Other useful packages of graphics

April 4, 2017

Contents

1 Lattice

The lattice package provides a comprehensive graphical system for visualizing univariate and multivariate data. In particular, many users turn to the lattice package because of its ability to easily generate trellis graphs.

- lattice: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like xyplot, bwplot, levelplot
- The lattice plotting system does not have a "two-phase" aspect with separate plotting and annotation like in base plotting
- All plotting/annotation is done at once with a single function call

1.1 Lattice Functions

The lattice package provides a wide variety of functions for producing univariate (dot plots, kernel density plots, histograms, bar charts, box plots), bivariate (scatter plots, strip plots, parallel box plots), and multivariate (3D plots, scatter plot matrices) graphs.

- xyplot: this is the main function for creating scatterplots
- bwplot: box-and-whiskers plots ("boxplots")
- histogram: histograms
- stripplot: like a boxplot but with actual points
- dotplot: plot dots on "violin strings"
- splom: scatterplot matrix; like pairs in base plotting system
- levelplot, contourplot: for plotting "image" data

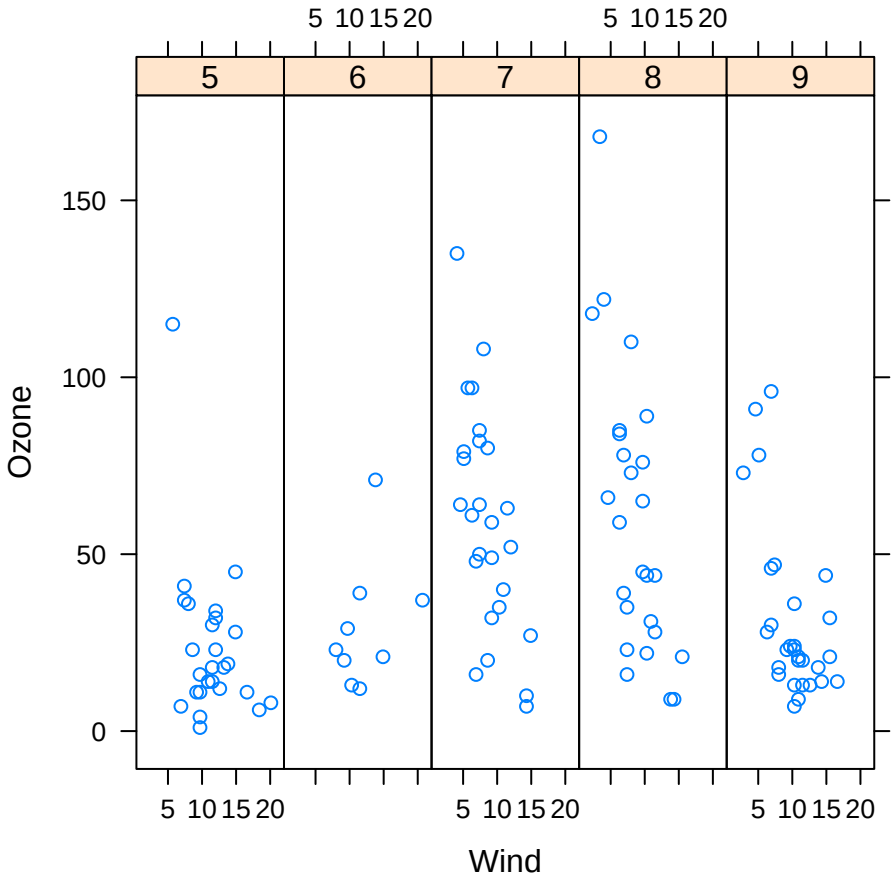
Lattice functions generally take a formula for their first argument, usually of the form

```
xyplot(y ~ x | f * g, data, option)
```

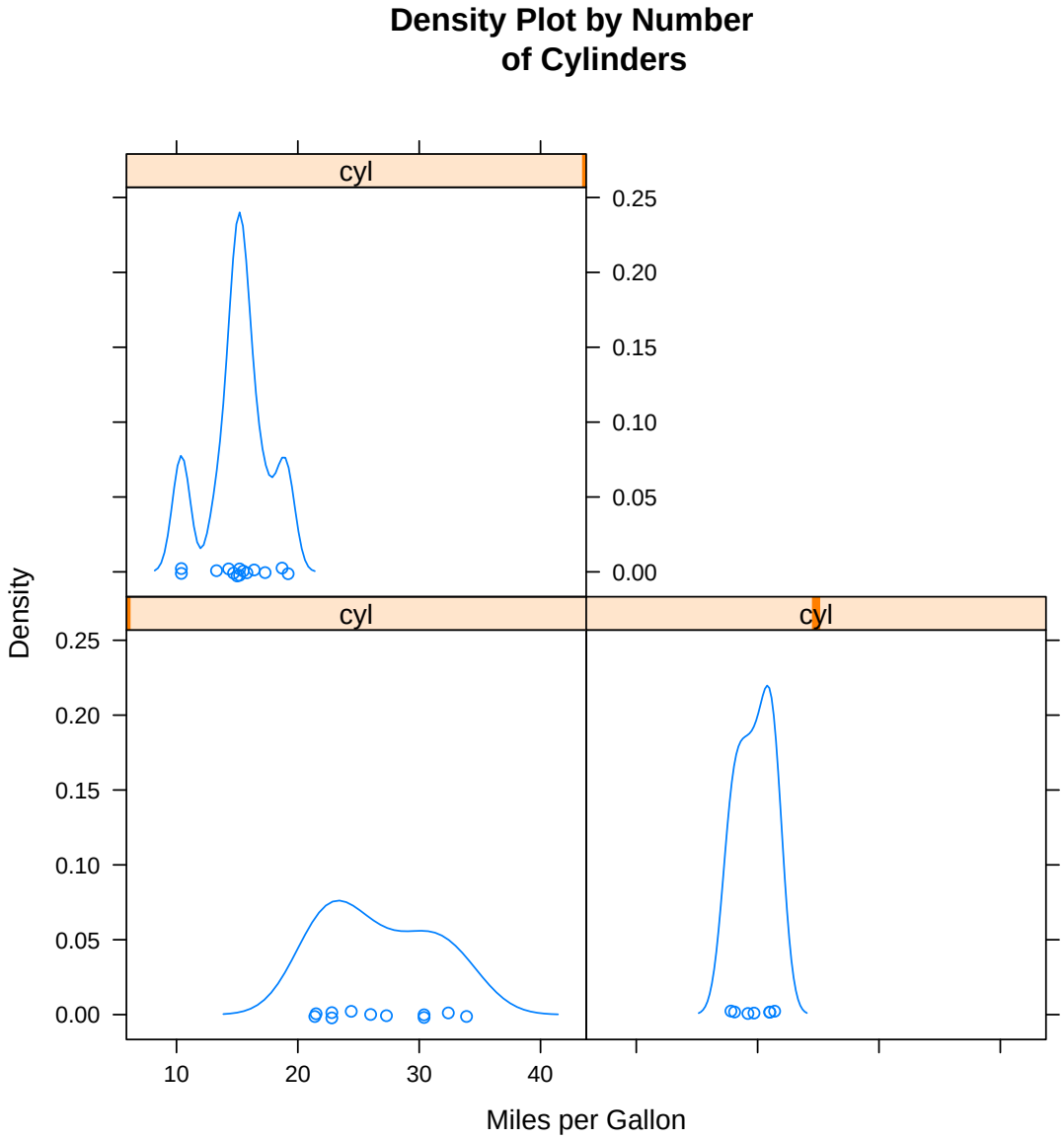
- We use the formula notation here, hence the ~.
- On the left of the ~ is the y-axis variable, on the right is the x-axis variable
- f and g are conditioning variables — they are optional
  - the * indicates an interaction between two variables
- The second argument is the data frame or list from which the variables in the formula should be looked up
  - If no data frame or list is passed, then the parent frame is used
- If no other arguments are passed, there are defaults that can be used.

lattice plot examples

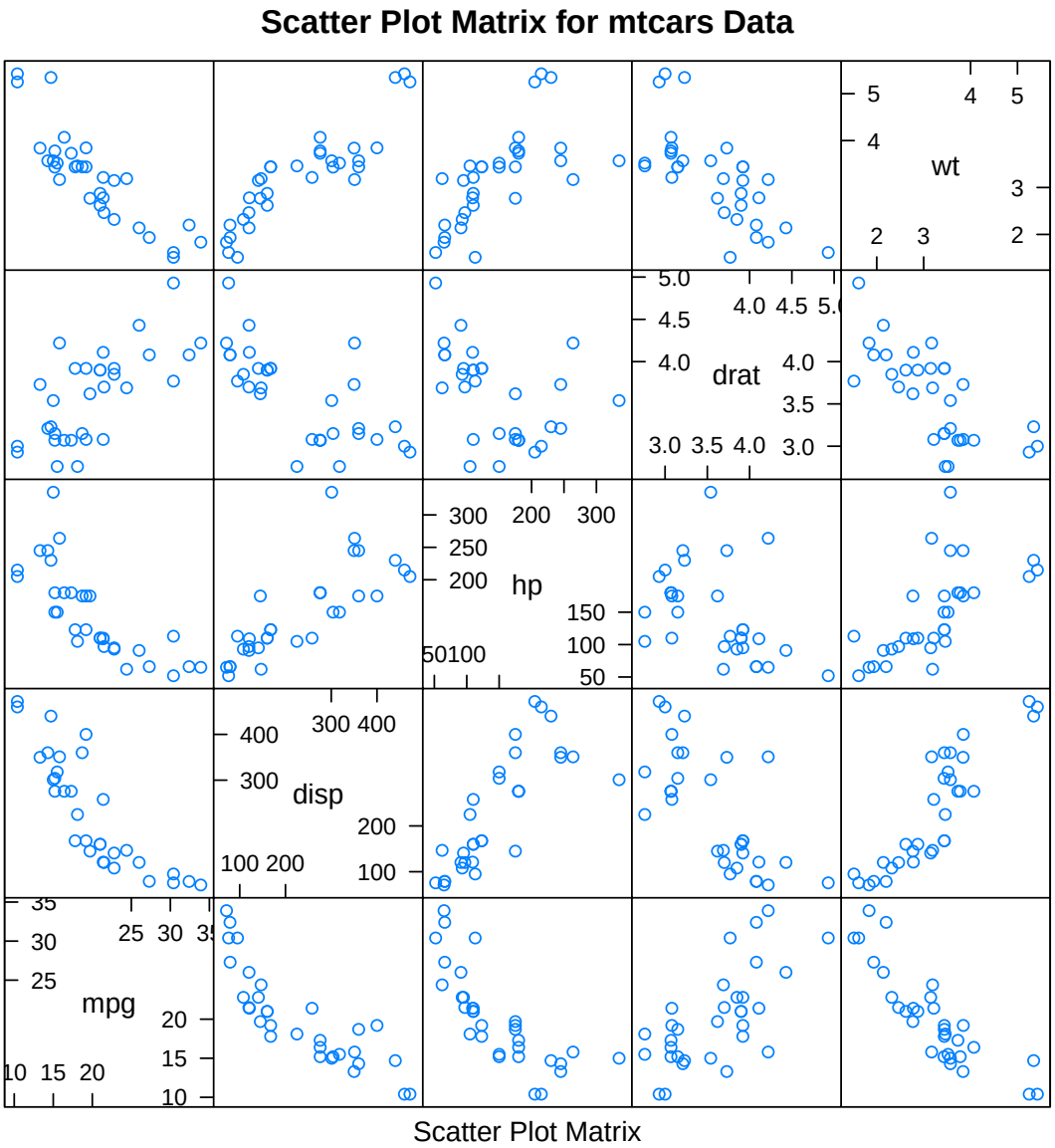
```
library(datasets)
library(lattice)
## Convert 'Month' to a factor variable
airquality <- transform(airquality, Month = factor(Month))
xyplot(Ozone ~ Wind | Month, data = airquality, layout = c(5, 1))
```



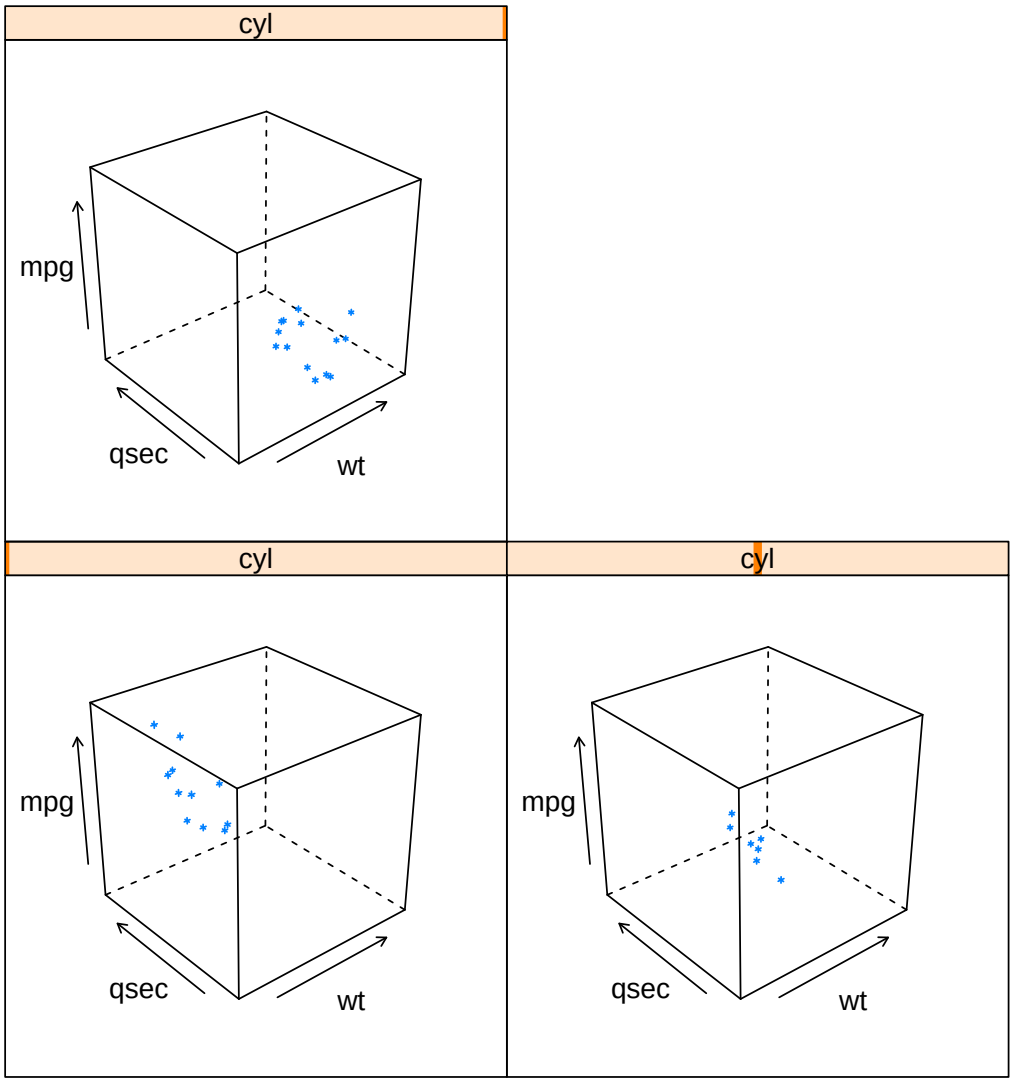
```
densityplot(~mpg | cyl, data=mtcars, main = "Density Plot by Number
    of Cylinders", xlab = "Miles per Gallon")
```



```
splom(mtcars[c(1, 3, 4, 5, 6)], main = "Scatter Plot Matrix for mtcars Data")
```

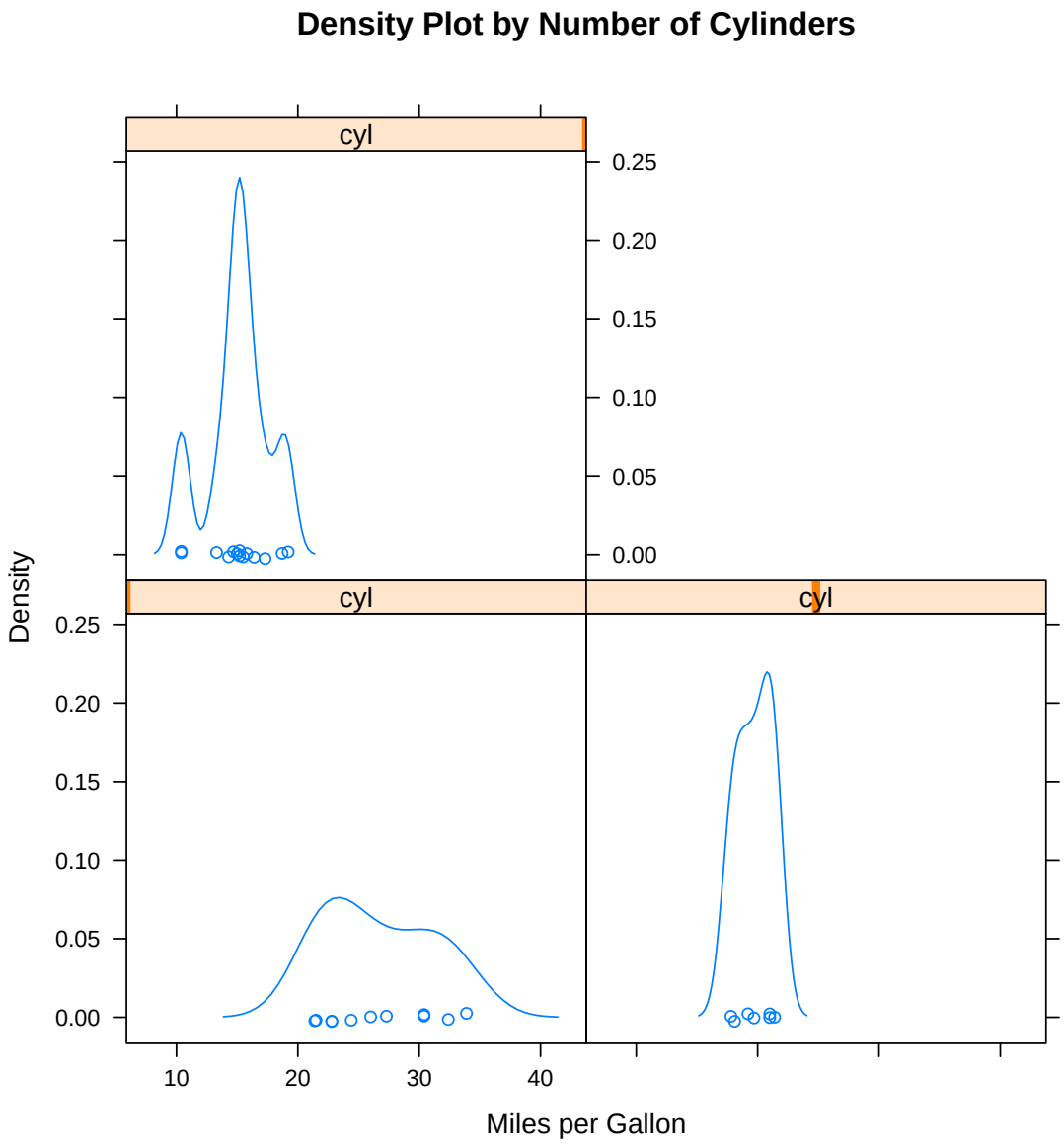


```
cloud(mpg ~ wt * qsec | cyl, data=mtcars, main = "3D Scatter Plots by Cylinders")
```

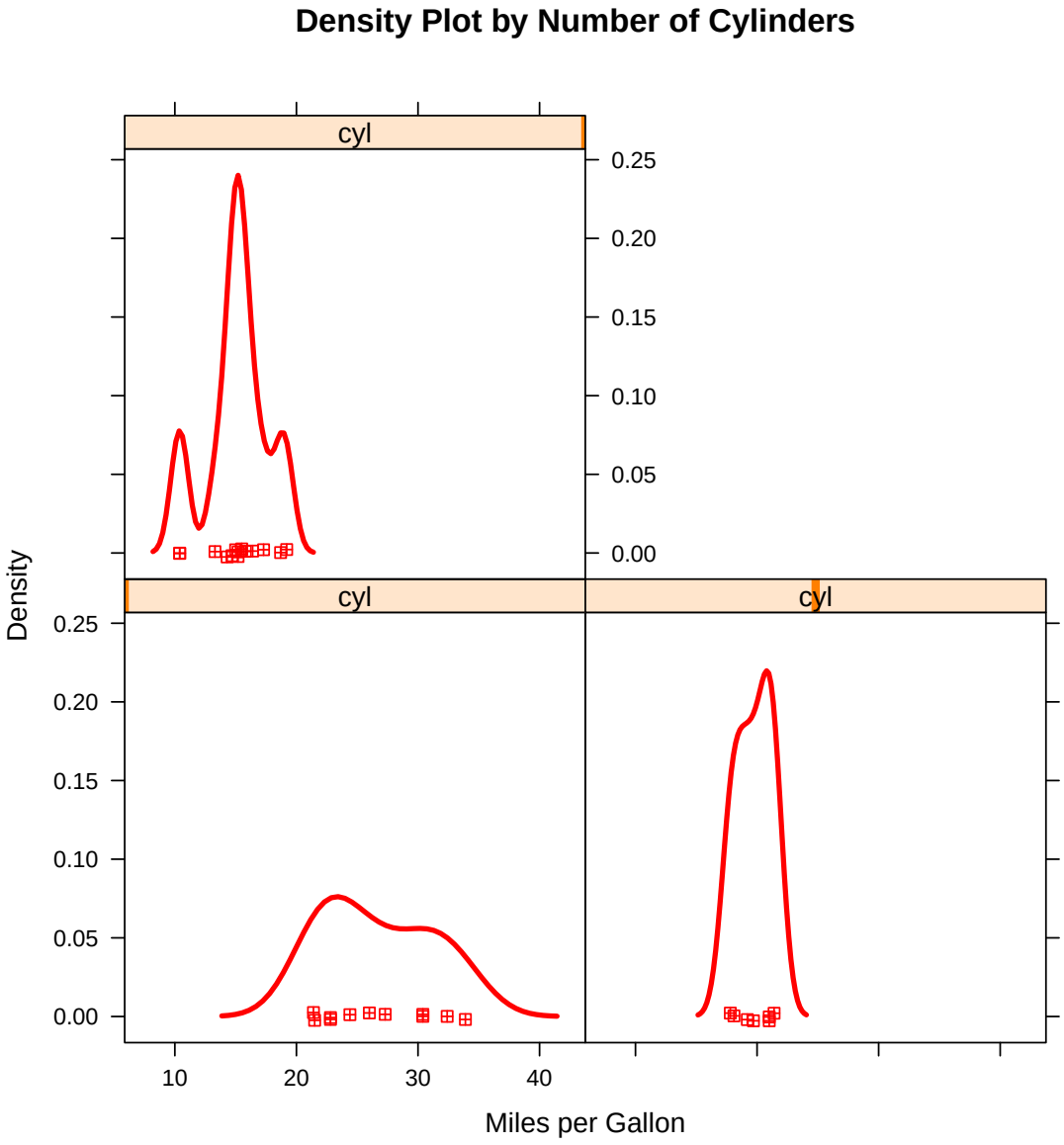


You can also use the update() function to modify a lattice graphic object.

```
library(lattice)
mygraph<-densityplot(~mpg | cyl, data=mtcars,
    main = "Density Plot by Number of Cylinders",
    xlab = "Miles per Gallon")
mygraph
```




```
update(mygraph, col="red", pch=12, cex=0.8, lwd=3)
```



1.2 Conditioning variables

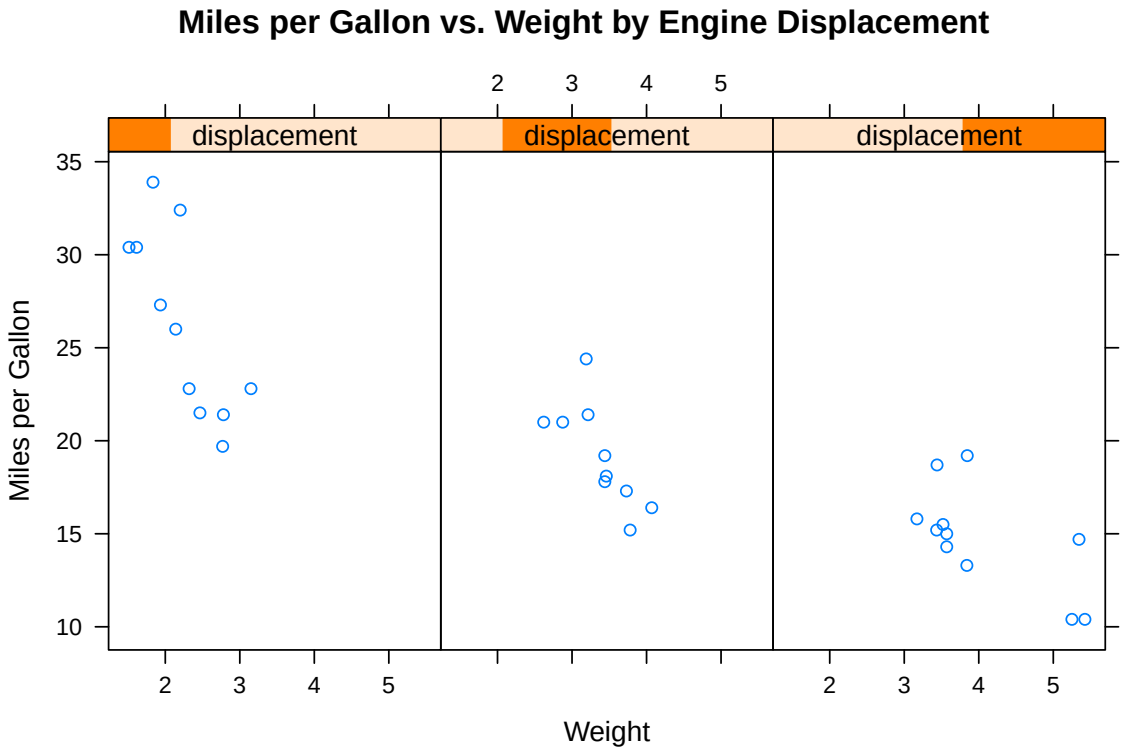
As you've seen, one of the most powerful features of lattice graphs is the ability to add conditioning variables. If one conditioning variable is present, a separate panel is created for each level. If two conditioning variables are present, a separate panel is created for each combination of levels for the two variables. It's rarely useful to include more than two conditioning variables.

Typically, conditioning variables are factors. But what if you want to condition on a continuous variable? One approach would be to transform the continuous variable into a discrete variable using R's cut() function . Alternatively, the lattice package provides functions for transforming a continuous variable into a data structure called a shingle. Specifically, the continuous variable is divided up into a series of (possibly) overlapping ranges.

```
myshingle <- equal.count(x, number=#, overlap=proportion)
```

will take continuous variable x and divide it up into # intervals, with proportion overlap, and equal numbers of observations in each range, and return it as the variable myshingle (of class shingle).

```
library(lattice)
displacement <- equal.count(mtcars$disp, number=3, overlap=0)
xyplot(mpg~wt|displacement, data=mtcars,
    main = "Miles per Gallon vs. Weight by Engine Displacement",
    xlab = "Weight", ylab = "Miles per Gallon",
    layout=c(3, 1), aspect=1.5)
```



Because engine displacement is a continuous variable, it has been converted to three nonoverlapping shingles with equal numbers of observations.

1.3 Graphic parameters

In previously section, we learned how to view and set default graphics parameters using the par() function . Although this works for graphs produced with R's native graphic system, lattice graphs are unaffected by these settings. Instead, the graphic defaults used by lattice functions are contained in a large list object that can be accessed with the trellis.par.get() function and modified through the trellis.par.set() function . The show.settings() function can be used to display the current graphic settings visually.

```
show.settings()
mysettings <- trellis.par.get()
mysettings$superpose.symbol
mysettings$superpose.symbol$pch <- c(1:15)
trellis.par.set(mysettings)
show.settings()
```

To learn more about lattice graphs, take a look the excellent text by Sarkar (2008) and its supporting website at http://lmdvr.r-forge.r-project.org. The Trellis Graphics User's Manual (http://cm.bell-labs.com/cm/ms/departments/sia/doc/trellis.user.pdf) is also an excellent source of information.

1.4 Some useful packages builted on lattice

Polar plots

The pie chart is a much-maligned graphical device for displaying counts or proportions as slices of a circle. The problem is that, in general, people are not good at perceiving the absolute or relative sizes of angles, compared to judging lengths or positions in normal cartesian coordinates.

However, pie charts are not the only way to present data on a polar coordinate system. It is possible to produce polar plots that represent values by lengths (of bars extending from the center) rather than angles and there are certain sorts of data, such as wind direction, that are naturally represented as an angle. For data that have a repeating cycle, such as hourly measurements over several days, a polar representation can also be useful for revealing periodic features.

The data used in this section are wind measurements from the New Zealand National Climate Database (http://cliflo.niwa.co.nz/). The data consist of wind speed and direction from daily (9am) observation for approximately two years (September 2008 to September 2010). Observations at each time point are taken from 11 different weather stations scattered around the Auckland region.

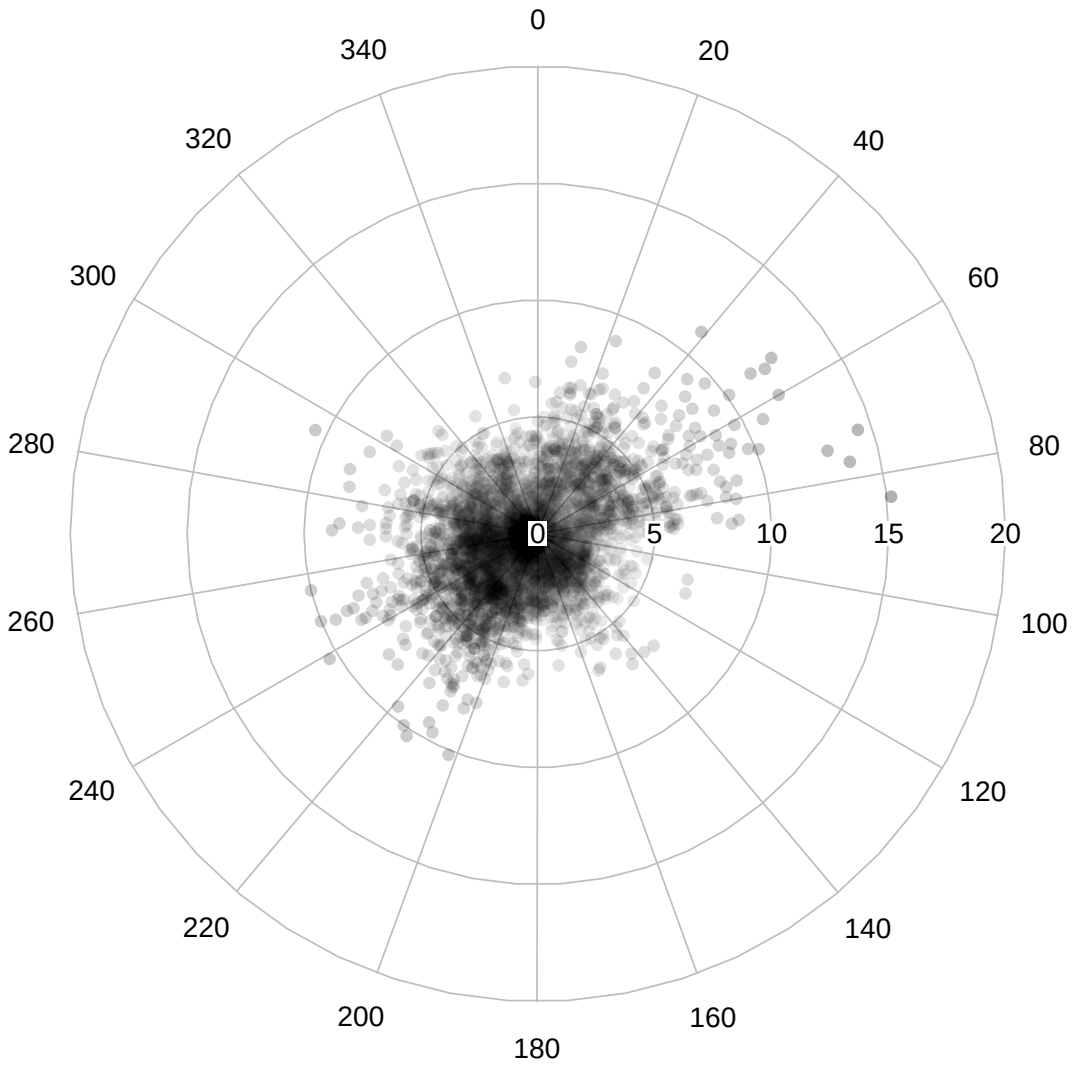
There is also a smaller data set consisting of hourly average wind speed based on data from 11 weather stations. There are hourly readings every day for one month (September 2010).

```
load("wind9am.rdata")
load("hourlySpeed.rdata")
head(wind9am)

##   Station                Date    Speed Dir
## 1   12325 2008-09-26 05:00:00 3.194444 211
## 2   12326 2008-09-26 05:00:00 3.194444 221
## 3   12327 2008-09-26 05:00:00 1.805556 212
## 4   12328 2008-09-26 05:00:00 1.611111 225
## 5   18195 2008-09-26 05:00:00 1.611111 215
## 6   22164 2008-09-26 05:00:00 2.805556 196

head(hourlySpeed)
```

```
##   hour day    Speed
## 1    0 237 1.626263
## 2    1 237 1.575758
## 3    2 237 1.618687
## 4    3 237 1.489899
## 5    4 237 1.138889
## 6    5 237 1.343434
```

The first function, polar.plot(), is from the plotrix package. This can be used to produce a variety of plots, but the code below is just a polar scatterplot of individual daily wind observations, with points plotted around the circle based on the wind direction and the distance from the center of the circle based on wind speed. The first two arguments provide the radius variable and the angle variable, and rp.type is specified here to plot data symbols at each point. The function normally draws its polar grid on top of the data symbols, so the code makes use of the standard painters model to first draw an empty plot with a grid and then draw another plot over the top to draw the actual data symbols. The data symbols are very dense at the center of the plot so the color of the data symbols is made semitransparent, with the level of transparency varying according to distance from the center.

```
#install.packages("plotrix")
library(plotrix)
with(wind9am,
     {
             polar.plot(Speed, Dir, rp.type="s",
                        start=90, clockwise=TRUE,
                        point.symbols=16,
                        point.col=rgb(0,0,0, .3*Speed/max(Speed)))
     })
```

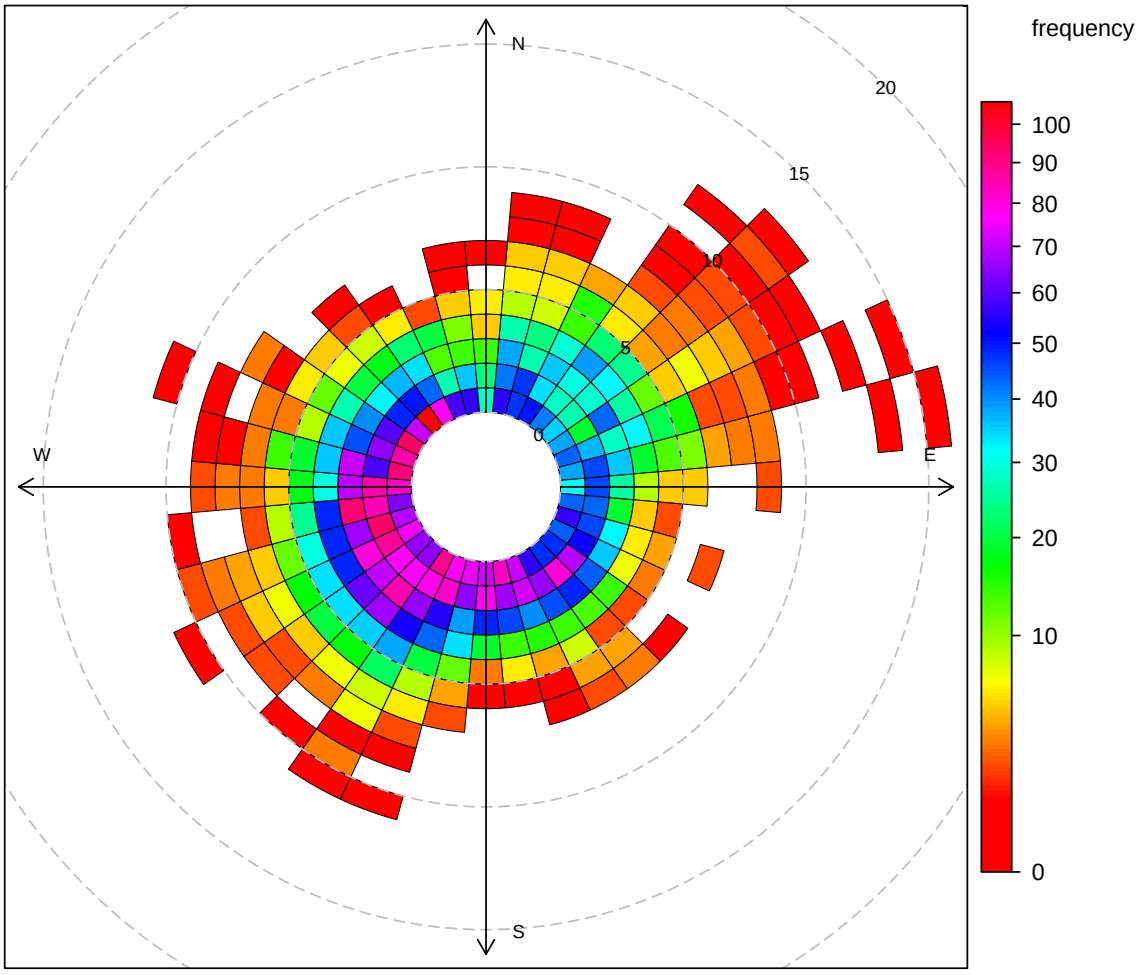


An important consideration with all polar plots is where to place the angle origin and which direction angles increase (clockwise or anti-clockwise), so all functions in this section provide some way to control these features. In this case, the start argument controls the origin and clockwise controls the direction.

Another way to view the distribution of these observations is provided by the polarFreq() function from the openair package. This function draws a polar image plot, with the color of each annulus sector determined by the number of points within that region. The data must be provided to this function with variables named ws for the radius variable and wd for the angle variable. There should also be a date variable, even though that is not always used.

```
#install.packages("openair")
library(openair)

## Warning:  package 'openair' was built under R version 3.3.3
## Loading required package:  maps
```

```
with(wind9am,
          polarFreq(data.frame(ws=Speed, wd=Dir, date=Date),
                                    cols=rainbow(256), border.col="black"))
```



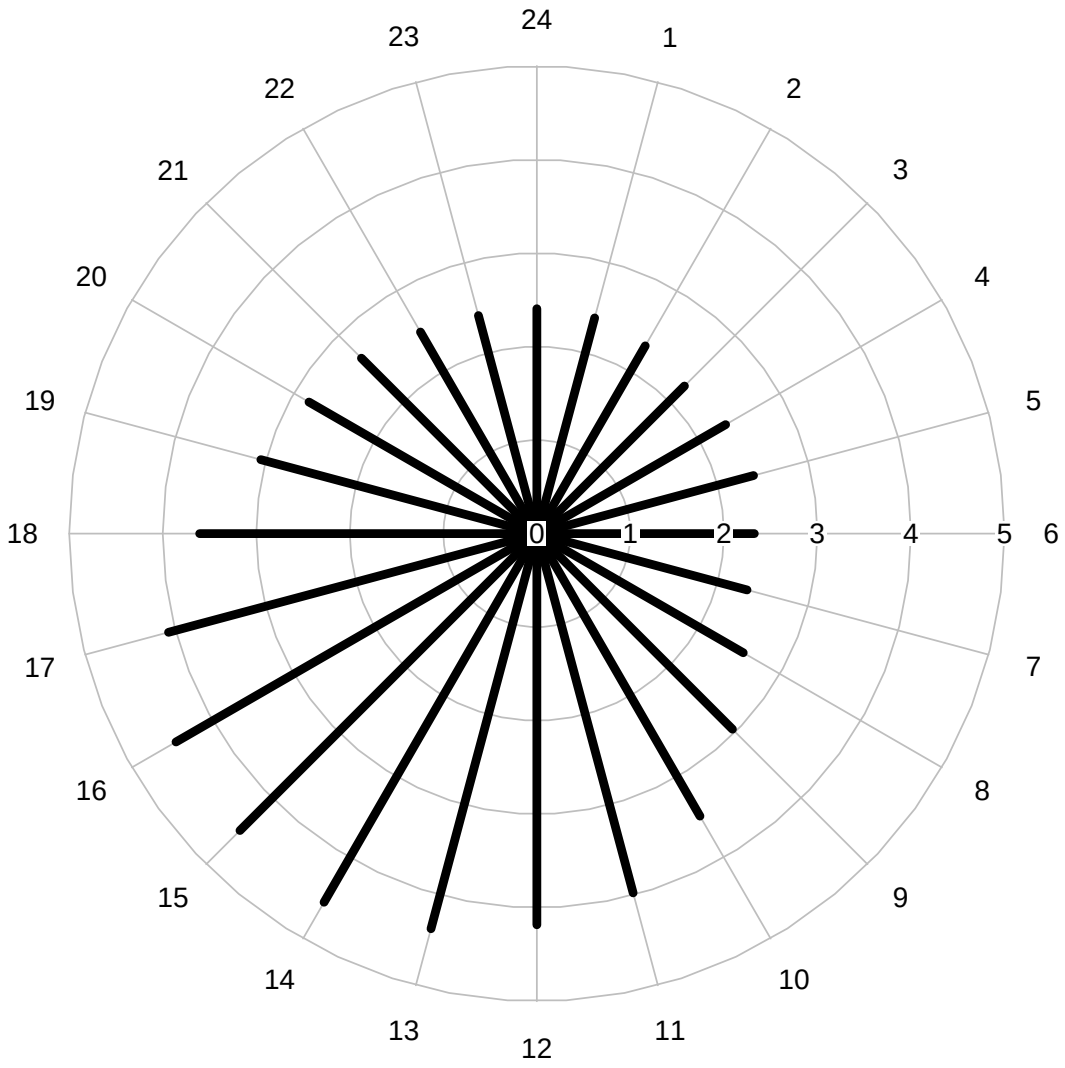
The functions in the openair package assume that the angle origin is pointing up (north) and that angles proceed clockwise, so there is no need to explicitly set the origin or angle direction in this case.

The previous two examples have demonstrated the use of points and areas within a polar plot. The next example shows one use of line segments, again using the polar.plot() function from plotrix. This plot is based on an aggregated form of the hourly wind data. The values are averaged across days to leave only 24 values, one for each hour of the day.

```
hourSpeed <- aggregate(hourlySpeed["Speed"],
                       list(hour=hourlySpeed$hour),
                       mean)
head(hourSpeed)
```

```
##   hour    Speed
## 1    0 2.404572
## 2    1 2.387312
## 3    2 2.318133
## 4    3 2.231682
## 5    4 2.330435
## 6    5 2.398445
```

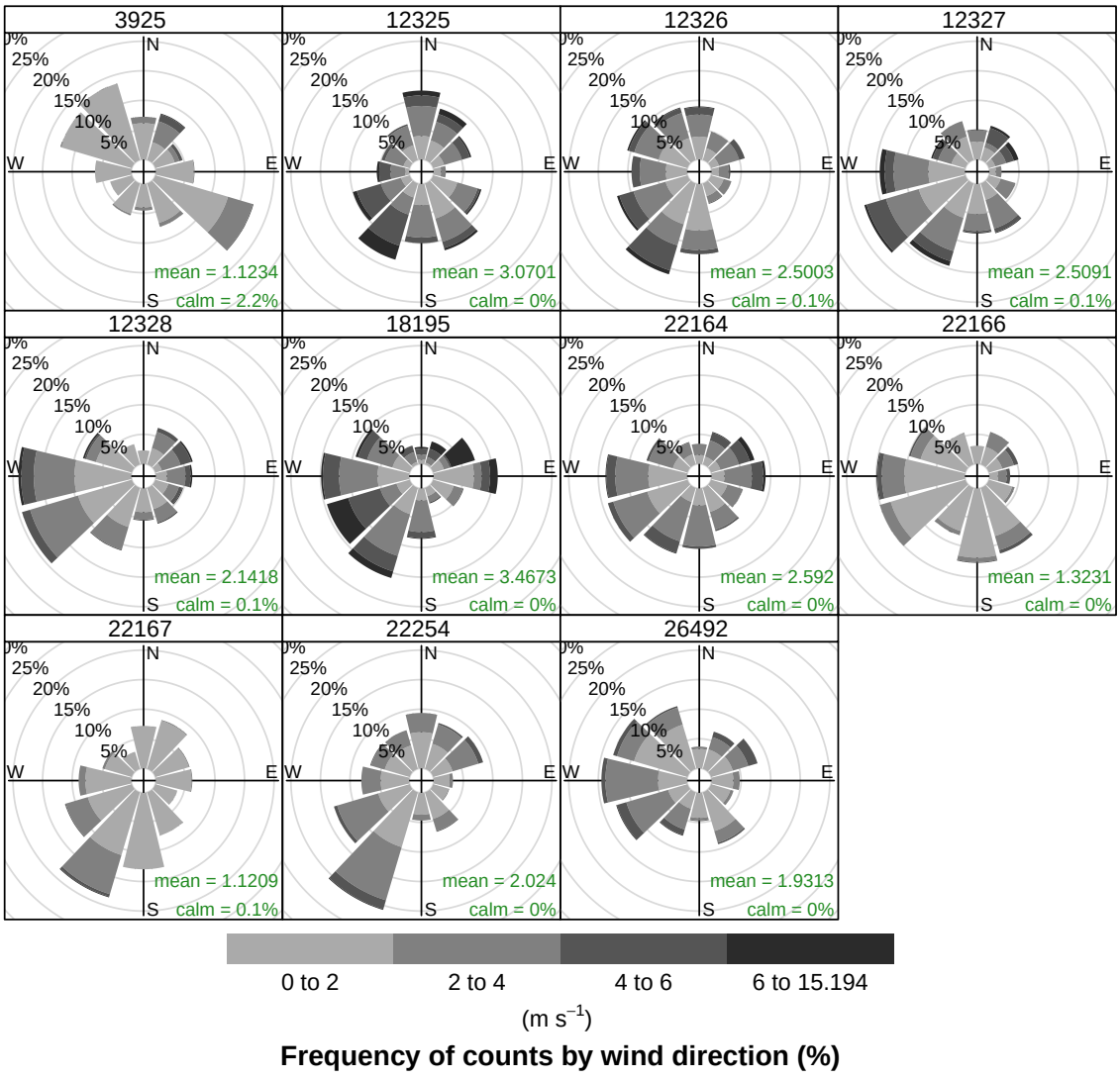
```
polar.plot(hourSpeed$Speed, hourSpeed$hour * 15,
           start=90, clockwise=TRUE, lwd=5,
           label.pos=seq(15, 360, 15), labels=1:24,
           radial.lim=c(0, 4.5))
```



Wind roses

In the specific case of plotting wind data, a specific style of plot called a wind rose is popular. This plots wedges (or "paddles") for a small set of angle ranges to represent how common each wind direction is. The wind speed is represented by breaking the wedge into separate bands, similar to a stacked bar chart. The openair package provides the windRose() function. As mentioned previously, this package is built on lattice so it is capable of multipanel wind roses.

```
with(wind9am,
     windRose(data.frame(ws=Speed, wd=Dir,
                         date=Date, station=factor(Station)),
              paddle=FALSE,
              type="station", cols=gray(4:1/6), width=2))
```



2 ggplot2

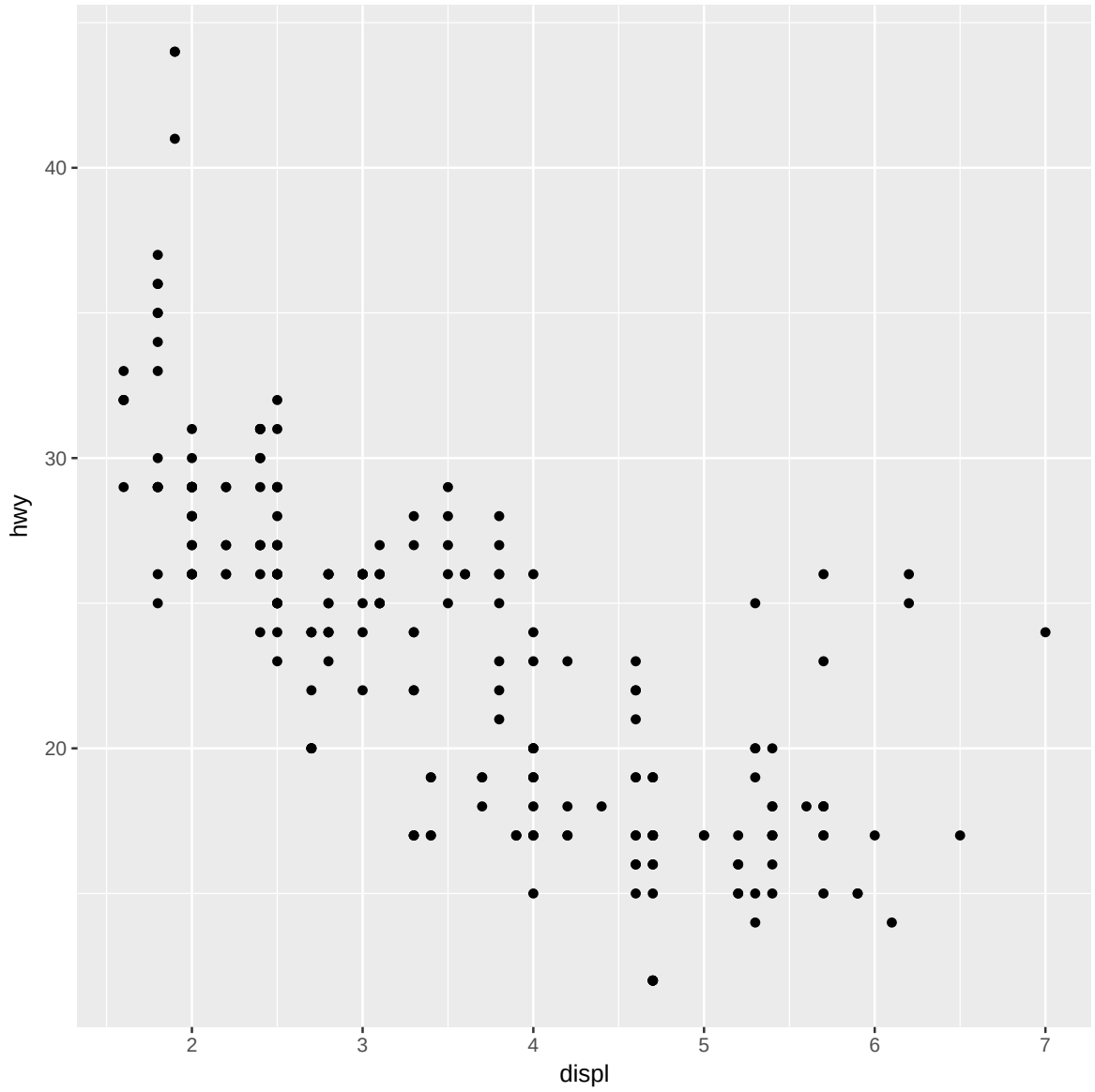
The ggplot2 package implements a system for creating graphics in R based on a comprehensive and coherent grammar. This provides a consistency to graph creation often lacking in R, and allows the user to create graph types that are innovative and novel.

2.1 qplot

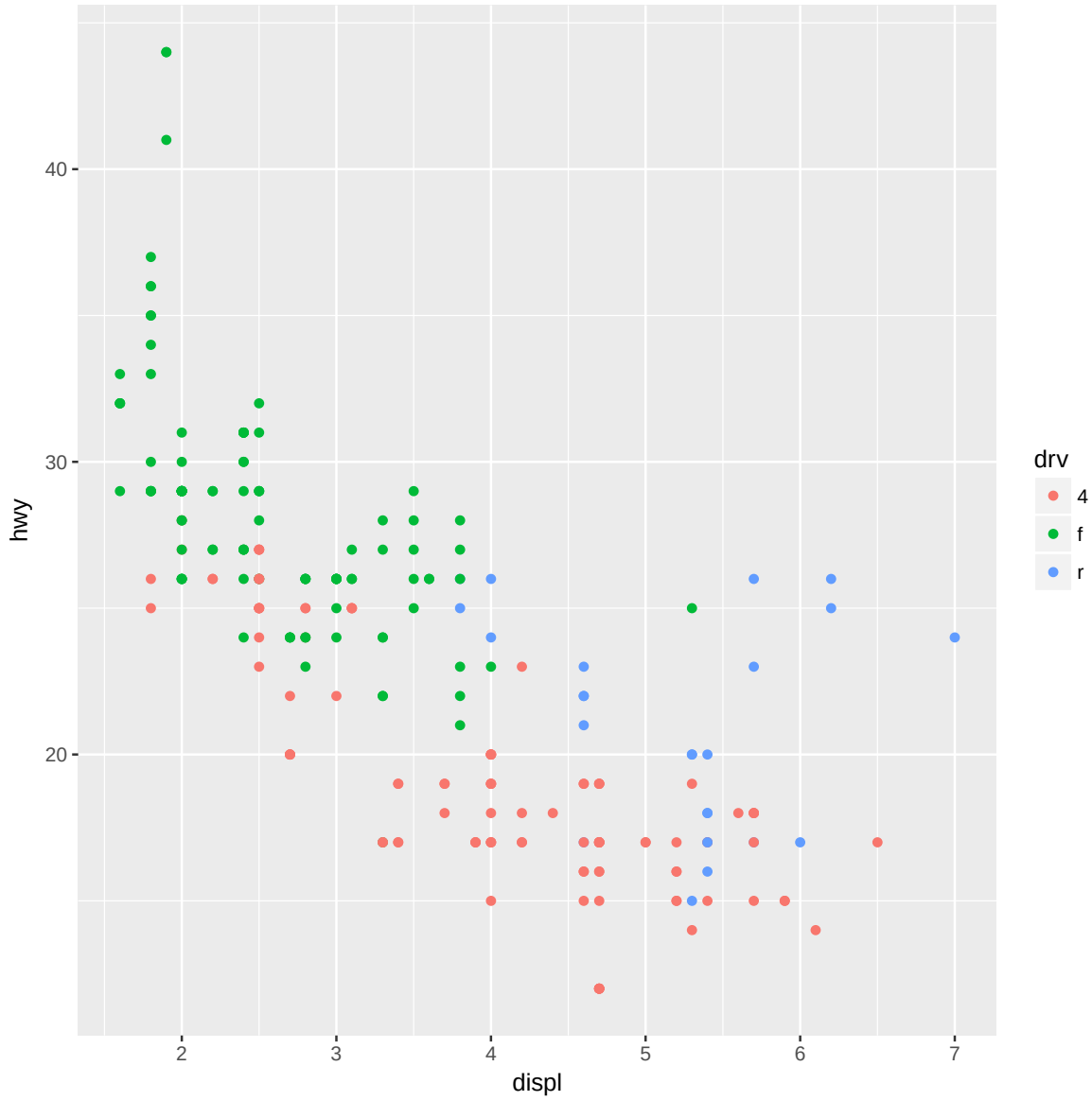
An example

The simplest approach for creating graphs in ggplot2 is through the qplot() or quick plot function.

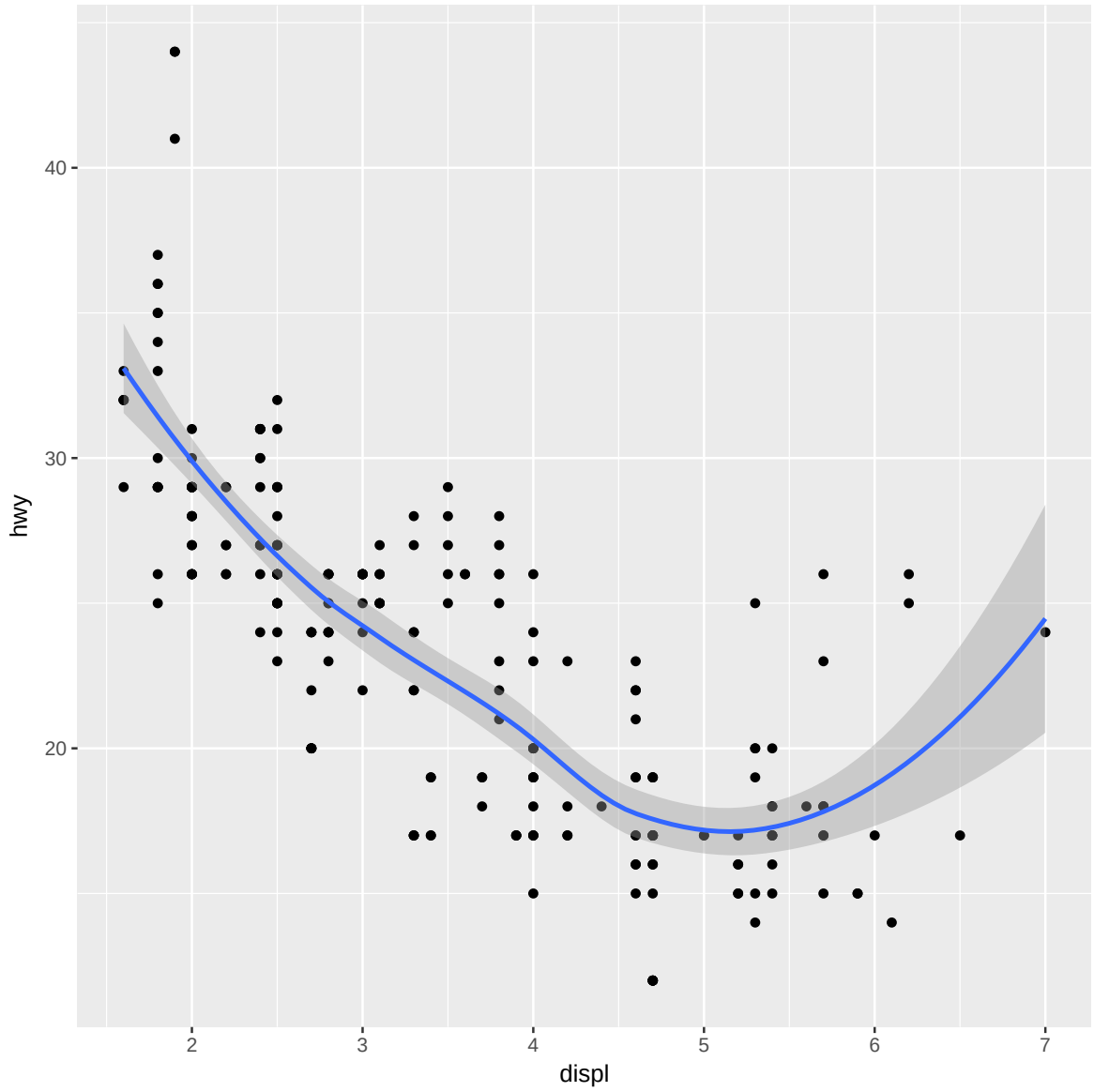
```
library("ggplot2")

## Warning:  package 'ggplot2' was built under R version 3.3.3

str(mpg)

## Classes 'tbl_df', 'tbl' and 'data.frame': 234 obs. of  11 variables:
##  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
##  $ model       : chr  "a4" "a4" "a4" "a4" ...
##  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr  "p" "p" "p" "p" ...
##  $ class       : chr  "compact" "compact" "compact" "compact" ...

qplot(displ,hwy,data=mpg)
```




```
qplot(displ,hwy,data=mpg,color=drv)
```

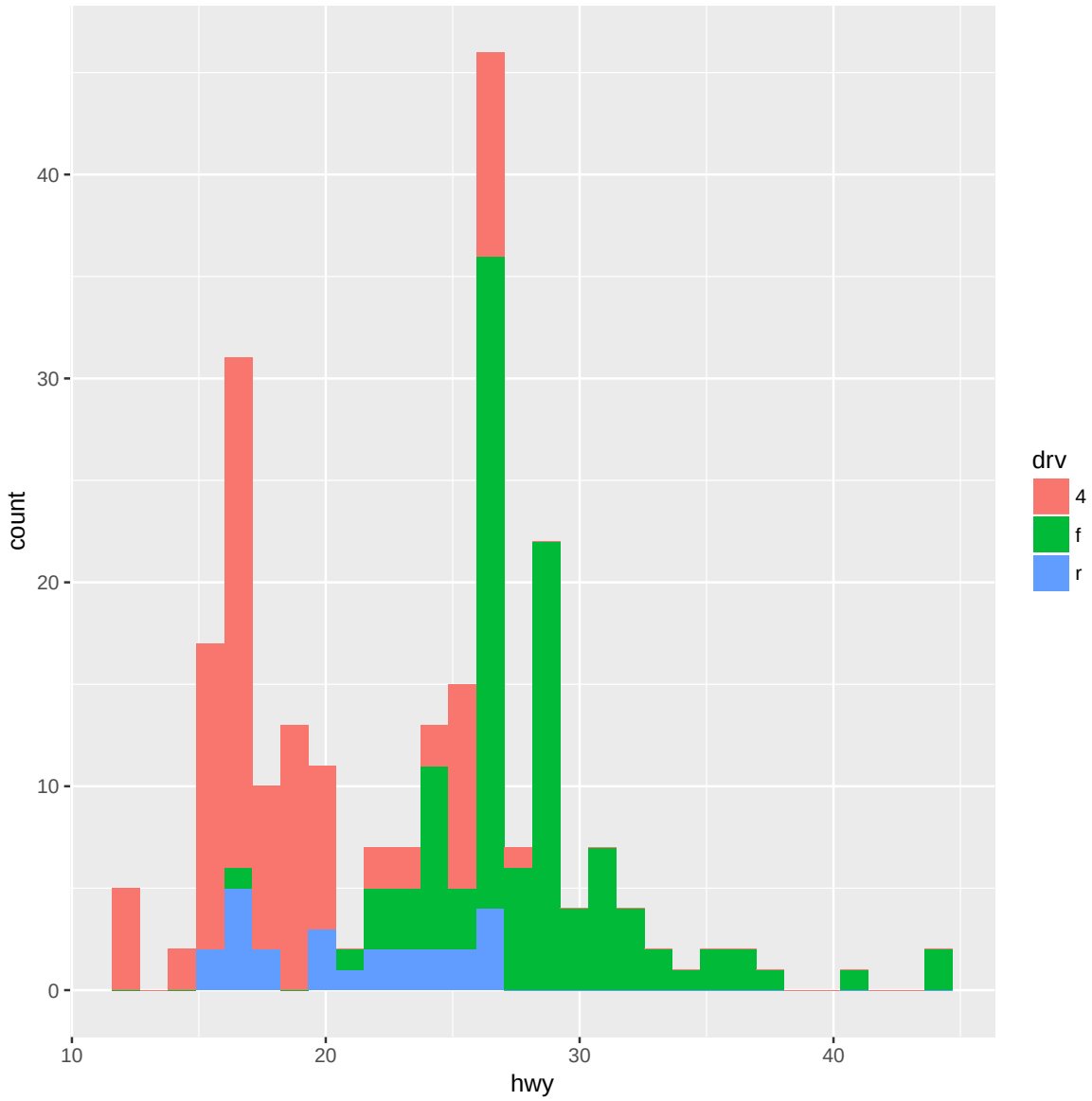



```
qplot(displ, hwy, data = mpg, geom = c("point", "smooth"))

## `geom_smooth()` using method = 'loess'
```

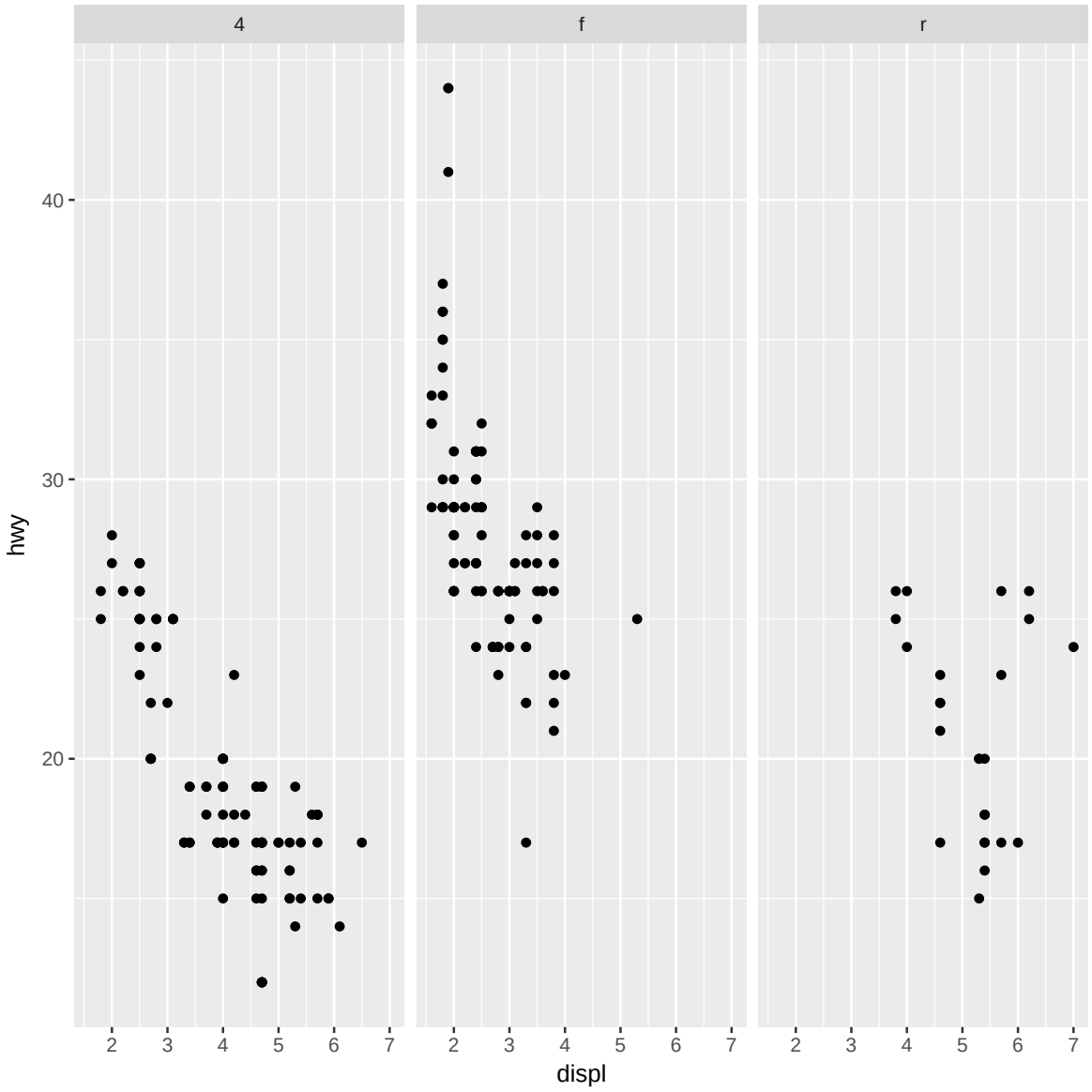



```
qplot(hwy, data = mpg, fill = drv)
```

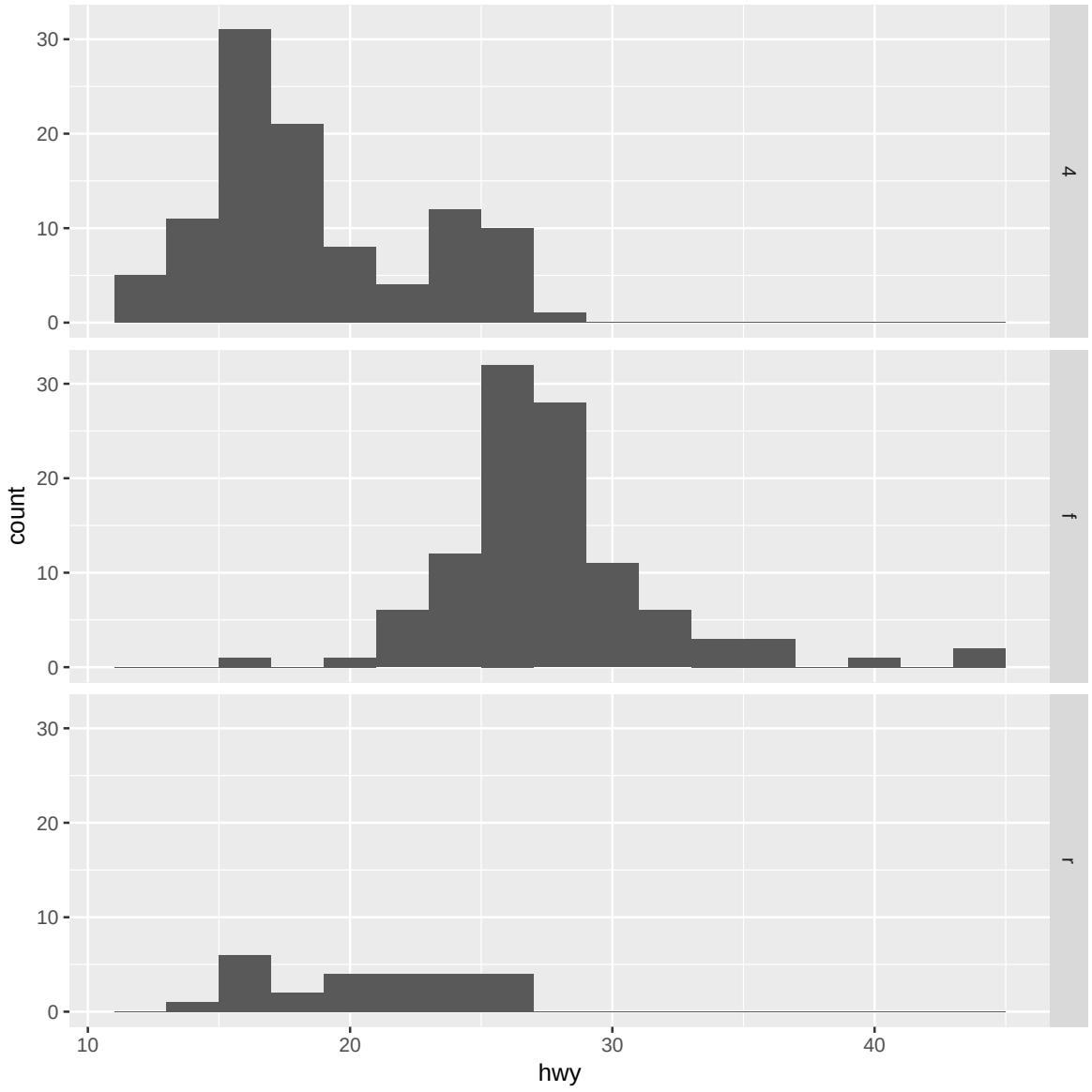
```
## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```



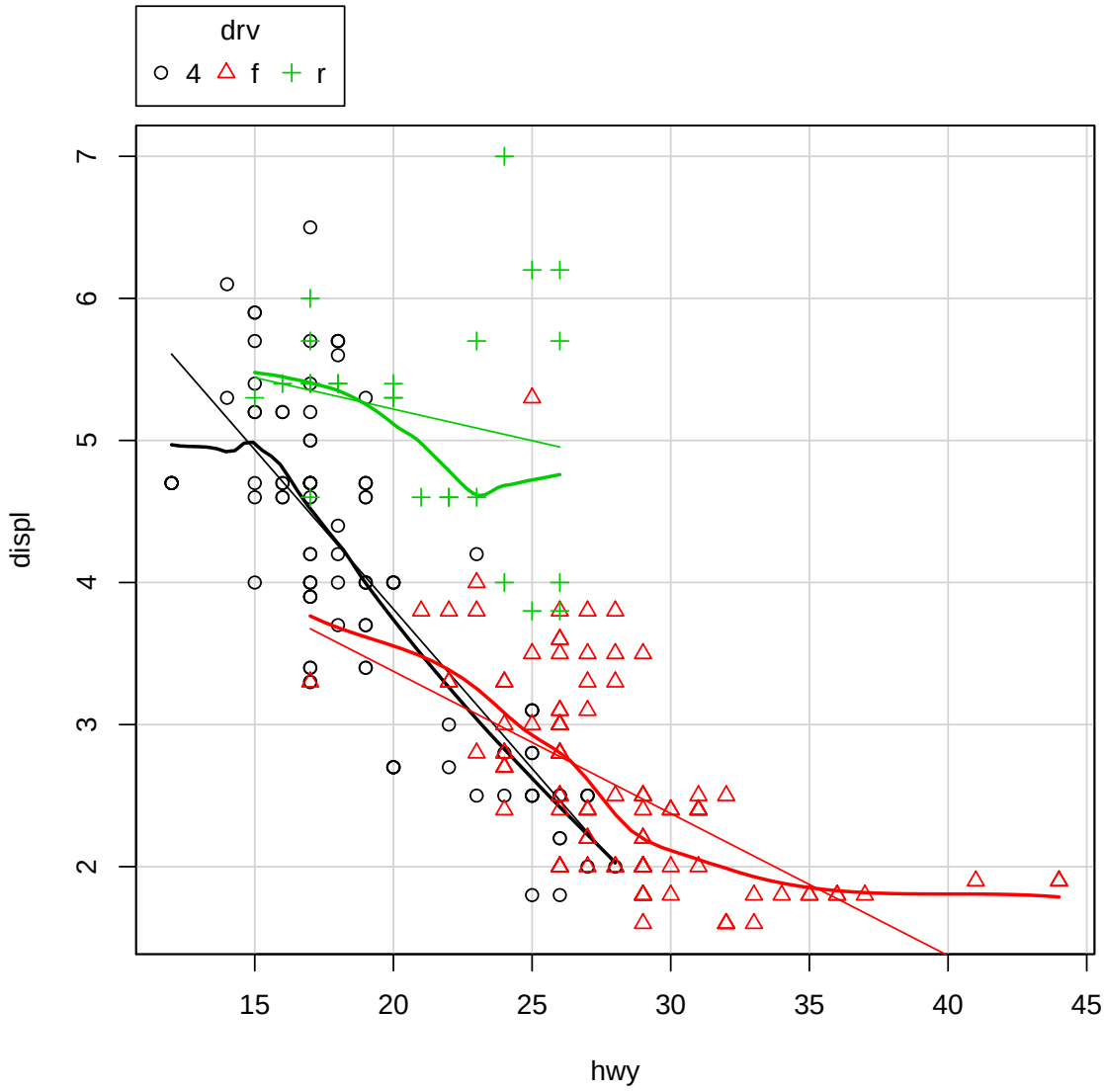

```
qplot(displ, hwy, data = mpg, facets = . ~ drv)
```



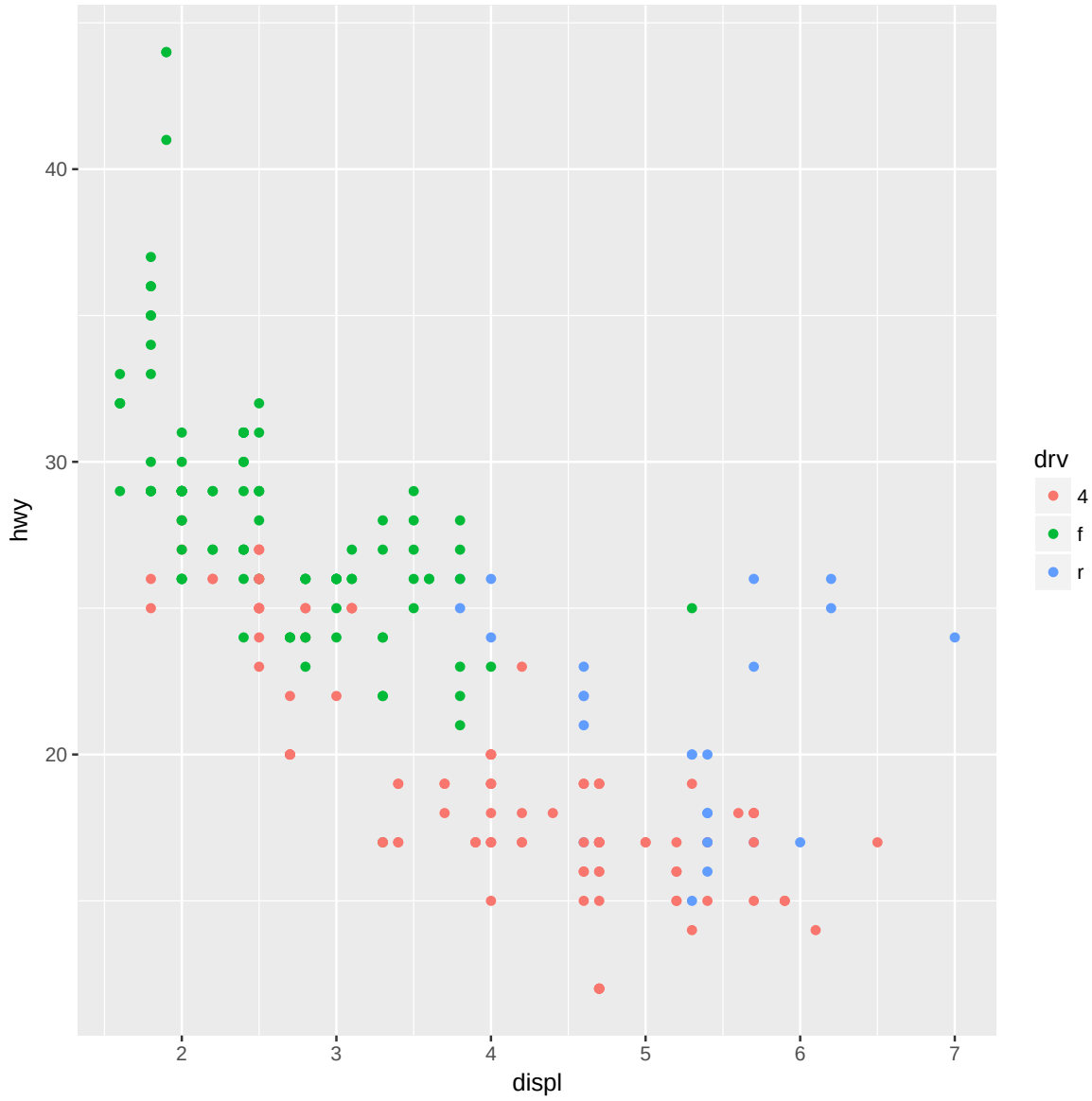

```
qplot(hwy, data = mpg, facets = drv ~ ., binwidth = 2)
```



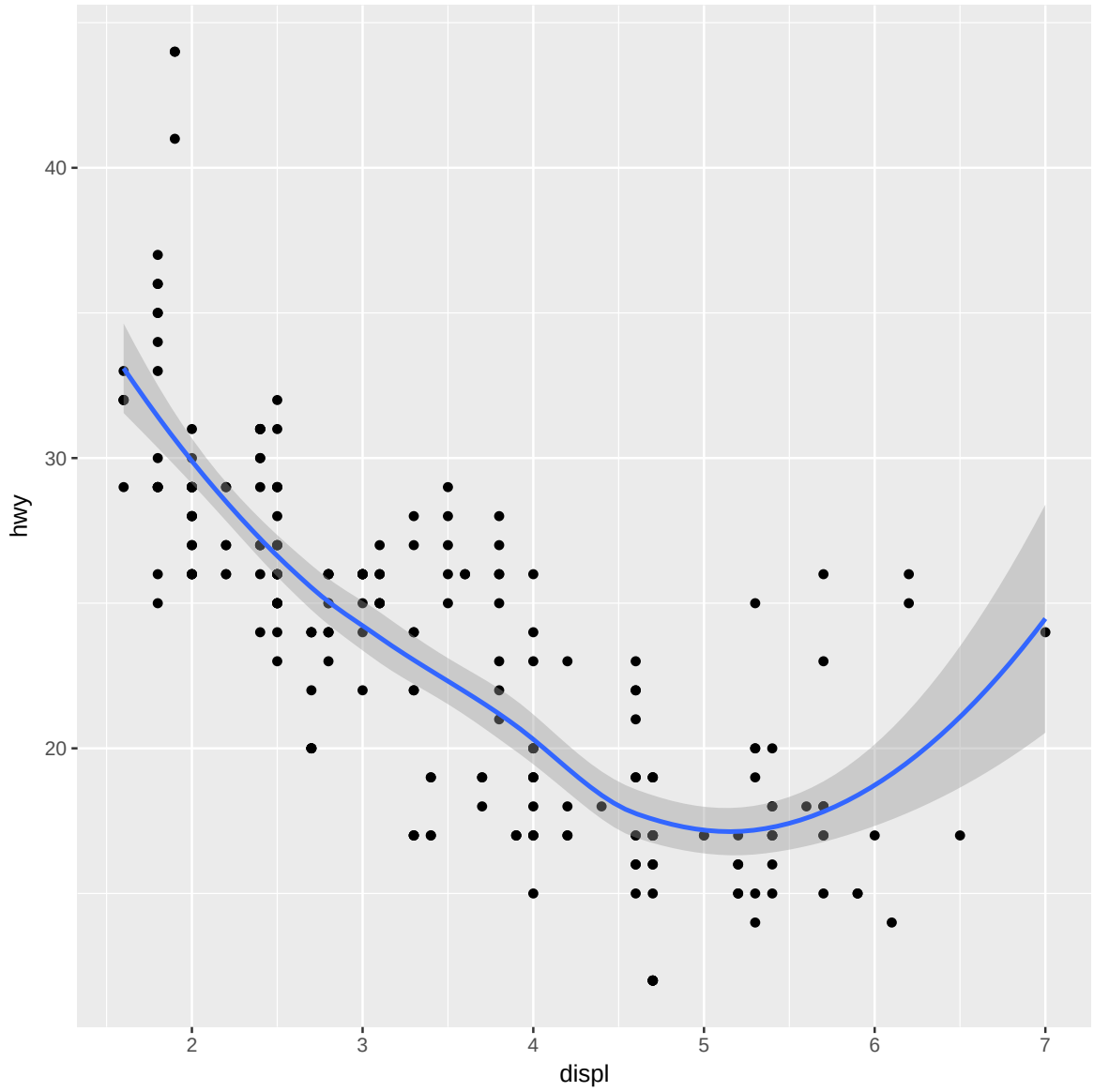

```
library(car)
scatterplot(displ~hwy|drv,data=mpg)
```



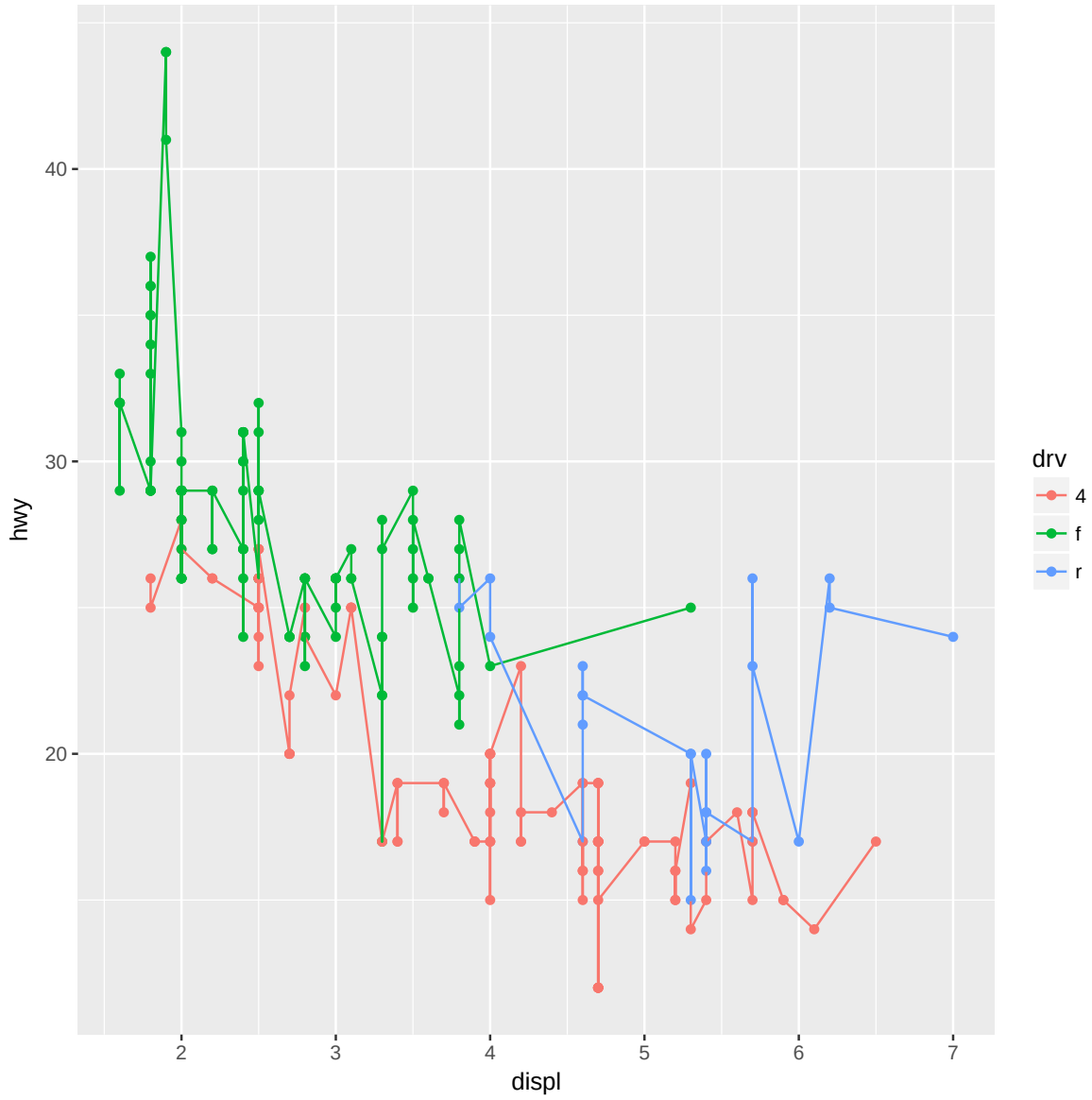

```
qplot(displ,hwy,data=mpg,color=drv)
```



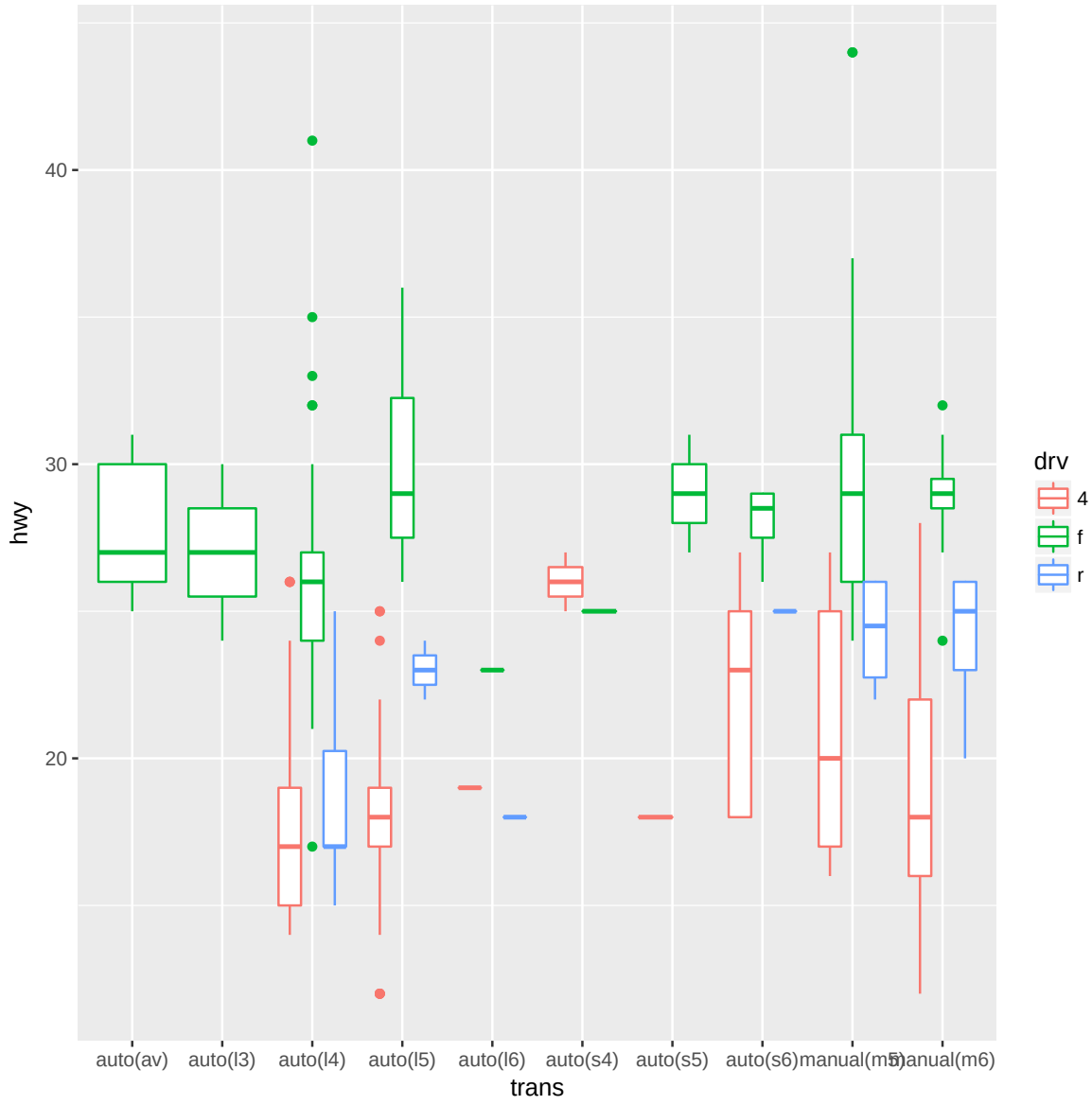

```
###point+line
qplot(displ, hwy, data = mpg, geom = c("point", "smooth"))

## `geom_smooth()` using method = 'loess'
```



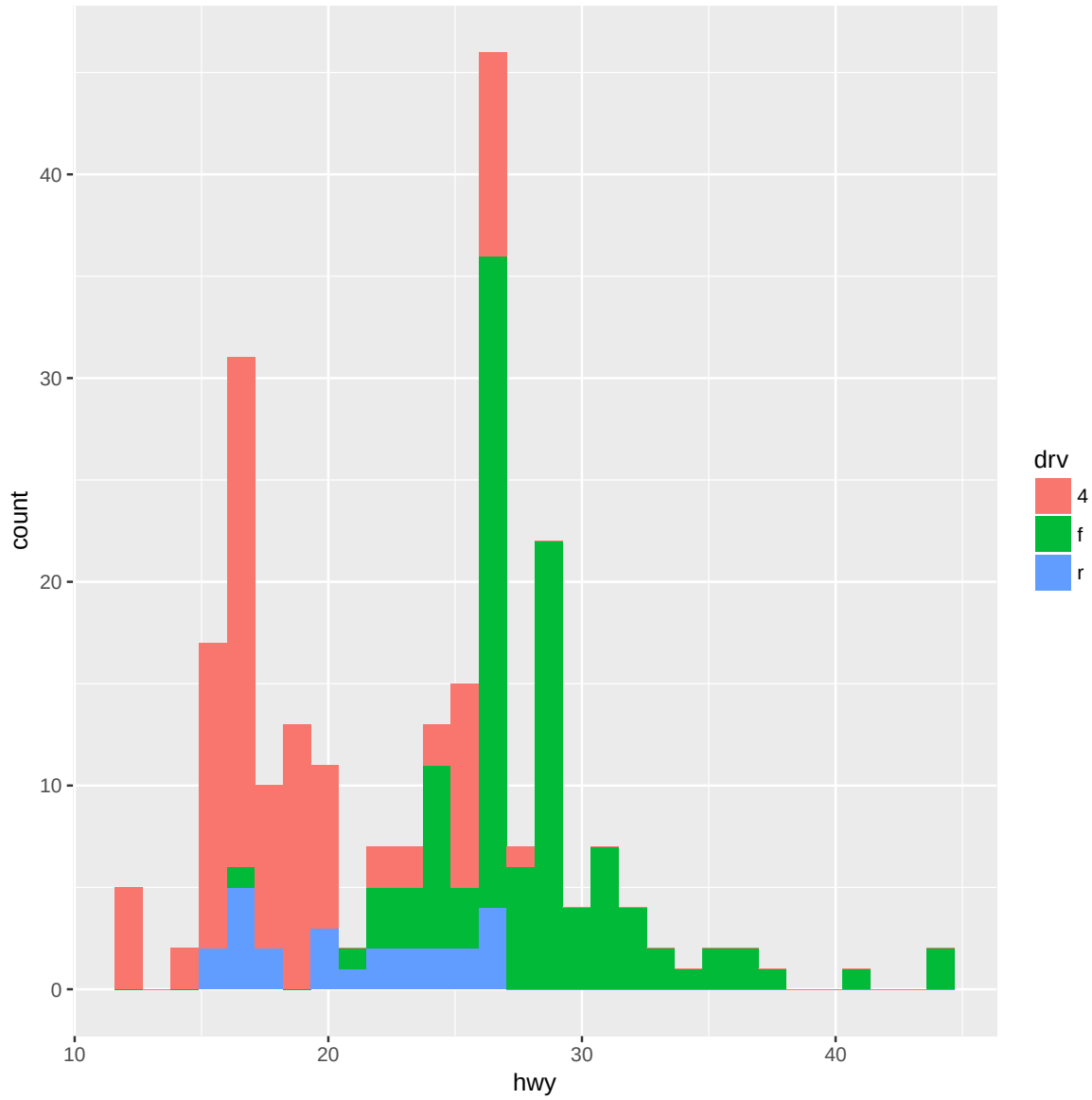

```
qplot(displ, hwy, data = mpg, color= drv, geom = c("point", "line"))
```



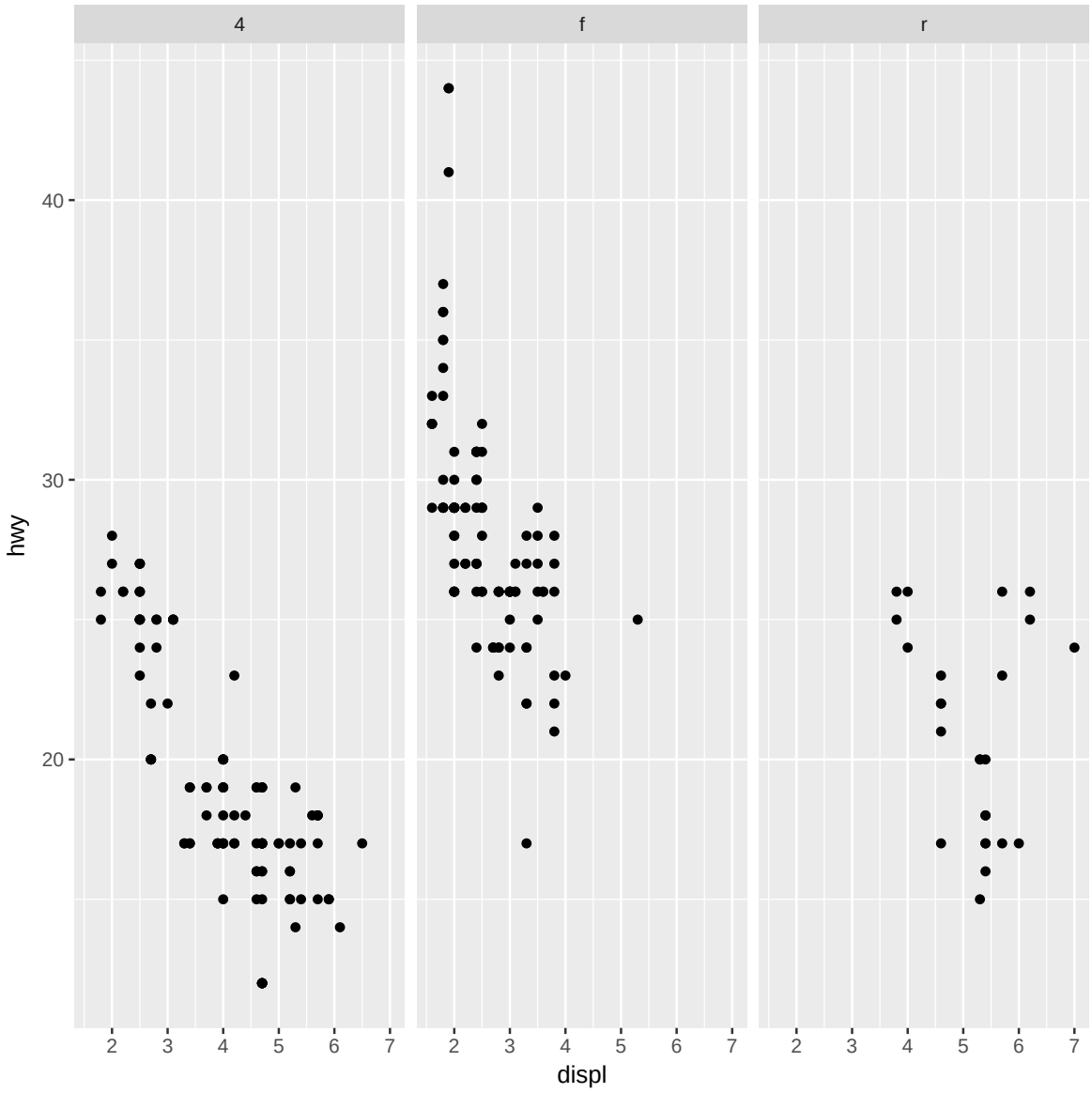

```
#boxplot
qplot(trans, hwy, data = mpg, color= drv, geom = "boxplot")
```



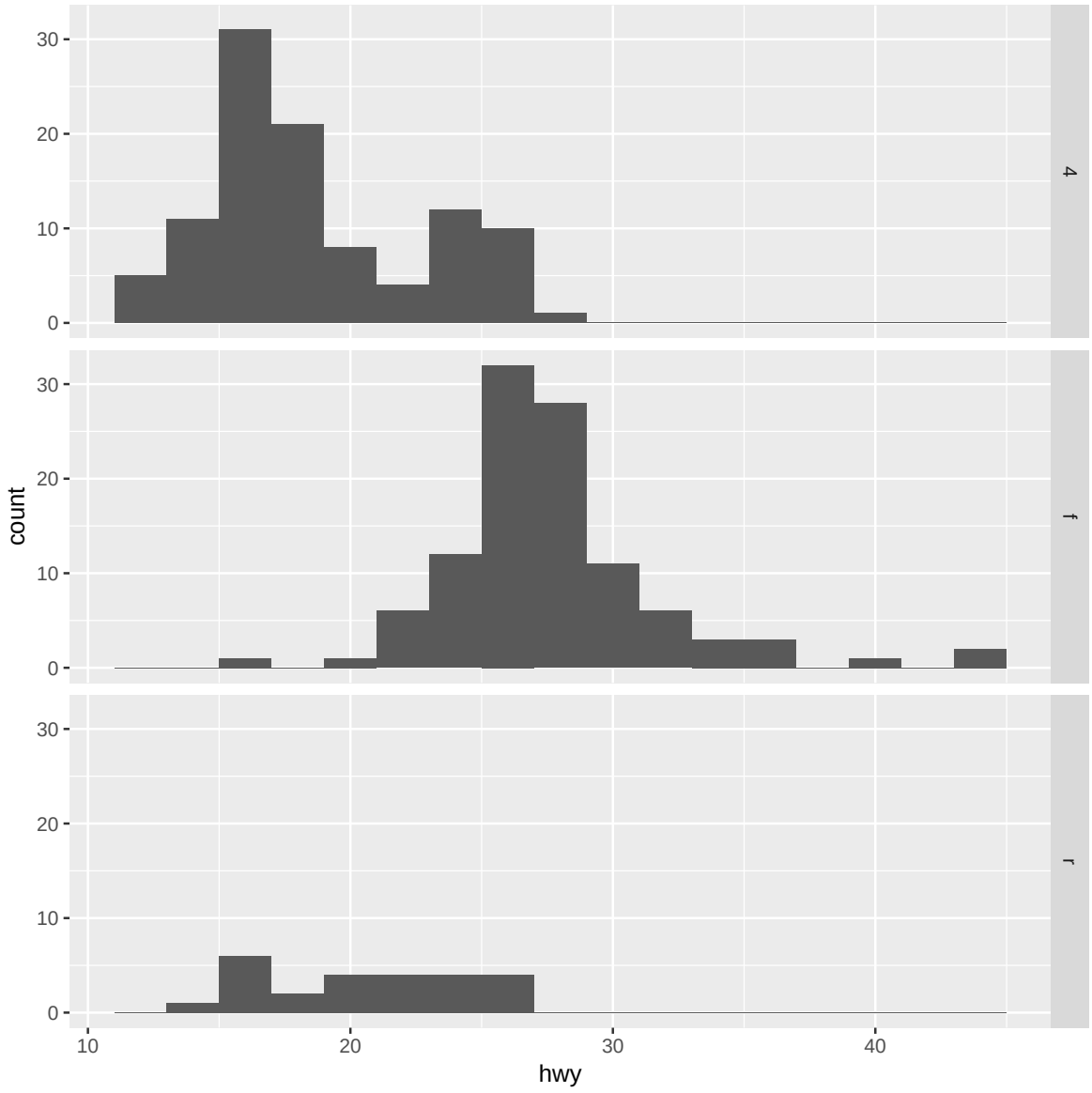

```
#Histogram
qplot(hwy, data = mpg, fill= drv, geom = "histogram")

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```




```
#qplot(hwy, data = mpg, fill = drv)
#Density qplot(hwy, data = mpg, color= drv, geom = "density")
#multi graph
qplot(displ, hwy, data = mpg, facets = . ~ drv)
```




```
qplot(hwy, data = mpg, facets = drv ~ ., binwidth = 2)
```




```
# save image of plot on disk
#ggsave(file="test.pdf")
#ggsave(file="test.jpeg", dpi=72)
#ggsave(file="test.svg", plot=p.tmp, width=10, height=5)
```

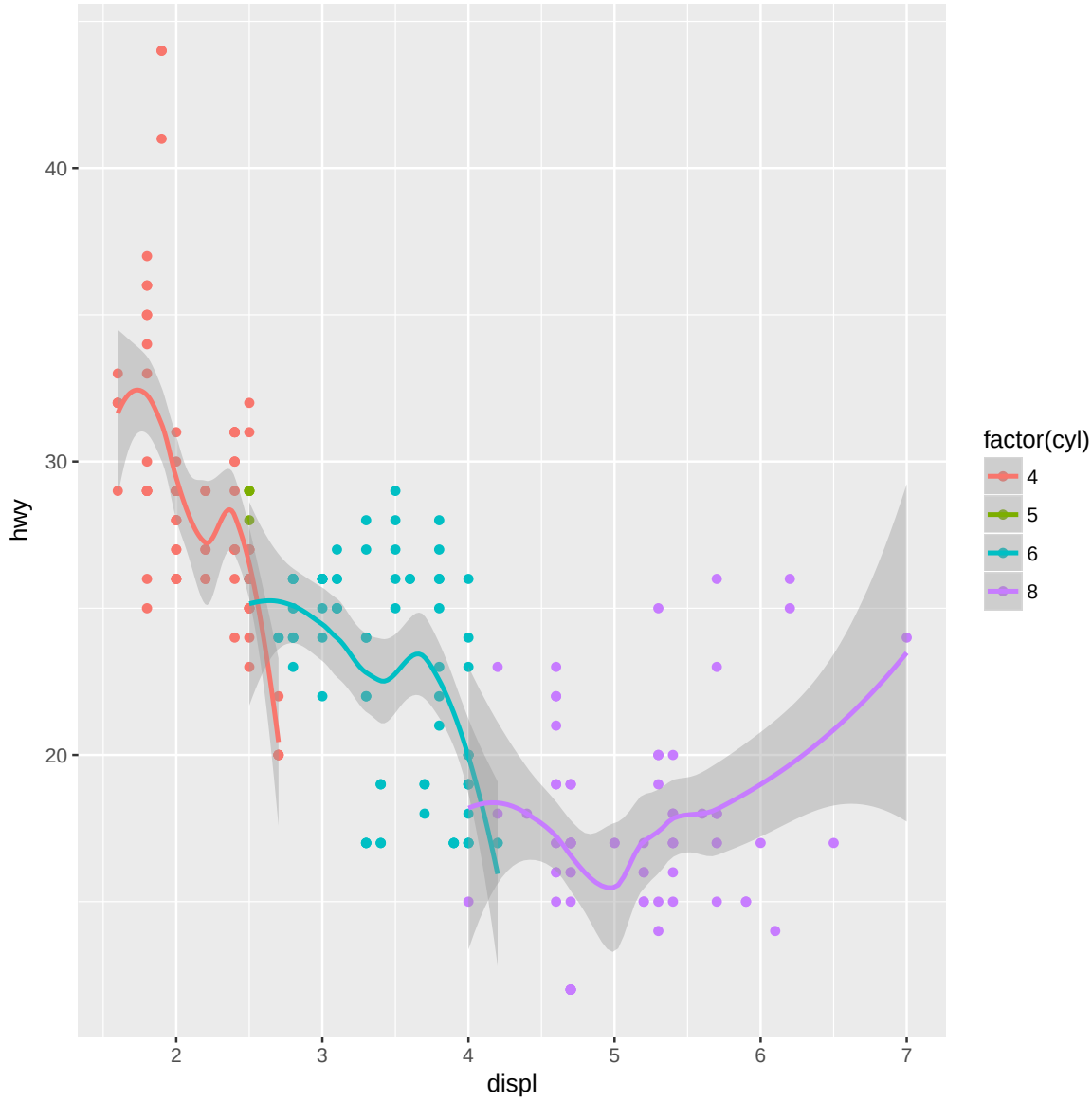
2.2 ggplot

Compare qplot and ggplot

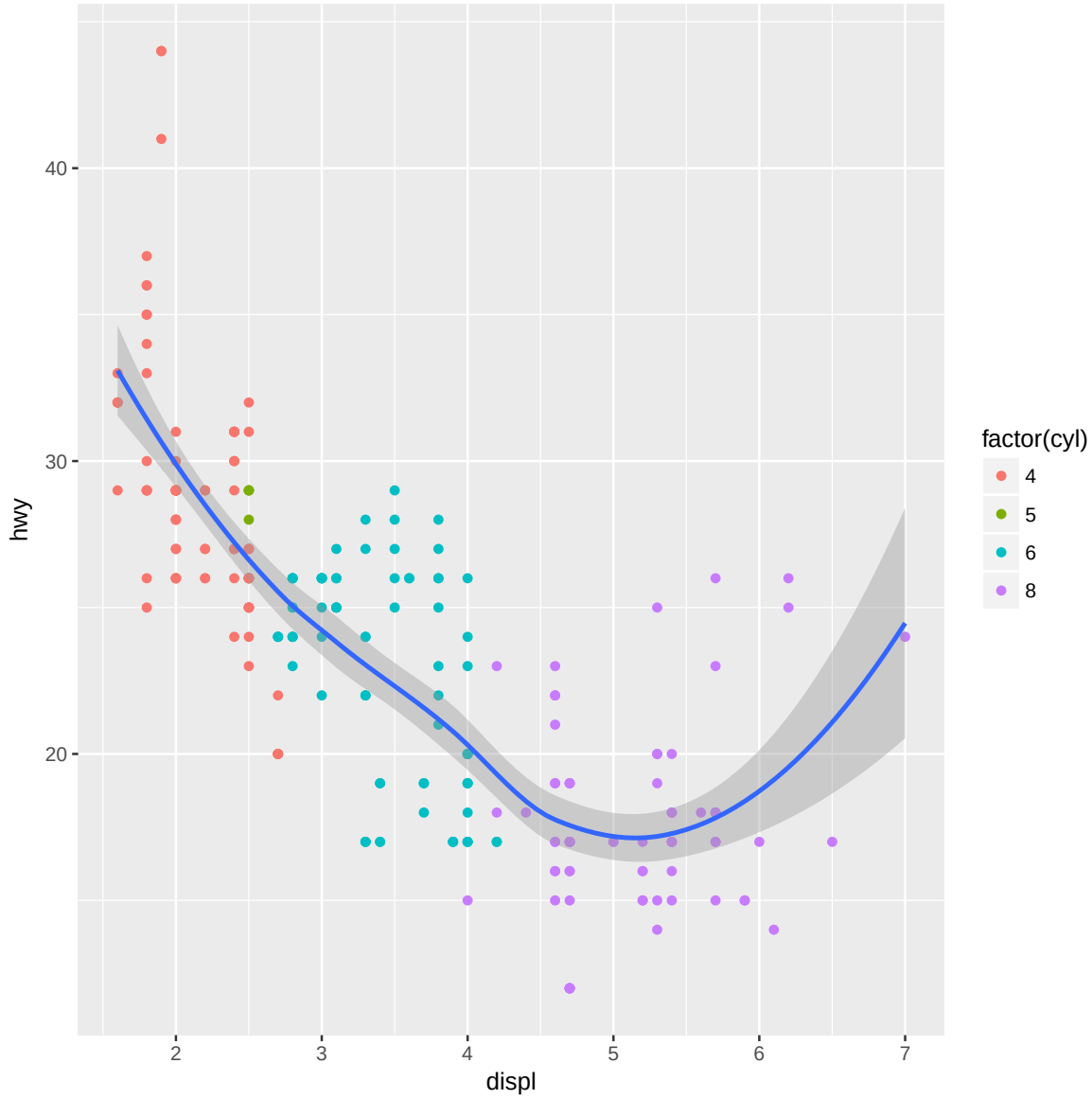
```
library("ggplot2")
### comparison qplot vs ggplot
# qplot histogram
```

```r
qplot(clarity, data=diamonds, fill=cut, geom="bar")
# ggplothistogram -> same output
ggplot(diamonds, aes(clarity, fill=cut)) + geom_bar()
```

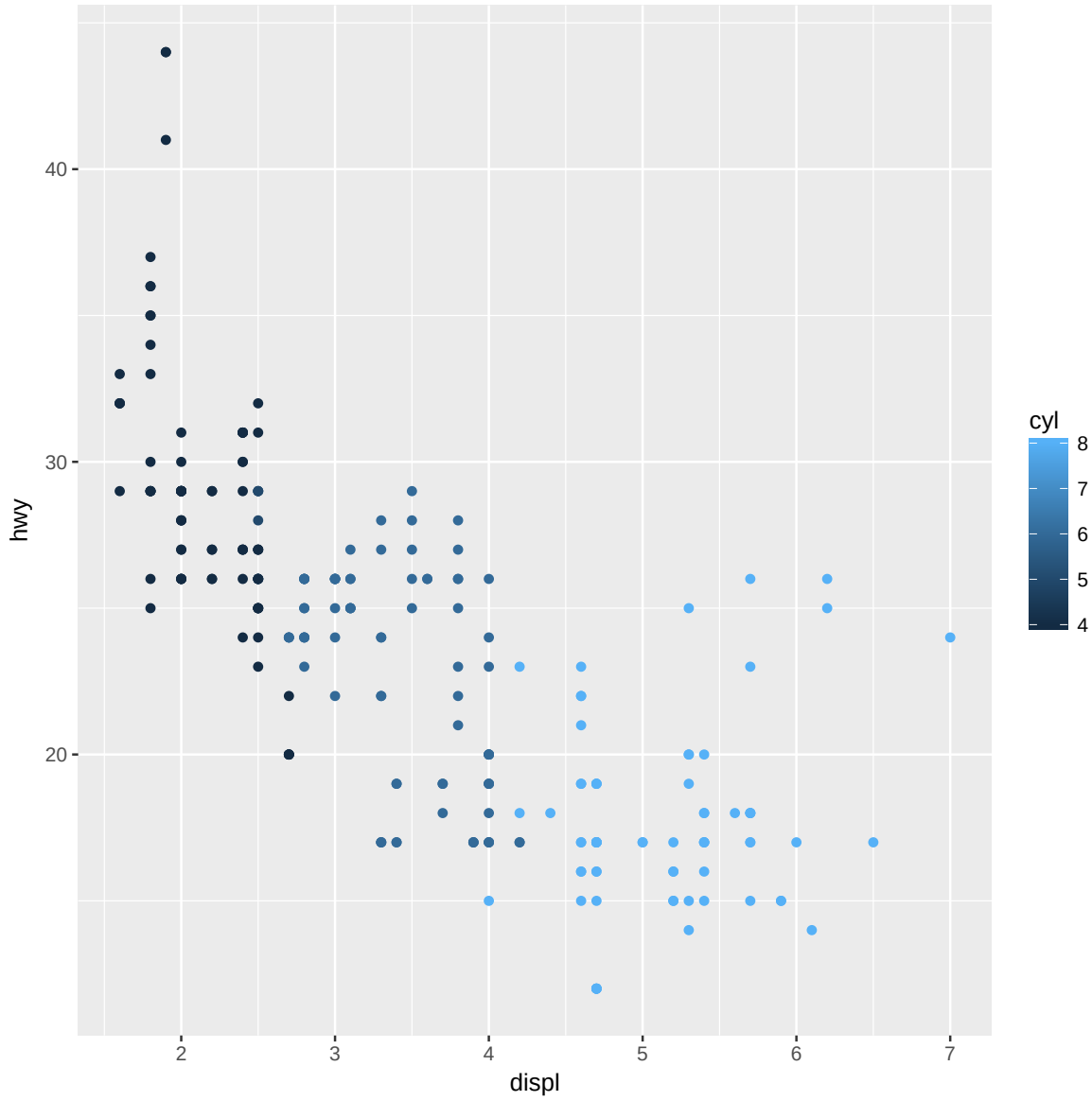
```r
library(ggplot2)
#1
p <- ggplot(data=mpg,aes(x=displ,y=hwy,colour=factor(cyl)))
p + geom_point() + geom_smooth()

## `geom_smooth()` using method = 'loess'
```



```
#2
p <- ggplot(mpg,aes(x=displ,y=hwy))
p + geom_point(aes(colour=factor(cyl))) + geom_smooth()

## `geom_smooth()` using method = 'loess'
```

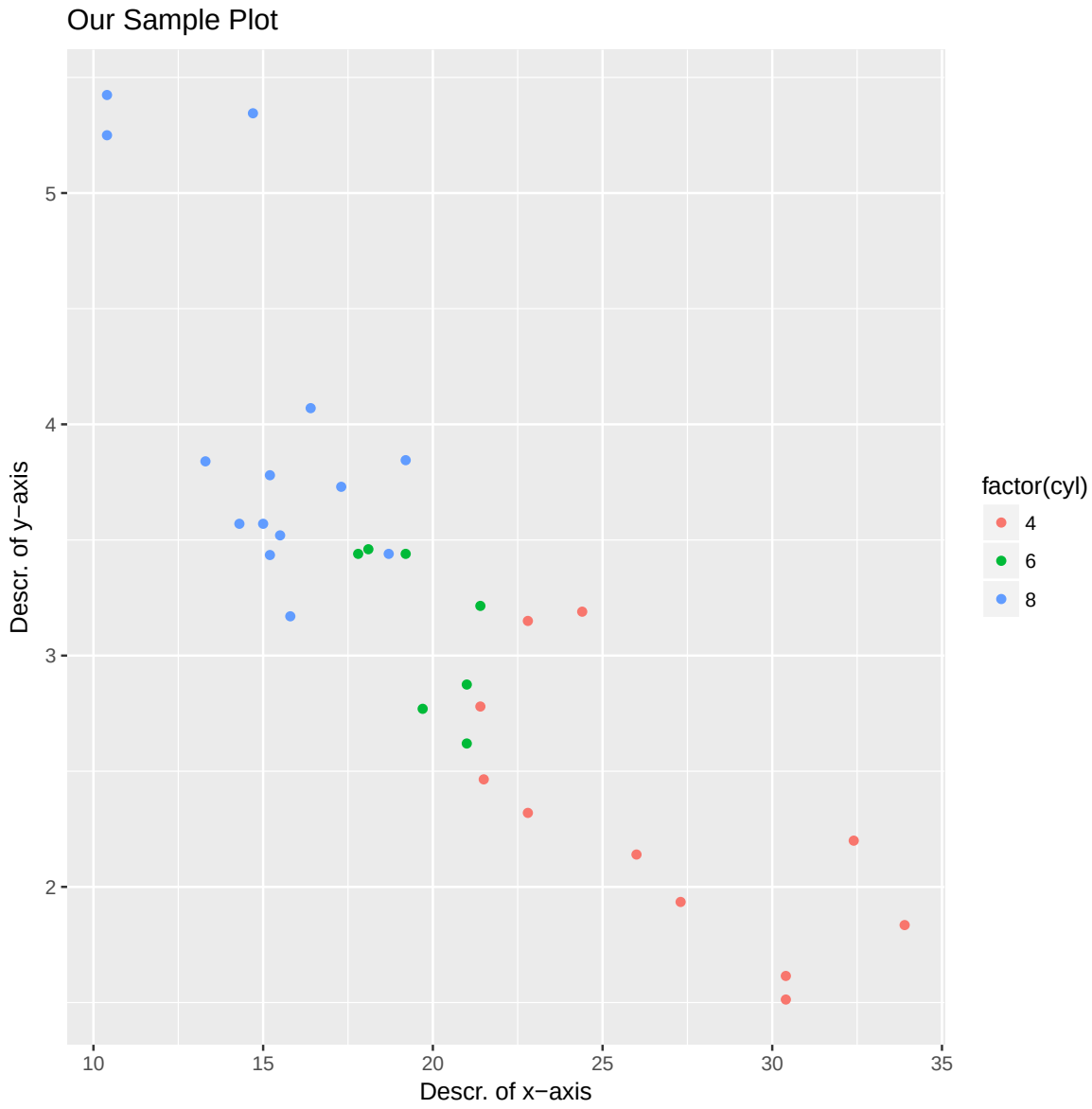



```
#3
p <- ggplot(mpg, aes(x=displ,y=hwy))
p + geom_point(aes(colour = cyl))
```

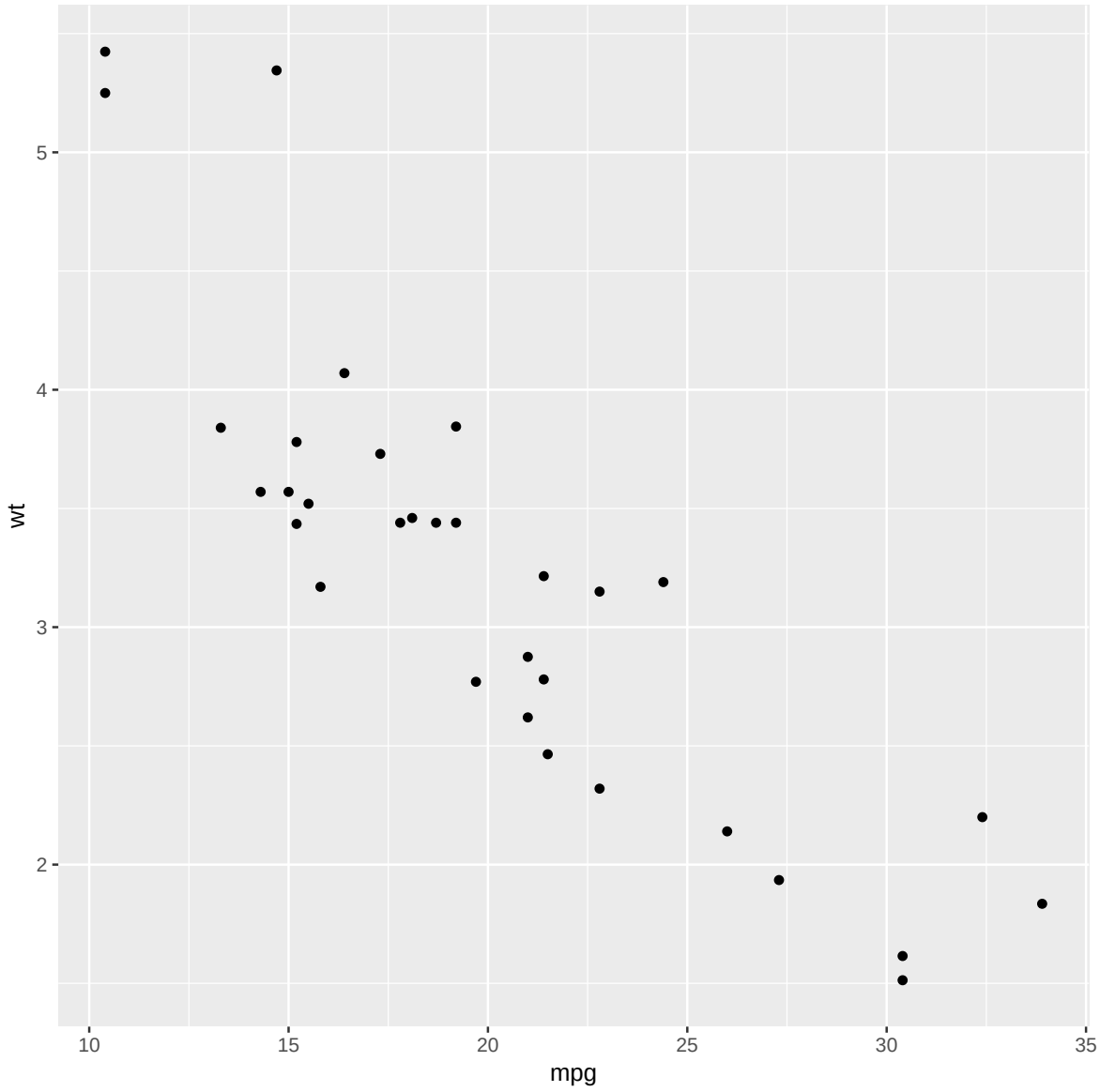


2.3 Other features

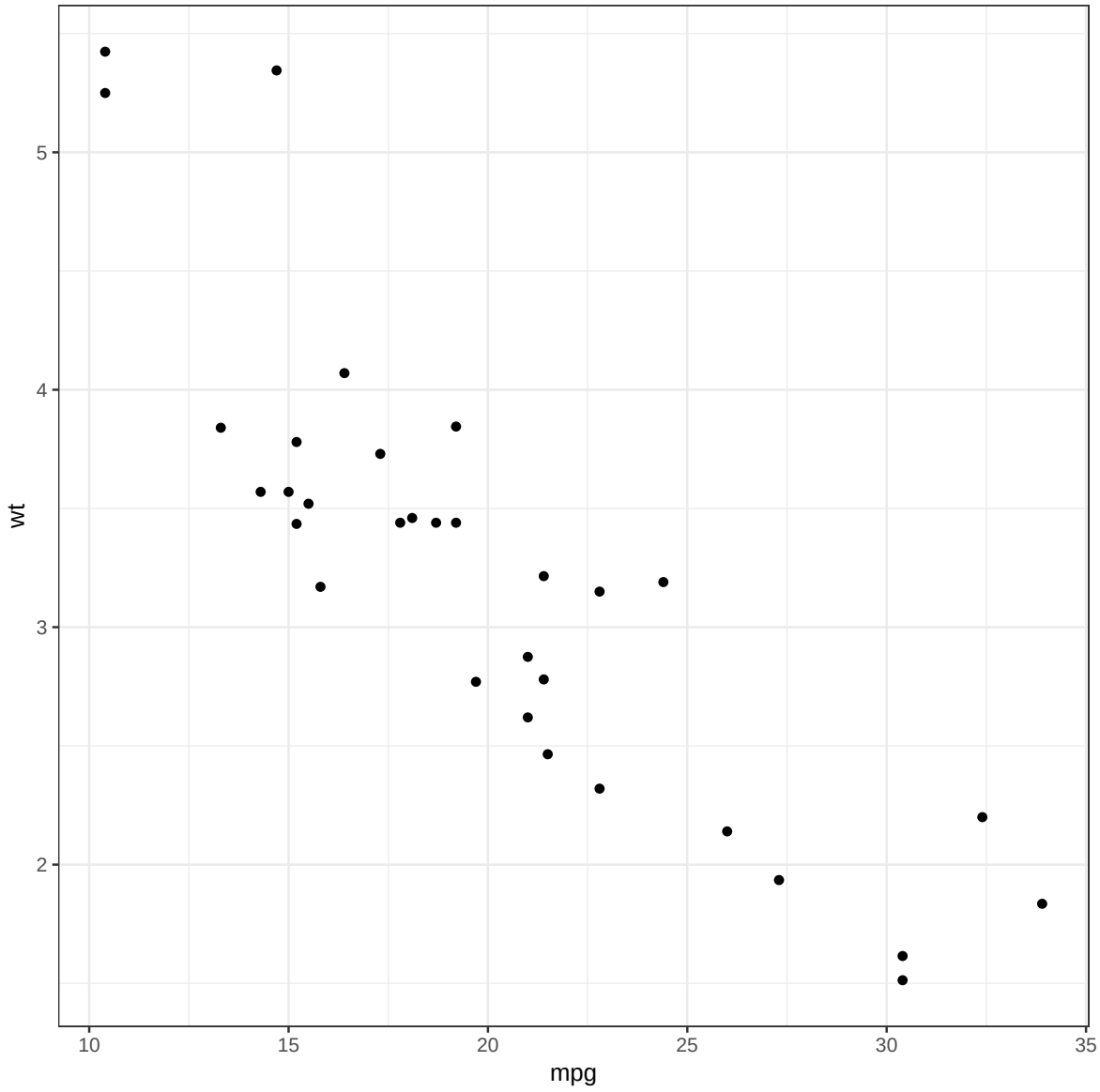
```
# changing text (directly in qplot / additional shortcut)
qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point", xlab="Descr. of x-axis", ylab="Descr. o
```



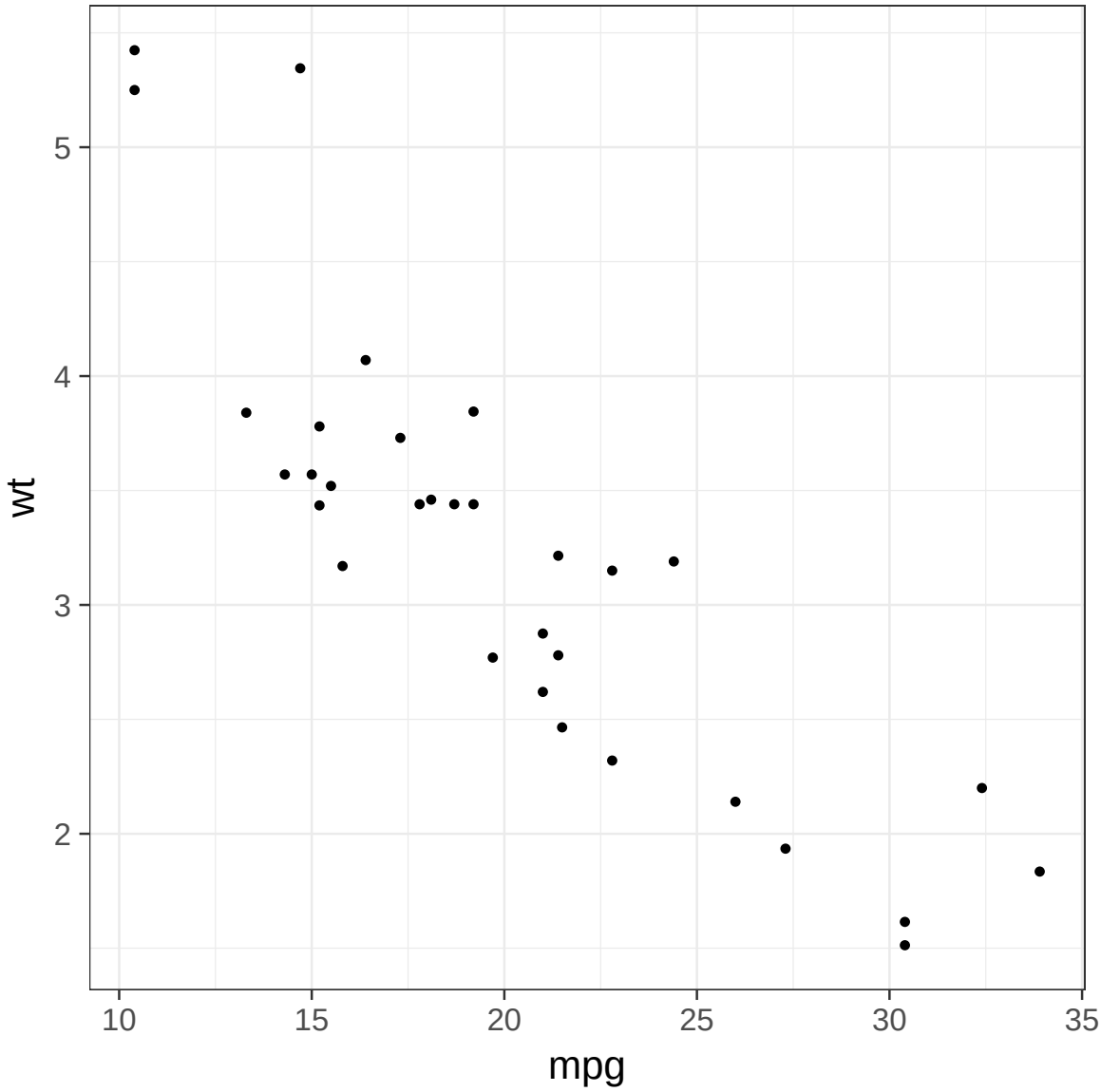

```
# use theme for plot only
qplot(mpg, wt, data=mtcars, geom="point")
```




```
qplot(mpg, wt, data=mtcars, geom="point") + theme_bw()
```




```
# change font-size for all labels (change base_size)
qplot(mpg, wt, data=mtcars, geom="point") + theme_bw(18)
```



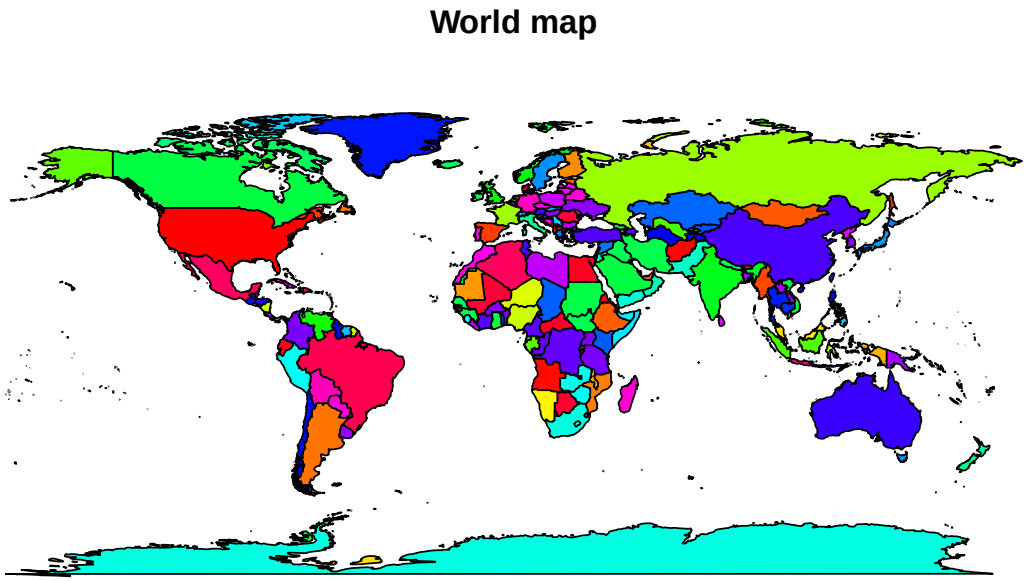
3 Maps

3.1 The maps package

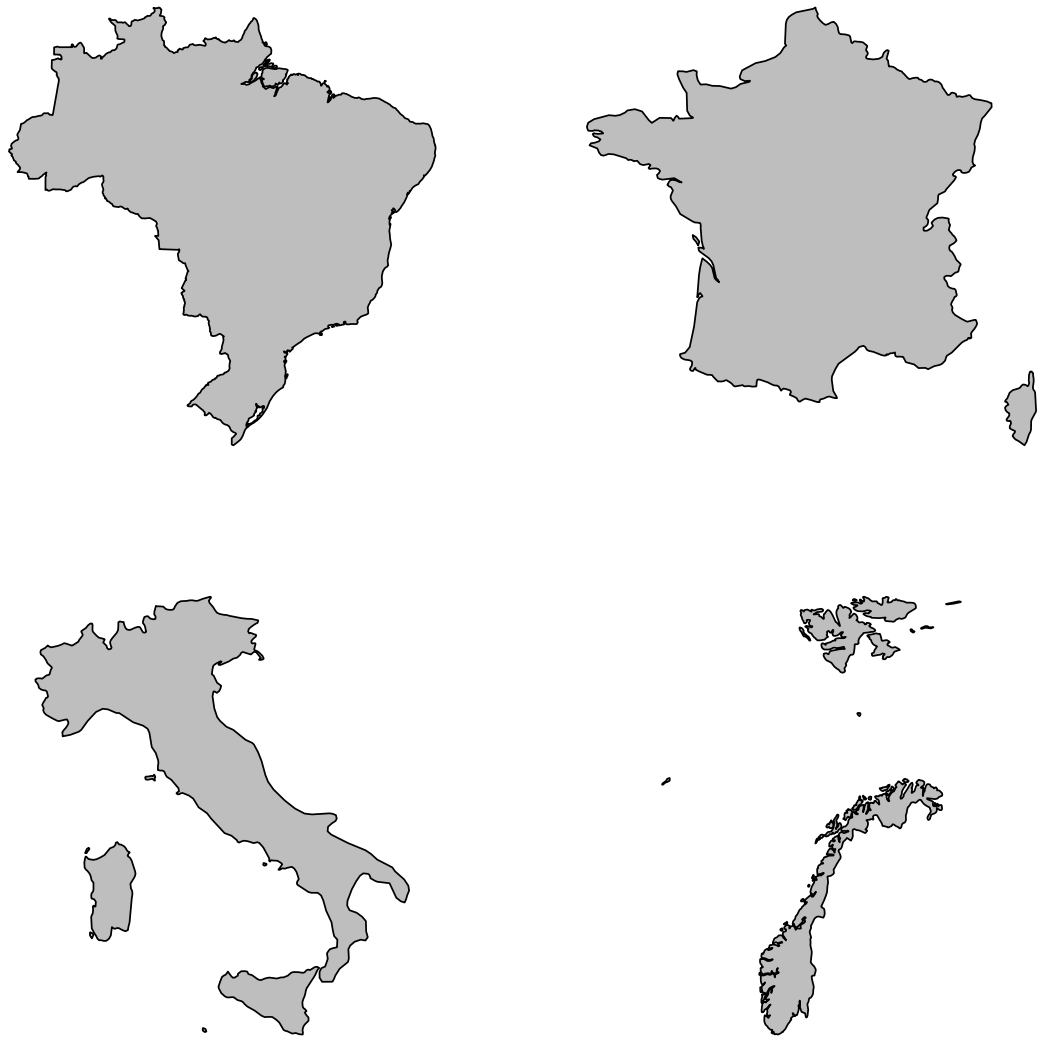
The simplest way to produce a map in R is with the maps package, because maps provides several of its own sets of map information. The maps package provides the function map() for drawing maps. The first argument to this function specifies the name of a "database" that contains the map information. By default, all polygons in a given database are drawn, but the region argument can be used to specify (by name) just a subset of polygons, and the xlim and ylim arguments may also be used to limit the drawing to just a particular range of longitude and latitude.

```
#install.packages('maps')
library(maps)
```

```r
#help(package='maps')
#1
map("world",fill=TRUE,col=rainbow(250),ylim=c(-90,90))
title("World map")
```



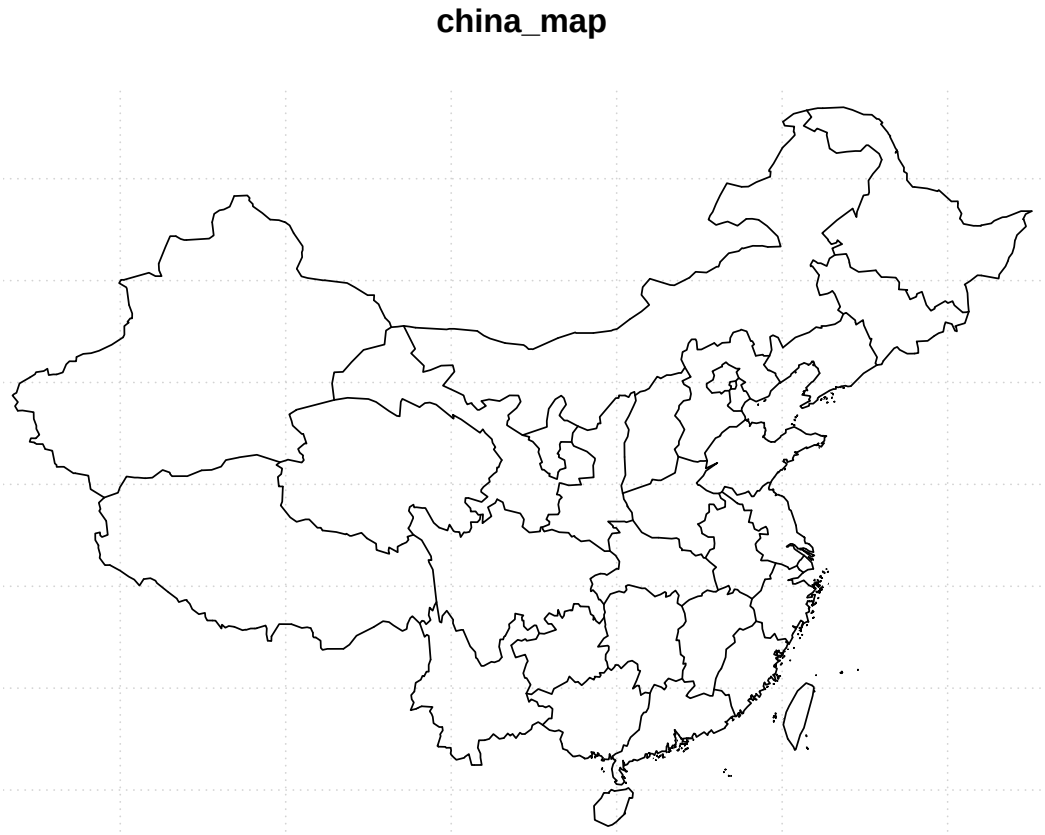

```r
#2
opar <- par(no.readonly = TRUE)
par(mfrow = c(2, 2))
par(mar = rep(1, 4))
map(regions="Brazil", fill=TRUE, col="gray")
map(regions="france", fill=TRUE, col="gray")
map(regions="italy", fill=TRUE, col="gray")
map(regions="Norway", fill=TRUE, col="gray")
```



```
par(opar)
```

Since there is no China data in “maps” package, we use package “mapdata” instead.

```
#install.packages('mapdata')
library(mapdata)

## Warning:  package 'mapdata' was built under R version 3.3.3

map("china",col="black",ylim=c(18,54),panel.first=grid())
title("china_map")
```



3.2 Shapefiles

A very common format for storing geographical information is the shapefile format. More accurately, such files are called ESRI shapefiles because the format is controlled by the ESRI company that produces GIS software. A shapefile is actually a collection of files; the geographical information is stored in several files with a common name stem and different suffixes (e.g., .shp, .shx, and .dbx).

The maptools package provides several functions to read shapefiles, the most general of which is the readShapePoly() function. The maptools package depends on the sp package, which is automatically loaded when maptools is loaded.

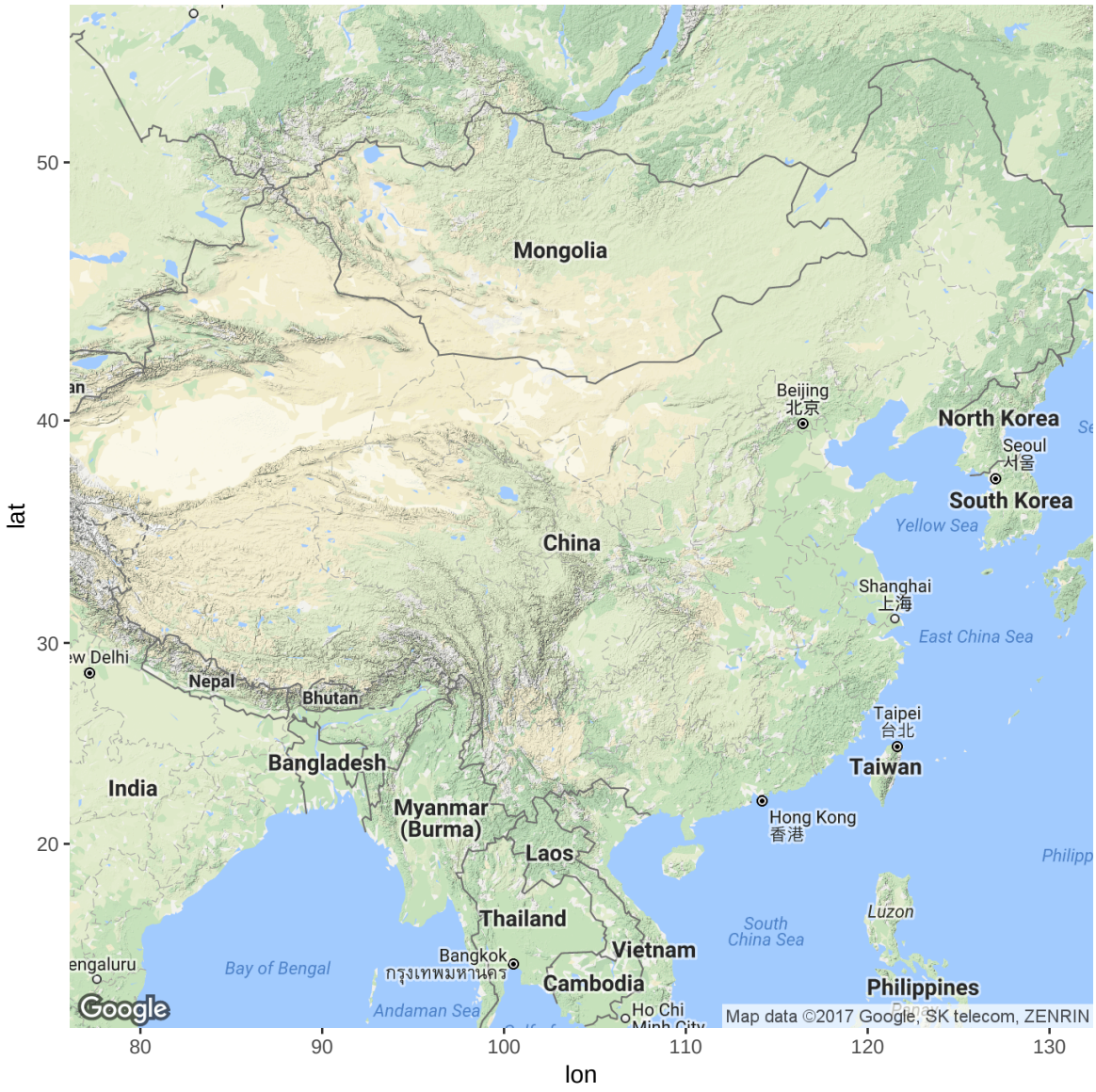
Say that you want to find a map of the World. Using a search engine such as Google or Yahoo!, search for "world shapefile". Download files from the following site: http://www.naturalearthdata.com/downloads/110m-cultural-vectors/ .

```r
#install.packages('maptools')
library(maptools)
map.2<-readShapePoly(file.choose())
#View(map.2)
summary(map.2)
#1
sp::plot(map.2,col=rainbow(200),ylim = c(18, 54), panel.first = grid(),axes=TRUE, border="gray")
#2
#install.packages('ggplot2')
library(ggplot2)
map.3<-fortify(map.2) #convert to data.frame
ggplot(map.3, aes(x = long, y = lat, group = group))+
              geom_polygon( )+
              geom_path(colour = "grey40")+
              scale_fill_manual(values=colours(),guide=FALSE)
```

3.3 ggmap

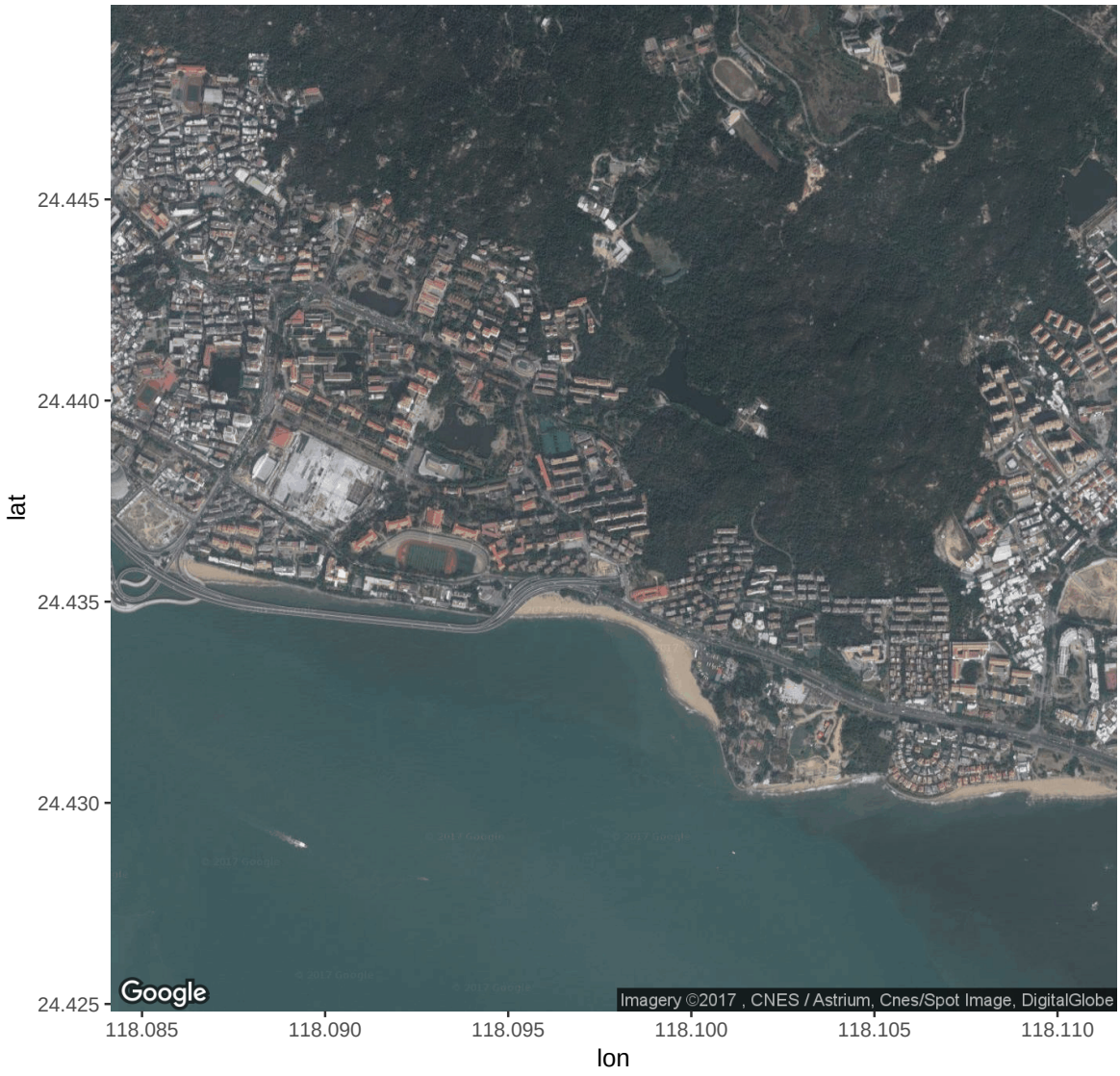
This package allow us to download data from google map.

```r
###NEED VPN
#install.packages('ggmap')
#install.packages('mapproj')
library(ggmap)
#library(mapproj)
## Download data from Google (need VPN)
map <- get_map(location = 'China', zoom = 4)

## Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=China&zoom=4&size=640x640&scal
## Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=China&sensor=false

ggmap(map)
```




```
#Beijing traffic map (from google)
map <- get_map(location = 'Xiamen', zoom = 13, maptype = 'roadmap')

## Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=Xiamen&zoom=13&size=640x640&sc
## Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=Xiamen&sensor=fals

ggmap(map)
```


```
#Satellite
map <- get_map(location = 'Xiamen University of China', zoom = 15,
               maptype = 'satellite')

## Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=Xiamen+University+of+China&zoo
## Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=Xiamen%20Universit

ggmap(map)
```



3.4 googleVis

```
#install.packages("googleVis")
library(googleVis)
G1 <- gvisGeoMap(Exports, locationvar='Country', numvar='Profit',
options=list(dataMode="regions"))
plot(G1)
```

4 Dynamic graphics

A dynamic plot is one which changes over time, such as a plot of a stock index that is constantly updated. The difference between a static plot and a dynamic plot is the difference between a photograph

and a video. The simplest approach to producing a dynamic plot with R is to generate a sequence of plots and show them rapidly one after the other. In simple cases, this can be achieved with a normal graphics window. For example, the following code produces a dot traveling around the circumference of a circle. On some platforms, the display may flicker badly as each new frame is drawn.

```
n <- 40
t <- seq(0, 2*pi, length=n)
x <- cos(t)
y <- sin(t)
for (i in 1:n) {
    plot.new()
    plot.window(c(-1, 1), c(-1, 1))
    lines(x, y)
    points(x[i], y[i], pch=16, cex=2)
    Sys.sleep(.05)
}
```

The animation package

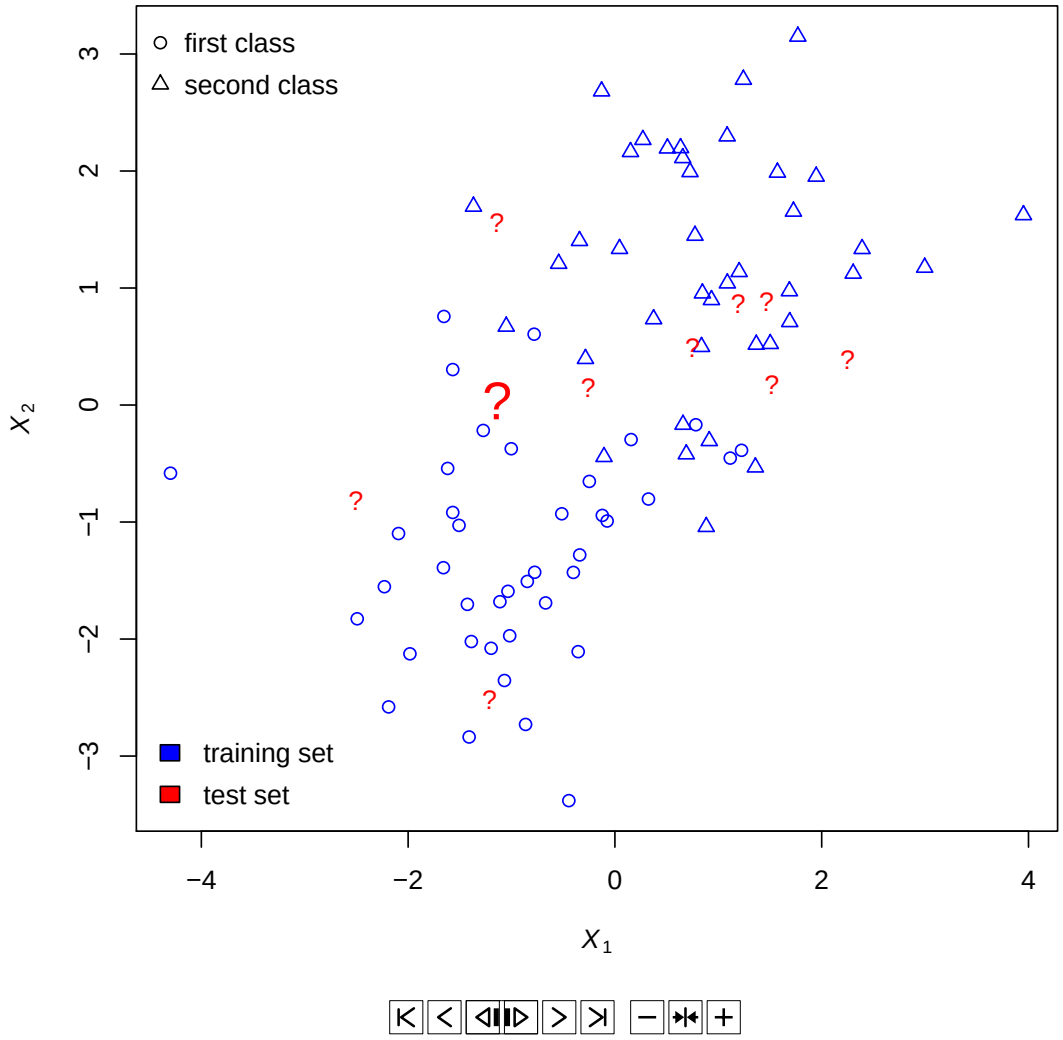
The animation package can produce animation files in a variety of formats, for example, an animated GIF, an Adobe Flash animation, or a web page with the animation embedded in it.

```
install.package("animation")
library(animation)
n <- 40
t <- seq(0, 2*pi, length=n)
x <- cos(t)
y <- sin(t)
orbit <- function() {
    par(pty="s", mar=rep(1, 4))
   for (i in 1:n) {
        plot.new()
        plot.window(c(-1, 1), c(-1, 1))
        lines(x, y)
        points(x[i], y[i], pch=16, cex=2)
    }
}
saveGIF(orbit(),interval=0.05,movie.name="orbit.gif")
```

Another Example (KNN)

```
library(animation)
#oopt = ani.options(interval = 2, nmax = ifelse(interactive(), 10, 2))
x = matrix(c(rnorm(80, mean = -1), rnorm(80, mean = 1)), ncol = 2, byrow = TRUE)
y = matrix(rnorm(20, mean = 0, sd = 1.2), ncol = 2)
saveLatex({knn.ani(train = x, test = y, cl = rep(c("first class", "second class"),
                                        each = 40), k = 30)
}, img.name = "KNN", ani.opts = "controls,loop,width=0.95\\textwidth",
```

```
 latex.filename = ifelse(interactive(), "KNN.tex", ""),
  interval = 1.5, nmax = 35, ani.dev = "pdf", ani.type = "pdf", ani.width = 7,
 ani.height = 7, documentclass = paste("\\documentclass{article}",
                                       "\\usepackage[papersize={7in,7in},margin=0.3in]{geometry}",
                                     sep = "\n"))

## \documentclass{article}
## \usepackage[papersize={7in,7in},margin=0.3in]{geometry}
##                  \usepackage{animate}
##                  \begin{document}
##                  \begin{figure}
##                  \begin{center}
##                  \animategraphics[controls,loop,width=0.95\textwidth]{0.666666666666667}{KNN}{0}{34
##                  \end{center}
##                  \end{figure}
##                  \end{document}
##

#ani.options(oopt)
```



5 Interactive graphics and Graphics GUIs

An interactive plot is dynamic in the sense that it can change rapidly, but the changes occur as a result of user input. The difference between a dynamic plot and an interactive plot is the difference between a video and a video game.

The iplots package

This package provides functions for creating plots that are similar to the standard plot types, but which respond to user interaction and are automatically linked. The iplots package is built on top of the rJava package so Java must be installed for this package to work.

```
install.packages("iplots")
library(iplots)
iplot(mtcars$disp, mtcars$mpg)
ibar(mtcars$gear)
iplot.set(1)
labels <- mapply("itext",
                 mtcars$disp, mtcars$mpg, rownames(mtcars),
                 MoreArgs=list(visible=FALSE), SIMPLIFY=FALSE)
olds <- NULL
while (!is.null(ievent.wait())) {
        if (iset.sel.changed()) {
                s <- iset.selected()
                if (length(s) > 1) {
                        lapply(labels[s], iobj.opt, visible = TRUE)
                }
                if (length(olds) > 1) {
                        lapply(labels[olds], iobj.opt, visible = FALSE)
                }
                olds <- s
       }
}
```

Graphics GUIs

This section is similar to the previous section on interactive graphics, because it also deals with graphics that change in response to user interaction.

```
#1
install.packages("Rcmdr")
library(Rcmdr)
#2
install.packages("playwith")
library(playwith)
playwith(xyplot(mpg ~ disp, mtcars))
playwith(xyplot(qsec ~ wt, mtcars),
          new=TRUE, link.to=playDevCur())
#3
install.packages("pmg")
library(pmg)
```

Exercise