Notes on using R

Pedro J. Aphalo

Git: tag ‘none’, committed with hash ‘none’ on ‘none’

Contents

Preface

This series of Notes cover different aspects of the use of R. They are meant to be use as a complement to a course or book, as explanations are short and terse. We do not discuss here statistics, just R as a tool and language for data manipulation and display.

1 Plots with ggplot2

1.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(plyr)
library(grid)
library(Hmisc)

## Loading required package:  lattice
## Loading required package:  survival
## Loading required package:  Formula
## Loading required package:  ggplot2
##
## Attaching package:  'Hmisc'
##
## The following objects are masked from 'package:plyr':
##
##    is.discrete, summarize
##
## The following objects are masked from 'package:base':
##
##    format.pval, round.POSIXt, trunc.POSIXt,
##    units

library(ggplot2)
library(scales)
library(rgdal)

## Loading required package:  methods
## Loading required package:  sp
## rgdal:  version:  1.1-3, (SVN revision 594)
##  Geospatial Data Abstraction Library extensions to R successfully
loaded
##  Loaded GDAL runtime:  GDAL 2.0.1, released 2015/09/15
##  Path to GDAL shared files:  C:/Program Files/R/R-3.2.3/library/rgdal/gdal
##  GDAL does not use iconv for recoding strings.
##  Loaded PROJ.4 runtime:  Rel.  4.9.1, 04 March 2015, [PJ_VERSION:
491]
##  Path to PROJ.4 shared files:  C:/Program Files/R/R-3.2.3/library/rgdal/proj
##  Linking to sp version:  1.2-1
```

1.2 Introduction

Being R extensible, in addition to the built-in plotting functions, there are several alternatives provided by packages. Of the general purpose ones, the most extensively used are `Lattice` and ggplot2. There are additional packages that add extra functionality to these packages.

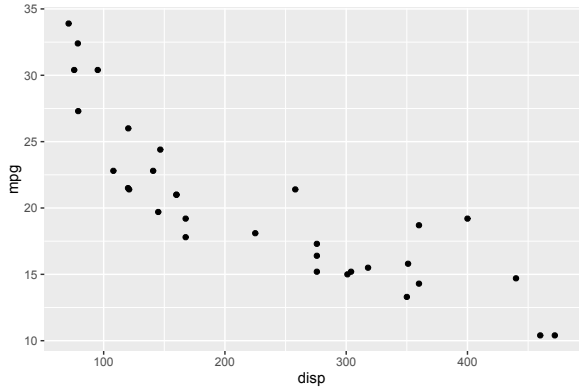
In the examples in this handbook we mainly use ggplot2, ggmap and ggtern. In this chapter we give an introduction to the 'grammar of graphics' and ggplot2. There is ample literature on the use of ggplot2, starting with very good reference documentation at `http://ggplot2.org/`. The book 'R Graphics Cookbook' Chang 2013 is very useful and should be always near you, when using the package, as it contains many worked out examples. Much of the literature available at this time is for older versions of ggplot2 but we here describe version 2.0.0, and highlight the most important incompatibilities that need to be taken into account when using older code as example. There is little well-organized literature on packages extending ggplot2 so we will describe some of them in later chapters.

1.3 Bases of plotting with ggplot2

What separates ggplot2 from base-R and trellis/lattice plotting functions is the use of a grammar of graphics (the reason behind 'gg' in the name of the package). What is meant by grammar in this case is that plots are assembled piece by piece from different 'nouns' and 'verbs'. Instead of using a single function with many arguments, plots are assembled by combining different elements with operators + and %+%. Furthermore, the constructions is mostly semantic-based and to a large extent how the plot looks when is printed, displayed or exported to a bitmap or vector graphics file is controlled by themes.

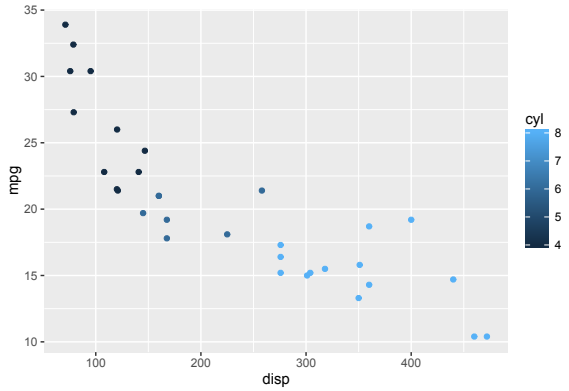
The grammar of graphics is based on aesthetics (`aes`) as for example color, geometric elements `geom_...` such as lines, and points, statistics `stat_...`, scales `scale_...`, labels `labs`, and themes `theme_...`. Plots are assembled from these elements, we start with a plot with two aesthetics, and one geometry. In the examples that follow we will use the `mtcars` data set included in R.

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) + geom_point()
```



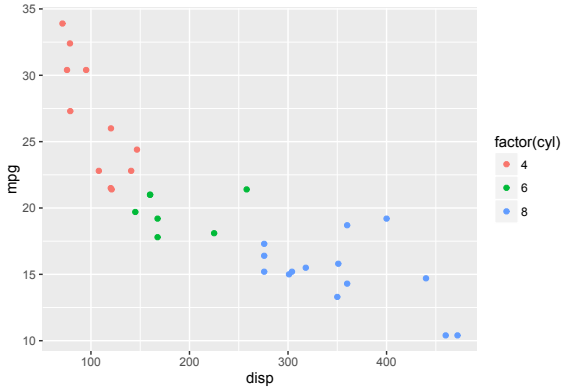
Aesthetics can be 'linked' to data variables, either continuous (numeric) or categorical (factor). Variable `cyl` is encoded in the `mtcars` dataframe as numeric values. Even though only three values are present, a continuous color scale is used.

```
ggplot(mtcars, aes(x = disp, y = mpg, colour = cyl)) + geom_point()
```



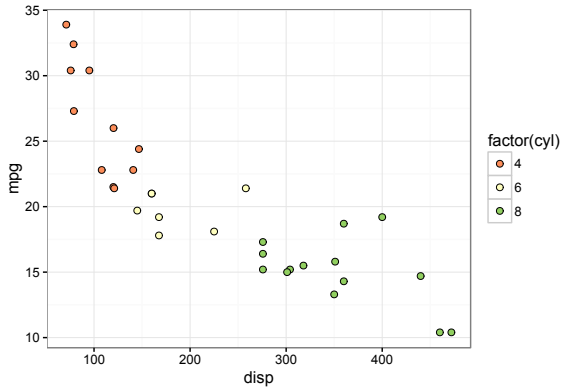
We can convert `cyl` into a factor 'on-the-fly' to force the use of a discrete color scale.

```
ggplot(mtcars, aes(x = disp, y = mpg, colour = factor(cyl))) + geom_point()
```



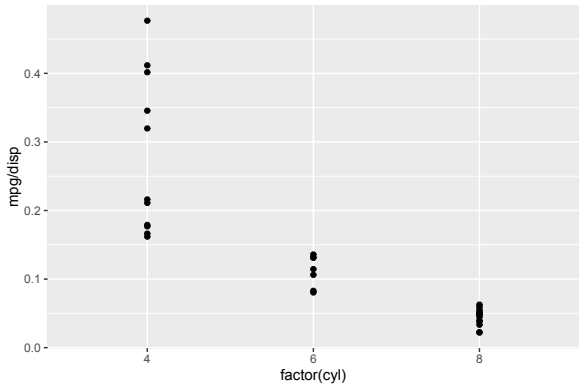
Each aesthetic, involves mapping of values in the data to aesthetic values such as colours. The mapping is defined by means of scales. If we now consider the `colour` aesthetic in the previous statement, a default discrete colour scale was used. In this case if we would like different colours used for the three values, but still have then selected automatically, we can select a different colour palette:

```
ggplot(mtcars, aes(x = disp, y = mpg, fill = factor(cyl))) +
  geom_point(shape = 21, size = rel(2)) +
  scale_fill_brewer(type = "div", palette = 8) +
  theme_bw()
```



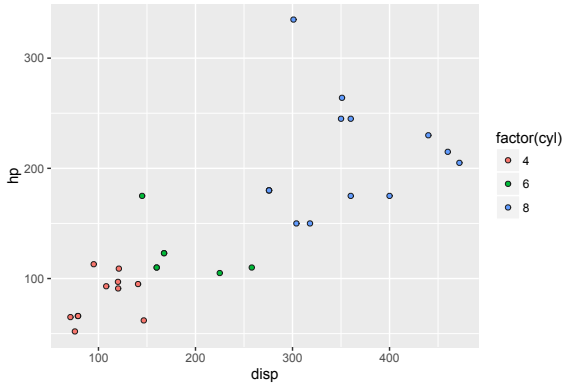
Data assigned to an aesthetic can be the ‘result of a computation’.

```
ggplot(mtcars, aes(x = factor(cyl), y = mpg / disp)) + geom_point()
```



Within `aes` the aesthetics are interpreted as being a function of the values in the data. If given outside `aes` they are interpreted as constants, which apply to one geom if given within the call to `geom_xxx` but outside `aes` or to the whole plot if given within `ggplot` but outside `aes`. The aesthetics and data given as `ggplot`'s arguments become the defaults for all the geoms, but geoms also accept aesthetics and data as arguments, which when supplied as arguments override the defaults.

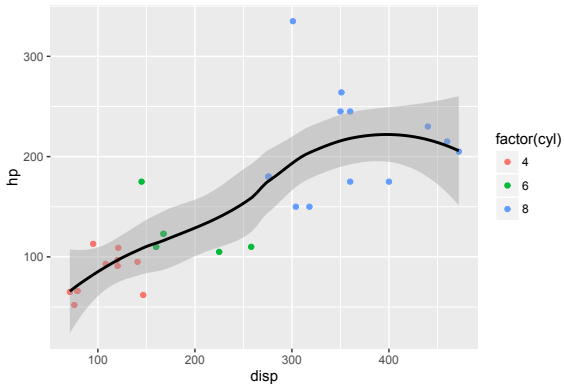
```
ggplot(mtcars, aes(x = disp, y = hp, fill = factor(cyl))) +
  geom_point(shape=21, colour="grey10")
```



In the next example we override the `color` aesthetic in `geom_smooth`[1], causing all the data to be fitted together

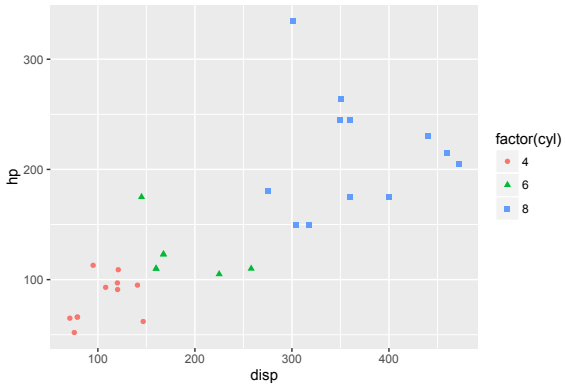
[1] Smoothing and curve fitted is discussed in more detail in section ??.

```
ggplot(mtcars, aes(x=disp, y=hp, colour=factor(cyl))) + geom_point() +
  geom_smooth(colour="black")
```



We can assign the same variable to more than one aesthetic, and the combined key will be produced automatically.

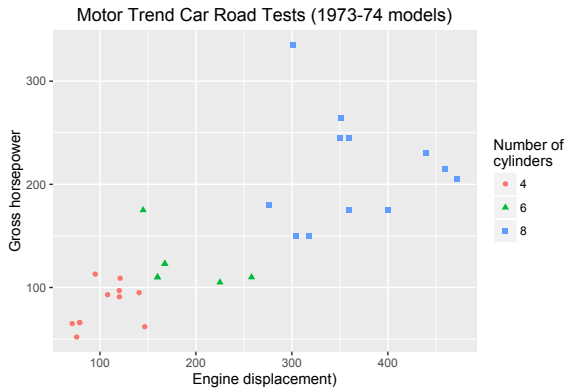
```
ggplot(mtcars, aes(x=disp, y=hp, colour=factor(cyl), shape=factor(cyl))) + geom_point(
```



We can change the labels for the different aesthetics, and give a title (\n means 'new line' and can be used to continue a label in the next line). In this case, if two aesthetics are linked to the same variable, the labels supplied should be identical, otherwise two separate keys will be produced.

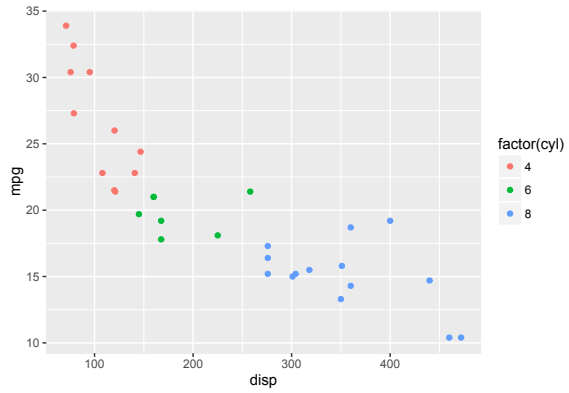
```
ggplot(mtcars, aes(x=disp, y=hp, colour=factor(cyl), shape=factor(cyl))) +
  geom_point() +
    labs(x="Engine displacement)",
       y="Gross horsepower",
       colour="Number of\ncylinders",
```

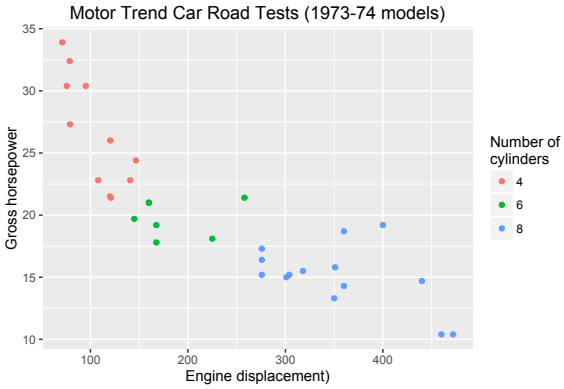
```
       shape="Number of\ncylinders",
       title="Motor Trend Car Road Tests (1973-74 models)")
```



We can assign a ggplot object or a part of it to a variable, and then assemble a new plot from the different pieces.

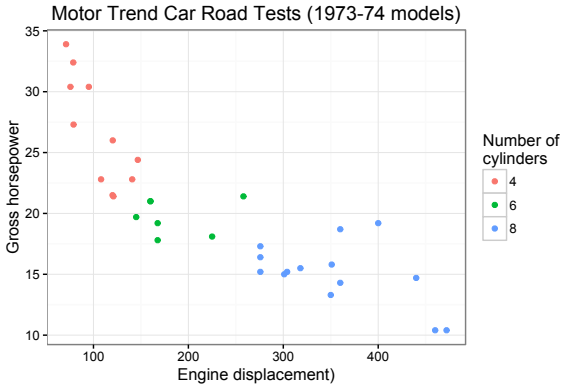
```
myplot <- ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_point()
mylabs <- labs(x="Engine displacement)",
       y="Gross horsepower",
       colour="Number of\ncylinders",
       shape="Number of\ncylinders",
       title="Motor Trend Car Road Tests (1973-74 models)")
myplot
myplot + mylabs
```

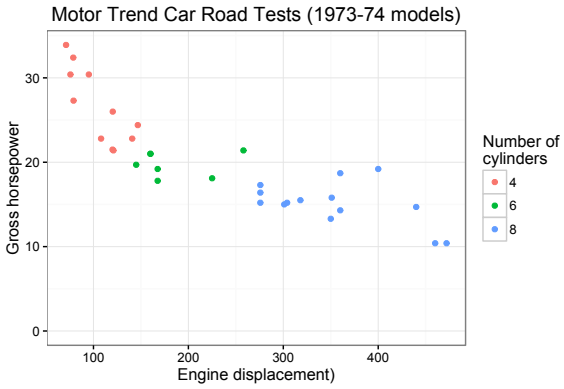




And now we can assemble them into plots.

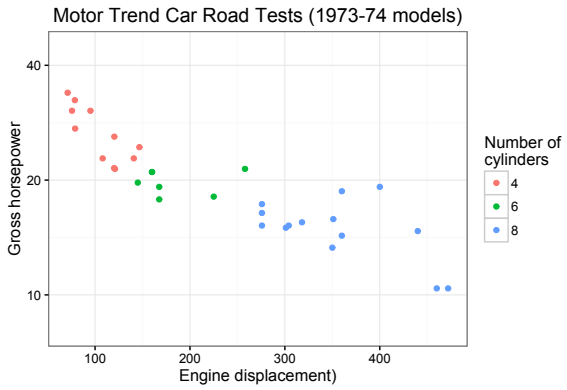
```
myplot + mylabs + theme_bw()
myplot + mylabs + theme_bw() + ylim(0, NA)
```





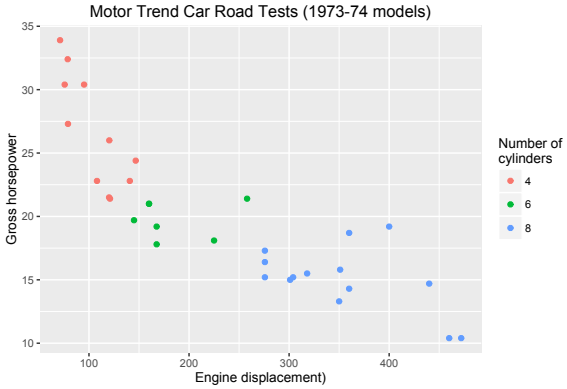
We can also save intermediate results.

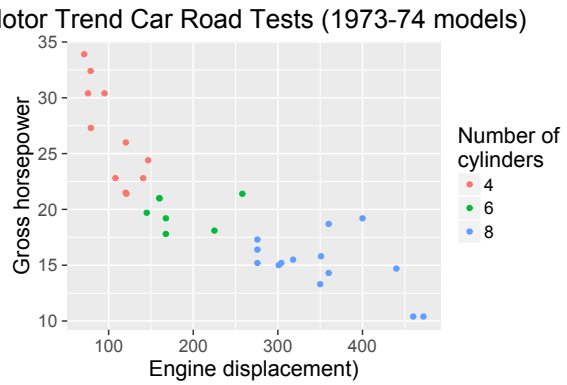
```
mylogplot <- myplot + scale_y_log10(breaks=c(10,20,40), limits=c(8,45))
mylogplot + mylabs + theme_bw()
```



The are a few predefined themes, even the default `theme_grey` can come in handy because the first parameter to themes is the point size used as reference to calculate all other font sizes. You can see in the two examples bellow, that the size of all text elements changes proportionally.

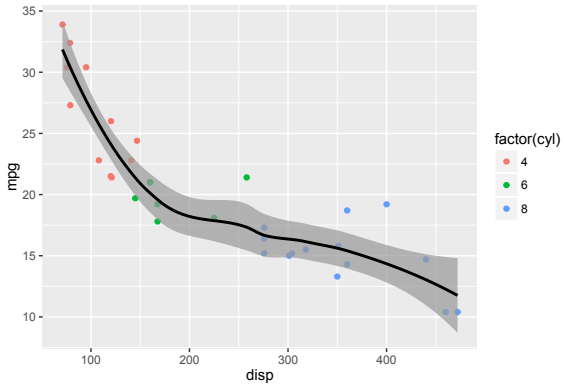
```
myplot + mylabs + theme_grey(10)
myplot + mylabs + theme_grey(16)
```





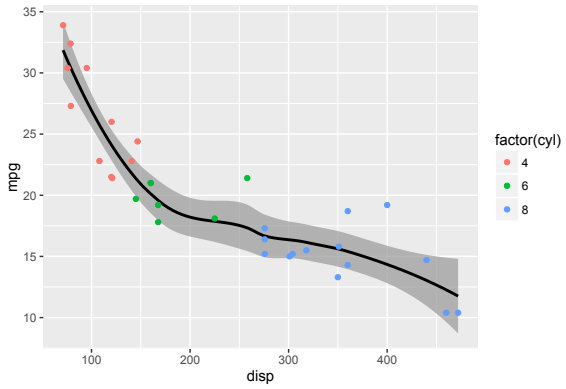
Be aware that the different geoms and elements can be added in almost any order to a ggplot object, but they will be plotted in the order that they are added. We use the `alpha` aesthetic to make the confidence band less transparent so that the example is easier to see in print.

```
ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_point() + geom_smooth(colour="black", alpha=0.7)
```



The plot looks different if the order of the geoms is swapped. The data points overlapping the confidence band are more clearly visible in this second example because they are above the shaded area instead of bellow it.

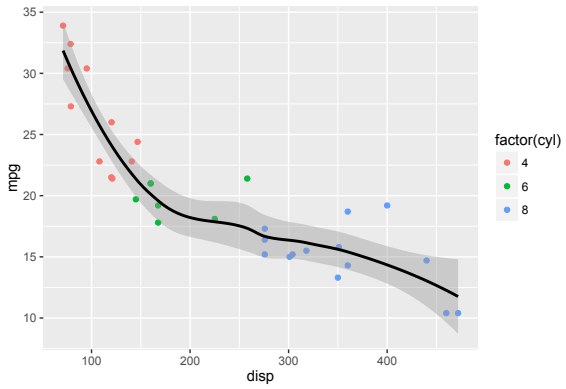
```
ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
   geom_smooth(colour="black", alpha=0.7) + geom_point()
```



1.4 Adding fitted curves, including splines

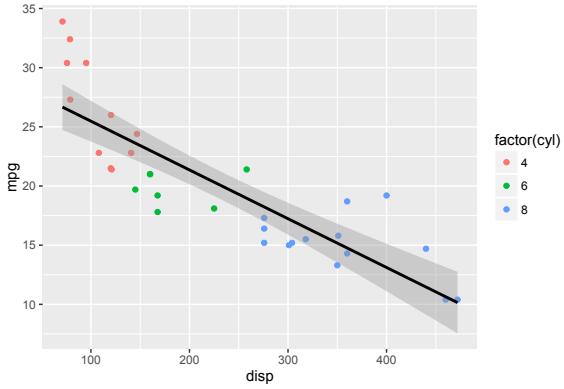
We will now show an example of use of `stat_smooth` using the default spline smoothing.

```
myplot + stat_smooth(colour="black")
```



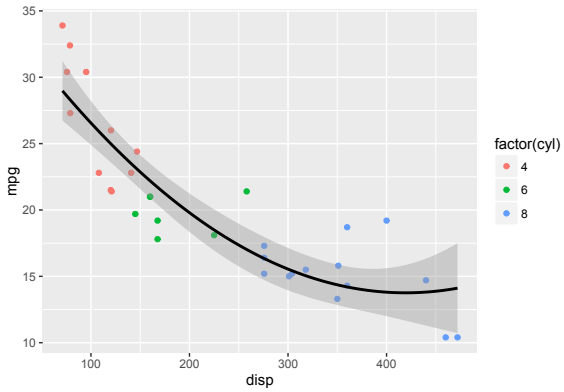
Instead of using the default spline, we can use a linear model fit. In this example we use a linear model, fitted by lm, as smoother:

```
myplot + stat_smooth(method="lm", colour="black")
```



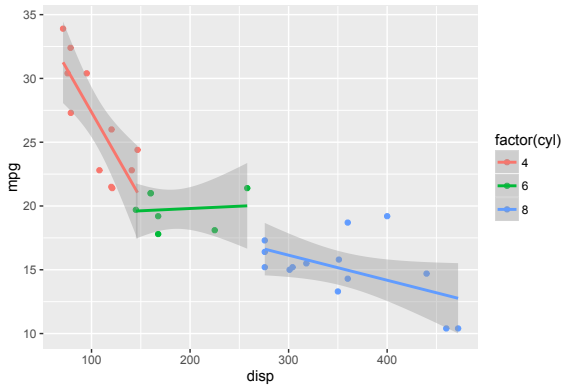
Instead of using the default linear regression as smoother, we can use a linear model fit. In this example we use a polynomial of order 2 fitted by lm.

```
myplot + stat_smooth(method="lm", formula=y~poly(x,2), colour="black")
```



If we do not use `colour="black"` then the colour aesthetics supplied to `ggplot` is used, and splits the data into three groups to which the model is fitted separately.

```
myplot + stat_smooth(method="lm")
```



It is possible to use other types of models, including GAM and GLM, as smoothers, but we will not give examples of the use these more advanced models.

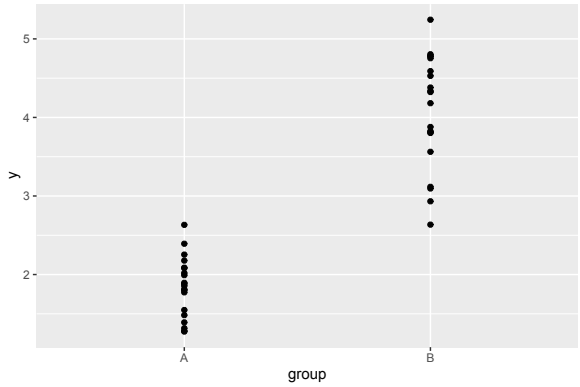
1.5 Adding statistical “summaries”

It is also possible to summarize data on-the-fly when plotting, but before showing this we will generate some normally distributed artificial data:

```
fake.data <- data.frame(
  y = c(rnorm(20, mean=2, sd=0.5), rnorm(20, mean=4, sd=0.7)),
  group = factor(c(rep("A", 20), rep("B", 20)))
  )
```

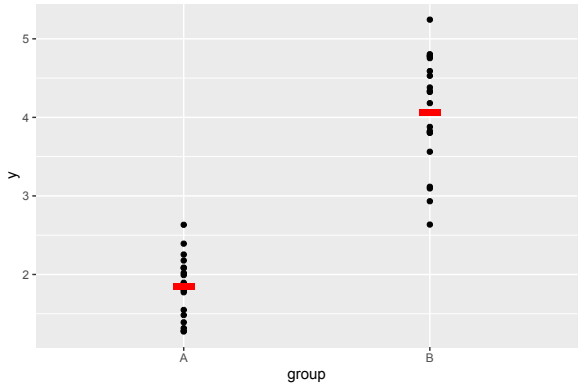
Now we use these data to plot means and confidence intervals by group:

```
fig2 <- ggplot(data=fake.data, aes(y=y, x=group)) + geom_point()
fig2
```

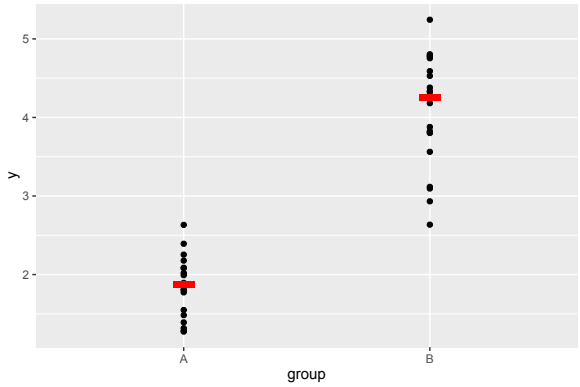


We have saved the base figure in `fig2`, so now we can play with different summaries. We first add just the mean. In this case we need to add as argument to `stat_summary` the geom to use, as the default one expects data for plotting error bars, in later examples, this is not needed.

```
fig2 + stat_summary(fun.y = "mean", geom="point",
                    colour="red", shape="-", size=20)
```

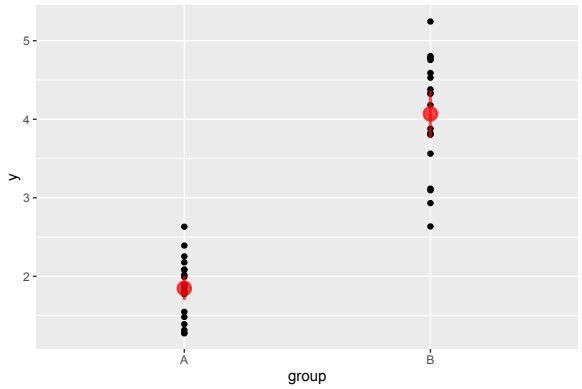



```
fig2 + stat_summary(fun.y = "median", geom="point",
                    colour="red", shape="-", size=20)
```



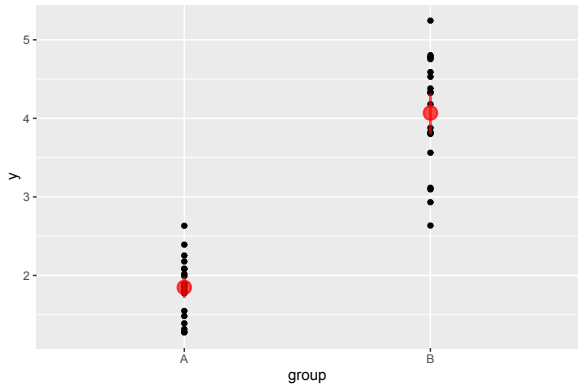
We can add the means and $p = 0.95$ confidence intervals not assuming normality (using the actual distribution of the data by bootstrapping):

```
fig2 + stat_summary(fun.data = "mean_cl_boot",
                    colour="red", size=1, alpha=0.7)
```



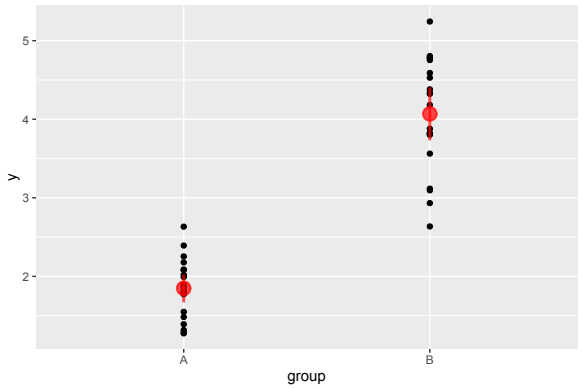
We can instead add the means and $p = 0.90$ confidence intervals, by supplying a value to parameter `conf.int`:

```
fig2 + stat_summary(fun.data = "mean_cl_boot",
                    fun.args = list(conf.int = 0.90),
                    colour = "red", size = 1, alpha = 0.7)
```



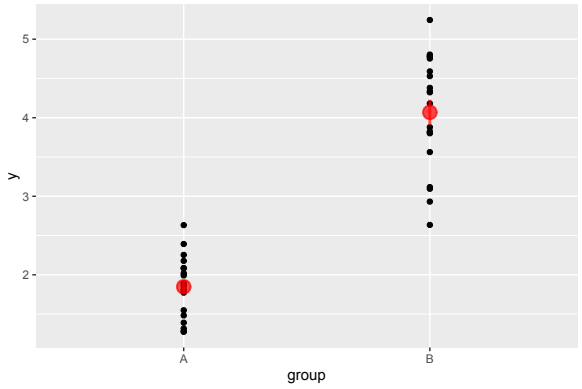
We can add the mean and $p = 0.95$ confidence intervals assuming normality (using the t distribution):

```
fig2 + stat_summary(fun.data = "mean_cl_normal",
                    colour="red", size=1, alpha=0.7)
```



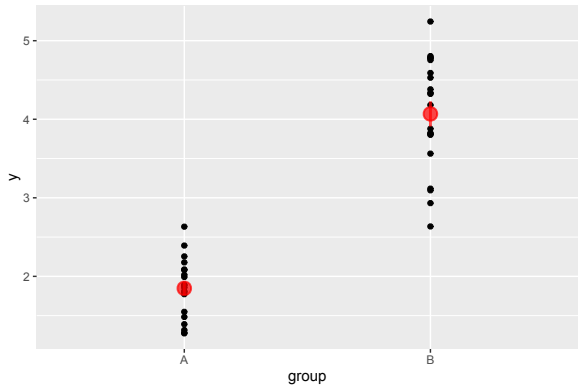
We can plot error bars corresponding to ±s.e. (standard errors). The function `mean_se` was first available in ggplot2 2.0.0.

```
fig2 + stat_summary(fun.data = "mean_se",
                    colour="red", size=1, alpha=0.7)
```



Or also one could use.

```
fig2 + stat_summary(fun.data = "mean_cl_normal", fun.args = list(mult = 1),
                    colour="red", size=1, alpha=0.7)
```

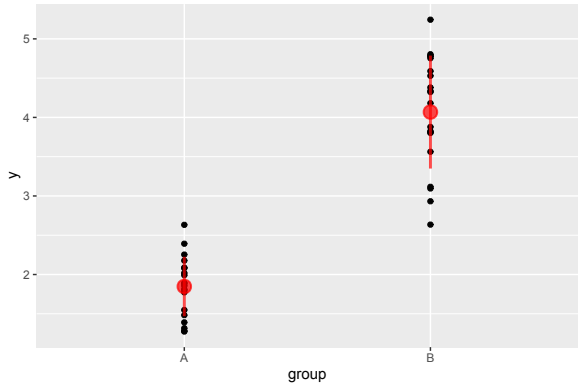


However, be aware that the code below, as used in earlier versions of ggplot2, needs to be rewritten as above.

```
fig2 + stat_summary(fun.data = "mean_cl_normal", mult = 1,
                    colour="red", size=1, alpha=0.7)
```

Finally we can plot error bars showing ±s.d. (standard deviation).

```
fig2 + stat_summary(fun.data = "mean_sdl",
                    fun.args = list(mult = 1),
                    colour="red", size=1, alpha=0.7)
```

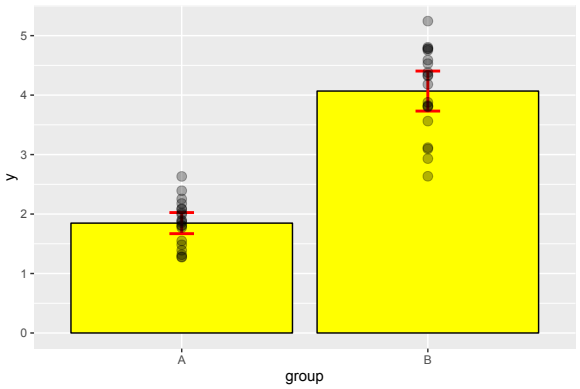


We do not show it here, but instead of using these functions (from package `Hmisc`) it is possible to define one's own functions, and remember that arguments to all functions, except for the first one containing the actual data should be supplied as a list through formal argument `fun.args`.

Finally we plot the means in a bar plot, with the observations superimposed and $p = 0.95$ C.I. (the order in which the geoms are added is important: by having

`geom_point` last it is plotted on top of the bars. In this case we set fill, colour and alpha (transparency) to constants, but in more complex data sets they can be assigned to factors in the data set.

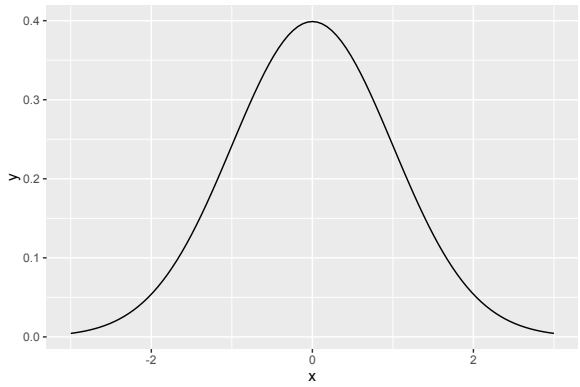
```
ggplot(data=fake.data, aes(y=y, x=group)) +
  stat_summary(fun.y = "mean", geom = "bar",
               fill="yellow", colour="black") +
  stat_summary(fun.data = "mean_cl_normal",
               geom = "errorbar",
               width=0.1, size=1, colour="red") +
  geom_point(size=3, alpha=0.3)
```



1.6 Plotting functions

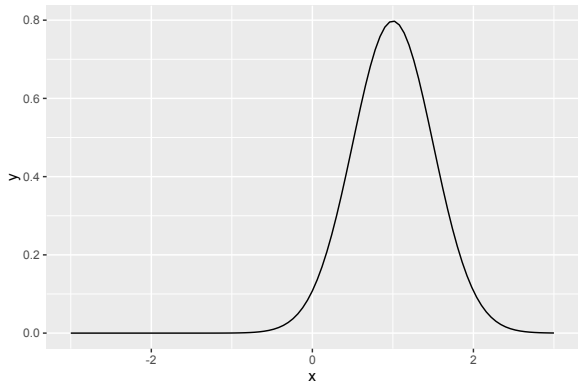
We can also directly plot functions, without need to generate data beforehand:

```
ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun=dnorm)
```



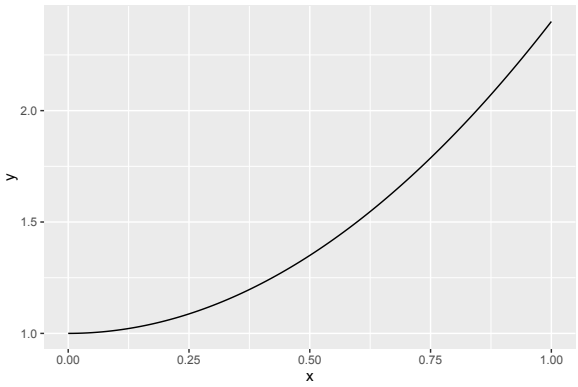
We can even pass additional arguments to a function:

```
ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun = dnorm, args = list(mean = 1, sd = .5))
```



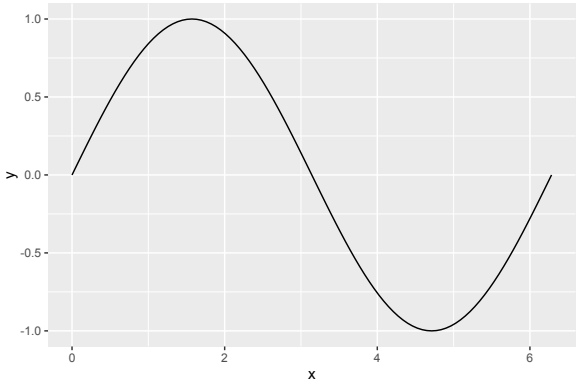
Of course, user-defined functions (not shown), and anonymous functions can also be used:

```
ggplot(data.frame(x=0:1), aes(x=x)) +
  stat_function(fun = function(x, a, b){a + b * x^2},
                args = list(a = 1, b = 1.4))
```



Here is another example of a predefined function, but in this case the default scale is not the best:

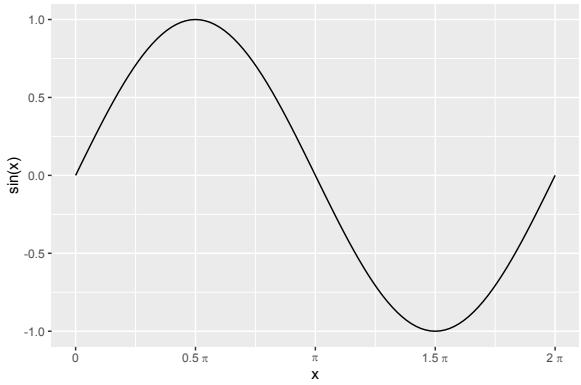
```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin)
```



In this case we need to change the x-axis scale to better suit the sin function and the use of radians as angular units[2].

```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
  labs(y="sin(x)")
```

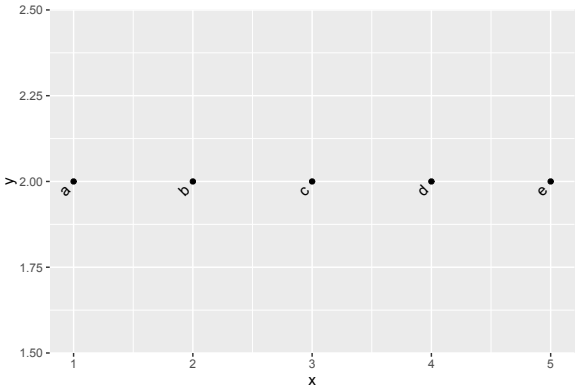
[2] The use of `expression` is explained in detail in section **??**, an the use of `scales` in section **??**.



1.7 Plotting text

One can use `geom_text` to add text labels to observations. The aesthetic `label` gives text and the usual aesthetics `x` and `y` the location of the labels. As one would expect the `colour` aesthetic can be also used for text. In addition `angle` and `vjust` and `hjust` can be used to rotate the label, and adjust its position. The default value of zero for both `hjust` and `vjust` centres the label. The centre of the text is at the supplied `x` and `y` coordinates. 'Vertical' and 'horizontal' for justification refer to the text, not the plot. This is important when `angle` is different from zero. Negative justification values, shift the label left or down, and positive values right or up. A value of 1 or -1 sets the text so that its edge is at the supplied coordinate. Values outside the range $-1 \ldots 1$ sift the text even further away.

```
my.data <-
  data.frame(x=1:5, y=rep(2, 5), label=paste(letters[1:5], " "))
ggplot(my.data, aes(x,y,label=label)) +
  geom_text(angle=45, hjust=1) + geom_point()
```



In this example we use `paste` (which uses recycling here) to add a space at the end of each label. Justification values outside the range $-1 \ldots 1$ are allowed, but are relative to the with of the label. As the default font used in this case has variable with characters, the justification would be inconsistent (e.g. try the code above but using `hjust` set to 3 instead of to 1 without pasting a space character to the labels.)

1.8 Scales

Scales map data onto aesthetics. There are different types of scales depending on the characteristics of the data being mapped: scales can be continuous or discrete. And of course, there are scales for different attributes of the plotted object, such as `colour`, `size`, position (`x, y, z`), `alpha` or transparency, `angle`, justification, etc. This means that many properties of, for example, the symbols used in a plot can be either set by a constant, or mapped to data. The most elemental mapping is `identity`, which means that the data is taken at its face value. In a numerical scale, say `scale_x_continuous`, this means that for example a '5' in the data is plotted at a position in the plot corresponding to the value '5' along the x-axis. A simple mapping could be a log10 transformation, that we can easily achieve with the pre-defined `scale_x_log10` in which case the position on the x-axis will be based on the logarithm of the original data. A continuous data variable can, if we think it useful for describing our data, be mapped to continuous scale either using an identity mapping or transformation, which for example could be useful if we want to map the value of a variable to the area of the symbol rather than its diameter.

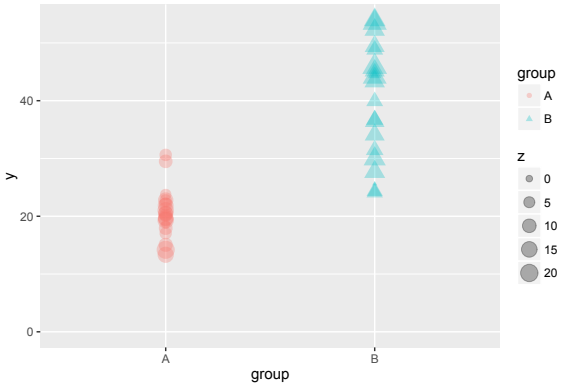
Discrete scales work in a similar way. We can use `scale_colour_identity` and have in our data a variable with values that are valid colour names like

"red" or "blue". However we can also assign the `colour` aesthetic to a factor with levels like "control", and "treatment", an these levels will be mapped to colours from the default palette, unless we chose a different one, or even use `scale_colour_manual` to assign whatever colour we want to each level to be mapped. The same is true for other discrete scales like symbol `shape` and `linetype`. Be aware that for example for colour, and 'numbers' there are both discrete and continuous scales available.

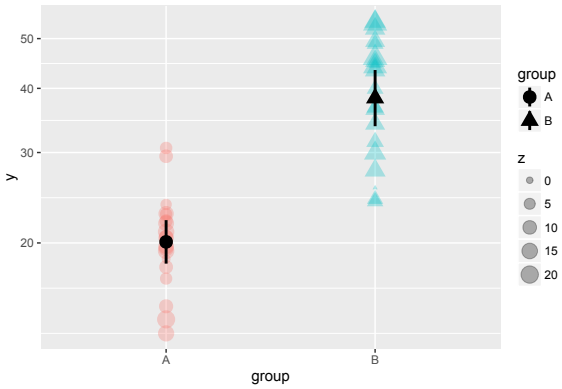
Advanced scale manipulation requires the package `scales` to be loaded. Some simple examples follow.

```
fake2.data <- data.frame(
  y = c(rnorm(20, mean=20, sd=5), rnorm(20, mean=40, sd=10)),
  group = factor(c(rep("A", 20), rep("B", 20))),
  z = rnorm(40, mean=12, sd=6)
  )
```

```
fig2 <-
  ggplot(data=fake2.data,
         aes(y=y, x=group, shape=group, colour=group, size=z)) +
  geom_point(alpha=0.3) + ylim(0, NA)
fig2
```



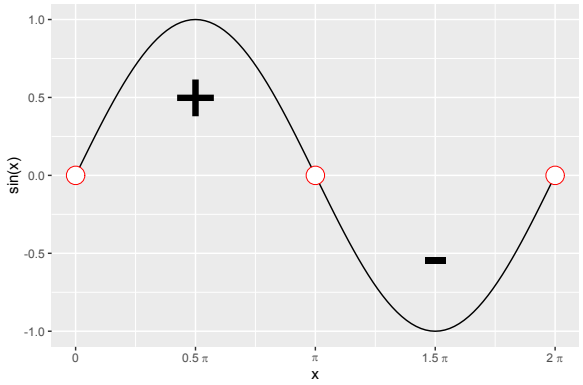

```
fig2 +
  scale_y_log10(breaks=c(10,20,30,40,50,60)) +
       stat_summary(fun.data = "mean_cl_normal",
                    colour="black", size=1, alpha=1)

## Scale for 'y' is already present.  Adding
## another scale for 'y', which will replace the
## existing scale.
```



1.9 Adding annotations

Annotations use the data coordinates of the plot, but do not 'inherit' data or aesthetics from the ggplot.

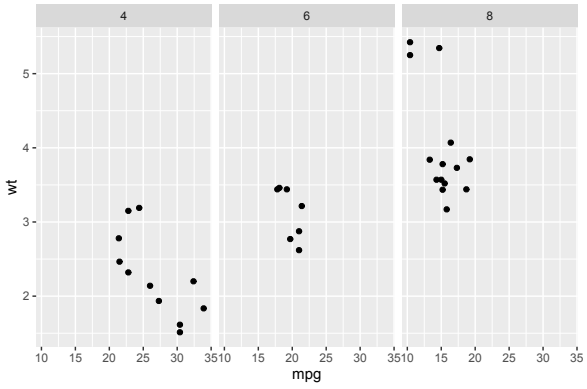
```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
  labs(y="sin(x)") +
  annotate(geom="text",
           label=c("+", "-"),
           x=c(0.5, 1.5) * pi, y=c(0.5, -0.5),
           size=20) +
  annotate(geom="point",
           colour="red",
           shape=21,
           fill="white",
           x=c(0, 1, 2) * pi, y=0,
           size=6)
```



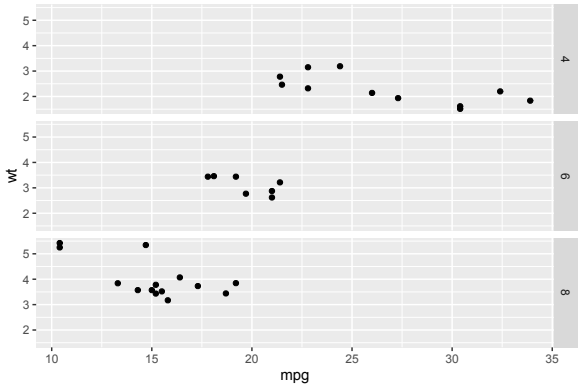
1.10 Using facets

Sets of coordinated plots are a very useful tool for visualizing data. These became popular through the `trellis` graphs in S, and the `lattice` package in R. The basic idea is to have row and/or columns of plots with common scales, all plots showing values for the same response variable. This is useful when there are multiple classification factors in a data set. Similarly looking plots but with free scales or with the same scale but a 'floating' intercept are sometimes also useful. In ggplot2 there are two possible types of facets: facets organized in a grid, and facets on along a single 'axis' but wrapped into several rows. In the examples below we use `geom_point` but faceting can be used with any geom, and even with maps and ternary plots.

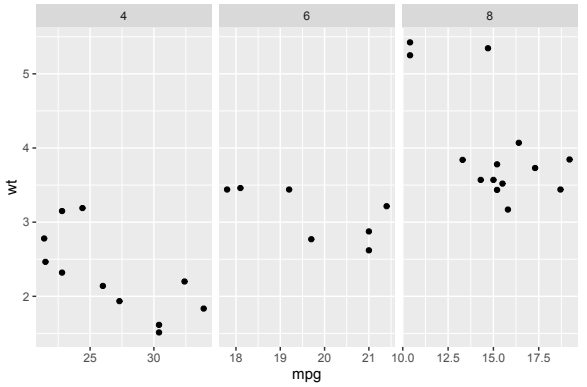
```
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
# With one variable
p + facet_grid(. ~ cyl)
```



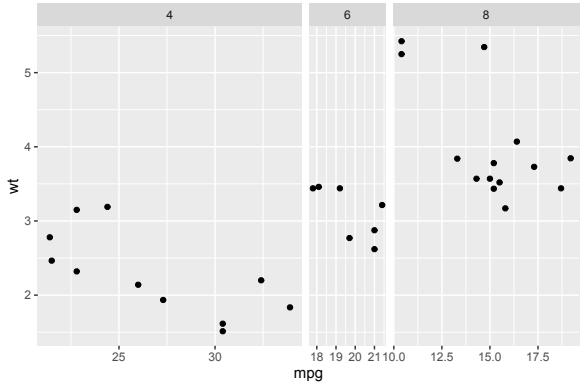

```
p + facet_grid(cyl ~ .)
```



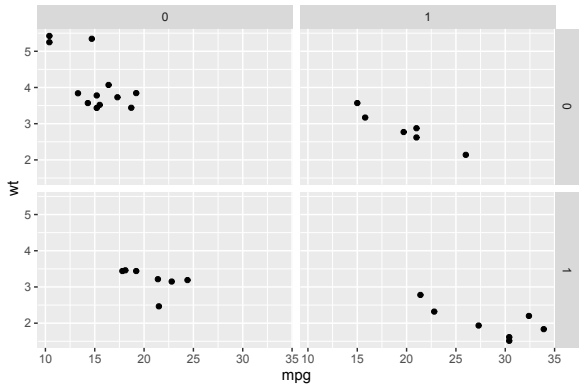

```
p + facet_grid(. ~ cyl, scales = "free")
```



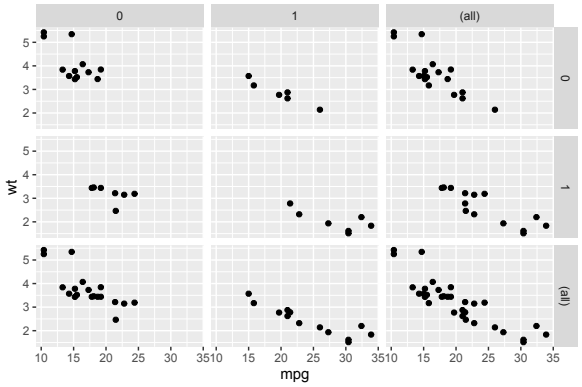
```
p + facet_grid(. ~ cyl, scales = "free", space = "free")
```



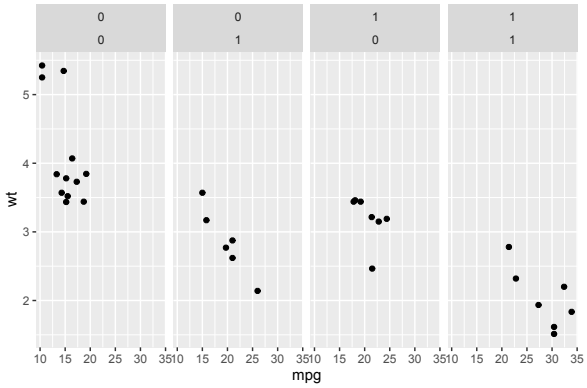

```
p + facet_grid(vs ~ am)
```



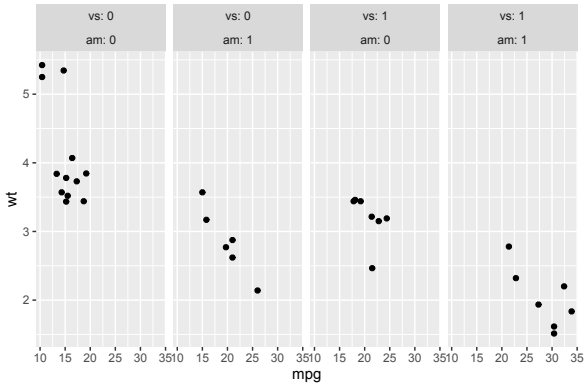

```
p + facet_grid(vs ~ am, margins=TRUE)
```



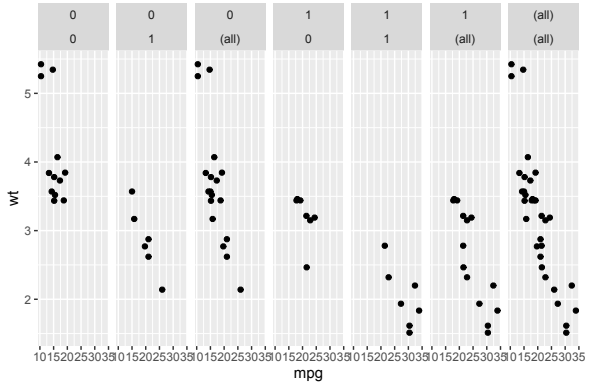

```
p + facet_grid(. ~ vs + am)
```



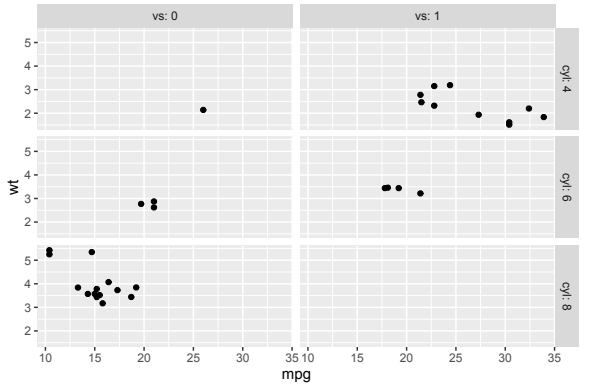

```
p + facet_grid(. ~ vs + am, labeller = label_both)
```



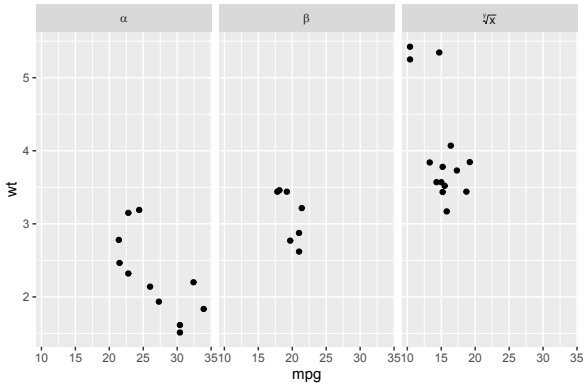
```
p + facet_grid(. ~ vs + am, margins=TRUE)
```



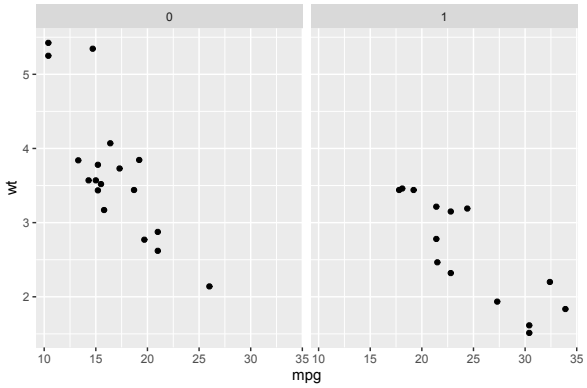

```
p + facet_grid(cyl ~ vs, labeller = label_both)
```



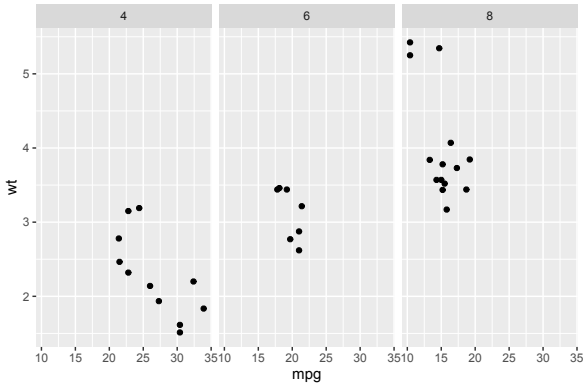

```
mtcars$cyl12 <- factor(mtcars$cyl, labels = c("alpha", "beta", "sqrt(x, y)"))
p1 <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
p1 + facet_grid(. ~ cyl12, labeller = label_parsed)
```



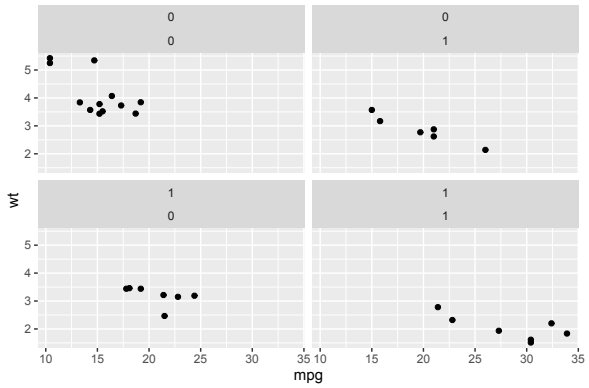

```
p + facet_grid(. ~ vs, labeller = label_bquote(alpha ^ .(x)))
```




```
p + facet_wrap(~ cyl)
```



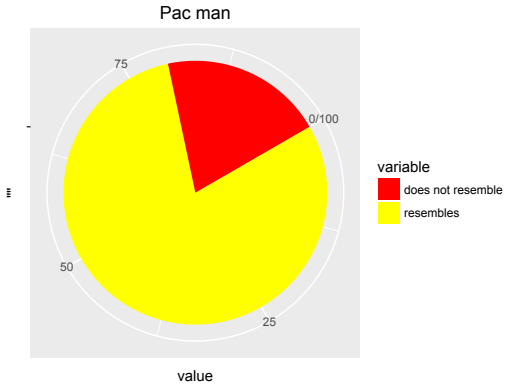
```
p + facet_wrap(~ vs + am, ncol=2)
```



1.11 Circular plots

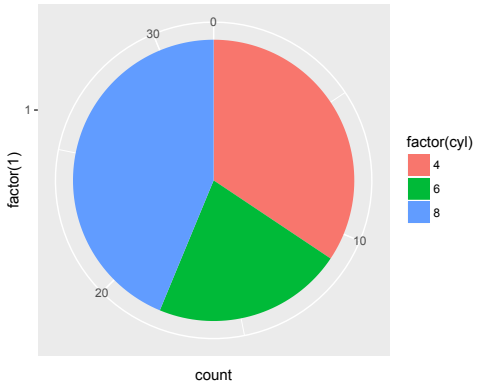
A funny example stolen from the ggplot2 website at `http://docs.ggplot2.org/current/coord_polar.html`.

```
# Hadley's favourite pie chart
df <- data.frame(
  variable = c("resembles", "does not resemble"),
  value = c(80, 20)
)
ggplot(df, aes(x = "", y = value, fill = variable)) +
  geom_bar(width = 1, stat = "identity") +
  scale_fill_manual(values = c("red", "yellow")) +
  coord_polar("y", start = pi / 3) +
  labs(title = "Pac man")
```



Something just a bit more useful, also stolen from the same page:

```
# A pie chart = stacked bar chart + polar coordinates
pie <- ggplot(mtcars, aes(x = factor(1), fill = factor(cyl))) +
 geom_bar(width = 1)
pie + coord_polar(theta = "y")
```

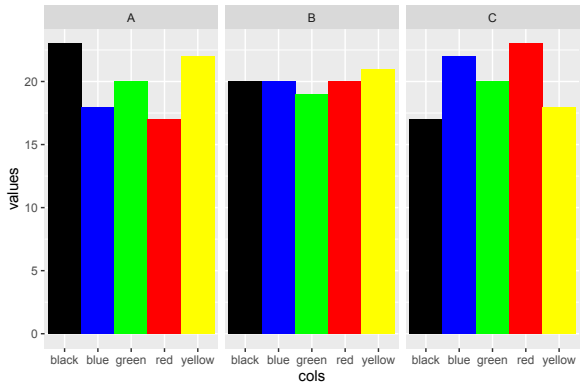


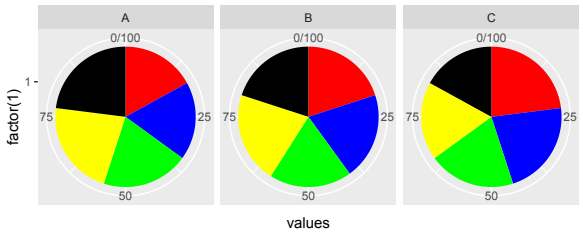
1.12 Pie charts vs. bar plots example

There is an example figure widely used in Wikipedia to show how much easier it is to 'read' bar plots than pie charts (`http://commons.wikimedia.org/wiki/File:Piecharts.svg?uselang=en-gb`).

Here is my ggplot2 version of the same figure, using much simpler code and obtaining almost the same result.

```
example.data <-
  data.frame(values = c(17, 18, 20, 22, 23,
                        20, 20, 19, 21, 20,
                        23, 22, 20, 18, 17),
             examples= rep(c("A", "B", "C"), c(5,5,5)),
             cols = rep(c("red", "blue", "green", "yellow", "black"), 3)
  )

ggplot(example.data, aes(x=cols, y=values, fill=cols)) +
  geom_bar(width = 1, stat="identity") +
  facet_grid(.~examples) +
  scale_fill_identity()
ggplot(example.data, aes(x=factor(1), y=values, fill=cols)) +
  geom_bar(width = 1, stat="identity") +
  facet_grid(.~examples) +
  scale_fill_identity() +
  coord_polar(theta="y")
```





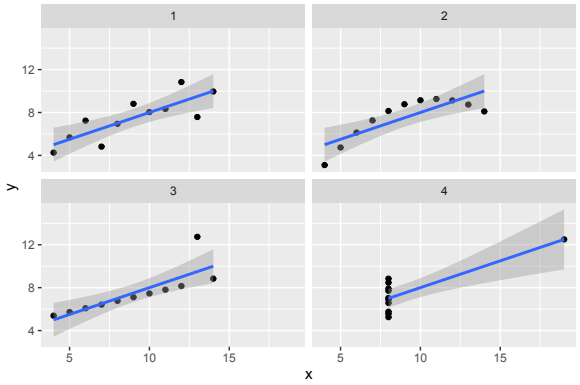
1.13 A classical example about regression

This is another figure from Wikipedia `http://commons.wikimedia.org/wiki/File:Anscombe.svg?uselang=en-gb`. The original code (not run):

```
svg("anscombe.svg", width=10.5, height=7)
par(las=1)

##-- some "magic" to do the 4 regressions in a loop:
ff <- y ~ x
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  ## or   ff2 <- as.name(paste("y", i, sep=""))
  ##      ff3 <- as.name(paste("x", i, sep=""))
  assign(paste("lm.",i,sep=""), lmi <- lm(ff, data= anscombe))
}

## Now, do what you should have done in the first place: PLOTS
op <- par(mfrow=c(2,2), mar=1.5+c(4,3.5,0,1), oma=c(0,0,0,0),
          lab=c(6,6,7), cex.lab=1.5, cex.axis=1.3, mgp=c(3,1,0))
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  plot(ff, data =anscombe, col="red", pch=21, bg = "orange", cex = 2.5,
       xlim=c(3,19), ylim=c(3,13),
       xlab=eval(substitute(expression(x[i]), list(i=i))),
       ylab=eval(substitute(expression(y[i]), list(i=i))))
  abline(get(paste("lm.",i,sep="")), col="blue")
}

dev.off()
```

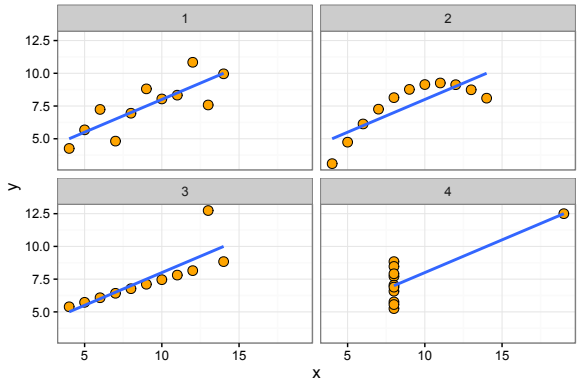
My version using ggplot2:

```
# we rearrange the data
my.mat <- matrix(as.matrix(anscombe), ncol=2)
my.anscombe <- data.frame(x = my.mat[ , 1],
                          y = my.mat[ , 2],
                          case=factor(rep(1:4, rep(11,4))))
# we draw the figure
ggplot(my.anscombe, aes(x,y)) +
  geom_point() +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2)
```



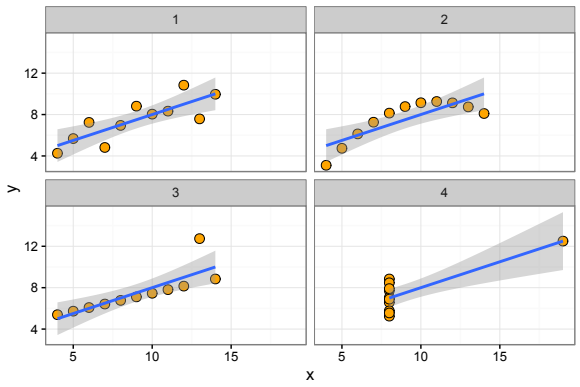
It is not much more difficult to make it look similar to the original

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm", se=FALSE) +
  facet_wrap(~case, ncol=2) +
  theme_bw()
```



Although I think that the confidence bands make the point of the example much clearer

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2) +
  theme_bw()
```



This classical example form Anscombe **xxx** demonstrates four very different data sets that yield exactly the same results when a linear regression model is fit to them, including $R^2 = 0.666$. It is usually presented as a warning about the need to check model fits beyond looking at R^2 and other parameter's estimates.

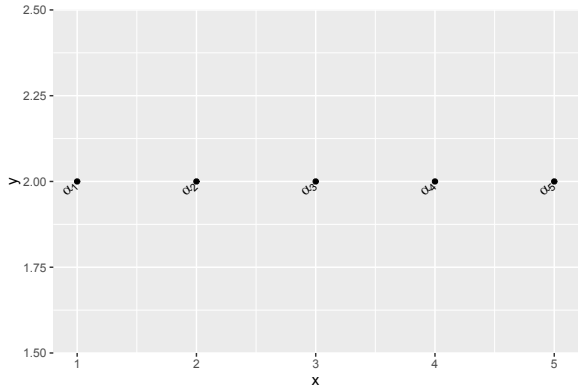
1.14 Advanced topics

1.15 Using `plotmath` expressions

Expressions are very useful but rather tricky to use because the syntax is unusual. In `ggplot` one can either use expressions explicitly, or supply them as character string labels, and tell `ggplot` to parse them. For titles, axis-labels, etc. (anything that is defined with `labs`) the expressions have to entered explicitly, or saved as such into a variable, and the variable supplied as argument. When plotting expressions using `geom_text` expression arguments should be supplied as character strings and the optional argument `parse=TRUE` used to tell the geom to interpret the labels as expressions. We will go through a few useful examples.

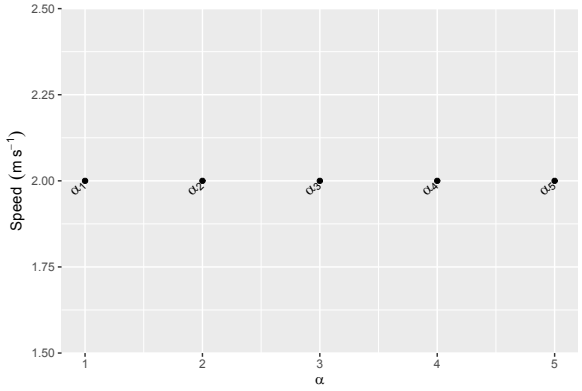
We will revisit the example from the previous section, but now using subscripted Greek α for labels. In this example we use as subscripts numeric values from another variable in the same dataframe.

```
my.data$greek.label <- paste("alpha[", my.data$x, "]", sep="")
(fig <- ggplot(my.data, aes(x,y,label=greek.label)) +
   geom_text(angle=45, hjust=1.2, parse=TRUE) + geom_point())
```



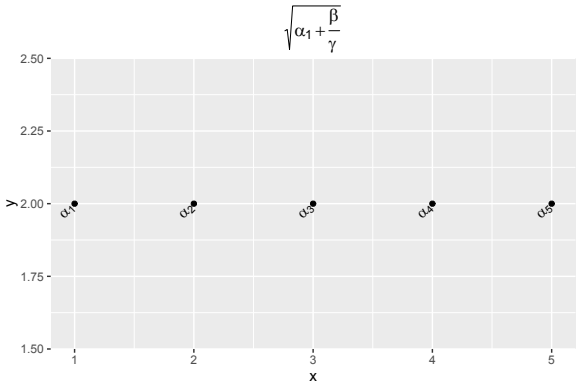
Setting an axis label with superscripts. The easiest way to deal with spaces is to use ‘ ’ or ‘ ’. One can connect pieces that would otherwise cause errors using ‘*’. If we

```
fig + labs(x=expression(alpha), y=expression(Speed~~(m~s^{-1})))
```



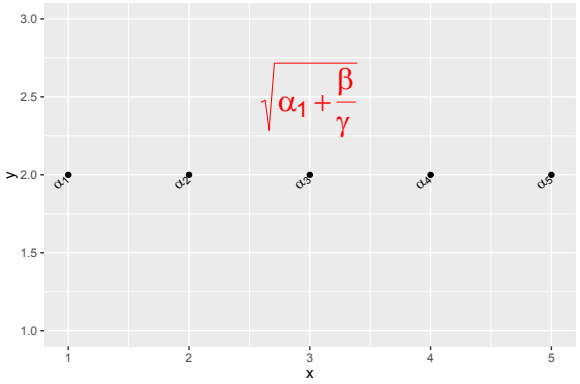
It is possible to store expressions in variables.

```
my.title <- expression(sqrt(alpha[1] + frac(beta, gamma)))
fig + labs(title=my.title)
```



Annotations are plotted ignoring the default aesthetics, but still make use of geoms, so labels for annotations also have to be supplied as character strings and parsed.

```
fig + ylim(1,3) +
  annotate("text", label="sqrt(alpha[1] + frac(beta, gamma))",
           y=2.5, x=3, size=8, colour="red", parse=TRUE)
```



We discuss how to use expressions as facet labels in section ??.

1.16 Generating output files

It is possible, when using RStudio, to directly export the displayed plot to a file. However, if the file will have to be generated again at a later time, or a series of plots need to be produced with consistent format, it is best to include the commands to export the plot in the script.

In R, files are created by printing to different devices. Printing is directed to a currently open device. Some devices produce screen output, others files. Devices depend on drivers. There are both devices that or part of R, and devices that can be added through packages.

A very simple example of PDF output (width and height in inches):

```
fig1 <- ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun=dnorm)
pdf(file="fig1.pdf", width=8, height=6)
print(fig1)
dev.off()
```

Encapsulated Postscript output (width and height in inches):

```
postscript(file="fig1.eps", width=8, height=6)
print(fig1)
dev.off()
```

There are Graphics devices for BMP, JPEG, PNG and TIFF format bitmap files. In this case the default units for width and height is pixels. For example we can generate TIFF output:

```
tiff(file="fig1.tiff", width=1000, height=800)
print(fig1)
dev.off()
```

```
try(detach(package:scales))
try(detach(package:plyr))
try(detach(package:Hmisc))
try(detach(package:ggplot2))
try(detach(package:grid))
```