# Contents

1 Brief Primer on Good Graphics  
1.1 Graphical Perception  
1.2 Advice  

2 Graphics Philosophies  

3 The Plot Function  
3.1 getting familiar with the function  
3.2 Default Plotting Methods  
3.3 Controlling the Plotting Region  
3.4 Example of Building a Scatterplot  
3.4.1 Adding a Legend  
3.4.2 Adding a Regression Line  
3.4.3 Identifying Points in the Plot  
3.5 Other Plots  

4 ggplots  
4.1 Deducer  
4.1.1 Scatterplot  
4.1.2 Bar Graph  
4.2 Back to Code  
4.3 Scatterplots  
4.4 Other Plots  
4.4.1 Histograms and Barplots  
4.4.2 Dotplot  
4.5 Faceting  

5 Exercises  

1 Brief Primer on Good Graphics

While we could define graphs in lots of different ways, Kosslyn (1994, 2) defines graphs as:

“a visual display that illustrates one or more relationships among numbers”

He goes on to define a graph as “successful” if:

“the pattern, trend or comparison it presents can be immediately apprehended.”

Graphs work by encoding quantitative and qualitative information with different graphical elements - points (plotting symbols), lines (patterns), colors, etc...

• Graphical perception is the visual decoding of this encoded information
• For our graphs to be successful, readers must be able to easily and efficiently decoded the quantitative and qualitative information.

Shah (2002) identifies three components to the process of visual decoding

1. Identify important features of the display (what Cleveland calls “Detection”) \(\rightarrow\) Characteristics of the display

2. Relate the visual feature to the phenomenon it represents in the observed world \(\rightarrow\) Knowledge of graphs

3. Interpret the relationships interps of concepts being quantified \(\rightarrow\) Knowledge of content (i.e., concepts being discussed).
1.1 Graphical Perception

We can use lots of different elements to encode numeric information

- Angle
- Area
- Color hue
- Color saturation
- Density
- Length (distance)
- Position along a common scale
- Position along identical, non-aligned scales
- Slope
- Volume

Figure 1: Position

Figure 2: Length
Figure 3: Slope

![Graph showing a line with increasing values as x increases.](image)

Figure 4: Angle

![Pie chart with angles](image)

GRAPHICAL

A measurement can be extracted by judgments along the horizontal scale. But we can also judge the local rate of change of $y$ as a function of $x$ by judging the relative values of the slopes of the connecting lines; the overall visual impression is an increasing trend in the local rate of change as $x$ increases. 

Figure 4.6 is a pie chart. To extract the percentages visually we can make angle comparisons. Figure 4.7 is a graph that shows three variables: $x$, $y$, and $z$. The first two, $x$ and $y$, are shown in the usual way by positions along the two scales, and $z$ is portrayed by the areas of the circles. Thus to visually decode the values of $z$ we must make area judgments.

Figure 4.8 is a scatterplot of two variables. One aspect of the data that we can judge from the graph is the relative number of points per unit area in different regions of the plane. To extract this information we must judge the visual density of the points; for example, the density of the point cloud appears to be greater in the middle than in the extremes in the upper right and lower left.
Figure 5: Area

The relative number of points per unit area in Y
different regions of the graph can be decoded by density judgments.

Figure 6: Density
Others

- Volume is used rarely to encode quantitative information in scientific graphs.
- Color hue (i.e., different colors) are often used to encode categorical (not quantitative) information. Cleveland does not discuss this much (book published in 1985).
- Color saturation (i.e., the intensity of the color) can be used to encode quantitative information.
  - Be careful about making graphs color-blind friendly
  - Realize that many graphs will may appear in color on the screen, but may not be printed in color
  - See http://colorbrewer2.org for more advice

Further, our visual processing system is drawn to make comparisons within rather than between groups

- When observations are clustered together, our minds tend to think they belong to the same group.
- Groups should be formed such that the intended comparisons are within, rather than between groups.

Cleveland identified a set of elementary perception tasks and the ordering (below from easiest to hardest) of their difficulty.

1. Position along a common scale
2. Position along identical, nonaligned scale
3. Length
4. Angle - Slope
5. Area
6. Volume
7. Color hue, color saturation, density

There are a number of other aspects of graphs that can make apprehension of the graphs main point more or less difficult.

- Explanation - features of the display were not (sufficiently) explained
- Discrimination - features of the display were hard to distinguish from each other because of the size or configuration.
• Construction - mistakes in the numbering/labeling of tick-marks, etc...

• Degraded Image - the reproduction of the image reduced the quality enough to make it hard to read.

Below are some comparisons we can consider to try to understand how to make graphs appropriately.

Figure 7: Make the Data Stand Out: Wrong

Figure 8: Make the Data Stand Out: Right
Figure 9: Use Prominent Graphical Elements

Figure 10: Do Not Obscure Points
Figure 11: Two Scale Lines, Reference Line, Ticks Outside

Figure 12: Do Not Clutter Data Region
Figure 13: Do Not Clutter Data Region II

Figure 14: Consider Juxtaposing Instead of Superposing
Figure 15: Use (but don’t overuse) Reference Lines

Figure 16: Labels if Necessary, but Don’t Obscure Data

Figure 17: Overlapping Symbols Must Be Distinguishable
Figure 18: Superposed Datasets Must be Distinguished
1.2 Advice

Plotting Symbols:

- Use open plotting symbols (rather than filled in)

Legends:

- Legends should be comprehensive and informative - should make the graph stand alone.
- Legends should:
  - Describe everything that is graphed
  - Draw attention to important aspects
  - Describe conclusions.

Readers should not be left “guessing” about what the graphical elements mean.

Error Bars:

Error bars (or regions) should be clearly labeled so as to convey their meaning. Common error bars are:

- $\pm 1$ SD.
- $\pm 1$ SE
- 95% Confidence Interval.

Axes (scales):

- Choose tick marks to include all (or nearly all) of the range of the data
- Choose scales so that the data fill up as much of the data region as possible
- Sometimes two different scales can be more informative
- Axes should be labeled (including units where applicable).
  - Log and level scales (bottom and top scale lines)
  - Differnet operationalization (e.g., age and year of birth).
- Use appropriate (i.e., the same) scales when comparing.
- Do not feel obliged to force scale lines to include zero.
- Use data transformations (e.g., logs) to improve resolution.
Aspect Ratio:

Cleveland found through experimentation, that $45^\circ$ was the angle at which we can best discern differences in slope. Thus, he developed a technique called “banking to $45^\circ$” that finds the optimal aspect ratio for graphs.

- If you’re using lattice, then using the option `aspect= 'xy'` will do the banking for you.
- If you’re using some different graphing tool (like R’s traditional graphics engine), then setting the aspect ratio to one will come close enough, usually.

2 Graphics Philosophies

R has four different graphics systems - base, grid, lattice and ggplot2, all with different philosophies about how graphs are to be made.

**base** The base system works like painting - you can sequentially add elements to a graph, but it is quite hard to take elements away (in fact, it is impossible). Layers can be added until the graph conveys just what the user wants.

**lattice** The lattice system is based on Cleveland’s Trellis system developed at Bell Labs and is built on top of the grid graphics system. These are particularly good for grouped data where multi-panel displays are needed/desired. These operate more like setting up a track of dominoes when you were a kid. You line them all up and then knock the first one down and all the others fall as you’ve arranged them. If you messed up the arrangement, it isn’t going to be as cool/pretty/interesting as you thought. Lattice graphs work the same way. All elements of the graph must be specified in a single call to the graphics command and if you don’t do it right, it will not be as cool/pretty/interesting as you want.

**ggplot2** Hadley Wickham wrote ggplot2 and describes it in his 2009 book as an implementation of Leeland Wilkinson’s “Grammar of graphics”. This builds a comprehensive grammar- (read model-) based system for generating graphs.

**grid** The grid system was written by Paul Murrell and is described in his 2006 book, which is now in its second edition. This system, while very flexible, is not something that users often interact with. We are much better off interacting with grid through lattice.

The base graphics sytem, lattice and grid are all downloaded automatically when you download R. ggplot2 must be downloaded separately and all but the base system must be loaded when you want to use them with the `library()` command.

As you can imagine, these different systems offer quite different solutions to creating high quality graphics. Depending on what exactly you’re trying to do, some things are more difficult, or impossible, in one of these systems, but not in the other. We will certainly see examples of this as the course progresses. I have had a change of heart
about these various systems recently. Originally, I did everything I possibly could in
the traditional graphics system and then only later moved to the lattice system. This
probably makes sense as the lattice system requires considerably more programming, but
does lots more neat stuff. I still don’t use ggplot2 much, but recognize that it can also
do quite powerful things.

We are going to spend our time today talking about the traditional graphics system.
We will spend some time later talking about what lattice graphs can do and why we
might want to use them, but for now, it’s just the traditional graphics.

3 The Plot Function

For the most part, in the traditional graphics system, graphs are initially made with the
\texttt{plot} function, though there are others, too. Then additional elements can be added as
you see fit.

3.1 getting familiar with the function

Let’s take a look at the help file for the \texttt{plot} command.

\begin{verbatim}
x: the coordinates of points in the plot. Alternatively, a
   single plotting structure, function or \_\_any R object with a
   'plot' method\_ can be provided.

y: the y coordinates of points in the plot, \_\_optional\_ if 'x' is
   an appropriate structure.

\ldots: Arguments to be passed to methods, such as graphical
   parameters (see 'par'). Many methods will accept the
   following arguments:

'\texttt{type}': what type of plot should be drawn. Possible types are
   * '\texttt{p}': for \textit{p}oints,
   * '\texttt{l}': for \textit{l}ines,
   * '\texttt{b}': for \textit{b}oth,
   * '\texttt{c}': for the lines part alone of 'b',
   * '\texttt{o}': for both \textit{e}\textit{o}verplotted,\textit{d},
   * '\texttt{h}': for \textit{e}\textit{h}istogram\textit{l}ike (or \textit{e}\textit{h}igh-density\textit{d})
     vertical lines,
   * '\texttt{s}': for stair \textit{s}teps,
\end{verbatim}
* "S" for other steps, see Details below,
* "n" for no plotting.

All other 'type's give a warning or an error; using, e.g., 'type = "punkte"' being equivalent to 'type = "p"' for S compatibility.

'main' an overall title for the plot: see 'title'.

'sub' a sub title for the plot: see 'title'.

'xlab' a title for the x axis: see 'title'.

'ylab' a title for the y axis: see 'title'.

'asp' the y/x aspect ratio, see 'plot.window'.

Now, we can load the Duncan data again and see how the plotting function works. The two following commands produce the same output within the plotting region, but have different axis labels.

```r
car
plot(prestige ~ income, data=Duncan)
```
There are a couple of interesting features here.

- You can either specify the plot with a formula as the first argument, `y ~ x` or with `x,y` as the first two arguments. The `data` argument only exists when you use the former method. Using the latter method, you have to provide the data in `dataset$variable` format, unless the data are attached.

- Whatever names you have for `x` and `y` will be printed as the x- and y-labels. These can be controlled with the `xlab` and `ylab` commands.

- The plotting symbols, by default are open circles. This can also be controlled with the `pch` option (more on this later).
3.2 Default Plotting Methods

In R, there are default methods for plotting all types of variables. By default method I simply mean that R looks at the context in which you’re asking for a graph and then makes what it thinks is a reasonable graph given the different types of data. All of the figures in 20 were called simply by using the `plot` command. R figured out by the types of variables being used what plot was most appropriate.

```r
# univariate scatterplot
plot(Duncan$income)

# histogram
plot(Duncan$type)

# bivariate scatterplot
plot(income ~ education, data=Duncan)

# two factors mosaic-plot
Duncan$inc.cat <- cut(Duncan$income, 3)
par(las=2, mar=c(5,7,2,3))
plot(inc.cat ~ type, data=Duncan, xlab="", ylab="")
```
# boxplot
plot(income ~ type, data= Duncan, xlab="")

## You try it

1. Load the R data file `strikes_small.rda` that was in your zip file.

2. Look through the variables by using `str` and `searchVarLabels(strikes, '')` (make sure the DAMisc package is loaded to use `searchVarLabels`). Using the data, make examples of all the plots above.
Figure 20: Default Plotting Methods

(a) One Numeric - scatterplot (with index as the x-axis)

(b) One factor - histogram

(c) Two Numeric Variables - scatterplot

(d) Two Factors - mosaic plot

(e) One Numeric, One Factor - boxplot
3.3 Controlling the Plotting Region

There are a number of commands we can use to control the size and features of the plotting region.

- We can control the limits of the x- and y-axis with xlim and ylim, respectively. Here, the limits are specified with a vector of indicating the desired minimum and maximum value of the axis.

  \[
  \text{plot(prestige ~ income, data=Duncan, xlim=c(0,100))}
  \]

3.4 Example of Building a Scatterplot

In R, you can open a plotting window and set its dimensions without actually making the points appear in the space. You can do this by specifying type='n'. You can also remove the axes from the space, by issuing the command axes=F. Finally, you could remove any axis labels by adding the commands xlab='', ylab='. The command, then, would look something like this:

\[
\text{plot(prestige ~ income, data=Duncan, type="n", xlab='', ylab='', axes=F)}
\]

This will open a graphics window that is completely blank. One reason that you may want to do this is to be able to control, more precisely, the elements of the graph. Let’s talk about adding some elements back in.

We can add the points by using the points command. You can see the arguments to the points command by typing help(points). The first two arguments to the command are x and y. You can also change the plotting symbol and the color.

- Plotting Symbols are governed by the pch argument. Below is a description of some common plotting symbols.

  ![Plotting Symbols](image)

- Color - Colors are controlled by the col command. (see help for rainbow, colors). The colors can be specified by col = 'black' or col = 'red'. Colors can also be specified with rgb() for red-green-blue saturation with optional transparency parameter.
• Lines - there are six different line-types which should be specified with the corresponding number:

```
1 2 3 4 5 6
```

• Line-width, controlled with `lwd` defaults to 1, so numbers bigger than 1 are thicker than the default and numbers smaller than 1 are thinner.

• Character Expansion is controlled by `cex`. This controls the size of the plotting symbols. The default is 1. Numbers in the range (0,1) make plotting symbols smaller than the default and values > 1 make the plotting symbols bigger than they would be otherwise.

There are lots of interesting things we can do with these. Let’s start by changing the plotting character and color of points.

```r
plot(prestige ~ income, data=Duncan, pch=16, col="blue",
     xlab='\% males earnings > $3500/year',
     ylab='\% of NORC raters indicating profession as good or excellent',
     main = 'Prestige versus Income
            (Duncan Data)')
```
Above, we simply plotted the points, so only two variables are involved here. What if we wanted to include a third variable? We could include information relating to occupation type by coloring the points different for different types of occupations and perhaps using different plotting symbols. To do this, we would have to specify the `pch` argument differently, but could use the rest of the commands we specified above to make the plot.

```r
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
     xlab='% males earnings > $3500/year',
     ylab='% of NORC raters indicating profession as good or excellent',
     main = 'Prestige versus Income
         (Duncan Data)'
)
```
Basically, what you are doing is you are specifying the plotting character for each point. We could also do this with colors.

cols <- c("blue", "black", "red")
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
     col = cols[as.numeric(Duncan$type)],
     xlab='% males earnings > $3500/year',
     ylab='% of NORC raters indicating profession as good or excellent',
     main = 'Prestige versus Income

     (Duncan Data)')
You’re doing this by allowing our vector of the three colors to be indexed by the occupation type. Nifty, right!?!

You try it

Using the `strikes_small.rda` object, do the following:

1. Plot the log of strike volume (+1) against one of the quantitative variables in the dataset.

2. Make a factor variable out of `sdlab_rep` such that there are three evenly-sized groups. Change the plotting symbols in the graph above based on the values of this variable.

3. Make the colors of the plotting symbols above a function of the factor you created in the last step.

You can also encode quantitative information in color with `colorRampPalette`.

```r
myRamp <- colorRampPalette(c("gray75", "gray25"))
cols <- myRamp(length(unique(Duncan$education)))
pchs <- c(15,16,17)
plot(prestige ~ income, data=Duncan, 
     pch=pchs[as.numeric(Duncan$type)], 
     col = cols[order(Duncan$education)],
     xlab='% males earnings > $3500/year',
     ylab='% of NORC raters indicating profession as good or excellent',
     main = 'Prestige versus Income (Duncan Data)')
```
Next, we can talk about adding elements to the plots.

### 3.4.1 Adding a Legend

We know what the points mean, but we can’t very well expect other people to know this unless we tell them. There is a function called `legend` that allows us to make a legend and stick it in the plot. The `legend` command has a number of arguments that should be specified.

- The first argument is the location of the plot. This can be specified in a number of ways. If you provide x and y coordinate values (with \texttt{x=\#} and \texttt{y=\#}), then this gives the coordinates for the top-left corner of the box containing the legend information. Otherwise, you can specify the location with \texttt{topleft}, \texttt{top}, \texttt{topright}, \texttt{right}, \texttt{bottomright}, \texttt{bottom}, \texttt{bottomleft}, \texttt{left} and \texttt{R} will put the legend near the edge of the plot in this position.

- The `legend` argument gives the text you want to be displayed for each point/line you’re describing.

- The point symbol and color are given by a vector to \texttt{pch} and \texttt{col}.

- If instead of points, you have lines, you can give the argument \texttt{lty} with the different line types being used (more on this later).

- \texttt{inset} gives the fraction of the plot region between the edges of the legend and the box around the plotting region (default is 0).

We can include a legend in our plot as follows:
You try it

Using the strikes_small.rda object, do the following:

1. Add a legend to the plot you made in the previous exercise

3.4.2 Adding a Regression Line

We can add a regression line to the previous graph with abline() which adds a line of specified slope and intercept to the plot.

```r
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
     col = cols[as.numeric(Duncan$type)],
     xlab='\% males earnings > $3500/year',
     ylab='\% of NORC raters indicating profession as good or excellent',
     main = 'Prestige versus Income\n (Duncan Data)')
legend("bottomright", c("Blue-collar", "Professional", "White-collar"),
       pch=1:3, col=cols, inset=.01)
```
If we wanted to add a different regression line for each occupation type, we can do that with three different calls to `abline()`. 

```r
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
     col = cols[as.numeric(Duncan$type)],
     xlab='% males earnings > $3500/year',
     ylab='% of NORC raters indicating profession as good or excellent',
     main = 'Prestige versus Income (Duncan Data)')
legend("bottomright", c("Blue-collar", "Professional", "White-collar"),
       pch=1:3, col=cols, inset=.01)
abline(lm(prestige ~ income, data=Duncan,
          subset=Duncan$type == "bc"), col=cols[1])
abline(lm(prestige ~ income, data=Duncan,
          subset=Duncan$type == "prof"), col=cols[2])
abline(lm(prestige ~ income, data=Duncan,
          subset=Duncan$type == "wc"), col=cols[3])
```
3.4.3 Identifying Points in the Plot

Points in the plot can be identified with the function `identify`. The `identify` will print a label next to points that you click on giving an indication of which point exactly is plotted. The important arguments are:

- Location is given by an x and y variable. These should be the same x and y variables you used to make the plot.

- The labels, given with the `label` option will provide R with text to print by each identified point.

- n gives the number of points we want to identify.

Let’s try to identify the two most outlying points on our graph.

```r
identify(x=Duncan$income, y=Duncan$prestige,
         labels=rownames(Duncan), n=2)
```

You can see here that the minister has more prestige on average than other jobs with similar percentage of males making over $3500 and that the conductor (railroad) has less prestige than its income would suggest.
You don’t have to set `n` to any number in particular, just remember to shut off the identifying if you don’t set it. On a mac, just right-clicking in the plot will turn off the identifying function. On Windows, you can right-click in the plot, then a dialog will pop up asking if you want to stop or continue. Not surprisingly, if you want to stop identifying, just click “stop”. There is another function called `locator()` which will simply return the (x,y) coordinate of the point clicked in the plotting region.

### You try it

Using the `strikes_small.rda` object, do the following:

1. Identify points in the previous plot you made by country and year. Note, you will have to make a variable that is country and year with `paste(strikes$abb, strikes$year, sep:="")`.

### 3.5 Other Plots

There are a few other plots that are both common and useful. I will talk about histograms, bar plots, density estimates and dot plots. Histograms can be done with the `hist()` command:

```r
hist(Duncan$prestige, nclass=10, xlab="Prestige", main="")
```
If you wanted to impose a kernel density estimate over the bars, you could use `lines()` with the `density()` command.

```r
hist(Duncan$prestige, nclass=10, xlab="Prestige", 
     main="", freq=F, col="gray80")
lines(density(Duncan$prestige), col="red", lwd=1.5)
```
You try it

Using the `strikes_small.rda` object, do the following:
1. Make a histogram of strike volume and impose a kernel density estimate.

Bar plots can be made first by aggregating a continuous variable over the values of some factor, then using `barplot`.

```r
bp1 <- by(Duncan$prestige, list(Duncan$type), mean)
par(mar=c(4,10,2,2))
barplot(bp1, horiz=T, las=1)
```

A dot-plot, often showing a set of estimates and confidence bounds arrayed relative to a categorical variable, is relatively common these days. Let’s do this by first making the confidence interval of repression for each region. Then we can make the dot-plot.

```r
library(gmodels)
ag <- aggregate(Duncan$prestige, list(Duncan$type), ci)
ag <- cbind(region = ag[,1], as.data.frame(ag$x))
names(ag) <- c("type", "est", "lower", "upper", "se")
ag
```

```
## type  est  lower  upper  se
## 1 bc 22.76190 14.54327 30.98054 3.939969
## 2 prof 80.44444 73.42991 87.45898 3.324717
## 3 wc 36.66667 24.29104 49.04230 4.814330
```
Now, we can make a dotplot:

```r
tag <- ag[order(ag$est),
dotchart(ag$est, ag$type, xlim=c(10,90),
    lcolor=NA, pch=16, xlab = "Prestige")
segments(ag$lower, 1:3, ag$upper, 1:3)
```

4 ggplots

Personally, I prefer the lattice system because I think Cleveland’s theoretical work is most compelling. However, Deducer makes a very friendly interface to ggplot graphics, so we’ll talk more extensively about those. According to Wickham (2009, p. 3), “a statistical graphic is a mapping from data to aesthetic attributes (color, shape size) of geometric objects (points, lines, bars).” The plot may contain transformations of the data that inform the scale or the coordinate system. Finally, faceting, or grouping, can generate the same plot for different subsets of the data. These components can be combined in various ways to make graphics. The main elements of a ggplot are:

**data/mappings** The data that you want to represent visually.
**geoms** represent the elements you see on the plot (lines, points, shapes, etc...)

**scales** map the values of the data into an aesthetic space through colors, sizes or shapes. They also draw legends or axes that allow users to read original data values from the plot.

**coord** the coordinate system describes how coordinates are mapped to a two-dimensional graph.

**facet** describes how to subset the data.

There are two methods for making plots. The first `qplot()` makes an immediate plot, that is the output from this function is a plot. The `ggplot()` function sets the parameters for the plot, but geoms need to be added to display information. Here are a couple of examples:

```r
library(car)
library(ggplot2)
data(Duncan)
qplot(income, prestige, data=Duncan, shape=type, color=type) +
  theme_bw() + theme(aspect.ratio=1) + scale_shape_manual(values=c(1,2,3))
```

```r
g <- ggplot(data=Duncan, aes(x=income, y=prestige, shape=type, color=type))
g + geom_point() + theme_bw() + theme(aspect.ratio=1) + scale_shape_manual(values=c(1,2,3))
```

Figure 21: GGplots

(a) qplot

(b) ggplot + geom

4.1 Deducer

Now, I want to move us over to Deducer in the JGR GUI. These are the tasks I want us to complete.
4.1.1 Scatterplot

- Load the car and ggplot2 packages.
- Load the Duncan data from the car package.
- Make a scatterplot of prestige (y) by income (x).
- Subset by type (different shapes/colors).
- Add regression line by group.
- Make the same plot as above, but using facets for the different values of type.

4.1.2 Bar Graph

- Make a factor out of prestige such that it is in 4 roughly equally sized groups (i.e., using quantiles).
- Make a bar graph of the new 4-group prestige variable.
- Subset the bar graph by type.

This will allow us the chance to A) investigate the various elements that can be added together and B) see what the code looks like. Deducer will spit out the R code to produce the graph.

You try it

<table>
<thead>
<tr>
<th>Load the strikes_small.rda object into Deducer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Try to recreate some of the plots you made before using Deducer</td>
</tr>
</tbody>
</table>

4.2 Back to Code

ggplot2 is Hadley Wickham's creation based on theoretical work by Leland Wilkinson. The basic idea is that graphs are built in a model-based fashion, where pieces are added or subtracted to or from existing plots. These “pieces” can be elements of the plot (e.g., points, lines, etc...) or they can be other parameters that govern how the plot looks. Some examples will help make sense of this.

4.3 Scatterplots

One way that we can make a scatterplot is by using the ggplot() command and then adding points.

```R
library(car)
library(ggplot2)
p <- ggplot(Duncan, aes(income, prestige))
p + geom_point(shape=1) + theme(aspect.ratio=1)
```
As you can see here, the default theme is to have a light gray background and filled-in black points (which I changed with `shape=1`). To make the plot have a white background, we add a theme statement:
Using the `qplot()` command, we can add colors, too.

```r
qplot(income, prestige, data=Duncan, shape=type, color=type) +
  theme_bw() + theme(aspect.ratio=1)
```
If you don’t like the plotting symbols chosen for you, you can change them as well by adding another element to the plot.

```r
qplot(income, prestige, data=Duncan, shape=type, color=type) +
  theme_bw() + scale_shape_manual(values=c(1,2,3)) +
  theme(aspect.ratio=1)
```

To change the colors back to something more familiar, we would have to first make them, and then add them to the plot.

```r
qplot(income, prestige, data=Duncan, shape=type, color=type) +
  theme_bw() + scale_shape_manual(values=c(1,2,3)) +
  scale_color_manual(values=c("blue", "black", "red")) +
  theme(aspect.ratio=1)
```
We could also add a regression line with the following:

```r
qplot(income, prestige, data=Duncan, shape=type, color=type) +
  theme_bw() + scale_shape_manual(values=c(1,2,3)) +
  scale_color_manual(values=c("blue", "black", "red")) +
  geom_smooth(method="lm", se=F) + theme(aspect.ratio=1)
```

Notice a couple of things here. First, because we defined the shapes and colors as depending on a variable, the regression lines also depend on that variable. So, we don’t have to explicitly say that we want different regression lines in any way other than making the points different shapes/colors relative to some variable. Also, the lines only plot over the range of the data. Adding the `fullrange=T` argument to `geom_smooth()` would change that behavior to something more like `abline()`.

In general, the parameters you can set with respect to colors, plotting symbols, etc... can be captured as below:
4.4 Other Plots

You can also make the whole set of other plots with `ggplot2` as well. Here are a few examples.

### 4.4.1 Histograms and Barplots

```r
p <- ggplot(Duncan, aes(prestige))
p2 <- p + xlim(1,101) + geom_histogram(aes(y = ..density..),
          color="black", fill="gray75", breaks=seq(1,101,by=10)) +
          theme_bw() + theme(aspect.ratio=1)
p2
```
and now with density lines

```r
p2 + geom_density()
```

```r
ag <- aggregate(Duncan$prestige, list(Duncan$type), mean)
names(ag) <- c("type", "prestige")
ggplot(ag, aes(x=type, y=prestige)) + geom_bar(stat="identity") +
  theme_bw() + theme(aspect.ratio=1)
```
### 4.4.2 Dotplot

```r
library(gmodels)
ag <- aggregate(Duncan$prestige, list(Duncan$type), ci)
ag <- cbind(region = ag[,1], as.data.frame(ag$x))
names(ag) <- c("type", "est", "lower", "upper", "se")
ag <- ag[order(ag$est), ]
ag$type <- factor(1:nrow(ag), labels=levels(Duncan$type))
p <- ggplot(ag, aes(est, type))
p + geom_point() + xlab("Mean Prestige") + ylab("") +
   geom_errorbarh(aes(xmin=lower, xmax=upper)) +
   theme(aspect.ratio=1)
```

![Dotplot Diagram]

### 4.5 Faceting

`ggplot2` is also really good with dependent data (like lattice).

```r
p <- ggplot(Duncan, aes(income, prestige))
p2 <- p + geom_point(shape=1) +
   facet_grid(~ type) + theme_bw() +
   theme(aspect.ratio=1)
p2
```
Adding a regression line in each panel is also easy:

\[
p2 + \text{geom\_smooth(} \text{method="lm", se=F, col="gray65", fullrange=T)} + \\
\text{geom\_smooth(} \text{method="lm", se=F)} + \text{theme(aspect\_ratio=1)}
\]

Taking the same example as in the Lattice lecture, we could plot model predictions using \text{ggplot2}.
mod1 <- lm(prestige ~ income + education + type, data=Duncan)
mod2 <- lm(prestige ~ poly(income, 3) + education + type, data=Duncan)

newdat <- data.frame(
  income = seq(min(Duncan$income), max(Duncan$income), length=25),
  education = mean(Duncan$education),
  type = "bc"
)
p1 <- as.data.frame(predict(mod1, newdata=newdat, interval="confidence"))
p2 <- as.data.frame(predict(mod2, newdata=newdat, interval="confidence"))

plot.dat <- as.data.frame(rbind(p1, p2))

plot.dat$model <- rep(c("Model 1", "Model 2"), each=25)
plot.dat$income <- rep(newdat$income, 2)
rownames(plot.dat) <- NULL
plot.dat1 <- plot.dat[which(plot.dat$model == "Model 1"), ]

mp <- ggplot(plot.dat, aes(x=income, y=fit), color=model) + theme_bw()
mp + geom_ribbon(aes(ymin=lwr, ymax=upr, fill=model), alpha=.25) +
  scale_fill_manual("Model", values=c("blue", "red")) +
  geom_line(aes(color=model), show_guide=F) +

In the same plot, we could do something like this:

mp <- ggplot(plot.dat, aes(x=income, y=fit, fill=model)) + theme_bw()
mp + geom_ribbon(aes(ymin=lwr, ymax=upr, fill=model), alpha=.25) +
  geom_line(aes(color=model), show_guide=F) +
scale_colour_manual(values=c("blue", "red")) +
theme(aspect.ratio=1)

## Warning: ‘show_guide’ has been deprecated. Please use ‘show.legend’ instead.
We could also make the histograms of assets by region from the Ornstein data.

```r
p <- ggplot(Ornstein, aes(x=log2(assets))) + theme_bw()
ra <- range(log2(Ornstein$assets))
p + geom_histogram(aes(y=.density..),
    breaks=pretty(ra, 10), color="black", fill="gray75") +
    facet_wrap(~nation) + geom_density() + theme(aspect.ratio=1)
```

![Histograms of assets by region](image)
Side-by-side barplots can be accomplished in a lot the same way as in Lattice.

```r
ag <- with(Chile, aggregate(income,
    list(region, education), mean, na.rm=T))
names(ag) <- c("region", "education", "mean_income")
p <- ggplot(ag, aes(x=region, y=mean_income)) + theme_bw()
p + geom_bar(stat="identity") + facet_grid(.~ education) + theme(aspect.ratio=1)
```
Dotplots by group can happen easily here, too.

```r
ag2 <- with(Chile, aggregate(income,
   list(region, education), ci, na.rm=T))
ag2 <- cbind(ag2[,1:2], ag2$x)
names(ag2) <- c("region", "education", "mean", "lower", "upper", "se")
ag2 <- ag2[order(ag2$education, ag2$mean), ]
ag2$region <- factor(ag2$region,
   levels=ag2$region[which(ag2$education == "PS")])
p <- ggplot(ag2, aes(y=region, x=mean))
p + geom_point() + facet_grid(education ~ .) + theme_bw() +
   geom_errorbarh(aes(xmin=lower, xmax=upper),
   size=.25)
```

5 Effects Package

The `effects` package is one of the most versatile and useful pieces of software for assessing non-linear/interactive effects in linear models or for assessing any effects in non-linear models.

- You can load the package by doing:
  ```r
  library(effects)
  ```
- The main command that does all of the work is `effect()` which has print and plot methods
The function provides predicted values holding other variables at fixed values (along with confidence intervals). The function can do this not only for individual terms, but for terms interacted with others and for non-linear transformations, polynomials, and splines. The great thing about this is that it is built on the lattice package and we can use what we know about lattice to modify it.

5.1 Some Linear Model Examples

5.1.1 Non-linear Transformations in Linear Models

First we have to run a model that has effects we want to plot. We’ll start with a model that has a non-linear transformation.

```r
library(car)
library(effects)
data(Prestige)
mod <- lm(prestige ~ log(income) + education + type, data=Prestige)
eff <- effect("log(income)", mod, default.levels=100)
names(eff)
```

```
## [1] "term"          "formula"
## [3] "response"     "variables"
## [5] "fit"           "x"
## [7] "x.all"        "model.matrix"
## [9] "data"         "discrepancy"
##[11] "offset"       "residuals"
##[13] "partial.residuals.range" "x.var"
##[15] "vcov"          "se"
##[17] "lower"        "upper"
##[19] "confidence.level" "transformation"
##[21] "family"
```

Now, we can use the plotting method for `effect` to plot those predictions. Notice, that I specified two colors as `colors`. The first controls the solid mean-line, the second controls the colors of the confidence bounds. Other graphical parameter arguments can be controlled, but more difficulty with `lty`, `lwd`, etc...

```r
plot(eff, colors=c("black", "black"))
```
Elements of the graph can be updated with the `update()` function. Some elements of the graph need to be “hard-coded” in, that is they are not allowed to be manipulated in call to `plot`. An example will help clarify. What if I wanted to rotate the x-axis labels 45 degrees. I should be able to provide this as an argument to `plot`:

```r
> plot(eff, colors=c("black", "black"), scales=list(x=list(rot=45)))
```

However, we can alter it with `update()`.

```r
p1 <- plot(eff, colors = c("black", "black"))
update(p1, scale = list(x = list(rot=45)))
```
5.1.2 Choosing “other” values of Covariates

The \texttt{effects} package holds the values of other variables constant at “interesting” values. We can choose the values at which those other variables are held constant in a couple of different ways:

- Using \texttt{typical} will allow you to set the values of all variables to something meaningful.

\begin{verbatim}
eff2 <- effect(“log(income)”, mod, 
default.levels=100, typical = median)
head(eff$model.matrix)
## (Intercept) log(income) education typeprof typewc
## 1 7.412160 10.7951 0.3163265 0.2346939
## 2 7.549965 10.7951 0.3163265 0.2346939
## 3 7.671060 10.7951 0.3163265 0.2346939
## 4 7.779061 10.7951 0.3163265 0.2346939
## 5 7.876527 10.7951 0.3163265 0.2346939
## 6 7.965332 10.7951 0.3163265 0.2346939

head(eff2$model.matrix)
## (Intercept) log(income) education typeprof typewc
## 1 7.412160 10.7951 0.3163265 0.2346939
## 2 7.549965 10.7951 0.3163265 0.2346939
\end{verbatim}
What you’ll notice is that the type dummy variables are set to their in-sample proportions, not as zeros or ones. This is problematic for some people, so we can move on to the other way of setting covariate values.

- We can also provide a value for all or some of the columns of the model matrix. We can do this with the `given.values` argument to `effect()`.

```r
eff3 <- effect("log(income)", mod, 
  default.levels=100, 
  given.values=c("typeprof"=1, 
               "typewc" = 0))
head(eff3$model.matrix)
## (Intercept) log(income) education typeprof typewc
## 1 1 7.412160 10.7951 1 0
## 2 1 7.549965 10.7951 1 0
## 3 1 7.671060 10.7951 1 0
## 4 1 7.779061 10.7951 1 0
## 5 1 7.876527 10.7951 1 0
## 6 1 7.965332 10.7951 1 0
```

Note that these values have to have precisely the same names as the model matrix (these are also the same names as the model coefficients).

```r
c coef(mod)
## (Intercept) log(income) education typeprof typewc
## -81.201867 10.487471 3.284486 6.750887 -1.439403
colnames(model.matrix(mod))
## [1] "(Intercept)" "log(income)" "education" "typeprof" "typewc"
```

All of the other variables will be included at typical values.

You try it

Using the `strikes_small.rda` object, do the following:
1. Model the log of strike volume (+1) as a function of the log of population density, unemployment and inflation
2. Plot the effect of the log of population density
5.1.3 Interactions

The \texttt{effects} package can also help us understand interaction effects.

\begin{verbatim}
mod4 <- lm(prestige ~ income*education + type, data=Prestige)
inc_eff <- effect("income*education", mod4, 
given.values=c("typeprof"=1, "typewc"=1))
plot(inc_eff, as.table=T)
\end{verbatim}

This provides us the effect of income as education takes on 10 different values. So the lines tell us how predicted values of prestige change with income for each of the different values of income.

\begin{verbatim}
inc_eff
##
## income*education effect
## education
## income 8 10 12 14
## 5000 35.44788 43.55465 51.66141 59.76818
## 10000 45.97000 51.97489 57.97978 63.98467
## 15000 56.49212 60.39513 64.29814 68.20115
## 20000 67.01424 68.81537 70.61651 72.41764
## 25000 77.53636 77.23562 76.93487 76.63412
\end{verbatim}

We could plot the other side of the interaction - the effect of education as income takes on different values:

\begin{verbatim}
plot(inc_eff, x.var="education", as.table=T)
\end{verbatim}
The `effects` package also does “the right” thing with factors and continuous variables interacted:

```r
mod5 <- lm(prestige ~ income*type + education, data=Prestige)
eff_cf <- effect("income*type", mod5)
pcf1 <- plot(eff_cf, layout=c(3,1))
pcf1
```
We can also plot the other side of the interaction, too:

```r
func2 <- effect("income*type", mod5,
               xlevels=list(income = quantile(Prestige$income, c(.1,.5,.9))))
pcf2 <- plot(func2, layout=c(3,1), x.var="type")
pcf2
```

You try it

Using the `strikes_small.rda` object, do the following:

1. Model the log of strike volume (+1) as a function of the log of population density, unemployment and inflation and the interaction of log population density and inflation.

2. Figure out what is going on in the interaction.

### 5.2 GLM Examples

The `effects` package is also good at plotting predicted probabilities (or other inverse-link transformations) from GLMs, for one variable holding all other variables constant at some value. Let’s run a logit model for vote for a democrat from the 1992 ANES.

```r
library(foreign)
nenes <- read.dta("anes1992.dta")
mod <- glm(votedem ~ age + pid +
           black + retnat, data=nes, family=binomial)
func <- effect("pid", mod,
               default.levels=25,
               given.values = c(
                 "pid" = 4,
                 "black" = 0,
                 "retnatsame" = 0,
                 "retnatworse" = 1))
plot(func)
```
Note that in the default version of the graph, we get what looks like a straight line. This is because the effect is plotted on the linear predictor and you see that we get unequal distances between tick-marks on the y-axis. We can change this behavior with rescale.axis=F.

```r
plot(eff, rescale.axis=F)
```

## NOTE: the rescale.axis argument is deprecated; use type instead
5.3 OL/MNL Examples

You can plot the predictions from Ordered logit/probit and MNL as well. Here, you have a couple more options.

```r
fh <- read.dta("fh2006_ss.dta")
library(MASS)
mod <- polr(freedomstat ~ log(pop) + gdppc10k + civ, data=fh)
summary(mod)

## Re-fitting to get Hessian

Call:
polr(formula = freedomstat ~ log(pop) + gdppc10k + civ, data = fh)

Coefficients:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
<th>Std. Error</th>
<th>t value</th>
</tr>
</thead>
<tbody>
<tr>
<td>log(pop)</td>
<td>-0.2556</td>
<td>0.09261</td>
<td>-2.7596</td>
</tr>
<tr>
<td>gdppc10k</td>
<td>0.5394</td>
<td>0.24689</td>
<td>2.1848</td>
</tr>
<tr>
<td>civIslamic</td>
<td>-1.3684</td>
<td>0.52840</td>
<td>-2.5897</td>
</tr>
<tr>
<td>civLatin American</td>
<td>1.4775</td>
<td>0.57628</td>
<td>2.5638</td>
</tr>
<tr>
<td>civOrthodox</td>
<td>0.2807</td>
<td>0.66007</td>
<td>0.4253</td>
</tr>
<tr>
<td>civOther</td>
<td>0.8716</td>
<td>0.49292</td>
<td>1.7681</td>
</tr>
<tr>
<td>civWestern</td>
<td>4.1342</td>
<td>1.16519</td>
<td>3.5481</td>
</tr>
</tbody>
</table>

Intercepts:

<table>
<thead>
<tr>
<th>Value</th>
<th>Std. Error</th>
<th>t value</th>
</tr>
</thead>
<tbody>
<tr>
<td>not free</td>
<td>partly free</td>
<td>-2.5367</td>
</tr>
<tr>
<td>partly free</td>
<td>free</td>
<td>-0.4965</td>
</tr>
</tbody>
</table>

Residual Deviance: 264.0614
AIC: 282.0614

We can use the \texttt{effects} package to get fitted probabilities for continuous covariates. For example:

```r
gdp.eff <- effect("gdppc10k", mod, 
                  default.levels=100)

## Re-fitting to get Hessian
plot(gdp.eff, rug=F)
```

58
You can also use the style='stacked' argument if you want to get something a bit different:

```r
plot(gdp.eff, xlab="GDP/capita (in $10000 US)",
     ylab = "Pr(Freedom Status)", style="stacked",
     colors=c("gray75", "gray50", "gray25"),
)
```

These work the same way for the multinomial logit, too.
6 Maps

One of the real benefits on R is that it is relatively easy to make good looking maps. You’ll need a shape file (and the associated auxiliary files) and potentially some extra data you want to plot. Generally, when plotting states, countries, counties, or any sort of administrative boundary, you read the data in as `SpatialPolygonsDataFrame`. All of the objects we have dealt with until now have been S3 objects. While this has many and varied implications, the most important for us is that we can list the names of the elements in the object with `names(object)` and once we find an element (let’s call it E), we could extract that element with `object$E`. With S4 objects, the analog to `names` is `slotNames(object)` and to extract element E, you would use `object@E`. So, below, where you see the code `world@data`, that is extracting the data frame associated with the polygons. Every polygon (country here) has an associated line in the data frame.

```r
library(rgdal)
library(ggplot2)
library(plyr)
world <- readOGR(".", layer="WorldCountries")

## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "WorldCountries"
## with 250 features
## It has 3 fields

wbmig <- read.csv("wbmig_R.csv")
wbmig <- na.omit(wbmig[,c("wbpropmig", "ccode")])
world@data <- join(world@data, wbmig, type="left")

## Joining by: ccode

q <- quantile(world@data$wbpropmig, c(0,.25,.5,.75,1), na.rm=T)
world@data$propmigcat <- cut(world@data$wbpropmig, breaks=q)
world@data$id <- world@data$placename

world.points <- fortify(world, region="id")
world.df <- join(world.points, world@data, by="id")
pl <- ggplot(world.df) +
aes(long,lat,group=group,fill=propmigcat) +
  geom_polygon() +
  geom_path(color="gray50", size=.1) +
  coord_equal() +
  scale_fill_brewer("", labels=c("Lowest Quartile", "2nd Quartile", "3rd Quartile", "Highest Quartile"))

pl + theme(line = element_blank(),
          axis.title.x = element_blank(),
          )
```
R also has lots of spatial statistics routines. One simple thing that you might have to do is create an adjacency matrix. You could do this as follows:

```r
library(spdep)
neighbors <- poly2nb(world)
adjmat <- nb2mat(neighbors, zero.policy=TRUE)
```

The adjacency matrix sums to 1 (so each entry is equal to \( \frac{1}{\text{#neighbors}} \)) in the rows (unless the polygon has no neighbors in which case it is all zeros). You can create a neighborhood average with:

```r
x <- ifelse(is.na(world@data$wbpropmig), 0, world@data$wbpropmig)
neighave <- adjmat %*% x
```

Now, the neighborhood average is a variable you could use in a model (to account for some degree of spatial dependence).

## 7 Interactivity

There are a few different means of generating interactive graphics in R depending on what you want. The following are interactive graphical options that don’t require you to write a function to do the graphing.

- `identify` and `locator` are simple, both in their implementation and capabilities. They make it easy to identify points in a plot. `locator` identifies any (x,y) coordinate you click with the mouse and `identify` uses the `locator` function to try to identify plotted points close to where you clicked the mouse. They are also automatically labeled.
iPlots offers linking and brushing of lots of univariate and bivariate plots (including scatterplots, histograms, bar plots). We will work through some of these examples in the workshop, but interactivity is hard to describe and display in a static handout like this one.

RGL offers real-time 3D visualization in R (where openGL does the heavy lifting). After we talk about writing functions, we’ll talk about some function-based approaches to interactive graphics. These include:

- **rpanel** - a package that makes it easy to re-render graphics based on sliders, check boxes, text, pull-down menus, etc... These are great tools for demonstration, but are hard to allow others arbitrary access. In that, they are not something you would host on a server and have other people play with.

- **shiny** - a web-based interactive visualization environment for R. Shiny is designed for getting interactive visualizations to the web. This is a competitor to the Tableau software. This is a great solution for small projects (i.e., those that you won’t need thousands of people to see and use, many of them concurrently). Thus, it is a great tool for academics.

- **rCharts** - an set of helper functions in R to produce D3.js graphics. These can be easily hosted on a website, though they do require you to have some place to host them. The interactivity is often more pointed or targeted here than it is with a shiny app, but it scales a lot more inexpensively.