Creatings Plots & Graphs in R

Joshua Wiley

Health Psychology Program
University of California, Los Angeles
wileyj@ucla.edu

Thursday, 26 April, 2012

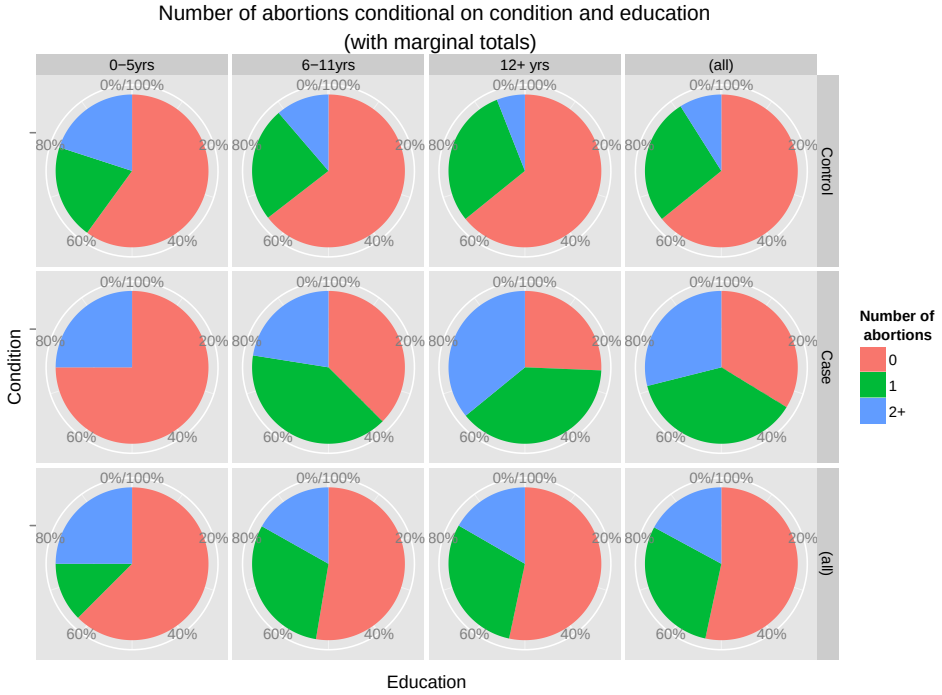


Table of Contents

- This presentation gives a *very* brief introduction to R. The point is just to give enough of a sense that the rest of the presentation makes some sense, not that you know R after this.
- Most time is spent describing and showing examples from the ggplot2 package for graphics, emphasizing bar charts, pie charts, and chloropleth maps
- I sneak in a few other graphs because I could not resist, some cool ones in ggplot2 and heatmaps.

Getting Started I

R is composed of a core set of packages that provide infrastructure and basic functionality and over 3,671 packages. Most regular R users use some interface. Popular ones include Rstudio (which I will demonstrate here) and Emacs + ESS. These provide features to make working with R easier and more efficient.

R http://www.r-project.org/

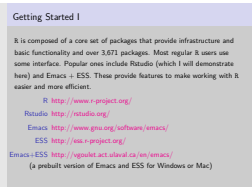
Rstudio http://rstudio.org/

Emacs http://www.gnu.org/software/emacs/

ESS http://ess.r-project.org/

Emacs+ESS http://vgoulet.act.ulaval.ca/en/emacs/

(a prebuilt version of Emacs and ESS for Windows or Mac)



- Core developers write and maintain core packages which include packages for the most common, basic tasks, and the R interpreter
- Thousands of community written packages extend this core to include and amazing among of flexibility
- Rstudio is a popular Integrated Development Environment (IDE) for R. Along similar lines is Eclipse + StatEt.
- Also popular is Emacs (a powerful text editor) + ESS (emacs speaks statistics), an Emacs add on that makes it "R aware"—automatically submit lines of code, get results, syntax highlighting, etc.

Getting Started II

Tutorial 1 http://www.burns-stat.com/pages/tutorials.html

read the tutorials for R beginning (∼ 1 day)

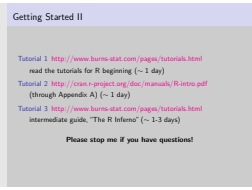
Tutorial 2 http://cran.r-project.org/doc/manuals/R-intro.pdf

(through Appendix A) (∼ 1 day)

Tutorial 3 http://www.burns-stat.com/pages/tutorials.html

intermediate guide, "The R Inferno" (∼ 1-3 days)

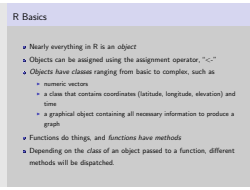
Please stop me if you have questions!



- When I say approximately 1 day, I mean 8-10 hours of work, but this is assuming that you are installing R, and actually playing with and trying code as you read
- Patrick Burns is a statistician who has written several popular tutorials
- The introduction to R manual is the official introductory manual
- I am guessing to thoroughly go through all of these would take about a week, after that you should have enough background to do basic things and learn on your own. For example, be able to read the documentation to learn a new function, and you will have the basic foundation to interact with other R users, ask and understand questions.

R Basics

- Nearly everything in R is an *object*
- Objects can be assigned using the assignment operator, "<-"
- *Objects have classes* ranging from basic to complex, such as
  - numeric vectors
  - a class that contains coordinates (latitude, longitude, elevation) and time
  - a graphical object containing all necessary information to produce a graph
- Functions do things, and *functions have methods*
- Depending on the *class* of an object passed to a function, different methods will be dispatched.

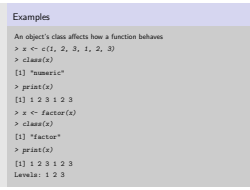


- This is all covered in the introductory manual, also see:
- *Introductory Statistics with R* by Peter Dalgaard (one of the core developers)
- *Using R for Introductory Statistics* by John Verzani
- `ggplot2, lattice, grid` tend to create graphical objects more than base graphics can

Examples

An object's class affects how a function behaves

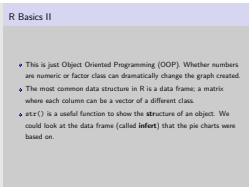
```
> x <- c(1, 2, 3, 1, 2, 3)
> class(x)
[1] "numeric"
> print(x)
[1] 1 2 3 1 2 3
> x <- factor(x)
> class(x)
[1] "factor"
> print(x)
[1] 1 2 3 1 2 3
Levels: 1 2 3
```

- A factor is essentially just indicating categorical data
- Compared with character data, factors can be ordered or have levels so that there can be a reference group. In statistical models, factors are usually automatically dummy coded.
- Depending on the class of the object, x, the behavior of `print` changes (it adds the levels info for factors)

R Basics II

- This is just Object Oriented Programming (OOP). Whether numbers are numeric or factor class can dramatically change the graph created.
- The most common data structure in R is a data frame; a matrix where each column can be a vector of a different class.
- `str()` is a useful function to show the **str**ucture of an object. We could look at the data frame (called **infert**) that the pie charts were based on.

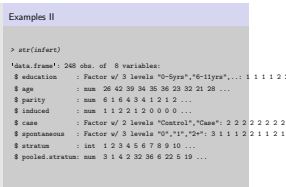


- Not only are data frames the most common, many functions create data frames by default and many others expect data frames
- For example, `read.table, read.csv` create data frames; `ggplot` the base function in the `ggplot2` package expects a data frame input.
- Beginners often run into difficulties and sometimes strange error messages because data are not in the correct format. `str` can help with this because it can show that a column you think is numeric is actually character. Perhaps on further investigation, "." were used for missing values, but not declared as such when using `read.table` to read the data in, so `R` treated the entire column as character.

Examples II

```
> str(infert)

'data.frame': 248 obs. of  8 variables:
 $ education     : Factor w/ 3 levels "0-5yrs","6-11yrs",..: 1 1 1 1 2 2 2 2 2 2 ...
 $ age           : num  26 42 39 34 35 36 23 32 21 28 ...
 $ parity        : num  6 1 6 4 3 4 1 2 1 2 ...
 $ induced       : num  1 1 2 2 1 2 0 0 0 0 ...
 $ case          : Factor w/ 2 levels "Control","Case": 2 2 2 2 2 2 2 2 2 2 ...
 $ spontaneous   : Factor w/ 3 levels "0","1","2+": 3 1 1 1 2 2 1 1 2 1 ...
 $ stratum       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ pooled.stratum: num  3 1 4 2 32 36 6 22 5 19 ...
```

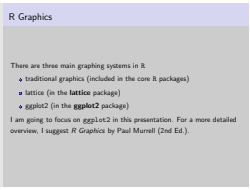
- Here you see that **infert** is a data frame where the observations are the number of rows and the variables the number of columns
- It also shows the name of every column (variable) in the data frame, the class of that particular variable, and a few example values from the first couple rows
- For factors it displays the number of levels, `num = numeric, int = integer, char = character`
- The "$" can be used to access a particular variable. For example, "infert$age" would point to the variable, age, in the dataset infert.

R Graphics

There are three main graphing systems in R

- traditional graphics (included in the core R packages)
- lattice (in the **lattice** package)
- ggplot2 (in the **ggplot2** package)

I am going to focus on ggplot2 in this presentation. For a more detailed overview, I suggest *R Graphics* by Paul Murrell (2nd Ed.).



- I call these "systems" because each one is heavily used and has other user written packages that extend their functionality. grid could be added to this list as it is the underlying framework used by both lattice and ggplot2, but it is low level, and many users use ggplot2 without ever directly using grid.
- *R Graphics* provides details about how and why these systems work. For a "cookbook", I would just search online. Many, many blogs and websites have numerous examples of graphs created in R. There are also some purchasable graphics cookbooks.

Table of Contents

- grammar of graphics plot 2 by Hadley Wickham
- Hadley Wickham is a statistician at Rice University
- ggplot2 has an active ($\sim$ 20 emails per day) support listserv and now developer listserv

ggplot2 background I

- Before we dive into examples of graphics in ggplot2, I want to give a bit of background.
- ggplot2 is based on the **g**rammar of **g**raphics, a framework for creating graphics.
- The idea is that graphics or data visualization generally can be broken down into basic low level pieces and then combined, like language, into a final product.

ggplot2 background I
- Before we dive into examples of graphics in ggplot2, I want to give a bit of background.
- ggplot2 is based on the grammar of graphics, a framework for creating graphics.
- The idea is that graphics or data visualization generally can be broken down into basic low level pieces and then combined, like language, into a final product.

- *Grammar of Graphics* is actually a theory (and a book) by Leland Wilkinson
- It is used now in IBM SPSS in what they call GPL or Graphics Processing Language
- Because it is very different from how many people think about plots and graphs, it can be tricky at first, but after learning it, I find it a very natural, flexible way to express data as graphics

ggplot2 background II

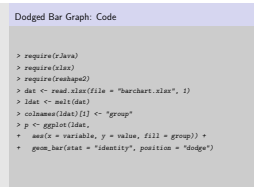
- For example, this sentence is valid: "This is a lamp." but the sentence can be modified without recreating it: "This is a red lamp.".
- This powerful concept allows basic aspects of graphs to be reused and slight modifications added rather than starting from scratch each time.
- Under this system, line plots and scatter plots are essentially the same. Both have data mapped to the x and y axes. The difference is the plotting symbol (called a **geom** in ggplot2) is a point or line. The data, axes, labels, titles, etc. can be identical in both cases.

ggplot2 background II
- For example, this sentence is valid: "This is a lamp." but the sentence can be modified without recreating it: "This is a red lamp.".
- This powerful concept allows basic aspects of graphs to be reused and slight modifications added rather than starting from scratch each time.
- Under this system, line plots and scatter plots are essentially the same. Both have data mapped to the x and y axes. The difference is the plotting symbol (called a geom in ggplot2) is a point or line. The data, axes, labels, titles, etc. can be identical in both cases.

- I will emphasize this theme of how plots are the same and where they differ throughout this presentation
- Hopefully this will make the grammar of graphics (and thus ggplot2) make more sense
- also, I believe it is a helpful way to conceptualize *any* graphic, even if it is not being made in a system that uses the grammar

Dodged Bar Graph: Code

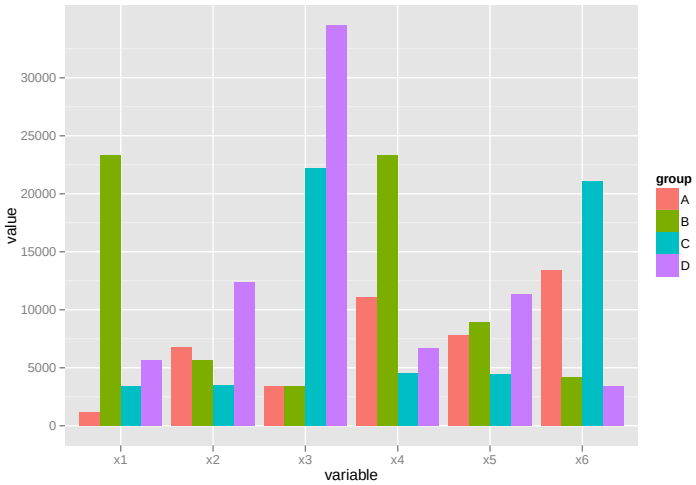
```
> require(rJava)
> require(xlsx)
> require(reshape2)
> dat <- read.xlsx(file = "barchart.xlsx", 1)
> ldat <- melt(dat)
> colnames(ldat)[1] <- "group"
> p <- ggplot(ldat,
+   aes(x = variable, y = value, fill = group)) +
+   geom_bar(stat = "identity", position = "dodge")
```

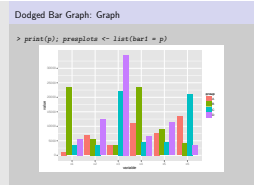



- First load the `rJava` and `xlsx` packages to read in .xlsx files, and the `reshape2` package for the `melt` function, which melts data from wide to long. `ggplot2` generally uses long data, so after reading the excel sheet in, we convert it to long, and then change the first column name to "group"
- to make the graph, call the `ggplot` function, which sets up the "base" of the graph. The data frame containing the variables comes first, `ldat` (for long dat), then the variables are mapped to aspects of the graph using `aes`.
- on the x axis is "variable" on the y axis is the "value" and the fill is "group"
- so far, there is no graph, as there is no geometric representation of our mappings; by adding + `geom_bar()`, the abstract mapping is graphed using bars. The x position of each bar is the variable, the height is the value, and the inside of the bars are filled with different colours based on group.
- a few additional options to `geom_bar` complete it. To make the bars be side by side, we "dodge" their positions

Dodged Bar Graph: Graph

```
> print(p); presplots <- list(bar1 = p)
```



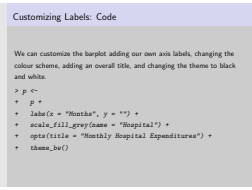


- we saw how to create the graphical object (called a grob) previously, but nothing was *rendered*
- in a typical interactive session, the graph would be rendered to a graphics device automatically, but here I am writing a script that includes R and LaTeXcode to create a presentation.
- you can render a saved grob by `printing` it.
- the semi colon is just to allow two commands to be put on the same line, we save the grob as "bar1" into a list (which can eventually contain many grobs) and store the list in the object `presplots`. This is because we will be reusing some of these later.

Customizing Labels: Code

We can customize the barplot adding our own axis labels, changing the colour scheme, adding an overall title, and changing the theme to black and white.

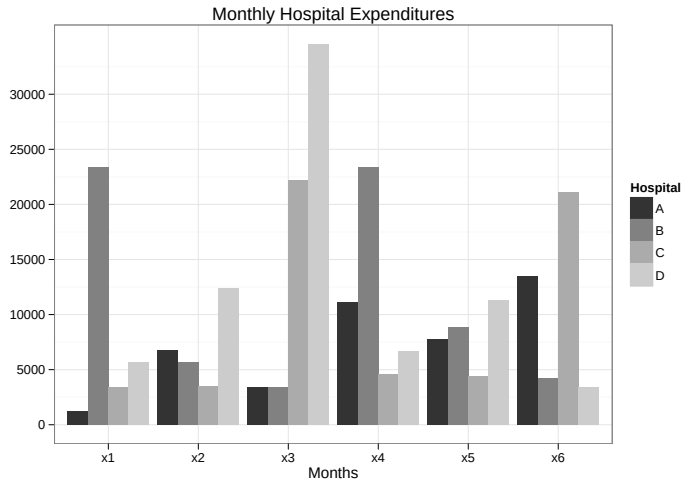
```
> p <-
+   p +
+   labs(x = "Months", y = "") +
+   scale_fill_grey(name = "Hospital") +
+   opts(title = "Monthly Hospital Expenditures") +
+   theme_bw()
```

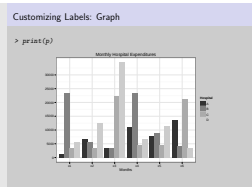



- One implication of the grammar of graphics is you can flexibly extending existing plots (just like sentences) by adding more.
- We previously made a dodged bar graph and stored the grob in `p`.
- We extend it by setting our own x- and y-axis labels using `labs`, controlling the color of the bars by using a grey scale fill with `scale_fill_grey`, adding an overall title using `opts`, and setting the theme to black and white (rather than the greyish background default) using `theme_bw`
- we again save the old grob plus our additions into the object `p`

Customizing Labels: Graph

```
> print(p)
```



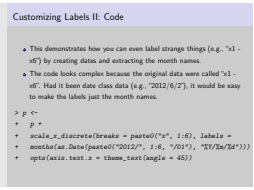


- We can see our title at the top center, x axis label at bottom center, by putting an empty character for y axis label, it is empty, and the bars are now grey scale (as is the legend).
- the black and white theme means that the plot background is white instead of the default grey
- by the way, the default grey was chosen theoretically—articles or books tend to be a mix of white paper and black text for an overall greyish appearance. This makes plots "jump out" if they are on a stark white background. That grey was designed to make them flow with the text more.

Customizing Labels II: Code

- This demonstrates how you can even label strange things (e.g., "x1 - x6") by creating dates and extracting the month names.
- The code looks complex because the original data were called "x1 - x6". Had it been date class data (e.g., "2012/6/2"), it would be easy to make the labels just the month names.

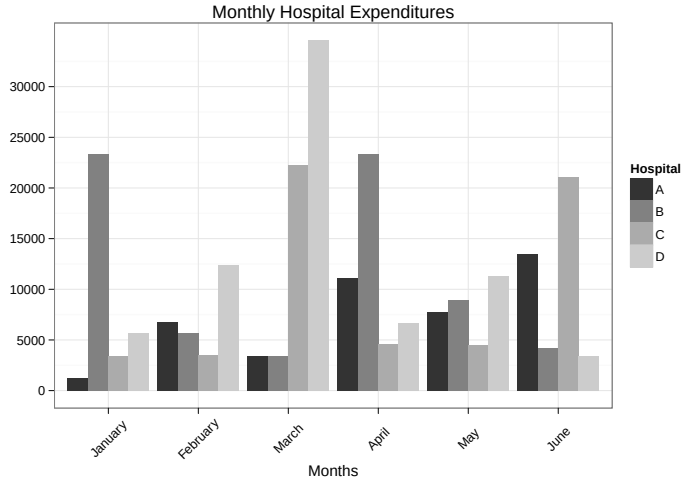
```
> p <-
+   p +
+   scale_x_discrete(breaks = paste0("x", 1:6), labels =
+   months(as.Date(paste0("2012/", 1:6, "/01"), "%Y/%m/%d"))) +
+   opts(axis.text.x = theme_text(angle = 45))
```

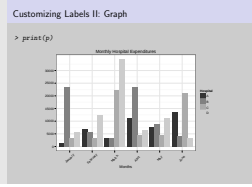



- Now we are going to take the previous graph a few steps farther, by using month names (January through June) for the tick labels rather than "x1 - x6". This is done using `scale_x_discrete`. `scale_x_` could take either `continuous` or `discrete`. We use `discrete` because the original values were discrete "x1". Had they been numeric, "1 - 6", we would have used `continuous`.
- We also rotate the tick labels to make them fit better and be more aesthetically pleasing for their length.

Customizing Labels II: Graph

```
> print(p)
```





- and here as before is the result rendering the updated grob

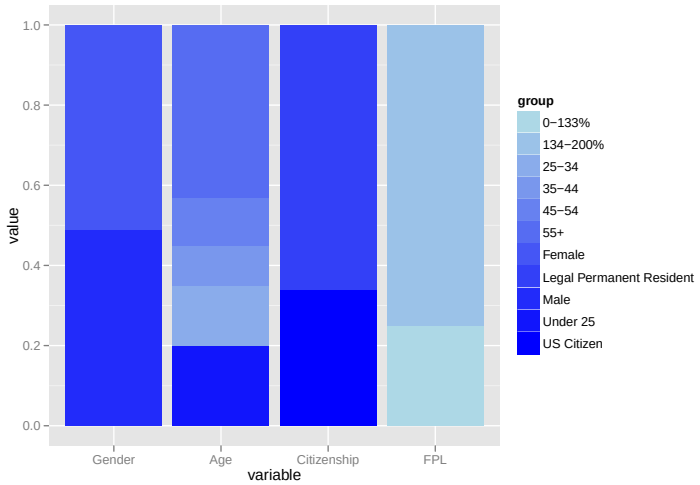
Stacked Bar: Code

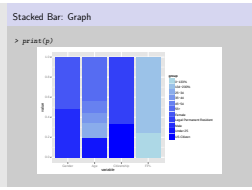
```
> dat2 <- read.xlsx(file = "barchart2.xlsx", 1)
> ldat2 <- na.omit(melt(dat2))
> colnames(ldat2)[1] <- "group"
> cols <- colorRamp(c("lightblue", "blue"))(
+   seq(0, 1, length.out = 11))
> cols <- rgb(cols[, 1], cols[, 2], cols[, 3],
+   maxColorValue = 255)
> p <- ggplot(ldat2,
+   aes(x = variable, y = value, fill = group)) +
+   geom_bar(stat = "identity", position = "stack") +
+   scale_fill_manual(values = cols)
```


- This new bar graph will be stacked instead of dodged. As before we read the data from Excel, convert to long, and rename the first column to "group".
- `na.omit` is used to drop missing (NA) values
- We can setup our own colour scheme using the `colorRamp` function to create a linear interpolation from lightblue to blue. Then we extract 11 equidistant colours from this continuous "ramp". Then use the `rgb` function to convert the RGB colours from `colorRamp` to hexadecimal for `ggplot2`.
- The data frame is `ldat2`, "variable" is on the x axis, "value" is on the y axis, "group" is the fill
- The plotting object/shape is a bar, but this time, we use the "stack" position rather than dodged.
- We can use `scale_fill_manual` to override the default colour scheme with our vector of colours, `cols`.

Stacked Bar: Graph

```
> print(p)
```



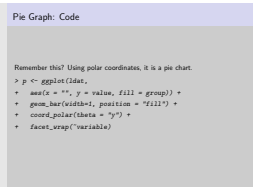


- When the colours are manually set, both the plot and the legend colours are updated so they match.
- This is a bit difficult to read because with 11 different shades from light blue to blue, the contrast between each one is small. The legend is also rather large. one option could be to place the text labels in the graph itself. Another option could be to use more disparate colours, so the contrast is higher and it is easier to tell them apart.

Pie Graph: Code

Remember this? Using polar coordinates, it is a pie chart.

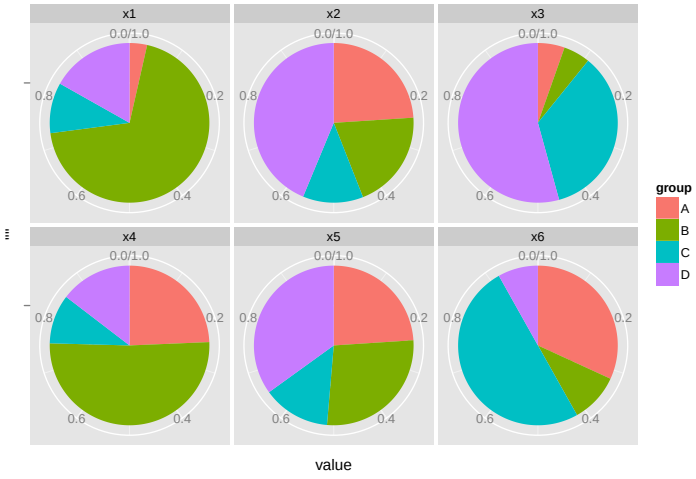
```
> p <- ggplot(ldat,
+   aes(x = "", y = value, fill = group)) +
+   geom_bar(width=1, position = "fill") +
+   coord_polar(theta = "y") +
+   facet_wrap(~variable)
```

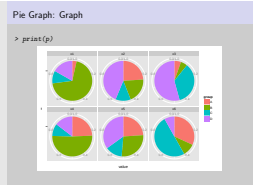



- the pie graph does not really need an "x" variable, but it does need something there, so I just used quotes
- for the bars, we used "stat = "identity"", but here we do not because each pie graph needs to sum to 1, so a transformation is silently done to percentages
- By using polar coordinates, the filled bar plots are wrapped in a circle, creating a pie
- `facet_wrap` breaks it down by each level of variable to give us six nice pie graphs

Pie Graph: Graph

```
> print(p)
```

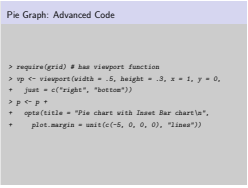




- Each pie graph is essentially a filled bar wrapped around a circle
- the faceting creates six pie graphs, so that each one corresponds to a bar from the bar graph

Pie Graph: Advanced Code

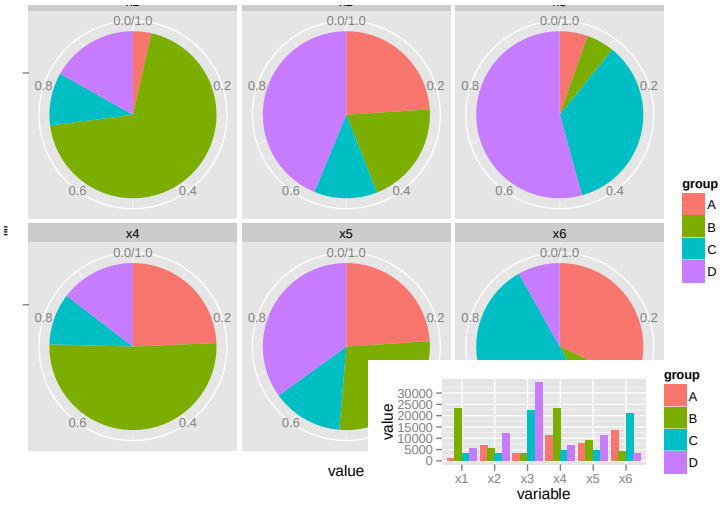
```
> require(grid) # has viewport function
> vp <- viewport(width = .5, height = .3, x = 1, y = 0,
+   just = c("right", "bottom"))
> p <- p +
+   opts(title = "Pie chart with Inset Bar chart\n",
+     plot.margin = unit(c(-5, 0, 0, 0), "lines"))
```

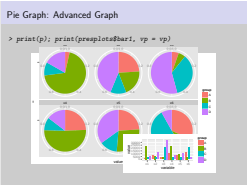



- With a bit of extra work, we can add the bar chart to the pie chart
- first we create a viewport that is located at x = 1, y = 0, and is right bottom justified
- x = 1 and y = 0 are in figure coordinates, that is the figure is scaled to a [0, 1] region, so these represent the bottom right corner
- then we tweak the pie graphs just a bit by adding a title and adjusting the margin
- the last step occurs next, during printing

Pie Graph: Advanced Graph

```
> print(p); print(presplots$bar1, vp = vp)
```





- We have printed graphical objects (grobs) before using the `print(p)` idiom
- Now there is a second call to print, this time using the bar chart we saved from awhile back and using the viewport we made. This causes it to print the graph, but rather than printing it in the whole figure, it just prints it in the little region we setup with the viewport.

Line Chart: Code

Here we are going to look at line and scatter plots. We create a base plot, but do not add shapes (lines or points, so we can demonstrate them.

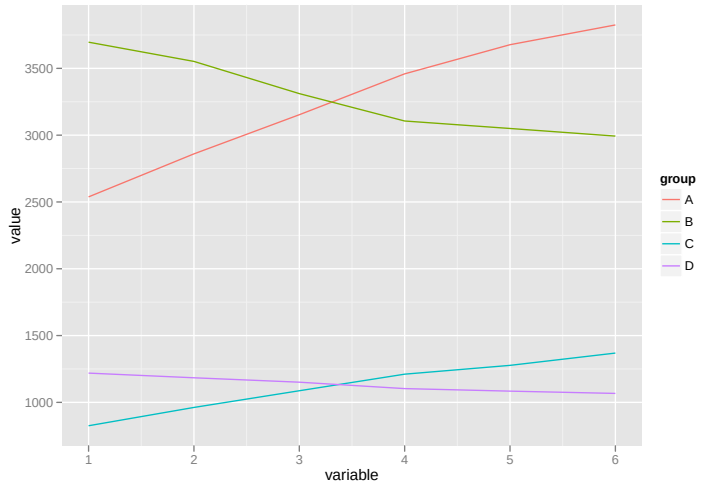
```
> dat3 <- read.xlsx(file = "linechart.xlsx",
+   1, header = FALSE)
> ldat3 <- melt(dat3)
> colnames(ldat3)[1] <- "group"
> ldat3$variable <- as.numeric(ldat3$variable)
> p <- ggplot(ldat3, aes(x = variable, y = value,
+   colour = group))
```


Line Chart: Code

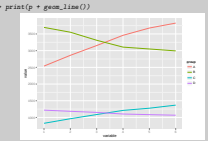
- The setup here is the same as before, read in data, make it long, and rename.
- We force the variable to be numeric using the `as.numeric` function
- The plot is almost exactly like everything else we have seen—we map some variable to x, y, but instead of trying to "fill" lines, we "colour" them by group. Then the geometric object we will use is `geom_line` instead of `geom_bar`.

Line Chart: Graph

```
> print(p + geom_line())
```



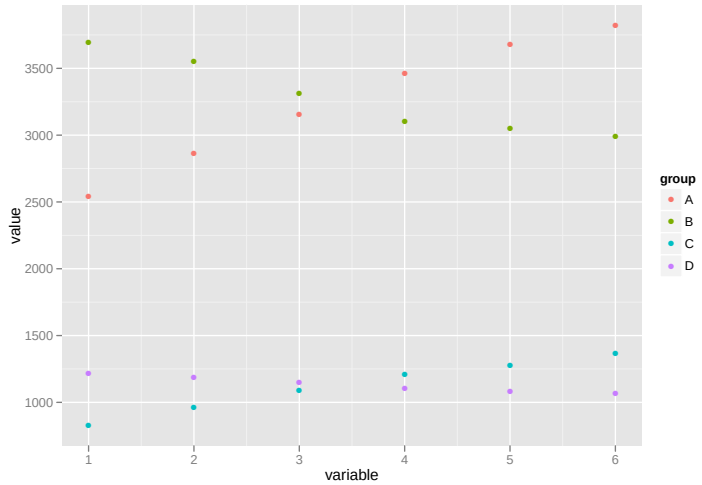
Line Chart: Graph

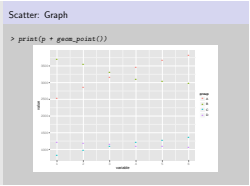


- Now we have a line plot, with separate lines for each group, each line coloured separately, and a legend automatically made.
- We are actually going to try several different geometric objects, just to show the options and how they work

Scatter: Graph

```
> print(p + geom_point())
```

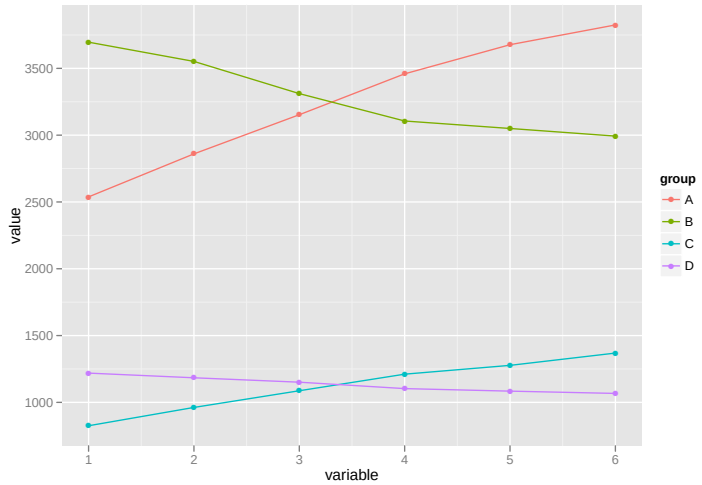


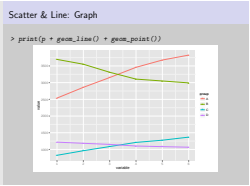


- Now we have a scatter plot, with separate point colours for each group, and a legend automatically made.

Scatter & Line: Graph

```
> print(p + geom_line() + geom_point())
```

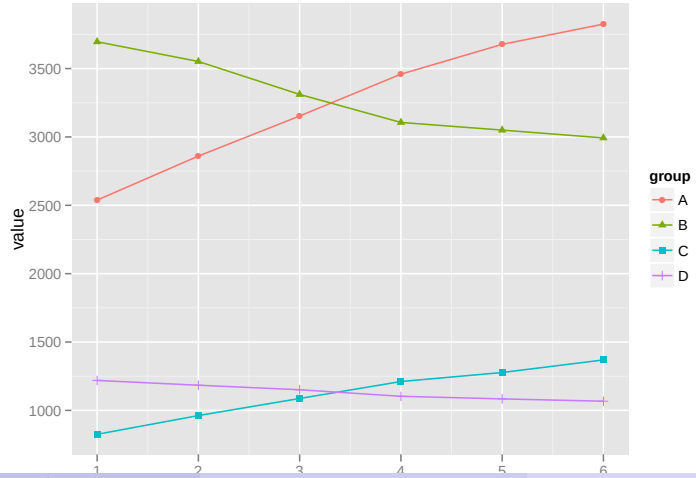


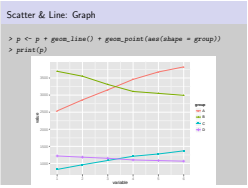


- Now we have a scatter plot and a line plot, just by adding both geometric shapes.

Scatter & Line: Graph

```
> p <- p + geom_line() + geom_point(aes(shape = group))
> print(p)
```

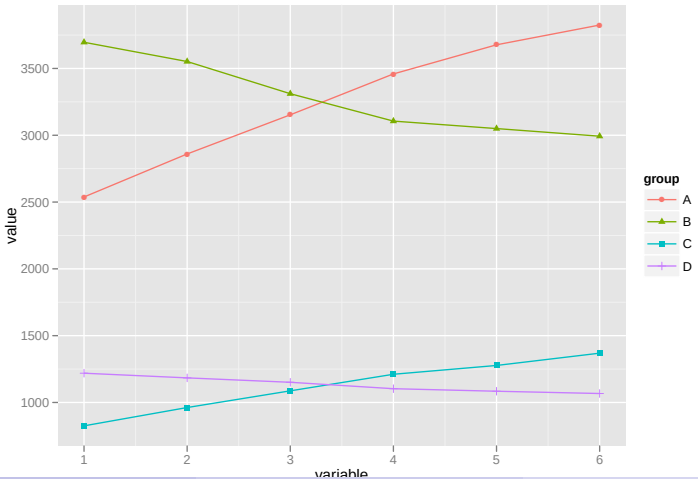


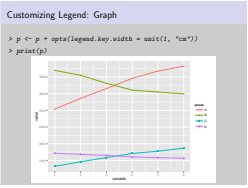


- Now in addition to separate colours, we have different shapes.

Customizing Legend: Graph

```
> p <- p + opts(legend.key.width = unit(1, "cm"))
> print(p)
```





- We can alter the legend (e.g., to make it wider and more easily visible) by adding an option setting the width.

Chloropleth Map: Data Setup I

```
> require(maps)
> require(mapproj)
> states <- map_data("state")[,
+   c("long", "lat", "group", "order", "region")]
> colnames(states)[5] <- "state"
> ## original data source
> ## dat <- read.csv("http://www.census.gov/popest/data/
> ##  national/totals/2011/files/NST_EST2011_ALLDATA.csv")
>
> dat <- read.csv("popdata.csv")
```


Chloropleth Map: Data Setup I

```
> require(maps)
> require(mapproj)
> states <- map_data("state")[,
+   c("long", "lat", "group", "order", "region")]
> colnames(states)[5] <- "state"
> ## original data source
> ## dat <- read.csv("http://www.census.gov/popest/data/
> ##  national/totals/2011/files/NST_EST2011_ALLDATA.csv")
>
> dat <- read.csv("popdata.csv")
```

- Now we are going to go in a bit of a different direction and look at how to create maps and plots of geospatial data on a map.
- first we will load some packages, `maps` and `mapproj`, `mapproj` allows for different projects of coordinates into a two dimensional space.
- We also need to get some data, from the `maps` package, we can get some state data (to create a map of the US with state borders) and from the US census, we will get birth and population data.

Chloropleth Map: Data Setup II

```
> births <- with(dat, {
+   data.frame(state = tolower(NAME),
+   kbirths2011 = BIRTHS2011/1000,
+   mpop2010 = CENSUS2010POP/1000000)
+ })
> births <- subset(births, state %in% states$state)
```

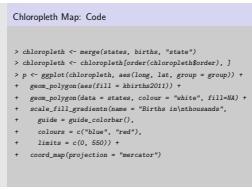

Chloropleth Map: Data Setup II

```
> births <- with(dat, {
+   data.frame(state = tolower(NAME),
+   kbirths2011 = BIRTHS2011/1000,
+   mpop2010 = CENSUS2010POP/1000000)
+ })
> births <- subset(births, state %in% states$state)
```

- Before the data are ready to use, we need to manipulate them some. We start with the census data converting the state names to lower case to match with our geospatial state data.
- we also convert births to births in thousands and population to millions
- the results are saved in a new data frame, births. Then, we subset the births data to only include those observations whose state names match the state names in our geospatial states data.

Chloropleth Map: Code

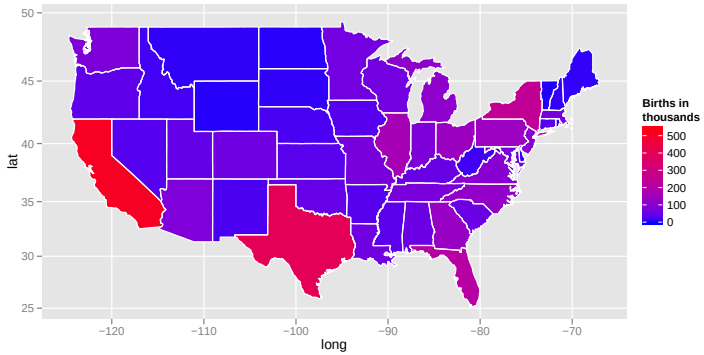
```
> chloropleth <- merge(states, births, "state")
> chloropleth <- chloropleth[order(chloropleth$order), ]
> p <- ggplot(chloropleth, aes(long, lat, group = group)) +
+   geom_polygon(aes(fill = kbirths2011)) +
+   geom_polygon(data = states, colour = "white", fill=NA) +
+   scale_fill_gradientn(name = "Births in\nthousands",
+     guide = guide_colorbar(),
+     colours = c("blue", "red"),
+     limits = c(0, 550)) +
+   coord_map(projection = "mercator")
```

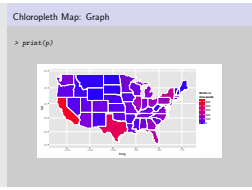



- The final step to a working data frame is to merge the two births and state geospatial data by state name.
- we see the familiar call to ggplot specifying the data frame, longitude is the x axis, latitude the y, and break it down by group.
- we add polygons (which will be the shape of the states) that are filled with the continuous variable thousands of births.
- we also add unfilled polygons that just have their outlines traced so it is easy to distinguish states
- finally we set the colours for the continuous births gradient to go from blue to red, and tell ggplot that the data are map data and to use the mercator projection.

Chloropleth Map: Graph

```
> print(p)
```

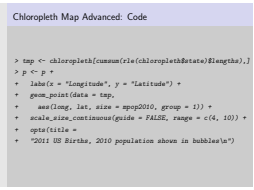




- here are the beautiful results of our work
- Alaska and other US provinces are left off, so we are just looking at the continental US. The legend is a nice colour bar with gradient showing the births in thousands.

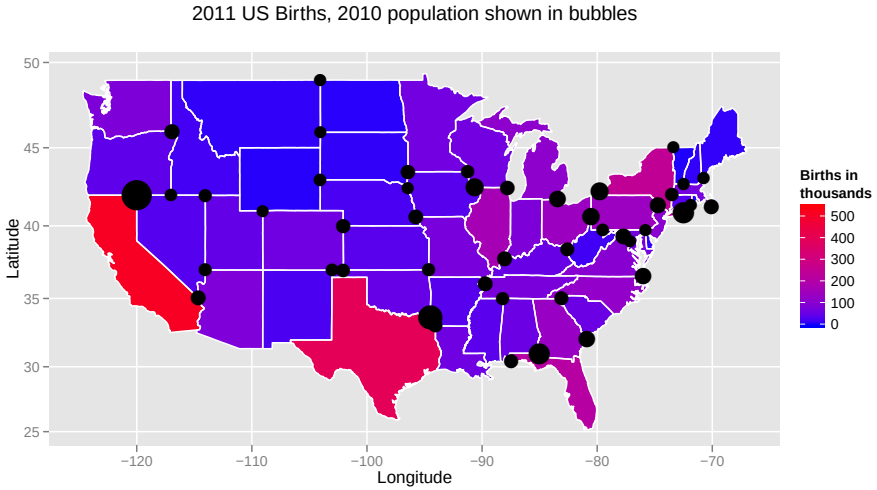
Chloropleth Map Advanced: Code

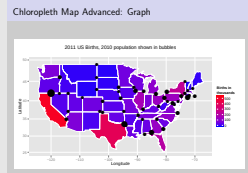
```
> tmp <- chloropleth[cumsum(rle(chloropleth$state)$lengths),]
> p <- p +
+   labs(x = "Longitude", y = "Latitude") +
+   geom_point(data = tmp,
+     aes(long, lat, size = mpop2010, group = 1)) +
+   scale_size_continuous(guide = FALSE, range = c(4, 10)) +
+   opts(title =
+   "2011 US Births, 2010 population shown in bubbles\n")
```

- If we had other data, we could also add that to our map. For example, from the census, we have the 2010 population of each state. We create a new dataset, `tmp` that is a subset of the full `chloropleth` data with just one observation per state.
- We can then use this new dataset, to plot the population of each state as a bubble plot. Something similar could be done with, for instance, county level data about mortality from heart disease or other diseases.

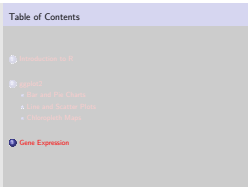
Chloropleth Map Advanced: Graph





- Note that there is no legend for bubble size because we turned it off using `guide = FALSE` earlier.
- In this case, the locations of the bubbles are not very sensible, but if we had the latitude and longitude of the centers of states or state capitals or something like that, it would work quite well.

Table of Contents



- I know this is different from what most of you are probably doing in your work, but I wanted to show it anyway, because it can give you ideas of what R can do and also potentially novel graphs or ways to visualize your own data.
- heatmaps of gene data are essentially just plots of matrices. If you have any matrix data, you could use this technique with it. For example, correlation or covariance matrices.

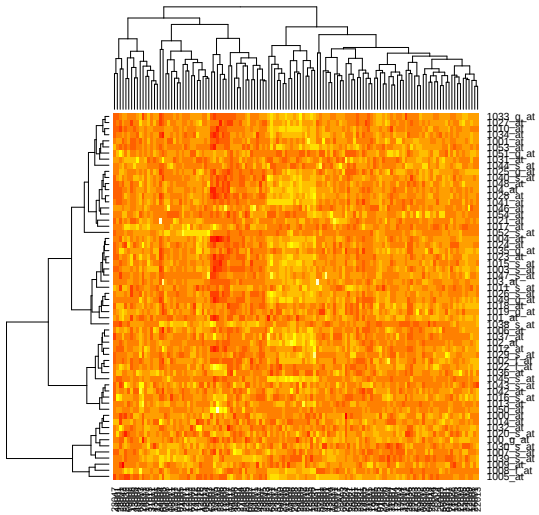
Gene Heatmap: Code

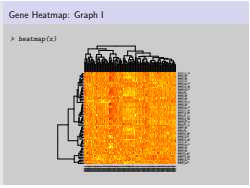
```
> ## install.packages("BiocInstaller", repos =
> ## "http://www.bioconductor.org/packages/2.11/bioc")
> ## require(BiocInstaller)
> ## biocLite("ALL")
>
> require(ALL)
> require("gplots")
> data("ALL")
> x <- exprs(ALL)[1:60, ]
```


- First we load the ALL package, because it has some gene expression (microarray) data. We also load a new graphing package, called, gplots. This actually builds on traditional or base graphics.
- Next we can bring in the data, and extract just the first 1 through 60 rows and store it in the matrix x.

Gene Heatmap: Graph I

```
> heatmap(x)
```

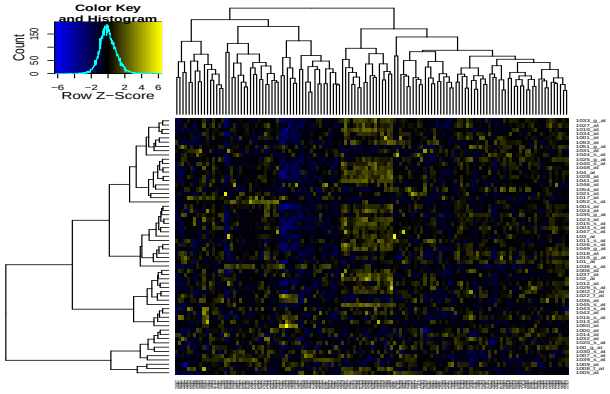


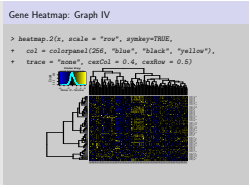


- `heatmap` is actually in core `R` not `gplots`. It just needs one argument: the matrix to be plotted. It automatically standardizes it, uses a clustering algorithm to create the dendrograms, and creates the nice final plot.

Gene Heatmap: Graph IV

```
> heatmap.2(x, scale = "row", symkey=TRUE,
+   col = colorpanel(256, "blue", "black", "yellow"),
+   trace = "none", cexCol = 0.4, cexRow = 0.5)
```





- We can customize the plot a lot more using the `heatmap.2` function in the `gplots` package.
- here, we tell it to scale (standardize) the matrix by row (column is also an option), ask for the key to be symmetric (so equally centered around 0). It is common for heatmaps to be red/black/green, but this is not very colorblind friendly, so instead we could use blue/black/yellow.
- Because there are so many labels, we shrink the text size to make it fit better using `cexCol` and `cexRow` (which stands for **c**haracter **ex**pansion).