Advanced Graphs

Statistics 427: R Programming

Module 13

2020

2-D plots

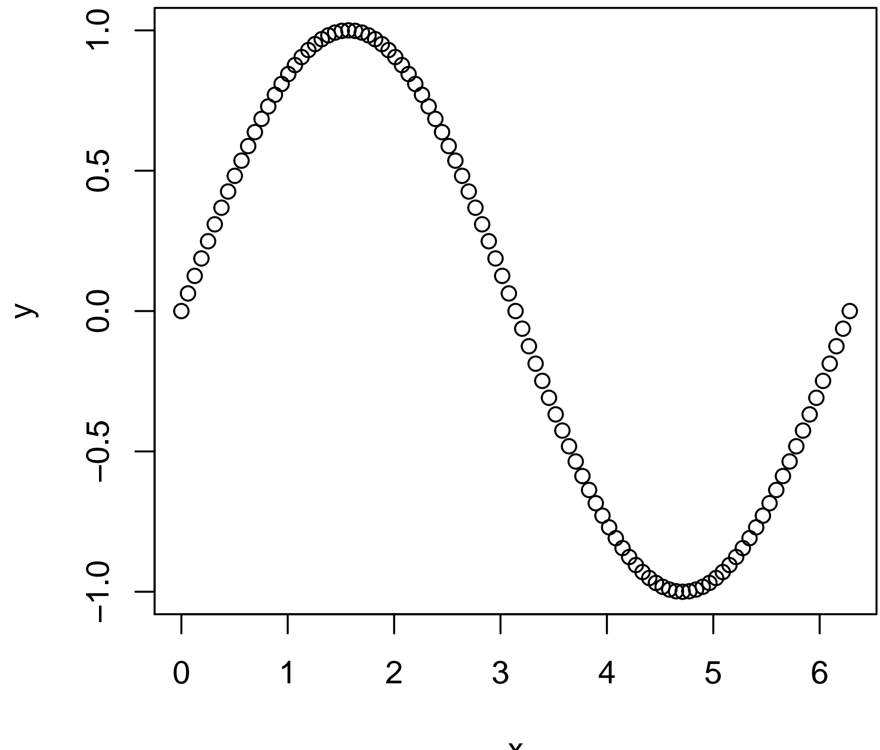
The two main functions for two-dimensional plotting in R are `plot()` and `matplot()` (introduced in last module).

The general form for `plot()` was discussed in earlier modules and we will see some new options to play with, while `matplot()` was briefly discussed.

plot() (yes again, but with more stuff)

`plot()` will produce the standard $x - y$ plots in Cartesian coordinates. It takes two vectors as arguments, x for horizontal axis and y for vertical axis.

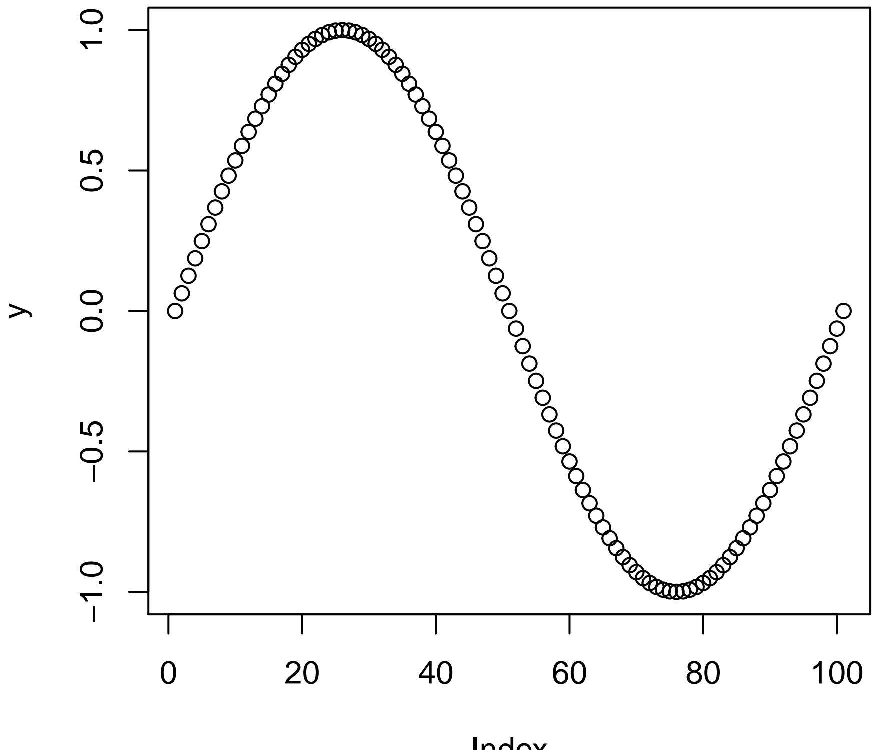
```
x=(0:100)*2*pi/100
y=sin(x)
plot(x,y)
```



plot()

It can work with just one vector of data, it will treat the vector as y and automatically generate an x vector of index values of 1:n.

```
x=(0:100)*2*pi/100
y=sin(x)
plot(y)
```



matplot()

General form of `matplot()`:

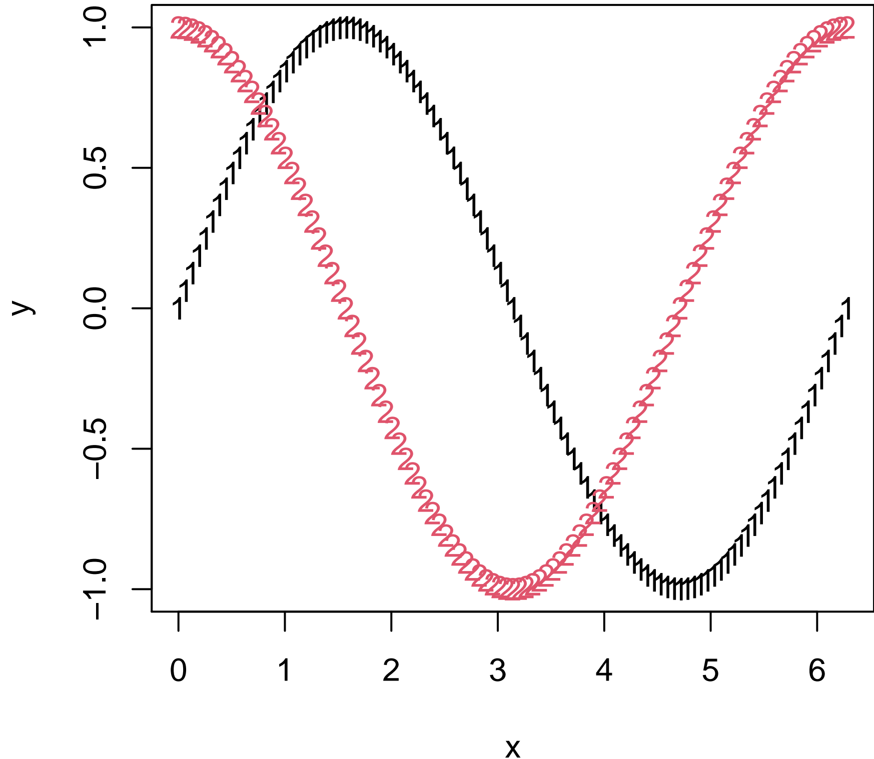
`matplot(x,y,type='p',...)`
`x` and `y`: vectors or matrices of data for plotting (`x` is horizontal, `y` is vertical)
`type`: ‘p’ (default) points, ‘l’ (lowercase L) line, ‘b’ both, etc.
`...`: more options

It does not include a title (`main=`) option; use of `title()` for graph titles.

`matplot(x,y); title('Title')`

matplot()

```
x=(0:100)*2*pi/100
y1=sin(x)
y2=cos(x)
y=cbind(y1,y2)
matplot(x,y); title('Plot of X and Y')
```



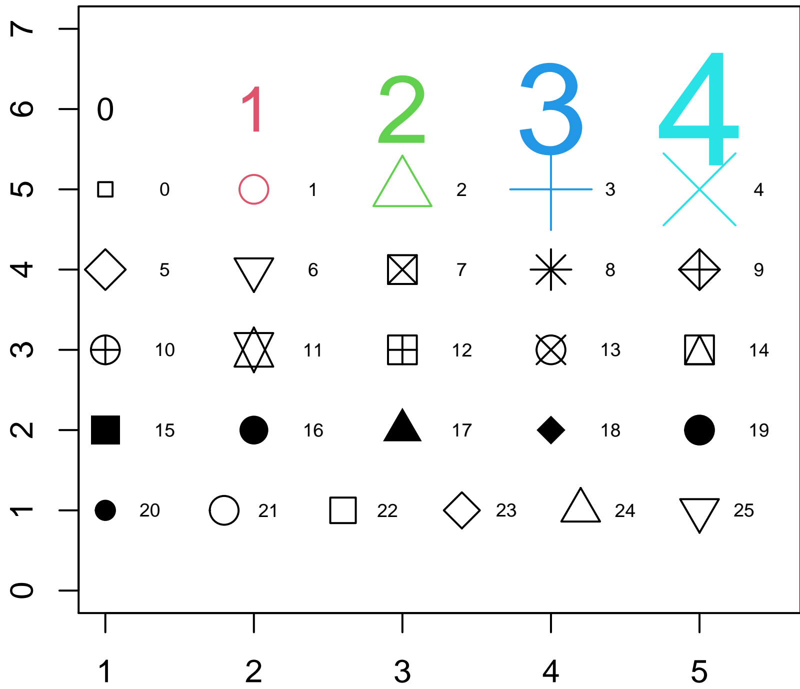
Options galore

Many options exist for customizing graphs. Options are usually invoked as additional arguments in the `plot()` or `matplot()` functions. The optional arguments can be entered in any order (after the vectors to be plotted), separated by commas. Most options have default settings that will be in force if you omit those arguments (like the `type=` option defaulting to `type='p'`). The default settings are usually pretty good, but there is always room for the option to improve. Some options have already been introduced in previous modules but some review is always good.

Data symbol types

Symbols used for the data points in a plot are designated with the `pch=` option (`pch` stands for plotting character); for example, `pch=1` would draw the points as (open) circles. The following slide is an example of some (code not shown in lecture slides but will be in code examples)

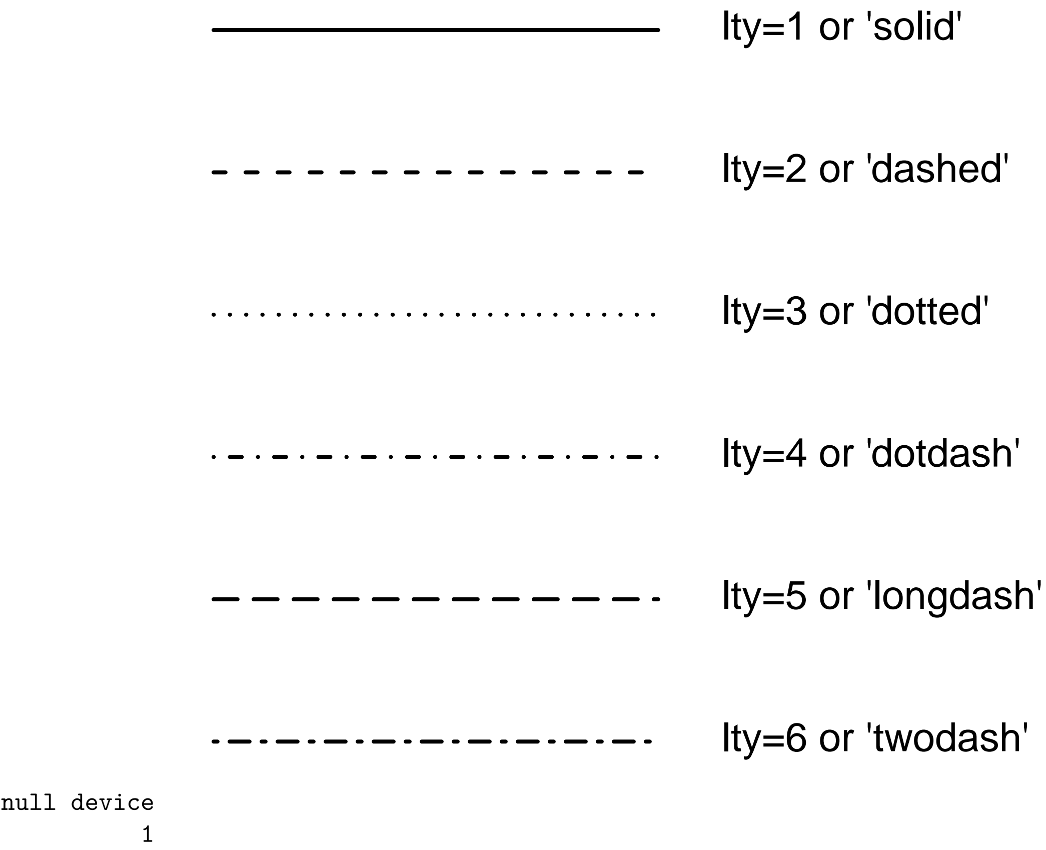
Data symbol types plot



Connecting line types

If a line is used to connect data points or portray curves, the style of the line is controlled by the `lty=` option. The common line types in the following slide is an example of the line types (code not shown in lecture slides but will be in code examples)

Connecting line types plot



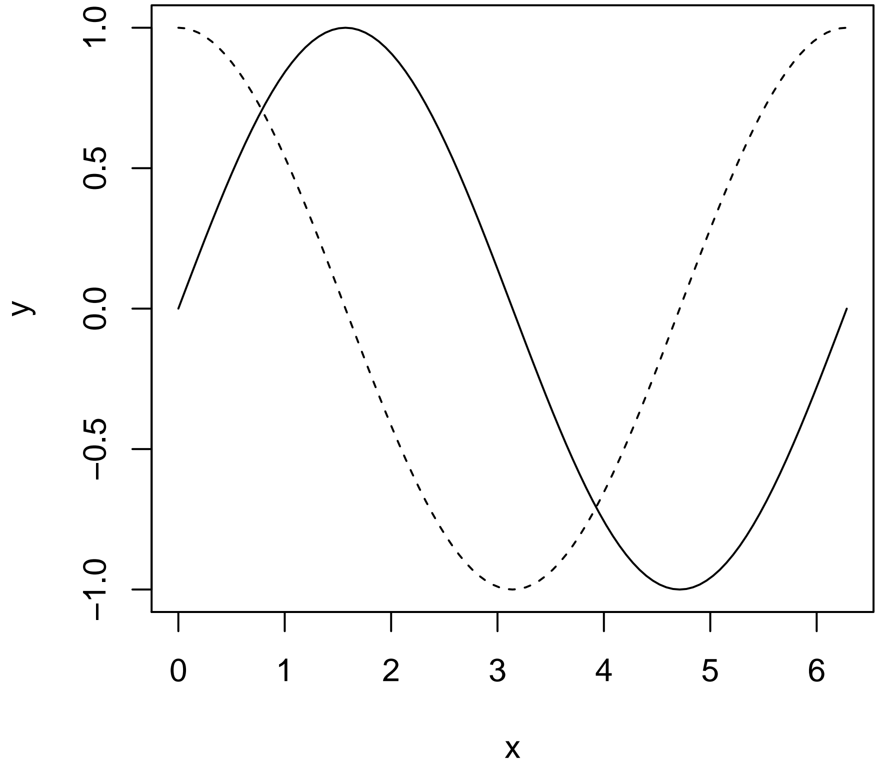

```
null device
          1
```

Connecting line types

Additionally, when using the `lty=` and other options in `matplot()`, setting the options to a vector value will cause the multiple plots to rotate sequentially through the list of different styles specified. As an example, in the next plot, using the `col='black'` option will suppress the use of different line colors in the `matplot()` function

Connecting line types plot

```
matplot(x,y,type='l',lty=c(1,2),col='black')
```

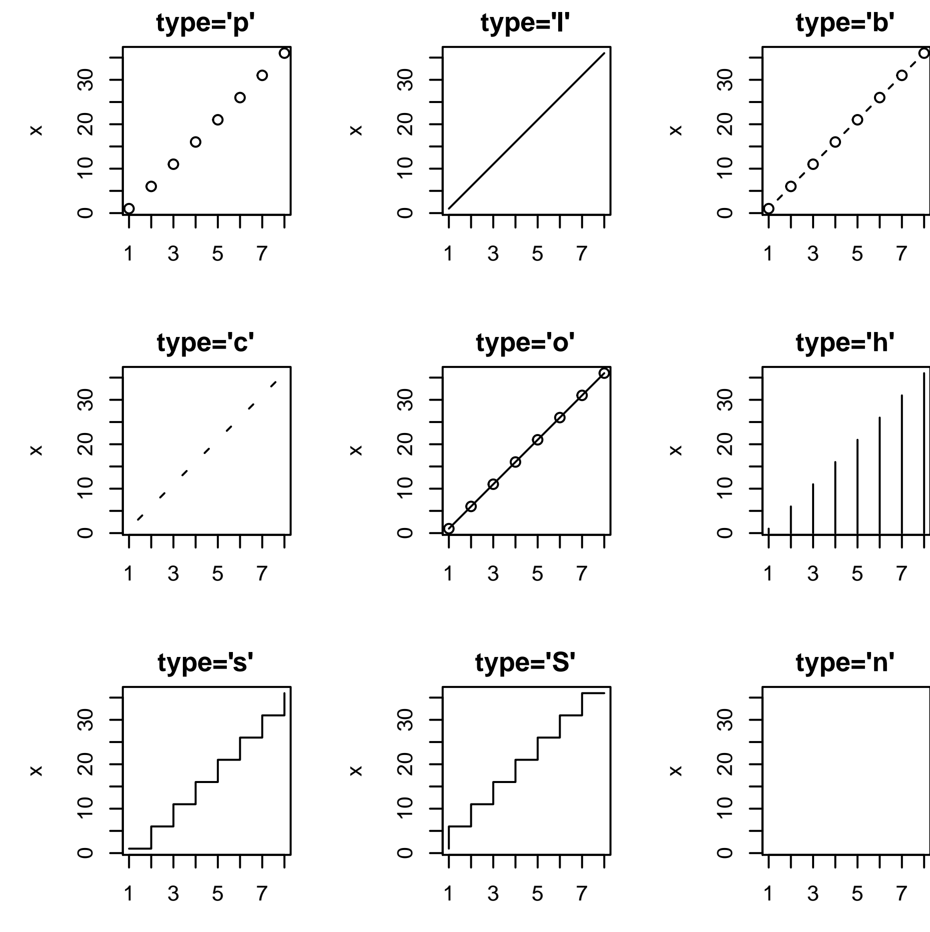


Plot types

Different plot types are specified with the `type=''` option. The default plot type is points with `type='p'`. The common types:

`'p'`: (default) points only
`'l'`: (lowercase L) for line only
`'o'`: (lowercase letter O) for both overplotted (lines and points)
`'b'`: both points and connecting lines
`'c'`: both points and connecting lines without lines touching points
`'h'`: vertical line drawn from each point to the x-axis (called comb plot)
`'s'`: staircase plot; neighboring points connected by horizontal line followed by a vertical line
`'S'`: other step plots, details in Help
`'n'`: no plot generated

Plot types. . . plot




```
null device
          1
```

Axis limits

The limits for the horizontal and vertical axes are specified respectively with the `xlim=` and `ylim=` options. The values needed for the limit options is a vector with two elements, from the smallest value to largest value, as in `c(a,b)`. By default, `R` will zoom into the graph within the limits of the function graphed, and will not show too much of sections of the quadrants with no data points.

If you want limits on both x and y to range from 0 to 10 rather than what the default does, then you would use: `xlim=c(0,10),ylim=c(0,10)`

Tic marks

The tic marks on the axes are controlled with `lab=` and `tcl=` options. The setting of the `lab=` option is a vector with two elements: the first is the number of tic marks on the horizontal axis and the second is the number of tic marks on the vertical axis. As an example, `lab=c(7,3)` gives an x-axis divided into eight intervals by seven (7) tic marks, and y divided into four intervals by three (3) tic marks. If you choose poorly, `R` will ignore you and override the `lab=` option if the choices do not work well in the display.

The `tcl=` option gives the length of the tic marks as a fraction of the height of a line of text. The setting `tcl` can be a negative number, in which case the tic marks will be on the outer side of the axes; a positive setting

(default) produces tic marks on the inner side of the axes.

Axis labels and suppression of axes

By default, the labels for the x- and y- axes are the names of the plotted vectors. The labels can be changed with `xlab=''` and `ylab=''` option.

Occasionally a plot without axes is better. The plot that shows the differing symbol types for the `pch` argument is done without axes (and without a plot… but semantics).

Axes are suppressed by the option `axes=F` option and the axis labels can be suppressed with the option `ann=F`

Sizes of symbols, labels, line widths, and axes

The size of the plotting characters can be altered with the `cex=` option. The numerical value specified for `cex` is a multiplier of the default character size, which is 1; a `cex=2` option will give symbol sizes twice the normal size while a `cex=.5` option will give symbol sizes half the normal size. Sizes of other plot aspects are controlled similarly as multiplier values through the following options:

`cex.axis=`: axis width multiplier
`cex.lab=`: axis label text size multiplier
`cex.main`: main title text size multiplier (`main=''` in `plot()` or `title()`)
`cex.sub`: subtitle text size multiplier
`lwd=`: plotting line width multiplier

Adding points

You can additional points or model curves to an existing, open plot with the `points()` and `matpoints()` functions. The `points()` function add to the `plot()` function and the `matpoint()` add to the `matplot()`. The two functions have many (not all) arguments that are the same for the appearance of data and curves. Options that attempt to alter aspects of the existing graphs such as the limits of axes, will however, not work.

Adding lines

You can additional lines to an existing, open plot with the `lines()` and `matlines()` functions. The `lines()` function add to the `plot()` function and the `matlines()` add to the `matplot()`. The functions take two vectors as arguments: the x- and y-coordinates. The line is drawn between each successive pair of points given by the vectors. The optional arguments governing line types and styles can be used.

Adding lines

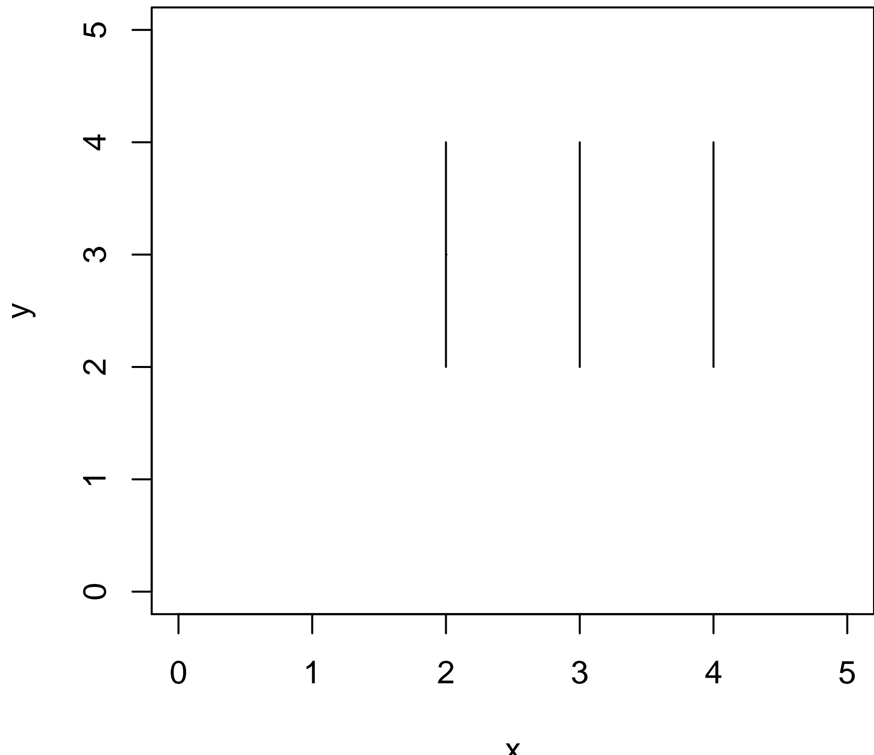
Nonconnecting (disjunct) line segments can be added to an existing, open plot with the `segments()` function. An alternative is to invoke multiple `lines()` commands. The `segments()` function takes four vector arguments. The first two are the x- and y- coordinates, respectively, of the points at which the line segments are to originate and the second two vectors are the x- and y- coordinates, respectively, of the points at which the line segments are to terminate.

Successive different plotting styles for the segments can be specified (in this and other functions) by assigning vector values for the plotting style options.

Adding points, lines, segments

```
x0=c(2,2,3,4)
x1=x0
y0=c(2,3,2,2)
y1=c(4,3,4,4)
```

```
x=c(0,3)
y=x
plot(x,y,type='l',lty=0,xlim=c(0,5),ylim=c(0,5))
segments(x0,y0,x1,y1)
```



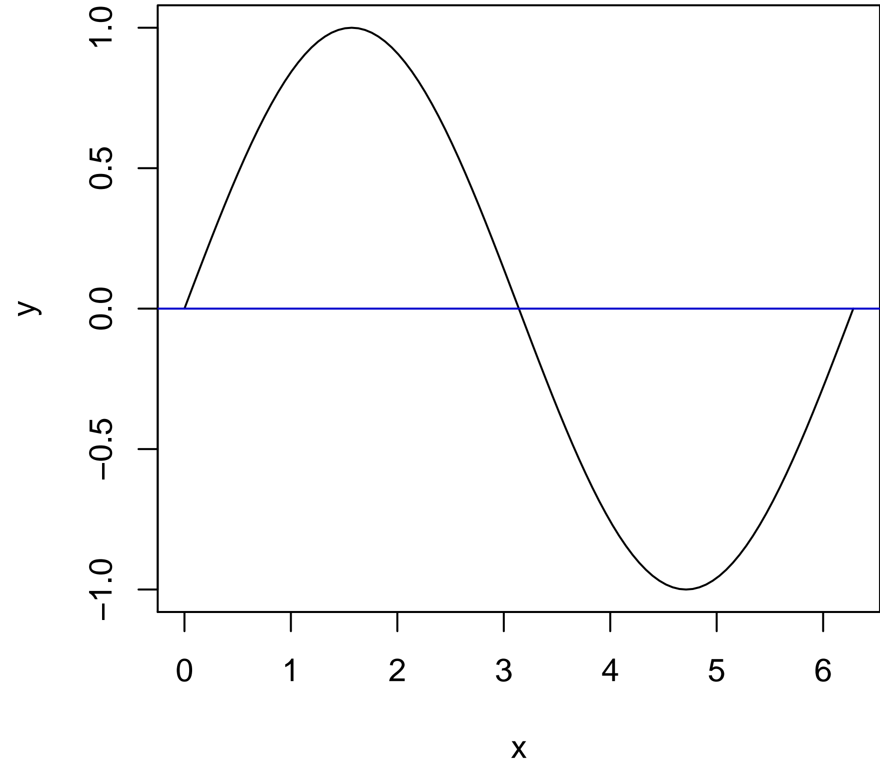
More lines

Lines can also be drawn across the entire graph with `abline()`. The function can take the following four forms:

`abline(a,b)`: `a` is slope and `b` is intercept of a single line to be drawn
`abline(h=)`: draws a horizontal line at the y value specified by `h=`
`abline(v=)`: draws a vertical line at the x value specified by `v=`
`abline(fit)`: regression (trend) line of model called `fit`

`abline()` example

```
x=(0:100)*2*pi/100
y=sin(x)
plot(x,y,type='l',lty=1)
abline(h=0,lty=1,col='mediumblue')
```

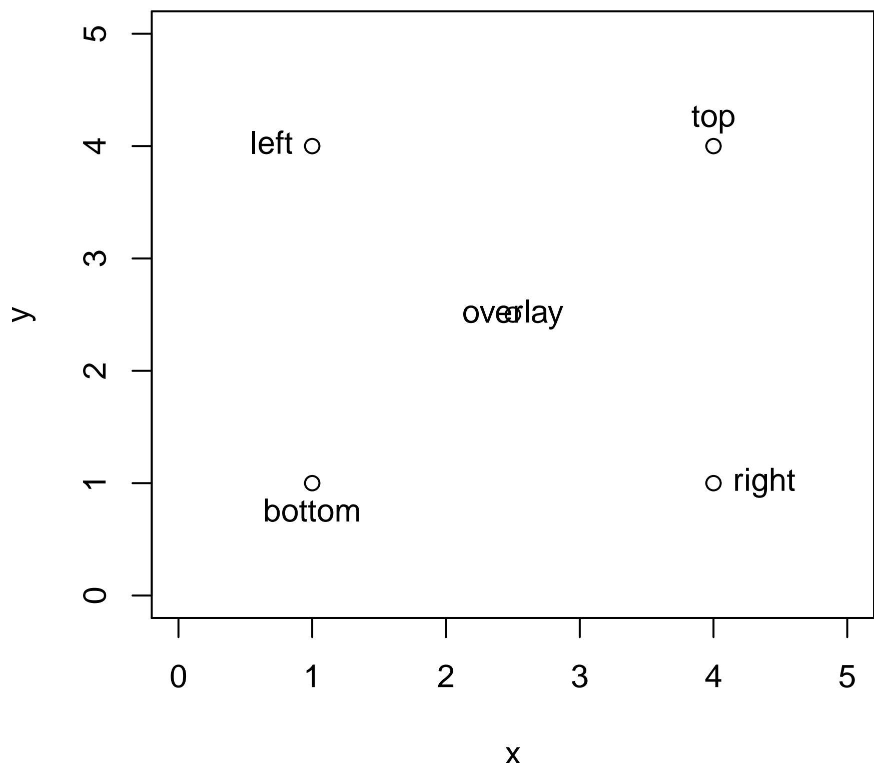


Adding text

Text can be added to the plotting region with `text()`; the function takes two arguments, giving the sets of $x - y$ coordinates where the text items are to be located. An additional character vector argument gives the respective string texts to be drawn. The `pos=` option has possible values 1, 2, 3, 4, giving the position of the text in relation to the coordinate point.

`text()` example

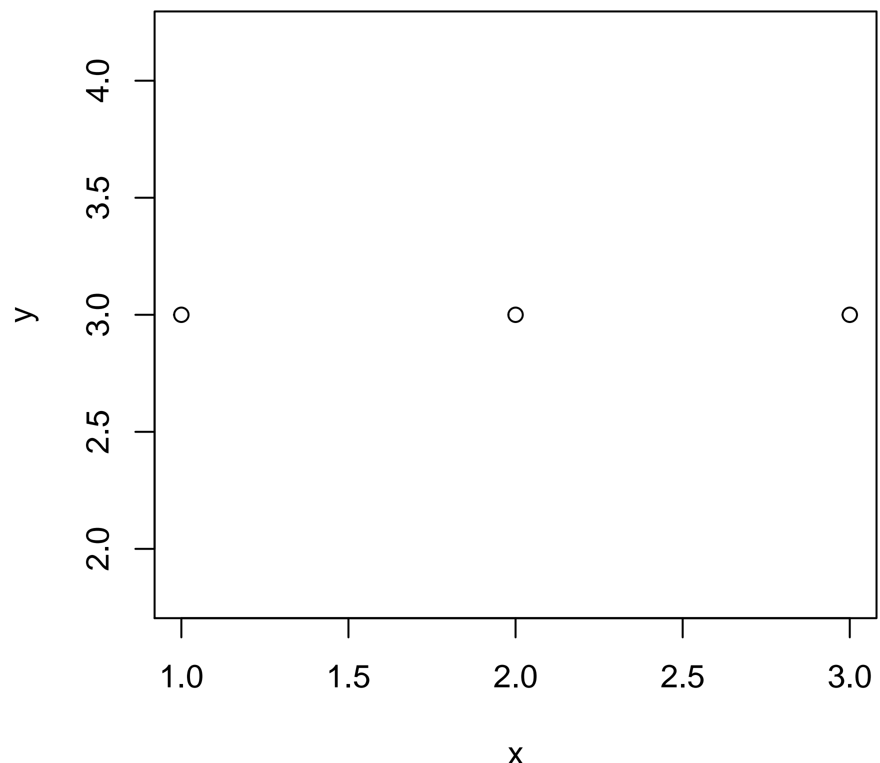
```
x=c(2.5,1,1,4,4)
y=c(2.5,1,4,4,1)
plot(x,y,xlim=c(0,5),ylim=c(0,5))
text(x[2:5],y[2:5],c('bottom','left','top','right'),pos=1:4)
text(x[1],y[1],c('overlay'))
```



Titles and subtitles

Plot titles and subtitles can be added with `title()`

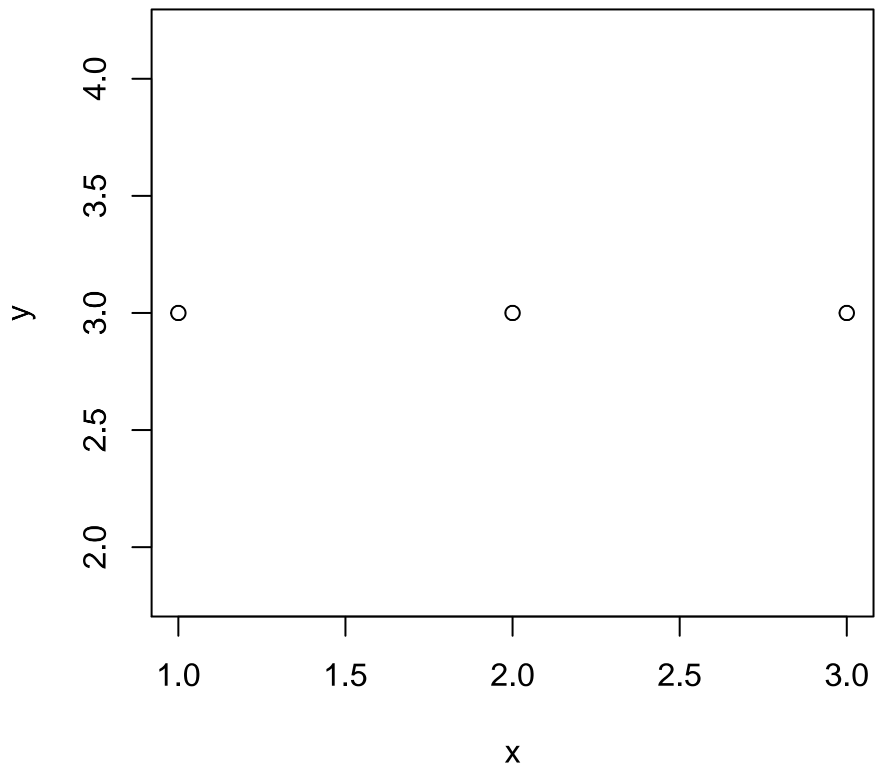
```
x=1:3; y=c(3,3,3)
plot(x,y)
title('wow what a plot')
```



Titles and subtitles

The same effect can be achieved with the `main=` option within the `plot()` function (and other (but not all) graphing functions)

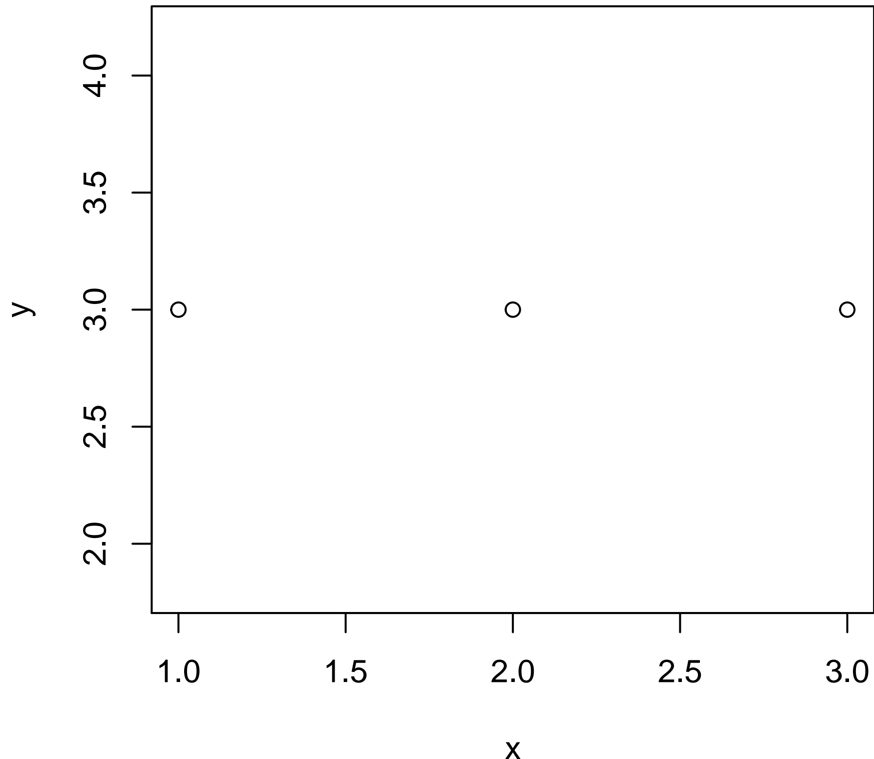
```
plot(x,y,main='wow what a plot')
```



Titles and subtitles

If the text argument in either case is a character vector, then a title with multiple lines will be produced

```
plot(x,y,main=c('wow','what a plot'))
```

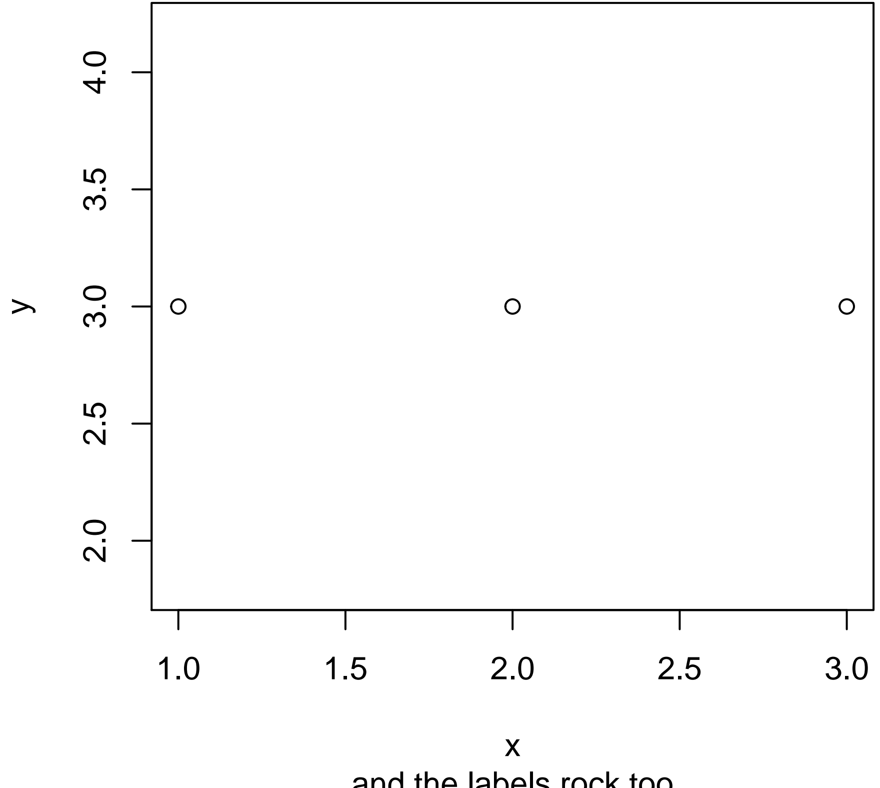


Titles and subtitles

If the text argument in either case is a character vector, then a title with multiple lines will be produced. The `sub=` option works as well in the `plot()` function. Using `title()` gives more flexibility, such as when an overall title is desired for a graph with multiple panels.

Titles and subtitles

```r
plot(x,y)
title(main='wow what a plot',sub='and the labels rock too')
```

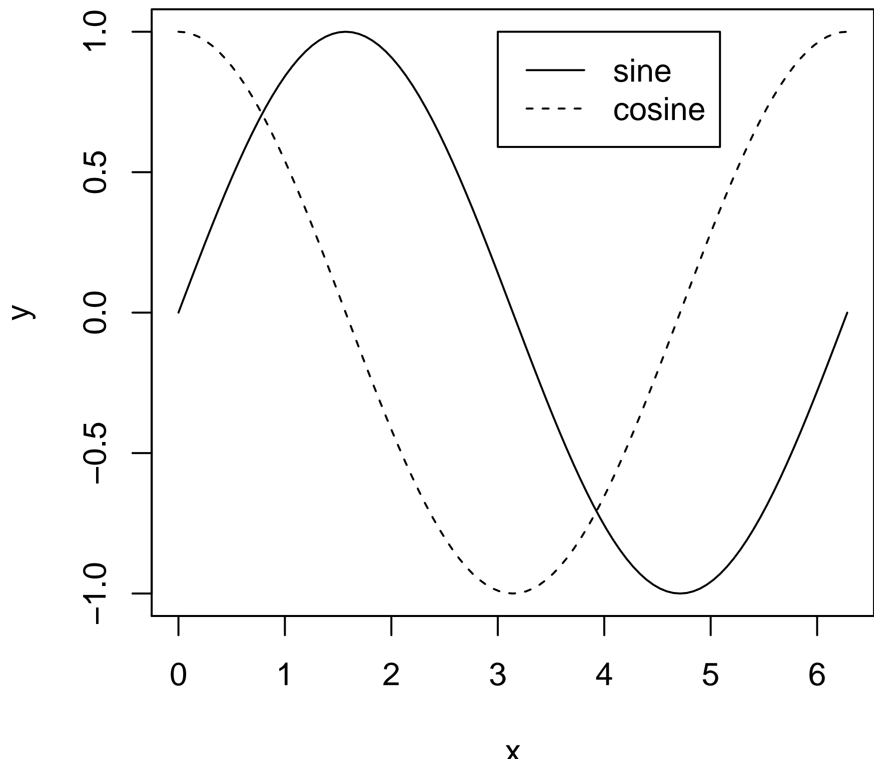


Legends

Legends can be added with `legend()`. The first two arguments are the x- and y-coordinates of the upper left of the legend box. The next arguments give the text strings for the legend entries and sets of plotting symbols that are to appear in the box, such as `pch=` and/or `lty=`. It is helpful to draw the plot first and then look for a nice open area for the position of the legend.

Legend

```
x=(0:100)*2*pi/100
y1=sin(x)
y2=cos(x)
y=cbind(y1,y2)
matplot(x,y,type='l',lty=c(1,2),col=1)
legend(3,1,c('sine','cosine'),lty=c(1,2))
```



Global and local options

When options are invoked in the `plot()`, `matplot()`, or other graphing functions, they are *local* to that function call (they will only be changed for the use of the function). All the default settings are resumed for other plots created in the same work session. The `par()` function (plotting parameters function) can be used to fix display settings *globally* for a sequence of plots.

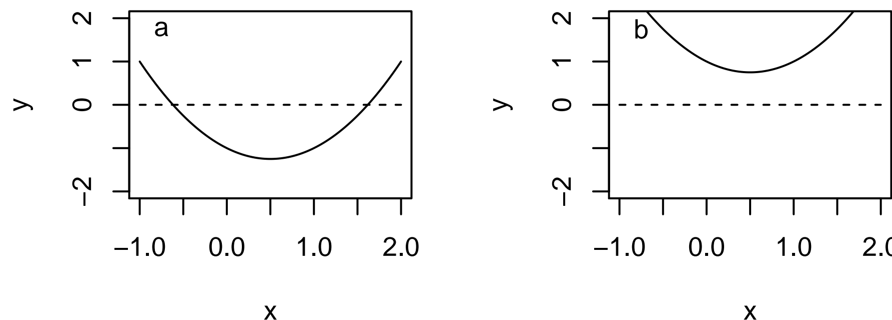
For example, say you want to keep the text size for all plots the same, or any other such change. The `par()` function will accept most of these types of arguments that are normally used in the plotting functions. When an option is subsequently changed in a plotting function, the change remains in effect only for that plot. The option then reverts to whatever was specified in `par()`.

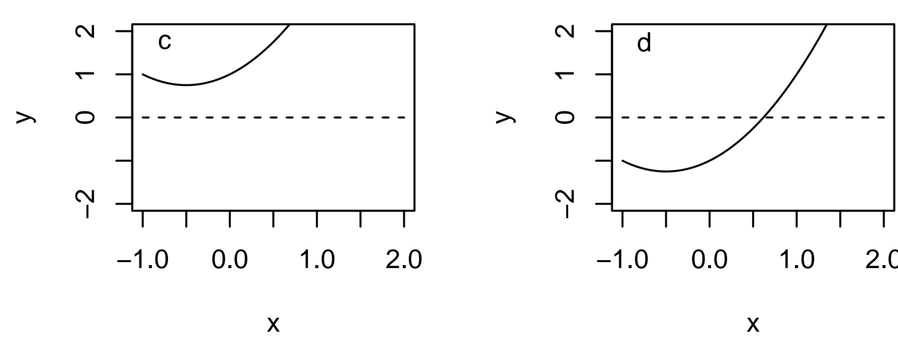
Multiple panels

A figure can be built with multiple panels, each panel having a different plot, as shown in plot with the different `type=` options. The construction can be accomplished a few different ways, one is the `layout()` function. The main argument is a matrix containing the integers $1, 2, \ldots, k$, where k is the number of panels to be drawn. The matrix indicates the position in the figure of each panel. The first plot is drawn in panel 1, the second in panel 2, and so on. My default, `matrix()` function reads in data by columns. The code for the example will not be shown in the lecture notes but will be in the Module 13 code on the class website.

It will include: `layout(matrix(c(1,2,3,4),2,2))`

Four quadratic functions



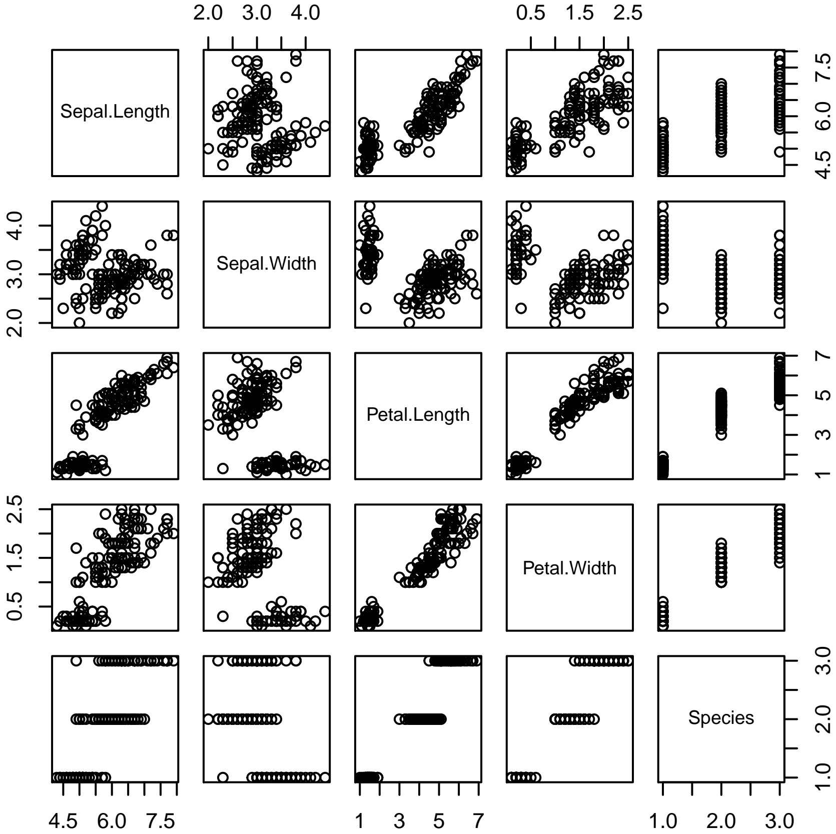


Scatterplot matrix

A scatterplot matrix is an array of scatterplots of every pair of quantitative variables in a dataset. The plot is good for exploring potential relationships between variables in complex data. The scatterplot matrix can be easily obtained with the `plot()` command. The argument must be a data frame of *all* quantitative variables.

Scatterplot matrix

```
plot(iris)
```



3-D plots

Two functions to discuss for 3-D plots are the surface (wire mesh) plots and contour plots. Surface plots works with a grid of $x - y$ values. Over each value of x and y, a value, for example, z, is recorded or calculated. The z-values can be thought of a heights in a landscape recorded at coordinates given by the combinations of x and y values. The plot links adjacent z-values with lines, as if the landscape is being covered with a flexible screen or wire mesh.

The `persp(x,y,z,...)` function needs three arguments: `x` and `y` are locations of grid lines where `z` will be measured, `z` is a matrix containing the values to be plotted

Surface plot

As an example, the sum of squares surface we saw last module. For a given dataset, the sum of the squared errors was minimized to find the intercept and slope corresponding to the "best" prediction line (fit). Each different pair of intercept and slope values gives a different sum of squares value. Visualize the least squares estimates of intercept and slope as the location of the lowest point in a "valley", with the elevation at any location in the valley given by the sum of squares value.

Old Faithful surface plot

```
set.seed(2) # so you get the same sample I do
OF=data.frame(faithful)
faith=OF[sample(nrow(OF),20),]
x=faith$eruptions; y=faith$waiting
```

Old Faithful surface plot

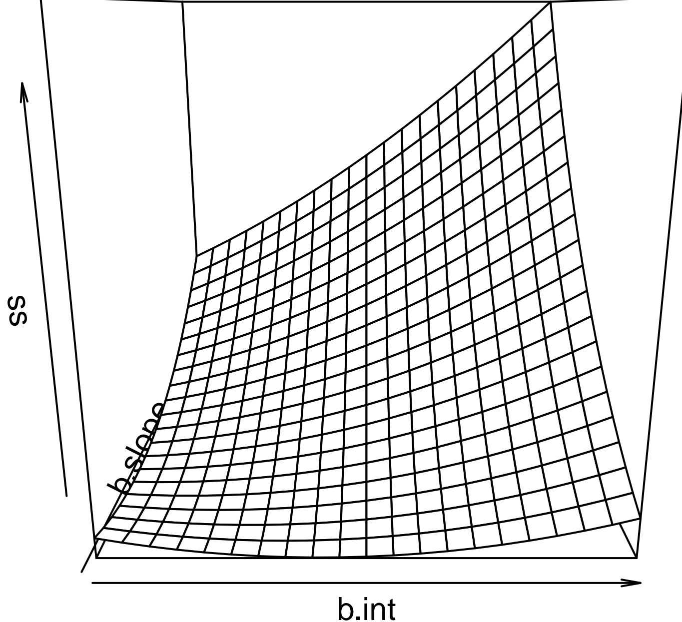
```
n=length(y) # sample size
X=matrix(1,n,2) # form X matrix with col1 all 1s
X[,2]=x # col2 has observations from predictor variable
```

Old Faithful surface plot

```
b.int=(0:20)*10/20+30 # range from 30-40
b.slope=(0:20)*4/20+10 # range from 10-14
ss=matrix(0,length(b.int),length(b.slope))
```

Old Faithful surface plot

```
for(i in 1:length(b.int)){
  for(j in 1:length(b.slope)){
    b=rbind(b.int[i],b.slope[j])
    ss[i,j]=sum((y-X%*%b)^2)
  }
}
persp(b.int,b.slope,ss)
```



Surface plot

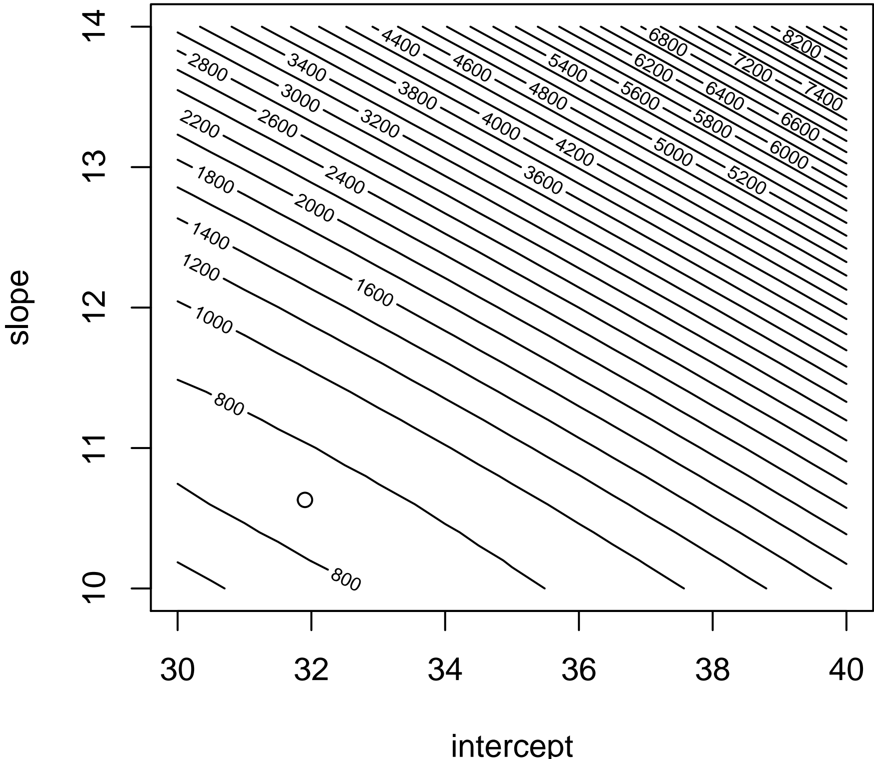
The landscape of the sum of squares looks to be a long and narrow valley with a nearly flat valley bottom. The least squares estimates of intercept and slope produce a sum of squares value only slightly lower than neighboring locales along the valley bottom.

Contour plot

Think of what a topographic map of the sum of squares valley might look like; that map is a contour plot. The function `contour()` takes the same type of data arguments as `persp()`.

Old faithful contour plot

```
contour(b.int,b.slope,ss,nlevels=30,xlab='intercept',ylab='slope')
bhat=solve(t(X)%*%X,t(X)%*%y)
points(bhat[1],bhat[2])
```



Color

Please, please, please, please, please... use it judiciously and sparingly in graphics. When color is used, it can detract from the main information that is attempting to be portrayed; the main issue is that color is a distraction. When symbols in a graph have different colors, the brain is searching for what the meaning of the color is. Additionally, a small but appreciable fraction of the population has varying degrees of colorblindness.

Colors

That being said, there are a collection of options that control colors in plotting functions.

`col=`: colors of data symbols, lines, and bars
`col.axis=`: color of axes
`col.lab=`: color of `x` and `y` labels
`col.main=`: color of main titles
`col.sub=`: color of subtitles

Colors can be specified, there are so many. Type `colors()` in the console to see 657 colors recognized by `R` in color options.

`ggplot()` syntax

To access the `ggplot()` functions, install the `ggplot2` package and load it. The syntax for `ggplot()` is a bit different than what you have grown accustomed to. The main difference is that, unlike with `plot()`, `matplot()`, etc, `ggplot()` works with data frames and not individual vectors. The data needed to create plots is usually within the data frame supplied to `ggplot()` itself or its respective `geoms` (more later).

General form of `ggplot()`

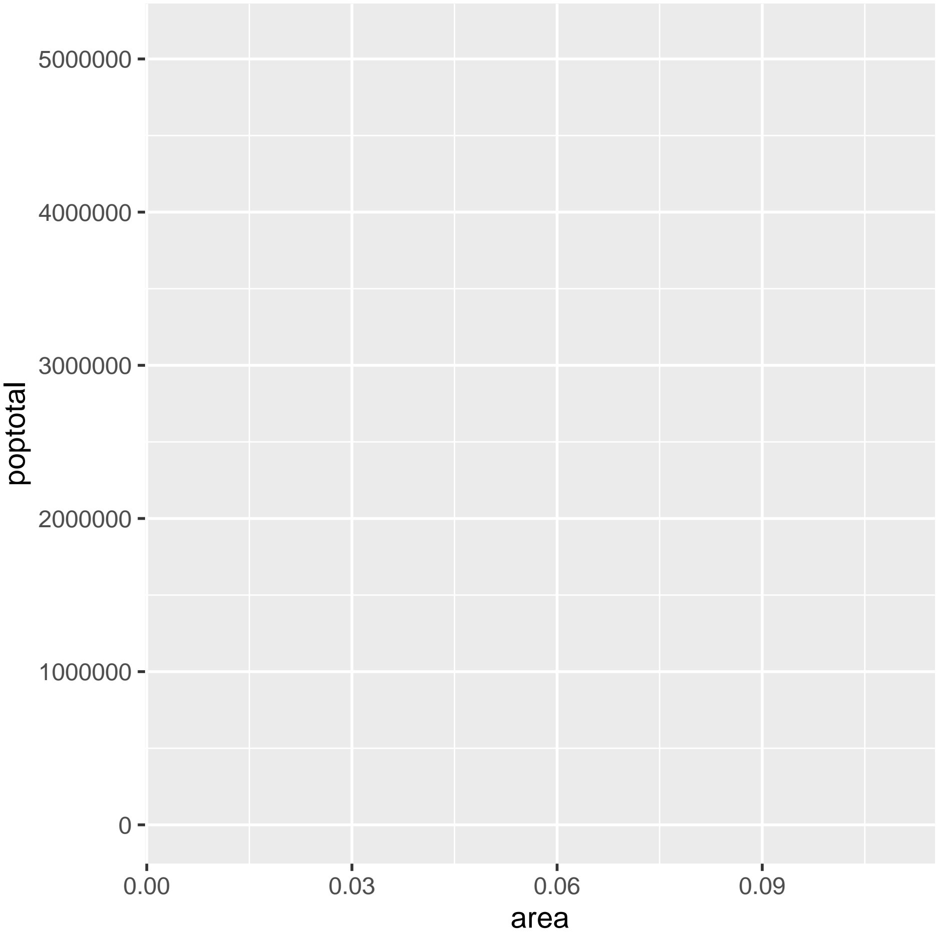
`ggplot()` initializes a ggplot object. It can be used to declare the input data frame for a graphic and to specify the set of plot aesthetics intended to be common throughout all subsequent layers unless specifically overridden.

`ggplot(data= ,mapping=aes(),...)`
`data=`: default dataset to use for plot; if not already a data.frame, it will be converted
`mapping`: default list of aesthetic mappings to use for plot; if not specified, must be supplied in each layer added to the plot
`...`: other options

Clear as mud, right? An interesting feature is that adding enhancements to the plot by adding more layers (and themes) to an existing plot created using the `ggplot()` function.

A first scatterplot

```
options(scipen=999)  # turn off scientific notation
library(ggplot2); midwest=read.csv("http://goo.gl/G1K41K")
ggplot(midwest,aes(x=area,y=poptotal)) # initialize ggplot
```



A first scatterplot

Where are the data points?!?

Notice what the note says on the line `ggplot(...)` "initialize ggplot". All that does, without adding more layers and such, is give a blank graph.

Another thing to note is the `aes` option specifies the columns to be used, but nothing at this point.

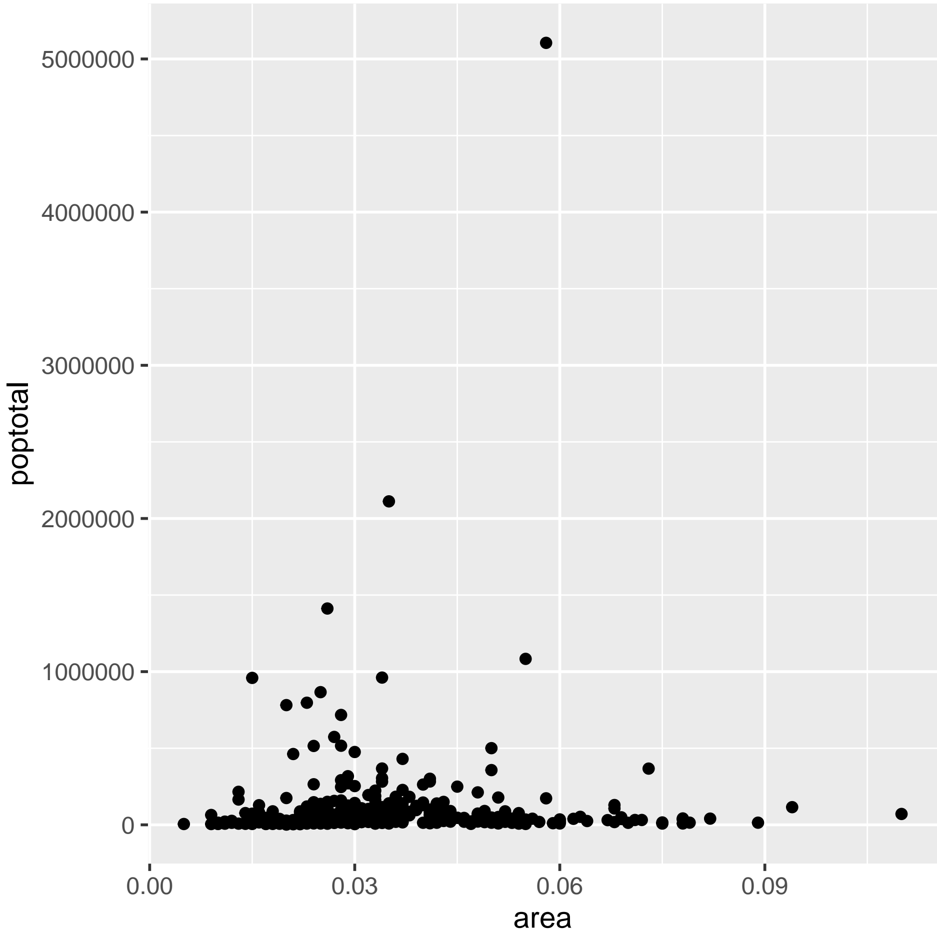
Adding a `geom` layer

Now add the scatterplot (the points from the data) on top of the blank graph with a `geom` layer called `geom_point`

```
ggplot(data= ,mapping=aes(),...)+
geom_point()
```

A first scatterplot with points

```r
ggplot(midwest,aes(x=area,y=poptotal))+ # initialize ggplot
  geom_point()
```



Adding a line of best fit

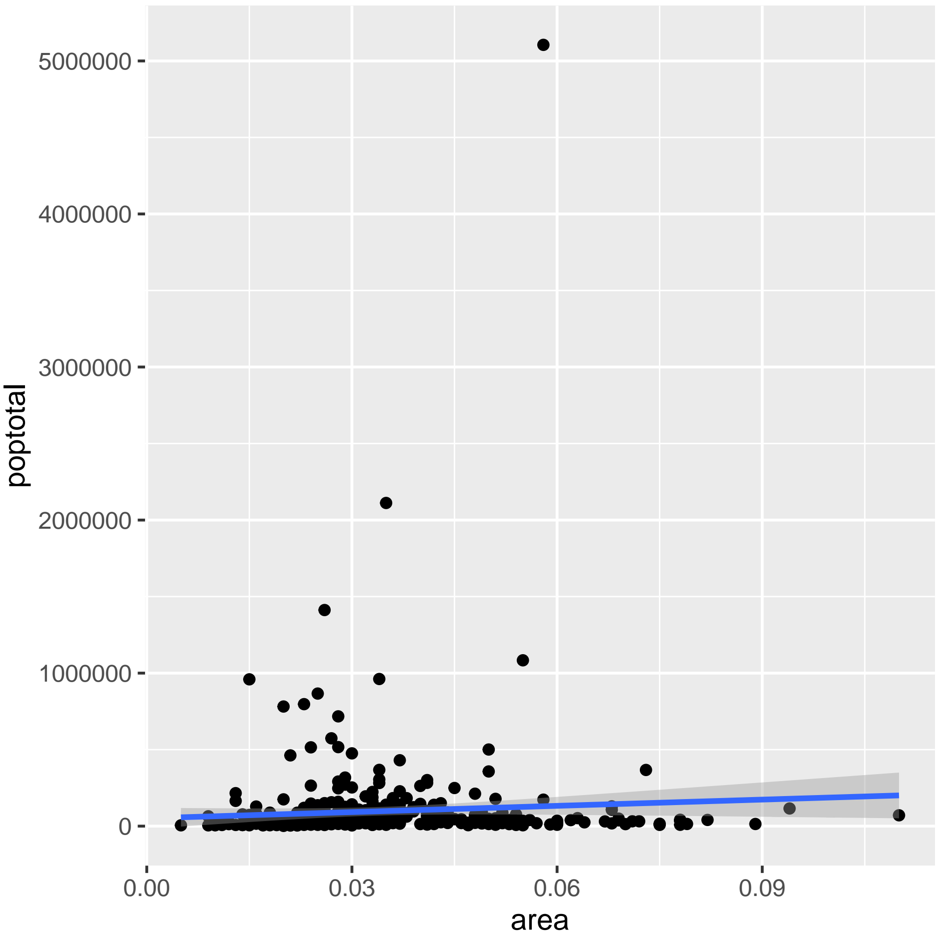
Adding a line from a model can be done with `geom_smooth(method=)` with `lm` for linear model. By default the line with also have confidence bands shaded in around the line; take them off with the `se=F` option in the `geom_smooth()` argument.

```
ggplot(data= ,mapping=aes(),...)+
geom_point()+
geom_smooth(method='lm')
```

Many times, it is easier to create an object with the arguments and then plot the object, but it is up to you.

A first scatterplot with points and best fit line

```
g=ggplot(midwest,aes(x=area,y=poptotal))+geom_point()+geom_smooth(method='lm')
plot(g)
```



Modifying axes limits

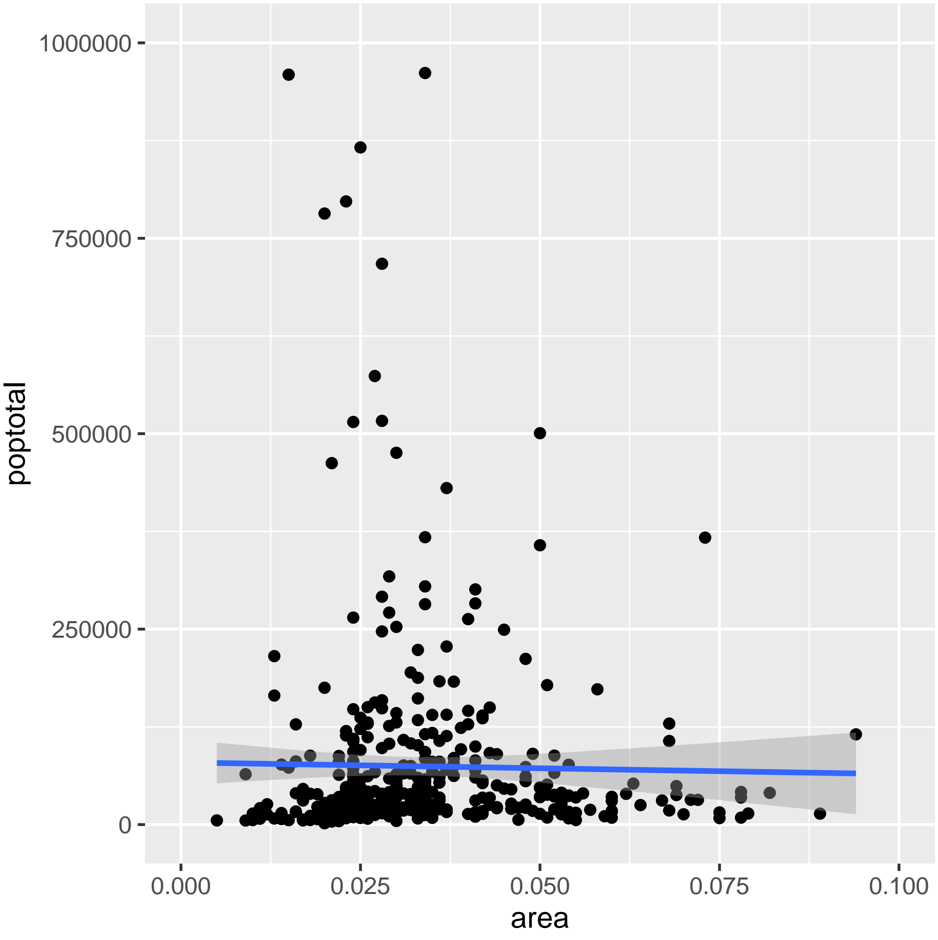
The x and y axes can be modified in two ways. The first way is similar to the way `plot()` and other base graphs options with `xlim` and `ylim`. The second way uses `coord_cartesian(xlim=,ylim=)`

```
g=ggplot(data,aes())+geom_point()+
geom_smooth(method='lm')
g+xlim(c(a,b))+ylim(c(c,d))
```

```
g=ggplot(data,aes())+geom_point()+
geom_smooth(method='lm')
g1=g+coord_cartesian(xlim=c(a,b)+ylim=c(c,d))
plot(g1)
```

A first scatterplot also with axes limits I

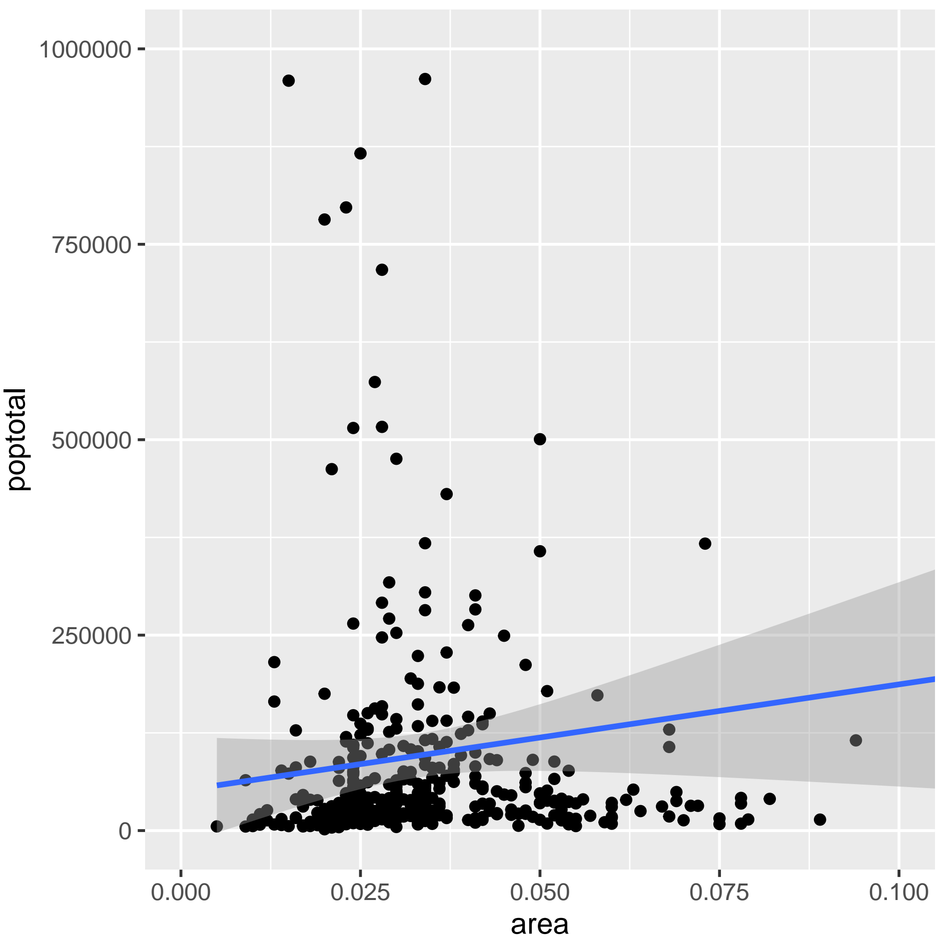
```
g=ggplot(midwest,aes(x=area,y=poptotal))+geom_point()+geom_smooth(method='lm')
g+xlim(c(0,0.1))+ylim(c(0,1000000)) # will delete points outside this
```




```
# also changes slope of line
```

A first scatterplot also with axes limits II

```
g1=g+coord_cartesian(xlim=c(0,0.1),ylim=c(0,1000000)) # zooms in
plot(g1)
```



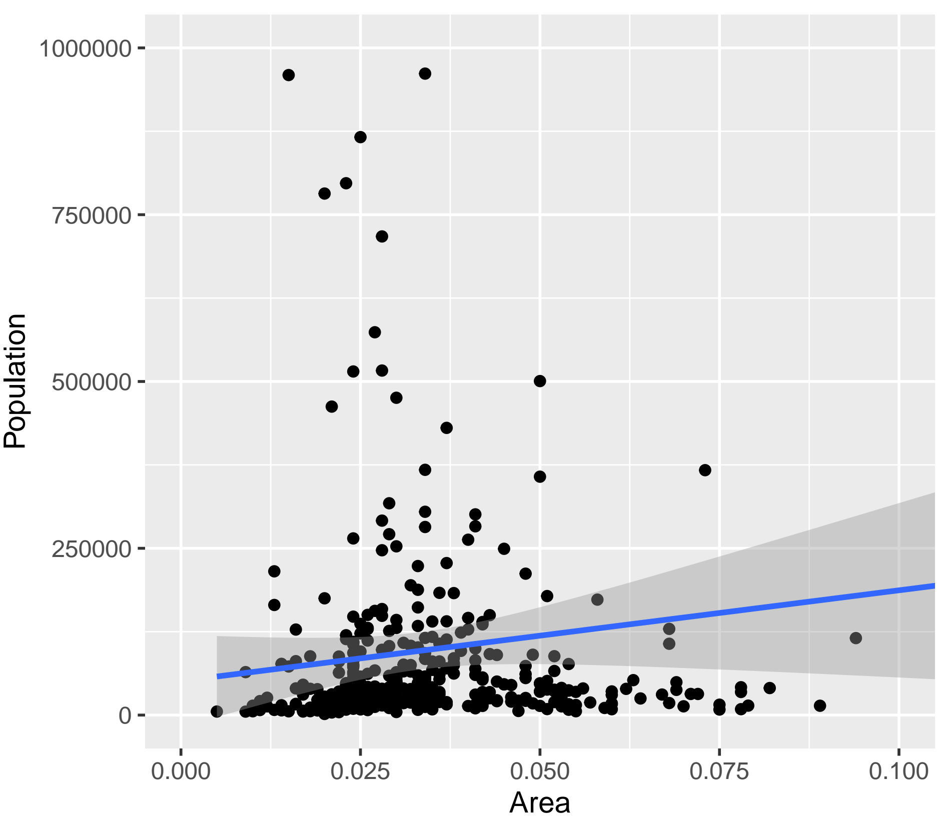
Title and axis labels

The function `labs(title='',subtitle='',y='',x='',caption='')` can be used to add a title, subtitle, x- and y-axis labels, and a caption.

```
g=ggplot(data,aes())+geom_point()+
geom_smooth(method='lm')
g1=g+coord_cartesian(xlim=c(a,b)+ylim=c(c,d))
g1+labs(title='',subtitle='',y='',x='',caption='')
```

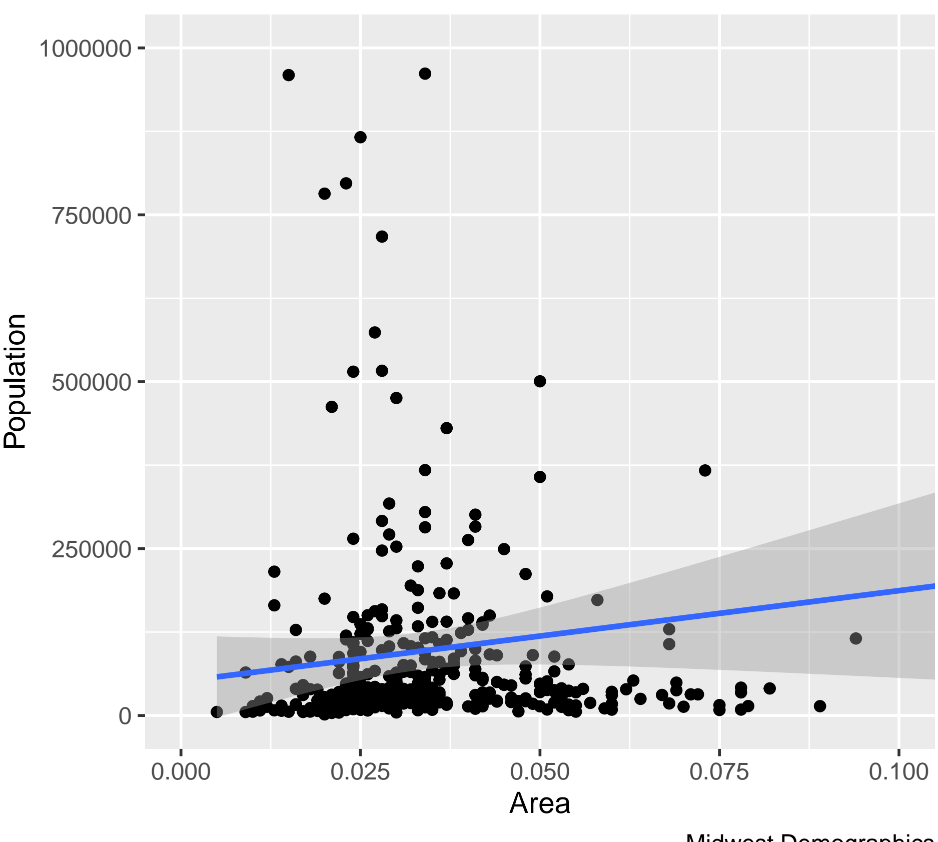
A first scatterplot also with titles limits I

```
g1=g+coord_cartesian(xlim=c(0,0.1),ylim=c(0,1000000))
g1+labs(title='Area Vs Population',subtitle='From midwest dataset',y='Population',
        x='Area',caption='Midwest Demographics')
```



Full function call

```
ggplot(midwest,aes(x=area,y=poptotal))+
  geom_point()+
  geom_smooth(method='lm')+
  coord_cartesian(xlim=c(0,0.1),ylim=c(0,1000000))+
  labs(title='Area Vs Population',subtitle='From midwest dataset',y='Population',
       x='Area',caption='Midwest Demographics')
```

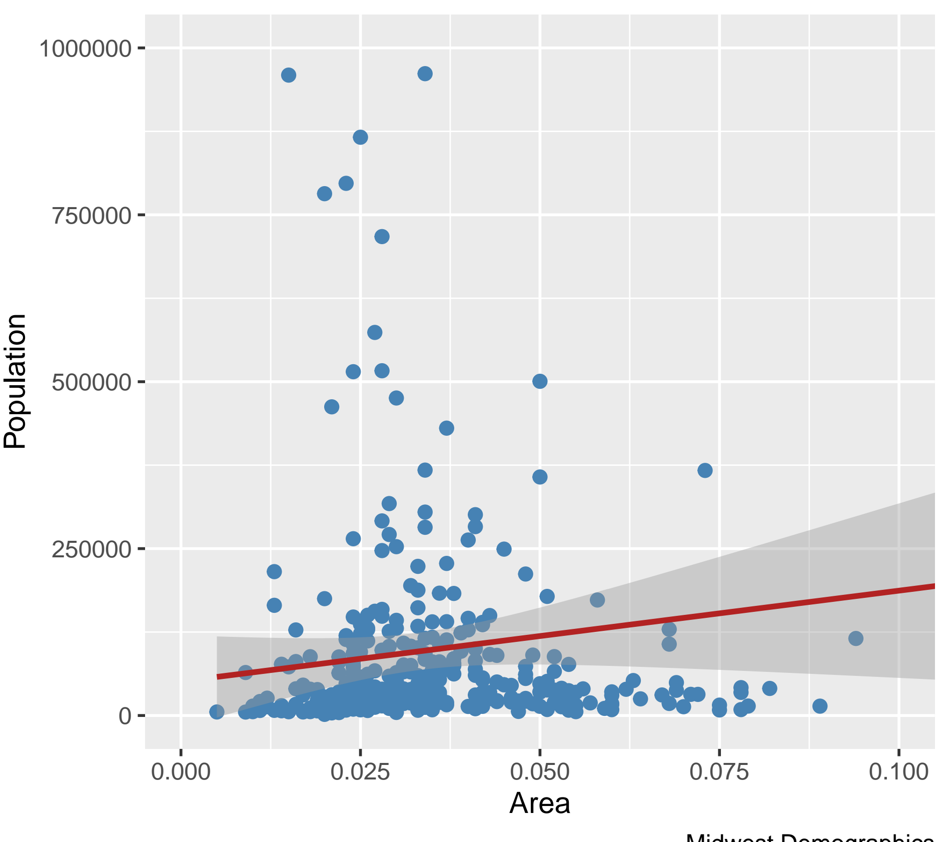


Colors and point sizes

Colors and point sizes can be modified in the `geom_point()` argument with `geom_point(col=,size=)`. The `col` and `size` options can be used again within the `geom_smooth()` layer. Colors can be static or dynamic (set or vary by categories).

A first scatterplot with static colors and point sizes

```
ggplot(midwest,aes(x=area,y=poptotal))+
  geom_point(col='steelblue',size=2)+
  geom_smooth(method='lm',col='firebrick')+
  coord_cartesian(xlim=c(0,0.1),ylim=c(0,1000000))+
  labs(title='Area Vs Population',subtitle='From midwest dataset',y='Population',
        x='Area',caption='Midwest Demographics')
```

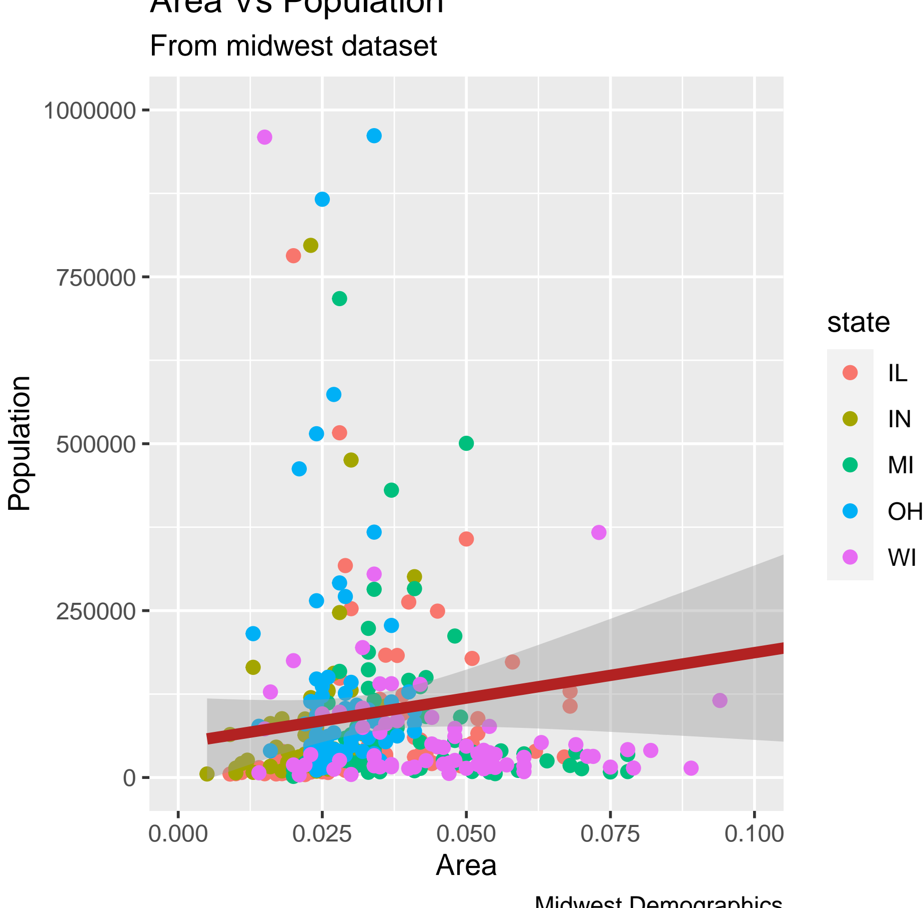


A first scatterplot with changing colors

If colors are desired to vary based on another column in the source dataset (`data`), it must be specified within the `aes()` function within `geom_point()`. Each point is colored on the level of the column it belongs because of `aes(col=)`; additionally along with `col`, `size`, `shape`, `stroke` (thickness of boundary), and `fill` (fill color) can be used to distinguish between groupings.

A first scatterplot with static colors and point sizes

```
g2=ggplot(midwest,aes(x=area,y=poptotal))+
  geom_point(aes(col=state),size=2)+
  geom_smooth(method='lm',col='firebrick',size=2)+
  coord_cartesian(xlim=c(0,0.1),ylim=c(0,1000000))+
  labs(title='Area Vs Population',subtitle='From midwest dataset',y='Population',
        x='Area',caption='Midwest Demographics')
plot(g2)
```

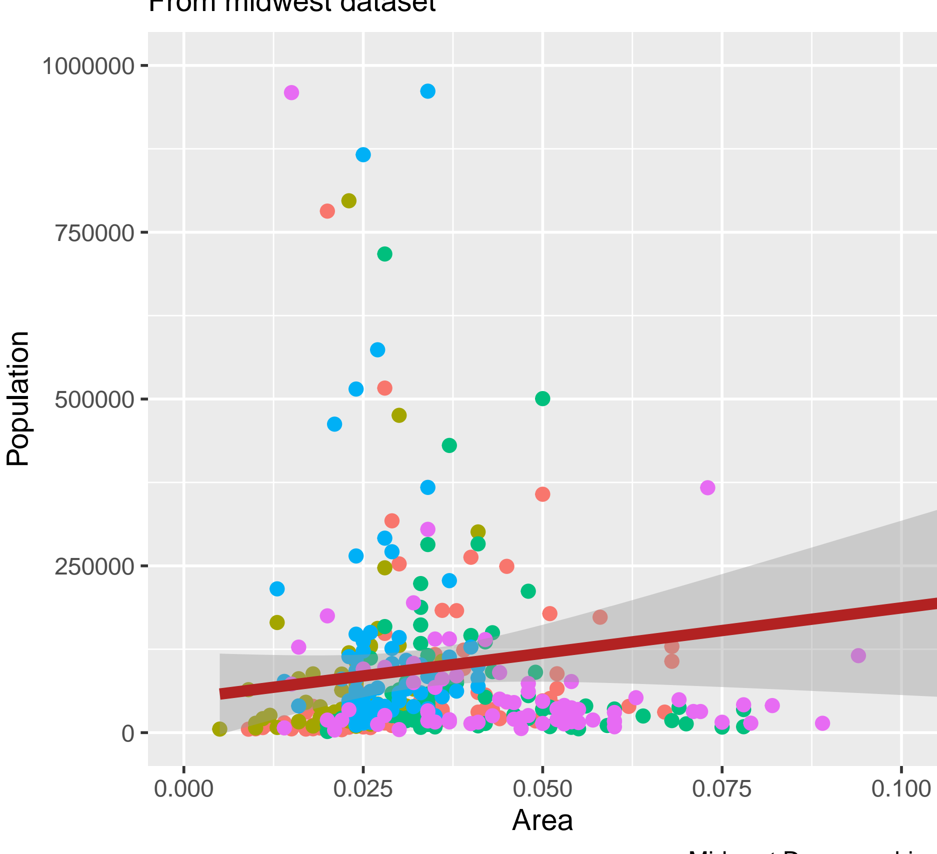


Legend and color palette

The legend is automatically added. It can be removed by setting the `legend.position` to `None` within a `theme()` function. The color palette can be changed out entirely (web search for R color brewer palettes or `library(RColorBrewer) head(brewer.pal.info, 10)` will show the first 10 palettes).

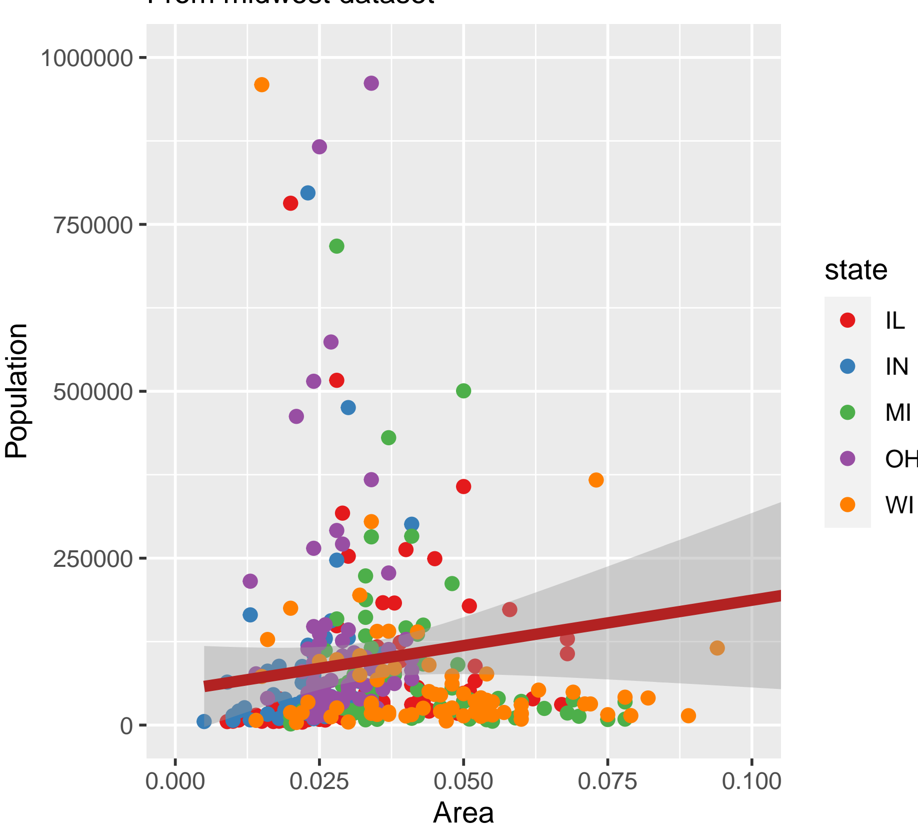
Legend suppression

```
g2+theme(legend.position='None')
```



Change color palette

```
g2+scale_colour_brewer(palette='Set1')
```



More. . . so much more

There are so many tutorials on YouTube and various websites that would be more than worth your time to look at if you want to learn more about `ggplot()`. This lecture should be a decent way to get you started and a bit more comfortable with its syntax.