Week 4

Home <- Week 3 Week 5->

Contents

References

Dalgaard 2008
Wickham, H. 2009 *ggplot2: Elegant graphics for data analysis*
Paradis, E. 2005 *R for Beginners* [PDF]
R Graph Gallery
Tufte, E. 2001 *The Visual Display of Quantitative Information*

To install today

```
install.packages("RColorBrewer")
```

Base Graphics

As Wickham points out in his book on his R graphics package, the R base graphics system has a pen on paper design. The basic way it works is you set up the plotting area, and then you add elements to it. Once you've added an element to a plot, it cannot be removed, so it's good idea to keep a little plotting script going so you can re-plot your tweaks more easilly.

Plotting Basics

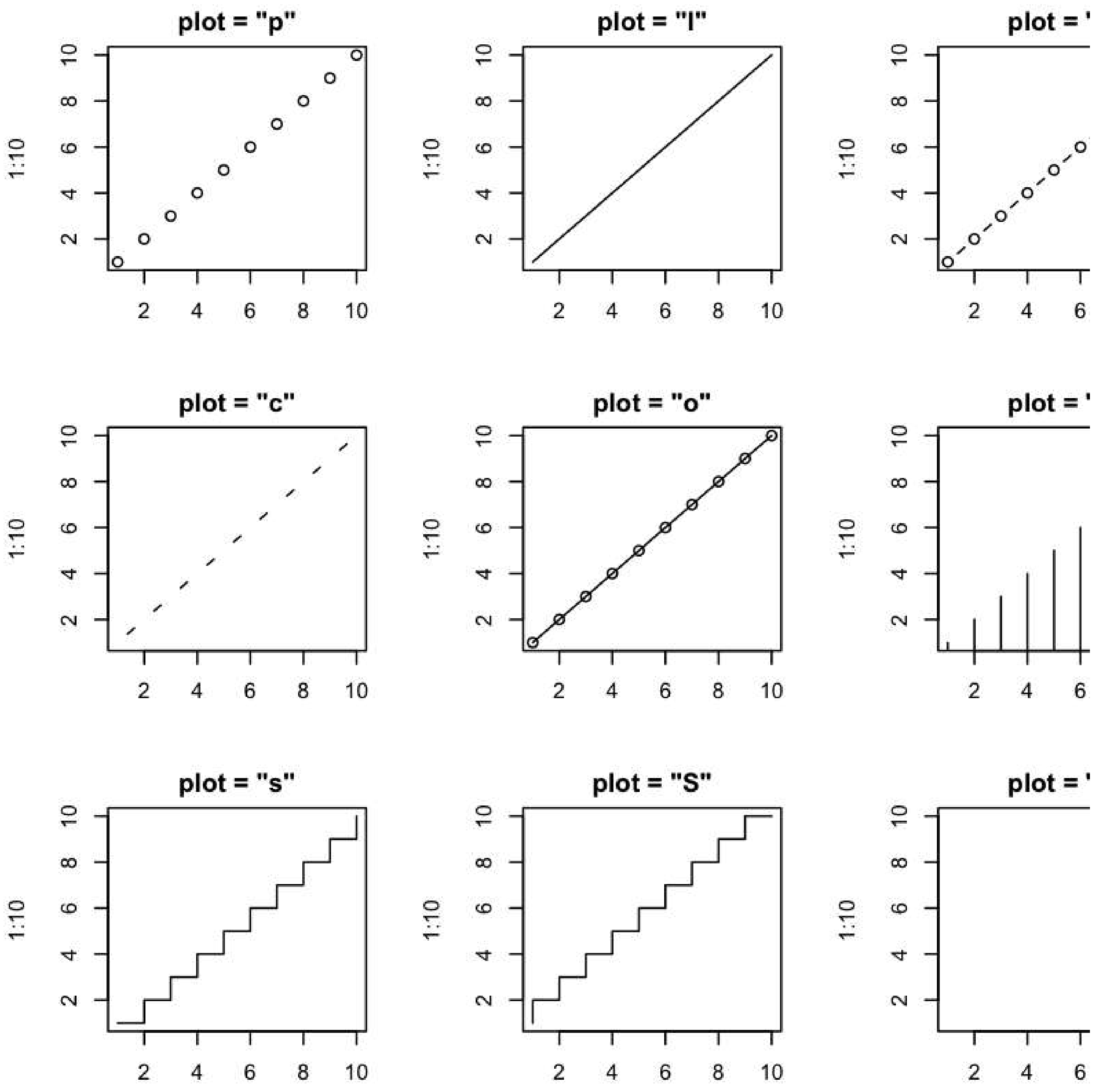
The most basic command you can use to produce a plot is `plot()`. The first two arguments to `plot()` are taken to be x and y coordinates. If you only provide one vector of values, this is used as the y-coordinate, and it is plotted against the index values of the vector.

Plot Types

There are 9 basic ways for R to plot a set of points. To use one of these, define the `type` argument one of these ways:

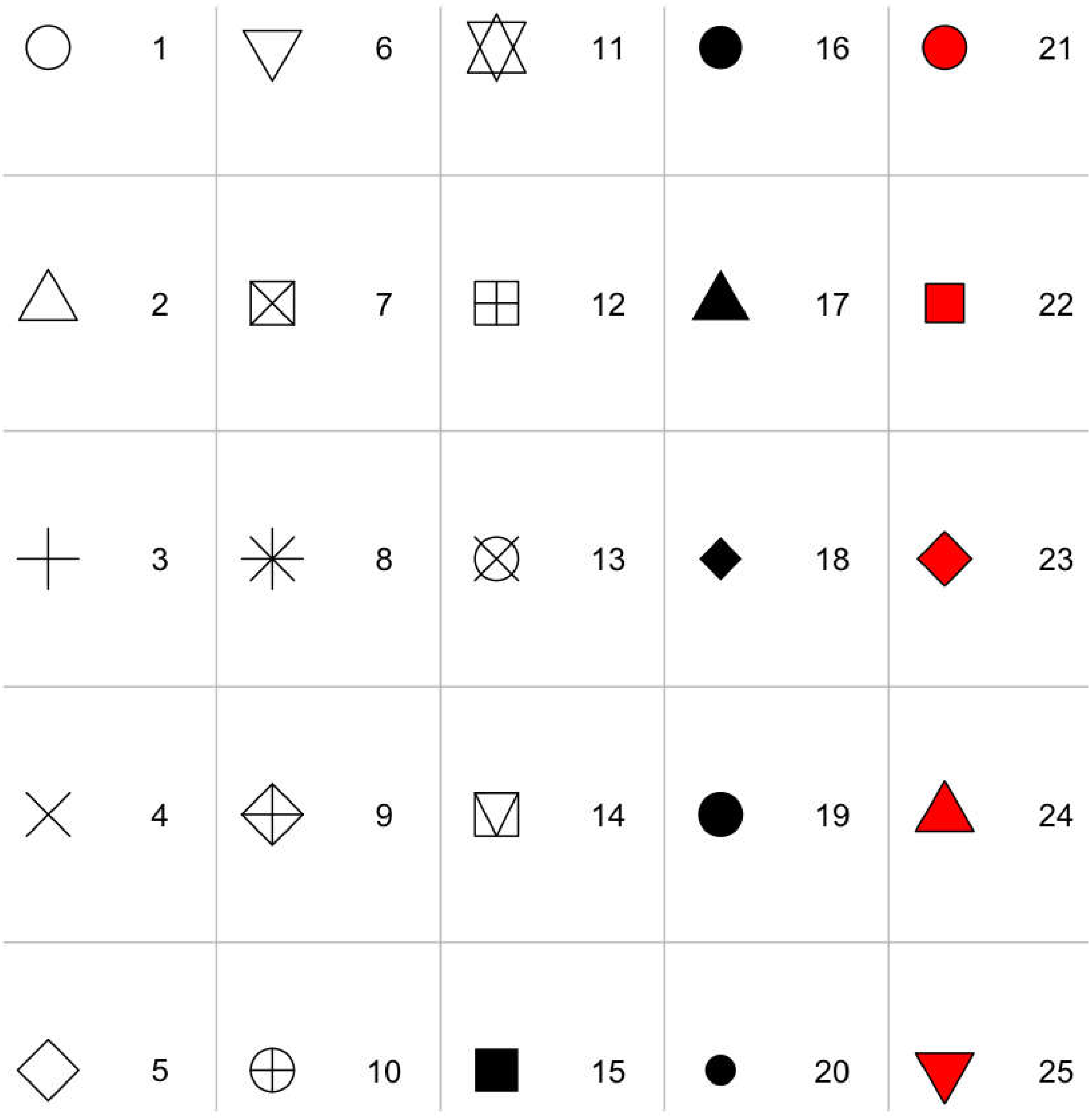
- `p`: points (this is the default)
- `l`: a line
- `b`: "both", points connected by line segments
- `c`: just the connecting segments of "b"
- `o`: "overplotted", points and lines overplotted
- `h`: a histogram
- `s`: stair
- `S`: alternative stairs
- `n`: none

Here's what the output looks like:



Point Types

There are 25 different plotting points that can be used by defining the argument `pch` (which I believe stands for "point character"). Points 21 thru 25 are multi-color points. Define the fill color for these with `bg` (which stands for "background").

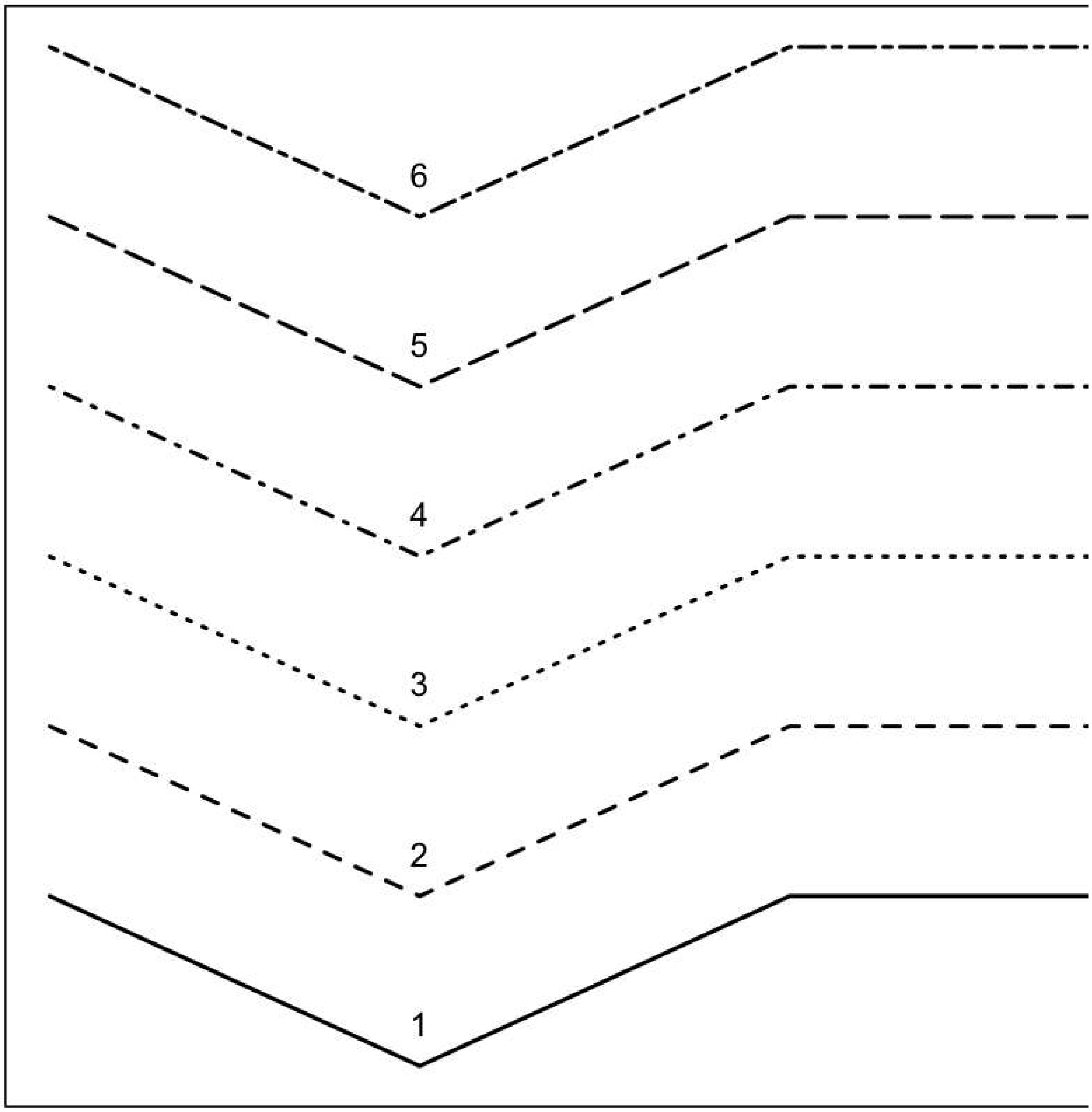


You can also give `pch` any single character string, and it will plot that.

```
plot(1:10,pch = c("@","$","#","&","%"),main = "Sarge says to
Beetle...")
```

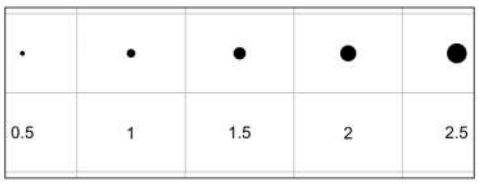
Line Types

There are 6 line types you can use by defining `lty` with 1 thru 6.



Weights and Sizes

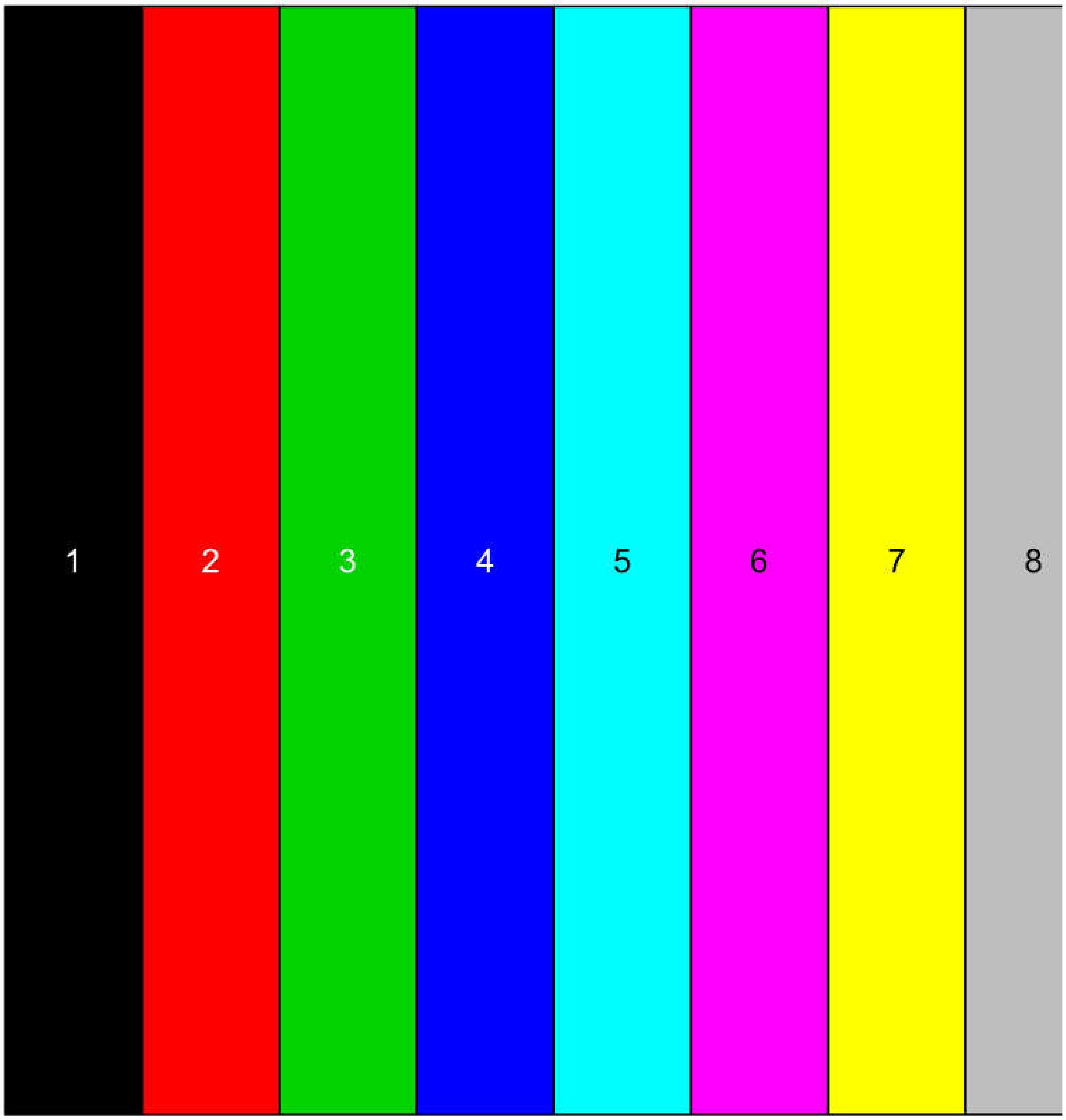
To control the size of plotted points, give the argument `cex` (for **c**haracter **ex**pansion I think) a value *x*, which will multiply the size of the character *x* times.



There are also `cex` arguments for different plot elements: `cex.axis, cex.lab, cex.main` and `cex.sub`

Colors

R has 8 default system colors, which can be assigned to a point or line by passing a numeric value of 1:8 to `col`



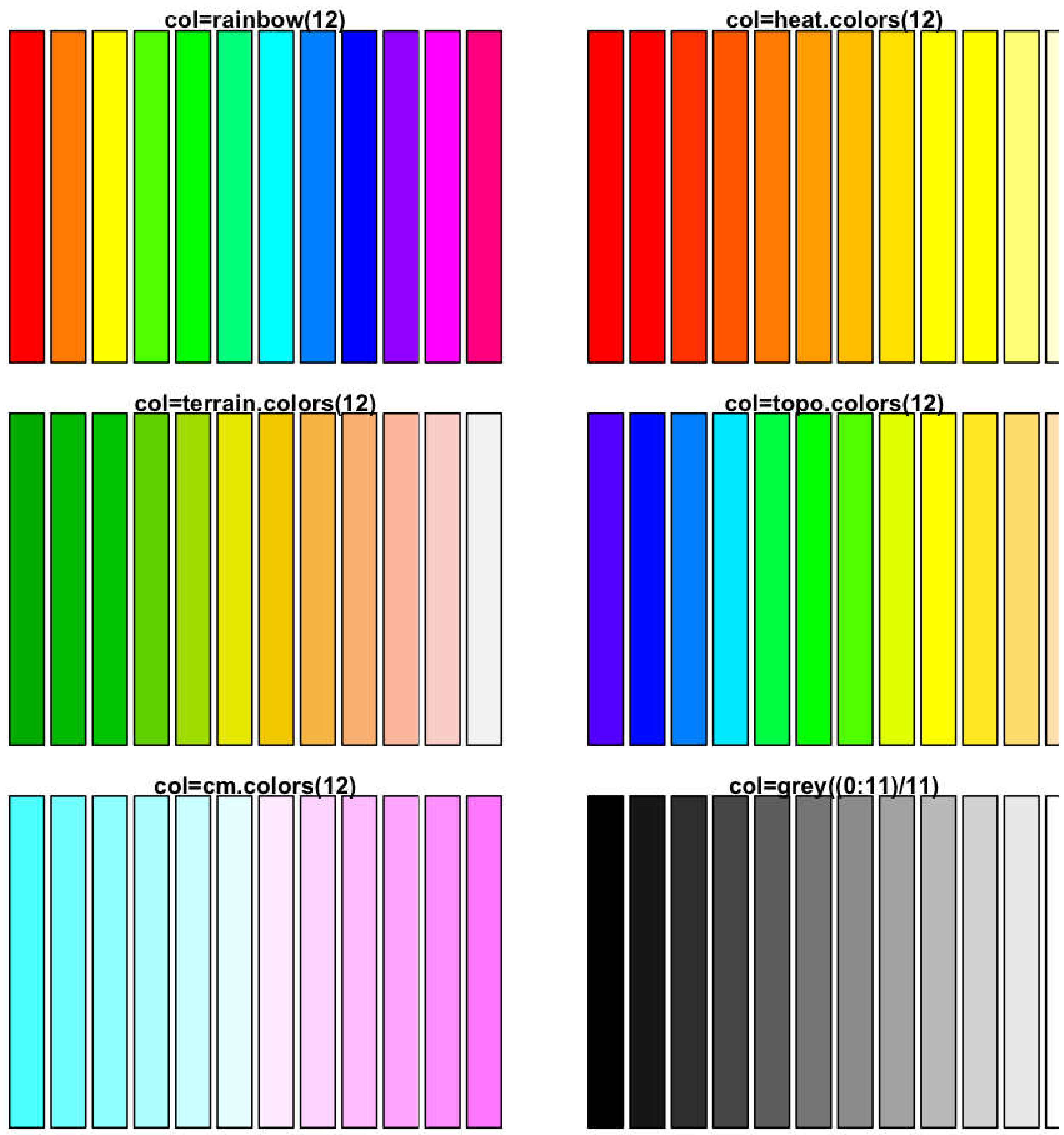
However, R has 657 named colors you can call upon by passing the name of the color to `col`. To see all the names, type `colors()`. You can read more about the named colors in R here, and there is a full chart available here [PDF].

There are particular color palettes that you could select colors from as well:

- `rainbow()`
- `heat.colors()`
- `terrain.colors()`
- `topo.colors()`
- `cm.colors()`
- `grey()`

If you give any of these functions a number, *n*, it will return that number of colors from the palette as a vector. There are also some other arguments you can pass to them as well, such as `alpha`, which controls transparency. They all share the same help page, so inspect that for more details.

The `grey()` palette is different, however. It takes a vector of values between 0 and 1 instead. 0 returns black, 1 returns white, and values in between return the appropriate percent grey.



Color Brewer

The RColorBrewer package is designed to provide a number of attractive palettes to suit different needs. You can explore the differen palettes interactively here. Enter the following to inspect the palettes you have to choose from.

```
library(RColorBrewer)
display.brewer.all()
?brewer.pal() ##to the help page
```

RGB, HSV, HCL

Of course, if you control the rgb, hsv or hcl color spaces, you can hand design your own color palettes. I don't so I won't talk any more about it, but it is a possibility

Building up Plots

With the information above, you can make some of the most basic plots. Here, we'll go over how to build plots from the ground up. The simplest way to think about it is that first you define the paper on which the plots will be drawn, then you define the plotting region in which you will draw, and then you add elements to that region.

`par()`

`par()` stands for "graphical **par**ameters", and in essence defines the paper on which plots will be drawn, as well as certain parameters which will be inherited by each plot on the page. Most graphical parameters can be set for each individual plot, but some must be set for the whole page. I'll go over some of these. To see the full list of definable parameters, see `?par`

`mfcol, mfrow`

If you want to have multiple plots per page, then you define it as follows:

```
n.col <- 2
n.row <- 2
par(mfrow = c(n.col,n.row))
```

Now, every time you define a new plot, it will fill in the page like this:

plot 1	plot 2
plot 3	plot 4

If you used `mfcol` instead, then the page will fill in like this:

plot 1	plot 3
plot 2	plot 4

`mar`

The `mar` argument sets the margins in terms of "lines" between the plotting region and the edge of the page. By default, they are set to `c(5,4,4,2)+0.1`. The numbers in the vectors stand for c(bottom, left, top, right).

bg

When defined within `par()`, `bg` changes the background color of the plotting area. By default, this is set to `"transparent"`, and I would suggest keeping it this way unless you have some other good reason.

Plot Region

Assuming you've set up your page as you like, then you need to define the plotting region with `plot()`.

x and y limits

By default, R will set the limits of the x and y axes to accommodate the range of x and y values passed to it. However, you can hand adjust the range of the axes with `xlim` and `xlim`. They take a vector of two numerical values: `c(from,to)`.

There are two immediate applications I can think of for changing the x and y ranges. First, when plotting vowel systems, the axis values run in reverse. `range()` will return the range of values, and `rev()` will reverse them.

```
plot(F2,F1,xlim = rev(range(F2)),ylim = rev(range(F1)))
```

The second case I can think of is i you have the age of a speaker or text, and want to plot some kind of variable in relation to this age. To represent the time course of a change on the x-axis, you need to reverse the order of the ages along it.

```
plot(Age, Variable,xlim = rev(range(Age)))
```

Axes

By default `plot()` will draw axes with ticks and labels at automatically determined places. If you would like to hand roll your own axes, set `axes = F` in the original `plot()` command, and add axes with `axis()`.

The first argument to `axis()` is which side to add the axis to. 1 = bottom, 2 = left, 3 = top and 4 = right. With no other arguments, this will add the axes as they would have been automatically in `plot()`. To have different labels on the axis, you need to provide a vector of labels to `labels`, and a vector of where to place them to `at`. The following code from the `axis()` help page demonstrates some of the possible customization of axes.

```
plot(1:7, rnorm(7), main = "axis() examples", type = "s", xaxt =
"n", frame = FALSE, col = "red")

axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:
axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)
```

Bounding Box

By default, `plot()` will produce a bounding box around the plotted region. To suppress this, pass `frame = F` to `plot()`. To change the box type, define `bty` with one of the following character values: `"l","7","c","u","]"`. The box type drawn will look like the capital form of these characters, (default is `"o"`). If you want to customise the color or line type of the box, then set `frame = F` in the original plot, and add the box with `box()`, passing the appropriate `bty, col, lty, lwd` values as desired.

Titles and Axis Labels

As far as I can tell, axis labels can only be defined in the original plot call. By default labels will be the names of the x and y arguments given to `plot()`. You'll frequently want to change these, especially if the argment utilize the `$` referencing. Do this by giving `xlab` or `ylab` the desired character sting.

To add a main title to a plot, pass a character string to `main`.

Adding Elements

Once you have defined a plotting window, it is possible to add more points, lines, etc to the plot. Exercise caution, however, since you may add points and lines which will be cut off by the plotting region. If this happens, it's necessary to go back and tweak `xlim` or `ylim` to be large enough to include the points you'll add.

`points()`

Just like it sounds, this adds points to a plot. Give it x and y value, and you can pass any other color, size or shape argument to it that you want.

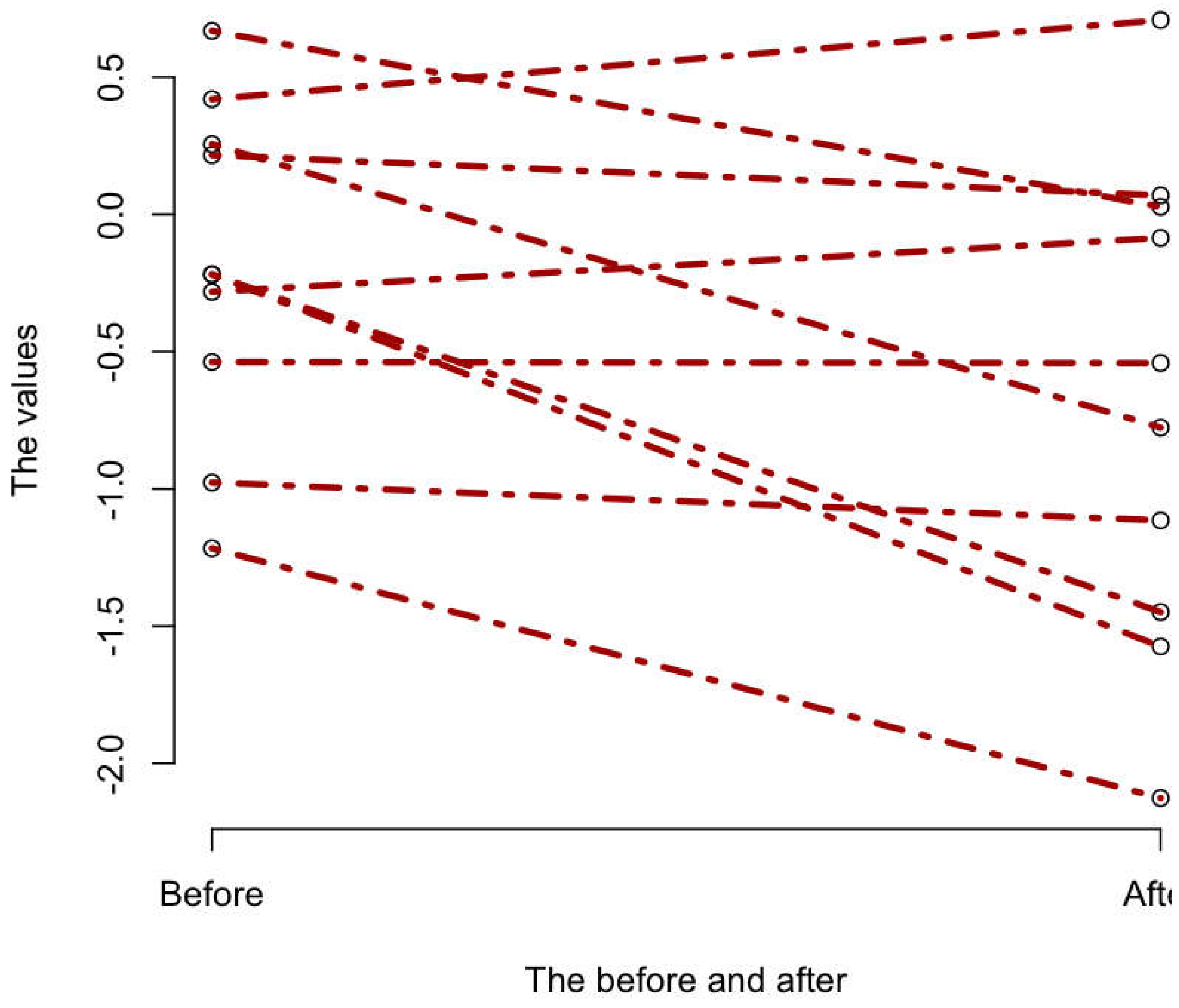
`lines()`

Again, just like it sounds, this will draw lines from x and y values.

`segments()`

This draws line segments from one point to another. It takes 4 vectors of arguments: From-x,From-y,To-x,To-y.

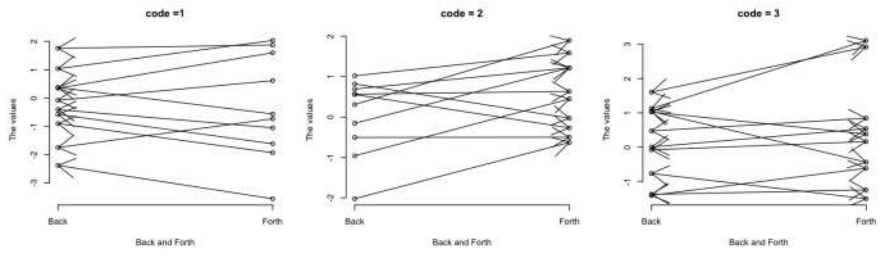
```
a <- rnorm(10)
b <- a - rnorm(10)
plot(rep(1,10),a, xlim = c(1,2), ylim = range(a,b) ,xlab = "The
before and after",ylab = "The values",axes = F)

points(rep(2,10),b)
axis(2)
axis(1,labels = c("Before","After"),at = 1:2)
segments(rep(1,10),a,rep(2,10),b,lty = 4,lwd = 3, col = "darkred")
```



arrows()

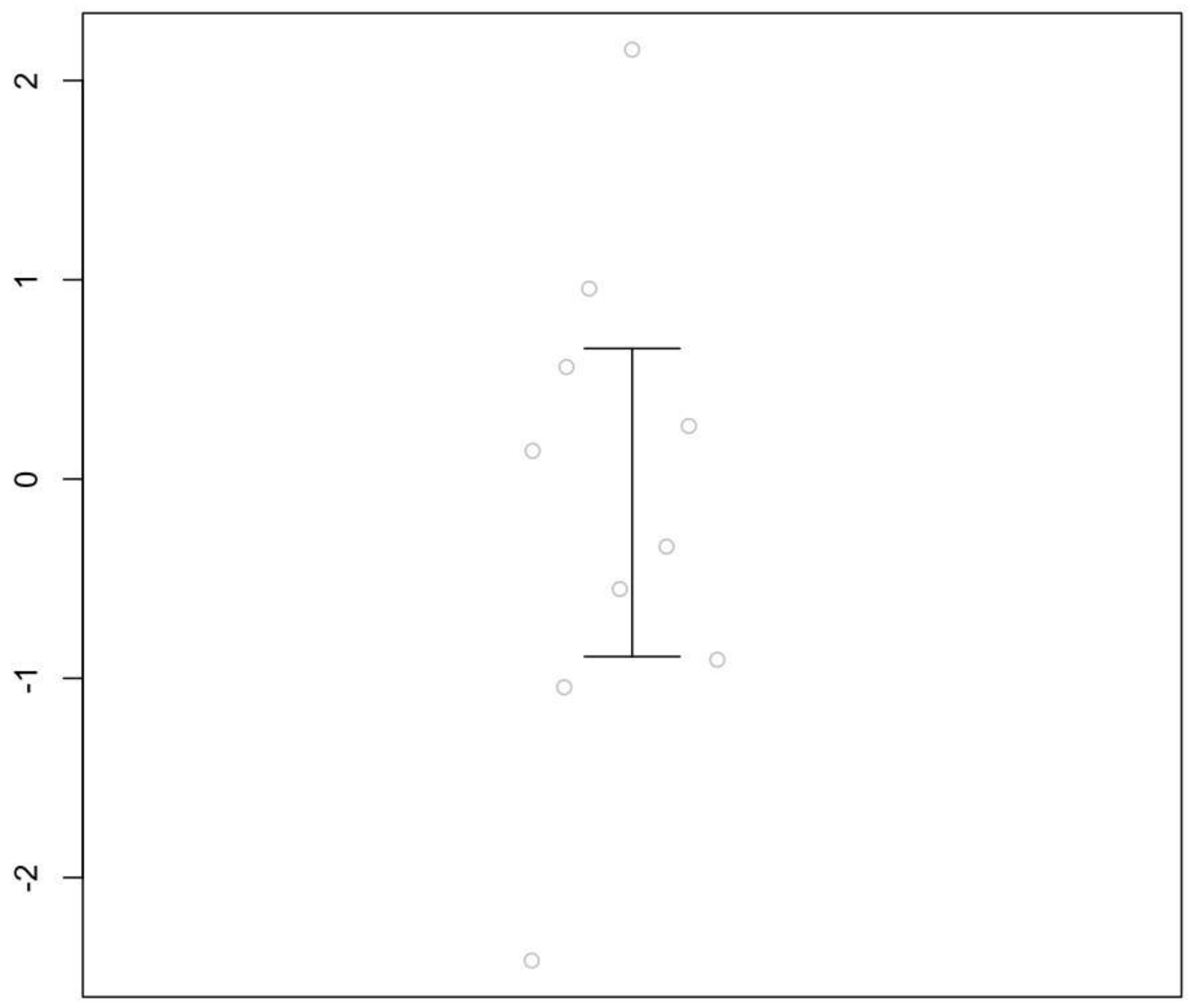
`arrows()` does much the same as `segments()`, but now draws an arrow head. There are a few more possible arguments to `arrows()`. `code` defines where the arrow head is drawn: 1 = from location; 2 = to location; 3 = both ends. `angle` defines the angle at which the arrow heads should be drawn from the segment. `length` defines the length in inches that the arrow head edges should extend.

```
a <- rnorm(10)
b <- a - rnorm(10)
plot(rep(1,10),a, xlim = c(1,2), ylim = range(a,b) ,xlab = "Back
and Forth",ylab = "The values",axes = F)
points(rep(2,10),b)
axis(2)
axis(1,labels = c("Back","Forth"),at = 1:2)
arrows(rep(1,10),a,rep(2,10),b,code = 1)
##or
arrows(rep(1,10),a,rep(2,10),b,code = 2)
##or
arrows(rep(1,10),a,rep(2,10),b,code = 3)
```



Note: This is a nice cheap way to draw some error bars.

```
a <- rnorm(10)
stripchart(a,method = "jitter", vert = T,pch = 1,col ="grey")
a.mean <- mean(a)
SEM <- sd(a)/sqrt(10)
a.high <- a.mean+(SEM*1.96)
a.low <- a.mean-(SEM*1.96)
arrows(1,a.high,1,a.low, angle = 90, code = 3)
```



`text()`

If you want to plot text on certain points, then you can use the `text()` function. It takes x and y values, and a vector of character strings to plot on the points.

`abline()`

There are a few ways to use `abline()`. In the simplest form it takes two arguments, `a` and `b`. It plots a line where `a` = intercept and `b` = slope. You can also plot simple horizontal and vertical lines by defining `h` and `v` values instead.

When look at simple regressions next week, we'll give `abline()` an `lm` object as an argument, and it'll plot the simple regression line.

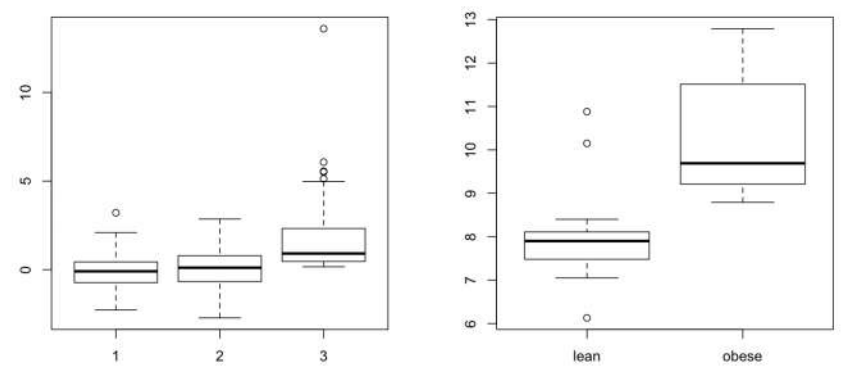
Higher Level Plots

There are a number of higher level plots in R that have their own kind of unique behavior and arguments. These are some of the more useful ones.

Boxplots

Even though some people don't like boxplots, they can be useful for eyballing the distribution of data. The box indicates the location of the first and third quartile, with the median indicated with a bar. Whiskers are drawn out to 1.5 times the length of the box, and any points outside this range are plotted (these can be considered outliers).

```
boxplot(rnorm(50),rnorm(100),rlnorm(100))
library(ISwR)
boxplot(expend~stature,data = energy)
```



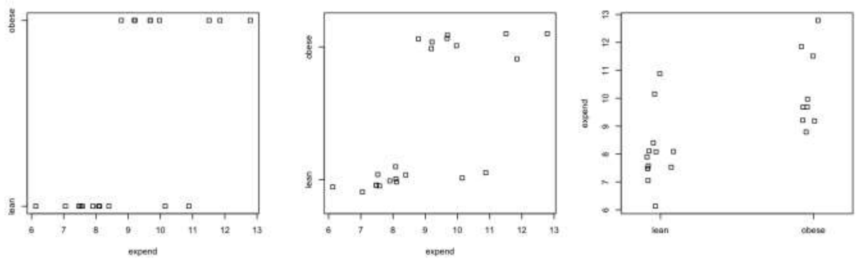
Some important arguments for `boxplot()` are:

- names: A vector of names for the labels
- col: color to fill the box with
- border: color for the lines to be drawn with

Stripchart

When you want to compare two groups along a single dimension, a stripchart is a good alternative to the box plot. By default, `stripchart` will use `method = "overplot"`, but I prefer `method = "jitter"`.

```
stripchart(expend~stature,data = energy)
stripchart(expend~stature, data = energy, method = "jitter")
stripchart(expend~stature, data = energy, method = "jitter",vert =
T)
```

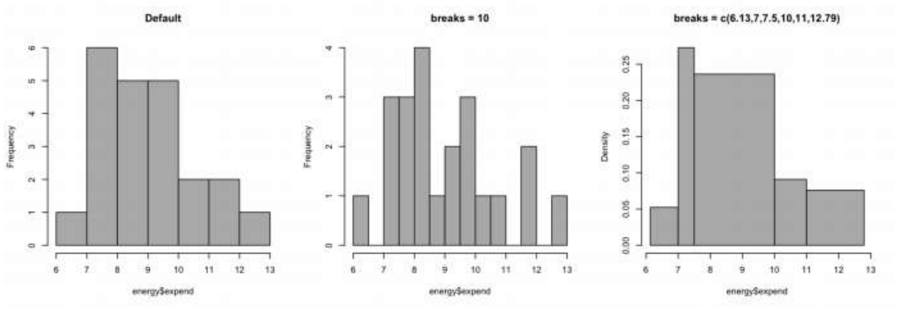


Histogram

You can easily plot a histogram of continuous data with `hist()`.

```
hist(energy$expend)
```

You can control bars and their width with the `breaks` argument. If you pass a number *n* to `breaks`, it will produce a histogram with about that many bars. You could also pass a vector of numerical values to `breaks`, in which case it will draw bars according to the intervals you specify.



Bar Chart

Bar charts are exactly what they sound like. They can take either a vector of numbers to be the bar height, or they can take a table or matrix to produce grouped bar charts. Here is some data that Dalgaard reproduces:

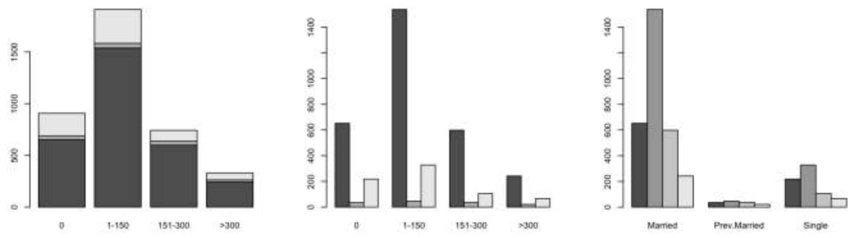
```
caff.marital<-matrix(c
(652,1537,598,242,36,46,38,21,218,327,106,67),nrow = 3, byrow = T)
colnames(caff.marital)<-c("0","1-150","151-300",">300")
rownames(caff.marital)<-c("Married","Prev.Married","Single")
```

By default, R produces a stacked bar chart. Define `beside` as true for side by side bar charts

```
barplot(caff.marital)
barplot(caff.marital,beside = T)
```

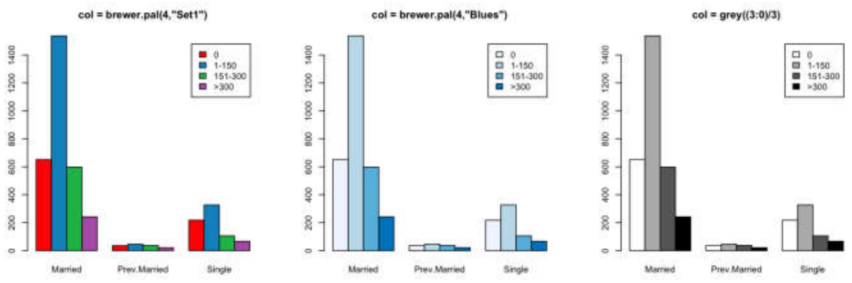
For this data, it makes more sense for marital status to define the groups, so we'll transpose the matrix with `t()`

```
barplot(t(caff.marital),beside = T)
```



Now let's beautify it with some colors and a legend.

```
brewer.pal(4,"Set1")->pal
barplot(t(caff.marital),beside = T,col = pal,legend.text=colnames
(caff.marital))
brewer.pal(4,"Blues")->pal
barplot(t(caff.marital),beside = T,col = pal,legend.text=colnames
(caff.marital))
grey((3:0)/3)->pal
barplot(t(caff.marital),beside = T,col = pal,legend.text=colnames
(caff.marital))
```



If you want to display the data as percentages per marital group, we can do this with `prop.table()`, which we'll go over more when we do categorical data.

```
prop.table(caff.marital,1)
barplot(t(prop.table(caff.marital,1)),beside = T,col =
pal,legend.text=colnames(caff.marital))
```

Dot Chart

Dot charts are a lot like bar charts, but points along a line represent bar height rather than a bar. It is a somewhat simpler way of printing the same data.

```
dotplot(t(caff.marital))
```

Pie Charts

Don't use pie charts.

Interaction Plot

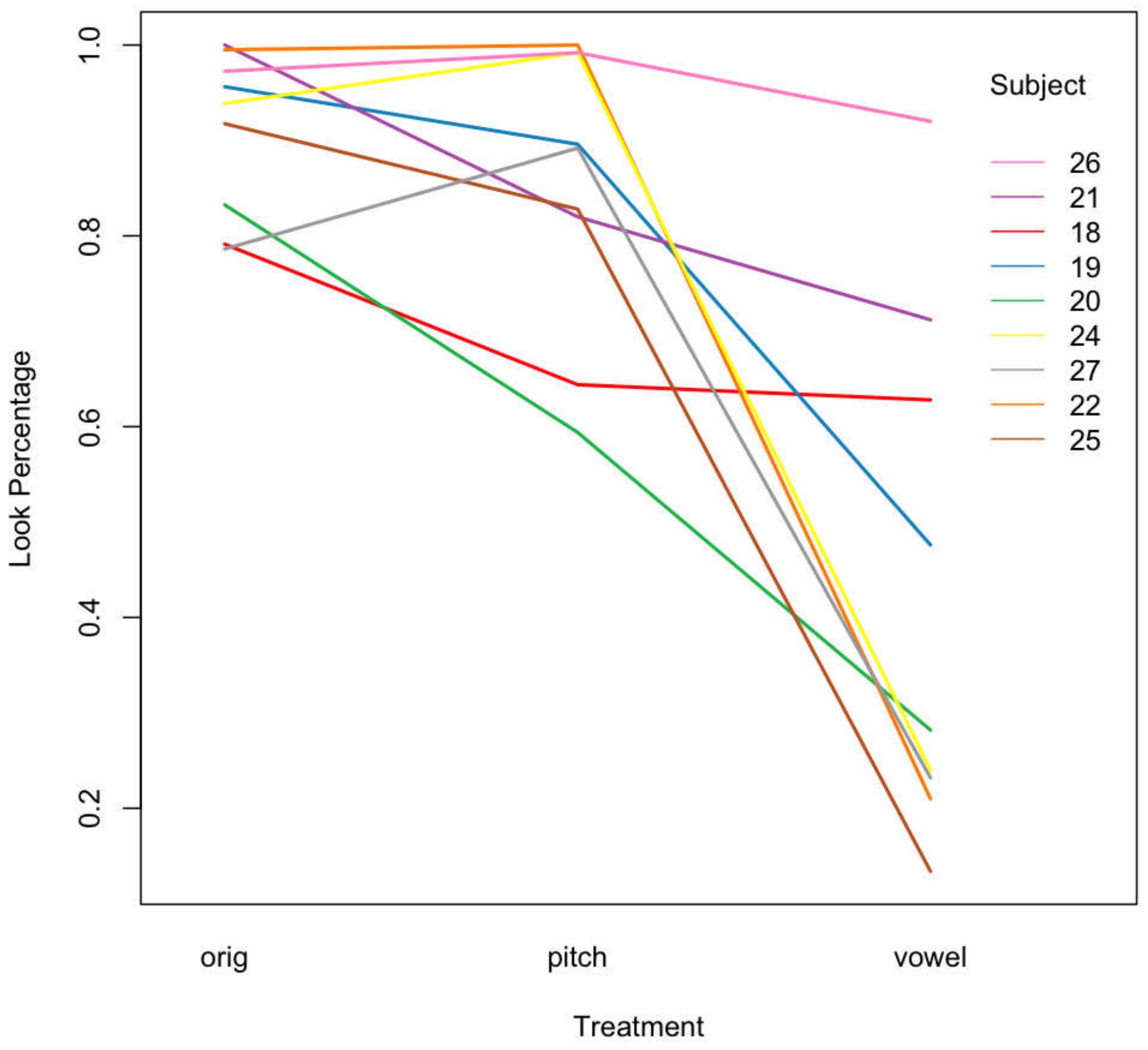
Interaction plots are a nice way to visualize data across two dimensions across different groups. Here's an example from Quam and Swingley (submitted) as available from the MLM reading group page, where they have percentage of looking to a target for three conditions across 32 subjects. This'll be a pretty large interaction plot, so we'll just take a subset.

```
deebo<-read.csv
("http://www.ling.upenn.edu/~kgorman/papers/deebo.csv")
sdeebo<- subset(deebo,Subject %in% 18:27)
```

The arguments `interactionplot()` take are ordered this way:

1. x-axis variable
2. groups
3. y-axis variable

```
pal<-brewer.pal(9,"Set1")

interaction.plot(sdeebo$Treatment, sdeebo$Subject,
sdeebo$LookPercentage, col = pal, lty = 1, lwd = 2, trace.label =
"Subject", xlab = "Treatment",ylab = "Look Percentage")
```



Legends, Labels, Etc.

There are some functions that will allow you to interact with a plot, either to extract or add points and labels. You can also use them to interactively place a legend.

`locator()`

The `locator()` function can be used for a few purposes. After having created a plot, calling `locator()` with no arguments will return a list of x and y coordinates that you clicked on in a plot before right, or CMD clicking. If you know a finite number of points you want to extract, you can define this with `n`.

If you pass a `type` argument (one of "p", "l", or "o"), `locator()` will actually draw points or lines on the coordinates you click on within a plot.

```
plot(1:10)
locator() ##right click, or CMD click to end
```

```
plot(1:10)
locator(n = 3, type = "o",lwd = 3,lty = 2,pch = 3)
```

identify()

If you want to identify specific points with a label, you can use `identify()`. After creating a plot, pass x and y coordinates of points to `identify()`, a long with the vector you would like to label them with. There are a few other arguments you can specify to control how labels will appear. I would suggest looking at `?identify` to see how they work.

```
plot(1:10,1:10)
identify(1:10,1:10,letters[1:10]) ## right click or CMD click to
end
```

legend()

Of course, what would a plot be without a legend? The higher level plots, like `interaction.plot()` or `barplot()` have arguments within the plotting call for defining a legend. For the lower level plots, you need to define the legend yourself. Legends are very cusomizable, but here is the basic use.

The first argument to `legend()` is where it should go. You can put it on the center of any side of the plot with one of `"top", "bottom", "right", "left"` or in any corner with one of `"topright", "topleft", "bottomright", "bottomleft"`. You could also pass x and y coordinate values for the location of the top-left corner of the legend, but we'll come back to that

You next need to provide a character vector for the legend labels to the argument `legend`. Hopefully you'll have a vector of some sort already lying around.

Lastly, you need to determine what kind of symbol should appear next to each label. If you've used color to distinguish groups, pass the vector of colors to `fill`. If you've used different point characters, pass that vector to `pch`. If you've used different line types, pass that vector to `lty`.

Now, if you want to legend to be located somewhere other than the 4 sides or 4 corners, a nifty trick (from Dalgaard) is to give the first location argument `locator(n=1)`. The legend will then be plotted wherever you click on the plot.

```
plot(1:9,col = brewer.pal(9,"Set1"),pch = 1:9)
legend("topleft",legend = letters[1:9], fill = brewer.pal
(9,"Set1"))
legend("bottomright",legend = c
("one","two","three","four","five","six","seven","eight","nine"),pch
= 1:9)
legend(locator(n=1),legend = c("not present"),fill = "green")
```