alpha=alpha, beta=betav), coeff[, 1],
+     check.attributes=FALSE)
> # Calculate regression standard errors directly
> betas <- c(alpha=alpha, beta=betav)
> fittedv <- (alpha + betav*pricelag)
> residuals <- (retsp - fittedv)
> prices2 <- sum((pricelag - mean(pricelag))^2)
> betasd <- sqrt(sum(residuals^2)/prices2/(nrows-2))
> alphasd <- sqrt(sum(residuals^2)/(nrows-2)*(1:nrows + mean(pricela
> cbind(direct=c(alphasd=alphasd, betasd=betasd), lm=coeff[, 2])
> all.equal(c(alphasd=alphasd, betasd=betasd), coeff[, 2],
+     check.attributes=FALSE)
> # Compare mean reversion parameter theta
> c(theta=(-thetav), round(coeff[2, ], 3))
> # Compare equilibrium price mu
> c(priceq=priceq, estimate=-coeff[1, 1]/coeff[2, 1])
> # Compare actual and estimated parameters
> coeff <- cbind(c(thetav*priceq, -thetav), coeff[, 1:2])
> rownames(coeff) <- c("drift", "theta")
> colnames(coeff)[1] <- "actual"
> round(coeff, 4)
```

The Schwartz Process

The *Ornstein-Uhlenbeck* prices can be negative, while actual prices are usually not negative.

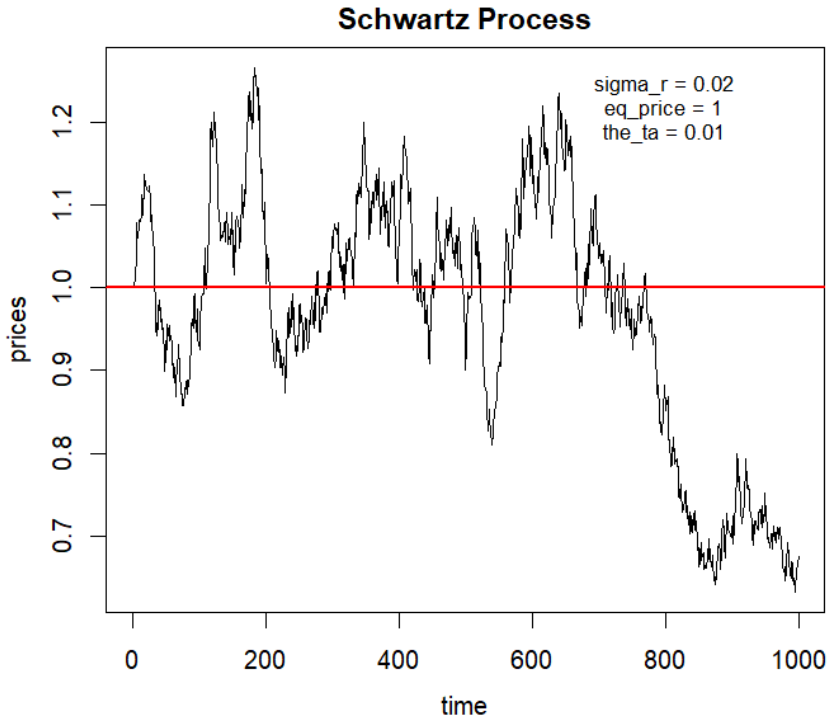
So the *Ornstein-Uhlenbeck* process is better suited for simulating the logarithm of prices, which can be negative.

The *Schwartz* process is the exponential of the *Ornstein-Uhlenbeck* process, so it avoids negative prices by compounding the percentage returns r_i instead of summing them:

$$r_i = \log p_i - \log p_{i-1} = \theta\,(\mu - p_{i-1}) + \sigma\,\xi_i$$

$$p_i = p_{i-1}\exp(r_i)$$

Where the parameter θ is the strength of mean reversion, σ is the volatility, and ξ_i are random normal *innovations* with zero mean and unit variance.




```
> # Simulate Schwartz process
> retsp <- numeric(nrows)
> pricev <- numeric(nrows)
> pricev[1] <- exp(sigmav*innov[1])
> set.seed(1121)  # Reset random numbers
> for (i in 2:nrows) {
+   retsp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+   pricev[i] <- pricev[i-1]*exp(retsp[i])
+ }  # end for
```

```
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Schwartz Process")
> legend("topright",
+  title=paste(c(paste0("sigmav = ", sigmav),
+                paste0("priceq = ", priceq),
+                paste0("thetav = ", thetav)),
+              collapse="\n"),
+  legend="", cex=0.8, inset=0.12, bg="white", bty="n")
> abline(h=priceq, col='red', lwd=2)
```

The Dickey-Fuller Process

The *Dickey-Fuller* process is a combination of an *Ornstein-Uhlenbeck* process and an *autoregressive* process.

The returns r_i are equal to the sum of a mean reverting term plus *autoregressive* terms:

$$r_i = \theta(\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \sigma \xi_i$$

$$p_i = p_{i-1} + r_i$$

Where μ is the equilibrium price, σ is the volatility of returns, θ is the strength of mean reversion, and ξ_i are standard normal *innovations*.

Then the prices follow an *autoregressive* process:

$$p_i = \theta\mu + (1 + \varphi_1 - \theta)p_{i-1} + (\varphi_2 - \varphi_1)p_{i-2} + \ldots + (\varphi_n - \varphi_{n-1})p_{i-n} - \varphi_n p_{i-n-1} + \sigma \xi_i$$

The sum of the *autoregressive* coefficients is equal to $1 - \theta$, so if the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_i exhibits mean reversion and has no *unit root*.

```
> # Define Dickey-Fuller parameters
> init_price <- 0.0;  priceq <- 1.0
> thetav <- 0.01;  nrows <- 1000
> coeff <- c(0.1, 0.39, 0.5)
> # Initialize the data
> innov <- rnorm(nrows, sd=0.01)
> retsp <- numeric(nrows)
> pricev <- numeric(nrows)
> # Simulate Dickey-Fuller process using recursive loop in R
> retsp[1] <- innov[1]
> pricev[1] <- init_price
> retsp[2] <- thetav*(priceq - pricev[1]) + coeff[1]*retsp[1] + inno
> pricev[2] <- pricev[1] + retsp[2]
> retsp[3] <- thetav*(priceq - pricev[2]) + coeff[1]*retsp[2] + coef
> pricev[3] <- pricev[2] + retsp[3]
> for (it in 4:nrows) {
+   retsp[it] <- thetav*(priceq - pricev[it-1]) + retsp[(it-1):(it-3
+   pricev[it] <- pricev[it-1] + retsp[it]
+ }  # end for
> # Simulate Dickey-Fuller process in Rcpp
> prices_cpp <- HighFreq::sim_df(init_price=init_price, eq_price=pri
> # Compare prices
> all.equal(pricev, drop(pricev_cpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (it in 4:nrows) {
+   retsp[it] <- thetav*(priceq - pricev[it-1]) + retsp[(it-1):(it-3
+   pricev[it] <- pricev[it-1] + retsp[it]
+   }},
+   Rcpp=HighFreq::sim_df(init_price=init_price, eq_price=priceq, th
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Augmented Dickey-Fuller ADF Test for Unit Roots

The *Augmented Dickey-Fuller ADF* test is designed to test the *null hypothesis* that a time series has a *unit root*.

The *ADF* test fits an autoregressive model for the prices p_i:

$$r_i = \theta(\mu - p_{i-1}) + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \sigma\,\xi_i$$

$$p_i = p_{i-1} + r_i$$

Where μ is the equilibrium price, σ is the volatility of returns, and θ is the strength of mean reversion.

ε_i are the *residuals*, which are assumed to be standard normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

If the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_i exhibits mean reversion and has no *unit root*.

The *null hypothesis* is that prices have a unit root ($\theta = 0$, no mean reversion), while the alternative hypothesis is that it's *stationary* ($\theta > 0$, mean reversion).

The *ADF* test statistic is equal to the *t*-value of the θ parameter: $t_\theta = \hat{\theta}/SE_\theta$ (which follows a distribution different from the t-distribution).

The function `tseries::adf.test()` performs the *ADF* test.

```
> set.seed(1121); innov <- matrix(rnorm(1e4, sd=0.01))
> # Simulate AR(1) process with coefficient=1, with unit root
> arimav <- HighFreq::sim_ar(coeff=matrix(1), innov=innov)
> x11(); plot(arimav, t="l", main="AR(1) coefficient = 1.0")
> # Perform ADF test with lag = 1
> tseries::adf.test(arimav, k=1)
> # Perform standard Dickey-Fuller test
> tseries::adf.test(arimav, k=0)
> # Simulate AR(1) with coefficient close to 1, without unit root
> arimav <- HighFreq::sim_ar(coeff=matrix(0.99), innov=innov)
> x11(); plot(arimav, t="l", main="AR(1) coefficient = 0.99")
> tseries::adf.test(arimav, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with mean reversion
> init_price <- 0.0; priceq <- 0.0; thetav <- 0.1
> pricev <- HighFreq::sim_ou(init_price=init_price, eq_price=priceq,
+   theta=thetav, innov=innov)
> x11(); plot(pricev, t="l", main=paste("OU coefficient =", thetav))
> tseries::adf.test(pricev, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with zero reversion
> thetav <- 0.0
> pricev <- HighFreq::sim_ou(init_price=init_price, eq_price=priceq,
+   theta=thetav, innov=innov)
> x11(); plot(pricev, t="l", main=paste("OU coefficient =", thetav))
> tseries::adf.test(pricev, k=1)
```

The common practice is to use a small number of lags in the *ADF* test, and if the residuals are autocorrelated, then to increase them until the correlations are no longer significant.

If the number of lags in the regression is zero: $n = 0$ then the *ADF* test becomes the standard *Dickey-Fuller* test: $r_i = \theta(\mu - p_{i-1}) + \varepsilon_i$.

draft: Calculating the ADF Test Statistic

Calculate the *ADF* Test statistic using matrix algebra.

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 a regression model to determine if the price time series p_i exhibits mean reversion:

$$r_i = \theta p_{i-1} + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \xi_i$$

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

$$p_i = (1+\theta) p_{i-1} + \varphi_1 r_{i-1} + \ldots + \varphi_n r_{i-n} + \xi_i$$

If the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_i exhibits mean reversion and has no *unit root*.

The *null hypothesis* is that the price process has a unit root ($\theta = 0$, no mean reversion), while the alternative hypothesis is that it's *stationary* ($\theta > 0$, mean reversion).

The *ADF* test statistic is equal to the *t*-value of the θ parameter: $t_\theta = \hat{\theta}/SE_\theta$ (which follows a distribution different from the t-distribution).

The common practice is to perform the *ADF* test with a small number of lags, and if the residuals are autocorrelated, then to increase the number of lags until the correlations are no longer significant.

If the number of lags in the regression is zero: $n = 0$ then the *ADF* test becomes the standard *Dickey-Fuller* test: $r_i = \theta p_{i-1} + \xi_i$.

The function `tseries::adf.test()` performs the *ADF* test.

```
> nrows <- 1e3
> # Perform ADF test for AR(1) with small coefficient
> set.seed(1121)
> arimav <- arima.sim(n=nrows, model=list(ar=0.01))
> tseries::adf.test(arimav)
> # Perform ADF test for AR(1) with large coefficient
> set.seed(1121)
> arimav <- arima.sim(n=nrows, model=list(ar=0.99))
> tseries::adf.test(arimav)
> # Perform ADF test with lag = 1
> tseries::adf.test(arimav, k=1)
> # Perform Dickey-Fuller test
> tseries::adf.test(arimav, k=0)
```

Sensitivity of the ADF Test for Detecting Unit Roots

The *ADF null hypothesis* is that prices have a unit root, while the alternative hypothesis is that they're *stationary*.

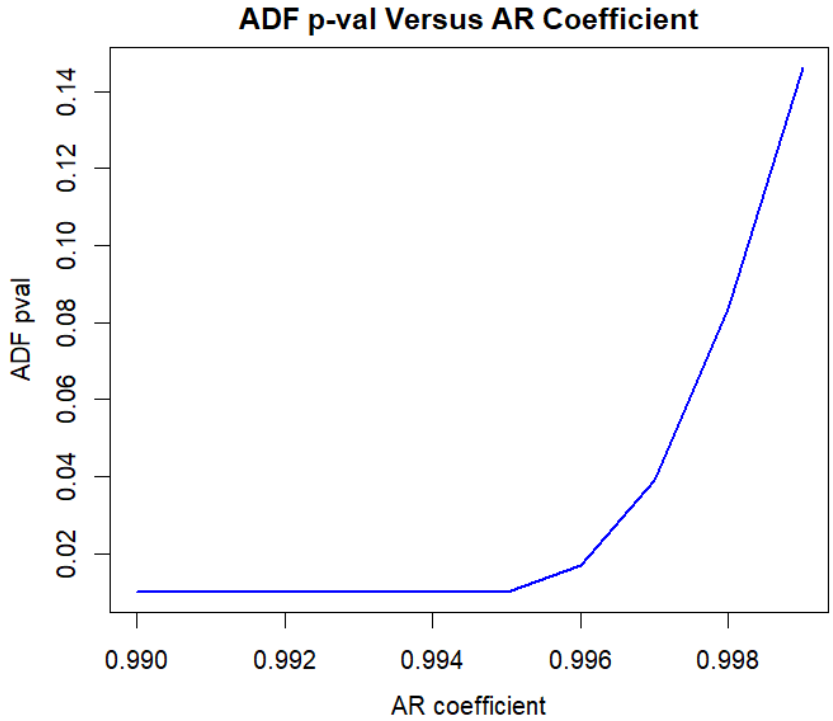
The *ADF* test has low *sensitivity*, i.e. the ability to correctly identify time series with no *unit root*, causing it to produce *false negatives* (*type II* errors).

This is especially true for time series which exhibit mean reversion over longer time horizons. The *ADF* test will identify them as having a *unit root* even though they are mean reverting.

Therefore the *ADF* test often requires a lot of data before it's able to correctly identify *stationary* time series with *no unit root*.

A *true negative* test result is that the *null hypothesis* is `TRUE` (pricev have a unit root), while a *true positive* result is that the *null hypothesis* is `FALSE` (pricev are stationary).

The function `tseries::adf.test()` assumes that the data is *normally distributed*, which may underestimate the standard errors of the parameters, and produce *false positives* (*type I* errors) by incorrectly rejecting the null hypothesis of a unit root process.




```
> # Simulate AR(1) process with different coefficients
> coeffv <- seq(0.99, 0.999, 0.001)
> retsp <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> adft <- sapply(coeffv, function(coeff) {
+   arimav <- filter(x=retsp, filter=coeff, method="recursive")
+   adft <- suppressWarnings(tseries::adf.test(arimav))
+   c(adfstat=unname(adft$statistic), pval=adft$p.value)
+ })  # end sapply
> dev.new(width=6, height=4, noRStudioGD=TRUE)
> # x11(width=6, height=4)
> plot(x=coeffv, y=adft["pval", ], main="ADF p-val Versus AR Coeffic
+      xlab="AR coefficient", ylab="ADF pval", t="l", col="blue", lw
> plot(x=coeffv, y=adft["adfstat", ], main="ADF Stat Versus AR Coeff
+      xlab="AR coefficient", ylab="ADF stat", t="l", col="blue", lw
```

Fitting Time Series to Autoregressive Models

An *autoregressive* process *AR(n)* for the time series of returns r_i:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$$

The coefficients φ can be calculated using linear regression, with the *response* equal to $\mathbf{r}$, and the columns of the *predictor matrix* $\mathbb{P}$ equal to the lags of $\mathbf{r}$:

$$\varphi = \mathbb{P}^{-1}\mathbf{r}$$

An intercept term can be added to the above formula by adding a unit column to the predictor matrix $\mathbb{P}$.

Adding the intercept term produces slightly different coefficients, depending on the mean of the returns.

The function `stats::ar.ols()` fits an *AR(n)* model, but it produces slightly different coefficients than linear regression, because it uses a different calibration procedure.

```
> # Specify AR process parameters
> nrows <- 1e3
> coeff <- matrix(c(0.1, 0.39, 0.5)); ncoeff <- NROW(coeff)
> set.seed(1121); innov <- matrix(rnorm(nrows))
> # arimav <- filter(x=innov, filter=coeff, method="recursive")
> # Simulate AR process using HighFreq::sim_ar()
> arimav <- HighFreq::sim_ar(coeff=coeff, innov=innov)
> # Fit AR model using ar.ols()
> arfit <- ar.ols(arimav, order.max=ncoeff, aic=FALSE)
> class(arfit)
> is.list(arfit)
> drop(arfit$ar); drop(coeff)
> # Define predictor matrix without intercept column
> predictor <- sapply(1:ncoeff, rutils::lagit, input=arimav)
> # Fit AR model using regression
> predinv <- MASS::ginv(predictor)
> coeff <- drop(predinv %*% arimav)
> all.equal(drop(arfit$ar), coeff, check.attributes=FALSE)
```

draft: Calibrating Autoregressive Models Using Maximum Likelihood

An *autoregressive* process *AR(n)* defined as:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$$

Can be expressed as a *multivariate* linear regression model, with the *response* equal to r_i, and the columns of the *predictor matrix* equal to the lags of r_i.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series, using the *maximum likelihood* method (which may give slightly different coefficients than the linear regression model).

```
> # Specify AR process parameters
> nrows <- 1e3
> coeff <- c(0.1, 0.39, 0.5); ncoeff <- NROW(coeff)
> # Simulate AR process using C_rfilter()
> set.seed(1121); innov <- rnorm(nrows, sd=0.01)
> arimav <- .Call(stats:::C_rfilter, innov, coeff,
+   double(nrows + ncoeff))[-(1:ncoeff)]
>
>
> # wippp
> # Calibrate ARIMA model using regression
> # Define predictor matrix
> arimav <- (arimav - mean(arimav))
> predictor <- sapply(1:3, rutils::lagit, input=arimav)
> # Calculate de-meaned returns matrix
> predictor <- t(t(predictor) - colMeans(predictor))
> predinv <- MASS::ginv(predictor)
> # Regression coefficients with response equal to arimav
> coeff <- drop(predinv %*% arimav)
>
> all.equal(arima_fit$coef, coeff, check.attributes=FALSE)
```

The Standard Errors of the *AR(n)* Coefficients

The *standard errors* of the fitted *AR(n)* coefficients are proportional to the standard deviation of the fitted residuals.

Their *t*-values are equal to the ratio of the fitted coefficients divided by their standard errors.

```
> # Calculate the regression residuals
> fittedv <- drop(predictor %*% coeff)
> residuals <- drop(arimav - fittedv)
> # Variance of residuals
> residvar <- sum(residuals^2)/(nrows-NROW(coeff))
> # predictor matrix squared
> predictor2 <- crossprod(predictor)
> # Calculate covariance matrix of AR coefficients
> covar <- residvar*MASS::ginv(predictor2)
> coeffsd <- sqrt(diag(covar))
> # Calculate t-values of AR coefficients
> coefftv <- drop(coeff)/coeffsd
```

Order Selection of *AR(n)* Model

Order selection means determining the *order parameter* n of the *AR(n)* model that best fits the time series.

The order parameter n can be set equal to the number of significantly non-zero *partial autocorrelations* of the time series.

The order parameter can also be determined by only selecting coefficients with statistically significant t-values.

Fitting an *AR(n)* model can be performed by first determining the order n, and then calculating the coefficients.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series.

The function `auto.arima()` from the package *forecast* performs order selection, and calibrates an *AR(n)* model to a univariate time series.

```
> # Fit AR(5) model into AR(3) process
> predictor <- sapply(1:5, rutils::lagit, input=arimav)
> predinv <- MASS::ginv(predictor)
> coeff <- drop(predinv %*% arimav)
> # Calculate t-values of AR(5) coefficients
> residuals <- drop(arimav - drop(predictor %*% coeff))
> residvar <- sum(residuals^2)/(nrows-NROW(coeff))
> covar <- residvar*MASS::ginv(crossprod(predictor))
> coeffsd <- sqrt(diag(covar))
> coefftv <- drop(coeff)/coeffsd
> # Fit AR(5) model using arima()
> arima_fit <- arima(arimav, order=c(5, 0, 0), include.mean=FALSE)
> arima_fit$coef
> # Fit AR(5) model using auto.arima()
> library(forecast)  # Load forecast
> arima_fit <- forecast::auto.arima(arimav, max.p=5, max.q=0, max.d=
> # Fit AR(5) model into VTI returns
> retsp <- drop(zoo::coredata(na.omit(rutils::etfenv$returns$VTI)))
> predictor <- sapply(1:5, rutils::lagit, input=retsp)
> predinv <- MASS::ginv(predictor)
> coeff <- drop(predinv %*% retsp)
> # Calculate t-values of AR(5) coefficients
> residuals <- drop(retsp - drop(predictor %*% coeff))
> residvar <- sum(residuals^2)/(nrows-NROW(coeff))
> covar <- residvar*MASS::ginv(crossprod(predictor))
> coeffsd <- sqrt(diag(covar))
> coefftv <- drop(coeff)/coeffsd
```

draft: *AR(n)* Order Selection Using Information Criteria

Fitting a time series to an *AR(n)* model requires selecting the *order* parameter n.

The *order* parameter n of the *AR(n)* model is equal to the number of non-zero *partial autocorrelations* of the time series.

Order selection means determining the order n of the *AR(n)* model that best fits the time series.

Calibrating an *AR(n)* model is a two-step process: first determine the order n of the *AR(n)* model, and then calculate the coefficients.

The function `auto.arima()` from the package *forecast* performs order selection, and calibrates an *AR(n)* model to a univariate time series.

The function `arima()` from the base package *stats* fits an *AR(n)* model to a univariate time series.

The function `auto.arima()` from the package *forecast* automatically calibrates an *AR(n)* model to a univariate time series.

An *autoregressive* process *AR(n)* defined as:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$$

Can be solved as a *multivariate* linear regression model,

```
> # Calibrate ARIMA model using auto.arima()
> # library(forecast)  # Load forecast
> forecast::auto.arima(arimav, max.p=3, max.q=0, max.d=0)
> # Calibrate ARIMA model using arima()
> arima_fit <- arima(arimav, order=c(3,0,0), include.mean=FALSE)
> arima_fit$coef
> # Calibrate ARIMA model using auto.arima()
> # library(forecast)  # Load forecast
> forecast::auto.arima(arimav, max.p=3, max.q=0, max.d=0)
> # Calibrate ARIMA model using regression
> arimav <- as.numeric(arimav)
> # Define predictor matrix for arimav
> predictor <- sapply(1:3, rutils::lagit, input=arimav)
> # Generalized inverse of predictor matrix
> predinv <- MASS::ginv(predictor)
> # Regression coefficients with response equal to arimav
> coeff <- drop(predinv %*% arimav)
> all.equal(arima_fit$coef, coeff, check.attributes=FALSE)
```

The Yule-Walker Equations

To lighten the notation we can assume that the time series r_i has zero mean $\mathbb{E}[r_i] = 0$ and unit variance $\mathbb{E}[r_i^2] = 1$. ($\mathbb{E}$ is the expectation operator.)

Then the *autocorrelations* of r_i are equal to: $\rho_k = \mathbb{E}[r_i r_{i-k}]$.

If we multiply the *autoregressive* process *AR(n)*: $r_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$, by r_{i-k} and take the expectations, then we obtain the Yule-Walker equations:

$$\begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \vdots \\ \rho_n \end{pmatrix} = \begin{pmatrix} 1 & \rho_1 & \cdots & \rho_{n-1} \\ \rho_1 & 1 & \cdots & \rho_{n-2} \\ \rho_2 & \rho_1 & \cdots & \rho_{n-3} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n-1} & \rho_{n-2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \vdots \\ \varphi_n \end{pmatrix}$$

The Yule-Walker equations relate the *autocorrelation coefficients* ρ_i with the coefficients of the *AR(n)* process φ_i.

The Yule-Walker equations can be solved for the *AR(n)* coefficients φ_i using matrix inversion.

```
> # Compute autocorrelation coefficients
> acfv <- rutils::plot_acf(arimav, lag=10, plot=FALSE)
> acfv <- drop(acfv$acf)
> nrows <- NROW(acfv)
> acf1 <- c(1, acfv[-nrows])
> # Define Yule-Walker matrix
> ywmat <- sapply(1:nrows, function(lagg) {
+   if (lagg < nrows)
+     c(acf1[lagg:1], acf1[2:(nrows-lagg+1)])
+   else
+     acf1[lagg:1]
+ })  # end sapply
> # Generalized inverse of Yule-Walker matrix
> ywmatinv <- MASS::ginv(ywmat)
> # Solve Yule-Walker equations
> ywcoeff <- drop(ywmatinv %*% acfv)
> round(ywcoeff, 5)
> coeff
```

The Durbin-Levinson Algorithm for Partial Autocorrelations

The *partial autocorrelations* ϱ_i are the estimators of the coefficients φ_i of the *AR(n)* process.

The *partial autocorrelations* ϱ_i can be calculated by inverting the Yule-Walker equations.

The *partial autocorrelations* ϱ_i of an *AR(n)* process can be computed recursively from the autocorrelations ρ_i using the Durbin-Levinson algorithm:

$$\varrho_{i,i} = \frac{\rho_i - \sum_{k=1}^{i-1} \varrho_{i-1,k}\rho_{i-k}}{1 - \sum_{k=1}^{i-1} \varrho_{i-1,k}\rho_k}$$

$$\varrho_{i,k} = \varrho_{i-1,k} - \varrho_{i,i}\varrho_{i-1,i-k} \quad (1 \leq k \leq (i-1))$$

The diagonal elements $\varrho_{i,i}$ are updated first using the first equation. Then the off-diagonal elements $\varrho_{i,k}$ are updated using the second equation.

The *partial autocorrelations* are the diagonal elements: $\varrho_i = \varrho_{i,i}$

The Durbin-Levinson algorithm solves the Yule-Walker equations efficiently, without matrix inversion.

The function `pacf()` calculates and plots the *partial autocorrelations* using the Durbin-Levinson algorithm.

```
> # Calculate PACF from acf using Durbin-Levinson algorithm
> acfv <- rutils::plot_acf(arimav, lag=10, plotobj=FALSE)
> acfv <- drop(acfv$acf)
> nrows <- NROW(acfv)
> pacfv <- numeric(2)
> pacfv[1] <- acfv[1]
> pacfv[2] <- (acfv[2] - acfv[1]^2)/(1 - acfv[1]^2)
> # Calculate PACF recursively in a loop using Durbin-Levinson algor
> pacfvl <- matrix(numeric(nrows*nrows), nc=nrows)
> pacfvl[1, 1] <- acfv[1]
> for (it in 2:nrows) {
+   pacfvl[it, it] <- (acfv[it] - pacfvl[it-1, 1:(it-1)] %*% acfv[(i
+   for (it2 in 1:(it-1)) {
+     pacfvl[it, it2] <- pacfvl[it-1, it2] - pacfvl[it, it] %*% pacf
+   }  # end for
+ }  # end for
> pacfvl <- diag(pacfvl)
> # Compare with the PACF without loop
> all.equal(pacfv, pacfvl[1:2])
> # Calculate PACF using pacf()
> pacfv <- pacf(arimav, lag=10, plot=FALSE)
> pacfv <- drop(pacfv$acf)
> all.equal(pacfv, pacfvl)
```

Forecasting Autoregressive Processes

An *autoregressive* process *AR(n)*:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

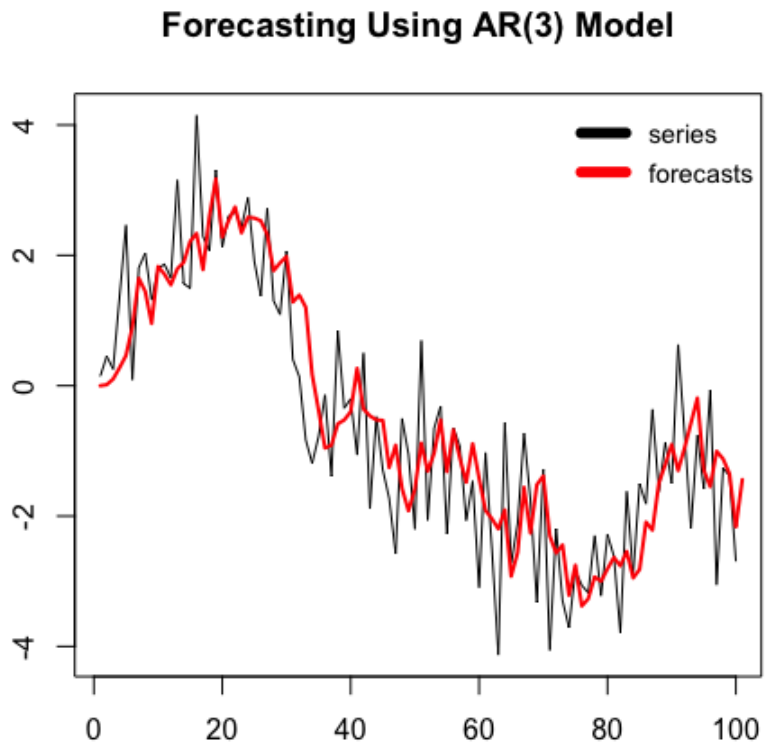
Can be simulated using the function `filter()` with the argument `method="recursive"`.

Recursive filtering can be performed even faster by directly calling the compiled `C++` function `stats:::C_rfilter()`.

The one step ahead *forecast* f_i is equal to the *convolution* of the time series r_i with the *AR(n)* coefficients:

$$f_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n}$$

```
> nrows <- 1e2
> coeff <- c(0.1, 0.39, 0.5); ncoeff <- NROW(coeff)
> set.seed(1121); innov <- rnorm(nrows, sd=0.01)
> # Simulate AR process using filter()
> arimav <- filter(x=innov, filter=coeff, method="recursive")
> arimav <- as.numeric(arimav)
> # Simulate AR process using C_rfilter()
> arimafast <- .Call(stats:::C_rfilter, innov, coeff,
+   double(nrows + ncoeff))
> all.equal(arimav, arimafast[-(1:ncoeff)],
+   check.attributes=FALSE)
```




```
> # Forecast AR(3) process using loop in R
> forecastv <- numeric(NROW(arimav)+1)
> forecastv[1] <- 0
> forecastv[2] <- coeff[1]*arimav[1]
> forecastv[3] <- coeff[1]*arimav[2] + coeff[2]*arimav[1]
> for (it in 4:NROW(forecastv)) {
+   forecastv[it] <- arimav[(it-1):(it-3)] %*% coeff
+ }  # end for
> # Plot with legend
> x11(width=6, height=4)
```

Fast Forecasting of Autoregressive Processes

The one step ahead *forecast* f_i is equal to the *convolution* of the time series r_i with the *AR(n)* coefficients:

$$f_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n}$$

The above *convolution* can be quickly calculated by using the function `filter()` with the argument `method="convolution"`.

The convolution can be calculated even faster by directly calling the compiled C++ function `stats:::C_cfilter()`.

The forecasts can also be calculated using the predictor matrix multiplied by the *AR(n)* coefficients.

```
> # Forecast using filter()
> filterfast <- filter(x=arimav, sides=1,
+   filter=coeff, method="convolution")
> filterfast <- as.numeric(filterfast)
> # Compare excluding warmup period
> all.equal(forecastv[-(1:ncoeff)], filterfast[-(1:(ncoeff-1))],
+     check.attributes=FALSE)
> # Filter using C_cfilter() compiled C++ function directly
> filterfast <- .Call(stats:::C_cfilter, arimav, filter=coeff,
+                sides=1, circular=FALSE)
> # Compare excluding warmup period
> all.equal(forecastv[-(1:ncoeff)], filterfast[-(1:(ncoeff-1))],
+     check.attributes=FALSE)
> # Filter using HighFreq::roll_conv() Rcpp function
> filterfast <- HighFreq::roll_conv(matrix(arimav), matrix(coeff))
> # Compare excluding warmup period
> all.equal(forecastv[-(1:ncoeff)], filterfast[-(1:(ncoeff-1))],
+     check.attributes=FALSE)
> # Define predictor matrix for forecasting
> predictor <- sapply(0:(ncoeff-1), function(lagg) {
+   rutils::lagit(arimav, lagg=lagg)
+ })  # end sapply
> # Forecast using predictor matrix
> filterfast <- c(0, drop(predictor %*% coeff))
> # Compare with loop in R
> all.equal(forecastv, filterfast, check.attributes=FALSE)
```

Forecasting Using predict.Arima()

The forecasts of the *AR(n)* process can also be calculated using the function `predict()`.

The function `predict()` is a *generic function* for forecasting based on a given model.

The *method* `predict.Arima()` is *dispatched* by `R` for calculating predictions from *ARIMA* models produced by the function `stats::arima()`.

The *method* `predict.Arima()` returns a prediction object which is a `list` containing the predicted value and its standard error.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series, using the *maximum likelihood* method (which may give slightly different coefficients than the linear regression model).

```
> # Fit ARIMA model using arima()
> arima_fit <- arima(arimav, order=c(3,0,0), include.mean=FALSE)
> arima_fit$coef
> coeff
> # One-step-ahead forecast using predict.Arima()
> predictv <- predict(arima_fit, n.ahead=1)
> # Or directly call predict.Arima()
> # predictv <- predict.Arima(arima_fit, n.ahead=1)
> # Inspect the prediction object
> class(predictv)
> names(predictv)
> class(predictv$pred)
> unlist(predictv)
> # One-step-ahead forecast using matrix algebra
> forecastv <- drop(arimav[nrows:(nrows-2)] %*% arima_fit$coef)
> # Compare one-step-ahead forecasts
> all.equal(predictv$pred[[1]], forecastv)
> # Get information about predict.Arima()
> ?stats:::predict.Arima
```

The Forecasting Residuals

The *forecasting residuals* ε_i are equal to the differences between the actual values r_i minus their *forecasts* f_i: $\varepsilon_i = r_i - f_i$.

Accurate forecasting of an *AR(n)* process requires knowing its coefficients.

If the coefficients of the *AR(n)* process are known exactly, then its *in-sample residuals* ε_i are equal to its *innovations* ξ_i: $\varepsilon_i = r_i - f_i = \xi_i$.

In practice, the *AR(n)* coefficients are not known, so they must be fitted to the empirical time series.

If the *AR(n)* coefficients are fitted to the empirical time series, then its *residuals* are *not* equal to its *innovations*.

```
> # Calculate the in-sample forecasting residuals
> residuals <- (arimav - forecastv[-NROW(forecastv)])
> # Compare residuals with innovations
> all.equal(innov, residuals, check.attributes=FALSE)
> plot(residuals, t="l", lwd=3, xlab="", ylab="",
+      main="ARIMA Forecast Errors")
```

draft: The Standard Errors of Forecasts from Autoregressive Processes

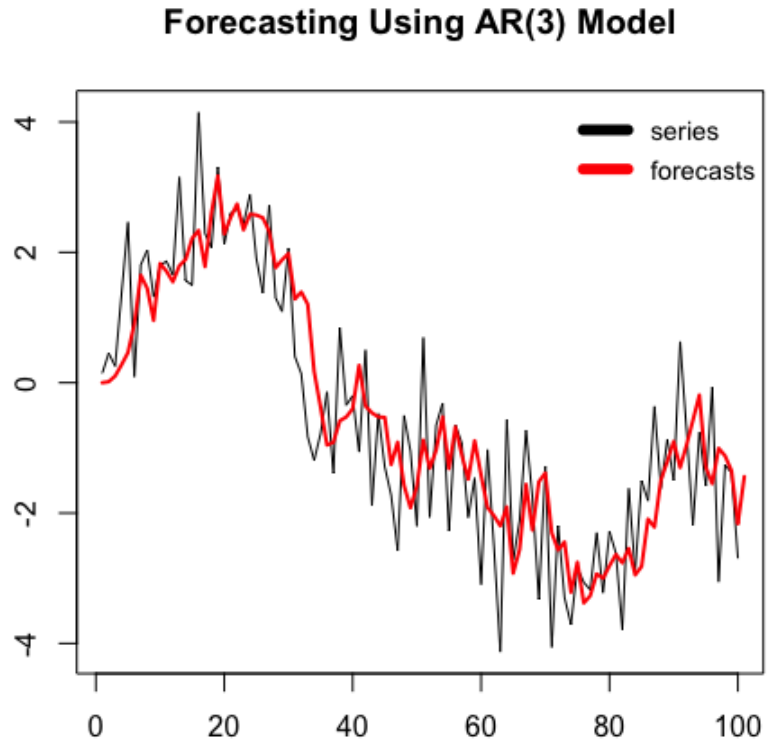
Trivial: The variance of the predicted value is equal to the predictor vector multiplied by the covariance matrix of the regression coefficients.

The one step ahead *forecast* f_i of the time series r_i using the process *AR(n)* is defined as:

$$f_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n}$$

The function `filter()` with the argument `method="convolution"` calculates the convolution of a vector with a filter

```
> # Simulate AR process using filter()
> nrows <- 1e2
> coeff <- c(0.1, 0.39, 0.5); ncoeff <- NROW(coeff)
> set.seed(1121)
> arimav <- filter(x=rnorm(nrows), filter=coeff, method="recursive")
> arimav <- as.numeric(arimav)
> # Forecast AR(3) process
> forecastv <- numeric(NROW(arimav))
> forecastv[2] <- coeff[1]*arimav[1]
> forecastv[3] <- coeff[1]*arimav[2] + coeff[2]*arimav[1]
> for (it in 4:NROW(forecastv)) {
+   forecastv[it] <- arimav[(it-1):(it-3)] %*% coeff
+ }  # end for
> # Forecast using filter()
> forecasts_filter <- filter(x=arimav, sides=1,
+   filter=coeff, method="convolution")
> class(forecasts_filter)
> all.equal(forecastv[-(1:4)],
+   forecasts_filter[-c(1:3, NROW(forecasts_filter))],
+   check.attributes=FALSE)
> # Compare residuals with innovations
> residuals <- (arimav-forecastv)
> tail(cbind(innov, residuals))
>
>
```



Accurate forecasting requires knowing the order n of the *AR(n)* process and its coefficients.

```
> # Plot with legend
> plot(arimav, main="Forecasting Using AR(3) Model",
+   xlab="", ylab="", type="l")
> lines(forecastv, col="orange", lwd=3)
> legend(x="topright", legend=c("series", "forecasts"),
+  col=c("black", "orange"), lty=1, lwd=6,
```

Fitting and Forecasting Autoregressive Models

In practice, the *AR(n)* coefficients are not known, so they must be fitted to the empirical time series first, before forecasting.

Forecasting using an autoregressive model is performed by first fitting an *AR(n)* model to past data, and calculating its coefficients.

The fitted coefficients are then applied to calculating the *out-of-sample* forecasts.

The model fitting procedure depends on two unknown *meta-parameters*: the order n of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define AR process parameters
> nrows <- 1e3
> coeff <- c(0.5, 0.0, 0.0); ncoeff <- NROW(coeff)
> set.seed(1121); innov <- rnorm(nrows, sd=0.01)
> # Simulate AR process using C_rfilter()
> arimav <- .Call(stats:::C_rfilter, innov, coeff,
+   double(nrows + ncoeff))[-(1:ncoeff)]
> # Define order of the AR(n) forecasting model
> ordern <- 5
> # Define predictor matrix for forecasting
> predictor <- sapply(1:ordern, rutils::lagit, input=arimav)
> colnames(predictor) <- paste0("pred", 1:NCOL(predictor))
> # Specify length of look-back interval
> look_back <- 100
> # Invert the predictor matrix
> rangev <- (nrows-look_back):(nrows-1)
> predinv <- MASS::ginv(predictor[rangev, ])
> # Calculate fitted coefficients
> coeff <- drop(predinv %*% arimav[rangev])
> # Calculate forecast
> drop(predictor[nrows, ] %*% coeff)
```

Rolling Forecasting of Autoregressive Models

The stock returns r_i are fitted into an *autoregressive* process *AR(n)* with a constant intercept term φ_0:

$$r_i = \varphi_0 + \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

The *AR(n)* coefficients φ are calibrated using linear regression:

$$\varphi = \mathbb{P}^{-1}\mathbf{r}$$

Where the *response* is equal to the stock returns $\mathbf{r}$, and the columns of the *predictor matrix* $\mathbb{P}$ are equal to the lags of $\mathbf{r}$

The *AR(n)* coefficients φ are recalibrated at every point in time on a rolling look-back interval of data.

The fitted coefficients φ are then used to calculate the one-day-ahead, out-of-sample return forecasts f_i:

$$f_i = \varphi_0 + \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n}$$

```
> # Calculate a vector of daily VTI log returns
> retsp <- na.omit(rutils::etfenv$returns$VTI)
> dates <- zoo::index(retsp)
> retsp <- as.numeric(retsp)
> nrows <- NROW(retsp)
> # Define response equal to returns
> response <- retsp
> # Define predictor as a rolling sum
> nagg <- 5
> predictor <- rutils::roll_sum(retsp, look_back=nagg)
> # Define predictor matrix for forecasting
> order_max <- 5
> predictor <- sapply(1+nagg*(0:order_max), rutils::lagit,
+                     input=predictor)
> predictor <- cbind(rep(1, nrows), predictor)
> # Perform rolling forecasting
> look_back <- 100
> forecastv <- sapply((look_back+1):nrows, function(endp) {
+   # Define rolling look-back range
+   startp <- max(1, endp-look_back)
+   # Or expanding look-back range
+   # startp <- 1
+   rangev <- startp:(endp-1)
+   # Invert the predictor matrix
+   predinv <- MASS::ginv(predictor[rangev, ])
+   # Calculate fitted coefficients
+   coeff <- drop(predinv %*% response[rangev])
+   # Calculate forecast
+   drop(predictor[endp, ] %*% coeff)
+ })  # end sapply
> # Add warmup period
> forecastv <- c(rep(0, look_back), forecastv)
```

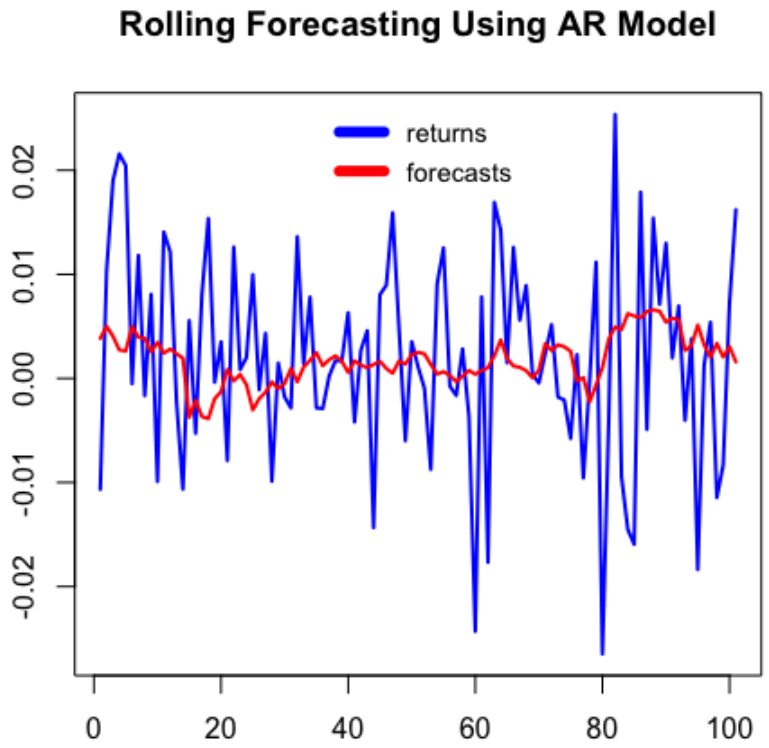
Mean Squared Error of the Autoregressive Forecasting Model

The accuracy of a forecasting model can be measured using the *mean squared error* and the *correlation*.

The mean squared error (*MSE*) of a forecasting model is the average of the squared forecasting residuals ε_i, equal to the differences between the actual values r_i minus the *forecasts* f_i: $\varepsilon_i = r_i - f_i$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(r_i - f_i)^2$$

```
> # Mean squared error
> mean((retsp - forecastv)^2)
> # Correlation
> cor(forecastv, retsp)
```




```
> # Plot forecasting series with legend
> x11(width=6, height=4)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> plot(retsp[(nrows-look_back):nrows], col="blue",
+      xlab="", ylab="", type="l", lwd=2,
+      main="Rolling Forecasting Using AR Model")
> lines(forecastv[(nrows-look_back):nrows], col="red", lwd=2)
> legend(x="top", legend=c("returns", "forecasts"),
+  col=c("blue", "red"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

Backtesting Function for the Forecasting Model

The *meta-parameters* of the *backtesting* function are the order *n* of the *AR(n)* model and the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

Backtesting is the simulation of a model on historical data to test its forecasting accuracy.

The autoregressive forecasting model can be *backtested* by calculating forecasts over either a *rolling* or an *expanding* look-back interval.

If the start date is fixed at the first row then the look-back interval is *expanding*.

The coefficients of the *AR(n)* process are fitted to past data, and then applied to calculating out-of-sample forecasts.

The *backtesting* procedure allows determining the optimal *meta-parameters* of the forecasting model: the order *n* of the *AR(n)* model and the length of look-back interval (`look_back`).

```
> # Define backtesting function
> sim_forecasts <- function(response, nagg=5, ordern=5,
+                           look_back=100, rollp=TRUE) {
+   nrows <- NROW(response)
+   # Define predictor as a rolling sum
+   predictor <- rutils::roll_sum(response, look_back=nagg)
+   # Define predictor matrix for forecasting
+   predictor <- sapply(1+nagg*(0:ordern), rutils::lagit,
+                       input=predictor)
+   predictor <- cbind(rep(1, nrows), predictor)
+   # Perform rolling forecasting
+   forecastv <- sapply((look_back+1):nrows, function(endp) {
+     # Define rolling look-back range
+     if (rollp)
+ startp <- max(1, endp-look_back)
+     else
+     # Or expanding look-back range
+     startp <- 1
+     rangev <- startp:(endp-1)
+     # Invert the predictor matrix
+     designinv <- MASS::ginv(predictor[rangev, ])
+     # Calculate fitted coefficients
+     coeff <- drop(designinv %*% response[rangev])
+     # Calculate forecast
+     drop(predictor[endp, ] %*% coeff)
+   })  # end sapply
+   # Add warmup period
+   forecastv <- c(rep(0, look_back), forecastv)
+   # Aggregate the forecasts
+   rutils::roll_sum(forecastv, look_back=nagg)
+ }  # end sim_forecasts
> # Simulate the rolling autoregressive forecasts
> forecastv <- sim_forecasts(response=retvti, ordern=5, look_back=1(
> c(mse=mean((retvti - forecastv)^2), cor=cor(retvti, forecastv))
```

Forecasting Dependence On the Look-back Interval

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order *n* of the *AR(n)* model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

```
> look_backs <- seq(20, 200, 20)
> forecastv <- sapply(look_backs, sim_forecasts, response=retsp,
+                 nagg=nagg, ordern=ordern)
> colnames(forecastv) <- look_backs
> msev <- apply(forecasts, 2, function(x) mean((retsp - x)^2))
> # Plot forecasting series with legend
> plot(x=look_backs, y=msev,
+   xlab="look-back", ylab="MSE", type="l", lwd=2,
+   main="MSE of AR(5) Forecasting Model")
```