Data manipulation and visualization with R

3.1 Basic visualization functions

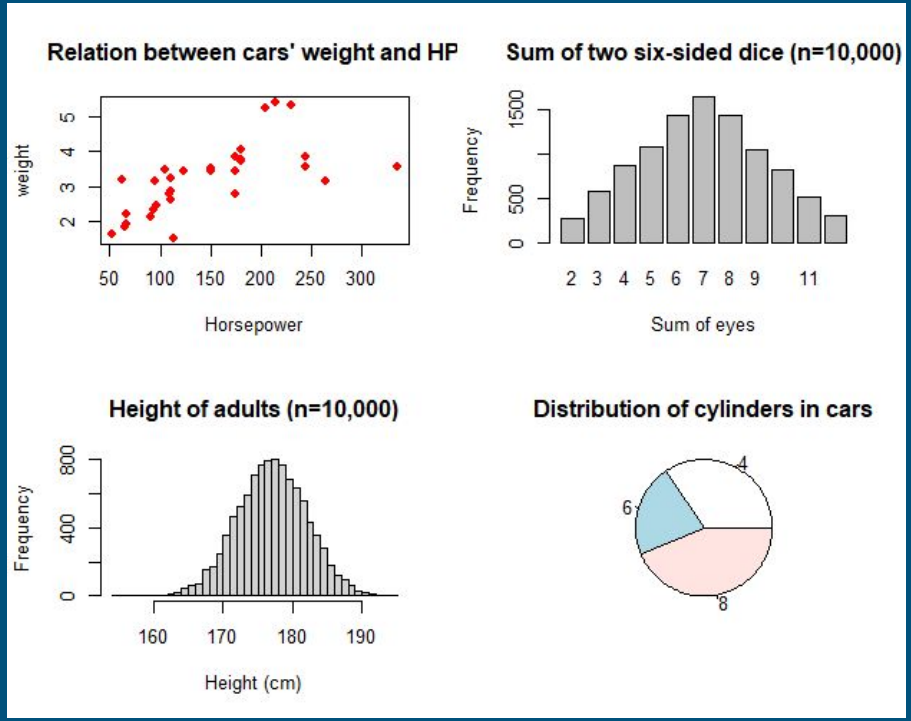
Hannes Seller, Jonathan Young (2020)

Introduction

The language R provides many functions to visualize data in different ways: scatter plots, histograms, bar plots or pie charts to name only a few.

In this course we will look at the most common visualization techniques that are provided by R (without the use of additional packages, which will be subject to the next course):

- Visual data exploration
- Scatter plots (x/y plots)
- Histograms
- Bar plots
- Pie charts
- Combining plots
- Exporting plots to files



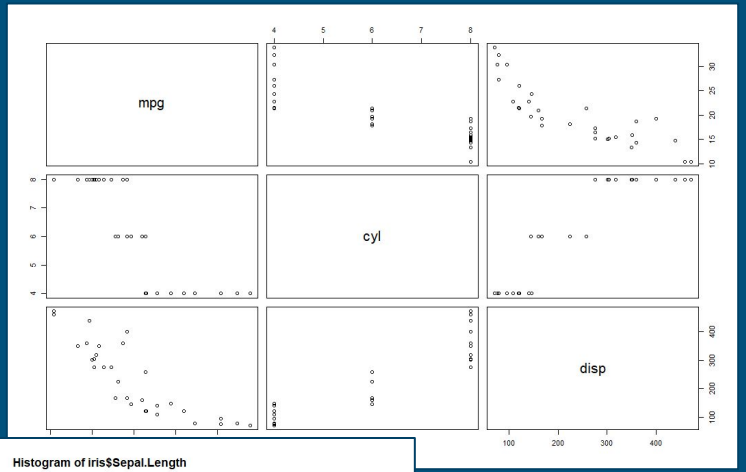
3.1 Basic visualization functions | Hannes Seller, Jonathan Young (2020)

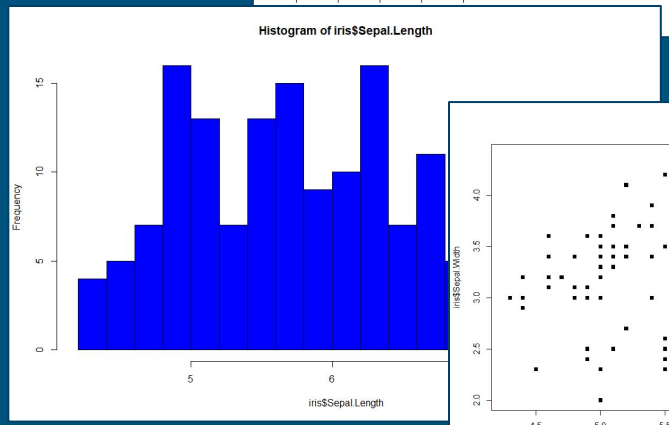
Visual data exploration

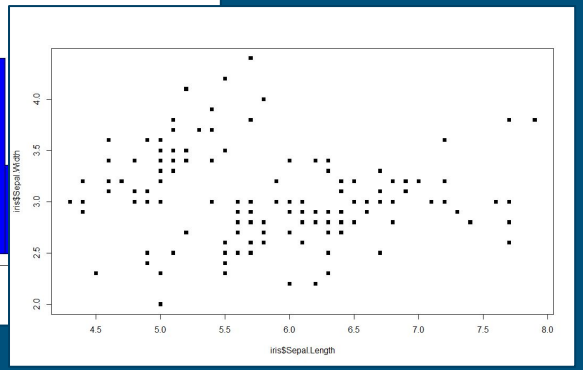
Let's have a quick look!

An important initial step in data analysis is visual data exploration. An image says more than a thousand words, and a plot of data points more than any table could. Visual exploration is often considered "quick and dirty", as plots do not have to be aesthetically pleasing to convey first impressions of the dataset.

From an x/y plot, tendencies and outliers in the data might be observable. A histogram provides a quick glance at the data's distribution. Visualizations with standard R functions are ideal for quick explorations, however, they might not deliver the most presentable graphics. In this course, we will focus on standard functions for the purpose of exploration, hence, we do not care too much about their looks. The upcoming course, in contrast, will focus on "nicer" visualizations using the package ggplot2.







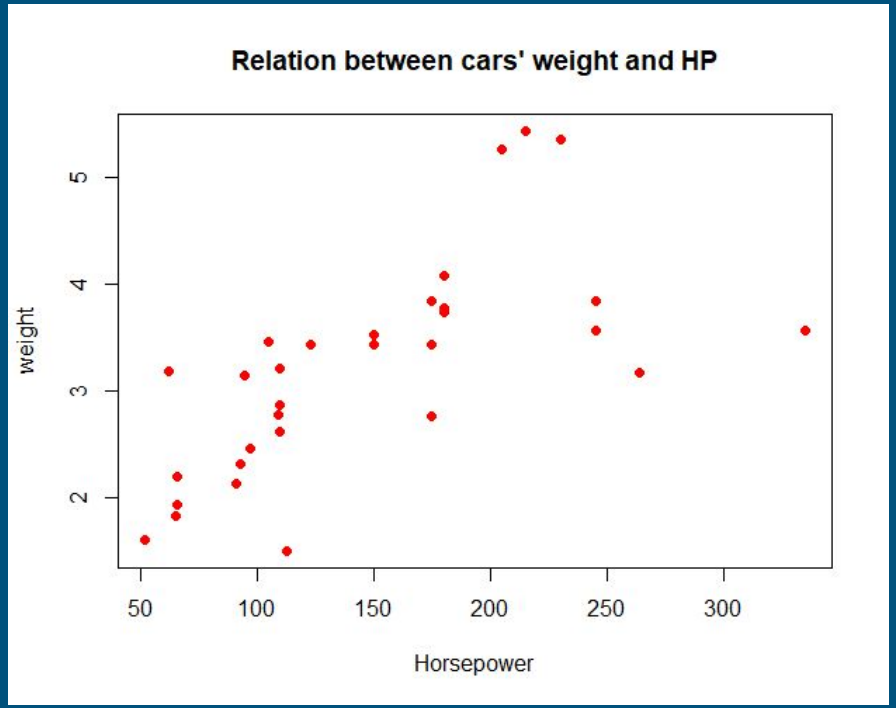
Scatter plots

plot( )

Scatter plots (also known as x/y plots) visualize one variable on the horizontal x-axis and one variable on the vertical y-axis. These plots are ideal to show the relation between two variables (e.g. a car's weight and its horsepower) or mathematical functions. They are created by the plot( ) function. The function has following essential parameters:

- x, y # input for x and y variables
- col, pch # color and shape of dots
- xlab, ylab # labels for x and y axes
- main # plot title
- type # type of visualization
- lty, lwd # type and width of lines
- xlim, ylim # cropping x and y axes

We will now go through an example and gradually build a plot.



Scatter plots

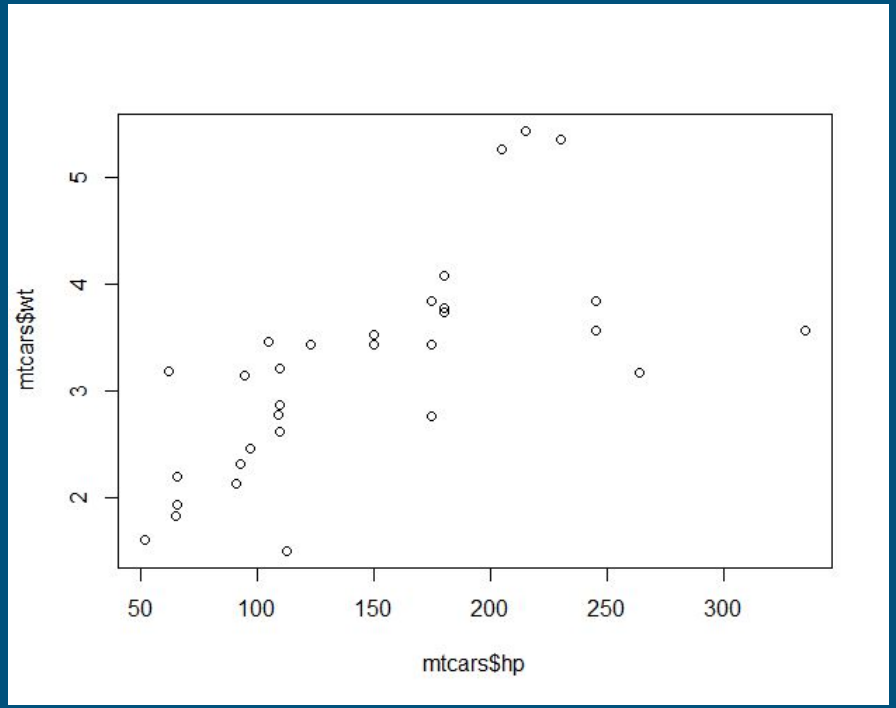
plot(x, y)

The input for the x and y variables has to fulfill two conditions:

- The two variables must be formatted as vectors
- The Variables must be the same length

The following code gives the most basic visualization of two variables (non-solid dots and axes labels based on vector names) of the dataset mtcars which contains the data of 32 cars. (This dataset is always available within RStudio.) The variables can be assigned to the parameters x and y. If not specified, R will interpret the first entry as x and the second as y.

```
plot(x=mtcars$hp, y=mtcars$wt) # specifying x and y
plot(mtcars$hp, mtcars$wt)     # implied x and y variables
```



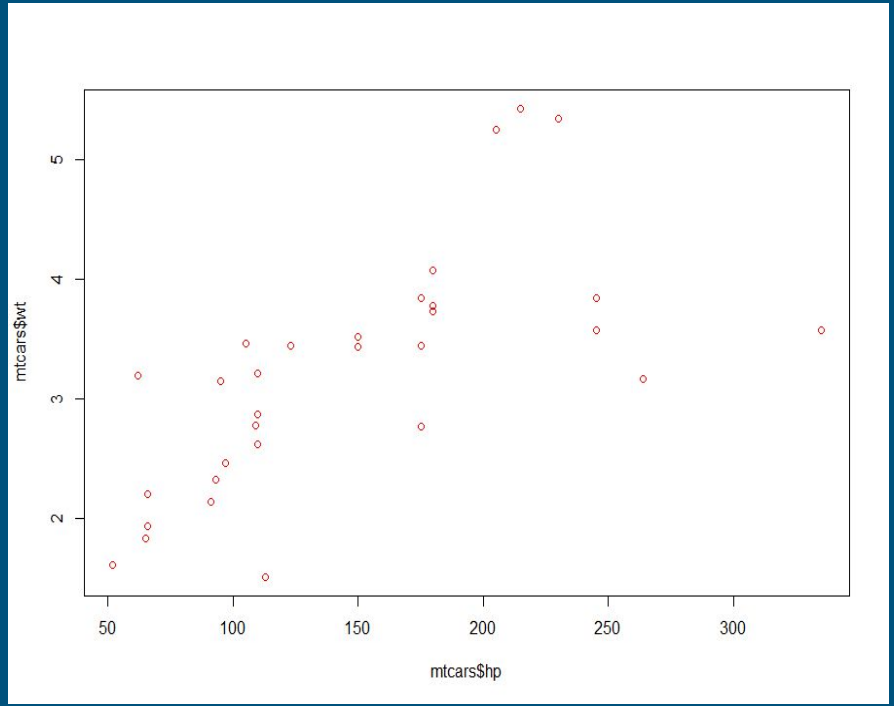
Scatter plots

plot(x, y, col)

The col parameter determines the color of dots. There are different ways to specify the color:

- Color names like “red”, “blue” or “green” are easy and intuitive. They are humanly readable which makes the code easy to maintain. A full list of usable colors is given here: (http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf)
- Colors based on hexadecimal numbers representing the share of Red/Green/Blue (https://www.color-hex.com/)
- There are also functions that create colors palettes like the rainbow(n) function creating a vector with n different colors.

```
plot(x=mtcars$hp, y=mtcars$wt, col="red") # color names
plot(x=mtcars$hp, y=mtcars$wt, col="#FF0000") # HEX codes
plot(x=mtcars$hp, y=mtcars$wt, col=rainbow(1)) # functions
```



Scatterplots

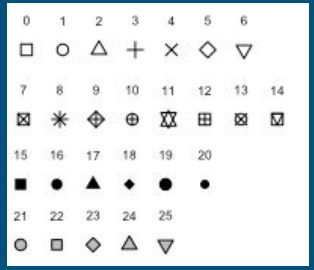
plot(x, y, col, pch)

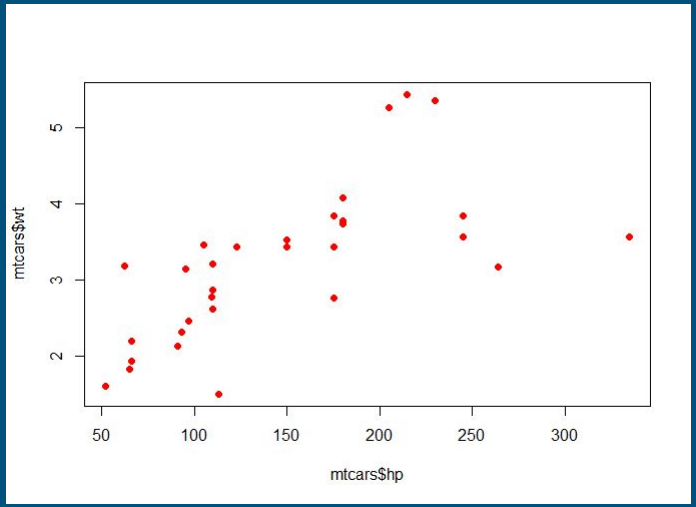
The pch parameter stands for “plot character” and determines the visual representation of each data point. By default the plot uses non-solid dots (i.e. colored outlines of empty circles).

All options can be displayed in RStudio calling ?pch in the console. A 0 creates non-solid squares, while a 15 creates solid squares filled by the chosen color.

It is highly recommended to use only a few basic options here, e.g. circles, triangles and squares to keep the plot simple.

```
# solid (i.e. colored) circles
plot(x=mtcars$hp, y=mtcars$wt, col="red", pch=16)
```





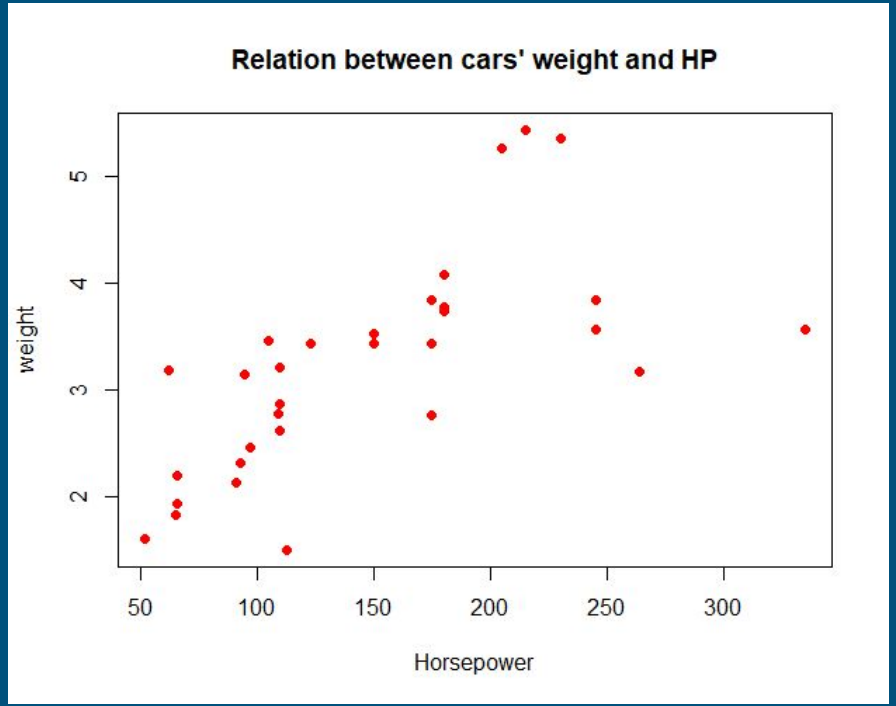
Scatterplots

plot(x, y, col, xlab, ylab, main)

The parameters xlab and ylab use strings (vectors containing text) to add labels to the x and y axes while main adds a title to the plot.

The following example creates the plot previously shown as an example. Please note that the code of the plot( ) function is here written over several lines. Adding line breaks after parameters can be useful to make the code easier to read and maintain. Thus, even annotations can be added into the function's call.

```
plot(mtcars$hp, mtcars$wt, # horsepower against weight
    col="red", pch=16,  #red filled circles
    xlab="Horsepower", ylab="Weight",
    main="Relation between cars' weight and HP"
)
```



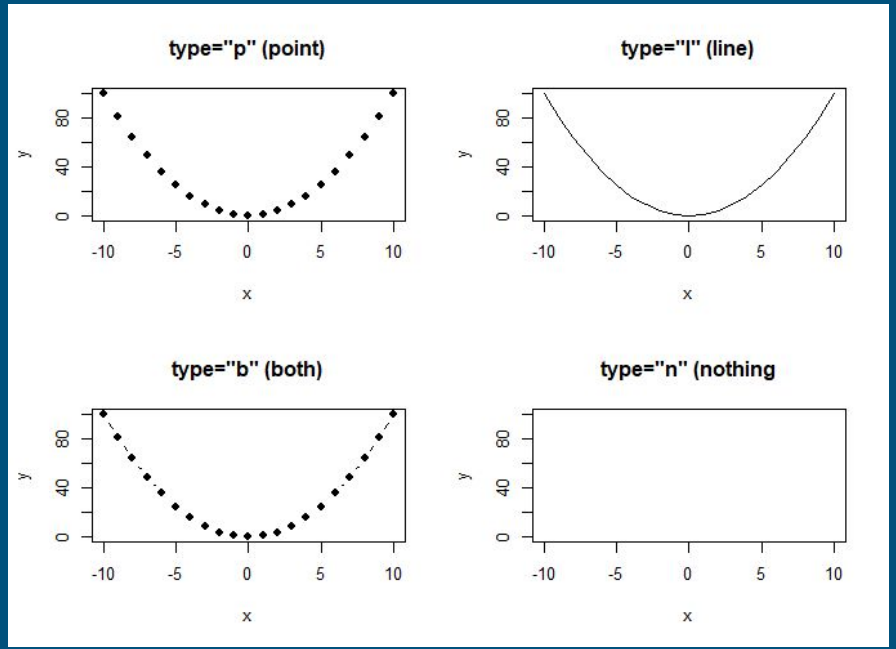
Scatter plots

plot(x, y, type)

The type parameter can be used to switch from separated dots to connected lines. However, this might not be meaningful for all types of data (like the cars that are individual observations). Line-typed scatter plots can be used to display mathematical functions though.

The example shows four different plot types. Type “p” is the default and will be used if no type is specified.

```
x <- c(-10:10)      # x variable: all integers from -10 to 10
y <- x^2            # y variable: x squared
plot(x, y, pch=16, type="p", main='type="p" (point)')
plot(x, y, pch=16, type="l", main='type="l" (line)')    # note: minor L
plot(x, y, pch=16, type="b", main='type="b" (both)')
plot(x, y, pch=16, type="n", main='type="n" (nothing')
```

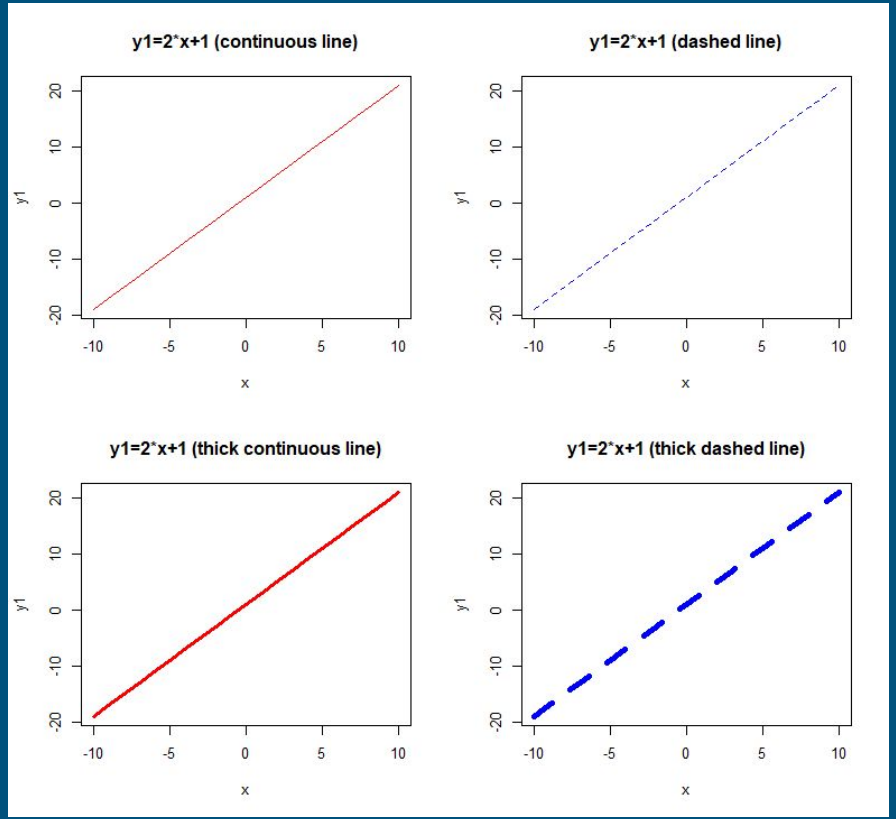


Scatter plots

plot(x, y, lty, lwd)

The lines can further be adjusted using the parameters lty (line type) and lwd (line width). Please note the difference between setting the type to visualization to a line (type="l") and changing the type of the line (lty=...). The line types (lty) are indicated by integers with 1 being the default continuous line and 2 being a dashed line. (All higher integers give variations of a dashed lines.) The width is 1 by default and cannot be lower than that. A width of 2 or 3 is usually more than enough to emphasize a certain plot.

```
x <- c(-10:10)      # x variable: all integers from -10 to 10
y <- 2*x+1          # function y = 2*x+1
plot(x,y, type="l", col="red", lty=1, lwd=1)
plot(x,y, type="l", col="blue", lty=2, lwd=1)
plot(x,y, type="l", col="red", lty=1, lwd=3)
plot(x,y, type="l", col="blue", lty=2, lwd=5)
```



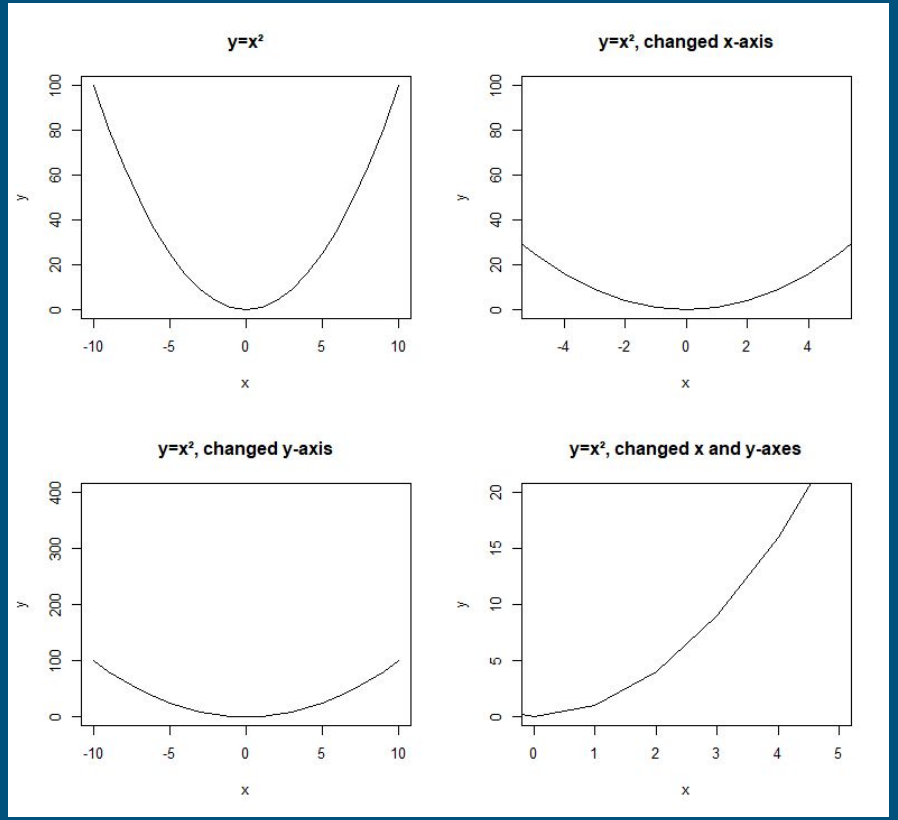
Scatter plots

plot(x, y, xlim, ylim)

The parameters xlim and ylim determine how much of each axis is shown. By default, the plot is sized to include all data points x/y. xlim and ylim require vectors that specify the interval, e.g. xlim=c(-5,5) reaches from x=-5 to x=5 . Thus, the user can limit which data points will be included in the graph or zoom into a specific area of a mathematical function.

Be aware of cropped or stretched axes as they can be used to manipulate how a reader interprets the plot. For example, the second plot suggests a smaller slope of the function.

```
plot(x,y, pch=16, type="l", main='y=x²')
plot(x,y, pch=16, type="l", xlim=c(-5,5))
plot(x,y, pch=16, type="l", ylim=c(0,400))
plot(x,y, pch=16, type="l", ,xlim=c(0,5), ylim=c(0,20))
```



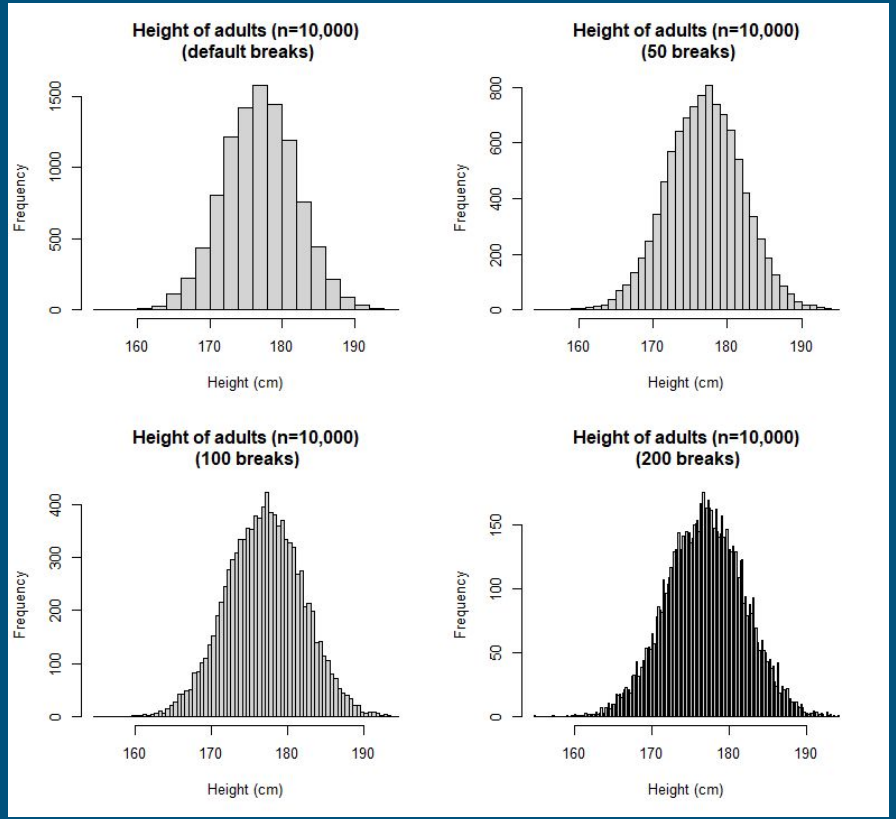
Histograms

hist(x)

Histograms are usually used to show distributions of datasets and frequencies of specific values or value intervals.

In this example we simulate the height of an adult population and visualize them with hist( ):

```
# creating 10,000 normally distributed values with …
# … a mean of 177 and a standard deviation of 5:
x <- rnorm(10000, 177, 5)

hist(x, col="lightgray", xlab="Height (cm)")
hist(x, breaks=50, col="lightgray", xlab="Height (cm)")
hist(x, breaks=100, col="lightgray", xlab="Height (cm)")
hist(x, breaks=200, col="lightgray", xlab="Height (cm)")
```



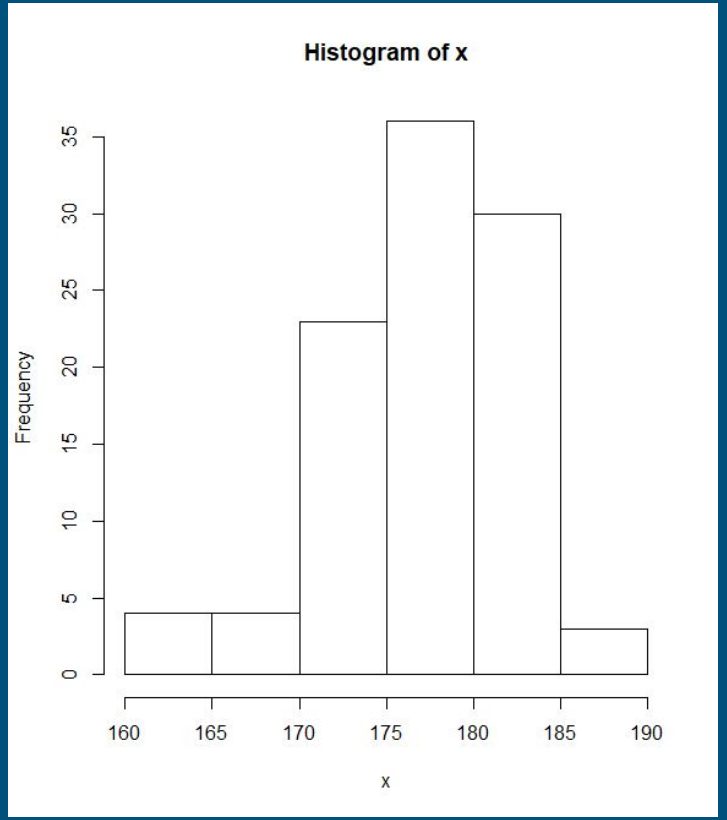
Histograms

h <- hist(x)

The hist( ) creates a histogram object that can be assigned to a variable. Thus, statistics about the distribution become available.

In this example, the first bin includes all numbers (4 cases) from 160 to 165. The individual counts divided by their sum give the percentage of each bin, e.g. 4% for the first bin.

```
# creating numbers with seed 42 (to get the same random numbers every time)
set.seed(42); x <- rnorm(n=100, m=177, sd=5)

h <- hist(x)
h$breaks          # output: 160 165 170 175 180 185 190
h$counts          # output: 4  4 23 36 30  3
h$counts/sum(h$counts) # output: 0.04 0.04 0.23 0.36 0.30 0.03
```

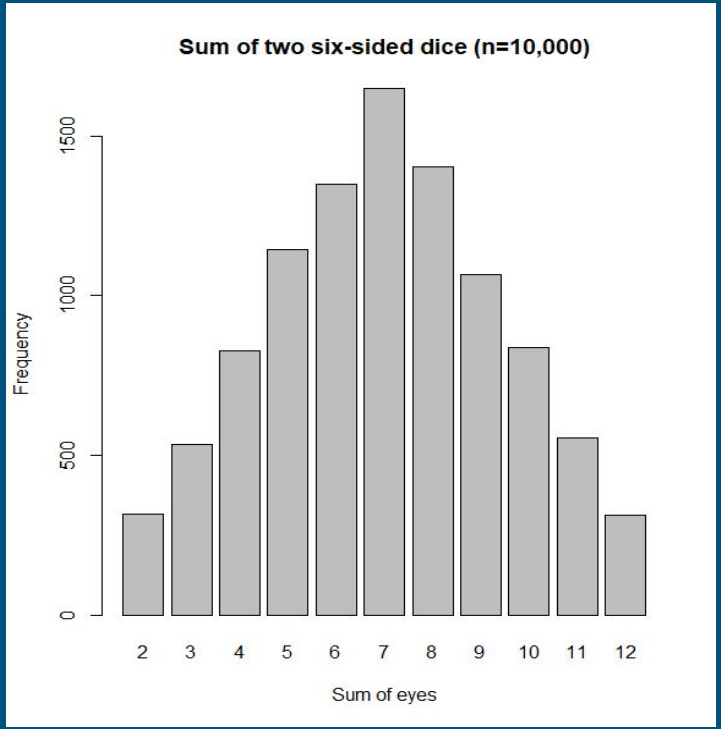


Bar plots

barplot(table(x), xlab, ylab, main, col)

Bar plots and histograms look rather similar -- both show frequencies and distributions of data -- but are used for different types of data. (The differences will be discussed on the next slide.) The barplot( ) creates a bar for each unique data category (e.g. biological sex, school grades, etc.) within a data vector and shows how often it appears in the dataset. The table( ) function calculates the frequencies of all data categories of a vector (e.g. male: 23, female: 25) and can be used as input for the barplot( ). The plot( ) parameters like “xlab”, “ylab”, “main” and “col” can also be used for bar plots. While the parameters “xlim” and “ylim” are applicable, they usually mess up the graph and should be avoided.

```
# the sum of two six-sided dice, 10,000 simulations
dice <- sample(1:6,10000, T) + sample(1:6,10000, T)

# barplot of frequencies of unique values in dataset “dice”
barplot(table(dice))
```



Bar plots

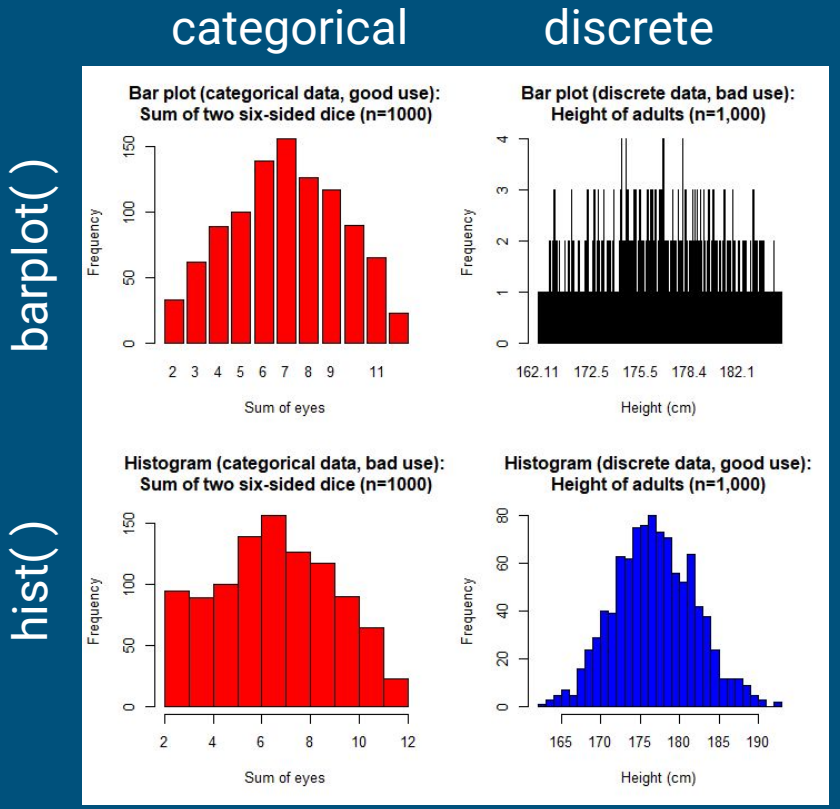
Bar plot( ) or hist( )?

Bar plots and histograms look rather similar -- both show frequencies and distributions of data -- but are used for different types of data.

Bar plots are better used when the data is categorical, i.e. there is a finite set of possible values (e.g. biological sexes, defined income groups, school grades). In the left column, barplot( ) is better as the categories are clearly distinguishable. The histogram, for examples, puts values for "2" and "3" in one bin, as there are only 10 bins for 11 categories.

Histograms are better used for discrete or continuous data that might have an infinite amount of different possible values (e.g. measurements like height in cm). In the right column, the barplot( ) is a bad choice as each value (160.01, 160.02, …) is given its own bar. While the histogram shows less detail for each unique value, the overall frequency is clearer.

(Code on next slide.)



Bar plots

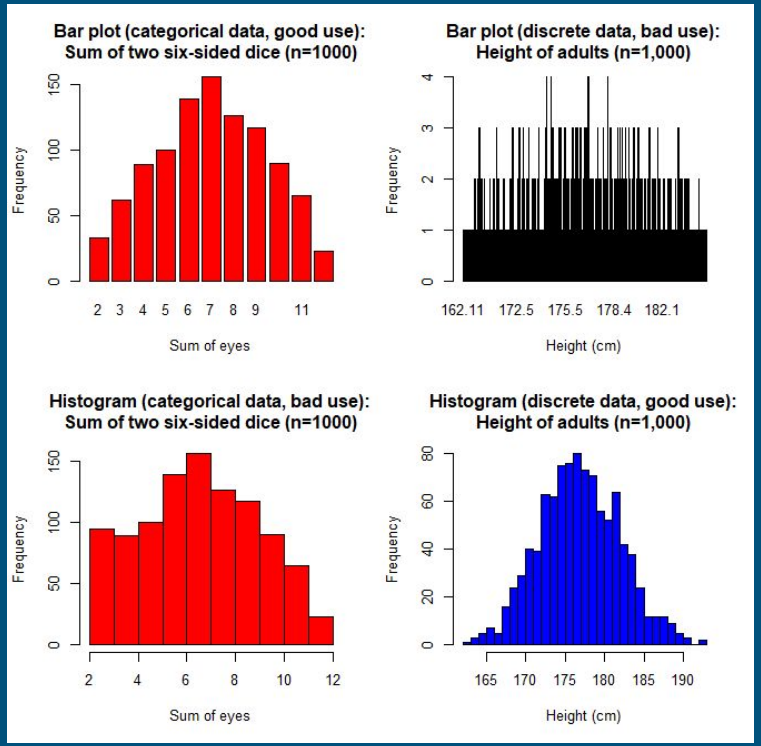
Bar plot( ) or hist( )?

```
# create random data
dice <- sample(1:6,1000, T) + sample(1:6,1000, T)
height <- round(rnorm(1000, 177, 5),2)

# top left:
barplot(table(dice),  xlab="Sum of eyes", col="red")

#top right:
barplot(table(height),xlab="Height (cm)", col="blue")

# bottom left:
hist(dice, xlab="Sum of eyes", col="red")

#bottom right:
hist(height, breaks=40, col="blue", xlab="Height (cm)")
```

categorical discrete

barplot( )

hist( )



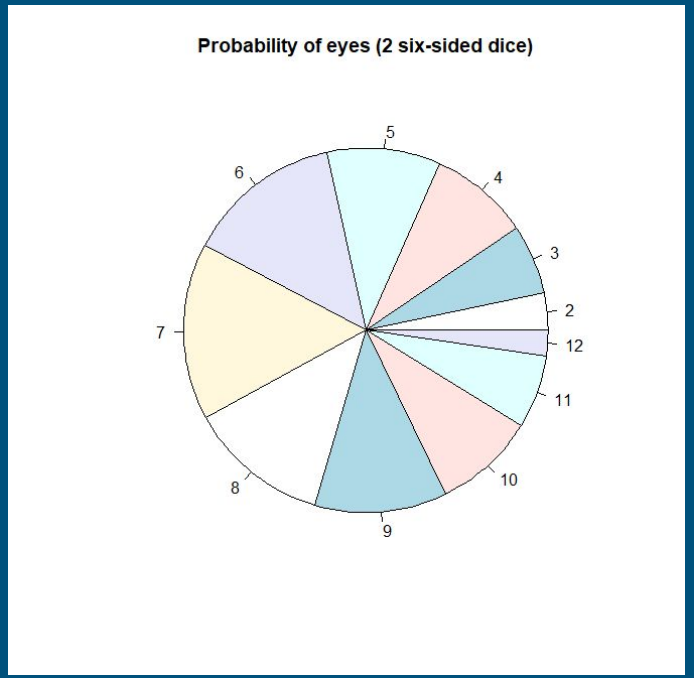
Pie charts

pie(table(x), label, main)

Pie charts show the frequency of unique elements in a dataset (like bar plots) as shares of a circle area.

They also use the table( ) of a data vector as primary input. The parameter “label” does not refer to axes but to the annotations put next to each slice of pie. We can use the names( ) of the data table( ), as shown below, or manually specify a vector of names (e.g. c(“one eye”, “two eyes”, “three eyes”)).

```
# the sum of two six-sided dice, 10,000 simulations
dice <- sample(1:6,10000, T) + sample(1:6,10000, T)

# barplot of frequencies of unique values in dataset “dice”
pie(x=table(dice), label=names(table(dice)),
   main="Probability of eyes (2 six-sided dice)")
```



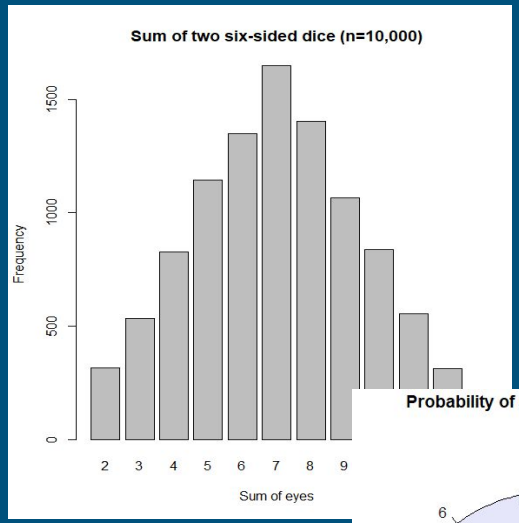
Pie charts

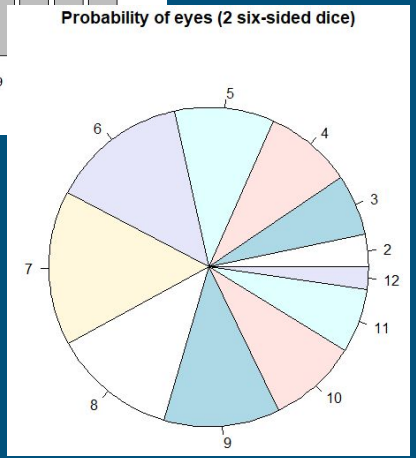
“The Golden Rule of Pie Charts”

The “Golden Rule of Pie Charts” states: “Do not use pie charts.”

While immensely popular (especially since the advent of Microsoft Excel), pie charts are often times not the most effective way to present information. The bar plot and the pie chart on the side display the same data. The bar plot enables a direct comparison between categories, e.g. “8” is more likely than “6”. The slices of a pie chart make a direct comparison more difficult -- try to order the slices by size!

In most cases plots with horizontally aligned bars or all categories stacked onto one bar are a better choice. Pie charts are only viable if only two or three categories are shown at once (e.g. YES/NO). Please note that some people have impaired vision and may struggle with a graph that relies on colors to distinguish categories, which is usually the case for pie charts.



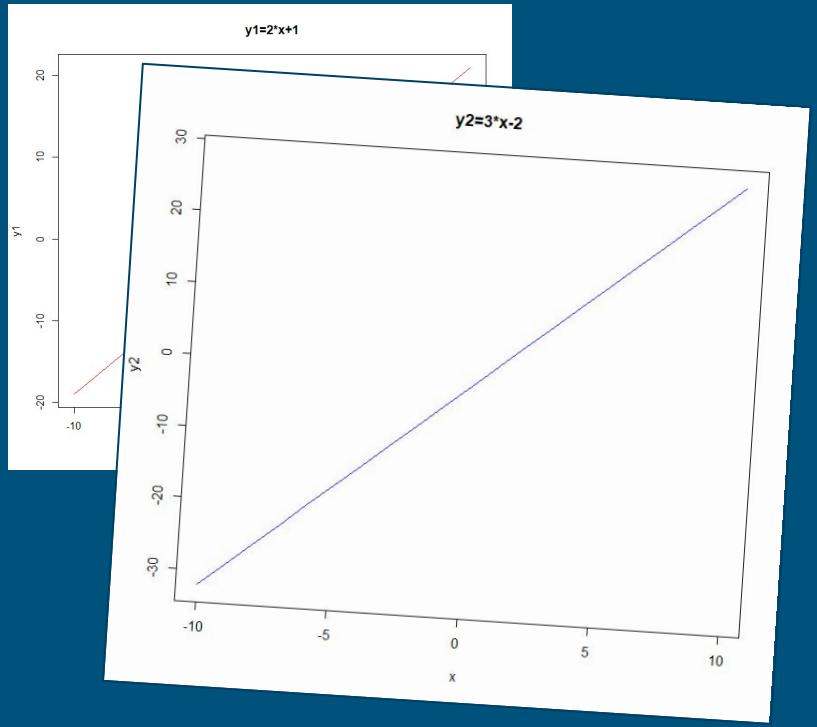


Combining plots

Multiple graphs in one plot

In case we want to include multiple graphs in the same plot, we cannot simply call two plot( ) functions after each other as the second one will create an independent instance of a plot.

```
x <- c(-10:10)      #x: all integers from -10 to 10
y1 <- 2*x+1         #function 1: x=2x+1
y2 <- 3*x -2        #function 2: x=3x-2

# plot first function:
plot(x,y1, type="l", col="red", main="y1=2*x+1")

# plot second function … in a new plot
plot(x,y2, type="l", col="blue", main="y2=3*x-2")
```



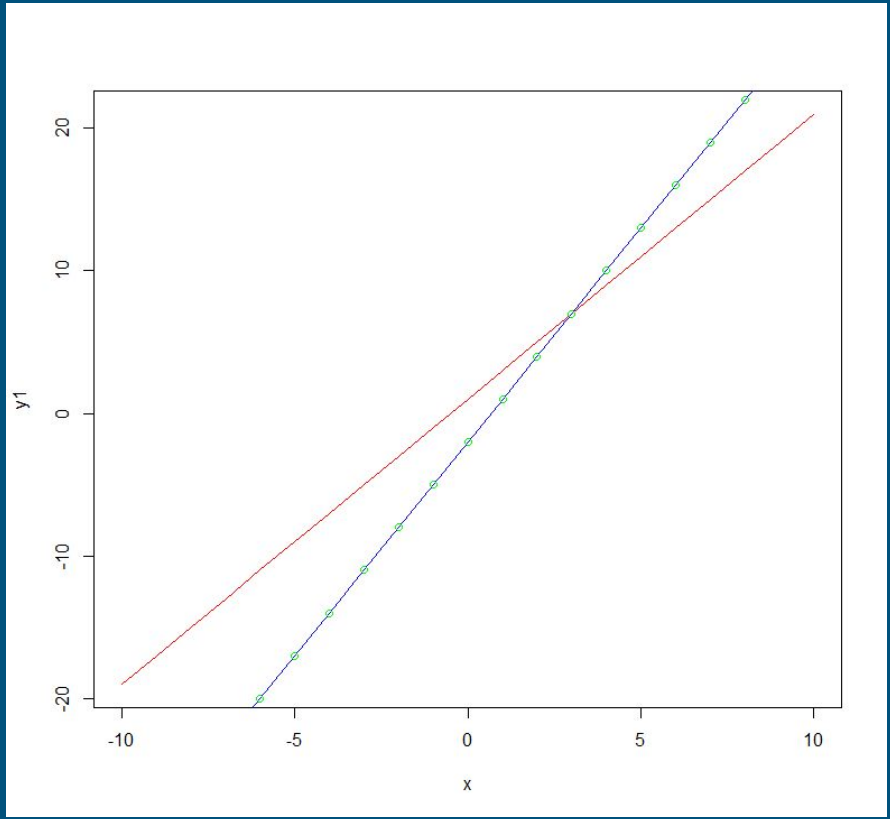
Combining plots

Multiple graphs in one plot

In this example we added two more graphs by using lines( ) and points( ). When these functions are called, they will add to the last call of plot( ). They cannot work on their own.

The additional graphs are not fully shown as the axes intervals only depend on plot( ). We fix this in the next step!

```
# plot first function:
plot(x,y1, type="l", col="red")

# add second function with blue lines (type="l"):
lines(x,y2, col="blue")

# add second function with green points (type="p"):
points(x,y2, col="green")
```

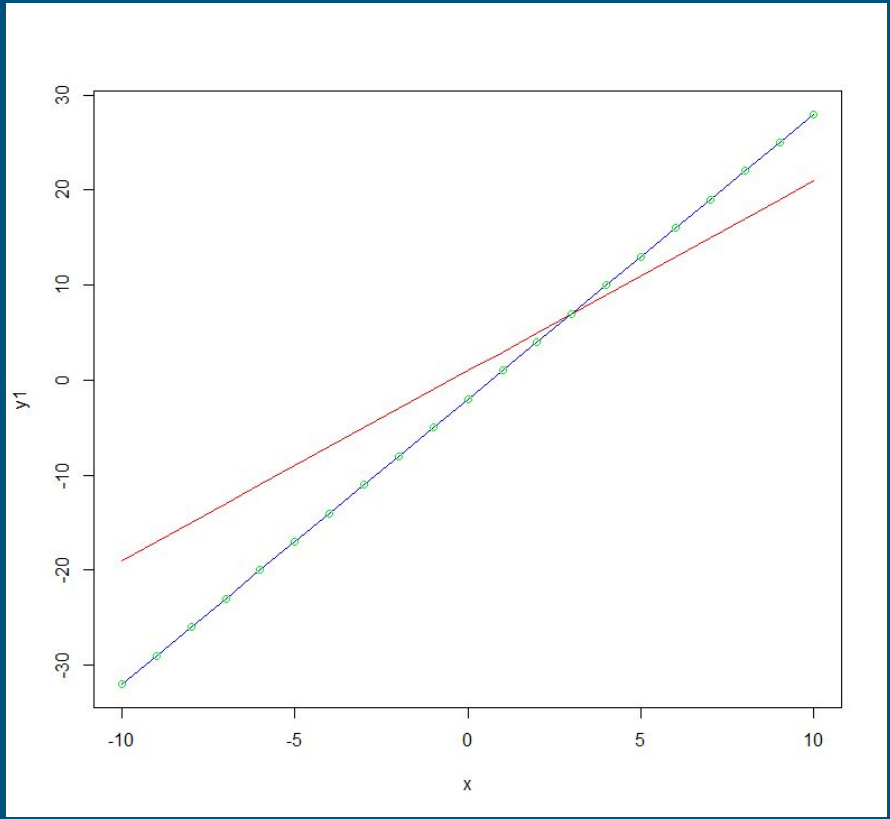


Combining plots

Multiple graphs in one plot

The range(x) function returns a vector showing the smallest and biggest number that a vector x contains. In this case the two functions have different intervals on the y-axis due to the different slopes. We can use range( ) to adjust xlim and ylim dynamically. Now all data points are shown in the plot!

```
range(y1)        #-19 21      range of y1
range(y2)        #-32 28      range of y2
range(c(y1,y2))  #-32 28      range of y1 and y2 combined

# adjust the ylim using range
plot(x,y1, type="l", col="red", ylim=range(c(y1,y2)))
lines(x,y2, col="blue")
points(x,y2, col="green")
```



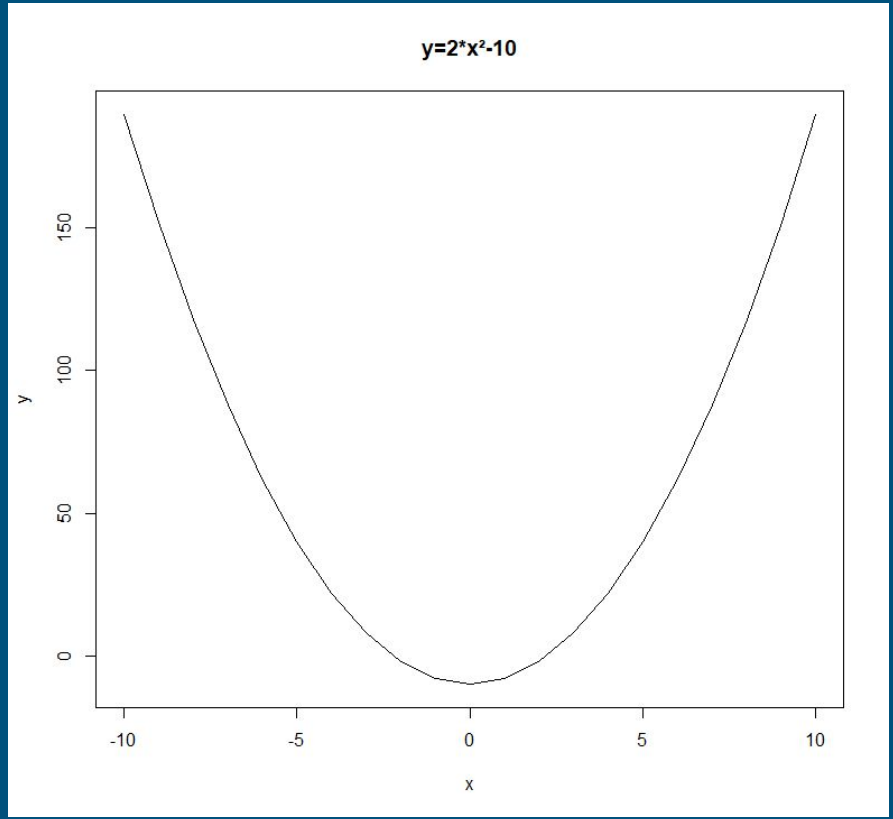
Combining plots

Adding axes to plots (abline( ))

Does this function's ($y=2x^2-10$) y value go below 0? Certainly, for the y value for x=0 must be -10. But it is difficult to see in the plot. Sometimes it can be handy to add horizontal or vertical lines to a plot to show where the axes would be.

The function abline( ) can be used to add linear functions ($y=a+bx$) to a plot without calculating the numbers or adjusting intervals. The parameter "a" indicates the intercept (the y value we get when x=0), while "b" indicates the slope (b=2, for each unit in the positive x direction, we move 2 units in the positive y direction).

```
abline(a=0, b=1)        # y=0+1x
abline(a=1, b=2)        # y=1+2x
```

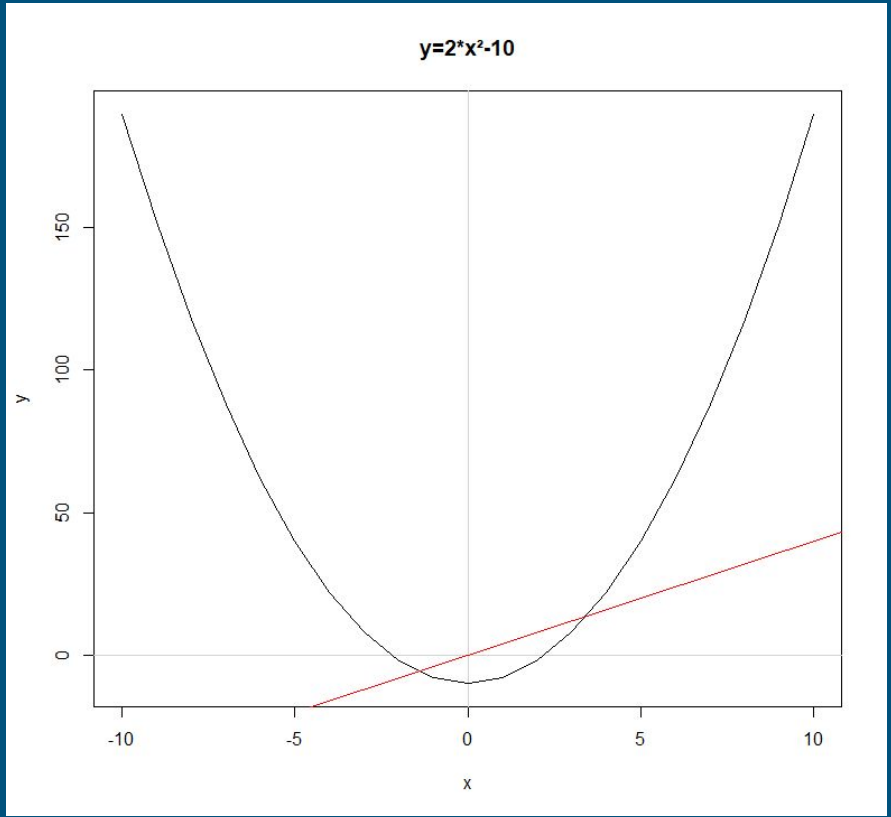


Combining plots

Adding axes to plots (abline( ))

The following plot is enhanced using several abline( ) functions.

```
x <- c(-10:10)      # x: all integers from -10 to 10
y <- 2*x^2 - 10     # y: function y=2x²-10

# create an empty plot (type="n")
# thus, the axes are drawn first and not on top of the graph
plot(x,y, pch=16, type="n", main='y=2*x²-10')

# add axes as ablines:
abline(h=0, col="lightgray")  # [h]orizontal line: y=0 (x axis)
abline(v=0, col="lightgray")  # [v]ertical line: x=0 (y axis)
lines(x,y) # y=2x²-10, drawn after the axes to be on top
abline(a=0, b=4, col="red") # first derivative of y: y'=0+4x
```

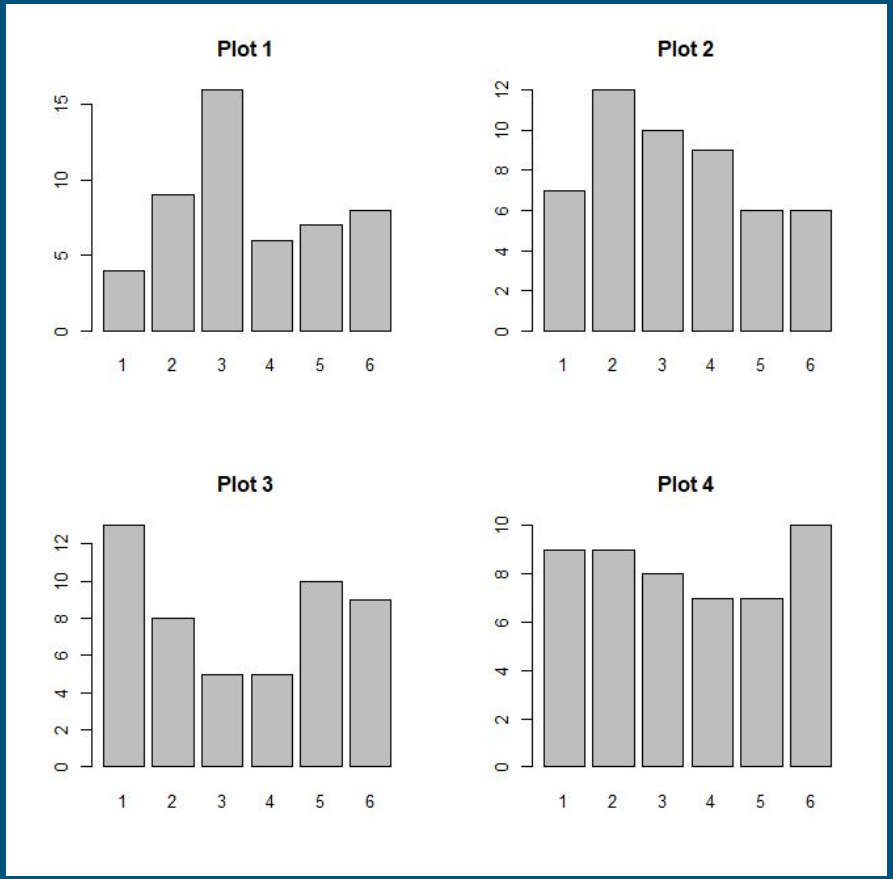


Combining plots

Multiple plots in a shared frame

Not all graphs can (or should) be combined into a single plot. The command “par(mfrow=c(m, n))” creates a raster with m rows and n columns in which the next m times n plots are inserted. First we specify the number of [R]ows m and then the number of [C]olumns n (mnemonic aid: [R]e[C]tangle). It is recommended to reset the frame back to 1 by 1 using “par(mfrow=c(1,1))” after all desired plots are printed so the next set of plots will be displayed individually again.

```
par(mfrow=c(2,2)) # open frame: 2 rows, 2 columns
barplot(table(sample(1:6,50, T)), main="Plot 1")
barplot(table(sample(1:6,50, T)), main="Plot 2")
barplot(table(sample(1:6,50, T)), main="Plot 3")
barplot(table(sample(1:6,50, T)), main="Plot 4")
par(mfrow=c(1,1)) # reset frame back to individual display
```



Exporting plots to files

Manual export

R and RStudio offer several ways to export plots to files.

After a plot is created, the "Export" button shown on the side can be used to create either an image (PNG, JPEG, TIFF, SVG) or a PDF file. While JPEGs have smaller file sizes due to compression, their quality is also limited. PNGs are usually preferable (e.g. on websites or in presentations). TIFF and SVG offer higher quality when the plot is intended to be scaled to a bigger image, e.g. for posters.

The option "Copy to Clipboard..." lets the user paste the plot into an existing document. WMFs (Windows Metafiles) can be scaled freely in Microsoft Word or PowerPoint. Bitmaps are uncompressed raster graphics.

Exporting plots to files

Scripted export

File export can be part of the R script. This is preferable when a large number of plots is created, e.g. as part of a for-loop. Four steps are needed:

1. Set the path where files should be saved
2. Set image properties
3. Create plot
4. Export file

It can look like this:

```
setwd("C:/...")                                          # step 1
jpeg("rplot4.jpg", width=1080, height=480, pointsize=25)  # step 2
barplot(table(sample(1:6,50, T)), main="Rolling 50 dice") # step 3
dev.off()                                                # step 4
```

The first step is explained in chapter 1.2. This can be omitted when the filename in the second step contains a complete and valid path.

For the second step, any of these functions: jpeg( ), png( ), tiff( ), bmp( ) can be used to create a respective file. By typing “?jpeg” (etc.) into the console, their parameters are explained within the help window. Common parameters include the size (height and width) as well as the font size (pointsize). Please note that the file name’s suffix (“.jpeg”) must coincide with the function called (jpeg( )).

For the third step many examples are already given within this presentation. Even multiple plots framed together can be exported into a single image file.

The fourth step creates the file in the specified working directory.

Conclusion

What have we learned?

In this lesson we looked at standard ("Vanilla R") functions that create different types of plots and visualizations. hist( ) and bar plot( ) are great to explore distributions within a dataset, while plot( ) creates two-dimensional graphs that visualize tendencies and dependencies between two data variables. Instead of pie charts, we can usually find a better visualization method. Plots can be combined within a single coordinate system or as independent plots within a frame. Many more functions exist to create specific visualizations.

Standard visualizations are best used for data exploration, i.e. quickly creating many different plots to get a better understanding of the dataset. Their appearance does not have to be perfect for this purpose. Data exploration is often "quick and dirty".

For the purpose of data presentation, we want to explore the package "ggplot2" in the next section. This package creates plots that are usually more aesthetically pleasing. It also provides more options to control a plot's appearance, e.g. through background rasters or color scaling.