FRE7241 Algorithmic Portfolio Management

Lecture#2, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

January 30, 2018

NYU | TANDON SCHOOL OF ENGINEERING

Package *quantmod* for Quantitative Financial Modeling

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

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

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

```
> # load package quantmod
> library(quantmod)
> # get documentation for package quantmod
> # get short description
> packageDescription("quantmod")
> # load help page
> help(package="quantmod")
> # list all datasets in "quantmod"
> data(package="quantmod")
> # list all objects in "quantmod"
> ls("package:quantmod")
> # remove quantmod from search path
> detach("package:quantmod")
```

ETF Dataset

```
> # ETF symbols for asset allocation
> sym_bols <- c("VTI", "VEU", "IEF", "VNQ",
+   "DBC", "VXX", "XLY", "XLP", "XLE", "XLF",
+   "XLV", "XLI", "XLB", "XLK", "XLU", "VYM",
+   "IVW", "IWB", "IWD", "IWF")
> # read etf database into data frame
> etf_list <- read.csv(
+   file='C:/Develop/R/lecture_slides/data/etf_list.csv',
+          stringsAsFactors=FALSE)
> rownames(etf_list) <- etf_list$Symbol
> # subset etf_list only those ETF's in sym_bols
> etf_list <- etf_list[sym_bols, ]
> # shorten names
> etf_names <- sapply(etf_list$Name,
+                 function(name) {
+   name_split <- strsplit(name, split=" ")[[1]]
+   name_split <-
+     name_split[c(-1, -length(name_split))]
+   name_match <- match("Select", name_split)
+   if (!is.na(name_match))
+     name_split <- name_split[-name_match]
+   paste(name_split, collapse=" ")
+ })  # end sapply
> etf_list$Name <- etf_names
> etf_list["IEF", "Name"] <- "Treasury Bond Fund"
> etf_list["XLY", "Name"] <- "Consumer Discr. Sector Fund"
```

Symbol	Name	Fund.Type
VTI	Total Stock Market	US Equity ETF
VEU	FTSE All World Ex US	Global Equity ETF
IEF	Treasury Bond Fund	US Fixed Income ETF
VNQ	REIT ETF - DNQ	US Equity ETF
DBC	DB Commodity Index Trac	Commodity Based ETF
VXX	S&P500 VIX Futures	Commodity Based ETN
XLY	Consumer Discr. Sector Fund	US Equity ETF
XLP	Consumer Staples Sector Fund	US Equity ETF
XLE	Energy Sector Fund	US Equity ETF
XLF	Financial Sector Fund	US Equity ETF
XLV	Health Care Sector Fund	US Equity ETF
XLI	Industrial Sector Fund	US Equity ETF
XLB	Materials Sector Fund	US Equity ETF
XLK	Technology Sector Fund	US Equity ETF
XLU	Utilities Sector Fund	US Equity ETF
VYM	Large-cap Value	US Equity ETF
IVW	S&P 500 Growth Index Fund	US Equity ETF
IWB	Russell 1000	US Equity ETF
IWD	Russell 1000 Value	US Equity ETF
IWF	Russell 1000 Growth	US Equity ETF

ETFs with names *X** represent industry sector funds,

ETFs with names *I** represent style funds (value, growth),

IWB is the Russell 1000 small-cap fund,

VXX is the VIX volatility fund,

Downloading Time Series Data Using Package *quantmod*

The function getSymbols() downloads time series data into the specified *environment*,

getSymbols() creates objects in the specified *environment* from the input strings (names),

It then assigns the data to those objects, without returning them as a function value, as a *side effect*,

By default, getSymbols() downloads for each symbol the daily *OHLC* prices and trading volume (Open, High, Low, Close, Adjusted, Volume),

The method getSymbols.yahoo accepts arguments "from" and "to" which specify the date range for the data,

If the argument "auto.assign" is set to FALSE, then getSymbols() returns the data, instead of assigning it silently,

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Alpha Vantage* and *Quandl* as the only major providers of free daily *OHLC* stock prices,

```
> library(quantmod)  # load package quantmod
> env_etf <- new.env()  # new environment for da
> # download data for sym_bols into env_etf from
> getSymbols.av(sym_bols, adjust=TRUE, env=env_e
+   output.size="full", api.key="T7JPW54ES8G7531
> # getSymbols(sym_bols, env=env_etf, adjust=TRU
```

```
> ls(env_etf)  # list files in env_etf
> # get class of object in env_etf
> class(get(x=sym_bols[1], envir=env_etf))
> # another way
> class(env_etf$VTI)
> colnames(env_etf$VTI)
> head(env_etf$VTI, 3)
> # get class of all objects in env_etf
> eapply(env_etf, class)
> # get class of all objects in R workspace
> lapply(ls(), function(ob_ject) class(get(ob_je
```

Adjusting Stock Prices Using Package *quantmod*

Traded stock and bond prices experience jumps after splits and dividends, and must be adjusted to account for them,

The function `adjustOHLC()` adjusts *OHLC* prices,

The function `get()` retrieves objects that are referenced using character strings, instead of their names,

The `assign()` function assigns a value to an object in a specified *environment*, by referencing it using a character string (name),

The functions `get()` and `assign()` allow retrieving and assigning values to objects that are referenced using character strings,

If the argument `"adjust"` in function `getSymbols()` is set to `TRUE`, then `getSymbols()` returns adjusted data,

```
> # check of object is an OHLC time series
> is.OHLC(env_etf$VTI)
> # adjust single OHLC object using its name
> env_etf$VTI <- adjustOHLC(env_etf$VTI,
+                      use.Adjusted=TRUE)
>
> # adjust OHLC object using string as name
> assign(sym_bols[1], adjustOHLC(
+     get(x=sym_bols[1], envir=env_etf),
+     use.Adjusted=TRUE),
+   envir=env_etf)
>
> # adjust objects in environment using vector of strings
> for (sym_bol in sym_bols) {
+   assign(sym_bol,
+    adjustOHLC(get(sym_bol, envir=env_etf),
+               use.Adjusted=TRUE),
+    envir=env_etf)
+ }  # end for
```

Extracting Prices Using Package *quantmod*

Data can be extracted from an *environment* by coercing it into a `list`, and then subsetting and merging it into an *xts* using the function `do.call()`,

A list of *xts* can be flattened into a single *xts* using the function `do.call()`,

The function `do.call()` executes a function call using a function name and a list of arguments,

The function `do.call()` passes the list elements individually, instead of passing the whole list as one argument,

The function `do_call()` from package *rutils* performs the same operation as `do.call()`, but using recursion, which is much faster and uses less memory,

The extractor (accessor) functions `Ad()`, `Vo()`, etc., extract columns from *OHLC* data,

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list,

```
> # extract and merge all data, subset by symbol
> price_s <- do.call(merge,
+   as.list(env_etf)[sym_bols])
> # or
> price_s <- rutils::do_call(cbind,
+   as.list(env_etf)[sym_bols])
> # extract and merge adjusted prices, subset by
> price_s <- rutils::do_call(cbind,
+   lapply(as.list(env_etf)[sym_bols], Ad))
> # same, but works only for OHLC series
> price_s <- rutils::do_call(cbind,
+   eapply(env_etf, Ad)[sym_bols])
> # drop ".Adjusted" from colnames
> colnames(price_s) <-
+   sapply(colnames(price_s),
+     function(col_name)
+ strsplit(col_name, split="[.]")[[1]])[1, ]
> tail(price_s[, 1:2], 3)
> # which objects in global environment are clas
> unlist(eapply(globalenv(), is.xts))
> # save xts to csv file
> write.zoo(price_s,
+   file='etf_series.csv', sep=",")
> # copy price_s into env_etf and save to .RData
> assign("price_s", price_s, envir=env_etf)
> save(env_etf, file='etf_data.RData')
```

Calculating Returns from Adjusted Prices

```
> # calculate returns from adjusted prices
> re_turns <- lapply(env_etf$price_s, function(x
+ # dailyReturn returns single xts with bad colu
+   daily_return <- dailyReturn(x_ts)
+   colnames(daily_return) <- names(x_ts)
+   daily_return
+ })  # end lapply
>
> # "re_turns" is a list of xts
> class(re_turns)
> class(re_turns[[1]])
>
> # flatten list of xts into a single xts
> re_turns <- rutils::do_call(cbind, re_turns)
```

```
> class(re_turns)
> dim(re_turns)
> head(re_turns[, 1:3])
> # copy re_turns into env_etf and save to .RData
> assign("re_turns", re_turns, envir=env_etf)
> save(env_etf, file='etf_data.RData')
```

Managing Data Inside Environments

The function `as.environment()` coerces objects (lists) into an environment,

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list,

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects,

```
> start_date <- "2012-05-10"; end_date <- "2013-11-20"
> # subset all objects in environment and return as envir
> new_env <- as.environment(eapply(env_etf, "[",
+             paste(start_date, end_date, sep="/")))
> # subset only sym_bols in environment and return as envi
> new_env <- as.environment(
+   lapply(as.list(env_etf)[sym_bols], "[",
+    paste(start_date, end_date, sep="/")))
> # extract and merge adjusted prices and return to envir
> assign("price_s", do.call(merge,
+          lapply(ls(env_etf), function(sym_bol) {
+            x_ts <- Ad(get(sym_bol, env_etf))
+            colnames(x_ts) <- sym_bol
+            x_ts
+          })), envir=new_env)
> # get sizes of OHLC xts series in env_etf
> sapply(mget(sym_bols, envir=env_etf), object.size)
> # extract and merge adjusted prices and return to envir
> col_name <- function(x_ts)
+   strsplit(colnames(x_ts), split="[.]")[[1]][1]
> assign("price_s", do.call(merge,
+          lapply(mget(env_etf$sym_bols, envir=env_etf),
+                 function(x_ts) {
+                   x_ts <- Ad(x_ts)
+                   colnames(x_ts) <- col_name(x_ts)
+                   x_ts
+          })), envir=new_env)
```

Plotting *OHLC* Time Series Using chartSeries()

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

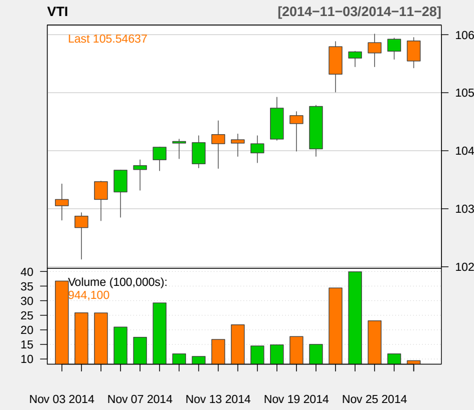
The argument "type" determines the type of plot (candlesticks, bars, or lines),

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

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

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

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



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

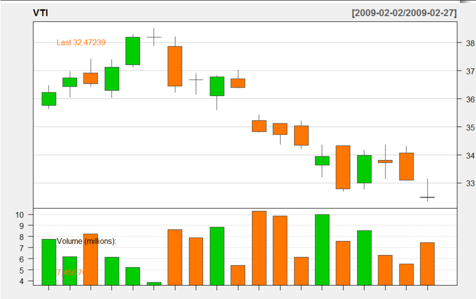
The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

Redrawing Plots Using reChart()

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

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

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



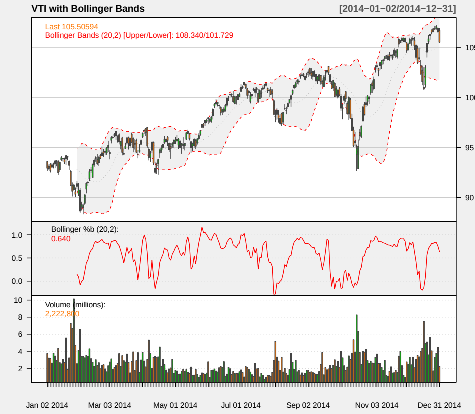
Plotting Technical Indicators Using chartSeries()

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

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

The function newTA() allows defining new technical indicators,

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



Adding Indicators and Lines Using addTA()

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

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

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

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

```
> oh_lc <- rutils::env_etf$VTI["2009-02/2009-03"]
> VTI_adj <- Ad(oh_lc); VTI_vol <- Vo(oh_lc)
> # calculate volume-weighted average price
> VTI_vwap <- TTR::VWAP(price=VTI_adj,
+ volume=VTI_vol, n=10)
> # plot OHLC candlechart with volume
> chartSeries(oh_lc, name="VTI plus VWAP",
+         theme=chartTheme("white"))
> # add VWAP to main plot
> addTA(ta=VTI_vwap, on=1, col='red')
> # add price minus VWAP in extra panel
> addTA(ta=(VTI_adj-VTI_vwap), col='red')
```

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

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

Shading Plots Using addTA()

addTA() accepts Boolean vectors for shading of plots,

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

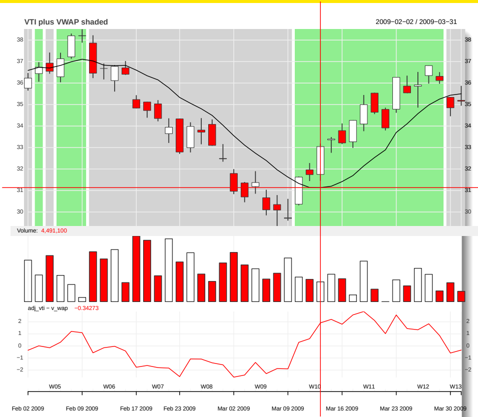
```
> # plot OHLC candlechart with volume
> chartSeries(oh_lc, name="VTI plus VWAP shaded"
+         theme=chartTheme("white"))
> # add VWAP to main plot
> addTA(ta=VTI_vwap, on=1, col='red')
> # add price minus VWAP in extra panel
> addTA(ta=(VTI_adj-VTI_vwap), col='red')
> # add background shading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

Plotting Time Series Using chart_Series()

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

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

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



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

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

Plot and Theme Objects of chart_Series()

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

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

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

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

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

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

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

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

```
> # extract plot object
> ch_ob <- chart_Series(x=oh_lc, plot=FALSE)
> class(ch_ob)
> ls(ch_ob)
> class(ch_ob$get_ylim)
> class(ch_ob$set_ylim)
> # ls(ch_ob$Env)
> class(ch_ob$Env$actions)
> plot_theme <- chart_theme()
> class(plot_theme)
> ls(plot_theme)
```

Customizing chart_Series() Plots

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

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

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

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

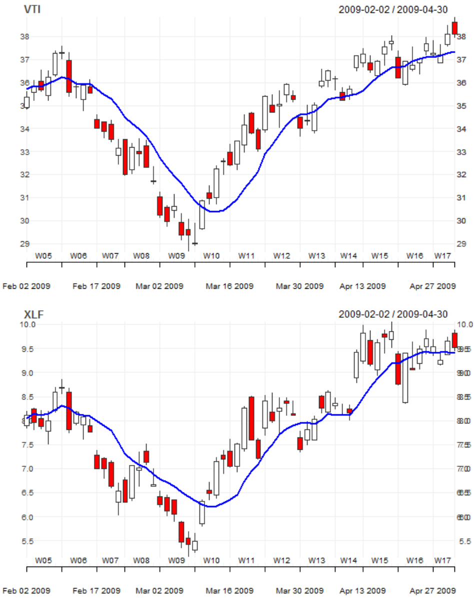
```
> oh_lc <- rutils::env_etf$VTI["2010-04/2010-05"]
> # extract, modify theme, format tick marks "%b %d"
> plot_theme <- chart_theme()
> plot_theme$format.labels <- "%b %d"
> # create plot object
> ch_ob <- chart_Series(x=oh_lc,
+                   theme=plot_theme, plot=FALSE)
> # extract ylim using accessor function
> y_lim <- ch_ob$get_ylim()
> y_lim[[2]] <- structure(
+   range(Ad(oh_lc)) + c(-1, 1),
+   fixed=TRUE)
> # modify plot object to reduce y-axis range
> ch_ob$set_ylim(y_lim)  # use setter function
> # render the plot
> plot(ch_ob)
```

Plotting chart_Series() in Multiple Panels

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

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

```
> # calculate VTI and XLF volume-weighted averag
> VTI_vwap <-
+   TTR::VWAP(price=Ad(rutils::env_etf$VTI),
+       volume=Vo(rutils::env_etf$VTI), n=10)
> XLF_vwap <-
+   TTR::VWAP(price=Ad(rutils::env_etf$XLF),
+       volume=Vo(rutils::env_etf$XLF), n=10)
> # open graphics device, and define
> # plot area with two horizontal panels
> x11(); par(mfrow=c(2, 1))
> ch_ob <- chart_Series(  # plot in top panel
+   x=env_etf$VTI["2009-02/2009-04"],
+   name="VTI", plot=FALSE)
> add_TA(VTI_vwap["2009-02/2009-04"],
+  lwd=2, on=1, col='blue')
> ch_ob <- chart_Series(  # plot in bottom panel
+   x=env_etf$XLF["2009-02/2009-04"],
+   name="XLF", plot=FALSE)
> add_TA(XLF_vwap["2009-02/2009-04"],
+  lwd=2, on=1, col='blue')
```



Plotting *OHLC* Time Series Using Package *dygraphs*

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

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

```
> library(dygraphs)
> # calculate volume-weighted average price
> oh_lc <- rutils::env_etf$VTI
> VTI_vwap <- TTR::VWAP(price=quantmod::Ad(oh_lc
+     volume=quantmod::Vo(oh_lc), n=20)
> # add VWAP to OHLC  data
> oh_lc <- cbind(oh_lc[, c(1:3, 6)],
+            VTI_vwap)["2009-02/2009-04"]
> # create dygraphs object
> dy_graph <- dygraphs::dygraph(oh_lc)
> # convert dygraphs object to candlestick plot
> dy_graph <- dygraphs::dyCandlestick(dy_graph)
> # render candlestick plot
> dy_graph
> # candlestick plot using pipes syntax
> dygraphs::dygraph(oh_lc) %>% dyCandlestick()
> # candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dygraph(oh_lc))
```

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

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

dygraphs OHLC Plots With Background Shading

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

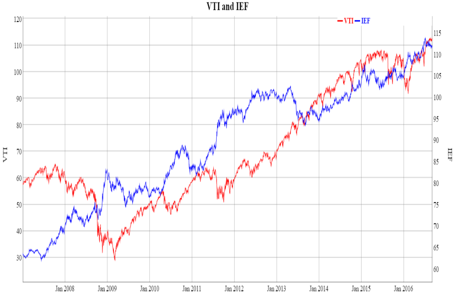
```
> # create candlestick plot with background shad
> in_dex <- index(oh_lc)
> in_dic <-
+   rutils::diff_xts(oh_lc[, 4] > oh_lc[, "VWAP"
> in_dic <- rbind(cbind(which(in_dic==1), 1),
+   cbind(which(in_dic==(-1)), -1))
> in_dic <- in_dic[order(in_dic[, 1]), ]
> in_dic <- rbind(c(1, -in_dic[1, 2]), in_dic,
+   c(NROW(oh_lc), -in_dic[NROW(in_dic), 2]))
> in_dic <-
+   data.frame(in_dex[in_dic[, 1]], in_dic[, 2])
> # create dygraphs object
> dy_graph <- dygraphs::dygraph(oh_lc) %>%
+   dyCandlestick()
> # add shading
> for (i in 1:(NROW(in_dic)-1)) {
+   if (in_dic[i, 2] == 1)
+     dy_graph <- dy_graph %>% dyShading(from=in_dic[i, 1], to=in_dic[i+1, 1], color="lightgreen
+   else
+     dy_graph <- dy_graph %>% dyShading(from=in_dic[i, 1], to=in_dic[i+1, 1], color="antiquewhi
+ }  # end for
> # render plot
> dy_graph
```

dygraphs Plots With Two "y" Axes

The function dyAxis() from package *dygraphs* adds customized axes to a *dygraphs* plot object,

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

```
> library(dygraphs)
> # prepare VTI and IEF prices
> price_s <- cbind(Ad(rutils::env_etf$VTI),
+             Ad(rutils::env_etf$IEF))
> col_names <- rutils::get_name(colnames(price_s
> colnames(price_s) <- col_names
>
> # dygraphs plot with two y-axes
> library(dygraphs)
> dygraphs::dygraph(price_s, main=paste(col_name
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(col_names[2], axis="y2", col=c("red", "blue"))
```



Downloading The *S&P500* Index Time Series From *Yahoo*

The *S&P500* stock market index is a capitalization-weighted average of the 500 largest U.S. companies, and covers about 80% of the U.S. stock market capitalization,

Yahoo provides daily *OHLC* prices for the *S&P500* index (symbol *^GSPC*), and for the *S&P500* total return index (symbol *^SP500TR*),

But special characters in some stock symbols, like `"-"` or `"^"` are not allowed in `R` names,

For example, the symbol *^GSPC* for the *S&P500* stock market index isn't a valid name in `R`,

The function `setSymbolLookup()` creates valid names corresponding to stock symbols, which are then used by the function `getSymbols()` to create objects with the valid names,

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Alpha Vantage* and *Quandl* as the only major providers of free daily *OHLC* stock prices,

```
> # assign name SP500 to ^GSPC symbol
> setSymbolLookup(
+   SP500=list(name="^GSPC", src="yahoo"))
> getSymbolLookup()
> # view and clear options
> options("getSymbols.sources")
> options(getSymbols.sources=NULL)
> # download S&P500 prices into env_etf
> getSymbols("SP500", env=env_etf,
+     adjust=TRUE, from="1990-01-01")
> chart_Series(x=env_etf$SP500["2016/"],
+        TA="add_Vo()",
+        name="S&P500 index")
```

Downloading The *DJIA* Index Time Series From *Yahoo*

The Dow Jones Industrial Average (*DJIA*) stock market index is a price-weighted average of the 30 largest U.S. companies (same number of shares per company),

Yahoo provides daily *OHLC* prices for the *DJIA* index (symbol *^DJI*), and for the *DJITR* total return index (symbol *DJITR*),

But special characters in some stock symbols, like `"-"` or `"^"` are not allowed in `R` names,

For example, the symbol *^DJI* for the *DJIA* stock market index isn't a valid name in `R`,

The function `setSymbolLookup()` creates valid names corresponding to stock symbols, which are then used by the function `getSymbols()` to create objects with the valid names,

```
> # assign name DJIA to ^DJI symbol
> setSymbolLookup(
+   DJIA=list(name="^DJI", src="yahoo"))
> getSymbolLookup()
> # view and clear options
> options("getSymbols.sources")
> options(getSymbols.sources=NULL)
> # download DJIA prices into env_etf
> getSymbols("DJIA", env=env_etf,
+     adjust=TRUE, from="1990-01-01")
> chart_Series(x=env_etf$DJIA["2016/"],
+        TA="add_Vo()",
+        name="DJIA index")
```

Scraping *S&P500* Stock Index Constituents From Websites

The *S&P500* index constituents change over time, and *Standard & Poor's* replaces companies that have decreased in capitalization with ones that have increased,

The *S&P500* index may contain more than 500 stocks because some companies have several share classes of stock,

The *S&P500* index constituents may be scraped from websites like Wikipedia, using dedicated packages,

The function `getURL()` from package *RCurl* downloads the *HTML* text data from a URL,

The function `readHTMLTable()` from package *XML* extracts tables from *HTML* text data or from a remote URL, and returns them as a list of data frames or matrices,

`readHTMLTable()` can't parse secure URLs, so they must first be downloaded using function `getURL()`, and then parsed using `readHTMLTable()`,

```
> library(RCurl)  # load package RCurl
> library(XML)  # load package XML
> # download text data from URL
> sp_500 <- getURL(
+   "https://en.wikipedia.org/wiki/List_of_S%26P
> # extract tables from the text data
> sp_500 <- readHTMLTable(sp_500,
+                   stringsAsFactors=FALSE)
> str(sp_500)
> # extract colnames of data frames
> lapply(sp_500, colnames)
> # extract S&P500 constituents
> sp_500 <- sp_500[[1]]
> head(sp_500)
> # create valid R names from symbols containing
> sp_500$names <- gsub("-", "_", sp_500$Ticker)
> sp_500$names <- gsub("[.]", "_", sp_500$names)
> # write data frame of S&P500 constituents to C
> write.csv(sp_500,
+   file="C:/Develop/R/lecture_slides/data/sp500
+   row.names=FALSE)
```

Downloading *S&P500* Time Series Data From *Yahoo*

Before time series data for S&P500 constituents can be downloaded from *Yahoo*, it's necessary to create valid names corresponding to symbols containing special characters like "-",

The function `setSymbolLookup()` creates a lookup table for *Yahoo* symbols, using valid names in `R`,

For example *Yahoo* uses the symbol "BRK-B", which isn't a valid name in `R`, but can be mapped to "BRK_B", using the function `setSymbolLookup()`,

```
> library(HighFreq)  # load package HighFreq
> # load data frame of S&P500 constituents from CSV file
> sp_500 <- read.csv(file="C:/Develop/R/lecture_slides/dat
+      stringsAsFactors=FALSE)
> # register symbols corresponding to R names
> for (in_dex in 1:NROW(sp_500)) {
+   cat("processing: ", sp_500$Ticker[in_dex], "\n")
+   setSymbolLookup(structure(
+     list(list(name=sp_500$Ticker[in_dex])),
+     names=sp_500$names[in_dex]))
+ }  # end for
> env_sp500 <- new.env()  # new environment for data
> # remove all files (if necessary)
> rm(list=ls(env_sp500), envir=env_sp500)
> # download data and copy it into environment
> rutils::get_symbols(sp_500$names,
+    env_out=env_sp500, start_date="1990-01-01")
> # or download in loop
> for (sym_bol in sp_500$names) {
+   cat("processing: ", sym_bol, "\n")
+   rutils::get_symbols(sym_bol,
+    env_out=env_sp500, start_date="1990-01-01")
+ }  # end for
> save(env_sp500, file="C:/Develop/R/lecture_slides/data/s
> chart_Series(x=env_sp500$BRK_B["2016/"], TA="add_Vo()",
+        name="BRK-B stock")
```

Downloading Time Series Data From *Alpha Vantage*

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Alpha Vantage* and *Quandl* as the only major providers of free daily *OHLC* stock prices,

But *Quandl* doesn't provide free *ETF* prices, while *Alpha Vantage* does,

The function `getSymbols()` has a *method* for downloading time series data from *Alpha Vantage*, called `getSymbols.av()`,

Users must first obtain an *Alpha Vantage* API key, and then pass it in `getSymbols.av()` calls:
https://www.alphavantage.co/

```
> # load data frame of S&P500 constituents from CSV file
> sp_500 <- read.csv(file="C:/Develop/R/lecture_slides/dat
> sym_bols <- sp_500$co_tic
> env_sp500 <- new.env()  # new environment for data
> # remove all files (if necessary)
> rm(list=ls(env_sp500), envir=env_sp500)
> # download in while loop from Alpha Vantage and copy int
> down_loaded <- sym_bols %in% ls(env_sp500)
> it_er <- 0
> while (((NROW(down_loaded) - sum(down_loaded)) > 0) & (i
+   # Boolean vector of symbols already downloaded
+   down_loaded <- sym_bols %in% ls(env_sp500)
+   # download data and copy it into environment
+   for (sym_bol in sym_bols[!down_loaded]) {
+     cat("processing: ", sym_bol, "\n")
+     tryCatch(  # with error handler
+   getSymbols(sym_bol, adjust=TRUE, env=env_sp500,
+              output.size="full", api.key="T7JPW54ES8G75
+ # error handler captures error condition
+ error=function(error_cond) {
+   print(paste("error handler: ", error_cond))
+ },  # end error handler
+ finally=print(paste("sym_bol=", sym_bol))
+     )  # end tryCatch
+   }  # end for
+   it_er <- it_er + 1
+   Sys.sleep(2*60)
+ }  # end while
```

Managing Stock Symbols and Adjusting *OHLC* Prices

The column names for symbol "LOW" must be renamed for the extractor (accessor) function Lo() to work properly,

The function adjustOHLC() with argument use.Adjusted=TRUE, adjusts all the *OHLC* price columns, using the *Adjusted* price column,


```
> # rename "LOW" colnames to "L_OWES"
> colnames(env_sp500$LOW) <-
+   sapply(colnames(env_sp500$LOW),
+     function(col_name) {
+ col_name <- strsplit(col_name, split="[.]")[[1
+ paste("L_OWES", col_name[2], sep=".")
+     })
> assign("L_OWES", env_sp500$LOW, envir=env_sp500)
> rm(LOW, envir=env_sp500)
> # adjust all OHLC in environment using vector of strings
> for (sym_bol in ls(env_sp500)) {
+   assign(sym_bol,
+     adjustOHLC(get(x=sym_bol, envir=env_sp500), use.Adjusted=TRUE),
+     envir=env_sp500)
+ }  # end for
> save(env_sp500,
+   file="C:/Develop/R/lecture_slides/data/sp500.RData")
> chart_Series(x=env_sp500$L_OWES["2017-06/"],
+   TA="add_Vo()", name="LOWES stock")
```

Downloading *FRED* Time Series Data

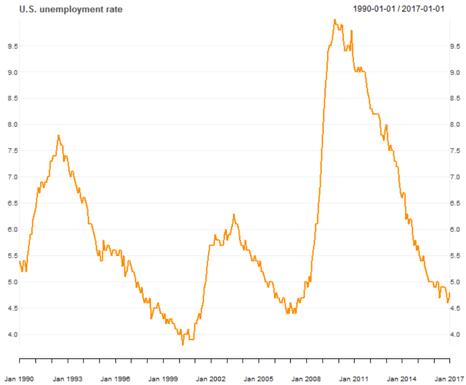
FRED is a database of economic time series maintained by the Federal Reserve Bank of St. Louis:

http://research.stlouisfed.org/fred2/

The function getSymbols() downloads time series data into the specified *environment*,

getSymbols() can download *FRED* data with the argument "src" set to FRED,

If the argument "auto.assign" is set to FALSE, then getSymbols() returns the data, instead of assigning it silently,




```
> # download U.S. unemployment rate data
> unemp_rate <- getSymbols("UNRATE",
+              auto.assign=FALSE,
+              src="FRED")
> # plot U.S. unemployment rate data
> chart_Series(unemp_rate["1990/"],
+       name="U.S. unemployment rate")
```

The *Quandl* Database

Quandl is a distributor of third party data, and offers several million financial, economic, and social datasets,

Much of the *Quandl* data is free, while premium data can be obtained under a temporary license,

Quandl offers online help and a guide to its datasets:
https://www.quandl.com/help/r
https://www.quandl.com/browse
https://www.quandl.com/blog/getting-started-with-the-quandl-api
https://www.quandl.com/blog/stock-market-data-guide

Quandl offers stock prices, stock fundamentals, financial ratios, indexes, options and volatility, earnings estimates, analyst ratings, etc.:
https://www.quandl.com/blog/api-for-stock-data

```
> install.packages("devtools")
> library(devtools)
> # install package Quandl from github
> install_github("quandl/R-package")
> library(Quandl)  # load package Quandl
> # register Quandl API key
> Quandl.api_key("pVJi9Nv3V8CD3Js5s7Qx")
> # get short description
> packageDescription("Quandl")
> # load help page
> help(package="Quandl")
> # remove Quandl from search path
> detach("package:Quandl")
```

Quandl has developed an `R` package called *Quandl* that allows downloading data from *Quandl* directly into `R`,

To make more than 50 downloads a day, you need to register your *Quandl* API key using the function `Quandl.api_key()`,

Downloading Time Series Data from *Quandl*

Quandl data can be downloaded directly into R using the function Quandl(),

The dots "..." argument of the Quandl() function accepts additional parameters to the *Quandl API*,

Quandl datasets have a unique *Quandl Code* in the format "database/ticker", which can be found on the *Quandl* website for that dataset:
https://www.quandl.com/data/WIKI?keyword=aapl

WIKI is a user maintained free database of daily prices for 3,000 U.S. stocks,
https://www.quandl.com/data/WIKI

SEC is a free database of stock fundamentals extracted from *SEC 10Q* and *10K* filings (but not harmonized),
https://www.quandl.com/data/SEC

RAYMOND is a free database of harmonized stock fundamentals, based on the *SEC* database,
https://www.quandl.com/data/RAYMOND-Raymond
https://www.quandl.com/data/RAYMOND-Raymond?keyword=aapl

```
> library(quantmod)  # load package quantmod
> # download EOD AAPL prices from WIKI free data
> price_s <- Quandl(code="WIKI/AAPL",
+              type="xts", start_date="1990-01-01
> x11(width=14, height=7)
> chart_Series(price_s["2016", 1:4],
+     name="AAPL OHLC prices")
> # add trade volume in extra panel
> add_TA(price_s["2016", 5])
> # download euro currency rates
> price_s <- Quandl(code="BNP/USDEUR",
+     start_date="2013-01-01",
+     end_date="2013-12-01", type="xts")
> # download multiple time series
> price_s <- Quandl(code=c("NSE/OIL", "WIKI/AAPL
+     start_date="2013-01-01", type="xts")
> # download AAPL gross profits
> prof_it <- Quandl("RAYMOND/AAPL_GROSS_PROFIT_Q
+     type="xts")
> chart_Series(prof_it, name="AAPL gross profits
> # download Hurst time series
> price_s <- Quandl(code="PE/AAPL_HURST",
+     start_date="2013-01-01", type="xts")
> chart_Series(price_s["2016/", 1],
+        name="AAPL Hurst")
```

Stock Index and Instrument Metadata on *Quandl*

Instrument metadata specifies properties of instruments, like its currency, contract size, tick value, delievery months, start date, etc.

Quandl provides instrument metadata for stock indices, futures, and currencies:

https://www.quandl.com/blog/useful-lists

Quandl also provides constituents for stock indices, for example the *S&P500*, *Dow Jones Industrial Average*, *NASDAQ Composite*, *FTSE 100*, etc.

```
> # load S&P500 stock Quandl codes
> sp_500 <- read.csv(
+   file="C:/Develop/R/lecture_slides/data/sp500
+   stringsAsFactors=FALSE)
> # replace "-" with "_" in symbols
> sp_500$free_code <-
+   gsub("-", "_", sp_500$free_code)
> head(sp_500)
> # vector of symbols in sp_500 frame
> tick_ers <- gsub("-", "_", sp_500$ticker)
> # or
> tick_ers <- matrix(unlist(
+   strsplit(sp_500$free_code, split="/"),
+   use.names=FALSE), ncol=2, byrow=TRUE)[, 2]
> # or
> tick_ers <- do_call_rbind(
+   strsplit(sp_500$free_code, split="/"))[, 2]
```

Downloading Multiple Time Series from *Quandl*

Time series data for a portfolio of stocks can be downloaded by performing a loop over the function `Quandl()` from package *Quandl*,

The `assign()` function assigns a value to an object in a specified *environment*, by referencing it using a character string (name),

```
> env_sp500 <- new.env()  # new environment for data
> # remove all files (if necessary)
> rm(list=ls(env_sp500), envir=env_sp500)
> # Boolean vector of symbols already downloaded
> down_loaded <- tick_ers %in% ls(env_sp500)
> # download data and copy it into environment
> for (tick_er in tick_ers[!down_loaded]) {
+   cat("processing: ", tick_er, "\n")
+   da_ta <- Quandl(code=paste0("WIKI/", tick_er),
+               start_date="1990-01-01",
+               type="xts")[, -(1:7)]
+   colnames(da_ta) <- paste(tick_er,
+     c("Open", "High", "Low", "Close", "Volume"), sep=".'
+   assign(tick_er, da_ta, envir=env_sp500)
+ }  # end for
> save(env_sp500, file="C:/Develop/R/lecture_slides/data/s
> chart_Series(x=env_sp500$XOM["2016/"], TA="add_Vo()",
+        name="XOM stock")
```

Aggregations Over Look-back Intervals

A time *period* is defined as the time between two neighboring points in time,

A time *interval* is defined as the time spanned by one or more neighboring time *periods*,

A *look-back interval* is a time *interval* for performing aggregations over the past, starting from a *startpoint* and ending at an *endpoint*,

The *startpoints* are the *endpoints* lagged by the interval width (number of periods in the interval),

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

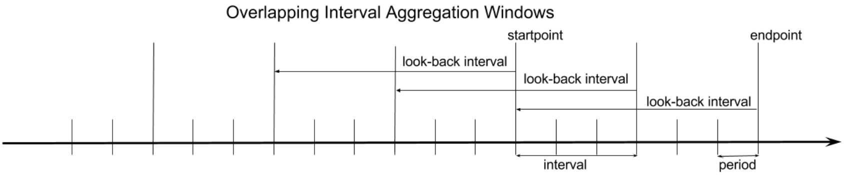
A rolling aggregation is specified by a vector of look-back *intervals* at each point in time,

An example of a rolling aggregation are moving average prices,

An interval aggregation is specified by a vector of look-back *intervals* attached at *endpoints* spanning multiple time *periods*,

An example of a non-overlapping interval aggregation are monthly asset returns,

An example of an overlapping interval aggregation are trailing 12-month asset returns calculated monthly,



Performing Rolling 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 `look_back` 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),

```
> price_s <- Cl(rutils::env_etf$VTI)
> end_points <- seq_along(price_s)  # define end points
> len_gth <- NROW(end_points)
> look_back <- 22  # number of data points per look-back i
> # start_points are multi-period lag of end_points
> start_points <- c(rep_len(1, look_back-1),
+     end_points[1:(len_gth-look_back+1)])
> # define list of look-back intervals for aggregations ov
> look_backs <- lapply(seq_along(end_points),
+   function(in_dex) {
+     start_points[in_dex]:end_points[in_dex]
+ })  # end lapply
> # define aggregation function
> agg_regate <- function(x_ts) c(max=max(x_ts), min=min(x_
> # perform aggregations over look_backs list
> agg_regations <- sapply(look_backs,
+     function(look_back) agg_regate(price_s[look_back])
+ )  # end sapply
> # coerce agg_regations into matrix and transpose it
> if (is.vector(agg_regations))
+   agg_regations <- t(agg_regations)
> agg_regations <- t(agg_regations)
> # coerce agg_regations into xts series
> agg_regations <- xts(agg_regations,
+                      order.by=index(price_s[end_points]))
```

Performing Rolling 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 look_backs list
> agg_regations <- lapply(look_backs,
+     function(look_back) agg_regate(price_s[look_back])
+ )  # end lapply
> # rbind list into single xts or matrix
> agg_regations <- rutils::do_call_rbind(agg_regations)
> # convert into xts
> agg_regations <- xts::xts(agg_regations,
+     order.by=index(price_s))
> agg_regations <- cbind(agg_regations, price_s)
> # plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "green")
> x11()
> chart_Series(agg_regations, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(agg_regations),
+   bg="white", lty=c(1, 1, 1), lwd=c(6, 6, 6),
+   col=plot_theme$col$line.col, bty="n")
```

Defining Functionals for Rolling Aggregations

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

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 look_back 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 look_back,

A second interval may be one of the

```
> # define functional for rolling aggregations
> roll_agg <- function(x_ts, look_back, FUN, ...) {
+ # define end points at every period
+   end_points <- seq_along(x_ts)
+   len_gth <- NROW(end_points)
+ # define starting points as lag of end_points
+   start_points <- c(rep_len(1, look_back-1),
+     end_points[1:(len_gth-look_back+1)])
+ # define list of look-back intervals for aggregations ov
+ look_backs <- lapply(seq_along(end_points),
+   function(in_dex) {
+     start_points[in_dex]:end_points[in_dex]
+ })  # end lapply
+ # perform aggregations over look_backs list
+   agg_regations <- lapply(look_backs,
+     function(look_back) FUN(x_ts[look_back], ...)
+   )  # end lapply
+ # rbind list into single xts or matrix
+   agg_regations <- rutils::do_call_rbind(agg_regations)
+ # coerce agg_regations into xts series
+   if (!is.xts(agg_regations))
+     agg_regations <- xts(agg_regations, order.by=index(x
+   agg_regations
+ }  # end roll_agg
> # define aggregation function
> agg_regate <- function(x_ts)
+   c(max=max(x_ts), min=min(x_ts))
> # perform aggregations over rolling interval
```

Benchmarking Speed of Rolling Aggregations

The speed of rolling 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(x_ts)
+   c(max=max(x_ts), min=min(x_ts))
> # define aggregation function that returns an xts
> agg_xts <- function(x_ts)
+   xts(t(c(max=max(x_ts), min=min(x_ts))),
+ order.by=end(x_ts))
> # benchmark the speed of aggregation functions
> library(microbenchmark)
> summary(microbenchmark(
+   agg_vector=roll_agg(price_s, look_back=look_back,
+             FUN=agg_vector),
+   agg_xts=roll_agg(price_s, look_back=look_back,
+             FUN=agg_xts),
+   times=10))[, c(1, 4, 5)]
```

Benchmarking Functionals for Rolling Aggregations

Several packages contain functionals designed for performing rolling aggregations:

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

These functionals don't require specifying the *endpoints*, and instead calculate the *endpoints* from the rolling 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 rolling 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
> agg_regate <- function(x_ts)  max(x_ts)
> # perform aggregations over a rolling interval
> agg_regations <- xts::rollapply.xts(price_s, width
+                FUN=agg_regate, align="right")
> # perform aggregations over a rolling interval
> library(PerformanceAnalytics)  # load package Perfo
> agg_regations <- apply.rolling(price_s,
+                width=look_back, FUN=agg_regate)
> # benchmark the speed of the functionals
> library(microbenchmark)
> summary(microbenchmark(
+   roll_agg=roll_agg(price_s, look_back=look_back,
+                FUN=max),
+   roll_xts=xts::rollapply.xts(price_s, width=look_
+                   FUN=max, align="right"),
+   apply_rolling=apply.rolling(price_s,
+                          width=look_back, FUN=max),
+   times=10))[, c(1, 4, 5)]
```

Rolling 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 rolling sum of an an *xts* series,

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

But rolling standard deviations and higher moments can't be easily calculated using cumsum(),

```
> # rolling sum using cumsum()
> roll_sum <- function(x_ts, look_back) {
+   cum_sum <- cumsum(na.omit(x_ts))
+   out_put <- cum_sum - lag(x=cum_sum, k=look_back)
+   out_put[1:look_back, ] <- cum_sum[1:look_back, ]
+   colnames(out_put) <- paste0(colnames(x_ts), "_stdev")
+   out_put
+ }  # end roll_sum
> agg_regations <- roll_sum(price_s, look_back=look_back)
> # define list of look-back intervals for aggregations ov
> look_backs <- lapply(seq_along(end_points),
+   function(in_dex) {
+     start_points[in_dex]:end_points[in_dex]
+ })  # end lapply
> # perform rolling aggregations using apply loop
> agg_regations <- sapply(look_backs,
+     function(look_back) sum(price_s[look_back])
+ )  # end sapply
> head(agg_regations)
> tail(agg_regations)
> # benchmark the speed of both methods
> library(microbenchmark)
> summary(microbenchmark(
+   roll_sum=roll_sum(price_s, look_back=look_back),
+   s_apply=sapply(look_backs,
+     function(look_back) sum(price_s[look_back])),
+   times=10))[, c(1, 4, 5)]
```

Performing Rolling Aggregations Using Package *TTR*

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

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

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

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

```
> # library(TTR)  # load package TTR
> # benchmark the speed of TTR::runSum
> library(microbenchmark)
> summary(microbenchmark(
+   cum_sum=cumsum(coredata(price_s)),
+   roll_sum=rutils::roll_sum(price_s, win_dow=l
+   run_sum=TTR::runSum(price_s, n=look_back),
+   times=10))[, c(1, 4, 5)]
```

Rolling *Weighted* Aggregations Using Package *RcppRoll*

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

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

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

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

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

```
> library(RcppRoll)  # load package RcppRoll
> wid_th <- 22  # number of data points per look
> # calculate rolling sum using rutils
> prices_mean <-
+   rutils::roll_sum(price_s, win_dow=wid_th)
> # calculate rolling sum using RcppRoll
> prices_mean <- RcppRoll::roll_sum(price_s,
+                  align="right", n=wid_th)
> # benchmark the speed of RcppRoll::roll_sum
> library(microbenchmark)
> summary(microbenchmark(
+   cum_sum=cumsum(coredata(price_s)),
+   rcpp_roll_sum=RcppRoll::roll_sum(price_s, n=
+   roll_sum=rutils::roll_sum(price_s, win_dow=w
+   times=10))[, c(1, 4, 5)]
> # calculate EWMA sum using RcppRoll
> weight_s <- exp(0.1*1:wid_th)
> prices_mean <- RcppRoll::roll_mean(price_s,
+ align="right", n=wid_th, weights=weight_s)
> prices_mean <- cbind(price_s,
+   rbind(coredata(price_s[1:(look_back-1), ]),
> colnames(prices_mean) <- c("SPY", "SPY EWMA")
> # plot EWMA prices with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red")
> x11()
> chart_Series(prices_mean, theme=plot_theme,
+          name="EWMA prices")
```

Performing Rolling Aggregations Using Package *caTools*

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

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

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

The rolling *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,

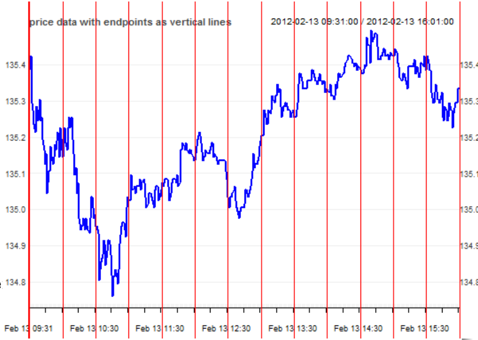
```
> library(caTools)  # load package "caTools"
> # get documentation for package "caTools"
> packageDescription("caTools")  # get short des
> help(package="caTools")  # load help page
> data(package="caTools")  # list all datasets i
> ls("package:caTools")  # list all objects in "
> detach("package:caTools")  # remove caTools fr
> # median filter
> look_back <- 11
> price_s <- Cl(HighFreq::SPY["2012-02-01/2012-0
> med_ian <- runmed(x=price_s, k=look_back)
> # vector of rolling volatility
> vol_at <- runsd(x=price_s, k=look_back,
+           endrule="constant", align="center")
> # vector of rolling quantiles
> quan_tiles <- runquantile(x=price_s,
+             k=look_back, probs=0.9,
+             endrule="constant",
+             align="center")
```

Defining Equally Spaced *Endpoints* of a Time Series

Endpoints are a vector of indices that divide a time series into non-overlapping intervals,

Endpoints may be specified as integers or as date-time objects,

```
> library(HighFreq)  # load package HighFreq
> # extract daily closing VTI prices
> price_s <- Cl(rutils::env_etf$VTI)
> # define number of data points per interval
> look_back <- 22
> # number of look_backs that fit over price_s
> n_row <- NROW(price_s)
> num_agg <- n_row %/% look_back
> # if n_row==look_back*num_agg then whole number
> # of look_backs fit over price_s
> end_points <- (1:num_agg)*look_back
> # if (n_row > look_back*num_agg)
> # then stub interval at beginning
> end_points <-
+   n_row-look_back*num_agg + (0:num_agg)*look_back
> # stub interval at end
> end_points <- c((1:num_agg)*look_back, n_row)
> # plot data and endpoints as vertical lines
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- "blue"
> chart_Series(price_s, theme=plot_theme,
+   name="prices with endpoints as vertical lines")
> abline(v=end_points, col="red")
```



Endpoints may be equally spaced, with a fixed number of data points between neighboring *endpoints*,

Endpoints start at 0 to allow the same number of data points in each equally spaced interval,

If all the data points don't fit into a whole number of intervals, then a stub interval is needed to fit the remaining data points,

Determining Calendar *Endpoints* of *xts* Time Series

The function endpoints() from package *xts* extracts the indices of the last observations in each calendar period of time of an *xts* series,

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

The *endpoints* 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
> end_points <- xts::endpoints(price_s, on="hour
> head(end_points)
> # extract the last observations in each hour
> head(price_s[end_points, ])
```

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 *endpoints*,

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,

```
> end_points <- # define end_points with beginning stub
+   n_row-look_back*num_agg + (0:num_agg)*look_back
> len_gth <- NROW(end_points)
> # start_points are single-period lag of end_points
> start_points <- c(1, end_points[1:(len_gth-1)]+1)
> # define list of look-back intervals for aggregations o
> look_backs <- lapply(seq_along(end_points),
+   function(in_dex) {
+     start_points[in_dex]:end_points[in_dex]
+ })  # end lapply
> look_backs[[1]]
> look_backs[[2]]
> # perform sapply() loop over look_backs list
> agg_regations <- sapply(look_backs,
+     function(look_back) {
+ x_ts <- price_s[look_back]
+ c(max=max(x_ts), min=min(x_ts))
+   })  # end sapply
> # coerce agg_regations into matrix and transpose it
> if (is.vector(agg_regations))
+   agg_regations <- t(agg_regations)
> agg_regations <- t(agg_regations)
> # coerce agg_regations into xts series
> agg_regations <- xts(agg_regations,
+     order.by=index(price_s[end_points]))
> head(agg_regations)
> # plot aggregations with custom line colors
> plot_theme <- chart_theme()
```

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 *endpoints*,

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 look_backs list
> agg_regations <- lapply(look_backs,
+     function(look_back) {
+ x_ts <- price_s[look_back]
+ c(max=max(x_ts), min=min(x_ts))
+     })  # end lapply
> # rbind list into single xts or matrix
> agg_regations <- rutils::do_call_rbind(agg_regations)
> # coerce agg_regations into xts series
> agg_regations <- xts(agg_regations,
+     order.by=index(price_s[end_points]))
> head(agg_regations)
> # plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> chart_Series(agg_regations, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(agg_regations),
+   bg="white", lty=c(1, 1), lwd=c(6, 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 *endpoints*,

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 *endpoints*:

- `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 rolling aggregations over end_po
> roll_agg <- function(x_ts, end_points, FUN, ...) {
+   len_gth <- NROW(end_points)
+ # start_points are single-period lag of end_points
+   start_points <- c(1, end_points[1:(len_gth-1)]+1)
+ # perform aggregations over look_backs list
+   agg_regations <- lapply(look_backs,
+     function(look_back) FUN(x_ts[look_back], ...))  # en
+ # rbind list into single xts or matrix
+   agg_regations <- rutils::do_call_rbind(agg_regations)
+   if (!is.xts(agg_regations))
+     agg_regations <-  # coerce agg_regations into xts se
+     xts(agg_regations, order.by=index(x_ts[end_points]))
+   agg_regations
+ }  # end roll_agg
> # apply sum() over end_points
> agg_regations <-
+   roll_agg(price_s, end_points=end_points, FUN=sum)
> agg_regations <-
+   period.apply(price_s, INDEX=end_points, FUN=sum)
> # benchmark the speed of aggregation functions
> summary(microbenchmark(
+   roll_agg=roll_agg(price_s, end_points=end_points, FUN=
+   period_apply=period.apply(price_s, INDEX=end_points, F
+   times=10))[, c(1, 4, 5)]
> agg_regations <- period.sum(price_s, INDEX=end_points)
> head(agg_regations)
```

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 *endpoints*, because they determine the *endpoints* from the calendar periods,

```
> # load package HighFreq
> library(HighFreq)
> # extract closing minutely prices
> price_s <- Cl(HighFreq::SPY["2012-02-01/2012-0
> # apply "mean" over daily periods
> agg_regations <- apply.daily(price_s, FUN=sum)
> head(agg_regations)
```

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 *endpoints* and a *look-back interval*,

The *startpoints* are defined as the *endpoints* 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*),

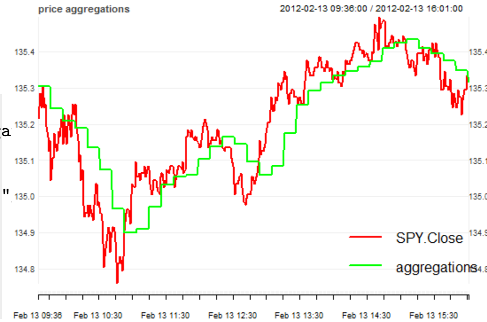
The variable `look_back` is equal to the number of end points in the *look-back interval*, while (`look_back - 1`) is equal to the number of intervals in the look-back,

```
> end_points <- # define end_points with beginning stub
+   n_row-look_back*num_agg + (0:num_agg)*look_back
> len_gth <- NROW(end_points)
> num_points <- 4  # number of end points in look-back int
> # start_points are multi-period lag of end_points
> start_points <- c(rep_len(1, num_points-1),
+   end_points[1:(len_gth-num_points+1)])
> # define list of look-back intervals for aggregations ov
> look_backs <- lapply(seq_along(end_points),
+   function(in_dex) {
+     start_points[in_dex]:end_points[in_dex]
+ })  # end lapply
> # perform lapply() loop over look_backs list
> agg_regations <- lapply(look_backs,
+     function(look_back) {
+ x_ts <- price_s[look_back]
+ c(max=max(x_ts), min=min(x_ts))
+     })  # end lapply
> # rbind list into single xts or matrix
> agg_regations <- rutils::do_call_rbind(agg_regations)
> # coerce agg_regations into xts series
> agg_regations <- xts(agg_regations,
+     order.by=index(price_s[end_points]))
> # plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("red", "green")
> chart_Series(agg_regations, theme=plot_theme,
+        name="price aggregations")
```

Extending Interval Aggregations

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

```
> tail(agg_regations, 22)
> agg_regations <- na.omit(zoo::na.locf(agg_rega
> # plot aggregations with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red", "
> chart_Series(agg_regations, theme=plot_theme,
+        name="price aggregations")
> legend("top", legend=colnames(agg_regations),
+   bg="white", lty=c(1, 1, 1), lwd=c(6, 6, 6),
+   col=plot_theme$col$line.col, bty="n")
```

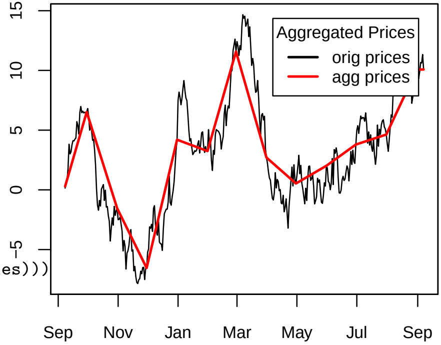


Performing Interval Aggregations of *zoo* Time Series

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,

```
> # create zoo time series of random returns
> in_dex <- Sys.Date() + 0:365
> zoo_series <-
+   zoo(rnorm(NROW(in_dex)), order.by=in_dex)
> # create monthly dates
> dates_agg <- as.Date(as.yearmon(index(zoo_series)))
> # perform monthly mean aggregation
> zoo_agg <- aggregate(zoo_series, by=dates_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)
> # extract aggregated zoo
> zoo_agg <- zoo_agg[index(zoo_series), 2]
```



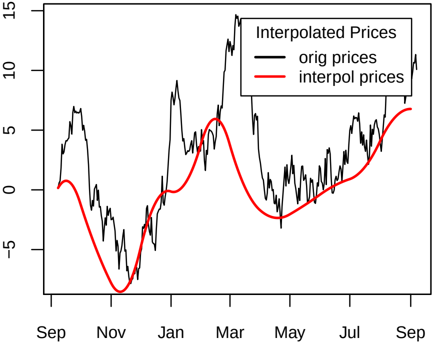

```
> # plot original and aggregated cumulative retu
> plot(cumsum(zoo_series), xlab="", ylab="")
> lines(cumsum(zoo_agg), lwd=2, col="red")
> # add legend
> legend("topright", inset=0.05, cex=0.8,
+  title="Aggregated Prices",
+  leg=c("orig prices", "agg prices"),
+  lwd=2, bg="white", col=c("black", "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=dates_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[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"))
```

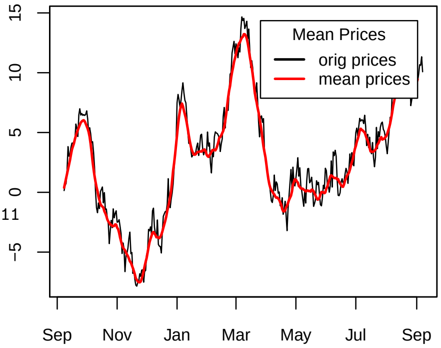


Performing Rolling Aggregations Over *zoo* Time Series

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

- `rollapply()` performing aggregations over a rolling (sliding) interval,
- `rollmean()` calculating rolling means,
- `rollmedian()` calculating rolling median,
- `rollmax()` calculating rolling 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, fromLast=TRUE)
> # extract mean zoo
> zoo_mean <- zoo_mean[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"))
```



The argument `align="right"` determines that aggregations are taken from the past,

Parallel Computing in R

Parallel Computing in R

Parallel computing means splitting a computing task into separate sub-tasks, and then simultaneously computing the sub-tasks on several computers or CPU cores,

There are many different packages that allow parallel computing in R, most importantly package *parallel*, and packages `foreach`, `doParallel`, and related packages:

http://cran.r-project.org/web/views/HighPerformanceComputing.html
http://blog.revolutionanalytics.com/high-performance-computing/
http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/

R Base Package *parallel*

The package *parallel* provides functions for parallel computing using multiple cores of CPUs,

The package *parallel* is part of the standard R distribution, so it doesn't need to be installed.

http://adv-r.had.co.nz/Profiling.html#parallelise
https://github.com/tobigithub/R-parallel/wiki/R-parallel-package-overview

Packages `foreach`, `doParallel`, and Related Packages

http://blog.revolutionanalytics.com/2015/10/updates-to-the-foreach-package-and-its-friends.html

Parallel Computing Using Package *parallel*

The package *parallel* provides functions for parallel computing using multiple cores of CPUs,

The package *parallel* is part of the standard R distribution, so it doesn't need to be installed,

Different functions from package *parallel* need to be called depending on the operating system (*Windows*, *Mac-OSX*, or *Linux*),

```
> library(parallel)  # load package parallel
> # get short description
> packageDescription("parallel")
> # load help page
> help(package="parallel")
> # list all objects in "parallel"
> ls("package:parallel")
```

Performing Parallel Loops Using Package *parallel*

Some computing tasks naturally lend themselves to parallel computing, like for example performing loops,

Different functions from package *parallel* need to be called depending on the operating system (*Windows*, *Mac-OSX*, or *Linux*),

The function `mclapply()` performs apply loops (similar to `lapply()`) using parallel computing on several CPU cores under *Mac-OSX* or *Linux*,

Under *Windows*, a cluster of `R` processes (one per each CPU core) need to be started first, by calling the function `makeCluster()`,

Mac-OSX and *Linux* don't require calling the function `makeCluster()`,

The function `parLapply()` is similar to `lapply()`, and performs apply loops under *Windows*, using parallel computing on several CPU cores,

The function `stopCluster()` stops the `R` processes running on several CPU cores,

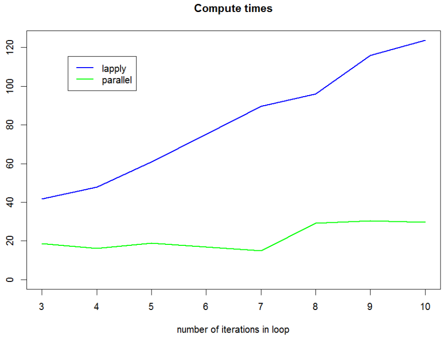
```
> library(parallel)  # load package parallel
> # calculate number of available cores
> num_cores <- detectCores() - 1
> # define function that pauses execution
> paws <- function(x, sleep_time) {
+   Sys.sleep(sleep_time)
+   x
+ }  # end paws
> # perform parallel loop under Mac-OSX or Linux
> paw_s <- mclapply(1:10, paws, mc.cores=num_cor
+           sleep_time=0.01)
> # initialize compute cluster under Windows
> clus_ter <- makeCluster(num_cores)
> # perform parallel loop under Windows
> paw_s <- parLapply(clus_ter, 1:10, paws,
+            sleep_time=0.01)
> library(microbenchmark)  # load package microb
> # compare speed of lapply versus parallel comp
> summary(microbenchmark(
+   l_apply=lapply(1:10, paws, sleep_time=0.01),
+   parl_apply=
+     parLapply(clus_ter, 1:10, paws, sleep_time
+   times=10)
+ )[, c(1, 4, 5)]
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

Computing Overhead of Parallel Computing

Parallel computing requires additional resources and time for distributing the computing tasks and collecting the output, which produces a computing overhead,

Therefore parallel computing can actually be slower for small computations, or for computations that can't be naturally separated into sub-tasks,

```
> # compare speed of lapply with parallel comput
> iter_ations <- 3:10
> compute_times <- sapply(iter_ations,
+   function(max_iterations, sleep_time) {
+     out_put <- summary(microbenchmark(
+ lapply=lapply(1:max_iterations, paws,
+                sleep_time=sleep_time),
+ parallel=parLapply(clus_ter, 1:max_iterations
+         paws, sleep_time=sleep_time),
+ times=10))[, c(1, 4)]
+     structure(out_put[, 2],
+         names=as.vector(out_put[, 1]))
+     }, sleep_time=0.01)
> compute_times <- t(compute_times)
> rownames(compute_times) <- iter_ations
```




```
> plot(x=rownames(compute_times),
+      y=compute_times[, "lapply"],
+      type="l", lwd=2, col="blue",
+      main="Compute times",
+      xlab="number of iterations in loop", ylab
+      ylim=c(0, max(compute_times[, "lapply"]))
> lines(x=rownames(compute_times),
+ y=compute_times[, "parallel"], lwd=2, col="gre
> legend(x="topleft", legend=colnames(compute_ti
+  inset=0.1, cex=1.0, bg="white",
+  lwd=2, lty=c(1, 1), col=c("blue", "green"))
```

Parallel Computing Over Matrices

Very often we need to perform time consuming calculations over columns of matrices,

The function parCapply() performs an apply loop over columns of matrices using parallel computing on several CPU cores,

```
> # define large matrix
> mat_rix <- matrix(rnorm(7*10^5), ncol=7)
> # define aggregation function over column of m
> agg_regate <- function(col_umn) {
+   out_put <- 0
+   for (in_dex in 1:NROW(col_umn))
+     out_put <- out_put + col_umn[in_dex]
+   out_put
+ }  # end agg_regate
> # perform parallel aggregations over columns o
> agg_regations <-
+   parCapply(clus_ter, mat_rix, agg_regate)
> # compare speed of apply with parallel computi
> summary(microbenchmark(
+   ap_ply=apply(mat_rix, MARGIN=2, agg_regate),
+   parl_apply=
+     parCapply(clus_ter, mat_rix, agg_regate),
+   times=10)
+ )[, c(1, 4, 5)]
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

Initializing Parallel Clusters Under *Windows*

Under *Windows* the child processes in the parallel compute cluster don't inherit data and objects from their parent process,

Therefore the required data must be either passed into `parLapply()` via the dots `"..."` argument, or by calling the function `clusterExport()`,

Objects from packages must be either referenced using the double-colon operator `"::"`, or the packages must be loaded in the child processes,

```
> ba_se <- 2
> # fails because child processes don't know ba_
> parLapply(clus_ter, 2:4,
+     function(exponent) ba_se^exponent)
> # ba_se passed to child via dots ... argument:
> parLapply(clus_ter, 2:4,
+     function(exponent, ba_se) ba_se^exponent,
+     ba_se=ba_se)
> # ba_se passed to child via clusterExport:
> clusterExport(clus_ter, "ba_se")
> parLapply(clus_ter, 2:4,
+     function(exponent) ba_se^exponent)
> # fails because child processes don't know zoo
> parSapply(clus_ter, c("VTI", "IEF", "DBC"),
+     function(sym_bol)
+       NROW(index(get(sym_bol, envir=rutils::en
> # zoo function referenced using "::" in child
> parSapply(clus_ter, c("VTI", "IEF", "DBC"),
+     function(sym_bol)
+       NROW(zoo::index(get(sym_bol, envir=rutil
> # package zoo loaded in child process:
> parSapply(clus_ter, c("VTI", "IEF", "DBC"),
+     function(sym_bol) {
+       stopifnot("package:zoo" %in% search() ||
+       NROW(index(get(sym_bol, envir=rutils::en
+     })  # end parSapply
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
```

Reproducible Parallel Simulations Under *Windows*

Simulations use pseudo-random number generators, and in order to perform reproducible results, they must set the *seed* value, so that the number generators produce the same sequence of pseudo-random numbers,

The function `set.seed()` initializes the random number generator by specifying the *seed* value, so that the number generator produces the same sequence of numbers for a given *seed* value,

But under *Windows* `set.seed()` doesn't initialize the random number generators of child processes, and they don't produce the same sequence of numbers,

The function `clusterSetRNGStream()` initializes the random number generators of child processes under *Windows*,

The function `set.seed()` does initialize the random number generators of child processes under *Mac-OSX* and *Linux*,

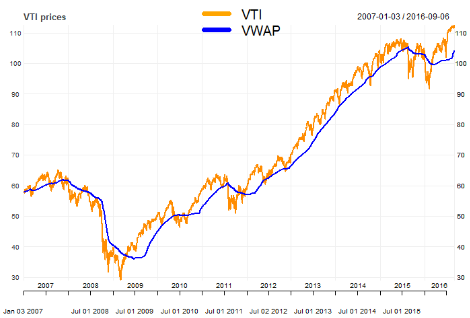
```
> library(parallel)  # load package parallel
> # calculate number of available cores
> num_cores <- detectCores() - 1
> # initialize compute cluster under Windows
> clus_ter <- makeCluster(num_cores)
> # set seed for cluster under Windows
> # doesn't work: set.seed(1121)
> clusterSetRNGStream(clus_ter, 1121)
> # perform parallel loop under Windows
> out_put <- parLapply(clus_ter, 1:70, rnorm, n=
> sum(unlist(out_put))
> # stop R processes over cluster under Windows
> stopCluster(clus_ter)
> # perform parallel loop under Mac-OSX or Linux
> out_put <- mclapply(1:10, rnorm, mc.cores=num_
```

Moving Average Technical Indicators

The Volume-Weighted Average Price (*VWAP*) is defined as the sum of prices multiplied by trading volumes, divided by the sum of volumes,

Moving averages (such as *VWAP*) are often used to define technical indicators (trading signals),

```
> # calculate open, close, and lagged prices
> op_en <- Op(rutils::env_etf$VTI)
> cl_ose <- Cl(rutils::env_etf$VTI)
> close_adj <- (cl_ose - as.numeric(cl_ose[1, ])
> prices_lag <- rutils::lag_it(cl_ose)
> # define aggregation interval and calculate VW
> look_back <- 150
> VTI_vwap <- HighFreq::roll_vwap(rutils::env_etf$VTI,
+         look_back=look_back)
> # calculate VWAP indicator
> in_dic <- sign(cl_ose - VTI_vwap)
> # determine dates right after VWAP has crosse
> trade_dates <- (rutils::diff_it(in_dic) != 0)
> trade_dates <- which(trade_dates) + 1
```



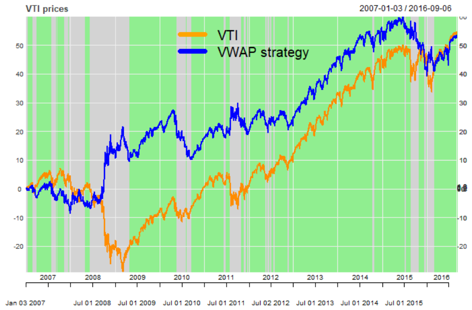

```
> # plot prices and VWAP
> chart_Series(x=cl_ose,
+   name="VTI prices", col="orange")
> add_TA(VTI_vwap, on=1, lwd=2, col="blue")
> legend("top", legend=c("VTI", "VWAP"),
+   bg="white", lty=1, lwd=6,
+   col=c("orange", "blue"), bty="n")
```

Moving Average Crossover Strategy

In a trend-following *Moving Average Crossover* strategy, when the current price crosses above the *VWAP*, then the strategy switches its position to long risk, and vice versa,

The strategy trades at the *Open* price in the next period after prices cross the *VWAP*, to reflect that in practice it's impossible to trade immediately,

```
> # calculate positions, either: -1, 0, or 1
> position_s <- NA*numeric(NROW(rutils::env_etf$
> position_s[1] <- 0
> position_s[trade_dates] <- in_dic[trade_dates]
> position_s <- na.locf(position_s)
> position_s <- xts(position_s, order.by=index(
> position_lagged <- rutils::lag_it(position_s)
> # calculate daily profits and losses
> pnl_s <- position_lagged*(cl_ose - prices_lag)
> pnl_s[trade_dates] <- position_lagged[trade_da
+   (op_en[trade_dates] - prices_lag[trade_date
+   position_s[trade_dates] *
+   (cl_ose[trade_dates] - op_en[trade_dates])
> # calculate annualized Sharpe ratio of strateg
> sqrt(252)*sum(pnl_s)/sd(pnl_s)/NROW(pnl_s)
```




```
> # plot prices and VWAP
> pnl_s <- xts(cumsum(pnl_s), order.by=index((ru
> close_adj <- (cl_ose - as.numeric(cl_ose[1, ])
> chart_Series(x=close_adj, name="VTI prices", c
> add_TA(pnl_s, on=1, lwd=2, col="blue")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=c("VTI", "VWAP strategy")
+  inset=0.1, bg="white", lty=1, lwd=6,
+  col=c("orange", "blue"), bty="n")
```

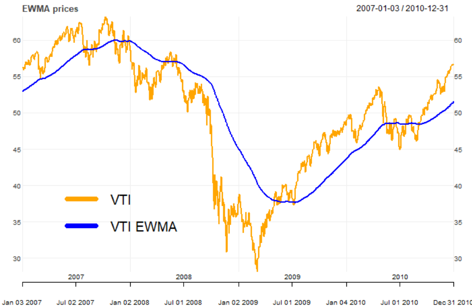
EWMA Price Technical Indicator

The *Exponentially Weighted Moving Average Price* (*EWMA*) is defined as the weighted average of prices over a rolling interval:

$$P_i^{EWMA} = (1 - \exp(-\lambda)) \sum_{j=0}^{\infty} \exp(-\lambda j) P_{i-j}$$

Where the decay parameter λ determines the rate of decay of the *EWMA* weights, with larger values of λ producing faster decay, giving more weight to recent prices, and vice versa,

```
> # select OHLC data
> oh_lc <- rutils::env_etf$VTI
> # calculate close prices
> cl_ose <- quantmod::Cl(oh_lc)
> close_adj <- (cl_ose - as.numeric(cl_ose[1, ])
> # define length for weights and decay paramete
> wid_th <- 251
> lamb_da <- 0.01
> # calculate EWMA prices
> weight_s <- exp(-lamb_da*1:wid_th)
> weight_s <- weight_s/sum(weight_s)
> ew_ma <- stats::filter(cl_ose, filter=weight_s, sides=1)
> ew_ma[1:(wid_th-1)] <- ew_ma[wid_th]
> ew_ma <- xts(cbind(cl_ose, ew_ma),
+         order.by=index(oh_lc))
> colnames(ew_ma) <- c("VTI", "VTI EWMA")
```




```
> # plot EWMA prices with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> chart_Series(ew_ma["2007/2010"], theme=plot_th
+         name="EWMA prices")
> legend("bottomleft", legend=colnames(ew_ma),
+  inset=0.1, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

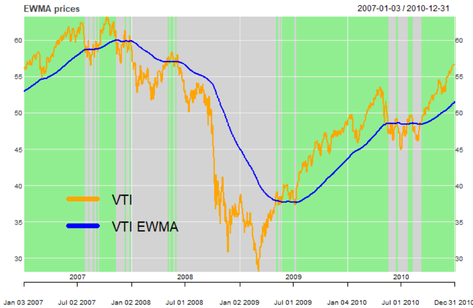
Simulating The *EWMA* Crossover Strategy

In a trend-following *EWMA Crossover* strategy, the risk position switches depending if the current price is above or below the *EWMA*,

If the current price crosses above the *EWMA*, then the strategy switches its risk position to a fixed unit of long risk, and if it crosses below, to a fixed unit of short risk,

The strategy holds the same position until the *EWMA* crosses over the current price (either from above or below), and then it switches its position,

The strategy is therefore always either in a long risk, or in a short risk position,




```
> # determine dates right after EWMA has crossed
> in_dic <- sign(cl_ose - ew_ma[, 2])
> trade_dates <- (rutils::diff_it(in_dic) != 0)
> trade_dates <- which(trade_dates) + 1
> # calculate positions, either: -1, 0, or 1
> position_s <- rep(NA_integer_, NROW(cl_ose))
> position_s[1] <- 0
> position_s[trade_dates] <-
+   rutils::lag_it(in_dic)[trade_dates]
> position_s <- na.locf(position_s)
> position_s <- xts(position_s, order.by=index(oh_lc))
```

```
> # plot EWMA prices with position shading
> chart_Series(ew_ma["2007/2010"], theme=plot_th
+        name="EWMA prices")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("bottomleft", legend=colnames(ew_ma),
+  inset=0.1, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Estimating the Transaction Costs of Trading

For institutional investors, the *bid-offer spread* (difference between the *offer* minus the *bid* prices) for liquid *stocks* and *ETFs* is often estimated to be about 10 basis points (bps),

In reality the *bid-offer spread* is not static and depends on many factors, such as market liquidity (trading volume), volatility, and time of day,

Broker commissions are an additional trading cost, but they depend on the size of the trades and on the type of investors, with institutional investors usually enjoying smaller commissions,

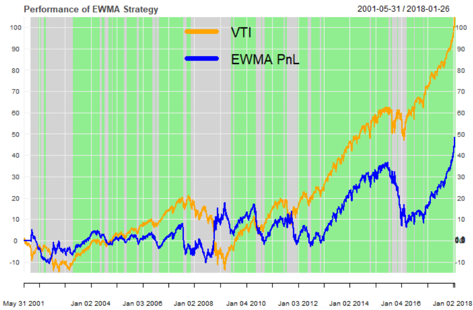
```
> # bid_offer is equal to 10 bps for liquid ETFs
> bid_offer <- 0.001
> # calculate open and lagged prices
> op_en <- Op(oh_lc)
> prices_lag <- rutils::lag_it(cl_ose)
> position_lagged <- rutils::lag_it(position_s)
> # calculate transaction costs
> cost_s <- 0.0*position_s
> cost_s[trade_dates] <- 0.5*bid_offer*abs(posit
```

Performance of *EWMA* Crossover Strategy

The strategy trades at the *Open* price on the next day after prices cross the *EWMA*, since in practice it may not be possible to trade immediately,

The Profit and Loss (*PnL*) on a trade date is the sum of the realized *PnL* from closing the old position, plus the unrealized *PnL* after opening the new position,

```
> # calculate daily profits and losses
> re_turns <- position_lagged*(cl_ose - prices_l
> re_turns[trade_dates] <-
+   position_lagged[trade_dates] *
+   (op_en[trade_dates] - prices_lag[trade_date
+   position_s[trade_dates] *
+   (cl_ose[trade_dates] - op_en[trade_dates]) -
+   cost_s
> # calculate annualized Sharpe ratio of strateg
> sqrt(252)*sum(re_turns)/sd(re_turns)/NROW(re_t
> pnl_s <- cumsum(re_turns)
> pnl_s <- cbind(close_adj, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
```




```
> # plot EWMA PnL with position shading
> chart_Series(pnl_s, theme=plot_theme,
+        name="Performance of EWMA Strategy")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(pnl_s),
+  inset=0.05, bg="white", lty=1, lwd=6,
+  col=plot_theme$col$line.col, bty="n")
```

Function for *EWMA* Crossover Strategy

The *EWMA* strategy can be simulated by a single function, which allows the analysis of its performance depending on its parameters,

The function simu_ewma() performs a simulation of the *EWMA* strategy, given an *OHLC* time series of prices, and a decay parameter λ,

The function simu_ewma() returns the *EWMA* strategy positions and returns, in a two-column *xts* time series,

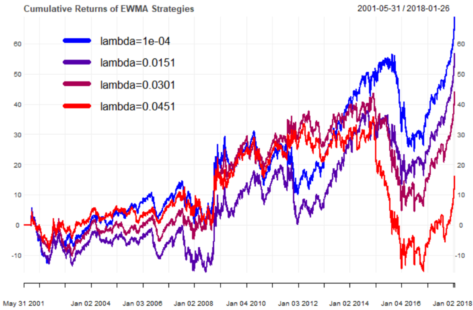
```
> simu_ewma <- function(oh_lc, lamb_da=0.01, wid_th=251, b
+   # calculate EWMA prices
+   weight_s <- exp(-lamb_da*1:wid_th)
+   weight_s <- weight_s/sum(weight_s)
+   cl_ose <- quantmod::Cl(oh_lc)
+   ew_ma <- stats::filter(as.numeric(cl_ose), filter=weig
+   ew_ma[1:(wid_th-1)] <- ew_ma[wid_th]
+   # determine dates right after EWMA has crossed prices
+   in_dic <- tre_nd*xts::xts(sign(as.numeric(cl_ose) - ew
+   trade_dates <- (rutils::diff_it(in_dic) != 0)
+   trade_dates <- which(trade_dates) + 1
+   trade_dates <- trade_dates[trade_dates<NROW(oh_lc)]
+   # calculate positions, either: -1, 0, or 1
+   position_s <- rep(NA_integer_, NROW(cl_ose))
+   position_s[1] <- 0
+   position_s[trade_dates] <- rutils::lag_it(in_dic)[trad
+   position_s <- xts::xts(na.locf(position_s), order.by=i
+   op_en <- quantmod::Op(oh_lc)
+   prices_lag <- rutils::lag_it(cl_ose)
+   position_lagged <- rutils::lag_it(position_s)
+   # calculate transaction costs
+   cost_s <- 0.0*position_s
+   cost_s[trade_dates] <- 0.5*bid_offer*abs(position_lagg
+   # calculate daily profits and losses
+   re_turns <- position_lagged*(cl_ose - prices_lag)
+   re_turns[trade_dates] <- position_lagged[trade_dates]
+   out_put <- cbind(position_s, re_turns)
+   colnames(out_put) <- c("positions", "returns")
+   out_put
```

Simulating Multiple Trend-following *EWMA* Strategies

Multiple *EWMA* strategies can be simulated by calling the function `simu_ewma()` in a loop over a vector of λ parameters,

But `simu_ewma()` returns an *xts* time series, and `sapply()` cannot merge *xts* time series together,

So instead the loop is performed using `lapply()` which returns a list of *xts*, and the list is merged into a single *xts* using functions `rutils::do_call()` and `cbind()`,




```
> source("C:/Develop/R/lecture_slides/scripts/ew
> lamb_das <- seq(0.0001, 0.05, 0.005)
> # perform lapply() loop over lamb_das
> re_turns <- lapply(lamb_das, function(lamb_da) {
+   # simulate EWMA strategy and calculate re_turns
+   simu_ewma(oh_lc=oh_lc, lamb_da=lamb_da,
+       wid_th=wid_th)[, "returns"]
+ })  # end lapply
> re_turns <- rutils::do_call(cbind, re_turns)
> colnames(re_turns) <- paste0("lambda=", lamb_
```

```
> # plot EWMA strategies with custom line colors
> column_s <- seq(1, NCOL(re_turns), by=3)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NROW(colu
> chart_Series(cumsum(re_turns[, column_s]),
+   theme=plot_theme, name="Cumulative Returns o
> legend("topleft", legend=colnames(re_turns[, c
+   inset=0.1, bg="white", cex=0.8, lwd=rep(6, N
+   col=plot_theme$col$line.col, bty="n")
```

Simulating *EWMA* Strategies Using Parallel Computing

Simulating *EWMA* strategies naturally lends itself to parallel computing, since the simulations are independent from each other,

The function `parLapply()` is similar to `lapply()`, and performs apply loops under *Windows*, using parallel computing on several CPU cores,

The resulting list of time series can then be collapsed into a single *xts* series using the functions `rutils::do_call()` and `cbind()`,

```
> # initialize compute cluster under Windows
> library(parallel)
> clus_ter <- makeCluster(detectCores()-1)
> clusterExport(clus_ter,
+   varlist=c("oh_lc", "wid_th", "simu_ewma"))
> # perform parallel loop over lamb_das under Wi
> re_turns <- parLapply(clus_ter, lamb_das,
+         function(lamb_da) {
+   library(quantmod)
+   # simulate EWMA strategy and calculate re_tu
+   simu_ewma(oh_lc=oh_lc,
+     lamb_da=lamb_da, wid_th=wid_th)[, "returns
+ })  # end parLapply
> # perform parallel loop over lamb_das under Ma
> re_turns <- mclapply(lamb_das,
+         function(lamb_da) {
+   library(quantmod)
+   # simulate EWMA strategy and calculate re_tu
+   simu_ewma(oh_lc=oh_lc,
+     lamb_da=lamb_da, wid_th=wid_th)[, "returns
+ })  # end mclapply
> stopCluster(clus_ter)  # stop R processes over
> re_turns <- rutils::do_call(cbind, re_turns)
> colnames(re_turns) <- paste0("lambda=", lamb_d
```

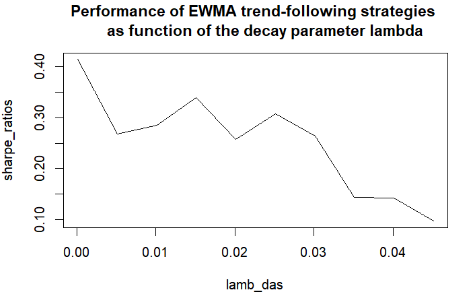
Performance of Trend-following *EWMA* Strategies

The *Sharpe* ratios of *EWMA* strategies with different λ parameters can be calculated by performing an `sapply()` loop over the *columns* of returns,

`sapply()` treats the columns of *xts* time series as list elements, and loops over the columns,

Performing loops in R over the *columns* of returns is acceptable, but R loops over the *rows* of returns should be avoided,

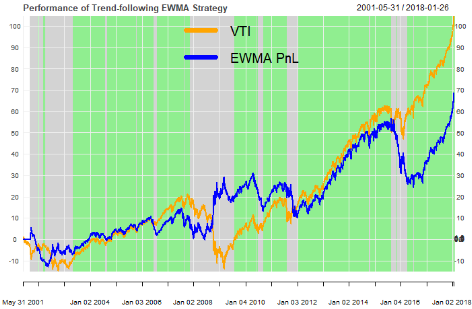
The performance of trend-following *EWMA* strategies depends on the λ parameter, with larger λ parameters performing worse than smaller ones,




```
> sharpe_ratios <- sqrt(252)*sapply(re_turns, function(x_ts) {
+   # calculate annualized Sharpe ratio of strategy returns
+   sum(x_ts)/sd(x_ts)
+ })/NROW(re_turns)  # end sapply
> plot(x=lamb_das, y=sharpe_ratios, t="l",
+      main="Performance of EWMA trend-following strategies
+      as function of the decay parameter lambda")
> trend_returns <- re_turns
> trend_sharpe_ratios <- sharpe_ratios
```

Optimal Trend-following *EWMA* Strategy

The best performing trend-following *EWMA* strategy has a relatively small λ parameter, corresponding to slower weight decay (giving more weight to past prices), and producing less frequent trading,




```
> # simulate best performing strategy
> ewma_trend <- simu_ewma(oh_lc=oh_lc,
+   lamb_da=lamb_das[which.max(sharpe_ratios)],
+   wid_th=wid_th)
> position_s <- ewma_trend[, "positions"]
> pnl_s <- cumsum(ewma_trend[, "returns"])
> pnl_s <- cbind(close_adj, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
> # plot EWMA PnL with position shading
> plot_theme$col$line.col <- c("orange", "blue")
> chart_Series(pnl_s, theme=plot_theme,
+        name="Performance of Trend-following EWMA Strategy")
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(pnl_s),
+   inset=0.05, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

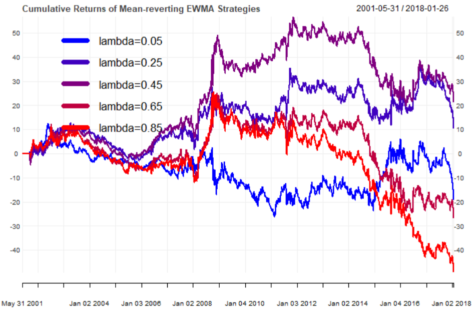
Simulating Multiple Mean-reverting *EWMA* Strategies

Multiple *EWMA* strategies can be simulated by calling the function simu_ewma() in a loop over a vector of λ parameters,

But simu_ewma() returns an *xts* time series, and sapply() cannot merge *xts* time series together,

So instead the loop is performed using lapply() which returns a list of *xts*, and the list is merged into a single *xts* using functions rutils::do_call() and cbind(),

```
> source("C:/Develop/R/lecture_slides/scripts/ew
> lamb_das <- seq(0.05, 1.0, 0.05)
> # perform lapply() loop over lamb_das
> re_turns <- lapply(lamb_das, function(lamb_da) {
+   # simulate EWMA strategy and calculate re_turns
+   simu_ewma(oh_lc=oh_lc, lamb_da=lamb_da,
+       wid_th=wid_th, tre_nd=(-1))[, "returns"]
+ })  # end lapply
> re_turns <- rutils::do_call(cbind, re_turns)
> colnames(re_turns) <- paste0("lambda=", lamb_d
```




```
> # plot EWMA strategies with custom line colors
> column_s <- seq(1, NCOL(re_turns), by=4)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NROW(colu
> chart_Series(cumsum(re_turns[, column_s]),
+   theme=plot_theme, name="Cumulative Returns o
> legend("topleft", legend=colnames(re_turns[, c
+   inset=0.1, bg="white", cex=0.8, lwd=rep(6, N
+   col=plot_theme$col$line.col, bty="n")
```

Performance of Mean-reverting *EWMA* Strategies

The *Sharpe* ratios of *EWMA* strategies with different λ parameters can be calculated by performing an `sapply()` loop over the *columns* of returns,

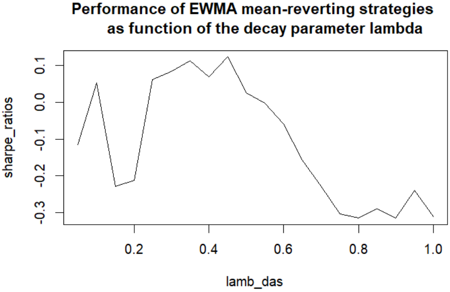
`sapply()` treats the columns of *xts* time series as list elements, and loops over the columns,

Performing loops in R over the *columns* of returns is acceptable, but R loops over the *rows* of returns should be avoided,

The performance of mean-reverting *EWMA* strategies depends on the λ parameter, with performance decreasing for very small or very large λ parameters,

For too large λ parameters, the trading frequency is too high, causing high transaction costs,

For too small λ parameters, the trading frequency is too low, causing the strategy to miss profitable trades,




```
> sharpe_ratios <- sqrt(252)*sapply(re_turns, fu
+   # calculate annualized Sharpe ratio of strat
+   sum(x_ts)/sd(x_ts)
+ })/NROW(re_turns)  # end sapply
> plot(x=lamb_das, y=sharpe_ratios, t="l",
+      main="Performance of EWMA mean-reverting
+      as function of the decay parameter lambda
> revert_returns <- re_turns
> revert_sharpe_ratios <- sharpe_ratios
```

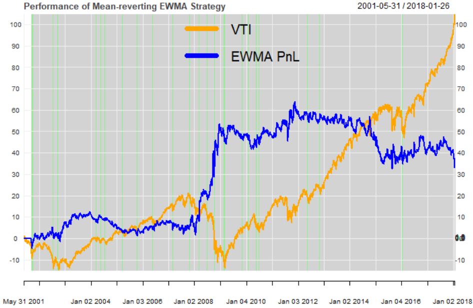
Optimal Mean-reverting *EWMA* Strategy

Reverting the rules of the trend-following *EWMA* strategy creates a mean-reverting strategy,

The best performing mean-reverting *EWMA* strategy has a relatively large λ parameter, corresponding to faster weight decay (giving more weight to recent prices), and producing more frequent trading,

But a too large λ parameter also causes very high trading frequency, and high transaction costs,

```
> # simulate best performing strategy
> ewma_revert <- simu_ewma(oh_lc=oh_lc,
+   lamb_da=lamb_das[which.max(sharpe_ratios)],
+   wid_th=wid_th, tre_nd=(-1))
> position_s <- ewma_revert[, "positions"]
> pnl_s <- cumsum(ewma_revert[, "returns"])
> pnl_s <- cbind(close_adj, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
```

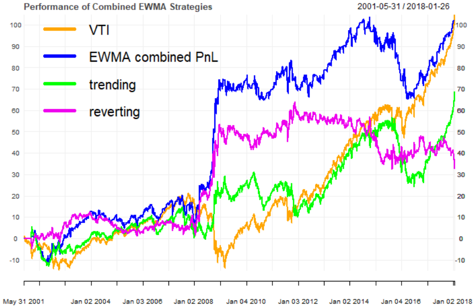



```
> # plot EWMA PnL with position shading
> plot_theme$col$line.col <- c("orange", "blue")
> chart_Series(pnl_s, theme=plot_theme,
+         name="Performance of Mean-reverting EWM
> add_TA(position_s > 0, on=-1,
+  col="lightgreen", border="lightgreen")
> add_TA(position_s < 0, on=-1,
+  col="lightgrey", border="lightgrey")
> legend("top", legend=colnames(pnl_s),
+   inset=0.05, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Combining Trend-following and Mean-reverting Strategies

The returns of trend-following and mean-reverting strategies are usually negatively correlated to each other, so combining them can achieve significant diversification of risk,

```
> # calculate correlation between trend-followin
> trend_ing <- ewma_trend[, "returns"]
> colnames(trend_ing) <- "trend"
> revert_ing <- ewma_revert[, "returns"]
> colnames(revert_ing) <- "revert"
> close_rets <- rutils::diff_it(cl_ose)
> corr_matrix <- cor(cbind(trend_ing, revert_ing
> corr_matrix
> # calculate combined strategy
> com_bined <- trend_ing + revert_ing
> colnames(com_bined) <- "combined"
> # calculate annualized Sharpe ratio of strateg
> sqrt(252)*sapply(
+   cbind(close_rets, trend_ing, revert_ing, com
+   function(x_ts) sum(x_ts)/sd(x_ts))/NROW(com_
> pnl_s <- cumsum(com_bined)
> pnl_s <- cbind(close_adj, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA combined Pnl
```



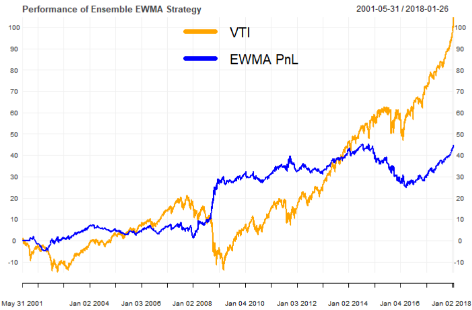

```
> chart_Series(pnl_s, theme=plot_theme,
+        name="Performance of Combined EWMA Stra
> add_TA(cumsum(trend_ing), on=1, lwd=2, col="gr
> add_TA(cumsum(revert_ing), on=1, lwd=2, col="m
> legend("topleft", legend=c(colnames(pnl_s), "t
+  inset=0.05, bg="white", lty=rep(1, 4), lwd=re
+  col=c(plot_theme$col$line.col, "green", "mage
```

Ensemble of *EWMA* Strategies

Instead of selecting the best performing *EWMA* strategy, one can choose a weighted average of strategies (ensemble), which corresponds to allocating positions according to the weights,

The weights can be chosen to be proportional to the Sharpe ratios of the *EWMA* strategies,

```
> sharpe_ratios <- c(trend_sharpe_ratios, revert
> weight_s <- sharpe_ratios
> weight_s[weight_s<0] <- 0
> weight_s <- weight_s/sum(weight_s)
> re_turns <- cbind(trend_returns, revert_return
> avg_returns <- re_turns %*% weight_s
> avg_returns <- xts(avg_returns, order.by=index
> pnl_s <- cumsum(avg_returns)
> pnl_s <- cbind(close_adj, pnl_s)
> colnames(pnl_s) <- c("VTI", "EWMA PnL")
> # plot EWMA PnL without position shading
> chart_Series(pnl_s, theme=plot_theme,
+        name="Performance of Ensemble EWMA Strategy")
> legend("top", legend=colnames(pnl_s),
+   inset=0.05, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```



Aggregations Over Look-back and Look-forward Intervals

Overlapping aggregations can be specified by a vector of *look-back* intervals attached at *end points*,

For example, we may specify aggregations at monthly *end points*, over overlapping 12-month *look-back* intervals,

The variable `look_back` is equal to the number of *end points* in the *look-back* interval,

The *start points* are the *end points* lagged by the length of the *look-back* interval,

The *look-back* intervals are spanned by the vectors of *start points* and *end points*,

Non-overlapping aggregations can also be calculated over a list of *look-forward* intervals (`look_fwds`),

The *look-back* intervals should not overlap with the *look-forward* intervals, in order to avoid data snooping,

```
> # end of month end_points
> end_points <- rutils::calc_endpoints(re_turns,
+              inter_val="months")
> len_gth <- NROW(end_points)
> # define 12-month look-back interval
> look_back <- 12
> # start_points are end_points lagged by look-b
> start_points <- c(rep_len(1, look_back-1),
+   end_points[1:(len_gth-look_back+1)])
> # Perform loop over end_points and calculate a
> # agg_fun <- function(re_turns) sum(re_turns)/
> agg_fun <- function(re_turns) sum(re_turns)
> back_aggs <- sapply(1:(len_gth-1), function(it
+   sapply(re_turns[start_points[it_er]:end_poin
+ })  # end sapply
> back_aggs <- t(back_aggs)
> # define forward (future) endpoints
> fwd_points <- end_points[c(2:len_gth, len_gth)
> fwd_rets <- sapply(1:(len_gth-1), function(it_
+   sapply(re_turns[(end_points[it_er]+1):fwd_po
+ })  # end sapply
> fwd_rets <- t(fwd_rets)
```

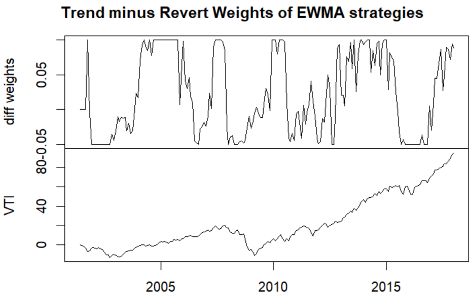
EWMA Momentum Portfolio Weights

In *momentum* strategies the portfolio weights are proportional to the past performance of the assets,

Constraints are also be applied to the weights to limit the portfolio *leverage* or its market *beta*,

To limit the portfolio leverage, the weights can be scaled so that the sum of their absolute values is equal to 1,

The weights can also be de-meaned (their sum is equal to zero), to create long-short portfolios with small betas,




```
> # calculate weight_s proportional to back_aggs
> weight_s <- back_aggs
> weight_s[weight_s<0] <- 0
> # scale weight_s so their sum is equal to 1
> weight_s <- weight_s/rowSums(weight_s)
> # set NA values to zero
> weight_s[is.na(weight_s)] <- 0
> sum(is.na(weight_s))
> in_dex <- index(re_turns[end_points[-len_gth]]
> trend_weights <- rowMeans(weight_s[, 1:NCOL(t
> revert_weights <- rowMeans(weight_s[, -(1:NCO
> diff_weights <- xts(trend_weights-revert_weig
> close_adj <- (cl_ose - as.numeric(cl_ose[1, ]))
> # de-mean weight_s so their sum is equal to 0
```

```
> # plot the mean weights of EWMA Strategies
> zoo::plot.zoo(cbind(diff_weights,
+   close_adj[end_points[-len_gth]]),
+   oma = c(3, 0, 3, 0), mar = c(0, 4, 0, 1),
+   xlab=NULL, ylab=c("diff weights", "VTI"),
+   main="Trend minus Revert Weights of EWMA str
> best_worst <- xts(cbind(bes_t, wors_t), order.
> zoo::plot.zoo(best_worst,
+   oma = c(3, 0, 3, 0), mar = c(0, 4, 0, 1),
+   xlab=NULL, ylab=c("best EWMA", "worst EWMA")
+   main="Best and Worst EWMA strategies")
```

Backtesting the *EWMA Momentum* Strategy

Backtesting is the testing of a forecasting model using historical data,

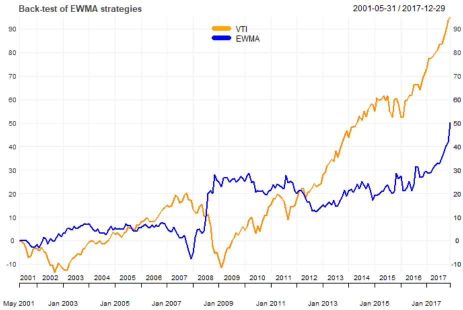
Backtesting is a type of *cross-validation* applied to time series data,

Backtesting is performed by *training* the model on past data defined by the *look-back* intervals, and then *testing* the model on future data defined by the *look-forward* intervals,

The hypothetical out-of-sample *momentum* strategy returns can be calculated by multiplying the `fwd_rets` by the forecast *ETF* portfolio weights,

The *training* data is specified by the *look-back* intervals (`past_aggs`), and the forecasts are applied to the future data defined by the *look-forward* intervals (`fwd_rets`),

```
> # calculate backtest returns
> pnl_s <- rowSums(weight_s * fwd_rets)
> pnl_s <- xts(pnl_s, order.by=in_dex)
> colnames(pnl_s) <- "ewma momentum"
> close_rets <- rutils::diff_it(cl_ose[in_dex])
> cor(cbind(pnl_s, close_rets))
> pnl_s <- cumsum(pnl_s)
```




```
> # plot the backtest
> chart_Series(x=close_adj[end_points[-len_gth]]
+   name="Back-test of EWMA strategies", col="or
> add_TA(pnl_s, on=1, lwd=2, col="blue")
> legend("top", legend=c("VTI", "EWMA"),
+  inset=0.1, bg="white", lty=1, lwd=6,
+  col=c("orange", "blue"), bty="n")
> # shad_e <- xts(index(pnl_s) < as.Date("2008-0
> # add_TA(shad_e, on=-1, col="lightgrey", borde
> # text(x=7, y=0, labels="warmup period")
```

Backtesting Functional for *Momentum* Strategies

The functional `back_test()` performs a *back-test* simulation,

Backtesting is a type of *cross-validation* applied to time series data, and consists of:

- aggregating past historical data (returns, etc.) into performance statistics (Sharpe ratios, etc.),
- applying a trading rule and forming a portfolio (*training* the model),
- *testing* the portfolio performance *out-of-sample* on future data,

```
> # define back-test functional
> back_test <- function(re_turns, price_s, agg_f
+     look_back=12, re_balance="months", bid_off
+     end_points=rutils::calc_endpoints(re_turns
+     with_weights=FALSE, ...) {
+   stopifnot("package:quantmod" %in% search() |
+   # define start_points and forward (future) e
+   len_gth <- NROW(end_points)
+   n_col <- NCOL(re_turns)
+   start_points <- c(rep_len(1, look_back-1), e
+   fwd_points <- end_points[c(2:len_gth, len_gt
+   # Perform loop over end_points and calculate
+   agg_s <- sapply(1:(len_gth-1), function(it_e
+     c(back_aggs=sapply(re_turns[start_points[i
+     fwd_rets=sapply(re_turns[(end_points[it_er
+   })  # end sapply
+   agg_s <- t(agg_s)
+   # Select aggregations over look-back and loo
+   back_aggs <- agg_s[, 1:n_col]
+   fwd_rets <- agg_s[, n_col+1:n_col]
+   # Calculate portfolio weights
+   weight_s <- back_aggs/rowSums(abs(back_aggs)
+   weight_s[is.na(weight_s)] <- 0
+   colnames(weight_s) <- colnames(re_turns)
+   # Calculate profits and losses
+   end_points <- end_points[-len_gth]
+   price_s <- price_s[end_points, ]
+   pnl_s <- rowSums(weight_s * fwd_rets / price
+   pnl_s <- xts(pnl_s, index(re_turns[end_point
```

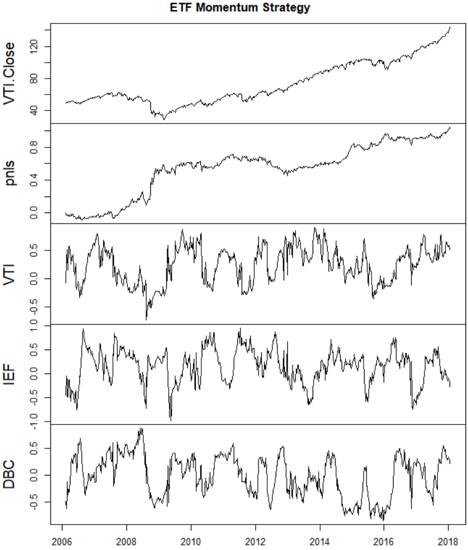
Backtesting the *Momentum* Strategy for an *ETF* Portfolio

The *momentum* strategy can applied to a portfolio of *ETFs*,

The hypothetical out-of-sample *momentum* strategy returns can be calculated by multiplying the fwd_rets by the forecast *ETF* portfolio weights,

The *training* data is specified by the *look-back* intervals (past_aggs), and the forecasts are applied to the future data defined by the *look-forward* intervals (fwd_rets),

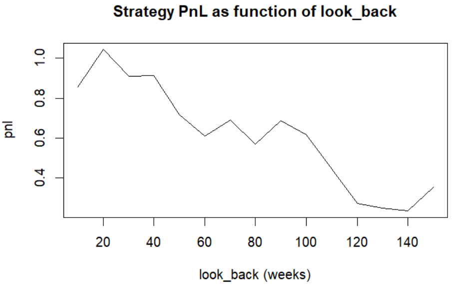
```
> # Calculate ETF prices and simple returns
> sym_bols <- c("VTI", "IEF", "DBC")
> price_s <- rutils::env_etf$price_s[, sym_bols]
> price_s <- zoo::na.locf(price_s)
> price_s <- na.omit(price_s)
> re_turns <- rutils::diff_it(price_s)
> # Perform back-test
> agg_fun <-
+   function(re_turns) sum(re_turns)/sd(re_turns
> pnl_s <- back_test(re_turns=re_turns, price_s=
+   re_balance="weeks", look_back=20, agg_fun=ag
+   with_weights=TRUE)
> zoo::plot.zoo(cbind(cl_ose[index(pnl_s)], pnl_
+   oma = c(3, 1, 3, 0), mar = c(0, 4, 0, 1), nc
+   xlab=NULL, main="ETF Momentum Strategy")
```



Parameter Optimization

The performance of the *momentum* strategy depends on the length of the *look-back interval* used for calculating past performance,

```
> look_backs <- seq(10, 150, by=10)
> pro_files <- sapply(look_backs, function(x) {
+   last(back_test(re_turns=re_turns, price_s=pr
+     re_balance="weeks", look_back=x, agg_fun=a
+ })  # end sapply
> plot(y=pro_files, x=look_backs, t="l",
+   main="Strategy PnL as function of look_back"
+   xlab="look_back (weeks)", ylab="pnl")
```

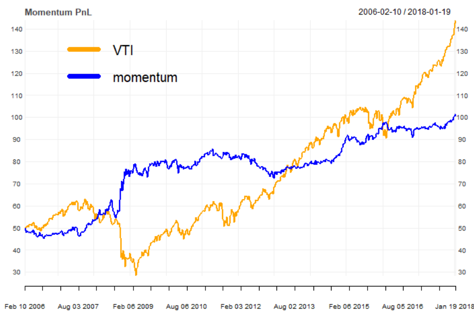


Momentum Strategy Performance

The hypothetical out-of-sample *momentum* strategy returns can be calculated by multiplying the `fwd_rets` by the forecast *ETF* portfolio weights,

The *training* data is specified by the *look-back* intervals (past_aggs), and the forecasts are applied to the future data defined by the *look-forward* intervals (`fwd_rets`),

```
> # bind model returns with VTI
> da_ta <- as.numeric(cl_ose[index(pnl_s)][1, 1]
> da_ta <- cbind(cl_ose[index(pnl_s)], da_ta*pnl
> colnames(da_ta) <- c("VTI", "momentum")
```



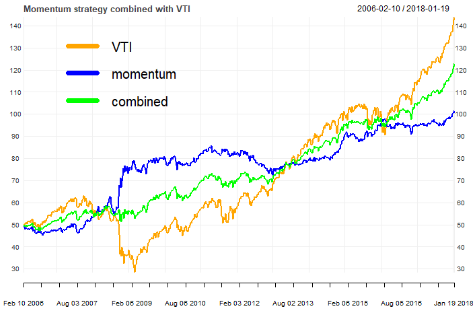

```
> # plot momentum strategy with VTI
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> chart_Series(da_ta, theme=plot_theme, lwd=c(2,
+         name="Momentum PnL")
> legend("topleft", legend=colnames(da_ta),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Combining the *Momentum* and Static Strategies

The *momentum* strategy has attractive returns compared to a static buy-and-hold strategy,

But the *momentum* strategy suffers from draw-downs called *momentum crashes*, especially after the market rallies from a sharp-sell-off,

This suggests that combining the *momentum* strategy with a static buy-and-hold strategy can achieve significant diversification of risk,




```
> # combine momentum strategy with static
> da_ta <- cbind(da_ta, 0.5* (da_ta[, "VTI"] + d
> colnames(da_ta) <- c("VTI", "momentum", "combi
> # calculate strategy annualized Sharpe ratios
> sapply(da_ta, function(cumu_lative) {
+   x_ts <- na.omit(diff(log(cumu_lative)))
+   sqrt(252)*sum(x_ts)/sd(x_ts)/NROW(x_ts)
+ })  # end sapply
> # calculate strategy correlations
> cor(na.omit(diff(log(da_ta))))
```

```
> # plot momentum strategy combined with VTI
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue",
> chart_Series(da_ta, theme=plot_theme,
+          name="Momentum strategy combined with V
> legend("topleft", legend=colnames(da_ta),
+   inset=0.1, bg="white", lty=c(1, 1, 1), lwd=c
+   col=plot_theme$col$line.col, bty="n")
```

Momentum Strategy Versus the *All-Weather* Portfolio

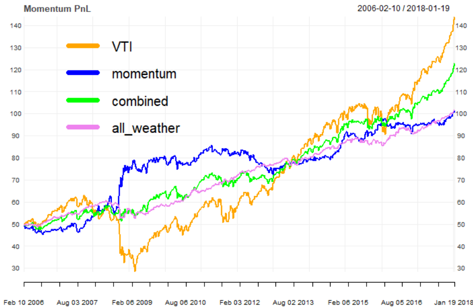
The *All-Weather* portfolio is a static portfolio of bonds (55%), stocks (30%), and commodities and precious metals (15%), and was designed by Bridgewater Associates, the largest hedge fund in the world:

https://www.bridgewater.com/research-library/the-all-weather-strategy/

http://www.nasdaq.com/article/remember-the-allweather-portfolio-its-having-a-killer-yea

The three different asset classes (bonds, stocks, commodities) provide positive returns under different economic conditions (recession, expansion, inflation),

```
> # Define all-weather symbols and weights
> weight_s <- c(0.30, 0.55, 0.15)
> all_weather <- re_turns %*% weight_s
> all_weather <- cumsum(all_weather)
> all_weather <- xts(all_weather, index(re_turns
> all_weather <- all_weather + as.numeric(cl_ose
> colnames(all_weather) <- "all_weather"
> # combine momentum strategy with all-weather
> da_ta <- cbind(da_ta, all_weather)
> # calculate strategy annualized Sharpe ratios
> sapply(da_ta, function(cumu_lative) {
+   x_ts <- na.omit(diff(log(cumu_lative)))
+   sqrt(252)*sum(x_ts)/sd(x_ts)/NROW(x_ts)
```




```
> # plot momentum strategy, combined, and all-we
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue",
> chart_Series(da_ta, theme=plot_theme, lwd=2, n
> legend("topleft", legend=colnames(da_ta),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

The combination of bonds, stocks, and commodities in the *All-Weather* portfolio is designed to provide positive returns under most economic conditions, without the costs of trading,

Homework Assignment

Required

- Read all the lecture slides in `FRE7241_Lecture_2.pdf`, and run all the code in `FRE7241_Lecture_2.R`,

Recommended

Read the following sections in the file `numerical_analysis.pdf`:

- *Numerical Calculations*,
- *Optimizing* `R` *Code for Speed and Memory Usage*,
- *Writing Fast* `R` *Code Using Vectorized Operations*,
- Run the code corresponding to the above sections from `numerical_analysis.R`

Read the following sections in the file `R_environment.pdf`:

- *Environments in* `R`,
- *Data Input and Output*,
- Run the code corresponding to the above sections from `R_environment.R`