STAT 210
Applied Statistics and Data Analysis:
Graphics in R
Low-level commands and graphical windows

Joaquín Ortega

joaquin.ortegasanchez@kaust.edu.sa

Low-level commands

Low-level commands

So far we have looked at high-level plotting commands in R, which are the functions that produce graphs.

Low level commands are useful to add information to existing plots.

Legends

Legends

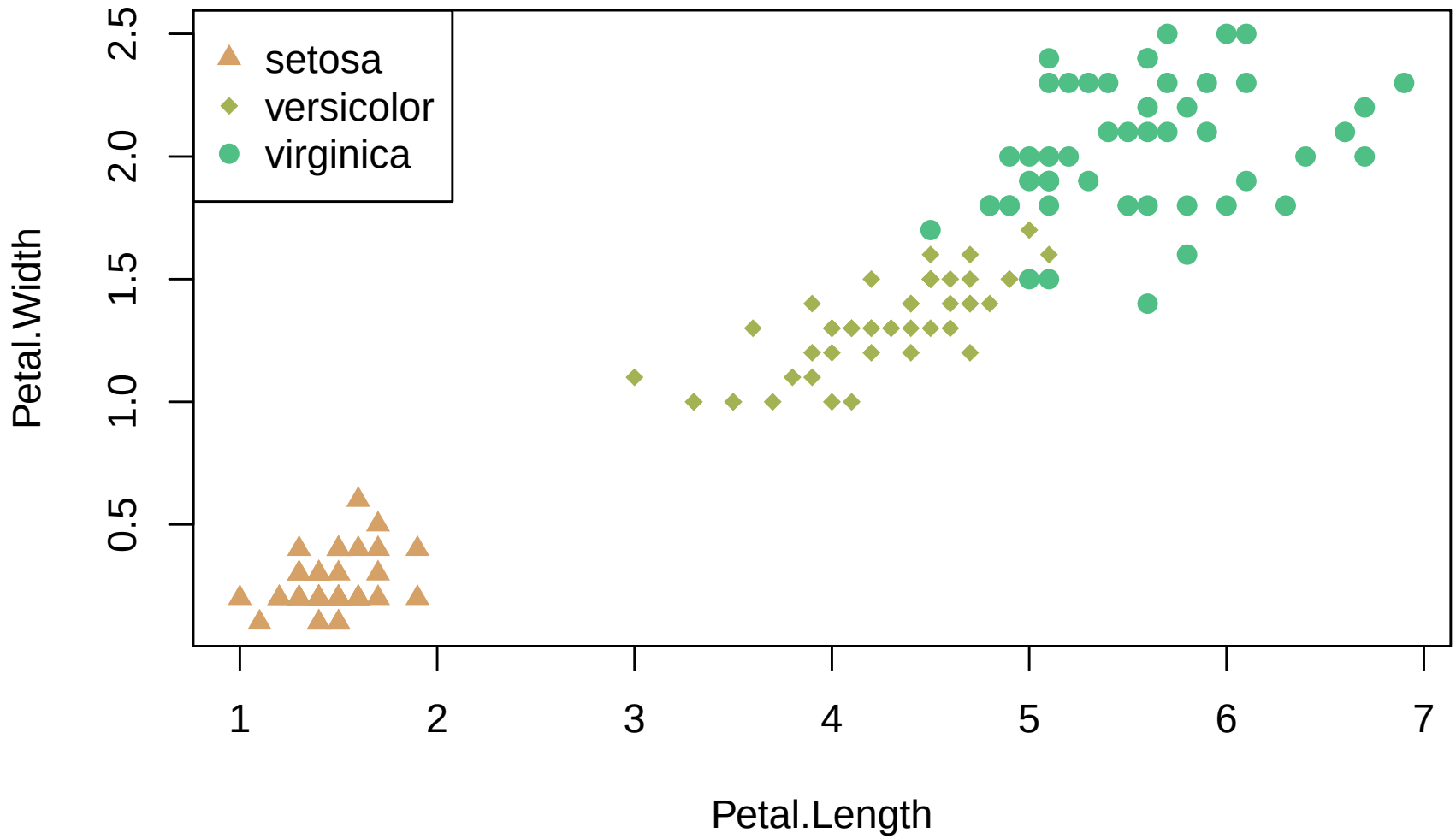
The command `legend` adds a legend to a plot in `R`. Syntax:

```
legend(x, y = NULL, legend, fill, col, bg)
```

- x and y: coordinates to position the legend
- legend: text of the legend
- fill: colors to use for filling the boxes beside the legend text
- col: colors of lines and points in the legend text
- bg: the background color for the legend box.

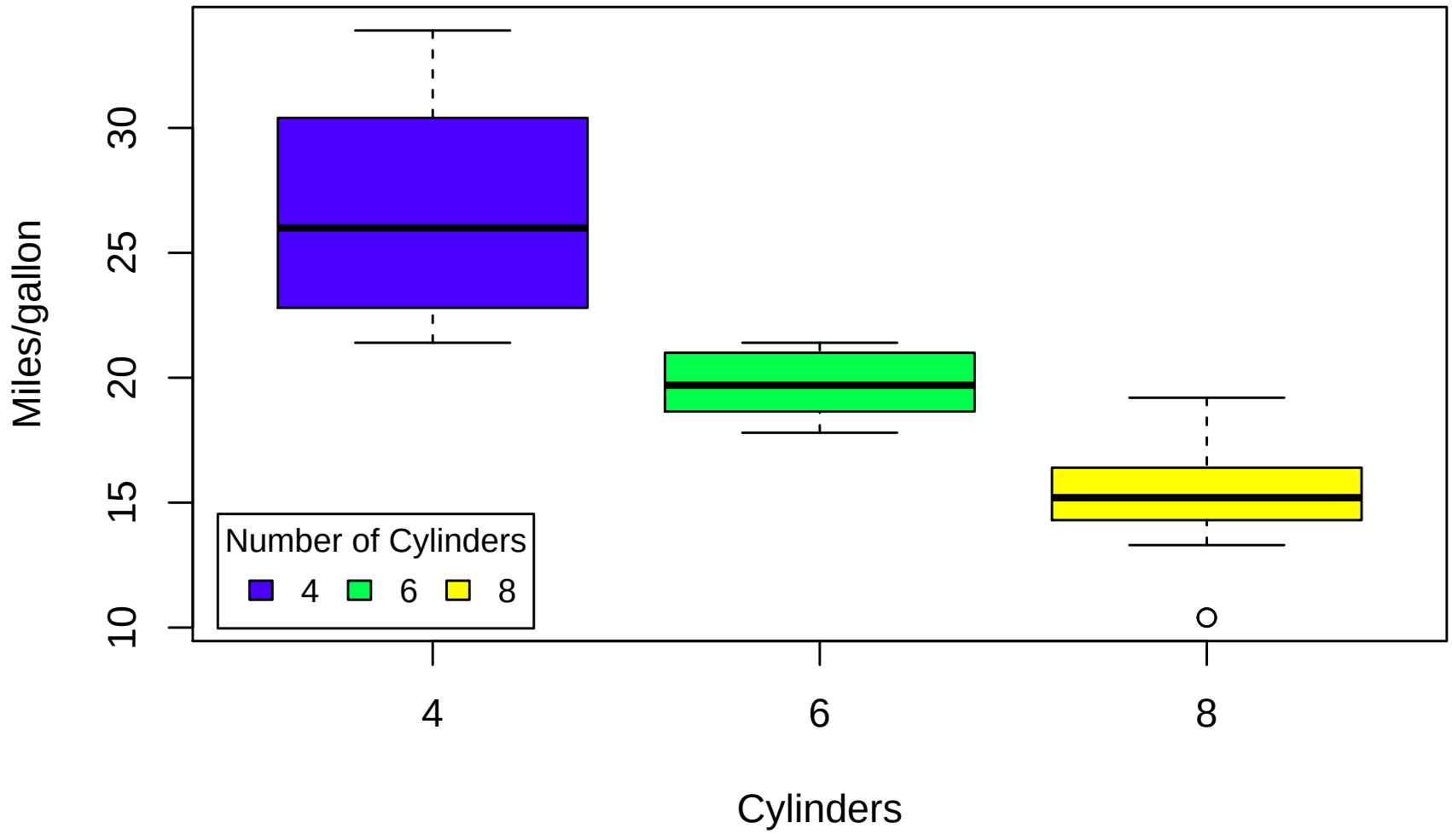
Legends

```
cols <- hcl.colors(15, "Set 2")
with(iris, plot(Petal.Length,Petal.Width,
     pch=16+as.numeric(Species),
     col = cols[1+2*as.numeric(iris$Species)]))
legend('topleft',legend=c('setosa','versicolor','virginica'),
       col=cols[c(3,5,7)], pch=16+(1:3))
```



Legends

```
boxplot(mpg~cyl, xlab="Cylinders", ylab="Miles/gallon",
        col=topo.colors(3))
legend("bottomleft", inset=.02, title="Number of Cylinders",
   c("4","6","8"), fill=topo.colors(3), horiz=TRUE, cex=0.8)
```



points

points

The function `points` adds points to an active graph.

As an example we add points to a boxplot.

Since many repeated values overlap in the graph, we use the function `jitter` to add some noise in the horizontal direction, to be able to tell these points apart.

points

```
attach(iris)
boxplot(Petal.Length ~ Species, data = iris)
points(jitter(as.numeric(Species), factor = 0.2), Petal.Length,
       col = cols[c(6,8,10)][as.numeric(Species)])
```

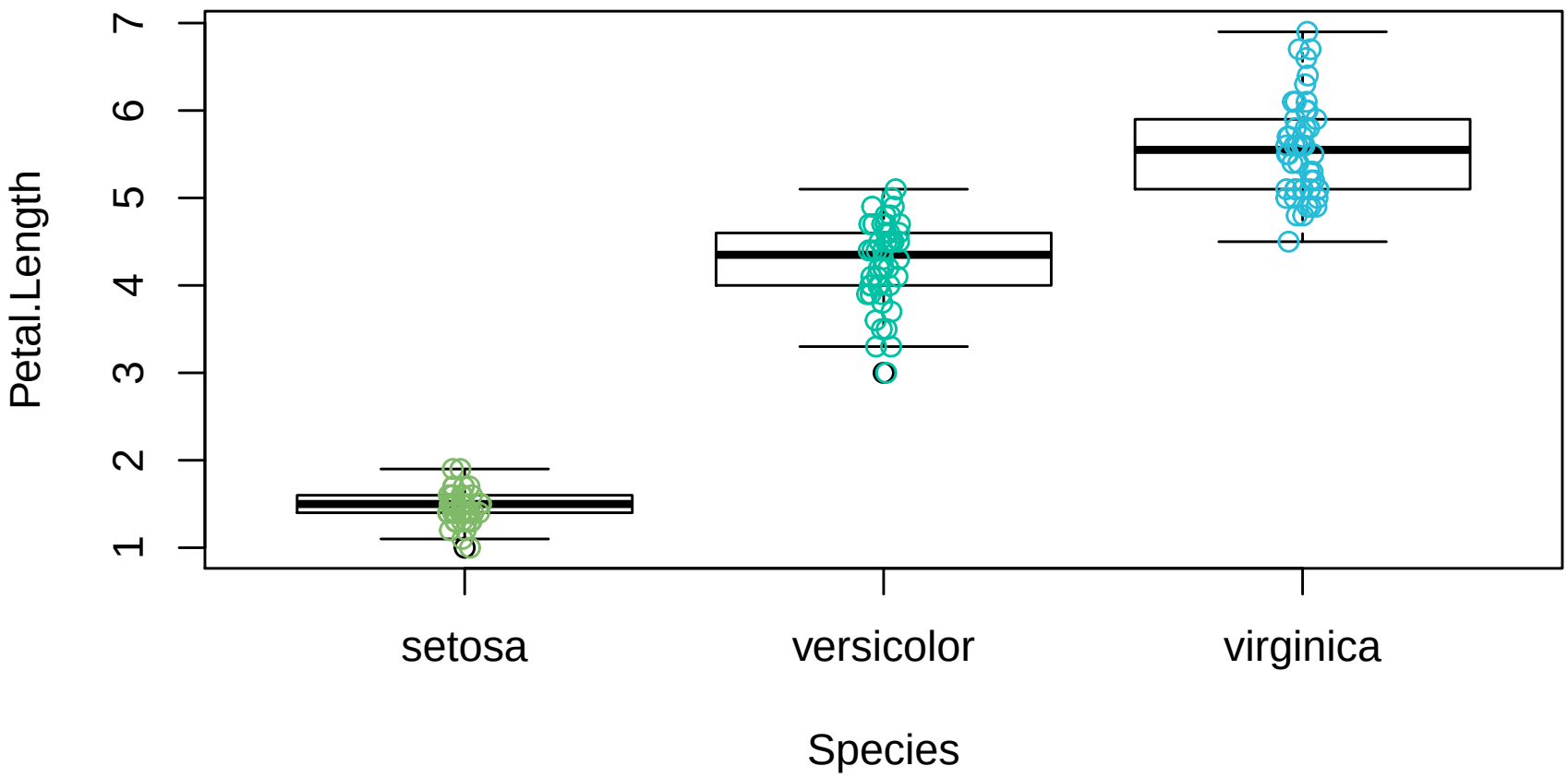


Figure 1: Boxplots of petal length as a function of species for the iris dataset. Points were added with the function `points()`

lines

`lines`

This function adds lines to an active graph.

The dataset `Indometh` has data on concentration (mcg/ml) of the compound indometacin as a function of time for six subjects.

We want to make a graph of the six concentration curves using different colors for each subject. One way to do this is to plot first the curve for subject 1 using `plot` and then add the other curves using `lines` in a `for` loop.

lines

```
plot(conc[Subject == 1] ~ time[Subject == 1],
     data = Indometh,col=cols[2],
     type = 'o', ylim = c(0,2.7), xlab = 'time (hr)',
     ylab = 'concentration (mcg/ml)',
     main='Concentration of indometacin')
for(i in 2:6){
  lines(conc[Subject == i] ~ time[Subject == i],
       data = Indometh, type = 'o', col = cols[2*i])
  }
legend('topright', legend = c('1','2','3','4','5','6'),
       col = cols[2*(1:6)], pch = rep(1,6), lwd = rep(1,6))
```

lines

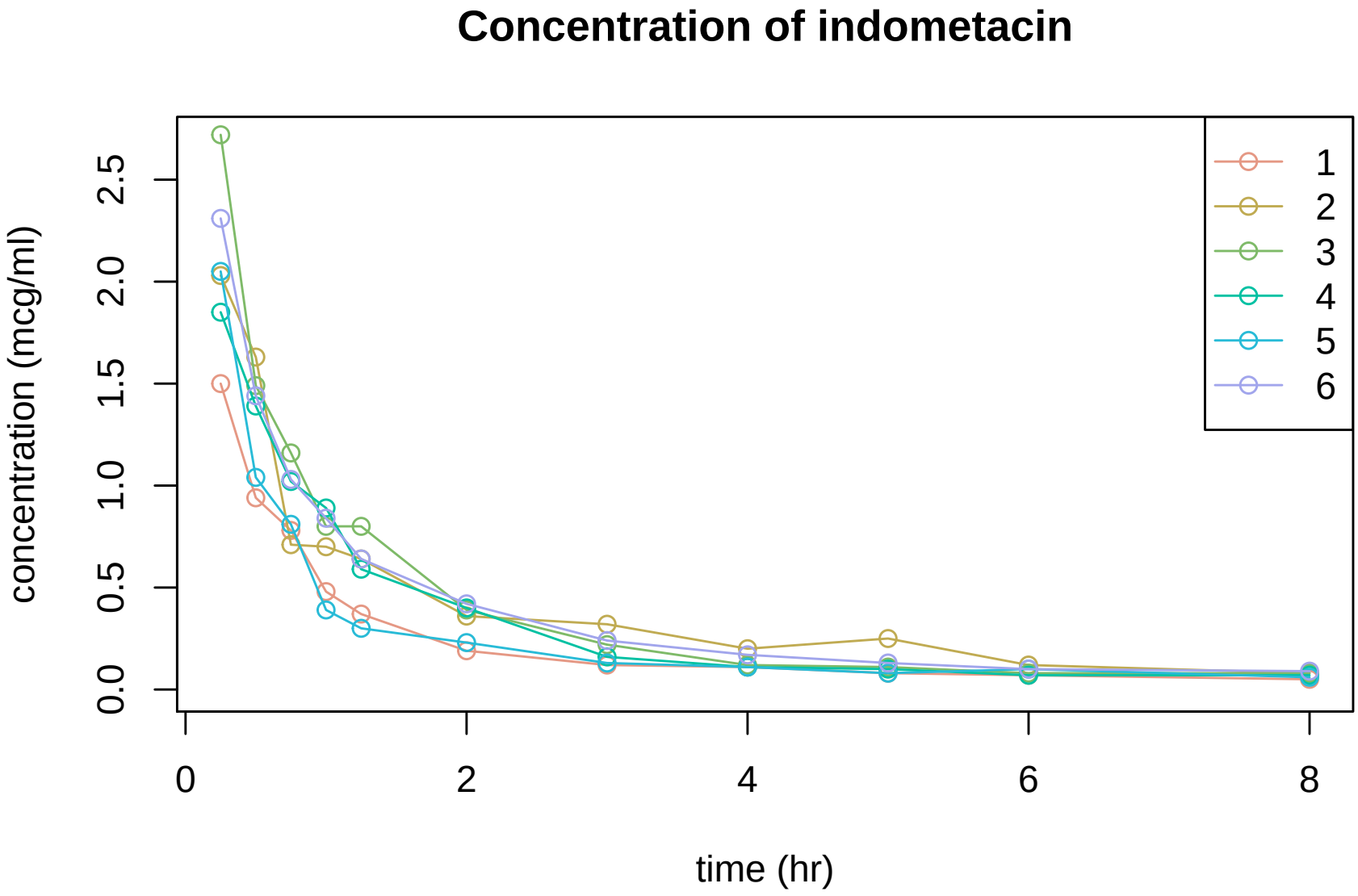


Figure 2: Example of the use of ‘lines()‘

abline

abline

`abline` draws straight lines. The syntax depends on the way the straight line is defined.

```
plot(cars); abline(h=60); abline(v=15)
```

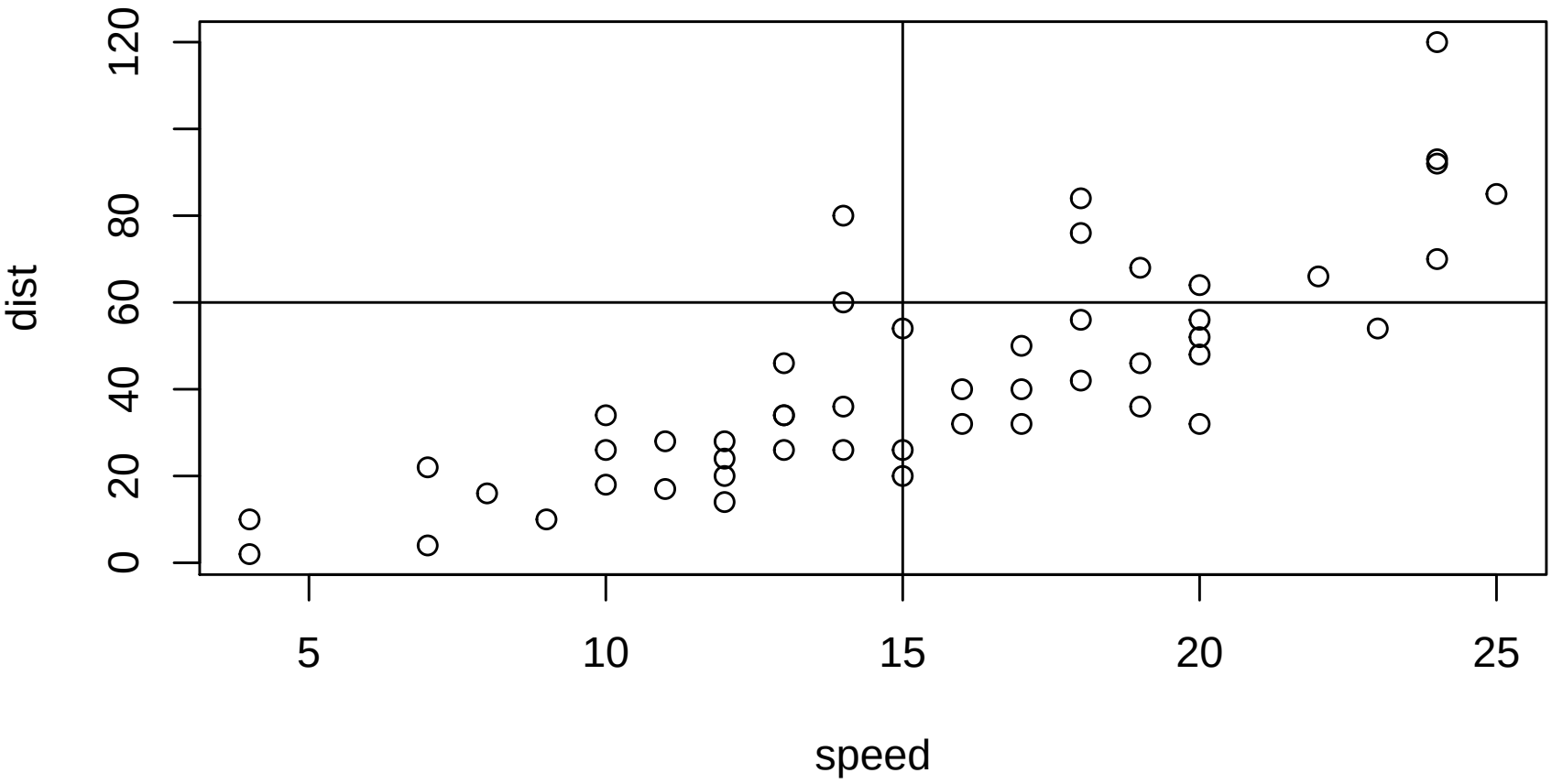


Figure 3: Example of the use of `abline`

abline

To overplot the regression line on a graph we can combine `abline` with the regression function `lm`,

```
plot(cars)
abline(reg = lm(dist ~ speed, data = cars))
```

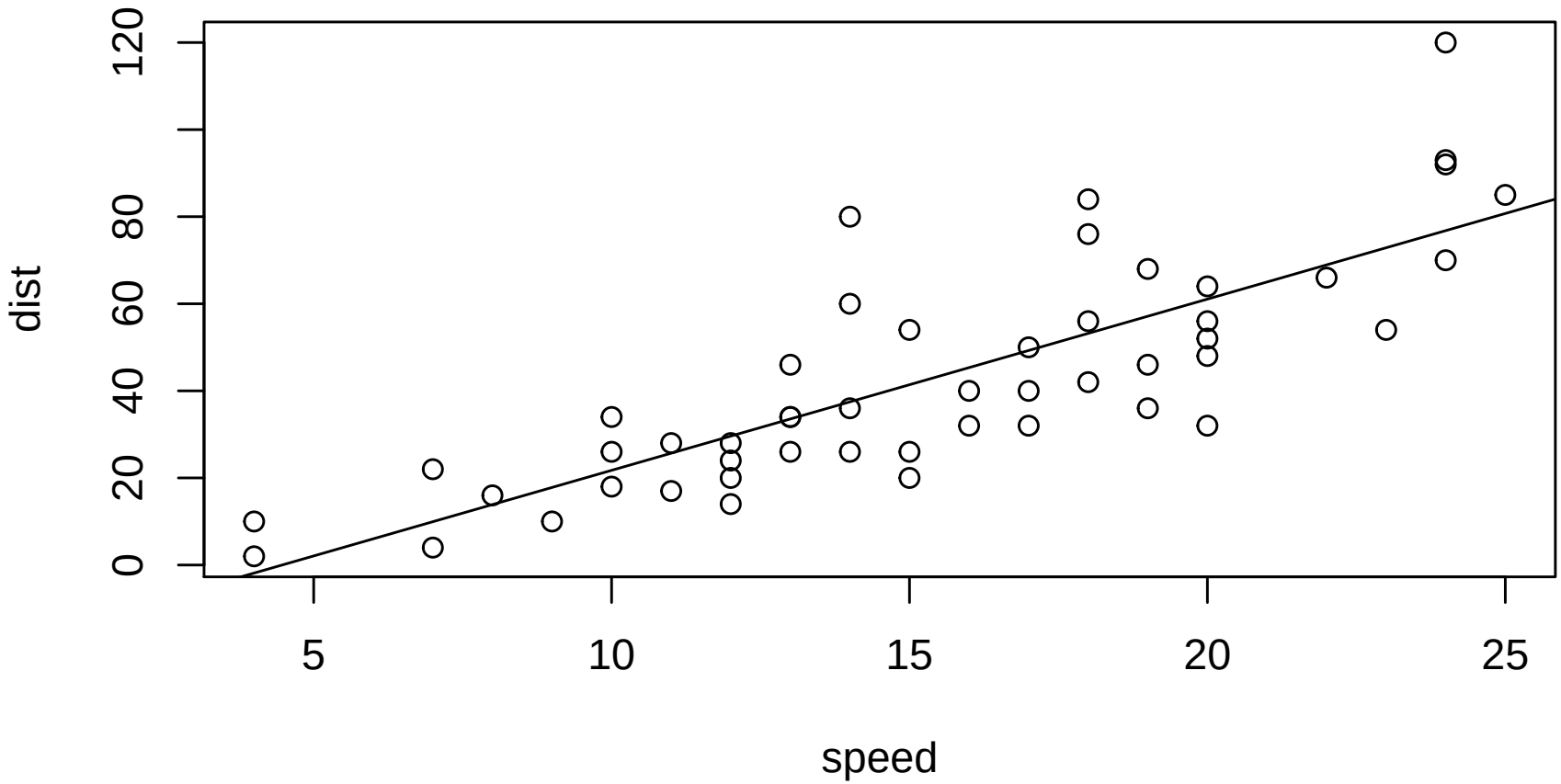
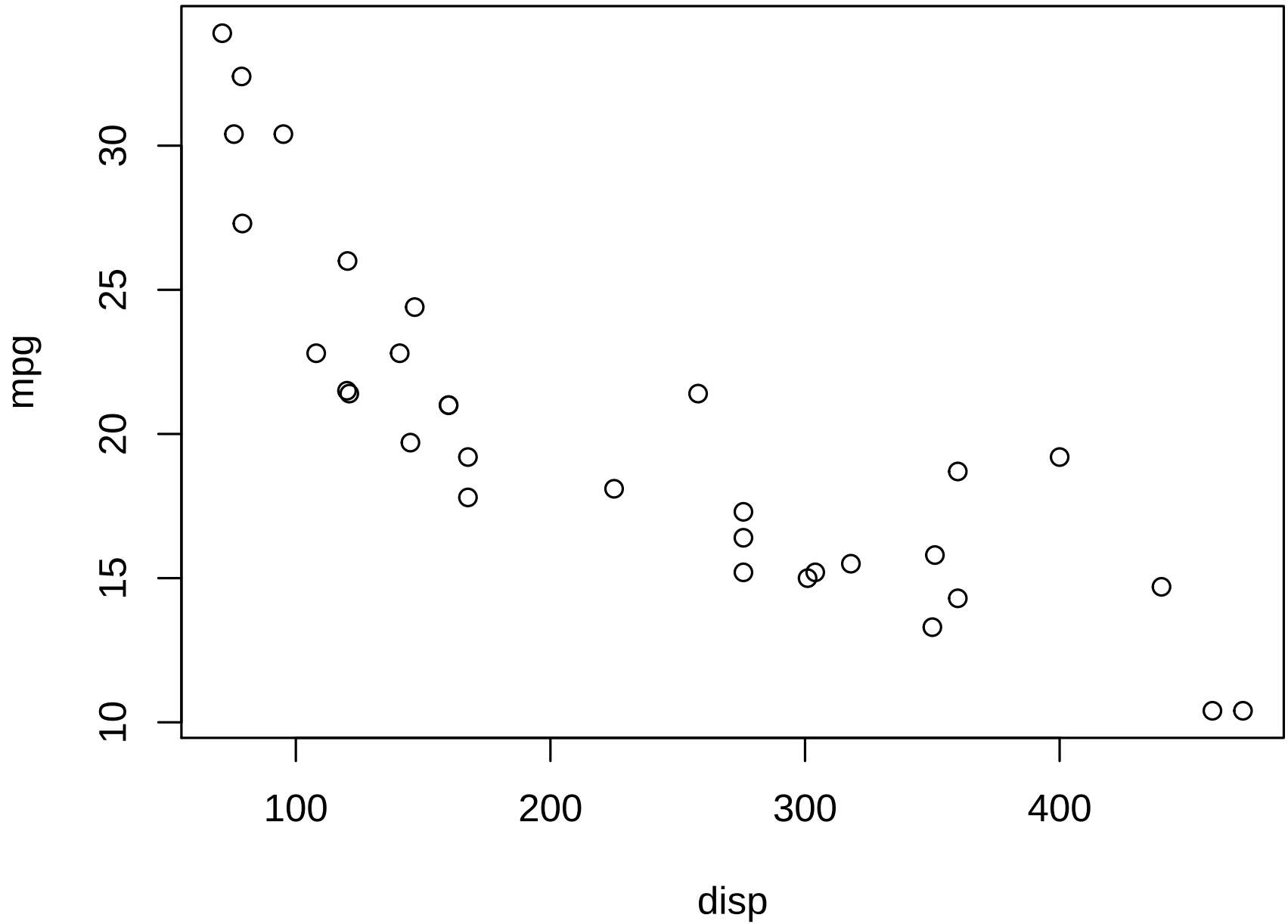


Figure 4: Example of the use of `abline` to add a regression line to a scatterplot

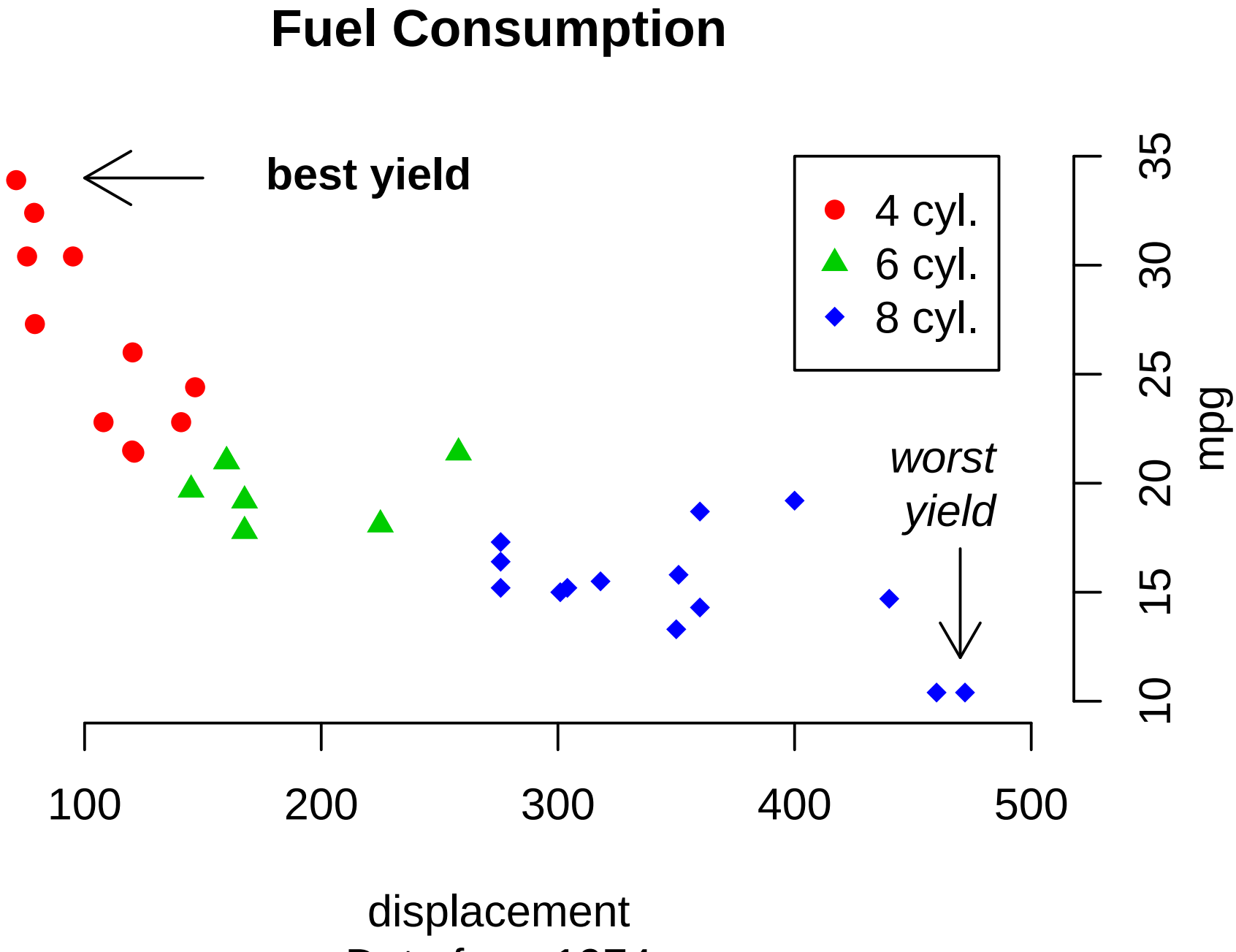
Other Commands

Other Commands

```
with(mtcars,plot(disp, mpg))
```



Other Commands



Other Commands

```
par(mar = c(5,4,4,4.5))
with(mtcars, {
plot(disp, mpg, type='n',axes=F, ylim = c(10,35), xlim = c(
points(disp[cyl==4], mpg[cyl==4], pch=16, col=2)
points(disp[cyl==6], mpg[cyl==6], pch=17, col=3)
points(disp[cyl==8], mpg[cyl==8], pch=18, col=4)
axis(1)
axis(4)
mtext('mpg',4, line = 2)
title('Fuel Consumption', 'Data from 1974')
arrows(470, 17, 470, 12, code = 2, length = 0.15)
text(485,20,'worst\nyield',font=3,adj=1)
arrows(100,34, 150,34, code=1, length = 0.2)
text(220, 34, 'best yield', font=2)
leg.txt <- c('4 cyl.', '6 cyl.', '8 cyl.')
legend(400,35, leg.txt,col=2:4, pch=16:18)
})
```

Graphical Parameters

Graphical Parameters

In addition to low-level commands, it is possible to modify the display of graphs using graphical parameters.

These can be used as graphical function options (but this doesn't always work) or with the function `par` to permanently change the parameters, i.e. the graphs that are made next will also use the new parameters.

There are 73 graphical parameters. The complete list can be viewed using the `?par` instruction.

We will not review them in detail and will only give some examples

Graphical Parameters

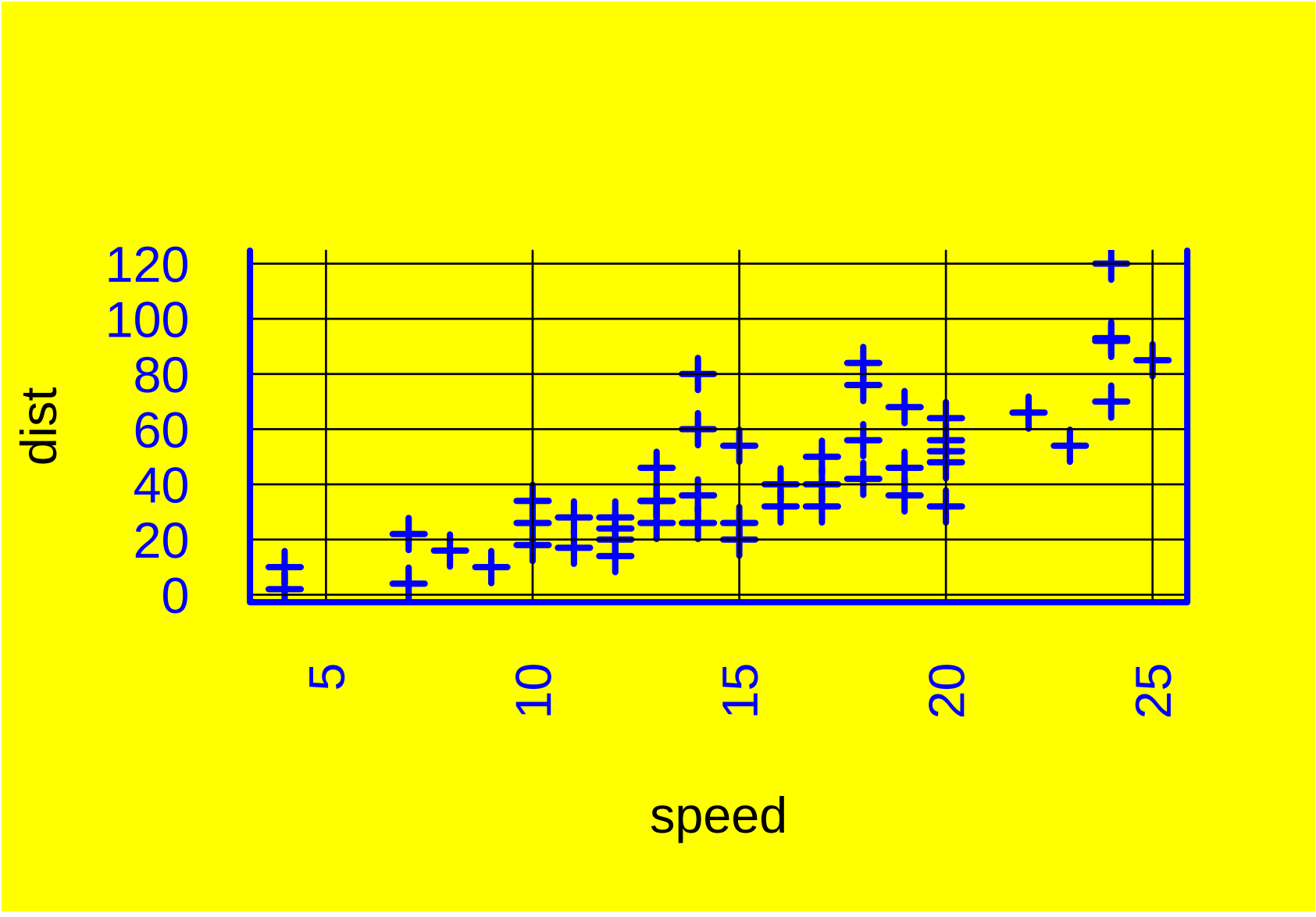
```
par()
```

Gives a list of the current values for all the parameters.

```
old.par <- par(no.readonly = TRUE)
par(bg=7, bty='u', cex=1.5, col='blue', col.axis=4,
    font=2,lty='dashed', lwd=3, pch=3, las=2, tck =1)
```

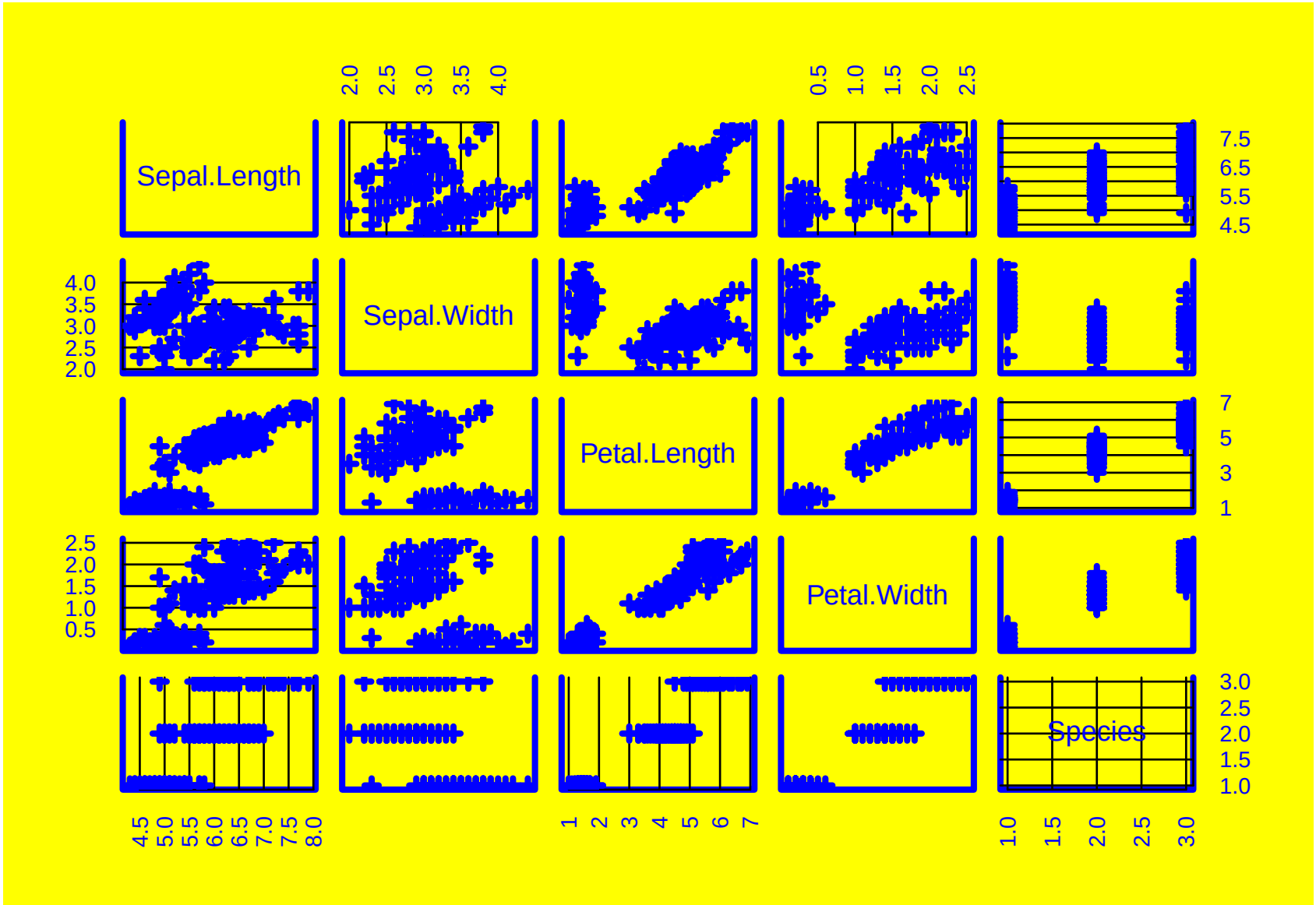
Stores in `old.par` the current values for the parameters that can be set by the user (some parameters can only be read but not changed).

Graphical Parameters



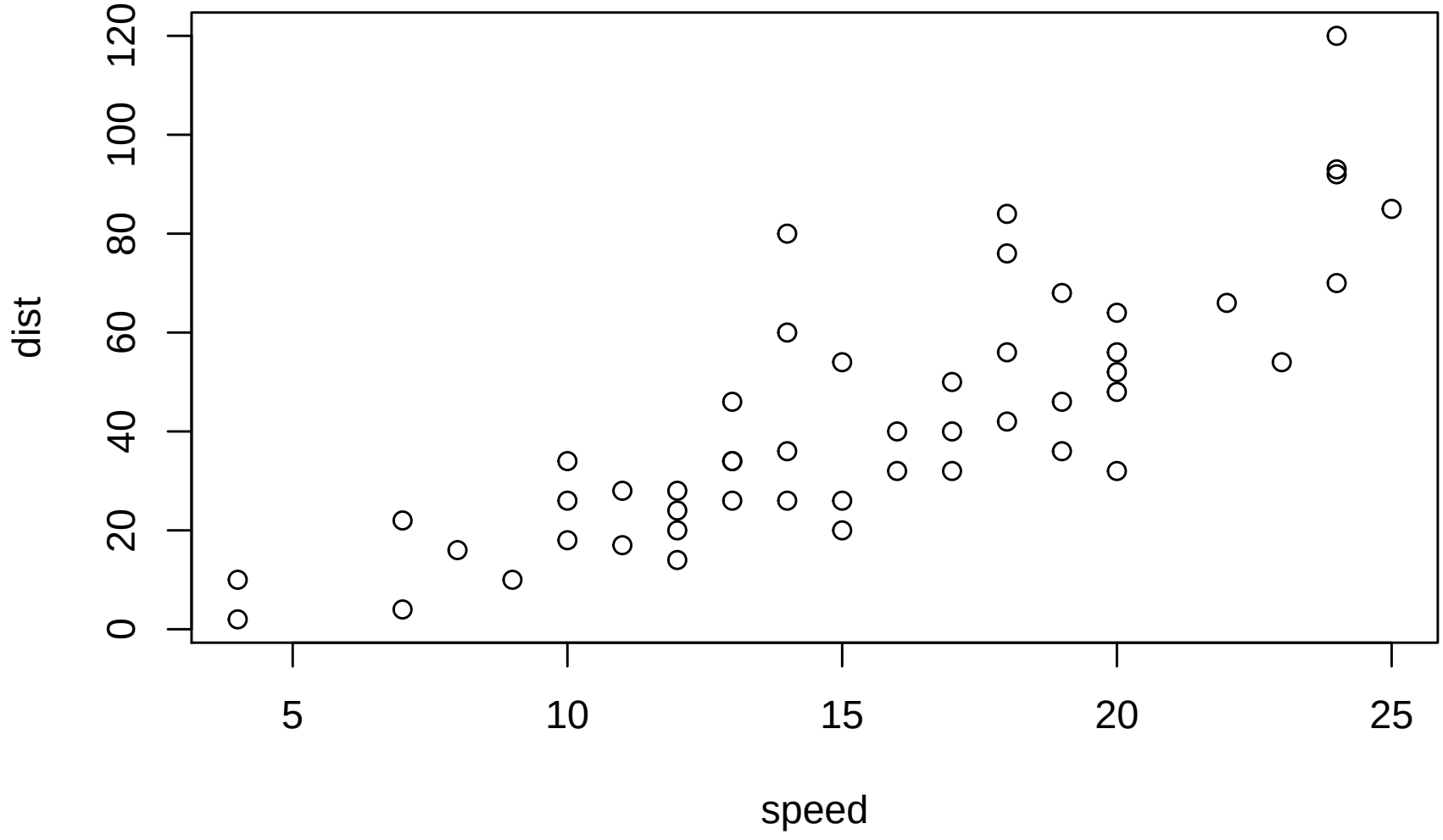
Graphical Parameters

```
plot(iris)
```



Graphical Parameters

```
par(old.par); plot(cars)
```



Graphical Windows

Graphical Windows

There are a few ways to split a graphical window to display several graphs simultaneously.

One possibility is to modify the graphical parameters using the function `par()` with the arguments `mfrow` or `mfcol`.

Another possibility is to use the `split.screen(c(m,n))` instruction that divides the window into m rows and n columns. Panels can be selected with `screen(r)` for $1 \leq r \leq m \cdot n$.

`erase.screen()` deletes the last graph.

These functions are incompatible with functions such as `coplot` or `layout` and should not be used with multiple graphics devices.

Graphical Windows

```
split.screen(c(2, 1))          # split display into two screens
split.screen(c(1, 2), 2)       # split bottom half in two
plot(1:10)                     # screen 3 is active, draw plot
erase.screen()                 # forgot label, erase and redraw
plot(1:10, ylab = "ylab 3")
screen(1)                      # prepare screen 1 for output
plot(1:10)
screen(4)                      # prepare screen 4 for output
plot(1:10, ylab = "ylab 4")
screen(1, FALSE)             # return to screen 1, but do not clear
plot(10:1, axes = FALSE,
     lty = 2, ylab = "")   # overlay second plot
axis(4)                        # add tic marks to right-hand axis
title("Plot 1")
close.screen(all = TRUE)       # exit split-screen mode

plot(1:10)
```

layout

layout

Another function that allows you to divide the graphic window is `layout`, which divides it into several panels in which the graphics will be drawn successively.

The argument is a matrix of integers that indicates the number of divisions. For example, to divide the device into four parts, we can use

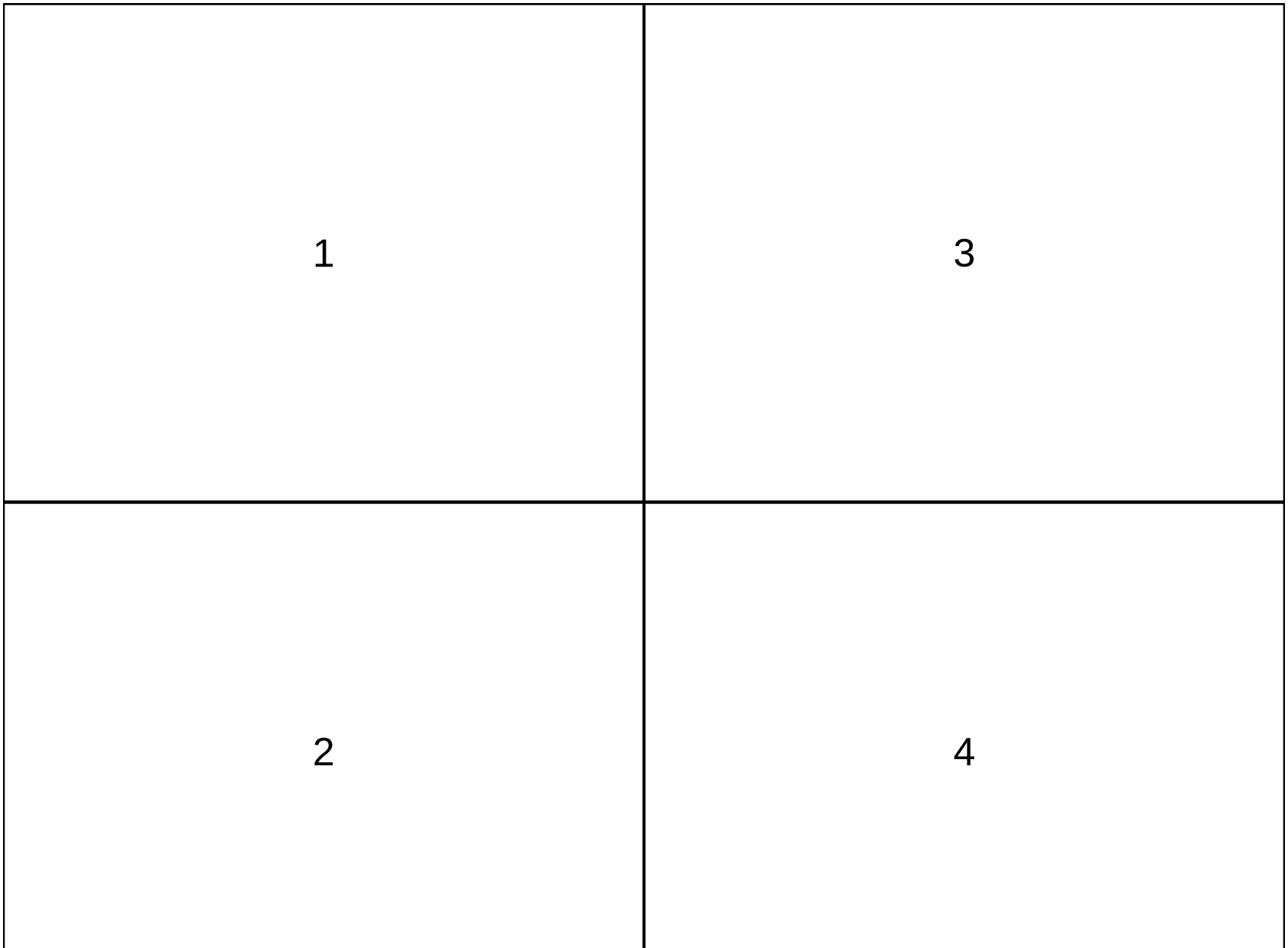
```
(mat1 <- matrix(1:4,2,2))
```

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

```
layout(mat1)
```

layout

```
layout.show(4)
```



layout

The following examples show some of the possibilities

```
layout(matrix(1:6,3,2))
layout.show(6)
layout(matrix(1:6,2,3))
layout.show(6)
layout(matrix(1:6,3,2,byrow=TRUE)) > layout.show(6)
(m <- matrix(c(1:3,3), 2, 2))
layout(m)
layout.show(3)
```

layout

By default `layout()` divides the windows in equal-sized panels, but this can be modified with the options `widths` and `heights`

```
m <- matrix(1:4, 2, 2)
layout(m, widths=c(1,3), heights=c(3,1))
layout.show(4)
m <- matrix(c(1,1,2,1), 2, 2)
layout(m, widths=c(2,1), heights=c(1,2))
layout.show(2)
```

Matrix entries may take the value 0, allowing more complex layouts:

```
m <- matrix(0:3, 2, 2)
layout(m, widths=c(1,3), heights=c(1,3))
layout.show(3)
```

layout

```
nf <- layout(matrix(c(2,0,1,3),2,2,byrow=TRUE),
             c(3,1), c(1,3), TRUE)
layout.show(nf)
par(mar=c(2,2,2,2))
 x <- pmin(3, pmax(-3, rnorm(50)))
 y <- pmin(3, pmax(-3, rnorm(50)))
 xrange <- c(-3,3)
 yrange <- c(-3,3)
 xhist <- hist(x,breaks=seq(-3,3,0.5),plot=FALSE)
 yhist <- hist(y,breaks=seq(-3,3,0.5),plot=FALSE)
 top <- max(c(xhist$counts, yhist$counts))

plot(x, y, xlim=xrange, ylim=yrange, xlab='',
             ylab='')
barplot(xhist$counts, axes=FALSE,
             ylim=c(0, top), space=0)
barplot(yhist$counts, axes=FALSE,
             xlim=c(0, top), space=0, horiz=TRUE)
```

locator

locator

This function allows the user to click within a graph and obtain the coordinates of the selected point. It is also possible to use it to place symbols where you click or to draw segments between the selected points. The syntax is

```
locator(n,type)
```

With this instruction, R expects the user to select *n* points on the active graph. The argument `type` allows us to draw on the selected points and has the same syntax as for high-level graphical commands. The default option is not to draw anything.

The `locator()` result is the coordinates of the points selected as a list with two components.

locator

```
plot(cars)
locator(3,type='n')
locator(2,type='l')
text(locator(1), 'Punto', adj=0)
```

identify

identify

This function can be used to identify data in a graph. The data closest to the mouse position is identified by clicking.

```
identify(x, y, labels).
```

The procedure is similar to locator, but instead of identifying the coordinates, the point is identified through labels. If labels are not present in the function call, the row in which the data is in the matrix is used as a label.

To finish the identification process press the right mouse button, and select Stop, or press escape.