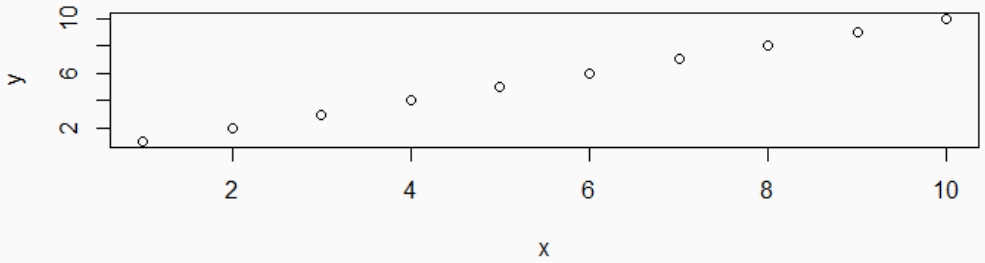
graphics in R

The following content focuses on the graphical functions in the ***base R graphics package***.

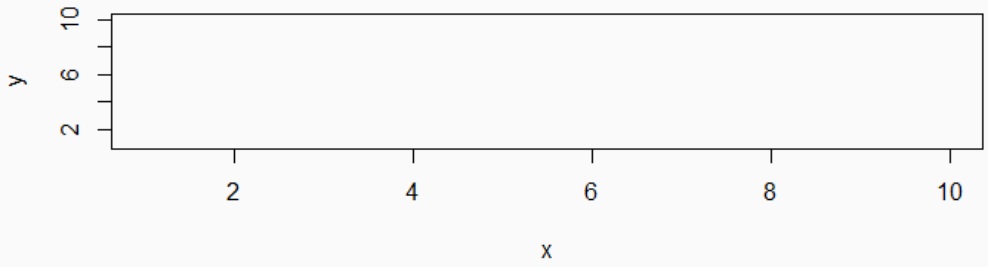
The **plot( )** function serves as the foundation for many of the R graphics capabilities. The **plot( )** function is a *generic* R function; a placeholder for a family of different functions called based on the object class.

```
1   > x <- 1:10
2   > y <- 1:10
3   > plot(x,y)
```



Note that the **plot( )** functions operates in stages; a graph can be built up in iterative stages by issuing a series of commands. For example, an empty base graph can be drawn that only contains the axis':

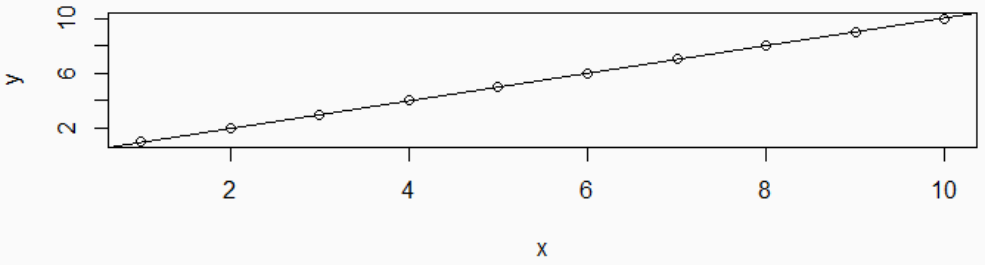
```
1   > plot(x,y, type = "n",
2   +      xlab = "x",
3   +      ylab = "y")
```



adding lines to plots in R

The **abline( )** function is used to add a line to the existing graph in the preceding illustration:

```
1   > plot(x,y)
2   > lmout <- lm(y ~ x)
3   > abline(lmout)
```

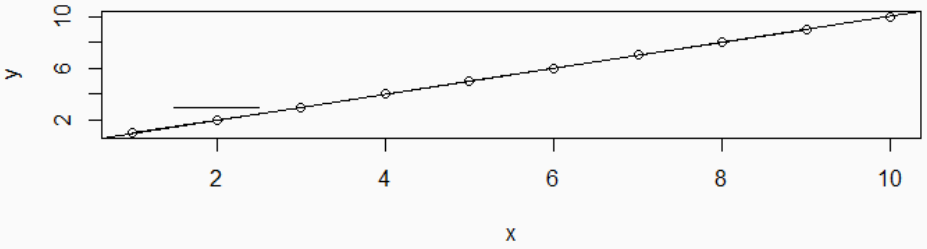


The above line is fitted to the graph based on the call to **lm( )** as a class instance containing the slope and intercept of the line. The regression line was assigned to the **lmout** variable and passed through **lm( )**.

Concretely, the **abline( )** functions takes the slope and intercepts computed by **lm( )** and specifically the **lmout$coefficients** feature from the prior regression function; superimposing the line onto the graph.

Additional lines can be included in the graph through the **lines( )** function; with the arguments of the **lines( )** function being a vectors of *x-values* and a vector of *y-values*:

```
1   >  lines(c(1.5,2.5), c(3,3))
```

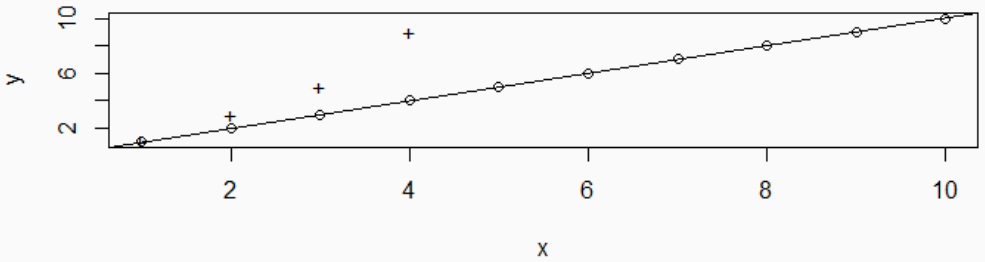


The lines can "connect the dots" through the use of the **type = 1** argument in the **plot( )** or **line( )** functions. Additionally, the **lty** parameter in **plot( )** can be used to specify the line type. The available types of lines in R's **plot( )** function can be examined by executing the **help(par)** function in the console.

adding points to plots in R

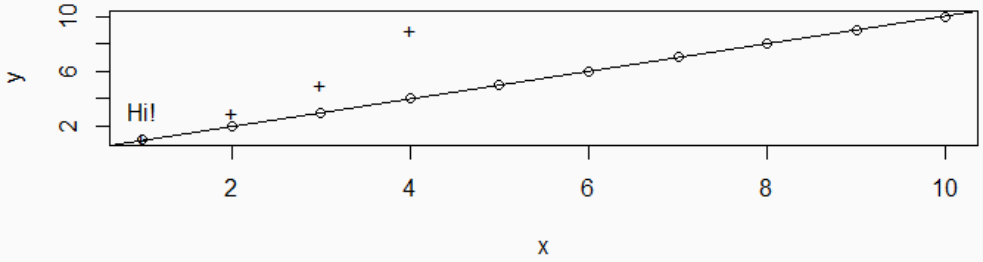
The **points( )** function adds a set of **(x,y)** and respective labels to the currently displayed graph.

```
1   > plot(x,y)
2   > abline(lmout)
3   > points(y1, pch = "+")
```



The **text( )** function is used to add text to a specified coordinate in the plot:

```
1   > text(3, "Hi!")
```



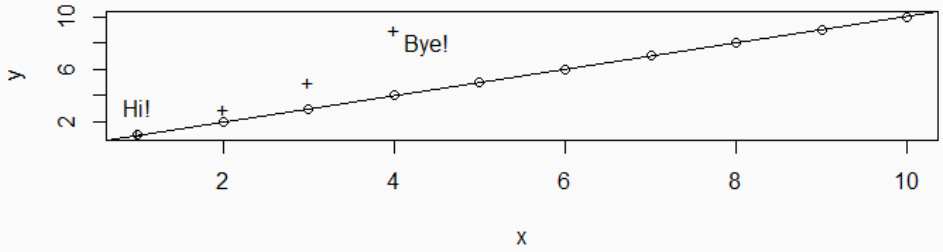
Similarly, the **legend( )** function can be called to add a legend; specified by the function arguments.

As illustrated by running the **text( )** function above, the exact location of text placement can sometimes be difficult. The solution to the latter is to utilize the **locator( )** function. The function works be converting the cursor to a crosshair, allowing the user to select any point on the graph and return the **(x,y)** coordinates:

```
1   > locator(1)  #the 1 specifies a single point to be selected and returned from the plot
2   $x
3   [1] 3.994866
4
5   $y
6   [1] 8.823904
```

Equally, the **locator( )** function can be used to place text as follows:

```
1   > text(locator(1), "Bye!")
```



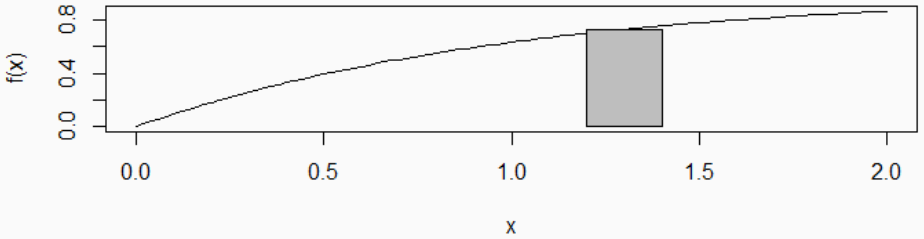
customizing graphs in R

The size of text printed to the graph can be specified by passing the **cex =** argument through **text( )**.

The **xlim( )** and **ylim( )** arguments within the **plot( )** or **points( )** function can be called to specify the axis'. Note that if the largest/longest graph/line is initially plotted first, the graph will properly display all others.

The **polygon( )** function draws arbitrary polygonal objects; below draws $f(x) = 1 - e^{-x}$ and adds a shape:

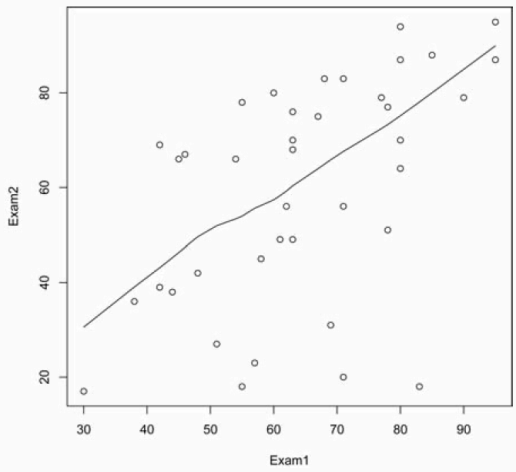
```
1   > f <- function(x)
2         return(1-exp(-x))
3   curve(f,0,2)
4   polygon(c(1.2,1.4,1.4,1.2),
5           c(0,0,f(1.3),
6             f(1.3)),
7           col="gray")
```



smoothing points in R

datasets plotted on graph are typically smoothed during the statistical analysis process. This is typically achieved through fitting a ***nonparametric regression estimator*** to the data. In R, the **lowess( )** and more recent **loess( )** functions apply regression smoothing to the data passed through their arguments.

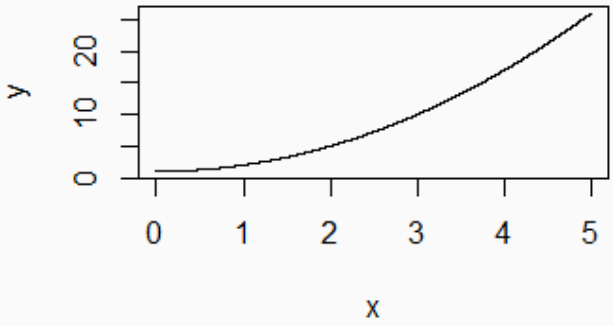
```
> plot(testscores)
> lines(lowess(testscores))
```



graphing explicit functions in R

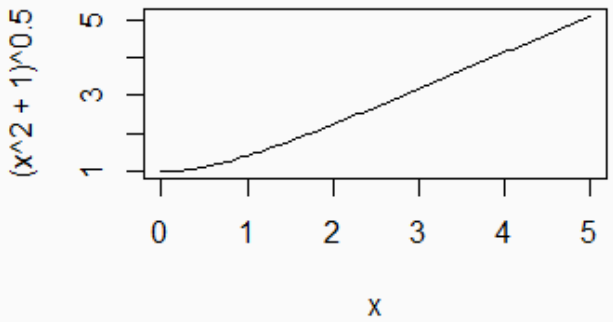
Given the function $g(t) = (t^2 + 1)^{-0.5}$ for t between **0** and **5**:

```
> g <- function(t) {
    return (t^2+1)^0.5 } # define g()
x <- seq(0,5,length=10000)
# x = [0.0004, 0.0008, 0.0012,..., 5]
y <- g(x)
# y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]
plot(x,y,type="l")
```



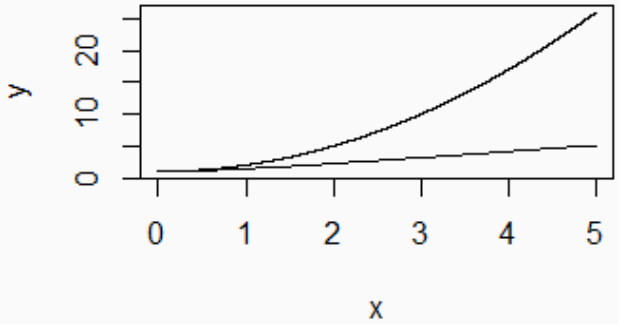
Note the **curve( )** function can be used in substitution of the verbose syntax above:

```
> curve((x^2+1)^0.5,0,5)
```



If the **curve( )** is to be added to an existing plot, the following code will do so along with visually smoothing:

```
> curve((x^2+1)^0.5,0,5, add = TRUE)
```



graphics devices in R

R graphic displays consist of various **graphic devices**. The screen is the default device, RStudio will be the graphic device if that IDE is being used. In order to **save** a graph to a file, an additional device is set up.

To open a file

```
> pdf("file")
```

To view open devices in R

```
> dev.list()
RStudioGD       png       pdf
        2         3         4
```

To view the currently active device in R

```
> dev.cur()
pdf
  4
```

To save displayed graphs in R

One method of saving the currently displayed graph in R is to copy the image to the open **pdf** device:

```
> dev.set(2)
RStudioGD
        2

> dev.copy(which = 4)
pdf
  4
```

The above is executed by reestablishing the screen as the current device and copying it to the **pdf** device.

However, it is best to set up the **pdf** device prior to running the graphics code, or rerunning prior to saving.

To close a graphics device in R

The **pdf** device must be closed prior to interacting with it outside of R.

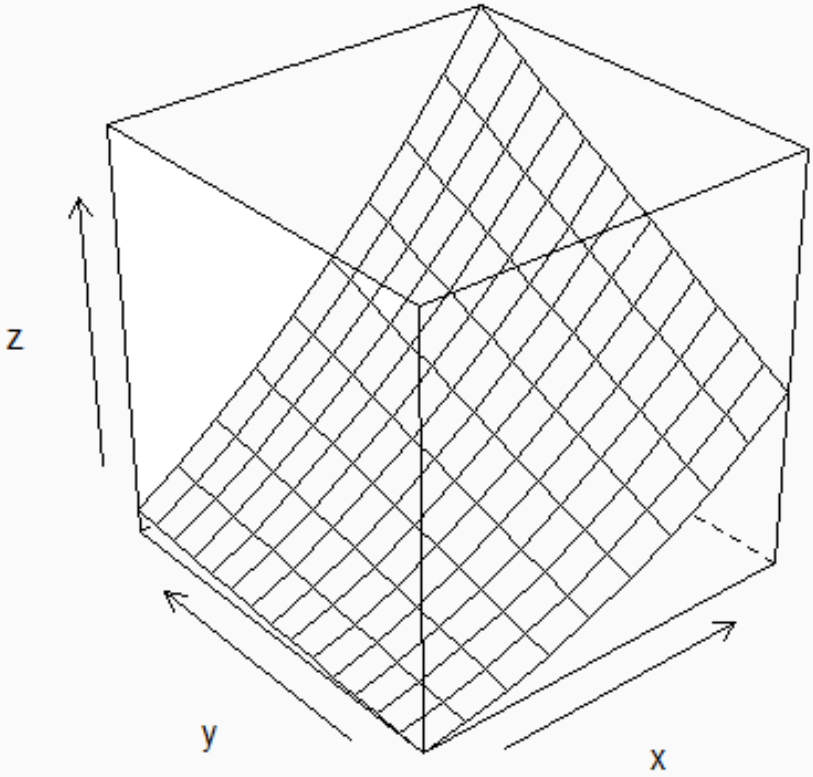
```
> dev.set(4)
pdf
  4

> dev.off()
RStudioGD
        2
```

The device an equally be closed through exiting the R session; it is best practice to manage devices through commands rather than assuming a closed session of R will reset to default devices.

creating three-dimensional plots in R

The are many function within R to plot 3-deminsional plots; the **persp( )** and **wireframe( )** functions draw surfaces; and the **cloud( )** function draws three-dimensional scatter plots. The following illustration plots a 3D surface plot using the **wireframe( )** function as a part of the **lattice** library.

```
> library(lattice)
> a <- 1:10
> b <- 1:15
> eg <- expand.grid(x=a,y=b)
> eg$z <- eg$x^2 + eg$x * eg$y
> wireframe(z ~ x+y, eg
```



In the illustration, the **expand.grid( )** function creates a 2-column dataframe **(x,y)**; will all possible combinations of the two input values. With **a = 10** and **b = 15**, the dataframe will be a total of **150 rows**. Column **z** is added as a function of columns **a** and **b**. The **wireframe( )** function is called to create the graph in regression model form; stating that **z** is to be graphed against **x** and **y**. The **wireframe( )** function connects all of the points, like a *surface***.** In contrast, the **cloud( )** function plots isolated points. Note that the **(x,y)** pairs passed through **wireframe( )** must form a rectangular grid.