MODULE-1
Data Visualization

Data Preparation

Before you can visualize your data, you first have to import the data from an external source into the R environment.

Importing data

- R is providing the flexibility to import data from almost any source, including text files, excel spreadsheets, statistical packages, and database management systems.
- After downloading the data we have to store it in the R folder where you will be storing the .R files to load into the R environment.

- **Text files-** Using **readr** Package

➢The **readr** package is developed by Hadley Wickham to deal with reading in large flat files.

➢The package provides replacements for functions like **read.table()** and **read.csv().**

➢This can be very helpful for “debugging” problems with your data before you get deep in data analysis.

➢R’s readr package provides an easy and flexible way to import all kinds of data into R environment.

➢You can load the readr package with the following command

```
install.packages("readr")
library(readr)
```

readr read functions in R:

Readr has six read functions that support reading data from six different file formats. These are as follows:

- **read_csv():** CSV files
- **read_tsv():** Tab-separated files
- **read_delim():** General delimited files
- **read_fwf():** Fixed-width files
- **read_table():** Tabular files where columns are separated by white space
- **read_log():** Web log files

Reading CSVs

- The best way to read data from a CSV file is to use **read.csv** with the sep argument preset to a comma (,). The result of using **read.csv** is a data.frame.
- **read.csv()** is used for reading "comma separated value" files
- Even though CSVs can hold numeric, text, date and other types of data, it is actually stored as text and can be opened in any text editor.
- The first argument to **read.table** is the full path of the file to be loaded. The file can be sitting on disk or even the Web.

  Syntax: read.csv(file, header = TRUE, sep = ",", dec = ".", …)

  where **file:** the path to the file containing the data to be imported into R.

  header: logical value. If TRUE, read.csv() assumes that your file has a header row, so row 1 is the name of each column. If that's not the case, you can add the argument header = FALSE.

  sep: the field separator character

  dec: the character used in the file for decimal points.

Similarly with **read.csv2()** and **read_csv()**

Example: # R program to read a file in table format

```
# Using read.csv()
myData = read.csv("basic.csv" , header = TRUE, sep = “,”)
print(myData)
```

Output:

	Name	Age	Qualification	Address
1	Amiya	18	MCA	BBS
2	Niru	23	Msc	BLS
3	Debi	23	BCA	SBP
4	Biku	56	ISC	JJP

Reading TSVs:

- The read_tsv is quite similar to the read_csv except for the fact that it is used to import tab-separted values.
- This method is also used for to read a tab separated (“\t”) values by using the help of **readr** package.

Syntax: read_tsv(file, col_names = TRUE)

where **file:** the path to the file containing the data to be read into R.

col_names: Either TRUE, FALSE, or a character vector specifying column names. If TRUE, the first row of the input will be used as the column names.

Example: # R program to read text file

```
# using readr package
# Import the readr library
library(readr)
# Use read_tsv() to read text file
myData = read_tsv(“data.tsv", col_names = FALSE)
print(myData)
```

read.delim():

- This method is used for reading "tab-separated value" files (".txt"). By default, point (".") is used as decimal points.
- The read_delim method can be used in cases where you have to read files that aredelimited by something other than comma or tab and when there is a need to specify the delimiter explicitly.

  Syntax: read.delim(file, header = TRUE, sep = "\t", dec = ".", …)

  where **file:** the path to the file containing the data to be read into R.

  header: a logical value. If TRUE, read.delim() assumes that your file has a header row, so row 1 is the name of each column. If that's not the case, you can add the argument header = FALSE.

  sep: the field separator character. "\t" is used for a tab-delimited file.

  dec: the character used in the file for decimal points.

Example: # R program reading a text file

```
# Read a text file using read.delim()
myData = read.delim("data.txt", header = FALSE)
print(myData)
```

or

```
myData=read_delim(“data.txt”, delim=“|”, header=FALSE)
library(readr)
# import data from a comma delimited file
Salaries <- read_csv("salaries.csv")
# import data from a tab delimited file
Salaries <- read_tsv("salaries.txt")
```


```
library(readr)
# import data from a comma delimited file
Salaries <- read_csv("salaries.csv")
# import data from a tab delimited file
Salaries <- read_tsv("salaries.txt")
```

```
"rank","discipline","yrs.since.phd","yrs.service","sex","salary"
"Prof","B",19,18,"Male",139750
"Prof","B",20,16,"Male",173200
"AsstProf","B",4,3,"Male",79750
"Prof","B",45,39,"Male",115000
"Prof","B",40,41,"Male",141500
"AssocProf","B",6,6,"Male",97000
```

- These function assume that the first line of data contains the variable names, values are separated by commas or tabs respectively, and that missing data are represented by blanks.

Excel spreadsheets

- The **readxl** package can import data from Excel workbooks. Both .xls and .xlsx formats are supported.
- **library**(readxl)
- *# import data from an Excel workbook*
- Salaries <- **read_excel**("salaries.xlsx", sheet=1)
- Since workbooks can have more than one worksheet, you can specify the one you want with the sheet option.
- The default is sheet=1.

Statistical packages

- The **haven** package provides functions for importing data from a variety of statistical packages.

library(haven)

 # import data from Stata

Example: Salaries <- **read_dta**("salaries.dta")

 # import data from SPSS

Salaries <- **read_sav**("salaries.sav")

 # import data from SAS

Salaries <- **read_sas**("salaries.sas7bdat")

Databases

- Importing data from a database requires additional steps.
- Depending on the database containing the data, the following packages can help: RODBC, RMySQL, ROracle, RPostgreSQL, RSQLite, and RMongo.
- In the newest versions of RStudio, you can use the Connections pane to quickly access the data stored in database management systems.

Cleaning data

- The process of cleaning your data can be the most time-consuming part of any data analysis. The most important steps are considered below. While there are many approaches, those using the dplyr and tidyr packages are some of the quickest and easiest to learn.

Package	Function	Use
dplyr	select	select variables/columns
dplyr	filter	select observations/rows
dplyr	mutate	transform or recode variables
dplyr	summarize	summarize data
dplyr	group_by	identify subgroups for further processing
tidyr	gather	convert wide format dataset to long format
tidyr	spread	convert long format dataset to wide format

Selecting variables

- The select function allows you to limit your dataset to specified variables (columns).

```
library(dplyr)
# keep the variables name, height, and gender
newdata <- select(starwars, name, height, gender)
# keep the variables name and all variables between mass and species inclusive
newdata <- select(starwars, name, mass:species)
# keep all variables except birth_year and gender
newdata <- select(starwars, -birth_year, -gender)
```

- Note: The dataset provides descriptions of 87 characters from the Starwars universe on 13 variables.

Selecting observations

- The filter function allows you to limit your dataset to observations (rows) meeting a specific criteria. Multiple criteria can be combined with the & (AND) and | (OR) symbols.

```
library(dplyr)
# select females
newdata <- filter(starwars, gender == "female")
# select females that are from Alderaan
newdata <- select(starwars, gender == "female" & homeworld == "Alderaan")
# select individuals that are from Alderaan, Coruscant, or Endor
newdata <- select(starwars, homeworld == "Alderaan" | homeworld == "Coruscant" |
                            homeworld == "Endor")
# this can be written more succinctly as
newdata <- select(starwars, homeworld %in% c("Alderaan", "Coruscant", "Endor"))
```

Creating/Recoding variables

- The mutate function allows you to create new variables or transform existing ones.

```
library(dplyr)
# convert height in centimeters to inches, and mass in kilograms to pounds
newdata <- mutate(starwars, height = height * 0.394,
                            mass = mass * 2.205)
```

- The ifelse function can be used for recoding data.

The format is ifelse(test, return if TRUE, return if FALSE).


```
library(dplyr)
# if height is greater than 180 then heightcat = "tall", otherwise heightcat = "short"
newdata <- mutate(starwars, heightcat = ifelse(height > 180, "tall",
                                               "short")
```

Or

```
# set heights greater than 200 or less than 75 to missing
newdata <- mutate(starwars, height = ifelse(height < 75 | height > 200,
                                            NA, height)


# convert any eye color that is not black, blue or brown, to other
newdata <- mutate(starwars, eye_color = ifelse(eye_color %in%
                                               c("black", "blue", "brown"), eye_color, "other")
```

Summarizing data

- The summarize function can be used to reduce multiple values down to a single value (such as a mean).
- It is often used in conjunction with the group_by function, to calculate statistics by group.
- In the code below, the na.rm=TRUE option is used to drop missing values before calculating the means.

```
library(dplyr)
# calculate mean height and mass
newdata <- summarize(starwars, mean_ht = mean(height, na.rm=TRUE),
                                mean_mass = mean(mass, na.rm=TRUE))
newdata
```

```
## # A tibble: 1 x 2
## mean_ht      mean_mass
##     <dbl>             <dbl>
## 1    174.              97.3
# calculate mean height and weight by gender
newdata <- group_by(starwars, gender)
newdata <- summarize(newdata, mean_ht = mean(height, na.rm=TRUE),
                                mean_wt = mean(mass, na.rm=TRUE))
newdata
```

```
## # A tibble: 5 x 3
##   gender        mean_ht mean_wt
##   <chr>           <dbl>   <dbl>
## 1 female           165.    54.0
## 2 hermaphrodite    175.  1358.
## 3 male             179.    81.0
## 4 none             200.   140.
## 5 <NA>             120.    46.3
```

Using pipes

- Packages like **dplyr** and **tidyr** allow you to write your code in a compact format using the pipe **%>%** operator.
- The **%>%** operator passes the result on the left to the first parameter of the function on the right.

```r
library(dplyr)

# calculate the mean height for women by species
newdata <- filter(starwars,
                  gender == "female")
newdata <- group_by(species)
newdata <- summarize(newdata,
                     mean_ht = mean(height, na.rm = TRUE))

# this can be written as
newdata <- starwars %>%
  filter(gender == "female") %>%
  group_by(species) %>%
  summarize(mean_ht = mean(height, na.rm = TRUE))
```

Reshaping data

Some graphs require the data to be in wide format, while some graphs require the data to be in long format. You can convert a wide dataset to a long dataset using **gather().**

```
library(tidyr)
long_data <- gather(wide_data,
                    key="variable",
                    value) ="value“,
                    sex:income
```

You can convert a long dataset to a wide dataset using **spread().**

```
library(tidyr)
wide_data <- spread(long_data, variable,
                    value)
```

Table 1.2: Wide data

id	name	sex	age	income
01	Bill	Male	22	55000
02	Bob	Male	25	75000
03	Mary	Female	18	90000

Table 1.3: Long data

id	name	variable	value
01	Bill	sex	Male
02	Bob	sex	Male
03	Mary	sex	Female
01	Bill	age	22
02	Bob	age	25
03	Mary	age	18
01	Bill	income	55000
02	Bob	income	75000
03	Mary	income	90000

Missing data

- Real data are likely to contain missing values. There are three basic approaches to dealing with missing data:
  1. feature selection,
  2. listwise deletion, and
  3. imputation.
- Here we are using **msleep** dataset from the **ggplot2** package. The msleep dataset describes the sleep habits of mammals and this dataset contains missing values on several variables.

1. Feature selection

In feature selection, you delete variables (columns) that contain too many missing values.

```
data(msleep, package="ggplot2")

# what is the proportion of missing data for each variable?
pctmiss <- colSums(is.na(msleep))/nrow(msleep)
round(pctmiss, 2)
```

```
##         name        genus         vore        order conservation
##         0.00         0.00         0.08         0.00         0.35
##  sleep_total    sleep_rem  sleep_cycle        awake      brainwt
##         0.00         0.27         0.61         0.00         0.33
##       bodywt
##         0.00
```

2. Listwise deletion

Listwise deletion involves deleting observations (rows) that contain missing values on *any* of the variables of interest.

 # Create a dataset containing genus, vore, and conservation. Delete any rows containing missing data.

newdata <- **select**(msleep, genus, vore, conservation)

newdata <- **na.omit**(newdata)

3. Imputation

- Imputation involves replacing missing values with “reasonable” guesses.
- There are several approaches, and some packages are available for imputation techniques. They are such packages like **VIM(Visualization and Imputation of Missing Values), mice, Amelia** and **missForest**.
- Here we will use the kNN function from the VIM package to replace missing values with imputed values.

- Basically, for each case with a missing value, the *k* most similar cases not having a missing value are selected.
  - If the missing value is **numeric**, the **mean of those *k* cases is used** as the **imputed value**.
  - If the missing value is **categorical**, the **most frequent value from the *k* cases is used.**

```
# Impute missing values using the 5 nearest neighbors
library(VIM)
newdata <- kNN(msleep, k=5)
```

Introduction to Base Graphics

- Producing quality supporting graphics for analysis is the hardest part.
- R provides excellent graphing capabilities, both in the base installation and with add-on packages such as **lattice** and **ggplot2**.
- Presenting some simple graphs using base graphics and then show their counterparts in **ggplot2**.
- Graphics are used in statistics primarily for two reasons: exploratory data analysis (EDA) and presenting results. Both are incredibly important.

Data Visualization

- Data Visualization is one of the most important processes in data science.
- Relationship between the variables can sometimes more easily be understood visually than on the data.
- We are having different data visualization tools offered by different software companies, but R has built-in capabilities for data visualization.
- The visual representations in R is also called as **Graphics**.
- The main aim of this data visualization is how to draw different kinds of graphs, plots and charts for the datasets.

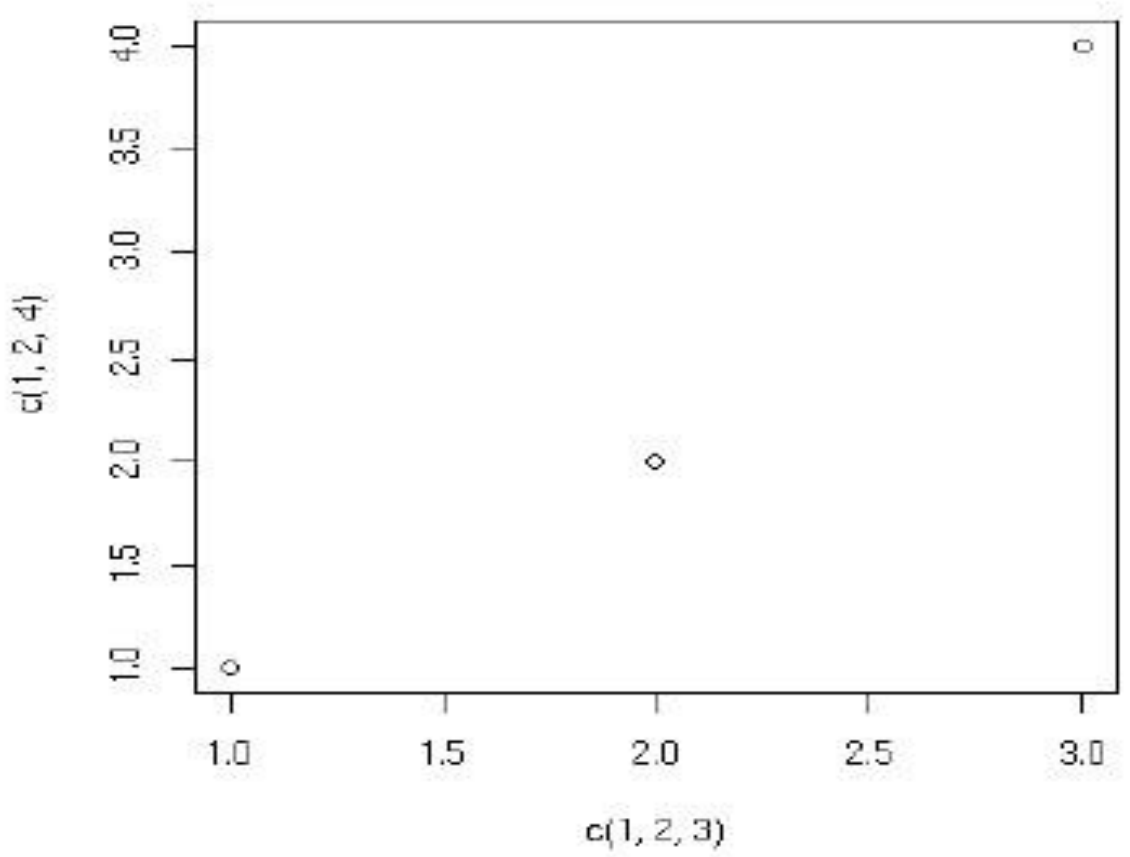
Creating Graphs

The foundational function for creating graphs: **plot().**

The Workhorse of R Base Graphics: The plot() Function

- The most used plotting function in R programming is the plot() function.
- plot() is a generic function, means the function that is actually called depends on the class of the object on which it is called.
- Let's see what happens when we call plot() with an X vector and a Y vector, which are interpreted as a set of pairs in the (x,y) plane.

  Example: plot(c(1,2,3), c(1,2,4))which means plotting the points (1,1), (2,2), and (3,4)

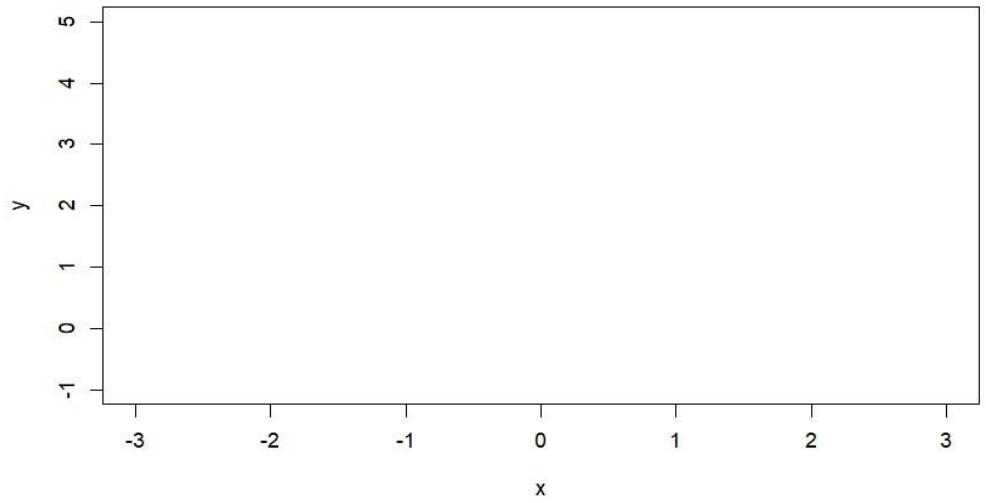


- This is a simple point plot. The points in the graph are denoted by empty circles.

Labeling axes:

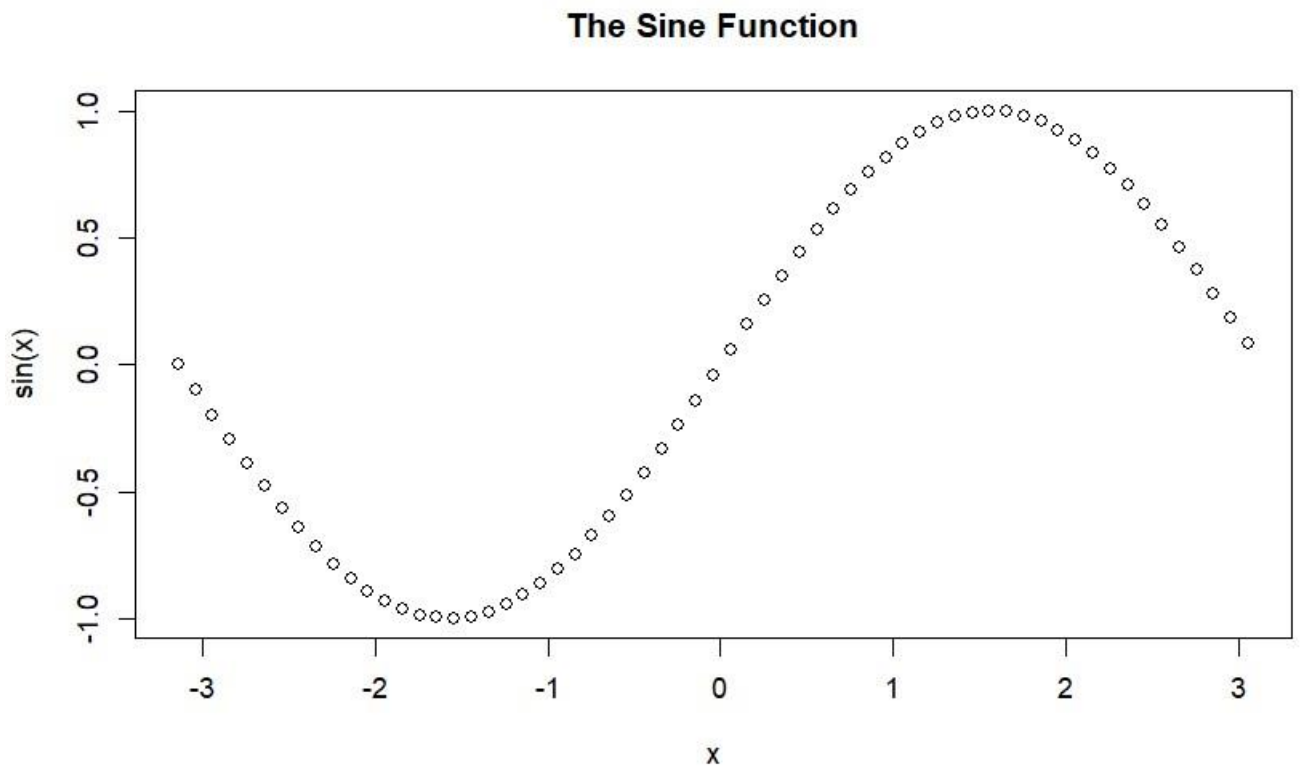
Example: plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y“)

- This draws axes labeled ***x*** and ***y***.
- The horizontal **(*x*)** axis ranges from −3 to 3.
- The vertical **(*y*)** axis ranges from −1 to 5.
- The argument **type="n"** means that there is nothing in the graph.



Adding titles:

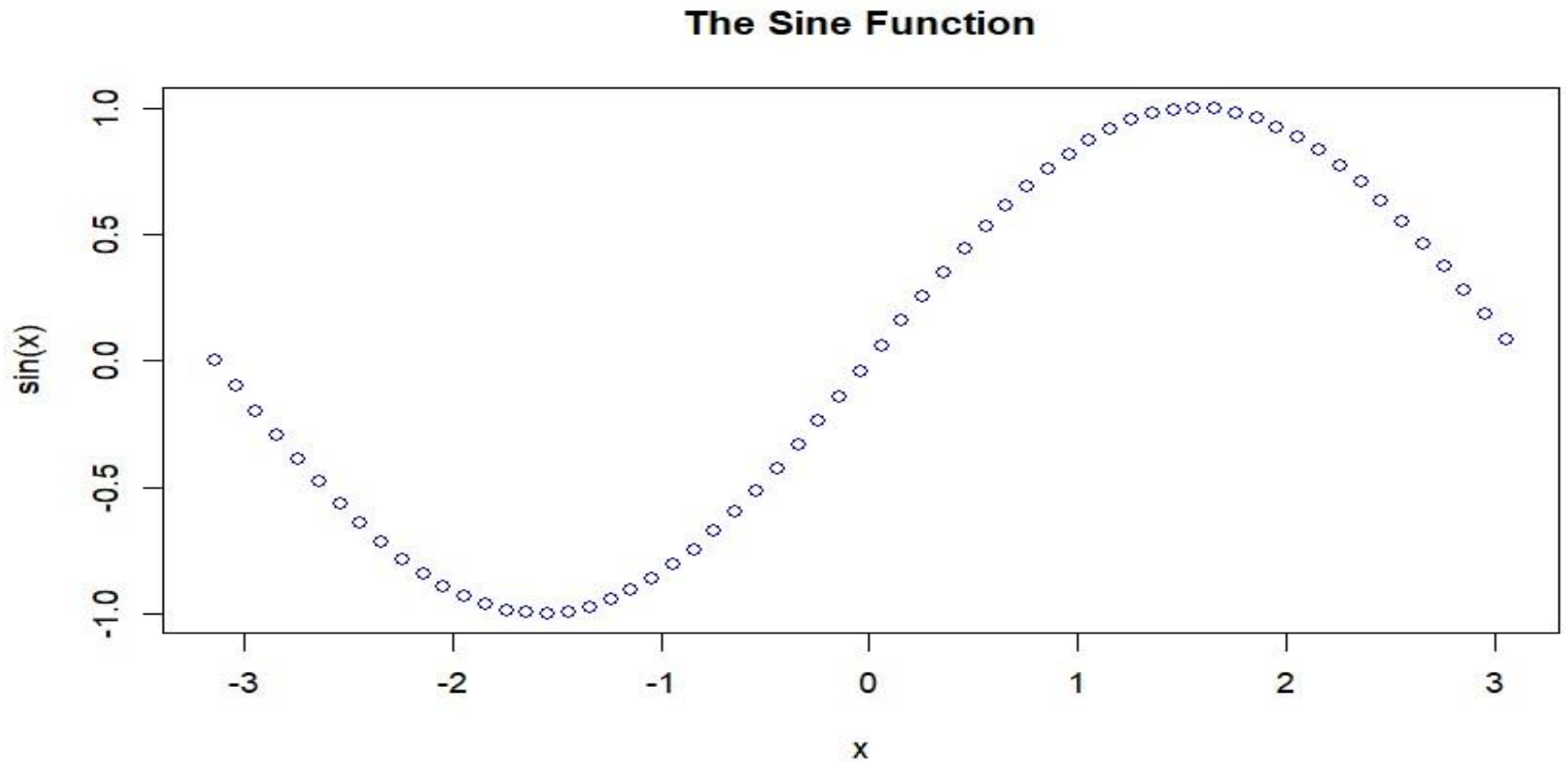
Example: plot(x, sin(x), main="The Sine Function", ylab="sin(x)")



Changing Color and Plot Type

- We can see above that the plot is of circular points and black in color. This is the default color.
- We can change the plot type with the argument type. It accepts the following strings and has the given effect.
  - ✓"p" – points
  - ✓"l" - lines
  - ✓"b" - both points and lines
  - ✓"c" - empty points joined by lines
  - ✓"o" – over plotted points and
  - ✓lines "s" and "S" - stair steps
  - ✓"h" - histogram-like vertical lines
  - ✓"n" - does not produce any points or lines

Example: plot(x, sin(x), main="The Sine Function",
ylab="sin(x)", type="l", col="blue")



Adding Lines: The abline() Function

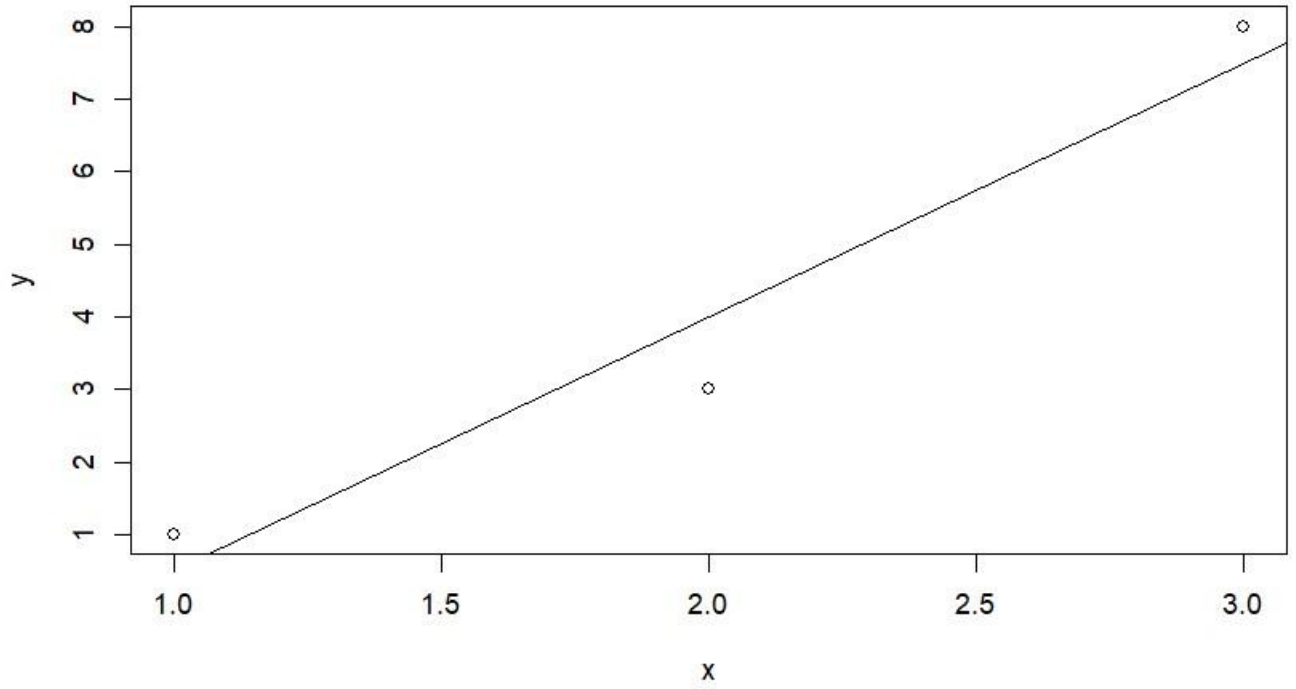
- ✓Straight lines can be added to an existing plot using the simple abline() function. The **abline()** function can be used to add vertical, horizontal or **regression** lines to plot.

Example:

```
x <- c(1,2,3)
y <- c(1,3,8)
plot(x,y)
lmout <- lm(y ~ x)
abline(lmout)
```

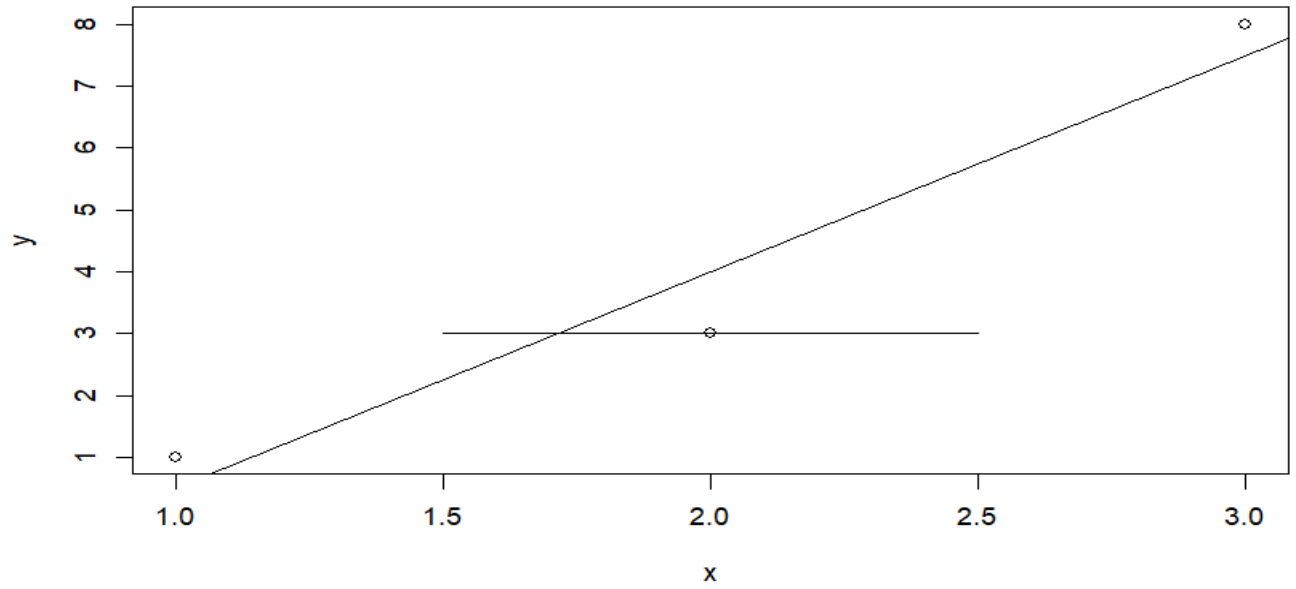
- ✓After the call to plot(), the graph will simply show the three points, along with the *x*- and *y*- axes.
- ✓The call to abline() then adds a line to the current graph.
- ✓The call to the linear-regression function **lm()** is a class instance containing the slope and intercept of the fitted line

- Class instance is assigned to **lmout.** we call abline() function this simply draws a straight line, with the function's arguments treated as the intercept and slope of the line.



- You can add more lines by using the **lines()** function. The two basic arguments to lines() are a vector of ***x*-values** and a vector of ***y*-values**.
- These are interpreted as (*x*,*y*) pairs representing points to be added to the current graph, with lines connecting the points.

  Example: lines(c(1.5,2.5),c(3,3))

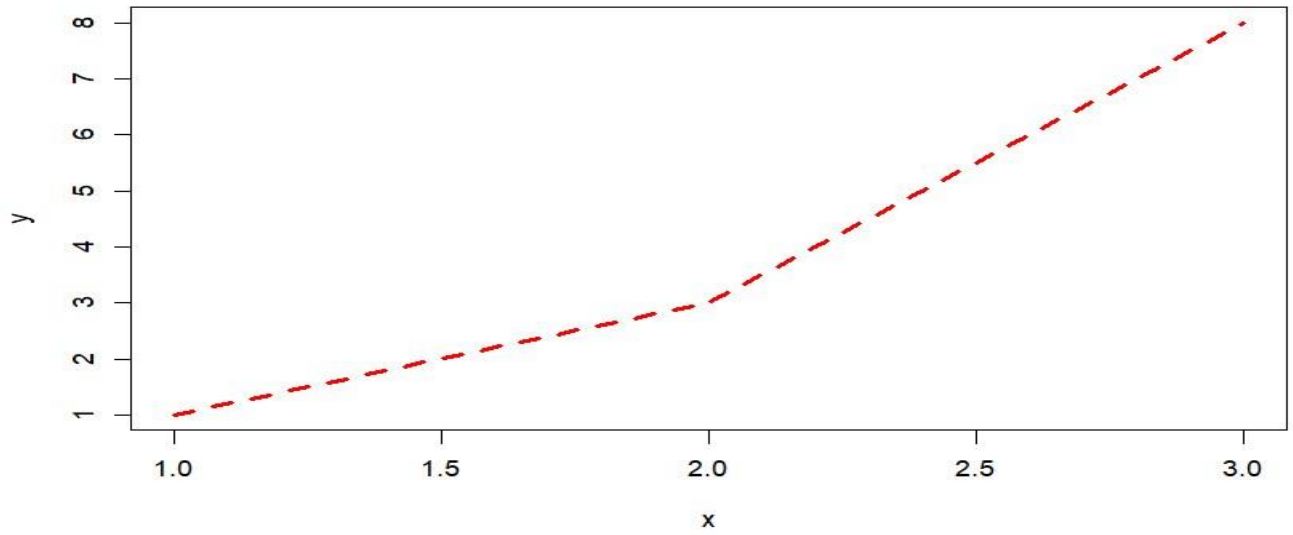


- If you want the lines to “connect the dots,” then include type="l" in your call to lines() or to plot().

Example: plot(x, y, type="l")

- Other parameter for plot() are lty- type of line (solid, dashed, black line, dotted line) and lwd- change line width

Example: plot(x, y, type="l“ ,col="red",lty=2, lwd=3)



Starting a New Graph While Keeping the Old Ones

- Each time you call plot(), directly or indirectly, the current graph window will be replaced by the new one. If you don't want that to happen, then use the command for your operating system:
- On **Linux** systems, call **X11().**
- On a **Mac**, call **macintosh().**
- On **Windows**, call **windows().**
- For instance, suppose you wish to plot two histograms of vectors X and Y and view them side by side. On a Linux system, you would type the following:

```
hist(x)
x11()
hist(y)
```

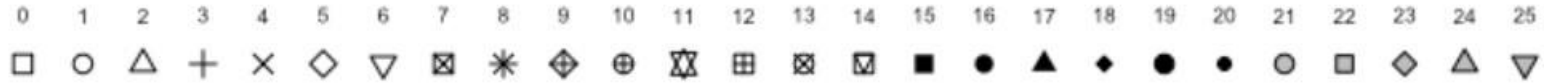
Adding Points: The points() Function

- points() is used to add a group of points of specified shapes, size and color to an existing plot.
- It is a generic function to draw a sequence of points at the specified coordinates. The specified character(s) are plotted, centered at the coordinates.

Syntax: points(x, y, cex, pch, col)

where **x, y:** Vector of coordinates
cex: size of points
pch: shape of points
col: color of points

Here pch with different shapes are values are also available.

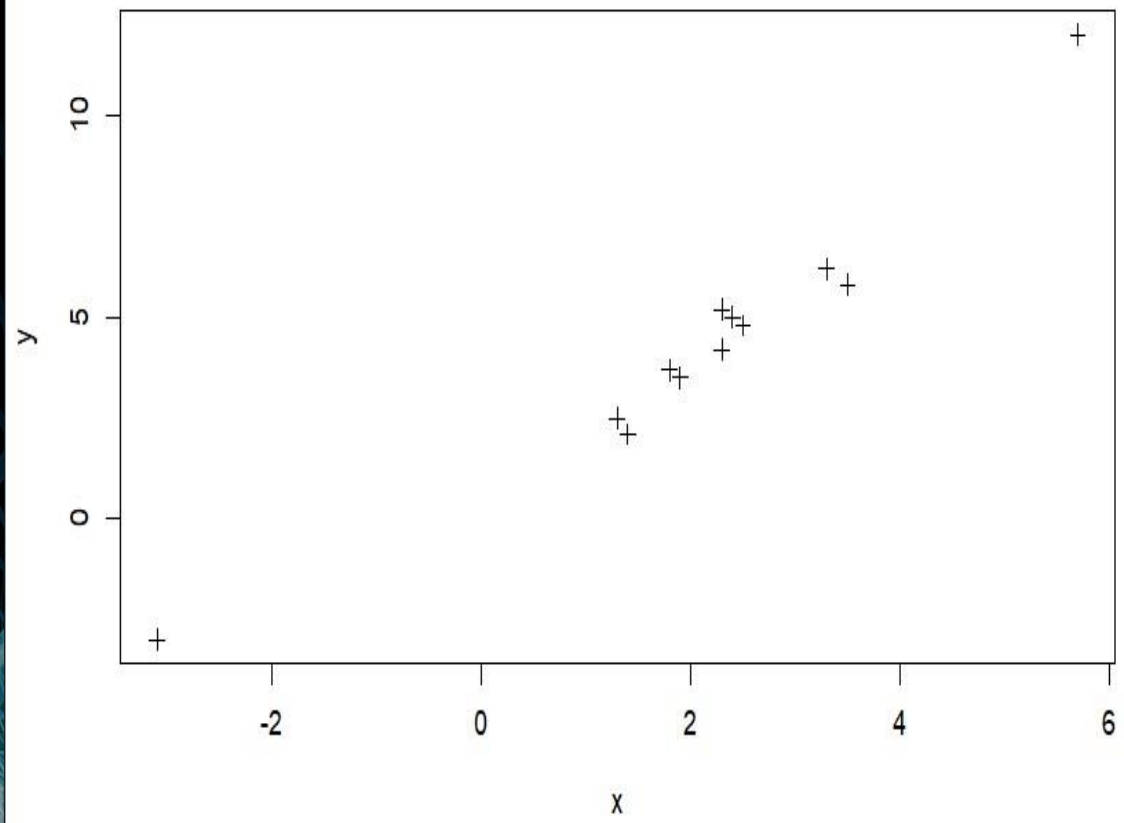


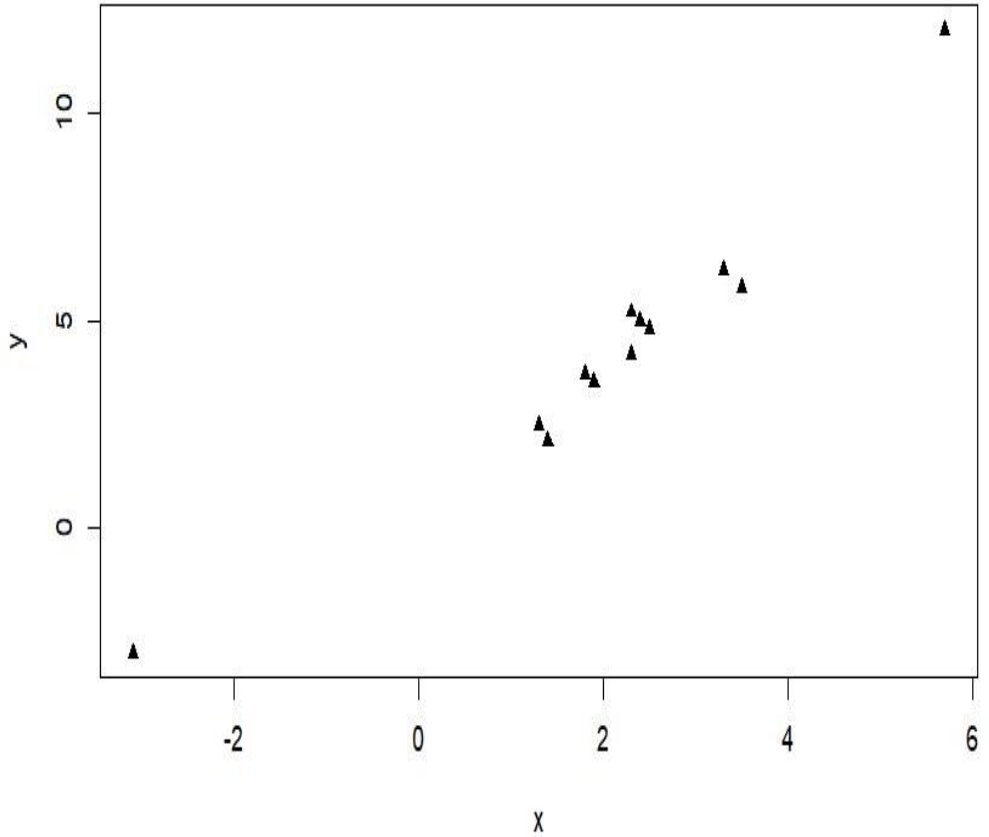
- pch = 19: solid circle,
- pch = 20: bullet (smaller solid circle, 2/3 the size of 19),
- pch = 21: filled circle,
- pch = 22: filled square,
- pch = 23: filled diamond,
- pch = 24: filled triangle point-up,
- pch = 25: filled triangle point down.

Most commonly we use this shapes in the plot().

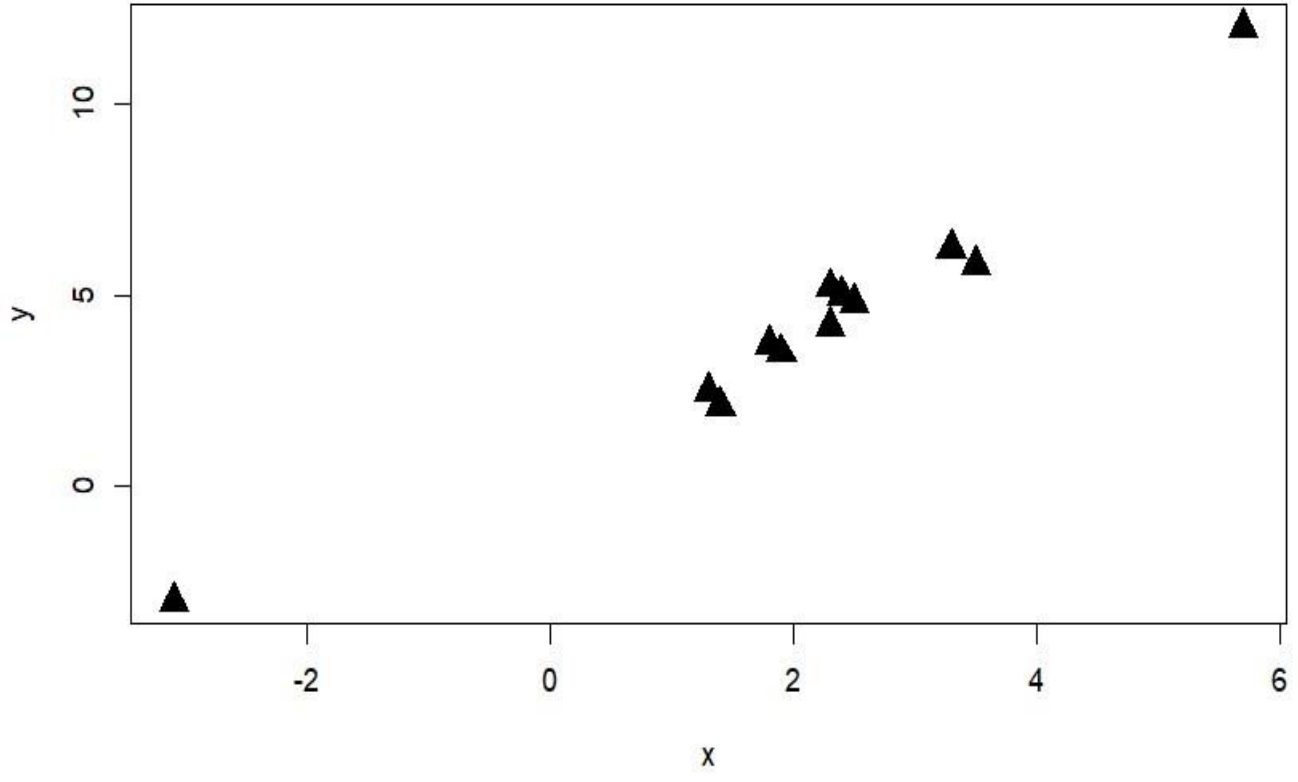
Example: # Creating coordinate vectors

```
x <- c(1.3, 3.5, 1.4, -3.1, 5.7, 2.4, 3.3, 2.5, 2.3, 1.9, 1.8, 2.3)
y <- c(2.5, 5.8, 2.1, -3, 12, 5, 6.2, 4.8, 4.2, 3.5, 3.7, 5.2)s
# Plotting the graph
plot(x, y, cex = 1, pch = 3, xlab ="x", ylab ="y", col ="black")
plot(x, y, cex = 1, pch = 17, xlab ="x", ylab ="y", col ="black")
```


y
10
5
0
-2
0
2
4
6
x


y
10
5
0
-2
0
2
4
6
x

- cex- character (or symbol) expansion: a numerical vector.



- **par()** is used to set or query graphical parameters. We can use the par() function in R to create multiple plots at once.
- We can divide the frame into the desired grid, add a margin to the plot or change the background color of the frame by using the par() function.
- By using par() we can develop our plot

1. Create Multiple plots in the same window
2. Increase or Decrease margin Around Plot
3. Change axis and tick label size

Syntax: par(mfrow, mar, cex.lab, cex.axis, bg)

where, **mfrow:** determines a vector with row and column values for the grid.

mar: determines the margin vector in which margins are in the order: bottom, left, top, right

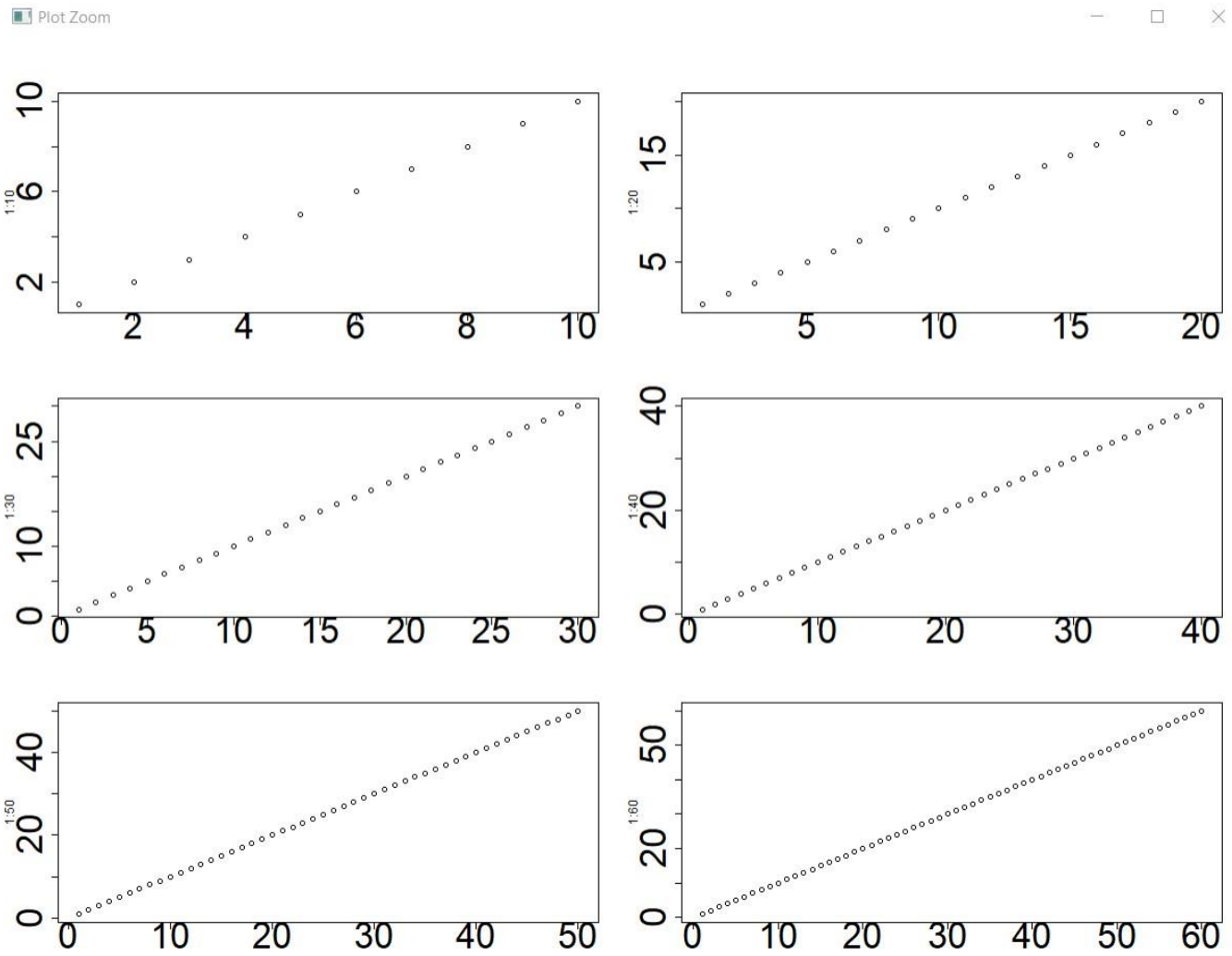
cex.lab: determines the size of the axis label.

cex.axis: determines the size of the tick labels.

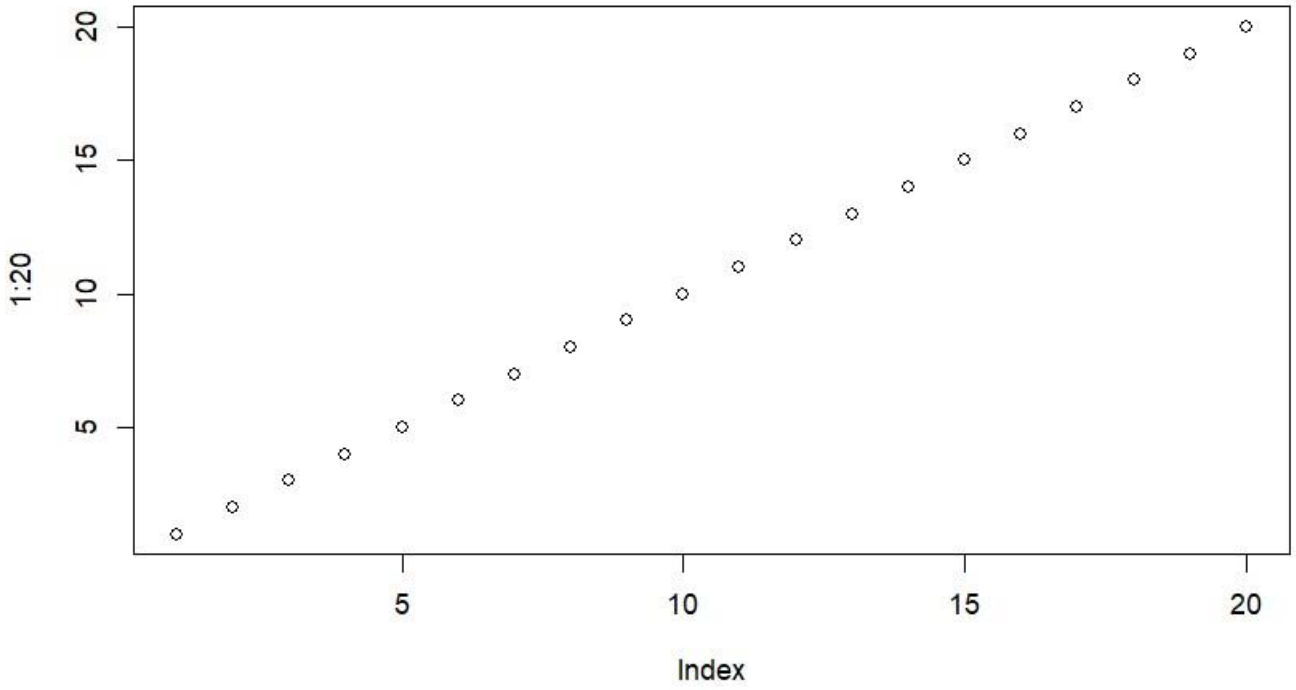
bg: determines the color of the background of the window.

Program:

```
# divide window into a 3X2 grid
par( mfrow= c(3,2) )
# add plots to window
plot( 1:10 )
plot( 1:20 )
plot( 1:30 )
plot( 1:40 )
plot( 1:50 )
plot( 1:60 )
```



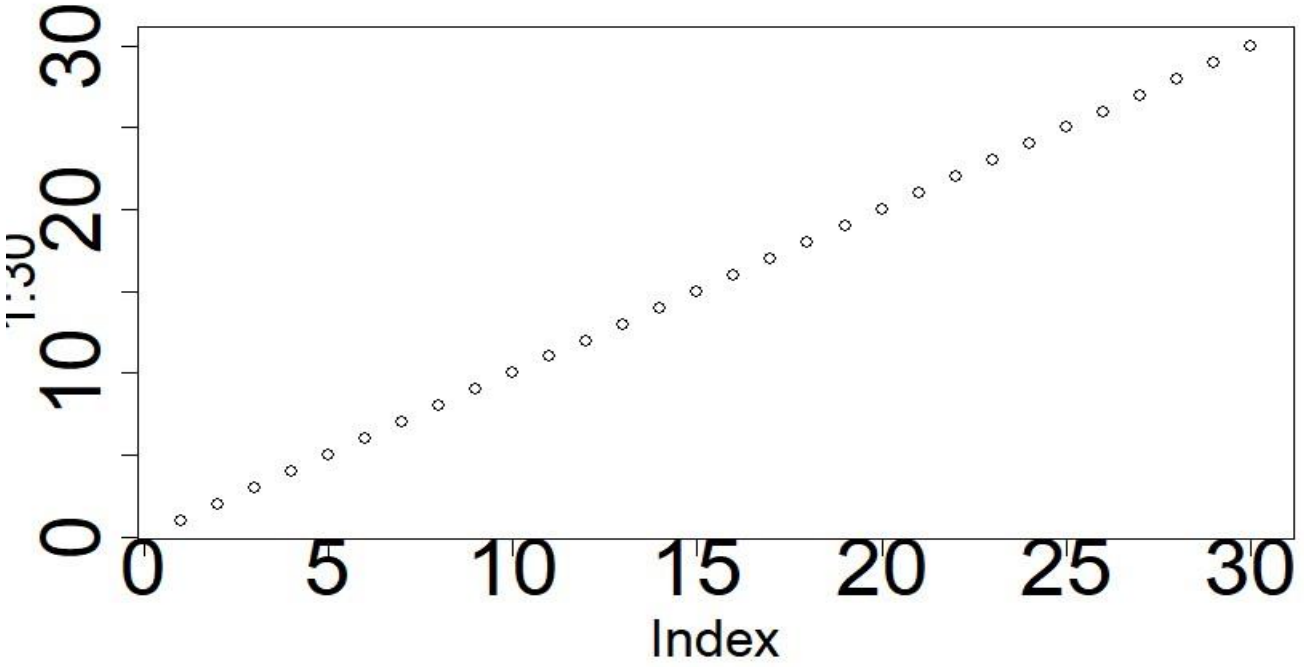
```
# Add margin using par() function
par( mar= c(1,10,12,1))
# create plot
plot( 1:20 )
```



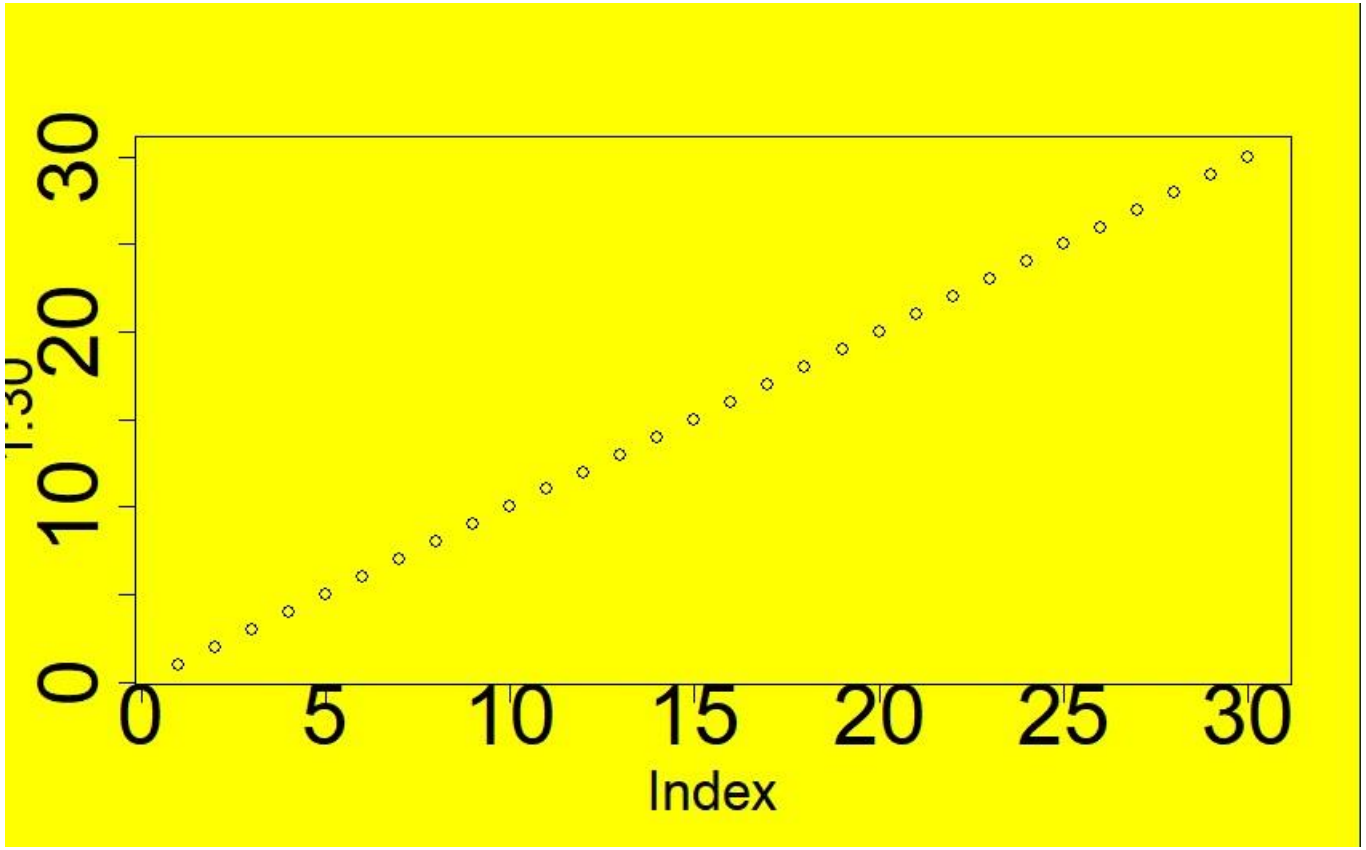
Changing Character Sizes: The cex Option

- The cex (for *character expand*) function allows you to expand or shrink characters within a graph, but with a larger font.

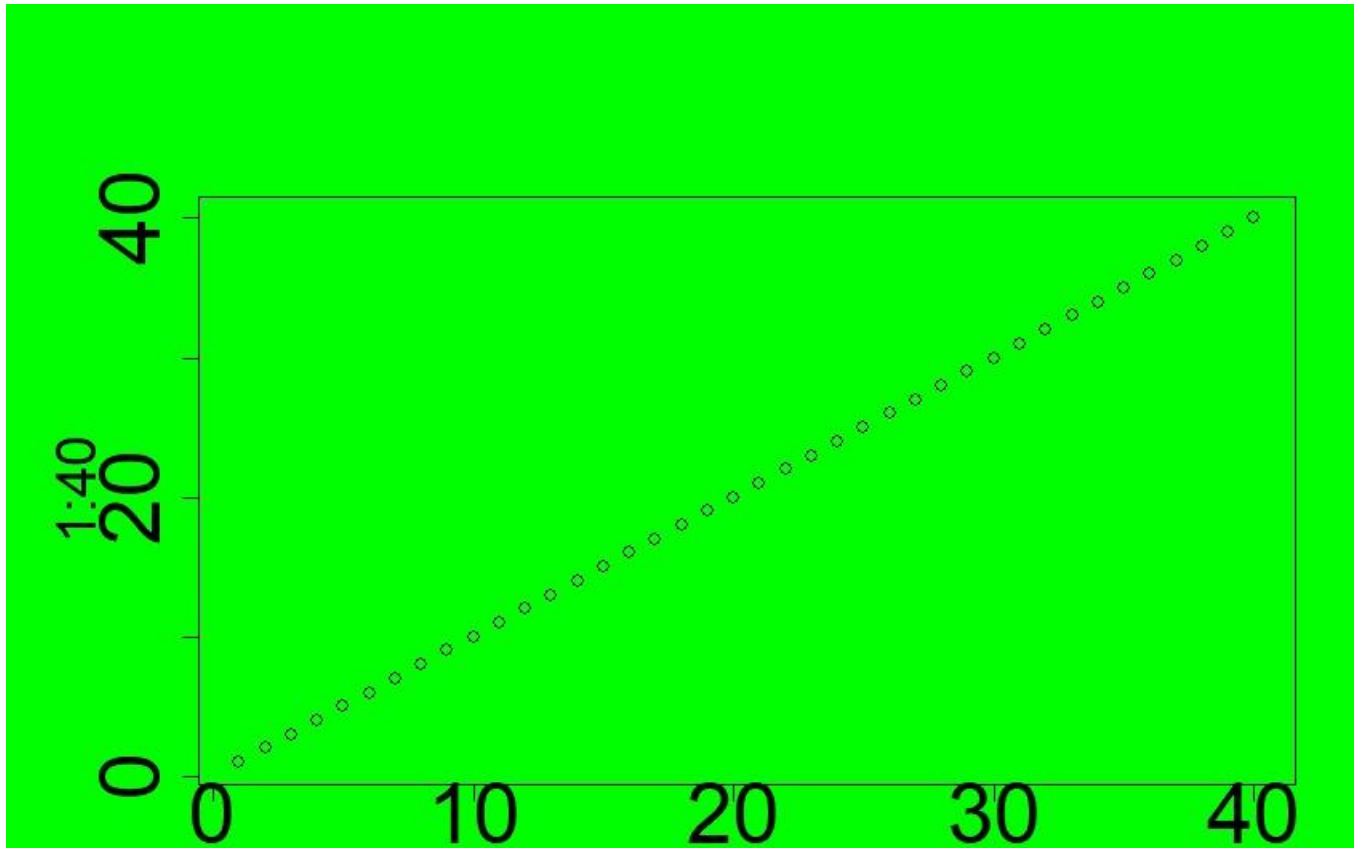
```
# set axis and tick label size with par() function
par( cex.axis= 3, cex.lab= 2)
# draw plot
plot(1:30)
```



```
# set green color for background of plot
par( bg = "yellow" )
# draw a plot
plot( 1:30 )
```



```
par( mar= c(2,6,6,1), cex.axis= 3, cex.lab= 2, bg = "blue")
plot( 1:40 )
```



legend():

- It allows you to add a legend to a plot in base R.
- Legends are useful to add more information to the plots and enhance the user readability.
- It involves the creation of titles, indexes, placement of plot boxes in order to create a better understanding of the graphs plotted.
- The summarized syntax of the function with the most common arguments is described in the following block:

Syntax: legend(x, y, # Coordinates (x also accepts keywords)

legend, # Vector with the name of each group

fill, # Creates boxes in the legend with the specified colors

col = par("col"), # Color of lines or symbols

border = "black", # Fill box border color

lty, lwd, # Line type and width

pch, # Add pch symbols to legend lines or boxes

bty = "o", # Box type (bty = "n" removes the box)

bg = par("bg") # Background color of the legend

box.lwd = par("lwd"), # Legend box line width

box.lty = par("lty"), # Legend box line type

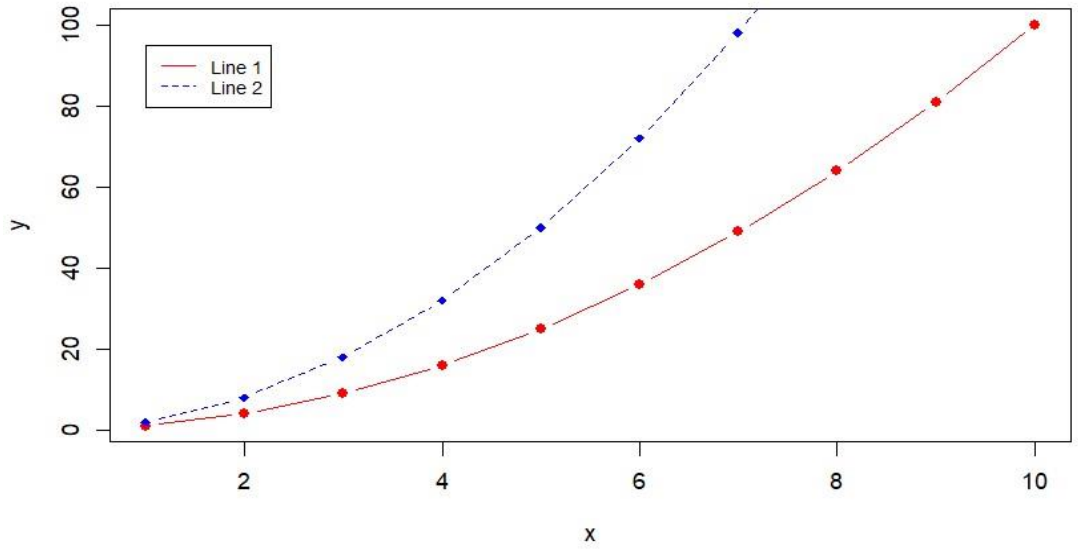
box.col = par("fg"), # Legend box line color

cex = 1, # Legend size

horiz = FALSE # Horizontal (TRUE) or vertical (FALSE) legend

title = NULL # Legend title)

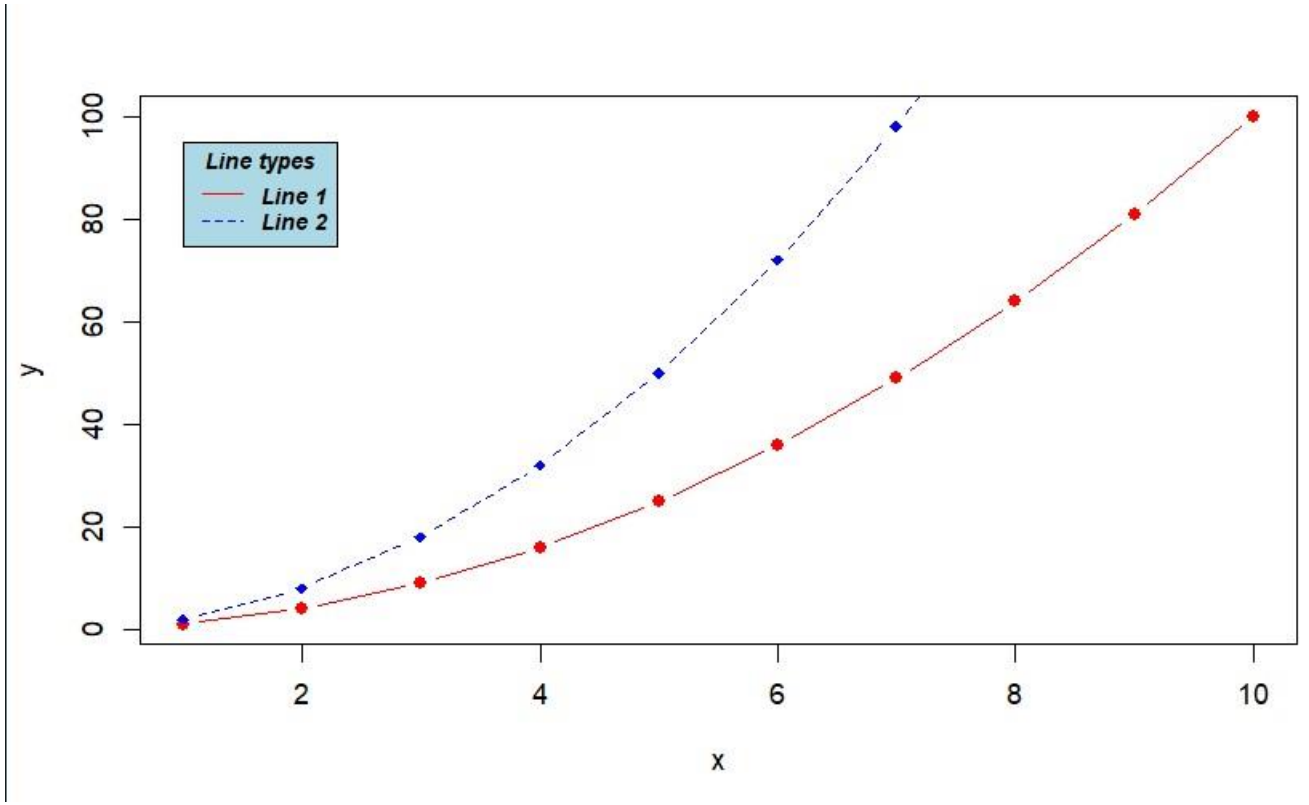
```
# Generate some data
x<-1:10; y1=x*x; y2=2*y1
plot(x, y1, type="b", pch=19, col="red", xlab="x", ylab="y")
# Add a line
lines(x, y2, pch=18, col="blue", type="b", lty=2)
# Add a legend
legend(1, 95, legend=c("Line 1", "Line 2"), col=c("red", "blue"), lty=1:2, cex=0.8)
```



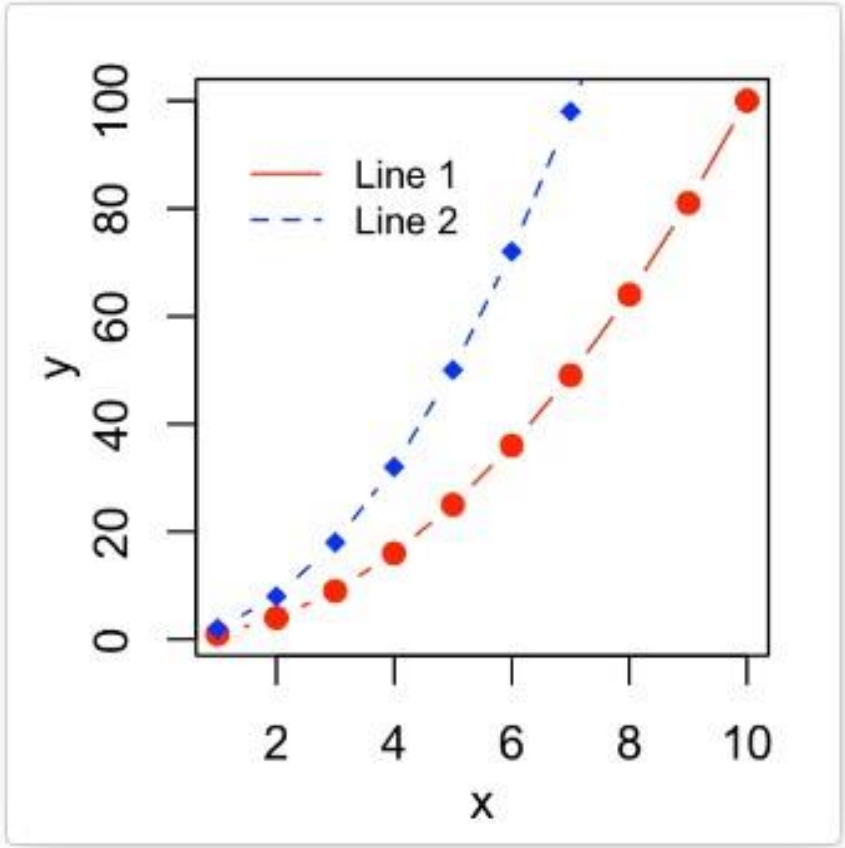
Title, text font and background color of the legend box

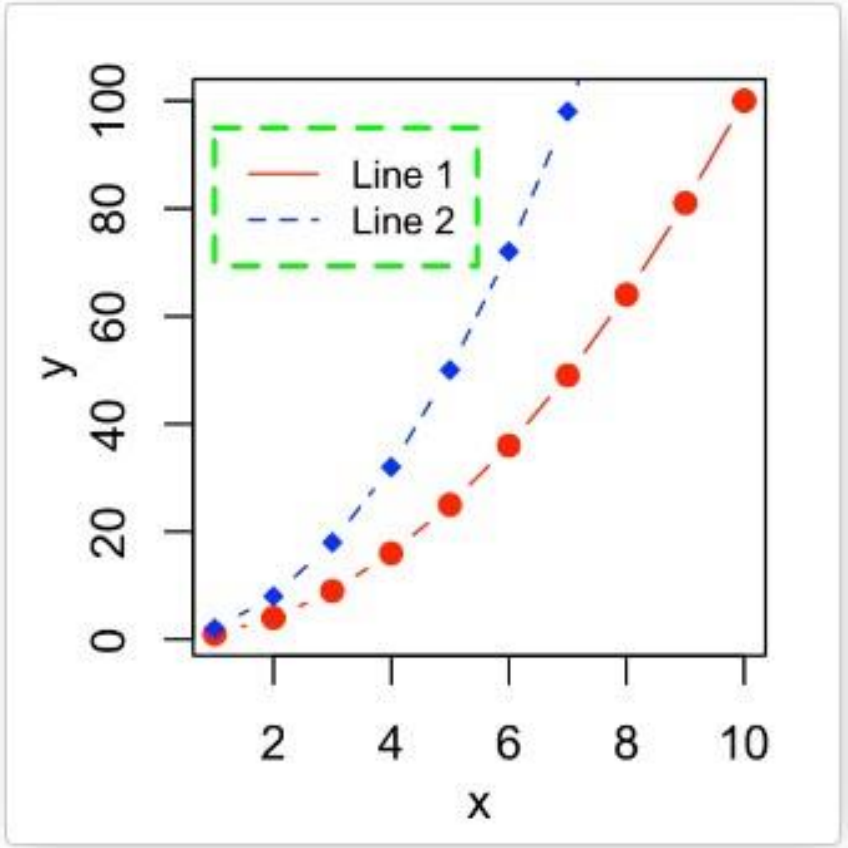
- The arguments below can be used :
  - **title**: The title of the legend
  - **text.font**: an integer specifying the font style of the legend text; possible values are :
    - ✓**1**: normal
    - ✓**2**: bold
    - ✓**3**: italic
    - ✓**4**: bold and italic
  - **bg**: background color of the legend box
- we can create a custom plot function to avoid repeating the **R** code


```
makePlot<-function(){
  x<-1:10; y1=x*x; y2=2*y1
  plot(x, y1, type="b", pch=19, col="red", xlab="x", ylab="y")
  lines(x, y2, pch=18, col="blue", type="b", lty=2)
}
makePlot()
# Add a legend to the plot
legend(1, 95, legend=c("Line 1", "Line 2"),
       col=c("red", "blue"), lty=1:2, cex=0.8,
       title="Line types", text.font=4, bg='lightblue')
```


Line types
Line 1
Line 2
y
x
0
20
40
60
80
100
2
4
6
8
10

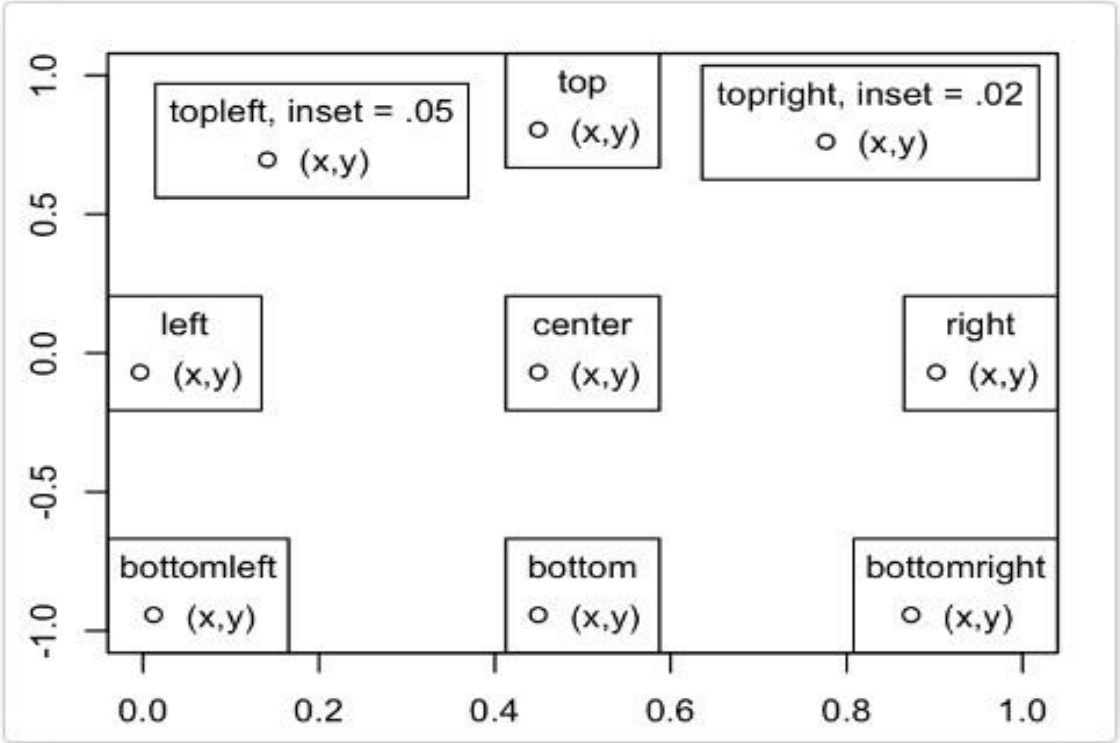

```
# Remove legend border using box.lty = 0
makePlot()
legend(1, 95, legend=c("Line 1", "Line 2"),
       col=c("red", "blue"), lty=1:2, cex=0.8,
       box.lty=0)
# Change the border
makePlot()
legend(1, 95, legend=c("Line 1", "Line 2"),
       col=c("red", "blue"), lty=1:2, cex=0.8,
       box.lty=2, box.lwd=2, box.col="green")
```


Line 1
Line 2
y
0
20
40
60
80
100
2
4
6
8
10
x

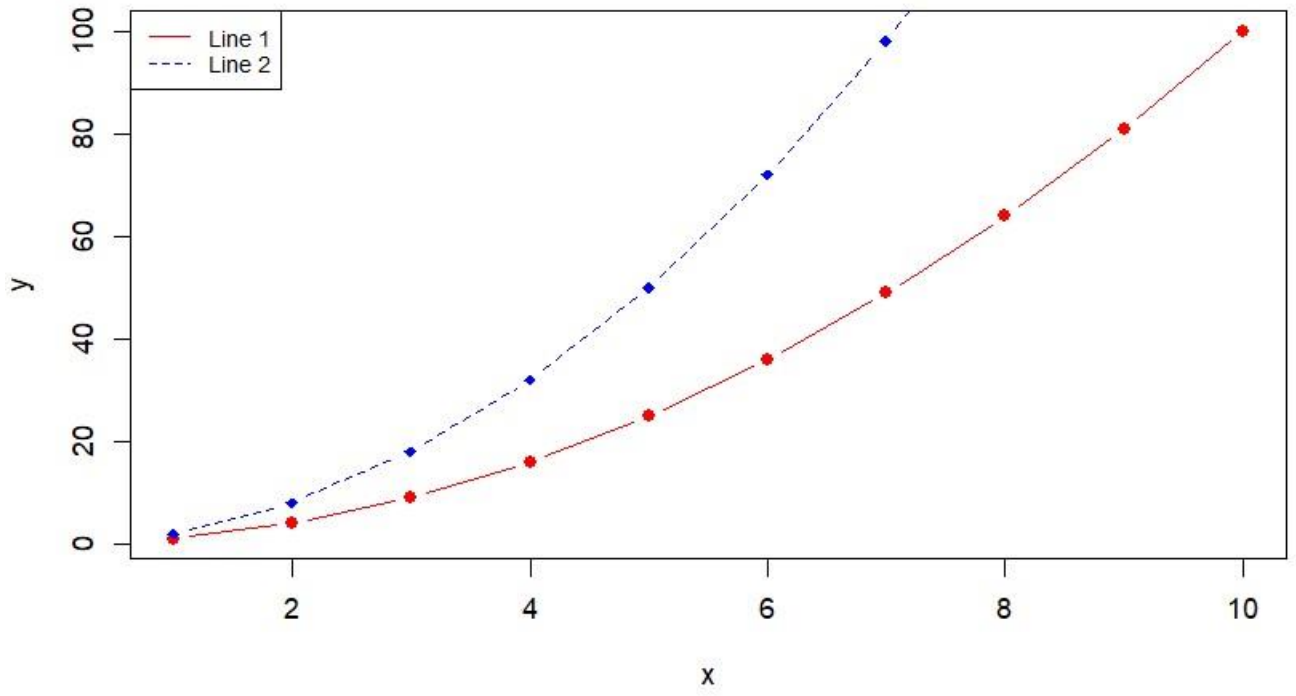

Line 1
Line 2
y
0
20
40
60
80
100
2
4
6
8
10
x

Specify legend position by keywords

The position of the legend can be specified also using the following keywords : "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center".



```
makePlot()
legend("topleft", legend=c("Line 1", "Line 2"),
       col=c("red", "blue"), lty=1:2, cex=0.8)
```



locator():

- For fining a good position location we keep on trying with different x- and y-coordinates.
- But R is providing use a best function without wasting time on checking coordinates.
- The function that helps to locate x- and y-coordinates is the locator() function. We can simply call the function and then click the mouse at the desired spot in the graph.
- Once you click the function tell you the exact x and y-coordinates of your clicked point.
- If you want to click more than once then you need to specify the number of times in the function.

Syntax: locator()

Example: hist(c(12,5,13,25,16))

locator(1) – which indicates only one click

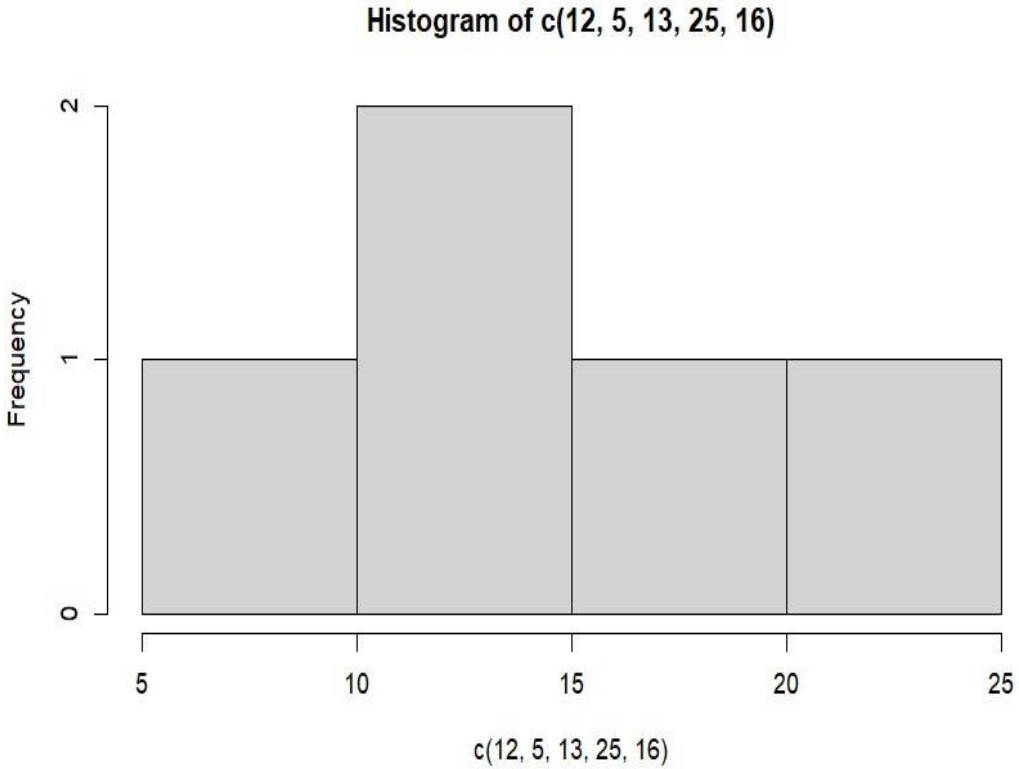
locator(3) – which indicates three clicks with three different coordinates

```
> locator(1)
$x
[1] 15.49695

$y
[1] 0.6012152

> hist(c(12,5,13,25,16))
> locator(3)
$x
[1]  6.964963 14.988025 22.023173

$y
[1] 0.6706240 0.9482589 0.9987380
```

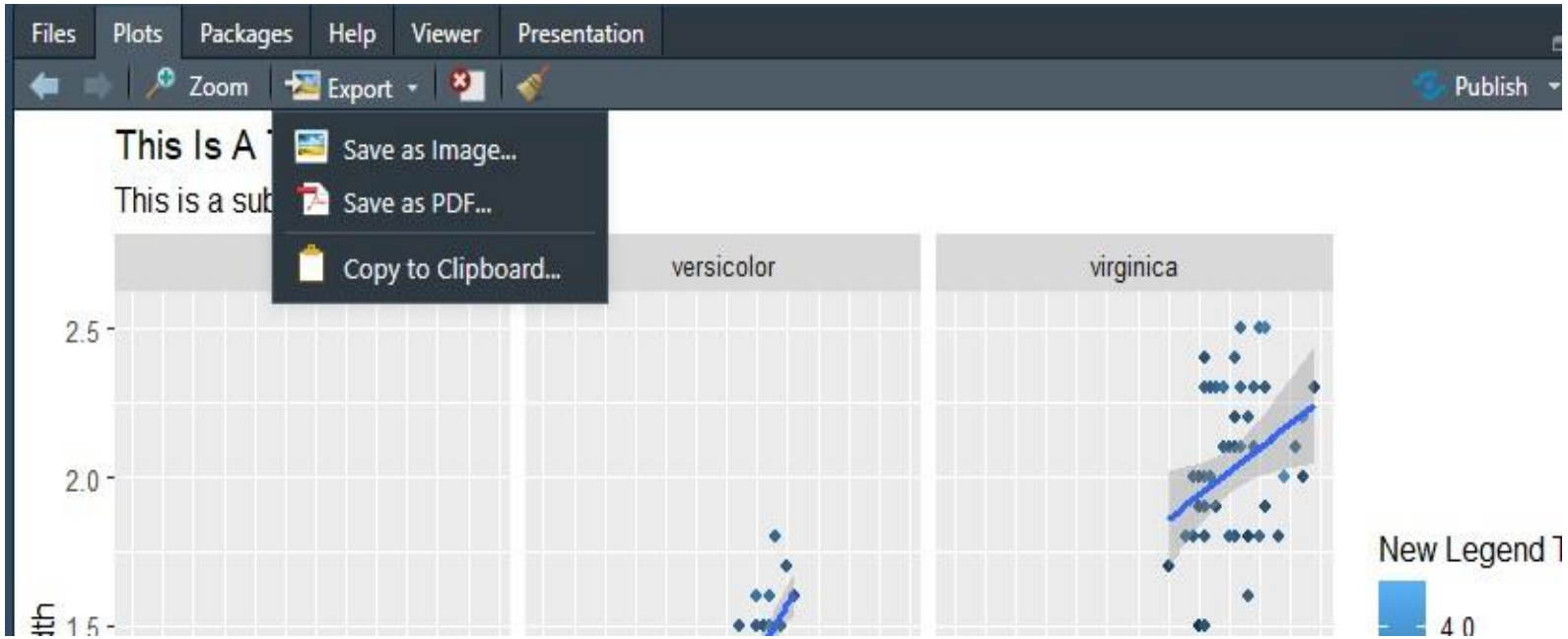


Saving Graphs to Files

GITAM
(DEEMED TO BE UNIVERSITY)
(Estd. u/s 3 of the UGC Act, 1956)
VISAKHAPATNAM • HYDERABAD • BENGALURU

For saving graphs in R we are having two ways.

1. **Via menus:** To save a graph using the RStudio menus, go to the Plots tab and choose Export.



2. **Via code:** ggplot2 graph can be saved as an object. Then you can use the **ggsave()** function to save the graph to disk.

```
# save a graph
library(ggplot2)
p <- ggplot(mtcars,
            aes(x = wt , y = mpg)) +
  geom_point()
ggsave(p, filename = "mygraph.png")
```

- The graph will be saved in the format defined by the file extension (*png* in the example above). Common formats are *pdf*, *jpeg*, *tiff*, *png*, *svg*, and *wmf* (windows only).

Introduction to R Graphics

- Data Visualization is one of the most important processes in data science.
- Visualization is the art of examining distributions and relationships between variables using visual representations(graphics).
- Relationships between variables can sometimes more easily be understood visually than relying only on predictive modelling, statistics, and on the data manipulation.
- R has built-in capabilities for data visualization. These capabilities can of course be extended by recourse to external packages. Graphics made for a particular dataset can be reused and adapted or another with relatively little effort.
- R is providing 4 different packages which is helpful in data visualization. They are as follows:
    1. Data Visualization with **Base** Graphics
    2. Data Visualization with **Grid** Graphics
    3. Data Visualization with **Lattice** Graphics
    4. Data Visualization with **ggplot2** Graphics

Having powerful and flexible systems for graphics is one of R's biggest strengths.

Base Graphics: Contained in the graphics package distributed in base R.

Grid graphics: grid package is distributed in base R.

- Contains low-level graphics functions.
- Useful as a platform for developing and implementing higher-level graphics functions and systems.

Lattice Graphics: lattice package also distributed in base R.

- Mimics and extends trellis graphics from S and S-PLUS.
- Characteristic feature is plots with multiple panels.
- Built on grid.

ggplot2 Graphics: ggplot2 package available on CRAN.

- Based on Leland Wilkinson's ideas articulated in his book, The Grammar of Graphics.
- Sophisticated and powerful system. Not too hard to learn.
- Built on grid.

Base Graphics

- High level functions—Produce a new complete plot on the current graphics device.
  - **plot():** generic function capable of a wide variety of plot types.
  - **boxplot():** single and side-by-side boxplots.
  - **hist():** histograms.
  - **qqplot()**, **qqnorm()**, **qqline():** quantile-quantile plots.
  - **dotchart()**, **stripchart():** dot plots.
  - **image()**, **contour()**, **persp():** 3d plots.
  - **pairs():** scatter plot matrices.
- Low level functions—Add features to an existing plot.
  - **lines(), points(), symbols(), segments(), arrows():** add various features. Most have syntax similar to plot().
  - **abline(), curve():** add lines or curves from output of a model or at reference locations.
  - **title(), legend():** add a title or legend.
  - **axis():** adds an axis with fine control of its appearance.
  - **text(), mtext():** add text within the plotting region or in the margins

High-level functions in lattice

Function	Default Display
`histogram()`	Histogram
`densityplot()`	Kernel Density Plot
`qqmath()`	Theoretical Quantile Plot
`qq()`	Two-sample Quantile Plot
`stripplot()`	Stripchart (Comparative 1-D Scatter Plots)
`bwplot()`	Comparative Box-and-Whisker Plots
`dotplot()`	Cleveland Dot Plot
`barchart()`	Bar Plot
`xyplot()`	Scatter Plot
`splom()`	Scatter-Plot Matrix
`contourplot()`	Contour Plot of Surfaces
`levelplot()`	False Color Level Plot of Surfaces
`wireframe()`	Three-dimensional Perspective Plot of Surfaces
`cloud()`	Three-dimensional Scatter Plot
`parallel()`	Parallel Coordinates Plot