R graphics

Alberto Garfagnini

Università di Padova

R lecture 5

Saving in R (1)

Saving R objects

- sometimes we need to save object created in R
- to save the current R session, so that it can be loaded at a later stage to continue working on it:

```
save(list = ls(all=TRUE), file = "my-session")
```

- a binary file will be produced and saved on disk
- everything can be loaded, at a later stage, with the following command:

```
load(file= "my-session")
```

Saving R history

- sometimes we need to save only the lines of code that have been typed in an R session

```
savehistory(file = "my-history.R")
```

- a text file with all the command is saved on disk
- to retrieve history, type:

```
loadhistory(file = "my-history.R")
```

Saving in R (2)

Saving graphics

- graphics can be saved in either pdf or postscript to include them in a report
- the procedure is to open a new pdf or postscript device, with the `pdf()` or `postscript()` functions
- then all commands needed to create the graphics can be typed in the R session, and once finished, the device has to be closed with the `dev.off()` function. Example:

```
pdf("my-plot.pdf")
hist(rnorm(10000))
dev.off()
```

Saving data produced within R

- let's suppose we have produced a vector we want to save on disk

  `nbnumbers <- rnbinom(1000, size=1, mu=1.2)`
- and we want to save them in a file, in a single column

  `write(nbnumbers,"nbnumbers.txt",1)`
- if, instead, we want to save them in a matrix like format

```
xmat <- matrix(rpois(100000,0.75),nrow=1000)
write.table(xmat,"table.txt",col.names=F,row.names=F)
```

- we have saved 1000 rows each of 100 Poisson random numbers with $\lambda = 0.75$

R graphics systems

`base` graphics

- a pen on paper model : you can only draw on top of a plot, no modification or deletion of existing content possible
- no user accessible representation of a graphics, only appearance on the screen
- fast primitives, but with limited scope

`grid` graphics

- developed by Paul Murrell (started in his PhD work)
- graphical objects can be represented independently of the plot and modified later
- a system of viewports makes it easier to lay out complex graphics

`lattice` graphics

- developed by Deepayan Sarkar
- use grid graphics to implement the trellis graphics system of Cleveland

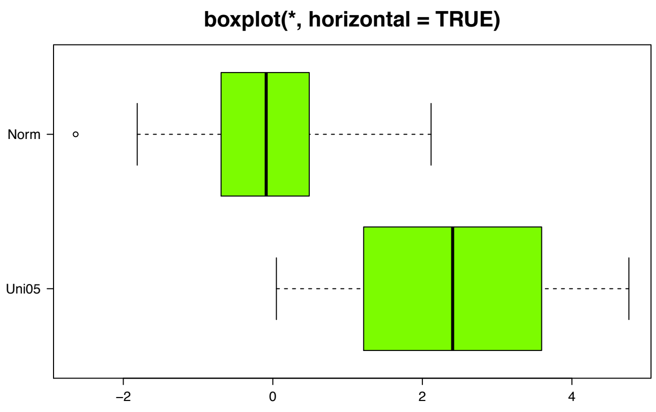
`ggplot2`

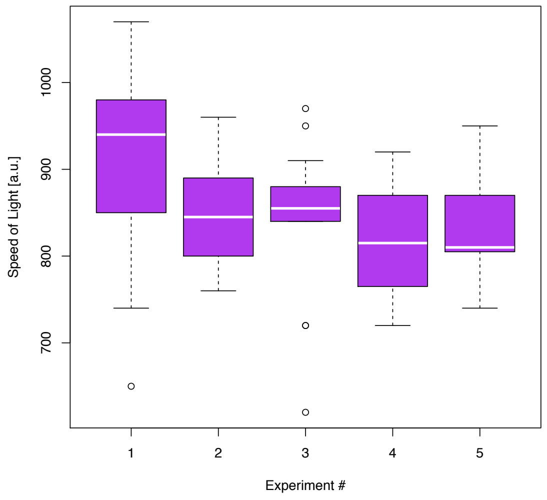
- a data visualization package creatd by Hadley Wickham in 2005
- it implements L. Wilkinson's Grammar of Graphics: a general scheme for data visualization which breaks up graphs into semantic components such as scales and layers

The R `boxplot()` function

- is a one-dimensional plot, known also as the box-and-whisker plot
- may be displayed vertically or horizontally
- the boxplot is always based on three quantities: top and bottom of the box are determined by the upper and lower quantiles; the band inside the box is the median
- the whiskers are created according to the purpose of the analyses and defined by the experimenter

```
mat <- cbind(Uni05 = (1:100)/21,
             Norm = rnorm(100))
df1 <- as.data.frame(mat)
par(las = 1)
boxplot(df1, horizontal = TRUE)

boxplot(Speed ~ Expt,
        data = morley,
        xlab = "Experiment #",
        ylab = "Speed of Light",
         col = "darkorchid2",
      medcol = "white")
```

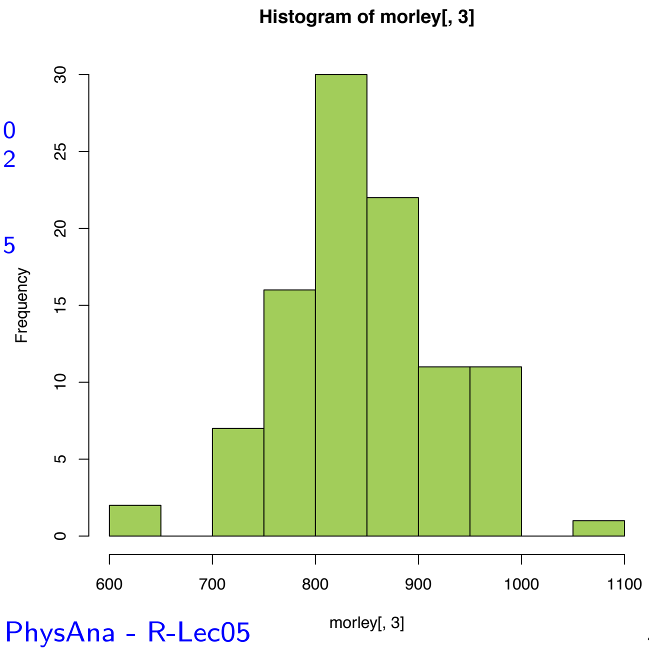




The R `hist()` function

- an `histogram` object has a complex structure
- its data can be accessed using the `$ + name` syntax; as example, `hm$breaks` is a vector with the bin limits

```
hm <- hist(morley[,3], col="darkolivegreen3",
           xlab="Speed of light [a.u.]", main="Michelson Morley")

str(hm)
#> $breaks
#>  [1]  600  650  700  750  800  850  900
#>  [8]  950 1000 1050 1100

#> $counts
#>  [1]  2  0  7 16 30 22 11 11  0  1

#> $density
#>  [1] 0.0004 0.0000 0.0014 0.0032 0.0060
#>  [6] 0.0044 0.0022 0.0022 0.0000 0.0002

#> $mids
#>  [1]  625  675  725  775  825  875  925
#>  [8]  975 1025 1075

#> $xname
#> [1] "morley[, 3]"

#> $equidist
#> [1] TRUE

#> attr(,"class")
#> [1] "histogram"
```



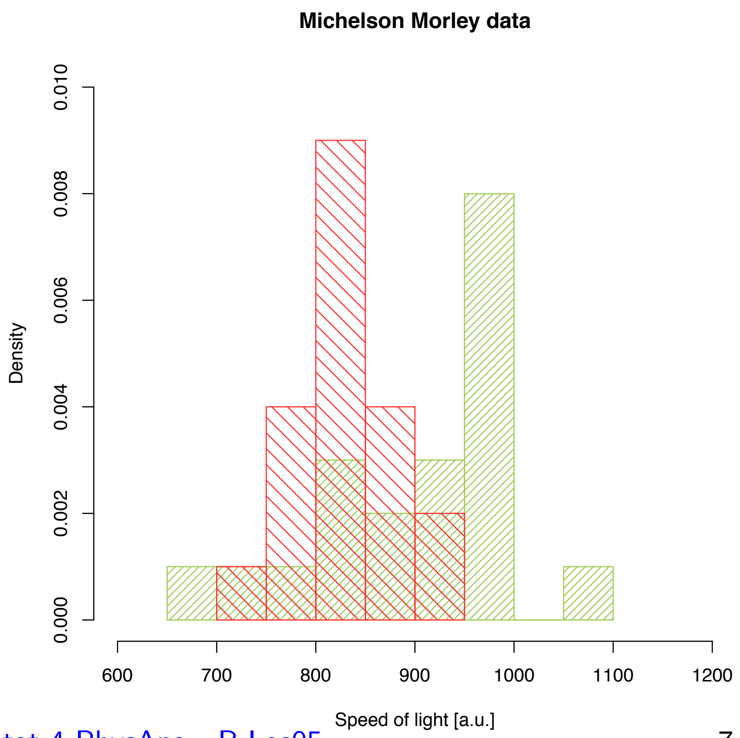
hist() function main parameters

- `hist(v, main, xlab, xlim, ylim, breaks, col, border)`
  - `v` a vector with numeric values used in histogram
  - `main` indicates title of the chart
  - `col` is used to set color of the bars
  - `border` is used to set border color of each bar
  - `xlab` is used to give description of x-axis
  - `xlim` specifies the range of values on the x-axis
  - `ylim` stands for the range of values on the y-axis
  - `breaks` is used to mention the width of each bar
  - `frame = FALSE` removes the box around the plot

Superimposing histograms

- the `add = TRUE` histogram parameters will do the job
- the option `freq = TRUE` will ensure that, in case of non equal number of observations, the heights of an interval remain the same

```r
exp_1 <- morley[[3]][morley$Expt==1]
exp_5 <- morley[[3]][morley$Expt==5]

hist(exp_1, col="darkolivegreen3",
            density=20, freq=TRUE,
            xlim=c(600,1200),
            ylim=c(0,0.01),
     xlab="Speed of light [a.u.]",
     main="Michelson Morley data")

hist(exp_5, col="firebrick1",
            density=10, freq=TRUE,
            angle=-45,
            add=TRUE)

legend(x=1100, y=6.5,
       c("exp_1", "exp_5"),
       col=c("darkolivegreen3",
             "firebrick1"),
       pch="-", cex=1.25)
```

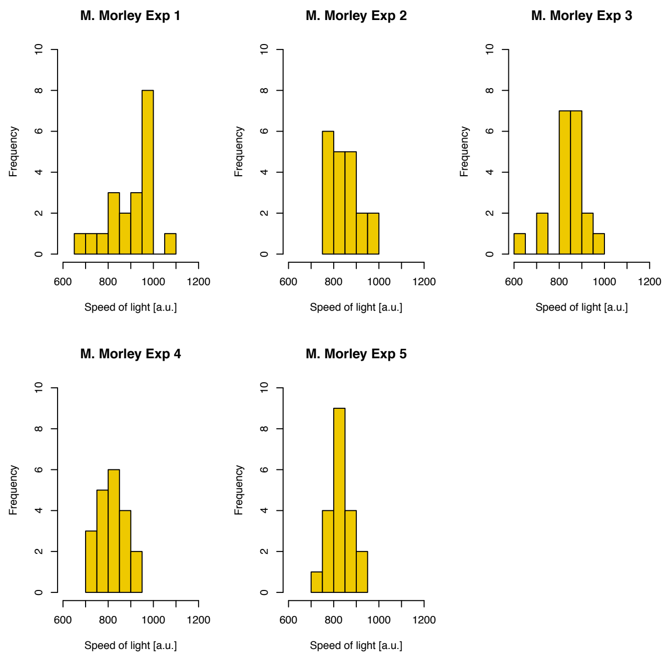


More plots on one page

- through par() it is possible to query or specify graphical parameters
- we divide the plot area in 2-row, 3-columns
- but since we have only 5 histograms, this leaves an empty plot area

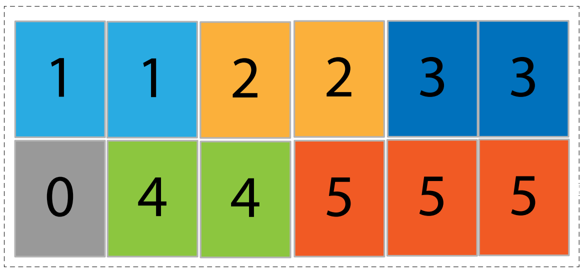
  CheatSheet: https://raw.githubusercontent.com/rstudio/cheatsheets/master/how-big-is-your-graph.pdf

```
# Save the old par versions

old_par <- par()

# Divide the graphical area in
#
2-rows, 3 columns

par(mfrow=c(2,3))

for (n_exp in 1:5) {
    h_text <- paste("M. Morley exp",
                    n_exp, sep=" ")

    hist(morley[morley$Exp==n_exp,3],
         col="gold2",
         xlim=c(600,1200),
         ylim=c(0,10),
         xlab="Speed of light [a.u.]",
         main=h_text)
}
```

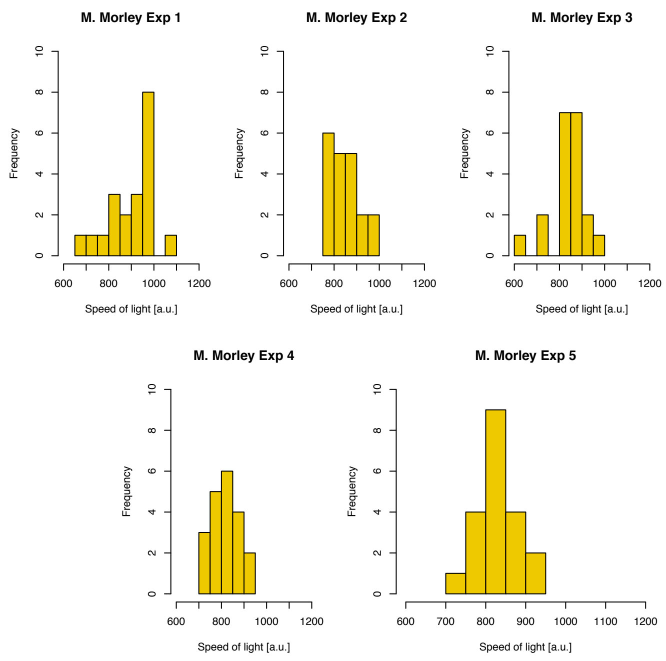


More plots on the page with layout()

- the layout(mat) function divides the device up into as many rows and columns as there are in matrix 'mat'
- a value of 0 says that such parts should not be used for plots



```
# Divide the area: 2 rows 6 columns

p_area <- matrix(c(1,1,2,2,3,3,
                   0,4,4,5,5,5),
                   nrow=2, ncol=6,
                   byrow=TRUE)

p_area
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    2    2    3    3
[2,]    0    4    4    5    5    5

layout(p_area)

for (n_exp in 1:5) {
   h_text <- paste("M. Morley exp",
                   n_exp, sep=" ")
   hist(morley[morley$Exp==n_exp,3],
        col="gold2",
        xlim=c(600,1200),
        ylim=c(0,10),
        xlab="Speed of light [a.u.]",
        main=h_text)
}
```



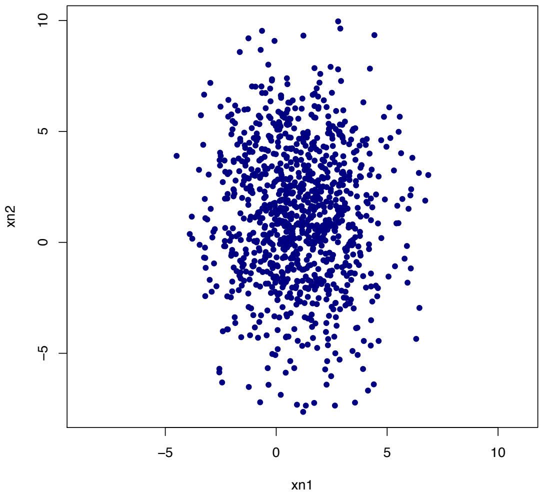
The R scatter plot() function

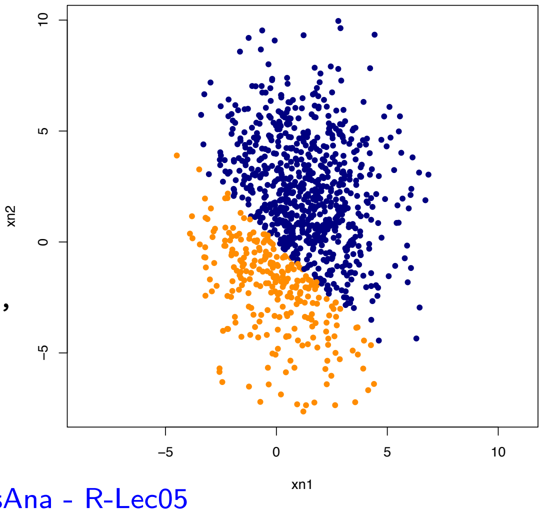
- the plot() function allows to produce a scatter plot of one variable versus the other
- the asp = value parameter allows to keep the y/x aspect ratio to a fixed value
- asp=1 sets the same scale for both x and y axis, even if the plot window is re-scaled

```
set.seed(34761542)
xn1 <- rnorm(1000, 1, 2)
xn2 <- rnorm(1000, 1, 3)
plot( xn1, xn2, pch = 20,
      col = "navy", cex=1.25,
      asp = 1 )
```

- it is possible to use a third variable for a color

```
xcontrol <- jitter(xn1+xn2, 2)
plot( xn1, xn2, pch = 20, cex=1.25,
      col = ifelse(xcontrol>0,
                   "navy", "darkorange"),
     asp = 1 )
```

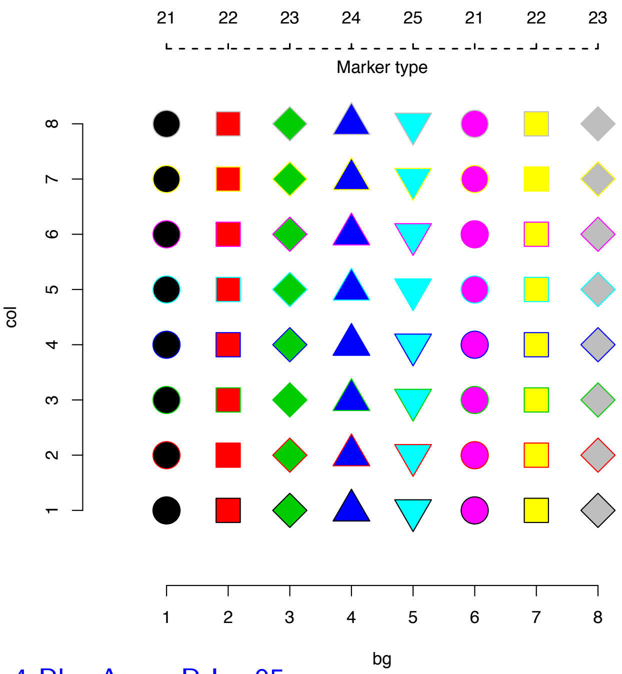




Colors for symbols and axes

- with the parameter pch it is possible to specify the plotting symbols
- moreover, it is possible to set the background bg, and fill Colors col, separately
- the option freq = TRUE will ensure that, in case of non equal number of observations, the heights of an interval remain the same

```
plot(0:9, 0:9,
     type="n", axes=FALSE,
     ylab="col", xlab="bg")
for (i in 1:8) {
     points(1:8, rep(i,8),
            pch=c(21,22,23,24,25),
            bg=1:8, col=i, cex=3.5)
}
axis(1, at=1:8)
axis(2, at=1:8)

axis(3, at=1:8,
    c(21,22,23,24,25,21,22,23),
    lty=2, lwd=1.5)
text(4.5, 9, "Marker␣type")
```



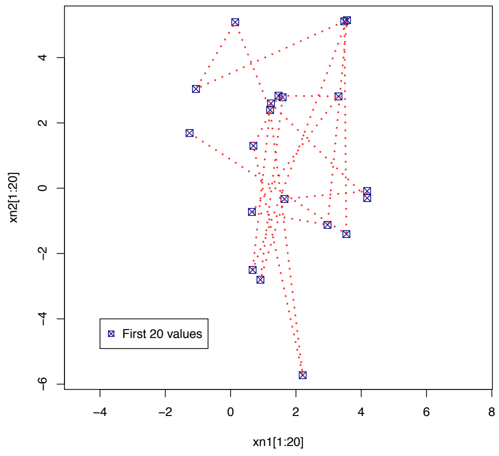
Joining points with lines()

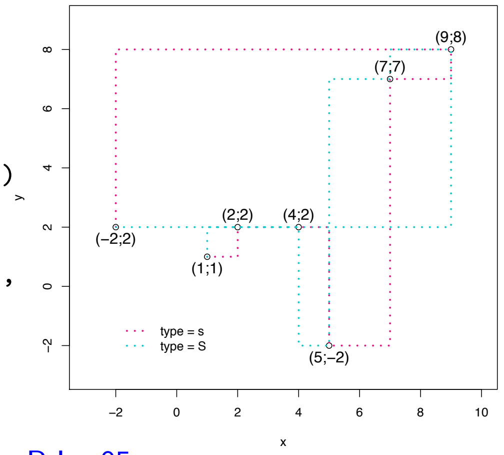
- the primitive lines() allows to connect points with lines
- the line go across the points

```
plot( xn1[1:20], xn2[1:20],
      pch = 7, cex=1.25, col = "navy")

lines(xn1[1:20], xn2[1:20],
      col = "firebrick1", lty=3)
```

- with option type='s', a stepped line going across first and then up (or down)
- with option type='S', a stepped line goes first up (or down) and then across

```
lines(x, y, col="deeppink2", type="s")
lines(x, y, col="darkturquoise", type="S")
tpos <- c(1,3,3,1,3,3,1)
text(x, y,
     labels=paste("(",x,";",y,")",sep=""),
     pos=tpos, offset=0.5, cex=1.25)
```

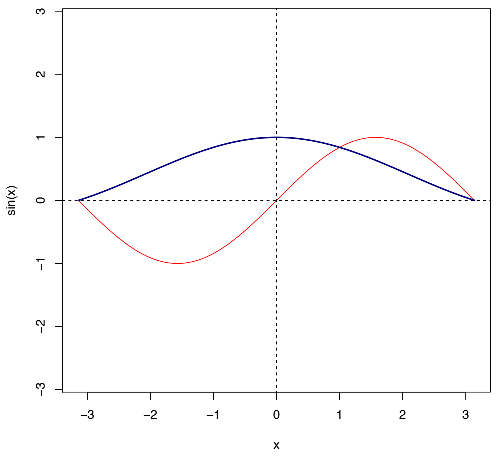


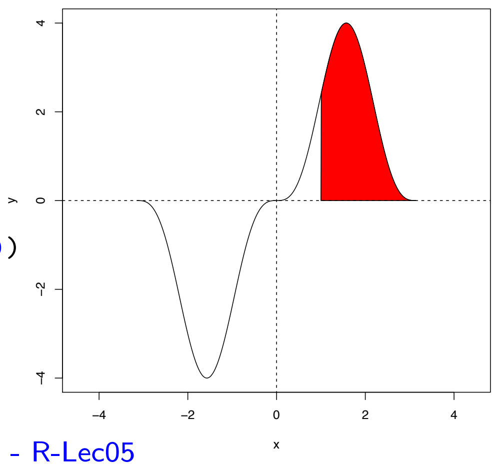


Plotting curves

- curve() allows to plot an analytical function
- abline() draws a line, given the intercept and slope

```
curve(sin(x), -pi, pi, col="red", asp=1)

# The x-axis, going through (0,0)
abline(0,0,lty=2)
# The y-axis, going through (0,0)
abline(0,10000,lty=2)

curve(sin(x)/x, -pi, pi, col="navy",
      lw=2, add=T) # add to plot


# Similar way to plot a curve
x <- seq(-pi, pi, 0.01)
y <- 4*sin(x)^3
plot(x,y, type="l", asp=1)

polygon( c(1,x[x>=1]),
         c(0,y[x>1]), col="red", angle=10)
abline(0,0, lty=2)
abline(0,10000, lty=2)
```

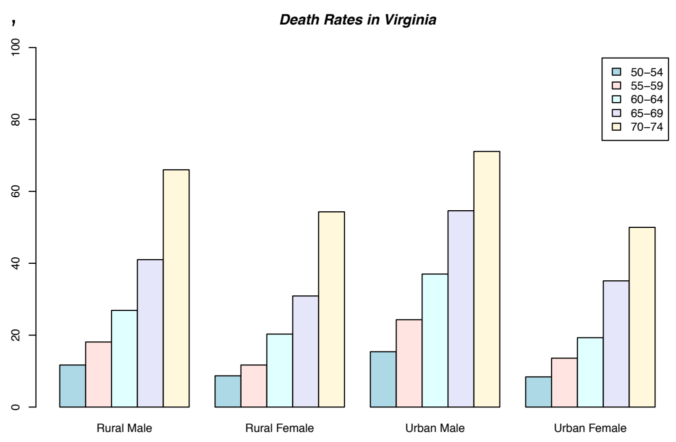




barplot

- a barplot shows the relationship between a numeric variable and a categorical variable
- R creates a barplot with vertical or horizontal bars
- be careful: a barplot is not an histogram !

```r
barplot(VADeaths, beside = TRUE,
        col = c("lightblue",
                "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths),
          ylim = c(0, 100))
title(main = "Death␣Rates␣in␣Virginia")

VADeaths
#>         Rural  Rural   Urban  Urban
#>          Male Female    Male Female
#> 50-54    11.7    8.7    15.4    8.4
#> 55-59    18.1   11.7    24.3   13.6
#> 60-64    26.9   20.3    37.0   19.3
#> 65-69    41.0   30.9    54.6   35.1
#> 70-74    66.0   54.3    71.1   50.0
```



The Grammar of Graphics

- created by Wilkinson in 2005 to describe the features living behind all statistical graphics
- it has the following components:
  - layer: are used to create the objects on a plot. They are defined by five basic parts: data (the source of the information to be visualized), mapping (how variables are applied to the plot), statistical transformation (which transform the data by summarizing the information), geometric object (controls the type of the plot to be created) and position adjustment
  - scale : controls how data is mapped to aesthetic attributes (ex: scale for colors)
  - coordinate system : maps the position of objects onto the plane of the plot and controls how axes and grid lines are drawn
  - faceting : can be used to split data into subsets of the entire dataset

Wilkinson L., *The grammar of graphics. Statistics and computing* 2005, Springer, New York

ggplot2 example

- every plot has three components :

1▷ data to be plotted

2▷ aesthetics, a set of mappings between variables in the data and visual properties

3▷ geoms, a layer that describes how to render each observation

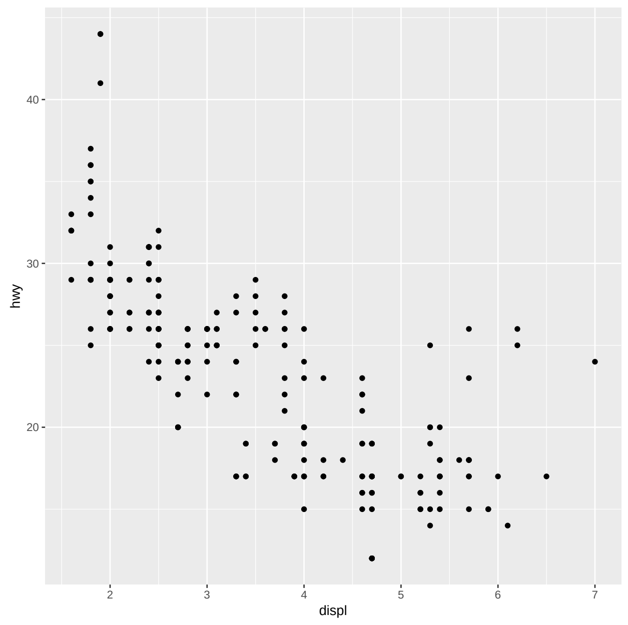
```
ggplot(mpg, aes(x=displ, y=hwy)) +
            geom_point()
```

- the produced scatter plot is defined by:

1▷ data = mpg dataframe

2▷ aesthetics = the x position is the engine mapping, while y is the fuel economy

3▷ geom = points

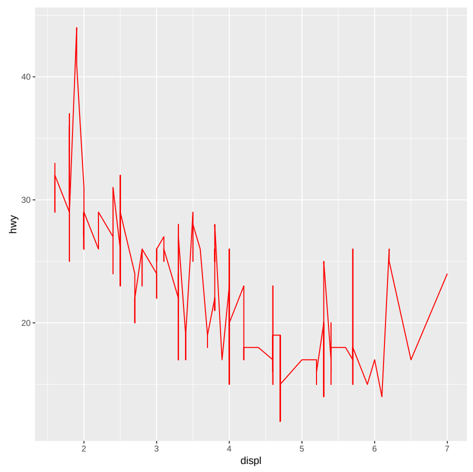


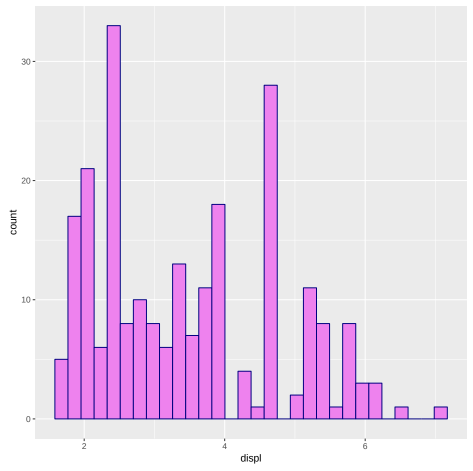
Important : data and aesthetic mappings are supplied in ggplot(), additional layers are added on with the '+' operator

ggplot2 example

- once a plot is created, it is possible to draw it using different rendering

```
p1 <- ggplot(mpg, aes(displ, hwy))

# the plot of the previous example
p1 + geom_point()


# new : points connected with lines
p1 + geom_line()


p2 <- ggplot(mpg, aes(displ))

p2 + geom_histogram(col="navy",
                    fill="violet")

p2 + geom_histogram(col='navy',
                    fill='orchid2',
                    binwidth=0.3)
```

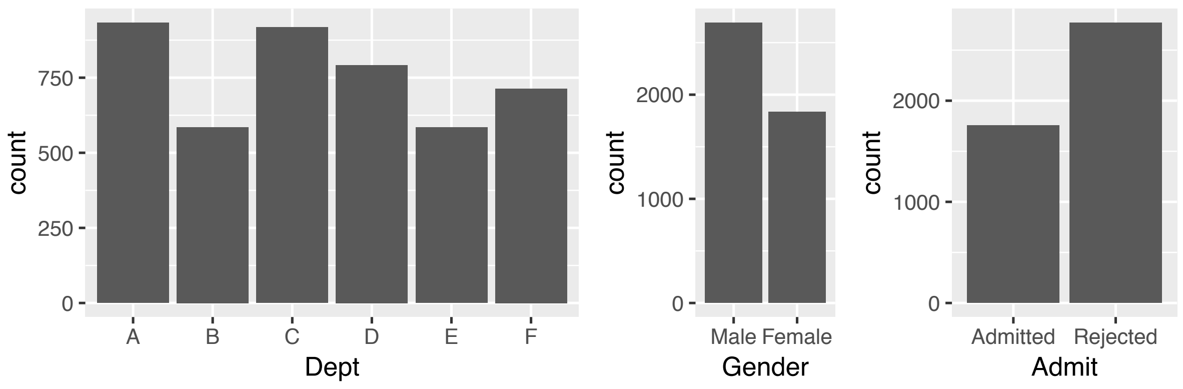




ggplot2 barchart example

- the UCBAdmissions data set contains data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex
- we create separate barcharts for the variables: department, gender, and admitted or rejected

```
library(gridExtra)

ucba <- as.data.frame(UCBAdmissions)

a <- ggplot(ucba, aes(Dept)) + geom_bar(aes(weight=Freq))
b <- ggplot(ucba, aes(Gender)) + geom_bar(aes(weight=Freq))
c <- ggplot(ucba, aes(Admit)) + geom_bar(aes(weight=Freq))

grid.arrange(a, b, c, nrow=1, widths=c(7,3,3))
```



References

Books

- P. Murrell, *R graphics*, Chapman & Hall/CRC, 2006, 978-1-58488-486-6
- D. Sarkar, *Lattice: Multivariate Data Visualization with R*, Use R! series, Springer, 2008, 978-0-387-75968-5
  http://lmdvr.r-forge.r-project.org/figures/figures.html
- H. Wickham, *ggplot2, Elegant Graphics for Data Analysis*, Use R! series, Springer, 2016, 978-3-319-24275-0
  https://ggplot2-book.org/

Tutorials

- P. Murrell, *Introduction to R graphics*,
  https://www.stat.auckland.ac.nz/~paul/RGraphics/chapter1.pdf
  https://www.stat.auckland.ac.nz/~paul/RGraphics/RGraphicsChapters-1-4-5.pdf
- https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/

Dates and time in R

R date/time

- measurement of time is highly idiosyncratic. Successive years start on different days of the week. There are months with different number of days. Leap years help complicating, adding an extra day on February every fourth year
- notations is also different: European, Asiatic and Americans put the day and the month in different years: 3/4/2006 can be the 3rd of April or the 4th of March.
- the `Sys.time()` function shows how dates and times is handled in R

```
#> Sys.time()
#> [1] "2021-03-23 21:00:57 CET"
```

- The baseline for expressing today's date and time in seconds is January 1st, 1970:

```
(tnow <- Sys.time())
#> [1] "2021-03-23 21:02:31 CET"
as.numeric(tnow)
#> [1] 1616529752
```

- R uses `POSIX` system for representing dates and times

```
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
```

R date/time classes : POSIXlt and POSIXct

- POSIXlt gives a list containing separate vectors for the year, month, day of the week, day within the year, . . .
- it is very useful as a categorical explanatory variable
- POSIXct gives a vector containing the date and time expressed as a continuous variable that you can use in regression models (it is the number of seconds since the beginning of 1970).

- it is possible to convert from one representation to the other

```
tnow <- Sys.time()
time.list <- as.POSIXlt(tnow)

class(as.POSIXlt(tnow))
#> [1] "POSIXlt" "POSIXt"

unlist(time.list)
#>       sec                min                hour               mday
#> "31.6148"               "2"               "21"               "23"
#>       mon               year               wday               yday
#>       "2"             "121"                "2"               "81"
#>     isdst               zone             gmtoff
#>       "0"             "CET"             "3600"
```

Reading date/times data from files

- once dates in the format DD/MM/YYYY are read with read.data, they are read as characters, by default

```
data <- read.table("dates.txt", header=TRUE)
head(data)
#>   cnt       date
#> 1   3 12/02/2021
#> 2   4 13/02/2021
#> 3  12 14/02/2021
#> 4   8 15/02/2021

attach(data)

date
#> [1] 12/02/2021 13/02/2021 14/02/2021 15/02/2021

class(date)
#> [1] "character"
```

- data are not recognized by R as being dates
- to convert a factor, or a character string into a POSIXlt object, the strptime() function is used

```
Rdate <-strptime(as.character(date), "%d/%m/%Y")
class(Rdate)
#> [1] "POSIXlt" "POSIXt"
```

strptime() function

```
str(unclass(Rdate))
#> List of 11
#>  $ sec   : num [1:4] 0 0 0 0
#>  $ min   : int [1:4] 0 0 0 0
#>  $ hour  : int [1:4] 0 0 0 0
#>  $ mday  : int [1:4] 12 13 14 15
#>  $ mon   : int [1:4] 1 1 1 1
#>  $ year  : int [1:4] 120 120 120 120
#>  $ wday  : int [1:4] 2 3 4 5
#>  $ yday  : int [1:4] 42 43 44 45
#>  $ isdst : int [1:4] 0 0 0 0
#>  $ zone  : chr [1:4] "CET" "CET" "CET" "CET"
#>  $ gmtoff: int [1:4] NA NA NA NA
```

- let's add the R-formatted date to our data frame

```
data <- data.frame(data,Rdate)
data
#>   cnt       date      Rdate
#> 1   3 12/02/2021 2021-02-12
#> 2   4 13/02/2021 2021-02-13
#> 3  12 14/02/2021 2021-02-14
#> 4   8 15/02/2021 2021-02-15
```

Dates and times arithmetic's

The following calculations are possible:

- time ± number
- time1 - time2
- time1 `logical-op` time2
- where `logical-op` is one of `==`, `!=`, <, <=, >, >=

```
y2 <- as.POSIXlt("2021-02-18")
y1 <- as.POSIXlt("2021-01-26")

y2 - y1
#> Time difference of 23 days

y1 + y2
#> Error in `+.POSIXt`(y1, y2) :
#> binary '+' is not defined for "POSIXt" objects
```

The following calculations are possible:

- it is possible to add or subtract a number of seconds or a `difftime` object from a dat-time object, but they cannot be added
- always convert dates and times into `POSIXlt` objects before staring any calculations

The difftime() and as.difftime() functions

- evaluating the difference between two dates and times involves the `difftime()` function

```
difftime("2021-02-18","2021-01-26")
#> Time difference of 23 days
```

- if only the number of days is needed, use

```
as.numeric(difftime("2021-02-18","2021-01-26"))
#> [1] 23
```

- the same operation can be applied to times, as well

```
t1 <- as.difftime("12:43:12")
t2 <- as.difftime("7:00:00")
t1-t2
#> Time difference of 5.72 hours

as.numeric(t1-t2)
#> [1] 5.72
```

Generating sequences of dates

- it may be useful to generate sequences of dates by years, months, weeks or days

```
seq(as.POSIXlt("2019-08-01"), as.POSIXlt("2019-10-12"), "1␣week")
#> [1] "2019-08-01 CEST" "2019-08-08 CEST" "2019-08-15 CEST"
#> ...
#> [9] "2019-09-26 CEST" "2019-10-03 CEST" "2019-10-10 CEST"

seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2029-10-12"), "3␣years")
#> [1] "2019-04-01 CEST" "2022-04-01 CEST" "2025-04-01 CEST"
#> [3] "2028-04-01 CEST"
```

- a number, instead of a recognized character string, will be interpreted as a number of seconds

```
seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2019-04-12"), 1024)
#> [1] "2019-04-01 00:00:00 CEST" "2019-04-01 00:17:04 CEST"
#> ...
#> [929] "2019-04-11 23:57:52 CEST"
```

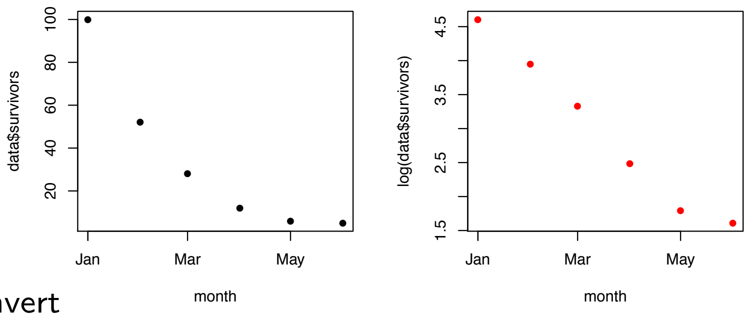
- as for other type of seq, the length of the vector can be specified instead of the final date:

```
seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2019-04-12"), length=3)
#> [1] "2019-04-01 00:00:00 CEST" "2019-04-06 12:00:00 CEST"
#> [3] "2019-04-12 00:00:00 CEST"
```

Regression using dates and times : 1

- an experiment was performed observing the number of insects over 6 months

```
data <- read.table("timereg.txt", header=T)
data
#>   survivors       date
#> 1       100 01/01/2011
#> 2        52 01/02/2011
#> 3        28 01/03/2011
#> 4        12 01/04/2011
#> 5         6 01/05/2011
#> 6         5 01/06/2011
```



- as before, we use strptime() to convert a data string into a date-time object

```
dl <-strptime(data$date, "%d/%m/%Y")

class(dl)
#> [1] "POSIXlt" "POSIXt"

mode(dl)
#> [1] "list"

# Let's plot the data
par(mfrow=c(2,2))
plot(dl, data$survivors, pch=16, xlab="month")
plot(dl, log(data$survivors), pch=16, col="red", xlab="month")
```

Regression using dates and times : 2

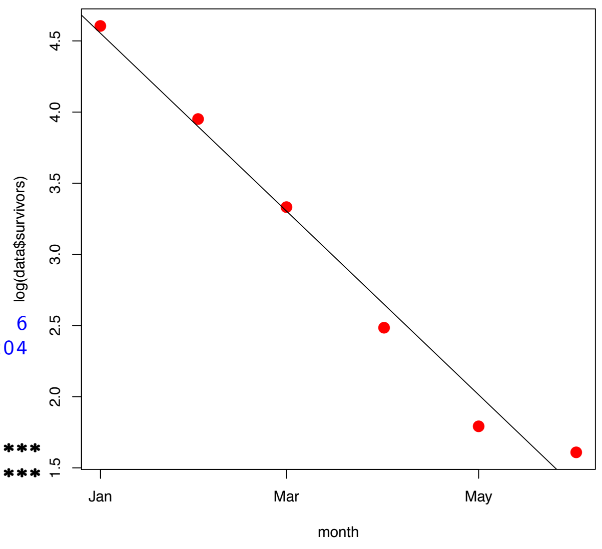
- plotting the data suggests an exponential decay in the survivor variable

```
model <- lm(log(data$survivors)~dl)
#> Error in model.frame.default(formula = log(data$survivors) ~ dl :
#>   invalid type (list) for variable 'dl'
```

- the reason for the error is that we cannot use a list as an explanatory variable in a linear model
- we need to convert from a list (class = POSIXlt) to a continuous numeric variable (class = POSIXct)

```
dc <- as.POSIXct(dl)
model <- lm(log(data$survivors)~dc)
abline(model)
summary(model)
```

-
```
Call:
lm(formula = log(data$survivors) ~ dc)

Residuals:
       1        2        3        4        5        6
 0.05178  0.05416  0.02792 -0.16395 -0.22195  0.25204

Coefficients:
              Estimate  Std. Error t value Pr(>|t|)
(Intercept)  3.216e+02   2.286e+01   14.07 0.000148 ***
dc          -2.450e-07   1.758e-08  -13.94 0.000154 ***
```



The `lubridate` package https://lubridate.tidyverse.org/

- the packages makes it easier to work with date and times
- it allows to create date from strings:

```
ymd('2021-03-22')
#> [1] "2021-03-22"
d1 <- ymd('2021-03-22')
d2 <- mdy('03-22-2021')
d3 <- dmy('22-03-2021')

d1-d2; d1-d3
#> Time difference of 0 days
#> Time difference of 0 days
```

- but also from unquoted numbers

```
> dmy(22032021)
[1] "2021-03-22"
```

- it has simple functions to get and set components of a date-time, such as `year()`, `month()`, `day()`, `hour()`, `minute()` and `second()`

```
> day(dmy(22032021))
[1] 22
```

- it expands the mathematical operations to be performed with date-time objects: 3 new time span classes (`durations`, `periods` and `intervals`, borrowed from http://joda.org