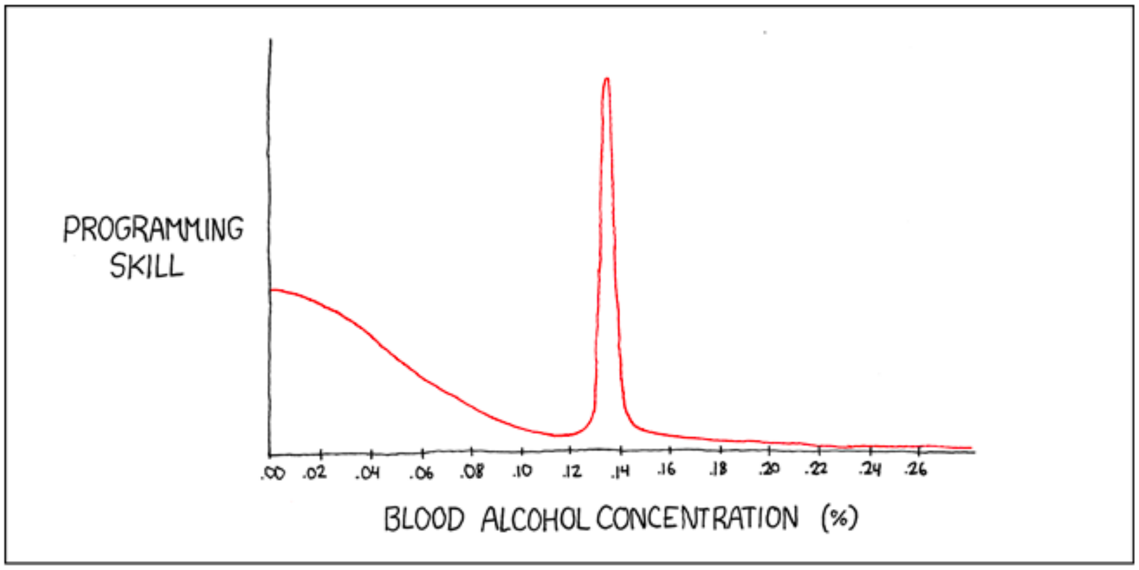
R Workshop, Session 5
Publication-Quality Graphics

by Rebecca Clark, once again by standing on the shoulders of many other R aficionados



Session 5, Lesson 1. Publication-Quality Graphics With `base` and `lattice`

This material has been adapted from Paul Murrell's book, *R Graphics*. I highly recommend the book.

1.A. Overview

There are two major methods for producing graphics in R, `base` graphics and `grid` graphics. So far, we have worked with `base` graphics, which are generated in the same way you would draw with ink on paper. You build up an image by drawing fixed things on it, and things that are drawn are permanent, although you could cover them with something else or move on to a clean sheet of paper if they got too ugly. They allow easy production of good quality scientific plots.

The `grid` package is the basis for newer graphics systems. You have access to the individual pieces of a graph and can modify them - it's more like a physical model being built and displayed, rather than just drawn. Both the `lattice` and `ggplot2` packages provide functions for high-level plots based on `grid` graphics. It's possible to use both packages without knowing about any of the underlying grid concepts, but in some cases there can be benefits to viewing the plots as grid output.

Note that grid graphics are designed to be "device independent." Directions are given for where to draw, and these commands work on any device, but the actual appearance will depend on the device (e.g. paper vs. computer screen) because the capabilities of different devices vary.

1.B. Base graphics: high-level functions

Simply use these to describe the plot, and R will draw it. See the accompanying script file for examples.

barplot()
dotchart()
pie()
hist()
boxplot()
plot() # use **type=""** to indicate plot type; look at **?plot** and **?par** for other optional arguments
qqplot() # Used for examining univariate or bivariate distributions

Which to use? Consider the type of data, your audience, and how one's visual system decodes the graph. People are good at eyeballing lengths and positions, not as good at slopes and angles, and bad at areas, volumes, and shades of color. So bar lengths and areas are easy as bar charts (be careful to base them at zero!), dot charts are better for indicating positions, and pie charts aren't a good idea, period.

1.C. Base graphics: Low-level functions

Use these to annotate your plot:

points(x, y, …)
lines(x, y, …)
text(x, y, labels, …) # Adds text into the graph
abline(a, b, …) # Adds the line y=a + bx
abline(h=y, …) # Adds a horizontal line
abline(v=x, …) # Adds a vertical line
polygon(x, y, …) # Adds a closed and possibly filled polygon
segments(x0, y0, x1, y1, …) # Draws line segments
arrows(x0, y0, x1, y1, …) # Draws arrows
symbols(x, y, …) # Draws circles, squares, rectangles, stars, thermometers, boxplots
legend(x, y, legend, …) # Draws a legend

Elements outside the plot region
title(main, sub, xlab, ylab, …) # Adds a main title, subtitle, and x- and/or y-axis labels
mtext(text, side, line, …) # Draws text in the margins
axis(side, at, labels, …) # Adds an axis to the plot
box(…) adds a box around the plot region

1.D. Setting graphical parameters

You have two chances to do this: when opening the plotting device, e.g.
windows(…) # For MS Windows machines
x11(…) or **X11(…)** # Opening a screen device in Unix-like systems
quartz(…) # Mac OSX
postscript(…) # Opening a file for Postscript output for printing
pdf(…)
jpeg(…)
png(…)

And after the device is open, you can use the **par(…)** function to set graphical parameters. There are a TON of options, but here are some highlights:
mfrow=c(m, n) # Draw m rows and n columns of plots
mfg=c(i, j) # Draw the next figure in row i and column j next
ask=TRUE # Tell you before erasing a plot to draw a new one
cex=1.5 # For Character EXpansion; there are separate cex.axis/etc. parameters for specific text regions
mar=c(side1, side2, side3, side4) # Sets margins to the given numbers of lines of text on each side
oma=c(side1, side2, side3, side4) # Sets the outer margins outside the array of plots
usr=c(x1, x2, y1, y2) # Sets the coordinate system within the plot with x and y coordinates on the given ranges.

You can use **par** in several ways depending on how you put together the arguments:
par("mfrow") # Will tell you the current value
par(mfrow=c(1,2)) # Lets you set one value.

1.E. Saving Your Output

If you want to save output, open a plotting device that will do this (**postscript, pdf, jpeg, png, pictex, xfig, bitmap, win.metafile, bmp…**). To finish and close the plotting device, use **dev.off()** at the end.

The output format will dictate whether multiple pages are supported - for instance, PostScript and PDF will allow it, but PNG will not. It's often possible to instead specify that each page of output produce a separate file with the argument **onefile=FALSE**. Then specify a pattern for the filename like **file="myplot%03d"** so that **%03d** is replaced by a three-digit number (padded with zeros)

1.F. The `lattice` package

The `lattice` package implements the Trellis Graphics system, with some extensions. Simple uses work and look like traditional graphics functions. So, why bother?

1. *Superior default appearance* - e.g. default symbols make it easier to distinguish between groups, and there are subtle improvements like horizontal y-axis tick labels
2. There are *powerful extensions* of plots.
3. The output is *grid output*, so you can use powerful grid features for annotating, editing, and saving graphics output.

Lattice plotting functions tend to have long lists of arguments, but can produce a wide range of different types of output. Many of the arguments are shared, so look at the help documentation for xyplot() as a starting point for explanations. Note that lattice plots use a formula argument, which can vary considerably. This is extremely handy for complex types of data.

Lattice function	Traditional analogue, where it exists
barchart()	**barplot()**
bwplot()	**boxplot()**
densityplot()	*none*
dotplot()	**dotchart()**
histogram()	**hist()**
qqmath()	**qqnorm()**
stripplot()	**stripchart()**
qq()	**qqplot()**
xyplot()	**plot()**
levelplot()	**image()**
contourplot()	**contour()**
cloud()	*none*
wireframe()	**persp()**
splom()	**pairs()**
parallel()	*none*

We'll do the bulk of our exploration of lattice plots in the accompanying script file, but here are several additional hints for tweaking lattice plots.

Parameter settings
Organized in parameter groups:

plot.line(col, lty, lwd)
plot.symbol(cex, col, font, pch)

Some have "global" effects, like font size. Others are more specific, like **strip.background**.

Arranging plots
To arrange panels and strips within a plot: examine the **layout** and **aspect** arguments.

```
layout=c(rows, columns, pages)
aspect=1 #square panels
```

Adding output
Use **trellis.focus()** to return to a particular panel or strip of the current lattice plot, then add further output with **llines()** or **lpoints()**, for instance.

Session 5, Lesson 2. Publication-Quality Graphics with `ggplot2`

2.A. Overview

As mentioned earlier, `ggplot2` relies on the grid graphics system, but it uses a different approach and syntax for creating graphics. It was built as an extension of Wilkinson's "grammar of graphics" (2005), and is based around thinking about why we create visualizations – to better understand our data. Since visualization is just part of the data analysis process, it needs to be coupled with transformation and modeling to build understanding. `ggplot2` was designed with this in mind.

The process of using `ggplot2` involves thinking about how your data will be presented visually, and then describing that representation using a declarative language built out of a set of independent building blocks, kind of like nouns and verbs, that allow you to build up a plot piece by piece.

Hadley Wickham, the author of `ggplot2`, also notes that there are some major benefits to the fact that `ggplot2` is code-based and programmable, unlike many other visualization tools that rely on a graphical user interface. Programmability (as you have hopefully already noticed) facilitates reproducibility, automation, and communication – all key to good science. It also simplifies the process of iteratively updating figures.

2.B. Getting started

There is a basic plotting function, **qplot()**, designed to function in a manner similar to the plotting functions for base graphics. However, in the interest of covering the `ggplot2` grammar, we are going to skip over working with **qplot()**. The chapter from Wickham's book on `ggplot2` that covers **qplot()** is available through the `ggplot2` documentation site:

http://docs.ggplot2.org/current/qplot.html

The Grammar

A layered grammar defines a plot as a combination of elements, listed here and described in greater detail below.

- A default dataset, plus a set of *mappings* from *variables* to *aesthetics*
- One or more *layers*, each composed of: a geometric object, a statistical transformation, and a position adjustment (plus, optionally, a dataset and aesthetic mappings)
- One *scale* for each aesthetic mapping
- A *coordinate system*
- The *faceting specification*

`ggplot2` goes through a series of steps to process data based on these concepts, and in the end the processed data are used to render *geoms*, the geometric shapes within the plot (points, lines, polygons, box plots, error bars, etc…).

2.C. ggplot2 *geoms*

For translating between ggplot2 and the other plotting packages, it's easiest to compare the other plotting packages to ggplot2's *geoms*, although note that deciding upon a *geom* happens later in the plotting process as compared to choosing a function in lattice or base graphics. See also: http://docs.ggplot2.org/current/translate_qplot_lattice.html

Lattice function	Traditional analogue	ggplot2 *geom*
barchart()	**barplot()**	**geom_bar(), geom_bin2d()**
bwplot()	**boxplot()**	**geom_boxplot()**
densityplot()	*none*	**geom_density(), geom_density2d()**
dotplot()	**dotchart()**	**geom_dotplot()**
histogram()	**hist()**	**geom_histogram()**
qqmath()	**qqnorm()**	*Slightly complicated. See here: http://stats.stackexchange.com/questions/12392/how-to-compare-two-datasets-with-q-q-plot-using-ggplot2*
stripplot()	**stripchart()**	**geom_point(position = "jitter")**
qq()	**qqplot()**	***stat_qq(), or use qplot(sample = y)**
xyplot()	**plot()**	**geom_point()**
levelplot()	**image()**	**geom_tile()**
contourplot()	**contour()**	**stat_contour(), geom_density2d()**
cloud()	*none*	*none*
wireframe()	**persp()**	*none*
splom()	**pairs()**	**See ?plotmatrix()**
parallelplot()	*none*	*Requires data manipulation. See here: http://learnr.wordpress.com/2009/07/15/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-5/*

2.D. Basic syntax

Two pieces of the above description drive ggplot2's flexibility and efficiency, *layers* and *aesthetic mappings*. A ggplot2 object is composed of one or more *layers*, each containing a different graphical object, or *grob*.

The **ggplot()** function defines the plot's *base layer,* including the name of the input data frame and associations between a subset of its variables and their roles in the graph (base mappings).

```
ggplot(iris, aes(Sepal.Length, Sepal.Width))
```

This won't produce any output, however, because it doesn't specify the geometry we want to use to display the data – points, lines, boxplots, barplots – any of the options from the table above, or even more (see the online documentation at http://docs.ggplot2.org).

That requires a second element added to the command:

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) + geom_point()
```

The *geom* does not need any arguments because it will take the relevant information (data and aesthetic mappings) from the default provided in the base layer. It's possible to keep adding additional layers, too – say, **geom_smooth()** to add a smoothed "conditional mean." [fyi, The default fit for **geom_smooth()** is "loess" for smaller sample sizes (<1000) and "gam" for larger sample sizes.]

Each *geom* function has arguments that can be used to control the *geom's* appearance. For example, you can specify a linear fit, *without* a standard error by using `geom_smooth(method="lm", se=FALSE)`.

2.E. *geoms* versus *stats*

In ggplot2, *geoms* are functions that convert transformed numeric data into some type of geometric object (points, lines, bars, box plots). The functions that do the transformations are called *stats*. While *stats* and *geoms* are independent of one another, every *geom* has a default stat (e.g. **stat_bin** is used for **geom_bar** and **geom_histogram**), and *stats* have default *geoms* (e.g. **geom_ribbon()** and **geom_smooth()** for **stat_smooth()**), and sometimes you may have reasons to call a *stat* directly to tinker with a data transformation and then specify a consequent *geom*. Note also that **stat_summary()** is something of a special case – see some of the additional resources listed at the end for more details if you want precise control of the presentation of summary statistics.

2.F. *mapping* vs. *setting*

Sometimes you may want to use an object's attributes to symbolize something about it – say, using an object's color, shape, or size to distinguish it from other objects. In those cases, in the `ggplot2` parlance, you are *mapping* that characteristic from within the aesthetics function (**aes()**). In contrast, at other points you may wish to just display everything with a fixed attribute of some particular sort – say, changing the default shape to triangles. In this case, you should *set* that characteristic within the relevant *geom* layer (e.g. **geom_point(shape=17)**).

Now, take a look at some examples in greater detail in the accompanying sample script.

2.G. Polishing your plots

Guides

A *guide* is a graphical object that aids in the interpretation of a statistical graphic. The two classes are *positional*, and *nonpositional*. Axes are positional guides, referring to the range of values in a single direction for a continuous variable, or to levels of a discrete factor. *Nonpositional* guides usually have to be indicated with the help of a legend, to illustrate the relationship between individual values of an aesthetic and the corresponding variable values. Use the **guides()** function to manipulate aspects of a legend or color bar guide.

Scales
Scale functions are used to control the *rendering* of a guide, and all generally allow the specification of: breaks, values, and labels. Positional axes tend to have the form **scale_dir_type**, as in **scale_x_discrete**, or **scale_y_continuous**. There are some built-in scale transformations as well (**scale_x_log10**), but most scale transformations need to be defined through the `scales` package. To adjust legend-related scale functions, use scale functions of the form **scale_aes_type** – i.e. **scale_fill_gradient** or **scale_colour_manual**.

Themes
The *theming* system controls non-data aspects of the plot. This system got a major overhaul in version 0.9.2, so Wickham's book is no longer the best source of information for how to update these aspects of plots. To observe the defaults, use **theme_get()** – similar to using **par()** for base graphics. You can set the theme globally with **theme_set()** or adjust the theme for a specific piece of a specific plot by adding **+ theme()** details to the plot object. There are also several default themes to choose among. The initial one is **theme_grey()**, chosen based on specific aesthetic principles; **theme_bw()** may get you fairly close to a plot ready for scientific publication. I tend to put the following at the beginning of my R script files to generate my desired default appeareance:

```
theme_set(theme_minimal())
theme_update(axis.line = element_line(),
 panel.grid.major=element_blank(),
 panel.grid.minor=element_blank(),
 strip.background=element_rect(colour="gray"))
```

Axis and Main Titles
Modify with **xlab()**, **ylab()** and **ggtitle()**, or use **labs(x = "X thing", y = "Y thing", title = "Main title")**

Positional adjustments
Use to avoid overplotting – implement with **geom_jitter()** or use **position= "jitter"** within **geom_point()**.
Also use to arrange elements of bar charts (**position = stack**, **dodge**, **fill**, or **identity**)

Coordinate systems
Cartesian (default): **coord_cartesian**
Equal-scale Cartesian coordinates: **coord_equal**
Swap x and y: **coord_flip**
Transformed Cartesian coordinates: **coord_trans**
Map projections: **coord_map**
Polar coordinates: **coord_polar**

Annotations
Plots can be embellished with things like text labels, fitted equations, *P* values, tables, pictures, or even inset graphics. See the documentation at docs.ggplot2.org for **annotate()**.

2.H. Faceting

This is the ggplot2 alternative to trellis plots. There are two functions, **facet_wrap()** and **facet_grid()**. **facet_wrap()** allows conditioning with one added variable, reshaping a 1D ribbon of plots into a 2D arrangement – use **ncol** or **nrow** to control exact arrangement. **facet_grid()** uses formula notation to generate a 2D grid of graphics panels by column and/or row – as in, **facet_grid(rows ~ columns)**. Note that there are different options for **scales** when faceting – they can be **"fixed"** (the default, all facets have identical scales), **"free_x"**, **"free_y"**, or **"free"**. The default arrangement (**as.table=TRUE**) is to place the highest values in the lower-right corner; if this is set to **as.table=FALSE**, the plots are rearranged with the highest value in the upper right.

2.I. Arranging multiple plots, saving plots

One method for arranging multiple plots involves the use of the **annotation_custom()** function, in conjunction with some functions from the `grid` and `gridExtra` packages. These allow you to assign the plot information itself to an object, as follows:

```
g <- ggplotGrob(plot) # The image information is saved as
"g"
```

Then:
```
p + annotation_custom(grob = g, xmin = 20, xmax = 48, ymin
= -330, ymax = 640)
```

But note that this gets complicated for faceted plots. The `gridExtra` package is also useful for positioning multiple plots on a page without having to learn the details of grid graphics. Once each plot is saved as an R object (as illustrated above), they can all be combined in one page with **grid.arrange()**:

```
grid.arrange(plot1, plot2, plot3, plot4, ncol=2)
```

Lastly, the **ggsave()** function can be used to save the current plot – it's a wrapper for:

```
pdf("myplot.pdf")
print(p) # Where p is a ggplot object
dev.off()
```

There are more options in the **ggsave()** help documentation (e.g. size, file format, etc).

Resources and References

Another thorough introductory tutorial, with more examples:
http://jofrhwld.github.io/avml2012/

The online documentation for ggplot2 is an invaluable reference tool once you have some practice: http://docs.ggplot2.org

A library of lattice graphics from *Lattice: Multivariate Data Visualization with R,* reproduced in ggplot2. Start here and work forwards or backwards:
http://learnr.wordpress.com/2009/07/15/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-part-5/

Or see this for a pdf version of the whole (amazing!) project:
http://learnr.wordpress.com/2009/08/26/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-final-part/

ggplot2 wiki on Github: https://github.com/hadley/ggplot2/wiki

Chang, W. 2012. *R Graphics Cookbook*. O'Reilly Media, 416 pp. Companion website:
http://www.cookbook-r.com/Graphs/

Ito, K., & D. Murphy. 2013. Application of *ggplot2* to pharmacometric graphics. *CPT: Pharmacometrics & Systems Pharmacology 2*, e79. Doi:10.1038/psp.2013.56

Wickham, H. 2009. *ggplot2: Elegant Graphics for Data Analysis* (Use R!). Springer, Berlin, 213 pp.

Wilkinson, L. 2005. *The Grammar of Graphics* (*Statistics and Computing)*, 2nd edition. Springer, Berlin, 690 pp.