# Unit 13: Graphics

December 2, 2020

This unit first discusses some general concepts and principles of graphics relevant more broadly than R, and then provides implementation details for graphics in R. Note that there is a bunch of example code in the demo code file that is not shown here in these notes.

References:

- Adler
- Chambers
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, R Graphics (available electronically through OskiCat: http://uclibs.org/PID/110697)
- R intro manual (R-intro) on CRAN
- There is a nice overview on creating good graphics in R at http://teachpress.environmentalinformatics-marburg.de/2013/07/creating-publication-quality-graphs-in-r-7

R has several general graphics packages: *graphics*, *grid*, *lattice*, and *ggplot2*. *graphics* is the original graphics package. *grid* is more object-oriented, and is quite powerful and useful if you're involved in serious graphics programming. *lattice* and *ggplot2* are more recent packages that use *grid* to provide the user with high-level graphics capabilities. More details on *grid* can be found in Murrell's *R Graphics* and more details on *ggplot2* can be found in Hadley Wickham's *ggplot2: Elegant Graphics for Data Analysis.*

## 1 Good practices for graphics

There are a number of principles that can be used in developing and critiquing graphics. But first let's see some examples that show those principles being violated.

## 1.1 Some example graphics

1. The file *shell.pdf* in the repository has an example of a crazy pie chart from an advertisement in the NY Times from December 2014.

2. This NY Times article presents time-series graphics about how many Olympic medals have been won by different countries. What do you like or not like about the graphical approach?

3. The article *gordonFinch2015.pdf* presents several variations on a scatterplot of demographic data that illustrate how different presentations can focus attention and allow for interpretation of different aspects of a dataset. Consider Figures 1, 2, and 3 in the pdf. For each figure, what relationships are easy to see and what relationships are hard to see?

4. In Figure 1, you can see a graph that we'll discuss in class. What aspects of the graph could be improved? What aspects do you like?

5. The article *wainer1984.pdf* in the repository is rather old but the ideas are still relevant and while the example figures are dated in terms of appearance, the same issues arise with more modern-looking graphics.

6. The article *gelmanUnwin2013.pdf* in the repository presents a modern-day reinterpretation of a famous graphic from Florence Nightingale regarding causes of death in the British Army during the Crimean War in the 1850s.

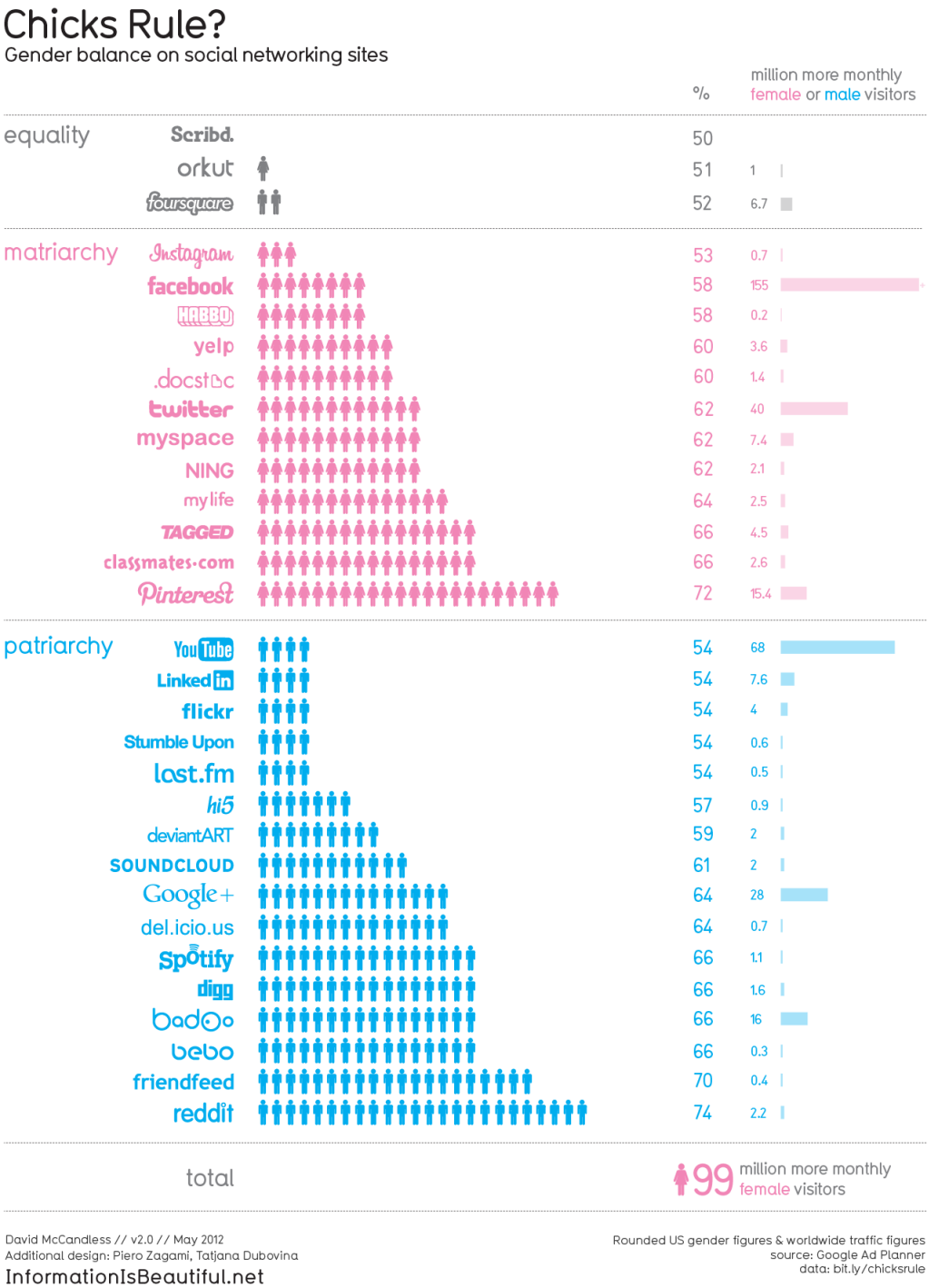



*Figure 1. Example graphic indicated in Section 1.1, item #4.*

## 1.2 Best practices

Here's a list of some guidelines to consider in creating graphics.

1. Have a high density of information to space
2. Show the data clearly: are the relationships and patterns that are clearly seen in the graph the ones you want to represent?
3. Can you reorder groups or variables to better illustrate the key points?
4. Strategies for going beyond two dimensions
   (a) Use color, but avoid if unnecessary (saves printing costs in journal articles)
   (b) Use varying symbol or line types
   (c) Use multiple panel plots
5. Avoid 3-d graphics unless they truly add information
6. Avoid stacked barplots (see demo code) as it's hard to assess anything except the total and the baseline category.
7. Studies indicate that humans have a hard time comparing areas, volumes, or angles, so try to avoid plots that represent data using any of these, including pie charts. Instead use position or length (horizontal is better than vertical) to display data values
8. Label axes and include units.
9. Keep the ranges of axes (and other features) the same for multiple panels, when possible
10. Use a legend where appropriate
11. Use vector graphics formats such as PDF or Postscript/EPS as these scale without pixelation when resized. Raster formats such as JPEG, PNG, BMP, TIFF don't rescale well and when they have high resolution also have large file sizes. See Section 6.

Rob Hyndman has a list of 20 rules for good graphics, including some of the ones above. There is also a list of guidelines in the *gordonFinch2015.pdf* article in the repository.

# 2 Base R graphics (the *graphics* package)

The material here mainly gives high-level information and information on adjusting your graphics, rather than telling you how to make particular plots using R's base graphics. The *graphicsCommands.pdf* file in the github repository and Chapter 14 of Adler provide information about many of the core plotting functions. These include high-level functions such as *plot()*, *matplot()*, *pairs()*, *coplot()*, *hist()*, *density()*, and *boxplot()*, as well as low-level functions for adding information to a plot such as *lines()*, *points()*, *abline()*. Adler goes into great detail on many of these, both in the context of the *graphics* package and that of the *lattice* package.

## 2.1 Background

The basic components of R base graphics are

1. high-level functions (e.g., *plot()*, *boxplot()*, etc.) for producing an entire plot
2. low-level functions for adding components to existing plots. The ability to build up a complicated plot piecewise is one of the strengths of R's graphics. These low-level functions include *abline()*, *arrows()*, *axis()*, *legend()*, *lines()*, *points()*, *rug()*, *text()*, *mtext(), title()*, *symbols()*.
3. graphics parameters controlled through (1) *par()* or (2) as arguments to a graphics function call. These parameters change the appearance of a plot or plots (e.g., *mai*, *pch*, *col*)

R graphics work by sequentially plotting graphics elements, so subsequent items may paint over initial elements. One technique when this is an issue is making your colors transparent - see the color subsection below.

Note that as we saw when talking about OOP, the core *plot()* function does both scatterplots and is a generic method used for plotting a wide variety of objects, e.g., *plot.lm()*.

I won't say much about the high-level and low-level functions, but a wide range of these exist. In particular low-level functions allow you to add almost anything you might want to add: arbitrary lines, polygons, symbols, text, arrows, boundary lines in maps, etc.

## 2.2 Graphics parameters

We can set graphics parameters through various plotting functions (for temporary changes applying only to the current plotting command) or through the *par()* function (for a permanent change), allowing us to customize the layout or appearance of a plot. Let's take a look at some of these and what they mean. The demo code shows some additional more complete code examples.

Most of you know how to create a multi-panel plot:

```
par(mfrow=c(4, 2)) # 4 rows, 2 columns of subplots
```

Changing the margin sizes (and axis information spacing) is one of the most common modifications one needs to do, particularly when producing a multi-panel plot and when creating an output file. One often wants to reduce the size of the inner margins. This helps maximize the information to white space and increase the resolution of your plot. Sometimes one needs extra space in the outer margin of a multi-panel plot. Positioning and lengths within the graphics window can occur with relative units; i.e., the device domain is $(0, 1) \times (0, 1)$, or in physical units (commonly inches), or in "lines" of text. Some values I commonly use are

```
par(mai = c(.5, .5, .1, .1)) # manipulate inner margins of subplots
par(mgp = c(1.8, .7, 0)) # manipulate spacing of axis ticks, labels, text
par(omi = c(0, 0, .3, 0)) # manipulate outer margin of full plot
```

Here for the margins, the first number is the bottom, the next the left, the third the top, and the fourth the right margin, so with *omi*, I've made some space (0.3 inches) in the outer margin at the top.

Note that if you change graphics options using *par()* (including within a function call!), the values change permanently (so it's like pass by reference). One way to be able to go back is

```
oldpar = par(no.readonly = TRUE)
par(cex = 3); plot(x, y)
par(oldpar); plot(x, y)
```

So if you create a function that changes the graphics parameters, you should use the above strategy to reset so you don't surprise your users.

Here are the various layout parameters and their units: *din* (device size, inches), *pin* (plot size, inches), *fin* (figure size, inches), *mai* (margin size, inches) or *mar* (margin size, lines), *omi* (outer margin, inches) or *oma* (outer margin, lines), *mex* (# text lines per interline spacing), and *plt* (plot region as fraction of figure region). *cex* controls size of points and text in general when called through *par()*, but only controls the size of points when called within a plotting function, while *cex.lab*, *cex.axis*, and *cex.main* control the size of axis labels, axis values, and the title. Similarly for *col*, *col.lab*, *col.axis* and *col.main*.

Here are some other things you can control: text justification (*adj*), font size (*cex*, *csi*), font type (*font*), rotation of text (*srt*), color (*col*), line type (*lty*), line width (*lwd*), plotting character symbol (*pch*), type/presence of boundary box (*bty*), log-scale axes (*log*), axis labels (*{x,y}lab*),

axis limits (*{x,y}lim*), a variety of details about the axis limits, labels, and ticks (*lab*, *las*, *tck*, *{x,y}axp*, *{x,y}axs*, *{x,y}axt*), and whether to plot axes (*axes*).

**Some additional tidbits** You can force subplots to have the same axis ranges by manipulating *xlim* and *ylim*.

It can be handy to put axis labels in the outer margin of a multi-panel plot:

```
x = rnorm(10); y = rnorm(10); par(mfrow = c(2, 2))
for(i in 1:4) plot(x, y, xlab = '', ylab = '')
mtext("my x variable", 1, line = -1, outer = TRUE)
mtext("my y variable", 2, line = -1, outer = TRUE)
```

Note if we wanted to put the label further towards the edge of the plot, we'd need to use *omi* or *oma* to create an outer margin, and then we could use a value of *line* that is greater than -1 to put the text further away from the plotting regions.

You can create multi-line text by just using "\n" in your character string.

To plot outside the plot region, set `xpd = TRUE` - otherwise anything outside the region is "clipped".

## 2.3 Adding information to a plot sequentially

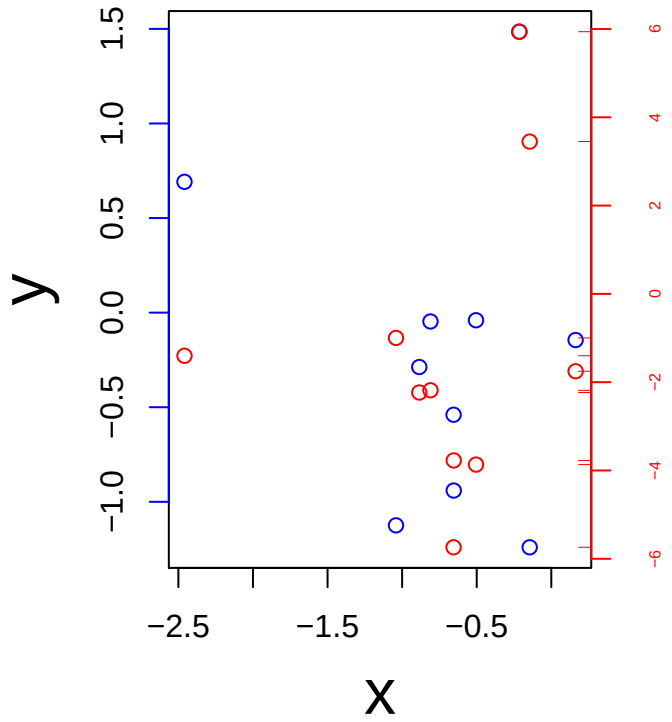
There are lots of low-level functions you can use to add components to a plot. These include *abline()*, *arrows()*, *axis()*, *legend()*, *lines()*, *points()*, *rug()*, *text()*, *mtext()*, *title()*, *symbols()*, *hline()*, *vline()*. A basic strategy for customizing components of a plot is to use the high-level function to plot the basics, specifying parameter arguments that leave out some components (e.g., `xaxt = 'n'`, `bty = 'n'`, `xlab = ''`). At its most extreme, you can create the structure of a plot without plotting any data for customizing everything:

```
plot(x, y, type = "n") # plot with nothing in it
```

With *title()*, you can add main and subtitles and axis labels, as we saw in the demo code for the previous subsection. This gives you control over positioning using the *line* and *outer* arguments. *rug()* will add tick marks for each observation along a chosen axis. *locator()* allows you to choose a location for certain items (e.g., a legend) with your mouse.

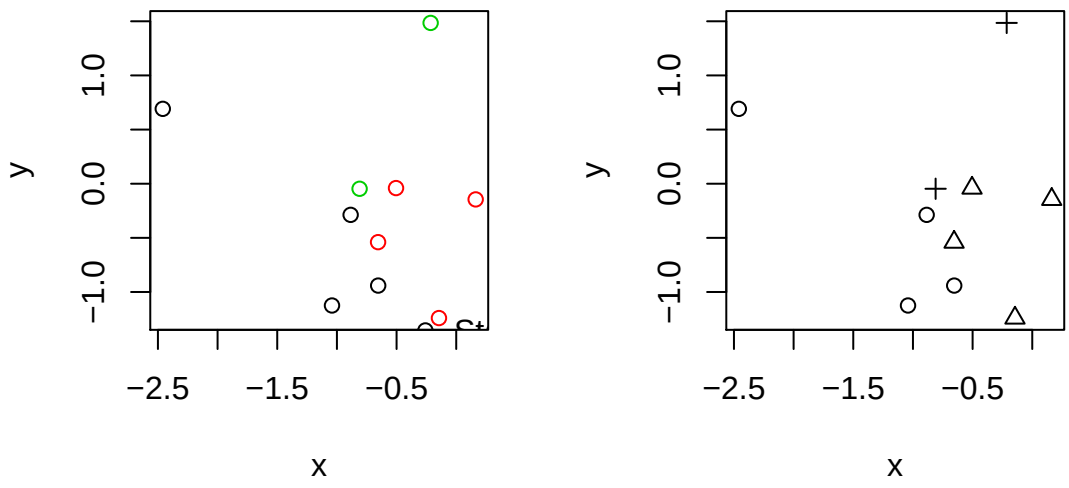
You an create custom axes with the *axis()* function, adding the axis, ticks, and axis value labels. Here's an example of plotting two sets of data with different axis values. It also shows the use of the new argument to overplot on an existing plot. Often one would initially exclude (some) axes and a boundary box from the plot:

```r
x = rnorm(10); y = rnorm(10); z = rnorm(10, 0, 5)
par(mfrow = c(1, 1), mai = c(1, 1, .1, .8))
# plot y points in blue
plot(x, y, yaxt = "n", bty = "n", col = "blue", cex.lab = 2)
box() # add box in black
axis(side = 2, col = 'blue')  # add left-side axis in blue
# this causes next call to plot to overplot instead of
#   replacing the original plot:
par(new = TRUE)
# plot z points in red
plot(x, z, col = 'red', bty = 'n', ann = FALSE, xaxt = 'n', yaxt = 'n')
# add right-side axis in red
axis(side = 4, col = 'red', cex.axis = .5, col.axis = "red")
rug(z, side = 4, col = "red") # add a rug
```




**Adding an additional dimension** Let's consider other ways to pack information on another dimension into a plot:

```r
par(mfrow = c(1, 2))
groups = sample(1:3, 10, replace = TRUE)
plot(x, y, col = groups)
legend(-.5, -1, legend = c("Stats", "PoliSci", "Math"),
       col = 1:3, pch = 1, bty = "n")
# change the first two arguments (the x,y coords of the upper left
#   of the legend) to position the legend in a different place

# alternatively you can position the legend manually
plot(x, y, pch = groups)
```




```r
# right-click to position legend manually (commented out for pdf compilatio
#legend(locator(), legend = c("Stats", "PoliSci", "Math"), pch = 1:3)
```

You can also use this approach with different line widths (*lwd*) and line styles/types (*lty*) when that is relevant.

## 2.4 Interacting with graphics

R is not well set up for interactive work, but one handy function is *identify()*, which is handy for both identifying points (such as outliers) and marking a specific point.

```r
plot(x, y)
identify(x, y, plot = FALSE)
# now left-click on individual points and
# R will tell you the index of the point
# right-click to exit the interactive mode and see the id's
# of the points you clicked
identify(x, y, labels = "Critical point")
```

If you want to do more with interactive graphics (which can be very helpful for high dimensional data), check out the *GGobi* software.

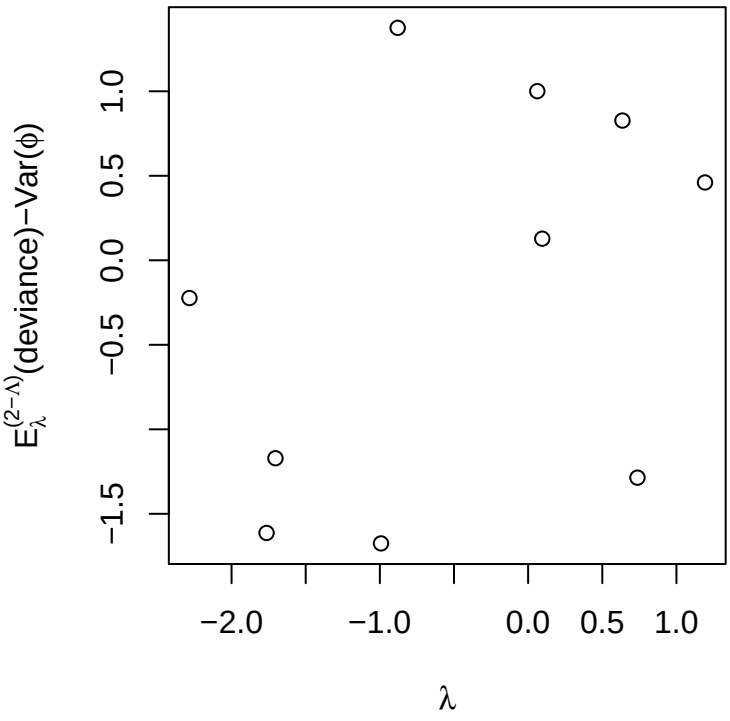
## 2.5 Using mathematical expressions in plots

### 2.5.1 latex2exp

The *latex2exp* package seems to make it much easier to work with than the *plotmath* syntax described in the next subsection.

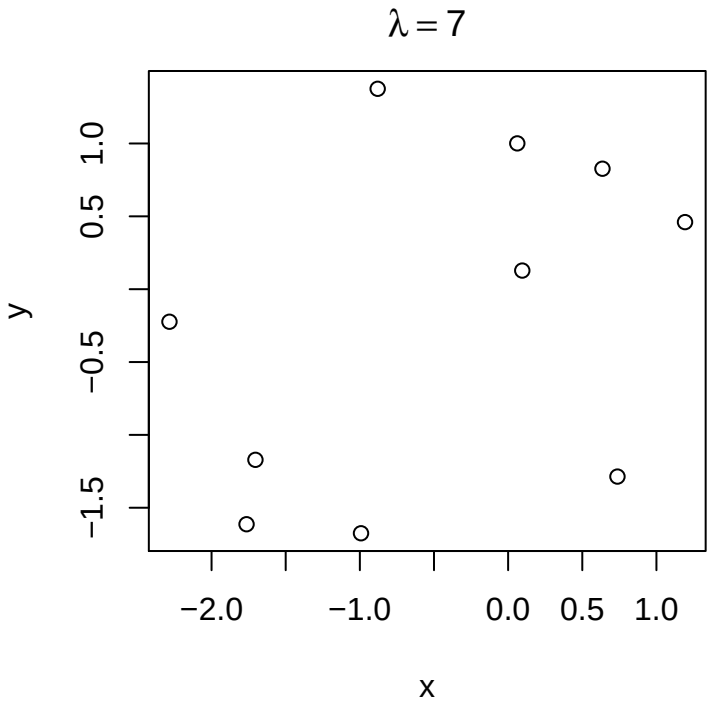
The *TEX* function takes a LaTeX expression and converts it to an object that can be used in place of a string or plotmath syntax in plotting contexts. Note that you need to escape the usual backslashes in LaTeX syntax, so you'll end up with double backslashes. And for math notation, you need the LaTeX syntax within a math environment.

```r
library(latex2exp)
par(mai = c(1, 1, .1, .1))
x = rnorm(10); y = rnorm(10)
plot(x, y, xlab = TeX("$\\lambda$"),
ylab = TeX("$E_{\\lambda}^{(2-\\Lambda)}(deviance)-Var(\\phi)$"))
```




If you want to combine *latex2exp* with evaluation of R code, try the following

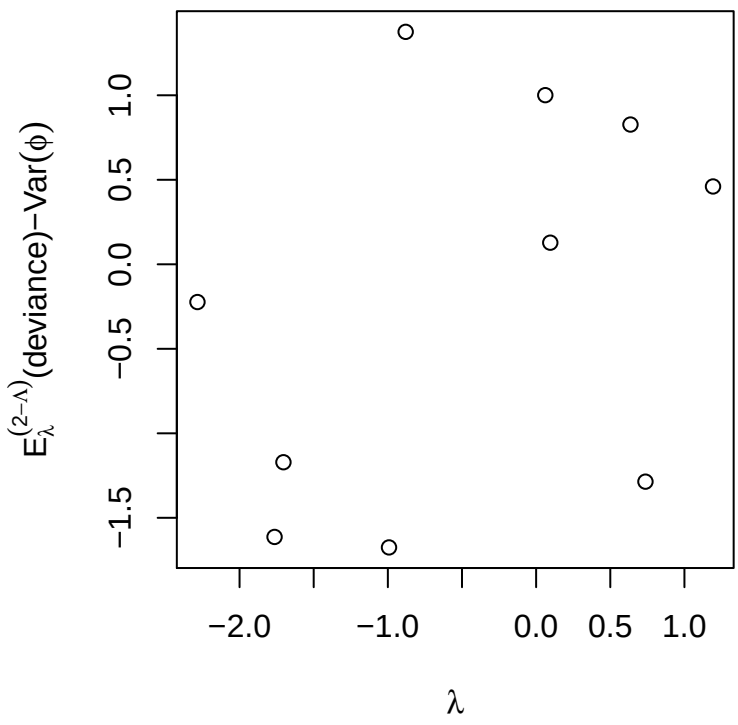
```
library(latex2exp)
lambdaVal <- 7
par(mai = c(1, 1, .5, .1))
plot(x, y, main = TeX(paste0("$\\lambda = $", lambdaVal)))
```




### 2.5.2 plotmath

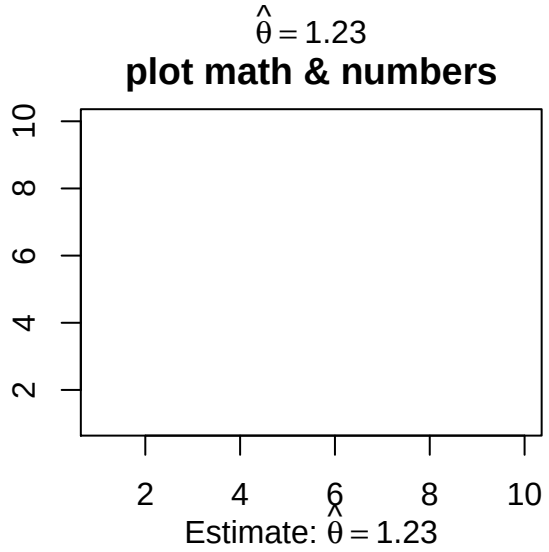
We can use Greek letters and mathematical symbols in labels using *expression()* to enclose the mathematical expression and *paste()* to combine elements.

```
par(mai = c(1, 1, .1, .1))
plot(x, y, xlab = expression(lambda),
     ylab = expression(paste(E[lambda]^(2-Lambda),
                             "(deviance)-",Var(phi))))
```




Here's an example of using a variable within an expression. This is quite powerful if you are producing multiple plots and the text changes for each plot:

```
par(mai = c(.7, .5, .4, .1), omi = c(0,0,.2,0))
plot(1:10, type="n", xlab="", ylab="", main = "plot math & numbers")
thetaVal <- 1.23
mtext(bquote(hat(theta) == .(thetaVal)), side = 3, line= 1.5)
                        # put text on top, 1.5 lines out

mtext(substitute(paste("Estimate: ", hat(theta) == thetaVal),
                 list(thetaVal = thetaVal)), side = 1, line= 2)
```




```
            # this puts the text on the bottom, 2 "lines" out
```

*bquote()* treats its argument as an R language object but substitutes in for terms wrapped in "*.()*", behaving somewhat like *substitute()*. An alternative approach uses *substitute()*:

For a full set of mathematical syntax and examples of such substitutions, see `?plotmath` as well as `example(plotmath)` and `demo(plotmath)`.

For plotting text, it's helpful to keep in mind that we can choose a display of text based on the encoding of the characters and a font in which to display the characters. So if you need characters not in the standard Latin character scheme, it should be doable by choosing a different encoding.

## 2.6 Laying out panels (subplots)

We can create more complicated arrangements of panels than the rectangular layout of *mfrow* and *mfcol*.

One approach is use *layout()*. Here we give it a matrix with as many rows and columns as we want in the plot and assign values, 1, 2, ... to the appropriate rows and columns of the matrix, with the values serving to identify each panel.

```
layout(matrix(c(1, 1,
                0, 2),
              nr = 2, byrow = TRUE))
## so this says to create the first subpanel (the 1s)
## as the entire first row (the 1,1 and 1,2 subpanels combined)
## and the second subpanel as the 2,2 subpanel, with nothing
```

```
## in the 2,1 subpanel
layout.show(n = 2) # the numbering goes from 1 to 2

layout(matrix(c(1, 1, 1, 1,
                4, 3, 2, 2), nr = 2, byrow = TRUE))
layout.show(n = 4) # numbering in the values in the matrix goes from 1 to 4
```

Let's see the example from `?layout`, which slickly plots a scatterplot with marginal histograms.

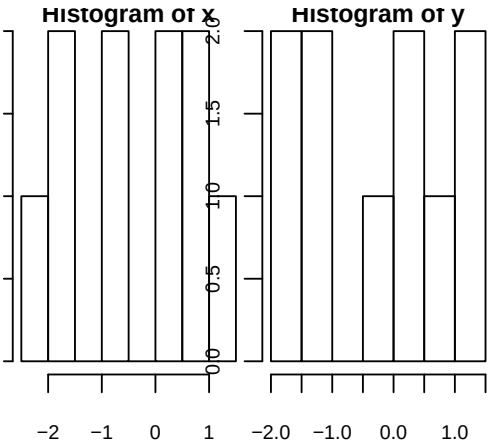
We can use *split.screen()* as an alternative to *layout()*. Here's a basic example (plot not shown) that allows us to choose which panel to plot in:

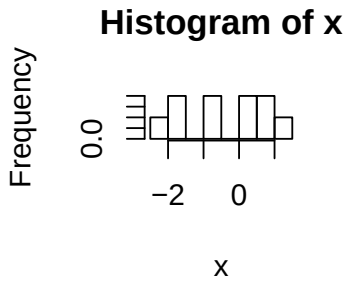
```
split.screen(figs = c(2, 3))  # 2 row by 3 column grid of subplots
screen(3) # plot in the 3rd subplot (position 1,3)
plot(x, y)
hist(x) # whoops - the hist() overwrote the plot() in the 3rd subplot
## this indicate we need to change screens manually
```

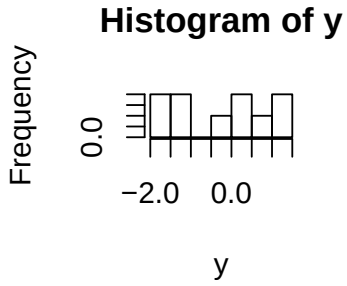
*split.screen()* can be used with a 4-column matrix where each row indicates the xrange and yrange *(xl, xu, yl, yu)* of the given subplot where *xl* and *xu* are the lower and upper x-axis values and *yl* and *yu* for the y-axis values. The values should all be in between 0 and 1. Note that overplotting is possible.

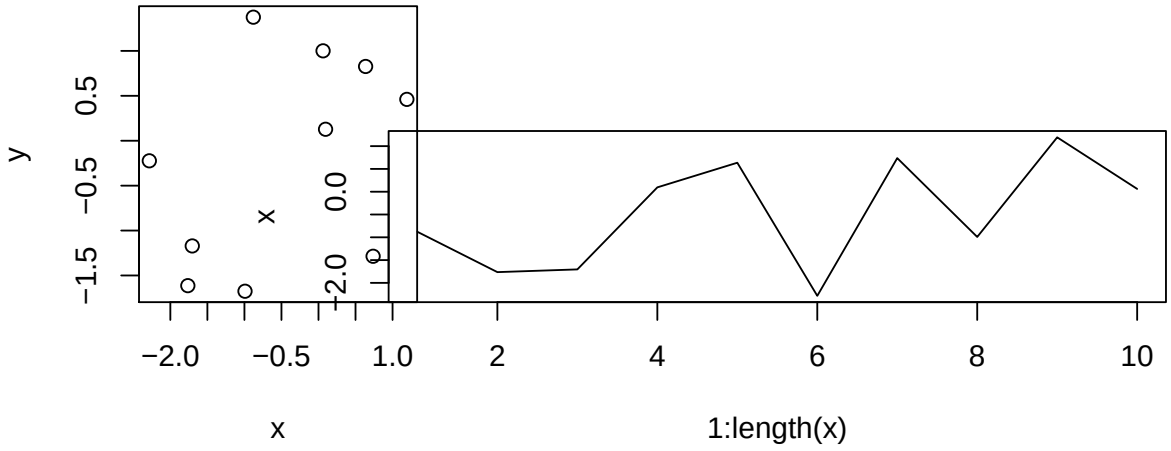
```
split.screen(figs = matrix(c(0, .4, 0, .5,
   # xl, xu, yl, yu values to position the first subplot
                .7, 1, .7, 1, # position for second subplot
                .7, 1, .4, .7, # etc.
                .2, 1, 0, .4,
                0, .4, .7, 1), nc = 4, byrow = TRUE))

## [1] 1 2 3 4 5

screen(1); plot(x, y)
screen(2); hist(x)
screen(3); hist(y)
screen(4); plot(1:length(x), x, type = "l")
# notice the overplotting because panel 4 overlaps panel 1
```

```
# in the x-dimension

## you can continue subdividing, but need to be careful that
## the margins don't get too big for the remaining space
## (and that text is not too large)
screen(5)
split.screen(figs = matrix(c(0, .5, 0, 1,
                .5, 1, 0, 1), nr =2, byrow = TRUE))

## [1] 6 7

screen(6); par(mai = c(.05, .05, .05, .05), cex = .7); hist(x)
screen(7); par(mai = c(.05, .05, .05, .05), cex = .7); hist(y)
```




```
close.screen(all = TRUE)
```

In the *grid* package, positioning subpanels is done via *viewports*.

## 2.7 Plotting in three dimensions and mapping

In general for plotting $z \sim x + y$, either in an actual spatial setting or as a function of two variables, I prefer the *image()* function (*levelplot()* in lattice), which uses color to represent different levels. *image.plot()* from the fields package nicely adds a color bar legend. *contour()* and *persp()* are other options but I personally find them less informative most of the time: *contour()* because it requires

the viewer to read the contour line values and *persp()* because some of the features often hide other features.

```
n = 20; xs = ys = 1:n;
gr = expand.grid(xs, ys)
# Cholesky decomposition for generating correlated normals
U = chol(exp(-rdist(gr)/6))
image.plot(1:n, 1:n, matrix(crossprod(U, rnorm(n^2)), n, n),
           col = tim.colors(32))
```

If you do want to make a 3-d plot of a surface, *persp()* will do it, with x, y, z arguments similar to *image()* and also having arguments for the angle of view (the rotation around the z-axis – *theta*) and the viewing angle (up or down – *phi*). You may be able to choose a set of angles that show the surface with the least amount of hidden material. See the demo code for using *image()*, *contour()*, and *persp()* to show topography in a mapping context.

R has a lot of tools for mapping in a variety of packages – for more examples see the demo code. In particular you can import standard GIS shape files and overlay boundaries on a map using the *spdep* package. R also has state and national boundaries as part of the *maps* package (see the demo code for more examples). If you like *ggplot2* (see the next section), check out Hadley Wickham's *ggmap* package.

```
plot(x, y)
map('state', add = TRUE) # adds state boundaries
```

If you're using color in maps, *RColorBrewer* package is good for choosing colors for unordered levels, sequential ordering, and two-way diverging color ordering (see `?display.brewer.all`) and the *ColorBrewer* website provides recommendations.

# 3 Lattice and ggplot2

Here we'll see some of the main kinds of graphs, contrasting the syntax in *lattice* and *ggplot2*.

The *lattice* package provides an alternative set of graphics functions based on the Trellis graphics system. *ggplot2* is another popular alternative developed by Hadley Wickham. In both packages, the style is prescriptive, which in some cases is good (think LaTeX vs. *Microsoft Word*). Both packages are well set up for dealing with conditioning either through sets of panels or through grouping by color or symbol. In the case of multi-panel plots, the style exploits the inter-relatedness

of the information, with 'strips' giving labelling information, and avoidance of repetition of axis information.

Here are the basic relationships amongst variables that we can represent graphically:

- $y$ # distribution of y
- $y \sim x$ # relationship between x and y, considering y as a function of x
- $y \sim x \mid A$ # relationship between x and y conditional on the values of A
- $y \sim x \mid A * B$ # relationship between x and y conditional on combinations of A and B
- $z \sim x * y$ # 3D relationship between x, y, and z, with z a function of x and y

Let's see the two packages in action. We'll use the *Comparative Political Data Set*, which contains data on some political/economic variables for some of the industrialized countries. In the data file, *vturn* is voter turnout, *realgdpgr* is growth in GDP (the size of the country's economy), *outlays* is government spending as a percentage of GDP, and *unemp* is the unemployment rate. Metadata are available at
http://www.ipw.unibe.ch/content/team/klaus_armingeon/comparative_political_data_sets/index_eng.html.

Both *lattice* and *ggplot2* work by adding layers to a base plot. In both cases, you produce an object with multiple layers and then a *print()* method applied to the object produces the plot.

## 3.1 Comparing *lattice* and *ggplot2* by example

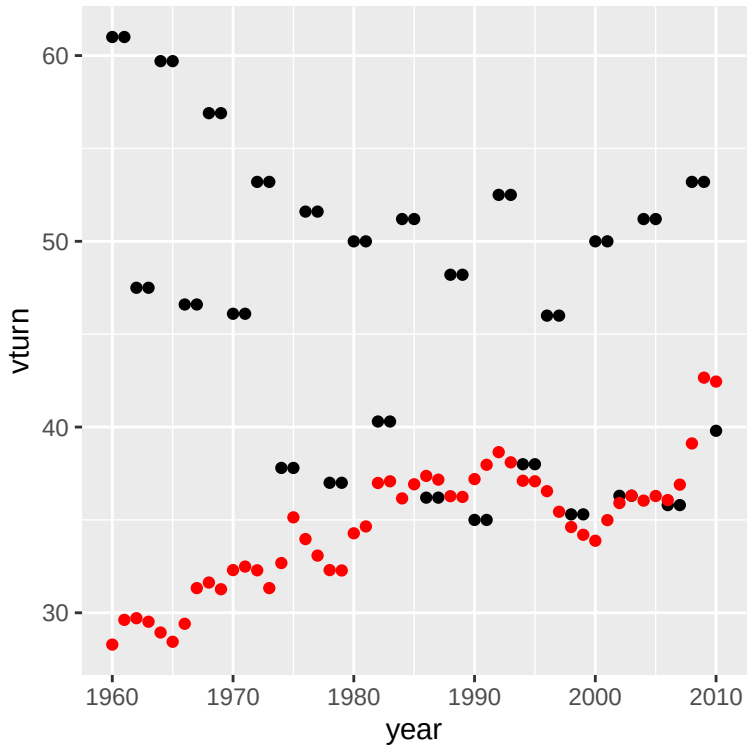
### 3.1.1 Basic intro to *ggplot2*

Here's a schematic of the *ggplot* syntax::

```
ggplot(data= , aes(x= ,y= , [options]))+geom_xxxx()+...+...+...
```

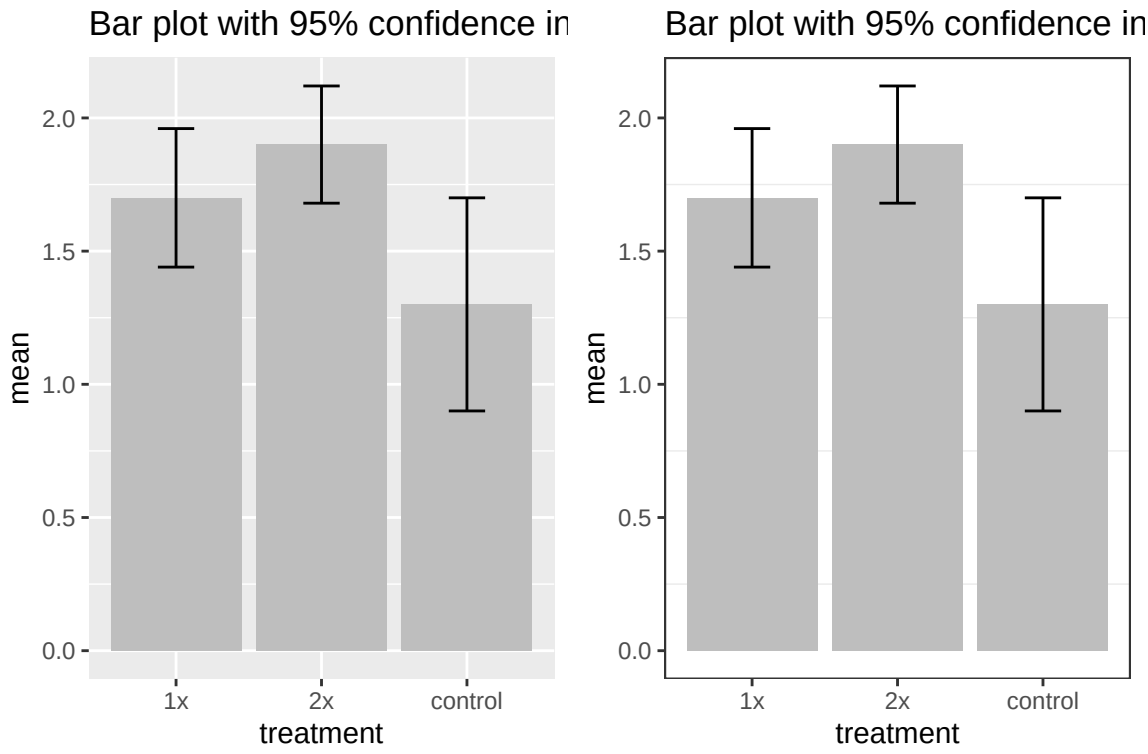
The variables are part of the "aesthetic" argument. You can see the layering effect by comparing the same graph with different colors for each layer

```
library(ggplot2)

cpds <- read.csv('../data/cpds.csv')
usa <- cpds[cpds$country == "USA", ]
ggplot(data = usa, aes(x = year, y = vturn)) +
  geom_point(color = "black") +
  geom_point(data = usa, aes(x = year, y = outlays), color = "red")
```




Here's another example where we layer on a bunch of features and create two versions of the plot as different objects. This also shows how to create a multipanel plot in *ggplot2*.

```
require(gridExtra, quietly=TRUE)

dat <- data.frame(treatment = c('control', '1x', '2x'),
  mean = c(1.3, 1.7, 1.9), se = c(.2, .13, .11))
fig1 <- ggplot(dat, aes(x = treatment, y = mean)) +
  geom_bar(position = position_dodge(), stat="identity", fill="grey") +
  geom_errorbar(aes(ymin=mean-2*se, ymax=mean+2*se), width = 0.25) +
  ggtitle("Bar plot with 95% confidence intervals")  # plot title
fig2 <- fig1 +
  theme_bw() + # remove grey background (because Tufte said so)
  theme(panel.grid.major = element_blank())
  # remove x and y major grid lines (because Tufte said so)
grid.arrange(fig1, fig2, nrow = 1, ncol = 2)
```

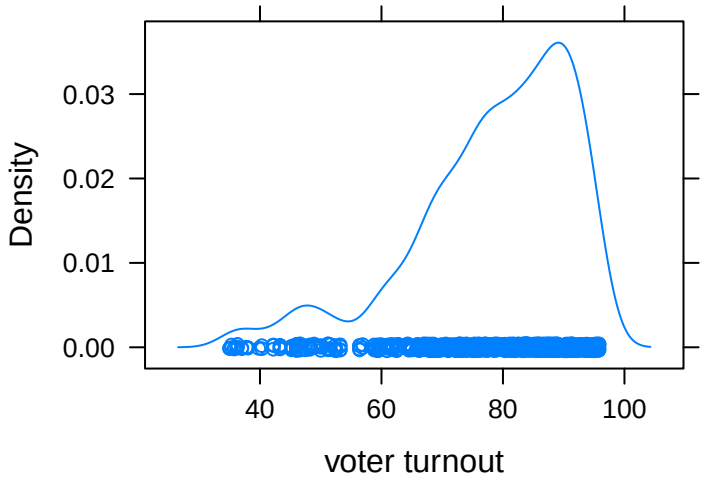



Here's a bit more syntax for modifying labels and axis limits and what-not.

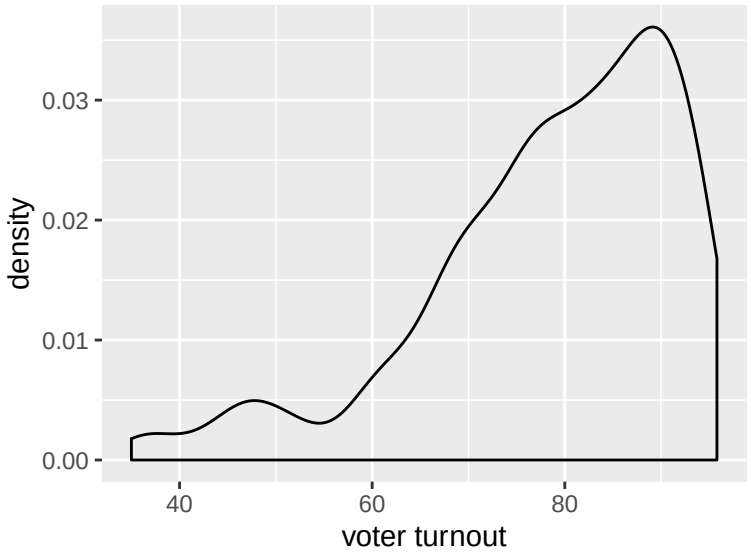
```
ggplot(data = cpds, aes(x = year, y = vturn)) + geom_point() +
  xlab(label = "The X Label") +  ylab(label = "The Y Lab") +
  xlim(1980, 1989) + ggtitle(label = "The Title")
```

### 3.1.2 Density plots

```
densityplot(~vturn, data = cpds, xlab = 'voter turnout')
```



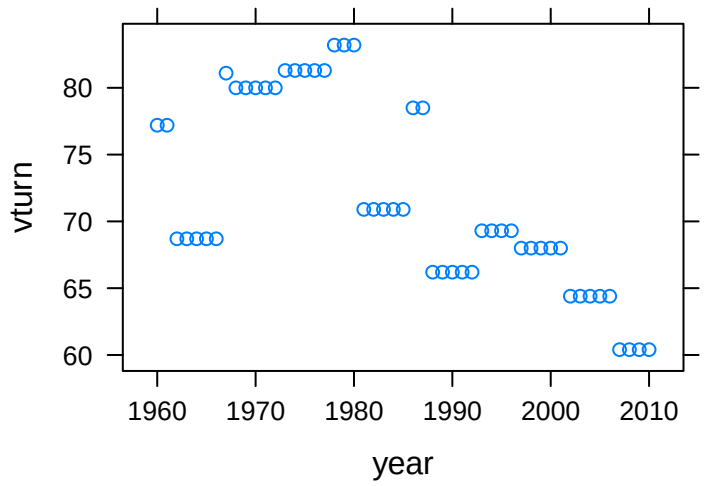

```
ggplot(data = cpds, aes(x = vturn)) + geom_density() +
  xlab(label = "voter turnout")

## Warning:  Removed 39 rows containing non-finite values
## (stat_density).
```



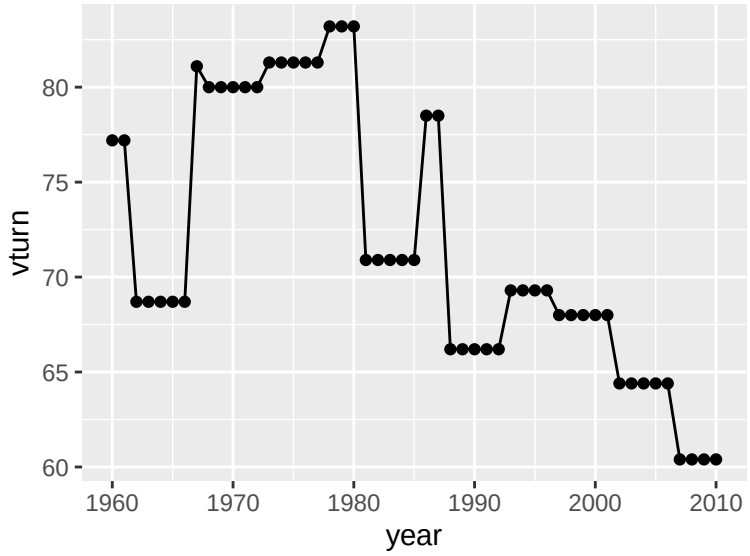

### 3.1.3 Scatterplots

In addition to seeing how to do scatterplots, we also see how to do built-in subsetting here.

```
xyplot(vturn ~ year, data = cpds, subset = country == "France")
```




```
ggplot(data = subset(cpds, subset = country == "France"),
       aes(x = year, y = vturn)) + geom_point() + geom_line()
```



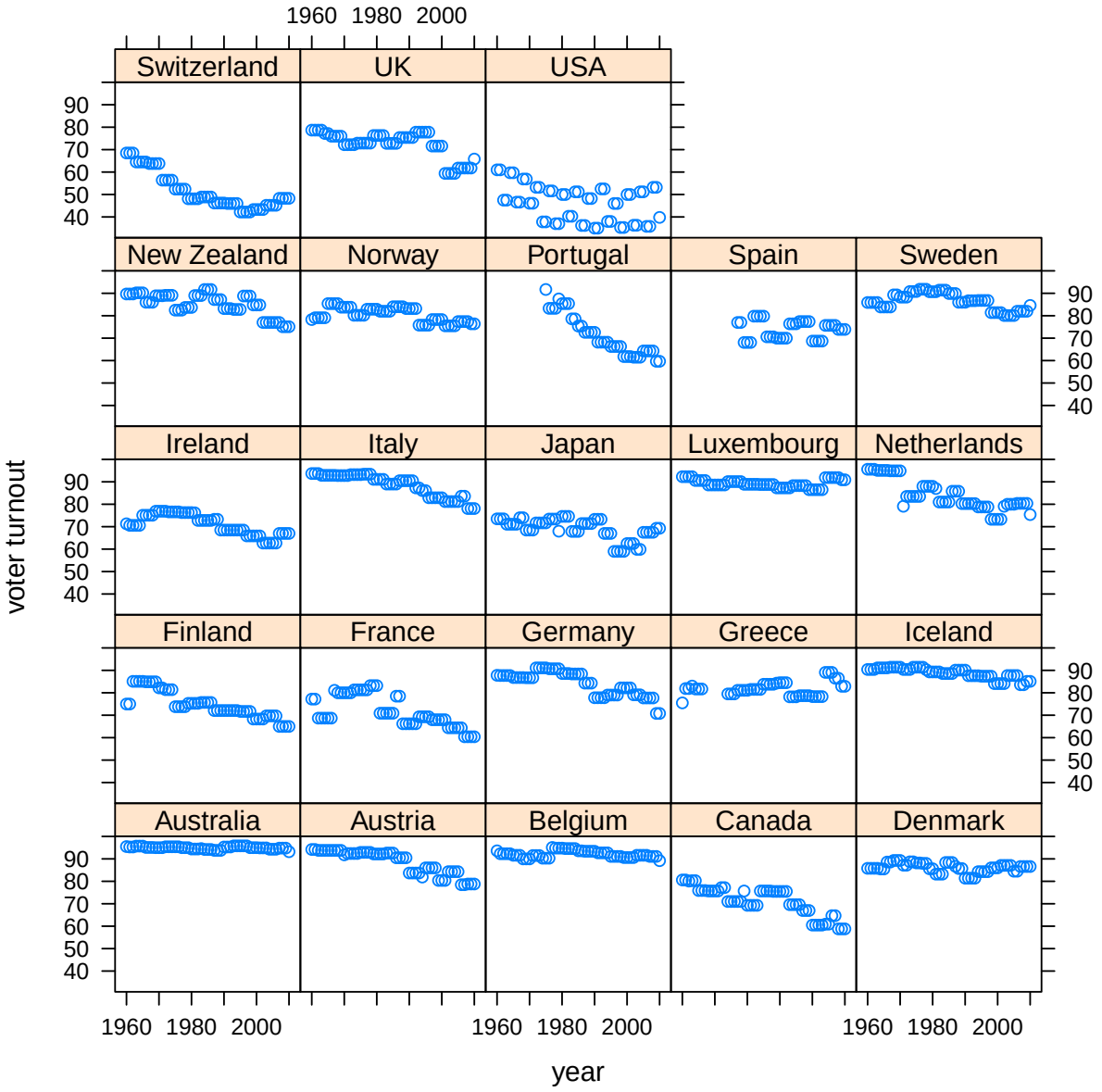

*ggpairs()* in *ggplot2* produces a scatterplot matrix showing the relationships of all pairs of a set of variables that deals with both continuous and discrete variables.

### 3.1.4 Conditioning

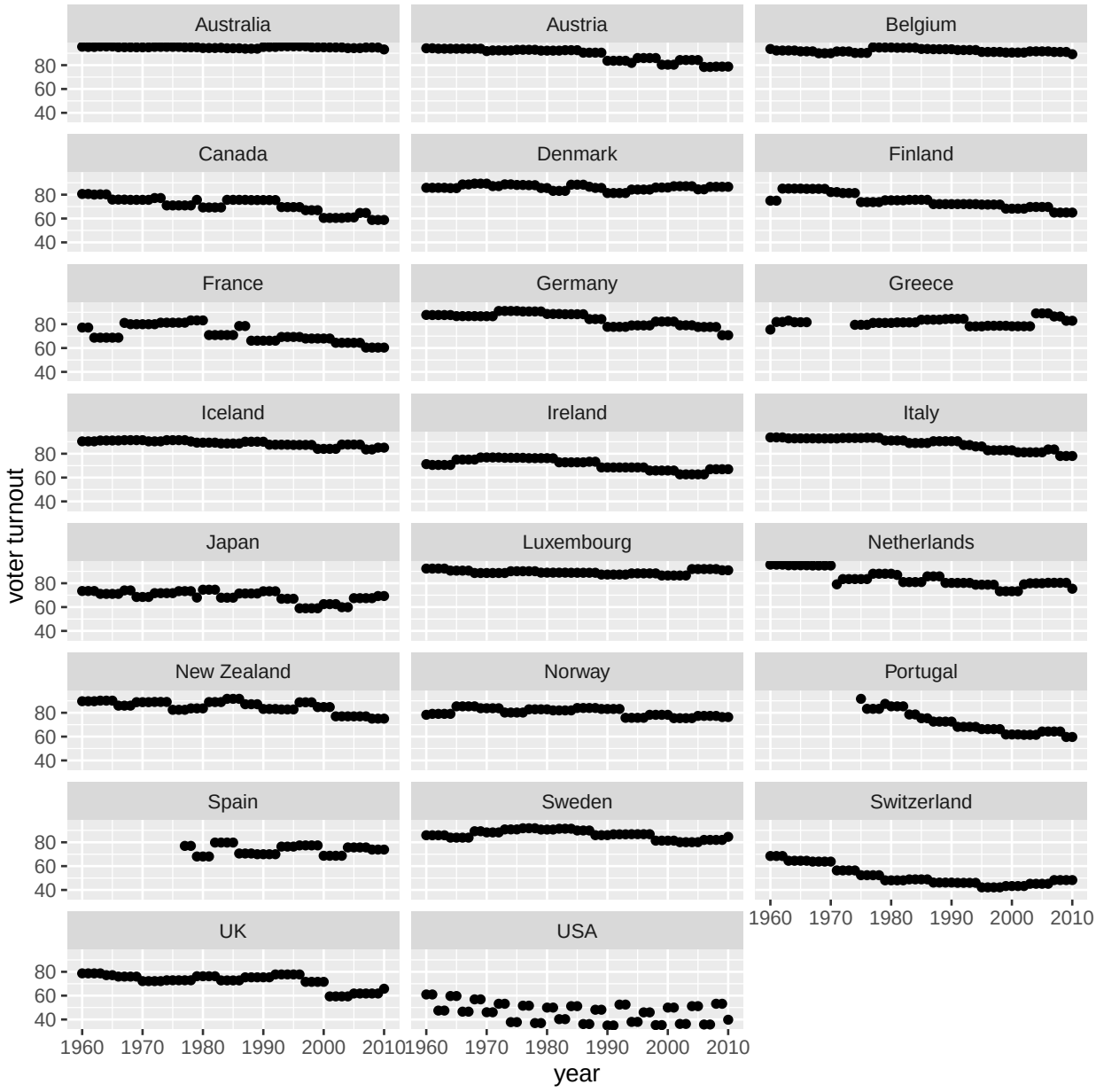
Here's how we create so-called trellis graphics, which create a series of plots, stratified by some conditioning variable, in this case by country. In ggplot terminology, this is called "faceting".

This is a powerful approach to dealing with multivariate data and is often useful for revealing patterns in data.

```
xyplot(vturn ~ year | country, data = cpds, ylab = 'voter turnout')
```



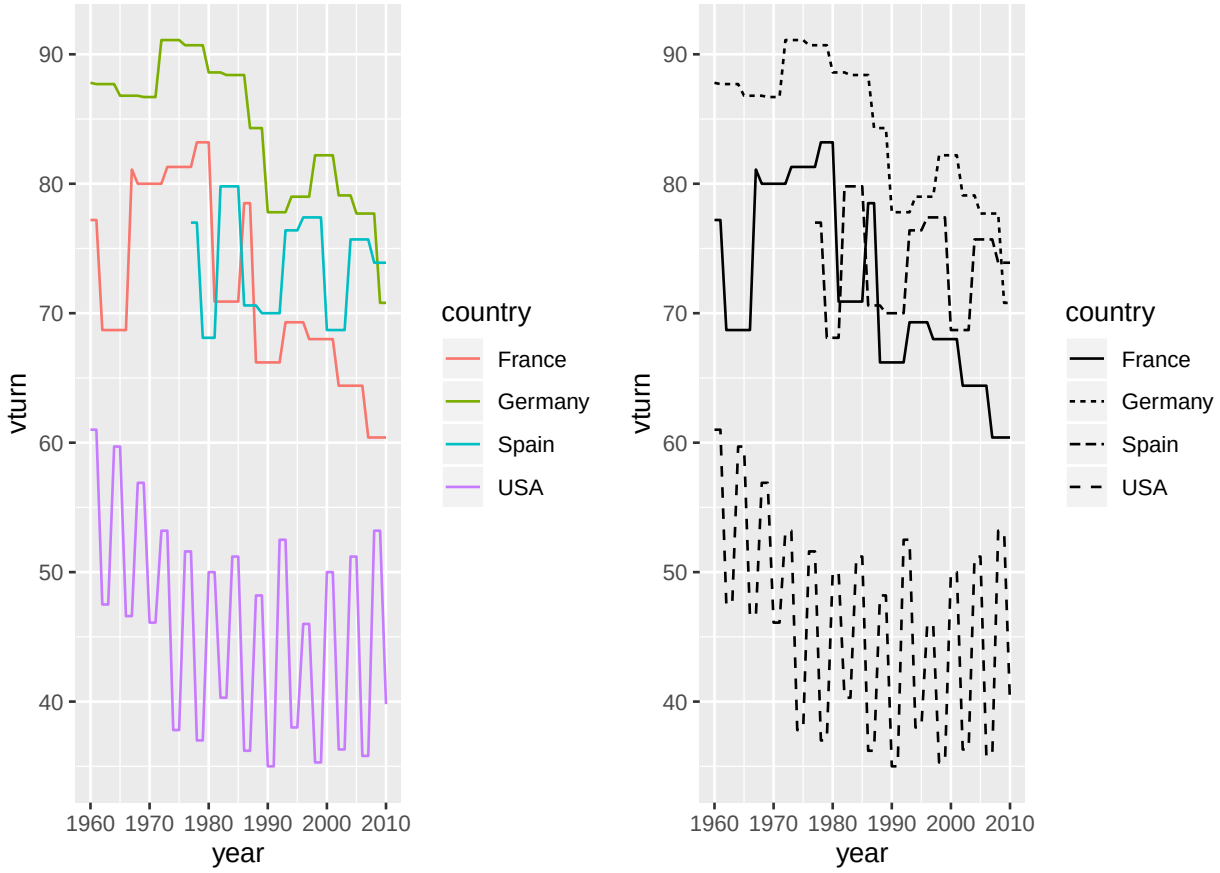

```
ggplot(data = cpds, aes(x = year, y = vturn)) + geom_point() +
  facet_wrap(~country, ncol = 3) + ylab(label = 'voter turnout')

## Warning:  Removed 39 rows containing missing values (geom_point).
```




In ggplot2, here's how we condition by using different colors or symbols within a single panel. Note the third plot in which we annotate the symbols based on a fourth variable - unemployment in this case.

```
countrySet <- c("USA", "Germany", "France", "Spain")
sub <- subset(cpds, subset = country %in% countrySet)

fig1 <- ggplot(data = sub,
       aes(x = year, y = vturn, color = )) +
       geom_line(aes(color = country))
```

```
fig2 <- ggplot(data = sub,
       aes(x = year, y = vturn, linetype = )) +
  geom_line(aes(linetype = country))

grid.arrange(fig1, fig2, nrow = 1, ncol = 2)

## Warning:  Removed 17 rows containing missing values (geom_path).
## Warning:  Removed 17 rows containing missing values (geom_path).
```



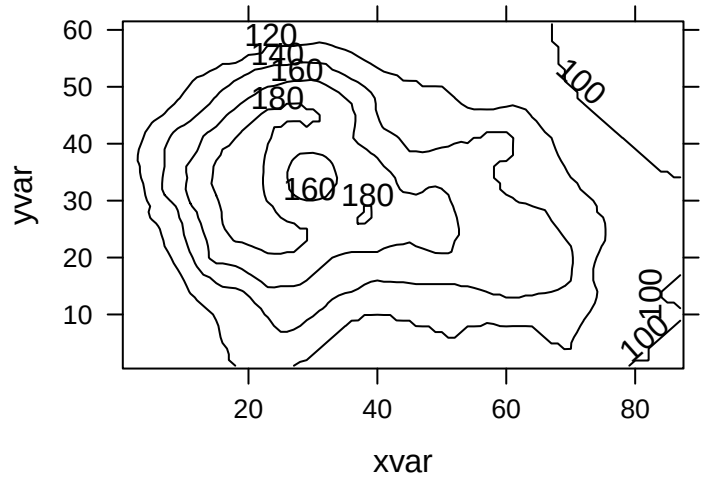

```
ggplot(data = sub,
     aes(x = year, y = vturn, color = , size = )) +
    geom_point(aes(color = country, size = unemp))

## Warning:  Removed 17 rows containing missing values (geom_point).
```


### 3.1.5 Contour plots

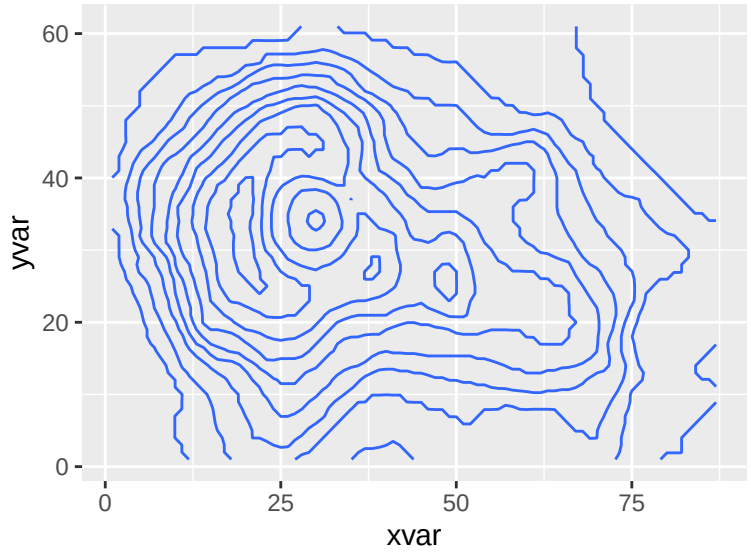
In general, I advise people to stay away from plots that attempt to show three dimensions, because it is very hard to see the whole plot without some of the features hiding other parts of the plot. Rather, contour and image plots are generally a better choice.

Here are contour plots:

```
require(reshape, quietly = TRUE)

data(volcano)
volcano3d <- melt(volcano)
names(volcano3d) <- c("xvar", "yvar", "zvar")
contourplot(zvar ~ xvar + yvar, data = volcano3d)
```

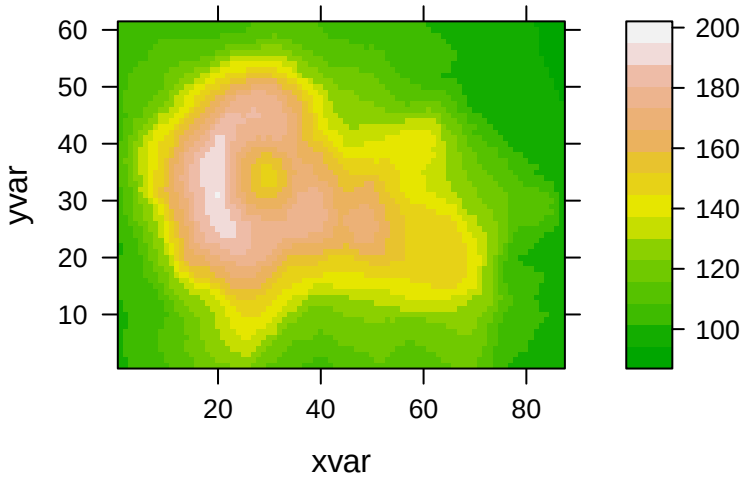



```
ggplot(data = volcano3d, aes(x = xvar, y = yvar, z = zvar)) +
  geom_contour()
```

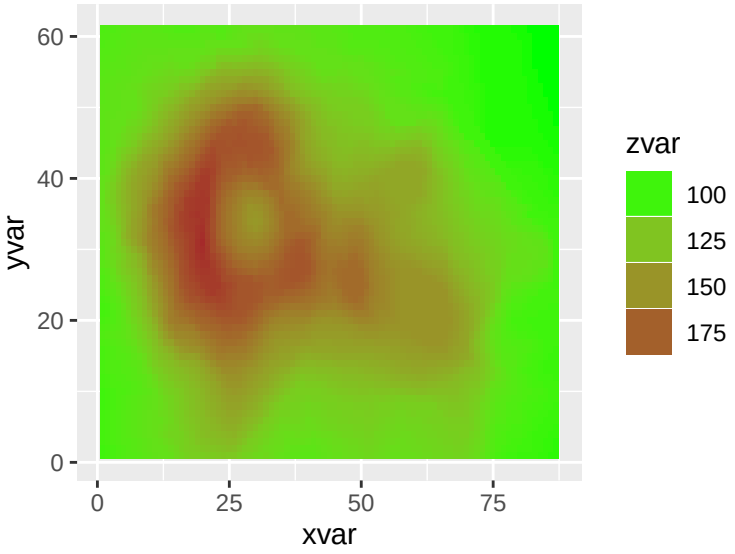



and here are image plots, which show surfaces based on variations in color:

```
levelplot(zvar ~ xvar + yvar, data = volcano3d,
          col.regions = terrain.colors(20))
```




```
ggplot(data = volcano3d, aes(x = xvar, y = yvar, z = zvar)) +
  geom_tile(aes(fill = zvar)) +
  scale_fill_gradient(low = 'green', high = 'brown', guide = 'legend')
```




I'm not a fan of that color scheme for the ggplot2 graphic, but didn't have time to figure out something better.

## 3.2 Some other details on *lattice*

The analog of *par()* for lattice is *trellis.par.set()* and *trellis.par.get()*. Often you can get a nice lattice graphic without much work, but modifying aspects of a lattice graphic may take some work.

Chapter 7 of the Chambers book discusses this to some degree. Note that *lattice* graphics are based on the *grid* package, so serious monkeying may require learning something about *grid*.

**Modularity and the *panel()* function** *Lattice* functions are highly modular and have consistent argument names (see p. 308 of Adler), unlike in *graphics*. The high-level functions take care of the overall logistics while low-level panel functions do the actual plotting within each panel. To add features to the panels, you need to do it through the *panel()* function argument:

```
library(MASS)
xyplot(time ~ dist, data = hills,
       panel = function(x, y, ...){
         panel.xyplot(x, y, ...)
         panel.lmline(x, y, type = "l")
         panel.loess(x, y, ...)
       }
)

## example with scatterplot matrix
splom( ~ hills,
      panel = function(x, y, ...){
        panel.xyplot(x, y, ...)
        panel.loess(x, y, ...)
      })
```

We can manipulate subpanels in a lattice graphic:

```
xyplot(Petal.Length ~ Sepal.Length | Species, iris, layout = c(2, 2))
trellis.focus("panel", 1, 2)
do.call("panel.lmline", trellis.panelArgs())
```

# 4 Animations and interactive graphics on the web

To create animations, check out the *animation* package for animations that can run in R or in HTML.

To create interactive graphics that users can interact with on the web, check out Shiny.

# 5 Graphics devices

Graphics are plotted on a *device*. In the old days when computer monitors were not high resolution or in color, this referred to a physical device, but nowadays this is a general term that denotes the context in which the plot is being made: typically on screen or as a file in a particular file format. The standard device in a UNIX environment is X11, basically a graphics window set up in the X11 windowing system. On-screen plotting is generally done with R interacting with a window manager for the operating system, so R is not interacting directly with the physical display. Often one needs to iterate to get a plot to look good when printed to a file; in particular the aspect (width to height ratio) (e.g., you can specify width and height in *pdf()*), the margin sizes relative to the size of the core plot, and size of plotting symbols and text relative to the size of the plot. That is the relative sizes when seen in a graphics window on the screen may be very different when printed to a file.

You can have multiple graphics windows open at once; you'll need to explicitly call the function for opening a device to set up any additional ones. *dev.cur()* tells the number of the active one and *dev.set()* allows to change it.

# 6 Graphics file formats

The *pdf()* function is a workhorse for creating an output file containing your R graphics. Analogues of *pdf()* include *postscript()*, *png()*, *tiff()*, *bmp()*, and *jpeg()*. If you have already made the plot in the graphics window and want to export it, you can use *dev.copy2pdf()*.

Note: Lattice graphics are optimized for the current plotting device, so using *dev.copy2pdf()* and the like is a bad idea since the graphic will have been optimized for the computer screen window. Instead use *pdf()* and the like to make the plot within the device.

## 6.1 Vectorized vs. rasterized file formats

pdf and postscript images are vectorized - in general elements of the image are symbolic objects (such as points, text, symbols, line segments, etc.) and when an image is resized, the items rescale appropriately for the new size without losing resolution. In contrast, with a rasterized format such as JPEG, individual pixels are plotted, and when an image is rescaled, in particular enlarged, one is stuck with the resolution that one used in plotting the figure (i.e., one has the original pixels but if you zoom, you only show some of them, losing resolution). One downside to vectorized images is that with a lot of points or line segments, they can be very large. For 2-d images such as created by *image()*, rasterized formats do make some sense inherently, though the other features in the file

(such as any text) is also rasterized.

## 6.2 File formats for journal articles

For inserting into Latex files for journal articles, I typically use *pdf()* in combination with *pdflatex* to compile the Latex file. In some cases for large images I will use *jpeg* or *png* files for figures in a paper.

Journals sometimes request either *postscript* or *encapsulated postscript* (EPS) format images. If you create a single R plot (as opposed to multiple pages), using *postscript()*, the result should be EPS compatible. You can do this with the arguments
`horizontal = FALSE, onefile = FALSE, paper = "special"`
or by calling *setEPS()*. If you have a postscript file that is not EPS, *ps2epsi* is a Ghostscript tool that can do the conversion from *ps* to *eps* (i.e., *epsi*). EPS is a standardized form of postscript and contains a bounding box describing the bounds of the image. EPSI files contain a bitmapped preview image in the "Interchange" format that systems can use to show an approximation of the image even if they can't render (i.e., display) postscript.

Discussing much about postscript is beyond our scope, but note that postscript files are just text in a particular language that the printer uses for generating a printout. So you can use *less* or *emacs* to view the file in UNIX. You'll see that it says that it's EPS and has a bounding box. It's occasionally handy to search for and replace text in a postscript file manually rather than regenerating the file. For example I could search for '*noise*' in *example.eps* (created in the demo code) and replace with something else. I could also monkey with the positioning, which should involve the numbers just before the text string. You can also do this sort of thing in pdf files as well.

## 6.3 Conversion utilities

UNIX has a lot of utilities for converting between image formats. Windows and Mac also have GUI-style programs for doing this.

In UNIX, *pdftops* will convert pdf to postscript and with the optional argument *-eps* to encapsulated postscript, while *ps2epsi* will create encapsulated postscript. *gs* (Ghostscript) will do a lot of different manipulations of ps and pdf files, including converting to jpeg and other formats and merging and splitting pages of pdf files. Here are some examples of command-line calls from within a UNIX shell:

```
# convert from ps to jpeg
gs -dNOPAUSE -r[xres]x[yres] -sDEVICE=jpeg -sOutputFile=file.jpg
file.ps
```

```
# extract pages from a pdf
gs -sDEVICE=pdfwrite -dNOPAUSE -dQUIET -dBATCH -dFirstPage=m
    -dLastPage=n -sOutputFile=out.pdf in.pdf
# merge pdf files into one pdf
gs -dNOPAUSE -sDEVICE=pdfwrite -sOUTPUTFILE=out.pdf -dBATCH in1.pdf

    in2.pdf in3.pdf
```

# 7 Colors

The default colors can be seen with *palette()*. Using *col=i* in a plot uses the ith element of the output of *palette()*. You can change the palette:

`palette(c("black", "yellowgreen", "purple")`

See *colors()* for the colors available by name. You can also use RGB levels, discussed next.

## 7.1 Colorspaces

Colors live in a 3-dimensional space that can be parameterized in several ways. One standard parameterization is RGB, which is a set of three numbers indicating the intensity of red, green and blue. We can use RGB levels to specify colors in R.

```
rgb(0.5, 0.5, 0) # each number is specified on scale of [0, 1]
plot(x, y, col = rgb(0.5, 0.5, 0))
col2rgb("yellowgreen") # on scale of {0,...,255}

n <- 16
pie(rep(1, n), col = rainbow(n))
  # rainbow varies hue while keeping s and v constant
pie(rep(1, n), col = rainbow(n, s = .5))  # reduce saturation
pie(rep(1, n), col = rainbow(n, v = .75)) # reduce brightness

library(colorspace)
pie(rep(1, n), col = rainbow_hcl(n, c = 70, l = 70))
                     # colors in the HCL colorspace
## none of these colors stand out more than others, unlike the RGB rainbow
```

Notice *rgb()* gives us back the color as a hexadecimal number (*#RRGGBB*), where each of *RR*, *GG*, and *BB* is 2-digit hexadecimal number (base 16) in the range 0 (00) to 255 (FF), so red is #FF0000. A string in this format can be used to specify colors and you'll run across this in R if you work with colors.

Another parameterization is HSV: *hue*, *saturation* (colorfulness metric), and *value* (brightness). Let's see the demo code to see how colors vary as we change HSV values using *rainbow()*.

A parameterization that uses a more absolute measure of colorfulness than saturation is HCL (hue, chroma, luminance). In the example in the demo code, none of the colors stands out more than the others.

The *colorspace* package provides tools for manipulating colors.

## 7.2 Color sequences

If we're using color to illustrate a continuous range of values, we need a meaningful color sequence. To construct a continuous color set giving a sequence of colors you can use a variety of color schemes: *rainbow()*, *heat.colors()*, *terrain.colors()*, *topo.colors()*, *temp.colors()*, and (in the *fields* package), *tim.colors()*. I know Tim! He likes to fish.

The main thing to avoid is a sequence in which the colors do not appear to vary smoothly or in some cases may not even appear monotonic. Let's examine a variety of the sequences (see the demo code).

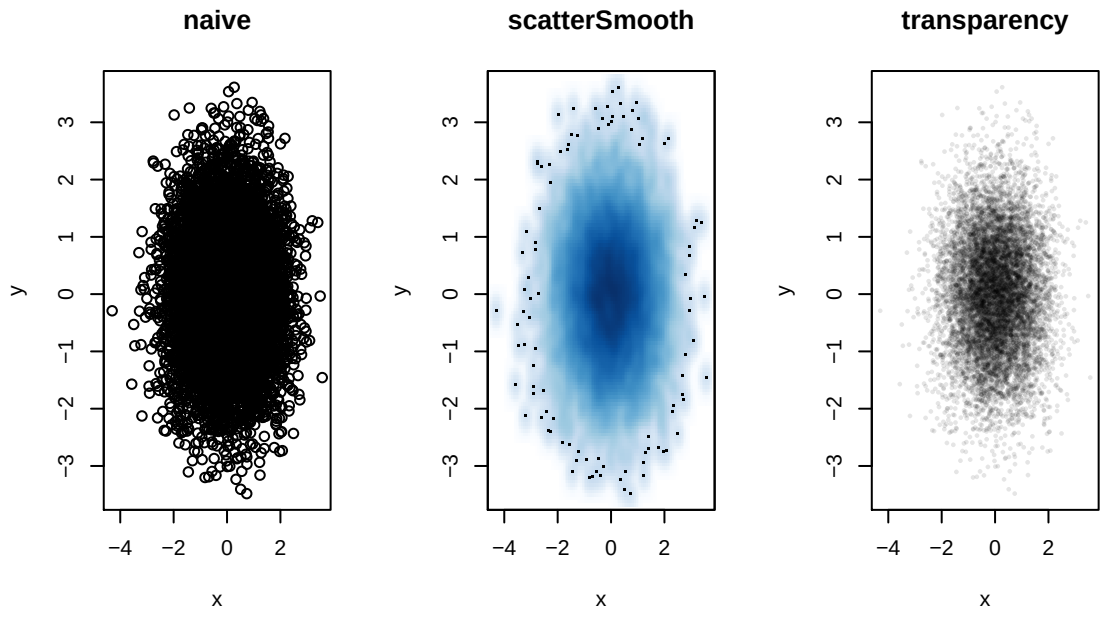
*temp.colors()* is a good blue to red "diverging" color scheme that emphasizes magnitudes around a central point, with two hues - one for each direction.

The *RColorBrewer* package is good for choosing colors for unordered levels, sequential ordering, and two-way diverging color ordering and the *ColorBrewer* website provides recommendations. We'll see an example in the section on mapping.

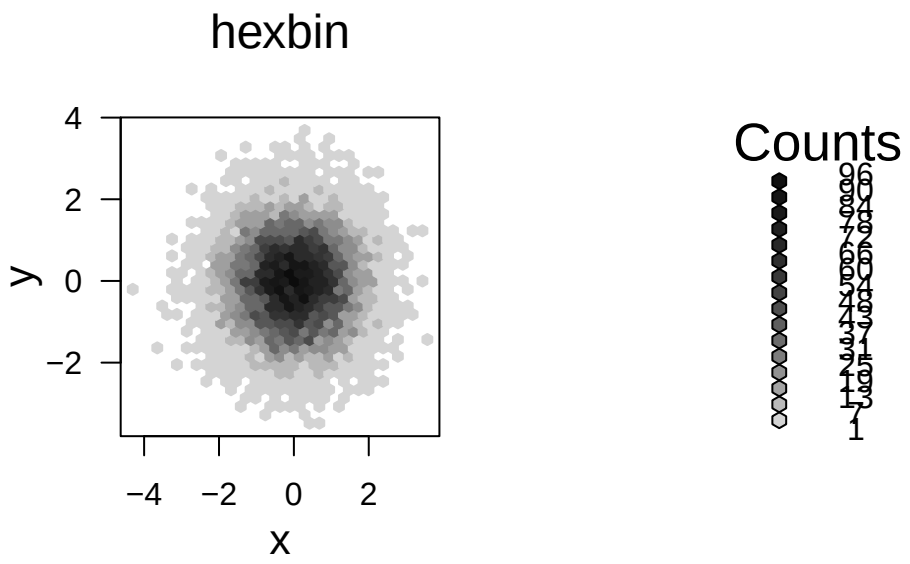
## 7.3 Overplotting of points

As a sidenote, if you have a scatterplot with many points that will overplot each other (as well as creating a huge file), consider the *scatterSmooth()* function as well as the *hexbin* package. The former creates a two-d density plots with outlying individual points included, while the latter creates an empirical two-d density by binning into hexagonal areas. A third approach is to have your color be partly transparent, so that overplotting results in darker colors. Note that this may not work on all devices. We can specify transparency level as either the 4th number in *rgb()* on a scale of 0 (transparent) to 1 (opaque - the default), or as a fourth hexadecimal number on the scale of 00 to FF (0 to 255); e.g., *#FF000080* would be half-transparent red, since 80 is one-half of FF in base 16.

```
require(hexbin, quietly = TRUE)
x <- rnorm(10000); y <- rnorm(10000)
par(mfrow = c(1, 3))
plot(x, y, main = 'naive')
smoothScatter(x, y, main = 'scatterSmooth')
plot(x, y, col = rgb(0, 0, 0, .1), pch = 16,
  cex = .5, main = 'transparency')
```




```
par(mfrow = c(1, 1))
bin <- hexbin(x,y)
plot(bin, main = 'hexbin')
```




## 7.4 Colorblindness

One thing to be aware of is that 7-8% of men are color blind. As we see in the demo code, the standard result of this is to make it difficult to distinguish red and green, so one may want to avoid color schemes that have both of these in them. We can use *dichromat()* from the *dichromat* package to assess the effect of colorblindness on viewing of one's images; see more in the demo code file.

```
library(dichromat)
showpal <- function(colors){ # helper function to show colors
  n <- length(colors)
  plot(1:n, rep(1, n), col = colors, pch = 16, cex = 4)
}

dev.off() # close the graphics windows to clear out old color stuff
par(mfrow=c(2, 1))
showpal(palette()) # show default palette colors
# here's how those look with standard colorblindness
showpal(dichromat(palette()))
## notice red and green similarity
```