R Graphics

An overview of possibilities

Kévin Cazelles and Nicolas Casajus
Université du Québec à Rimouski

Outline

- ✓ Introduction
- ✓ The `graphics` package basis
- ✓ Composition and multi-panel plotting
- ✓ Graphics automation and exporting
- ✓ Resources
- ✓ Exercises

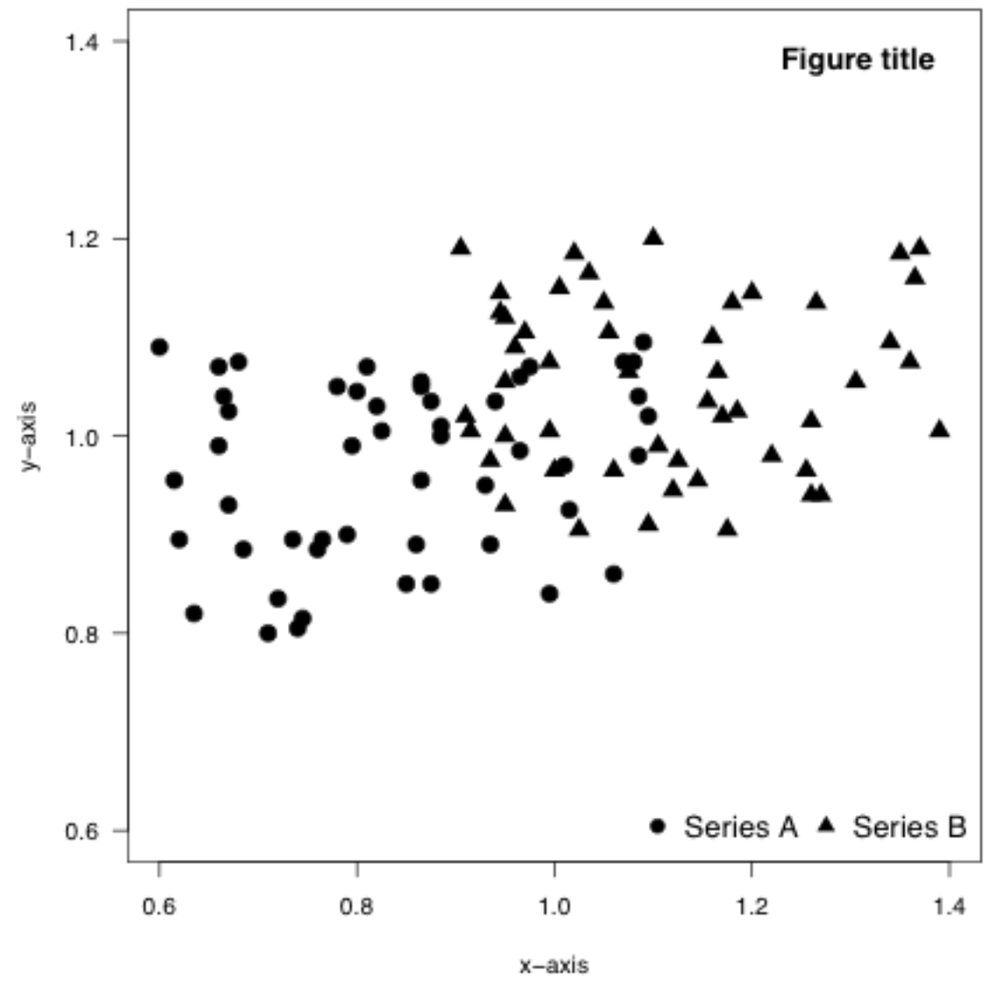
The importance of graphics

> “A picture is worth a thousand words”

- ✓ Visual summary of data / information
- ✓ More efficient than table and text
- ✓ Useful for exploring data
    - trends, correlations, cycles, outliers, etc.
- ✓ Essential for presenting results

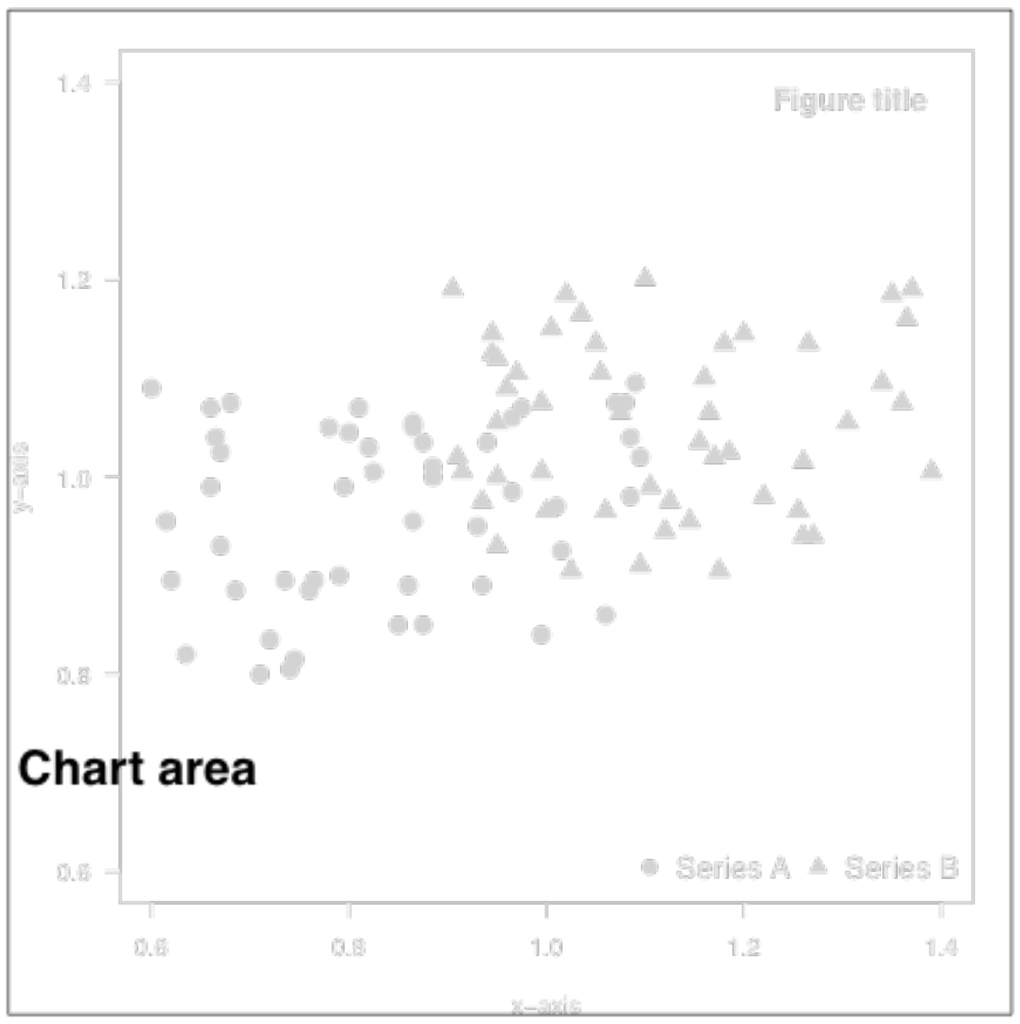
- ✓ But a bad graph can lie about data
- ✓ And sometimes a graphic is not the solution

The components of a graphic



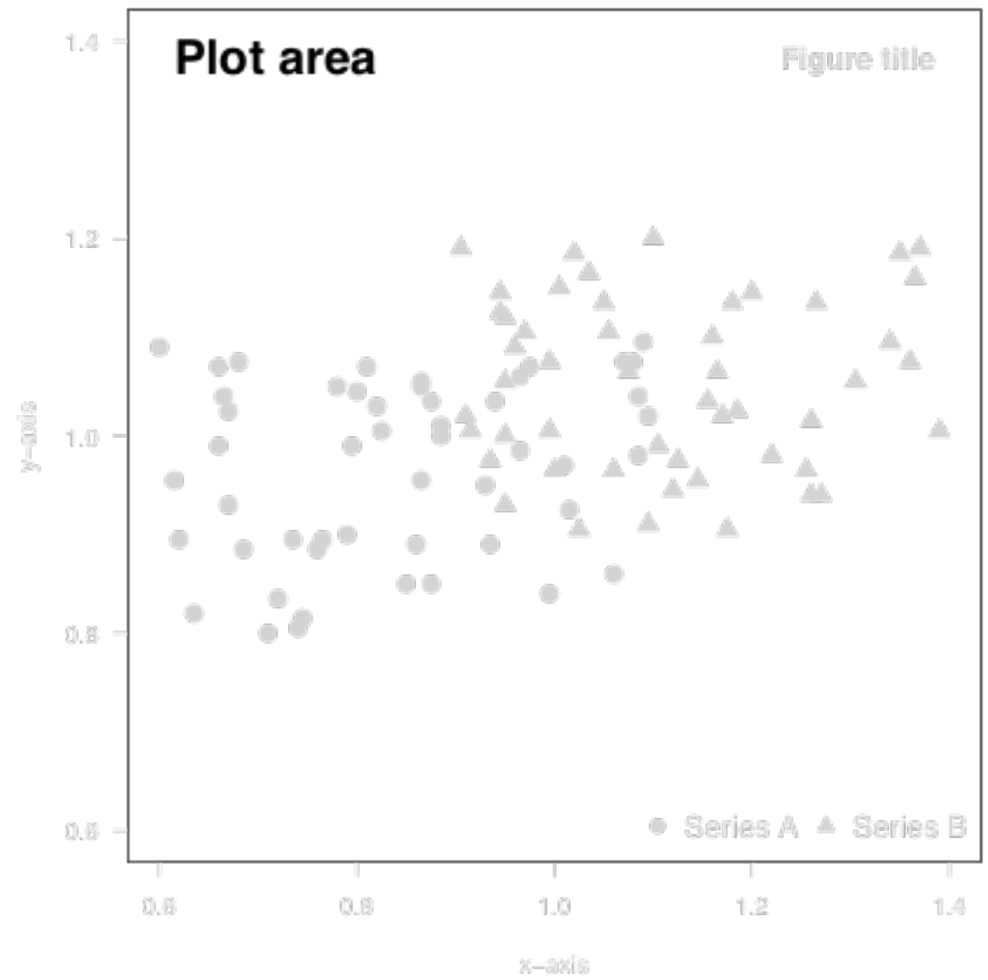
The components of a graphic

- ✓ Chart area



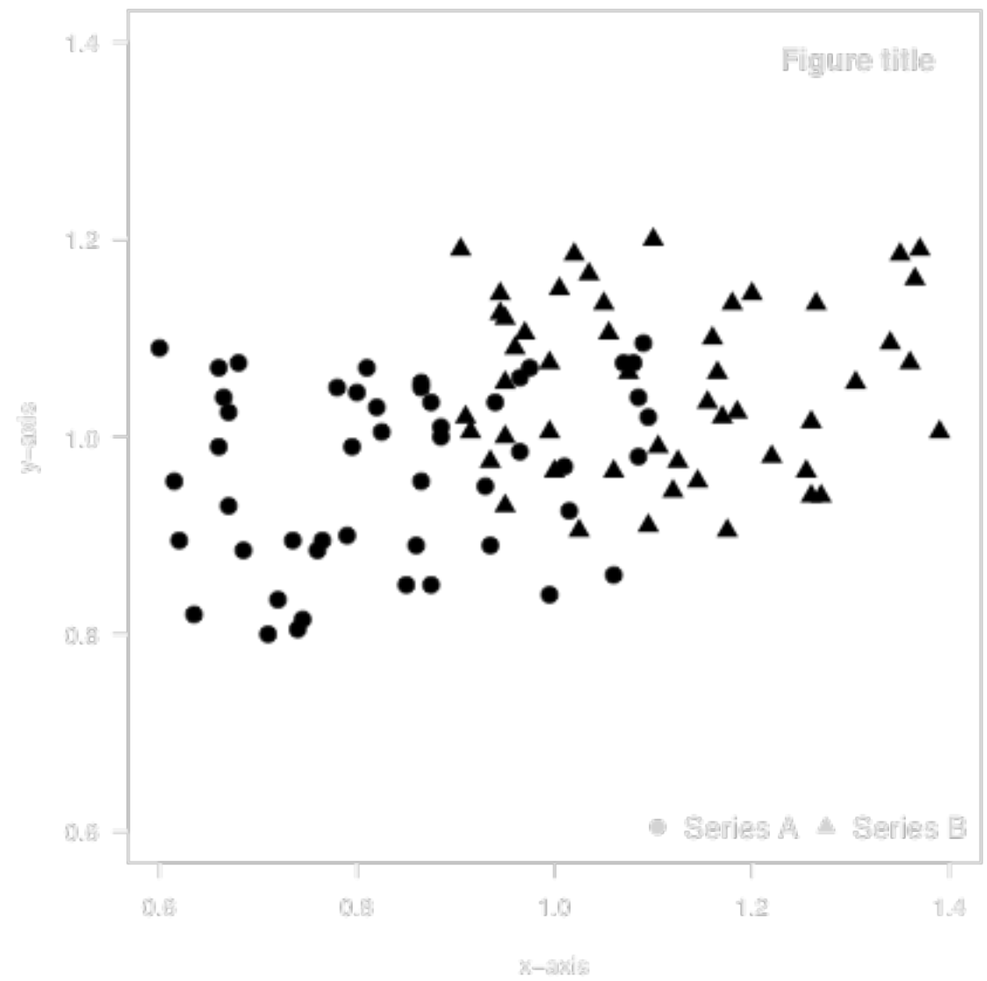
The components of a graphic

- ✓ Chart area
- ✓ Plot area



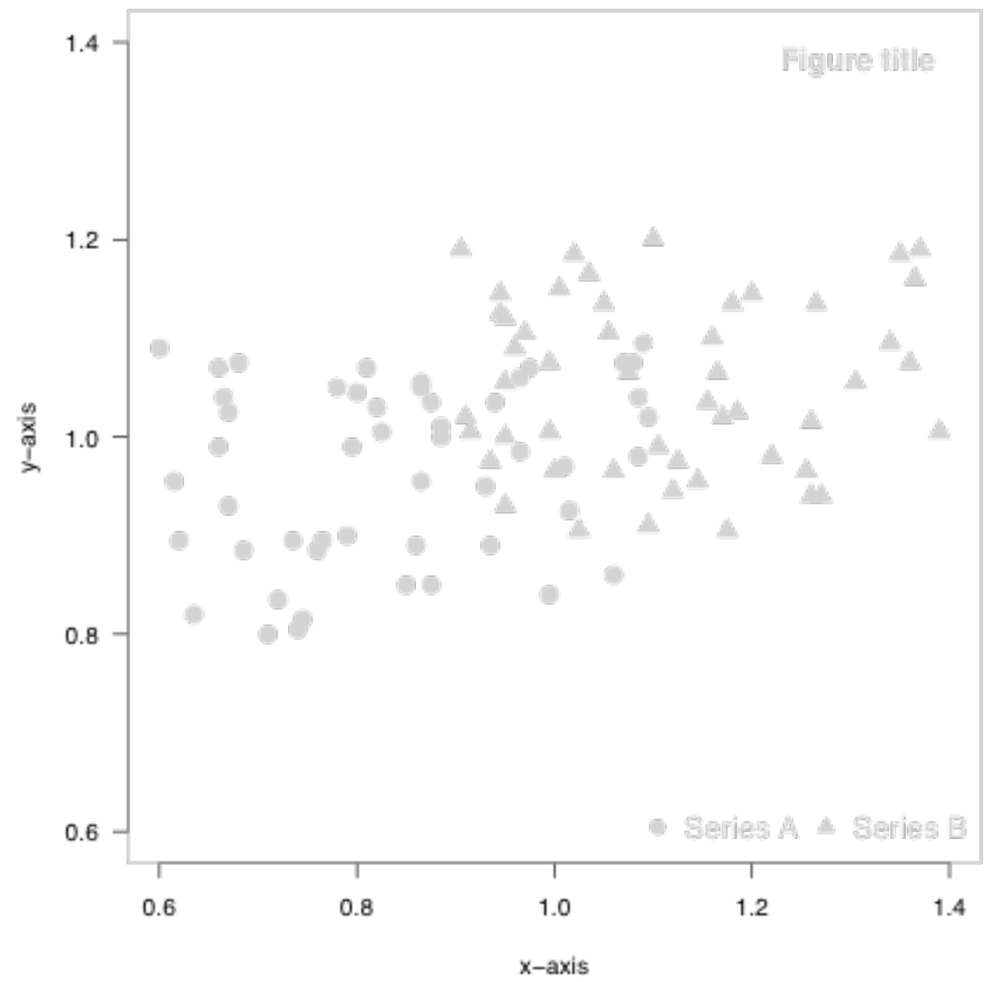
The components of a graphic

- ✓ Chart area
- ✓ Plot area
- ✓ Data representation



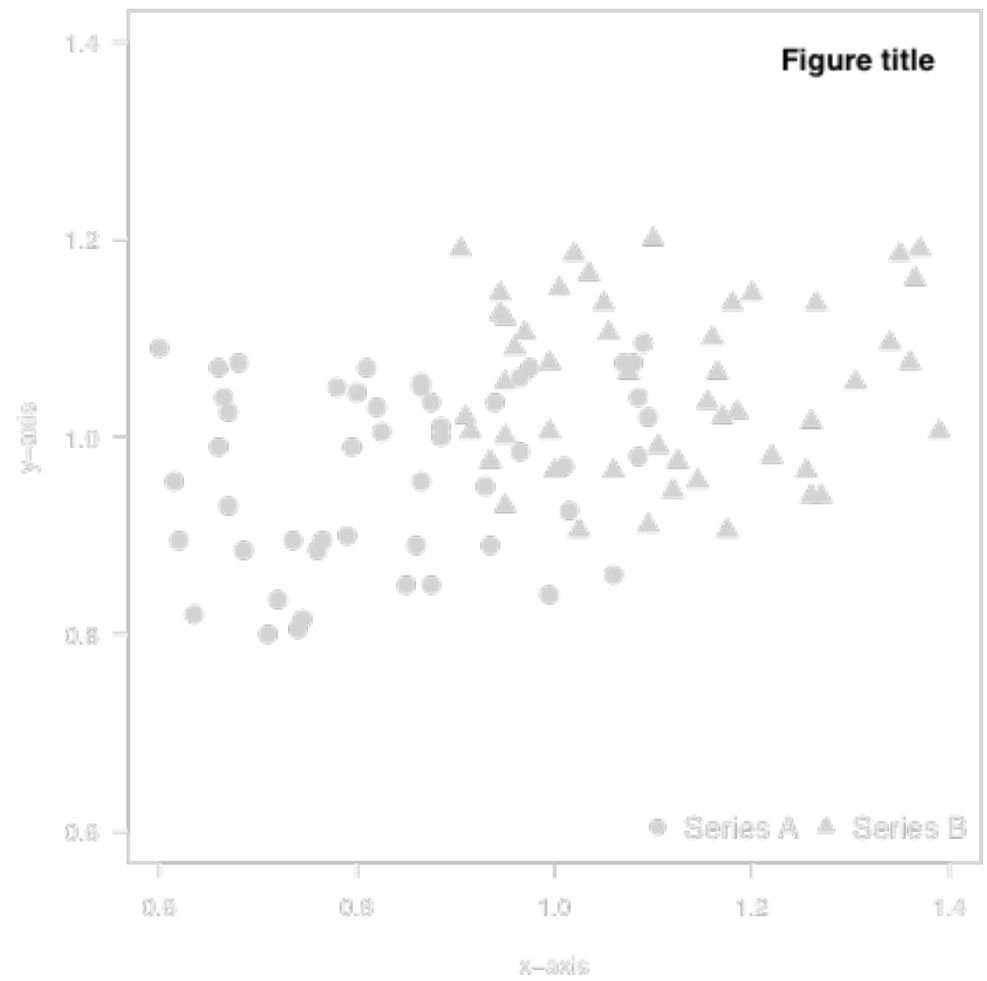
The components of a graphic

- ✓ Chart area
- ✓ Plot area
- ✓ Data representation
- ✓ Axis (scaling, labeling)



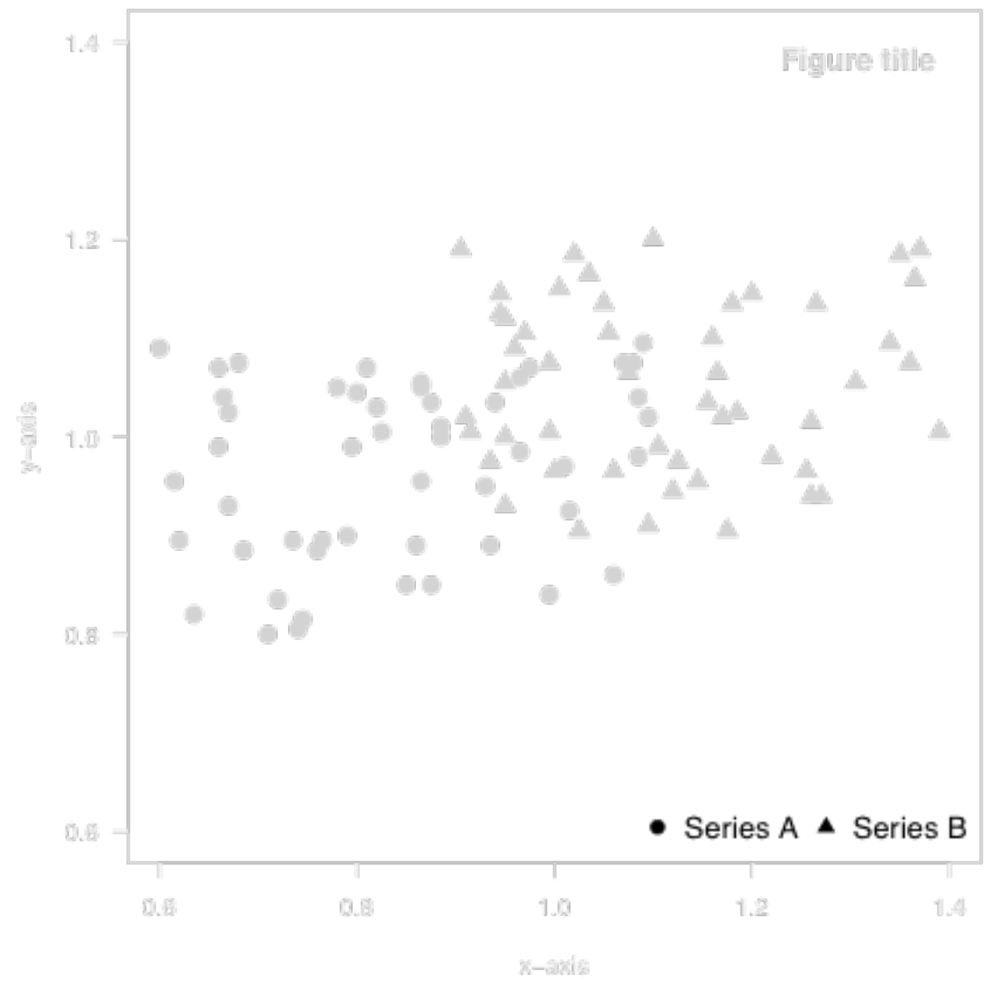
The components of a graphic

- ✓ Chart area
- ✓ Plot area
- ✓ Data representation
- ✓ Axis (scaling, labeling)
- ✓ Figure title



The components of a graphic

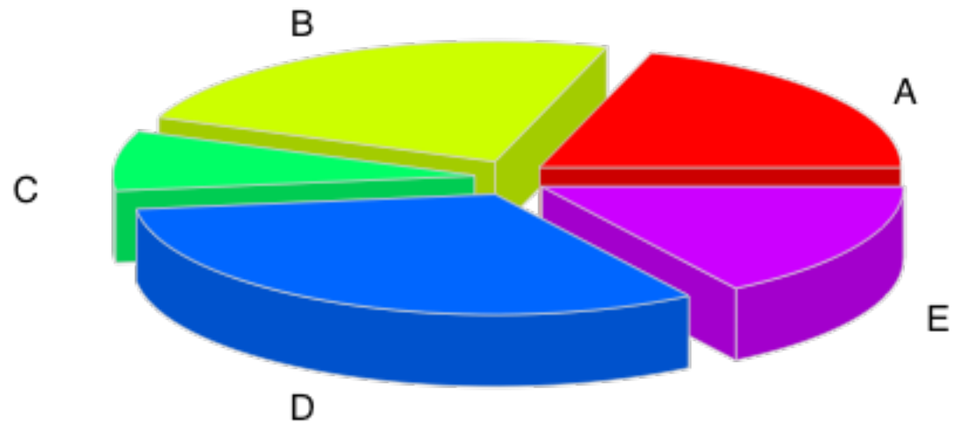
- ✓ Chart area
- ✓ Plot area
- ✓ Data representation
- ✓ Axis (scaling, labeling)
- ✓ Figure title
- ✓ Legend



Some guidelines for better graphics

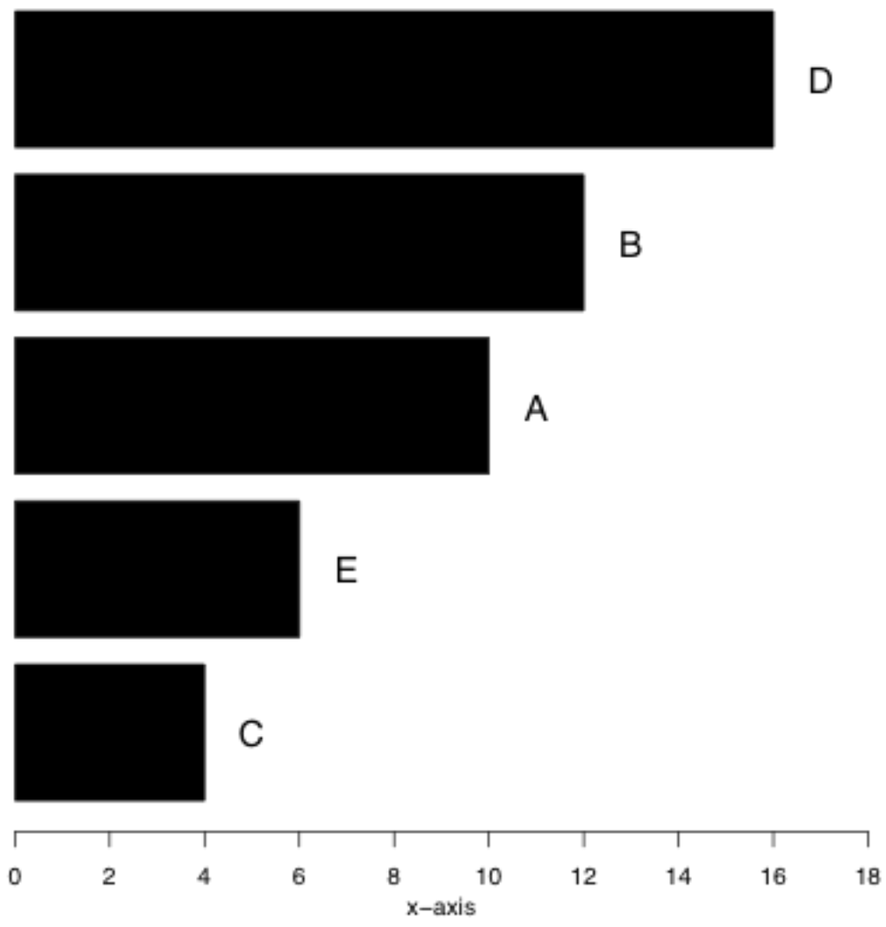
Some guidelines for better graphics

- ✓ Do not use pie chart
- ✓ Do not use 3D (never)
- ✓ Use consistent colors



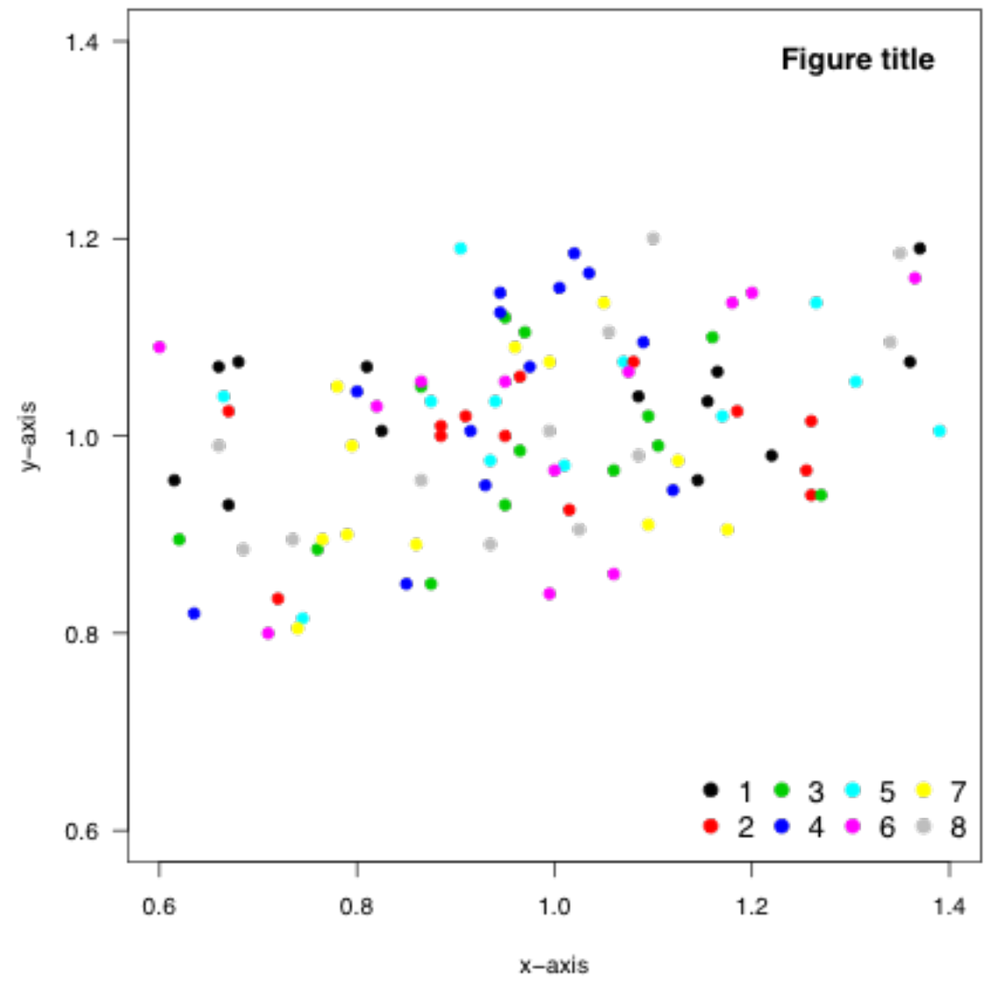
Some guidelines for better graphics

- ✓ Do not use pie chart
- ✓ Do not use 3D (never)
- ✓ Use consistent colors
- ✓ Do prefer this representation



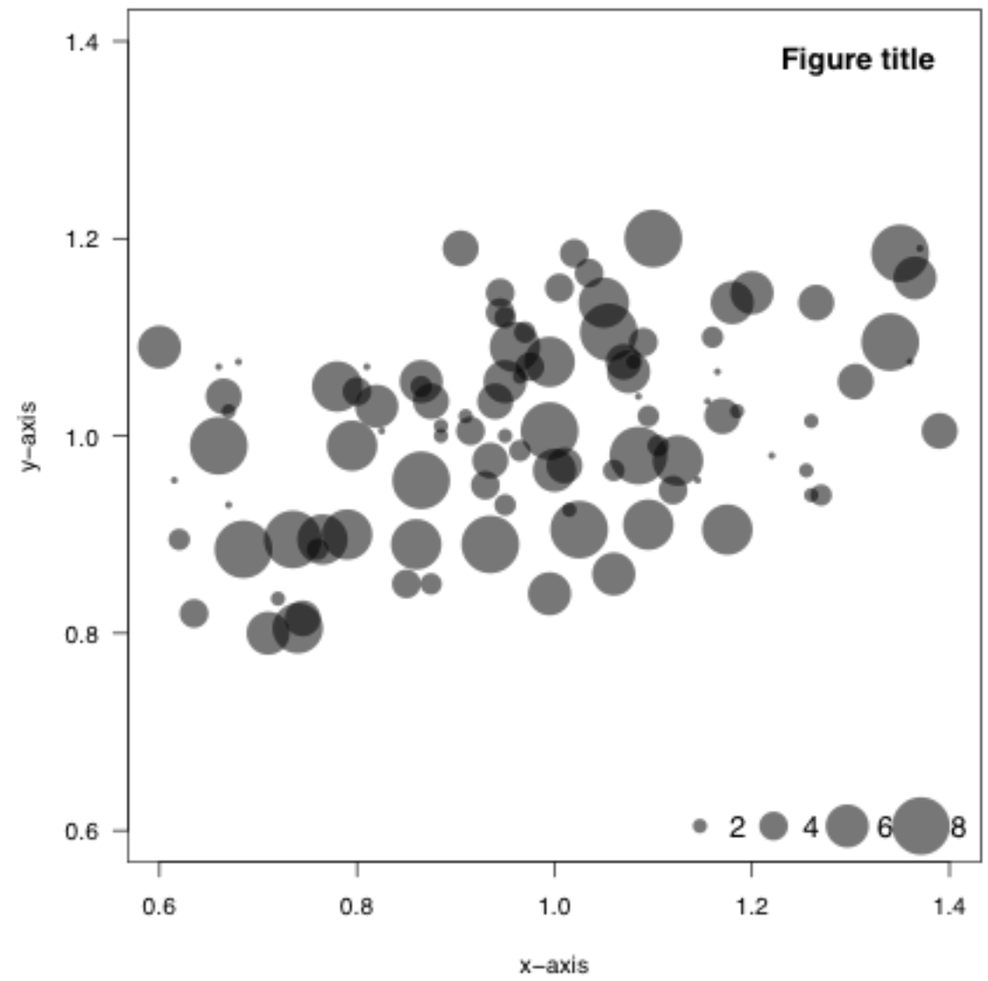
Some guidelines for better graphics

- ✓ Do not use more than 6 colors
- ✓ Do not use high contrast color



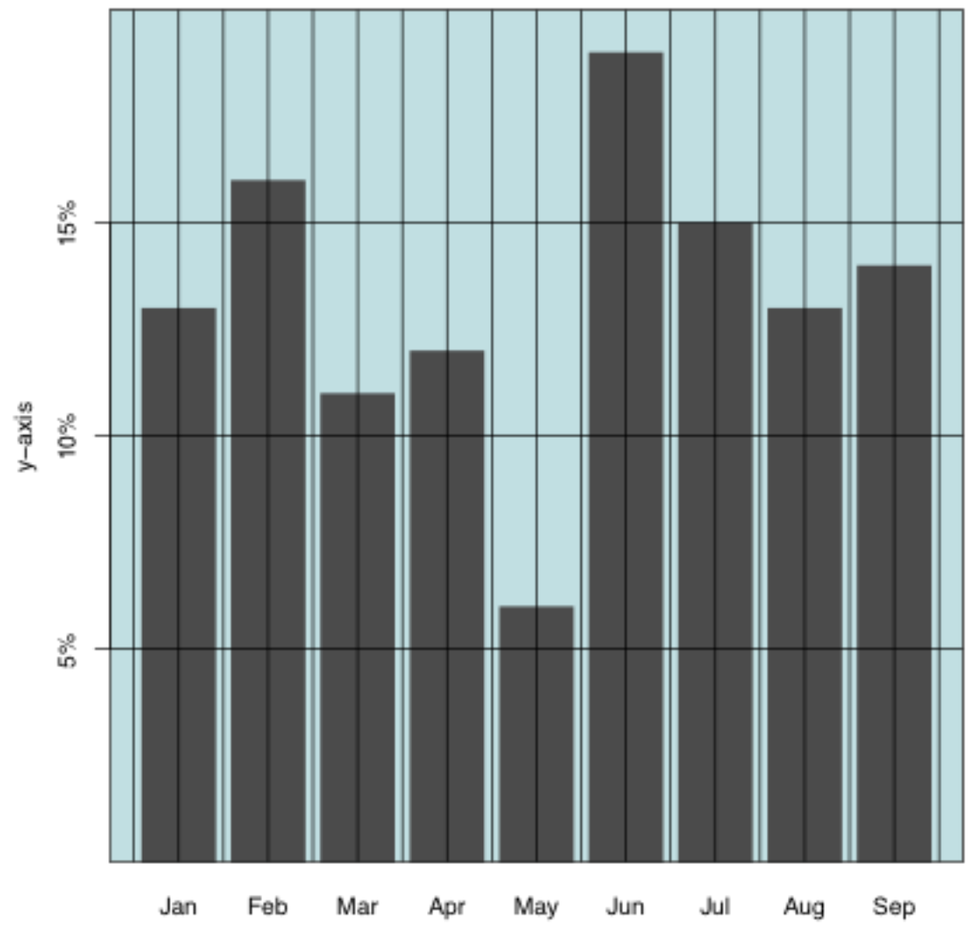
Some guidelines for better graphics

- ✓ Do not use more than 6 colors
- ✓ Do not use high contrast color
- ✓ Sometimes sizes and symbols are better



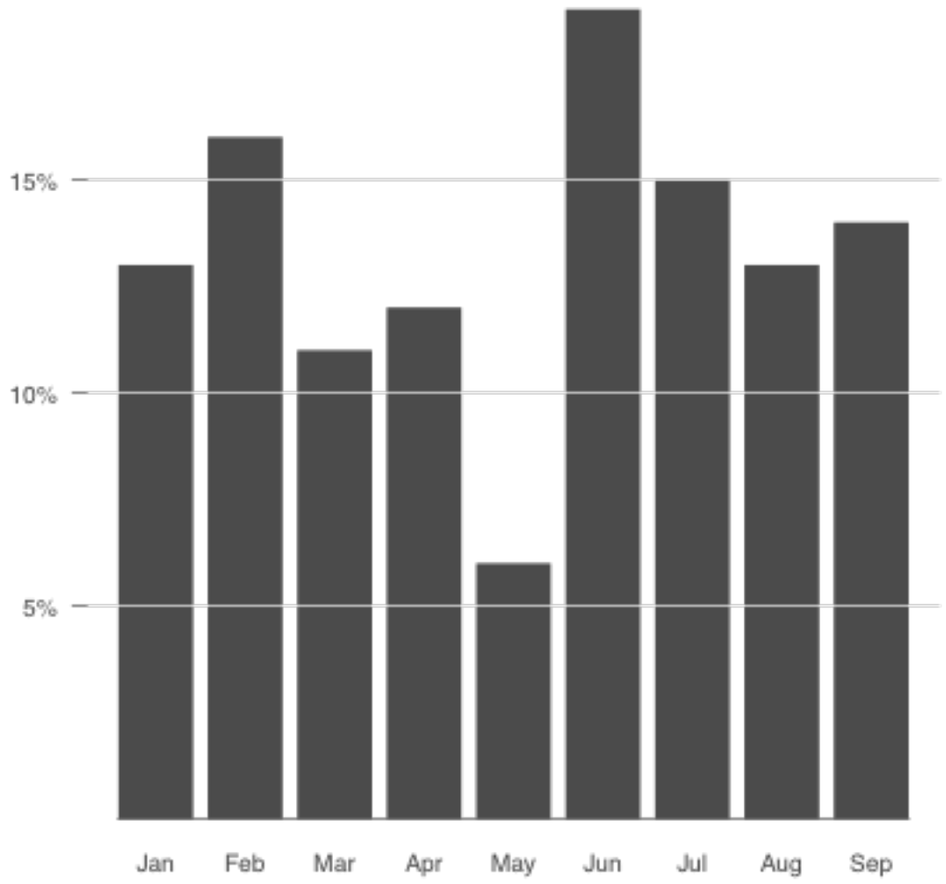
Some guidelines for better graphics

- ✓ Write textual informations horizontally
- ✓ Do not use distracting elements
- ✓ Do not add chart junk



Some guidelines for better graphics

- ✓ Write textual informations horizontally
- ✓ Do not use distracting elements
- ✓ Do not add chart junk
- ✓ Think about the Data-Ink ratio (Tufte, 1983)



Some guidelines for better graphics

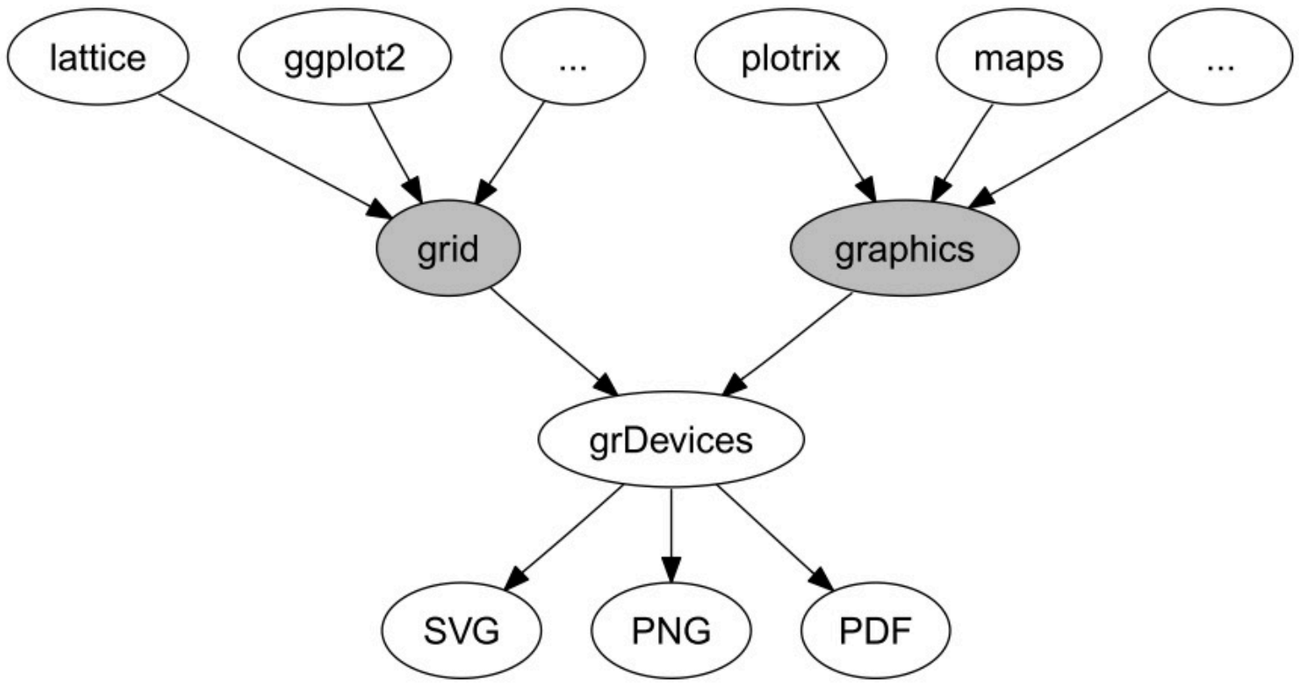
> Each element of a graph has to help understanding data

> Choose the graphic that shows what you want to show

The R system

- ✓ Software environment for statistical computing and graphics
- ✓ Open-source, free and multiplatform
- ✓ Widely used in the scientific community
- ✓ Programming language
- ✓ Implementation of the S programming language
- ✓ The core system is extended through user-created packages
- ✓ You can do what you want with R

The R system

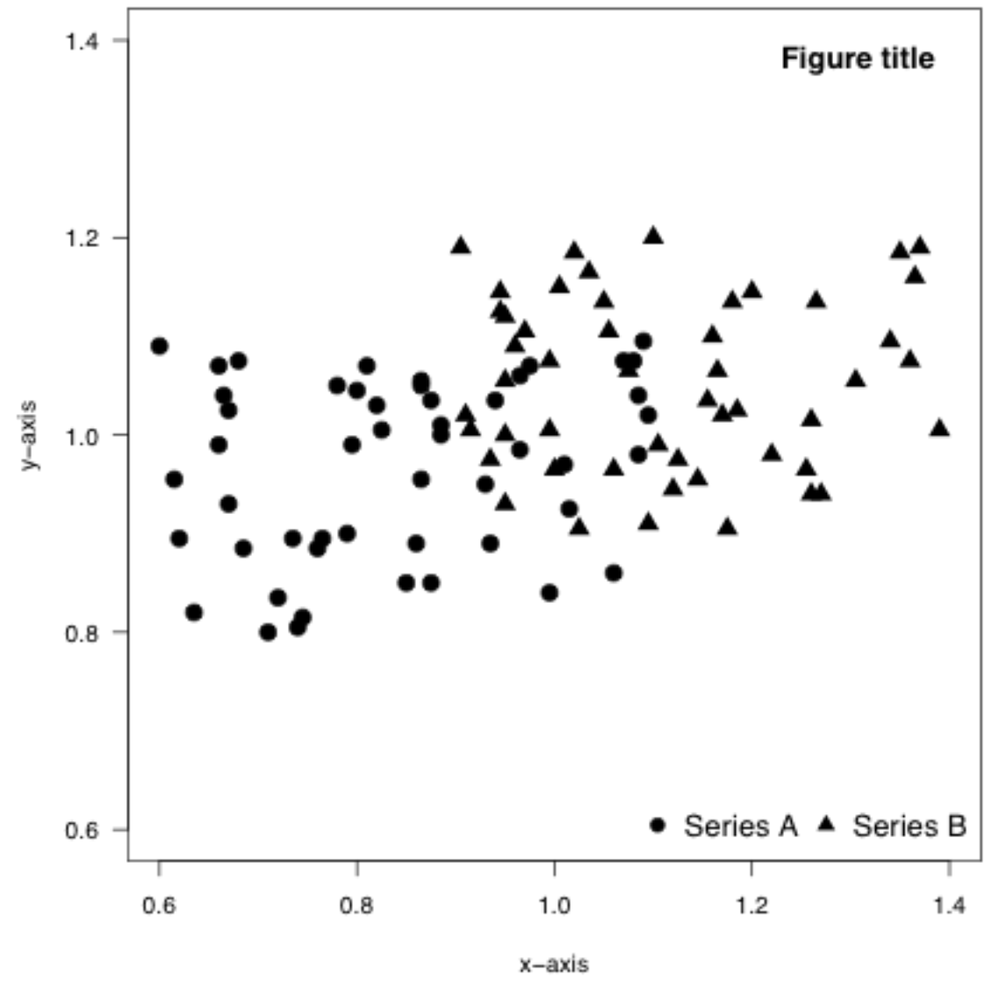


Murrell, P. (2015) The gridGraphics Package. The R Jounal.

Graphical packages - graphics

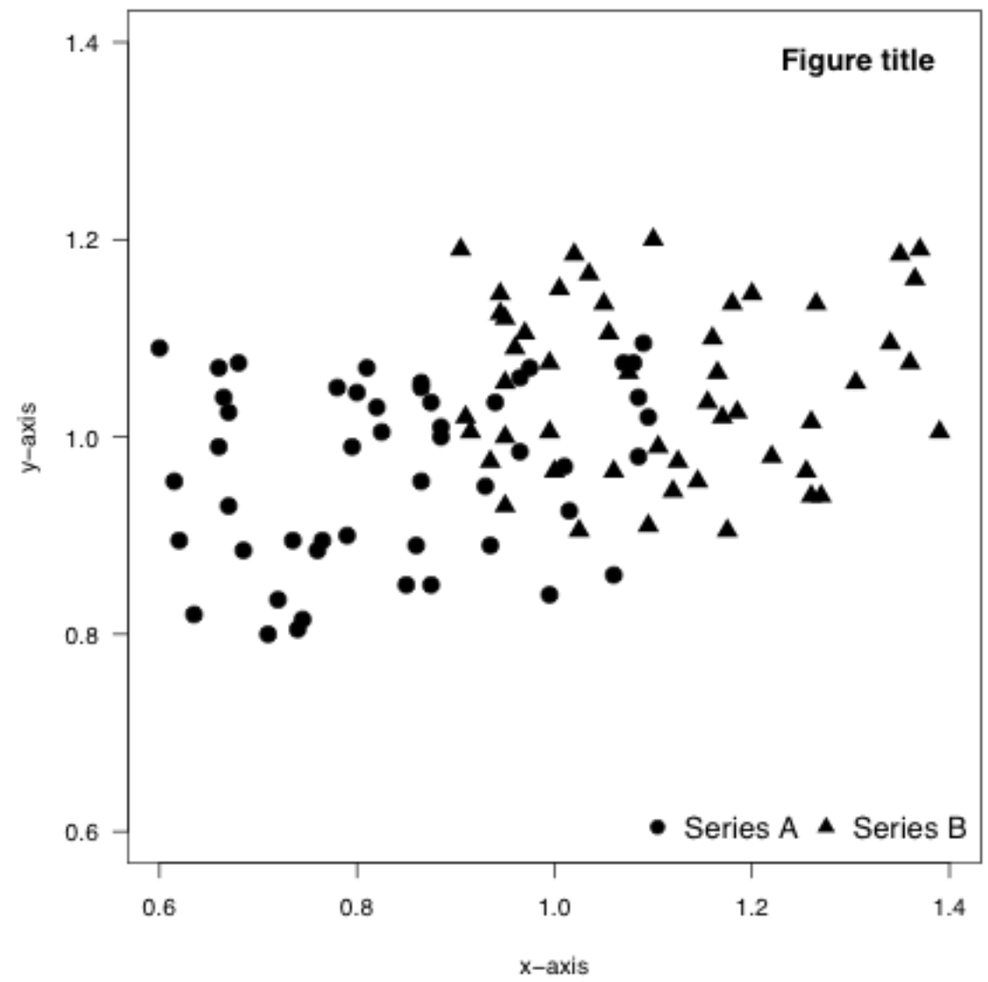
- ✓ Base package
- ✓ S-like plotting functions
- ✓ Contains the famous function `plot()`
- ✓ And a lot of well-known functions:
  `boxplot()`, `barplot()`, `hist()`,
  `lines()`, `points()`, `legend()`, etc.

```
plot(x, y, ...)
points(x, y, ...)
title(main, ...)
legend(legend, ...)
```



Graphical packages - grid

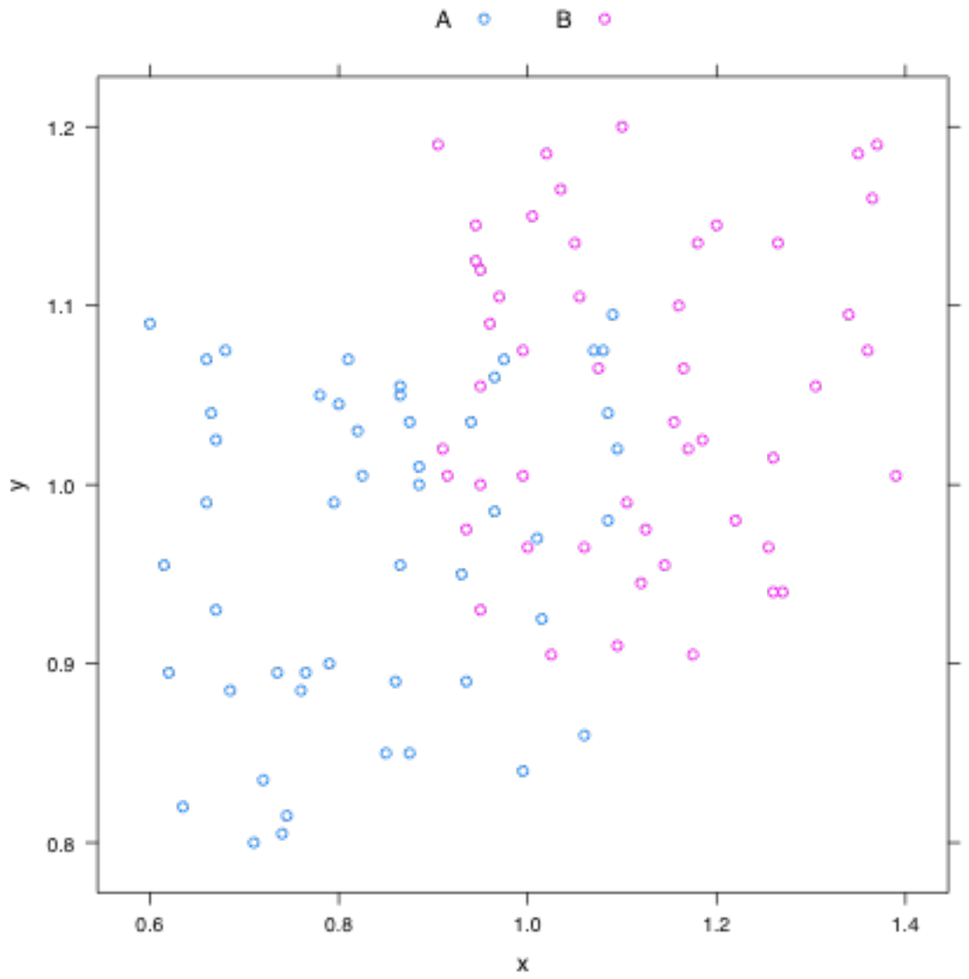
- ✓ An alternative set of graphical functions
- ✓ Well-suited for developers
- ✓ `ggplot2` is based on this package



Graphical packages - lattice

- ✓ Based on the `grid` package
- ✓ High-level system inspired by Trellis graphics
- ✓ Specialized on multivariate data
- ✓ and multipanel figures

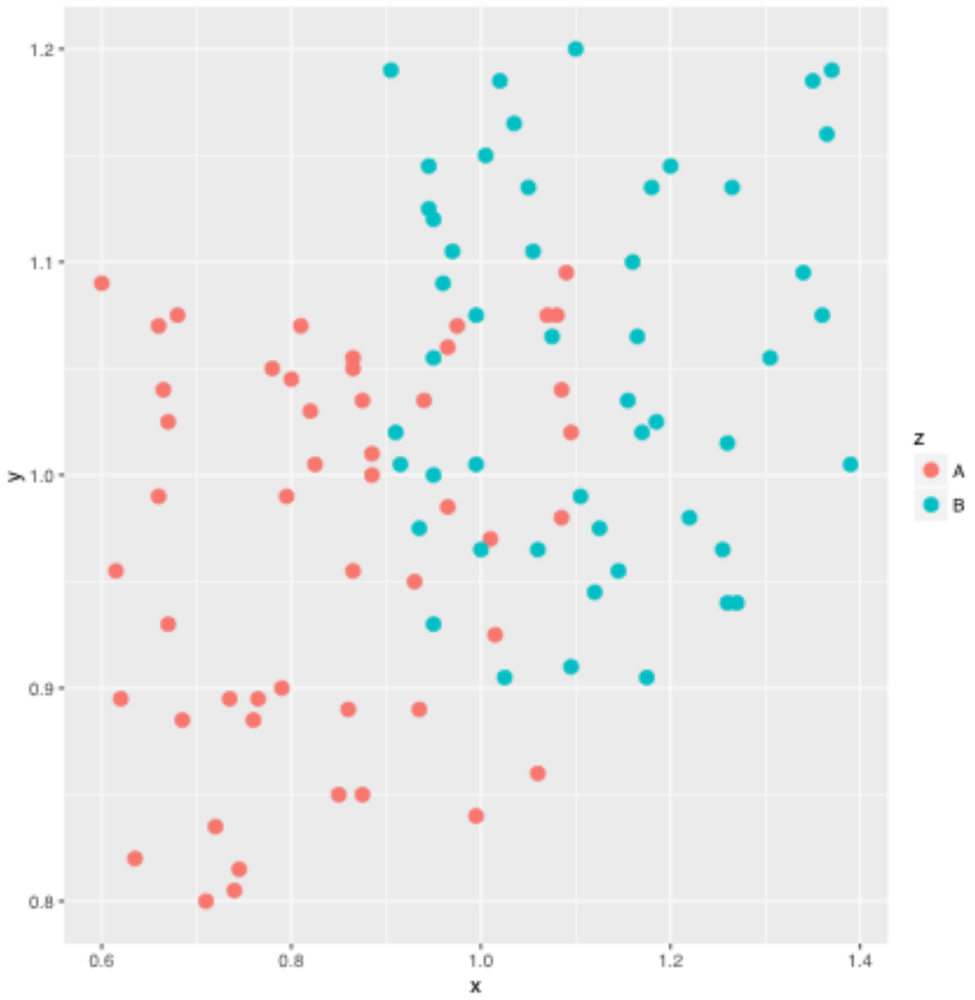
```
library(lattice)
xyplot(y ~ x, dat, group = z,
       auto.key = list(columns = 2))
```



Graphical packages - ggplot2

- ✓ Also based on the `grid` package
- ✓ A complete plotting system for R
- ✓ Based on the Grammar of Graphics
- ✓ But introduces its own syntax
- ✓ and requires a long time to master it

```
library(ggplot2)
p <- ggplot(data = dat,
            aes(x = x, y = y, colour = z))
p <- p + geom_point(size = 3)
p
```

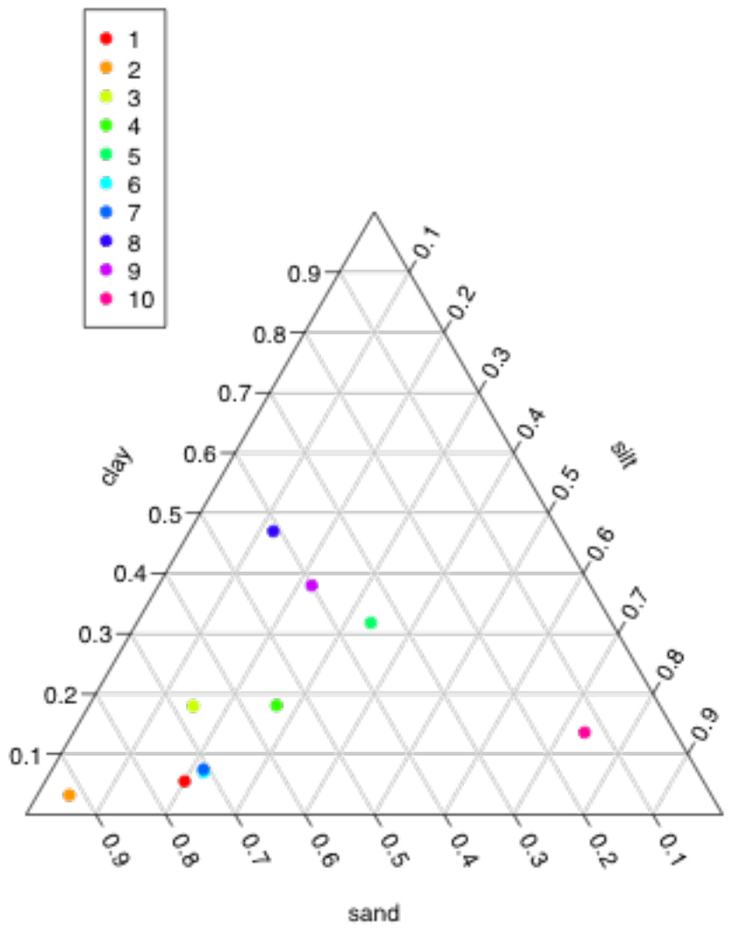


- ✓ See the QCBS workshop on ggplot2

Graphical packages - plotrix

- ✓ Based on the `graphics` package
- ✓ Contains a lots of specialized plots (i.e. polar plots)
- ✓ and various labeling, axis and color scaling functions

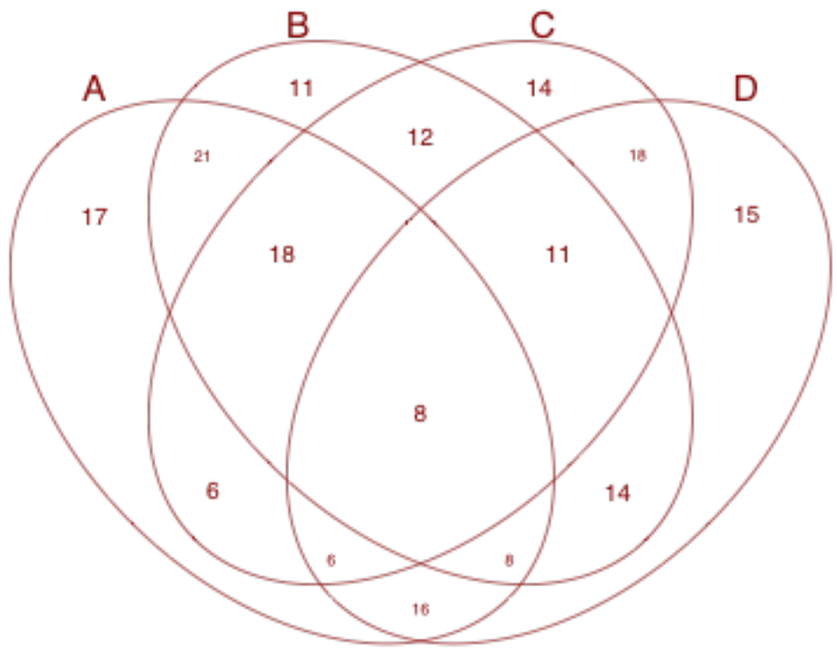
```
library(plotrix)
data(soils)
triax.plot(...)
```



Graphical packages - gplots

- ✓ Based on the `graphics` package
- ✓ Adds enhanced versions of standard plots (e.g. `boxplot2`)
- ✓ and some extra functions (e.g. Venn diagram)

```
library(gplots)
venn(...)
```



Graphical packages - others

- ✓ More than 80 others graphical packages
- ✓ For an overview see this R task view
- ✓ For a more exhaustive list see this post

- ✓ On this workshop we will only use the `graphics` package

Graphical parameters

- ✓ Default values of graphical parameters are stored in `par()`
- ✓ `par()` is an object
  - we can get the value of a parameter
- ✓ `par()` is also a function
  - we can change the default values

```
## How many graphical parameters?
length(par())
## [1] 72

## Let's get the default value of text color
par()$col
## [1] "black"

## Let's set 'red' for text color
par(col = 'red')

## Check
par()$col
## [1] "red"

## We're good!
```

Graphical parameters

- ✓ Important: when you change the value of one parameter, the new value affects all the graphs until the graphical window is closed

Graphical parameters

- ✓ Important: when you change the value of one parameter, the new value affects all the graphs until the graphical window is closed
- ✓ A recommendation:
  - Save the default par(): `opar <- par()`
  - Change the values: `par(col='red')`
  - Do the graph
  - Restaure the old par(): `par(opar)`

Graphical parameters

- ✓ Important: when you change the value of one parameter, the new value affects all the graphs until the graphical window is closed
- ✓ A recommendation:
  - Save the default par(): `opar <- par()`
  - Change the values: `par(col = 'red')`
  - Do the graph
  - Restaure the old par(): `par(opar)`
- ✓ Some graphical parameters can also be changed directly in plotting functions

High-level vs. low-level plotting functions

High-level plotting functions

- ✓ Open a new graphical window
- ✓ Or erase the content of the previous window
- ✓ Examples: `plot()`, `boxplot()`, `barplot()`, `hist()`, etc.

High-level vs. low-level plotting functions

High-level plotting functions

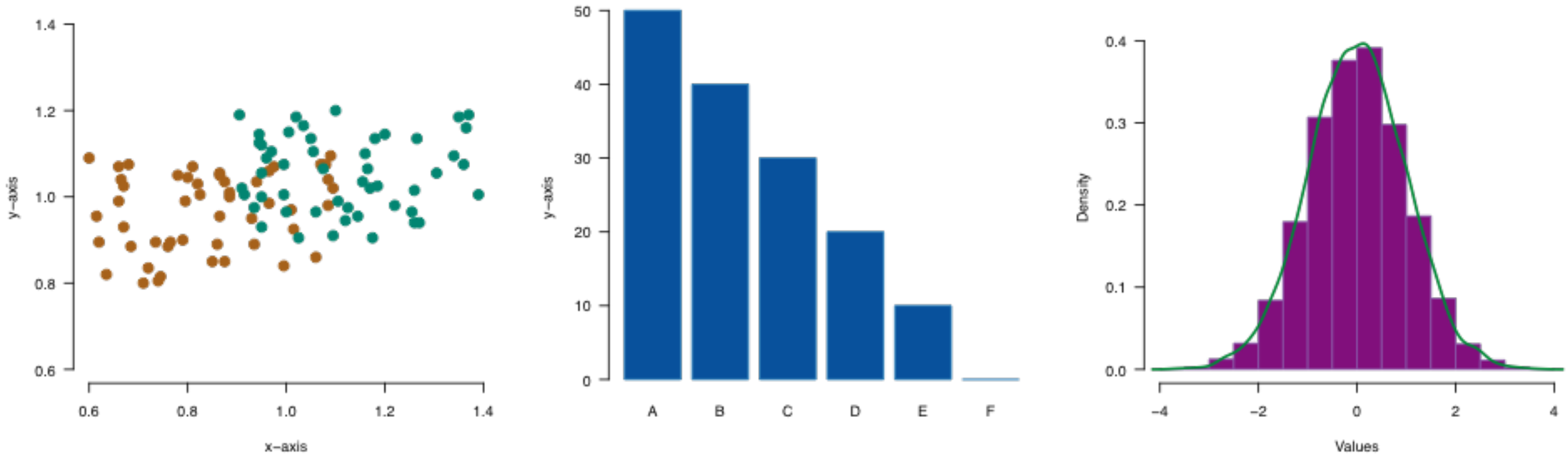
- ✓ Open a new graphical window
- ✓ Or erase the content of the previous window
- ✓ Examples: `plot()`, `boxplot()`, `barplot()`, `hist()`, etc.

Low-level plotting functions

- ✓ Work only when a graphical window is open
- ✓ Add content to the active window
- ✓ Examples: `lines()`, `points()`, `axis()`, `legend()`, etc.

High-level vs. low-level plotting functions

> You only need to know one high-level plotting function: `plot()`



Let's take a look at the data

- ✓ Random data with no particular sense
- ✓ Three variables:
  - x and y: quantitative variables
  - z: qualitative variable (factor)

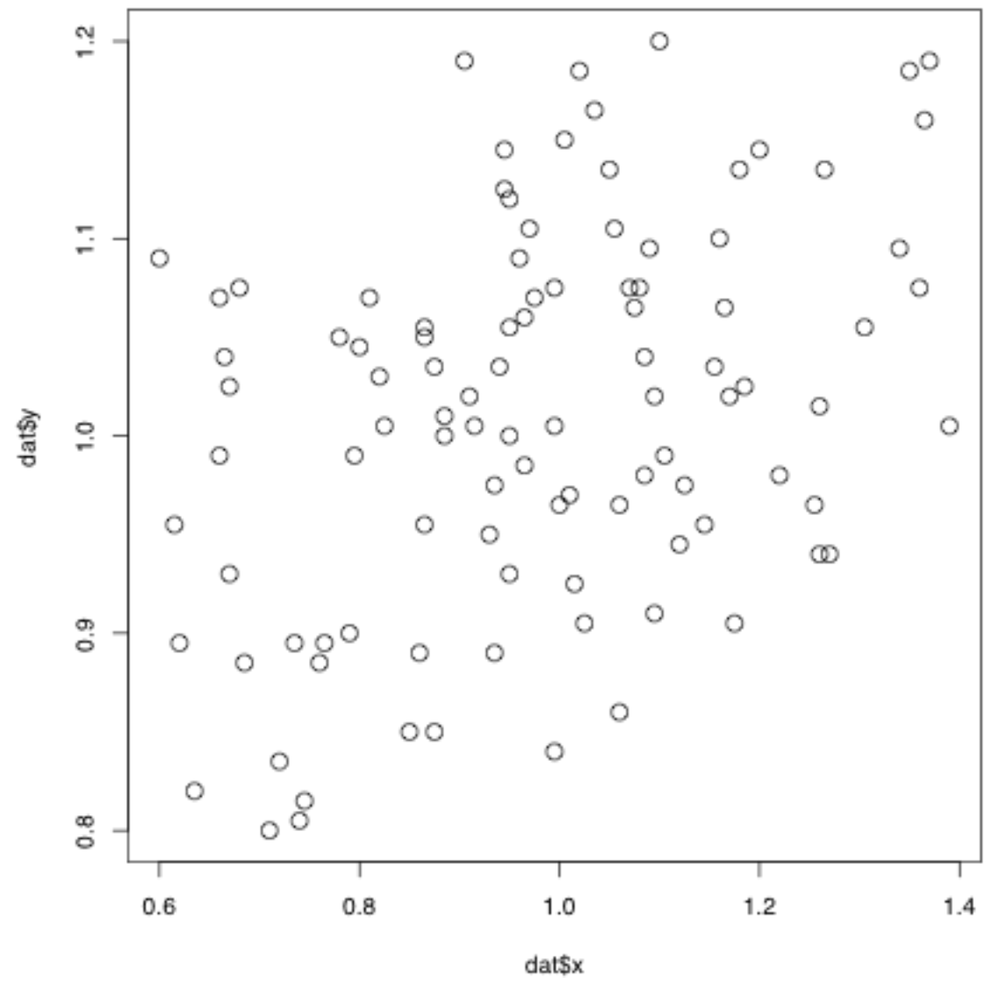
```
load('../data/xyz.RData')

head(dat)
##       x     y z
## 1 0.680 1.075 A
## 2 0.720 0.835 A
## 3 0.865 1.050 A
## 4 0.800 1.045 A
## 5 0.745 0.815 A
## 6 0.995 0.840 A

summary(dat$z)
##  A  B
## 50 50
```

An empty plot

- ✓ The default plot
- ✓ Quite ugly, isn't it?

```
plot(x = dat$x, y = dat$y)
```

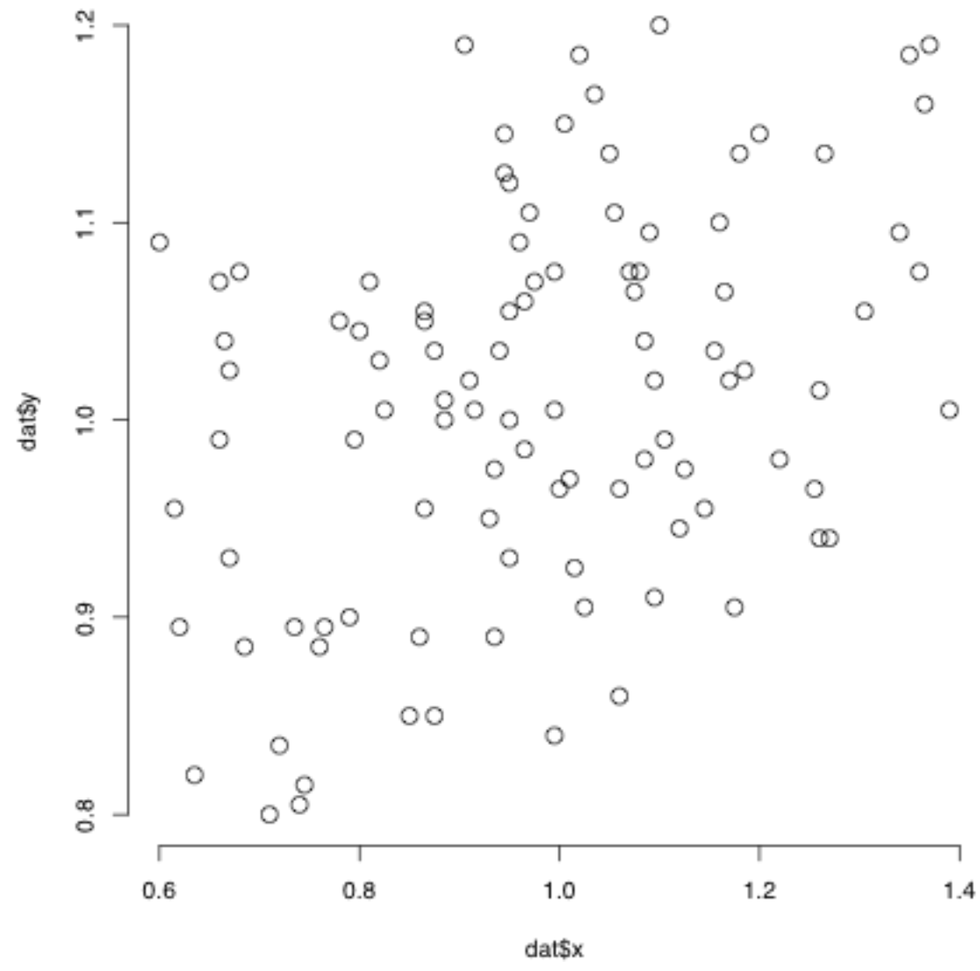
- ✓ Now we are going to remove each component of the graph to create an empty plot



An empty plot

- ✓ First let's remove the box
- ✓ with the argument `bty` (default: `'o'`)

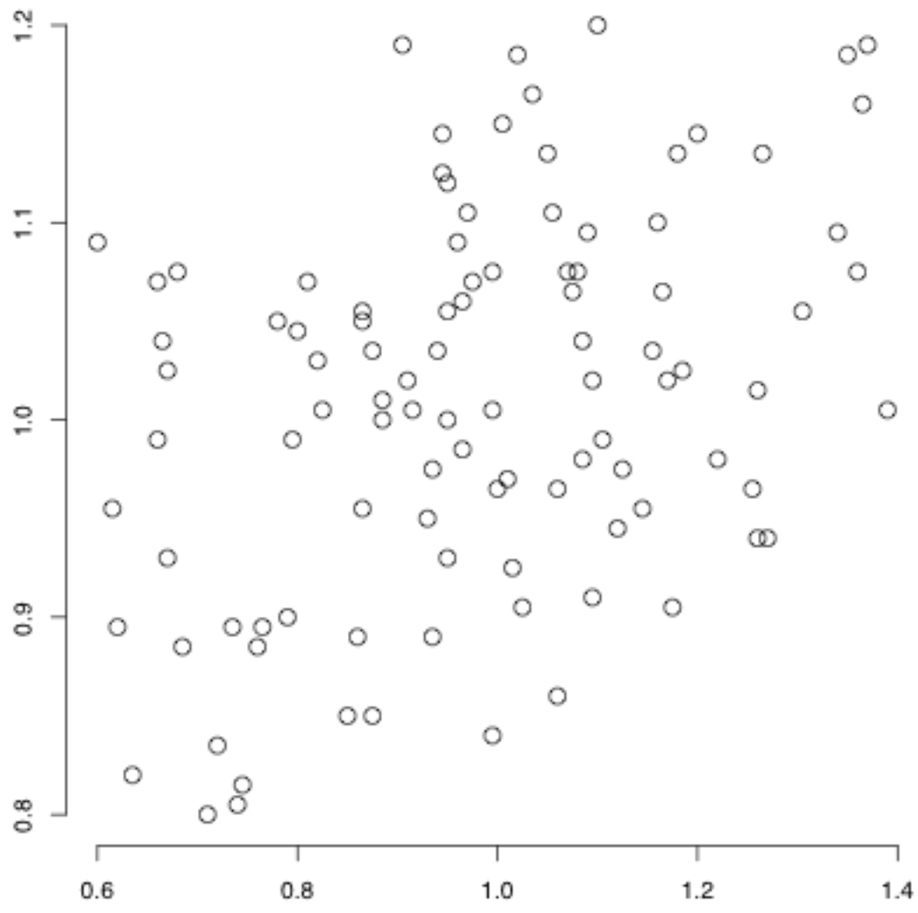
```
plot(x = dat$x, y = dat$y,
     bty = 'n')
```



An empty plot

- ✓ Now let's remove the textual annotation
- ✓ with the argument `ann` (default: `'TRUE'`)

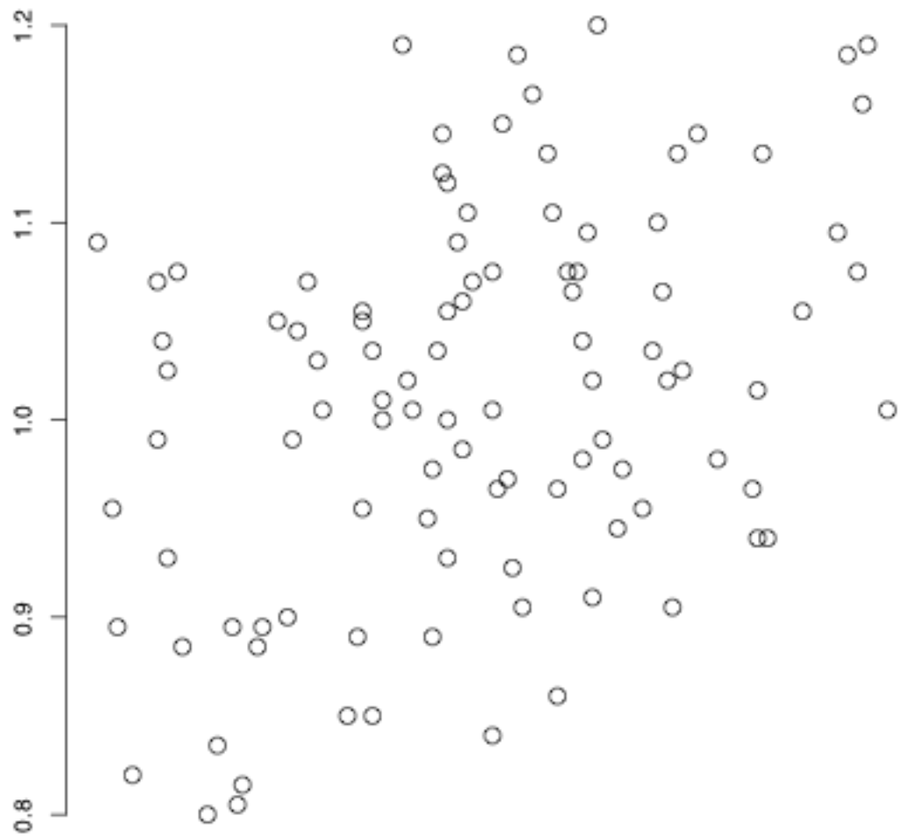
```
plot(x = dat$x, y = dat$y,
     bty = 'n',
     ann = FALSE)
```



An empty plot

- ✓ Let's remove the x-axis
- ✓ with the argument `xaxt` (default: `'s'`)

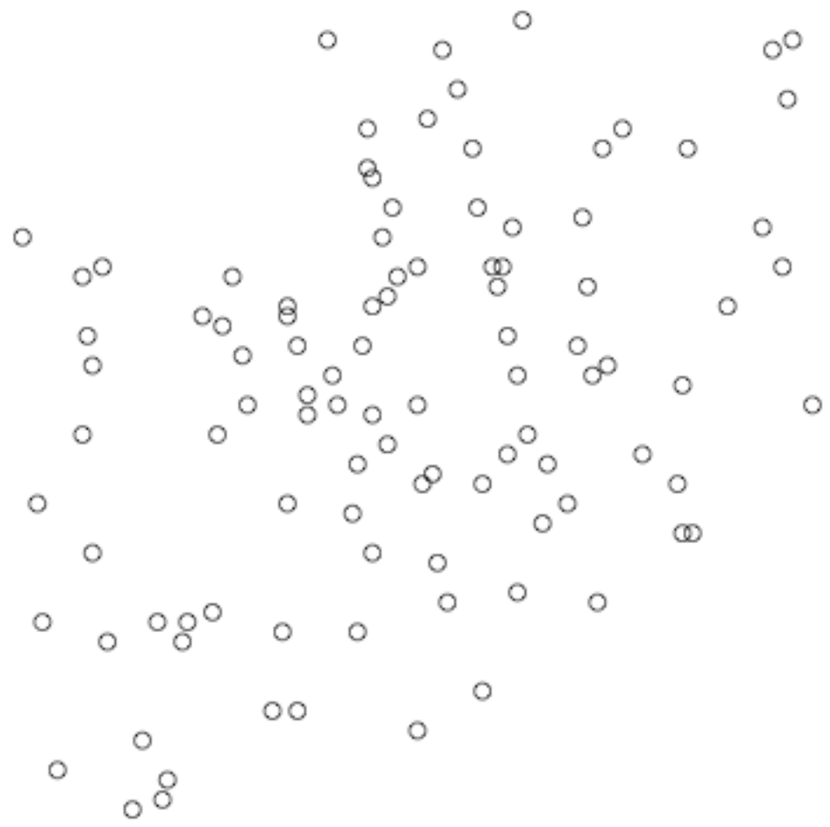
```
plot(x = dat$x, y = dat$y,
     bty = 'n',
     ann = FALSE,
     xaxt = 'n')
```



An empty plot

- ✓ And the y-axis
- ✓ with the argument `yaxt` (default: `'s'`)

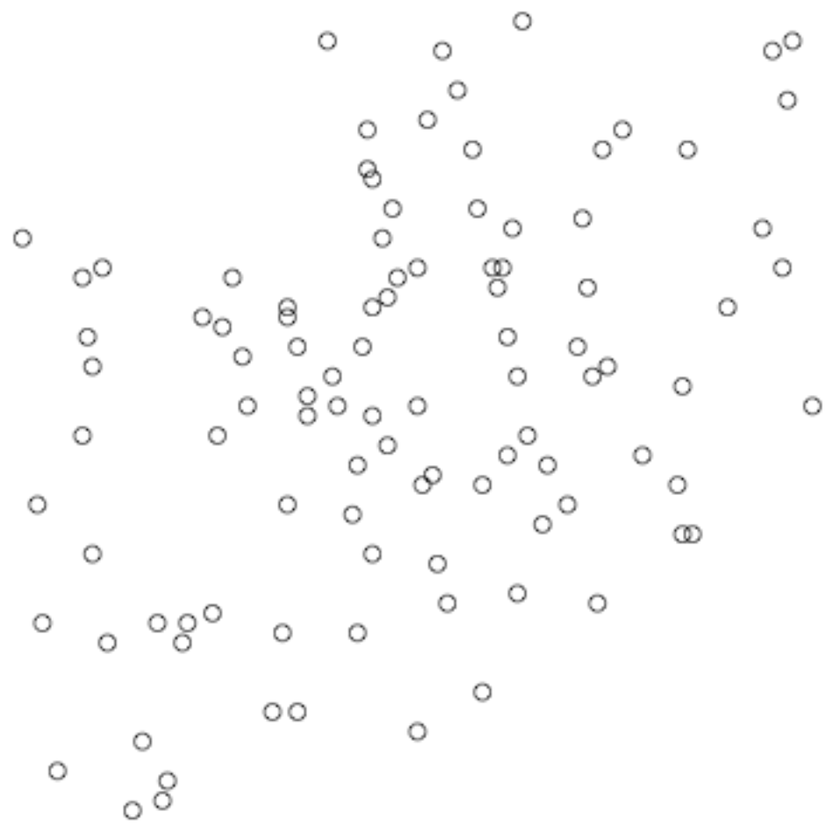
```
plot(x = dat$x, y = dat$y,
     bty = 'n',
     ann = FALSE,
     xaxt = 'n',
     yaxt = 'n')
```



An empty plot

- ✓ Using `axes=FALSE` is the same as:
- ✓ `bty='n'+xaxt='n'+yaxt='n'`

```
plot(x = dat$x, y = dat$y,
     ann = FALSE,
     axes = FALSE)
```



An empty plot

- ✓ Finally let's remove data
- ✓ with the argument `type` (default: `'p'`)

```
plot(x = dat$x, y = dat$y,
     ann = FALSE,
     axes = FALSE,
     type = 'n')
```

An empty plot

- ✓ Finally let's remove data
- ✓ with the argument `type` (default: `'p'`)

```
plot(x = dat$x, y = dat$y,
     ann = FALSE,
     axes = FALSE,
     type = 'n')
```

- ✓ In an empty plot, visual information is not displayed but the graph is defined in the window
- ✓ It is now possible to use low-level plotting functions such as `points()` or `axis()`

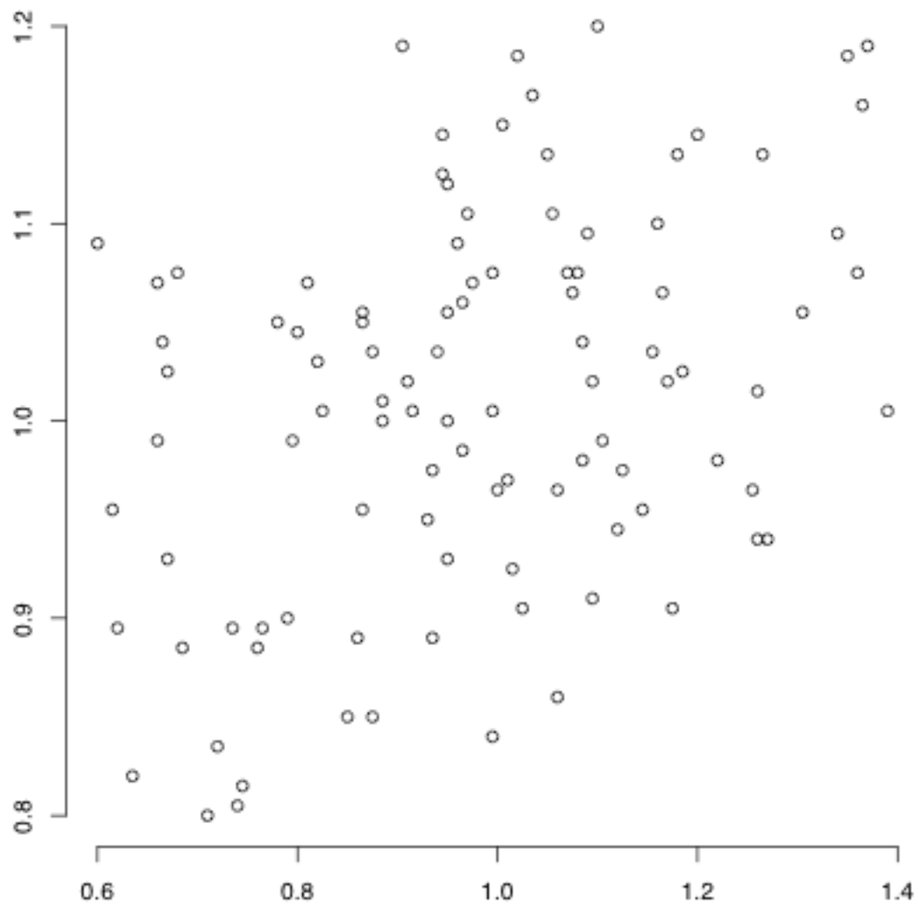
An empty plot, and now what?

- ✓ Now we've got an empty plot
- ✓ We are going to add some informations to:
    - discover useful low-level plotting functions,
    - improve the quality of the default plot
- ✓ Let's go!

Adding points

- ✓ We will use the function `points()`
- ✓ It shares some arguments with the function `plot()`

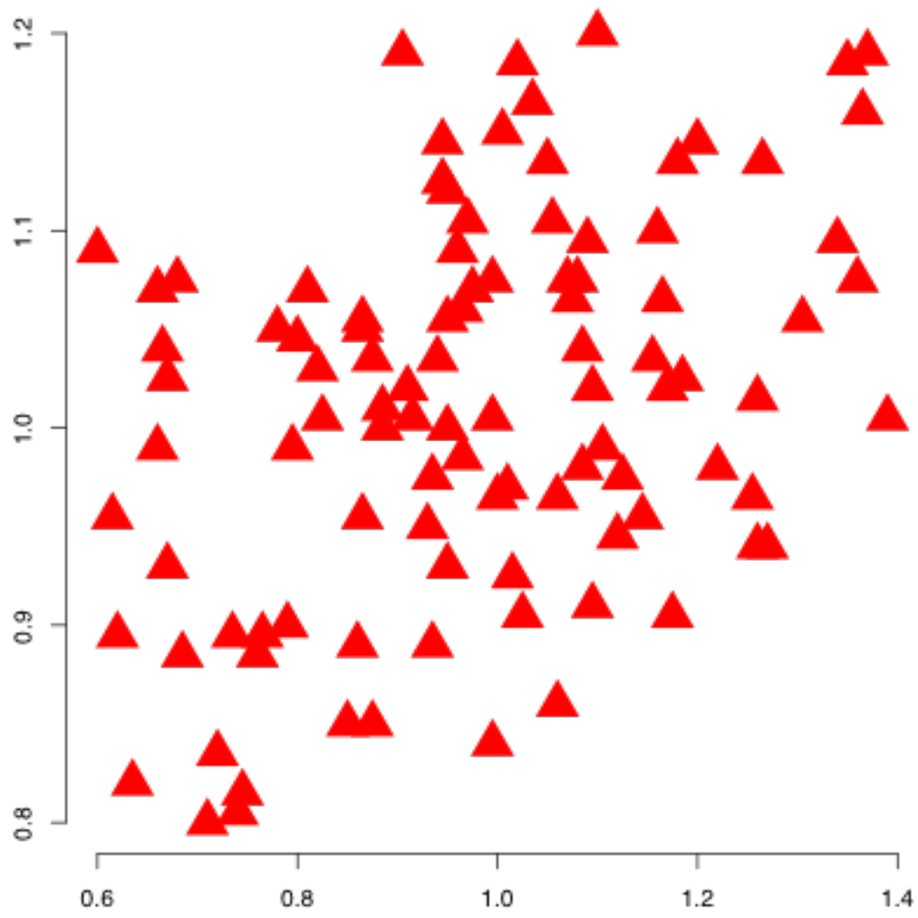
```
## Empty plot
plot(x = dat$x, y = dat$y, ann = FALSE,
     bty = 'n', type = 'n')

## Adding points (default settings)
points(x = dat$x, y = dat$y)
```



Adding points

- ✓ We can customize the points with:
  - `cex`, the size
  - `col`, the color
  - `pch`, the symbol

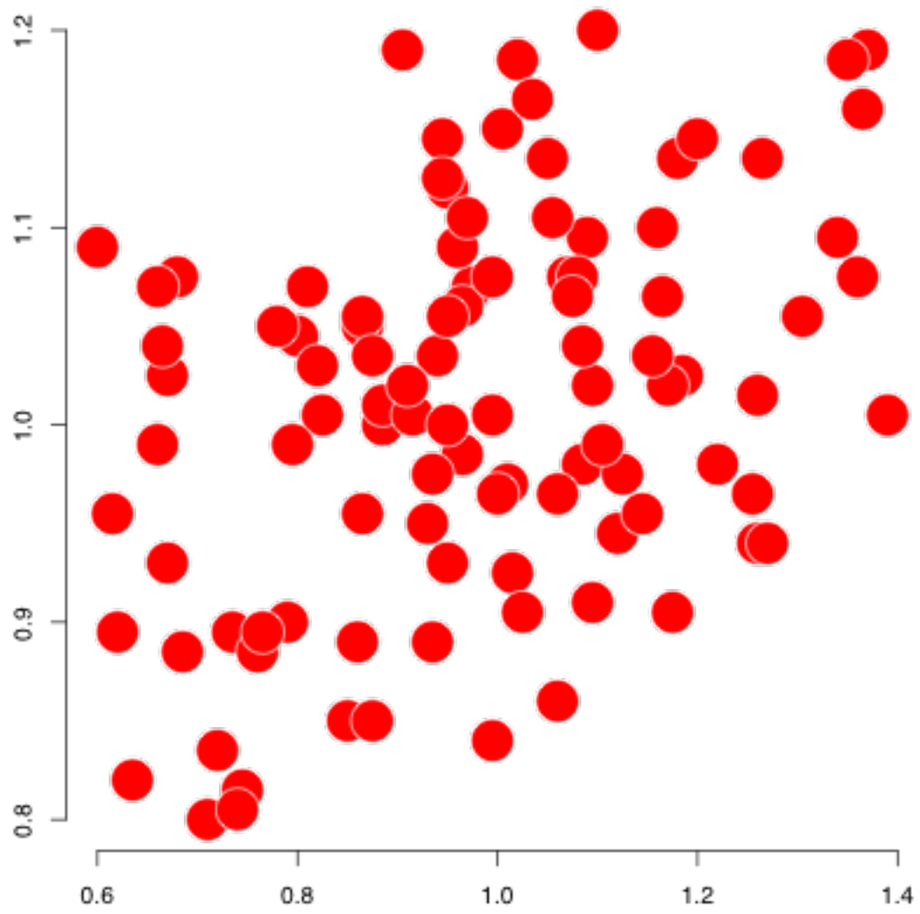
```
## Empty plot
plot(x = dat$x, y = dat$y, ann = FALSE,
     bty = 'n', type = 'n')

## Adding points (user settings)
points(x = dat$x, y = dat$y,
       cex = 3, col = 'red', pch = 17)
```



Adding points

- ✓ Some symbols have two colors:
  - `col`: the border color,
  - `bg`: the background color
- ✓ This is the case for `pch` = 21 to 25

```
## Empty plot
plot(x = dat$x, y = dat$y, ann = FALSE,
     bty = 'n', type = 'n')

## Adding points (user settings)
points(x = dat$x, y = dat$y,
       cex = 4, pch = 21,
       col = 'white', bg = 'red')
```



Adding lines

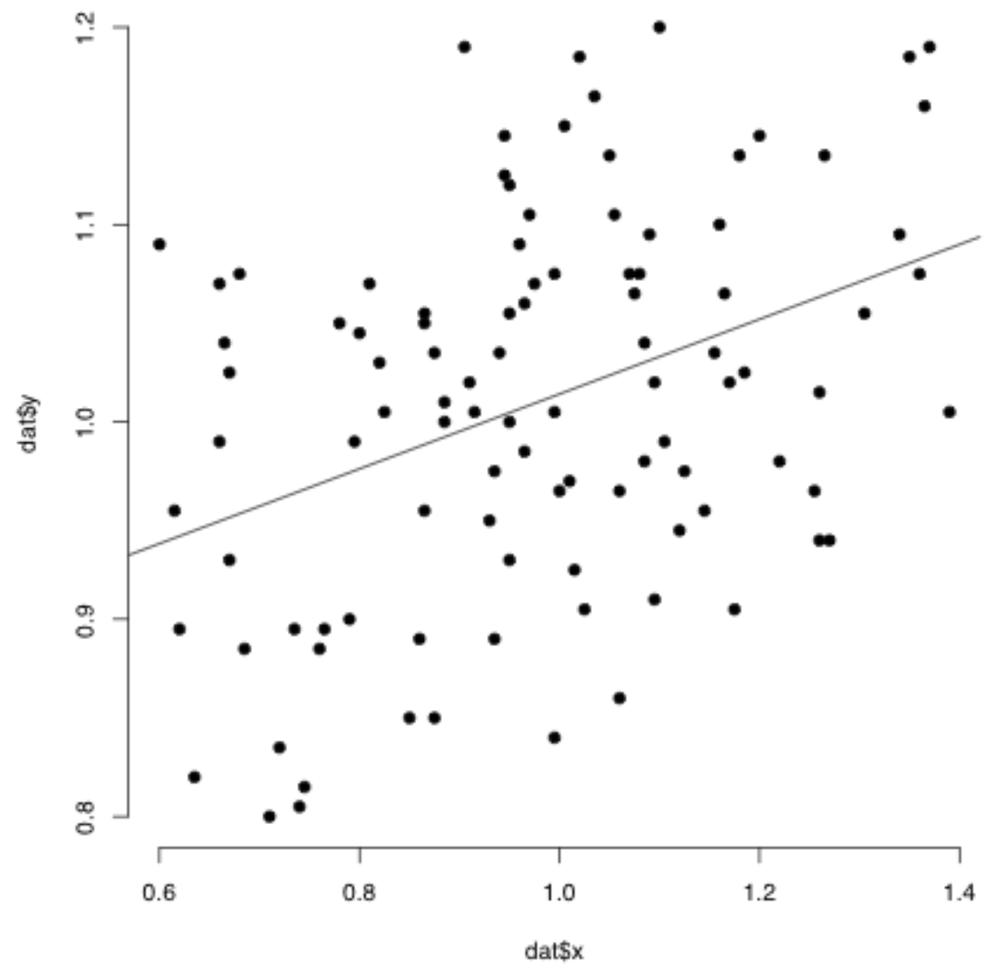
- ✓ Four functions allow to plot lines:
  - `points()`
  - `lines()`
  - `abline()`
  - `segments()`
- ✓ We will illustrate these functions with a linear regression

```
## Linear regression
(mod <- lm(y ~ x, data = dat))
##
## Call:
## lm(formula = y ~ x, data = dat)
##
## Coefficients:
## (Intercept)            x
##      0.8246       0.1895

(a <- coefficients(mod)[1])
## (Intercept)
##   0.8246096

(b <- coefficients(mod)[2])
##         x
## 0.1894751
```

Adding lines

- ✓ First, let's try the function `abline()`
- ✓ with the first way

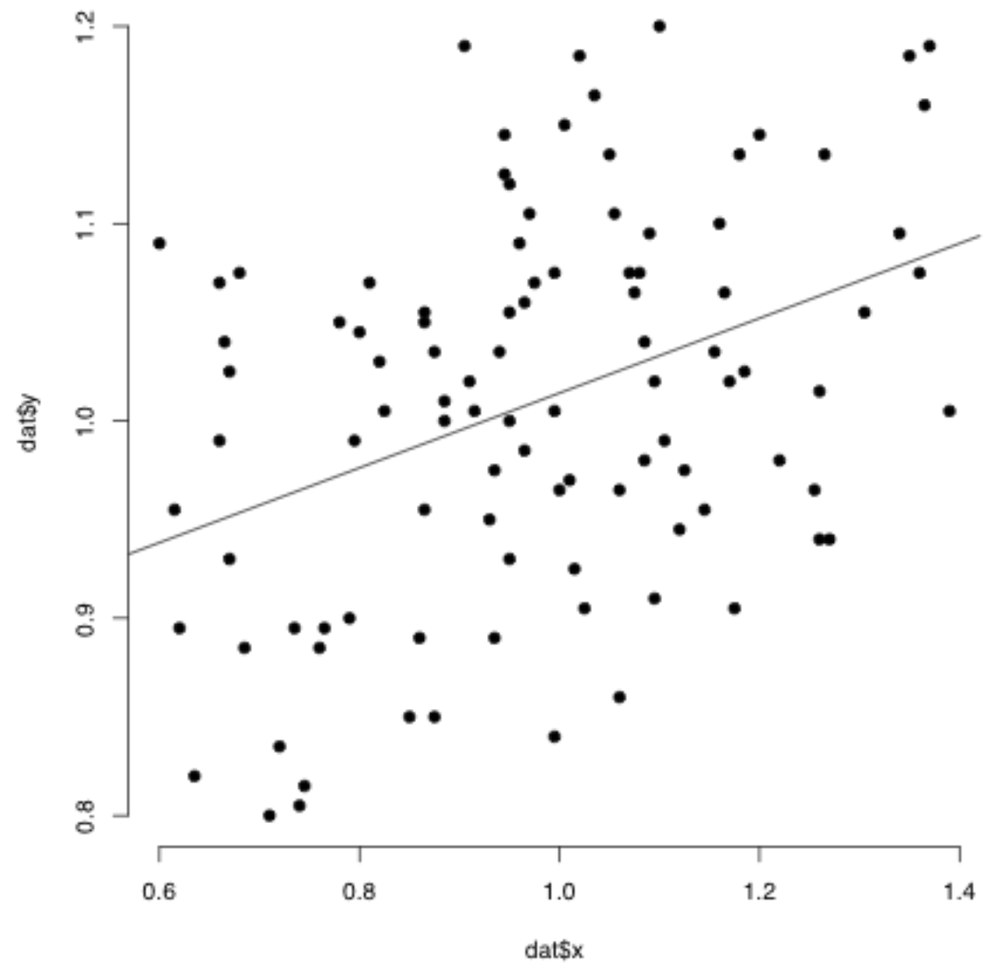
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding line (default settings)
abline(reg = mod)
```



Adding lines

- ✓ The second way is to specify model parameters

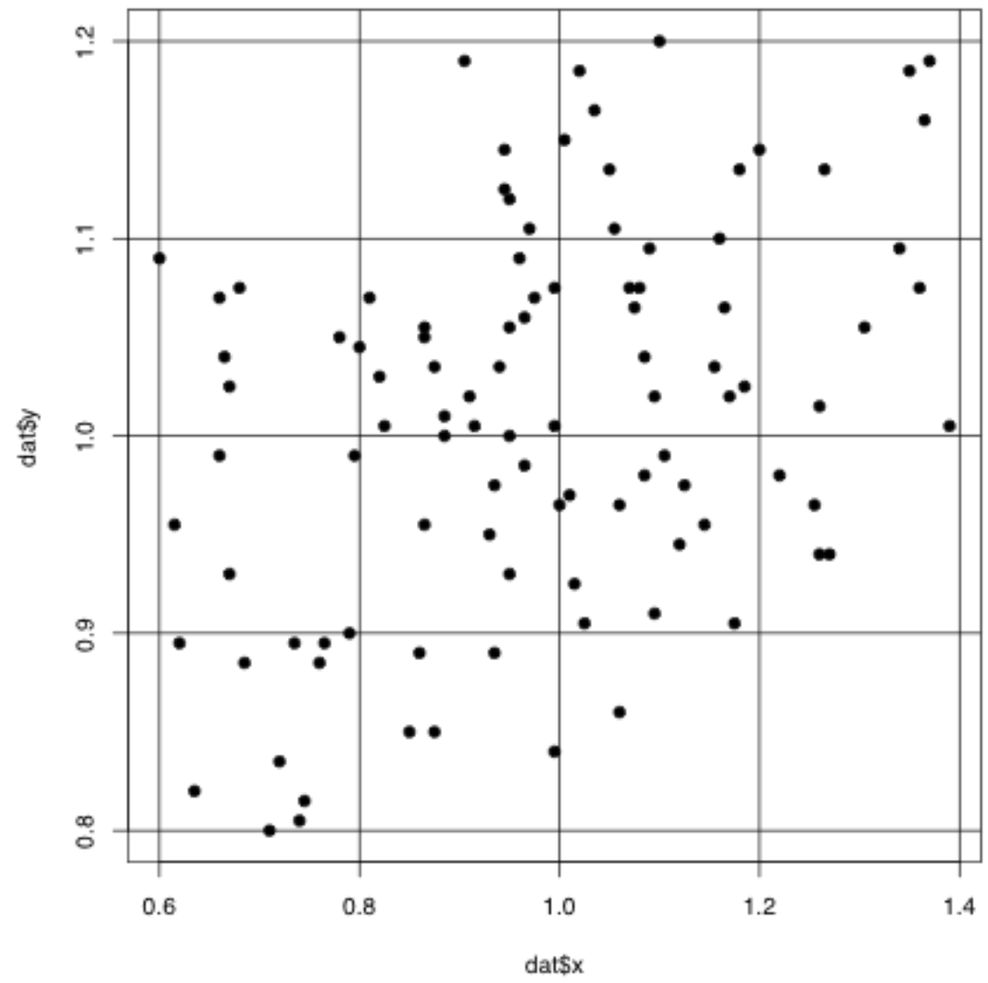
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding line (default settings)
abline(a = a, b = b)
```



Adding lines

✓ The function `abline()` allows to draw horizontal and vertical lines

```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'o',
     type = 'p', pch = 19)

## Adding line (default settings)
abline(h = 0.8)
abline(h = seq(0.9, 1.2, by = 0.1))
abline(v = seq(0.6, 1.4, by = 0.2))
```



Adding lines

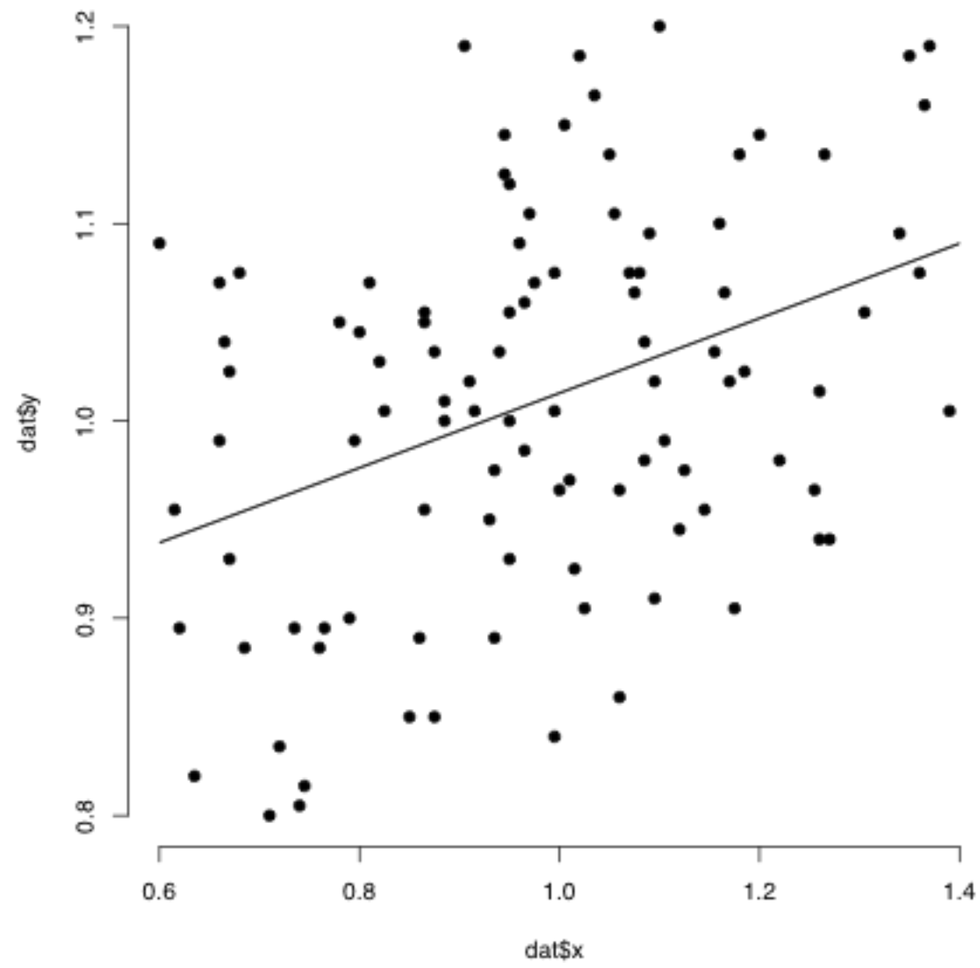
- ✓ Now take a look at the functions `lines()` and `points()`
- ✓ But first, we are going to predict the model on new data

```
## New data frame
mat <- data.frame(x = seq(0.6, 1.4, by = 0.05))

## Model prediction
ypred <- predict(object = mod, newdata = mat)
```

Adding lines

- ✓ Let's add model regression with the functions `lines()` and `points()`

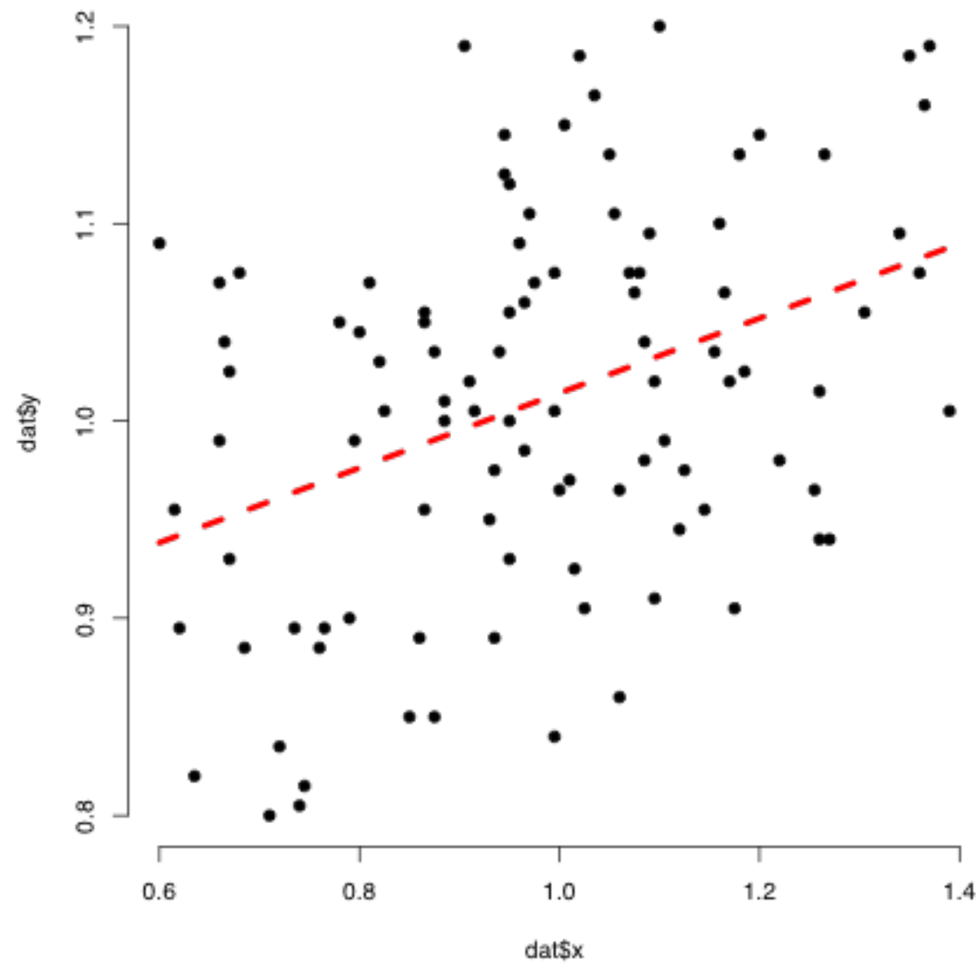
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding line (default settings)
lines(x = mat$x, y = ypred)

## Or, with the function points()
points(x = mat$x, y = ypred, type = 'l')
```



Adding lines

- ✓ We can customize the lines with:
  - `lwd`, the line width
  - `col`, the line color
  - `lty`, the line type

```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding line (user settings)
lines(x = mat$x, y = ypred,
      col = 'red', lwd = 4, lty = 2)
```



Adding polygons

- ✓ To add a polygon, the function is `polygon()`
- ✓ A special, the rectangle can be drawn with `rect()`
- ✓ Let's predict again the model, but this time with the standard error

```
## Model prediction with se
ypred <- predict(object = mod, newdata = mat,
                 se.fit = TRUE)

class(ypred)
## [1] "list"

names(ypred)[1:2]
## [1] "fit"    "se.fit"
```

Adding polygons

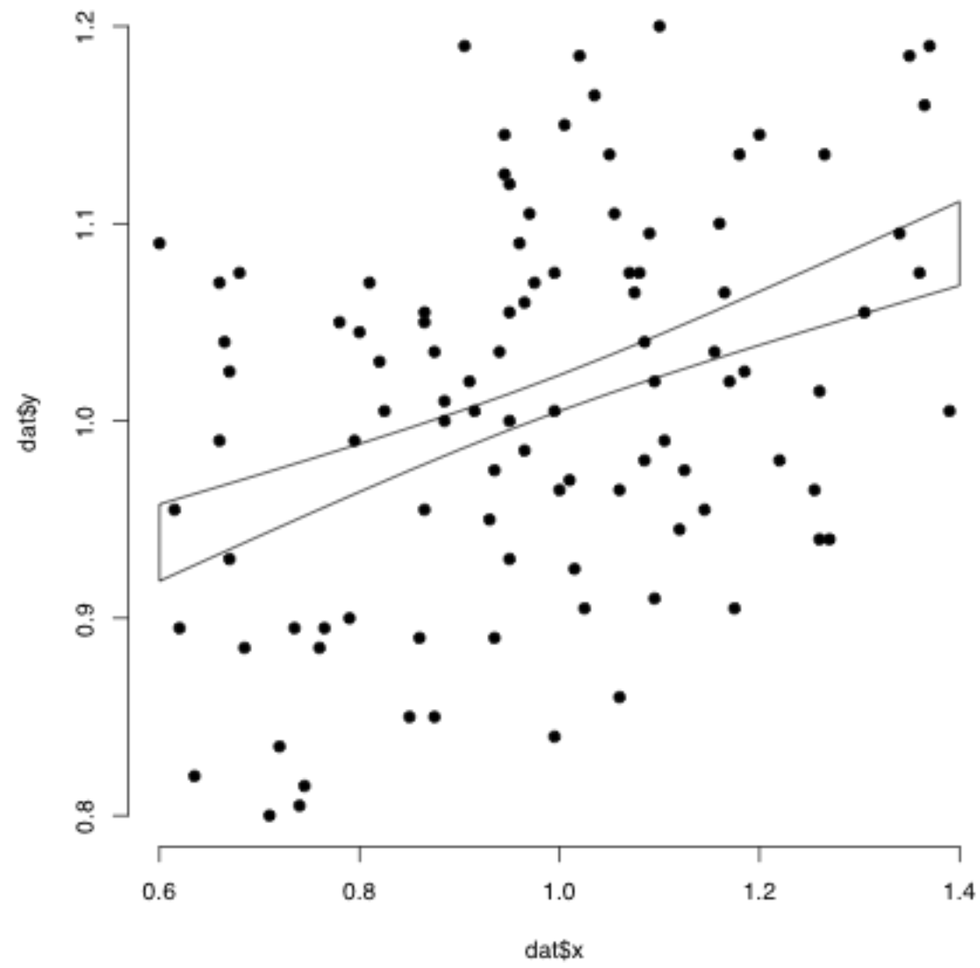
- ✓ We are going to add the error envelope with the function `polygon()`
- ✓ So, let's calculate the coordinates of this envelope

```
## Superior interval
xsup <- mat[ , 'x']
ysup <- ypred$fit + ypred$se.fit

## Inferior interval
xinf <- mat[ , 'x']
yinf <- ypred$fit - ypred$se.fit

## Reverse sort of inf.
xinf <- mat[nrow(mat) : 1, 'x']
yinf <- yinf[length(yinf) : 1]
```

Adding polygons

- ✓ Let's add model error envelope

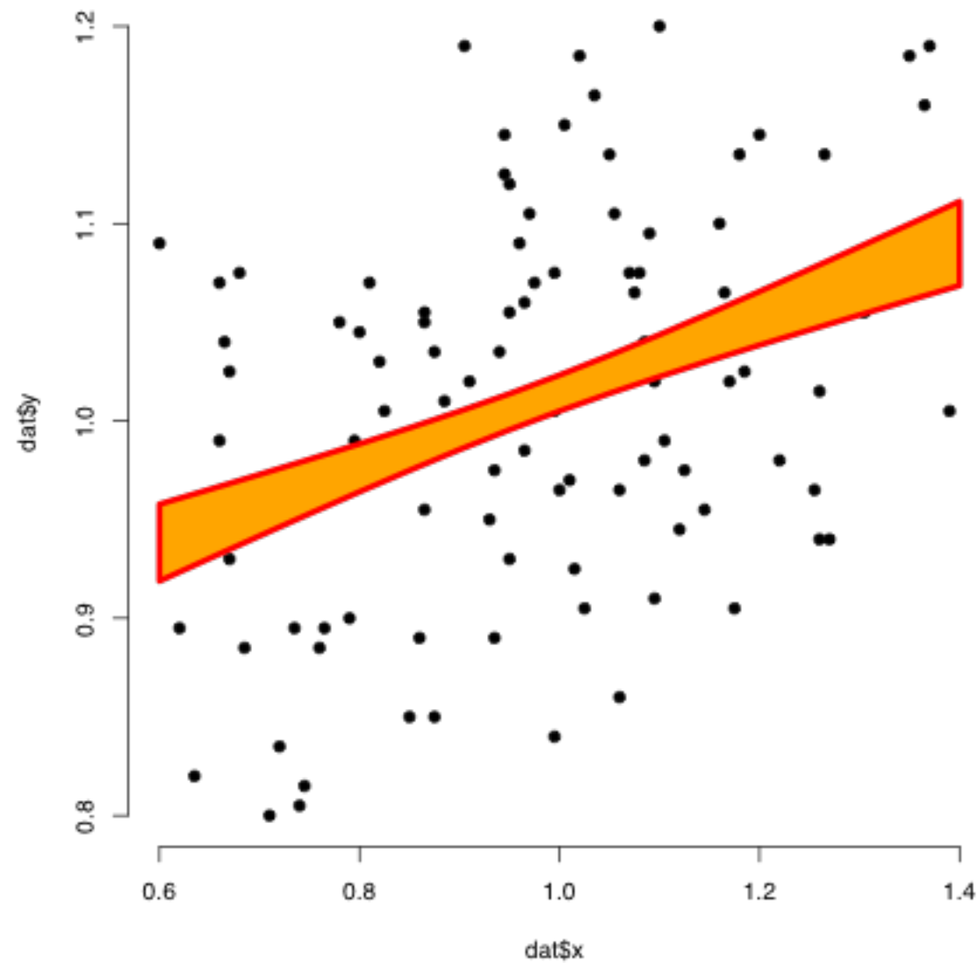
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding error envelope
polygons(x = c(xsup, xinf), y = c(ysup, yinf))
```



Adding polygons

- ✓ We can customize the polygon with:
  - `border`, the border color
  - `col`, the background color
  - `lwd`, the border width

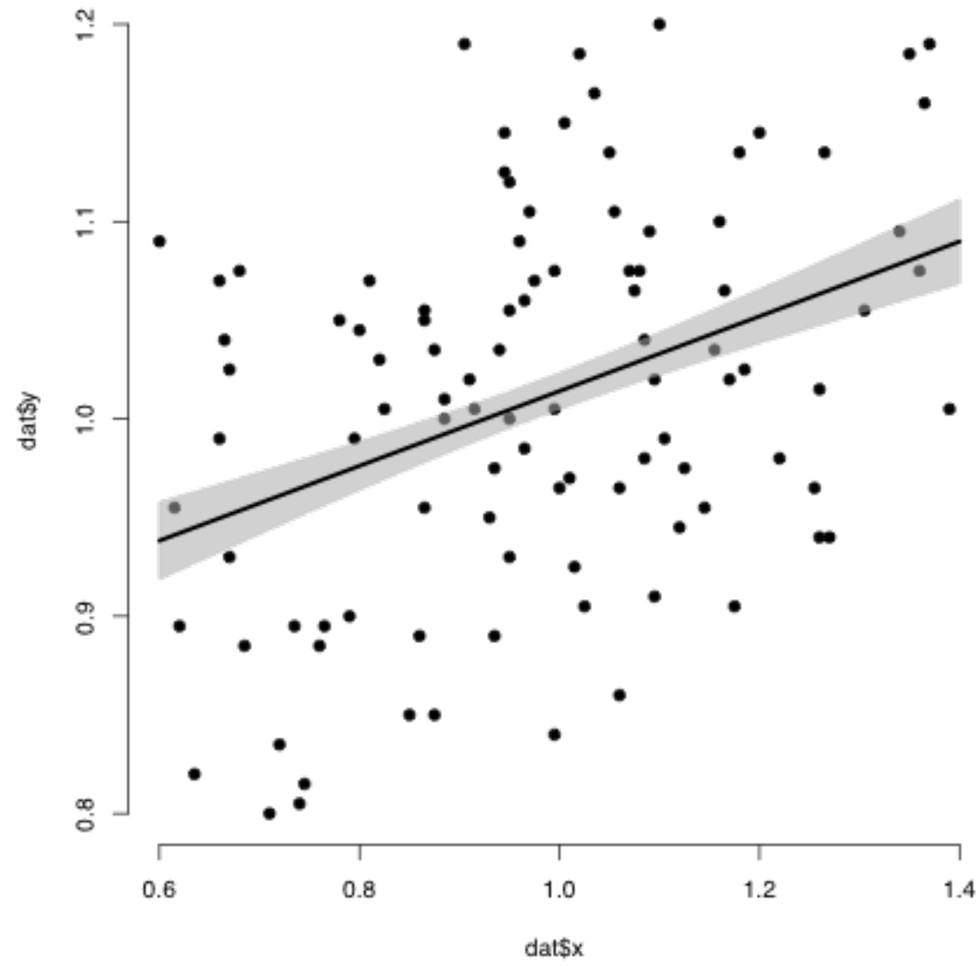
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding error envelope
polygon(x = c(xsup, xinf), y = c(ysup, yinf),
        col = 'orange', border = 'red',
        lwd = 4)
```



Adding polygons

✓ Finally

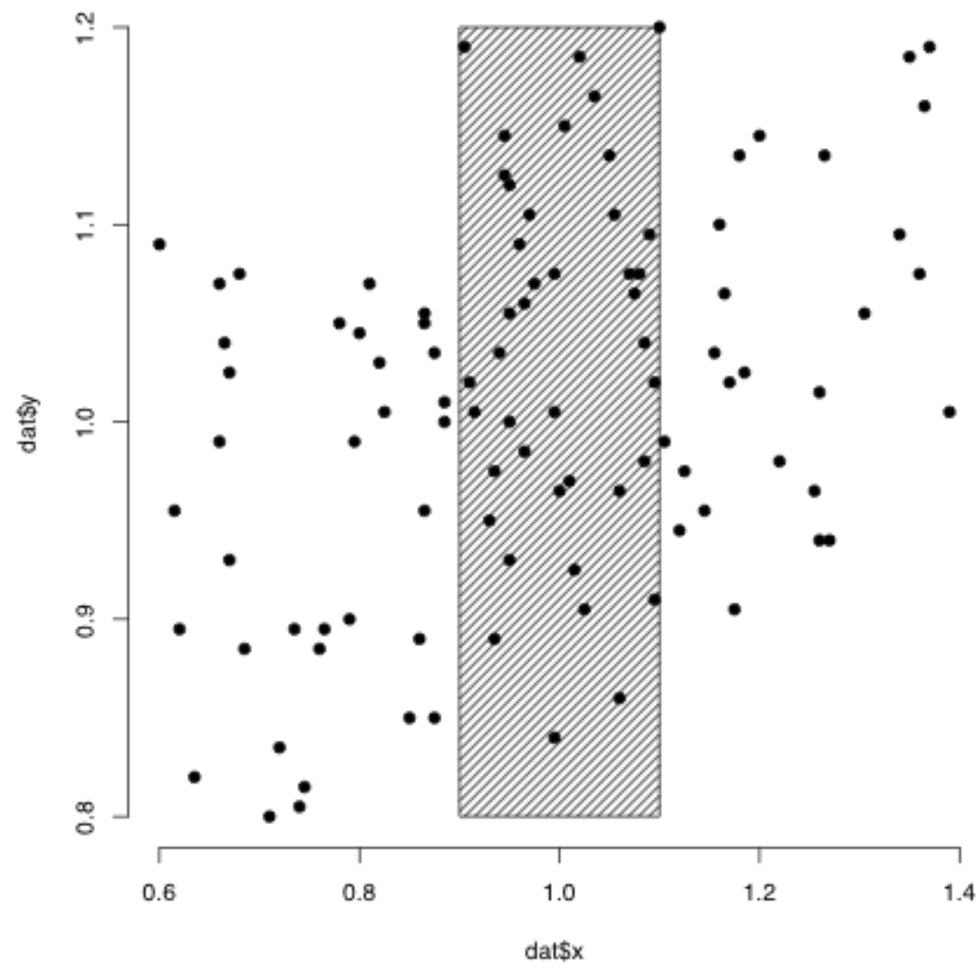
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding error envelope
polygon(x = c(xsup, xinf), y = c(ysup, yinf),
        col = '#aaaaaa88', border = '#aaaaaa88')

## Adding model regression
lines(x = mat$x, y = ypred$fit, lwd = 3)
```



Adding polygons

- ✓ The function `rect()` is appropiated when you want to add draw rectangle
- ✓ Here is an example

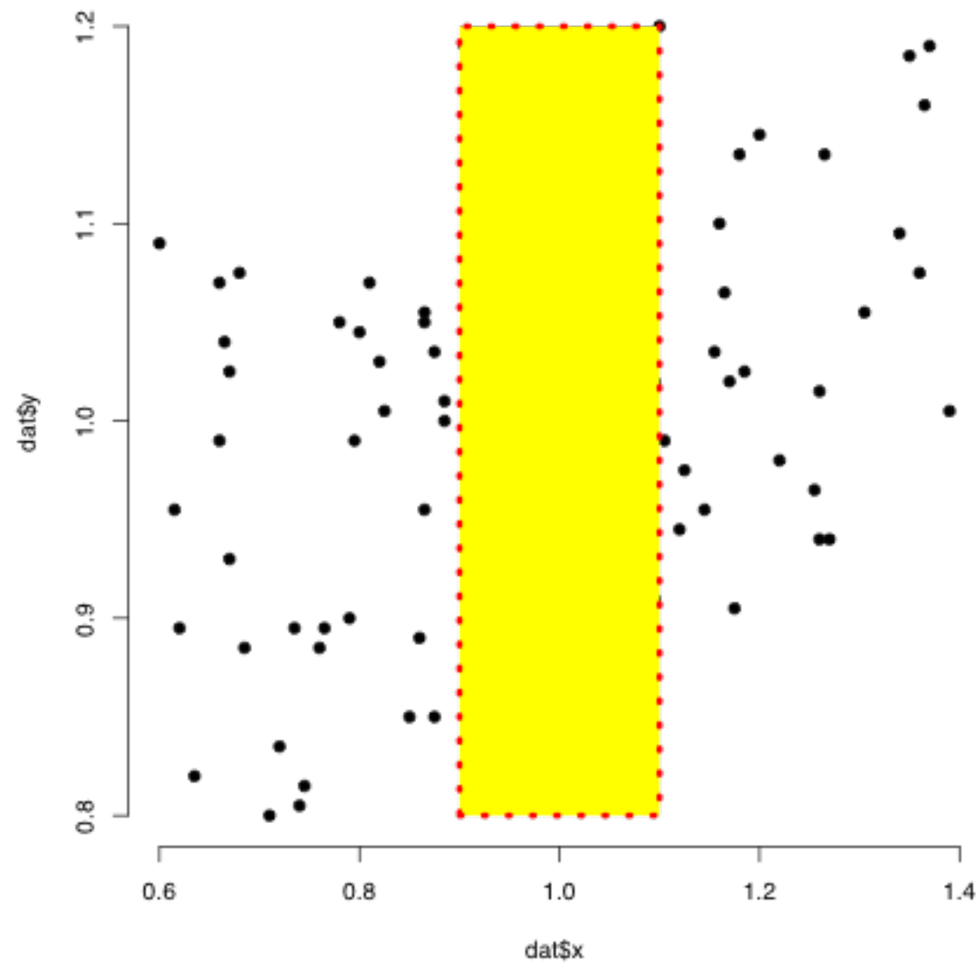
```
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

rect(xleft = 0.9, ybottom = 0.8,
     xright = 1.1, ytop = 1.2,
     density = 20, angle = 45)
```



Adding polygons

- ✓ You also can customize the rectangle

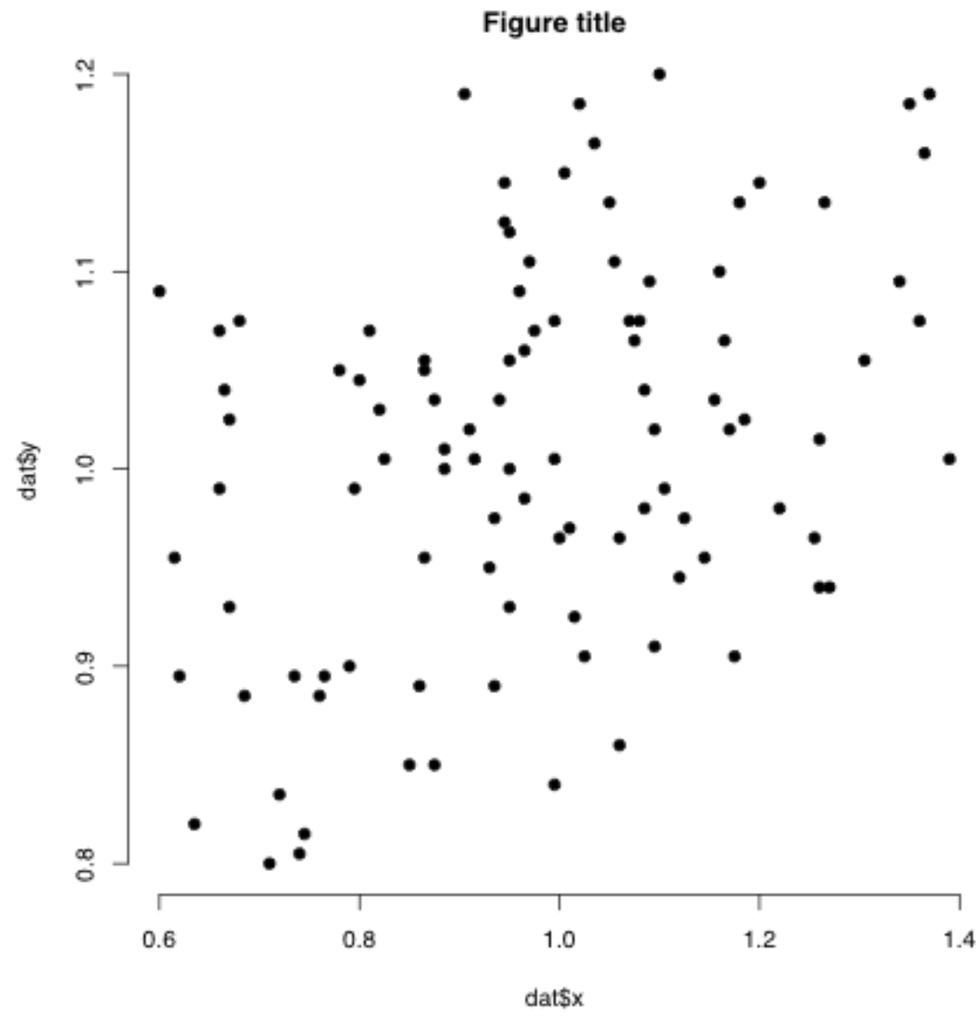
```
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

rect(xleft = 0.9, ybottom = 0.8,
     xright = 1.1, ytop = 1.2,
     col = 'yellow', border = 'red',
     lwd = 4, lty = 3)
```



Adding textual informations

- ✓ Let's see now how to add text
- ✓ First let's add a main title with the function `title()`

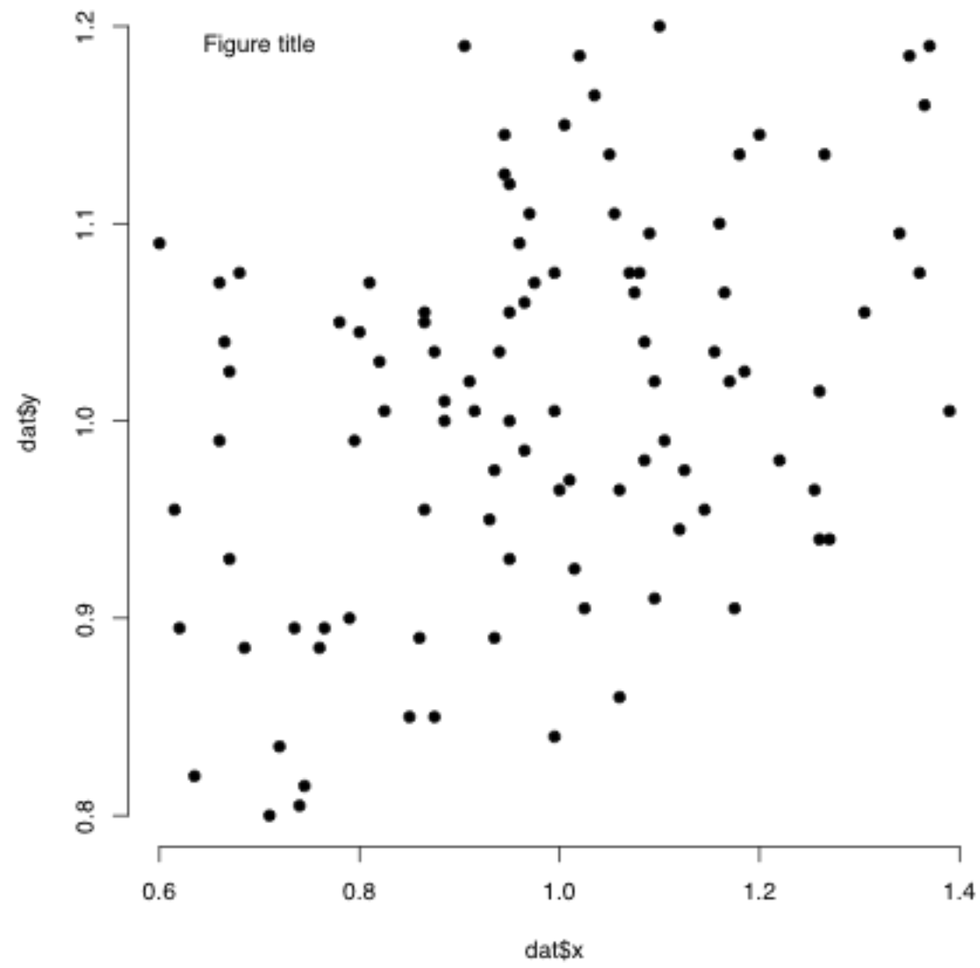
```
## Basic plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding title
title(main = 'Figure title')
```



Adding textual informations

- ✓ What about adding text in the plot area?
- ✓ We will use the function `text()`
- ✓ Here is a first example

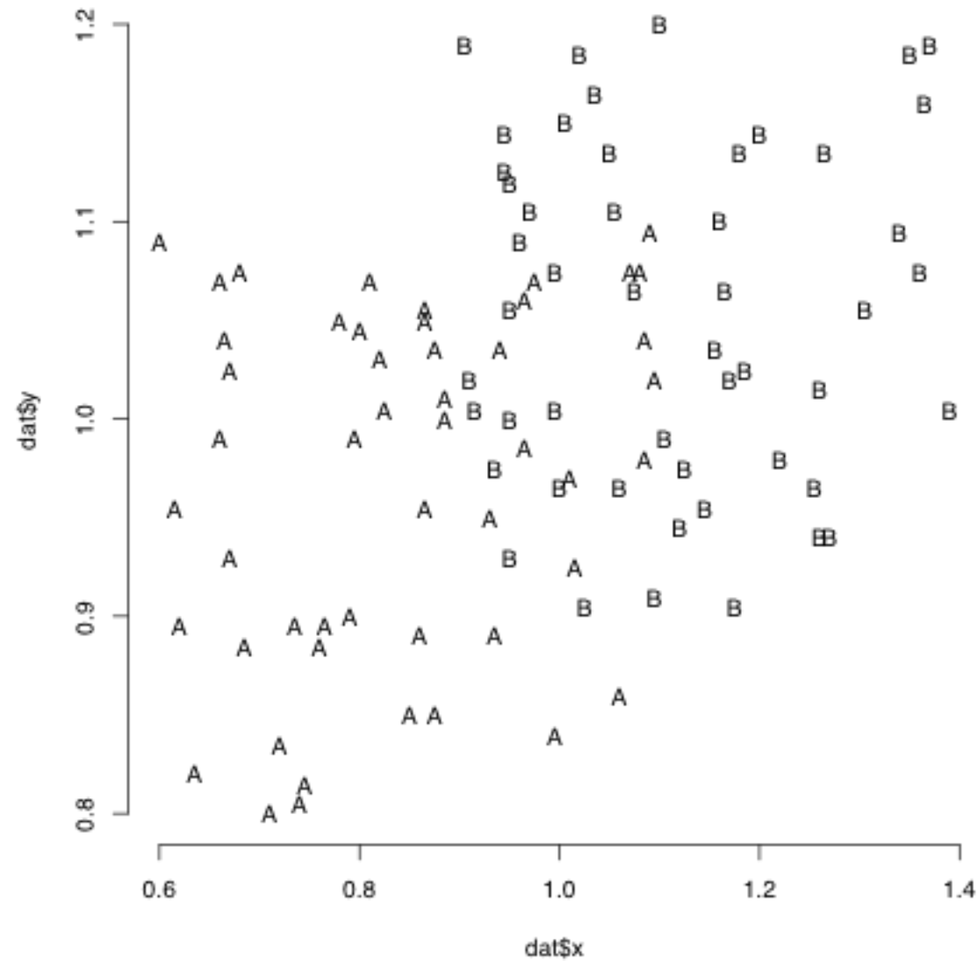
```
## Basic plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19)

## Adding text
text(x = 0.7, y = 1.19,
     labels = 'Figure title')
```



Adding textual informations

- ✓ And another example

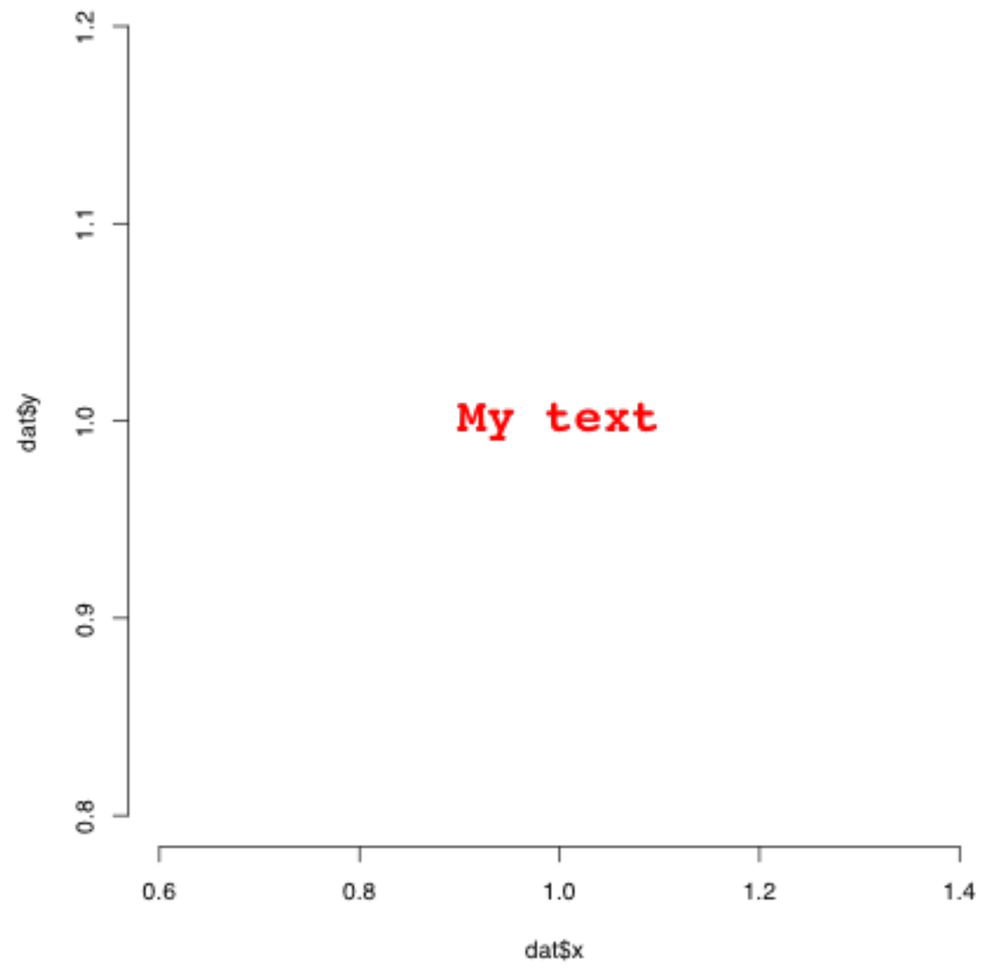
```
## Empty plot
plot(x = dat$x, y = dat$y, bty = 'n', type = 'n')

## Adding text
text(x = dat$x, y = dat$y, labels = dat$z)
```



Adding textual informations

- ✓ Let's customize a little the text with:
  - `cex`, the size
  - `col`, the color
  - `font`, the font (bold, italic, etc.)
  - `family`, the typeface

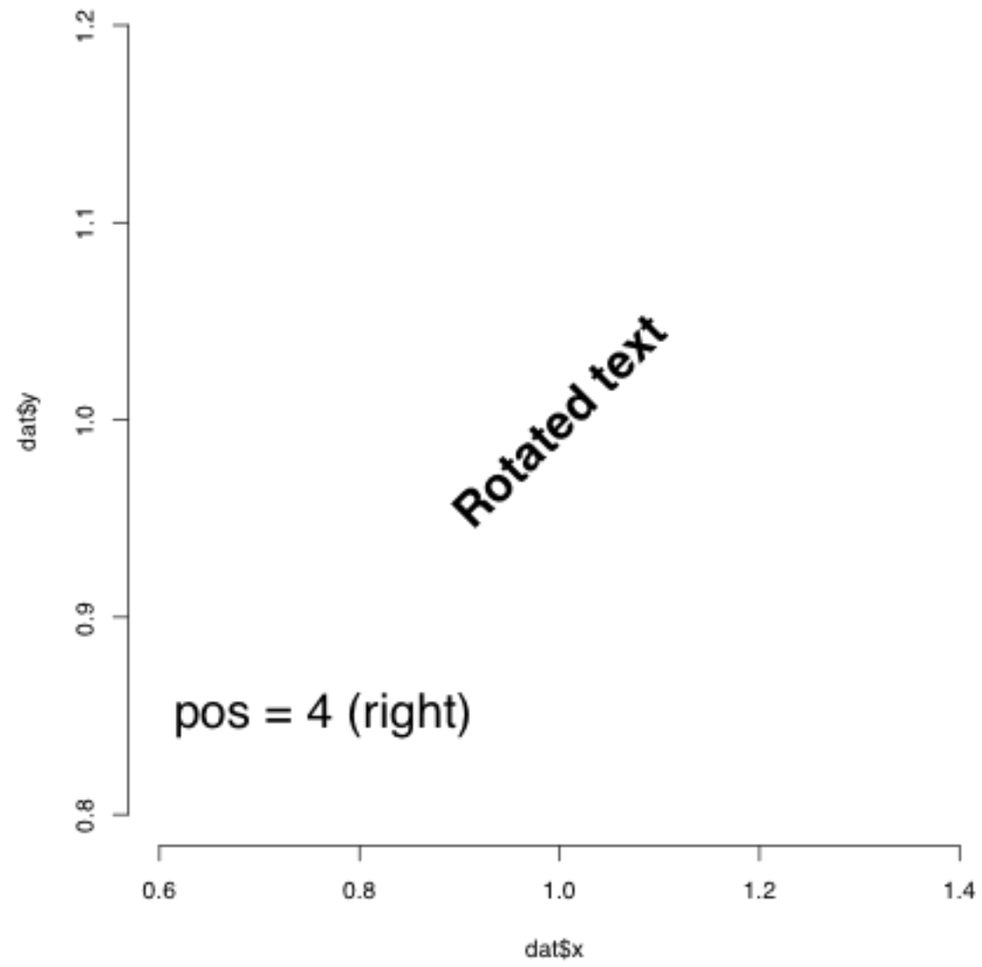
```
## Basic plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'n')

## Adding text
text(x = 1, y = 1, labels = 'My text', col = 'red',
     cex = 2, font = 2, family = 'mono')
```



Adding textual informations

- ✓ Finally, let's customize the orientation and position
  - `srt`, the rotation angle
  - `pos`, the position from coordinates

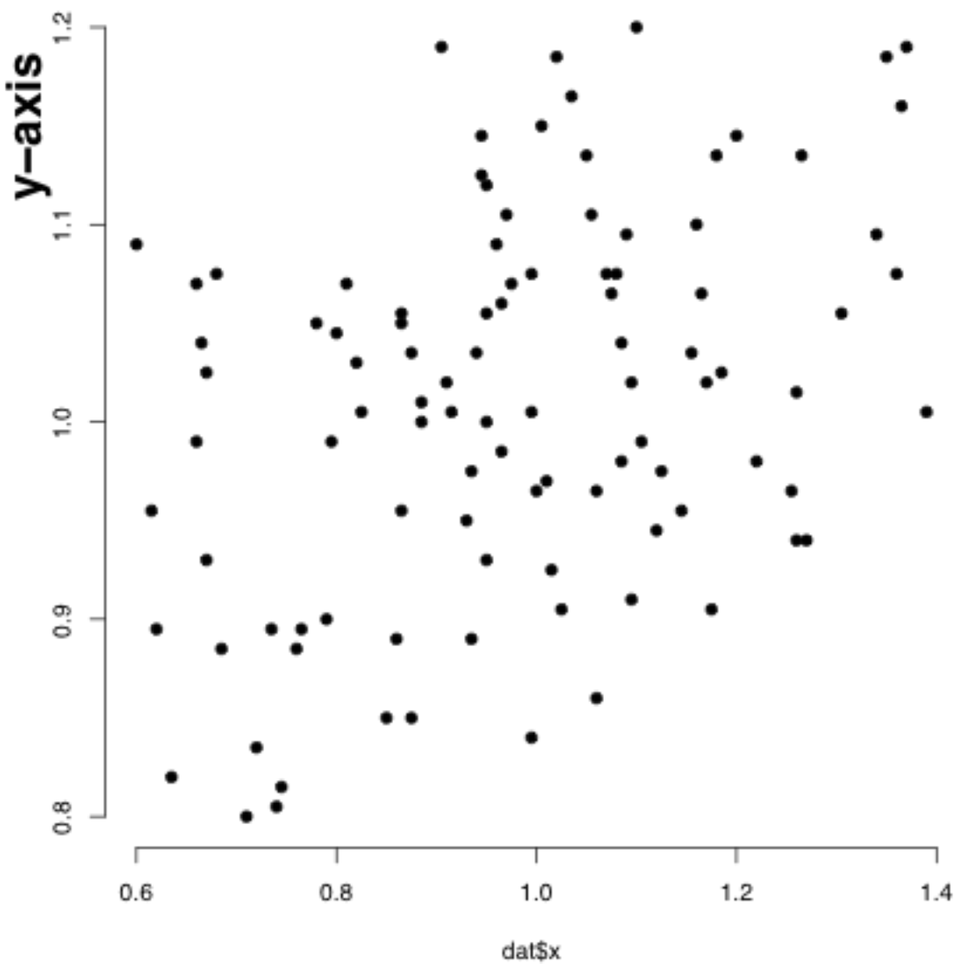
```
## Basic plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'n')

## Adding text
text(x = 0.6, y = 0.85, labels = 'pos = 4 (right)',
     cex = 2, font = 1, pos = 4)
text(x = 1, y = 1, labels = 'Rotated text',
     cex = 2, font = 2, srt = 45)
```



Adding textual informations

- ✓ But the function `text()` can't add text outside the plot area
- ✓ except if `par(xpd=TRUE)`
- ✓ We will use the function `mtext()`

```
## Basic plot
plot(x = dat$x, y = dat$y, bty = 'n',
     type = 'p', pch = 19, ylab = '')


## Adding text
mtext(text = 'y-axis', side = 2, line = 2,
      at = 1.15, font = 2, cex = 2)
```



Back to colors

- ✓ R has some predefine colors palettes

```
## Basic colors
palette()
## [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
## [8] "gray"

## and 657 others colors
colors()[1:10]
##  [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1"
##  [5] "antiquewhite2" "antiquewhite3" "antiquewhite4" "aquamarine"
##  [9] "aquamarine1"   "aquamarine2"
```

Back to colors

- ✓ You can define colors in the RGB system with `rgb()`
- ✓ You specify value between 0 to 1 for each primary colors
- ✓ For example:

```
red    <- rgb(red = 1,   green = 0,   blue = 0)
yellow <- rgb(red = 1,   green = 1,   blue = 0)
gray1  <- rgb(red = 0.5, green = 0.5, blue = 0.5)
```

Back to colors

- ✓ You can define colors in the RGB system with `rgb()`
- ✓ You specify value between 0 to 1 for each primary colors
- ✓ For example:

```
red    <- rgb(red = 1,   green = 0,   blue = 0)
yellow <- rgb(red = 1,   green = 1,   blue = 0)
gray1  <- rgb(red = 0.5, green = 0.5, blue = 0.5)
```

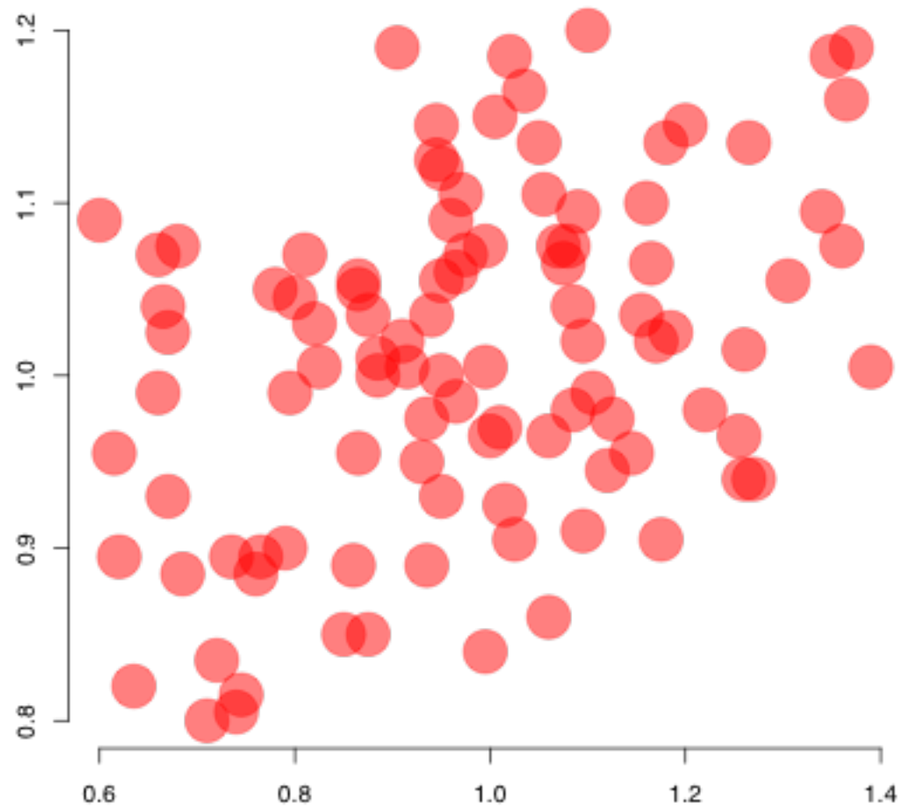
- ✓ The `alpha` argument controls for opacity (default = `1`)
- ✓ Its values vary from 0 (transparent) to 1 (opaque)
- ✓ For example

```
## Transparent red
redp <- rgb(red = 1, green = 0, blue = 0,
            alpha = .8)
```

Back to colors

✓ Let's see an application

```
## Empty plot
plot(x = dat$x, y = dat$y, ann = FALSE,
     bty = 'n', type = 'n')

## Transparent red
redp <- rgb(red = 1, green = 0, blue = 0,
            alpha = .5)

## Adding points
points(x = dat$x, y = dat$y, col = redp,
       pch = 19, cex = 4)
```



Back to colors

- ✓ You can also define colors in the hexadecimal system
- ✓ Each primary colors is define by two values varying from 0 to 9 and A to F
- ✓ For example:

```
red    <- '#FF0000'
yellow <- '#FFFF00'
gray1  <- '#888888'
```

Back to colors

- ✓ You can also define colors in the hexadecimal system
- ✓ Each primary colors is define by two values varying from 0 to 9 and A to F
- ✓ For example:

```
red    <- '#FF0000'
yellow <- '#FFFF00'
gray1  <- '#888888'
```

- ✓ To add transparency, we have to add two hexadecimal character at the end
- ✓ For example,

```
## Transparent red
redp <- '#FF000088'
```

Adding axis

- ✓ The function `axis()` allows to add axis
- ✓ Here is an example of usage

```
## Empty plot
plot(x = dat$x, y = dat$y, pch = 19)

## Adding top-axis
axis(side = 3, at = seq(0.6, 1.4, by = 0.1),
     labels = seq(0.6, 1.4, by = 0.1), las = 1)

## Adding right-axis
axis(side = 4, at = seq(0.8, 1.2, by = 0.1),
     labels = format(seq(0.8, 1.2, by = 0.1)),
     las = 2)
```

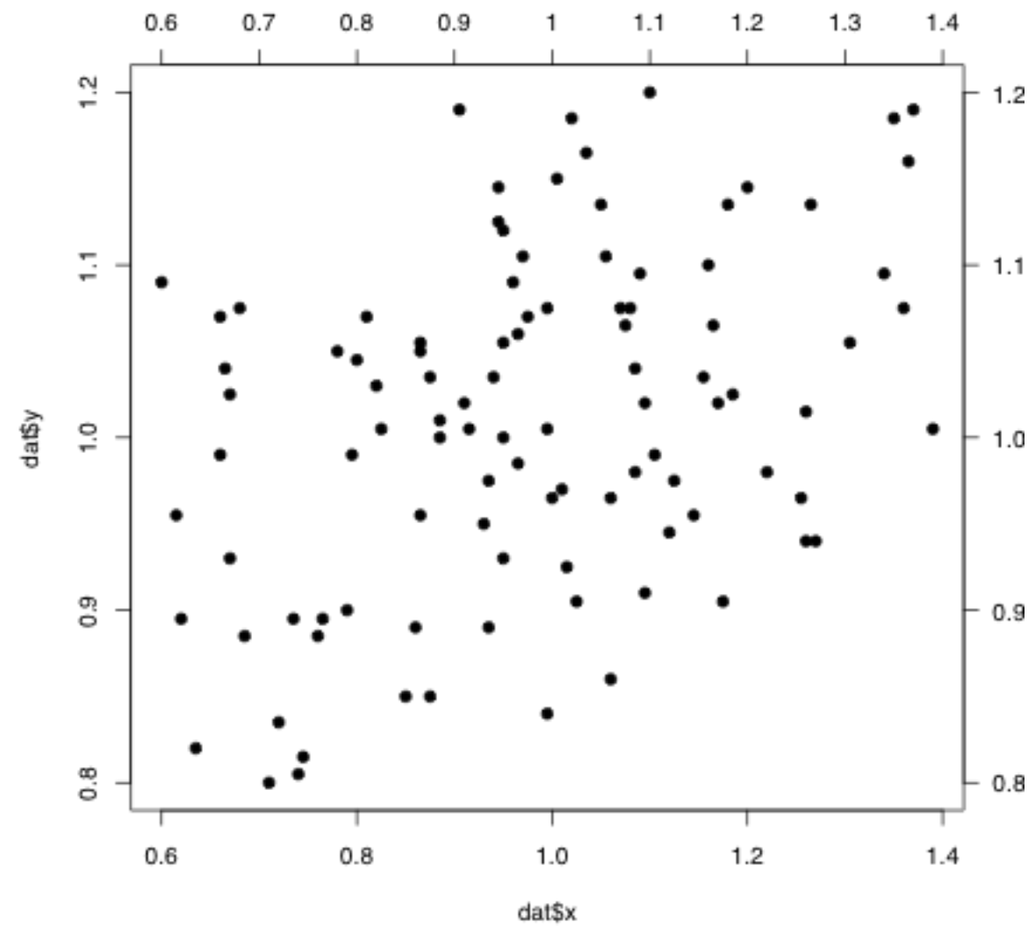
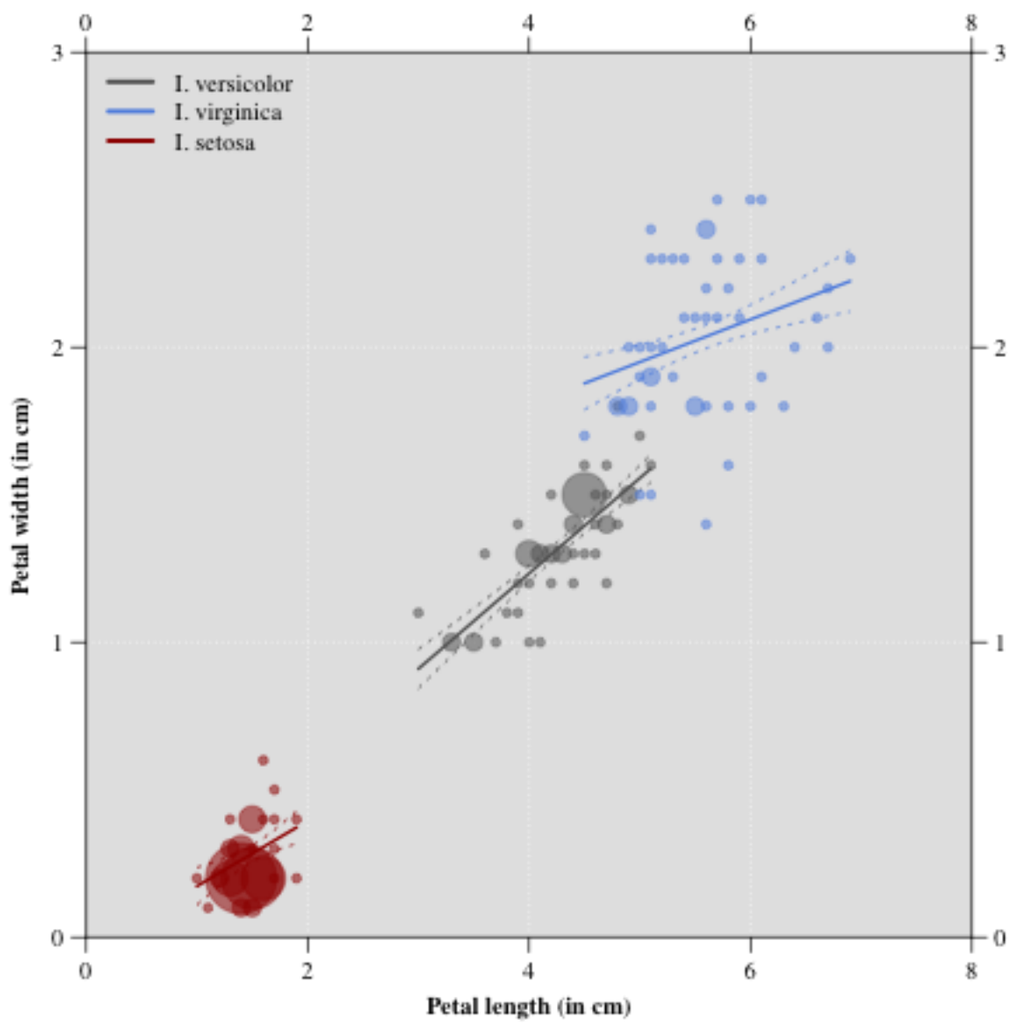


Figure margins

- ✓ To change the figure margins you have to change the values of the parameter `mar` in the `par()`
- ✓ The order is the follow: bottom, left, top and right
- ✓ For example:

```
par(mar = c(4, 4, 4, 4))
```

Exercise 1


I. versicolor
I. virginica
I. setosa
Petal width (in cm)
Petal length (in cm)
0
2
4
6
8
0
1
2
3

Exercise 1

- ✓ Objective: reproduce this figure
- ✓ Using the dataset `iris.txt` (Dropbox)
- ✓ The size of the bubble is proportional to `n`
- ✓ Lines represent regression model and standard error

```
head(tab)
##   species petal.l petal.w n
## 1  setosa     1.4     0.2 8
## 2  setosa     1.3     0.2 4
## 3  setosa     1.5     0.2 7
## 4  setosa     1.7     0.4 1
## 5  setosa     1.4     0.3 3
## 6  setosa     1.5     0.1 2
```

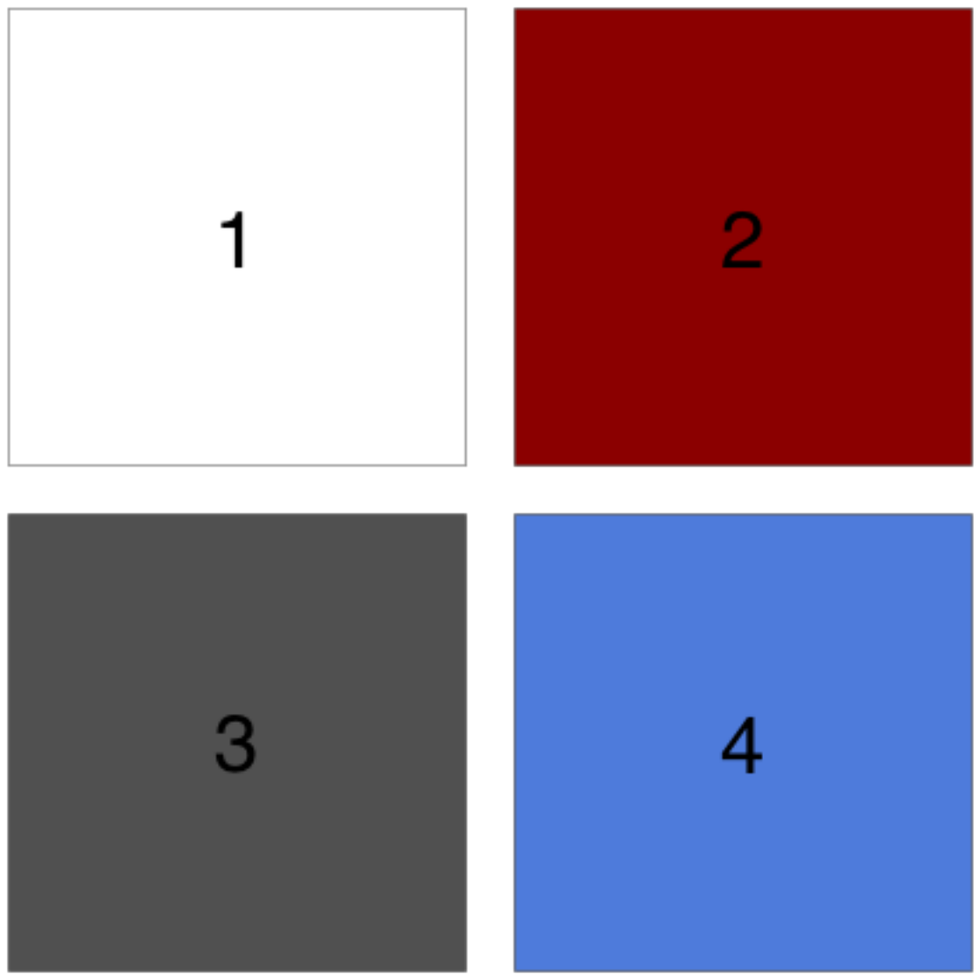
Dividing the output device

- ✓ `mfrow` and `mfcol` in `par()`

```
par(mfrow=c(2,2))
```

or

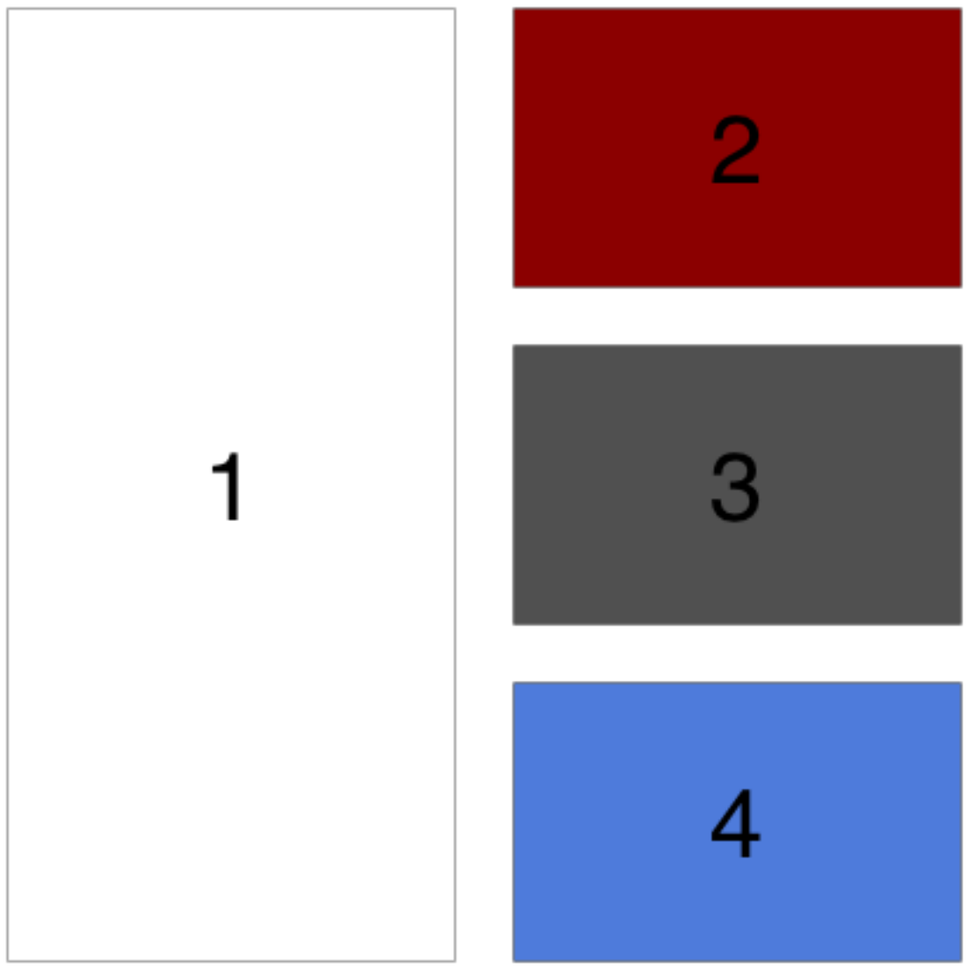
```
par(mfcol=c(2,2))
```



Dividing the output device

- ✓ `mfrow` and `mfcol` in `par()`
- ✓ `split.screen()`

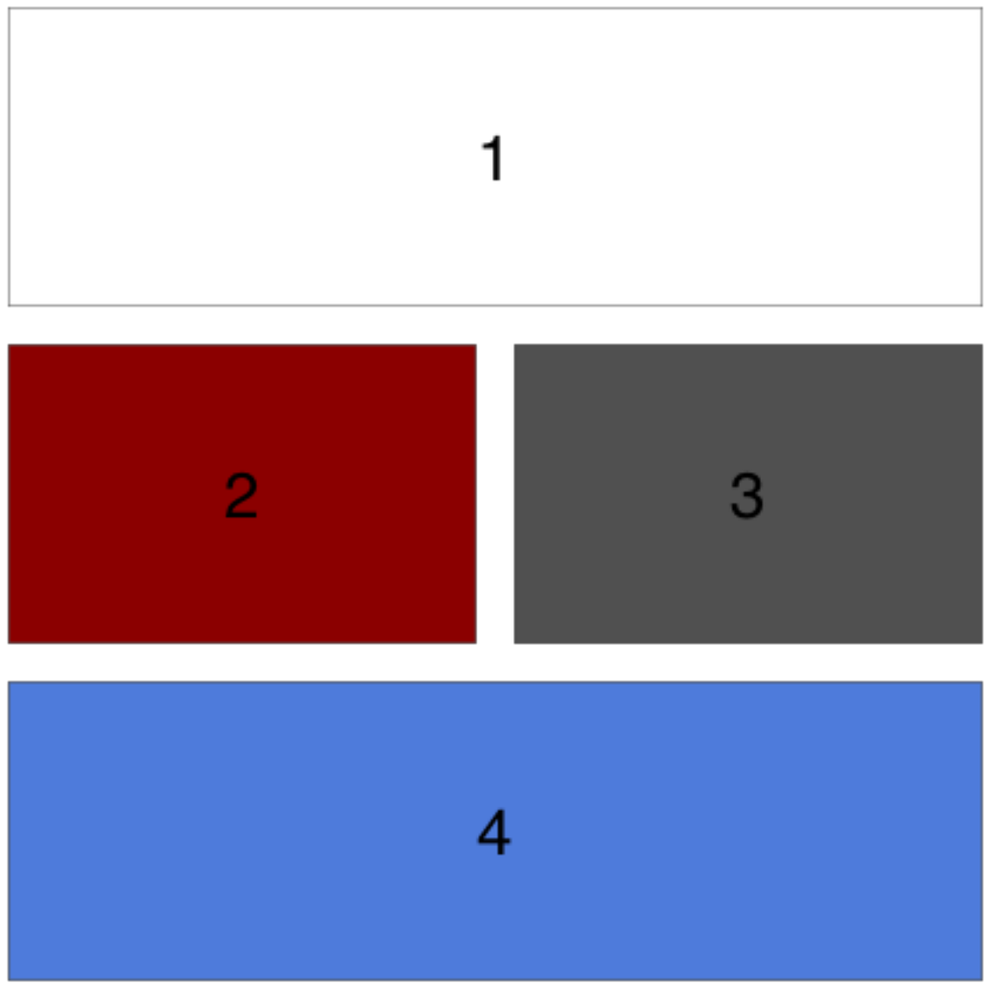
```
split.screen(c(1, 2))
split.screen(c(3, 1), screen = 2)
```



Dividing the output device

- ✓ mfrow and mfcol in `par()`
- ✓ `split.screen()`
- ✓ layout()

```
mat_lay <- matrix(c(1,2,4,1,3,4),nrow=3)
layout(mat_lay)
```



More about 'layout()'

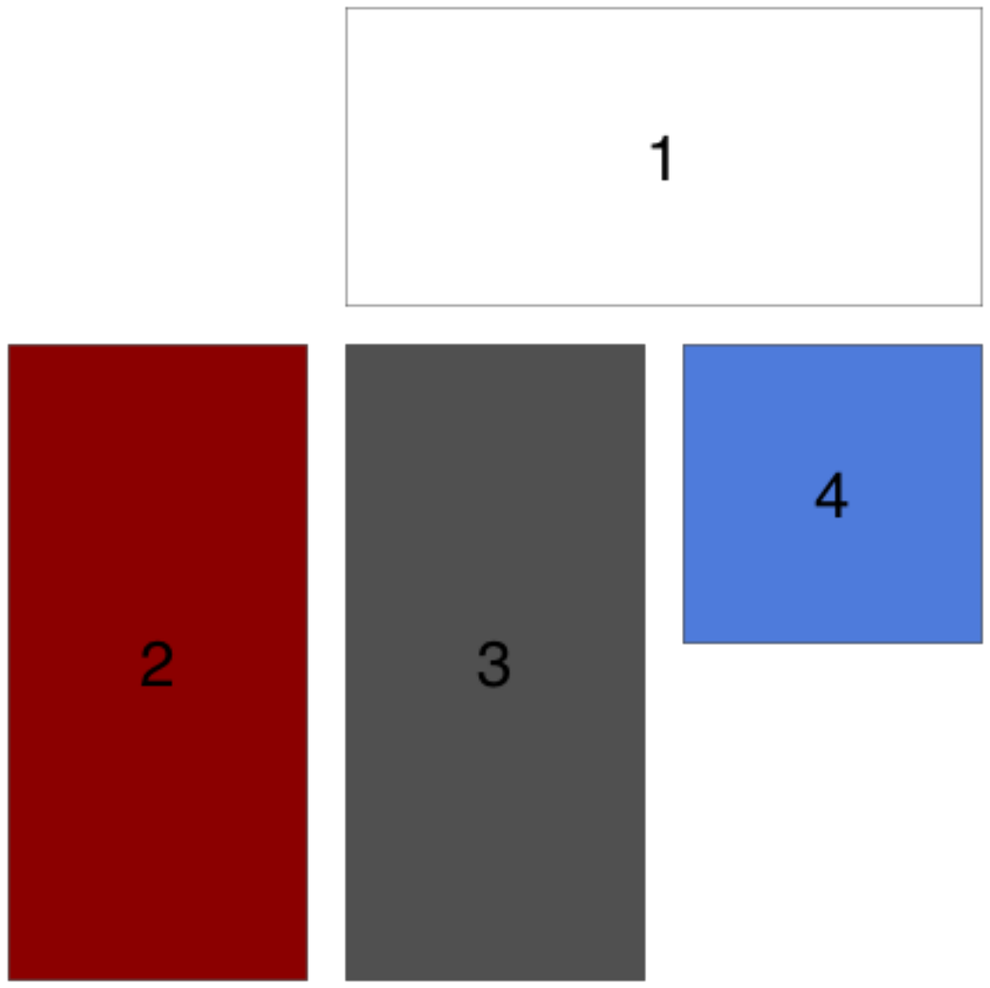
```
mat_lay <- matrix(c(1,2,4,1,3,4), nrow=3)
layout(mat_lay)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    3
## [3,]    4    4
```

More about 'layout()'

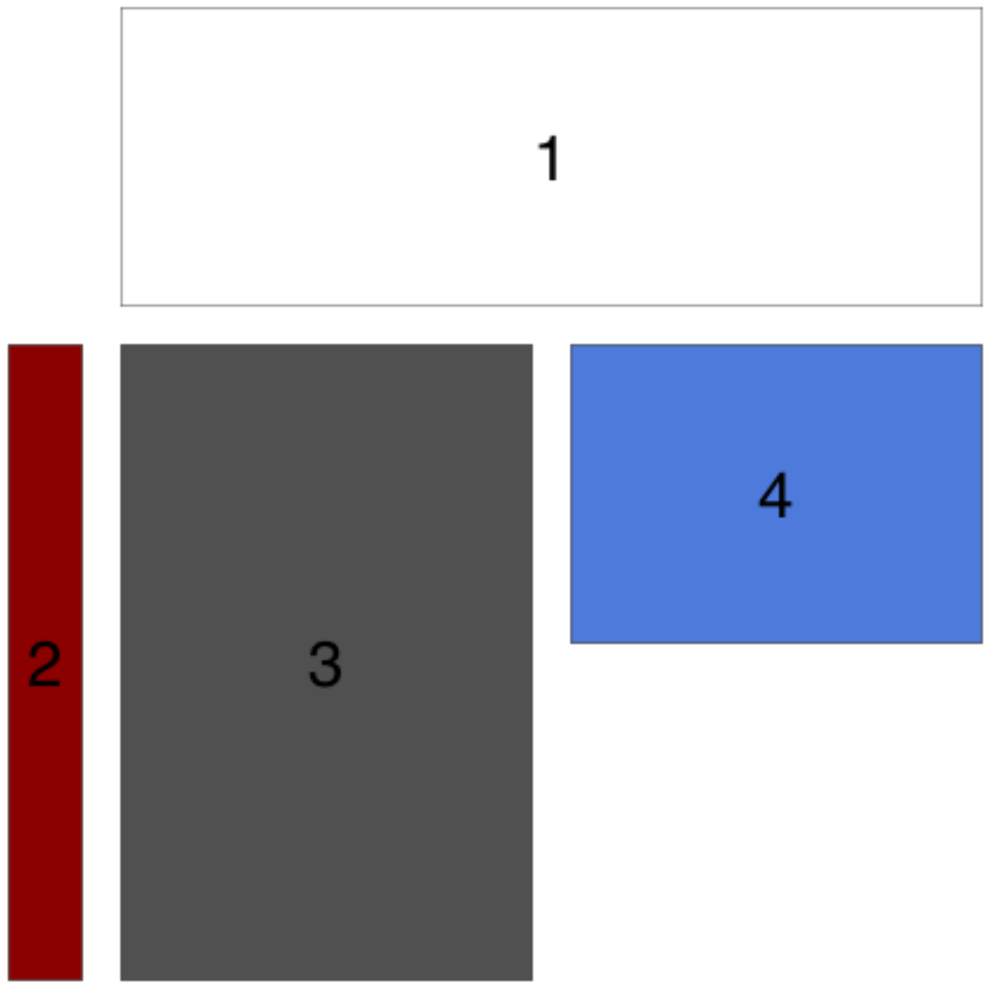
```
mat_lay <- matrix(c(0,2,2,1,3,3,1,4,0), nrow=3)
layout(mat_lay)
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    2    3    4
## [3,]    2    3    0
```



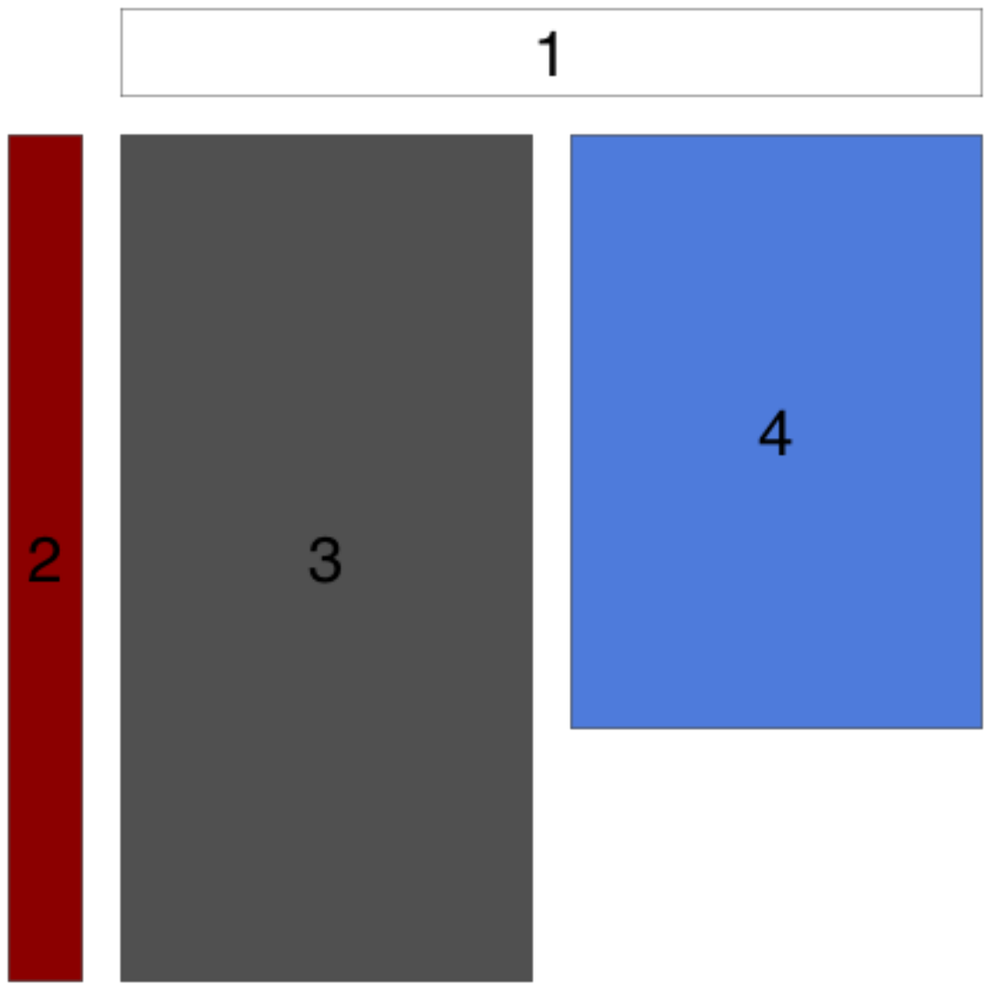
More about 'layout()'

```
mat_lay <- matrix(c(0,2,2,1,3,3,1,4,0),nrow=3)
layout(mat_lay, widths=c(.25,1,1))
```



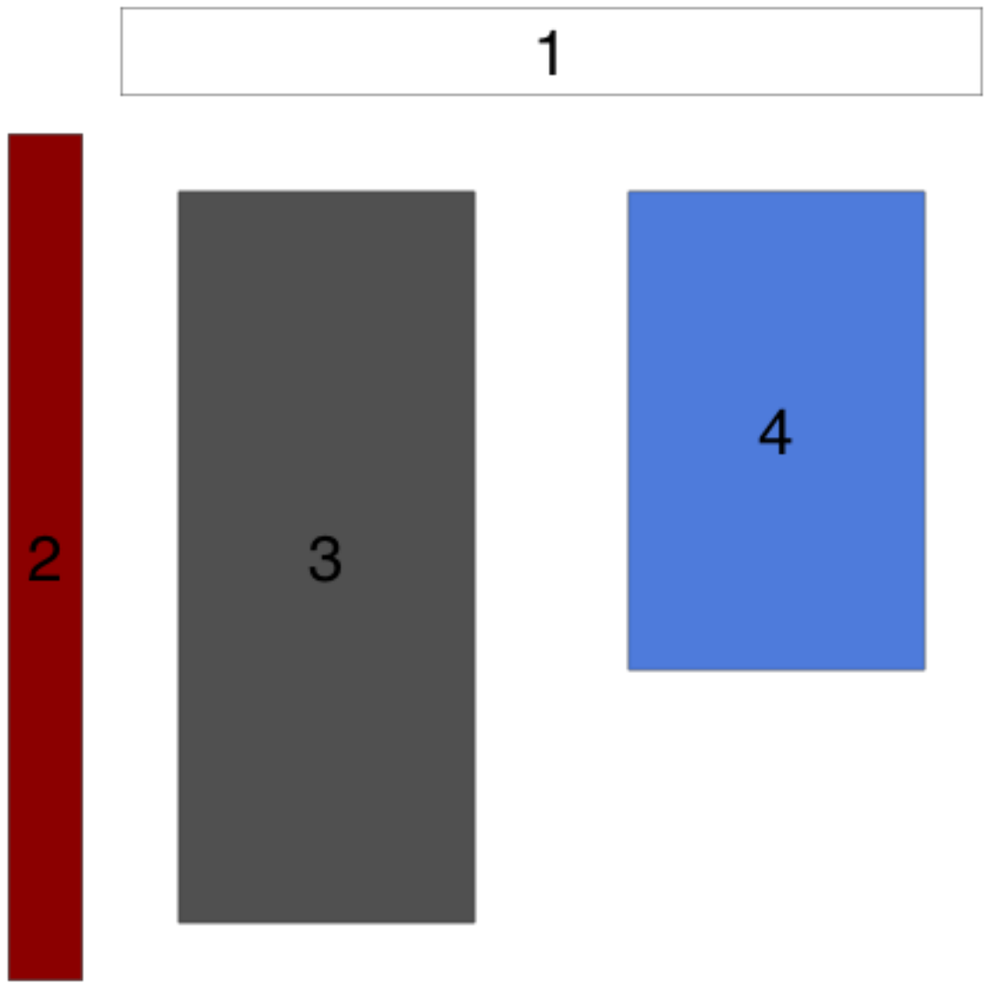
More about 'layout()'

```
mat_lay <- matrix(c(0,2,2,1,3,3,1,4,0),nrow=3)
layout(mat_lay, widths=c(.25,1,1),
  heights=c(.25,1,.25))
```



Combining 'layout()' and 'mar'

```
mat_lay <- matrix(c(0,2,2,1,3,3,1,4,0),nrow=3)
layout(mat_lay, widths=c(.25,1,1),
  heights=c(.2,1,.4))

for (i in 1:4) {
  if (i<3) par(mar=rep(1,4)) else par(mar=rep(4,4))
  eplot()
  fillIt(col=i)
  text(0,0, labels=i, cex=4)
}
```

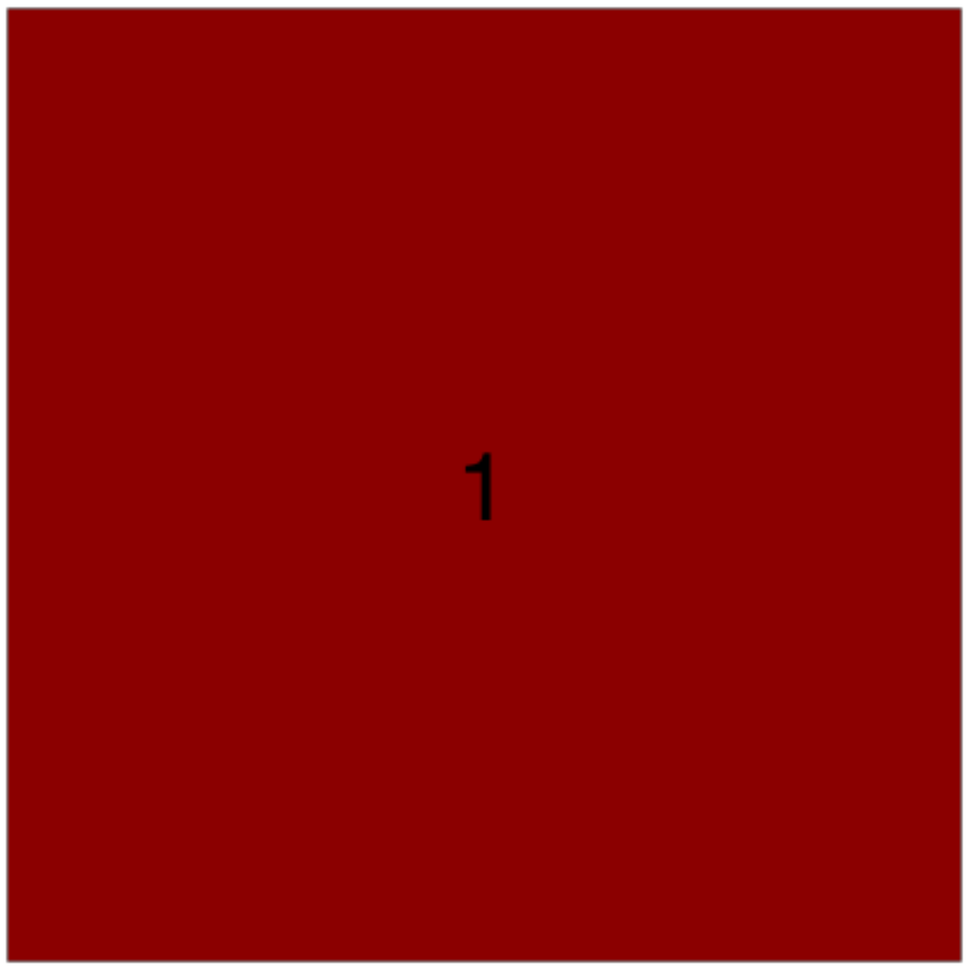


Embedded plots

- ✓ You must call `new=TRUE` and specifying `fig` in `par()`:

1. create your first plot;

```
plot(...)
```

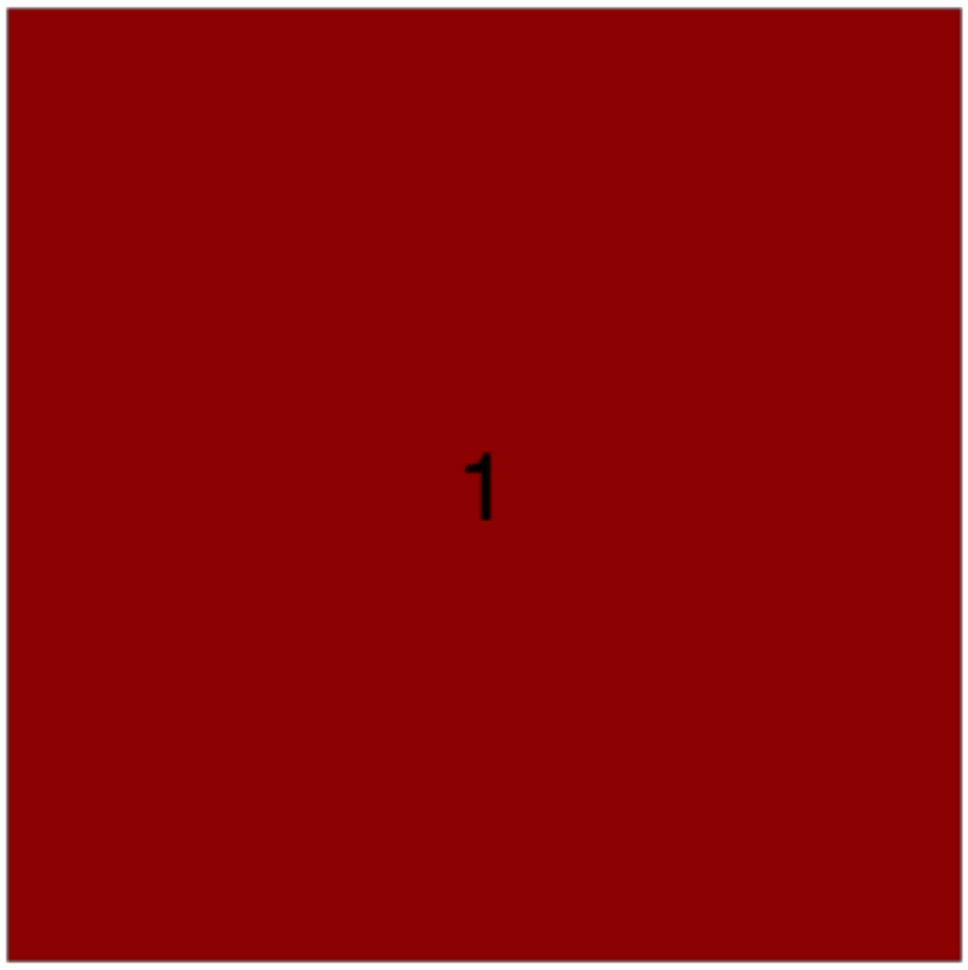


Embedded plots

✓ You must call `new=TRUE` and specifying `fig` in `par()`:

1. create your first plot;
2. use `par()`;

```
plot(...)
par(new=TRUE, fig=c(0.5,1,0.5,1))
```

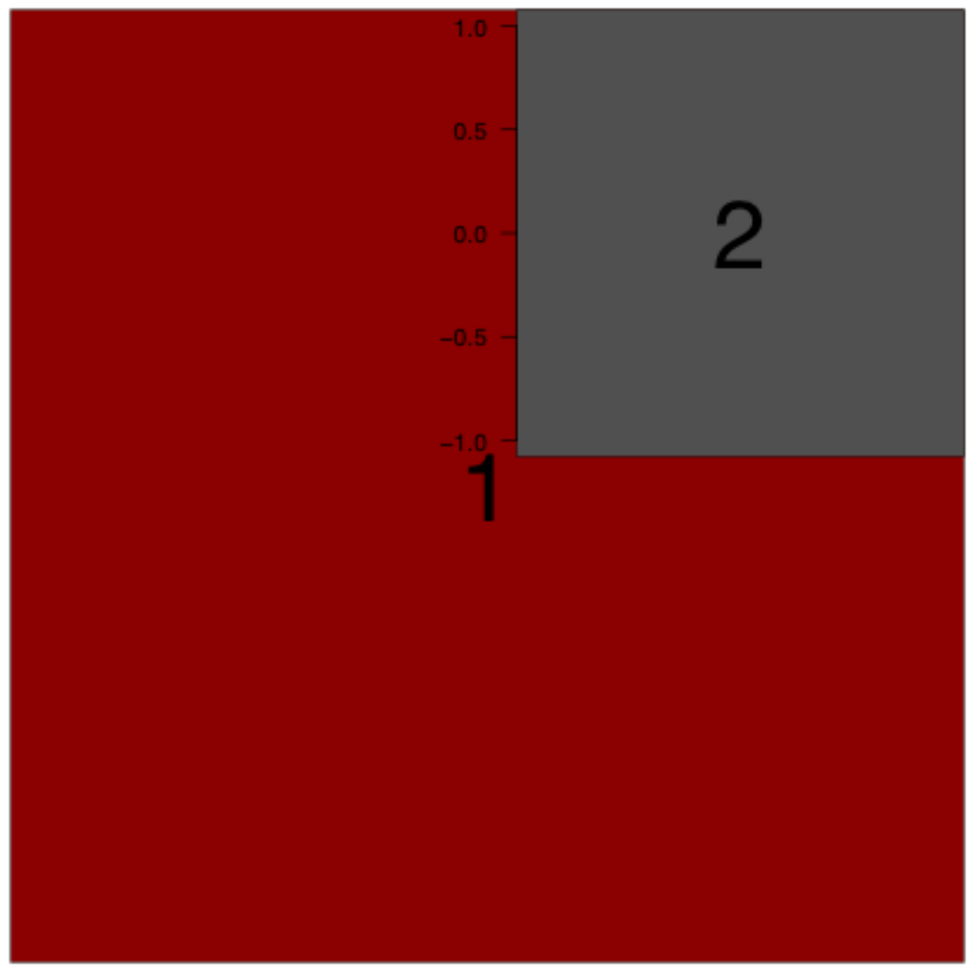


Embedded plots

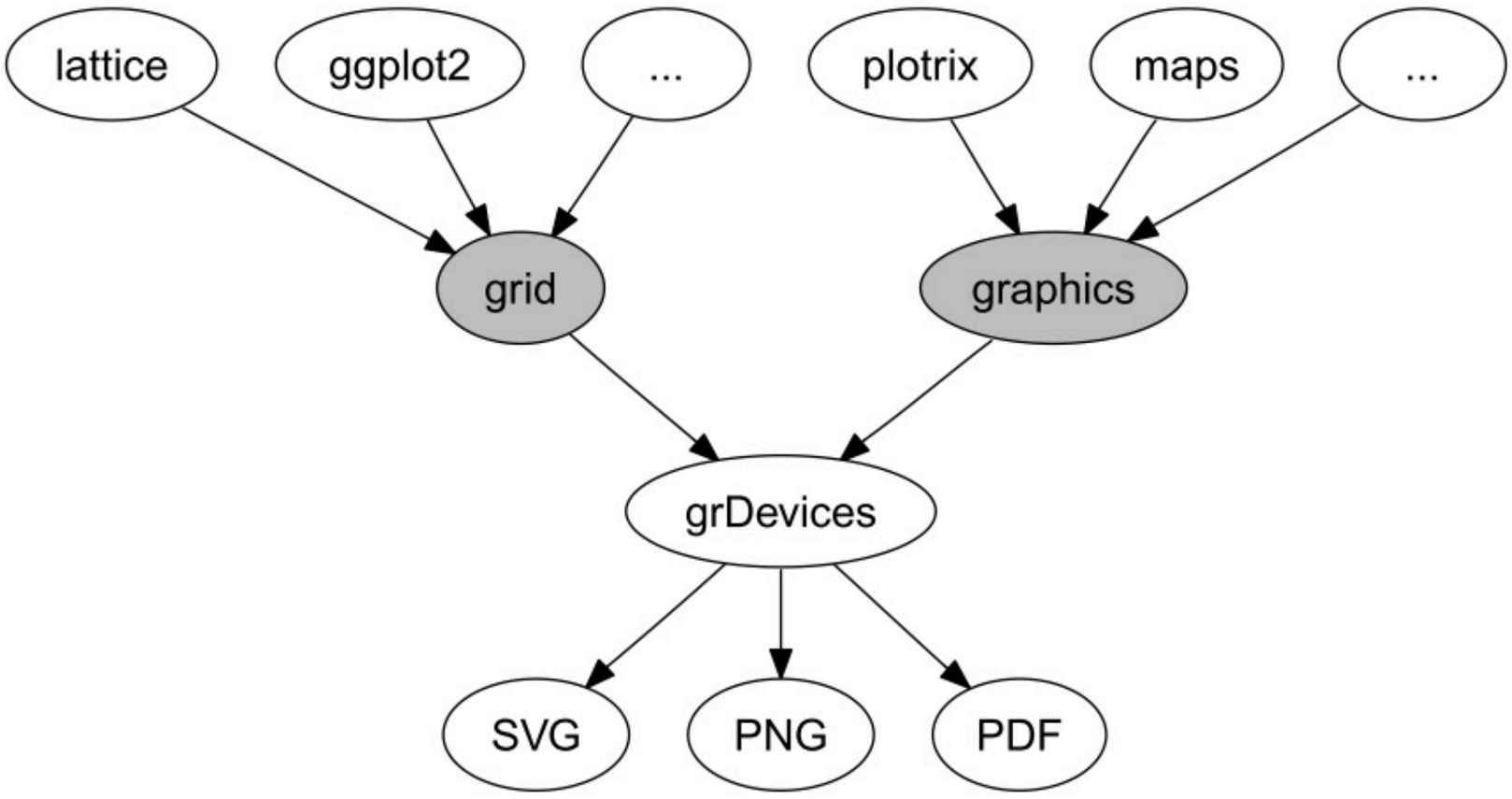
- ✓ You must call `new=TRUE` and specifying `fig` in `par()`:

1. create your first plot;
2. use `par()`;
3. add your embedded plot;

```
plot(...)
par(new=TRUE, fig=c(0.5,1,0.5,1))
plot(...)
```



grDevices



Murrell, P. (2015) The gridGraphics Package. The R Jounal.

grDevices

```
options('device')
```

✓ Devices available :

- Quartz
- X11
- pdf, jpeg, svg, ...
- in add-on package :
  - rgl package (OpenGL website)
  - Internet browsers googleVis

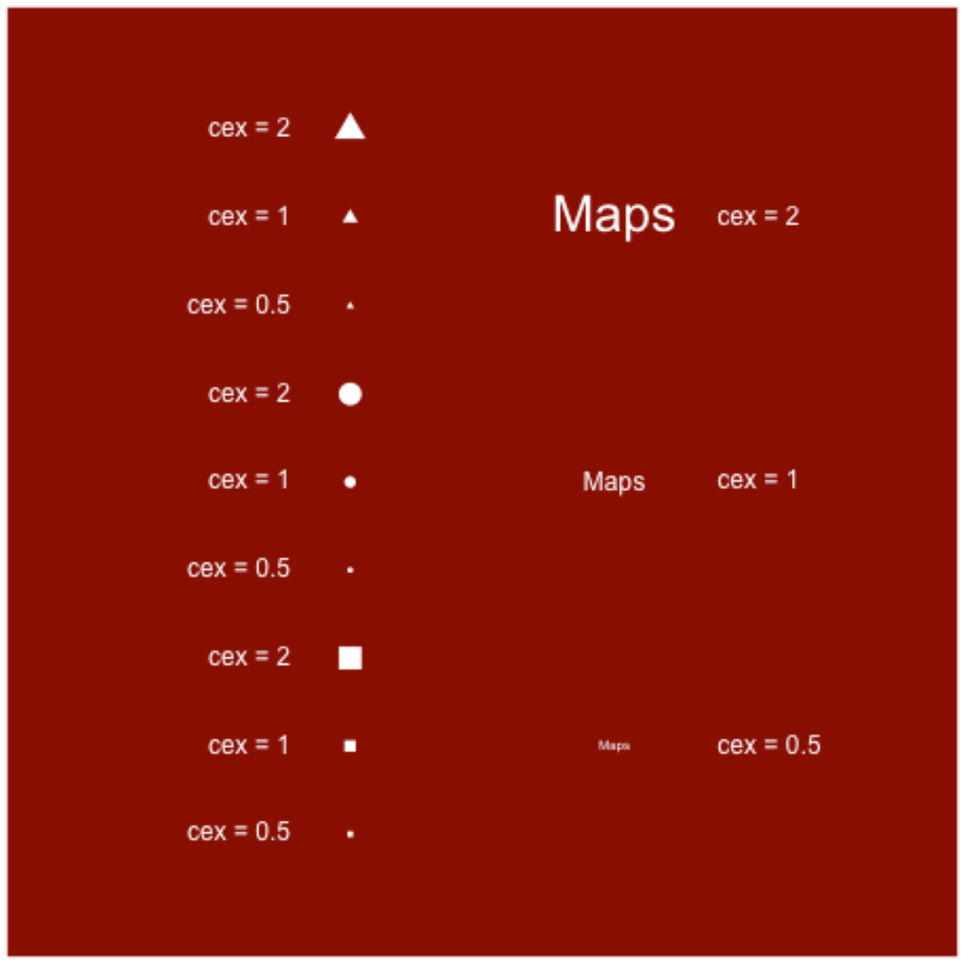
Exporting figures as bitmap files

✓ `bmp()`, `jpeg()`, `png()`, `tiff()`

```
?jpeg
```

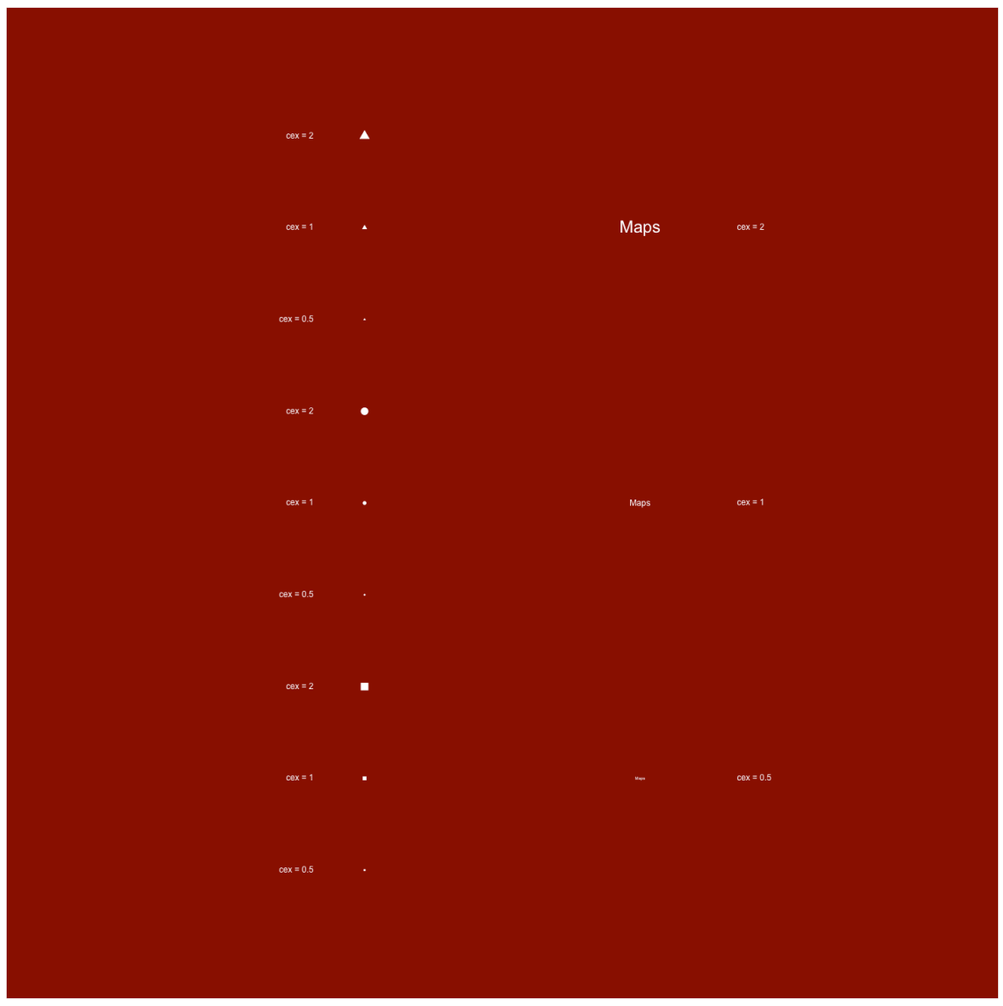
✓ use them:

```
png(filename, width=480, height=480)
...
dev.off()
```



Exporting figures as bitmap files

```
png(filename, width=1440, height=1440)
...
dev.off()
```

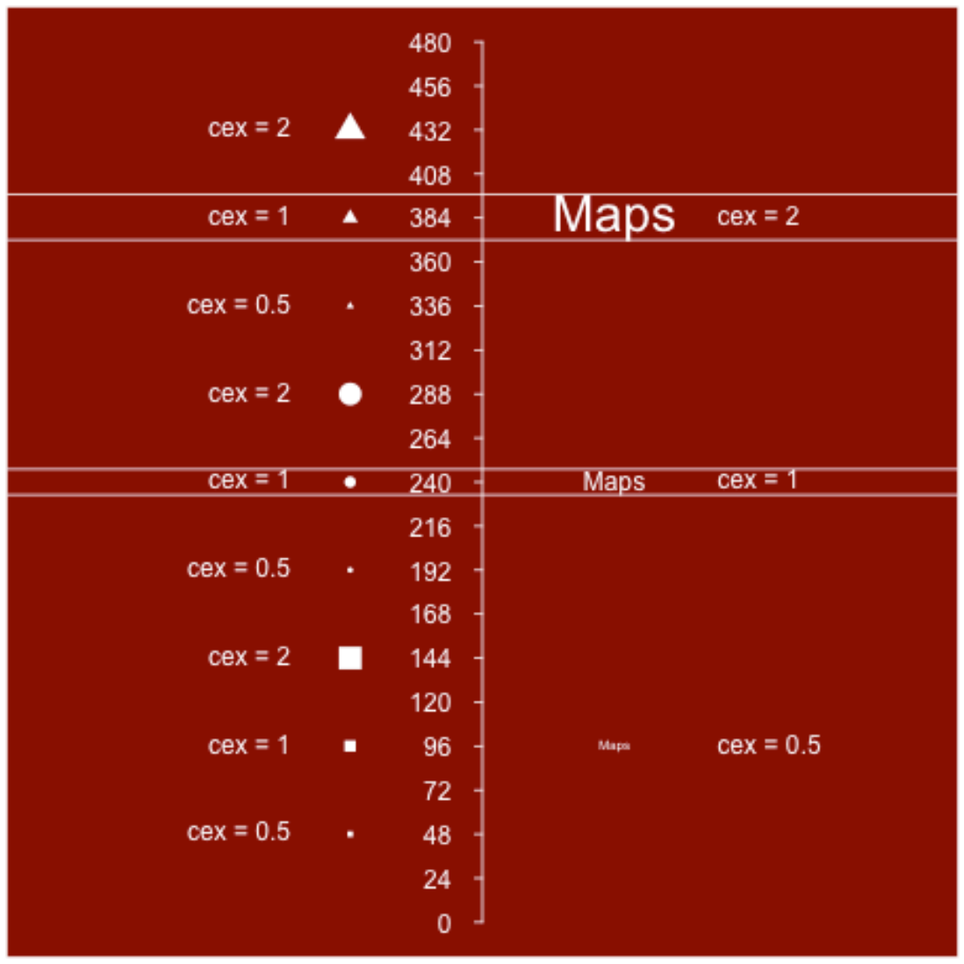


Exporting figures as bitmap files

- ✓ pixel (px) = small colored square;
- ✓ `width=480` + `height=480` = grid of 480x480px ;
- ✓ point (pt) = unit of length (measures the height of a font);
- ✓ 1pt = 1/72 inch;
- ✓ `pointsize` of plotted text = how many points your font will use (size of the text);
- ✓ resolution `res` (in px per inch, ppi) links pixel and size;
- ✓ `res` determines how many pixels = 1pt;
- ✓ if `res=72` then one point will equal exactly one pixel.
- ✓ `res=72` + `width=480` + `height=480` = 6.667x6.667in => 16.9*16.9cm
- ✓ `res=300` + `width=480` + `height=480` = 1.6x1.6in => 4.06cmx4.06cm
- ✓ font 12 points => 0.42cm

Exporting figures as bitmap files

```
jpeg(filename, res=72,
  pointsize=12, width=480, height=480)
...
dev.off()
```



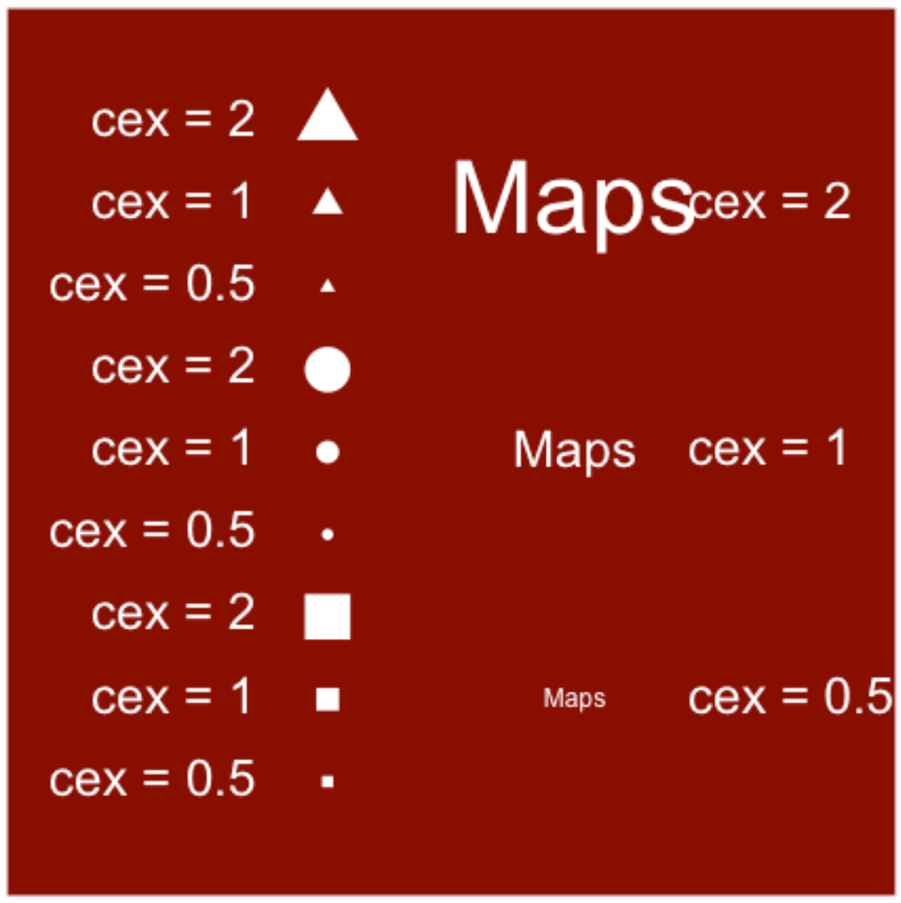
Exporting figures as bitmap files

- ✓ pointsize=12
- ✓ res=72
- ✓ cex=1

- ✓ pointsize=12
- ✓ res=72
- ✓ cex=2

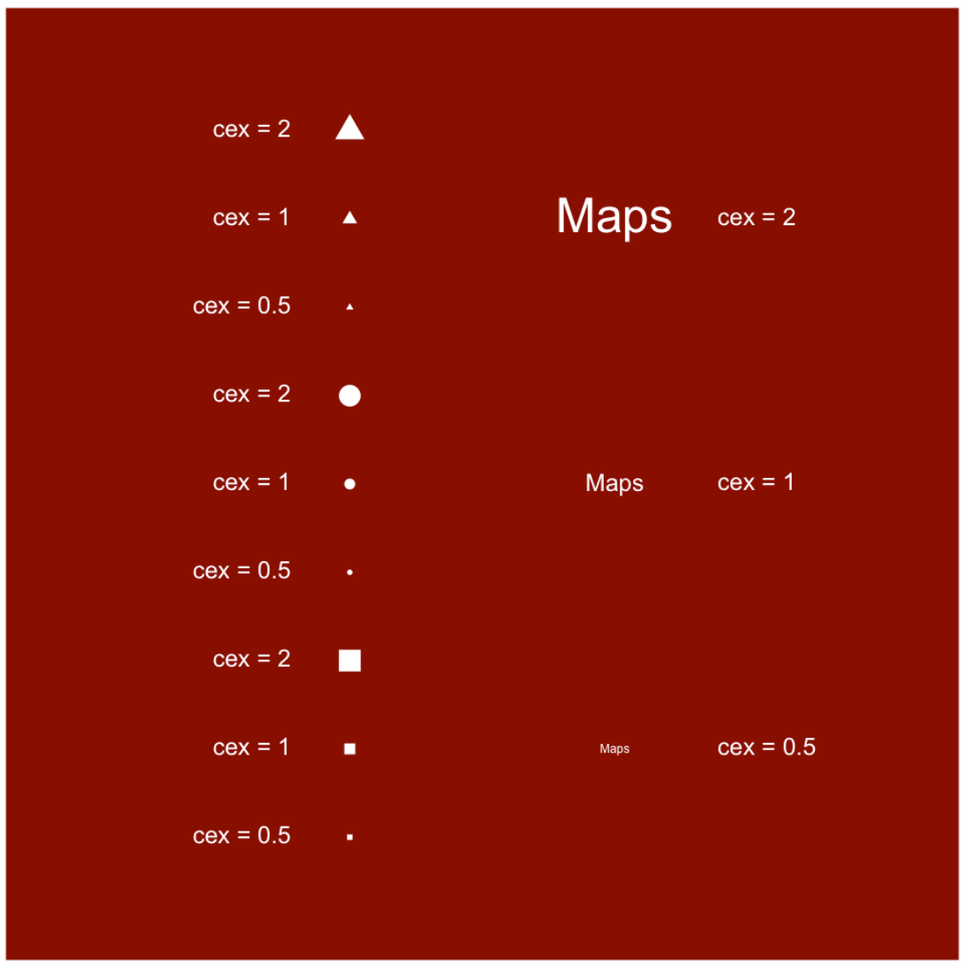
Exporting figures as bitmap files

```
png(filename, res=144)
...
dev.off()
```



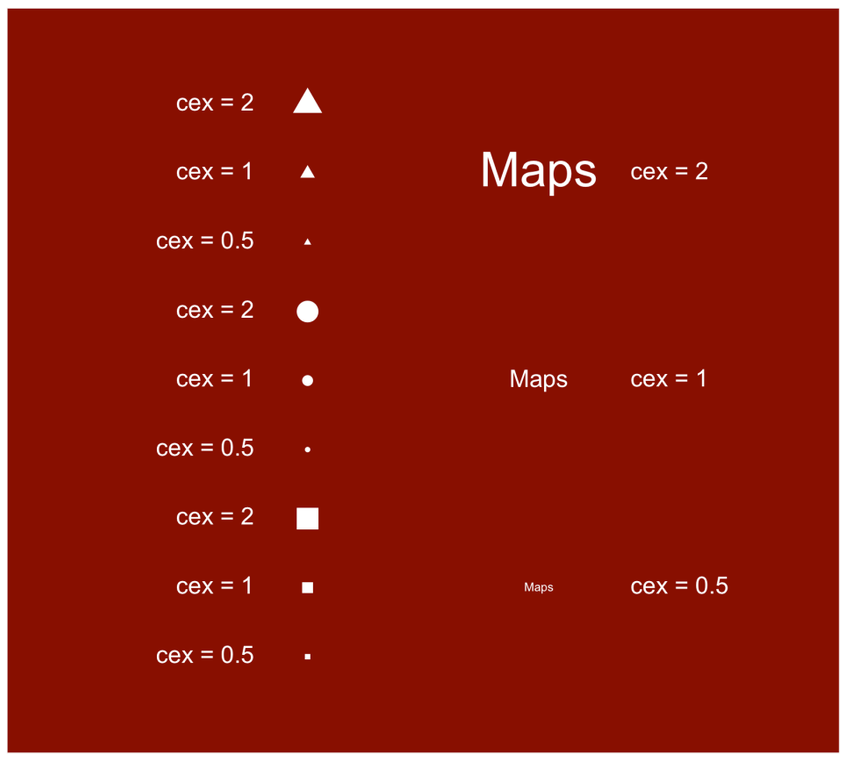
Exporting figures as bitmap files

```
png(filename, res=144,
  height=7, width=7, unit="in")
...
dev.off()
```



Exporting figures as bitmap files

```
png(filename, res=288,
   height=7, width=7, unit="in")
...
dev.off()
```

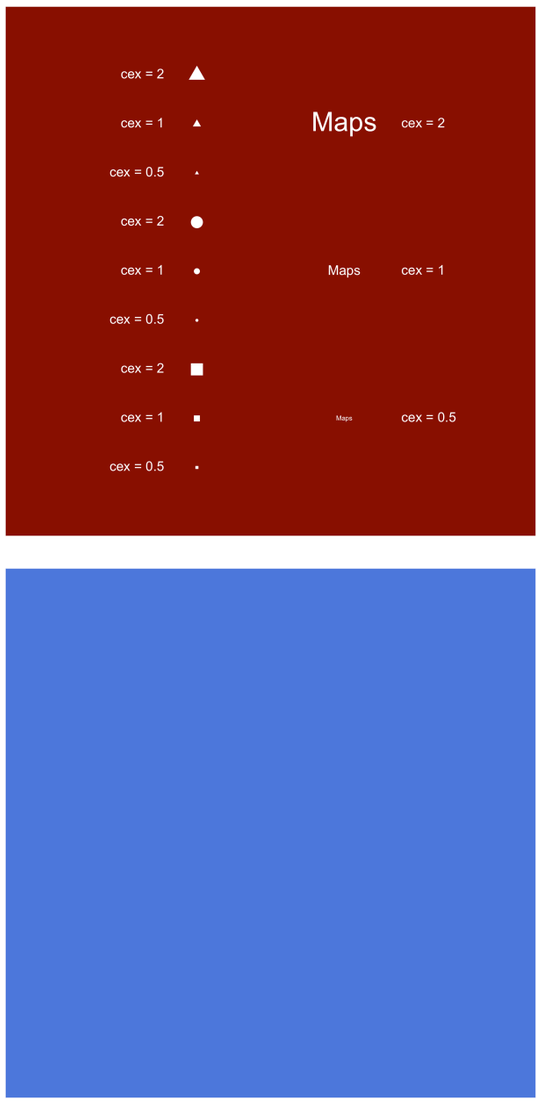


Exporting figures as bitmap files

```
png(filename, res=288,
  height=7, width=7, unit="in")
...
dev.off()
```

Exporting figures as bitmap files

```
png(filename, res=288,
  height=2*7, width=7, unit="in")
...
dev.off()
```



Exporting figures as vector files

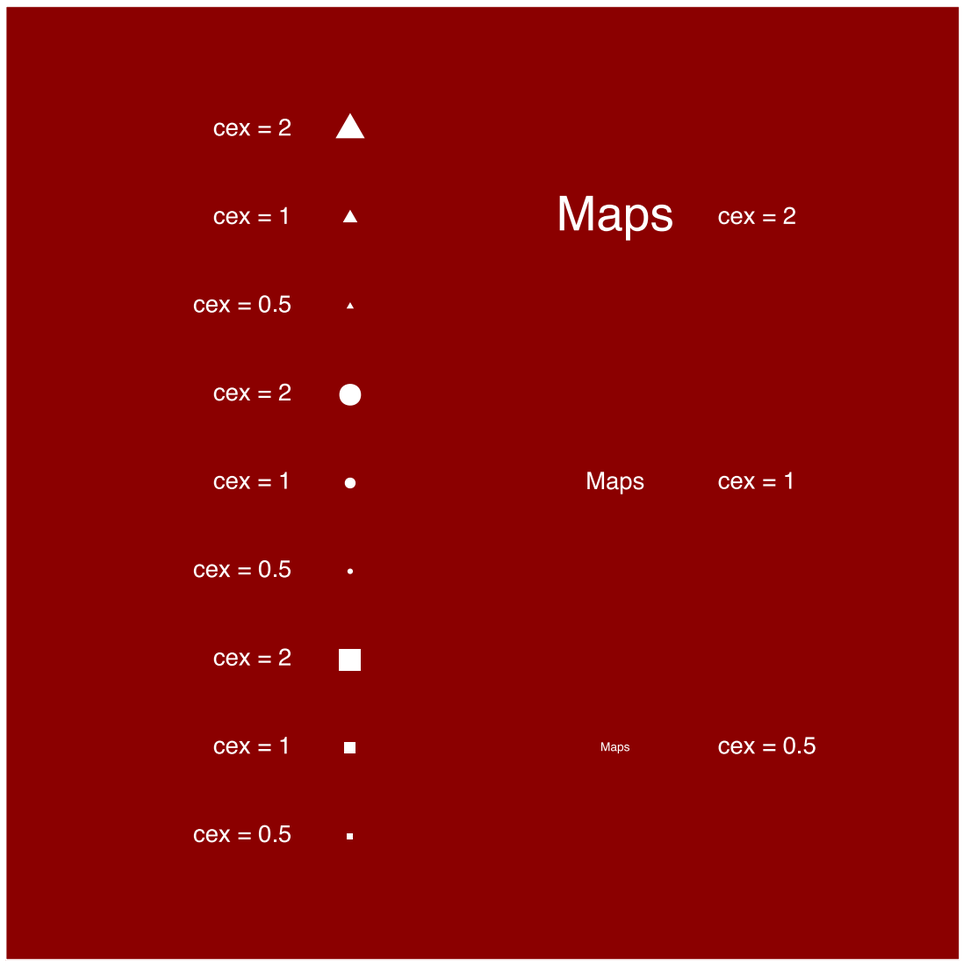
- ✓ `pdf()`;
- ✓ Cairo :
  - `cairo_pdf()`
  - `cairo_ps()`
  - `svg()`

Exporting figures as vector files

```
pdf(fil, pointsize=12,
  height=2*7, width=7)
...
dev.off()
```

Exporting figures as vector files

```
svg(filename, pointsize=12,
  height=2*7, width=7)
...
dev.off()
```



Resources

- ✓ CRAN task view fro graphs
- ✓ more packages indexed
- ✓ ggplot2 website
- ✓ Color: An interesting blog post

Three challenges

1. "refaire une figure du samedi"
2. a fig quite complicated...
3. Code your own boxplot function.