R403: Probabilistic and Statistical Computations with R

Topic 13: Introduction to R Graphics

Kaloyan Ganev

2022/2023

Lecture Contents

Introduction

Introduction

- Graphs are indispensable in studying and presenting data and results
- They provide a lot of insight and hints on where to go next with data analysis
- As you are aware, there are graphs to display any type of data
- R has enormous and flexible capabilities to chart data
- In general, there are two command types related to graphical output in R: commands to create a basic plot, and commands to tweak the output to one's liking
- We will discuss also specialized packages that have lots of tweaking pre-programmed so that you don't need to spend time on it but focus on more important stuff

Base R Plotting Capabilities

The plot() Function

- Generate some random and some not-so-random data:

```
x <- 1000*runif(50)
y <- 0.8*x + rnorm(50, mean = 0, sd = 75)
```

- We will start with scatterplots, as they are usually among the most frequently used ones
- To plot y against x, just type:

```
plot(x,y)
```

- Usually we have the response variable on the vertical axis, and the variable that influences it – on the horizontal axis
- The plot is not so impressive but does the job (for now)
- We will use it as a basis for expanding on R's charting capabilities

Tweaking the Basic Plots

- Let's create a data frame to see how data frame data is used in plots:

```
df1 <- as.data.frame(cbind(x,y))
```

- Rename the variables to something more meaningful:

```
colnames(df1) <- c("inc", "cons")
```

- ...and plot:

```
plot(df1$inc, df1$cons)
```

- The first thing that looks ugly are the axis labels; a chart title is also missing
- This can be corrected by adding some options to the graph:

```
plot(df1$inc, df1$cons, xlab = "Aggregate income", ylab = "
    Aggregate consumption", main = "Some macroeconomic aggregates
    ")
```

Tweaking the Basic Plots (2)

- The scales of the two variables can be tweaked by respectively `xlim` and `ylim` as in:

```
plot(df1$inc, df1$cons, xlab = "Aggregate income", ylab = "
    Aggregate consumption", main = "Some macroeconomic aggregates
    ", xlim = c(0,2000), ylim = c(0, 2000))
```

- Sometimes one would want to change the type of plotting character (empty dot by default)
- There are many available options which can be found here: https://www.r-bloggers.com/2021/06/r-plot-pch-symbols-different-point-shapes-in-r/
- Symbols are selected with the `pch` option, their size is controlled with the `cex` option, and their color – with the `col` option; see the following example:

```
plot(df1$inc, df1$cons, xlab = "Aggregate income", ylab = "
    Aggregate consumption", main = "Some macroeconomic aggregates
    ", pch = 20, cex = 2, col = "red")
```

Tweaking the Basic Plots (3)

- There are actually many more graphics options that can be utilized
- There is a default set of such parameters that is contained in a pre-specified list in R
- This list can be called with the `par()` function
- That same function allows to set the parameters so that they can be used as default
- The latter also prevents lots of typing when no (significant) changes of your graphs are necessary, and also makes code much neater
- In order to be able to go back to the original parameters, sometimes it is a good idea to save them in an object in your workspace:

```
saved_par <- par()
```

Tweaking the Basic Plots (4)

- We will use an example to demonstrate setting parameters with `par()`
- The following code shows some playing with it:

```
par(
    bg = "lightgray", # background of chart
    bty = "l", # box type around chart
    cex = 2.5, # magnification of symbols
    cex.axis = 0.4, # size of axis symbols relative to cex
    cex.lab = 0.5, # size of labels relative to cex
    cex.main = 0.67, # size of main title relative to cex
    col = "darkorange", # colour of symbols
    col.axis = "blue", # colour of axes symbols
    col.lab = "blue", # colour of axis labels
    col.main = "darkgreen", # colour of main title
    family = "serif",
    fig = saved_par$fig,
    fin = saved_par$fin,
    mar = c(4,4,2,2),
    pch = 20 # symbol for plotting
    )
```

Tweaking the Basic Plots (5)

- Now plot your figure with:

```
plot(df1$inc, df1$cons, xlab = "Aggregate income", ylab = "
    Aggregate consumption", main = "Some macroeconomic aggregates
    ")
```

- To switch back to default parameters:

```
par(saved_par)
```

- Let's now add a categorical variable to the data frame and name it "country":

```
f1 <- factor(sample(c(1:3),length(x),replace = T))
df1$country <- f1
```

- We can now plot the same data but colour each country differently:

```
plot(df1$inc, df1$cons, xlab = "Aggregate income", ylab = "
    Aggregate consumption", main = "Some macroeconomic aggregates
    ", col = df1$country, pch = 20, cex = 2)
```

Line Plots

- Still the `plot()` command is used
- As scatterplots use sorted observations, it is not advisable to turn them into line plots
- Instead, we take an individual variable and plot it, setting some options, too
- Start with a blanc plot to which we will later on add the lines:

```
plot(df1$inc, type = "l", lty = 0, ylab = "BGN", main = "Income
    and consumption")
```

- `type = "l"` is the option that makes it a line chart
- `lty` sets the type of line to use: can be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as character strings ”blank”, ”solid”, ”dashed”, ”dotted”, ”dotdash”, ”longdash”, or ”twodash”)

Line Plots (2)

- To add the first variable as a line:

```
lines(df1$inc, type = "l", lty = 1, lwd = 2, col = "red")
```

- Here, `lwd` sets line width
- To add another line to the same graph:

```
lines(df1$cons, type = "l", lty = 6, lwd = 2, col = "blue")
```

- Finally, we might add a legend:

```
legend(40,970, c("Income","Consumption"), lty=c(1,6), lwd=c(2,2),
     col=c("red","blue"))
```

Plotting Graphs of Functions

- The `curve()` function is used for that purpose
- Take the following function:

```
fun1 <- function(x){
  3*x^3 + 2*x^2 - 7*x + 11
  }
```

- ...and plot it:

```
curve(fun1, -10,10, lwd = 2, col = "red")
```

- To add coordinate axes:

```
abline(h = 0, lty = 2)
abline(v = 0, lty = 2)
```

Plotting Graphs of Functions (2)

- Too add more functions, e.g.:

```
fun2 <- function(x){
  1000*cos(x)
  }
curve(fun2, -10, 10, add = T, lwd = 2, col = "blue")
```

- Pay attention to the `add = T` option!

Pie Charts

- Created with the `pie()` function
- Example:

```
pie_data <- c(19,24,28,17,35)
pie_labels <- c("Apples", "Pears", "Cherries", "Oranges", "
    Bananas")
pie(pie_data, pie_labels, col = rainbow(length(pie_data)), main =
     "Fruit consumption")
```

- 3D pie charts are also possible, e.g. using the **plotrix** package:

```
library(plotrix)
pie3D(pie_data, labels = pie_labels, explode = 0.2, radius = 0.8,
     main = "Fruit consumption in 3D", labelcex = 1, labelcol = "
    blue")
```

Bar Plots

- Created with the `barplot()` function
- Both vertical and horizontal bars can be produced
- An example of vertical bars:

```
barplot(df1$inc, border = "darkgreen", col = "orange", xlab = "
    Index", ylab = "BGN", main = "Income levels")
```

- Same example but with horizontal bars:

```
barplot(df1$inc, horiz = T, border = "darkgreen", col = "orange",
     ylab = "Index", xlab = "BGN", main = "Income levels")
```

Bar Plots (2)

- To plot two variables together:

```
bar_data1 <- sample(30:50, 3)
bar_data2 <- sample(30:40, 3)
barplot(cbind(bar_data1,bar_data2), col = c("darkred", "
    darkorange", "yellow"), names.arg = c("My variable", "Your
    variable"))
```

- The same, with bars beside each other:

```
barplot(cbind(bar_data1,bar_data2), col = c("darkred", "
    darkorange", "yellow"), names.arg = c("My variable", "Your
    variable"), beside = T)
```

- Of course, there are many more parameters available

Histograms

- Generated by means of `hist()` and its options
- Example:

```
z <- rnorm(500)
hist(z, border = "darkblue", col = "orange", breaks = 20, freq =
    FALSE)
```

- If `freq = TRUE` then counts are displayed instead of the density which is output in the above example
- Check this one out too:

```
hist(z, border = "darkblue", col = "darkgreen", density = 40,
    breaks = 20, freq = FALSE)
```

- Here, the `density` option provides the density of shading lines used; their angle can be controlled with `angle`

Box Plots

- Created with `boxplot()`
- Simple example:

```
boxplot(df1$inc, col = "#D85625") # uses an HTML colour
```

- The example from my ANOVA lecture:

```
avgbuy <- read.csv("three_stores.csv")
boxplot(avgbuy, col = c("red","green","blue"))
```

Pairs Plots

- Each pairs plot is actually a matrix of scatter plots
- Used when your data set contains more than two variables and you would like to explore visually the (possible) association between any two variables
- To illustrate, we will use Fisher's `iris` dataset which is readily available in R
- Graphs are produced with:

```
iris_df <- as.data.frame(iris)
pairs(iris_df[,1:4], col = iris_df[,5])
```

Plotting 3D Surfaces

- R has some core functionality in this respect
- It is realized through the `persp()` function
- A simple example:

```
dome <- function(x,y){
  -(x^2 + y^2)
  }
x <- seq(from=-3, to = 3, by=0.1)
y <- seq(from=-3, to = 3, by=0.1)
z <- outer(x,y,dome)
persp(x,y,z,col="blue",theta=70,phi=-10)
```

Multiple Plots in One Graph

- To do that, you need first to create a special matrix which will hold the plots
- In fact, this matrix already exists and you've been using it all along
- It is just of size 1×1
- What is necessary is to resize it
- This is done again with the `par()` function using the `mfrow` option
- For example, in order to create a graph that has four plots in it (2×2):

```
par(mfrow = c(2,2))
plot(rnorm(100), type = "p")
plot(rnorm(100), type = "l")
plot(rnorm(100), type = "s")
plot(rnorm(100), type = "b")
```

Multiple Plots in One Graph (2)

- What if you need an unequal number of plots in each row/column of the graph?
- This is achieved with the `layout()` function
- Its argument is a matrix of integers in which each unique integer stands for a single object
- For example, to create a graph which has one plot in its first column and two plots in the second column, you use the following:

```
pos_m <- matrix(c(1,1,2,3), nrow = 2)
layout(pos_m)
plot(rnorm(100), type = "p")
plot(rnorm(100), type = "l")
plot(rnorm(100), type = "s")
```

- To return to the single-plot layout you can for example type:

```
par(mfrow = c(1,1))
```

Saving Graphs to Disk

- It is possible to export and save you graphical output to various graphical formats
- This is a very convenient feature which allows you to later on use graphs in your documents (paper, thesis, etc.)
- The functionality is provided by the **grDevices** package (comes with the base distribution)[1]
- Each export format is treated as a device
- Supported popular formats are BMP, JPEG, PNG, TIFF, PDF, Postscript, etc.

[1] See the full documentation here: https://stat.ethz.ch/R-manual/R-devel/library/grDevices/html/00Index.html.

Saving Graphs to Disk (2)

- There are two ways of exporting and saving
- First, you turn on the relevant device:

```
png("linegr1.png", height=600, width=800)
```

- Then you create the graph:

```
plot(rnorm(100), type = "l")
```

- Finally, you turn off the device:

```
dev.off()
```

- In this case, the graph is not visualised in R

Saving Graphs to Disk (3)

- The second approach boils down to copying and exporting the latest graphical output:

```
plot(rnorm(1000), type = "l", lwd = 1.5, col = "red", main = "
    Gaussian white noise")
dev.copy2pdf(file = "wn.pdf", height=6, width=8)
dev.copy2ps(file = "wn.eps", height=6, width=8)
```

- Note that in the last two examples (pdf and ps) no `dev.off()` is necessary

An introduction to the lattice package

The **lattice** Package

- R has a parallel graphics system called **grid**
- This system provides only low-level graphics functions but does not allow to produce complete plots
- **lattice** is one of the packages[2] that builds upon this system by providing high-level functions that allow creating complete graphs
- Load it with:

```
library(lattice)
```

[2] **ggplot2** is the other one considered later.

The **lattice** Package (2)

- The most basic plot is very similar to the ones considered with base R:

```
xyplot(cons ~ inc, data = df1)
```

- Analogically, they can be tweaked further, e.g. through:

```
df1$index <- c(1:length(df1$cons))
xyplot(cons ~ index, data = df1, type="o", lty = 2, pch=24, main=
    "Income vs. consumption")
```

- Bar charts are produced with `barchart()`:

```
barchart(cons ~ index, horizontal = FALSE, data = df1, main="
    Consumption", col = "darkred")
```

- Box (and whiskers) plots

```
rnd1 <- rnorm(1000)
bwplot(rnd1, col = "red", fill = "green")
```

The **lattice** Package (3)

- Histograms and density plots

```
histogram(rnd1, col = "orange")
densityplot(rnd1, lwd = 3)
```

- Q-Q plots:

```
qqmath(rnd1)
```

- Other capabilities of the **lattice** package may be explored by checking out the demo:

```
demo(lattice)
```

- There is also a package called **latticeExtra** which (as its name tells) extends **lattice**'s capabilities and range of graphs produced (see `http://latticeextra.r-forge.r-project.org/#panel.quantile&theme=default`)

An Introduction to the ggplot2 Package

The **ggplot2** Package

- As usual, in order to use it, first install it
- The name of the package comes from the title of a book whose conceptual models it implements (*The Grammar of Graphics* by L. Wilkinson)
- There are two ways to create graphics using the **ggplot2** package:
  - Using the `qplot()` command – to create quick plots
  - Using the `ggplot()` command (and the associated ones) to use the full potential of the package
- We will discuss each of the two in turn

Using `qplot()`

- `qplot()` is similar in its application to `plot()`
- One needs to specify only the data to be used:

```
qplot(cons, inc, data = df1, main = "Consumption vs. income")
```

- Quick plots are pre-formatted with default package settings
- There are some more tweaking options unused above but it is a better idea, if you need further graph customization (or, in fact, in principle), to learn more and use `ggplot()`

The `ggplot()` Command

- **ggplot2** sticks, as already mentioned, to the paradigm of the *The Grammar of Graphics* book
- This paradigm specifies that instead of using a separate function for each type of graph, graphs should be constructed by means of a small set of functions – where each function produces a graph component
- In other words, graphs are being constructed by means of layers and layer elements, each one using a specific function and its options
- The first essential command is `ggplot()`: it creates an empty plot
- In a way, you can think of the empty plot as of a painter's canvas with a prime (ground) layer applied to it
- When created, it waits for the painting to be created on it

The `aes()` and `geom()` Commands

- `aes()` stands for *aesthetics*
- Specifies how variables in the data are mapped to visual properties (aesthetics) of the so-called *geoms*
- *Geoms* themselves are the graphics shapes (lines, dots, bars, etc.) used to display the data
- The `geom()` command in its various forms adds the geoms to the graph

A Simple Example

- Take once again our consumption and income example (with random data)
- To create the base layer of the graph and save the result in a graph object, type the following:

```
gg_graph1 <- ggplot(df1)
```

- Then add the aesthetics together with the preferred shape

```
gg_graph1 <- gg_graph1 + geom_point(aes(x = inc, y = cons), size
     = 6, colour = "red")
```

- By this, we are actually adding a new layer to the graph (using the plus operator)
- In the above, we also added some extra formatting options

List of Geoms and Aesthetics

Geom	Description	Aesthetics
`geom_point()`	Data symbols	x, y, shape,fill
`geom_line()`	Line (ordered on x)	x, y, linetype
`geom_path()`	Line (original order)	x, y, linetype
`geom_text()`	Text labels	x, y, label, angle, hjust, vjust
`geom_rect()`	Rectangles	xmin, xmax, ymin, ymax, fill, linetype
`geom_polygon()`	Polygons	x, y, fill, linetype
`geom_segment()`	Line segments	x, y, xend, yend, linetype
`geom_bar()`	Bars	x, fill, linetype, weight
`geom_histogram()`	Histogram	x,fill,linetype,weight
`geom_boxplot()`	Boxplots	x, y, fill, weight
`geom_density()`	Density	x, y, fill, linetype
`geom_contour()`	Contour lines	x, y, fill, linetype
`geom_smooth()`	Smoothed line	x, y, fill, linetype
ALL	color, size, group	

(Table borrowed from Murrell (2012), p. 152)

Some More Examples with `ggplot()`

- `ggplot()` can be used to plot histograms and densities; for a demo, let's use the data found in the `iris` dataset
- The two sets of commands are as follows:

```
gg_hist <- ggplot(iris_df)
gg_hist <- gg_hist + geom_histogram(aes(x = Petal.Length, color=
    Species, fill=Species), alpha=I(0.5))

gg_dens <- ggplot(iris_df)
gg_dens <- gg_dens + geom_density(aes(x = Petal.Length, color=
    Species, fill=Species, alpha=I(0.5)))
```

- In order to make a box plot:

```
gg_box <- ggplot(iris_df)
gg_box <- gg_box + geom_boxplot(aes(x = Species, y = Petal.Length
    , fill=Species, alpha=I(0.5)))
```

- Here, `alpha` controls transparency
- etc.

Setting Scales in **ggplot2**

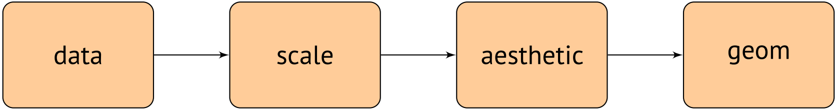
- Scales concern axes and legends
- Usually scales are appropriately set by the **ggplot2** package so there is no real need to tweak the limits of variables' values
- However, through adjusting scales' parameters it is for example possible to change the automatically set axis labels
- An example:

```
gg_points <- ggplot(df1)
gg_points <- gg_points + geom_point(aes(x = inc, y = cons, color
    = country, fill=country), size = I(6), alpha=I(0.5))
gg_points <- gg_points + scale_x_continuous(name="Household
    income (EUR)") +
        scale_y_continuous(name="Household consumption (EUR)")
```

Setting Scales in **ggplot2** (2)

- If still a change in an axis' limits is needed, it is achieved through the `limits = c(x_value, y_value)` option
- The colours of dots, respectively the legend, can be done in the following way:

```
gg_points <- gg_points +  scale_colour_manual(values = c("orange"
     ,"darkblue", "darkgreen"))
```



List of Scale Types and Parameters

Scale	Description	Parameters
`scale_x_continuous()`	Continuous axis	expand, trans
`scale_x_discrete()`	Categorical axis	
`scale_x_date()`	Date axis	major, minor, format
`scale_shape()`	Symbol shape legend	
`scale_linetype()`	Line pattern legend	
`scale_color_manual()`	Symbol/line color legend	values
`scale_fill_manual()`	Symbol/bar fill legend	values
`scale_size()`	Symbol size legend	trans, to
ALL		name, breaks, labels, limits

(Table borrowed from Murrell (2012), p. 157)

Statistical Transformations

- It is possible to map aesthetics not directly to raw (untransformed) data but to their transformations via statistical functions
- For example, this adds a polynomial regression estimate to the graph:

```
gg_points <- gg_points + stat_smooth(aes(x = inc, y = cons),
    method=lm, formula = y ~ poly(x,2), level=0.95)
```

Facets

- Data can be broken into subsets and then a separate plot for each subset can be made
- For this purpose, `facet_wrap()` is used
- This function requires a formula as an argument
- The formula provides a description of the the variable that will be used to subset the data
- We will use the income and consumption data frame to illustrate this

```
gg_facets <- ggplot(df1)
gg_facets <- gg_facets + geom_point(aes(x=inc, y=cons), colour =
    "red", size = I(3))
gg_facets <- gg_facets + facet_wrap(~ country, nrow=2)
```

Themes

- As it already became clear, **ggplot2** separates graph elements into data and non-data ones
- The data-related elements are represented by geoms, and the appearance of geoms is controlled by aesthetics
- Themes are collections of graphical parameters to control non-data elements
- We show some examples of themes in code but we will not dig into the details
- Details, however, allow customization of individual theme elements

Annotations

- Annotations are in general text labels placed over graphs for displaying additional information not directly inferable from the data
- Of course, this means placing an additional layer over the graph
- Check out the following two:

```
gg2 <- gg2 + geom_point(aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_text(aes(x = Sepal.Width, y = Sepal.Length, label =
      Species, color = Species))

gg2 <- gg2 + geom_point(aes(x = Sepal.Width, y = Sepal.Length)) +
    geom_label(aes(x =    Sepal.Width, y = Sepal.Length, label =
        Species, color = Species))
```

Annotations (2)

- An annotation layer can also be created in the following way:

```
gg3 <- ggplot(df1)
gg3 <- gg3 + geom_point(aes(x = inc, y = cons), size = 6, colour
    = "red")
gg3 <- gg3 + annotate("text", x = 100, y = 700, label = "A text
    annotation", size = I(6), color = "blue")

gg3 <- gg3 + annotate("rect", xmin = 100, xmax =  300, ymin = 0,
    ymax = 300, alpha = 0.2)

gg3 <- gg3 + annotate("segment", x = 600, xend =  900, y = 800,
    yend = 0, color = "darkgreen")
```

An introduction to the ggvis package

The **ggvis** Package

- Created by the RStudio team, a quite new project
- Current version is 0.4 (far from version 1.0 but still very promising)
- Like **ggplot2**, it is based on the *Grammar of Graphics* philosophy
- However, in addition it combines the above with the **Vega** model[3] which allows to draw raster graphics in the HTML 5 canvas or vector graphics in the svg format using JavaScript
- This makes it possible to render graphics in a standard web browser while allowing interactive plots
- As RStudio in fact possesses many features of browsers, it is possible to visualize graphics directly in it

[3] https://vega.github.io/vega/

The **ggvis** Package (2)

- The full (available as of now) documentation can be found here: http://ggvis.rstudio.com/
- We will not consider the details of the package as they are not directly relevant to the programme courses
- We will only take a look at the online examples

An Introduction to the rbokeh Package

The **rbokeh** Package

- **rbokeh** is similar to **ggvis**
- Also quite new, current version 0.5.0
- Uses the **Bokeh** library which has interfaces to Python, Scala, R, and Julia
- Again, we will refrain from exploring its syntax, instead we will look again at the online examples: `https://hafen.github.io/rbokeh/`
- As you can see there, development is in its relatively early stages
- Nevertheless, it is a promising project, too

Plotting time series

Plotting Time Series

- Plotting time series is in general not different from plotting other objects
- However, here we discuss it for two purposes:
  1. To make a smooth transition to real-life data
  2. To make a quick review of plotting approaches using time series objects
- Start with places where many datasets can be found so that we can use them for illustration purposes: https://www.quora.com/Where-can-I-find-large-datasets-open-to-the-public
- We will for the examples the Quandl database
- This database has an R interface to directly download data to your R IDE

Quandl

- In order to explore the contents of Quandl, it is necessary to create an account at https://www.quandl.com/
- Other than that, you don't need login credentials to download data to R
- Install the package and load it:

```
library(Quandl)
```

- Note that it also loads automatically **xts** (and, of course, **zoo**)
- Let's download data on the price of gold from the Bundesbank database:

```
data1 <- Quandl("BUNDESBANK/BBK01_WT5511")
```

- A dataframe is created

Plotting the Data

- We can use standard plotting functionality to plot the data:

```
plot(data1$Date, data1$Value, type = "l")
```

- The plot will be roughly correct except for the fact that it does not refer to 'real' time series data

```
class(data1)
```

- It is not a good idea to convert to a **ts** object as the data is of irregular frequency
- Therefore, it is better to have them in the **xts** format:

```
xts1 <- as.xts(data1$Value, order.by = data1$Date)
names(xts1)[1] <- "gold_price"
```

Plotting the Data (2)

- It turns out so that **xts** introduces its own plotting functionality after loading
- Check out:

```
plot(xts1[,1], col = "red", main = "Gold Price")
```

- Actually, `plot()` is in this case `plot.xts()`
- Good, but not very customizable
- Let's plot the same series with **lattice**

```
library(lattice)
xyplot(xts1, type=c("l","g"), xlab="",main="Gold Price")
```

- Now create a new **xts** object containing also percentage changes besides levels:

```
xts2 <- merge(xts1[,1],diff.xts(log(xts1[,1])))
colnames(xts2)[2] <- "change"
```

Plotting the Data (3)

- To plot the two series together but in split plots:

```
xyplot(xts2, type=c("l","g"), strip = strip.custom(factor.levels
    = c("Gold Price, USD","Percentage change")), main="Gold Price
    ")
```

- To tweak it further, save the **lattice** parameters for later use with:

```
savepar <- trellis.par.get()
```

- Then set the new parameters and plot:

```
trellis.par.set(strip.background = list(col="#0080ff"))
xyplot(xts2, type=c("l","g"), strip = strip.custom(factor.levels
    = c("Gold Price, USD","Percentage change")), par.strip.text =
     list(col="white", font = 2), main="Gold Price")
```

- To return to the old parameters:

```
trellis.par.set(savepar)
```

Plotting the Data (4)

- The last plotting approach that will be considered explicitly is that using **ggplot2**

```
gg1 <- ggplot(data = xts2) +
  geom_line(aes(x = Index, y = gold_price), colour = "#0080ff") +
  theme_bw() +
  ggtitle("Gold price, USD") +
  theme(plot.title = element_text(face="bold", colour = "red"))
gg1

gg2 <- ggplot(xts2) +
  geom_line(aes(x = Index, y = change), colour = "#0080ff") +
  theme_bw() +
  ggtitle("Change, %") +
  theme(plot.title = element_text(face="bold", colour = "red"))
gg2
```

Plotting the Data (5)

- With the **gridExtra** package, for example, it is possible to combine the above two graphs into a single one:

```
library(gridExtra)
grid.arrange(gg1, gg2, ncol=1)
```

Plotting the Data (6)

- What if we want several series in a single plot?
- Let's first download some more data (exchange rates, unimportant otherwise):

```
data2 <- Quandl("BOE/XUDLGBD")
data3 <- Quandl("BOE/XUDLADD")
```

- Generate **xts** objects:

```
xts3 <- as.xts(data2$Value, order.by = data2$Date)
xts4 <- as.xts(data3$Value, order.by = data3$Date)
```

- Merge them and change column names:

```
xts5 <- merge.xts(xts3, xts4)
colnames(xts5) <- c("GBP_USD", "AUD_USD")
```

Plotting the Data (8)

- Make the graph:

```
ggplot(xts5)  +
  geom_line(aes(x = Index, y = GBP_USD, color = "GBP/USD")) +
  geom_line(aes(x = Index, y = AUD_USD, color = "AUD/USD")) +
  scale_color_manual("", values = c("darkred", "darkblue")) +
  theme_bw() +
  xlab("") +
  ylab("") +
  annotate("rect",
        xmin = as.Date("2008-01-01"),
        xmax = as.Date("2014-12-18"),
        ymin = min(xts5[,1]),
        ymax = max(xts5[,2]), fill = "orange", alpha = I(0.2))
```

Don't Miss Out!

Plotly! (https://plot.ly/)

R package:
https://cran.r-project.org/web/packages/plotly/index.html

References

- Sarkar, D. (2008): *Lattice: Multivariate Data Visualization with R*, Springer
- Teutonico, D. (2015): *ggplot2 Essentials*, Packt Publishing
- Wickham, H. (2009): *ggplot2: Elegant Graphics for Data Analysis*, Springer
- Zuur, A., E. Ieno and E. Meesters (2009): *A Beginner's Guide to R*, Springer, Ch. 5