

R Programming

UNIT – V

Graphics

Creating Graphs:

To begin, we will look at the foundational function for creating graphs: `plot()`.

The `plot()` function:

The `plot()` function is the basic and most commonly used function in R for creating graphs.

Syntax:

`plot(x, y, type, main, xlab, ylab, col, pch)`

In the above syntax,

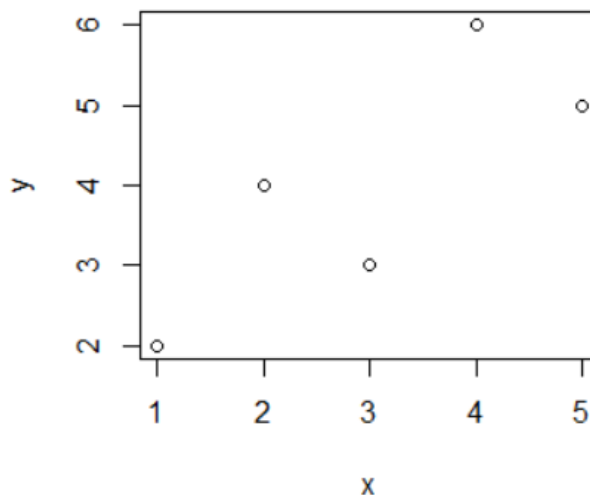
- `x, y` : Coordinates,
- `type` : Graph type ("p" for points, "l" for line, "b" for both, "n" for nothing)
- `main` : Title of the graph
- `xlab` : Label for X-axis
- `ylab` : Label for Y-axis
- `col` : Color of the graph
- `pch` : Shape of the points

- **Creating a Basic Plot:**

Creating a basic plot using only x and y coordinates.

Example:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y)
# Output:
```

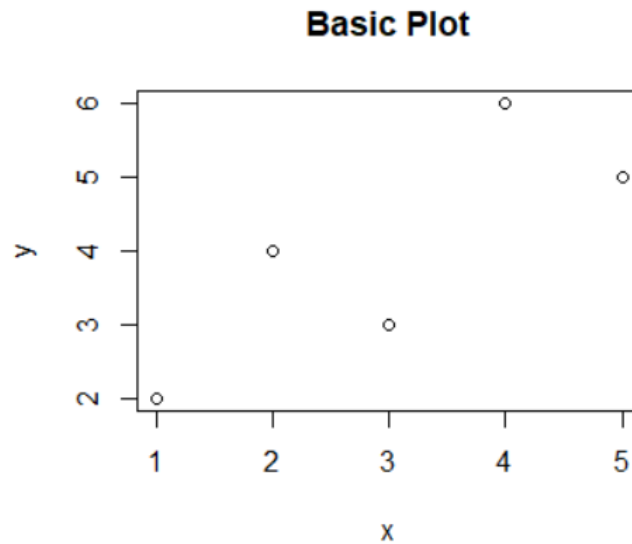


- **Adding a Title:**

Adding a Title for the graph using “*main*” argument.

Example:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Basic Plot")
# Output:
```

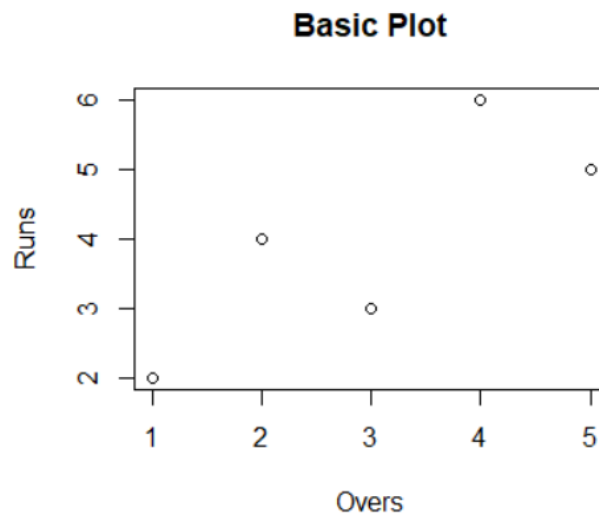


- **Adding Labels for X and Y axis:**

Adding a Labels for X and Y axis using “*xlab* and *ylab*” arguments.

Example:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Basic Plot", xlab="Overs", ylab="Runs")
# Output:
```

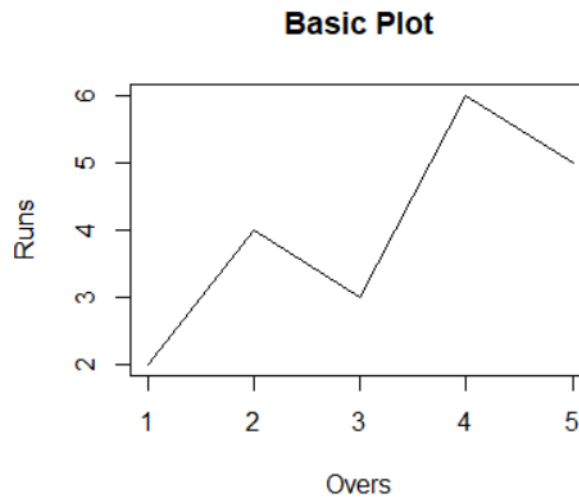


- **Graph Type:**

You can create different types of the Graphs using “*type*” argument. The *type* argument has different possible values. Those are "p" for points, "l" for line, "b" for both, "n" for nothing. By default, the graph type is points(p).

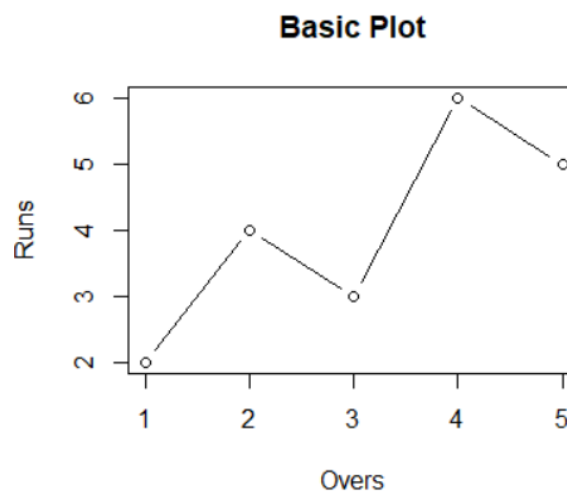
Example: *type* = “l”

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Basic Plot", xlab="Overs", ylab="Runs", type = "l")
# Output:
```



Example: *type* = “b”

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Basic Plot", xlab="Overs", ylab="Runs", type = "b")
# Output:
```



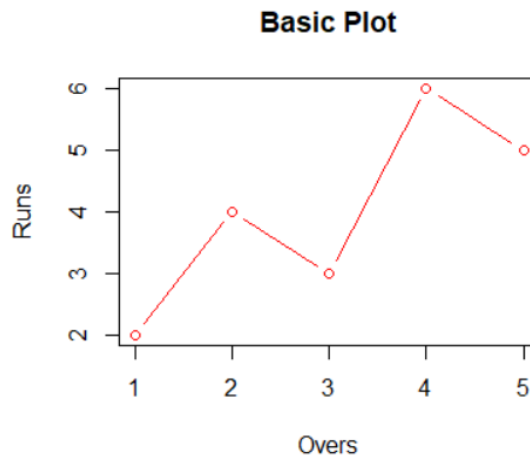
- **Color of the graph:**

You can specify the color of the graph using “*col*” argument. By default, the color is “black.”

Example:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Basic Plot", xlab="Overs", ylab="Runs", type ="b",
col="red")
```

Output:



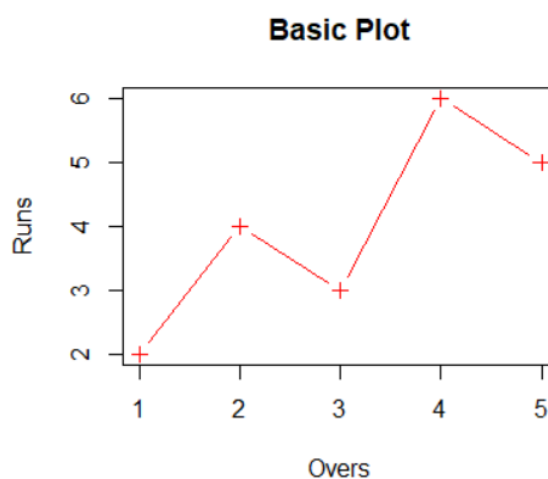
- **Shape of the points:**

You can specify the shape of the points using “*pch*” argument. By default, the shape of the points is “circle”. The possible values for *pch* are from 1 to 25.

Example:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Basic Plot", xlab="Overs", ylab="Runs", type ="b",
col="red", pch ="3")
```

Output:



Adding Lines: The `abline()` Function

The **`abline()`** is short for "add line" — it is used to add **horizontal, vertical, or regression lines** to an existing plot.

Syntax:

`abline(a, b, col, lty, lwd)`

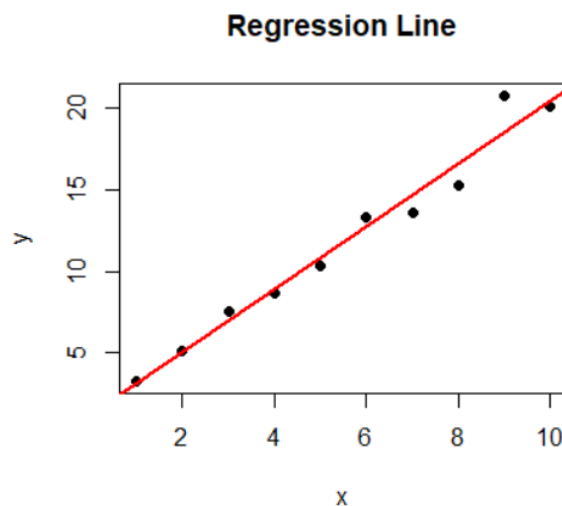
In the above syntax,

a : Intercept of the line (for $y = a + b \cdot x$)
b : Slope of the line
col : Color of the line
lty : Line type (solid, dashed, etc.)
lwd : Line width (thickness)

- **Add a Regression Line to a Scatter Plot:**

Example:

```
> x <- 1:10  
> y <- 2 * x + 1 + rnorm(10) # y = 2x + 1 with some noise  
> plot(x, y, main = "Regression Line", pch = 16)  
> abline(lm(y ~ x), col = "red", lwd = 2)  
# Output:
```

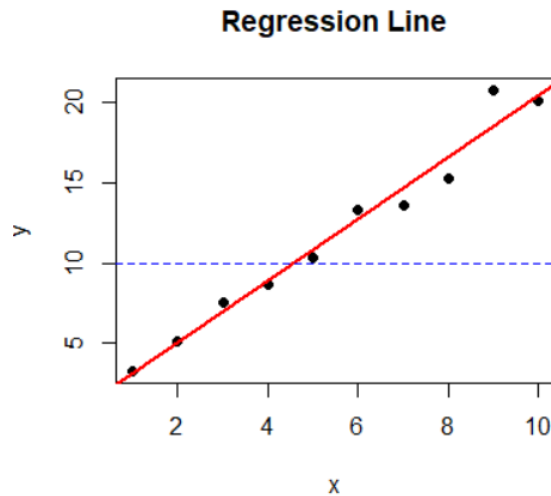


In the above code, **`lm(y ~ x)`** fits a linear model; **`abline()`** draws that line.

- **Add a Horizontal Line:**

```
> abline(h = 10, col = "blue", lty = 2)
```

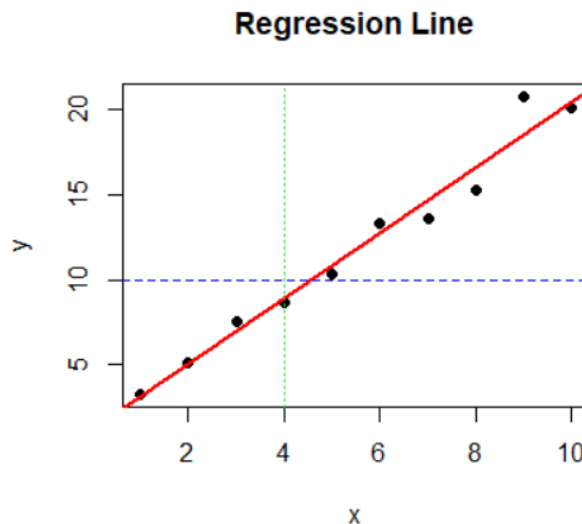
In this, $h = 10$ adds a horizontal line at $y = 10$ with a blue color dashed line.



- **Add a Vertical Line:**

```
> abline(v = 4, col = "green", lty = 3)
```

In this, $v = 4$ adds a vertical line at $x = 4$ with a green color dotted line.



Starting a New Graph While Keeping the Old Ones:

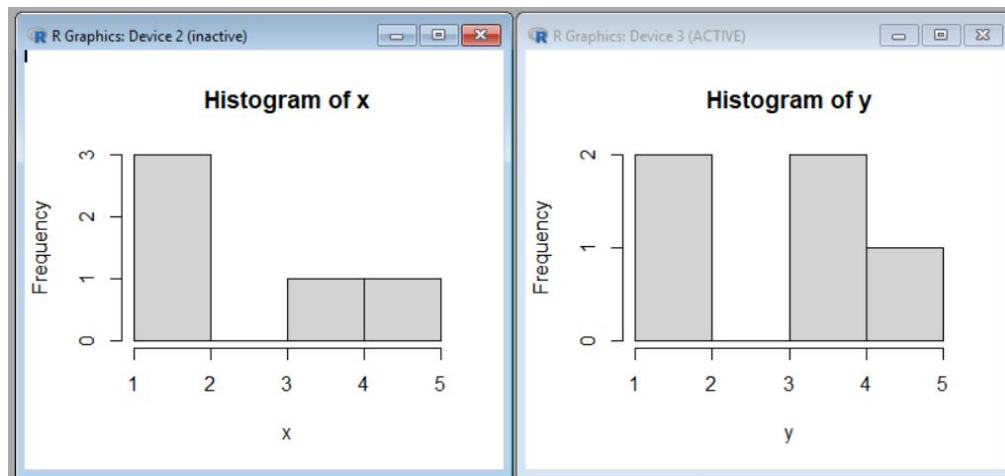
Each time you call `plot()`, directly or indirectly, the current graph window will be replaced by the new one. If you don't want that to happen, use the command for your operating system:

- 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 Windows system, you would type the following:

```
> x <- c(5,2,1,2,4)
> y <- c(4,5,4,2,1)
> hist(x)
> windows()
> hist(y)
```

Output:



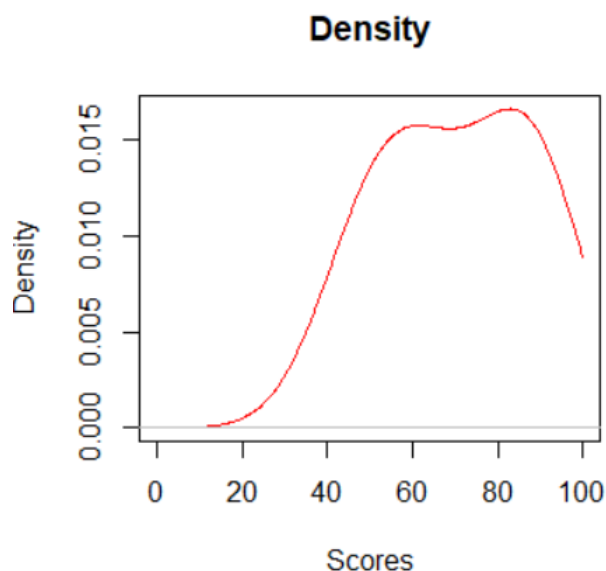
Two Density Estimates on the Same Graph:

Plotting **two density estimates on the same graph** is a very common and useful technique in R, especially to compare distributions of two datasets. For this, we use *density()* and *lines()* functions.

Example:

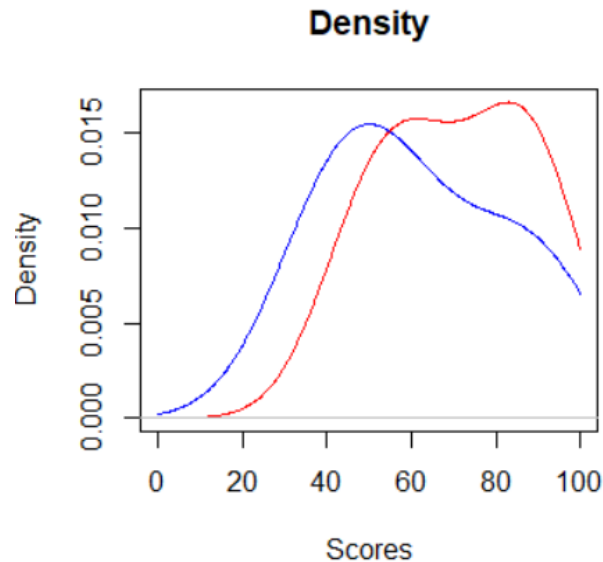
```
> exam1 <- c(78,88,92,56,46,64)
> exam2 <- c(63,52,36,82,94,45)
> d1 = density(exam1,from=0,to=100)
> d2 = density(exam2,from=0,to=100)
> plot(d1,main="Density",xlab="Scores",col="red")
```

In the above code, First, we compute nonparametric density estimates from the two variables (exam1 and exam2), saving them in objects d1 and d2 by using *density()* function. We then call *plot()* to draw the curve for exam 1, at which point the plot looks like below,



We then call `lines()` to add exam 2's curve to the graph, at which point the plot looks like below.

```
> lines(d2,col="blue")
```



The `plot()` function starts the graph and also draws the **first curve**. Then, the `lines()` function is used to **add the second curve** on top of the same graph.

Adding a Legend: The legend() Function:

The `legend()` function is used to add a legend to a multicurve graph. This could tell the viewer something like, "The red curve is for the exam1, and the blue curve displays the data for the exam2."

Syntax:

```
legend(pos, legend, col, lty, lwd, ...)
```

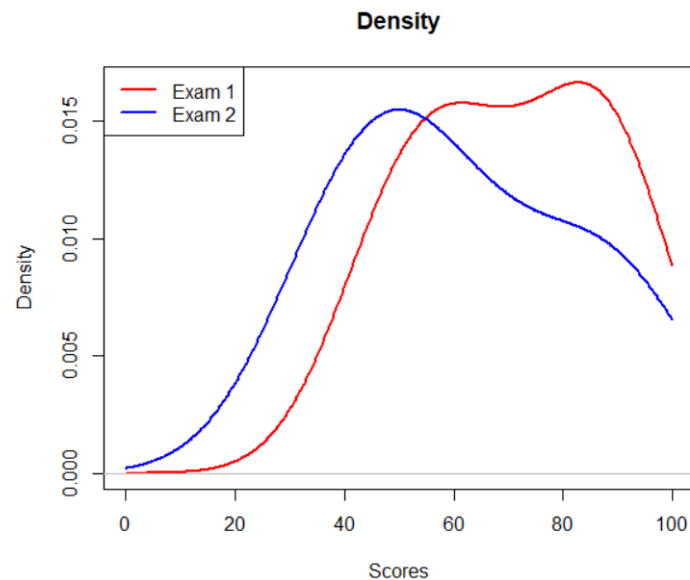
In the above syntax,

pos : Position of the legend (like "topright")
legend : A vector of labels
col : Colors of lines or points
lty : Line type (1 = solid, 2 = dashed, etc.)
lwd : Line width

Example:

```
> exam1 <- c(78,88,92,56,46,64)
> exam2 <- c(63,52,36,82,94,45)
> d1 = density(exam1, from = 0, to = 100)
> d2 = density(exam2, from = 0, to = 100)
# Plot first density
> plot(d1, main = "Density", xlab = "Scores", col = "red", lwd = 2)
# Add second density
> lines(d2, col = "blue", lwd = 2)
# Add legend
> legend("topleft", legend = c("Exam 1", "Exam 2"), col = c("red",
"blue"), lty = 1, lwd = 2)
```

Output:



Adding Text: The text() Function

The text() function is used to add labels or custom text to specific locations on a plot.

Syntax:

text(x, y, labels, col, pos, cex, ...)

In the above syntax,

x, y : Location on the plot where the text goes

labels : The text you want to show

col : Text color

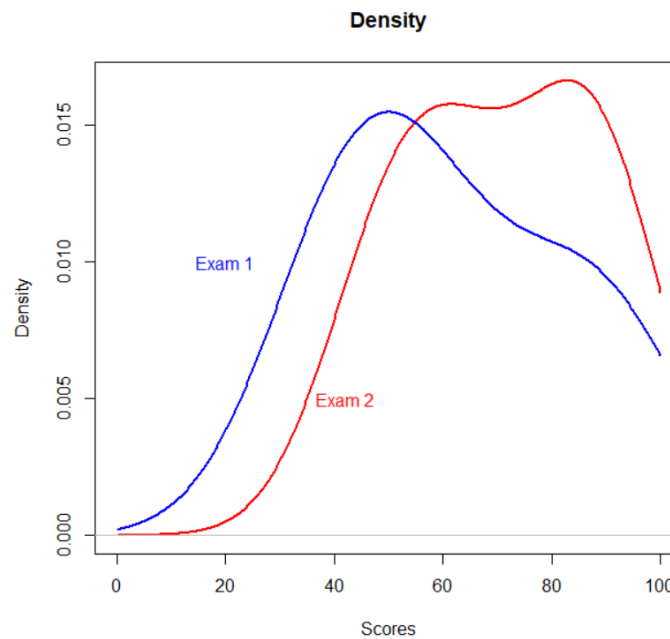
pos : Position relative to the point (1=below, 2=left, 3=above, 4=right)

cex : Text size (default is 1)

Example:

```
> exam1 <- c(78,88,92,56,46,64)
> exam2 <- c(63,52,36,82,94,45)
> d1 = density(exam1, from = 0, to = 100)
> d2 = density(exam2, from = 0, to = 100)
> # Plot first density
> plot(d1, main = "Density", xlab = "Scores", col = "red", lwd = 2)
> # Add second density
> lines(d2, col = "blue", lwd = 2)
> # Adding Text to Lines
> text(20,0.010,"Exam 1",col="blue")
> text(42,0.005,"Exam 2",col="red")
```

Output:



Customizing Graphs:

You have seen how easy it is to create basic graphs by starting with the `plot()` function. Now, you can make your graphs better and more detailed using the many features and tools that R offers.

Changing Character Sizes: The `cex` Option

The ***cex*** stands for ***character expand***. It controls the size of Points, Text, and Labels.

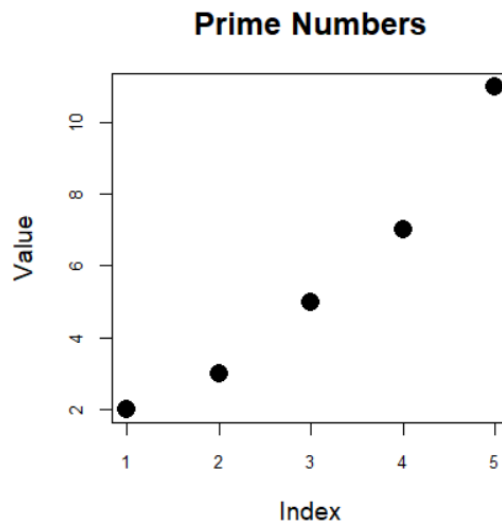
Basic Usage

<code>cex</code>	: Size of plot symbols (like points)
<code>cex.main</code>	: Size of main title
<code>cex.lab</code>	: Size of axis labels
<code>cex.axis</code>	: Size of axis tick labels
<code>cex.names</code>	: Size of bar names (for barplot)

Example 1:

```
> x <- c(1,2,3,4,5)
> y <- c(2,3,5,7,11)
> plot(x, y, pch = 16, cex = 2,      # big points
      main = "Prime Numbers",
      xlab = "Index", ylab = "Value",
      cex.main = 1.5,                # larger title
      cex.lab = 1.2,                 # larger axis labels
      cex.axis = 0.8)                # smaller axis numbers
```

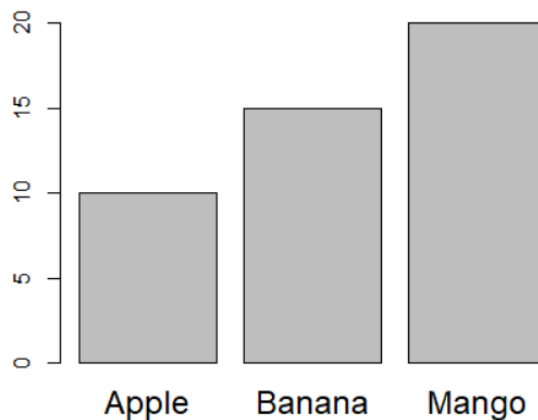
Output:



Example 2: Barplot with Bigger Names

```
> counts <- c(10, 15, 20)
> names <- c("Apple", "Banana", "Mango")
> barplot(counts, names.arg = names, cex.names = 1.5)
```

Output:



Changing the Range of Axes: The xlim and ylim Options

You may wish to have the ranges on the x- and y-axes of your plot be broader or narrower by using *xlim* and/or *ylim* options.

- **xlim**: Sets the range of values shown on the **x-axis**
- **ylim**: Sets the range of values shown on the **y-axis**

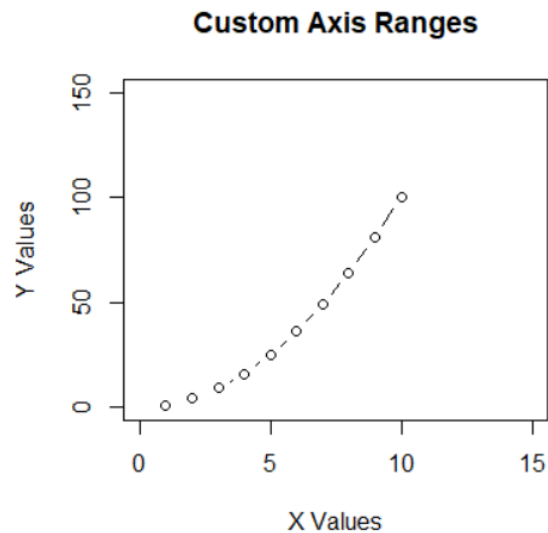
They both take a vector of two numbers: the minimum and maximum of the range.

For example, `xlim=c(0,10)` specifies a range on the x-axis of 0 to 10 and `ylim=c(0,90)` specifies a range on the y-axis of 0 to 90.

Example:

```
> x <- 1:10
> y <- x^2
> plot(x, y, type = "b",           # line and points
      xlim = c(0, 15),           # x-axis goes from 0 to 15
      ylim = c(0, 150),          # y-axis goes from 0 to 150
      main = "Custom Axis Ranges",
      xlab = "X Values", ylab = "Y Values")
```

Output:



Adding a Polygon: The polygon() Function

The ***polygon()*** function is used to draw a filled shape (usually between lines or around points) by connecting a series of (x, y) coordinates. You can use it to highlight areas under curves, create custom shaded areas, and more.

Syntax:

```
polygon(x, y, col = "color", border = "color")
```

In the above,

x: a vector of x coordinates

y: a vector of y coordinates (same length as *x*)

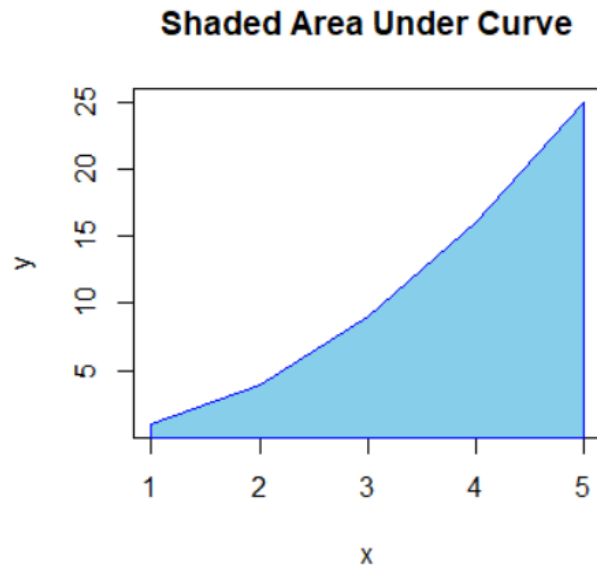
col: fill color of the polygon

border: color of the outline (optional)

Example:

```
> x <- 1:5
> y <- x^2
> # Plot the curve
> plot(x, y, type = "l", main = "Shaded Area Under Curve")
> # Add polygon below the curve
> polygon(c(x, rev(x)), c(rep(0, length(y)), rev(y)),
      col = "skyblue", border = "blue")
```

Output:



Explanation:

In the above example,

$$x = [1, 2, 3, 4, 5]$$

$$y = [1, 4, 9, 16, 25]$$

rev(x) means reverse x:

$$\text{rev}(x) = [5, 4, 3, 2, 1]$$

So,

$$c(x, \text{rev}(x)) = [1, 2, 3, 4, 5, 5, 4, 3, 2, 1]$$

rep(0, length(y)) (for shading under curve):

$$\text{rep}(0, 5) = [0, 0, 0, 0, 0]$$

$$\text{rev}(y) = [25, 16, 9, 4, 1]$$

$$c(\text{rep}(0, \text{length}(y)), \text{rev}(y)) = [0, 0, 0, 0, 0, 25, 16, 9, 4, 1]$$

You are building a path like this:

1. Start from left to right along x-axis (bottom line): $(1,0) \rightarrow (2,0) \rightarrow (3,0) \rightarrow (4,0) \rightarrow (5,0)$
2. Go **up and back** along the curve: $(5,25) \rightarrow (4,16) \rightarrow (3,9) \rightarrow (2,4) \rightarrow (1,1)$
3. This forms a **looped shape** under the curve — perfect for shading.

Smoothing Points: The lowess() and loess() Functions

The lowess() and loess() in R, are used to **draw smooth curves through scattered data** to better understand trends.

Both are smoothing techniques:

- **lowess()**: Stands for *Locally Weighted Scatterplot Smoothing*. Simple and good for small to medium datasets.
- **loess()**: More advanced, can handle more complex relationships and is slightly more flexible than lowess().

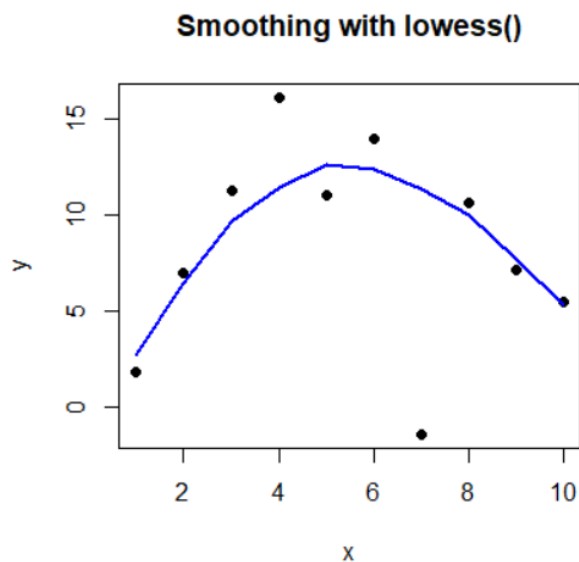
Example: lowess() function

```
> # Sample scattered data
> x <- 1:10
> y <- x + rnorm(10, 0, 5) # some noisy data

> # Plot the original data points
> plot(x, y, main = "Smoothing with lowess()", pch = 19)

> # Add smoothed line
> lines(lowess(x, y), col = "blue", lwd = 2)

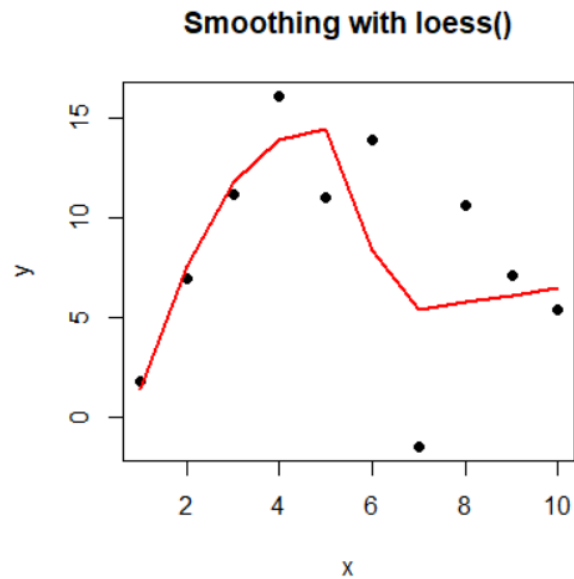
# Output:
```



Example: loess() function

```
> # Use loess to fit a smooth curve
> fit <- loess(y ~ x)
> # Get smoothed y-values for the x-values
> smoothed <- predict(fit)
> # Plot
> plot(x, y, main = "Smoothing with loess()", pch = 19)
> lines(x, smoothed, col = "red", lwd = 2)
```

Output:



Graphing Explicit Functions

An **explicit function** is one where you can directly express **y in terms of x**, like:

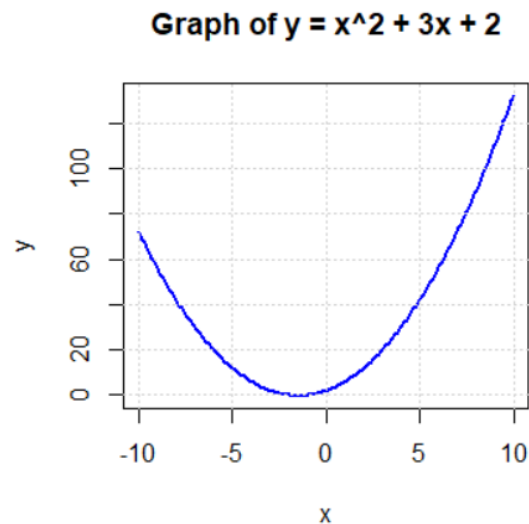
$$y = x^2 + 3x + 2$$

In R, you can graph such functions easily using `plot()`.

Example:

```
> x <- seq(-10, 10, 0.1)      # x-values from -10 to 10
> y <- x^2 + 3*x + 2          # Compute y-values
> plot(x, y, type = "l", main = "Graph of y = x^2 + 3x + 2",
       xlab = "x", ylab = "y", col = "blue", lwd = 2)
> grid()                      # Add grid lines
```

Output:



Saving Graphs to Files

R graphics can use different display devices. The screen is the default one. To save a graph as a file, you need to choose a different device. Let us start by learning the basics of R graphics devices.

R Graphics Devices:

In R, the graphics devices are used to save plots as files like PNG, PDF, JPEG, etc.

Common Devices:

Format	Function	File Extension
PNG	png()	.png
JPEG	jpeg()	.jpg / .jpeg
PDF	pdf()	.pdf
BMP	bmp()	.bmp
TIFF	tiff()	.tiff / .tif
SVG	svg()	.svg

General Format:

`device_function ("filename", width=..., height=...)`

Example:

`png("my_plot.png", width=600, height=400)`

Saving the Graphs:

In R, you can save the graphs using device functions in various formats like PNG, PDF, JPEG, etc.

Example: Save to PNG

```
> # Open PNG device
> png("cricket_plot.png", width=600, height=400)
> # plot a Graph
> x <- c(1, 2, 3, 4, 5)
> y <- c(2, 4, 3, 6, 5)
> plot(x, y, main="Score Plot", xlab="Overs", ylab="Runs", type="b",
col="red", pch=16, lwd=2)
> # Close the device and save the file
> dev.off()
```

Output:

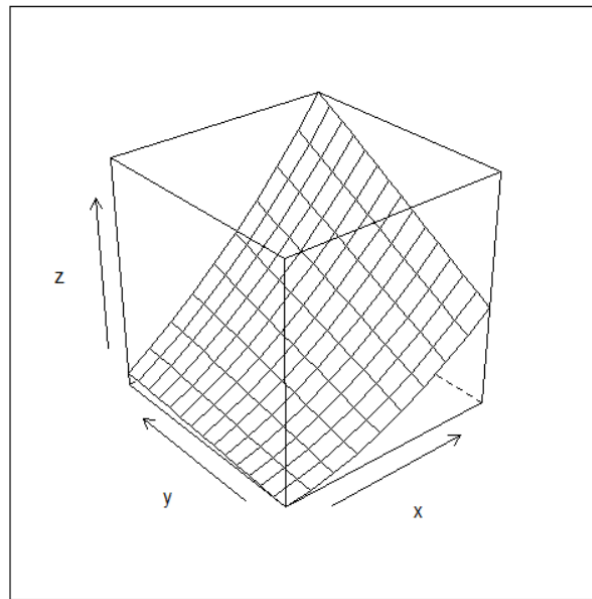
The above code saves the graph as “**cricket_plot.png**” with the width 600px and height 400px. And the function **dev.off()** close the device and save the file.

Creating Three-Dimensional Plots

R provides several functions for creating 3D plots. The *persp()* and *wireframe()* functions are used to plot data in three dimensions.

Example: using *wireframe()*

```
> 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)
# Output:
```



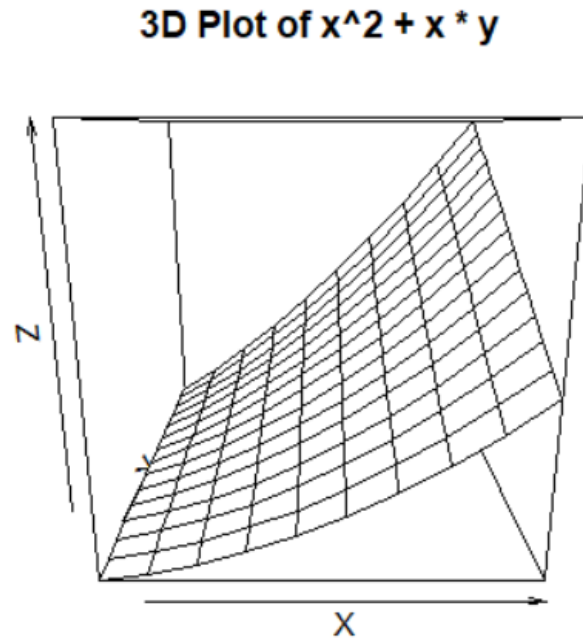
In the above example, First, we load the *lattice library*. Then the call to *expand.grid()* creates a **data frame**, consisting of two columns named x and y, in all possible combinations of the values of the two inputs. Here, a and b had 10 and 15 values, respectively, so the resulting data frame will have 150 rows.

We then added a third column, named z, as a function of the first two columns. The arguments, given in regression model form, specify that z is to be graphed against x and y.

Example: using *persp()*

```
> # Create x and y values
> x <- 1:10
> y <- 1:15
> # Create a matrix of z values based on x and y
> z <- outer(x, y, function(x, y) x^2 + x * y)
> # Make the 3D plot
> persp(x, y, z, main = "3D Plot of x^2 + x * y",
        xlab = "X", ylab = "Y", zlab = "Z")
```

Output:



While using ***persp()*** and ***wireframe()*** functions, all the points are connected as a surface (like connecting points by lines in two dimensions). In contrast, with ***cloud()***, the points are isolated.

Debugging

Debugging is the process of identifying, analyzing, and fixing bugs or errors in a program. In R, debugging is especially important due to the interpreted nature of the language, which makes some errors appear only during runtime.

Fundamental Principles of Debugging:

1. Understand the Error Types:

- **Syntax Errors:** Occur when R cannot parse the code (e.g., missing commas or parentheses).
- **Runtime Errors:** Happen during the execution phase (e.g., accessing a non-existent variable).
- **Logical Errors:** The code runs without error but produces incorrect results.

2. Use Informative Error Messages:

- Always read and interpret error messages carefully; they usually indicate what went wrong and where.
- Example: Error in eval(expr, envir, enclos) : object 'x' not found.

3. Use Print Statements for Tracing:

- Insert `print()` or `cat()` statements in your code to track variable values and flow.
- Helps detect where the program deviates from expected behavior.

4. Check Input and Output Values:

- Verify data types, expected input structures (e.g., vectors vs. lists), and function outputs.
- This prevents type-related errors.

5. Write Modular Code:

- Break your code into smaller functions, which makes it easier to identify where an issue originates.

6. Test with Simplified Data:

- Reduce complexity by testing your function or script with minimal data to isolate the problem.

Why Use a Debugging Tool in R:

In the old days, programmers would perform the debugging confirmation process by temporarily inserting print statements into their code and rerunning the program to see what printed out.

Example:

```
x <- 5
y <- calc(x)
if (x+y > 20) {
  u <- 1
  print("the 'if' was done")
} else {
  v <- 10
  print("the 'else' was done")
}
```

We would rerun the program and inspect the feedback printed out. We would then remove the print statements and put in new ones to track down the next bug.

This manual process is fine for one or two cycles, but it gets boring during a long debugging session. And worse, all that editing work distracts your attention, making it harder to concentrate on finding the bug.

So, debugging by inserting print statements into your code is slow, cumbersome, and distracting. If you are serious about programming in any particular language, you should seek a good debugging tool for that language.

Using a debugging tool will make it much easier to query the values of variables, check whether the if or the else gets executed, and so on. Moreover, if your bug causes an execution error, debugging tools can analyze it for you, possibly providing major clues as to the source of the error. All of this will increase your productivity substantially.

Using R Debugging Facilities:

R provides several built-in debugging facilities to help identify, inspect, and fix errors during code execution. These tools allow developers to pause execution, check variable states, and understand the sequence of function calls that led to an error.

1. Using *traceback()*:

- Shows the **call stack** after an error has occurred.
- Useful to see which function caused the error and what other functions were called.

Usage:

```
> some_function()
> traceback()
```

2. Using *browser()*:

- Inserts a **breakpoint** inside a function.
- Execution pauses, and you can interactively inspect and modify variables.

Usage:

```
my_function <- function(x) {  
  browser()  
  x + 10  
}  
my_function(5)
```

3. Using *debug()* and *undebug()*:

- Used to **step through** an entire function line by line.
- Helps inspect the flow and variable values at each step.

Usage:

```
debug(my_function)  
my_function(5)  
undebug(my_function)
```

4. Using *recover()*:

- Gives access to all active function calls when an error occurs.
- Allows the user to **browse the environment** of each function in the call stack.

Usage:

```
options(error = recover)
```

5. Using *trace()*:

- Allows you to insert code into a function to track its execution.
- Often used for **monitoring function calls** without editing the function definition.

Usage:

```
> trace("mean", tracer = quote(print("mean function called")))  
> mean(c(1, 2, 3))  
> untrace("mean")
```

Example:

```
bad_function <- function(x) {  
  y <- x + z # z is not defined  
  return(y)  
}  
  
# Use debugging tools  
debug(bad_function)  
bad_function(10)  
undebug(bad_function)
```

Moving Up in the World: More Convenient Debugging Tools

While R's base debugging tools like `browser()`, `debug()`, `traceback()`, and `recover()` are powerful, they can be manual and limited in large projects or interactive development. "Moving up in the world" means using more advanced and user-friendly debugging tools that streamline the process and provide better visualization, control, and convenience.

1. RStudio Built-in Debugger (IDE Debugging)

The RStudio IDE is the Most convenient tool for debugging in R.

Key Features:

- **Breakpoints:** Set breakpoints directly by clicking in the margin.
- **Step Through Code:** Use buttons to *Step Into*, *Step Over*, and *Continue*.
- **Environment Viewer:** Inspect all variables at each frame of execution.
- **Call Stack View:** See which functions were called and where you're paused.
- **Watch Window:** Monitor specific variables during execution.
- **No manual code changes needed** (like adding `browser()` or `debug()`).

2. `debugonce()`

- Similar to `debug()`, but only activates debugging for one call.
- Useful when you want to inspect a function *once* without altering future behavior.

Usage:

```
> debugonce(my_function)  
  
> my_function(10)
```

Ensuring Consistency in Debugging Simulation Code

If you are doing anything with random numbers, you will need to be able to reproduce the same stream of numbers each time you run your program during the debugging session. Without this, your bugs may not be reproducible, making them even harder to fix.

The `set.seed()` function controls this by reinitializing the random number sequence to a given value.

Consider this example:

```
> runif(3)
[1] 0.5775979 0.4588383 0.8354707
> runif(3)
[1] 0.4155105 0.4852900 0.6591892
> runif(3)
> set.seed(8888)
> runif(3)
[1] 0.5775979 0.4588383 0.8354707
> set.seed(8888)
> runif(3)
[1] 0.5775979 0.4588383 0.8354707
```

The call `runif(3)` generates three random numbers from the uniform distribution on the interval (0,1). Each time we make that call, we will get a different set of three numbers. But with `set.seed()`, we can start over and get the same sequence of numbers.

Syntax and Runtime Errors

In R, like in most programming languages, errors can generally be categorized into two main types:

- Syntax Errors
- Runtime Errors

Syntax Errors:

Syntax errors occur when the code violates the rules of the R language grammar. These errors are detected before the code is executed (i.e., at parsing time).

Common Causes:

- Missing parentheses, brackets, or braces
- Typos in function or variable names
- Incorrect use of operators or punctuation
- Improper string termination

Example:

```
> x <- c(1, 2, 3 # Missing closing parenthesis

> mean[1, 2] # Incorrect use of brackets instead of parentheses
```

Runtime Errors:

Runtime errors occur when R successfully parses the code but encounters a problem during execution. These errors only show up while the code is running.

Common Causes:

- Referring to undefined variables
- Invalid data types (e.g., adding a number to a string)
- Index out of bounds
- Division by zero
- Function misuse

Example:

```
> x <- "text"
> y <- x + 1 # Trying to add a number to a string

# Error in x + 1 : non-numeric argument to binary operator

> z <- log(-1)
# Warning message: In log(-1) : NaNs produced
```

Running GDB on R Itself:

You may have written some C code to interface to R and found it to be buggy. In order to run GDB on that C function, you must first run R itself through GDB.

Although you can invoke R through GDB from a shell command line, for our purposes here, I suggest using separate windows for R and GDB. Here is the procedure:

1. Start R in one window, as usual.
2. In another window, determine the ID number of your R process. In UNIX family systems, for instance, this is obtained by something like

ps -a.

3. In that second window, submit GDB's attach command with the R process number.
4. Submit the continue command to GDB.

You can set breakpoints in the R source code either before continuing or by interrupting GDB later with CTRL-C.

The R source code is dominated by S expression pointers (SEXP), which are pointers to C structs that contain an R variable's value, type, and so on. You can use the R internal function `Rf_PrintValue(s)` to inspect SEXP values. For example, if the SEXP is named `s`, then in GDB, type this:

call Rf_PrintValue(s)

This prints the value.