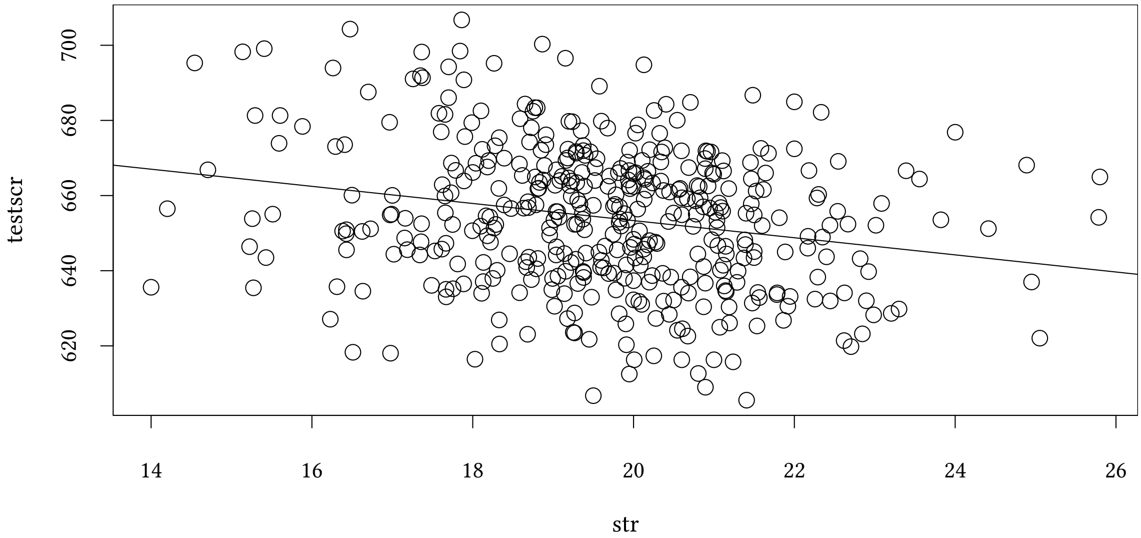
Introduction to R




```
Call:
lm(formula = testscr ~ str)

Residuals:
    Min      1Q  Median      3Q     Max 
-47.727 -14.251   0.483  12.822  48.540 

Coefficients:
            Estimate Std. Error t value   Pr(>|t|)    
(Intercept) 698.9330     9.4675  73.825    < 2e-16 ***
str          -2.2798     0.4798  -4.751 0.00000278 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 18.58 on 418 degrees of freedom
Multiple R-squared:  0.05124,Adjusted R-squared:  0.04897 
F-statistic: 22.58 on 1 and 418 DF,  p-value: 0.000002783
```


Oliver Kirchkamp

Contents

1 A Brief Introduction To R

For the purpose of the course we take R as an example for one statistical language. Even if you use other languages for your work, you will find that the concepts are similar.

2 First steps

2.1 Installing R

On the Homepage of the R project you find in the menu on the left a link Download / CRAN. This link leads to a list of "mirrors". On these mirrors you also find instructions how to install R on your OS.

- Learning R: On the Homepage of the R Projekt: Documentation (Manuals, FAQs, Contributed).
- Front-end: RStudio

Installing Libraries

If the command `library` complains about not being able to find the required library, then the library is most likely not installed. The command

```
install.packages("Ecdat")
```

installs the library `Ecdat`. Some installations have a menu "Packages" that allows you to install missing libraries. Users of operating systems of Microsoft find support at the FAQ for Packages.

2.2 Types and assignments

R knows about different *types* of data. We will meet some types in this chapter. To assign a number (or a value, or any object) to a variable, we use the operator `<-`

```
x <- 4
```

R stores the result of this assignment as `double`

```
typeof(x)

[1] "double"
```

Now we can use `x` in our calculations:

```
2 * x

[1] 8

sqrt(x)

[1] 2
```

Often our calculations will not only involve a single number (a scalar) but several which are connected as a vector. Several numbers are connected with `c`

```
x <- c(21,22,23,24,25,16,17,18,19,20)
x

 [1] 21 22 23 24 25 16 17 18 19 20
```

When we need a long list of subsequent numbers, we use the operator `:` or the function `seq`

```
21:30

 [1] 21 22 23 24 25 26 27 28 29 30

seq(21,30)

 [1] 21 22 23 24 25 26 27 28 29 30

y <- 21:30
```

The recycling rule

Combine vectors of the same length:

```
x <- c(21,22,23,24,25,16,17,18,19,20)
y <- 21:30
x + y

 [1] 42 44 46 48 50 42 44 46 48 50
```

Combine vectors of different length:

```
x * 10

 [1] 210 220 230 240 250 160 170 180 190 200
```

But also:

```
x * c(1,10)

 [1]  21 220  23 240  25 160  17 180  19 200

x * c(1,10,100)

 [1]   21  220 2300   24  250 1600   17  180 1900   20
```

Subsets

We can access single elements of a vector with `[]`

```
x[1]

[1] 21
```

When we want to access several elements at the same time, we simply use several indices (which are connected with c). We can use this to change the sequence of values (e.g. to sort).

```
x[c(3,2,1)]

[1] 23 22 21

x[3:1]

[1] 23 22 21
```

```
x

 [1] 21 22 23 24 25 16 17 18 19 20
```

(to sort a long vector we would use the function order).

```
order(x)

 [1]  6  7  8  9 10  1  2  3  4  5

x[order(x)]

 [1] 16 17 18 19 20 21 22 23 24 25
```

(order determines an "ordering", i.e. a sequence in which the elements of the vector should be used to be "ordered". We use x[...] to see the ordered result.)

Negative indices drop elements:

```
x[-1:-3]

[1] 24 25 16 17 18 19 20
```

Matrices and arrays

```
x

 [1] 21 22 23 24 25 16 17 18 19 20

matrix(x,nrow=2)

     [,1] [,2] [,3] [,4] [,5]
[1,]   21   23   25   17   19
[2,]   22   24   16   18   20
```

```
array(1:30,dim=c(3,5,2))

, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   16   19   22   25   28
[2,]   17   20   23   26   29
[3,]   18   21   24   27   30
```

Missings

```
x <- c( 1, 2, 3, 0/0, sqrt(-1), 1/0, NA)
x

[1]   1   2   3 NaN NaN Inf  NA

is.finite(x)

[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE

is.infinite(x)

[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE

is.nan(x)

[1] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE

is.na(x)

[1] FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE

NA == NA

[1] NA

NA & FALSE

[1] FALSE

NA | TRUE

[1] TRUE

1/Inf

[1] 0

Inf-2*Inf

[1] NaN
```

Logicals

Logicals can be either TRUE or FALSE. When we compare a vector with a number, then all the elements will be compared (this follows from the *recycling rule*, see below):

```
x <- c(21,22,23,24,25,16,17,18,19,20)
x == 20

 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE

x < 20

 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE

x >= 20

 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE

typeof(x < 20)

[1] "logical"
```

We can use logicals as indices, too:

```
x [ x < 20 ]

[1] 16 17 18 19
```

Characters

Not only numbers, also character strings can be assigned to a variable:

```
x <- "Mary"
typeof(x)

[1] "character"
```

We can also work with vectors of character strings:

```
x <- c("John","Mary","Jane")
x[2]

[1] "Mary"

x[3]<-"Lucy"
x

[1] "John" "Mary" "Lucy"
```

Factors

Often it is clumsy to store a string of characters again and again if this string appears in the dataset several times. We might, e.g., want to store whether an observation belongs to a man or a woman. This can be done in an efficient way by storing 2 for `"male"`, and 1 for `"female"`.

```
x <- factor(c("male","female",
              "female","male"))
typeof(x)

[1] "integer"

class(x)

[1] "factor"

levels(x)

[1] "female" "male"
```

```
x[2]

[1] female
Levels: female male

as.numeric(x)

[1] 2 1 1 2
```

Usually the first level in a factor is the level that comes first in the alphabet. If we do not want this, we can `relevel` a factor:

```
x<-relevel(x,"male")
x

[1] male   female female male
Levels: male female

as.numeric(x)

[1] 1 2 2 1
```

Note that the `meaning` of the values remains unchanged.

Sometimes, when we have more than only two levels, we want to order levels of a factor along a third variable. This is done by `reorder`.

```
y <- c(12,7,8,11)
x<-reorder(x,y)
x
```

```
[1] male   female female male
attr(,"scores")
  male female
  11.5    7.5
Levels: female male

as.numeric(x)

[1] 2 1 1 2
```

Lists

Lists allow us to combine different data types in one element:

```
x <- list(a=123,b="hello world",c=3)
x[[1]]

[1] 123

x[["a"]]

[1] 123

x$a

[1] 123

x$b

[1] "hello world"
```

Nested lists:

```
y <- list(g=456,h="hello world",i=x)
y$i$c

[1] 3

y[["i"]][["c"]]

[1] 3

typeof(y)

[1] "list"

class(y)

[1] "list"
```

Dataframes

Often we use "rectangular" data structures, i.e. lists where all elements are vectors of the same length.

```
x <- data.frame(a=1:3,b=c("a","b","c"))
x

  a b
1 1 a
2 2 b
3 3 c

x$a

[1] 1 2 3

x$b

[1] "a" "b" "c"
```

```
x[["b"]]

[1] "a" "b" "c"

x[,"b"]

[1] "a" "b" "c"

x[1:2,]

  a b
1 1 a
2 2 b

typeof(x)

[1] "list"
```

2.3 Random numbers

Random numbers can be generated for rather different distributions. R calculates pseudo-random numbers, i.e. R picks numbers from a very long list that appears random. Where we start in this long list is determined by `set.seed`:

```
set.seed(123)
```

10 pseudo-random numbers from a normal distribution can be obtained with

```
rnorm(10)

 [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

We get the same list when we initialise the list with the same starting value:

```
set.seed(123)
rnorm(10)

 [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

This is very useful, when we want to replicate the same "random" results.

10 uniformly distributed random numbers from the interval $[100, 200]$ can be obtained with

```
runif(10,min=100,max=200)

 [1] 188.9539 169.2803 164.0507 199.4270 165.5706 170.8530 154.4066 159.4142
 [9] 128.9160 114.7114
```

Often we use random numbers when we simulate (stochastic) processes. To replicate a process we use the command `replicate`. E.g.

```
replicate(10,mean(rnorm(100)))

 [1]  0.016749257 -0.024755975  0.061320514 -0.028205903  0.087712299
 [6] -0.025113287 -0.141043824  0.123989920  0.109293109 -0.002743263
```

takes 10 times the mean of each 100 pseudo-normally distributed random numbers.

2.4 Example Datasets

We just saw that the command `c` allows us to describe the elements of a vector. For long datasets this is not very convenient. R contains already a lot of example datasets. These datasets are, similar to statistical functions, organised in libraries. To save space and time R does not load all libraries initially. The command `library` allows us to load a library with a dataset at any time.

The library `Ecdat` provides a lot of interesting economic datasets. The library `memisc` gives access to some interesting functions that help us organising our data.

When we need a specific function and we do not know in which library to look for this function we can use the command `RSiteSearch` or the R Site Search Extension for Firefox.

The dataset `BudgetFood` is, e.g., contained in the libarary `Ecdat`.

```
library(Ecdat)
data(BudgetFood)
```

To see the first few records, we can use the command `head`:

```
head(BudgetFood)

      wfood  totexp age size town   sex
1 0.4676991 1290941  43    5    2   man
2 0.3130226 1277978  40    3    2   man
3 0.3764819  845852  28    3    2   man
4 0.4396909  527698  60    1    2 woman
5 0.4036149 1103220  37    5    2   man
6 0.1992503 1768128  35    4    2   man
```

The command `str` shows the structure of an object:

```
str(BudgetFood)

'data.frame': 23972 obs. of  6 variables:
 $ wfood : num  0.468 0.313 0.376 0.44 0.404 ...
 $ totexp: num  1290941 1277978 845852 527698 1103220 ...
 $ age   : num  43 40 28 60 37 35 40 68 43 51 ...
 $ size  : num  5 3 3 1 5 4 4 2 9 7 ...
 $ town  : num  2 2 2 2 2 2 2 2 2 2 ...
 $ sex   : Factor w/ 2 levels "man","woman": 1 1 1 2 1 1 1 2 1 1 ...
```

Usually we do *not* want to see many numbers. Instead we want to derive (in a structured way) a few numbers (parameters, confidence intervals, p-values,...)

The command `help` aids us in finding out the meaning of the numbers of the diferent columns of a dataset.

```
help(BudgetFood)
```

An important command to get a summary is `summary`

```
summary(BudgetFood)
```

How can we access specific columns from our dataset? Since R may have several datasets at the same time in its memory, there are several possibilities. One possibility is to append the name of the dataset `BudgetFood` with a `$` and then the name of the column.

```
BudgetFood$age

 [1] 43 40 28 60 37 35 40 68 43 51 43 48 51 58 61 53 58 64 50 50 47 76 49 44 49
[26] 51 56 63 30 70 29 60 50 56 36 46 43 32 45 34
 [ reached getOption("max.print") -- omitted 23932 entries ]
```

This is helpful when we work with several different datasets at the same time.

The example also shows that R does not flood our screen with long lists of numbers. Instead we only see the first few numbers, and then the text "`omitted ... entries`".

When we want to use only one dataset, then the command `attach` is helpful.

```
attach(BudgetFood)
age

 [1] 43 40 28 60 37 35 40 68 43 51 43 48 51 58 61 53 58 64 50 50 47 76 49 44 49
[26] 51 56 63 30 70 29 60 50 56 36 46 43 32 45 34
 [ reached getOption("max.print") -- omitted 23932 entries ]
```

From now on, all variables will first be searched in the dataset `BudgetFood`. When we no longer want this, then we say

```
detach(BudgetFood)
```

A third possibility is the command `with`:

```
with(BudgetFood,age)

 [1] 43 40 28 60 37 35 40 68 43 51 43 48 51 58 61 53 58 64 50 50 47 76 49 44 49
[26] 51 56 63 30 70 29 60 50 56 36 46 43 32 45 34
 [ reached getOption("max.print") -- omitted 23932 entries ]
```

We often use `with` when we use a function and want to refer to a specific dataset in this function. E.g. `hist` shows a histogram:

```
with(BudgetFood,hist(age))
```

Most commands have several options which allow you to fine-tune the result. Have a look at the help-page for `hist` (you can do this with `help(hist)`). Perhaps you prefer the following graph:

```
with(BudgetFood,hist(age,breaks=40,xlab="Age [years]",col=gray(.7),main="Spain"))
```

3 Functions

3.1 Introduction

R knows many built-in functions:

```
mean(x)
median(x)
max(x)
min(x)
length(x)
unique(c(1,2,3,4,1,1,1))
```

When we need more, we can write our own:

```
square <- function(x) {
  x*x
}
```

The last expression in a function (here `x*x`) is the return value. Now we can use the function.

```
square(7)

[1] 49
```

3.2 Arguments to functions

- Positional arguments.

- Named arguments.
- Default arguments.

```
myF <- function(a,b,c)
    a + b * c
myF(1,2,3)

[1] 7

myF(a=1,b=2,c=3)

[1] 7

myF(c=3,a=1,b=2)

[1] 7

myF(c=3,1,2)

[1] 7
```

... arguments

```
myF <- function(n,...)
    runif(n,...)
myF(10)

 [1] 0.5741897 0.5776351 0.5899991 0.2229283 0.1034923 0.7365060 0.6124038
 [8] 0.9512124 0.9598759 0.7448361

myF(10,-1)

 [1] -0.13033971 -0.65044973 -0.12942386  0.52491638  0.83805862  0.81339517
 [7]  0.63102566 -0.60049411 -0.90074883 -0.04210295

myF(10,min=-1)

 [1]  0.1807867 -0.8067595  0.9017805 -0.6745137  0.8430125 -0.7444499
 [7]  0.3254792  0.5680839  0.5292263  0.3451406

myF(n=10,min=-1,max=1)

 [1] -0.7686682  0.7026030  0.2359232  0.6728231 -0.6601436 -0.9794376
 [7] -0.3530605 -0.4606346  0.6505227  0.8673235
```

Local variables

```
myF <- function(n) {
  x <- n
  print(x)
  print(z)
}
x <- 7
z <- 8
myF(5)

[1] 5
[1] 8

x

[1] 7

myG <- function(n) {
  x <<- n
  print(x)
}
x <- 7
myG(5)

[1] 5

x

[1] 5
```

Closures

```
getStore <- function(store=0) {
  list(
    add = function(x) {
      store <<- store + x
    },
    show = function() {
      store
    }
  )
}
##
storeA <- getStore(0)
storeB <- getStore(-5)
storeA$show()

[1] 0

storeB$show()

[1] -5
```

```
storeA$add(10)
storeA$show()

[1] 10

storeB$add(20)
storeB$show()

[1] 15
```

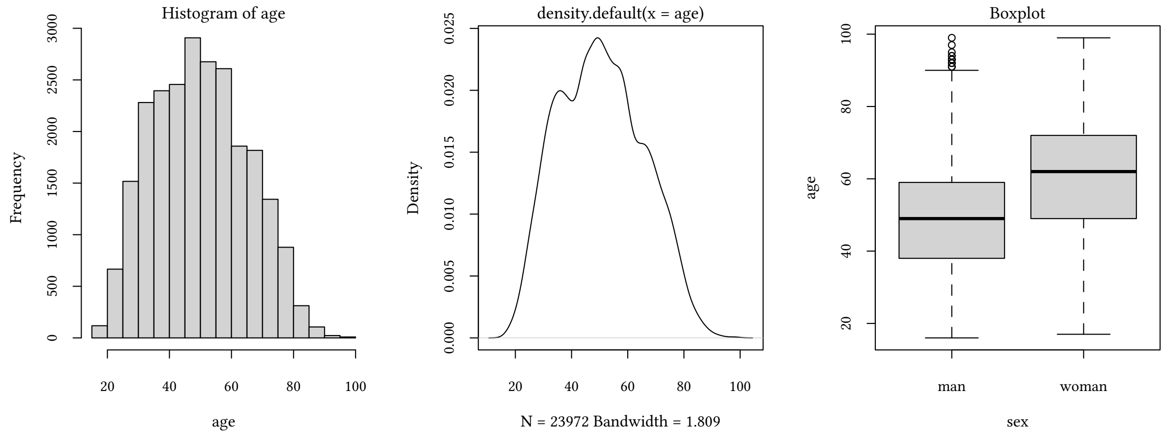
4 Graphs

There is more than one way to represent numbers as graphs.

4.1 Basic Graphs

Here are three basic graphs – they all show the same data:

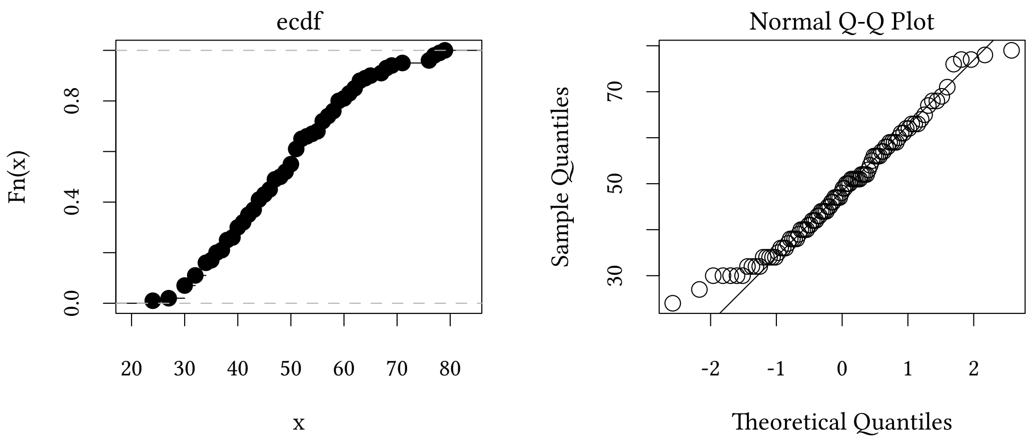
```
with(BudgetFood, {
  hist(age)
  plot(density(age))
  boxplot(age ~ sex,main="Boxplot")
})
```



Two further helpful plots are `ecdf` and `qqnorm`:

```
x <- sample(BudgetFood$age,
            100)
plot(ecdf(x),main="ecdf")
```

```
qqnorm(x)
qqline(x)
```



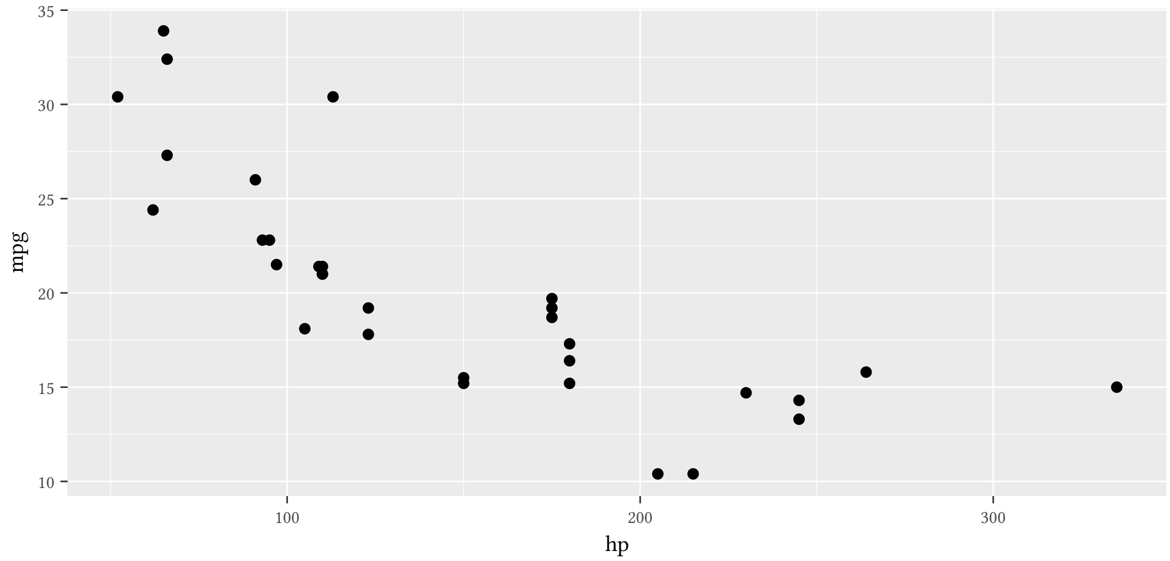
- Sometimes it is obvious how to prepare our data for these functions. Sometimes it is more complicated. Then other commands help and calculate an object that can be plotted (with `plot`)
  - `density`, `ecdf`, `xyplot`...
- Some commands then plot whatever we have prepared:
  - `plot`, `hist`, `boxplot`, `barplot`, `curve`, `mosaicplot`,...
- Yet other commands add something to an existing plot:
  - `points`, `text`, `lines`, `abline`, `qqline`...

4.2 ggplot2

R provides different functions to plot data: The basic `plot` command, the `lattice` library, and the `ggplot2` library are some of them.

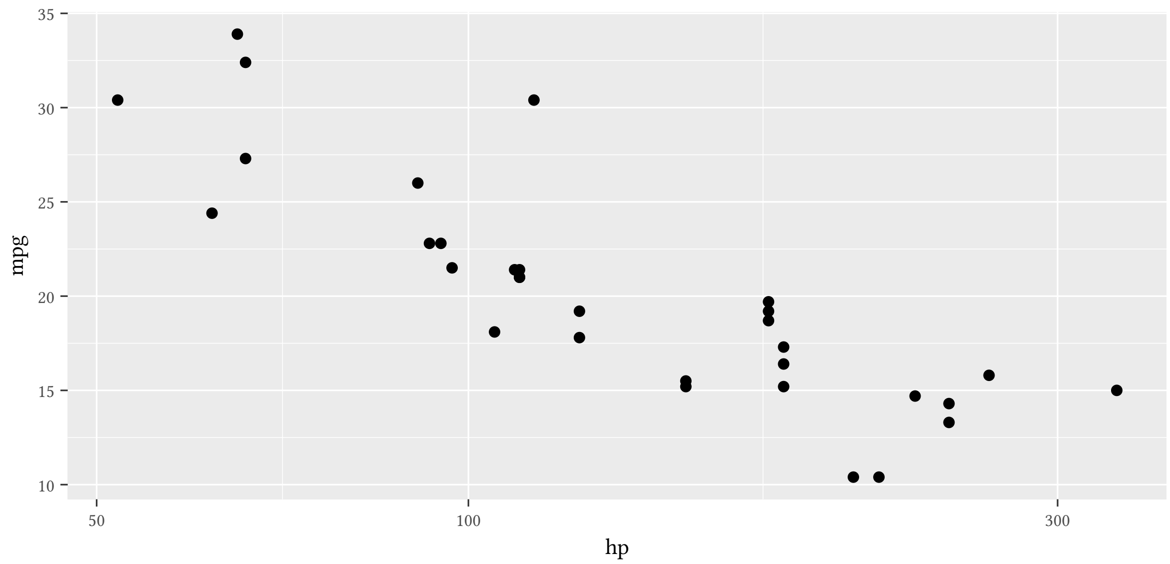
Here we have a look at `ggplot2`:

```
library(tidyverse)
mtcars %>%
    ggplot(aes(x=hp,y=mpg)) + geom_point()
```



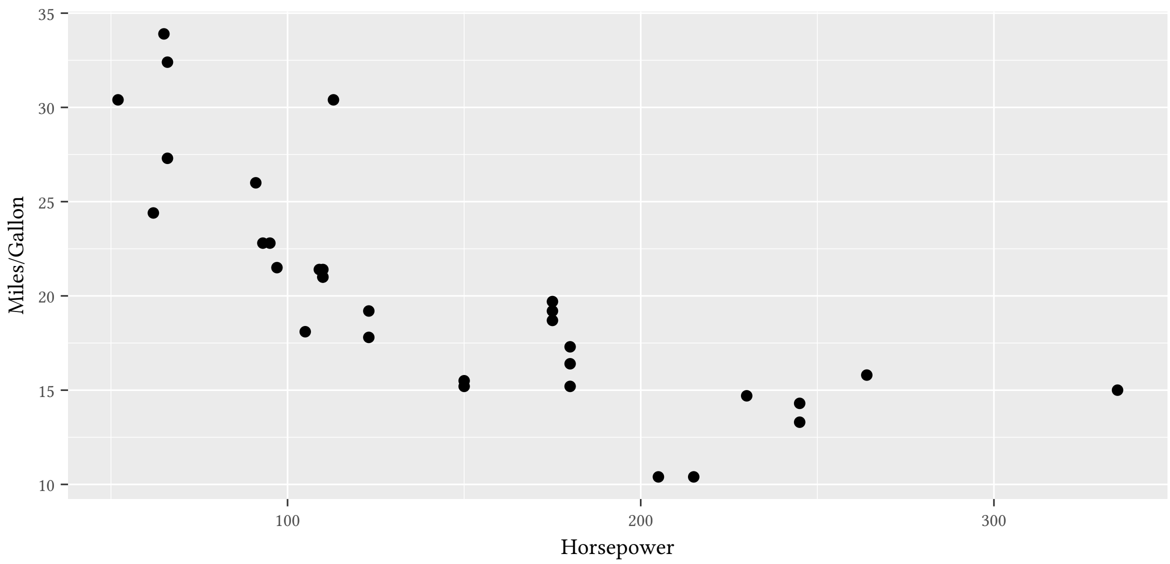
Scales

```
mtcars %>%
    ggplot(aes(x=hp,y=mpg)) + geom_point() + scale_x_log10()
```



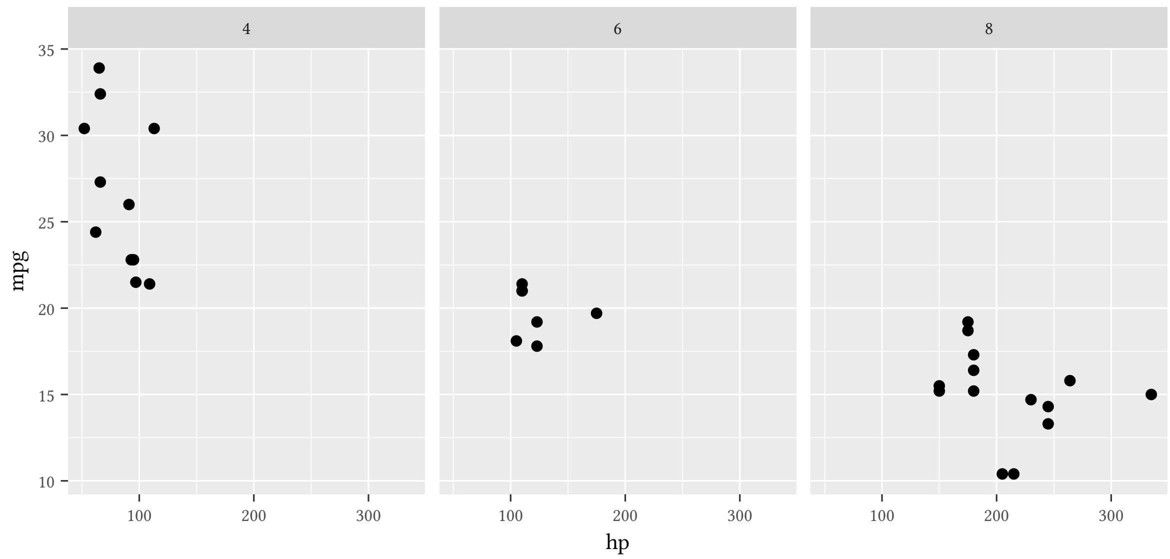
Labels

```
mtcars %>%
    ggplot(aes(x=hp,y=mpg)) + geom_point() + labs(x="Horsepower",y="Miles/Gallon")
```



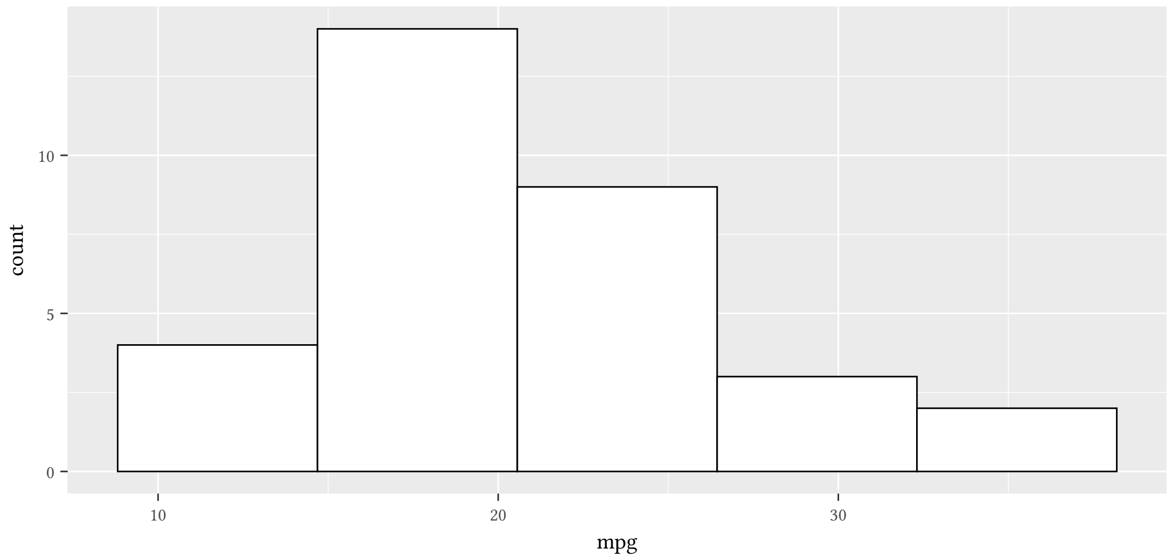
Facets

```
mtcars %>%
    ggplot(aes(x=hp,y=mpg)) + geom_point() + facet_grid(cols=vars(cyl))
```



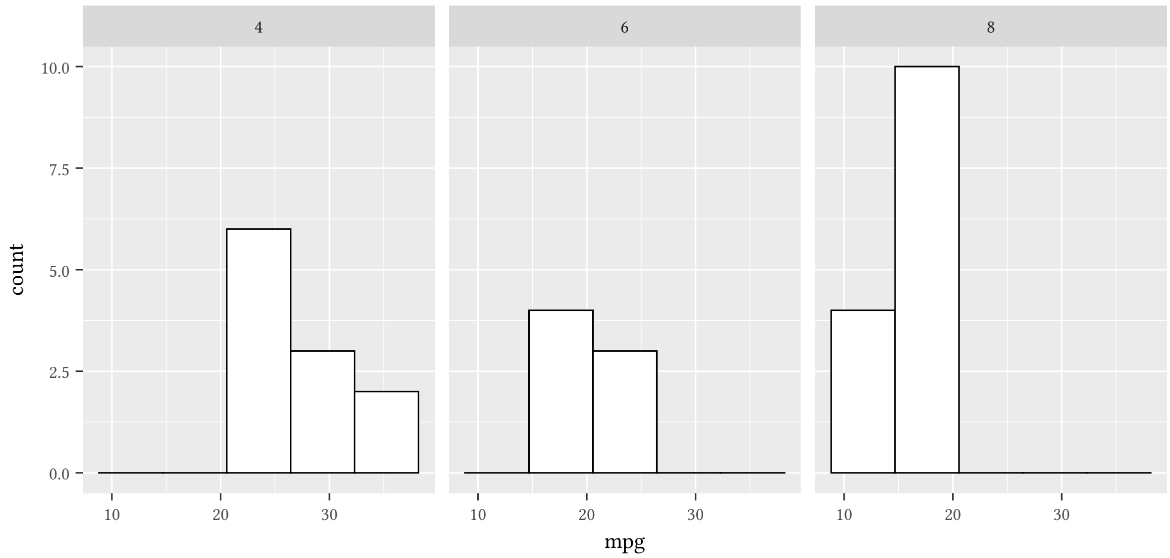
Univariate

```
mtcars %>%
    ggplot(aes(x=mpg)) + geom_histogram(fill='white',color='black',bins=5)
```



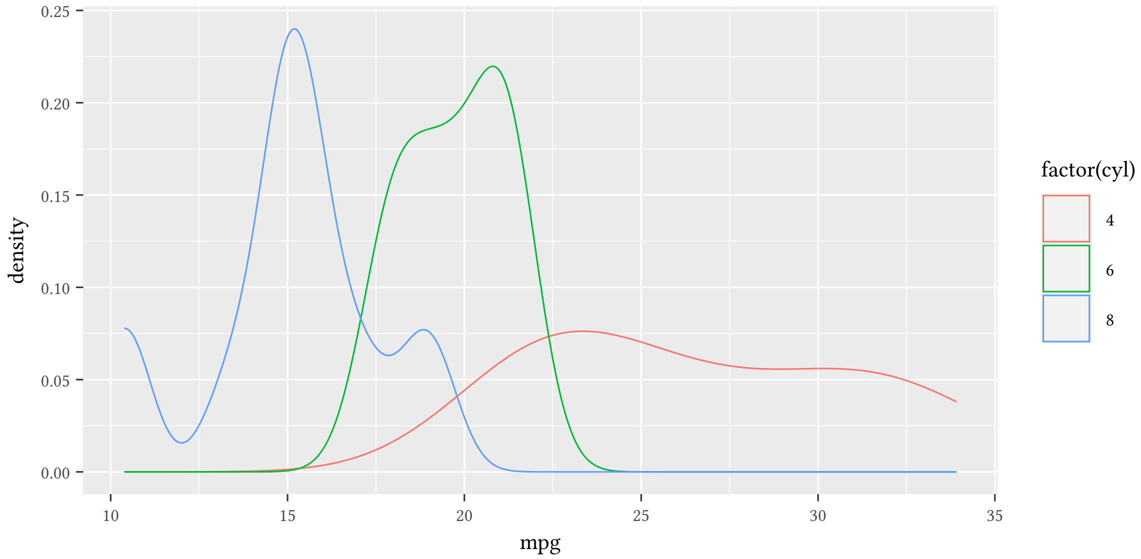
Comparing several distibutions with histograms is less obvious, still:

```
mtcars %>%
   ggplot(aes(x=mpg)) +
   geom_histogram(fill='white',color='black',bins=5) +
   facet_grid(cols=vars(cyl))
```



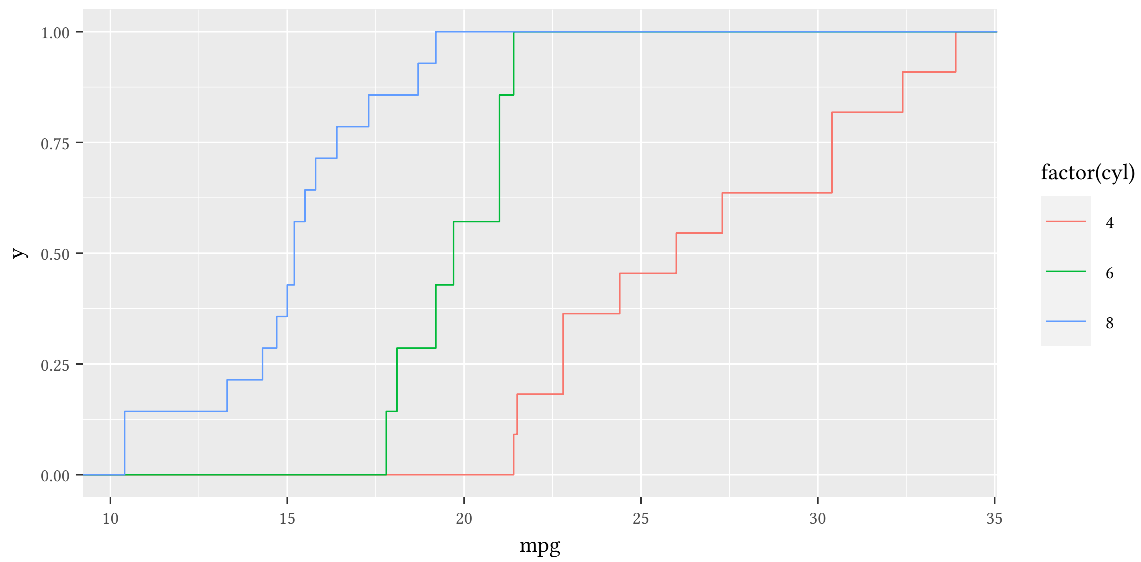
With densities it is easier to compare several distributions:

```
mtcars %>%
    ggplot(aes(x=mpg,color=factor(cyl))) + geom_density()
```



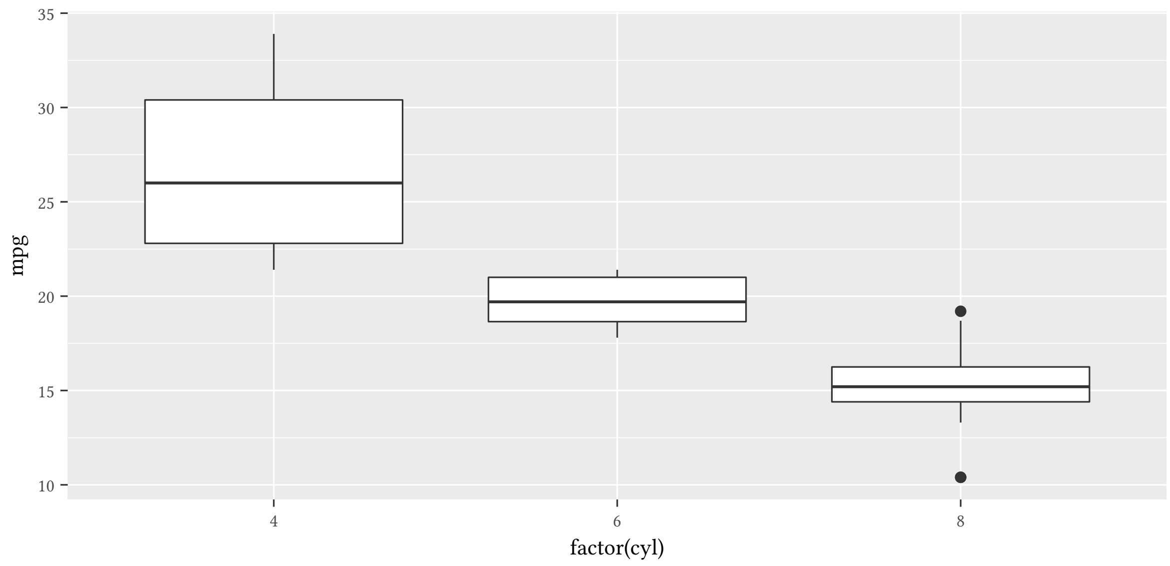
Densities need, however, a model in the background. How "smooth" is the distribution? Empirical cumulative distribution functions don't need such a model:

```
mtcars %>%
    ggplot(aes(x=mpg,color=factor(cyl))) + stat_ecdf()
```



Boxplots are useful, in particular when we have many categories:

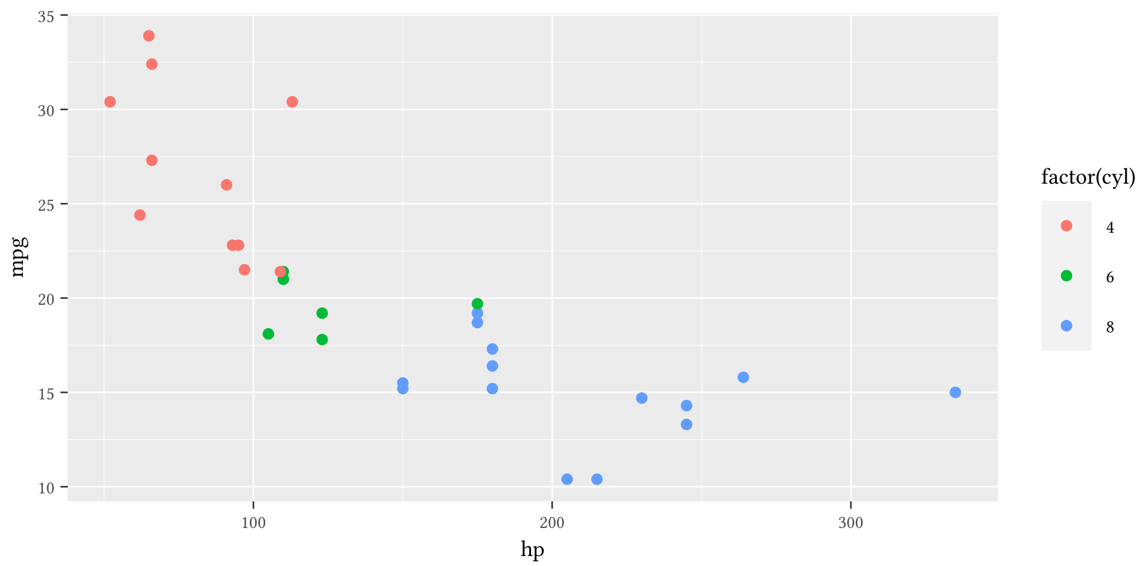
```
mtcars %>%
    ggplot(aes(x=factor(cyl),y=mpg)) + geom_boxplot()
```



Bivariate

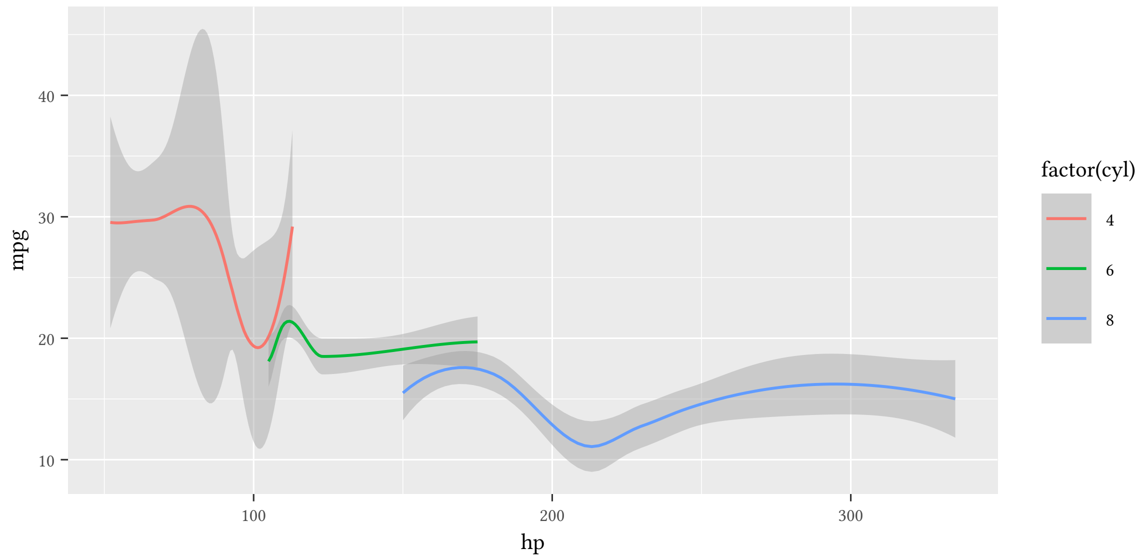
Here is a simple scatterplot:

```
mtcars %>%
    ggplot(aes(x=hp,y=mpg,color=factor(cyl))) + geom_point()
```



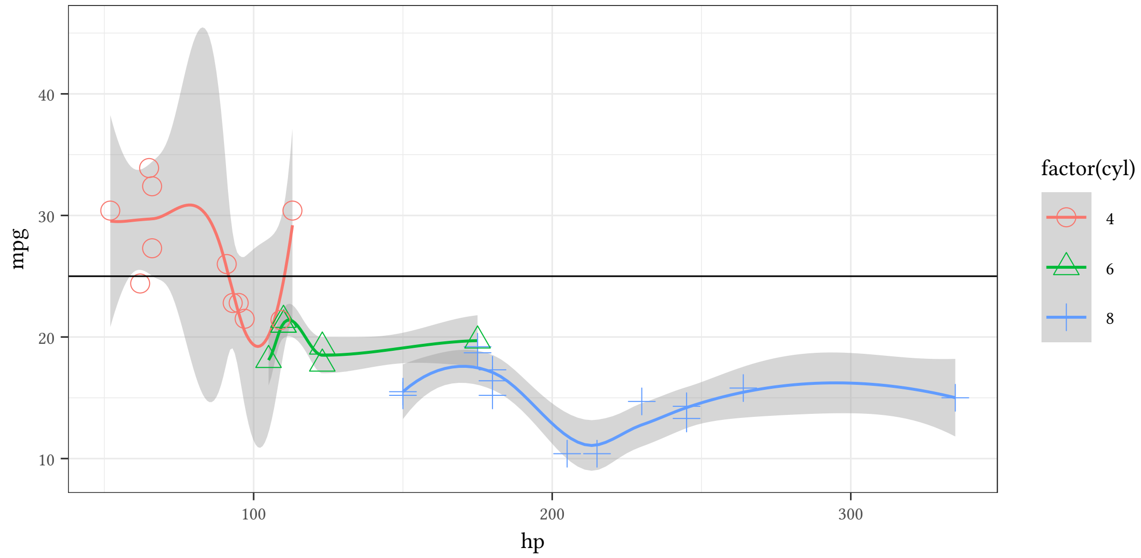
Here is a smooth line through the points:

```
mtcars %>%
    ggplot(aes(x=hp,y=mpg,color=factor(cyl))) + geom_smooth()
```



And here we combine line and points:

```
mtcars %>%
    ggplot(aes(x=hp,y=mpg,color=factor(cyl),shape=factor(cyl))) +
    geom_smooth() +
    geom_point(size=3) + scale_shape_manual(values=1:3) +
    geom_hline(yintercept=25) + theme_bw()
```



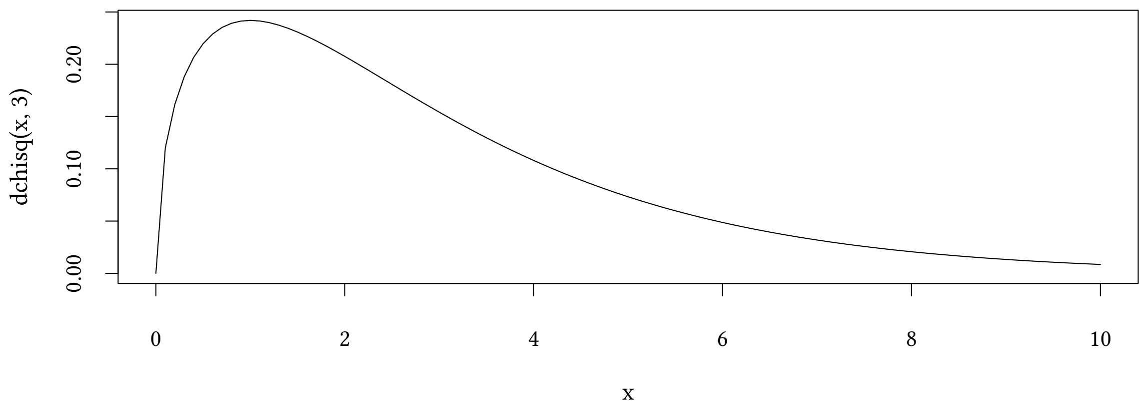
4.3 Basic plot

`plot` is the “basic” plot function of R.

Plotting functions

We can plot functions of `x` with `curve`.

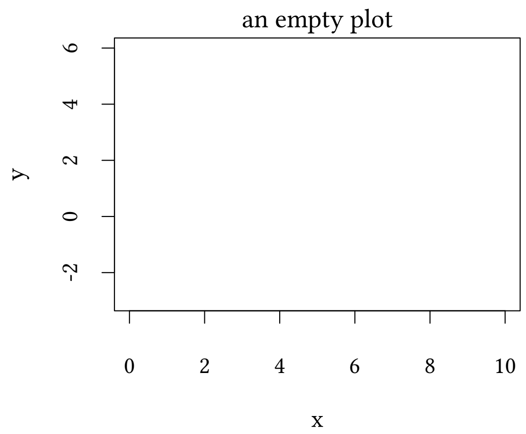
```
curve(dchisq(x,3),from=0,to=10)
```



Empty plots

Sometimes it is helpful to start with an empty plot. Then we have to help `plot` a little bit. Usually, `plot` can guess from the data the limits and labels of the axes. With an empty plot we have to specify them explicitely.

```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="an empty plot")
```

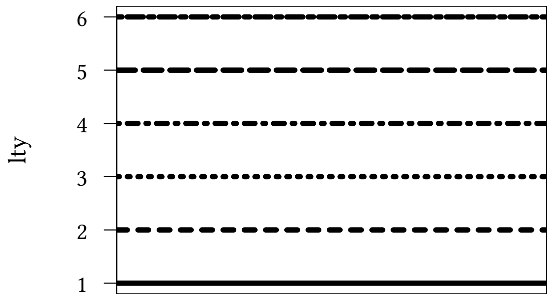


Line type

Almost all commands that draw lines follow the following conventions:

- `lty` linetype (`"dashed"`, `"dotted"`, or simply a number)

```
plot(NULL,ylim=c(1,6),xlim=c(0,1),xaxt="n",ylab="lty",las=1)
sapply(1:6,function(lty) abline(h=lty,lty=lty,lwd=5))
```



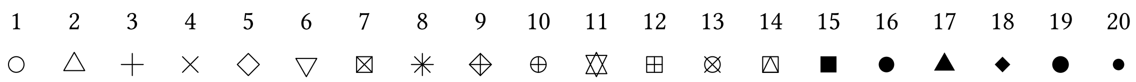
- `lwd` linewidth (a number)

- `col` colour (`"red"`, `"green"`, `gray(0.5)`)

Points

The character used to draw points is determined with `pch`.

```
range=1:20
plot(range,range/range,pch=range,frame=FALSE)
text(range,range/range+.2,range)
```

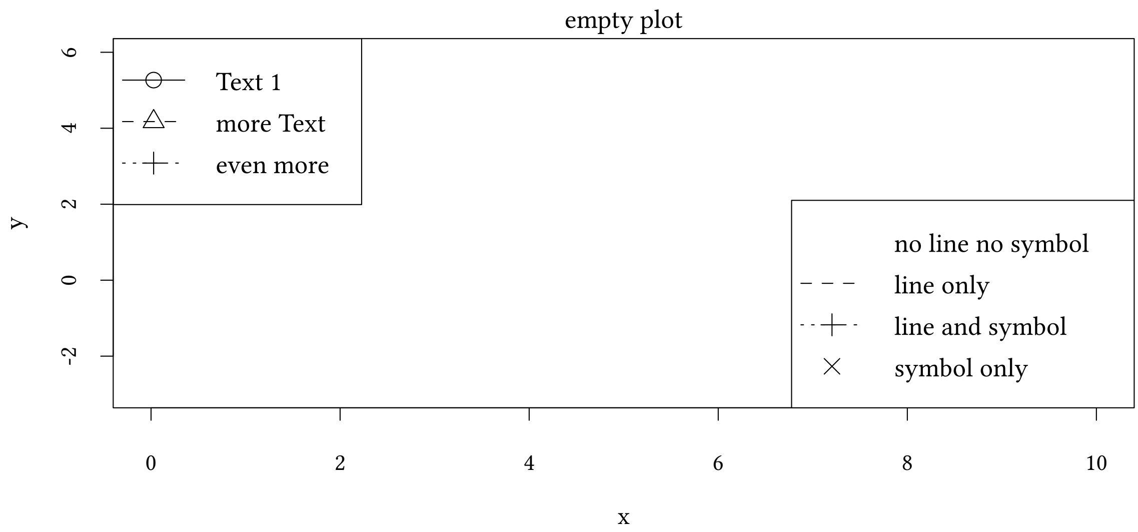


Legends

When we use more than one line or more than one symbol in our plot we have to explain their meaning. This is done in a legend.

Usually `legend` gets as an option a vector of linetypes `lty` and symbols `pch`. They will be used to construct example lines and symbols next to the actual text of the legend. If the `lty` or `pch` is `NA`, then no line or point is drawn.

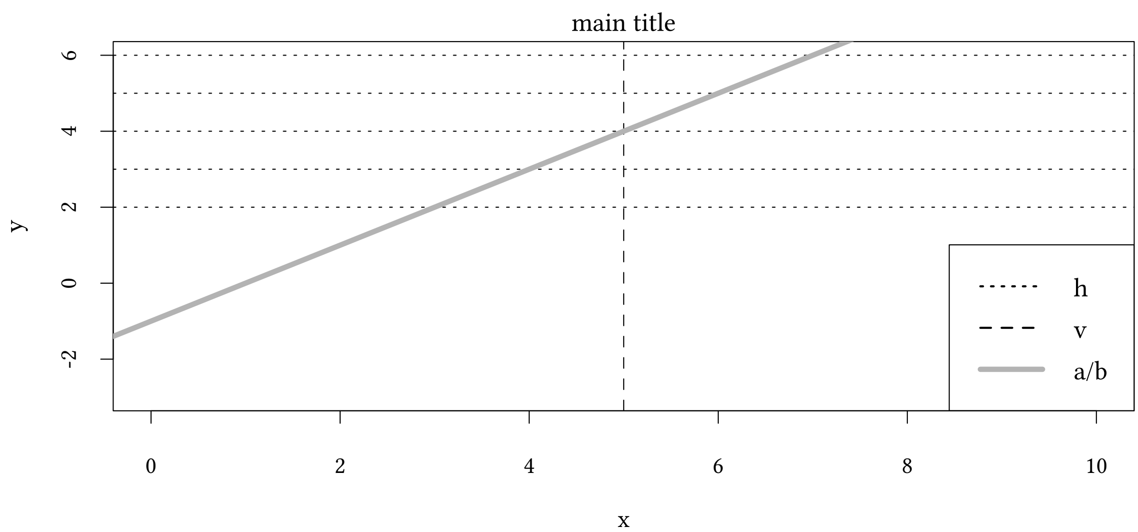
```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="empty plot")
legend("topleft",c("Text 1","more Text","even more"),lty=1:3,pch=1:3)
legend("bottomright",c("no line no symbol","line only","line and symbol","symbol only"),
       lty=c(NA,2,3,NA),pch=c(NA,NA,3,4),bg="white")
```



Auxiliary lines

The command `abline` allows us to add auxiliary lines to a plot.

```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="main title")
abline(h=2:6,lty="dotted")
abline(v=5,lty="dashed")
abline(a=-1,b=1,lwd=5,col=grey(.7))
legend("bottomright",c("h","v","a/b"),lty=c("dotted","dashed","solid"),col=c("black","black",grey(.7)),l
```



`abline` knows the following important parameters:

- `h=` for horizonal lines

- v= for vertical lines
- a=..., b=... for lines with intercept a and slope b

Note, that these arguments can be vectors if we want to draw several lines at the same time.

Axes

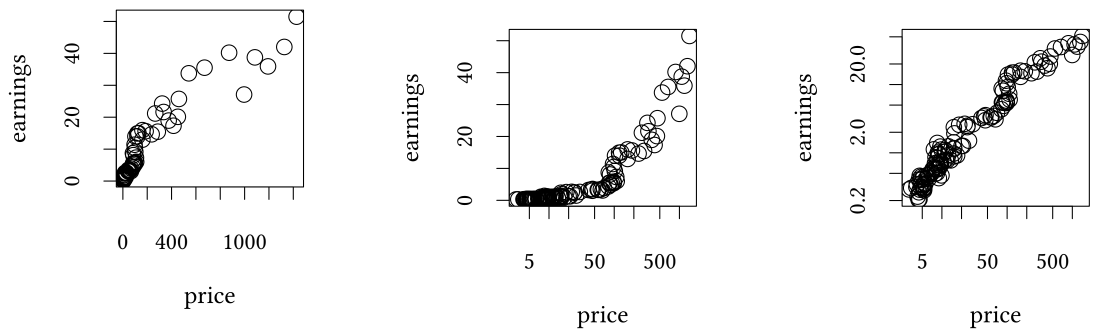
The options log='x', log='y' or log='xy' determine whether which axis is shown in a logarithmic style.

```
data(PE,package="Ecdat")
xx<-data.frame(PE)
attach(xx)
```

```
plot(price, earnings)
```

```
plot(price, earnings,
   log="x")
```

```
plot(price, earnings,
   log="xy")
```

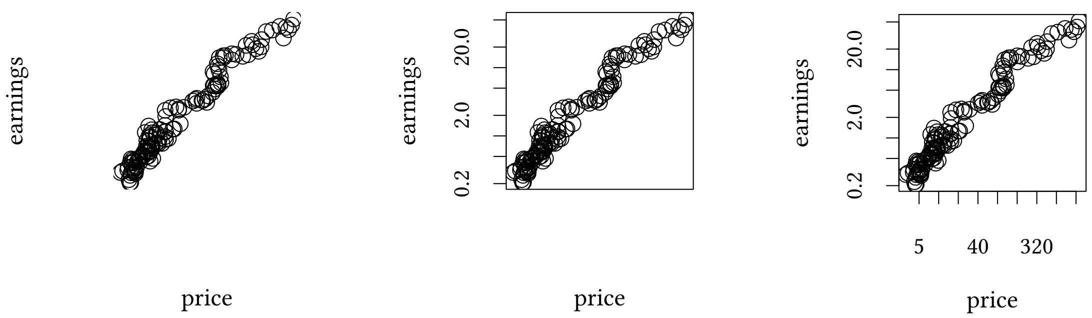


To gain more flexibility axis can draw a wide range of axes. Before using axis the previous axes can be removed entirely (axes=FALSE) or suppressed selectively (xaxt="n" or yaxt="n").

```
plot(price, earnings,
   log="xy",axes=FALSE)
```

```
plot(price, earnings,
   log="xy",xaxt="n")
```

```
plot(price, earnings,
   log="xy",xaxt="n")
axis(1,at=c(5,10,20,40,
   80,160,320,640,1280))
```

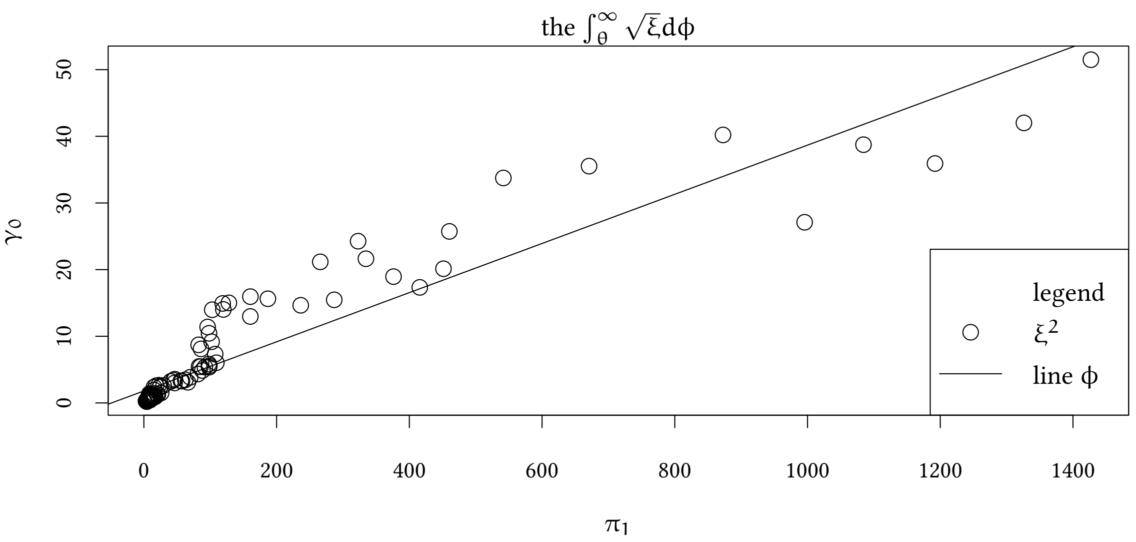


If we specify a lot of axes labels, as in the example above, R does not print them all if they overlap.

Fancy math

R can also render more than only textual labels. If you use `tikz` as an output device you can use LaTeX-notation. Otherwise you can use `plotmath`.

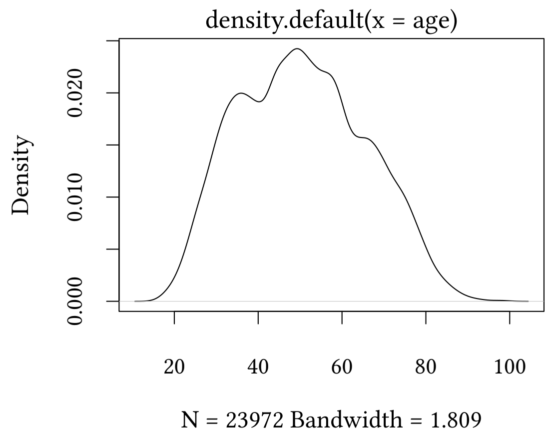
```
plot(price, earnings,xlab='$\\pi_1$',ylab='$\\gamma_0$',
     main="the $\\int_\\theta^\\infty \\sqrt{\\xi} d\\phi$")
abline(lm(earnings~price))
legend("bottomright",c("legend","$\\xi^2$","line $\\phi$"),pch=c(NA,1,NA),lty=c(NA,NA,1))
```



Several diagrams

Diagrams side by side To put several diagrams on one plot side by side we can call `par(mfrow=c(...))` or `layout` or `split.screen`.

```
par(mfrow=c(1,2))
with(BudgetFood, {
  hist(age)
  plot(density(age))
})
```

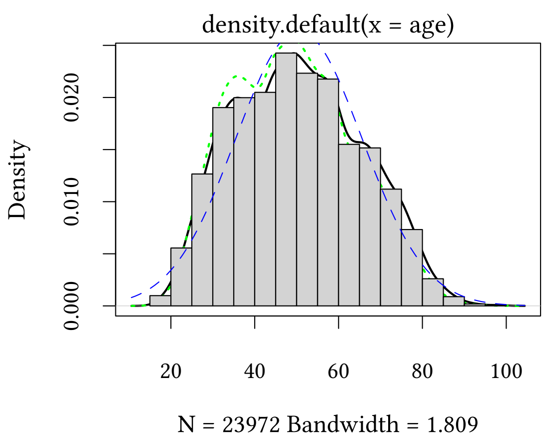


Superimposed graphs

- Anything that can create `lines` or `points` (like `density` or `ecdf`) can immediately be added to an existing plot.

- Plot-objects that would otherwise create a new figure (like `plot`, `hist`, or `curve`) can be added to an existing plot with the optional parameter `add=TRUE`.

```
with(BudgetFood, {
  plot(density(age),lwd=2)
  lines(density(age[sex=="man"],na.rm=TRUE),
        lty=3,lwd=2,col="green")
  hist(age,freq=FALSE,add=TRUE)
  curve(dnorm(x,mean(age),sd(age)),
        add = TRUE,lty=2,col="blue")
})
```



5 Files

Accessing the filesystem

- `getwd(), setwd(...)`
- `dir(...), list.files(...)`
- `file.info(...)`
- `file.create(...)`
- `file.exists(...)`
- `file.remove(...)`
- `file.rename(from, to)`
- `file.append(file1, file2)`
- `file.copy(from, to)`
- `file.symlink(from, to)`
- `file.link(from, to)`
- `tempfile()`

Rdata

- `save(x,y,z,file=...)`
- `save.image(file=...)`
- `load(...)`

CSV files

```
d <- head(mtcars,1)
write.table(d)

"mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear" "carb"
"Mazda RX4" 21 6 160 110 3.9 2.62 16.46 0 1 4 4

write.csv(d)

"","mpg","cyl","disp","hp","drat","wt","qsec","vs","am","gear","carb"
"Mazda RX4",21,6,160,110,3.9,2.62,16.46,0,1,4,4

write.csv2(d)
```

```
"";"mpg";"cyl";"disp";"hp";"drat";"wt";"qsec";"vs";"am";"gear";"carb"
"Mazda RX4";21;6;160;110;3,9;2,62;16,46;0;1;4;4

write.table(d,dec=",",quote=FALSE)

mpg cyl disp hp drat wt qsec vs am gear carb
Mazda RX4 21 6 160 110 3,9 2,62 16,46 0 1 4 4
```

Other files

Format	Library	Command
Stata (5-12)	foreign	read.dta, write.dta
Stata (13-)	readstata13	read.dta13, write.dta
Stata (8-15)	haven	read_dta
xls,xlsx	readxl	read_excel
xls,xlsx	WriteXLS	WriteXLS
xlsx	xlsx	read.xlsx, saveWorkbook...
z-Tree	zTree	zTreeTables, zTreeSbj

```
library(readxl)
fn<-readxl_example("datasets.xlsx")
excel_sheets(fn)

[1] "iris"     "mtcars"   "chickwts" "quakes"

read_excel(fn,sheet="quakes",range="A1:G5")

# A tibble: 4 x 7
    lat  long depth   mag stations ...6  ...7
  <dbl> <dbl> <dbl> <dbl>    <dbl> <lgl> <lgl>
1 -20.4  182.   562   4.8       41 NA    NA
2 -20.6  181.   650   4.2       15 NA    NA
3 -26    184.    42   5.4       43 NA    NA
4 -18.0  182.   626   4.1       19 NA    NA
```

6 Pipes

6.1 Pipes

Sometimes you want to apply functions of functions:

```
x <- 1:10
var(x)

[1] 9.166667

sqrt(var(x))

[1] 3.02765
```

Alternatively store intermediate results in a variable:

```
varx <- var(x)
sqrt(varx)

[1] 3.02765
```

Or use a pipe:

```
var(x) |> sqrt()

[1] 3.02765
```

So far the nesting is not so hard to understand. There is no need to rewrite the code using pipes. A deeply nested function can be harder to understand. Here is a more complicated example:

```
library(dplyr)
summarize(group_by(filter(mtcars,!is.na(am) & !is.na(cyl)),am,cyl),
          disp=mean(disp),hp=mean(hp))

# A tibble: 6 x 4
# Groups:   am [2]
     am   cyl  disp    hp
  <dbl> <dbl> <dbl> <dbl>
1     0     4 136.   84.7
2     0     6 205.  115.
3     0     8 358.  194.
4     1     4  93.6  81.9
5     1     6 155   132.
6     1     8 326   300.
```

To make the code more readable, we could store intermediate results in a variable (xx)

```
## summarize(group_by(filter(mtcars,!is.na(am) & !is.na(cyl)),am,cyl),disp=mean(disp),hp=mean(hp))
xf <- filter(mtcars,!is.na(am) & !is.na(cyl))
xg <- group_by(xf,am,cyl)
summarize(xg,disp=mean(disp),hp=mean(hp))

# A tibble: 6 x 4
# Groups:   am [2]
     am   cyl  disp    hp
  <dbl> <dbl> <dbl> <dbl>
1     0     4 136.   84.7
2     0     6 205.  115.
3     0     8 358.  194.
4     1     4  93.6  81.9
5     1     6 155   132.
6     1     8 326   300.
```

We could combine all this into a single chain of functions:
The %>% operator from from `dplyr` allows us to chain functions more transparently.

```
## summarize(group_by(filter(mtcars,!is.na(am) & !is.na(cyl)),am,cyl),disp=mean(disp),hp=mean(h
mtcars %>%
  filter(!is.na(am), !is.na(cyl))  %>%
  group_by(am,cyl) %>%
  summarise(disp=mean(disp),hp=mean(hp))

# A tibble: 6 x 4
# Groups:   am [2]
     am   cyl  disp    hp
  <dbl> <dbl> <dbl> <dbl>
1     0     4 136.   84.7
2     0     6 205.  115.
3     0     8 358.  194.
4     1     4  93.6  81.9
5     1     6 155   132.
6     1     8 326   300.
```

7 Control structures

7.1 Conditional evaluation

if

```
x <- runif(1)
if (x>0.5)
    print("large x")

[1] "large x"
```

```
if (x>0.5)
    print("large x") else
    print("small x")

[1] "large x"
```

```
x <- 1:10
ifelse(x>5,x*10,x/10)

 [1]   0.1   0.2   0.3   0.4   0.5  60.0  70.0  80.0  90.0 100.0
```

7.2 Loops

for

```
for (i in 1:5)
    cat(i)

12345
```

Ex ante we know all conditions of the loop.

while

```
i <- 1
while(i<6) {
    cat(i)
    i <- i+1
}

12345
```

We don't know conditions ex ante, but we can decide at the beginning of the loop.

repeat

```
i <- 1
repeat {
    cat(i)
    if ( i>=5)
        break
    i <- i+1
}

12345
```

Flexible: we don't know and we don't decide at the beginning.

8 Structuring data

8.1 sapply and lapply

When we want to apply a function to each element of a vector or a list, `sapply` helps:

```
range <- 1:3
square <- function(x)
  x*x
```

```
lapply(range,square)

[[1]]
[1] 1
```

```
[[2]]
[1] 4

[[3]]
[1] 9
```

lapply returns a list. sapply returns a vector (or a matrix,...).

```
sapply(range,square)

[1] 1 4 9
```

We do not have to define a name for a function:

```
sapply(range,function(x) x*x)

[1] 1 4 9
```

sapply can be faster than for.

apply

apply applies a function along one or more dimensions of an array:

```
example <- matrix(1:9,nrow=3) %>% print

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

apply(example,MARGIN=1,FUN=mean)

[1] 4 5 6

apply(example,MARGIN=2,FUN=mean)

[1] 2 5 8
```

Splitting by groups

```
with(mtcars,split(mpg,cyl))

$`4`
 [1] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26.0 30.4 21.4

$`6`
[1] 21.0 21.0 21.4 18.1 19.2 17.8 19.7

$`8`
 [1] 18.7 14.3 16.4 17.3 15.2 10.4 10.4 14.7 15.5 15.2 13.3 19.2 15.8 15.0
```

Often we want to perform a calculation for each group:

```
with(mtcars,sapply(split(mpg,cyl),mean))

       4        6        8
26.66364 19.74286 15.10000
```

`tapply` is a shorthand for the combination of `sapply` and `split`:

```
with(mtcars,tapply(mpg,cyl,mean))

       4        6        8
26.66364 19.74286 15.10000
```

Often we want to split entire dataframes, not only vectors:

The command `aggregate` groups our data by levels of one or several factors and applies a function to each group. In the following example the factor is `cyl`, the function is the `mean` which is applied to the variable `mpg`.

(this is similar to the `group_by(...) %>% summarise(...)` we had earlier)

```
with(mtcars,aggregate(mpg ~ cyl,FUN=mean))

  cyl      mpg
1   4 26.66364
2   6 19.74286
3   8 15.10000
```

Alternatively, with dplyr:

```
mtcars %>%
    group_by(cyl) %>%
    summarise(mean(mpg))

# A tibble: 3 x 2
    cyl `mean(mpg)`
  <dbl>       <dbl>
1     4        26.7
2     6        19.7
3     8        15.1
```

```
by(mtcars,mtcars$cyl,function(d) with(d,c(min=min(mpg),max=max(mpg)))) %>% sapply(c)

       4    6    8
min 21.4 17.8 10.4
max 33.9 21.4 19.2
```

With the `dplyr` library (from `tidyverse`) we can express this perhaps more clearly:

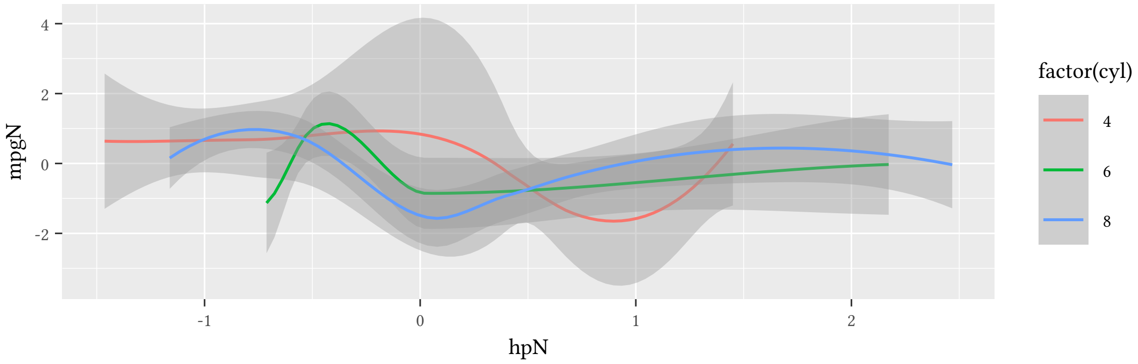
```
mtcars %>%
    group_by(cyl) %>%
    summarise(min=min(mpg),max=max(mpg))

# A tibble: 3 x 3
    cyl   min   max
  <dbl> <dbl> <dbl>
1     4  21.4  33.9
2     6  17.8  21.4
3     8  10.4  19.2
```

8.2 The tidyverse

The `tidyverse` is a collection of libraries that helps to structure and rearrange data. Above we have already met functions like `group_by` and `summarise`.

Mutate

```
mtcars %>%
    group_by(cyl) %>%
    mutate(mpgN = (mpg - mean(mpg))/sd(mpg),
           hpN = (hp - mean(hp))/sd(hp)) %>%
    ungroup() %>%
    ggplot(aes(x=hpN,y=mpgN,color=factor(cyl))) + geom_smooth()
```




```
mtcars %>% select(starts_with("m")) %>% head(2)

              mpg
Mazda RX4      21
Mazda RX4 Wag  21

mtcars %>% select(matches("m")) %>% head(2)

              mpg am
Mazda RX4      21  1
Mazda RX4 Wag  21  1
```

```
mtcars %>% head(1)

          mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4

mtcars %>% relocate(matches("m"),.after="gear") %>% head(1)

          cyl disp  hp drat   wt  qsec vs gear mpg am carb
Mazda RX4   6  160 110  3.9 2.62 16.46  0    4  21  1    4
```

```
mtcars %>% arrange(mpg) %>% head

                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
 [ reached 'max' / getOption("max.print") -- omitted 3 rows ]

mtcars %>% arrange(desc(cyl),desc(hp)) %>% head

                mpg cyl disp  hp drat   wt  qsec vs am gear carb
Maserati Bora  15.0   8  301 335 3.54 3.57 14.60  0  1    5    8
Ford Pantera L 15.8   8  351 264 4.22 3.17 14.50  0  1    5    4
Duster 360     14.3   8  360 245 3.21 3.57 15.84  0  0    3    4
 [ reached 'max' / getOption("max.print") -- omitted 3 rows ]
```

Often we join information from two different datasets. For the example, before we join, we have to create a second table:

```
secondTable<-read.csv(text="cyl,class
4,A
6,B
8,C")
mtcars %>% left_join(secondTable) %>% head

   mpg cyl disp  hp drat    wt  qsec vs am gear carb class
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4     B
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4     B
3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1     A
 [ reached 'max' / getOption("max.print") -- omitted 3 rows ]
```

left_join	observations in left table
right_join	observations in right table
inner_join	observations in both tables
full_join	all observations

```
example <- matrix(runif(25),ncol=5,dimnames=list(1:5,2000+1:5)) %>%
    data.frame %>% rownames_to_column("i") %>% print

  i      X2001     X2002      X2003      X2004     X2005
```

```
1 1 0.08063882 0.6727282 0.09341809 0.63250079 0.1798389
2 2 0.70731594 0.1621797 0.63992514 0.83075783 0.3286921
3 3 0.56366877 0.1085019 0.56404244 0.05135562 0.8548957
4 4 0.28280791 0.5334442 0.30553652 0.99233504 0.1843603
5 5 0.39205677 0.1031991 0.26600903 0.02208719 0.7953567

example %>% pivot_longer(cols=starts_with("X"))

# A tibble: 25 x 3
   i     name   value
   <chr> <chr>  <dbl>
 1 1     X2001 0.0806
 2 1     X2002 0.673
 3 1     X2003 0.0934
 4 1     X2004 0.633
 5 1     X2005 0.180
 6 2     X2001 0.707
 7 2     X2002 0.162
 8 2     X2003 0.640
 9 2     X2004 0.831
10 2     X2005 0.329
# ... with 15 more rows
```

```
example2 <- data.frame(Subject=1:5,year=2001:2005) %>%
    expand(Subject,year) %>%
    add_column(v=runif(25)) %>%
    print

# A tibble: 25 x 3
   Subject  year      v
     <int> <int>  <dbl>
 1       1  2001 0.859
 2       1  2002 0.472
 3       1  2003 0.648
 4       1  2004 0.325
 5       1  2005 0.466
 6       2  2001 0.942
 7       2  2002 0.245
 8       2  2003 0.0225
 9       2  2004 0.967
10       2  2005 0.260
# ... with 15 more rows
```

```
example2 %>%
    pivot_wider(names_from=year,values_from=v)

# A tibble: 5 x 6
  Subject `2001` `2002` `2003` `2004` `2005`
    <int>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1       1  0.859 0.472  0.648   0.325  0.466
2       2  0.942 0.245  0.0225  0.967  0.260
```

```
3        3  0.654 0.0754 0.969    0.919   0.868
4        4  0.424 0.781  0.300    0.289   0.999
5        5  0.987 0.480  0.762    0.329   0.236
```

```
example2 %>% pivot_wider(names_from=year,names_prefix="year.",values_from=v)

# A tibble: 5 x 6
  Subject year.2001 year.2002 year.2003 year.2004 year.2005
    <int>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
1       1     0.859     0.472    0.648      0.325     0.466
2       2     0.942     0.245    0.0225     0.967     0.260
3       3     0.654     0.0754   0.969      0.919     0.868
4       4     0.424     0.781    0.300      0.289     0.999
5       5     0.987     0.480    0.762      0.329     0.236
```

9 Tables

Tables of frequencies

The command `table` calculates a table of frequencies. Here we show only the first 16 columns:

```
library(magrittr)
mtcars %$%
    table(cyl ,gear )

   gear
cyl  3  4  5
  4  1  8  2
  6  2  4  1
  8 12  0  2
```

```
mtcars %$%
    table(cyl ,gear ) %>%
    prop.table(margin=1)

   gear
cyl          3          4          5
  4 0.09090909 0.72727273 0.18181818
  6 0.28571429 0.57142857 0.14285714
  8 0.85714286 0.00000000 0.14285714
```

```
mtcars %$%
    table(cyl ,gear ) %>%
    prop.table(margin=2)

   gear
cyl          3          4          5
  4 0.06666667 0.66666667 0.40000000
  6 0.13333333 0.33333333 0.20000000
  8 0.80000000 0.00000000 0.40000000
```

10 Regressions

Simple regressions can be estimated with `lm`. The operator ~ allows us to describe the regression equation. The dependent variable is written on the left side of ~, the indenpendent variables are written on the right side of ~.

```
lm (wfood ~ totexp,data=BudgetFood)


Call:
lm(formula = wfood ~ totexp, data = BudgetFood)

Coefficients:
  (Intercept)         totexp
 0.4950397225  -0.0000001348
```

The result is a bit terse. More details are shown with the command `summary`.

```
summary(lm (wfood ~ totexp,data=BudgetFood))


Call:
lm(formula = wfood ~ totexp, data = BudgetFood)

Residuals:
     Min       1Q   Median       3Q      Max
-0.49307 -0.09374 -0.01002  0.08617  1.06182

Coefficients:
                   Estimate      Std. Error t value Pr(>|t|)
(Intercept)  0.495039722500  0.001561819134  316.96   <2e-16 ***
totexp      -0.000000134849  0.000000001459  -92.41   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1422 on 23970 degrees of freedom
Multiple R-squared:  0.2627,Adjusted R-squared:  0.2626
F-statistic:  8540 on 1 and 23970 DF,  p-value: < 2.2e-16
```

11 Starting and stopping R

Whenever we start R, the program attempts to find a file `.Rprofile`, first in the current working directory, then in the home directory. If the file is found, it is “sourced”, i.e. all R commands in this file are executed. This is useful when we want to run the same commands whenever we start R. The following line

```
options(browser = "/usr/bin/firefox")
```

in `.Rprofile` makes sure that the help system of R always uses `firefox`.

Also when we quit R with the command `q()`, the application tries to make our life easier.

```
q()
```

R first asks us

```
Save workspace image? [y/n/c]:
```

Here we have the possibility to save all the data that we currently use (and that are in our workspace) in a file `.Rdata` in the current working directory. When we start R for the next time (from this directory) R automatically reads this file and we can continue our work.