FRE6871 R in Finance

Lecture#7, Spring 2023

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

May 8, 2023

Date Objects

R has a Date class for date objects (but without time).

The function as.Date() parses character strings and coerces numeric objects into Date objects.

R stores Date objects as the number of days since the *epoch* (January 1, 1970).

The function difftime() calculates the difference between Date objects, and returns a time interval object of class difftime.

The "+" and "-" arithmetic operators and the "<" and ">" logical comparison operators are overloaded to allow these operations directly on Date objects.

numeric *year-fraction* dates can be coerced to Date objects using the functions attributes() and structure().

```
> Sys.Date()  # Get today's date
> as.Date(1e3)  # Coerce numeric into date object
> datetime <- as.Date("2014-07-14")  # "%Y-%m-%d" or "%Y/%m/%d"
> datetime
> class(datetime)  # Date object
> as.Date("07-14-2014", "%m-%d-%Y")  # Specify format
> datetime + 20  # Add 20 days
> # Extract internal representation to integer
> as.numeric(datetime)
> date_old <- as.Date("07/14/2013", "%m/%d/%Y")
> date_old
> # Difference between dates
> difftime(datetime, date_old, units="weeks")
> weekdays(datetime)  # Get day of the week
> # Coerce numeric into date-times
> datetime <- 0
> attributes(datetime) <- list(class="Date")
> datetime  # "Date" object
> structure(0, class="Date")  # "Date" object
> structure(10000.25, class="Date")
```

POSIXct Date-time Objects

The POSIXct class in R represents *date-time* objects, that can store both the date and time.

The *clock time* is the time (number of hours, minutes and seconds) in the local *time zone*.

The *moment of time* is the *clock time* in the UTC *time zone*.

POSIXct objects are stored as the number of seconds that have elapsed since the *epoch* (January 1, 1970) in the UTC *time zone*.

POSIXct objects are stored as the *moment of time*, but are printed out as the *clock time* in the local *time zone*.

A *clock time* together with a *time zone* uniquely specifies a *moment of time*.

The function as.POSIXct() can parse a character string (representing the *clock time*) and a *time zone* into a POSIXct object.

POSIX is an acronym for "Portable Operating System Interface".

```
> datetime <- Sys.time()  # Get today's date and time
> datetime
> class(datetime)  # POSIXct object
> # POSIXct stored as integer moment of time
> as.numeric(datetime)
> # Parse character string "%Y-%m-%d %H:%M:%S" to POSIXct object
> datetime <- as.POSIXct("2014-07-14 13:30:10")
> # Different time zones can have same clock time
> as.POSIXct("2014-07-14 13:30:10", tz="America/New_York")
> as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # Format argument allows parsing different date-time string format
> as.POSIXct("07/14/2014 13:30:10", format="%m/%d/%Y %H:%M:%S",
+      tz="America/New_York")
```

Operations on POSIXct Objects

The "+" and "-" arithmetic operators are overloaded to allow addition and subtraction operations on POSIXct objects.

The "<" and ">" logical comparison operators are also overloaded to allow direct comparisons between POSIXct objects.

Operations on POSIXct objects are equivalent to the same operations on the internal integer representation of POSIXct (number of seconds since the *epoch*).

Subtracting POSIXct objects creates a time interval object of class difftime.

The method seq.POSIXt creates a vector of POSIXct *date-times*.

```
> # Same moment of time corresponds to different clock times
> timeny <- as.POSIXct("2014-07-14 13:30:10", tz="America/New_York")
> timeldn <- as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # Add five hours to POSIXct
> timeny + 5*60*60
> # Subtract POSIXct
> timeny - timeldn
> class(timeny - timeldn)
> # Compare POSIXct
> timeny > timeldn
> # Create vector of POSIXct times during trading hours
> timev <- seq(
+   from=as.POSIXct("2014-07-14 09:30:00", tz="America/New_York"),
+   to=as.POSIXct("2014-07-14 16:00:00", tz="America/New_York"),
+   by="10 min")
> head(timev, 3)
> tail(timev, 3)
```

Moment of Time and Clock Time

`as.POSIXct()` can also coerce `integer` objects into `POSIXct`, given an `origin` in time.

The same *moment of time* corresponds to different *clock times* in different *time zones*.

The same *clock times* in different *time zones* correspond to different *moments of time*.

```
> # POSIXct is stored as integer moment of time
> datetimen <- as.numeric(datetime)
> # Same moment of time corresponds to different clock times
> as.POSIXct(datetimen, origin="1970-01-01", tz="America/New_York")
> as.POSIXct(datetimen, origin="1970-01-01", tz="UTC")
> # Same clock time corresponds to different moments of time
> as.POSIXct("2014-07-14 13:30:10", tz="America/New_York") -
+   as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # Add 20 seconds to POSIXct
> datetime + 20
```

Methods for Manipulating POSIXct Objects

The generic function `format()` formats `R` objects for printing and display.

The method `format.POSIXct()` parses `POSIXct` objects into a `character` string representing the *clock time* in a given *time zone*.

The method `as.POSIXct.Date()` parses `Date` objects into `POSIXct`, and assigns to them the *moment of time* corresponding to midnight `UTC`.

`POSIX` is an acronym for "Portable Operating System Interface".

```
> datetime  # POSIXct date and time
> # Parse POSIXct to string representing the clock time
> format(datetime)
> class(format(datetime))  # Character string
> # Get clock times in different time zones
> format(datetime, tz="America/New_York")
> format(datetime, tz="UTC")
> # Format with custom format strings
> format(datetime, "%m/%Y")
> format(datetime, "%m-%d-%Y %H hours")
> # Trunc to hour
> format(datetime, "%m-%d-%Y %H:00:00")
> # Date converted to midnight UTC moment of time
> as.POSIXct(Sys.Date())
> as.POSIXct(as.numeric(as.POSIXct(Sys.Date())),
+      origin="1970-01-01",
+      tz="UTC")
```

POSIXlt Date-time Objects

The POSIXlt class in R represents *date-time* objects, that are stored internally as a list.

The function as.POSIXlt() can parse a character string (representing the *clock time*) and a *time zone* into a POSIXlt object.

The method format.POSIXlt() parses POSIXlt objects into a character string representing the *clock time* in a given *time zone*.

The function as.POSIXlt() can also parse a POSIXct object into a POSIXlt object, and as.POSIXct() can perform the reverse.

Adding a number to POSIXlt causes implicit coercion to POSIXct.

POSIXct and POSIXlt are two derived classes from the POSIXt class.

The methods round.POSIXt() and trunc.POSIXt() round and truncate POSIXt objects, and return POSIXlt objects.

```
> # Parse character string "%Y-%m-%d %H:%M:%S" to POSIXlt object
> datetime <- as.POSIXlt("2014-07-14 18:30:10")
> datetime
> class(datetime)  # POSIXlt object
> as.POSIXct(datetime)  # Coerce to POSIXct object
> # Extract internal list representation to vector
> unlist(datetime)
> datetime + 20  # Add 20 seconds
> class(datetime + 20)  # Implicit coercion to POSIXct
> trunc(datetime, units="hours")  # Truncate to closest hour
> trunc(datetime, units="days")  # Truncate to closest day
> methods(trunc)  # Trunc methods
> trunc.POSIXt
```

Time Zones and Date-time Conversion

date-time objects require a *time zone* to be uniquely specified.

UTC stands for "Universal Time Coordinated", and is synonymous with GMT, but doesn't change with Daylight Saving Time.

EST stands for "Eastern Standard Time", and is UTC - 5 hours.

EDT stands for "Eastern Daylight Time", and is UTC - 4 hours.

The function Sys.setenv() can be used to set the default *time zone*, but the environment variable "TZ" must be capitalized.

```
> Sys.timezone()  # Get time-zone
> Sys.setenv(TZ="UTC")  # Set time-zone to UTC
> Sys.timezone()  # Get time-zone
> # Standard Time in effect
> as.POSIXct("2013-03-09 11:00:00", tz="America/New_York")
> # Daylight Savings Time in effect
> as.POSIXct("2013-03-10 11:00:00", tz="America/New_York")
> datetime <- Sys.time()  # Today's date and time
> # Convert to character in different TZ
> format(datetime, tz="America/New_York")
> format(datetime, tz="UTC")
> # Parse back to POSIXct
> as.POSIXct(format(datetime, tz="America/New_York"))
> # Difference between New_York time and UTC
> as.POSIXct(format(Sys.time(), tz="UTC")) -
+   as.POSIXct(format(Sys.time(), tz="America/New_York"))
> # Set time-zone to New York
> Sys.setenv(TZ="America/New_York")
```

Manipulating Date-time Objects Using *lubridate*

The package *lubridate* contains functions for manipulating POSIXct date-time objects.

The ymd(), dmy(), etc. functions parse character and numeric *year-fraction* dates into POSIXct objects.

The mday(), month(), year(), etc. accessor functions extract date-time components.

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

The function date_decimal() converts numeric *year-fraction* dates into POSIXct objects.

```
> library(lubridate)  # Load lubridate
> # Parse strings into date-times
> as.POSIXct("07-14-2014", format="%m-%d-%Y", tz="America/New_York")
> datetime <- lubridate::mdy("07-14-2014", tz="America/New_York")
> datetime
> class(datetime)  # POSIXct object
> lubridate::dmy("14.07.2014", tz="America/New_York")
>
> # Parse numeric into date-times
> as.POSIXct(as.character(14072014), format="%d%m%Y",
+                  tz="America/New_York")
> lubridate::dmy(14072014, tz="America/New_York")
>
> # Parse decimal to date-times
> lubridate::decimal_date(datetime)
> lubridate::date_decimal(2014.25, tz="America/New_York")
> date_decimal(decimal_date(datetime), tz="America/New_York")
```

Time Zones Using *lubridate*

The package *lubridate* simplifies *time zone* calculations.

The package *lubridate* uses the *UTC time zone* as default.

The function `with_tz()` creates a date-time object with the same moment of time in a different *time zone*.

The function `force_tz()` creates a date-time object with the same clock time in a different *time zone*.

```
> datetime <- lubridate::ymd_hms(20140714142010,
+                tz="America/New_York")
> datetime
> # Get same moment of time in "UTC" time zone
> lubridate::with_tz(datetime, "UTC")
> as.POSIXct(format(datetime, tz="UTC"), tz="UTC")
> # Get same clock time in "UTC" time zone
> lubridate::force_tz(datetime, "UTC")
> as.POSIXct(format(datetime, tz="America/New_York"),
+      tz="UTC")
> # Same moment of time
> datetime - with_tz(datetime, "UTC")
> # Different moments of time
> datetime - force_tz(datetime, "UTC")
```

lubridate Time Span Objects

lubridate has two time span classes: `durations` and `periods`.

`durations` specify exact time spans, such as numbers of seconds, hours, days, etc.

The functions `ddays()`, `dyears()`, etc. return duration objects.

`periods` specify relative time spans that don't have a fixed length, such as months, years, etc.

`periods` account for variable days in the months, for Daylight Savings Time, and for leap years.

The functions `days()`, `months()`, `years()`, etc. return `period` objects.

```
> # Daylight Savings Time handling periods vs durations
> datetime <- as.POSIXct("2013-03-09 11:00:00", tz="America/New_York
> datetime
> datetime + lubridate::ddays(1)  # Add duration
> datetime + lubridate::days(1)  # Add period
>
> leap_year(2012)  # Leap year
> datetime <- lubridate::dmy(01012012, tz="America/New_York")
> datetime
> datetime + lubridate::dyears(1)  # Add duration
> datetime + lubridate::years(1)  # Add period
```

Adding Time Spans to Date-time Objects

periods allow calculating future dates with the same day of the month, or month of the year.

```
> datetime <- lubridate::ymd_hms(20140714142010, tz="America/New_Yor
> datetime
> # Add periods to a date-time
> c(datetime + lubridate::seconds(1), datetime + lubridate::minutes(
+   datetime + lubridate::days(1), datetime + lubridate::months(1))
>
> # Create vectors of dates
> datetime <- lubridate::ymd(20140714, tz="America/New_York")
> datetime + 0:2 * lubridate::months(1)  # Monthly dates
> datetime + lubridate::months(0:2)
> datetime + 0:2 * lubridate::months(2)  # bi-monthly dates
> datetime + seq(0, 5, by=2) * lubridate::months(1)
> seq(datetime, length=3, by="2 months")
```

End-of-month Dates

Adding monthly periods can create invalid dates.

The operators %m+% and %m-% add or subtract monthly periods to account for the varible number of days per month.

This allows creating vectors of end-of-month dates.

```
> # Adding monthly periods can create invalid dates
> datetime <- lubridate::ymd(20120131, tz="America/New_York")
> datetime + 0:2 * lubridate::months(1)
> datetime + lubridate::months(1)
> datetime + lubridate::months(2)
>
> # Create vector of end-of-month dates
> datetime %m-% lubridate::months(13:1)
```

Package *RQuantLib* Calendar Functions

The package *RQuantLib* is an interface to the *QuantLib* open source C/C++ library for quantitative finance, mostly designed for pricing fixed-income instruments and options.

The *QuantLib* library also contains calendar functions for determining holidays and business days in many different jurisdictions.

```
> library(RQuantLib)  # Load RQuantLib
>
> # Create daily date series of class "Date"
> dates <- Sys.Date() + -5:2
> dates
>
> # Create Boolean vector of business days
> # Use RQuantLib calendar
> is_busday <- RQuantLib::isBusinessDay(
+   calendar="UnitedStates/GovernmentBond", dates)
>
> # Create daily series of business days
> bus_index <- dates[is_busday]
> bus_index
```

Review of Date-time Classes in R

The Date class from the base package is suitable for *daily* time series.

The POSIXct class from the base package is suitable for *intra-day* time series.

The yearmon and yearqtr classes from the *zoo* package are suitable for *quarterly* and *monthly* time series.

```
> datetime <- Sys.Date()  # Create date series of class "Date"
> dates <- datetime + 0:365  # Daily series over one year
> head(dates, 4)  # Print first few dates
> format(head(dates, 4), "%m/%d/%Y")  # Print first few dates
> # Create daily date-time series of class "POSIXct"
> dates <- seq(Sys.time(), by="days", length.out=365)
> head(dates, 4)  # Print first few dates
> format(head(dates, 4), "%m/%d/%Y %H:%M:%S")  # Print first few dat
> # Create series of monthly dates of class "zoo"
> monthly_index <- yearmon(2010+0:36/12)
> head(monthly_index, 4)  # Print first few dates
> # Create series of quarterly dates of class "zoo"
> qrtly_index <- yearqtr(2010+0:16/4)
> head(qrtly_index, 4)  # Print first few dates
> # Parse quarterly "zoo" dates to POSIXct
> Sys.setenv(TZ="UTC")
> as.POSIXct(head(qrtly_index, 4))
```

Time Series Objects of Class *ts*

Time series are data objects that contain a *date-time* index and data associated with it.

The native time series class in R is *ts*.

ts time series are *regular*, i.e. they can only have an equally spaced *date-time* index.

ts time series have a `numeric` *date-time* index, usually encoded as a *year-fraction*, or some other unit, like number of months, etc.

For example the date `"2015-03-31"` can be encoded as a *year-fraction* equal to `2015.244`.

The *stats* base package contains functions for manipulating time series objects of class *ts*.

The function `ts()` creates a *ts* time series from a `numeric` vector or matrix, and from the associated *date-time* information (the number of data per time unit: year, month, etc.).

The `frequency` argument is the number of observations per unit of time.

For example, if the *date-time* index is encoded as a *year-fraction*, then `frequency=12` means 12 monthly data points per year.

```
> # Create daily time series ending today
> startd <- decimal_date(Sys.Date()-6)
> endd <- decimal_date(Sys.Date())
> # Create vector of geometric Brownian motion
> datav <- exp(cumsum(rnorm(6)/100))
> tstep <- NROW(datav)/(endd-startd)
> tseries <- ts(data=datav, start=startd, frequency=tstep)
> tseries  # Display time series
> # Display index dates
> as.Date(date_decimal(zoo::coredata(time(tseries))))
> # bi-monthly geometric Brownian motion starting mid-1990
> tseries <- ts(data=exp(cumsum(rnorm(96)/100)),
+        frequency=6, start=1990.5)
```

Manipulating *ts* Time Series

ts time series don't store their *date-time* indices, and instead store only a `"tsp"` attribute that specifies the index `start` and `end` dates and its `frequency`.

The *date-time* index is calculated as needed from the `"tsp"` attribute.

The function `time()` extracts the *date-time* index of a *ts* time series object.

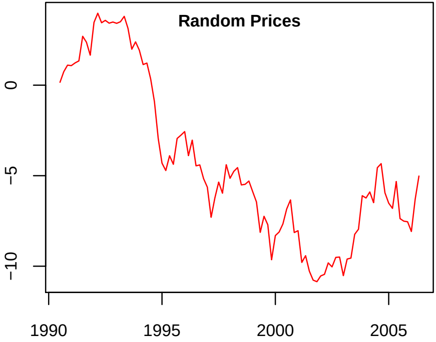
The function `window()` subsets the a *ts* time series object.

```
> # Show some methods for class "ts"
> matrix(methods(class="ts")[3:8], ncol=2)
> # "tsp" attribute specifies the date-time index
> attributes(tseries)
> # Extract the index
> tail(time(tseries), 11)
> # The index is equally spaced
> diff(tail(time(tseries), 11))
> # Subset the time series
> window(tseries, start=1992, end=1992.25)
```

Plotting *ts* Time Series Objects

The method `plot.ts()` plots *ts* time series objects.

```
> plot(tseries, type="l",  # Create plot
+      col="red", lty="solid", xlab="", ylab="")
> title(main="Random Prices", line=-1)  # Add title
```



EuStockMarkets Data

R includes a number of base packages that are already installed and loaded.

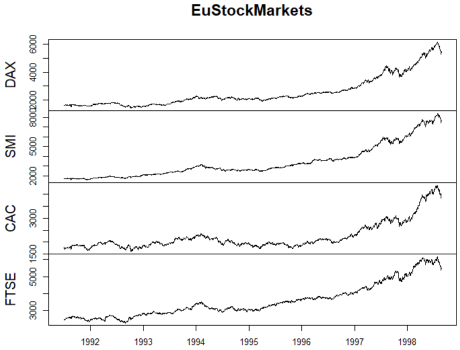
datasets is a base package containing various datasets, for example: EuStockMarkets.

The EuStockMarkets dataset contains daily closing prices of european stock indices.

EuStockMarkets is a mts() time series object.

The EuStockMarkets *date-time* index is equally spaced (*regular*), so the *year-fraction* dates don't correspond to actual trading days.

```
> class(EuStockMarkets)  # Multiple ts object
> dim(EuStockMarkets)
> head(EuStockMarkets, 3)  # Get first three rows
> # EuStockMarkets index is equally spaced
> diff(tail(time(EuStockMarkets), 11))
> # Plot all the columns in separate panels
> plot(EuStockMarkets, main="EuStockMarkets", xlab="")
```

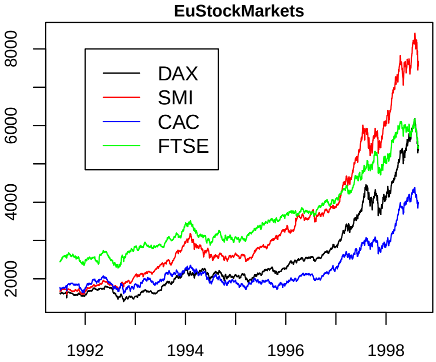


Plotting EuStockMarkets Data

The argument `plot.type="single"` for method `plot.zoo()` allows plotting multiple lines in a single panel (pane).

The four `EuStockMarkets` time series can be plotted in a single panel (pane).

```
> # Plot in single panel
> plot(EuStockMarkets, main="EuStockMarkets",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue", "green"))
> # Add legend
> legend(x=1992, y=8000,
+  legend=colnames(EuStockMarkets),
+  col=c("black", "red", "blue", "green"),
+  lwd=6, lty=1)
```



zoo Time Series Objects

The package *zoo* is designed for managing *irregular* time series and ordered objects of class *zoo*.

Irregular time series have *date-time* indeices that aren't equally spaced (because of weekends, overnight hours, etc.).

The function `zoo()` creates a *zoo* object from a `numeric` vector or matrix, and an associated *date-time* index.

The *zoo* index is a vector of *date-time* objects, and can be from any *date-time* class.

The *zoo* class can manage *irregular* time series whose *date-time* index isn't equally spaced.

```
> # Create zoo time series of random returns
> dates <- Sys.Date() + 0:3
> zoots <- zoo(rnorm(NROW(dates)), order.by=dates)
> zoots
> attributes(zoots)
> class(zoots)  # Class "zoo"
> tail(zoots, 3)  # Get last few elements
```

Operations on *zoo* Time Series

The function `zoo::coredata()` extracts the data contained in *zoo* object, and returns a vector or matrix.

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

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

The functions `start()` and `end()` return the time index values of the first and last elements of a *zoo* object.

The functions `cumsum()`, `cummax()`, and `cummin()` return cumulative sums, minima and maxima of a *zoo* object.

```
> zoo::coredata(zoots)  # Extract coredata
> zoo::index(zoots)  # Extract time index
> start(zoots)  # First date
> end(zoots)  # Last date
> zoots[start(zoots)]  # First element
> zoots[end(zoots)]  # Last element
> zoo::coredata(zoots) <- rep(1, 4)  # Replace coredata
> cumsum(zoots)  # Cumulative sum
> cummax(cumsum(zoots))
> cummin(cumsum(zoots))
```

Single Column *zoo* Time Series

Single column *zoo* time series usually don't have a dimension attribute (they have a NULL dimension), and they don't have a column name, unlike multi-column *zoo* time series.

Single column *zoo* time series without a dimension attribute should be avoided, since they can cause hard to detect bugs.

If a single column *zoo* time series is created from a single column matrices, then it have a dimension attribute, and can be assigned a column name.

```
> zoots <- zoo(matrix(cumsum(rnorm(100)), nc=1),
+   order.by=seq(from=as.Date("2013-06-15"), by="day", len=100))
> colnames(zoots) <- "zoots"
> tail(zoots)
> dim(zoots)
> attributes(zoots)
```

The lag() and diff() Functions

The method `lag.zoo()` returns a lagged version of a *zoo* time series, shifting the time index by "k" observations.

If "k" is positive, then `lag.zoo()` shifts values from the future to the present, and if "k" is negative then it shifts them from the past.

This is the opposite of what is usually considered as a positive *lag*.

A positive *lag* should replace the current value with values from the past (negative lags should replace with values from the future).

The method `diff.zoo()` returns the difference between a *zoo* time series and its proper lagged version from the past, given a positive *lag* value.

By default, the methods `lag.zoo()` and `diff.zoo()` omit any NA values they may have produced, and return shorter time series.

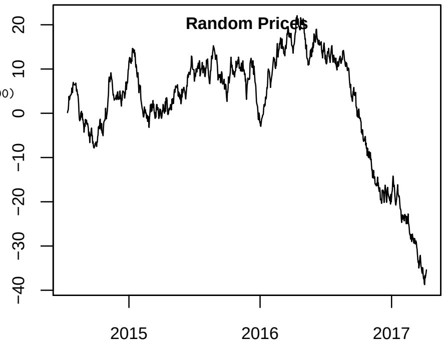
If the "na.pad" argument is set to TRUE, then they return time series of the same length, with NA values added where needed.

```
> zoo::coredata(zoots) <- (1:4)^2  # Replace coredata
> zoots
> lag(zoots)  # One day lag
> lag(zoots, 2)  # Two day lag
> lag(zoots, k=-1)  # Proper one day lag
> diff(zoots)  # Diff with one day lag
> # Proper lag and original length
> lag(zoots, -2, na.pad=TRUE)
```

Plotting *zoo* Time Series

zoo time series can be plotted using the generic function `plot()`, which dispatches the `plot.zoo()` method.

```
> library(zoo)  # Load package zoo
> # Create index of daily dates
> dates <- seq(from=as.Date("2014-07-14"), by="day", length.out=1000)
> # Create vector of geometric Brownian motion
> datav <- exp(cumsum(rnorm(NROW(dates))/100))
> # Create zoo series of geometric Brownian motion
> zoots <- zoo(x=datav, order.by=dates)
> # Plot using plot.zoo method
> plot(zoots, xlab="", ylab="")
> title(main="Random Prices", line=-1)  # Add title
```



Subsetting *zoo* Time Series

zoo time series can be subset in similar ways to `matrices` and *ts* time series.

The function `window()` can also subset *zoo* time series objects.

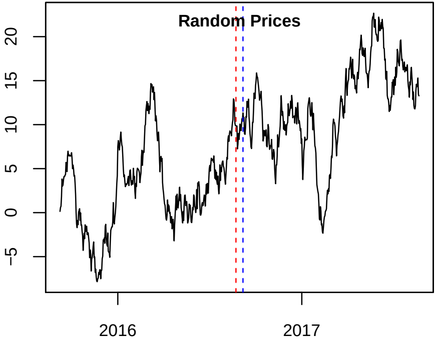
In addition, *zoo* time series can be subset using `Date` objects.

```
> # Subset zoo as matrix
> zoots[459:463, 1]
> # Subset zoo using window()
> window(zoots,
+  start=as.Date("2014-10-15"),
+  end=as.Date("2014-10-19"))
> # Subset zoo using Date object
> zoots[as.Date("2014-10-15")]
```

Sequential Joining *zoo* Time Series

zoo time series can be joined sequentially using function `rbind()`.

```
> library(zoo)  # Load package zoo
> # Create daily date series of class "Date"
> index1 <- seq(Sys.Date(), by="days", length.out=365)
> # Create zoo time series of random returns
> zoots1 <- zoo(rnorm(NROW(index1)), order.by=index1)
> # Create another zoo time series of random returns
> index2 <- seq(Sys.Date()+350, by="days", length.out=365)
> zoots2 <- zoo(rnorm(NROW(index2)), order.by=index2)
> # rbind the two time series - ts1 supersedes ts2
> zoots3 <- rbind(zoots1,
+   zoots2[zoo::index(zoots2) > end(zoots1)])
> # Plot zoo time series of geometric Brownian motion
> plot(exp(cumsum(zoots3)/100), xlab="", ylab="")
> # Add vertical lines at stitch point
> abline(v=end(zoots1), col="blue", lty="dashed")
> abline(v=start(zoots2), col="red", lty="dashed")
> title(main="Random Prices", line=-1)  # Add title
```



Merging *zoo* Time Series

zoo time series can be combined concurrently by joining their columns using function `merge()`.

Function `merge()` is similar to function `cbind()`.

If the `all=TRUE` option is set, then `merge()` returns the union of their dates, otherwise it returns their intersection.

The `merge()` operation can produce `NA` values.

```
> # Create daily date series of class "Date"
> index1 <- Sys.Date() + -3:1
> # Create zoo time series of random returns
> zoots1 <- zoo(rnorm(NROW(index1)), order.by=index1)
> # Create another zoo time series of random returns
> index2 <- Sys.Date() + -1:3
> zoots2 <- zoo(rnorm(NROW(index2)), order.by=index2)
> merge(zoots1, zoots2)  # union of dates
> # Intersection of dates
> merge(zoots1, zoots2, all=FALSE)
```

Managing NA Values

Binding two time series that don't share the same time index produces NA values.

There are two dedicated functions for managing NA values in time series:

- `stats::na.omit()` removes whole rows of data containing NA values.
- `zoo::na.locf()` replaces NA values with the most recent non-NA values prior to it (*locf* stands for *last observation carry forward*).

Copying the last non-NA values forward causes less data loss than removing whole rows of data.

`na.locf()` with argument `fromLast=TRUE` operates in reverse order, starting from the end.

But copying values forward requires initializing the first row of data, to guarantee that initial NA values are also over-written.

The initial NA *prices* can be initialized to the first non-NA price in the future, which can be done by calling `zoo::na.locf()` with the argument `fromLast=TRUE`.

But the initial NA values in *returns* data should be initialized to *zero*, without carrying data backward from the future, to avoid data *snooping*.

```
> # Create matrix containing NA values
> matrixv <- sample(18)
> matrixv[sample(NROW(matrixv), 4)] <- NA
> matrixv <- matrix(matrixv, nc=3)
> # Replace NA values with most recent non-NA values
> zoo::na.locf(matrixv)
> rutils::na_locf(matrixv)
> # Get time series of prices
> pricev <- mget(c("VTI", "VXX"), envir=rutils::etfenv)
> pricev <- lapply(pricev, quantmod::Cl)
> pricev <- rutils::do_call(cbind, pricev)
> sum(is.na(pricev))
> # Carry forward and backward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> pricev <- zoo::na.locf(pricev, na.rm=FALSE, fromLast=TRUE)
> sum(is.na(pricev))
> # Remove whole rows containing NA returns
> retp <- rutils::etfenv$returns
> sum(is.na(retp))
> retp <- na.omit(retp)
> # Or carry forward non-NA returns (preferred)
> retp <- rutils::etfenv$returns
> retp[1, is.na(retp[1, ])] <- 0
> retp <- zoo::na.locf(retp, na.rm=FALSE)
> sum(is.na(retp))
```

Managing NA Values in "xts" Time Series

The function `na.locf.xts()` from package *xts* is faster than `zoo::na.locf()`, but it only operates on time series of class "xts".

```
> # Replace NAs in xts time series
> pricev <- rutils::etfenv$pricev[, 1]
> head(pricev)
> sum(is.na(pricev))
> library(quantmod)
> pricezoo <- zoo::na.locf(pricev, na.rm=FALSE, fromLast=TRUE)
> pricexts <- xts:::na.locf.xts(pricev, fromLast=TRUE)
> all.equal(pricezoo, pricexts, check.attributes=FALSE)
> library(microbenchmark)
> summary(microbenchmark(
+   zoo=zoo::na.locf(pricev, fromLast=TRUE),
+   xts=xts:::na.locf.xts(pricev, fromLast=TRUE),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing Time Series Objects Into *zoo*

The generic function as.zoo() coerces objects into *zoo* time series.

The function as.zoo() creates a *zoo* object with a numeric *date-time* index, with *date-time* encoded as a *year-fraction*.

The *year-fraction* can be *approximately* converted to a Date object by first calculating the number of days since the *epoch* (1970), and then coercing the numeric days using as.Date().

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

The function date_decimal() provides a more accurate way of converting a *year-fraction* index to POSIXct.

```
> class(EuStockMarkets)  # Multiple ts object
> # Coerce mts object into zoo
> zoots <- as.zoo(EuStockMarkets)
> class(zoo::index(zoots))  # Index is numeric
> head(zoots, 3)
> # Approximately convert index into class "Date"
> zoo::index(zoots) <-
+   as.Date(365*(zoo::index(zoots)-1970))
> head(zoots, 3)
> # Convert index into class "POSIXct"
> zoots <- as.zoo(EuStockMarkets)
> zoo::index(zoots) <- date_decimal(zoo::index(zoots))
> head(zoots, 3)
```

Coercing *zoo* Time Series Into Class *ts*

The generic function as.ts() from package *stats* coerces time series objects (including *zoo*) into *ts* time series.

The function as.ts() creates a *ts* object with a frequency=1, implying a *"day"* time unit, instead of a *"year"* time unit suitable for *year-fraction* dates.

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

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

```
> # Create index of daily dates
> dates <- seq(from=as.Date("2014-07-14"), by="day", length.out=100(
> # Create vector of geometric Brownian motion
> datav <- exp(cumsum(rnorm(NROW(dates))/100))
> # Create zoo time series of geometric Brownian motion
> zoots <- zoo(x=datav, order.by=dates)
> head(zoots, 3)  # zoo object
> # as.ts() creates ts object with frequency=1
> tseries <- as.ts(zoots)
> tsp(tseries)  # Frequency=1
> # Get start and end dates of zoots
> startd <- decimal_date(start(zoots))
> endd <- decimal_date(end(zoots))
> # Calculate frequency of zoots
> tstep <- NROW(zoots)/(endd-startd)
> datav <- zoo::coredata(zoots)  # Extract data from zoots
> # Create ts object using ts()
> tseries <- ts(data=datav, start=startd, frequency=tstep)
> # Display start of time series
> window(tseries, start=start(tseries),
+  end=start(tseries)+4/365)
> head(time(tseries))  # Display index dates
> head(as.Date(date_decimal(zoo::coredata(time(tseries)))))
```

Coercing Irregular Time Series Into Class *ts*

Irregular time series cannot be properly coerced into *ts* time series without modifying their index.

The function as.ts() creates NA values when it coerces irregular time series into a *ts* time series.

```
> # Create weekday Boolean vector
> weekdayv <- weekdays(zoo::index(zoots))
> is_weekday <- !((weekdayv == "Saturday") |
+   (weekdayv == "Sunday"))
> # Remove weekends from zoo time series
> zoots <- zoots[is_weekday, ]
> head(zoots, 7)  # zoo object
> # as.ts() creates NA values
> tseries <- as.ts(zoots)
> head(tseries, 7)
> # Create vector of regular dates, including weekends
> dates <- seq(from=start(zoots),
+             by="day",
+             length.out=NROW(zoots))
> zoo::index(zoots) <- dates
> tseries <- as.ts(zoots)
> head(tseries, 7)
```

xts Time Series Objects

The package *xts* defines time series objects of class *xts*,

- Class *xts* is an extension of the *zoo* class (derived from *zoo*),
- Class *xts* is the most widely accepted time series class,
- Class *xts* is designed for high-frequency and *OHLC* data,
- Class *xts* contains many convenient functions for plotting, calculating rolling max, min, etc.

The function `xts()` creates a *xts* object from a `numeric` vector or matrix, and an associated *date-time* index.

The *xts* index is a vector of *date-time* objects, and can be from any *date-time* class.

The *xts* class can manage *irregular* time series whose *date-time* index isn't equally spaced.

```
> library(xts)  # Load package xts
> # Create xts time series of random returns
> dates <- Sys.Date() + 0:3
> xtsv <- xts(rnorm(NROW(dates)), order.by=dates)
> names(xtsv) <- "random"
> xtsv
> tail(xtsv, 3)  # Get last few elements
> first(xtsv)  # Get first element
> last(xtsv)  # Get last element
> class(xtsv)  # Class "xts"
> attributes(xtsv)
> # Get the time zone of an xts object
> indexTZ(xtsv)
```

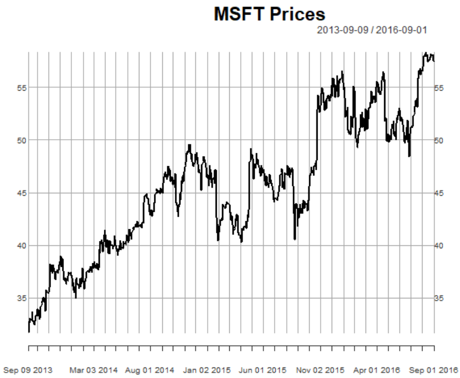
Coercing *zoo* Time Series Into Class *xts*

The function as.xts() coerces *zoo* time series into *xts* series.

as.xts() preserves the *index* attributes of the original time series.

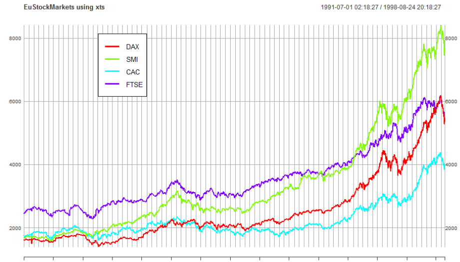
xts can be plotted using the generic function plot(), which dispatches the plot.xts() method.

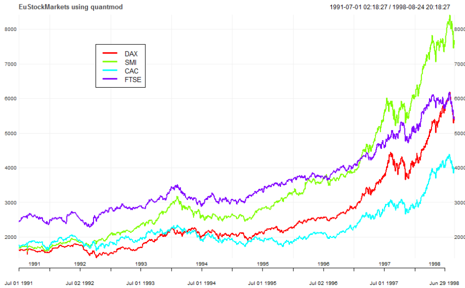
```
> library(xts)  # Load package xts
> # as.xts() coerces zoo series into xts series
> pricexts <- as.xts(pricezoo)
> dim(pricexts)
> head(pricexts[, 1:4], 4)
> # Plot using plot.xts method
> xts::plot.xts(pricexts[, "Close"], xlab="", ylab="", main="")
> title(main="MSFT Prices")  # Add title
```



Plotting Multiple *xts* Using Packages *xts* and *quantmod*

```
> library(lubridate)  # Load lubridate
> # Coerce EuStockMarkets into class xts
> xtsv <- xts(zoo::coredata(EuStockMarkets),
+       order.by=date_decimal(zoo::index(EuStockMarkets)))
> # Plot all columns in single panel: xts v.0.9-8
> colorv <- rainbow(NCOL(xtsv))
> plot(xtsv, main="EuStockMarkets using xts",
+      col=colorv, major.ticks="years",
+      minor.ticks=FALSE)
> legend("topleft", legend=colnames(EuStockMarkets),
+  inset=0.2, cex=0.7, , lty=rep(1, NCOL(xtsv)),
+  lwd=3, col=colorv, bg="white")
> # Plot only first column: xts v.0.9-7
> plot(xtsv[, 1], main="EuStockMarkets using xts",
+      col=colorv[1], major.ticks="years",
+      minor.ticks=FALSE)
> # Plot remaining columns
> for (colnum in 2:NCOL(xtsv))
+   lines(xtsv[, colnum], col=colorv[colnum])
> # Plot using quantmod
> library(quantmod)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colors
> chart_Series(x=xtsv, theme=plot_theme,
+        name="EuStockMarkets using quantmod")
> legend("topleft", legend=colnames(EuStockMarkets),
+  inset=0.2, cex=0.7, , lty=rep(1, NCOL(xtsv)),
+  lwd=3, col=colorv, bg="white")
```





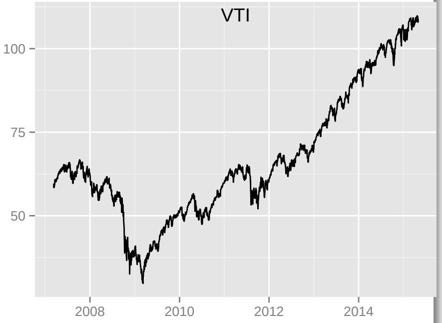
Plotting *xts* Using Package *ggplot2*

xts time series can be plotted using the package *ggplot2*.

The function qplot() is the simplest function in the *ggplot2* package, and allows creating line and bar plots.

The function theme() customizes plot objects.

```
> library(ggplot2)
> pricev <- rutils::etfenv$pricev[, 1]
> pricev <- na.omit(pricev)
> # Create ggplot object
> plotobj <- qplot(x=zoo::index(pricev),
+           y=as.numeric(pricev),
+           geom="line",
+           main=names(pricev)) +
+   xlab("") + ylab("") +
+   theme(  # Add legend and title
+     legend.position=c(0.1, 0.5),
+     plot.title=element_text(vjust=-2.0),
+     plot.background=element_blank()
+   )  # end theme
> # Render ggplot object
> plotobj
```

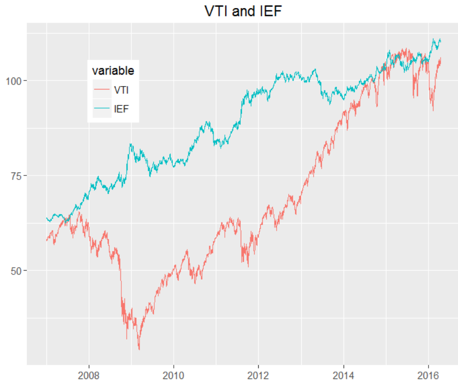


Plotting Multiple *xts* Using Package *ggplot2*

Multiple *xts* time series can be plotted using the function `ggplot()` from package *ggplot2*.

But *ggplot2* functions don't accept time series objects, so time series must be first coerced into `data frames`.

```
> library(rutils)  # Load xts time series data
> library(reshape2)
> library(ggplot2)
> pricev <- rutils::etfenv$pricev[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> # Create data frame of time series
> dframe <- data.frame(dates=zoo::index(pricev),
+     zoo::coredata(pricev))
> # reshape data into a single column
> dframe <-
+   reshape2::melt(dframe, id="dates")
> x11(width=6, height=5)  # Open plot window
> # ggplot the melted dframe
> ggplot(data=dframe,
+  mapping=aes(x=dates, y=value, colour=variable)) +
+  geom_line() +
+   xlab("") + ylab("") +
+   ggtitle("VTI and IEF") +
+   theme(  # Add legend and title
+     legend.position=c(0.2, 0.8),
+     plot.title=element_text(vjust=-2.0)
+   )  # end theme
```



Time series with multiple columns must be reshaped into a single column, which can be performed using the function `melt()` from package *reshape2*,

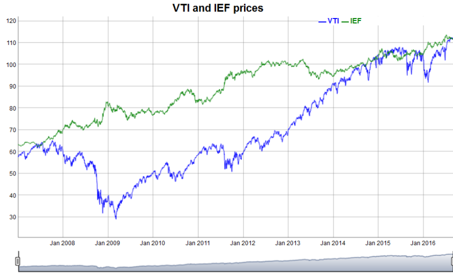
Interactive Time Series Plots Using Package *dygraphs*

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

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

The function `dyRangeSelector()` adds a date range selector to the bottom of a *dygraphs* plot.

```
> # Load rutils which contains etfenv dataset
> library(rutils)
> library(dygraphs)
> pricev <- rutils::etfenv$pricev[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> # Plot dygraph with date range selector
> dygraph(pricev, main="VTI and IEF prices") %>%
+   dyOptions(colors=c("blue","green")) %>%
+   dyRangeSelector()
```



The *dygraphs* package in R is an interface to the *dygraphs JavaScript* charting library.

Interactive *dygraphs* plots require running *JavaScript* code, which can be embedded in *html* documents, and displayed by web browsers.

But *pdf* documents can't run *JavaScript* code, so they can't display interactive *dygraphs* plots,

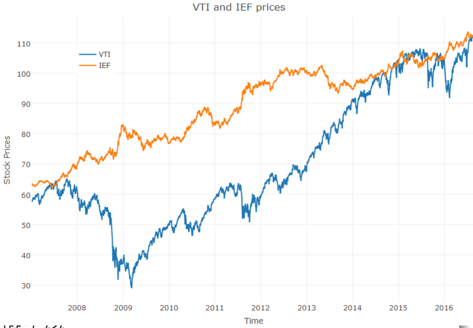
Interactive Time Series Plots Using Package *plotly*

The function `plot_ly()` from package *plotly* creates interactive plots from data residing in `data frames`.

The function `add_trace()` adds elements to a *plotly* plot.

The function `layout()` modifies the layout of a *plotly* plot.

```
> # Load rutils which contains etfenv dataset
> library(rutils)
> library(plotly)
> pricev <- rutils::etfenv$pricev[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> # Create data frame of time series
> dframe <- data.frame(dates=zoo::index(pricev),
+     zoo::coredata(pricev))
> # Plotly syntax using pipes
> dframe %>%
+   plot_ly(x=~dates, y=~VTI, type="scatter", mode="lines", name="VT
+   add_trace(x=~dates, y=~IEF, type="scatter", mode="lines", name='IEF') %>%
+   layout(title="VTI and IEF prices",
+    xaxis=list(title="Time"),
+    yaxis=list(title="Stock Prices"),
+    legend=list(x=0.1, y=0.9))
> # Or use standard plotly syntax
> plotobj <- plot_ly(data=dframe, x=~dates, y=~VTI, type="scatter", mode="lines", name="VTI")
> plotobj <- add_trace(p=plotobj, x=~dates, y=~IEF, type="scatter", mode="lines", name="IEF")
> plotobj <- layout(p=plotobj, title="VTI and IEF prices", xaxis=list(title="Time"), yaxis=list(title="Stock Prices"), legend=list(x=0.
> plotobj
```



Subsetting *xts* Time Series

xts time series can be subset in similar ways as *zoo* time series.

In addition, *xts* time series can be subset using date strings, or date range strings, for example: `["2014-10-15/2015-01-10"]`.

xts time series can be subset by year, week, days, or even seconds.

If only the date is subset, then a comma "," after the date range isn't necessary.

The function `.subset_xts()` allows fast subsetting of *xts* time series, which for large datasets can be faster than the bracket "[]" notation.

```
> # Subset xts using a date range string
> pricev <- rutils::etfenv$prices
> pricesub <- pricev["2014-10-15/2015-01-10", 1:4]
> first(pricevub)
> last(pricevub)
> # Subset Nov 2014 using a date string
> pricesub <- pricev["2014-11", 1:4]
> first(pricevub)
> last(pricevub)
> # Subset all data after Nov 2014
> pricesub <- pricev["2014-11/", 1:4]
> first(pricevub)
> last(pricevub)
> # Comma after date range not necessary
> all.equal(pricev["2014-11", ], pricev["2014-11"])
> # .subset_xts() is faster than the bracket []
> library(microbenchmark)
> summary(microbenchmark(
+   bracket=pricev[10:20, ],
+   subset=xts::.subset_xts(pricev, 10:20),
+   times=10))[, c(1, 4, 5)]
```

Fast Subsetting of *xts* Time Series

Subsetting of *xts* time series can be made much faster if the right operations are used.

Subsetting *xts* time series using `Boolean` vectors is usually faster than using date strings.

But the speed of subsetting can be reduced by additional operations, like coercing strings into dates.

```
> # Specify string representing a date
> datev <- "2014-10-15"
> # Subset prices in two different ways
> pricev <- rutils::etfenv$prices
> all.equal(pricev[zoo::index(pricev) >= datev],
+     pricev[paste0(datev, "/")])
> # Boolean subsetting is slower because coercing string into date
> library(microbenchmark)
> summary(microbenchmark(
+   boolean=(pricev[zoo::index(pricev) >= datev]),
+   date=(pricev[paste0(datev, "/")]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Coerce string into a date
> datev <- as.Date("2014-10-15")
> # Boolean subsetting is faster than using date string
> summary(microbenchmark(
+   boolean=(pricev[zoo::index(pricev) >= datev]),
+   date=(pricev[paste0(datev, "/")]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Subsetting Recurring *xts* Time Intervals

A *recurring time interval* is the same time interval every day, for example the time interval from 9:30AM to 4:00PM every day.

xts series can be subset on recurring time intervals using the `"T"` notation.

For example, to subset the time interval from 9:30AM to 4:00PM every day: `["T09:30:00/T16:00:00"]`

Warning messages that "timezone of object is different than current timezone" can be suppressed by calling the function options() with argument `"xts_check_tz=FALSE"`

```
> pricev <- HighFreq::SPY["2012-04"]
> # Subset recurring time interval using "T notation",
> pricev <- pricev["T10:30:00/T15:00:00"]
> first(pricev["2012-04-16"])  # First element of day
> last(pricev["2012-04-16"])  # Last element of day
> # Suppress timezone warning messages
> options(xts_check_tz=FALSE)
```

Binding *xts* Time Series by Rows

The function `rbind()` joins the rows of *xts* time series.

If the time series have overlapping time indices then the join produces duplicate rows with the same dates.

The duplicate rows can be removed using the function `duplicated()`.

The function `duplicated()` returns a Boolean vector indicating the duplicate elements of a vector.

The function `duplicated()` with argument `"fromLast=TRUE"` identifies duplicate elements starting from the end.

```
> # Create time series with overlapping time indices
> vti1 <- rutils::etfenv$VTI["/2015"]
> vti2 <- rutils::etfenv$VTI["2014/"]
> dates1 <- zoo::index(vti1)
> dates2 <- zoo::index(vti2)
> # Join by rows
> vti <- rbind(vti1, vti2)
> dates <- zoo::index(vti)
> sum(duplicated(dates))
> vti <- vti[!duplicated(dates), ]
> all.equal(vti, rutils::etfenv$VTI)
> # Alternative method - slightly slower
> vti <- rbind(vti1, vti2[!(zoo::index(vti2) %in% zoo::index(vti1))]
> all.equal(vti, rutils::etfenv$VTI)
> # Remove duplicates starting from the end
> vti <- rbind(vti1, vti2)
> vti <- vti[!duplicated(dates), ]
> vtifl <- vti[!duplicated(dates, fromLast=TRUE), ]
> all.equal(vti, vtifl)
```

Properties of *xts* Time Series

xts series always have a `dim` attribute, unlike *zoo*, which have no `dim` attribute when they only have one column of data.

zoo series with multiple columns have a `dim` attribute, and are therefore matrices.

But *zoo* with a single column don't, and are therefore vectors not matrices.

When a *zoo* is subset to a single column, the `dim` attribute is dropped, which can create errors.

```
> pricev <- rutils::etfenv$pricev[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> str(pricev)  # Display structure of xts
> # Subsetting zoo to single column drops dim attribute
> pricezoo <- as.zoo(pricev)
> dim(pricezoo)
> dim(pricezoo[, 1])
> # zoo with single column are vectors not matrices
> c(is.matrix(pricezoo), is.matrix(pricezoo[, 1]))
> # xts always have a dim attribute
> rbind(base=dim(pricev), subs=dim(pricev[, 1]))
> c(is.matrix(pricev), is.matrix(pricev[, 1]))
```

lag() and diff() Operations on *xts* Time Series

The methods `xts::lag()` and `xts::diff()` for *xts* series differ from those of package *zoo*.

By default, the method `xts::lag()` replaces the current value with values from the past (negative lags replace with values from the future).

The methods `zoo::lag()` and `zoo::diff()` shorten the series by the number of lag periods.

By default, the methods `xts::lag()` and `xts::diff()` retain the same number of elements, by padding with leading or trailing `NA` values.

In order to avoid padding with `NA` values, asset returns can be padded with zeros, and prices can be padded with the first or last elements of the input vector.

```
> # Lag of zoo shortens it by one row
> rbind(base=dim(pricezoo), lag=dim(lag(pricezoo)))
> # Lag of xts doesn't shorten it
> rbind(base=dim(pricev), lag=dim(lag(pricev)))
> # Lag of zoo is in opposite direction from xts
> head(lag(pricezoo, -1), 4)
> head(lag(pricev), 4)
```

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

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

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

Converting *xts* Time Series to Lower Periodicity

The function `to.period()` converts a time series to a lower periodicity (for example from hourly to daily periodicity).

`to.period()` returns a time series of open, high, low, and close values (*OHLC*) for the lower period.

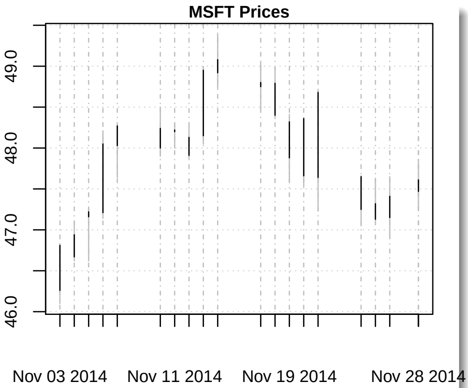
`to.period()` converts both univariate and *OHLC* time series to a lower periodicity.

```
> # Lower the periodicity to months
> pricem <- to.period(x=pricev, period="weeks", name="MSFT")
> # Convert colnames to standard OHLC format
> colnames(pricem)
> colnames(pricem) <- sapply(
+   strsplit(colnames(pricem), split=".", fixed=TRUE),
+   function(na_me) na_me[-1]
+   ) # end sapply
> head(pricem, 3)
> # Lower the periodicity to years
> pricesy <- to.period(x=pricem, period="years", name="MSFT")
> colnames(pricevy) <- sapply(
+   strsplit(colnames(pricevy), split=".", fixed=TRUE),
+   function(na_me) na_me[-1]
+   ) # end sapply
> head(pricevy)
```

Plotting *OHLC* Time Series Using plot.xts()

The method (function) plot.xts() can plot *OHLC* time series of class *xts*.

```
> library(xts)  # Load package xts
> # as.xts() coerces zoo series into xts series
> pricexts <- as.xts(pricezoo)
> # Subset xts using a date
> pricexts <- pricexts["2014-11", 1:4]
>
> # Plot OHLC using plot.xts method
> xts::plot.xts(pricexts, type="candles", main="")
> title(main="MSFT Prices")  # Add title
```



Time Series Classes in R

R and other packages contain a number of different time series classes:

- Class *ts* from base package *stats*:
  native time series class in R, but allows only *regular* (equally spaced) date-time index,
  not suitable for sophisticated financial applications,
- Class *zoo*: allows *irregular* date-time index,
  the *zoo* index can be from any *date-time* class,
- Class *xts* extension of *zoo* class: most widely accepted time series class,
  designed for high-frequency and *OHLC* data,
  contains convenient functions for plotting, calculating rolling max, min, etc.
- Class *timeSeries* from the *Rmetrics* suite,

```
> pricev <- as.ts(pricezoo)
> class(pricev)
> tail(pricev[, 1:4])
> library(xts)
> pricexts <- as.xts(pricezoo)
> class(pricexts)
> tail(pricexts[, 1:4])
```

Writing Text Strings

The function cat() concatenates strings and writes them to standard output or to files.

cat() interretsp its argument character string and its escape sequences ("\"), but doesn't return a value.

The function print() doesn't interpret its argument, and simply prints it to standard output and invisibly returns it.

Typing the name of an object in R implicitly calls print() on that object.

The function save() writes objects to compressed binary .RData files.

```
> cat("Enter\ttab")  # Cat() interretsp backslash escape sequences
> print("Enter\ttab")
>
> my_text <- print("hello")
> my_text  # Print() returns its argument
>
> # Create string
> my_text <- "Title: My Text\nSome numbers: 1,2,3,...\nRprofile file
>
> cat(my_text, file="mytext.txt")  # Write to text file
>
> cat("Title: My Text",  # Write several lines to text file
+     "Some numbers: 1,2,3,...",
+     "Rprofile files contain code executed at R startup,",
+     file="mytext.txt", sep="\n")
>
> save(my_text, file="mytext.RData")  # Write to binary file
```

Displaying Numeric Data

The function `print()` displays numeric data objects, with the number of digits given by the global option `"digits"`.

The function `sprintf()` returns strings formatted from text strings and numeric data.

```
> print(pi)
[1] 3.14
> print(pi, digits=10)
[1] 3.141592654
> getOption("digits")
[1] 3
> foo <- 12
> bar <- "weeks"
> sprintf("There are %i %s in the year", foo, bar)
[1] "There are 12 weeks in the year"
```

Reading Text from Files

The function `scan()` reads text or data from a file and returns it as a vector or a list.

The function `readLines()` reads lines of text from a connection (file or console), and returns them as a vector of `character` strings.

The function `readline()` reads a single line from the console, and returns it as a `character` string.

The function `file.show()` reads text or data from a file and displays in editor.

```
> # Read text from file
> scan(file="mytext.txt", what=character(), sep="\n")
>
> # Read lines from file
> readLines(con="mytext.txt")
>
> # Read text from console
> input <- readline("Enter a number: ")
> class(input)
> # Coerce to numeric
> input <- as.numeric(input)
>
> # Read text from file and display in editor:
> # file.show("mytext.txt")
> # file.show("mytext.txt", pager="")
```

Writing and Reading *Data Frames* from *Text* Files

The functions `write.table()` and `read.table()` write and read *data frames* from text files.

`write.table()` coerces objects to *data frames* before it writes them.

`read.table()` returns a *data frame*, without coercing non-numeric values to `factors` (so no need for the option `stringsAsFactors=FALSE`).

`write.table()` and `read.table()` can be used to write and read matrices from text files, but they have to be coerced back to matrices.

`write.table()` and `read.table()` are inefficient for very large data sets.

```
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> dframe <- data.frame(type=c("rose", "daisy", "tulip"),
+   color=c("red", "white", "yellow"),
+   price=c(1.5, 0.5, 1.0),
+   row.names=c("flower1", "flower2", "flower3"))  # end data.frame
> matrixv <- matrix(sample(1:12), ncol=3,
+   dimnames=list(NULL, c("col1", "col2", "col3")))
> rownames(matrixv) <- paste("row", 1:NROW(matrixv), sep="")
> # Write data frame to text file, and then read it back
> write.table(dframe, file="florist.txt")
> readf <- read.table(file="florist.txt")
> readf  # A data frame
>
> # Write matrix to text file, and then read it back
> write.table(matrixv, file="matrix.txt")
> readmat <- read.table(file="matrix.txt")
> readmat  # write.table() coerced matrix to data frame
> class(readmat)
> # Coerce from data frame back to matrix
> readmat <- as.matrix(readmat)
> class(readmat)
```

Copying *Data Frames* Between the *clipboard* and R

Data frames stored in the *clipboard* can be copied into R using the function `read.table()`.

Data frames in R can be copied into the *clipboard* using the function `write.table()`.

This allows convenient copying of *data frames* between R and `Excel`.

Data frames can also be manipulated directly in the R spreadsheet-style data editor.

Copying and pasting between the *clipboard* and R works well on Windows, but not on MacOS. There are some workarounds for MacOS:
Copy_paste_between_R_and_clipboard

```
> # Create a data frame
> dframe <- data.frame(small=c(3, 5), medium=c(9, 11), large=c(15, 1
>
> # Launch spreadsheet-style data editor
> dframe <- edit(dframe)
>
> # Copy the data frame to clipboard
> write.table(x=dframe, file="clipboard", sep="\t")
>
> # Wrapper function for copying data frame from R into clipboard
> # by default, data is tab delimited, with a header
> write_clip <- function(data, row.names=FALSE, col.names=TRUE, ...)
+   write.table(x=data, file="clipboard", sep="\t",
+       row.names=row.names, col.names=col.names, ...)
+ }  # end write_clip
>
> write_clip(data=dframe)
>
> # Wrapper function for copying data frame from clipboard into R
> # by default, data is tab delimited, with a header
> read_clip <- function(file="clipboard", sep="\t", header=TRUE, ..
+   read.table(file=file, sep=sep, header=header, ...)
+ }  # end read_clip
>
> dframe <- read.table("clipboard", header=TRUE)
> dframe <- read_clip()
```

Writing and Reading *Data Frames* From .csv Files

The easiest way to share data between R and Excel is through .csv files.

The functions write.csv() and read.csv() write and read *data frames* from .csv format files.

The functions write.csv() and read.csv() write and read *data frames* from .csv format files.

These functions are *wrappers* for write.table() and read.table().

read.csv() doesn't coerce non-numeric values to factors, so no need for the option stringsAsFactors=FALSE.

read.csv() reads row names as an extra column, unless the row.names=1 argument is used.

The argument "row.names" accepts either the number or the name of the column containing the row names.

The *.csv() functions are very inefficient for large data sets.

```
> # Write data frame to CSV file, and then read it back
> write.csv(dframe, file="florist.csv")
> readf <- read.csv(file="florist.csv")
> readf  # the row names are read in as extra column
> # Restore row names
> rownames(readf) <- readf[, 1]
> readf <- readf[, -1]  # Remove extra column
> readf
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> readf
```

Writing and Reading *Data Frames* From .csv Files (cont.)

The functions `write.csv()` and `read.csv()` can write and read *data frames* from .csv format files *without using row names*.

Row names can be omitted from the output file by calling `write.csv()` with the argument `row.names=FALSE`.

```
> # Write data frame to CSV file, without row names
> write.csv(dframe, row.names=FALSE, file="florist.csv")
> readf <- read.csv(file="florist.csv")
> readf  # A data frame without row names
```

Reading Data From Very Large .csv Files

Data from very large .csv files can be read in small chunks instead of all at once.

The function file() opens a connection to a file or an internet website URL.

The function read.csv() with the argument "nrows" reads only the specified number of rows from a connection and returns a *data frame*. The connection pointer is reset to the next row.

The function read.csv() with the argument "nrows" allows reading data sequentially from very large files that wouldn't fit into memory.

```
> # Open a read connection to a file
> con_read = file("/Users/jerzy/Develop/lecture_slides/data/etf_pric
> # Read the first 10 rows
> data10 <- read.csv(con_read, nrows=10)
> # Read another 10 rows
> data20 <- read.csv(con_read, nrows=10, header=FALSE)
> colnames(data20) <- colnames(data10)
> # Close the connection to the file
> close(con_read)
> # Open a read connection to a file
> con_read = file("/Users/jerzy/Develop/lecture_slides/data/etf_pric
> # Read the first 1000 rows
> data10 <- read.csv(con_read, nrows=1e3)
> colnamev <- colnames(data10)
> # Write to a file
> countv <- 1
> write.csv(data10, paste0("/Users/jerzy/Develop/data/temp/etf_price
> # Read remaining rows in a loop 10 rows at a time
> # Can produce error without getting to end of file
> while (isOpen(con_read)) {
+   datav <- read.csv(con_read, nrows=1e3)
+   colnames(datav) <- colnamev
+   write.csv(datav, paste0("/Users/jerzy/Develop/data/temp/etf_pric
+   countv <- countv + 1
+ }  # end while
```

Writing and Reading Matrices From .csv Files

The functions `write.csv()` and `read.csv()` can write and read matrices from `.csv` format files.

If row names can be omitted in the output file, then `write.csv()` can be called with argument `row.names=FALSE`.

If the input file doesn't contain row names, then `read.csv()` can be called without the `"row.names"` argument.

```
> # Write matrix to csv file, and then read it back
> write.csv(matrixv, file="matrix.csv")
> readmat <- read.csv(file="matrix.csv", row.names=1)
> readmat  # Read.csv() reads matrix as data frame
> class(readmat)
> readmat <- as.matrix(readmat)  # Coerce to matrix
> identical(matrixv, readmat)
> write.csv(matrixv, row.names=FALSE,
+     file="matrix_ex_rows.csv")
> readmat <- read.csv(file="matrix_ex_rows.csv")
> readmat <- as.matrix(readmat)
> readmat  # A matrix without row names
```

Writing and Reading Matrices (cont.)

There are several ways of writing and reading matrices from .csv files, with tradeoffs between simplicity, data size, and speed.

The function write.matrix() writes a matrix to a text file, without its row names.

write.matrix() is part of package *MASS*.

The advantage of function scan() is its speed, but it doesn't handle row names easily.

Removing row names simplifies the writing and reading of matrices.

The function readLines reads whole lines and returns them as single strings.

```
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> library(MASS)  # Load package "MASS"
> # Write to CSV file by row - it's very SLOW!!!
> MASS::write.matrix(matrixv, file="matrix.csv", sep=",")
> # Read using scan() and skip first line with colnames
> readmat <- scan(file="matrix.csv", sep=",", skip=1,
+   what=numeric())
> # Read colnames
> colnamev <- readLines(con="matrix.csv", n=1)
> colnamev  # this is a string!
> # Convert to char vector
> colnamev <- strsplit(colnamev, split=",")[[1]]
> readmat  # readmat is a vector, not matrix!
> # Coerce by row to matrix
> readmat <- matrix(readmat, ncol=NROW(colnamev), byrow=TRUE)
> # Restore colnames
> colnames(readmat) <- colnamev
> readmat
> # Scan() is a little faster than read.csv()
> library(microbenchmark)
> summary(microbenchmark(
+   read_csv=read.csv("matrix.csv"),
+   scan=scan(file="matrix.csv", sep=",",
+     skip=1, what=numeric()),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Reading Matrices Containing Bad Data

Very often data that is read from external sources contains elements with bad data.

An example of bad data are `character` strings within sets of `numeric` data.

Columns of numeric data that contain strings are coerced to `character` or `factor`, when they're read by `read.csv()`.

The function `as.numeric()` coerces complex data objects into `numeric` vectors, and removes all their *attributes*.

`as.numeric()` coerces strings that don't represent numbers into `NA` values.

```
> # Read data from a csv file, including row names
> matrixv <- read.csv(file="matrix_bad.csv", row.names=1)
> matrixv
> class(matrixv)
> # Columns with bad data are character or factor
> sapply(matrixv, class)
> # Coerce character column to numeric
> matrixv$col2 <- as.numeric(matrixv$col2)
> # Or
> # Copy row names
> rownames <- row.names(matrixv)
> # sapply loop over columns and coerce to numeric
> matrixv <- sapply(matrixv, as.numeric)
> # Restore row names
> row.names(matrixv) <- rownames
> # Replace NAs with zero
> matrixv[is.na(matrixv)] <- 0
> # matrix without NAs
> matrixv
```

Writing and Reading Time Series From *Text* Files

The package *zoo* contains functions `write.zoo()` and `read.zoo()` for writing and reading *zoo* time series from `.txt` and `.csv` files.

The functions `write.zoo()` and `read.zoo()` are *wrappers* for `write.table()` and `read.table()`.

The function `write.zoo()` writes the *zoo* series index as a character string in quotations `""`, to make it easier to read (parse) by `read.zoo()`.

Users may also directly use `write.table()` and `read.table()`, instead of `write.zoo()` and `read.zoo()`.

```
> # Create zoo with Date index
> dates <- seq(from=as.Date("2013-06-15"), by="day",
+         length.out=100)
> pricev <- zoo(rnorm(NROW(dates)), order.by=dates)
> head(pricev, 3)
> # Write zoo series to text file, and then read it back
> write.zoo(pricev, file="pricev.txt")
> pricezoo <- read.zoo("pricev.txt")  # Read it back
> all.equal(pricezoo, pricev)
> # Perform the same using write.table() and read.table()
> # First coerce pricev into data frame
> dframe <- as.data.frame(pricev)
> dframe <- cbind(dates, dframe)
> # Write pricev to text file using write.table
> write.table(dframe, file="pricev.txt",
+       row.names=FALSE, col.names=FALSE)
> # Read data frame from file
> pricezoo <- read.table(file="pricev.txt")
> sapply(pricezoo, class)  # A data frame
> # Coerce data frame into pricev
> pricezoo <- zoo::zoo(
+   drop(as.matrix(pricezoo[, -1])),
+   order.by=as.Date(pricezoo[, 1]))
> all.equal(pricezoo, pricev)
```

Writing and Reading Time Series From .csv Files

By default the functions `zoo::write.zoo()` and `zoo::read.zoo()` write data in *space*-delimited text format, but they can also write to *comma*-delimited `.csv` files by passing the parameter `sep=","`.

Single column *zoo* time series usually don't have a dimension attribute, and they don't have a column name, unlike multi-column *zoo* time series, and this can cause hard to detect bugs.

It's best to always pass the argument `"col.names=TRUE"` to the function `write.zoo()`, to make sure it writes a column name for a single column *zoo* time series.

Reading a `.csv` file containing a single column of data using the function `read.zoo()` produces a *zoo* time series with a `NULL` dimension, unless the argument `"drop=FALSE"` is passed to `read.zoo()`.

Users may also directly use `write.table()` and `read.table()`, instead of `write.zoo()` and `read.zoo()`.

```
> # Write zoo series to CSV file, and then read it back
> write.zoo(pricev, file="pricev.csv",
+     sep=",", col.names=TRUE)
> pricezoo <- read.zoo(file="pricev.csv",
+   header=TRUE, sep=",", drop=FALSE)
> all.equal(pricev, drop(pricezoo))
```

Writing and Reading Time Series With *Date-time* Index

The function `read.csv.zoo()` reads *zoo* time series from `.csv` files.

The function `xts::as.xts()` coerces *zoo* time series into *xts* series.

If the index of a *zoo* time series is a *date-time*, then `write.zoo()` writes the date and time fields as character strings separated by a *space* between them, inside quotations `""`.

Very often `.csv` files contain custom *date-time* formats, which need to be passed as parameters into `read.zoo()` for proper formatting.

The `"FUN"` argument of `read.zoo()` accepts a function for coercing the date and time columns of the input data into a *date-time* object suitable for the *zoo* index.

The function `as.POSIXct()` coerces character strings into `POSIXct` *date-time* objects.

```
> # Create zoo with POSIXct date-time index
> dates <- seq(from=as.POSIXct("2013-06-15"),
+         by="hour", length.out=100)
> pricev <- zoo(rnorm(NROW(dates)), order.by=dates)
> head(pricev, 3)
> # Write zoo series to CSV file, and then read it back
> write.zoo(pricev, file="pricev.csv",
+     sep=",", col.names=TRUE)
> # Read from CSV file using read.csv.zoo()
> pricezoo <- read.csv.zoo(file="pricev.csv")
> all.equal(pricev, pricezoo)
> # Coerce to xts series
> xtsv <- xts::as.xts(pricezoo)
> class(xtsv); head(xtsv, 3)
> # Coerce zoo series into data frame with custom date format
> dframe <- as.data.frame(pricev)
> dframe <- cbind(format(dates, "%m-%d-%Y %H:%M:%S"), dframe)
> head(dframe, 3)
> # Write zoo series to csv file using write.table
> write.table(dframe, file="pricev.csv",
+       sep=",", row.names=FALSE, col.names=FALSE)
> # Read from CSV file using read.csv.zoo()
> pricezoo <- read.zoo(file="pricev.csv",
+   header=FALSE, sep=",", FUN=as.POSIXct,
+   format="%m-%d-%Y %H:%M:%S", tz="America/New_York")
> # Or using read.csv.zoo()
> pricezoo <- read.csv.zoo(file="pricev.csv", header=FALSE,
+   format="%m-%d-%Y %H:%M:%S", tz="America/New_York")
> head(pricezoo, 3)
> all.equal(pricev, pricezoo)
```

Reading Time Series With `Numeric` *Date-time* Index

If the index of a time series is `numeric` (representing the *moment of time*, either as the number of days or seconds), then it must be coerced to a proper *date-time* class.

A convenient way of reading time series with a numeric index is by using `read.table()`, and then coercing the *data frame* into a time series.

The function `as.POSIXct.numeric()` coerces a `numeric` value representing the *moment of time* into a `POSIXct` *date-time*, equal to the *clock time* in the local *time zone*.

```
> # Read time series from CSV file, with numeric date-time
> datazoo <- read.table(file="/Users/jerzy/Develop/lecture_slides/da
+   header=TRUE, sep=",")
> # A data frame
> class(datazoo)
> sapply(datazoo, class)
> # Coerce data frame into xts series
> datazoo <- xts::xts(as.matrix(datazoo[, -1]),
+   order.by=as.POSIXct.numeric(datazoo[, 1], tz="America/New_York",
+                       origin="1970-01-01"))
> # An xts series
> class(datazoo)
> head(datazoo, 3)
```

Passing Arguments to the save() Function

The function `save()` writes objects to a binary file.

Object names can be passed into `save()` either through the `"..."` argument, or the `"list"` argument.

Objects passed through the `"..."` argument are not evaluated, so they must be either object names or character strings.

Object names aren't surrounded by quotes `""`, while character strings that represent object names are surrounded by quotes `""`.

Objects passed through the `"list"` argument are evaluated, so they may be variables containing character strings.

```
> var1 <- 1; var2 <- 2
> ls()  # List all objects
> ls()[1]  # List first object
> args(save)  # List arguments of save function
> # Save "var1" to a binary file using string argument
> save("var1", file="my_data.RData")
> # Save "var1" to a binary file using object name
> save(var1, file="my_data.RData")
> # Save multiple objects
> save(var1, var2, file="my_data.RData")
> # Save first object in list by passing to "..." argument
> # ls()[1] is not evaluated
> save(ls()[1], file="my_data.RData")
> # Save first object in list by passing to "list" argument
> save(list=ls()[1], file="my_data.RData")
> # Save whole list by passing it to the "list" argument
> save(list=ls(), file="my_data.RData")
```

Writing and Reading Lists of Objects

The function `load()` reads data from `.RData` files, and *invisibly* returns a vector of names of objects created in the workspace.

The vector of names can be used to manipulate the objects in loops, or to pass them to functions.

```
> rm(list=ls())  # Remove all objects
> # Load objects from file
> loadobj <- load(file="my_data.RData")
> loadobj  # vector of loaded objects
> ls()  # List objects
> # Assign new values to objects in  global environment
> sapply(loadobj, function(symbol) {
+   assign(symbol, runif(1), envir=globalenv())
+ })  # end sapply
> ls()  # List objects
> # Assign new values to objects using for loop
> for (symbol in loadobj) {
+   assign(symbol, runif(1))
+ }  # end for
> ls()  # List objects
> # Save vector of objects
> save(list=loadobj, file="my_data.RData")
> # Remove only loaded objects
> rm(list=loadobj)
> # Remove the object "loadobj"
> rm(loadobj)
```

Saving Output of R to a File

The function `sink()` diverts R *text* output (excluding graphics) to a file, or ends the diversion.

Remember to call `sink()` to end the diversion!

The function `pdf()` diverts graphics output to a *pdf* file (text output isn't diverted), in vector graphics format.

The functions `png()`, `jpeg()`, `bmp()`, and `tiff()` divert graphics output to graphics files (text output isn't diverted).

The function `dev.off()` ends the diversion.

```
> sink("sinkdata.txt")# Redirect text output to file
>
> cat("Redirect text output from R\n")
> print(runif(10))
> cat("\nEnd data\nbye\n")
>
> sink()  # turn redirect off
>
> pdf("Rgraph.pdf", width=7, height=4)  # Redirect graphics to pdf f
>
> cat("Redirect data from R into pdf file\n")
> myvar <- seq(-2*pi, 2*pi, len=100)
> plot(x=myvar, y=sin(myvar), main="Sine wave",
+    xlab="", ylab="", type="l", lwd=2, col="red")
> cat("\nEnd data\nbye\n")
>
> dev.off()  # turn pdf output off
>
> png("r_plot.png")  # Redirect graphics output to png file
>
> cat("Redirect graphics from R into png file\n")
> plot(x=myvar, y=sin(myvar), main="Sine wave",
+  xlab="", ylab="", type="l", lwd=2, col="red")
> cat("\nEnd data\nbye\n")
>
> dev.off()  # turn png output off
```

Package *data.table* for High Performance Data Management

The package *data.table* is designed for high performance data management.

The package *data.table* implements *data table* objects, which are a special type of *data frame*, and an extension of the *data frame* class.

Data tables are faster and more convenient to work with than *data frames*.

data.table functions are optimized for high performance (speed), because they are written in C++ and they perform operations by reference (in place), without copying data in memory.

Some of the attractive features of package *data.table* are:

- Syntax is analogous to SQL,
- Very fast writing and reading from files,
- Very fast sorting and merging operations,
- Subsetting using multiple logical clauses,
- Columns of type `character` are never converted to factors,

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

The package *data.table* has extensive documentation:
https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html
https://github.com/Rdatatable/data.table/wiki

Data Table Objects

Data table objects are a special type of *data frame*, and are derived from the class `data.frame`.

Data table objects resemble databases, with columns of different types of data, and rows of records containing individual observations.

The function `data.table::data.table()` creates a *data table* object.

Data table columns can be referenced directly by their names (without quotes), and their rows can be referenced without a following comma.

When a *data table* is printed (by typing its name) then only the top 5 and bottom 5 rows are displayed (unless `getOption("datatable.print.nrows")` is less than 100).

The operator `.N` returns the number of observations (rows) in the *data table*.

Data table computations are usually much faster than equivalent `R` computations, but not always.

```
> # Create a data table
> library(data.table)
> dtable <- data.table::data.table(
+   col1=sample(7), col2=sample(7), col3=sample(7))
> # Print dtable
> class(dtable); dtable
> # Column referenced without quotes
> dtable[, col2]
> # Row referenced without a following comma
> dtable[2]
> # Print option "datatable.print.nrows"
> getOption("datatable.print.nrows")
> options(datatable.print.nrows=10)
> getOption("datatable.print.nrows")
> # Number of rows in dtable
> NROW(dtable)
> # Or
> dtable[, NROW(col1)]
> # Or
> dtable[, .N]
> # microbenchmark speed of data.table syntax
> library(microbenchmark)
> summary(microbenchmark(
+   dt=dtable[, .N],
+   rcode=NROW(dtable),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Writing and Reading Data Using Package *data.table*

The easiest way to share data between `R` and `Excel` is through `.csv` files.

The function `data.table::fread()` reads from `.csv` files and returns a *data table* object of class `data.table`.

Data table objects are a special type of *data frame*, and are derived from the class `data.frame`.

The function `data.table::fread()` is over 6 times faster than `read.csv()`!

The function `data.table::fwrite()` writes to `.csv` files over 12 times faster than the function `write.csv()`, and 300 times faster than function `cat()`!

```
> # Read a data table from CSV file
> dir_name <- "/Users/jerzy/Develop/lecture_slides/data/"
> file_name <- file.path(dir_name, "weather_delays14.csv")
> dtable <- data.table::fread(file_name)
> class(dtable); dim(dtable)
> dtable
> # fread() reads the same data as read.csv()
> all.equal(read.csv(file_name),
+     setDF(data.table::fread(file_name)))
> # fread() is much faster than read.csv()
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=read.csv(file_name),
+   fread=setDF(data.table::fread(file_name)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Write data table to file in different ways
> data.table::fwrite(dtable, file="dtable.csv")
> write.csv(dtable, file="dtable2.csv")
> cat(unlist(dtable), file="dtable3.csv")
> # microbenchmark speed of data.table::fwrite()
> summary(microbenchmark(
+   fwrite=data.table::fwrite(dtable, file="dtable.csv"),
+   write_csv=write.csv(dtable, file="dtable2.csv"),
+   cat=cat(unlist(dtable), file="dtable3.csv"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Subsetting *Data Table* Objects

The square braces (brackets) "[]" operator subsets (references) the rows and columns of *data tables*.

Data table rows can be subset without a following comma.

Data table columns can be referenced directly by their names (without quotes, as if they were variables), after a comma.

Multiple *data table* columns can be referenced by passing a list of names.

The brackets "[]" operator is a *data.table* function, and all the commands inside the brackets "[]" are executed using code from the package *data.table*.

The dot `.()` operator is equivalent to the list function `list()`.

```
> # Select first five rows of dtable
> dtable[1:5]
> # Select rows with JFK flights
> jfk_flights <- dtable[origin=="JFK"]
> # Select rows JFK flights in June
> jfk_flights <- dtable[origin=="JFK" & month==6]
> # Select rows without JFK flights
> jfk_flights <- dtable[!(origin=="JFK")]
> # Select flights with carrier_delay
> dtable[carrier_delay > 0]
> # Select column of dtable and return a vector
> head(dtable[, origin])
> # Select column of dtable and return a dtable, not vector
> head(dtable[, list(origin)])
> head(dtable[, .(origin)])
> # Select two columns of dtable
> dtable[, list(origin, month)]
> dtable[, .(origin, month)]
> columnv <- c("origin", "month")
> dtable[, ..columnv]
> dtable[, month, origin]
> # Select two columns and rename them
> dtable[, .(orig=origin, mon=month)]
> # Select all columns except origin
> head(dtable[, !"origin"])
> head(dtable[, -"origin"])
```

Performing Computations on *Data Table* Columns

If the second argument in the brackets "[]" operator is a function of the columns, then the brackets return the result of the function's computations on those columns.

The second argument in the brackets "[]" can also be a list of functions, in which case the brackets return a vector of computations.

The brackets "[]" can evaluate most standard R functions, but they are executed using *data.table* code, which is usually much faster than the equivalent R functions.

The operator .N returns the number of observations (rows) in the *data table*.

```
> # Select flights with positive carrier_delay
> dtable[carrier_delay > 0]
> # Number of flights with carrier_delay
> dtable[, sum(carrier_delay > 0)]
> # Or standard R commands
> sum(dtable[, carrier_delay > 0])
> # microbenchmark speed of data.table syntax
> summary(microbenchmark(
+   dt=dtable[, sum(carrier_delay > 0)],
+   rcode=sum(dtable[, carrier_delay > 0]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Average carrier_delay
> dtable[, mean(carrier_delay)]
> # Average carrier_delay and aircraft_delay
> dtable[, .(carrier=mean(carrier_delay),
+      aircraft=mean(aircraft_delay))]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Number of flights from JFK
> dtable[origin=="JFK", NROW(aircraft_delay)]
> # Or
> dtable[origin=="JFK", .N]
> # In R
> sum(dtable[, origin]=="JFK")
```

Grouping *Data Table* Computations by Factor Columns

The *data table* brackets "[]" operator can accept three arguments: `[i, j, by]`

- `i`: the row index to select,
- `j`: a list of columns or functions on columns,
- `by`: the columns of factors to aggregate over.

The *data table* columns can be *aggregated* over categories (factors) defined by one or more columns passed to the "by" argument.

The "keyby" argument is similar to "by", but it sorts the output according to the categories used to group by.

Multiple *data table* columns can be referenced by passing a list of names.

The dot `.()` operator is equivalent to the list function `list()`.

```
> # Number of flights from each airport
> dtable[, .N, by=origin]
> # Same, but add names to output
> dtable[, .(flights=.N), by=.(airport=origin)]
> # Number of AA flights from each airport
> dtable[carrier=="AA", .(flights=.N), by=.(airport=origin)]
> # Number of flights from each airport and airline
> dtable[, .(flights=.N), by=.(airport=origin, airline=carrier)]
> # Average aircraft_delay
> dtable[, mean(aircraft_delay)]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Average aircraft_delay from each airport
> dtable[, .(delay=mean(aircraft_delay)), by=.(airport=origin)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+      by=.(airport=origin, month=month)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+      keyby=.(airport=origin, month=month)]
```

Sorting *Data Table* Rows by Columns

Standard R functions can be used inside the brackets "[]" operator.

The function `order()` calculates the permutation index, to sort a given vector into ascending order.

The function `setorder()` sorts the rows of a *data table* by reference (in place), without copying data in memory.

`setorder()` is over 10 times faster than `order()`, because it doesn't copy data in memory.

Several brackets "[]" operators can be chained together to perform several consecutive computations.

```
> # Sort ascending by origin, then descending by dest
> dtables <- dtable[order(origin, -dest)]
> dtables
> # Doesn't work outside dtable
> order(origin, -dest)
> # Sort dtable by reference
> setorder(dtable, origin, -dest)
> all.equal(dtable, dtables)
> # setorder() is much faster than order()
> summary(microbenchmark(
+   order=dtable[order(origin, -dest)],
+   setorder=setorder(dtable, origin, -dest),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Average aircraft_delay by month
> dtables[, .(mean_delay=mean(aircraft_delay)),
+       by=.(month=month)]
> # Chained brackets to sort output by month
> dtables[, .(mean_delay=mean(aircraft_delay)),
+   by=.(month=month)][order(month)]
```

Subsetting, Computing, and Grouping *Data Table* Objects

The special symbol `.SD` selects a subset of a *data table*.

The symbol `.SDcols` specifies the columns to select by the symbol `.SD`.

Inside the brackets `"[]"` operator, the `.SD` symbol can be treated as a virtual *data table*, and standard `R` functions can be applied to it.

The `"by"` argument can be used to group the outputs produced by the functions applied to the `.SD` symbol.

If the symbol `.SDcols` is not defined, then the symbol `.SD` returns the remaining columns not passed to the `"by"` operator.

```
> # Select weather_delay and aircraft_delay in two different ways
> dtable[1:7, .SD,
+      .SDcols=c("weather_delay", "aircraft_delay")]
> dtable[1:7, .(weather_delay, aircraft_delay)]
> # Calculate mean of weather_delay and aircraft_delay
> dtable[, sapply(.SD, mean),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> sapply(dtable[, .SD,
+      .SDcols=c("weather_delay", "aircraft_delay")], mean)
> # Return origin and dest, then all other columns
> dtable[1:7, .SD, by=.(origin, dest)]
> # Return origin and dest, then weather_delay and aircraft_delay co
> dtable[1:7, .SD, by=.(origin, dest),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> # Return first two rows from each month
> dtable[, head(.SD, 2), by=.(month)]
> dtable[, head(.SD, 2), by=.(month),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> # Calculate mean of weather_delay and aircraft_delay, grouped by o
> dtable[, lapply(.SD, mean),
+      by=.(origin),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> # Or simply
> dtable[, .(weather_delay=mean(weather_delay),
+          aircraft_delay=mean(aircraft_delay)),
+      by=.(origin)]
```

Modifying *Data Table* Objects by Reference

The special assignment operator ":=" allows modifying *data table* columns by reference (in place), without copying data in memory.

The computations on columns by reference can be *grouped* over categories defined by one or more columns passed to the "by" argument.

The computations are recycled to fit the size of each group.

The selected parts of columns can also be modified by reference, by combining the i and j arguments.

The special symbols .SD and .SDcols can be used to perform computations on several columns.

Modifying by reference is several times faster than standard R assignment.

```
> # Add tot_delay column
> dtable[, tot_delay := (carrier_delay + aircraft_delay)]
> head(dtable, 4)
> # Delete tot_delay column
> dtable[, tot_delay := NULL]
> # Add max_delay column grouped by origin and dest
> dtable[, max_delay := max(aircraft_delay), by=.(origin, dest)]
> dtable[, max_delay := NULL]
> # Add date and tot_delay columns
> dtable[, c("date", "tot_delay") :=
+        list(paste(month, day, year, sep="/"),
+             (carrier_delay + aircraft_delay))]
> # Modify select rows of tot_delay column
> dtable[month == 12, tot_delay := carrier_delay]
> dtable[, c("date", "tot_delay") := NULL]
> # Add several columns
> dtable[, c("max_carrier", "max_aircraft") := lapply(.SD, max),
+  by=.(origin, dest),
+  .SDcols=c("carrier_delay", "aircraft_delay")]
> # Remove columns
> dtable[, c("max_carrier", "max_aircraft") := NULL]
> # Modifying by reference is much faster than standard R
> summary(microbenchmark(
+   dt=dtable[, tot_delay := (carrier_delay + aircraft_delay)],
+   rcode=(dtable[, "tot_delay"] <- dtable[, "carrier_delay"] + dtab
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Adding *keys* to *Data Tables* for Fast Binary Search

The *key* of a *data table* is analogous to the row indices of a *data frame*, and it determines the ordering of its rows.

The function `data.table::setkey()` adds a *key* to a *data table*, and sorts the *data table* rows by reference according to the key.

`setkey()` creates the *key* from one or more columns of the *data frame*.

Subsetting rows using a *key* can be several times faster than standard R.

```
> # Add a key based on the "origin" column
> setkey(dtable, origin)
> haskey(dtable)
> key(dtable)
> # Select rows with LGA using the key
> dtable["LGA"]
> all.equal(dtable["LGA"], dtable[origin == "LGA"])
> # Select rows with LGA and JFK using the key
> dtable[c("LGA", "JFK")]
> # Add a key based on the "origin" and "dest" columns
> setkey(dtable, origin, dest)
> key(dtable)
> # Select rows with origin from JFK and MIA
> dtable[c("JFK", "MIA")]
> # Select rows with origin from JFK and dest to MIA
> dtable[.("JFK", "MIA")]
> all.equal(dtable[.("JFK", "MIA")],
+     dtable[origin == "JFK" & dest == "MIA"])
> # Selecting rows using a key is much faster than standard R
> summary(microbenchmark(
+   with_key=dtable[.("JFK", "MIA")],
+   standard_r=dtable[origin == "JFK" & dest == "MIA"],
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing *Data Table* Objects Into *Data Frames*

The functions data.table::setDT() and data.table::setDF() coerce *data frames* to *data tables*, and vice versa.

The *set* functions data.table::set*() perform their operations by reference (in place), without returning any values or copying data to a new memory location, which makes them very fast.

Data table objects can also be coerced into *data frames* using the function as.data.frame(), but it's much slower because it makes copies of data.

```
> # Create data frame and coerce it to data table
> dtable <- data.frame(col1=sample(7), col2=sample(7), col3=sample(7
> class(dtable); dtable
> data.table::setDT(dtable)
> class(dtable); dtable
> # Coerce dtable into data frame
> data.table::setDF(dtable)
> class(dtable); dtable
> # Or
> dtable <- data.table:::as.data.frame.data.table(dtable)
> # SetDF() is much faster than as.data.frame()
> summary(microbenchmark(
+   asdataframe=data.table:::as.data.frame.data.table(dtable),
+   setDF=data.table::setDF(dtable),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing xts Time Series Into *Data Tables*

An xts time series can be coerced into a *data table* by first coercing it into a *data frame* and then into a *data table* using the function data.table::setDT().

But then the time index of the xts series is coerced into strings, not dates.

An xts time series can also be coerced directly into a *data table* using the function data.table::as.data.table().

```
> # Coerce xts to a data frame
> pricev <- rutils::etfenv$VTI
> class(pricev); head(pricev)
> pricev <- as.data.frame(pricev)
> class(pricev); head(pricev)
> # Coerce data frame to a data table
> data.table::setDT(pricev, keep.rownames=TRUE)
> class(pricev); head(pricev)
> # Dates are coerced to strings
> sapply(pricev, class)
> # Coerce xts directly to a data table
> dtable <- as.data.table(rutils::etfenv$VTI,
+   keep.rownames=TRUE)
> class(dtable); head(dtable)
> # Dates are not coerced to strings
> sapply(dtable, class)
> all.equal(pricev, dtable, check.attributes=FALSE)
```

Package *fst* for High Performance Data Management

The package *fst* provides functions for very fast writing and reading of *data frames* from *compressed binary files*.

The package *fst* writes to *compressed binary files* in the `fst` fast-storage format.

The package *fst* uses the `LZ4` and `ZSTD` compression algorithms, and utilizes multithreaded (parallel) processing on multiple CPU cores.

The package *fst* has extensive documentation:
http://www.fstpackage.org/

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

Writing and Reading Data Using Package *fst*

The package *fst* allows very fast writing and reading of *data frames* from *compressed binary files* in the fst fast-storage format.

The function fst::write_fst() writes to .fst files over 10 times faster than the function write.csv(), and 300 times faster than function cat() write to .csv files!

The function fst::fread() reads from .fst files over 10 times faster than the function read.csv() from .csv files!

```
> # Read a data frame from CSV file
> dir_name <- "/Users/jerzy/Develop/lecture_slides/data/"
> file_name <- file.path(dir_name, "weather_delays14.csv")
> data.table::setDF(dframe)
> class(dframe); dim(dframe)
> # Write data frame to .fst file in different ways
> fst::write_fst(dframe, path="dframe.fst")
> write.csv(dframe, file="dframe2.csv")
> # microbenchmark speed of fst::write_fst()
> library(microbenchmark)
> summary(microbenchmark(
+   fst=fst::write_fst(dframe, path="dframe.csv"),
+   write_csv=write.csv(dframe, file="dframe2.csv"),
+   cat=cat(unlist(dframe), file="dframe3.csv"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # fst::read_fst() reads the same data as read.csv()
> all.equal(read.csv(file_name),
+     fst::read_fst("dframe.fst"))
> # fst::read_fst() is 10 times faster than read.csv()
> summary(microbenchmark(
+   fst=fst::read_fst("dframe.fst"),
+   read_csv=read.csv(file_name),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Random Access to Large Data Files

The package *fst* allows *random access* to very large *data frames* stored in compressed data files in the `.fst` format.

Data frames can be accessed *randomly* by loading only the selected rows and columns into memory, without fully loading the whole data frame.

function `fst::fst()` reads an `.fst` file and returns an *fst_table* reference object (pointer) to the data, without loading the whole data into memory.

The *fst_table* reference provides access to the data similar to a regular *data frame*, but it requires only a small amount of memory because the data isn't loaded into memory.

```
> # Coerce TAQ xts to a data frame
> library(HighFreq)
> taq <- HighFreq::SPY_TAQ
> taq <- as.data.frame(taq)
> class(taq)
> # Coerce data frame to a data table
> data.table::setDT(taq, keep.rownames=TRUE)
> class(taq); head(taq)
> # Get memory size of data table
> format(object.size(taq), units="MB")
> # Save data table to .fst file
> fst::write_fst(taq, path="/Users/jerzy/Develop/data/taq.fst")
> # Create reference to .fst file similar to a data frame
> refst <- fst::fst("/Users/jerzy/Develop/data/taq.fst")
> class(refst)
> # Memory size of reference to .fst is very small
> format(object.size(refst), units="MB")
> # Get sizes of all objects in workspace
> sort(sapply(mget(ls()), object.size))
> # Reference to .fst can be treated similar to a data table
> dim(taq); dim(refst)
> fst:::print.fst_table(refst)
> # Subset reference to .fst just like a data table
> refst[1e4:(1e4+5), ]
```

Reading Data From Excel Files

The package *readxl* reads data from Excel spreadsheet files into R.

The function read_excel() reads a single sheet (tab) from an Excel file.

The function read_xlsx() reads a single sheet (tab) from an Excel file in .xlsx format.

The functions from package *readxl* return a type of *data frame* called a *tibble* object.

The *tibble* classes tbl and tbl_df are derived from the *data frame* class data.frame.

tibble objects are also used by the package *dplyr*.

DataCamp offers a Tutorial on Importing Excel Files into R.

```
> # Install and load package readxl
> install.packages("readxl")
> library(readxl)
> dir_name <- "/Users/jerzy/Develop/lecture_slides/data"
> filev <- file.path(dir_name, "multi_tabs.xlsx")
> # Read a time series from first sheet of xlsx file
> tibblev <- readxl::read_xlsx(filev)
> class(tibblev)
> # Coerce POSIXct dates into Date class
> class(tibblev$Dates)
> tibblev$Dates <- as.Date(tibblev$Dates)
> # Some columns are character strings
> sapply(tibblev, class)
> sapply(tibblev, is.character)
> # Coerce columns with strings to numeric
> listv <- lapply(tibblev, function(x) {
+   if (is.character(x))
+     as.numeric(x)
+   else
+     x
+ })  # end lapply
> # Coerce list into xts time series
> xtsv <- xts::xts(do.call(cbind, listv)[, -1], listv[[1]])
> class(xtsv); dim(xtsv)
> # Replace NA values with the most recent non-NA values
> sum(is.na(xtsv))
> xtsv <- zoo::na.locf(xtsv, na.rm=FALSE)
> xtsv <- zoo::na.locf(xtsv, fromLast=TRUE)
```

Reading Multiple Sheets From Excel Files

The function `readxl::excel_sheets()` returns a vector of character strings with the names of all the sheets in an Excel spreadsheet.

The package *readxl* reads data from Excel spreadsheet files into R.

The function `read_excel()` reads a single sheet (tab) from an Excel file.

The function `read_xlsx()` reads a single sheet (tab) from an Excel file in `.xlsx` format.

The functions from package *readxl* return a type of *data frame* called a *tibble* object.

The *tibble* classes `tbl` and `tbl_df` are derived from the *data frame* class `data.frame`.

tibble objects are also used by the package *dplyr*.

```
> # Read names of all the sheets in an Excel spreadsheet
> namesv <- readxl::excel_sheets(filev)
> # Read all the sheets from an Excel spreadsheet
> sheets <- lapply(namesv, read_xlsx, path=filev)
> names(sheets) <- namesv
> # sheets is a list of tibbles
> sapply(sheets, class)
> # Create function to coerce tibble to xts
> to_xts <- function(tibblev) {
+   tibblev$Dates <- as.Date(tibblev$Dates)
+   # Coerce columns with strings to numeric
+   listv <- lapply(tibblev, function(x) {
+     if (is.character(x))
+       as.numeric(x)
+     else
+       x
+   })  # end lapply
+   # Coerce list into xts series
+   xts::xts(do.call(cbind, listv)[, -1], listv$Dates)
+ }  # end to_xts
> # Coerce list of tibbles to list of xts
> class(sheets)
> sheets <- lapply(sheets, to_xts)
> sapply(sheets, class)
> # Replace NA values with the most recent non-NA values
> sapply(sheets, function(xtsv) sum(is.na(xtsv)))
> sheets <- lapply(sheets, zoo::na.locf, na.rm=FALSE)
> sheets <- lapply(sheets, zoo::na.locf, fromLast=TRUE)
```

Performing Calculations in Excel Using R

Excel can run R using either VBA scripts, or through a *COM* interface (available on *Windows* only).

R can perform calculations and export its output to Excel files, or it can modify Excel files (requires packages using Java or Perl code).

Calculations in R and Excel can be combined in several different ways:

- Data from Excel can be exchanged with R via .csv files (simplest and best method),
- Excel can execute R commands using VBA scripts, and then import the R output from .csv files,
- An Excel add-in can execute R commands as Excel functions (relies on *COM* protocol, so works only for *Windows*): add-ins *BERT*, *RExcel*,
- R can modify Excel files and run Excel functions (requires packages using Java or Perl code): packages *xlsx*, *XLConnect*, *excel.link*,

```
> ### Perform calculations in R,
> ### And export to CSV files
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> # Subset data frame
> readf <- readf[readf[, "type"]=="daisy", ]
> # Write data frame to CSV file, with row names
> write.csv(readf, file="daisies.csv")
```

Running R Code from Excel

There are several ways of performing calculations in R and exporting the outputs to Excel:

- Export data from Excel via .csv files to R, perform the calculations in R, and import the outputs back to Excel via .csv files (simplest and best method),
- Run R from Excel using VBA scripts, and exchange data via .csv files,
- Run R from Excel using an Excel add-in, and execute R commands as Excel functions (relies on the *COM* protocol, so works only for *Windows*),

```
> ### Perform calculations in R,
> ### And export to CSV files
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> # Subset data frame
> readf <- readf[readf[, "type"]=="daisy", ]
> # Write data frame to CSV file, with row names
> write.csv(readf, file="daisies.csv")
```

Running R Code Using VBA Scripts

An R session can be launched from Excel using a VBA script (macro).

The VBA function shell() executes a program by running an executable *exe* file (with extension *exe*).

A VBA script can also run an R *batch* process.

The R *batch* process can write to .csv files, which can then be imported into Excel.

```
' VBA macro to run R process
Sub run_r()
 Call shell("R", vbNormalFocus)
End Sub
```

```
' VBA macro to run interactive R process
Sub run_rinteractive()
 Dim script_dir As String: script_dir = "C:\Develop\R\script
 Dim script_file As String: script_file = "plot_interactive.
 Dim log_file As String: log_file = "C:\Develop\R\scripts\lo
 Call shell("R --vanilla < " & script_dir & script_file & ">
End Sub
```

```
' VBA macro to run batch R process
Sub run_rbatch()
 Dim script_dir As String: script_dir = "C:\Develop\R\script
 Dim script_file As String: script_file = "plot_to_file.R"
 Dim log_file As String: log_file = "C:\Develop\R\scripts\lo
 Call shell("R --vanilla < " & script_dir & script_file & ">
End Sub
```

BERT Excel Add-in for Running R Code

BERT is an Excel add-in which allows executing R commands as Excel functions:

http://bert-toolkit.com/
http://bert-toolkit.com/bert-quick-start
https://github.com/sdllc/Basic-Excel-R-Toolkit/wiki
https://github.com/sdllc/Basic-Excel-R-Toolkit

BERT launches its own R process from Excel.

BERT can create its own menu in the Excel add-ins tab:
After installing *BERT*, click on upper-left *Office Button*, click Excel options, on the bottom of the window choose (Manage: *COM* Add-ins) Go, add the *COM* add-in BERTRibbon2x86.dll.

BERT relies on the *COM* protocol, so it works only for *Windows*.

```
' calculate sum of Excel cells using R
R.Add(B1:D1)

' remove NAs over Excel cell range using R function
R.na_omit(F2:H4)

' calculate eigenValues of Excel matrix using R function
R.EigenValues(A1:H8)
```

Package *googlesheets* for Interacting with *Google Sheets*

The package *googlesheets* allows interacting with *Google Sheets* using R commands.

If you already have a *Google* account, then your personal *Google Sheets* can be found at:
https://docs.google.com/spreadsheets/

The function gs_ls() listv the files in *Google Sheets*.

The function gs_title() registers a *Google* sheet, and returns a googlesheet object.

A googlesheet object contains information (metadata) about a *Google* sheet, such as its name and key, but not the sheet data itself.

The function gs_browse() opens a *Google* sheet in an internet browser.

You can find online a document about using *googlesheets*.

You can find online a document about managing authentication tokens.

```
> # Install latest version of googlesheets
> devtools::install_github("jennybc/googlesheets")
> # Load package googlesheets
> library(googlesheets)
> library(dplyr)
> # Authenticate authorize R to view and manage your files
> gs_auth(new_user=TRUE)
> # List the files in Google Sheets
> googlesheets::gs_ls()
> # Register a sheet
> google_sheet <- gs_title("my_data")
> # view sheet summary
> google_sheet
> # List tab names in sheet
> tab_s <- gs_ws_ls(google_sheet)
> # Set curl options
> library(httr)
> httr::set_config(config(ssl_verifypeer=0L))
> # Read data from sheet
> gs_read(google_sheet)
> # Read data from single tab of sheet
> gs_read(google_sheet, ws=tab_s[1])
> gs_read_csv(google_sheet, ws=tab_s[1])
> # Or using dplyr pipes
> google_sheet %>% gs_read(ws=tab_s[1])
> # Download data from sheet into file
> gs_download(google_sheet, ws=tab_s[1],
+       to="/Users/jerzy/Develop/lecture_slides/data/google_sheet.cs
> # Open sheet in internet browser
> gs_browse(google_sheet)
```

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

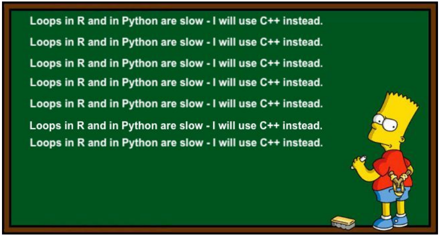
The package *Rcpp* allows calling C++ functions from R, by compiling the C++ code and creating R functions.

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

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

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

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




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

Function cppFunction() for Compiling C++ code

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

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

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

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

Performing Loops in *Rcpp Sugar*

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

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

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

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

Simulating Ornstein-Uhlenbeck Process Using *Rcpp*

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

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

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

Rcpp Attributes

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

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

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

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

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

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

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

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

Generating Random Numbers Using Logistic Map in *Rcpp*

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

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

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

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

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

Package *RcppArmadillo* for Fast Linear Algebra

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

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

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

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

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

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

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

// The function inner_mat() calculates the inner (dot) p
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
// It uses \emph{RcppArmadillo}.
//' @export
// [[Rcpp::export]]
double inner_mat(const arma::vec& vectorv2, const arma::
  return arma::as_scalar(trans(vectorv2) * (matrixv * ve
}  // end inner_mat
```

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

Simulating *ARIMA* Processes Using *RcppArmadillo*

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

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

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

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

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

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

  return arimav;
}  // end sim_arima
```

Fast Matrix Algebra Using *RcppArmadillo*

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

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

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

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

// Examples of \emph{RcppArmadillo} functions below

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

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

Fast Correlation Matrix Inverse Using *RcppArmadillo*

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

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

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

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

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

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

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

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

Portfolio Optimization Using *RcppArmadillo*

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

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

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

Strategy Backtesting Using *RcppArmadillo*

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

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

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

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

  // Return the strategy pnls
  return pnls;

}  // end back_test
```

Package *reticulate* for Running Python from RStudio

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

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

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

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

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

Running Python Under *reticulate*

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

Homework Assignment

No homework!