Time Series Univariate

FRE6871 & FRE7241, Fall 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

January 13, 2025

NYU | TANDON SCHOOL OF ENGINEERING

Defining Look-back Time Intervals

A time *period* is the time between two neighboring points in time.

A time *interval* is the time spanned by one or more time *periods*.

A *look-back interval* is a time *interval* for performing aggregations over the past, starting from a *start point* and ending at an *end point*.

The *start points* are the *end points* lagged by the *look-back interval*.

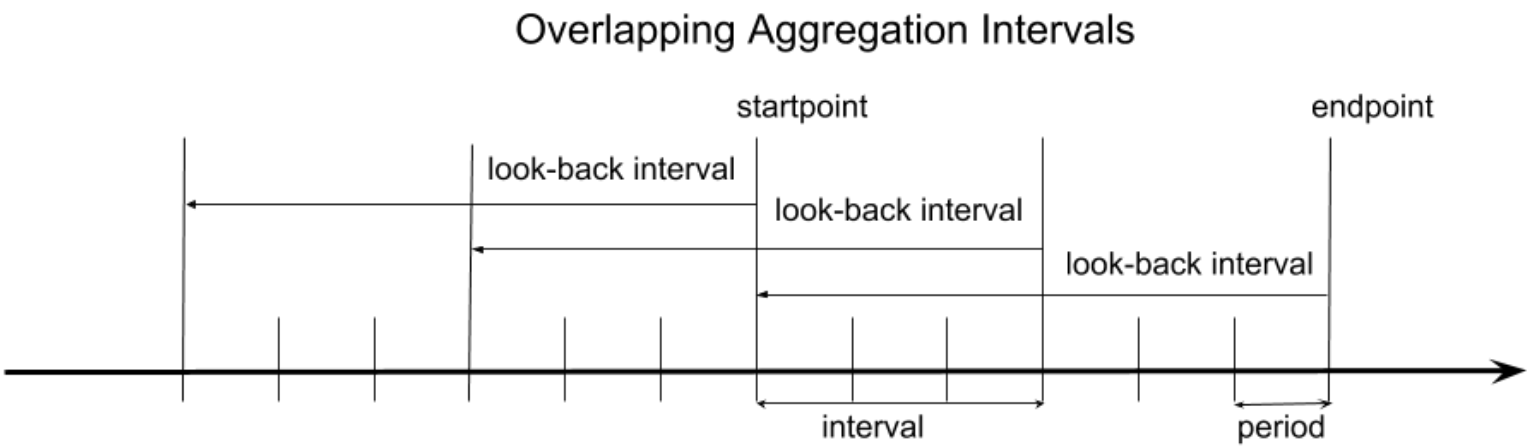
The look-back *intervals* may or may not *overlap* with their neighboring intervals.

A *trailing aggregation* is performed over a vector of *end points* in time.

An example of a trailing aggregation are moving average prices.

An *interval aggregation* is specified by *end points* separated by many time *periods*.

Examples of interval aggregations are monthly asset returns, or trailing 12-month asset returns calculated every month.



Defining *Trailing* Look-back Time Intervals

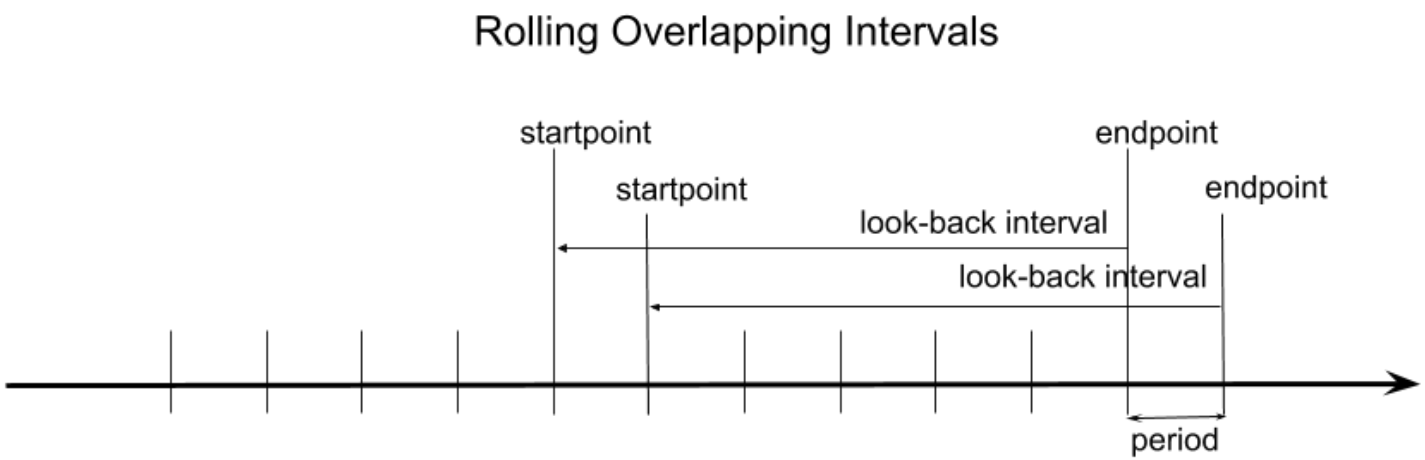
A *trailing aggregation* is performed over a vector of *end points* in time.

The first *end point* is equal to zero 0.

The *start points* are the *end points* lagged by the *look-back interval*.

An example of a trailing aggregation are moving average prices.

```
> ohlc <- rutils::etfenv$VTI
> # Number of data points
> nrows <- NROW(ohlc["2018-06/"])
> # Define endd at each point in time
> endd <- 0:nrows
> # Number of data points in lookb interval
> lookb <- 22
> # startp are endd lagged by lookb
> startp <- c(rep_len(0, lookb), endd[1:(NROW(endd)-lookb)])
> head(startp, 33)
```



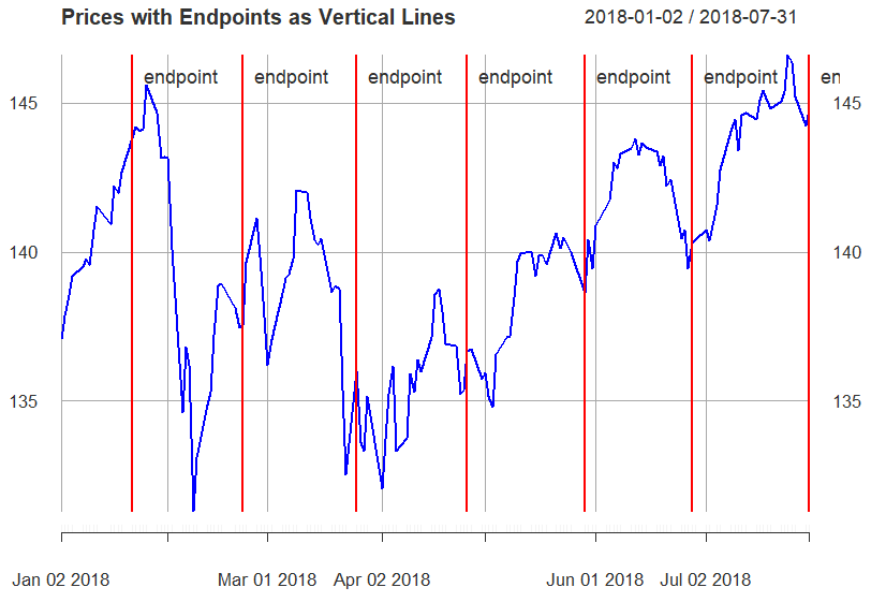
Defining Equally Spaced *end points* of a Time Series

The neighboring *end points* may be separated by a fixed number of periods, equal to `npoints`.

If the total number of data points is not an integer multiple of `npoints`, then a stub interval must be added either at the beginning or at the end of the *end points*.

The function `xts::endpoints()` extracts the indices of the last observations in each calendar period of an *xts* series.

```
> # Number of data points
> closep <- quantmod::Cl(ohlc["2018/"])
> nrows <- NROW(closep)
> # Number of periods between endpoints
> npoints <- 21
> # Number of npoints that fit over nrows
> nagg <- nrows %/% npoints
> # If(nrows==npoints*nagg then whole number
> endd <- (0:nagg)*npoints
> # Stub interval at beginning
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
> # Else stub interval at end
> endd <- c((0:nagg)*npoints, nrows)
> # Or use xts::endpoints()
> endd <- xts::endpoints(closep, on="weeks")
```




```
> # Plot data and endpoints as vertical lines
> plot.xts(closep, col="blue", lwd=2, xlab="", ylab="",
+    main="Prices with Endpoints as Vertical Lines")
> addEventLines(xts(rep("endpoint", NROW(endd)-1), zoo::index(closep
+         col="red", lwd=2, pos=4)
> # Or
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- "blue"
> quantmod::chart_Series(closep, theme=plot_theme,
+   name="prices with endpoints as vertical lines")
> abline(v=endd, col="red", lwd=2)
```

Defining *Overlapping* Look-back Time Intervals

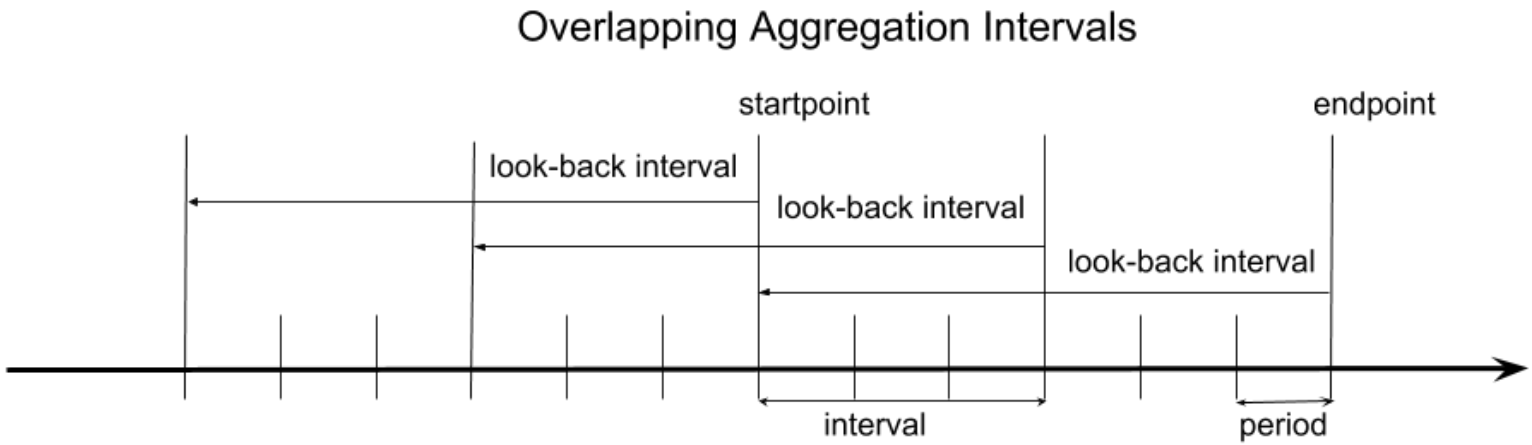
Overlapping time intervals can be defined if the *start points* are equal to the *end points* lagged by the *look-back interval*.

An example of an overlapping interval aggregation are trailing 12-month asset returns calculated every month.

```
> # Number of data points
> nrows <- NROW(rutils::etfenv$VTI["2019/"])
> # Number of npoints that fit over nrows
> npoints <- 21
> nagg <- nrows %/% npoints
> # Stub interval at beginning
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
```

The length of the *look-back interval* can be defined either as the number of data points, or as the number of *end points* to look back over.

```
> # lookb defined as number of data points
> lookb <- 252
> # startp are endd lagged by lookb
> startp <- (endd - lookb + 1)
> startp <- ifelse(startp < 0, 0, startp)
> # lookb defined as number of endd
> lookb <- 12
> startp <- c(rep_len(0, lookb), endd[1:(NROW(endd)- lookb)])
> # Bind startp with endd
> cbind(startp, endd)
```



Defining *Non-overlapping* Look-back Time Intervals

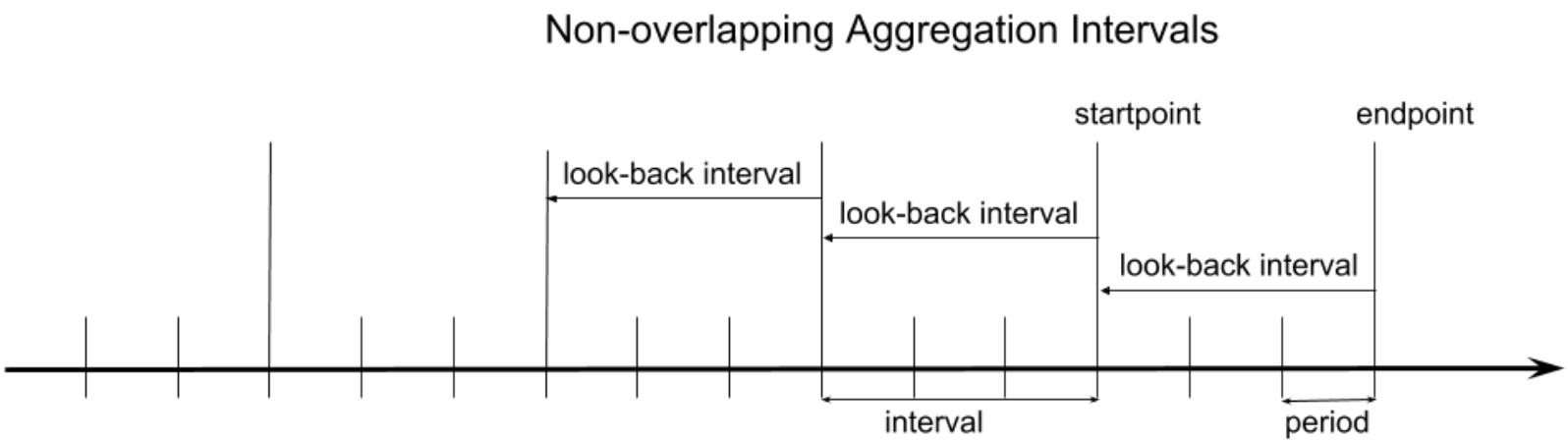
Non-overlapping time intervals can be defined if *start points* are equal to the previous *end points*.

In that case the look-back *intervals* are non-overlapping and *contiguous* (each *start point* is the *end point* of the previous interval).

If the *start points* are defined as the previous *end points* plus 1, then the *intervals* are *exclusive*.

Exclusive intervals are used for calculating *out-of-sample* aggregations over future intervals.

```
> # Number of data points
> nrows <- NROW(rutils::etfenv$VTI["2019/"])
> # Number of data points per interval
> npoints <- 21
> # Number of npointss that fit over nrows
> nagg <- nrows %/% npoints
> # Define endd with beginning stub
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
> # Define contiguous startp
> startp <- c(0, endd[1:(NROW(endd)-1)])
> # Define exclusive startp
> startp <- c(0, endd[1:(NROW(endd)-1)]+1)
```



Performing Trailing Aggregations Using sapply()

Aggregations performed over time series can be extremely slow if done improperly, therefore it's very important to find the fastest methods of performing aggregations.

The `sapply()` functional allows performing aggregations over the look-back *intervals*.

The `sapply()` functional by default returns a vector or matrix, not an *xts* series.

The vector or matrix returned by `sapply()` therefore needs to be coerced into an *xts* series.

The variable `lookb` is the size of the look-back interval, equal to the number of data points used for applying the aggregation function (including the current point).

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> endd <- 0:NROW(closep)  # End points at each point
> npts <- NROW(endd)
> lookb <- 22  # Number of data points per look-back interval
> # startp are multi-period lag of endd
> startp <- c(rep_len(0, lookb), endd[1:(npts - lookb)])
> # Define list of look-back intervals for aggregations over past
> lookbv <- lapply(2:npts, function(it) {
+     startp[it]:endd[it]
+ })  # end lapply
> # Define aggregation function
> aggfun <- function(xtsv) c(max=max(xtsv), min=min(xtsv))
> # Perform aggregations over lookbv list
> aggs <- sapply(lookbv,
+     function(lookb) aggfun(closep[lookb])
+ )  # end sapply
> # Coerce aggs into matrix and transpose it
> if (is.vector(aggs))
+   aggs <- t(aggs)
> aggs <- t(aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
```

Performing Trailing Aggregations Using lapply()

The lapply() functional allows performing aggregations over the look-back *intervals*.

The lapply() functional by default returns a list, not an *xts* series.

If lapply() returns a list of *xts* series, then this list can be collapsed into a single *xts* series using the function do_call_rbind() from package *rutils*.

The function chart_Series() from package *quantmod* can produce a variety of time series plots.

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

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.

```
> # Perform aggregations over lookbv list
> aggs <- lapply(lookbv,
+     function(lookb) aggfun(closep[lookb])
+ )  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> # Convert into xts
> aggs <- xts::xts(aggs, order.by=zoo::index(closep))
> aggs <- cbind(aggs, closep)
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "green")
> x11(width=6, height=5)
> quantmod::chart_Series(aggs, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6, y.intersp=0.4,
+   col=plot_theme$col$line.col, bty="n")
```

Defining Functionals for Trailing Aggregations

The functional `roll_agg()` performs trailing aggregations of its function argument `FUN`, over an *xts* series (`x_ts`), and a look-back interval (`lookb`).

The argument `FUN` is an aggregation function over a subset of `x_ts` series.

The dots `"..."` argument is passed into `FUN` as additional arguments.

The argument `lookb` is equal to the number of periods of `x_ts` series which are passed to the aggregation function `FUN`.

The functional `roll_agg()` calls `lapply()`, which loops over the length of series `x_ts`.

Note that two different intervals may be used with `roll_agg()`.

The first interval is the argument `lookb`.

A second interval may be one of the variables bound to the dots `"..."` argument, and passed to the aggregation function `FUN` (for example, an *EMA* window).

```
> # Define functional for trailing aggregations
> roll_agg <- function(xtsv, lookb, FUN, ...) {
+ # Define end points at every period
+   endd <- 0:NROW(xtsv)
+   npts <- NROW(endd)
+ # Define starting points as lag of endd
+   startp <- c(rep_len(0, lookb), endd[1:(npts- lookb)])
+ # Perform aggregations over lookbv list
+   aggs <- lapply(2:npts, function(it)
+     FUN(xtsv[startp[it]:endd[it]], ...)
+   )  # end lapply
+ # rbind list into single xts or matrix
+   aggs <- rutils::do_call(rbind, aggs)
+ # Coerce aggs into xts series
+   if (!is.xts(aggs))
+     aggs <- xts(aggs, order.by=zoo::index(xtsv))
+   aggs
+ }  # end roll_agg
> # Define aggregation function
> aggfun <- function(xtsv)
+   c(max=max(xtsv), min=min(xtsv))
> # Perform aggregations over trailing interval
> aggs <- roll_agg(closep, lookb=lookb, FUN=aggfun)
> class(aggs)
> dim(aggs)
```

Benchmarking Speed of Trailing Aggregations

The speed of trailing aggregations using `apply()` loops can be greatly increased by simplifying the aggregation function

For example, an aggregation function that returns a vector is over 13 times faster than a function that returns an *xts* object.

```
> # Define aggregation function that returns a vector
> agg_vector <- function(xtsv)
+   c(max=max(xtsv), min=min(xtsv))
> # Define aggregation function that returns an xts
> agg_xts <- function(xtsv)
+   xts(t(c(max=max(xtsv), min=min(xtsv))), order.by=end(xtsv))
> # Benchmark the speed of aggregation functions
> library(microbenchmark)
> summary(microbenchmark(
+   agg_vector=roll_agg(closep, lookb=lookb, FUN=agg_vector),
+   agg_xts=roll_agg(closep, lookb=lookb, FUN=agg_xts),
+   times=10))[, c(1, 4, 5)]
```

Benchmarking Functionals for Trailing Aggregations

Several packages contain functionals designed for performing trailing aggregations:

- `rollapply.zoo()` from package *zoo*,
- `rollapply.xts()` from package *xts*,
- `apply.rolling()` from package *PerformanceAnalytics*,

These functionals don't require specifying the *end points*, and instead calculate the *end points* from the trailing interval width.

These functionals can only apply functions that return a single value, not a vector.

These functionals return an *xts* series with leading `NA` values at points before the trailing interval can fit over the data.

The argument `align="right"` of `rollapply()` determines that aggregations are taken from the past.

The functional `rollapply.xts` is the fastest, about as fast as performing an `lapply()` loop directly.

```
> # Define aggregation function that returns a single value
> aggfun <- function(xtsv)  max(xtsv)
> # Perform aggregations over a trailing interval
> aggs <- xts:::rollapply.xts(closep, width=lookb,
+               FUN=aggfun, align="right")
> # Perform aggregations over a trailing interval
> library(PerformanceAnalytics)  # Load package PerformanceAnalytics
> aggs <- apply.rolling(closep, width=lookb, FUN=aggfun)
> # Benchmark the speed of the functionals
> library(microbenchmark)
> summary(microbenchmark(
+   roll_agg=roll_agg(closep, lookb=lookb, FUN=max),
+   roll_xts=xts:::rollapply.xts(closep, width=lookb, FUN=max, align="righ
+   apply_rolling=apply.rolling(closep, width=lookb, FUN=max),
+   times=10))[, c(1, 4, 5)]
```

Trailing Aggregations Using *Vectorized* Functions

The generic functions `cumsum()`, `cummax()`, and `cummin()` return the cumulative sums, minima, and maxima of *vectors* and *time series* objects.

The methods for these functions are implemented as *vectorized compiled* functions, and are therefore much faster than `apply()` loops.

The `cumsum()` function can be used to efficiently calculate the trailing sum of an an *xts* series.

Using the function `cumsum()` is over 25 times faster than using `apply()` loops.

But trailing volatilities and higher moments can't be easily calculated using `cumsum()`.

```
> # Trailing sum using cumsum()
> roll_sum <- function(xtsv, lookb) {
+   cumsumv <- cumsum(na.omit(xtsv))
+   datav <- (cumsumv - rutils::lagit(x=cumsumv, lagg=lookb))
+   datav[1:lookb, ] <- cumsumv[1:lookb, ]
+   colnames(datav) <- paste0(colnames(xtsv), "_stdev")
+   datav
+ }  # end roll_sum
> aggs <- roll_sum(closep, lookb=lookb)
> # Perform trailing aggregations using lapply loop
> aggs <- lapply(2:npts, function(it)
+     sum(closep[startp[it]:endd[it]])
+ )  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> head(aggs)
> tail(aggs)
> # Benchmark the speed of both methods
> library(microbenchmark)
> summary(microbenchmark(
+   roll_sum=roll_sum(closep, lookb=lookb),
+   s_apply=sapply(lookbv,
+     function(lookb) sum(closep[lookb])),
+   times=10))[, c(1, 4, 5)]
```

Filtering Time Series Using Function filter()

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

The function filter() with the argument method="convolution" calculates the *convolution* of the vector r_t with the filter φ_i:

$$f_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_p r_{t-p}$$

Where f_t is the filtered output vector, and φ_i are the filter coefficients.

filter() with method="recursive" calculates a *recursive* filter over the vector of random *innovations* ξ_t as follows:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_p r_{t-p} + \xi_t$$

Where r_t is the filtered output vector, and φ_i are the filter coefficients.

The *recursive* filter describes an *AR(p)* process, which is a special case of an *ARIMA* process.

filter() is very fast because it calculates the filter by calling compiled C++ functions.

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Calculate EMA prices using filter()
> lookb <- 21
> weightv <- exp(-0.1*1:lookb)
> weightv <- weightv/sum(weightv)
> pricef <- stats::filter(closep, filter=weightv,
+                         method="convolution", sides=1)
> pricef <- as.numeric(pricef)
> # filter() returns time series of class "ts"
> class(pricef)
> # Filter using compiled C++ function directly
> getAnywhere(C_cfilter)
> str(stats:::C_cfilter)
> priceff <- .Call(stats:::C_cfilter, closep,
+                  filter=weightv, sides=1, circular=FALSE)
> all.equal(as.numeric(pricef), priceff, check.attributes=FALSE)
> # Calculate EMA prices using HighFreq::roll_conv()
> pricecpp <- HighFreq::roll_conv(closep, weightv=weightv)
> all.equal(pricef[-(1:lookb)],
+     as.numeric(pricecpp)[-(1:lookb)],
+     check.attributes=FALSE)
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   filter=filter(closep, filter=weightv, method="convolution", side
+   priceff=.Call(stats:::C_cfilter, closep, filter=weightv, sides=1
+   cumsumv=cumsum(closep),
+   rcpp=HighFreq::roll_conv(closep, weightv=weightv)
+   ), times=10)[, c(1, 4, 5)]
```

Performing Trailing Aggregations Using Package *TTR*

The package *TTR* contains functions for calculating trailing aggregations over *vectors* and *time series* objects:

- `runSum()` for trailing sums,
- `runMin()` and `runMax()` for trailing minima and maxima,
- `runSD()` for trailing standard deviations,
- `runMedian()` and `runMAD()` for trailing medians and Median Absolute Deviations (*MAD*),
- `runCor()` for trailing correlations,

The trailing *TTR* functions are much faster than performing `apply()` loops, because they are *compiled* functions (compiled from `C++` or `Fortran` code).

But the trailing *TTR* functions are a little slower than using *vectorized compiled* functions such as `cumsum()`.

```
> # Calculate the trailing maximum and minimum over a vector of data
> roll_maxminr <- function(vecv, lookb) {
+   nrows <- NROW(vecv)
+   max_min <- matrix(numeric(2:nrows), nc=2)
+   # Loop over periods
+   for (it in 1:nrows) {
+     sub_vec <- vecv[max(1, it-lookb+1):it]
+     max_min[it, 1] <- max(sub_vec)
+     max_min[it, 2] <- min(sub_vec)
+   }  # end for
+   return(max_min)
+ }  # end roll_maxminr
> max_minr <- roll_maxminr(closep, lookb)
> max_minr <- xts::xts(max_minr, zoo::index(closep))
> library(TTR)  # Load package TTR
> max_min <- cbind(TTR::runMax(x=closep, n=lookb),
+            TTR::runMin(x=closep, n=lookb))
> all.equal(max_min[-(1:lookb), ], max_minr[-(1:lookb), ], check.att
> # Benchmark the speed of TTR::runMax
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=roll_maxminr(closep, lookb),
+   ttr=TTR::runMax(closep, n=lookb),
+   times=10))[, c(1, 4, 5)]
> # Benchmark the speed of TTR::runSum
> summary(microbenchmark(
+   vector_r=cumsum(coredata(closep)),
+   rutils=rutils::roll_sum(closep, lookb=lookb),
+   ttr=TTR::runSum(closep, n=lookb),
+   times=10))[, c(1, 4, 5)]
```

Trailing *Weighted* Aggregations Using Package *roll*

The package *roll* contains functions for calculating *weighted* trailing aggregations over *vectors* and *time series* objects:

- `roll_sum()`, `roll_max()`, `roll_mean()`, and `roll_median()` for *weighted* trailing sums, maximums, means, and medians,
- `roll_var()` for *weighted* trailing variance,
- `roll_scale()` for trailing scaling and centering of time series,
- `roll_lm()` for trailing regression,
- `roll_pcr()` for trailing principal component regressions of time series,

The *roll* functions are about `1,000` times faster than `apply()` loops!

The *roll* functions are extremely fast because they perform calculations in *parallel* in compiled `C++` code, using packages *Rcpp* and *RcppArmadillo*.

The *roll* functions accept *xts* time series, and they return *xts*.

```
> # Calculate trailing VTI variance using package roll
> library(roll)  # Load roll
> retp <- na.omit(rutils::etfenv$returns$VTI)
> lookb <- 22
> # Calculate trailing sum using roll::roll_sum
> sumroll <- roll::roll_sum(retp, width=lookb, min_obs=1)
> # Calculate trailing sum using rutils
> sumrutils <- rutils::roll_sum(retp, lookb=lookb)
> all.equal(sumroll[-(1:lookb), ],
+     sumrutils[-(1:lookb), ], check.attributes=FALSE)
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   cumsumv=cumsum(retp),
+   roll=roll::roll_sum(retp, width=lookb),
+   RcppRoll=RcppRoll::roll_sum(retp, n=lookb),
+   rutils=rutils::roll_sum(retp, lookb=lookb),
+   times=10))[, c(1, 4, 5)]
```

Trailing *Weighted* Aggregations Using Package *RcppRoll*

The package *RcppRoll* contains functions for calculating *weighted* trailing aggregations over *vectors* and *time series* objects:

- `roll_sum()` for *weighted* trailing sums,
- `roll_min()` and `roll_max()` for *weighted* trailing minima and maxima,
- `roll_sd()` for *weighted* trailing standard deviations,
- `roll_median()` for *weighted* trailing medians,

The *RcppRoll* functions accept *xts* objects, but they return matrices, not *xts* objects.

The trailing *RcppRoll* functions are much faster than performing `apply()` loops, because they are *compiled* functions (compiled from `C++` code).

But the trailing *RcppRoll* functions are a little slower than using *vectorized compiled* functions such as `cumsum()`.

```
> library(RcppRoll)  # Load package RcppRoll
> # Calculate trailing sum using RcppRoll
> sumroll <- RcppRoll::roll_sum(retp, align="right", n=lookb)
> # Calculate trailing sum using rutils
> sumrutils <- rutils::roll_sum(retp, lookb=lookb)
> all.equal(sumroll, coredata(sumrutils[-(1:(lookb-1))]),
+     check.attributes=FALSE)
> # Benchmark speed of trailing calculations
> library(microbenchmark)
> summary(microbenchmark(
+   cumsumv=cumsum(retp),
+   RcppRoll=RcppRoll::roll_sum(retp, n=lookb),
+   rutils=rutils::roll_sum(retp, lookb=lookb),
+   times=10))[, c(1, 4, 5)]
> # Calculate EMA prices using RcppRoll
> closep <- quantmod::Cl(rutils::etfenv$VTI)
> weightv <- exp(0.1*1:lookb)
> pricema <- RcppRoll::roll_mean(closep,
+ align="right", n=lookb, weights=weightv)
> pricema <- cbind(closep,
+   rbind(coredata(closep[1:(lookb-1), ]), pricema))
> colnames(pricema) <- c("VTI", "VTI EMA")
> # Plot an interactive dygraph plot
> dygraphs::dygraph(pricema)
> # Or static plot of EMA prices with custom line colors
> x11(width=6, height=5)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red")
> quantmod::chart_Series(pricema, theme=plot_theme, name="EMA prices
> legend("top", legend=colnames(pricema),
+  bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Performing Trailing Aggregations Using Package *caTools*

The package *caTools* contains functions for calculating trailing interval aggregations over a `vector` of data:

- `runmin()` and `runmax()` for trailing minima and maxima,
- `runsd()` for trailing standard deviations,
- `runmad()` for trailing Median Absolute Deviations (*MAD*),
- `runquantile()` for trailing quantiles,

Time series need to be coerced to *vectors* before they are passed to *caTools* functions.

The trailing *caTools* functions are very fast because they are *compiled* functions (compiled from `C++` code).

The argument `"endrule"` determines how the end values of the data are treated.

The argument `"align"` determines whether the interval is centered (default), left-aligned or right-aligned, with `align="center"` the fastest option.

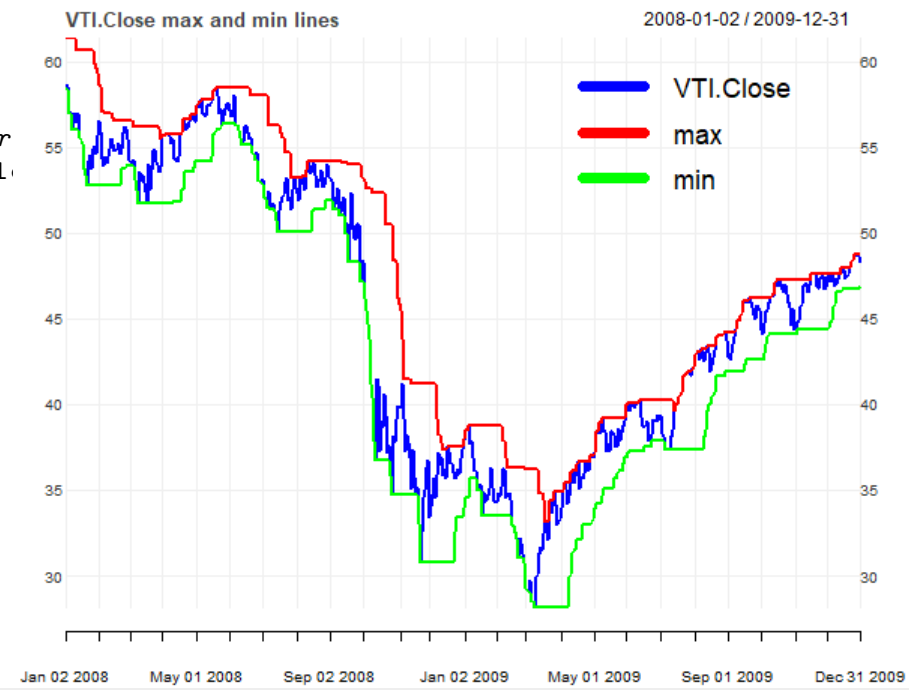
```
> library(caTools)  # Load package "caTools"
> # Get documentation for package "caTools"
> packageDescription("caTools")  # Get short description
> help(package="caTools")  # Load help page
> data(package="caTools")  # List all datasets in "caTools"
> ls("package:caTools")  # List all objects in "caTools"
> detach("package:caTools")  # Remove caTools from search path
> # Median filter
> lookb <- 2
> closep <- quantmod::Cl(HighFreq::SPY["2012-02-01/2012-04-01"])
> med_ian <- runmed(x=closep, k=lookb)
> # Vector of trailing volatilities
> sigmav <- runsd(x=closep, k=lookb,
+           endrule="constant", align="center")
> # Vector of trailing quantiles
> quantvs <- runquantile(x=closep, k=lookb,
+   probs=0.9, endrule="constant", align="center")
```

Performing Trailing Aggregations Using *RcppArmadillo*

RcppArmadillo functions for calculating trailing aggregations are often the fastest.

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

// export the function roll_maxmin() to R
// [[Rcpp::export]]
arma::mat roll_maxmin(const arma::vec& vecv,
                      const arma::uword& lookb) {
  arma::uword n_rows = vecv.size();
  arma::mat max_min(nrows, 2);
  arma::vec sub_vec;
  // startup period
  max_min(0, 0) = vecv[0];
  max_min(0, 1) = vecv[0];
  for (uword it = 1; it < lookb; it++) {
    sub_vec = vecv.subvec(0, it);
    max_min(it, 0) = sub_vec.max();
    max_min(it, 1) = sub_vec.min();
  }  // end for
  // remaining periods
  for (uword it = lookb; it <.n_rows; it++) {
    sub_vec = vecv.subvec(it- lookb + 1, it);
    max_min(it, 0) = sub_vec.max();
    max_min(it, 1) = sub_vec.min();
  }  // end for
  return max_min;
}  // end roll_maxmin
```




```
> # Compile Rcpp functions
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/R/Rcpp/roll_maxmin.cpp"
> max_minarma <- roll_maxmin(closep, lookb)
> max_minarma <- xts::xts(max_minr, zoo::index(closep))
> max_min <- cbind(TTR::runMax(x=closep, n=lookb),
+            TTR::runMin(x=closep, n=lookb))
> all.equal(max_min[-(1:lookb), ], max_minarma[-(1:lookb), ], check
> # Benchmark the speed of TTR::runMax
> library(microbenchmark)
> summary(microbenchmark(
+   arma=roll_maxmin(closep, lookb),
+   ttr=TTR::runMax(closep, n=lookb),
+   times=10))[, c(1, 4, 5)]
> # Dygraphs plot with max_min lines
> datav <- cbind(closep, max_minarma)
> colnames(datav)[2:3] <- c("max", "min")
```

Determining Calendar *end points* of *xts* Time Series

The function `xts::endpoints()` extracts the indices of the last observations in each calendar period of an *xts* series.

For example:
`endpoints(x, on="hours")`
extracts the indices of the last observations in each hour.

The *end points* calculated by `endpoints()` aren't always equally spaced, and aren't the same as those calculated from fixed intervals.

For example, the last observations in each day aren't equally spaced due to weekends and holidays.

```
> # Indices of last observations in each hour
> endd <- xts::endpoints(closep, on="hours")
> head(endd)
> # extract the last observations in each hour
> head(closep[endd, ])
```

Performing Non-overlapping Aggregations Using sapply()

The apply() functionals allow for applying a function over intervals of an *xts* series defined by a vector of *end points*.

The sapply() functional by default returns a vector or matrix, not an *xts* series.

The vector or matrix returned by sapply() therefore needs to be coerced into an *xts* series.

The function chart_Series() from package *quantmod* can produce a variety of time series plots.

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

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.

```
> # Extract time series of VTI log prices
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> # Number of data points
> nrows <- NROW(closep)
> # Number of data points per interval
> lookb <- 22
> # Number of lookbv that fit over nrows
> nagg <- nrows %/% lookb
> # Define endd with beginning stub
> endd <- c(0, nrows-lookb*nagg + (0:nagg)*lookb)
> # Define contiguous startp
> startp <- c(0, endd[1:(NROW(endd)-1)])
> # Define list of look-back intervals for aggregations over past
> lookbv <- lapply(2:NROW(endd), function(it) {
+     startp[it]:endd[it]
+ })  # end lapply
> lookbv[[1]]
> lookbv[[2]]
> # Perform sapply() loop over lookbv list
> aggs <- sapply(lookbv, function(lookb) {
+   xtsv <- closep[lookb]
+   c(max=max(xtsv), min=min(xtsv))
+ })  # end sapply
> # Coerce aggs into matrix and transpose it
> if (is.vector(aggs))
+   aggs <- t(aggs)
> aggs <- t(aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
> head(aggs)
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Performing Non-overlapping Aggregations Using lapply()

The apply() functionals allow for applying a function over intervals of an *xts* series defined by a vector of *end points*.

The lapply() functional by default returns a list, not an *xts* series.

If lapply() returns a list of *xts* series, then this list can be collapsed into a single *xts* series using the function do_call_rbind() from package *rutils*.

```
> # Perform lapply() loop over lookbv list
> aggs <- lapply(lookbv, function(lookb) {
+   xtsv <- closep[lookb]
+   c(max=max(xtsv), min=min(xtsv))
+ })  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
> head(aggs)
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme, name="price aggrega
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Performing Interval Aggregations Using period.apply()

The functional period.apply() from package *xts* performs *aggregations* over non-overlapping intervals of an *xts* series defined by a vector of *end points*.

Internally period.apply() performs an sapply() loop, and is therefore about as fast as an sapply() loop.

The package *xts* also has several specialized functionals for aggregating data over *end points*:

- period.sum() calculate the sum for each period,
- period.max() calculate the maximum for each period,
- period.min() calculate the minimum for each period,
- period.prod() calculate the product for each period,

```
> # Define functional for trailing aggregations over endd
> roll_agg <- function(xtsv, endd, FUN, ...) {
+   nrows <- NROW(endd)
+ # startp are single-period lag of endd
+   startp <- c(1, endd[1:(nrows-1)])
+ # Perform aggregations over lookbv list
+   aggs <- lapply(lookbv,
+     function(lookb) FUN(xtsv[lookb], ...))  # end lapply
+ # rbind list into single xts or matrix
+   aggs <- rutils::do_call(rbind, aggs)
+   if (!is.xts(aggs))
+     aggs <-  # Coerce aggs into xts series
+     xts(aggs, order.by=zoo::index(xtsv[endd]))
+   aggs
+ }  # end roll_agg
> # Apply sum() over endd
> aggs <- roll_agg(closep, endd=endd, FUN=sum)
> aggs <- period.apply(closep, INDEX=endd, FUN=sum)
> # Benchmark the speed of aggregation functions
> summary(microbenchmark(
+   roll_agg=roll_agg(closep, endd=endd, FUN=sum),
+   period_apply=period.apply(closep, INDEX=endd, FUN=sum),
+   times=10))[, c(1, 4, 5)]
> aggs <- period.sum(closep, INDEX=endd)
> head(aggs)
```

Performing Aggregations of *xts* Over Calendar Periods

The package *xts* has convenience wrapper functionals for `period.apply()`, that apply functions over calendar periods:

- `apply.daily()` applies functions over daily periods,
- `apply.weekly()` applies functions over weekly periods,
- `apply.monthly()` applies functions over monthly periods,
- `apply.quarterly()` applies functions over quarterly periods,
- `apply.yearly()` applies functions over yearly periods,

These functionals don't require specifying a vector of *end points*, because they determine the *end points* from the calendar periods.

```
> # Load package HighFreq
> library(HighFreq)
> # Extract closing minutely prices
> closep <- quantmod::Cl(rutils::etfenv$VTI["2019"])
> # Apply "mean" over daily periods
> aggs <- apply.daily(closep, FUN=sum)
> head(aggs)
```

Performing Aggregations Over Overlapping Intervals

The functional `period.apply()` performs aggregations over *non-overlapping* intervals.

But it's often necessary to perform aggregations over *overlapping* intervals, defined by a vector of *end points* and a *look-back interval*.

The *start points* are defined as the *end points* lagged by the interval width (number of periods in the *look-back interval*).

Each point in time has an associated *look-back interval*, which starts at a certain number of periods in the past (*start_point*) and ends at that point (*end_point*).

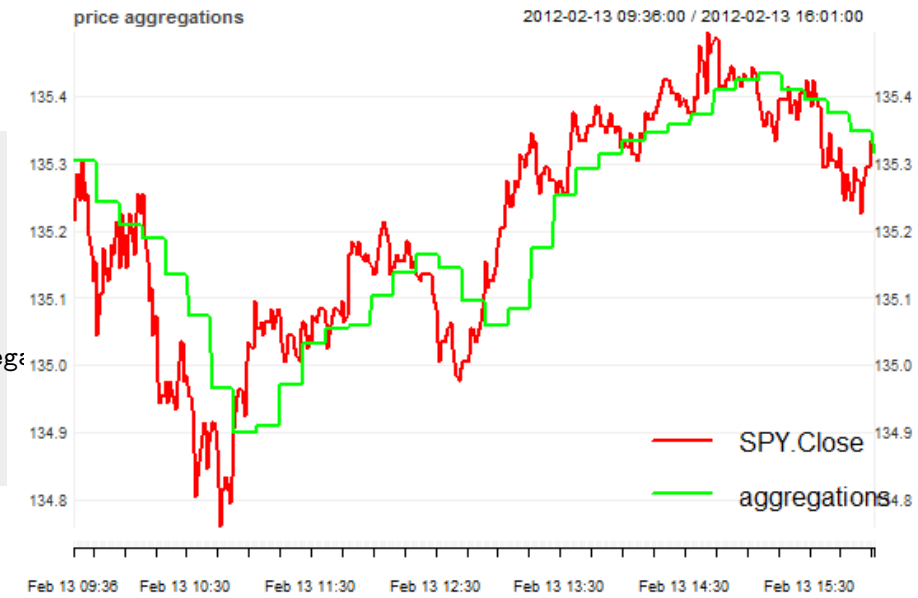
The variable `lookb` is equal to the number of end points in the *look-back interval*, while (`lookb - 1`) is equal to the number of intervals in the look-back.

```
> # Define endd with beginning stub
> npoints <- 5
> nrows <- NROW(closep)
> nagg <- nrows %/% npoints
> endd <- c(0, nrows-npoints*nagg + (0:nagg)*npoints)
> # Number of data points in lookb interval
> lookb <- 22
> # startp are endd lagged by lookb
> startp <- (endd - lookb + 1)
> startp <- ifelse(startp < 0, 0, startp)
> # Perform lapply() loop over lookback list
> aggs <- lapply(2:NROW(endd), function(it) {
+ xtsv <- closep[startp[it]:endd[it]]
+ c(max=max(xtsv), min=min(xtsv))
+ })  # end lapply
> # rbind list into single xts or matrix
> aggs <- rutils::do_call(rbind, aggs)
> # Coerce aggs into xts series
> aggs <- xts(aggs, order.by=zoo::index(closep[endd]))
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Extending Interval Aggregations

Interval aggregations produce values only at the *end points*, but they can be carried forward in time using the function `na.locf.xts()` from package *xts*.

```
> aggs <- cbind(closep, aggs)
> tail(aggs)
> aggs <- na.omit(xts:::na.locf.xts(aggs))
> # Plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "green")
> quantmod::chart_Series(aggs, theme=plot_theme, name="price aggrega
> legend("top", legend=colnames(aggs),
+   bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

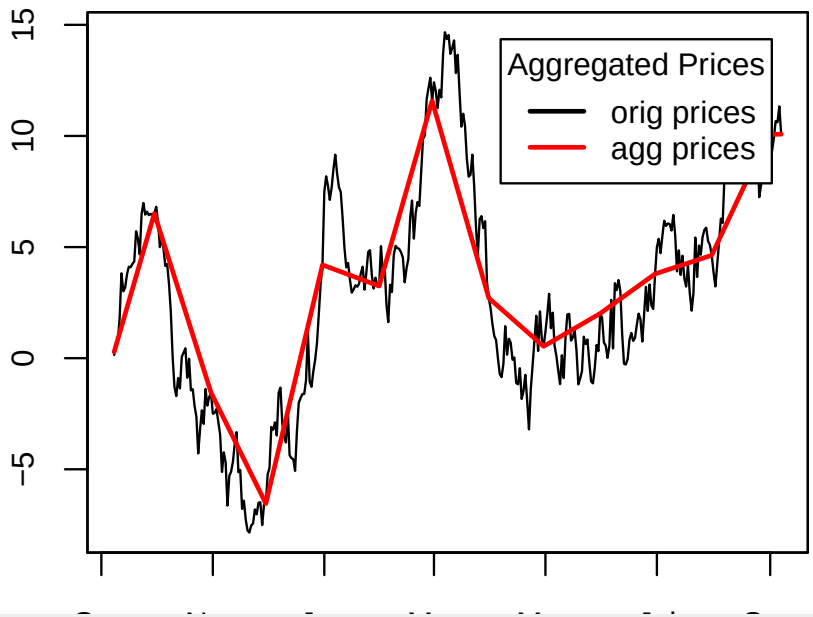


Performing Interval Aggregations of *zoo* Time Series

The method `aggregate.zoo()` performs aggregations of *zoo* series over non-overlapping intervals defined by a vector of aggregation groups (minutes, hours, days, etc.).

For example, `aggregate.zoo()` can calculate the average monthly returns.

```
> library(zoo)  # Load package zoo
> # Create zoo time series of random returns
> datev <- Sys.Date() + 0:365
> zoo_series <- zoo(rnorm(NROW(datev)), order.by=datev)
> # Create monthly dates
> dates_agg <- as.Date(as.yearmon(zoo::index(zoo_series)))
> # Perform monthly mean aggregation
> zoo_agg <- aggregate(zoo_series, by=datev_agg, FUN=mean)
> # Merge with original zoo - union of dates
> zoo_agg <- cbind(zoo_series, zoo_agg)
> # Replace NA's using locf
> zoo_agg <- na.locf(zoo_agg, na.rm=FALSE)
> # Extract aggregated zoo
> zoo_agg <- zoo_agg[zoo::index(zoo_series), 2]
```



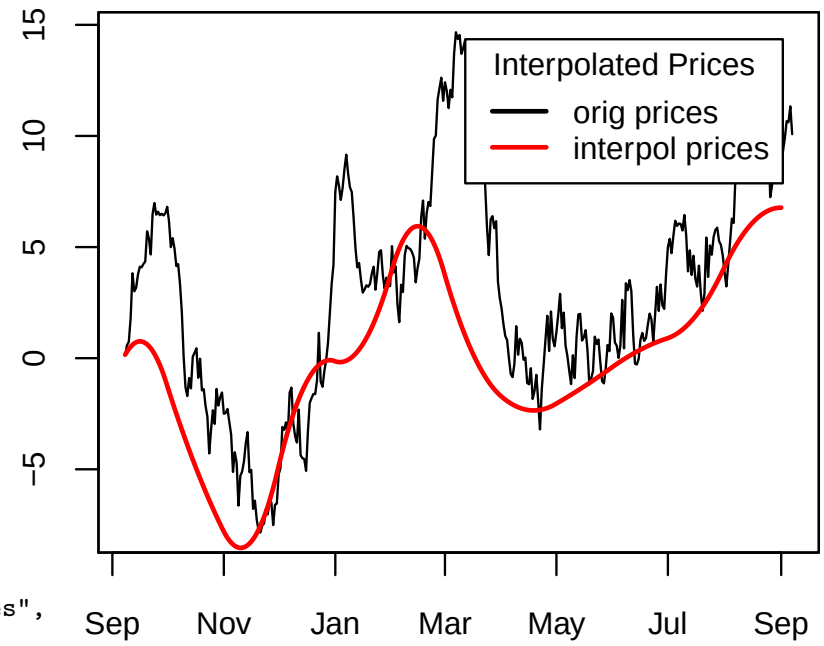

```
> # Plot original and aggregated cumulative returns
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_agg), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, bty="n",
+   title="Aggregated Prices", y.intersp=0.4,
+   leg=c("orig prices", "agg prices"),
+   lwd=2, bg="white", col=c("black", "red"))
```

Interpolating *zoo* Time Series

The package *zoo* has two functions for replacing `NA` values using interpolation:

- `na.approx()` performs linear interpolation,
- `na.spline()` performs spline interpolation,

```
> # Perform monthly mean aggregation
> zoo_agg <- aggregate(zoo_series, by=datev_agg, FUN=mean)
> # Merge with original zoo - union of dates
> zoo_agg <- cbind(zoo_series, zoo_agg)
> # Replace NA's using linear interpolation
> zoo_agg <- na.approx(zoo_agg)
> # Extract interpolated zoo
> zoo_agg <- zoo_agg[zoo::index(zoo_series), 2]
> # Plot original and interpolated zoo
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_agg), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title="Interpolated Prices",
+  leg=c("orig prices", "interpol prices"), lwd=2, bg="white",
+  col=c("black", "red"), bty="n", y.intersp=0.4)
```

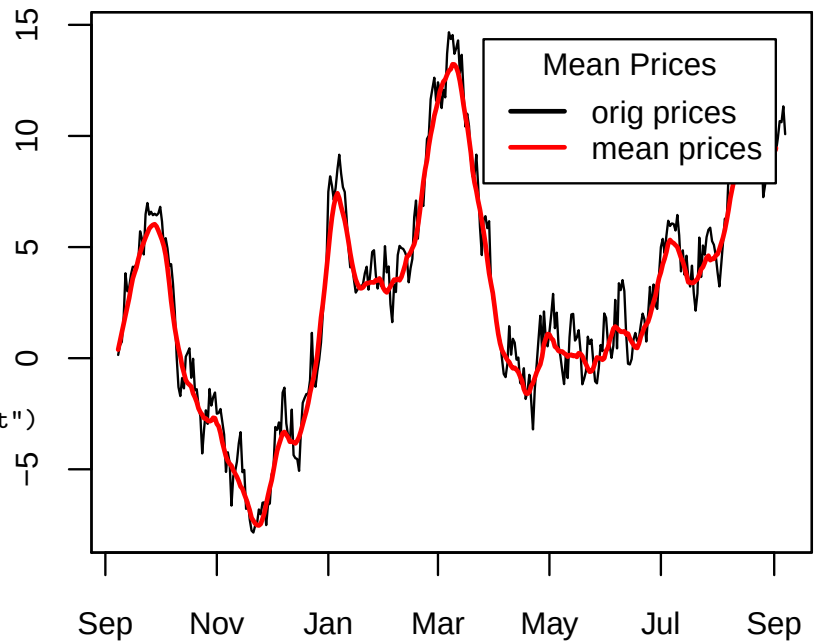


Performing Trailing Aggregations Over *zoo* Time Series

The package *zoo* has several functions for trailing calculations:

- `rollapply()` performing aggregations over a trailing (sliding) interval,
- `rollmean()` calculating trailing means,
- `rollmedian()` calculating trailing median,
- `rollmax()` calculating trailing max,

```
> # "mean" aggregation over interval with width=11
> zoo_mean <- rollapply(zoo_series, width=11, FUN=mean, align="right")
> # Merge with original zoo - union of dates
> zoo_mean <- cbind(zoo_series, zoo_mean)
> # Replace NA's using na.locf
> zoo_mean <- na.locf(zoo_mean, na.rm=FALSE, fromLast=TRUE)
> # Extract mean zoo
> zoo_mean <- zoo_mean[zoo::index(zoo_series), 2]
> # Plot original and interpolated zoo
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_mean), lwd=2, col="red")
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title="Mean Prices",
+  leg=c("orig prices", "mean prices"), lwd=2, bg="white",
+  col=c("black", "red"), bty="n", y.intersp=0.4)
```



aggregations are taken from the past,

The argument `align="right"` determines that

Brownian Motion

Brownian motion B_T is a stochastic process, with the increments $\mathrm{d}B_t$ which are independent and normally distributed, with mean zero and variance equal to $\mathrm{d}t$.

$$\mathrm{d}B_t = \xi_t \sqrt{\mathrm{d}t}$$

Where the ξ_t are random and independent *innovations* following the standard normal distribution $\phi(0, 1)$, with the expected values: $\mathbb{E}[\xi_t] = 0$, $\mathbb{E}[\xi_t^2] = 1$, and $\mathbb{E}[\xi_{t1}\xi_{t2}] = 0$.

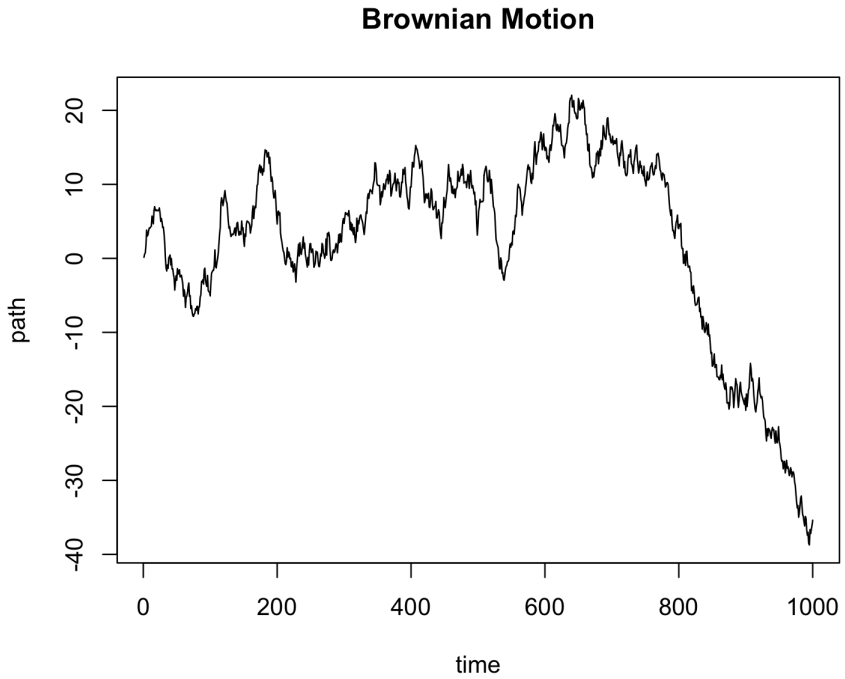
The Brownian motion path B_T is equal to the sum of its increments $\mathrm{d}B_t$:

$$B_T = \sum_{i=1}^{n} \mathrm{d}B_t$$

Where the number of time increments n is equal to the total time of evolution T divided by the increment size $\mathrm{d}t$: $n = T/\mathrm{d}t$.

The variance of Brownian motion is equal to the time of its evolution T:

$$\sigma^2 = \mathbb{E}[(\sum_{i=1}^{n} \xi_t \sqrt{\mathrm{d}t})^2] = \sum_{i=1}^{n} \mathbb{E}[\xi_t^2]\mathrm{d}t = T$$




```
> # Simulate a Brownian motion path
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> pathv <- cumsum(rnorm(nrows))
> plot(pathv, type="l", xlab="time", ylab="path",
+      main="Brownian Motion")
```

The Maximum Value of Brownian Motion

The distribution of the maximum value m of a Brownian motion path B_t can be calculated using the *reflection principle*.

The *reflection principle* states that the mirror image (reflection) of a Brownian motion path has the same probability as the original path.

The probability that the Brownian motion path B_t is at some point greater than some value m is the sum of the joint probability, that after the Brownian motion reaches the value m, it ends up greater than m, plus the joint probability that it ends up less than m:

$$p(B_t > m) = p((B_t > m) \& (B_T > m)) + p((B_t > m) \& (B_T < m))$$

By the *reflection principle*, both probabilities are equal, and also $p((B_t > m) \& (B_T > m)) = p(B_T > m)$, so that:

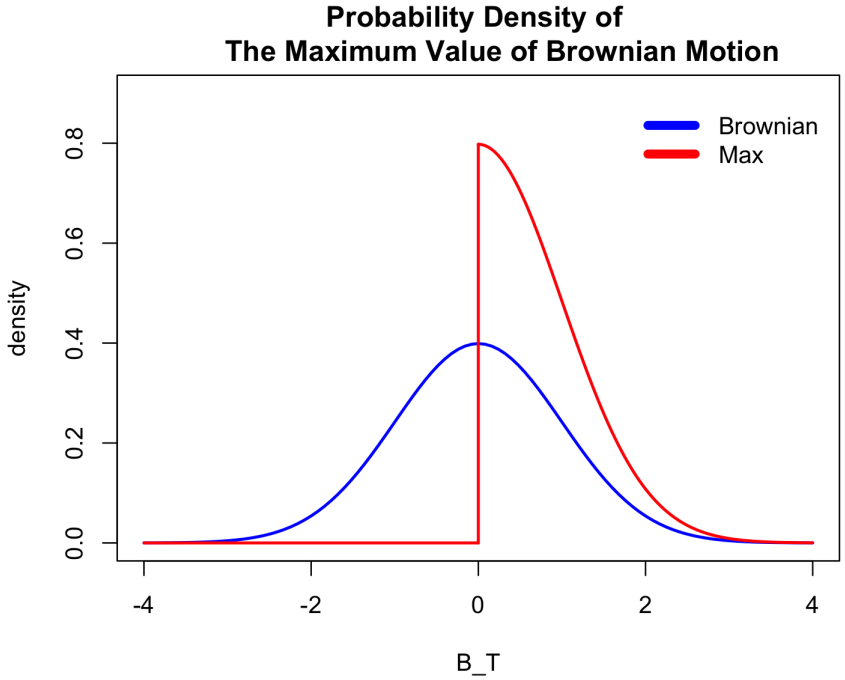
$$p(B_t > m) = 2p(B_T > m)$$

And the probability density of the maximum value m is equal to:

$$\varphi(m) = \sqrt{\frac{2}{\pi T}} e^{-\frac{m^2}{2T}}$$

The average value of the maximum value is equal to:

$$\bar{m} = \sqrt{\frac{2T}{\pi}}$$




```
> # Plot the density of Brownian Motion
> curve(expr=dnorm(x), xlim=c(-4, 4), ylim=c(0, 0.9),
+   xlab="B_T", ylab="density", lwd=2, col="blue")
> # Plot the density of the maximum of Brownian Motion
> curve(expr=2*dnorm(x), xlim=c(0, 4), xlab="", ylab="",
+   lwd=2, col="red", add=TRUE)
> lines(x=c(0, 0), y=c(0, sqrt(2/pi)), lwd=2, col="red")
> lines(x=c(-4, 0), y=c(0, 0), lwd=2, col="red")
> title(main="Probability Density of
+ The Maximum Value of Brownian Motion", line=0.5)
> legend("topright", inset=0.0, bty="n", y.intersp=0.4,
+  title=NULL, c("Brownian", "Max"), lwd=6,
+  col=c("blue", "red"))
```

The Range of Brownian Motion

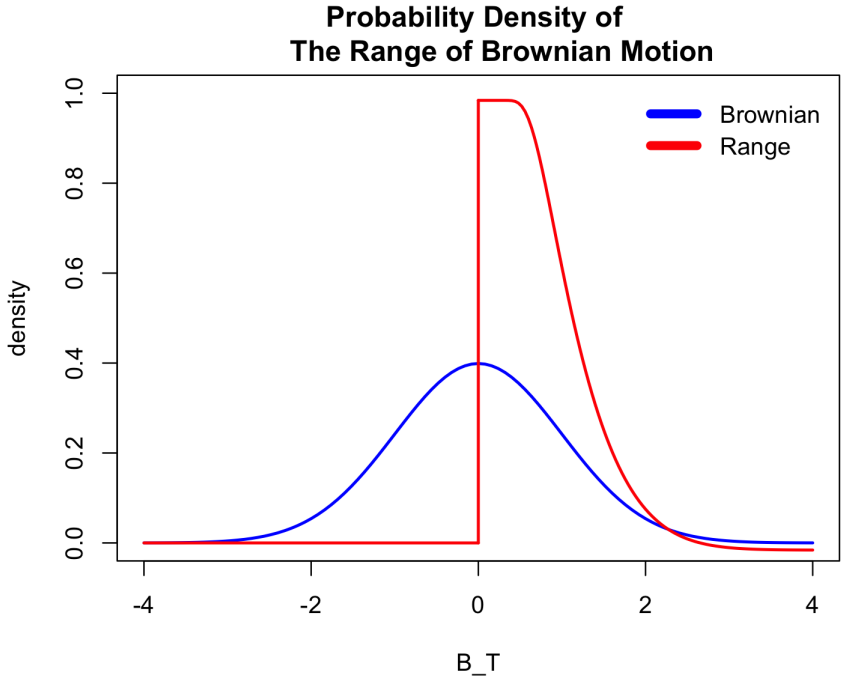
The range of a Brownian motion path B_t is equal to the difference between its maximum value minus its minimum value. The range is also called the drawdown.

The probability density of the range r is equal to the infinite series:

$$p(r) = 2 \sum_{n=1}^{\infty} \frac{\sin{(n-0.5)\pi}}{(n-0.5)\pi} (1 - e^{-\frac{(n-0.5)^2 \pi^2 T}{2r^2}})$$

The average value of the range is equal to:

$$\bar{r} = \sqrt{\frac{\pi T}{2}}$$




```
> # Series element
> fun1 <- function(n, r) { 2*sin((n-0.5)*pi)/((n-0.5)*pi) *
+   (1-exp(-((n-0.5)^2)*pi^2/2/r^2)) }
> # fun2 <- function(x) { sum(sapply(1:10, function(n) fun1(n, x))) }
> # fun2 <- function(x) { fun1(1, x) + fun1(2, x) + fun1(3, x) + fu
> # Sum of fun1
> fun2 <- function(x) {
+   valf <- 0
+   for (n in 1:20) { valf <- valf + fun1(n, x) }
+   return(valf)
+   } # end fun2
> # Theoretical average value of the range
> fun2(2)
> # Average value of the range from integration (not quite close)
> integrate(fun2, lower=0.01, upper=4)
```

```
> # Plot the density of Brownian Motion
> curve(expr=dnorm(x), xlim=c(-4, 4), ylim=c(0, 1.0),
+   xlab="B_T", ylab="density", lwd=2, col="blue")
> # Plot the density of the range of Brownian Motion
> curve(expr=fun2(x), xlim=c(0, 4), xlab="", ylab="",
+   lwd=2, col="red", add=TRUE)
> lines(x=c(0, 0), y=c(0, fun2(0.01)), lwd=2, col="red")
> lines(x=c(-4, 0), y=c(0, 0), lwd=2, col="red")
> title(main="Probability Density of
+ The Range of Brownian Motion", line=0.5)
> legend("topright", inset=0.0, bty="n", y.intersp=0.7,
+  title=NULL, c("Brownian", "Range"), lwd=6,
+  col=c("blue", "red"))
```

Monte Carlo Simulation

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

The *Monte Carlo* data samples can then be 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.

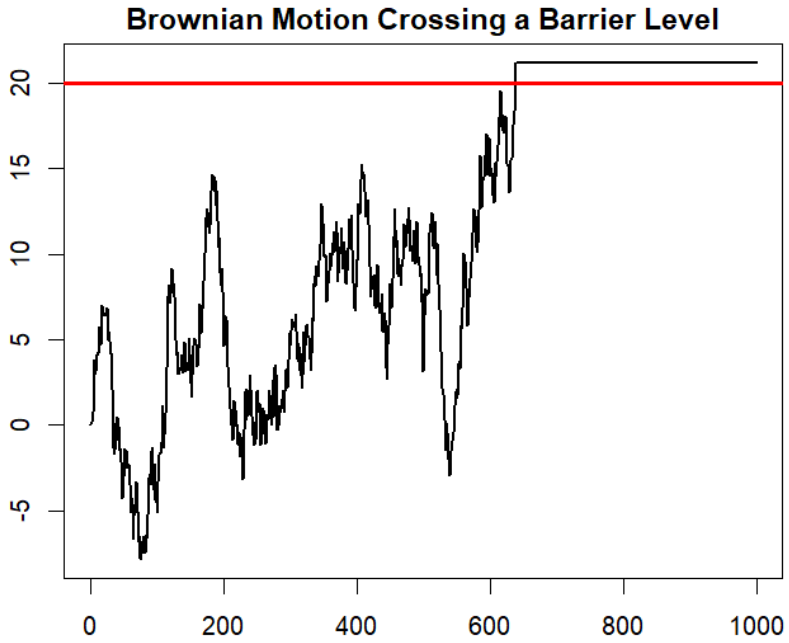
```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Sample from Standard Normal Distribution
> nrows <- 1000
> datav <- rnorm(nrows)
> # Sample mean - MC estimate
> mean(datav)
> # Sample standard deviation - MC estimate
> sd(datav)
> # Monte 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],
+   quantv = 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.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> barl <- 20  # Barrier level
> nrows <- 1000  # Number of simulation steps
> pathv <- numeric(nrows)  # Allocate path vector
> pathv[1] <- rnorm(1)  # 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)
```

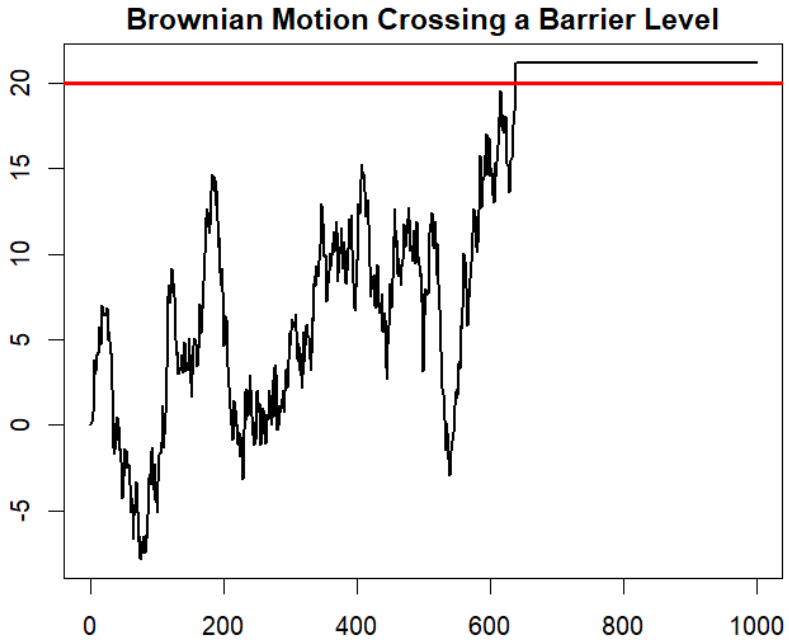


Simulating Brownian Motion Using Vectorized Functions

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.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> 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*:

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

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

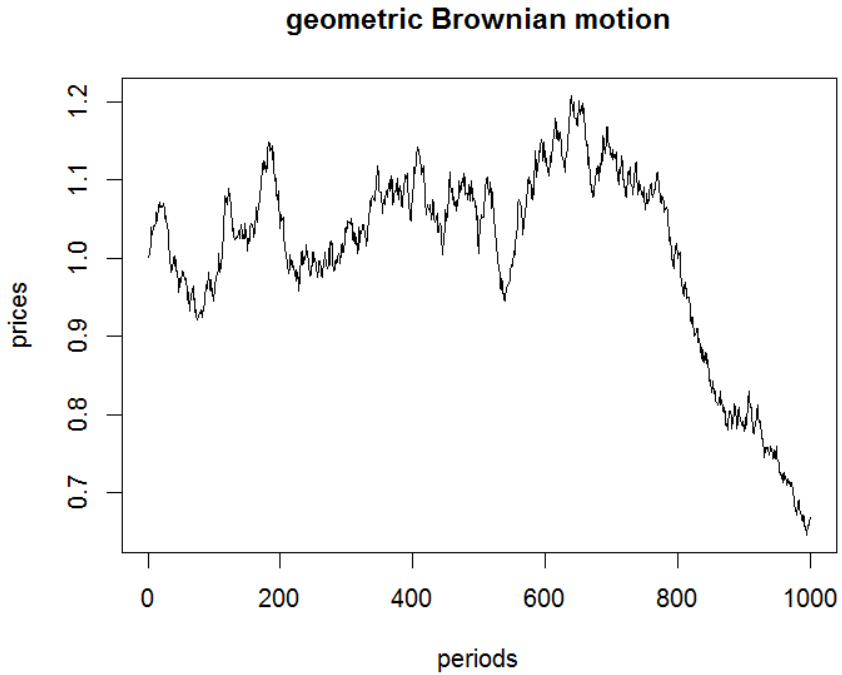
$$\mathrm{d}p_t = \mu p_t \mathrm{d}t + \sigma \, p_t \mathrm{d}B_t$$

Where σ is the volatility of asset returns, and B_t is a *Brownian Motion*, with $\mathrm{d}B_t$ following the normal distribution $\phi(0, \sqrt{\mathrm{d}t})$, with the volatility $\sqrt{\mathrm{d}t}$, equal to the square root of the time increment $\mathrm{d}t$.

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

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

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




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

Simulating Random *OHLC* Prices

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

The function `sample()` selects a random sample from a vector of data elements.

The function `sample()` with `replace=TRUE` selects samples with replacement (the default is `replace=FALSE`).

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

The *Log-normal* Probability Distribution

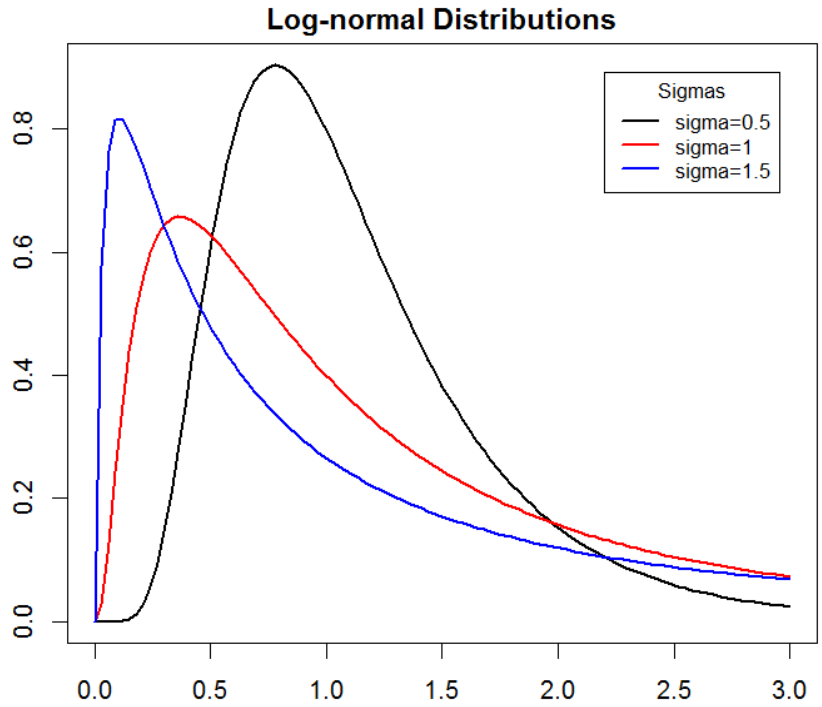
If x follows the *Normal* distribution $\phi(x, \mu, \sigma)$, then the exponential of x: $y = e^x$ follows the *Log-normal* distribution $\log \phi()$:

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

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

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

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




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

The Standard Deviation of *Log-normal* Prices

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

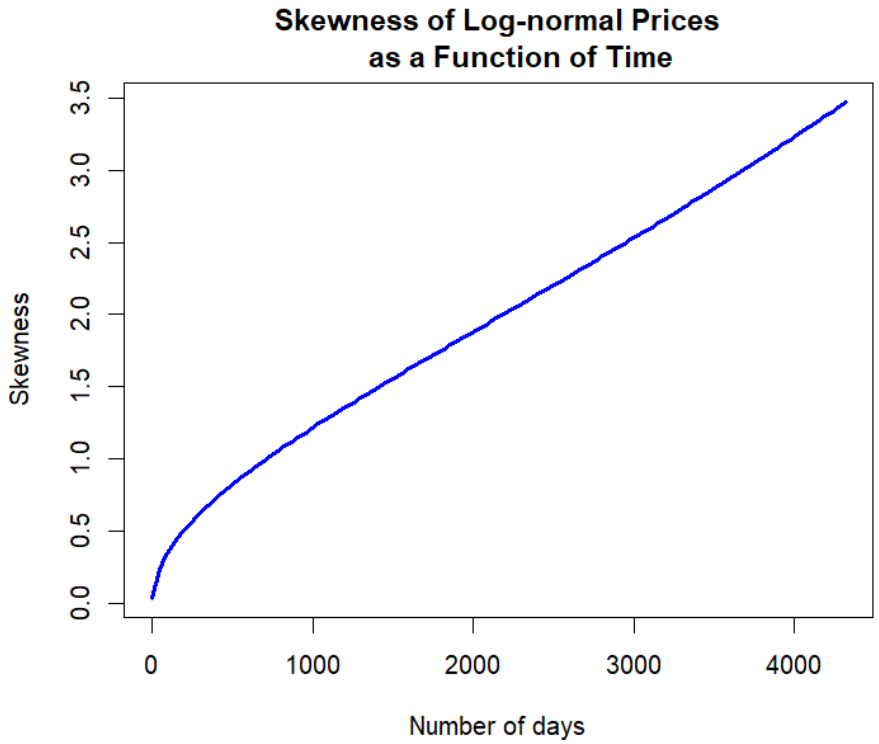
The standard deviation of *log-normal* prices is equal to the return volatility σ_r times the square root of time: $\sigma = \sigma_r \sqrt{t}$.

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

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

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

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




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

The Mean and Median of *Log-normal* Prices

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

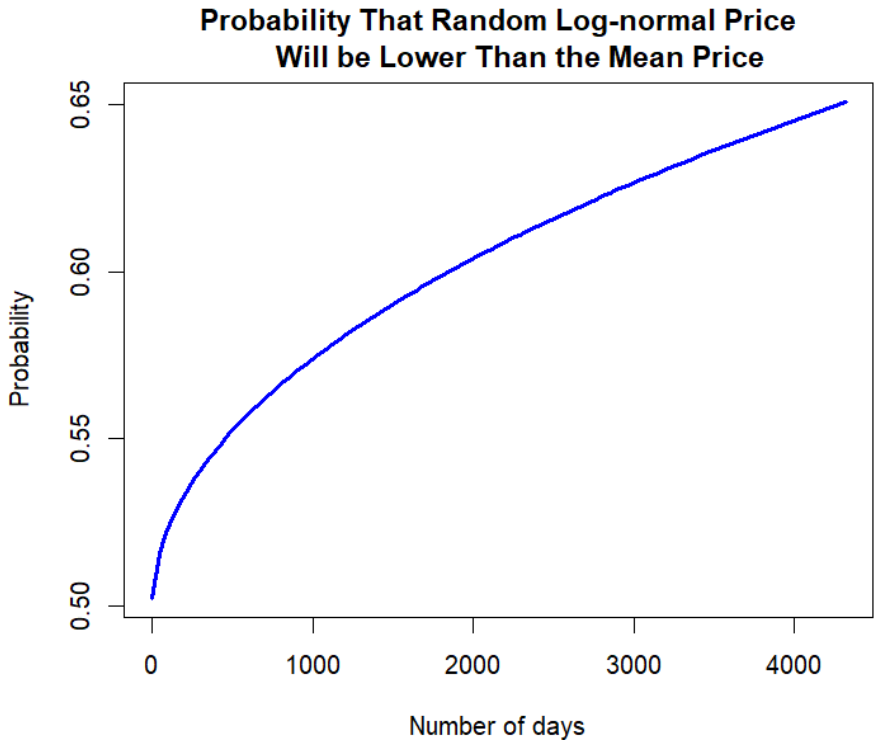
So if stock prices follow *Geometric Brownian motion* and are distributed *log-normally*, then a stock selected at random will have a high probability of havng a lower price than the mean expected price.

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



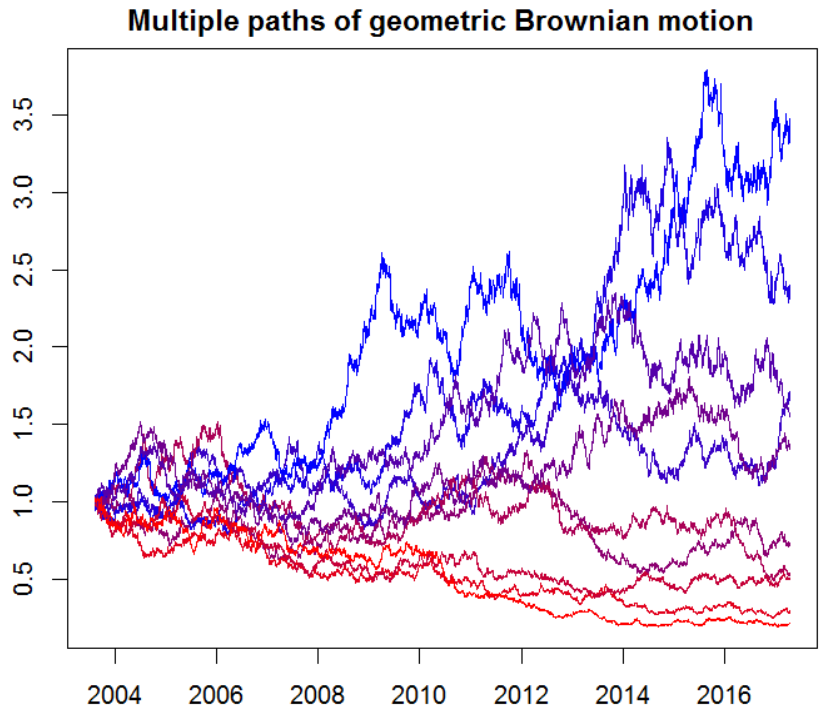

```
> # Probability that random log-normal price will be lower than the
> curve(expr=pnorm(sigmav*sqrt(x)/2),
+ xlim=c(1, nrows), lwd=3,
+ xlab="Number of days", ylab="Probability", col="blue",
+ main="Probability That Random Log-normal Price
+ Will be Lower Than the Mean Price")
```

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 the 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)
```

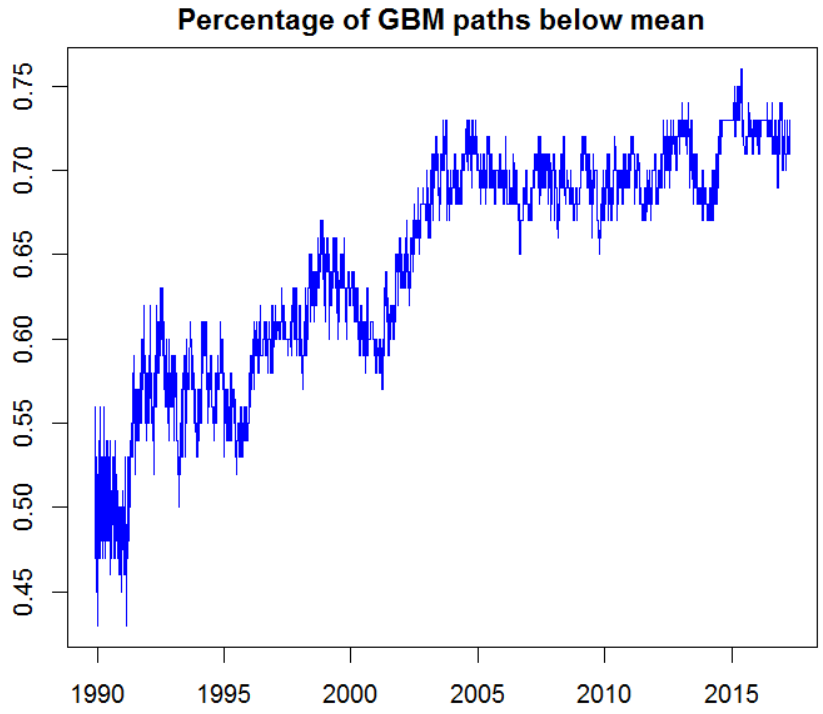


Distribution of Paths of Geometric Brownian Motion

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

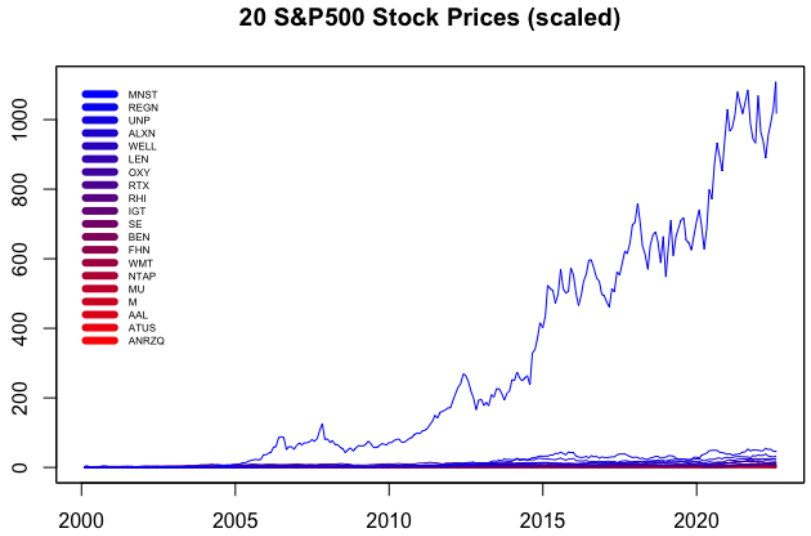
```
> # Define the daily volatility and growth rate
> sigmav <- 0.01; drift <- 0.0; nrows <- 10000
> npaths <- 100
> # Simulate multiple paths of geometric Brownian motion
> pricev <- rnorm(npaths*nrows, sd=sigmav) + drift - sigmav^2/2
> pricev <- matrix(pricev, nc=npaths)
> pricev <- exp(matrixStats::colCumsums(pricev))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



Time Evolution of Stock Prices

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

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



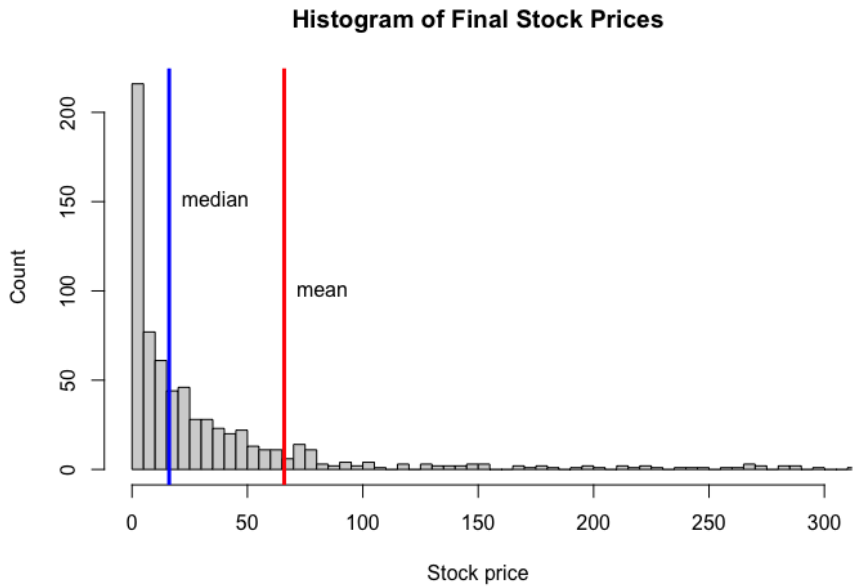

```
> # Plot xts time series of prices
> colorv <- colorRampPalette(c("red", "blue"))(NROW(symbolv))
> endd <- rutils::calc_endpoints(pricev, interval="weeks")
> plot.zoo(pricev[endd, symbolv], main="20 S&P500 Stock Prices (scal
+    xlab=NA, ylab=NA, plot.type="single", col=colorv)
> legend(x="topleft",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

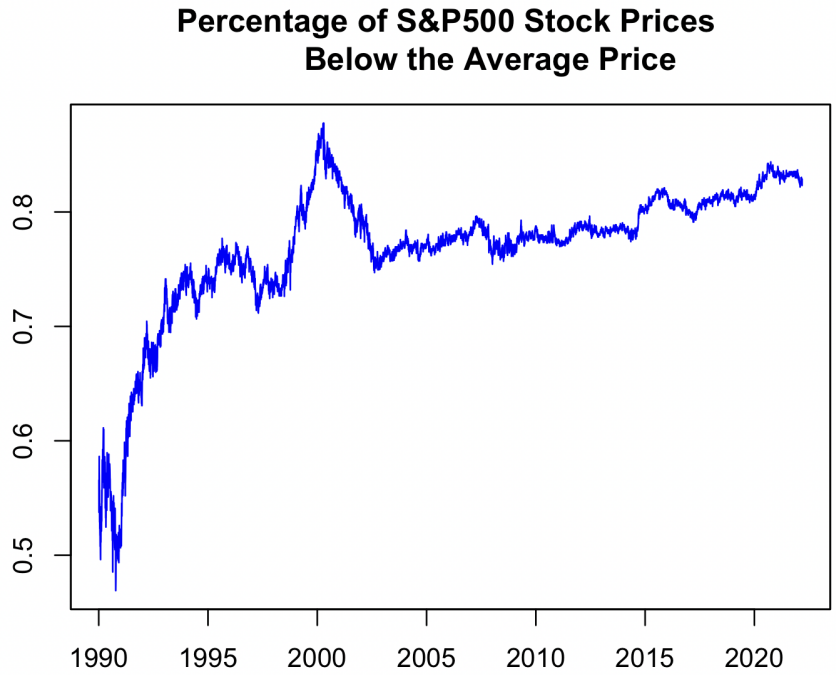
Distribution of Stock Prices Over Time

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")
```

draft: Fractional Brownian Motion

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

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

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

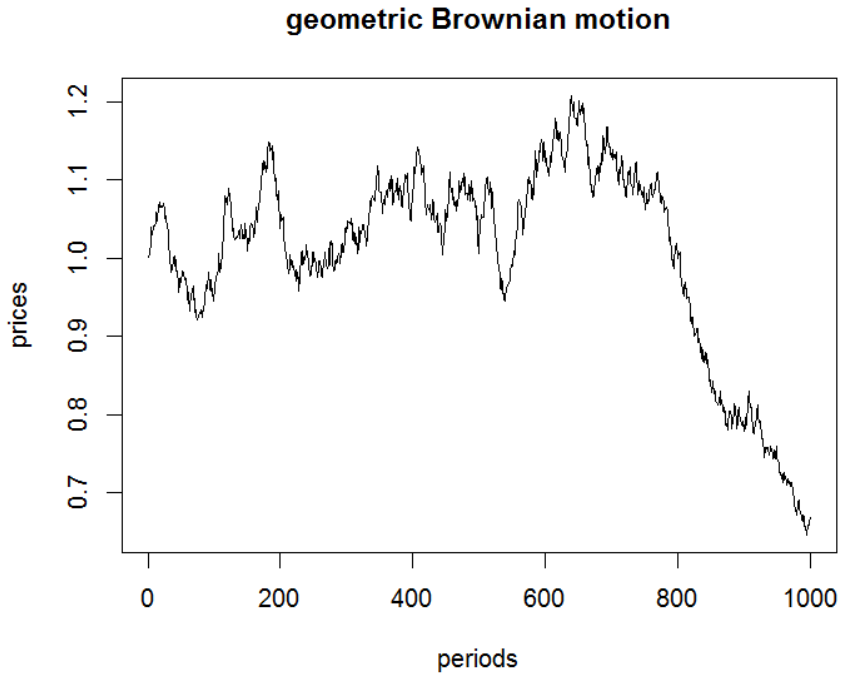
$$\mathrm{d}p_t = \mu p_t \mathrm{d}t + \sigma \, p_t \mathrm{d}B_t$$

Where σ is the volatility of asset returns, and B_t is a *Brownian Motion*, with $\mathrm{d}B_t$ following the normal distribution $\phi(0, \sqrt{\mathrm{d}t})$, with the volatility $\sqrt{\mathrm{d}t}$, equal to the square root of the time increment $\mathrm{d}t$.

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

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

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




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

Autocorrelation Function of Time Series

The *autocorrelation* of lag k of a time series of returns r_t is equal to:

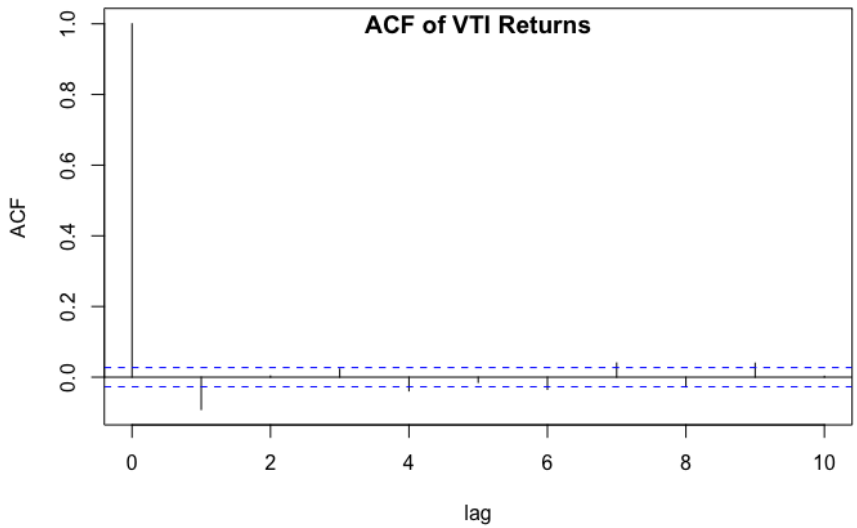
$$\rho_k = \frac{\sum_{t=k+1}^{n}(r_t - \bar{r})(r_{t-k} - \bar{r})}{(n-k)\,\sigma^2}$$

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
> retp <- na.omit(rutils::etfenv$returns$VTI)
> retp <- drop(zoo::coredata(retp))
> # Plot autocorrelations of VTI returns using stats::acf()
> stats::acf(retp, lag=10, xlab="lag", main="")
> title(main="ACF of VTI Returns", line=-1)
> # Calculate two-tailed 95% confidence interval
> qnorm(0.975)/sqrt(NROW(retp))
```



The *VTI* time series of returns has small, but statistically significant negative 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
> acfl <- acf(retp, plot=FALSE)
> summary(acfl)
> # Print the ACF data
> print(acfl)
> dim(acfl$acf)
> dim(acfl$lag)
> head(acfl$acf)
```

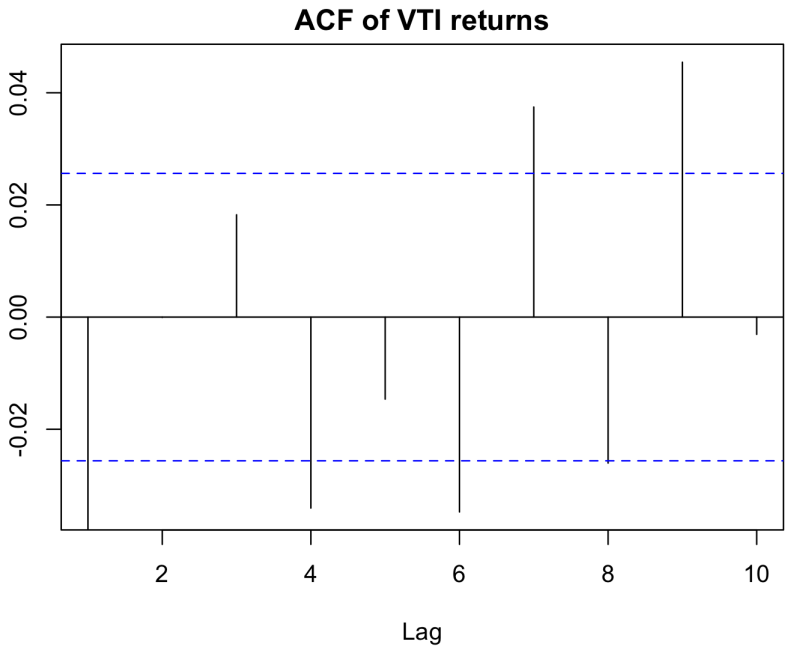
```
> plot_acf <- function(xtsv, lagg=10, plotobj=TRUE,
+                xlab="Lag", ylab="", main="", ...) {
+   # Calculate the ACF without a plot
+   acfl <- acf(x=xtsv, lag.max=lagg, plot=FALSE, ...)
+   # Remove first element of ACF data
+   acfl$acf <- array(data=acfl$acf[-1],
+     dim=c((dim(acfl$acf)[1]-1),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

Autocorrelations of Stock 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.

```
> # Autocorrelations of VTI returns
> rutils::plot_acf(retp, lag=10, main="ACF of VTI returns")
```



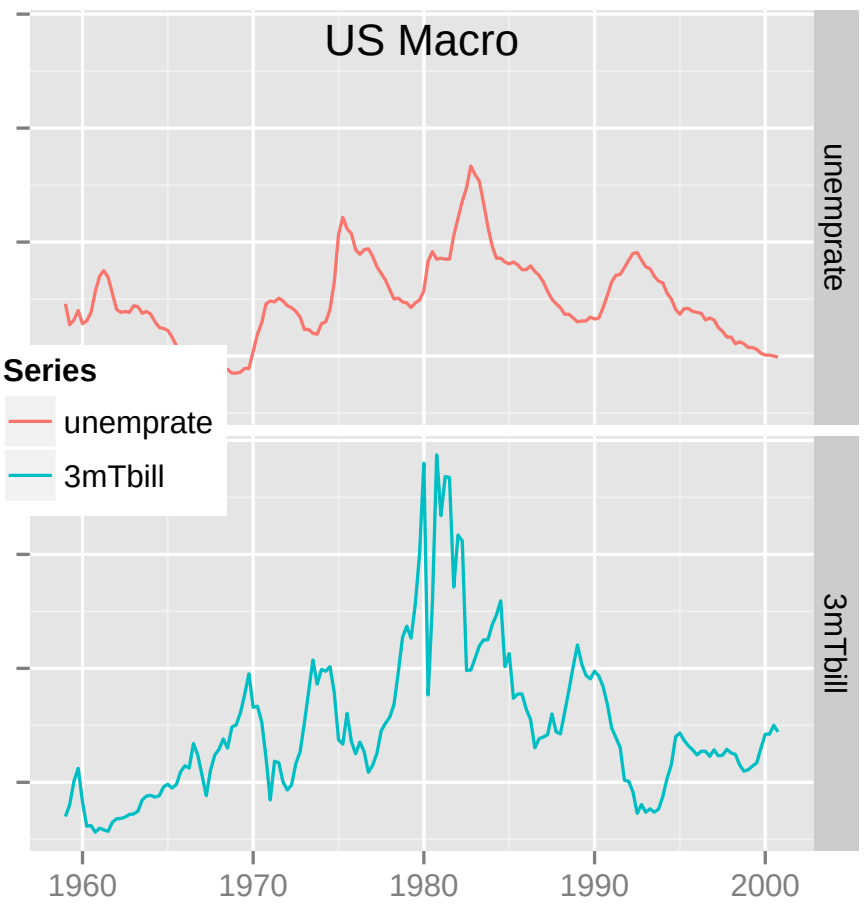
depr: U.S. Macroeconomic Data

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
```



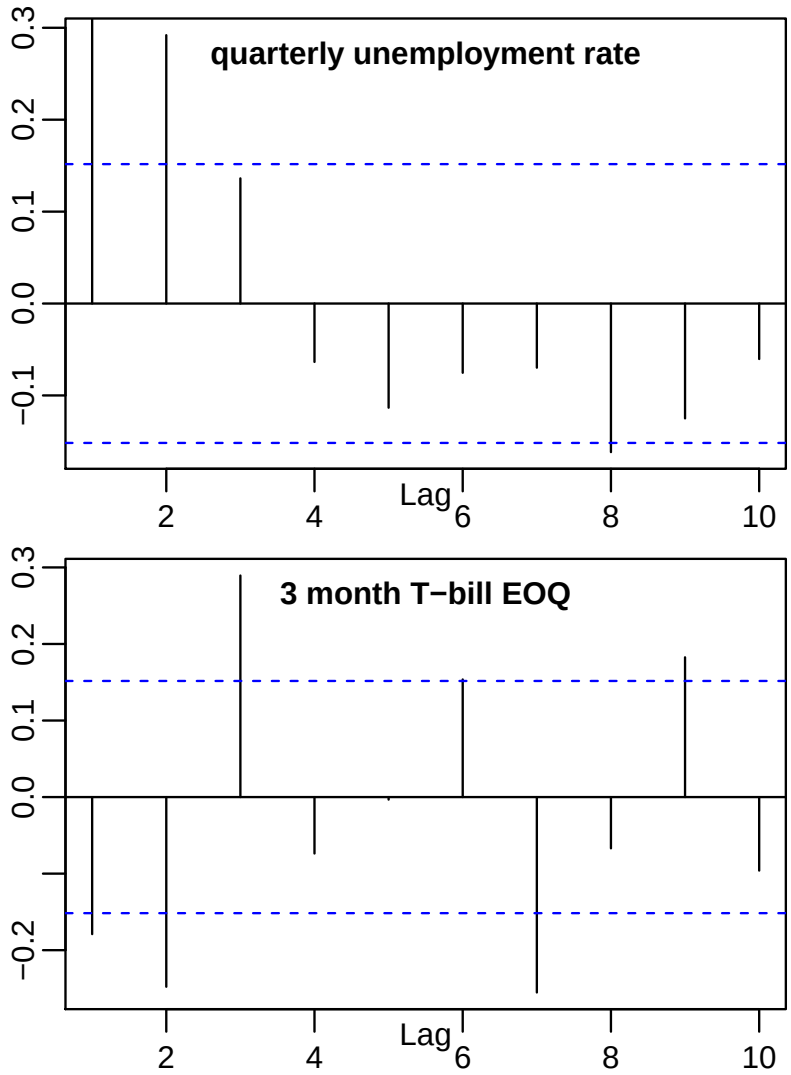
depr: Autocorrelations of Econometric Data

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")
```



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(retp, lag=10, type="Ljung")
> # Ljung-Box test for random returns
> Box.test(rnorm(NROW(retp)), 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
> retp <- rutils::etfenv$returns[, c("VTI", "XLF")]
> retp <- na.omit(retp)
> nrows <- NROW(retp)
> # Center (de-mean) and scale the returns
> retp <- apply(retp, MARGIN=2, function(x) (x-mean(x))/sd(x))
> apply(retp, MARGIN=2, sd)
> # Calculate the correlation
> drop(retp[, "VTI"] %*% retp[, "XLF"])/(nrows-1)
> corv <- cor(retp[, "VTI"], retp[, "XLF"])
> # Test statistical significance of correlation
> cortest <- cor.test(retp[, "VTI"], retp[, "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(retp, 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*

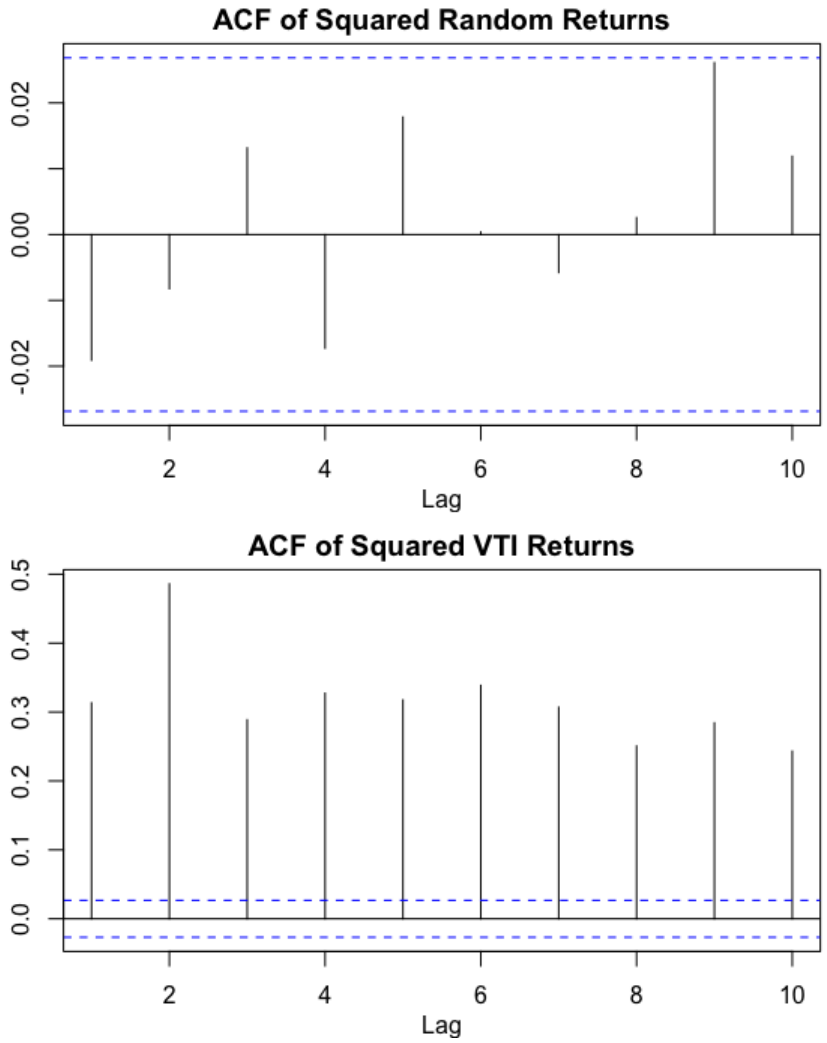
Autocorrelations of Squared VTI Returns

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.

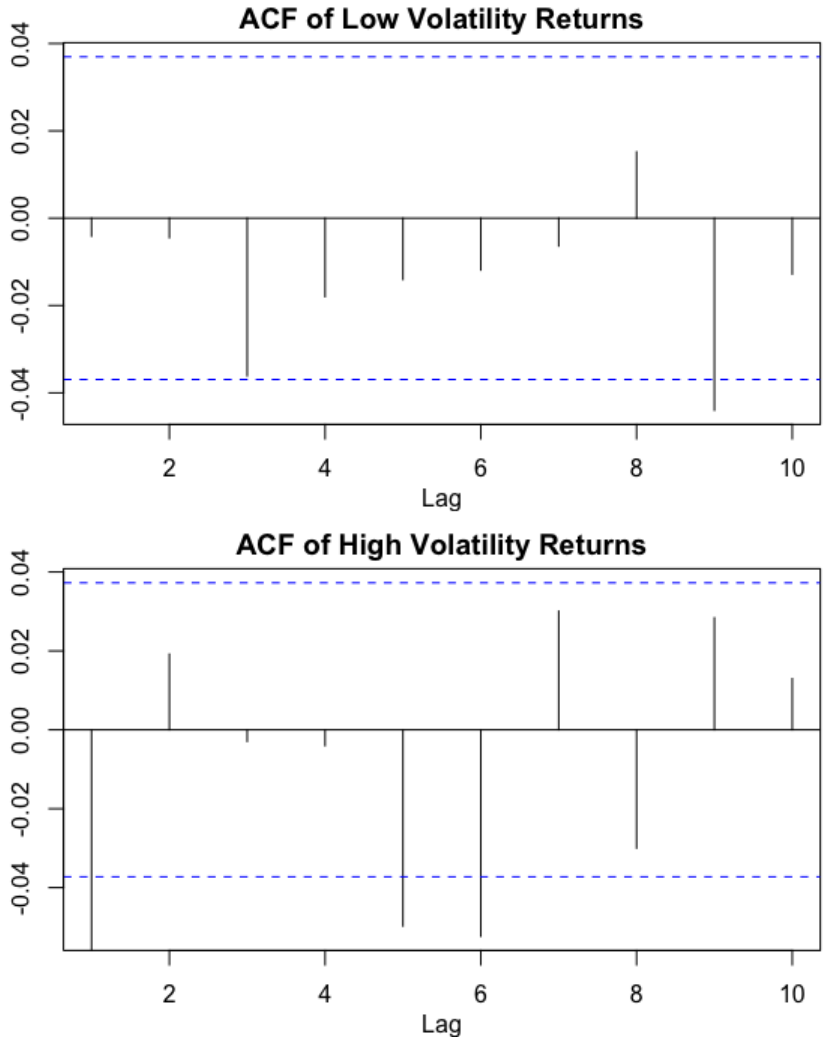
```
> # Open plot window under MS Windows
> x11(width=6, height=7)
> # Set two vertical plot panels
> par(mfrow=c(2,1))
> par(mar=c(3, 3, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> # Plot ACF of squared random returns
> rutils::plot_acf(rnorm(NROW(retp))^2, lag=10,
+   main="ACF of Squared Random Returns")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



Autocorrelations in Intervals of Low and High Volatility

Stock returns have significant negative autocorrelations in time intervals with high volatility, but much less in time intervals with low volatility.

```
> # Calculate the weekly end points
> endd <- rutils::calc_endpoints(retp, interval="weeks")
> npts <- NROW(endd)
> # Calculate the monthly VTI volatilities and their median volatil
> stdev <- sapply(2:npts, function(endp) {
+   sd(retp[endd[endp-1]:endd[endp]])
+ })  # end sapply
> medianv <- median(stdev)
> # Calculate the stock returns of low volatility intervals
> retlow <- lapply(2:npts, function(endp) {
+   if (stdev[endp-1] <= medianv)
+     retp[endd[endp-1]:endd[endp]]
+ })  # end lapply
> retlow <- rutils::do_call(c, retlow)
> # Calculate the stock returns of high volatility intervals
> rethigh <- lapply(2:npts, function(endp) {
+   if (stdev[endp-1] > medianv)
+     retp[endd[endp-1]:endd[endp]]
+ })  # end lapply
> rethigh <- rutils::do_call(c, rethigh)
> # Plot ACF of low volatility returns
> rutils::plot_acf(retlow, lag=10,
+  main="ACF of Low Volatility Returns")
> Box.test(retlow, lag=10, type="Ljung")
> # Plot ACF of high volatility returns
> rutils::plot_acf(rethigh, lag=10,
+  main="ACF of High Volatility Returns")
> Box.test(rethigh, lag=10, type="Ljung")
```

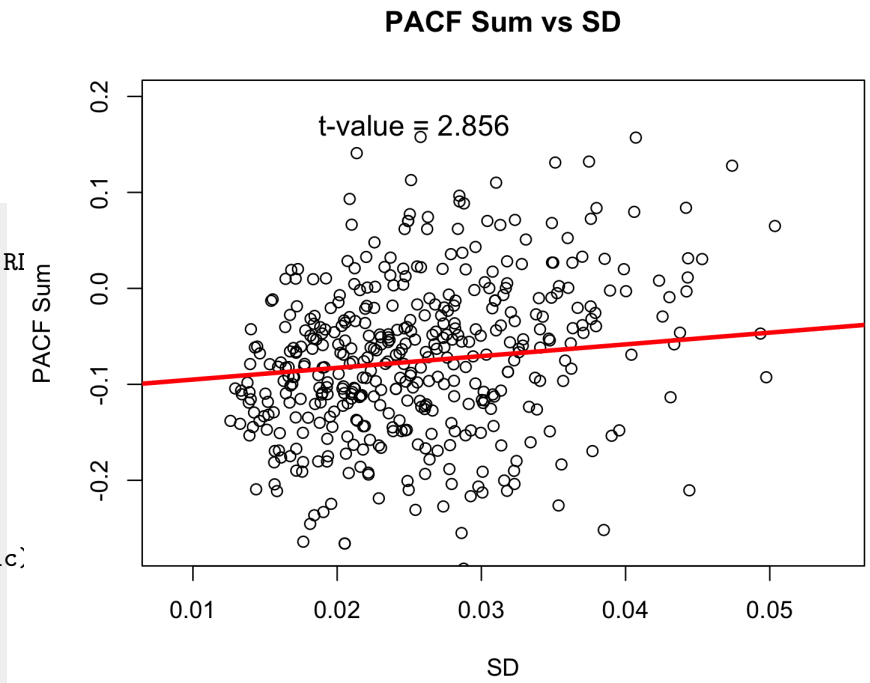


Autocorrelations of Low and High Volatility Stocks

Low volatility stocks have more significant negative autocorrelations than high volatility stocks.

The lowest volatility stocks have negative autocorrelations similar to *VTI*.

```
> # Load daily S&P500 stock returns
> load("/Users/jerzy/Develop/lecture_slides/data/sp500_returnstop.RI
> # Calculate the stock volatilities and the sum of the ACF
> library(parallel)  # Load package parallel
> ncores <- detectCores() - 1
> statm <- mclapply(retstock, function(retp) {
+   retp <- na.omit(retp)
+   # Calculate the sum of the ACF
+   acfsum <- sum(pacf(retp, lag=10, plot=FALSE)$acf)
+   # Calculate the Ljung-Box statistic
+   lbstat <- unname(Box.test(retp, lag=10, type="Ljung")$statistic)
+   c(stdev=sd(retp), acfsum=acfsum, lbstat=lbstat)
+ }, mc.cores=ncores)  # end mclapply
> statm <- do.call(rbind, statm)
> statm <- as.data.frame(statm)
> # Calculate the ACF sum for stock volatility quantiles
> confl <- seq(0.1, 0.9, 0.1)
> stdq <- quantile(statm[, "stdev"], confl)
> acfq <- quantile(statm[, "acfsum"], confl)
> plot(stdq, acfq, xlab="volatility", ylab="PACF Sum",
+      main="PACF Sum vs Volatility")
> # Compare the ACF sum for stock volatility quantile with VTI
> acfq[1]
> sum(pacf(na.omit(rutils::etfenv$returns$VTI), lag=10, plot=FALSE)
```

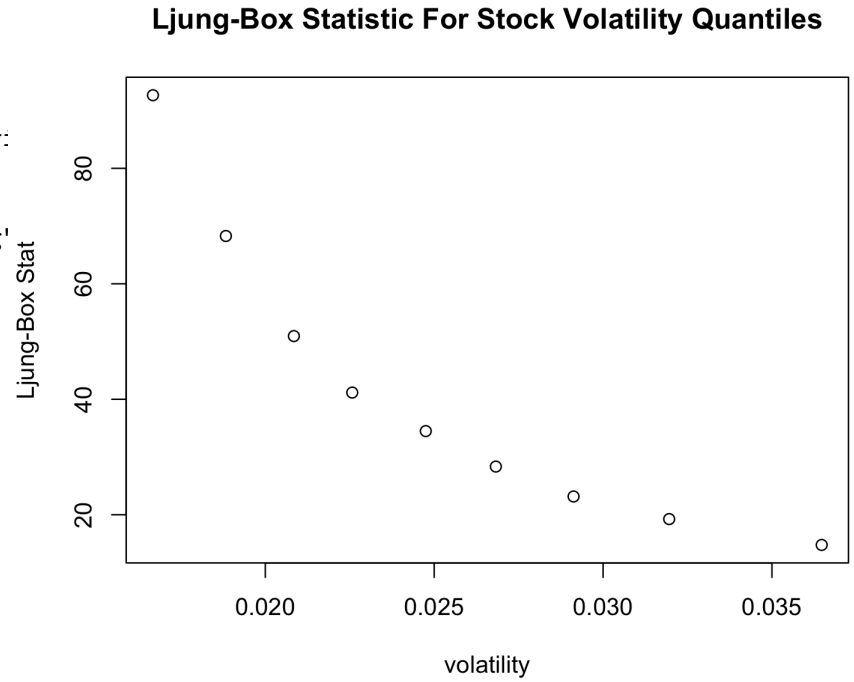



```
> # Scatter plot of the sum of the ACF vs the standard deviation
> regmod <- lm(acfsum ~ stdev, data=statm)
> plot(acfsum ~ stdev, data=statm, xlab="SD", ylab="PACF Sum",
+      xlim=c(0.5*min(stdq), 1.5*max(stdq)), ylim=c(1.5*min(acfq),
+      main="PACF Sum vs SD")
> abline(regmod, lwd=3, col="red")
> tvalue <- summary(regmod)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=mean(stdq), y=6*max(acfq),
+      lab=paste("t-value =", tvalue), lwd=2, cex=1.2)
```

Ljung-Box statistics of Low and High Volatility Stocks

The lowest volatility quantile of stocks has more significant negative autocorrelations than *VTI* does.

```
> # Compare the Ljung-Box statistics for lowest volatility stocks w
> lbstatq <- rev(quantile(statm[, "lbstat"], conf1))
> lbstatq[1]
> Box.test(na.omit(rutils::etfenv$returns$VTI), lag=10, type="Ljung'
```



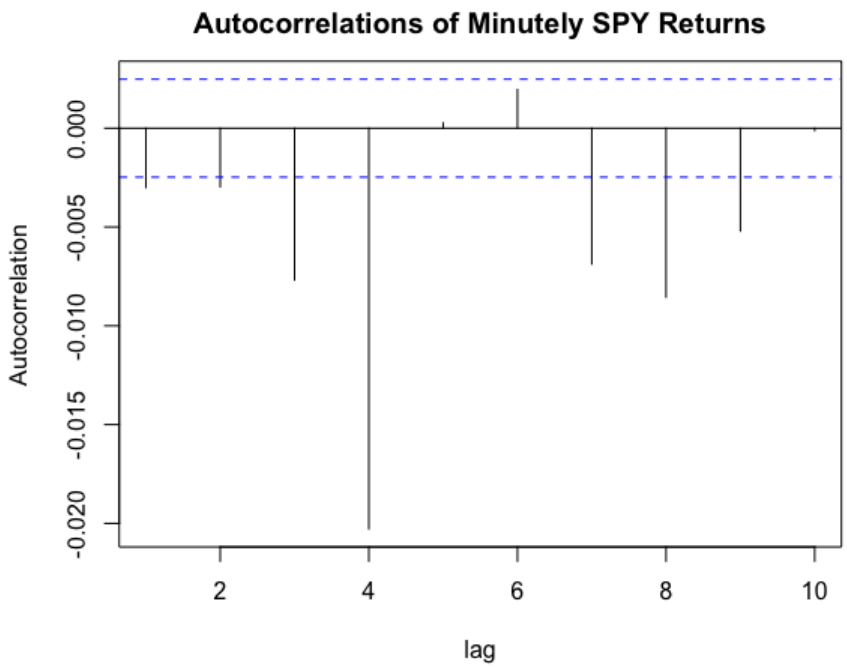

```
> # Plot Ljung-Box test statistic for volatility quantiles
> plot(stdq, lbstatq, xlab="volatility", ylab="Ljung-Box Stat",
+      main="Ljung-Box Statistic For Stock Volatility Quantiles")
```

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)
> retp <- rutils::diffit(closep)
> colnames(retp) <- "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
> acfl <- rutils::plot_acf(as.numeric(retp), lag=10,
+       xlab="lag", ylab="Autocorrelation", main="")
> title("Autocorrelations of Minutely SPY Returns", line=1)
> # Calculate the sum of autocorrelations
> sum(acfl$acf)
```



Autocorrelations as Function of Aggregation Interval

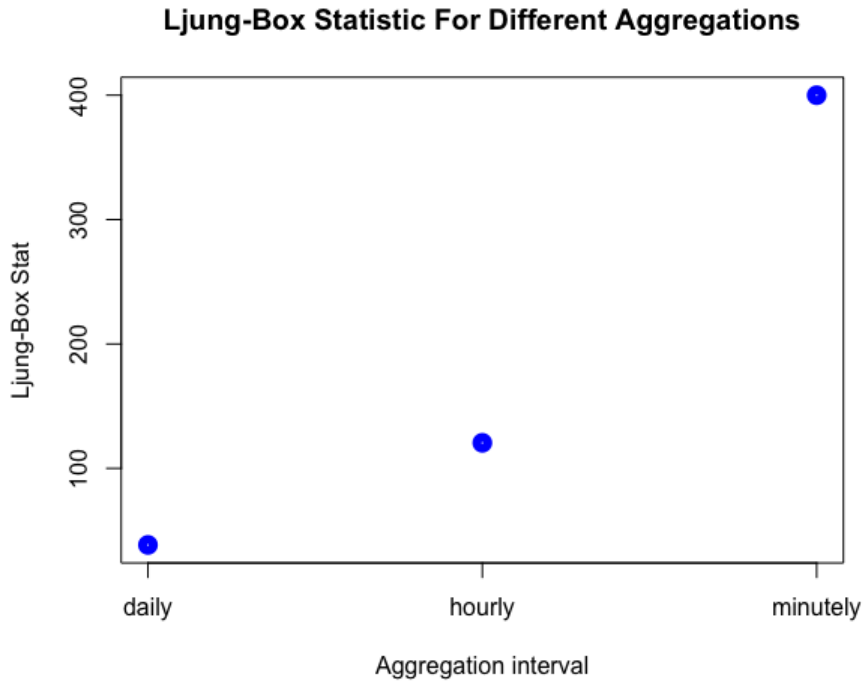
For *minutely SPY* returns, the *Ljung-Box* statistic is large and its *p*-value is very small, so we can conclude that *minutely SPY* returns have statistically significant autocorrelations.

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(retp, lag=10, type="Ljung")
> # Calculate hourly SPY percentage returns
> closeh <- quantmod::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 <- quantmod::Cl(xts::to.period(x=ohlc, period="days"))
> retd <- rutils::diffit(closed)
> # Ljung-Box test for daily SPY returns
> Box.test(retd, lag=10, type="Ljung")
```

```
> # Ljung-Box test statistics for aggregated SPY returns
> lbstat <- sapply(list(daily=retd, hourly=retsh, minutely=retp),
+   function(rets) {
+     Box.test(rets, lag=10, type="Ljung")$statistic
+ })  # end sapply
> # Plot Ljung-Box test statistic for different aggregation interval
> plot(lbstat, lwd=6, col="blue", xaxt="n",
+      xlab="Aggregation interval", ylab="Ljung-Box Stat",
+      main="Ljung-Box Statistic For Different Aggregations")
> # Add X-axis with labels
> axis(side=1, at=(1:3), labels=c("daily", "hourly", "minutely"))
```

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

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

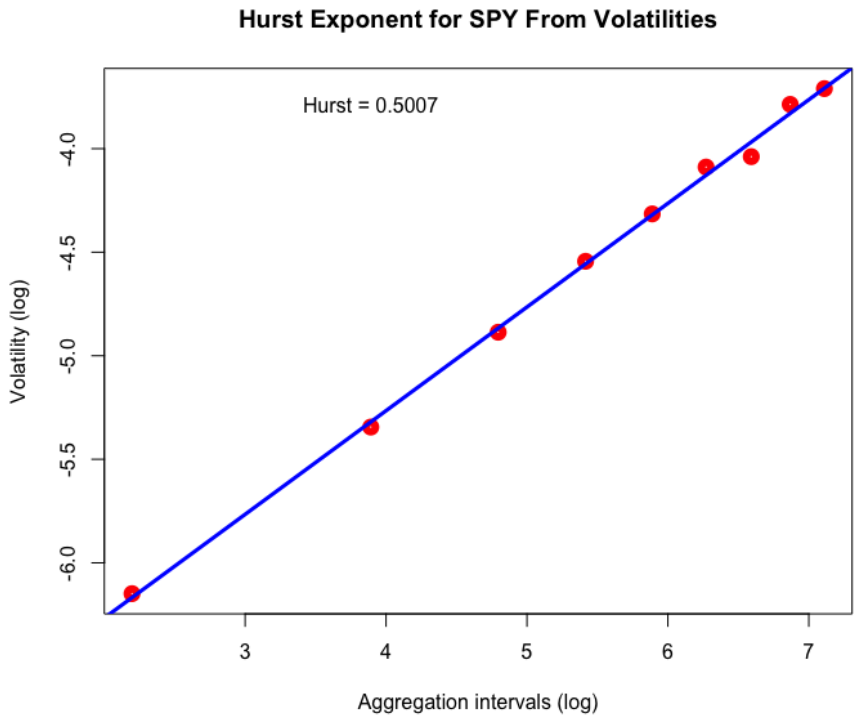
For a single aggregation interval, the *Hurst exponent* `H` is equal to:

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

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

$$H = \frac{\text{cov}(\log \sigma_t, \log \Delta t)}{\text{var}(\log \Delta t)}$$

```
> # Calculate volatilities for vector of aggregation intervals
> aggv <- seq.int(from=3, to=35, length.out=9)^2
> volv <- sapply(aggv, function(agg) {
+   naggs <- nrows %/% agg
+   endd <- c(0, nrows - naggs*agg + (0:naggs)*agg)
+   # endd <- rutils::calc_endpoints(closep, interval=agg)
+   sd(rutils::diffit(closep[endd]))
+ })  # 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()
> regmod <- lm(volog ~ agglog)
> coef(regmod)[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(retp)
> nrows <- NROW(closep)
> # Calculate the rescaled range
> agg <- 500
> naggs <- nrows %/% agg
> endd <- c(0, nrows - naggs*agg + (0:naggs)*agg)
> # Or
> # endd <- rutils::calc_endpoints(closep, interval=agg)
> rrange <- sapply(2:NROW(endd), function(np) {
+   indeks <- (endd[np-1]+1):endd[np]
+   diff(range(closep[indeks]))/sd(retp[indeks])
+ })  # end sapply
> mean(rrange)
> # Calculate the Hurst from single data point
> log(mean(rrange))/log(agg)
```

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`:

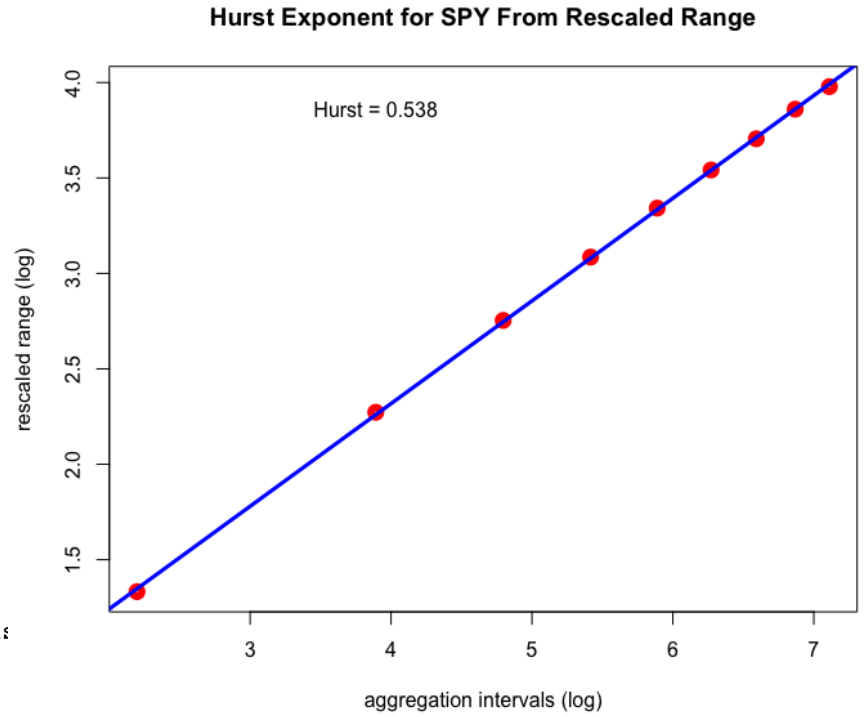
$$RS_{\Delta t} \propto \Delta t^{H}$$

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(agg) {
+ # Calculate the end points
+   naggs <- nrows %/% agg
+   endd <- c(0, nrows - naggs*agg + (0:naggs)*agg)
+ # Calculate the rescaled ranges
+   rrange <- sapply(2:NROW(endd), function(np) {
+     indeks <- (endd[np-1]+1):endd[np]
+     diff(range(closep[indeks]))/sd(retp[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()
> regmod <- lm(rangelog ~ agglog)
> coef(regmod)[2]
```



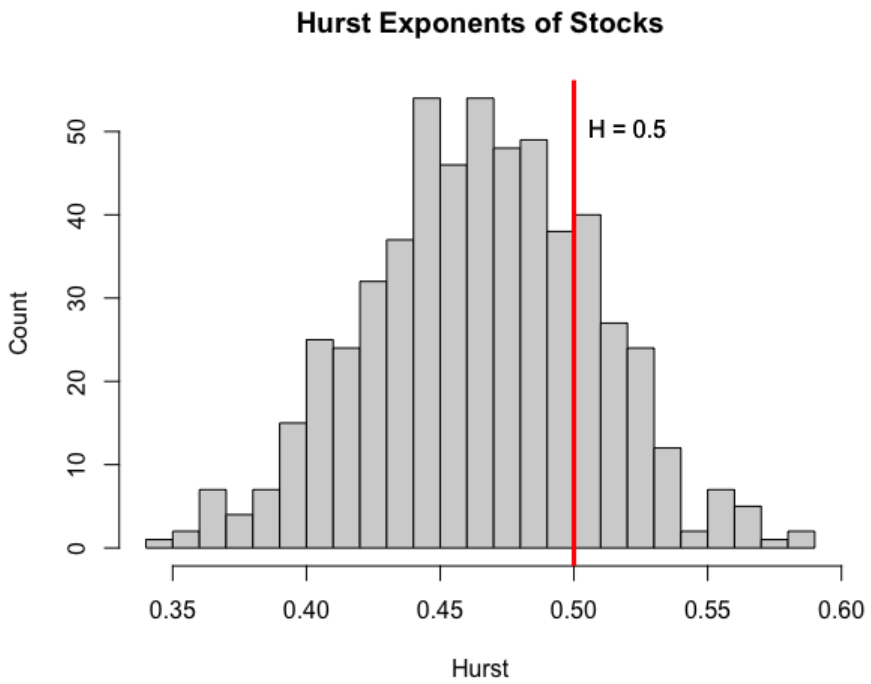

```
> plot(rangelog ~ agglog, lwd=6, col="red",
+      xlab="aggregation intervals (log)",
+      ylab="rescaled range (log)",
+      main="Hurst Exponent for SPY From Rescaled Range")
> abline(model, lwd=3, col="blue")
> text(agglog[2], rangelog[NROW(rangelog)-1],
+      paste0("Hurst = ", round(hurstexp,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

draft: Hurst Exponents of Fractional Brownian Motion

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)
```



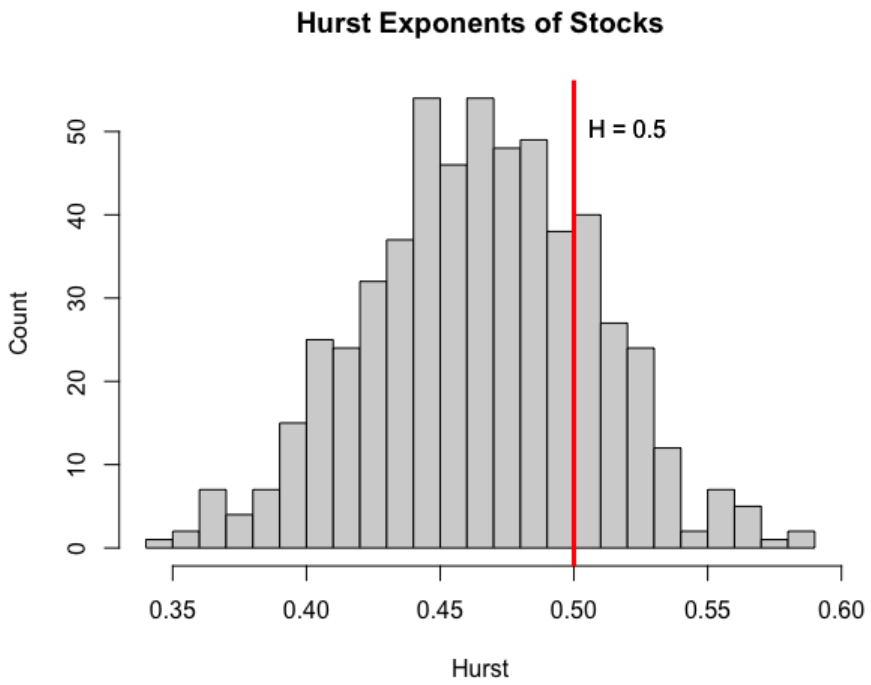

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=20, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of Stocks")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

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)
```



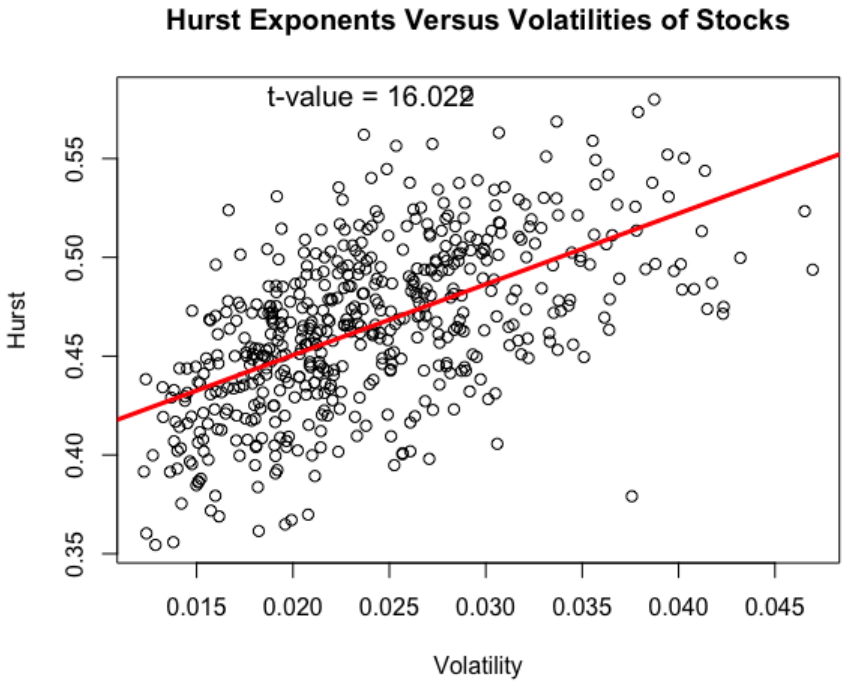

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=20, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of Stocks")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

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



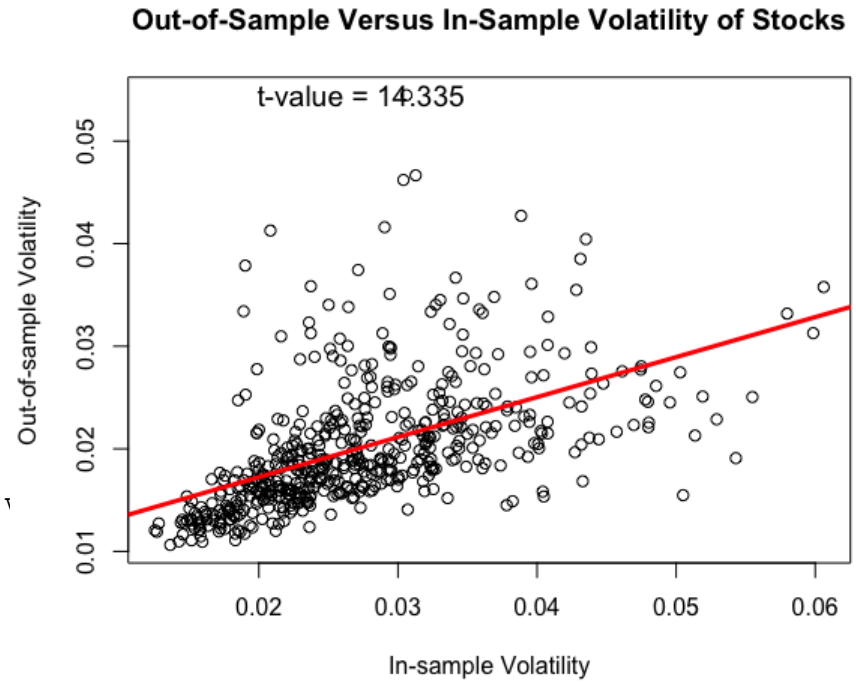

```
> # Plot scatterplot of the Hurst exponents versus volatilities
> plot(hurstv ~ volv, xlab="Volatility", ylab="Hurst",
+      main="Hurst Exponents Versus Volatilities of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(regmod)$coefficients[2, "t value"]
> tvalue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



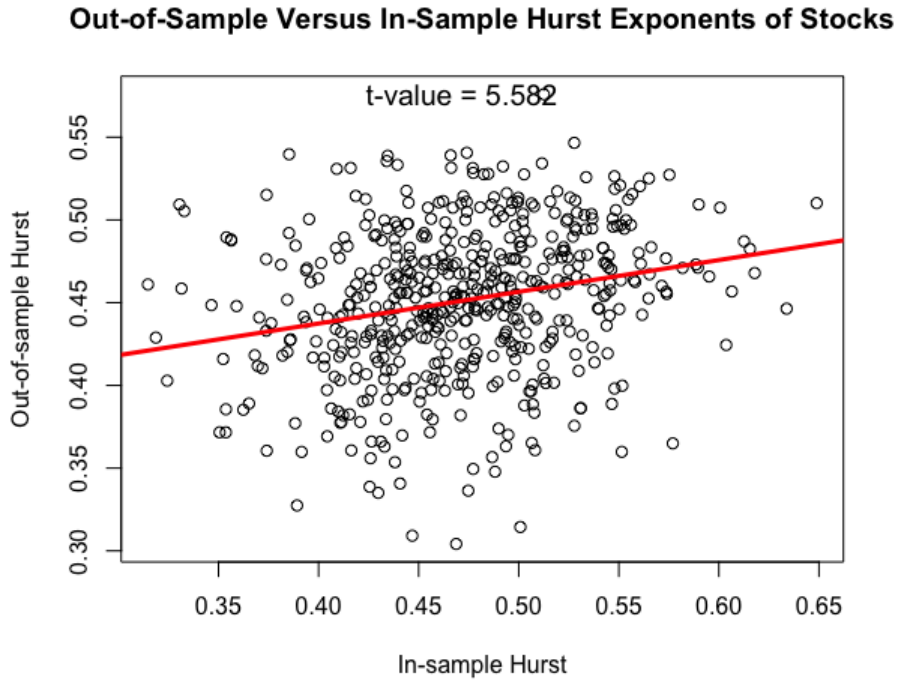

```
> # Plot scatterplot of the out-of-sample versus in-sample volatilit
> plot(volatos ~ volatis, xlab="In-sample Volatility", ylab="Out-of-
+     main="Out-of-Sample Versus In-Sample Volatility of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(regmod)$coefficients[2, "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
> regmod <- lm(hurstos ~ hurstis)
> summary(regmod)
```

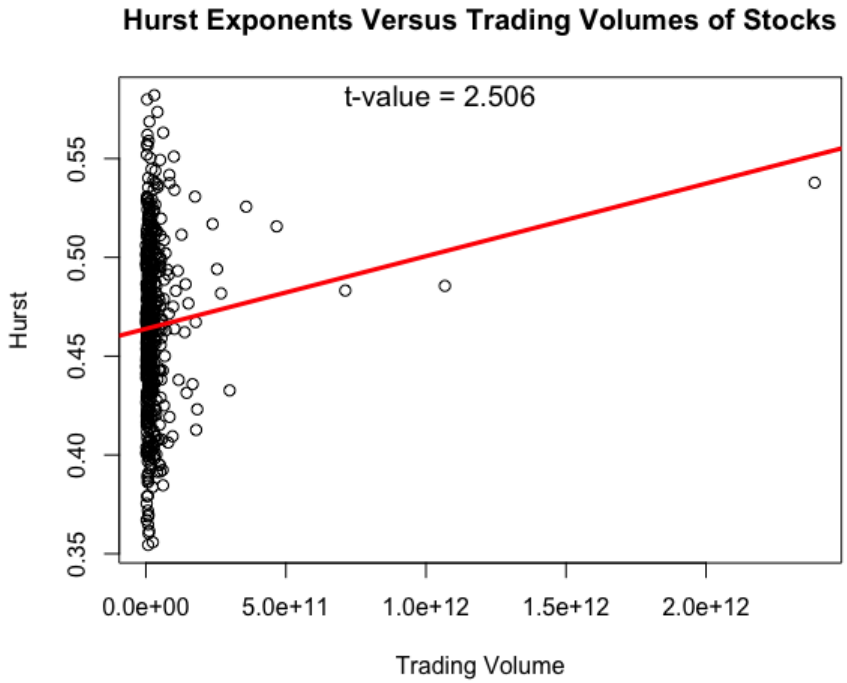
```
> # Plot scatterplot of the out-of-sample versus in-sample Hurst exp
> plot(hurstos ~ hurstis, xlab="In-sample Hurst", ylab="Out-of-sampl
+      main="Out-of-Sample Versus In-Sample Hurst Exponents of Stock
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(regmod)$coefficients[2, "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
> volum <- eapply(sp500env, function(ohlc) {
+     sum(quantmod::Vo(ohlc)["2000/"])
+ })  # end eapply
> # Remove NULL values
> volum <- volum[names(pricev)]
> volum <- unlist(volum)
> which.max(volum)
> # Calculate the number of NULL prices
> sum(is.null(volum))
> # Calculate the Hurst exponents of stocks
> hurstv <- sapply(pricev, HighFreq::calc_hurst, aggv=aggv)
> # Calculate the regression of the Hurst exponents versus trading volumes
> regmod <- lm(hurstv ~ volum)
> summary(regmod)
```



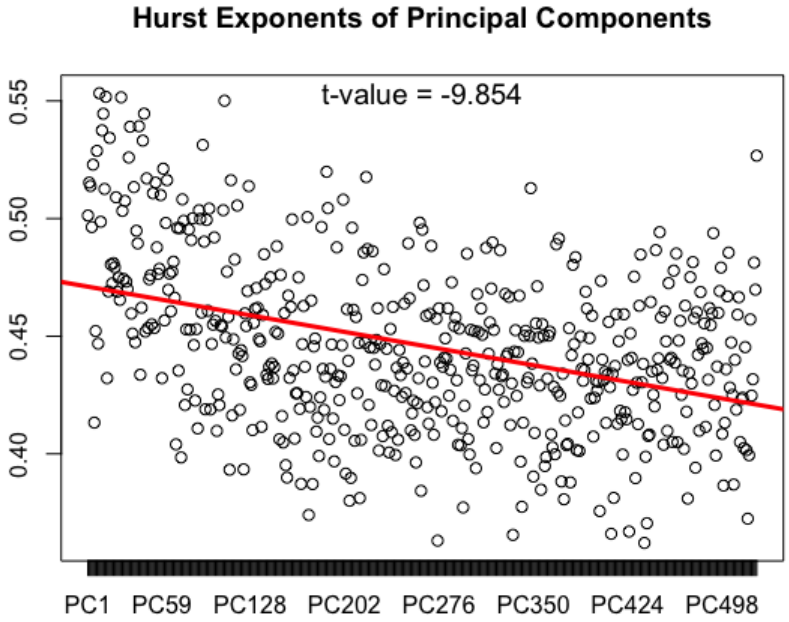

```
> # Plot scatterplot of the Hurst exponents versus trading volumes
> plot(hurstv ~ volum, xlab="Trading Volume", ylab="Hurst",
+      main="Hurst Exponents Versus Trading Volumes of Stocks")
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(regmod)$coefficients[2, "t value"]
> tvalue <- round(tvalue, 3)
> text(x=quantile(volum, 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
> retp <- lapply(pricev, rutils::diffit)
> retp <- rutils::do_call(cbind, retp)
> retp[is.na(retp)] <- 0
> sum(is.na(retp))
> # Drop ".Close" from column names
> colnames(retp[, 1:4])
> colnames(retp) <- rutils::get_name(colnames(retp))
> # Calculate PCA prices using matrix algebra
> eigend <- eigen(cor(retp))
> retpca <- retp %*% eigend$vectors
> pricepca <- xts::xts(matrixStats::colCumsums(retpca),
+                      order.by=index(retp))
> colnames(pricepca) <- paste0("PC", 1:NCOL(retp))
> # 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)
```

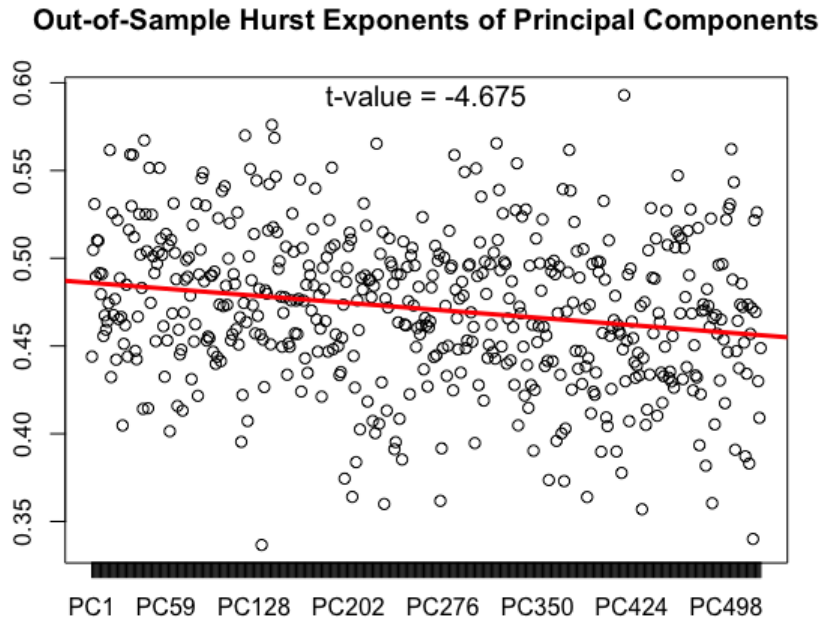
```
> # Plot the Hurst exponents of principal components without x-axis
> plot(hurstv, xlab=NA, ylab=NA, xaxt="n",
+      main="Hurst Exponents of Principal Components")
> # Add X-axis with PCA labels
> axis(side=1, at=(1:NROW(hurstv)), labels=names(hurstv))
> # Calculate the regression of the PCA Hurst exponents versus thei
> orderv <- 1:NROW(hurstv)
> regmod <- lm(hurstv ~ orderv)
> summary(regmod)
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(regmod)$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(retp["/2010"]))
> # Calculate out-of-sample PCA prices
> retpca <- retp["2010/"] %*% eigend$vectors
> pricepca <- xts::xts(matrixStats::colCumsums(retpca),
+                      order.by=index(retp["2010/"]))
> colnames(pricepca) <- paste0("PC", 1:NCOL(retp))
> # 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)
```



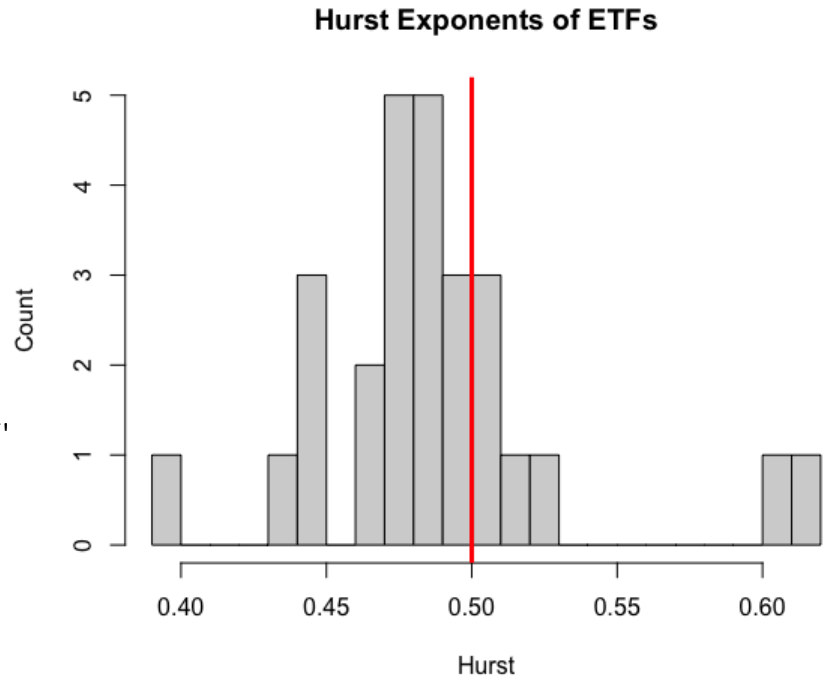

```
> # Plot the Hurst exponents of principal components without x-axis
> plot(hurstv, xlab=NA, ylab=NA, xaxt="n",
+      main="Out-of-Sample Hurst Exponents of Principal Components")
> # Add X-axis with PCA labels
> axis(side=1, at=(1:NROW(hurstv)), labels=names(hurstv))
> # Calculate the regression of the PCA Hurst exponents versus their
> regmod <- lm(hurstv ~ orderv)
> summary(regmod)
> # Add regression line
> abline(model, col='red', lwd=3)
> tvalue <- summary(regmod)$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$symbolv
> symbolv <- symbolv[!(symbolv %in% c("MTUM", "QUAL", "VLUE", "USMV'
> pricev <- lapply(rutils::etfenv$prices[, symbolv], function(x) {
+   log(na.omit(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(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)
```



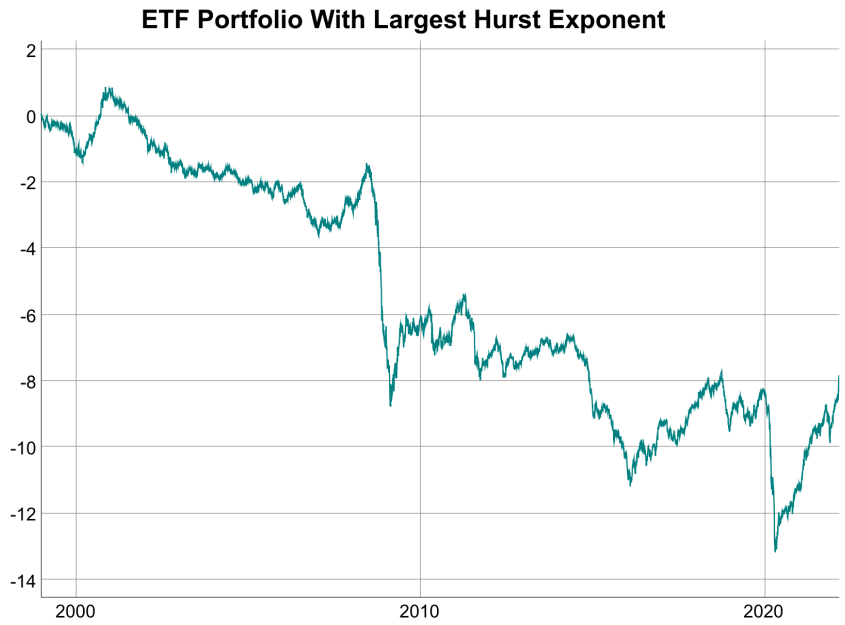

```
> # Plot a histogram of the Hurst exponents of stocks
> hist(hurstv, breaks=2e1, xlab="Hurst", ylab="Count",
+      main="Hurst Exponents of ETFs")
> # Add vertical line for H = 0.5
> abline(v=0.5, lwd=3, col='red')
> text(x=0.5, y=50, lab="H = 0.5", pos=4)
```

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




```
> # Dygraph of ETF portfolio with largest Hurst exponent
> wealthv <- xts::xts(cumsum(retp %*% weightv), zoo::index(retp))
> 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, retp=retp["/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(retp)
> # Calculate the in-sample Hurst exponent
> -calc_phurst(weightv, retp=retp["/2010"])
> # Calculate the out-of-sample Hurst exponent
> -calc_phurst(weightv, retp=retp["2010/"])
```

Autoregressive Processes

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

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$$

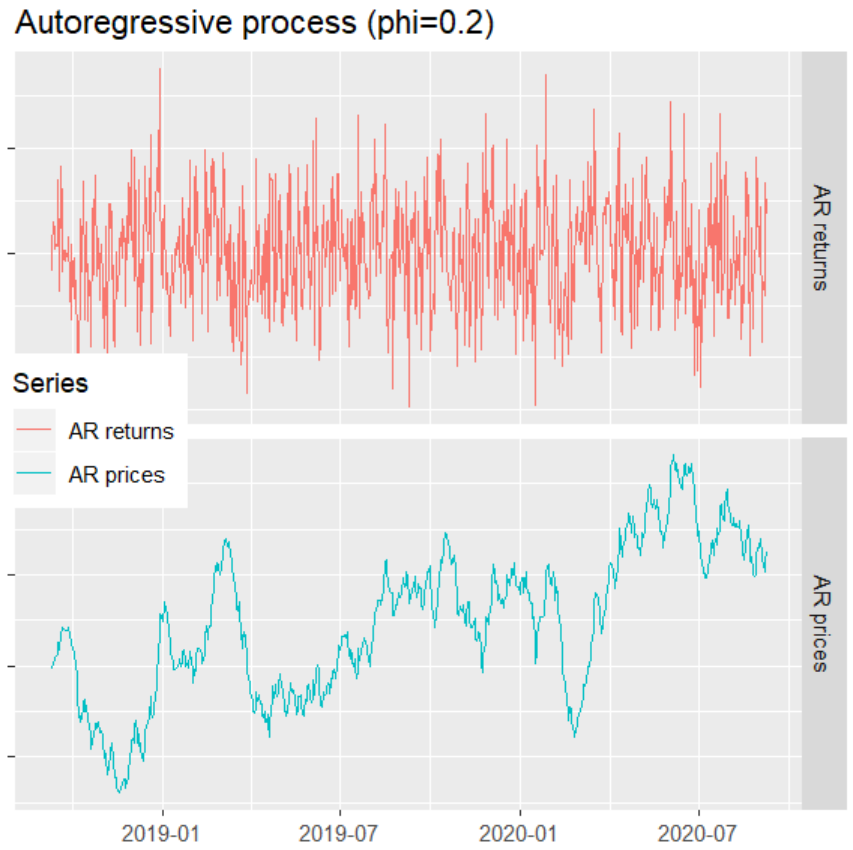
Where φ_i are the *AR(n)* coefficients, and ξ_t are standard normal *innovations*.

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_t 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, "Mersenne-Twister", sample.kind="Rejection")  # Res
> datev <- Sys.Date() + 0:728  # Two year daily series
> # AR time series of returns
> arimav <- xts(x=arima.sim(n=NROW(datev), model=list(ar=0.2)),
+           order.by=datev)
> arimav <- cbind(arimav, cumsum(arimav))
> colnames(arimav) <- c("AR returns", "AR prices")
```



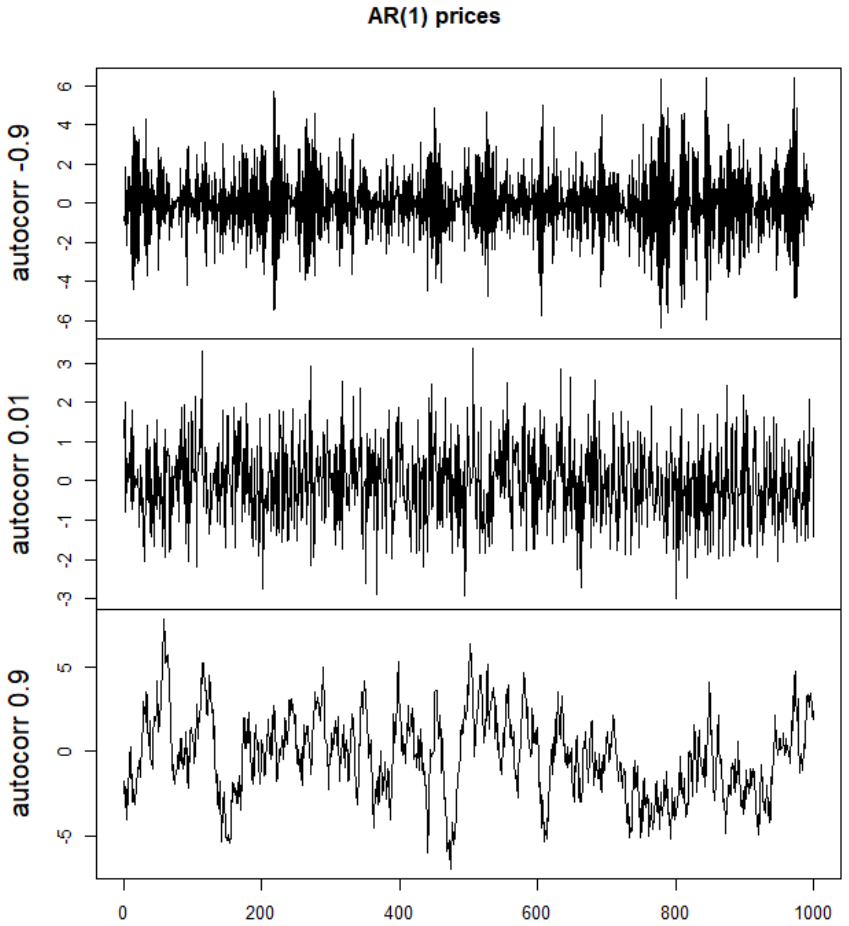

```
> library(ggplot2)  # Load ggplot2
> library(gridExtra)  # Load gridExtra
> autoplot(object=arimav, # ggplot AR process
+  facets="Series ~ .",
+  main="Autoregressive process (phi=0.2)") +
+   facet_grid("Series ~ .", scales="free_y") +
+   xlab("") + ylab("") +
+ theme(legend.position=c(0.1, 0.5),
+   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_t 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, "Mersenne-Twister", sample.kind="Rejection")  #
+   arima.sim(n=NROW(datev), 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=datev)
> 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)*:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$$

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"`.

The function `HighFreq::sim_ar()` simulates an *AR(n)* processes using C++ code.

```
> # Define AR(3) coefficients and innovations
> coeff <- c(0.1, 0.39, 0.5)
> nrows <- 1e2
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection"); innov
> # 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()
> arimaf <- filter(x=innov, filter=coeff, method="recursive")
> class(arimaf)
> all.equal(arimav, as.numeric(arimaf))
> # 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)
> # Fastest simulation of AR process using HighFreq::sim_ar()
> arimav <- HighFreq::sim_ar(coeff=matrix(coeff), innov=matrix(innov
> arimav <- drop(arimav)
> all.equal(arimav, arimacpp)
> # Benchmark the speed of the three methods of simulating AR proces
> library(microbenchmark)
> summary(microbenchmark(
+   Rloop={for (it in 4:NROW(arimav)) {
+     arimav[it] <- arimav[(it-1):(it-3)] %*% coeff + innov[it]
+   }},
+   filter=filter(x=innov, filter=coeff, method="recursive"),
+   cpp=HighFreq::sim_ar(coeff=matrix(coeff), innov=matrix(innov))
+   ), times=10)[, c(1, 4, 5)]
```

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, "Mersenne-Twister", sample.kind="Rejection")
> 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()
> arimaf <- filter(x=innov, filter=coeff, method="recursive")
> all.equal(arimaf[-(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 $r_t = \varphi r_{t-1} + \xi_t$), satisfies the recursive equation: $\rho_i = \varphi \rho_{i-1}$, with $\rho_1 = \varphi$.

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

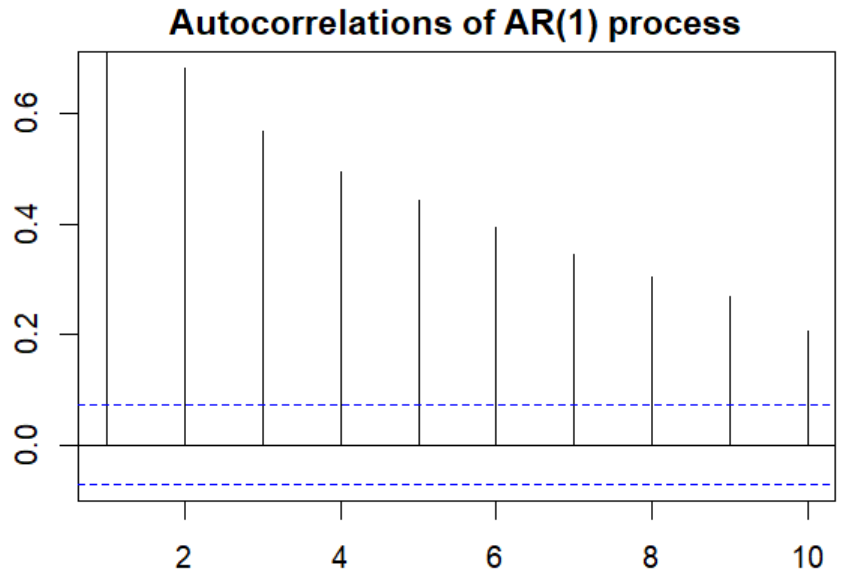
The *AR(1)* process can be simulated recursively:

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

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

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

If $\varphi = 1.0$ then the influence of the random innovations ξ_t persists indefinitely, so that the variance of r_t is proportional to 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
> acfl <- rutils::plot_acf(arimav, lag=10, xlab="", ylab="",
+   main="Autocorrelations of AR(1) process")
> acfl$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.

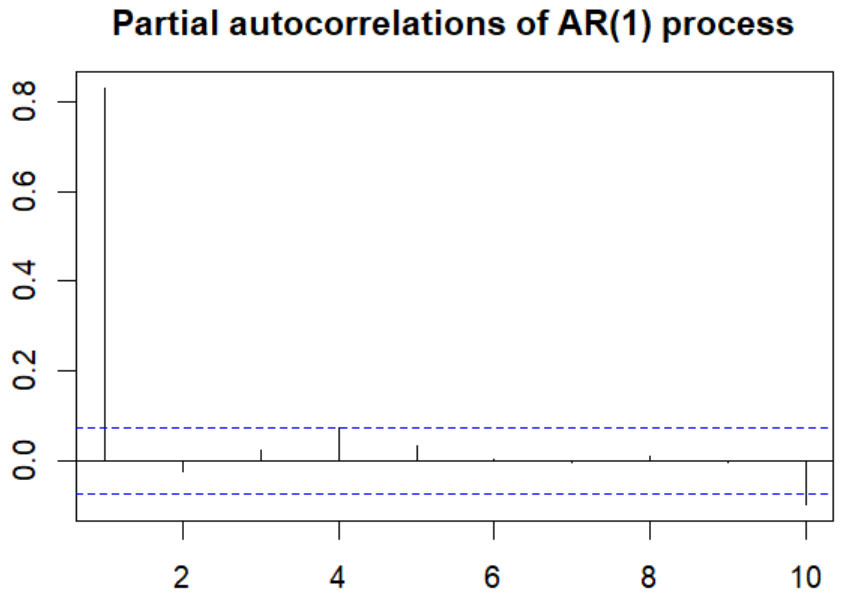
A linear combination of the time series and its own lag can be created, such that its lag 1 autocorrelation is zero.

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

draft: Higher Order Autocorrelations

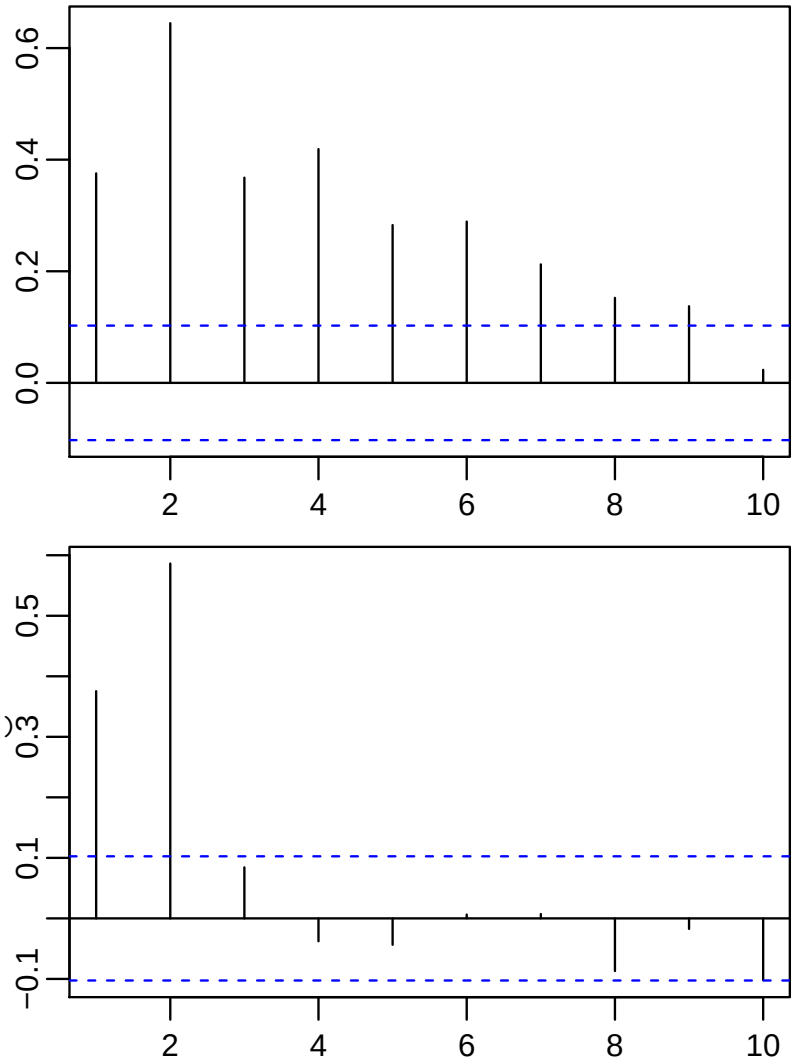
An *AR(3)* process of order *three* is defined by the formula:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \varphi_3 r_{i-3} + \xi_t$$

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)*:
$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$

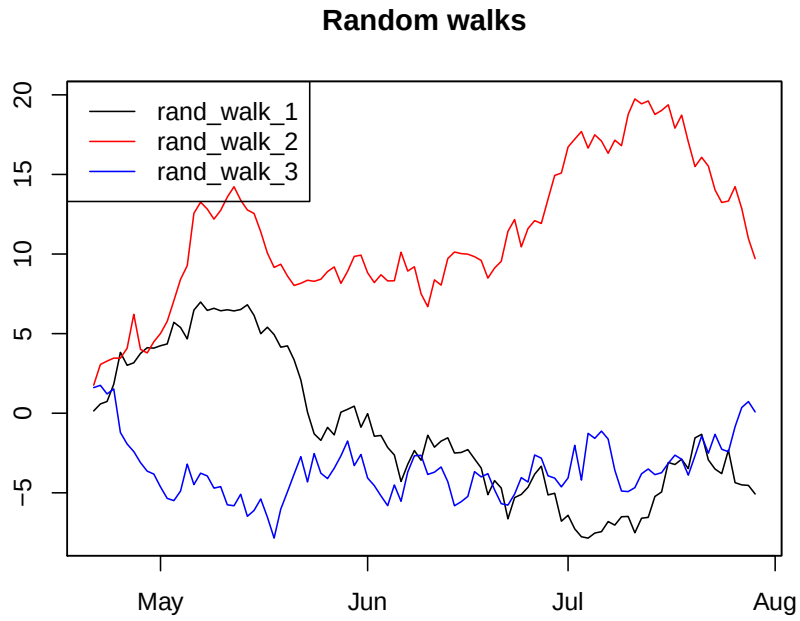
Has the following characteristic equation:
$1 - \varphi_1 z - \varphi_2 z^2 - \ldots - \varphi_n z^n = 0$

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*: $p_t = p_{t-1} + \xi_t$




```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> randw <- cumsum(zoo(matrix(rnorm(3*100), ncol=3),
+ order.by=(Sys.Date()+0:99)))
> colnames(randw) <- paste("randw", 1:3, sep="_")
> plot.zoo(randw, main="Random walks",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue"))
> # Add legend
> legend(x="topleft", legend=colnames(randw),
+  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: $p_t = \sum_{i=1}^{t} r_i$.

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

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

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:
$r_t = \varphi r_{t-1} + \xi_t$.

Then asset prices follow the process:
$p_t = (1 + \varphi)p_{t-1} - \varphi p_{t-2} + \xi_t$

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, "Mersenne-Twister", sample.kind="Rejection")
> 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, "Mersenne-Twister", sample.kind="Rejection")
> 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: $r_t = \varphi r_{t-1} + \xi_t$ 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: $r_t = r_{t-1} + \xi_t$

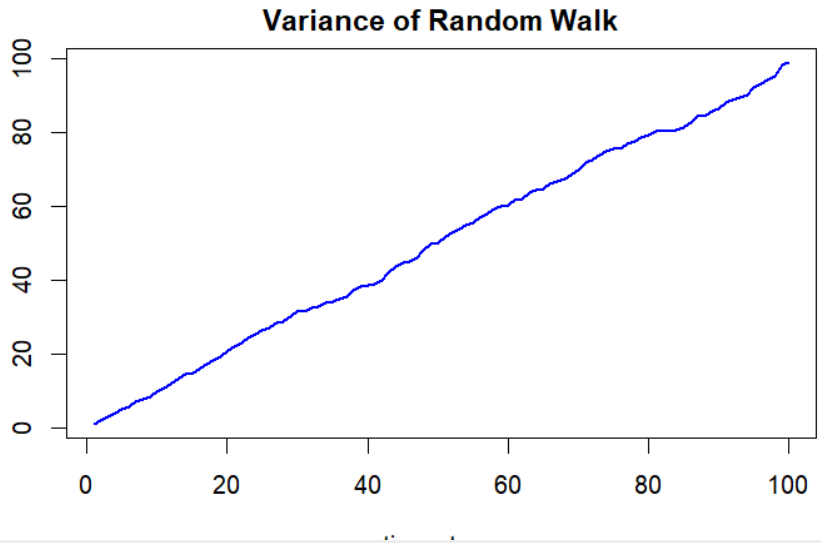
The above is called a *Brownian Motion*, and it's an example of a *unit root* process.

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

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 is not *stationary*.

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




```
> # Simulate random walks using apply() loops
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> randws <- matrix(rnorm(1000*100), ncol=1000)
> randws <- apply(randws, 2, cumsum)
> varv <- apply(randws, 1, var)
> # Simulate random walks using vectorized functions
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> randws <- matrixStats::colCumsums(matrix(rnorm(1000*100), ncol=100
> varv <- matrixStats::rowVars(randws)
> par(mar=c(5, 3, 2, 2), oma=c(0, 0, 0, 0))
> plot(varv, 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_t are equal to the random *innovations*:

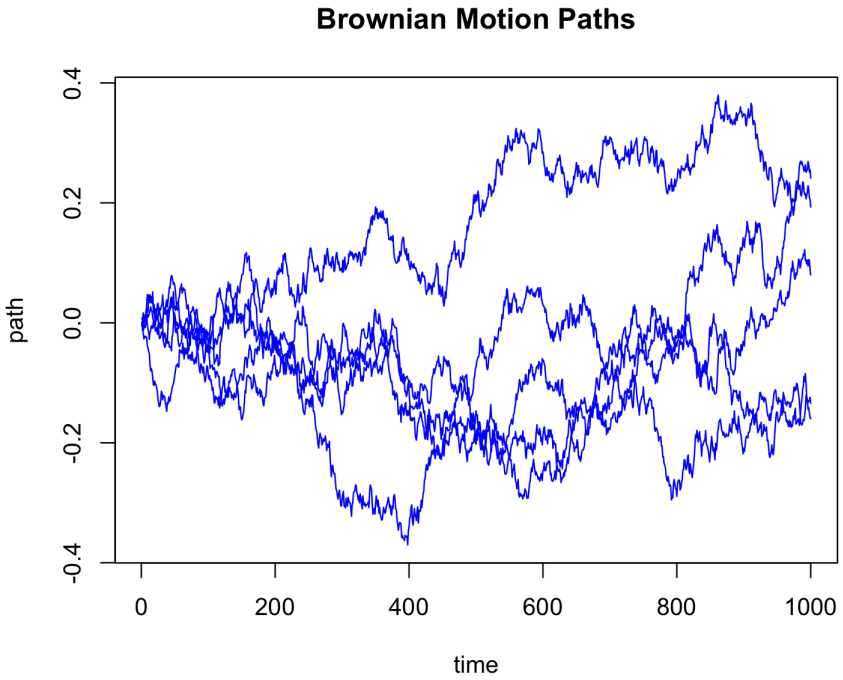
$$r_t = p_t - p_{t-1} = \sigma \, \xi_t$$

$$p_t = p_{t-1} + r_t$$

Where σ is the volatility of returns, and ξ_t 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 $\varphi = 1$:

$$p_t = \varphi p_{t-1} + \sigma \, \xi_t$$




```
> # 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)
> # Plot 5 paths of Brownian motion
> matplot(y=pricev, main="Brownian Motion Paths",
+   xlab="time", ylab="path",
+   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_t are equal to the difference between the equilibrium price μ minus the latest price p_{t-1}, times the mean reversion parameter θ, plus random *innovations*:

$$r_t = p_t - p_{t-1} = \theta\,(\mu - p_{t-1}) + \sigma\,\xi_t$$

$$p_t = p_{t-1} + r_t$$

Where σ is the volatility of returns, and ξ_t 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:

$$p_t = \theta\,\mu + (1 - \theta)\,p_{t-1} + \sigma\,\xi_t$$

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
> prici <- 0.0; priceq <- 1.0;
> sigmav <- 0.02; thetav <- 0.01; nrows <- 1000
> # Initialize the data
> innov <- rnorm(nrows)
> retp <- numeric(nrows)
> pricev <- numeric(nrows)
> retp[1] <- sigmav*innov[1]
> pricev[1] <- prici
> # Simulate Ornstein-Uhlenbeck process in R
> for (i in 2:nrows) {
+   retp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+   pricev[i] <- pricev[i-1] + retp[i]
+ }  # end for
> # Simulate Ornstein-Uhlenbeck process in Rcpp
> pricecpp <- HighFreq::sim_ou(prici=prici, priceq=priceq,
+   theta=thetav, innov=matrix(sigmav*innov))
> all.equal(pricev, drop(pricecpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (i in 2:nrows) {
+     retp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+     pricev[i] <- pricev[i-1] + retp[i]}},
+   Rcpp=HighFreq::sim_ou(prici=prici, priceq=priceq,
+     theta=thetav, innov=matrix(sigmav*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:

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

Where B_t is a *Brownian Motion*, with $\mathrm{d}B_t$ following the normal distribution $\phi(0, \sqrt{\mathrm{d}t})$, with the volatility $\sqrt{\mathrm{d}t}$, equal to the square root of the time increment $\mathrm{d}t$.

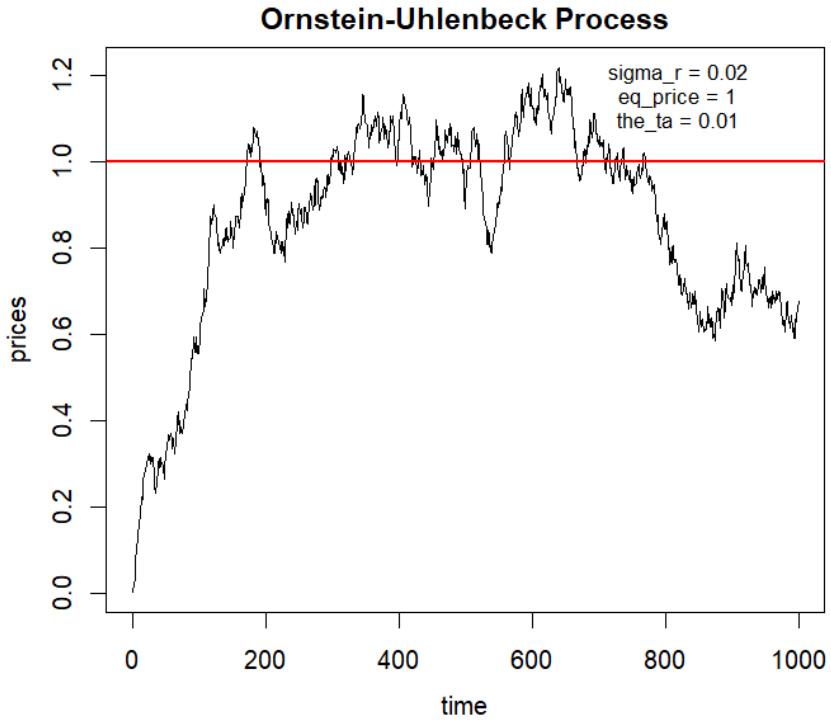
The solution of the *Ornstein-Uhlenbeck* process is given by:

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

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.

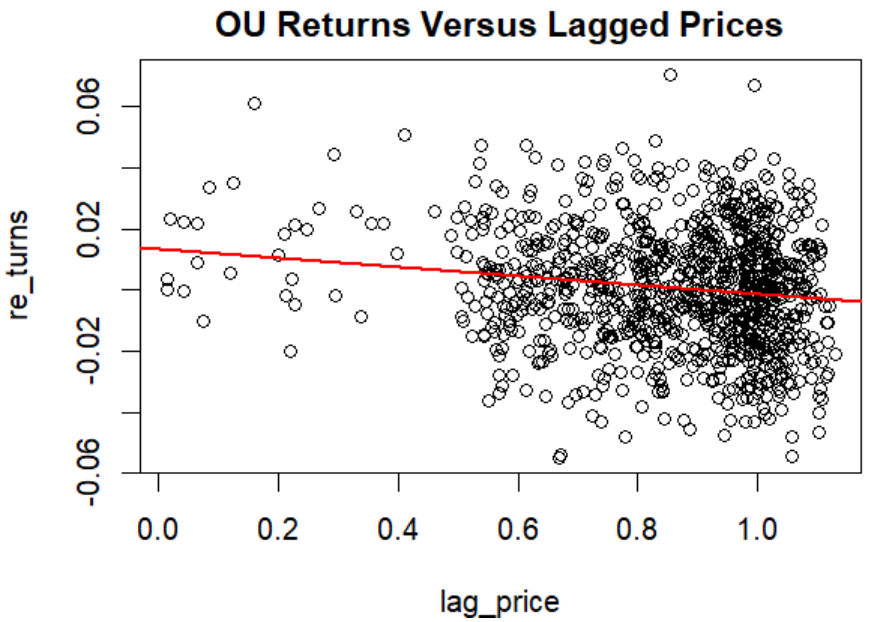



```
> plot(pricev, type="l", xlab="time", ylab="prices",
+      main="Ornstein-Uhlenbeck Process")
> legend("topright", title=paste(c(paste0("sigmav = ", sigmav),
+      paste0("priceq = ", ),
+      paste0("thetav = ", thetav)),
+    collapse="\n"),
+  legend="", cex=0.8, inset=0.1, bg="white", bty="n")
> 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.

```
> retp <- rutils::diffit(pricev)
> pricelag <- rutils::lagit(pricev)
> formulav <- retp ~ 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(retp))
> # Extract OU parameters from regression
> coeff <- summary(regmod)$coefficients
> # Calculate regression alpha and beta directly
> betac <- cov(retp, pricelag)/var(pricelag)
> alphac <- (mean(retp) - betac*mean(pricelag))
> cbind(direct=c(alpha=alphac, beta=betac), lm=coeff[, 1])
> all.equal(c(alpha=alphac, beta=betac), coeff[, 1],
+     check.attributes=FALSE)
> # Calculate regression standard errors directly
> betac <- c(alpha=alphac, beta=betac)
> fitv <- (alphac + betac*pricelag)
> resids <- (retp - fitv)
> prices2 <- sum((pricelag - mean(pricelag))^2)
> betasd <- sqrt(sum(resids^2)/prices2/(nrows-2))
> alphasd <- sqrt(sum(resids^2)/(nrows-2)*(1:nrows + mean(pricelag)
> 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.

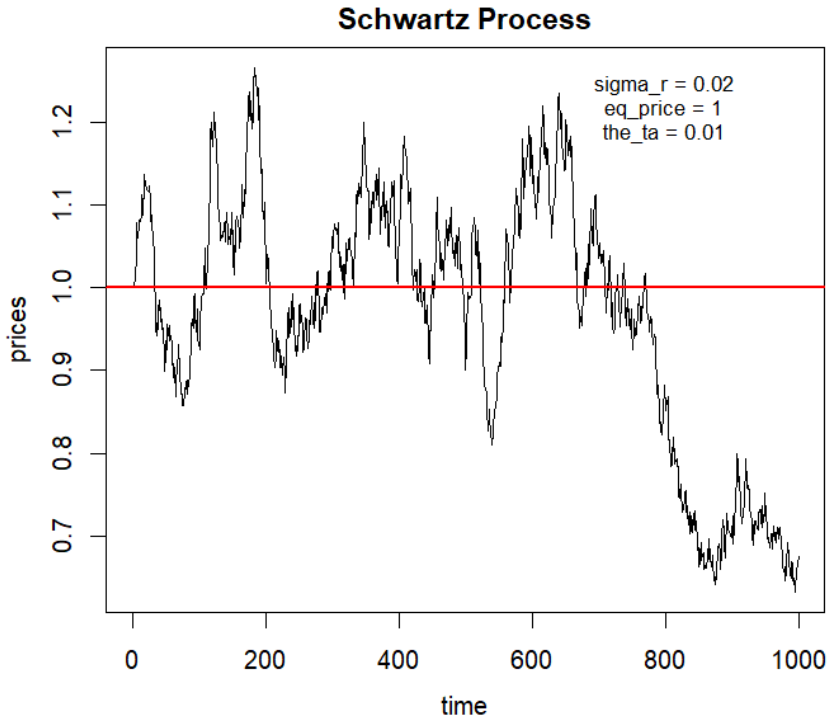
So the *Ornstein-Uhlenbeck* process is better suited for simulating the logarithm of prices, which can be negative.

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

$$r_t = \log p_t - \log p_{t-1} = \theta\,(\mu - p_{t-1}) + \sigma\,\xi_t$$

$$p_t = p_{t-1}\exp(r_t)$$

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




```
> # Simulate Schwartz process
> retp <- numeric(nrows)
> pricev <- numeric(nrows)
> pricev[1] <- exp(sigmav*innov[1])
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # Re
> for (i in 2:nrows) {
+   retp[i] <- thetav*(priceq - pricev[i-1]) + sigmav*innov[i]
+   pricev[i] <- pricev[i-1]*exp(retp[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_t are equal to the sum of a mean reverting term plus *autoregressive* terms:

$$r_t = \theta(\mu - p_{t-1}) + \varphi_1 r_{t-1} + \ldots + \varphi_n r_{t-n} + \sigma \xi_t$$

$$p_t = p_{t-1} + r_t$$

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

Then the prices follow an *autoregressive* process:

$$p_t = \theta\mu + (1 + \varphi_1 - \theta)p_{t-1} + (\varphi_2 - \varphi_1)p_{t-2} + \ldots + (\varphi_n - \varphi_{n-1})p_{t-n} - \varphi_n p_{t-n-1} + \sigma \xi_t$$

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_t exhibits mean reversion and has no *unit root*.

```
> # Define Dickey-Fuller parameters
> prici <- 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)
> retp <- numeric(nrows)
> pricev <- numeric(nrows)
> # Simulate Dickey-Fuller process using recursive loop in R
> retp[1] <- innov[1]
> pricev[1] <- prici
> retp[2] <- thetav*(priceq - pricev[1]) + coeff[1]*retp[1] +
+   innov[2]
> pricev[2] <- pricev[1] + retp[2]
> retp[3] <- thetav*(priceq - pricev[2]) + coeff[1]*retp[2] +
+   coeff[2]*retp[1] + innov[3]
> pricev[3] <- pricev[2] + retp[3]
> for (it in 4:nrows) {
+   retp[it] <- thetav*(priceq - pricev[it-1]) +
+     retp[(it-1):(it-3)] %*% coeff + innov[it]
+   pricev[it] <- pricev[it-1] + retp[it]
+ }  # end for
> # Simulate Dickey-Fuller process in Rcpp
> pricecpp <- HighFreq::sim_df(prici=prici, priceq=priceq,
+    theta=thetav, coeff=matrix(coeff), innov=matrix(innov))
> # Compare prices
> all.equal(pricev, drop(pricecpp))
> # Compare the speed of R code with Rcpp
> library(microbenchmark)
> summary(microbenchmark(
+   Rcode={for (it in 4:nrows) {
+   retp[it] <- thetav*(priceq - pricev[it-1]) + retp[(it-1):(it-3)]
+   pricev[it] <- pricev[it-1] + retp[it]
+   }},
+   Rcpp=HighFreq::sim_df(prici=prici, priceq=priceq, theta=thetav,
+   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_t:

$$r_t = \theta(\mu - p_{t-1}) + \varphi_1 r_{t-1} + \ldots + \varphi_n r_{t-n} + \sigma\,\xi_t$$

$$p_t = p_{t-1} + r_t$$

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_t 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 different distribution from the t-distribution).

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

```
> # Simulate AR(1) process with coefficient=1, with unit root
> innov <- matrix(rnorm(1e4, sd=0.01))
> arimav <- HighFreq::sim_ar(coeff=matrix(1), innov=innov)
> plot(arimav, t="l", main="Brownian Motion")
> # 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)
> plot(arimav, t="l", main="AR(1) coefficient = 0.99")
> tseries::adf.test(arimav, k=1)
> # Simulate Ornstein-Uhlenbeck OU process with mean reversion
> prici <- 0.0; priceq <- 0.0; thetav <- 0.1
> pricev <- HighFreq::sim_ou(prici=prici, priceq=priceq,
+   theta=thetav, innov=innov)
> 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(prici=prici, priceq=priceq,
+   theta=thetav, innov=innov)
> 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: $r_t = \theta(\mu - p_{t-1}) + \varepsilon_i$.

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_t exhibits mean reversion:

$$r_t = \theta p_{t-1} + \varphi_1 r_{t-1} + \ldots + \varphi_n r_{t-n} + \xi_t$$

where $p_t = p_{t-1} + r_t$, so that:

$$p_t = (1+\theta) p_{t-1} + \varphi_1 r_{t-1} + \ldots + \varphi_n r_{t-n} + \xi_t$$

If the mean reversion parameter θ is positive: $\theta > 0$, then the time series p_t 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 different distribution 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: $r_t = \theta p_{t-1} + \xi_t$.

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

```
> nrows <- 1e3
> # Perform ADF test for AR(1) with small coefficient
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> 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, "Mersenne-Twister", sample.kind="Rejection")
> 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*.

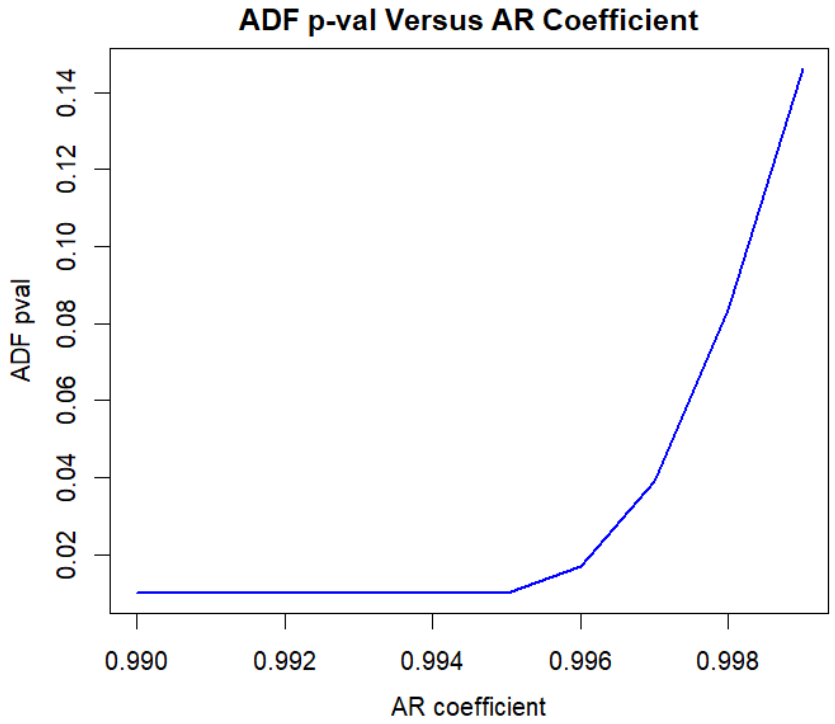
The *ADF* test has low *sensitivity*, i.e. the ability to correctly identify time series with no *unit root*, causing it to produce *false negatives* (*type II* errors).

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)
> retp <- as.numeric(na.omit(rutils::etfenv$returns$VTI))
> adft <- sapply(coeffv, function(coeff) {
+   arimav <- filter(x=retp, 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_t:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t = \sum_{j=1}^{n} \varphi_j r_{t-j} + \xi_t$$

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 `HighFreq::sim_ar()` simulates an *AR(n)* processes using `C++` code.

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, "Mersenne-Twister", sample.kind="Rejection"); innov
> # 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$ar); drop(coeff)
> # Define predictor matrix without intercept column
> predm <- sapply(1:ncoeff, rutils::lagit, input=arimav)
> # Fit AR model using regression
> predinv <- MASS::ginv(predm)
> coeff <- drop(predinv %*% arimav)
> all.equal(drop(arfit$ar), coeff, check.attributes=FALSE)
```

draft: Calibrating Autoregressive Models Using Maximum Likelihood

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

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t = \sum_{j=1}^{n} \varphi_j r_{t-j} + \xi_t$$

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

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, "Mersenne-Twister", sample.kind="Rejection"); innov
> 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))
> predm <- sapply(1:3, rutils::lagit, input=arimav)
> # Calculate centered returns matrix
> predm <- t(t(predm) - colMeans(predm))
> predinv <- MASS::ginv(predm)
> # Regression coefficients with response equal to arimav
> coeff <- drop(predinv %*% arimav)
>
> all.equal(arfit$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 model residuals
> fitv <- drop(predm %*% coeff)
> resids <- drop(arimav - fitv)
> # Variance of residuals
> residsd <- sum(resids^2)/(nrows-NROW(coeff))
> # Inverse of predictor matrix squared
> pred2 <- MASS::ginv(crossprod(predm))
> # Calculate covariance matrix of AR coefficients
> covmat <- residsd*pred2
> coefsd <- sqrt(diag(covmat))
> # Calculate t-values of AR coefficients
> coefft <- drop(coeff)/coefsd
> # Plot the t-values of the AR coefficients
> barplot(coefft, xlab="lag", ylab="t-value",
+   main="Coefficient t-values of AR Forecasting Model")
```

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
> predm <- sapply(1:5, rutils::lagit, input=arimav)
> predinv <- MASS::ginv(predm)
> coeff <- drop(predinv %*% arimav)
> # Calculate t-values of AR(5) coefficients
> resids <- drop(arimav - drop(predm %*% coeff))
> residsd <- sum(resids^2)/(nrows-NROW(coeff))
> covmat <- residsd*MASS::ginv(crossprod(predm))
> coefsd <- sqrt(diag(covmat))
> coefft <- drop(coeff)/coefsd
> # Fit AR(5) model using arima()
> arfit <- arima(arimav, order=c(5, 0, 0), include.mean=FALSE)
> arfit$coef
> # Fit AR(5) model using auto.arima()
> library(forecast)  # Load forecast
> arfit <- forecast::auto.arima(arimav, max.p=5, max.q=0, max.d=0)
> # Fit AR(5) model into VTI returns
> retp <- drop(zoo::coredata(na.omit(rutils::etfenv$returns$VTI)))
> predm <- sapply(1:5, rutils::lagit, input=retp)
> predinv <- MASS::ginv(predm)
> coeff <- drop(predinv %*% retp)
> # Calculate t-values of AR(5) coefficients
> resids <- drop(retp - drop(predm %*% coeff))
> residsd <- sum(resids^2)/(nrows-NROW(coeff))
> covmat <- residsd*MASS::ginv(crossprod(predm))
> coefsd <- sqrt(diag(covmat))
> coefft <- drop(coeff)/coefsd
```

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:

$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t = \sum_{j=1}^{n} \varphi_j r_{t-j} + \xi_t$$

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

The Yule-Walker Equations

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

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

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

If we multiply the *autoregressive* process *AR(n)*: $r_t = \sum_{j=1}^{n} \varphi_j r_{t-j} + \xi_t$, by r_{t-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 can be solved for the *AR(n)* coefficients φ_i using matrix inversion.

```
> # Compute autocorrelation coefficients
> acfl <- rutils::plot_acf(arimav, lag=10, plot=FALSE)
> acfl <- drop(acfl$acf)
> nrows <- NROW(acfl)
> acf1 <- c(1, acfl[-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 %*% acfl)
> 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
> acfl <- rutils::plot_acf(arimav, lag=10, plotobj=FALSE)
> acfl <- drop(acfl$acf)
> nrows <- NROW(acfl)
> pacfl <- numeric(2)
> pacfl[1] <- acfl[1]
> pacfl[2] <- (acfl[2] - acfl[1]^2)/(1 - acfl[1]^2)
> # Calculate PACF recursively in a loop using Durbin-Levinson algor
> pacfll <- matrix(numeric(nrows*nrows), nc=nrows)
> pacfll[1, 1] <- acfl[1]
> for (it in 2:nrows) {
+   pacfll[it, it] <- (acfl[it] - pacfll[it-1, 1:(it-1)] %*% acfl[(i
+   for (it2 in 1:(it-1)) {
+     pacfll[it, it2] <- pacfll[it-1, it2] - pacfll[it, it] %*% pacf
+   }  # end for
+ }  # end for
> pacfll <- diag(pacfll)
> # Compare with the PACF without loop
> all.equal(pacfl, pacfll[1:2])
> # Calculate PACF using pacf()
> pacfl <- pacf(arimav, lag=10, plot=FALSE)
> pacfl <- drop(pacfl$acf)
> all.equal(pacfl, pacfll)
```

Forecasting Autoregressive Processes

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

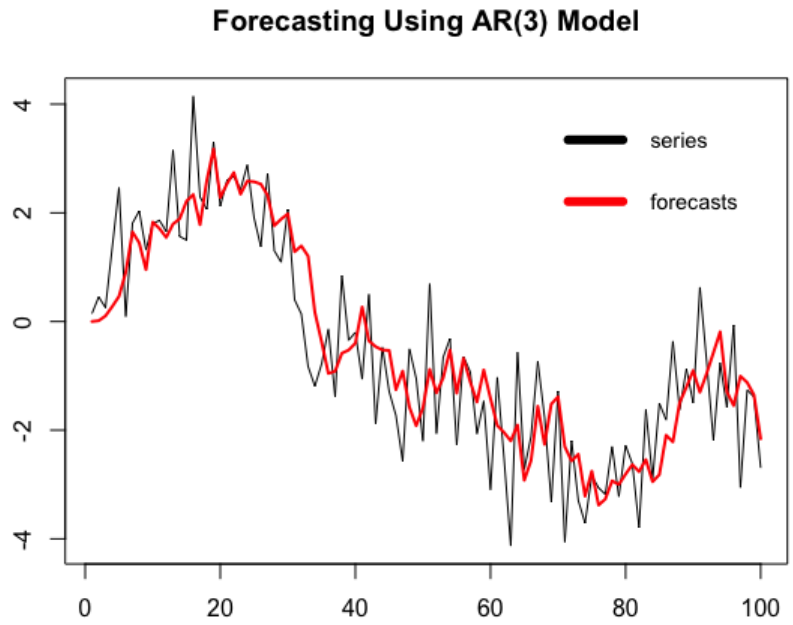
$$r_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$$

Can be simulated using the `HighFreq::sim_ar()`.

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

$$f_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n}$$

```
> # Simulate AR process using HighFreq::sim_ar()
> nrows <- 1e2
> coeff <- matrix(c(0.1, 0.39, 0.5)); ncoeff <- NROW(coeff)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection"); innov
> arimav <- HighFreq::sim_ar(coeff=coeff, innov=innov)
> # Forecast AR process using loop in R
> fcast <- numeric(nrows+1)
> fcast[2] <- coeff[1]*arimav[1]
> fcast[3] <- coeff[1]*arimav[2] + coeff[2]*arimav[1]
> for (it in 3:nrows) {
+   fcast[it+1] <- arimav[it:(it-2)] %*% coeff
+ }  # end for
```




```
> # Plot with legend
> plot(arimav, main="Forecasting Using AR(3) Model",
+   xlab="", ylab="", type="l")
> lines(fcast[-(nrows+1)], col="red", lwd=2)
> legend(x="topright", legend=c("series", "forecasts"),
+  col=c("black", "red"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

Fast Forecasting of Autoregressive Processes

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

$$f_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n}$$

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()
> convf <- filter(x=arimav, sides=1, filter=coeff, method="convoluti
> convf <- as.numeric(convf)
> # Compare excluding warmup period
> all.equal(fcast[-(1:ncoeff)], convf[-(1:(ncoeff-1))],
+ check.attributes=FALSE)
> # Filter using C_cfilter() compiled C++ function directly
> convf <- .Call(stats:::C_cfilter, arimav, filter=coeff,
+                sides=1, circular=FALSE)
> # Compare excluding warmup period
> all.equal(fcast[-(1:ncoeff)], convf[-(1:(ncoeff-1))],
+ check.attributes=FALSE)
> # Filter using HighFreq::roll_conv() Rcpp function
> convf <- HighFreq::roll_conv(arimav, coeff)
> # Compare excluding warmup period
> all.equal(fcast[-(1:ncoeff)], convf[-(1:(ncoeff-1))],
+ check.attributes=FALSE)
> # Define predictor matrix for forecasting
> predm <- sapply(0:(ncoeff-1), function(lagg) {
+   rutils::lagit(arimav, lagg=lagg)
+ })  # end sapply
> # Forecast using predictor matrix
> convf <- c(0, drop(predm %*% coeff))
> # Compare with loop in R
> all.equal(fcast, convf, 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()
> arfit <- arima(arimav, order=c(3,0,0), include.mean=FALSE)
> arfit$coef
> coeff
> # One-step-ahead forecast using predict.Arima()
> predm <- predict(arfit, n.ahead=1)
> # Or directly call predict.Arima()
> # predm <- predict.Arima(arfit, n.ahead=1)
> # Inspect the prediction object
> class(predm)
> names(predm)
> class(predm$pred)
> unlist(predm)
> # One-step-ahead forecast using matrix algebra
> fcast1 <- drop(arimav[nrows:(nrows-2)] %*% arfit$coef)
> # Compare one-step-ahead forecasts
> all.equal(predm$pred[[1]], fcast1)
> # 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_t minus their *forecasts* f_t: $\varepsilon_i = r_t - f_t$.

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_t: $\varepsilon_i = r_t - f_t = \xi_t$.

The forecasts have a lower volatility than the *AR(n)* process because the convolution procedure averages out the noise.

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

draft: The Standard Errors of Forecasts from Autoregressive Processes

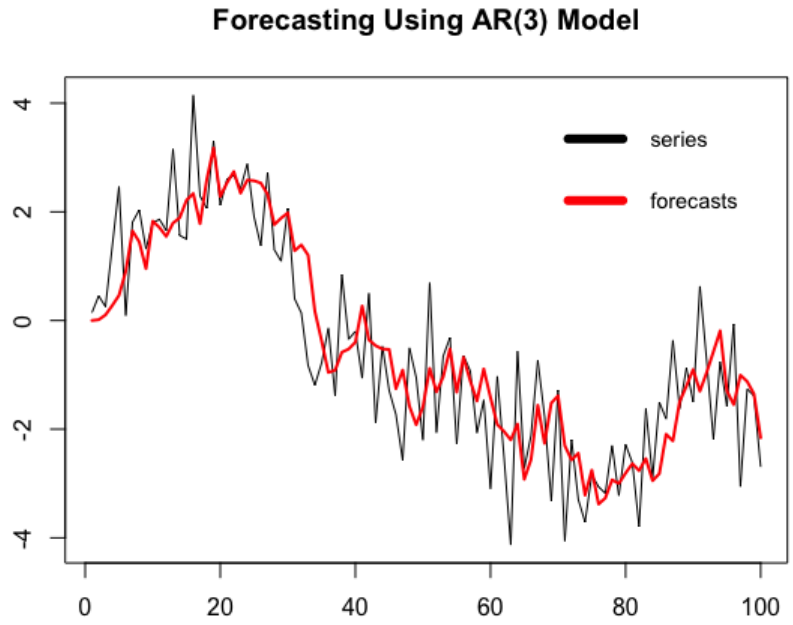
Trivial: The variance of the predicted value is equal to the predictor vector multiplied by the covariance matrix of the regression coefficients.

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

$$f_t = \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n}$$

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, "Mersenne-Twister", sample.kind="Rejection")
> arimav <- filter(x=rnorm(nrows), filter=coeff, method="recursive")
> arimav <- as.numeric(arimav)
> # Forecast AR(3) process
> fcast <- numeric(NROW(arimav))
> fcast[2] <- coeff[1]*arimav[1]
> fcast[3] <- coeff[1]*arimav[2] + coeff[2]*arimav[1]
> for (it in 4:NROW(fcast)) {
+   fcast[it] <- arimav[(it-1):(it-3)] %*% coeff
+ }  # end for
> # Forecast using filter()
> fcastf <- filter(x=arimav, sides=1,
+   filter=coeff, method="convolution")
> class(fcastf)
> all.equal(fcast[-(1:4)],
+   fcastf[-c(1:3, NROW(fcastf))],
+   check.attributes=FALSE)
> # Compare residuals with innovations
> resids <- (arimav-fcast)
> tail(cbind(innov, resids))
>
>
```



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(fcast, col="orange", lwd=3)
> legend(x="topright", legend=c("series", "forecasts"),
+  col=c("black", "orange"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

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 (`lookb`).

```
> # Define AR process parameters
> nrows <- 1e3
> coeff <- matrix(c(0.5, 0.0, 0.0)); ncoeff <- NROW(coeff)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection"); innov
> # Simulate AR process using HighFreq::sim_ar()
> arimav <- HighFreq::sim_ar(coeff=coeff, innov=innov)
> # Define order of the AR(n) forecasting model
> ordern <- 5
> # Define predictor matrix for forecasting
> predm <- sapply(1:ordern, rutils::lagit, input=arimav)
> colnames(predm) <- paste0("pred", 1:NCOL(predm))
> # Specify length of look-back interval
> lookb <- 100
> # Invert the predictor matrix
> rangev <- (nrows-lookb):(nrows-1)
> predinv <- MASS::ginv(predm[rangev, ])
> # Calculate fitted coefficients
> coeff <- drop(predinv %*% arimav[rangev])
> # Calculate forecast
> drop(predm[nrows, ] %*% coeff)
> # Actual value
> arimav[nrows]
```

Rolling Forecasting of Autoregressive Models

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

$$r_t = \varphi_0 + \varphi_1 r_{t-1} + \varphi_2 r_{t-2} + \ldots + \varphi_n r_{t-n} + \xi_t$$

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_t:

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

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

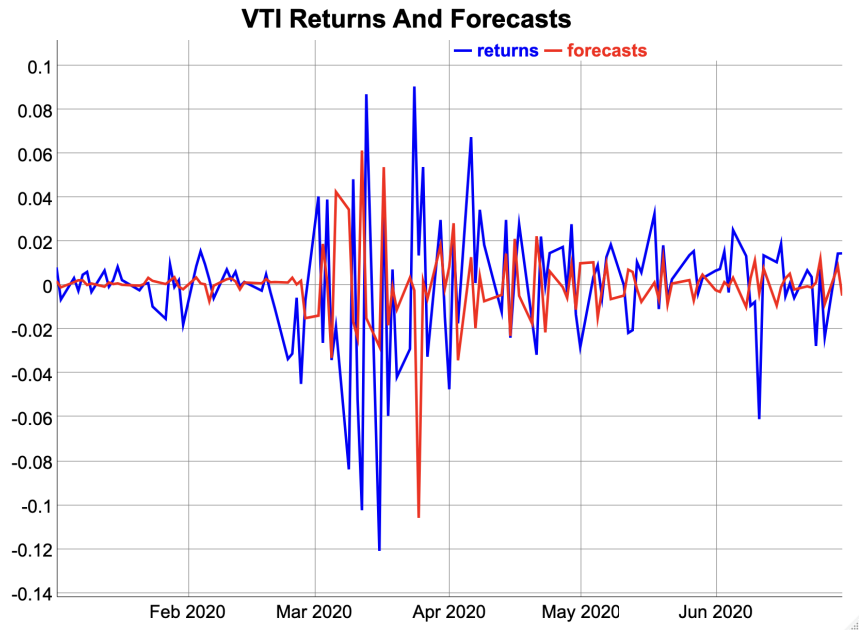
Mean Squared Error of the Autoregressive Forecasting Model

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 errors ε_i, equal to the differences between the *forecasts* f_t minus the actual values r_t: $\varepsilon_i = f_t - r_t$:

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

```
> # Calculate the correlation between forecasts and returns
> cor(fcasts, retp)
> # Calculate the forecasting errors
> errorf <- (fcasts - retp)
> # Mean squared error
> mean(errorf^2)
```




```
> # Plot the forecasts
> datav <- cbind(retp, fcasts)["2020-01/2020-06"]
> colnames(datav) <- c("returns", "forecasts")
> dygraphs::dygraph(datav,
+   main="VTI Returns And Forecasts") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=300)
```

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 (`lookb`).

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 (`lookb`).

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

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 (`lookb`).

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

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