Introduction to R language

M. Beccuti

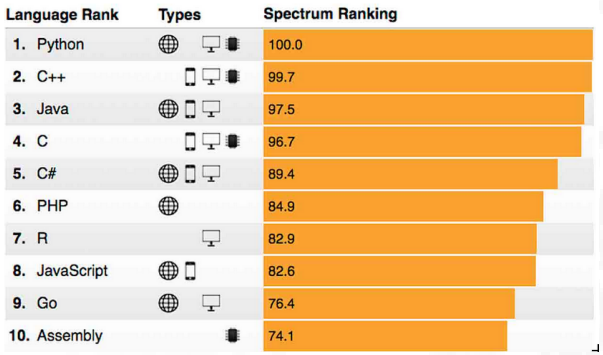
Università degli Studi di Torino

Bioinformatics Course

May 2019

The R Project

- Environment for statistical computing and graphics:
- Free software and Open-source;
- A simple programming language:
  - it is an open-source implementation of S language;
  - it is among the Top 10 Programming Languages in 2018 for *IEEE Spectrum Journal*;



- software and packages can be downloaded from:

  www.cran.r-project.org

- Versions of R exist of Windows, MacOS, Linux and various other Unix-like OS.

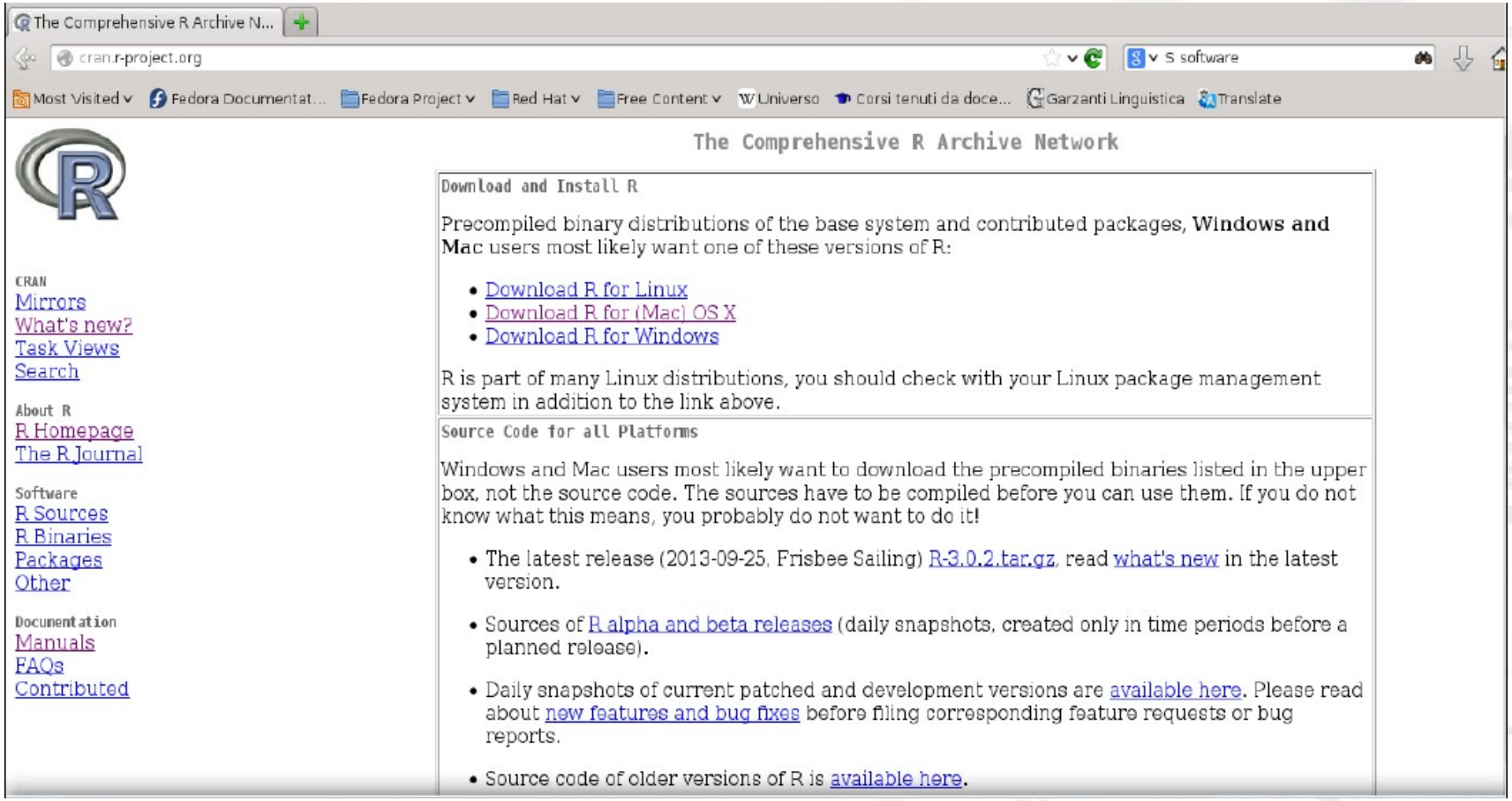
Why to use R language

- Implement many common statistical and bioinformatics procedures;
- Provide excellent graphics functionality;
- A convenient starting point for many data analysis projects
- Libraries (namely packages) can be automatically downloaded from:

  www.cran.r-project.org
  https://www.bioconductor.org/

- It is standard for data mining and statistical analysis;
- Efficient data structures make programming easier.

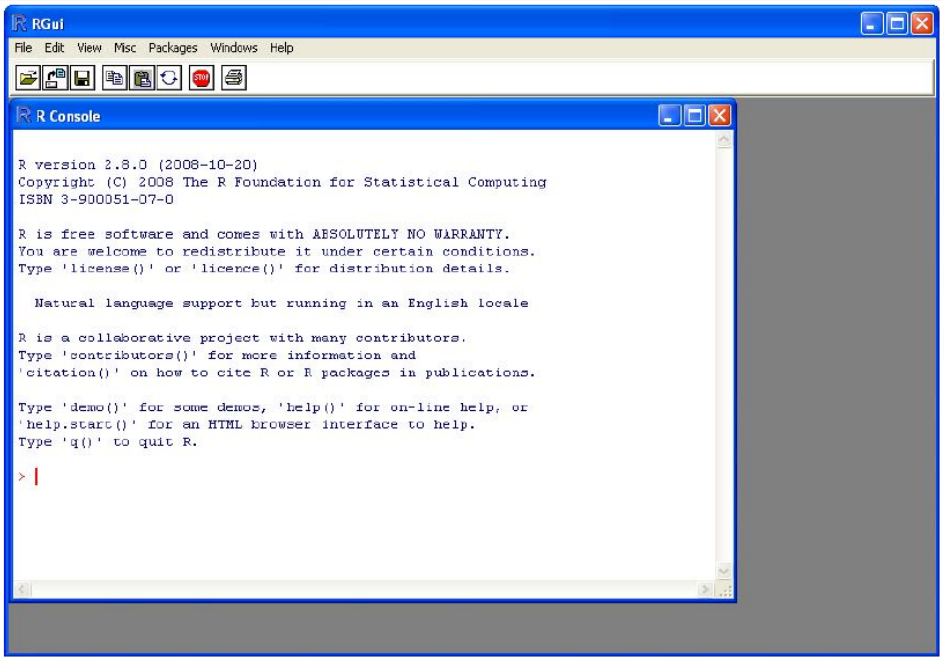
Download and Install R language



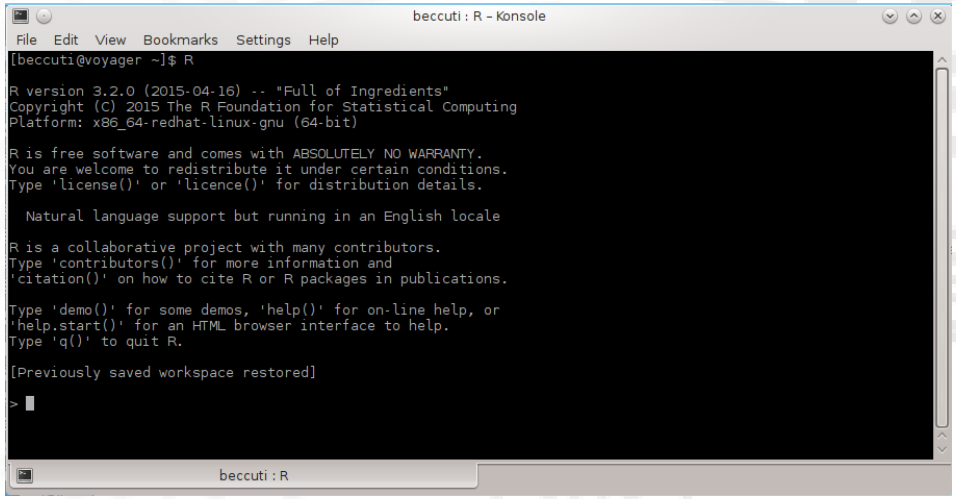
http://cran.mirror.garr.it/mirrors/CRAN/

Download the appropriate version (w.r.t. your OS) and follow the instructions to install the program.

R under GUI



from Windows

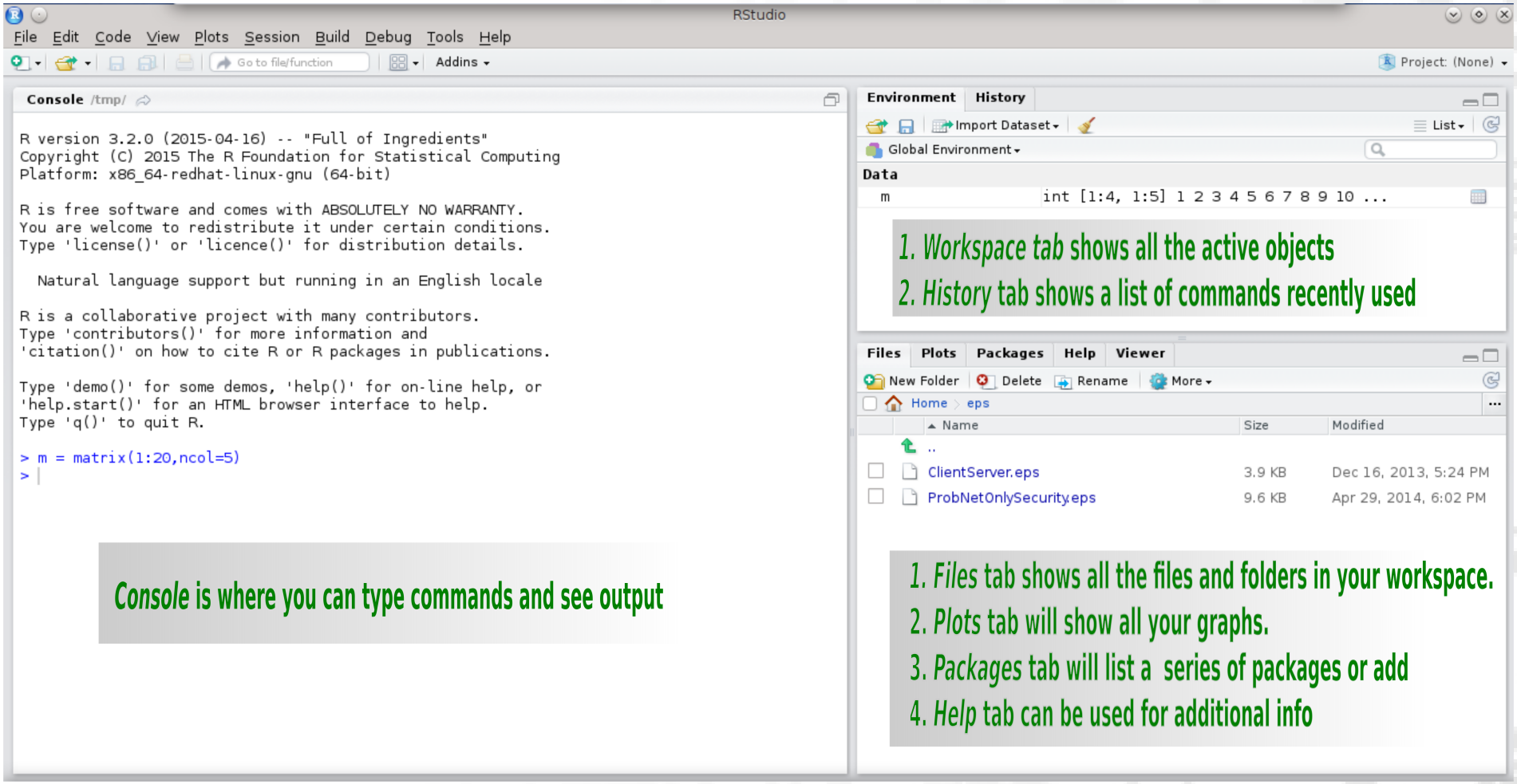


from Linux

R under GUI using Rstudio

RStudio allows the user to run R in a more user friendly environment.

It is open-source and available at `http://www.rstudio.com/`



Starting R

R can be started:

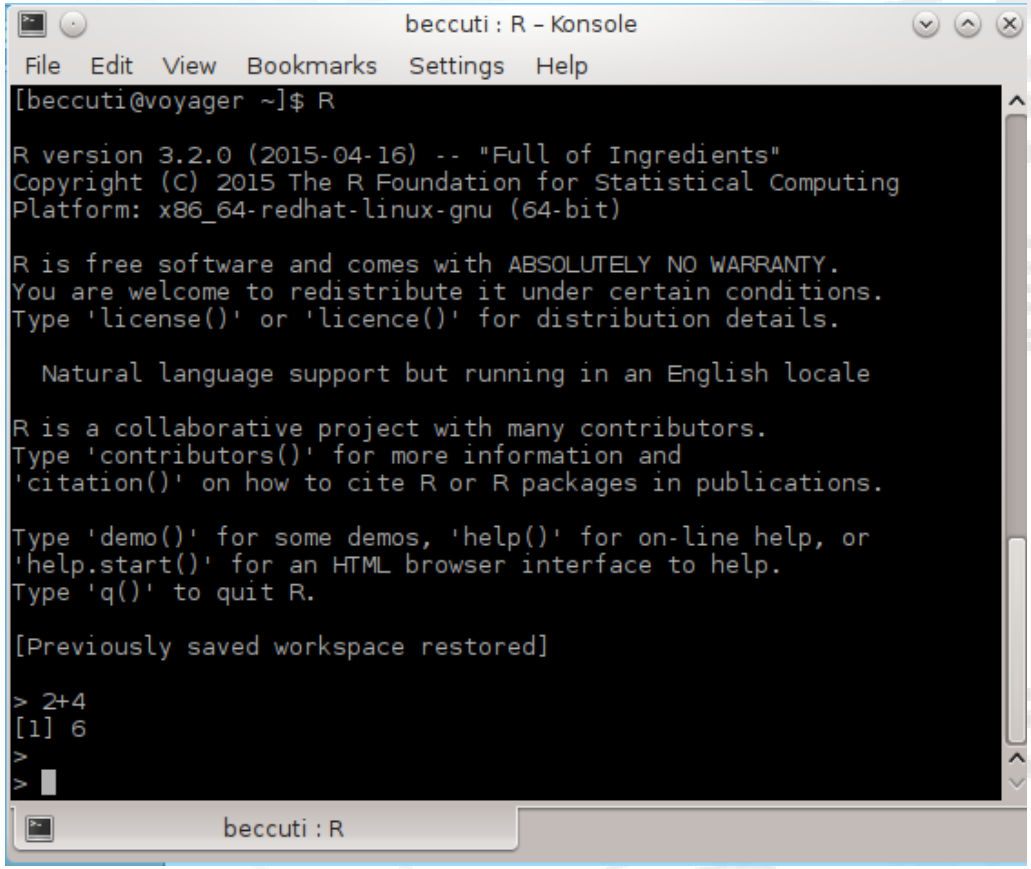
- by double-clicking on the R icon (e.g. Window);
- by double-clicking on the Rstudio icon (e.g. Window + Rstudio);
- by typing R in a shell (e.g. Linux).
- by typing rstudio in a shell (e.g. Linux + Rstudio).

How R works:

- R creates its objects in memory and saves them in a file called .RData (by default);
- Commands are recorded in an .Rhistory file, Command can be recalled using up- and down-arrow;
- Recalled commands may be edited;
- Commands may be abandoned by pressing <Esc>;
- To end your session type q() or just kill the window.
- A concept of *working directory* is introduced: each project is associated with a working folder containing each data.

Interactive R

- R defaults to an interactive mode;
- A prompt ">" is presented to users;
- Each input command is evaluated and a result returned;
- Commands
  - consist of expressions or assignments;
  - are separated by a semi-colon (;) or by a newline
  - can be grouped together using curly brackets({ and })



RStudio prompt and script

RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function | Addins | Project: (None)

Untitled7* | Untitled5* | Untitled8 | script.R*

Source on Save | Run | Source

1

1:1 (Top Level) R Script

Console ~/

```
Type 'demo()' for some demos, 'help()' for on
-line help, or
'help.start()' for an HTML browser interface
to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

>
```

Environment | History

Import Dataset | List

Global Environment

Environment is empty

Files | Plots | Packages | Help | Viewer

R: Solver for Ordinary Differential Equations (ODE) | Find in Topic

lsode {deSolve} R Documentation

Solver for Ordinary Differential Equations (ODE)

Description

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

$$dy/dt = f(t,y)$$

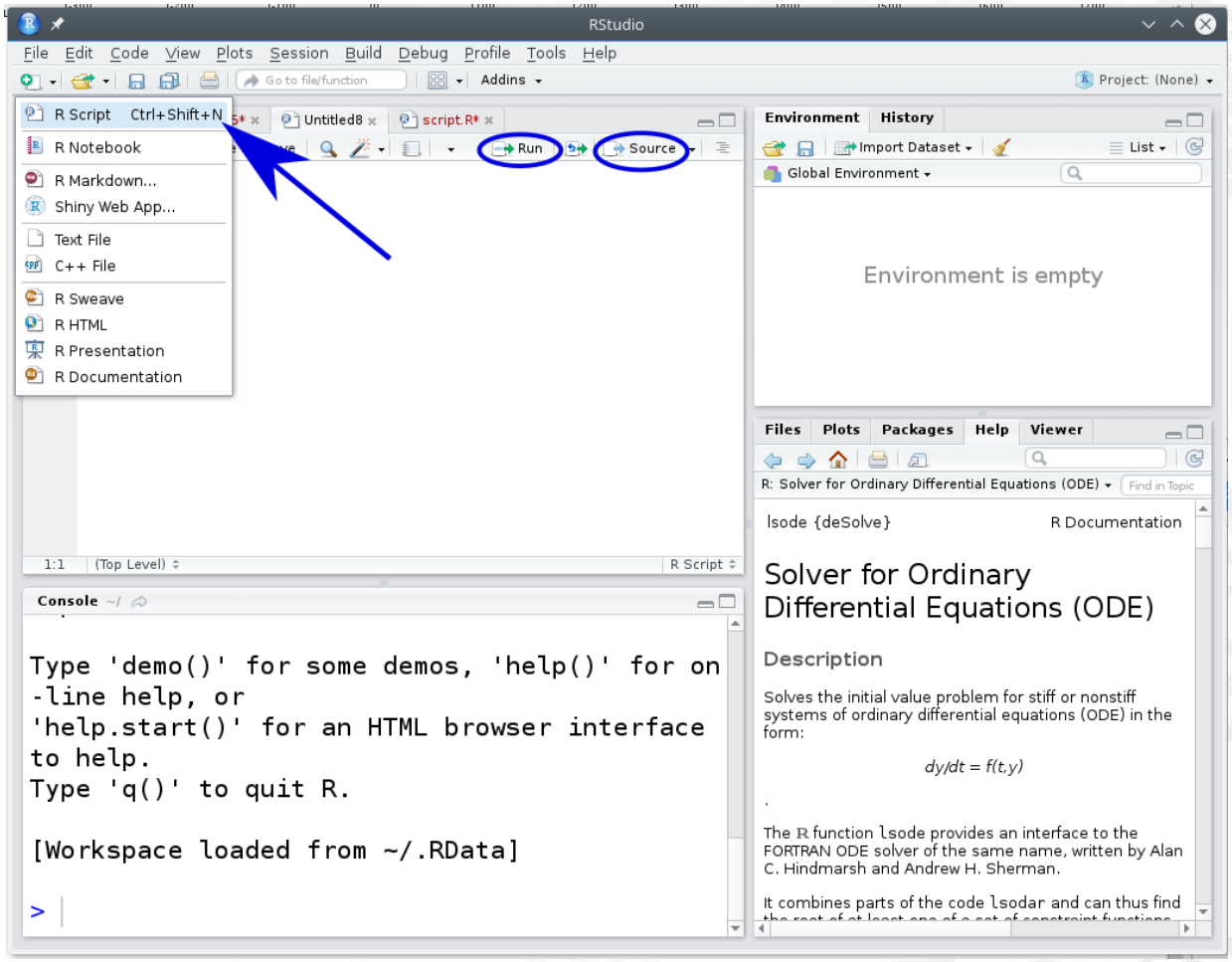
.

The R function lsode provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Andrew H. Sherman.

It combines parts of the code lsodar and can thus find

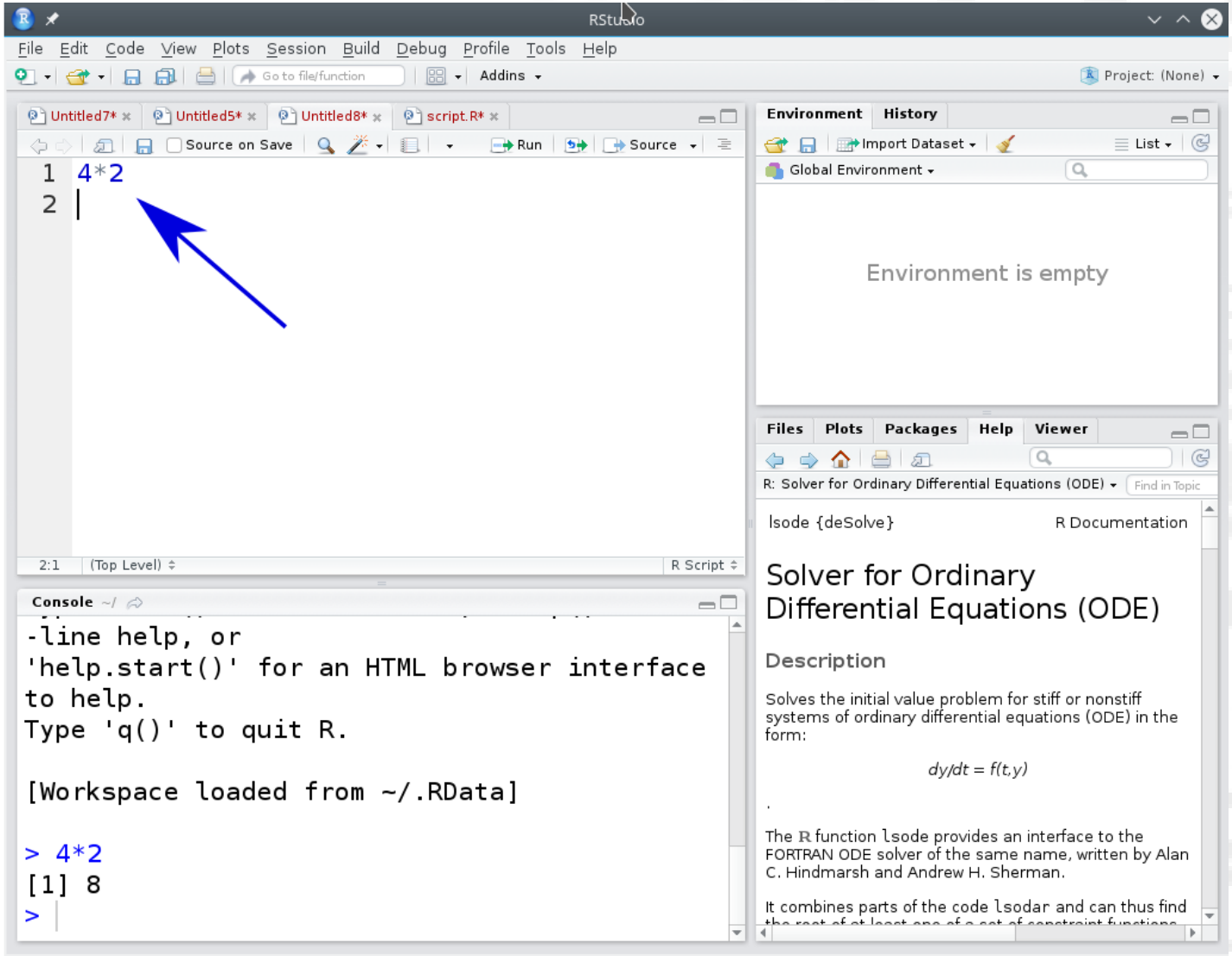
RStudio prompt and script

- R script can be used to save R commands into a file;
- Commands into R script can be executed line by line (clicking on Run) or globally (clicking on Source).



RStudio prompt and script

- Commands can be directly typed into the R script console.



R as a calculator

Simple Arithmetic

$> 3 + 4$

[1]7

Operator precedence

$> 2 + 3 * 5$

[1]17

Exponentiation

$> 3\hat{}5$

[1]243

Basic mathematical functions

$> exp(4)$

[1]54.59815

$> sqrt(4)$

[1]2

Predefined constant

$> pi$

[1]3.141593

$> Inf$

[1]Inf

Assignments in R

It is often required to store intermediate results so that they do not need to be re-typed over and over again. To assign a value of 324 to the variable X type:

$> X < -324$

or

$> X = 324$

Variable X can be used in next expressions:

Example

$> X$
[1]324

$> X + X$
[1]648

$> sqrt(X)$
[1]18

$> X = X + X; X$
[1]684

$> X/4$
[1]162

$> X\hat{}sqrt(X)$
[1]$1.54814e + 45$

Variable name in R

R is a case-sensitive language, hence x and X do not refer to the same variable.

Variable name:

- can be created using letters, digits and the . (dot) symbol;

  $> data1.address$
  $> d1_4.f$

- must not start with **a digit** or a . followed by a digit.

- some names are reserved by the system: *if, while, NULL, TRUE ...*

Variable type in R

Basic variable types are:

Numeric: integer, floating point values;

Boolean: values corresponding to **True** or **False**;

Strings: sequences of characters.

Type is determined automatically when variable is created with $<-$ or $=$ operator.

Data structures/Objects are: R provides types of different object.

Vector: a collection of elements (numbers, logical values and character strings) with same type;

Array: a generalization of a vector;

List: collections of objects of any type;
e.g. list of vectors, list of matrices, etc.

Data Frame an array in which the type of each element can be different;

Factor takes on a limited number of values;

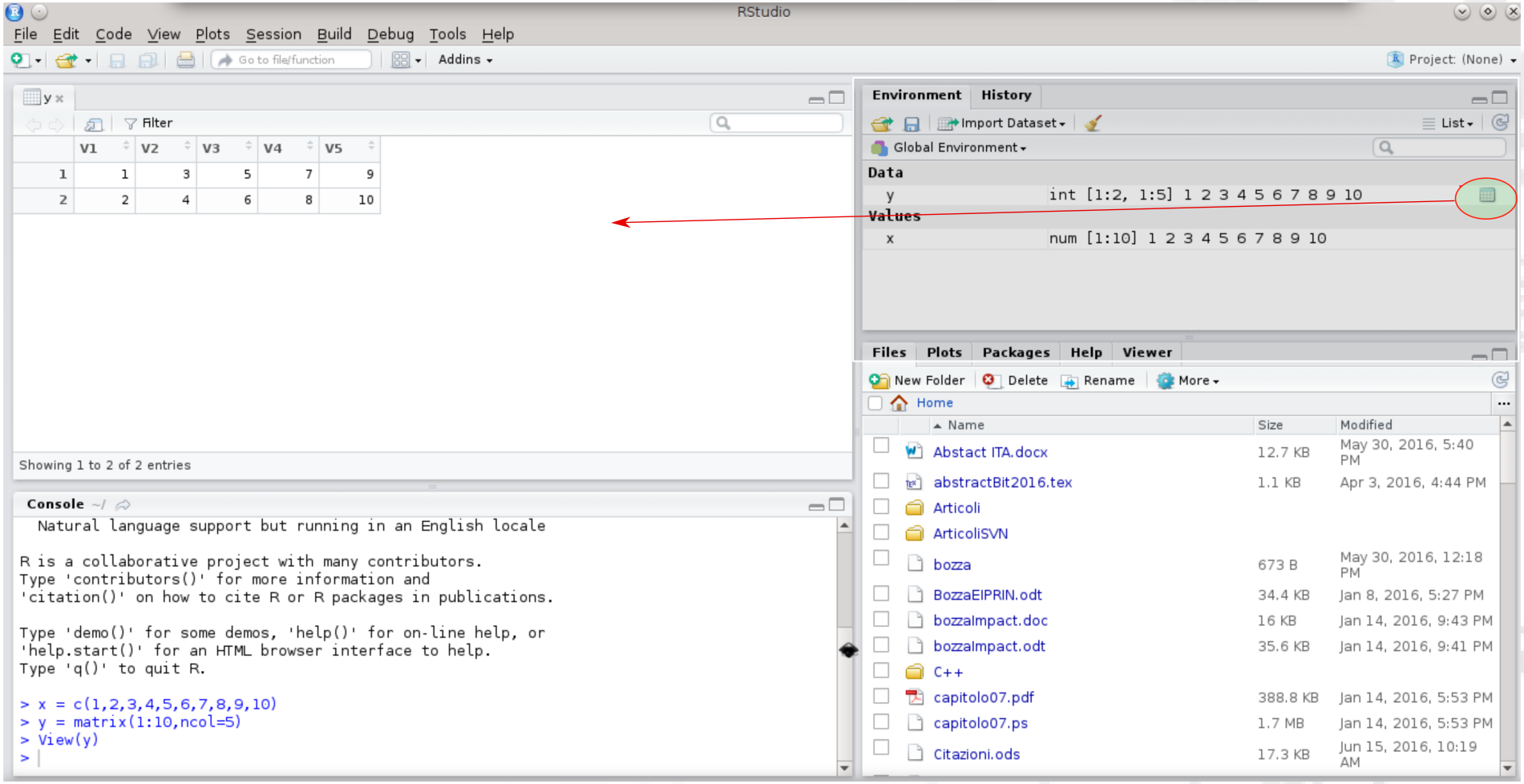
Variable in R

- During an R session, objects are created and stored by name;
- The command *ls()* displays all currently-stored objects (workspace);
- Objects can be removed using *rm(variable_name)*;
- All the objects in the workspace are removed using *rm(list=ls())*.

Observe

At the end of each R session, you are prompted to save your workspace. If you click Yes, all objects are written to the .RData file. When R is re-started, it reloads the workspace from this file and the command history stored in .Rhistory is also reloaded.

Variable in RStudio



Getting help in R

R provides a built-in help facility.

- To get more information on any specific function, e.g. sqrt(), the command is:
  help(sqrt)
  or
  ?sqrt

- help on features specified by special characters must enclose in single or double quotes (e.g. "[[") *help("[[")*

- Help is also available in HTML format by running *help.start()*

- For more information use
  ?help

Getting help in Rstudio

RStudio

File Edit Code View Plots Session Build Debug Tools Help

Go to file/function | Addins | Project: (None)

Console /tmp/

```
R version 3.2.0 (2015-04-16) -- "Full of Ingredients"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> m = matrix(1:20,ncol=5)
>
```

Environment | **History**

Import Dataset | List

Global Environment

Data

m	int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...

Files | **Plots** | **Packages** | **Help** | **Viewer**

R: List Objects | Find in Topic

ls {base} R Documentation

List Objects

Description

ls and objects return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, ls shows what data sets and functions a user has defined. When invoked with no argument inside a function, ls returns the names of the function's local variables: this is useful in conjunction with browser.

Usage

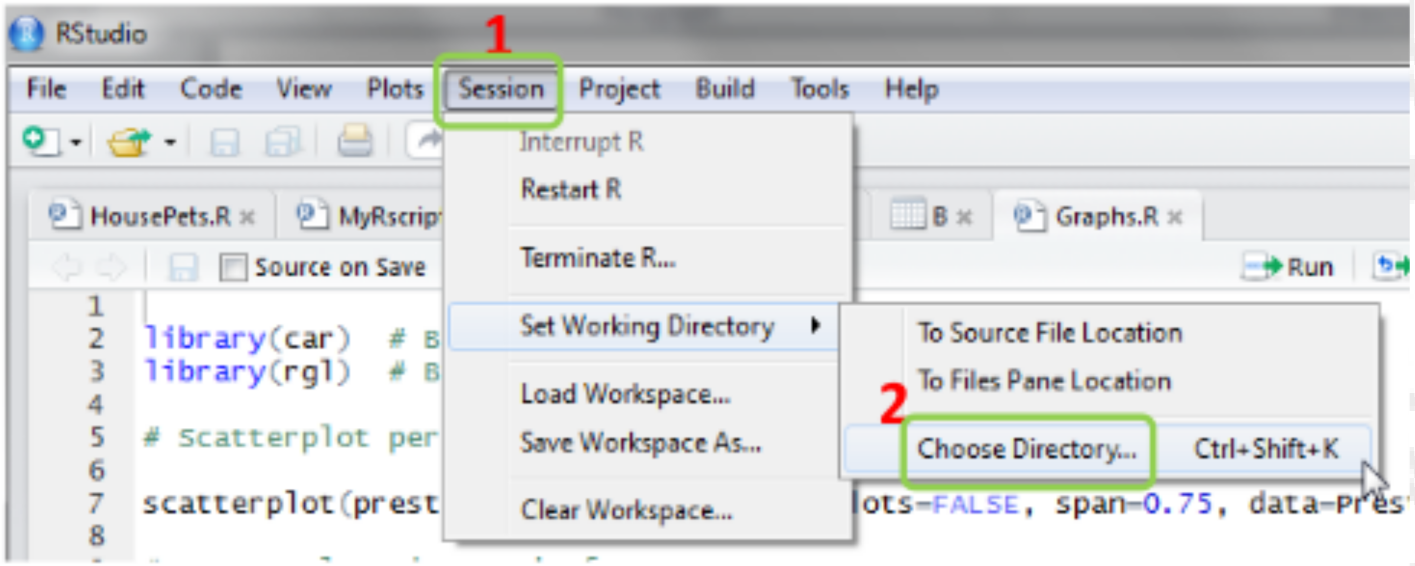
```
ls(name, pos = -1L, envir = as.environment(pos),
   all.names = FALSE, pattern, sorted = TRUE)
objects(name, pos= -1L, envir = as.environment(pos),
        all.names = FALSE, pattern, sorted = TRUE)
```

Working directory

Working directory in R:

- Working directory contains data and R scripts. It is a directory of the file-system;
- *getwd()* returns the current Working directory;
- *setwd("new_path")* sets Working directory;

Working directory in RStudio:



Packages in R

- R provides libraries of packages. Packages contain various functions and data sets for numerous purposes;
- Some packages are part of the basic installation. Others can be downloaded from CRAN:
  $> install.packages(\text{“}ggplot2\text{”})$
- To use functions and data sets of a package, it must be loaded into the workspace:
  $> library(ggplot2)$
- To check what packages are currently loaded into the workspace:
  $> search()$
- A loaded package can be removed:
  $> detach(\text{“}package:ggplot2\text{”})$

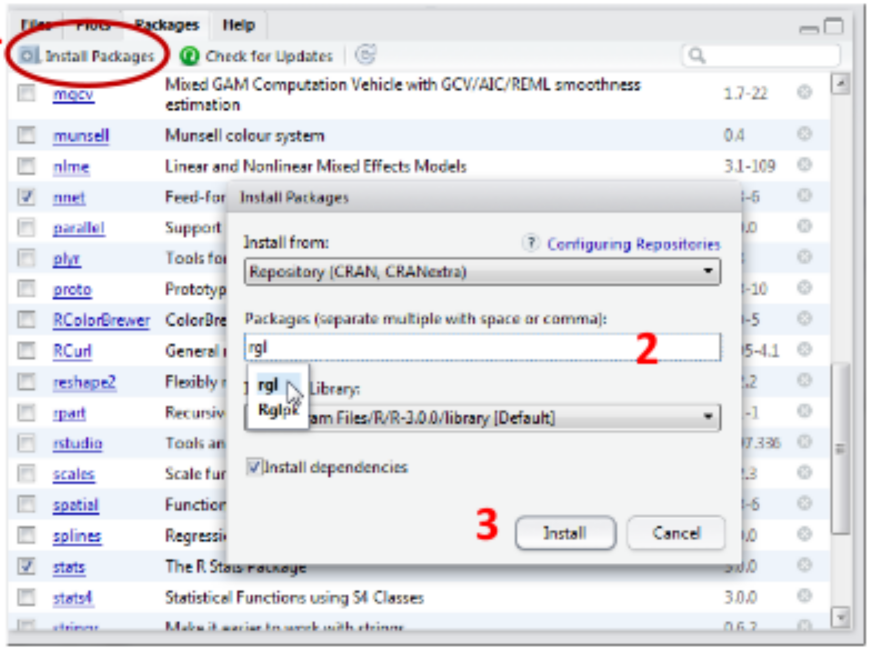
Observe:

if you terminated your session and start a new session with the saved workspace, you must load the packages again.

Packages in Rstudio

RCurl	General network (HTTP/FTP/...) client interface for R	1.95-4.1	
reshape2	Flexibly reshape data: a reboot of the reshape package.	1.2.2	
rpart	Recursive Partitioning	4.1-1	

Before



We focus on Package tab(bottom-right)

After

RCurl	General network (HTTP/FTP/...) client interface for R	1.95-4.1	
reshape2	Flexibly reshape data: a reboot of the reshape package.	1.2.2	
rgl	3D visualization device system (OpenGL)	0.93.952	
rpart	Recursive Partitioning	4.1-1	12

DSS/OTR

Vector in R

Vectors in R

- an ordered list of homogeneous elements;
- Vectors are the simplest type of object in R;
  There are 3 main types of vectors:
  - Numeric vectors;
  - Character vectors;
  - Logical vectors.
- To create a numeric vector x consisting of 6 numbers, 1.4, 6, 23.1, 65.43, 2.7, 55 use:

$$> x = c(1.4, 6, 23.1, 65.43, 2.7, 55)$$

or

$$> assign("x", c(1.4, 6, 23.1, 65.43, 2.7, 55))$$

Numeric vectors in R

- To print the contents of x:

  $> x$
  [1]1.4 6 23.1 65.43 2.7 55

  symbol [1] in front of the result is the index of the first element in the vector x.

- To access a particular element of x:

  $> x[1]$
  [1]1.4

  $> x[6]$
  [1]55

  $> x[c(1,6)]$
  [1]1.4 55

  $> x[-c(1,5)]$ **Operator - means: select all the elements except those ....**
  [1]6 23.1 65.43 55

Numeric vectors in R

- To modify a particular vector element:

  $> x[2] = 5$ **to modify the 2nd element of x in 5**
  [1]1.4 5 23.1 65.43 2.7 55

  $> x[4] = 5$
  [1]1.4 5 23.1 5 2.7 55

- To modify more than one vector elements:

  $> x[c(2, 4)] = c(6, 65.43)$
  [1]1.4 6 23.1 65.43 2.7 55

  $> y = x$
  $> y[y < 3] = 1$
  $> y$
  [1]1 6 23.1 65.43 1 55

Numeric vectors in R

- A vector can be used to do further assignments:

  $> y = c(x, 2, 3, x[c(1, 3)])$

  vector y with 10 entries is created:

  $> y$
  [1]1.4 6 23.1 65.43 2.7 55 2 3 1.4 23.1

- Operation are performed on each single element:

  $> x/10$
  [1]0.14 0.6 2.31 6.543 0.27 5.5

- Short vectors are "recycled" to match long ones (if it is possible):

  $> v = x[c(1, 2)] + y$ $x[c(1, 2)]$ **is repeated 5 times**
  $> v$
  [1]2.8 12 24.5 71.43 4.1 61 3.4 9 2.829.1

Numeric vectors in R

- Short vectors are "recycled" to match long ones (if it is possible)

  $> v = x + y$
  Warning message:
  In $x + y$ ***: longer object length is not a multiple of shorter object length***

- Some functions take vectors of values and produce results of the same length:

  sin, cos, tan, asin, acos, atan, log, exp, ...

  $> log(x)$
  [1]0.3364722 1.7917595 3.1398326 4.1809809 0.9932518 4.0073332

- Some functions return a single value:

  sum, mean, max, min, prod, ...

$> length(x)$
[1]6
$> sum(x)$
[1]153.63
$> sum(x)/lenght(x)$
[1]25.605

$> mean(x)$
[1]25.605
$> max(x)$
[1]65.43
$> min(x)$
[1]1.4

Numeric vectors in R

- Some special functions are:

  sort, cumsum, cumprod, pmax, pmin, range...

$> x$
[1]1.4 6 23.1 65.43 2.7 55

$> sort(x)$
[1]1.40 2.70 6.00 23.10 55.00 65.43

$> cumsum(x)$ **cumulative sums**
[1]1.40 7.40 30.50 95.93 98.63 153.63

$> y = c(2, 3, 5, 6, 100, 9)$
$> pmax(x, y)$ **max among 2 or more vector/scalar**
[1]2 6 23.1 65.43 100 55

$> pmin(x, y)$
[1]1.40 3 5 6 2.7 9

$> range(x)$
[1]1.40 65.43

How to generate sequences in R

- In R it is possible to generate sequences of numbers
  - using operator ":"
    $> 1 : 5$
    [1]1 2 3 4 5
  - using function seq()
    $> seq(1, 5)$
    [1]1 2 3 4 5
    $> seq(from = 1, to = 5)$
    [1]1 2 3 4 5
    We can also specify a step size (using by=value) or a length (using length=value) for the sequence.
    $> seq(1, 5, by = 0.5)$
    [1]1 1.5 2 2.5 3 3.5 4 4.5 5
    $> seq(from = 1, to = 5, length = 9)$
    [1]1 1.5 2 2.5 3 3.5 4 4.5 5
  - using function rep()
    $> rep(x, 3)$
    [1]1.40 6.00 23.10 65.43 2.70 55.00 1.40 6.00 23.10 65.43 2.70 55.00
    [13]1.40 6.00 23.10 65.43 2.70 55.00

Character vector in R

- A string is identify by " "
- A string vector is defined as well as number vector by c() operator
  $> y = c(\text{“}ROMA\text{”}, \text{“}MILANO\text{”}, \text{“}TORINO\text{”})$
- several functions in R to manipulate character vectors.

  paste, as.character, is.character, strsplit, substr. . .

$> paste(\text{“}HOME\text{”}, \text{“}WHILE\text{”}, \text{“}DOG\text{”}, sep = \text{“ : ”})$
$[1]\text{“}HOME : WHILE : DOG\text{”}$ **Concatenate char vectors**

$> x = c(1, 3, 45, 7)$
$> is.character(x)$ **test if an object is of type character**
$[1]FALSE$

$> is.character(as.character(x))$
$[1]TRUE$

$> Y = paste(\text{“}HOME\text{”}, \text{“}WHILE\text{”}, \text{“}DOG\text{”}, sep = \text{“ : ”})$
$> strsplit(Y, split = \text{“}O\text{”})$ **split the elements of Y into sub-strings w.r.t split string**
$[[1]]$
$[1]\text{“}H\text{”}\text{“}ME : WHILE : D\text{”}\text{“}G\text{”}$

$> substr(Y, 5, 10)$ **Extract or replace sub-strings in a character vector.**
$[1]\text{” : }WHILE\text{”}$

Logical vector in R

- A logical vector is a vector whose elements are TRUE, FALSE or NA.
- it is generated by conditions:

  $> x$

  [1]1.4 6 23.1 65.43 2.7 55

  $> logic = x > 34$

  [1]***FALSE FALSE FALSE TRUE FALSE TRUE***

  It compares each element of x with 34. It returns a vector the same length as x, with a value TRUE when the condition is met and FALSE when it is not.

- logical operators are >, >=, <, <=, ==, !=, &, |.

Factor in R

- A factor is a special type of vector used to a vector of data, usually taking a small number of distinct values. To store in statistical modeling data as factors insures that will be treated not as continuous variables but as categorical variable.
  - it is internally stored as a vector of integer values with a corresponding set of character values to use when the factor is displayed (an efficient way);
  - Factor's levels is always a character values;
- a factor is created as follows:

  $> f = factor(rep(c(\text{"}Control\text{"}, \text{"}Treated\text{"}), c(3,4)))$
  $[1] Control\ Control\ Control\ Treated\ Treated\ Treated\ Treated$
  Levels: Control Treated

- main factor operators:

  $> levels(f)$ **it returns the levels of a factor**

  $> summary(f)$ **it returns the frequencies associated with each level**

  $> str(f)$ **it returns a compact visualization of the factor**

Exercises on Vectors

1. Create a vector x with the following entries:
   $$3\ 4\ 1\ 1\ 2\ 1\ 6$$
   Check which elements of x are lower or equal to 2.
   Modify x so that all of the 1 values are changed to 0 values.
2. Create a vector y containing the elements of x that are greater than 2;
3. Create a sequence of numbers from 1 to 20 in steps of 0.25 and store in k.
   Change the elements in positions 4 and 5 in 11 and 12;
4. Concatenate x and y into a vector called Vec;
5. Display all objects in the workspace and then remove Vec.

Exercises on Vectors

- Create a vector x with the following entries:

  3 4 1 1 2 1 6

  Check which elements of x are lower or equal to 2.
  Modify x so that all of the 1 values are changed to 0 values.

  $> x = c(3, 4, 1, 1, 2, 1, 6)$

  $> x <= 2$

  $> x[x == 1] = 0$

Exercises on Vectors

- Create a vector y containing the elements of x that are greater than 2;

  $> y = x[x > 2]$

  $> y$

  [1]3 4 6

Exercises on Vectors

- Create a sequence of numbers from 1 to 20 in steps of 0.25 and store in k. Change the elements in positions 4 and 5 in 11 and 12

  $> k = seq(1, 20, by = 0.25)$

  $> k[c(4, 5)] = c(11, 12)$

Exercises on Vectors

- Concatenate x and y into a vector called Vec:

  $> Vec = c(x, y)$

  $> Vec$

  [1]3 4 1 1 2 1 6 3 4 6

Exercises on Vectors

- Display all objects in the workspace and then remove Vec.

  $> ls()$

  $[1]$"Vec" "x" "y" "z"

  $> rm(Vec)$

  $> rm(list = ls())$ **To remove all variables**

List in R

List in R

- it is an ordered collection of components;
- its components may be arbitrary R objects (vectors, data frame lists, . . .);
- function list() can be used to create lists:

```
> x = c(1 : 4)
> y = rep("ACT", 2)
> k = c(TRUE, TRUE)
> l1 = list(x, y, k)    it creates a list contains three vectors (i.e. x,y,k)
> l1
[[1]]
[1]1 2 3 4

[[2]]
[1]"ACT" "ACT"

[[3]]
[1]TRUE TRUE
```

List in R

- Two lists can be concatenated as follows:

  $> l2 = list(c("ACT"), 1:10)$
  $> l3 = c(l1, l2)$

- names can be associated with list elements:

  $> names(l1) = c(\text{“}first\text{”}, \text{“}second\text{”}, \text{“}third\text{”})$
  first
  [1]1 2 3 4

  second
  [1]“*ACT*” “*ACT*”

  third
  [1]*TRUE TRUE*

List in R

- We can access the list elements in the following two ways:
  1. how to access the element in first position in the list *l*1 returning a vector

     > *l*1[[1]]
     [1]1 2 3 4

     > *l*1$*first*
     [1]1 2 3 4

  2. how to access the first element in the vector in first position in the list *l*1

     > *l*1[[1]][1]
     [1]1

  3. how to return a new list containing the fist vector in the list *l*1

     > *l*1[1]
     [[1]]
     [1]1 2 3 4

Exercises on Lists

- Create the following three vector:
  1. X = {1, 5, 6, 19, 5};
  2. Y = {"HOME", "WOLF", "ROOM", NA}
  3. Z = {1.25, 1.50, 1.75, ...10}

  and stores them in the list $L1$.
- Give a name to each list element (using names function).
- Use the two different ways to access the 2nd element of the list $L1$.
- Access the 2nd element of the 3rd element of the list $L1$.
- Access the 2nd and 4th elements of the 1st element of the list $L1$.

Exercises on Lists

- Create the following three vectors:
  1. X = {1, 5, 6, 19, 5};
  2. Y = {"HOME", "WOLF", "ROOM", NA}
  3. Z = {1.25, 1.50, 1.75, ...10}

  and stores them in the list $L1$.

$> X = c(1, 5, 6, 19, 5)$
$> Y = c("HOME", "WOLF", "ROOM", NA)$
$> Z = seq(1, 10, by = 0.25)$
$> L1 = list(X, Y, Z)$

Exercises on Lists

- Give a name to each list element (using names function).

$> names(L1) = c("X", "Y", "Z")$
$> L1$
X
[1]1 5 6 19 5
Y
[1]"$HOME$" "$WOLF$" "$ROOM$" NA
...
...
...

Exercises on Lists

- Use the two different ways to access the 2nd element of the list $L1$.

$> L1[[2]]$
$[1]"HOME" \ "WOLF" \ "ROOM" \ NA$
$> L1[2]$
$\$Y$
$[1]"HOME" \ "WOLF" \ "ROOM" \ NA$

Exercises on Lists

- Access the 2nd element of the 3rd element of the list $L1$.

  $> L1[[3]][2]$
  $[1]1.25$

Exercises on Lists

- Access the 2nd and 4th elementes of the 1st element of the list $L1$.

  $> L1[[1]][c(2,4)]$
  [1]5 19

Data Frame in R

Data Frame in R

- It is used to storage data table in R;
- It can be considered as a matrix in which columns can contain different types;
- We can create data frames from pre-existing variables:

$$
\begin{aligned}
&> name = c("GENE1", "GENE2", "GENE3") \\
&> seq = c("ATCCT..", "CCTTT..", "CCAACT..") \\
&> count = c(100, 20, 4) \\
&> d = data.frame(name, seq, count) \\
&> d
\end{aligned}
$$

	name	seq	count
1	GENE1	ATCCT..	100
2	GENE2	CCTTT..	20
3	GENE3	CCACT..	4

Data Frame in R

Main operations:

- attributes(d) returns the data frame attributes:

  $> attributes(d)$
  $\$names$
  $[1]"name" \ "seq" \ "count"$
  $\$row.names$
  $[1]1 \ 2 \ 3$
  $\$class \ [1]"data.frame"$

- colnames(d) returns the names of data frame columns:

  $> colnames(d)$
  $[1]"name" \ "seq" \ "count"$

  $> colnames(d) = c(\text{“}c1\text{”}, \text{“}c2\text{”}, \text{“}c3\text{”}, \text{“}c4\text{”})$ **change column names.**

- rownames(d) returns the names of data frame rows:

  $> rownames(d)$
  $[1]1 \ 2 \ 3$

Indexing Data Frame in R

- it is possible to use the same method of matrices to access values of a data frame.

$> d$

	name	*seq*	*count*
1	*GENE*1	*ATCCT..*	100
2	*GENE*2	*CCTTT..*	20
3	*GENE*3	*CCACT..*	4

$> d[2,2]$ **gives the value in the 2nd row and 2nd column of d.**
[1]*CCTTT..*

$> d[2,]$ **gives the values in the 2nd row of d.**
[1]*GENE*2 *CCTTT..* 20

$> d[,3]$ **gives the values in the 3rd column of d.**
[1]100 20 4

Indexing Data Frame in R

- it is possible to use column name to access columns of a data frame.

$> d$

	name	*seq*	*count*
1	*GENE*1	*ATCCT..*	100
2	*GENE*2	*CCTTT..*	20
3	*GENE*3	*CCACT..*	4

$> d\$count$ **gives the values in the 3rd column of d.**
[1]100 20 4

- Selecting all data for cases that satisfy some criterion.
  $> d[d\$count \geq 20,]$

	name	*seq*	*count*
1	*GENE*1	*ATCCT..*	100
2	*GENE*2	*CCTTT..*	20

Data Frame in R

Main operations(2):

- summary(d) returns a summary of data frame:

$> summary(d)$

name	seq		count
GENE1 : 1	ATCCT.. : 1	Min. :	4.000
GENE2 : 1	CCTTT.. : 1	1stQu. :	12.00
GENE3 : 1	CCACT.. : 1	Median :	20.00
		Mean :	41.33
		3rdQu. :	60.00
		Max. :	100.00

- subset(d,cond) returns a subset of rows according to condition:

$> subset(d, d[, 3] > 10)$

	name	seq	count
1	GENE1	ATCCT..	100
2	GENE2	CCTTT..	20

Main operations(3):

- which(condition) gives the TRUE indices of a logical object.
  Then, it answers to the question "Which indices are TRUE?"

  $> which(d[,3] > 10)$
  [1] 1 2

  $> which(d[,3] == 20)$
  [1] 2

  $> which(d[,3]\%in\%1:20)$ **operator $\%in\%$ tests which elements of d are in 1:20.**
  [1] 2 3

  $> which(d[,1]\%in\%c("GENE1","GENE3"))$
  [1] 1 3

Exercises on Data Frames

- Create a data frame called *D* with the following data:

Firstname	Lastname	Age	Gender	Points
Alice	Ryan	37	F	278
Paul	Collins	34	M	242
Jerry	Burke	26	M	312
Thomas	Dolan	72	M	740
Marguerite	Black	18	F	177
Linda	McGrath	24	F	195

- Store the points for every person into a vector called *pts*, then calculate the average number of points received.

- Store the data for the females only into a data frame called *fpoints*, then calculate the summary.

Exercises on Data Frames

- Create a data frame called D with the following data:

Firstname	Lastname	Age	Gender	Points
Alice	Ryan	37	F	278
Paul	Collins	34	M	242
Jerry	Burke	26	M	312
Thomas	Dolan	72	M	740
Marguerite	Black	18	F	177
Linda	McGrath	24	F	195

$> Firstname = c(\text{“}Alice\text{”}, \text{“}Paul\text{”}, \text{“}Jerry\text{”}, \text{“}Thomas\text{”}, \text{“}Marguerite\text{”}, \text{“}Linda\text{”})$
$> Lastname = c(\text{“}Ryan\text{”}, \text{“}Collins\text{”}, \text{“}Burke\text{”}, \text{“}Dolan\text{”}, \text{“}Black\text{”}, \text{“}McGrath\text{”})$
$> Age = c(37, 34, 26, 72, 18, 24)$
$> Gender = c(\text{“}F\text{”}, \text{“}M\text{”}, \text{“}M\text{”}, \text{“}M\text{”}, \text{“}F\text{”}, \text{“}F\text{”})$
$> Points = c(278, 242, 312, 740, 177, 195)$
$> D = data.frame(Firstname, Lastname, Age, Gender, Points)$ **vector names are used as column names.**

Exercises on Data Frames

- Store the points for every person into a vector called *pts*, then calculate the average number of points received.

$> pts = D\$Points$
$> pts$
[1]278 242 312 740 177 195
$> mean(pts)$
[1]324

Exercises on Data Frames

- Store the data for the females only into a data frame called *fpoints*, then calculate the summary.

  $> fpoints = subset(D, D\$Gender == \text{“}F\text{”})$
  $summary(fpoints)$

Exercises on Data Frames

- The age for Paul Collins was entered incorrectly. Change his age to 48.
- Determine the maximum age of the males.
- Extract the data for people with more than 100 points and are over the age of 30.

Exercises on Data Frames

- The age for Paul Collins was entered incorrectly. Change his age to 48.

  $> D[2,3] = 48$

Exercises on Data Frames

- Determine the maximum age of the males.

$> max(subset(D, D\$Gender == \text{“}M\text{”})\$Age)$
[1]72

Exercises on Data Frames

- Extract the data for people with more than 100 points and are over the age of 30.

  $> subset(D, D\$Age > 30 \& D\$Points > 100)$

I/O in R language

Input from a file in R

- R provides a set of functions to read data from files:
  - read.table() is used to read data frames from formatted text files.
    A variable separator can be specified.
  - read.csv() is used to read data frames from comma separated variable files.
  - read.csv2() is used to read data frames from semicolon separated variable files.
  - load() is used to reload datasets written with the function save().
    Data are stored in binary format (more compact!!).

Input from a file in R

- read.table() reads a file in table format and creates a data frame from it,

read.table(file,header=FALSE, sep= " ", dec=".", stringAsFactors=TRUE ...)

file : the name of the file in which the data are stored;
header : a logical value indicating whether the file contains the names of the variables as its first line;
sep : the field separator character;
dec : the character used for decimal points;
stringAsFactors : logical: should character vectors be converted to factors?;
row.names : it can be a vector giving the actual row names, or a single number giving the column of the table which contains the row name;
... : optional arguments;

$> d = read.table(\text{“./example.txt”}, header = TRUE, sep = \text{“!”})$

$> b = read.table(\text{“./example1.txt”}, header = FALSE, sep = \text{“ ”})$

Input from a file in R

- read.csv() reads a file in table format and creates a data frame from it,

read.csv(file,header=FALSE, sep=",", dec=".",...)

file	: the name of the file in which the data are stored;
header	: a logical value indicating whether the file contains the names of the variables as its first line;
sep	: the field separator character;
dec	: the character used for decimal points;
stringAsFactors	: logical: should character vectors be converted to factors?;
row.names	: it can be a vector giving the actual row names, or a single number giving the column of the table which contains the row name
...	: optional arguments;

$> d = read.csv(\text{“./example.txt”}, header = TRUE)$

$> b = read.csv(\text{“./example1.txt”}, header = FALSE)$

Input from a file in R

- read.csv2() reads a file in table format and creates a data frame from it,

read.csv2(file,header=FALSE, sep=";", dec=".", ...)

file	: the name of the file in which the data are stored;
header	: a logical value indicating whether the file contains the names of the variables as its first line;
sep	: the field separator character;
dec	: the character used for decimal points;
stringAsFactors	: logical: should character vectors be converted to factors?;
row.names	: it can be a vector giving the actual row names, or a single number giving the column of the table which contains the row name
...	: optional arguments;

$> d = read.csv2(\text{“./example.txt”}, header = T)$

$> b = read.csv2(\text{“./example1.txt”}, header = F)$

Input from a file in R

- load() reload datasets written with the function save().

load(file, ...)

File : the name of the file in which the data are stored;
verbose = FALSE : if TRUE item names are printed;
... : optional arguments;

$> load(\text{“./example.data”})$

$> load(\text{“./example.data”}, verbose = T)$
Loading objects :
m

Writing a file in R

- R provides a set of functions to write data into files:
  - write.table() is used to write data frames into formatted text files.
    A variable separator can be specified.
  - write.csv() is used to write data frames into comma separated variable files.
  - write.csv2() is used to write data frames into semicolon separated variable files.
  - save() is used to save datasets into a binary file.
    Data are stored in binary format (more compact!!).

Writing a file in R

- write.table() is used to write data frames into formatted text files ,

write.table(x,file,col.names=TRUE,row.names=TRUE, sep=" ", dec=".", ...)

x	:	the object to be written;
file	:	the name of the file in which the data are stored;
col.names	:	if TRUE column names are stored;
row.names	:	if TRUE row names are stored;;
sep	:	the field separator character;
dec	:	the character used for decimal points;
...	:	optional arguments;

$> write.table(b, \text{“./example.txt”}, col.names = TRUE, row.names = TRUE, sep = \text{“!”})$

$> write.table(b, \text{“./example.txt”}, col.names = FALSE, row.names = FALSE, sep = \text{“,”})$

Writing a file in R

- write.csv() is used to write data frames into formatted text files ,

write.csv(x,file,col.names=TRUE,row.names=TRUE, sep=",", dec=".", ...)

x	:	the object to be written;
file	:	the name of the file in which the data are stored;
col.names	:	if TRUE column names are stored;
row.names	:	if TRUE row names are stored;;
sep	:	the field separator character;
dec	:	the character used for decimal points;
...	:	optional arguments;

$> write.csv(b, \text{“./example.txt”}, col.names = TRUE, row.names = TRUE)$

$> write.csv(b, \text{“./example.txt”}, col.names = FALSE, row.names = FALSE)$

Writing a file in R

- write.csv2() is used to write data frames into formatted text files ,

write.csv2(x,file,col.names=TRUE,row.names=TRUE, sep=";",dec=",", ...)

x	:	the object to be written;
file	:	the name of the file in which the data are stored;
col.names	:	if TRUE column names are stored;
row.names	:	if TRUE row names are stored;;
sep	:	the field separator character;
dec	:	the character used for decimal points;
...	:	optional arguments;

$> write.csv2(b, \text{“./example.txt”}, col.names = TRUE, row.names = TRUE)$

$> write.csv2(b, \text{“./example.txt”}, col.names = FALSE, row.names = FALSE)$

Writing a file in R

- save() writes an external representation of R objects to the specified file,

  save(...,file, ...)

  ... : a list of objects to be saved;
  file : the name of the file in which the data are stored;
  ... : optional arguments;

  $> save(b, c, file = \text{“./example.data”})$

Download and install a package in R

- In R, a package can be downloaded and installed from CRAN-like repositories or from local files;

install.packages(pkgs,rep=getOption("repos"))

pkgs : character vector of the names of packages to be downloaded;
rep : base URL(s) of the repositories to use.
Default CRAN repository.
... : optional arguments;

$> install.packages(\text{“}KDE\text{”})$

$> install.packages(path\_to\_file, repos = NULL, type = \text{"}source\text{"})$

Load a package in R

- In R a package must be loaded before being used;

library(package,....)

package : name of the package to be loaded;
... : optional arguments;

> *library*(*MASS*)

> *library*() **see all packages installed**

Download and install Bioconductor

- To install core packages, type the following in an R command window: ;

```
source("https://bioconductor.org/biocLite.R")
  try http if https does not work
biocLite()
```

- Install specific packages, e.g., *GenomicFeatures* and *AnnotationDbi*, with:

```
biocLite(c("GenomicFeatures", "AnnotationDbi"))
```

Save and Load the R workspace

- In R the workspace can be saved and loaded using:

```
save.image(file =".RData")
load(file =".RData")
```

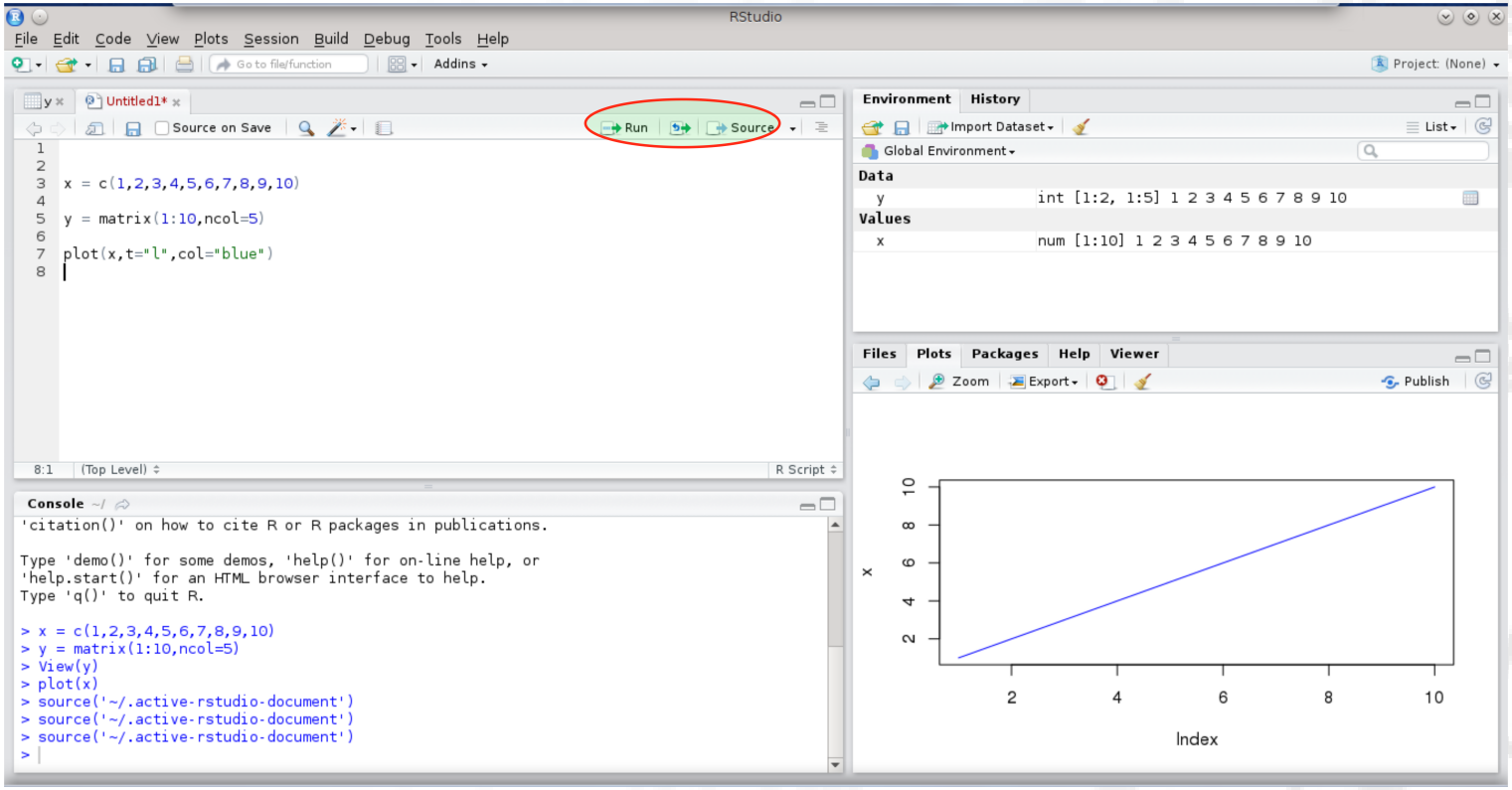
$> save.image(file = "OutputWorkspace")$

$> load(file = "OutputWorkspace")$

How to import R script

- An R-script is simply a text file containing commands;
- It must be in the Working Directory;
- It can be loaded in R using source("scriptFile")

Using RStudio (new window):



Exercises on input/ouput

- Save in the textual file "example.txt" the data frame trees;
- Load the data frame stored in the textual file "example.txt";
- Save in the textual file "example.csv" the data frame trees using ";" as variable separator;
- Load the data frame stored in the textual file "example.csv";
- Create a matrix with 1,000,000 elements and save it using "write.table" and "save".

Exercises on input/ouput

- Save in the textual file "example.txt" the data frame trees;

  $> write.table(trees, file = \text{“./example.txt”})$

- Load the data frame stored in the textual file "example.txt";

  $> D = read.table(file = \text{“./example.txt”})$

- Save in the textual file "example.csv" the data frame trees using ";" as variable separator;

  $> write.table(trees, file = \text{“./example.csv”}, sep = \text{”;”})$

- Load the data frame stored in the textual file "example.csv";

  $> K = read.table(file = \text{“./example.csv”}, sep = \text{”;”})$

Exercises on input/ouput

- Create a matrix with 1,000,000 elements and save it using "write.table" and "save".

$> m = matrix(1 : 1000000, ncol = 100000)$

$> write.table(m, file = \text{“./example.csv”})$

$> save(m, file = \text{“./example.csv”})$

Apply family in R

Apply family in R

- How to efficiently apply a function to each element of array, data frame and list.

  For instance: to apply a function to the rows/columns of a matrix

- Functions apply, lapply, sapply, tapply can be used:

  apply : only used for arrays/matrices;

  lapply : takes any data structure and gives a list of results;

  sapply : like lapply, but it tries to simplify the result to a vector or matrix if possible;

  tapply :allows us to apply a function on a subset of values grouped according to one or more factors.

Function apply()

- the apply function returns a vector or array of values obtained by applying a function to margins of an array or matrix.

apply(X, MARGIN, FUN, ...)

X	:	array;
MARGIN	:	1 for rows, 2 for columns;
FUN	:	one function to be applied;
...	:	optional arguments toFUN;

$> m$

	[, 1]	[, 2]
[1,]	−0.1767643	−0.1950407
[2,]	1.5306045	0.3307676
[3,]	−0.3806768	0.8992097

$> apply(m, 1, sum)$ **by rows**

[1] − 0.3718050 1.8613721 0.5185329

$> apply(m, 2, sum)$ **by columns**

[1]0.9731634 1.0349366

Function apply()

- the apply function returns a vector or array of values obtained by applying a function to margins of an array or matrix.

apply(X, MARGIN, FUN, ...)

X	:	array;
MARGIN	:	1 for rows, 2 for columns;
FUN	:	one function to be applied;
...	:	optional arguments toFUN;

$> m$

	[, 1]	[, 2]
[1,]	−0.1767643	−0.1950407
[2,]	1.5306045	0.3307676
[3,]	−0.3806768	0.8992097

$> apply(m, 1, max)$ **by rows**

$[1] - 0.1767643\ 1.5306045\ 0.8992097$

$> apply(m, 2, min)$ **by columns**

$[1] - 0.3806768\ \ - 0.1950407$

Function lapply()

- the lapply function returns a list where each element is the result of applying a function to the corresponding element of input data structure.

lapply(X, FUN, ...)

X : any data that can be compatible with a list;
FUN : one function to be applied;
... : optional arguments toFUN;

$> data()$ **to list in-built data set**
$> lapply(trees, mean)$ **trees is a in-built data set**
$Girth
[1]13.24839
$Height
[1]76
$Volume
[1]30.17097

```
> trees
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
5   10.7     81   18.8
6   10.8     83   19.7
7   11.0     66   15.6
8   11.0     75   18.2
9   11.1     80   22.6
```

Function sapply()

- the sapply function is a user-friendly version and wrapper of "lapply" by default returning a vector, matrix .

sapply(X, FUN, ...)

X : any data that can be compatible with a list/vector/matrix;
FUN : one function to be applied;
... : optional arguments toFUN;

$> data()$ **to list in-built data set**
$> sapply(trees, mean)$ **trees is a in-built data set**

Girth	***Height***	***Volume***
13.24839	76.00000	30.17097

```
> trees
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
5   10.7     81   18.8
6   10.8     83   19.7
7   11.0     66   15.6
8   11.0     75   18.2
9   11.1     80   22.6
```

Function tapply()

- the tapply function allows us to apply a function on a subset of values grouped according to one or more factors .

tapply(X, INDEX, FUN, ...)

X : any data that can be compatible with a list;
INDEX : list of one or more factors used to cluster X;
FUN : one function to be applied;
... : optional arguments toFUN;

> *library*(*MASS*) **to load MASS data set**

> *Cars93* **Car93 is a MASS data set**

	Manufacturer	Model	Type	Min.Price	Price	Max.Price	MPG.city
1	Acura	Integra	Small	12.9	15.9	18.8	25
2	Acura	Legend	Midsize	29.2	33.9	38.7	18
3	Audi	90	Compact	25.9	29.1	32.3	20
4	Audi	100	Midsize	30.8	37.7	44.6	19
5	BMW	535i	Midsize	23.7	30.0	36.2	22
6	Buick	Century	Midsize	14.2	15.7	17.3	22
7	Buick	LeSabre	Large	19.9	20.8	21.7	19
8	Buick	Roadmaster	Large	22.6	23.7	24.9	16
9	Buick	Riviera	Midsize	26.3	26.3	26.3	19

> *tapply*(*Cars93*$*Price*, *Cars93*$*Manufacturer*, *mean*) **Compute the average price for each brand**

Exercises on apply

- Compute sums of the columns of the hills data set;
- Compute row and column sums of a matrix 10x10 whose values are generated according to uniform distribution between 4 and 10;
- Use apply to calculate the standard deviation of the columns of a matrix;
- Create a list of vectors of varying length (using sample() function);
- Consider in-built data set "airquality" compute the average wind speed and ozone percentage with respect to "month" column.

Exercises on apply

- Compute sums of the columns of the hills data set;

  $> lapply(hills, sum)$

  $> sapply(hills, sum)$

Exercises on apply

- Compute row and column sums of a matrix 10x10 whose values are generated according to uniform distribution between 4 and 10

$> m = matrix(runif(100, min = 4, max = 10), ncol = 10)$
$> apply(m, 1, sum)$
$> apply(m, 2, sum)$

Exercises on apply

- Use apply to calculate the standard deviation of the columns of a matrix.

$> m = matrix(runif(100, min = 4, max = 10), ncol = 10)$
$> apply(m, 2, sd)$

Exercises on apply

- Create a list of vectors of varying length (using sample() function)

$> veclen = sample(11 : 40)$
$> mylist = lapply(veclen, runif)$

Exercises on apply

- Consider in-built data set "airquality" compute the average wind speed and ozone percentage with respect to "month" column.

  $> tapply(airquality\$Wind, airquality\$Month, mean)$

  $> tapply(airquality\$Ozone, airquality\$Month, mean, na.rm = TRUE)$

  na.rm=TRUE removes NA from mean computation

Function in R

- We have already used several examples of functions:

  mean(x) sd(x) plot(x, y, ...) lm(y $\sim$ x, ...) ....

- Functions are typically written if we need to compute the same thing for several data sets;

- Functions have a **name** and a **list of arguments** or **input objects**. For example, the argument to the function mean() is the vector x;

- Functions can also have a list of **output objects** returned when the function is terminated;

- A function must be written and loaded into R before it can be used.

A simple function in R

- A simple function can be constructed as follows:

```
function_name=function(arg1,arg2,...){
command1
command2
output
}
```

- You can define a function name;
- The function keyword specified that you are writing a function;
- Inside () you can outline the input objects;
- The commands occur inside {};
- The name of whatever output you want goes at the **end of the function**;
- Comments lines are denoted by #.

A simple function in R

- An example:

```
mysum=function(x,y){
  x + y
}
```

- This function is called mysum;
- It has two input arguments, called x,y.
- Whatever values are passed for x and y their sum will be computed and the result visualizes on the screen.
- The function must be loaded into R before being called.

A simple function in R

How to execute a new function:

- Write the function in a text editor;
- Copy the function in the R console.
  Type ls() into the console: the function now appears;
- Call the function using:

  $> mysum(3, 4)$
  [1]7

  $> mysum(y = 3, x = 4)$
  [1]7

  $> mysum(y = c(3, 6), x = c(4, 4))$
  [1]7 6

- Store the result into a variable sumXY:

  $> sumxy = mysum(3, 4)$

How to load a function from a file

- Command source() is used to read the file and execute/load the commands in the same sequence given in the file.

source(file,echo ...)

file : character string giving the pathname of the file;

echo : if TRUE, each expression is printed after parsing, before evaluation.

How to load a function from a file

- Command source() is used to read the file and execute/load the commands in the same sequence given in the file.

- Use a text editor to save the following function in the file "myfun1.r":

```
myfun=function(x,y,p){
  k = (x + y) * p
  return(k)
}
```

- Use command source() to load the function from the file:

```
> source("myfun1.r")
```

A simple function in R

- An example:

```
myfun=function(x,y,p){
  k = (x + y) * p
  return(k)
}
```

- Function myfun has 3 arguments;
- The command return specifies what the function returns, here the value of k;

$> myfun(3, 4, 7)$

$> res = myfun(3, 4, 7)$ **result is stored in res**

A more complex function in R

- The following function returns several values in the form of a list:

```
myfun1=function(x){
  the.mean = mean(x)
  the.sd = sd(x)
  the.min = min(x)
  the.max = max(x)
  return (list(mean = the.mean, stand.dev = the.sd,
  minimum = the.min, maximum = the.max))
}
```

A more complex function in R

- how to call myfun1:

$> x = rnorm(10)$

$> res = myfun1(x)$

$> res$

res

$\$mean$

[1]0.29713

$\$stand.dev$

[1]1.019685

$\$minimum$

$[1] - 1.725289$

$\$maximum$

[1]2.373015

Argument Matching in R

How does R know to match arguments?

Argument matching is done in a few different ways:

- The arguments are matched by their positions. The first supplied argument is matched to the first formal argument and so on.

  $> myfun(3, 4, 7)$ **x=3, y=4 and p=7**

- The arguments are matched by name. A named argument is matched to the formal argument with the same name:

  $> myfun(y = 4, x = 3, p = 7)$ **x=3, y=4 and p=7**

- Name matching happens first, then positional matching is used for any unmatched arguments.

Argument Matching in R

- Default values for some/all arguments can be specified:

```
myfun=function(x,y,p=10){
  k = (x + y) * p
  return(k)
}
```

- If a value for the argument p is not specified in the function call, a value of 10 is used.

```
> l = myfun(3, 4)
> l
[1]70
```

- If a value for p is specified, that value is used.

```
> l = myfun(3, 4, 2)
> l
[1]14
```

Exercises on functions

1. Write a function that when passed a number, returns the number squared, the number cubed, and the square root of the number;
2. Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the cat() function in R.) and creates a histogram of the data in a file;
3. Write a function that compares its two input vectors using a Q-Q plot. Moreover each vector must be compared with normal distribution re-using a Q-Q plot. Generate the two vectors according to a gamma distribution.

Exercises on function

- Write a function that when passed a number, returns the number squared, the number cubed, and the square root of the number;

```
myfun2=function(x){
  squared = x * x
  cubed = x * x * x
  root = sqrt(x)
  return (list(squared, cube, root))
}
```

Exercises on function

- Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the cat() function in R.) and creates a histogram of the data in a file;

```
myfun3=function(x,file="hist.png"){
  cat(x,": standard deviation is", sd(x),"\n")
  cat(x,": mean is", mean(x),"\n")
  png(file)
  hist(x, col ="blue", main ="Histogram of ; x")
  dev.off()
}
```

Exercises on function

- Write a function that compare its two input vectors using a Q-Q plot. Moreover each vector must be compared with normal distribution re-using a Q-Q plot. Generate the two vectors according to a gamma distribution.

```
myfun4=function(x,y){
  png("qqplot.png")
  par(mfrow = c(1,3))
  qqplot(x, y, main ="Q - Qplot of x VS y")
  qqnorm(x, main ="Q - Qplot of x VS normal")
  qqline(x, col ="red")
  qqnorm(y, main ="Q - Qplot of y VS normal")
  qqline(y, col ="red")
  dev.off()
}
```

```
> x = rgamma(100, shape = 1.5, rate = 3)
> y = rgamma(100, shape = 1.5, rate = 6)
> myfun4(x, y)
```

if Statement

- **Conditional execution:** the if statement has the form:

  if (condition){
  $expr_1$
  }
  else {
  $expr_2$
  }

  Condition is evaluated and returns a logical value (i.e. TRUE or FALSE.)
  If the condition is evaluated **TRUE**, $expr_1$ is executed , otherwise $expr_2$ is executed.
- Logical operators $\&\&, ||, ==, !=, >, <, >=, <=$ are used as the conditions in the if statement.

if Statement: a simple example

- The following function gives a demonstration of the use of if ... else:

```
checkMyfunction=function(number){
 if(number! = 1) {
  cat(number,"is not one \n")
 }
 else {
  cat(number,"is one \n")
 }
}
```

```
> checkMyfunction(1)
1 is one
> checkMyfunction(2)
2 is not one
```

if Statement: a second simple example

- The following function gives a demonstration of the use of && :

```
checkBetween=function(number){
 if((number >= 1)&&(number <= 10)) {
  cat(number,"is between one and ten \n")
 }
 else {
  cat(number,"isn't between one and ten \n")
 }
}
```

```
> checkBetween(2)
1 is between one and ten
> checkMyfunction(12)
12 isn't between one and ten
```

Nested if Statements

- The following function gives a demonstration of the use of if ... else if ... else:

```
checkNum=function(number){
 if(number == 0) {
  cat(number,"is zero \n")
 }
 else if(number < 0) {
  cat(number,"is negative \n")
 }
 else{
  cat(number,"is positive \n")
 }
}
```

For loop

- To loop/iterate through a certain number of repetitions a for loop is used.
  Its syntax is:

```
for (condition){
  command_1
  command_2
  .....
}
```

A simple example of a for loop:

```
MyLoop=function(x){
  cumsum = rep(0, length(x))
  if(!(is.numeric(x))) {
    cat(x,"must be numeric \n")
    return(cumsum)
  }
  cumsum[1] = x[1]
  for(i in 2 : length(x))
    cumsum[i] = cumsum[i-1] + x[i]
  return(cumsum)
}
```

For loop

- You can nest loops. In this cases indenting the code can be useful.

```
for (condition_1){
  command_1
  command_2
  for(condition_2){
    command_1
    command_2
  }
}
```

- for loops and multiply nested for loops are generally avoided when possible in R because they can be quite slow.

For loop

- Compare using function system.time() the function MyLoop()

```
MyLoop=function(x){
  cumsum = rep(0, length(x))
  if(!(is.numeric(x))) {
    cat(x,"must be numeric \n")
    return(cumsum)
  }
  cumsum[1] = x[1]
  for(i in 2 : length(x))
    cumsum[i] = cumsum[i-1] + x[i]
  return(cumsum)
}
```

and cumsum(). They have a different execution time.

```
> x = rnorm(1000000)
> system.time(cusum(x))
> system.time(MyLoop(x))
```

For loop

- Execution time of code portion can be measured using functions Sys.time() and difftime()

```
MyLoop=function(x){
  cumsum = rep(0, length(x))
  if(!(is.numeric(x))) {
    cat(x,"must be numeric \n")
    return(cumsum)
  }
  cumsum[1] = x[1]
  time1 = Sys.time()          # before loop
  for(i in 2 : length(x))
    cumsum[i] = cumsum[i-1] + x[i]
  time2 = Sys.time()          # after loop
  cat("Loop time : ", difftime(time2, time1, unit = "secs"), "sec.\n")
  return(cumsum)
}
```

While loop

- While loop can be used if the number of iterations required is not known beforehand;
- For example, if loop must continue until a certain condition is met.
- Its syntax is:

```
while (condition){
  command_1
  command_2
  .....
}
```

The loop continues while condition == TRUE.

While loop

- A simple example of a while loop:

```
MyLoop1=function(x){
  cumsum = rep(0, length(x))
  if(!(is.numeric(x))) {
    cat(x,"must be numeric \n")
    return(cumsum)
  }
  cumsum[1] = x[1]
  i = 2
  while(i <= length(x)){
    cumsum[i] = cumsum[i-1] + x[i]
    i = i + 1
  }
  return(cumsum)
}
```

next, break, statements

- The next statement can be used to discontinue one particular iteration of any loop. Useful if you want a loop to continue even if an error is found (error checking);
- The break statement completely terminates a loop. Useful if you want a loop to end if an error is found.

```
MyLogNext=function(x){
 for(i in 1 : length(x)){
  if(x[i] <= 0) {
   next
  }
  x[i] = log(x[i])
 }
 return(x)
}
```

```
MyLogNext1=function(x){
 for(i in 1 : length(x)){
  if(x[i] <= 0) {
   break
  }
  x[i] = log(x[i])
 }
 return(x)
}
```

next, break, statements

- The next statement can be used to discontinue one particular iteration of any loop. Useful if you want a loop to continue even if an error is found (error checking);
- The break statement completely terminates a loop. Useful if you want a loop to end if an error is found.

```
MyLogNext=function(x){
 for(i in seq_along(x)){
  if(x[i] <= 0) {
   next
  }
  x[i] = log(x[i])
 }
 return(x)
}
```

```
MyLogNext1=function(x){
 for(i in seq_along(x)){
  if(x[i] <= 0) {
   break
  }
  x[i] = log(x[i])
 }
 return(x)
}
```

Exercises on loops and functions

- Create a function find_value(), which takes as input a number b and a vector m, and returns first occurrence of b in m;
- Create a function find_all_value(), which takes as input a number b and a matrix m, and returns all the occurrences of b in m;
- Create a function translate(), which takes as input a numeric vector c and returns a string vector f such that $f[i] = "P"$ iff $c > 0$ otherwise $f[i] = "N"$.

Exercises on loops and functions

- Create a function find_value(), which takes as input a number b and a vector m, and returns first occurrence of b in m;

```
find_value=function(b,m){
 if(length(m) < 2) {
  cat("m size must be greater 1 \n")
  return(-1)
 }
 ind = 1
 while(ind <= length(m)){
  if(m[ind] == b)
    return(ind)
  ind = ind + 1
 }
 return(-1)
}
```

Exercises on loops and functions

- Create a function find_all_value(), which takes as input a number b and a matrix m, and returns all the occurrences of b in m;

```
find_all_value=function(b,m){
  f = NULL
  for(row in 1 : dim(m)[1]){
    for(col in 1 : dim(m)[2]){
      if(m[row, col] == b)
      if(length(f) == 0)
        f = list(c(row, col))
      else
        f = list(f, c(row, col))
    }
  }
  return(f)
}
```

Exercises on loops and functions

- Create a function translate(), which takes as input a numeric vector c and returns a string vector f such that $f[i] = "P"$ iff $c > 0$ otherwise $f[i] = "N"$.

```
translate=function(m){
  f = NULL
  if(!(is.numeric(x))) {
    cat(x,"must be numeric \n")
    return(f)
  }
  for(ind in 1 : length(m)){
    if(m[ind] > 0)
      f = c(f,"P")
    else
      f = c(f,"N")
  }
  return(f)
}
```

Plotting in R

Plotting in R

- R language provides a powerful graphical environment (2D and 3D plots);
- In R it is easy to generate high quality plots;
- It can generate plots in many different formats(devices):
  - directly on the screen output;
  - postscript format;
  - pdf (Adobe Portable Document Format);
  - jpeg (JPEG bitmap format);
  - png (PNG bitmap format);
  - wmf (Windows Metafile).

Plotting in R

R graphical functions can be classified as follows:

- High level graphical functions:
  - they draw a plot on a device;
  - plot, hist, pairs, boxplot, ...
- Adding functions:
  - to insert new components/objects into existing plots;
  - points, lines, abline, legend, title, mtext, ...
- Interacting functions:
  - they allow user to interact with graphics;
  - locator, identify

To see the many possibilities that R provides

> *demo(graphics)*

High level graphical functions in R

R command	Description
plot()	Generic function for plotting of R objects. It can generate different plots: lines, points, bars ...
hist()	It computes a histogram of the given data values.
boxplot()	It computes box plot of the given data values.
qqnorm()/qqplot()	It computes Quantile-quantile (Q-Q) plot. of the given data values.
pairs()	It computes a plot for multivariate variables.
coplot()	It computes a conditioning plots of two variables. conditioned by a third variable.
...	...

Function plot()

Basic plotting function is plot(). Possible arguments to plot() include:

plot(x,y, xlim, ylim, xlab, ylab, type,pch,col ...)

x,y	:	coordinates of points in the plot (y may be omitted);
xlim=c(lo,hi)	:	the x axe range is between lo and hi;
ylim=c(lo,hi)	:	the y axe range is between lo and hi;
xlab	:	label for x-axe;
ylab	:	label for y-axe;
type	:	what type of plot should be drawn (i.e. p,l,b,h,. . .);
lty	:	line type (if lines used)
lwd	:	line width (if lines used)
pch	:	symbols to use when plotting points
col	:	color to be used for everything.

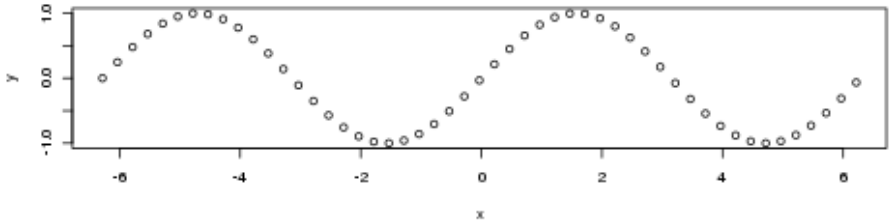
Function plot()

- A simple example:

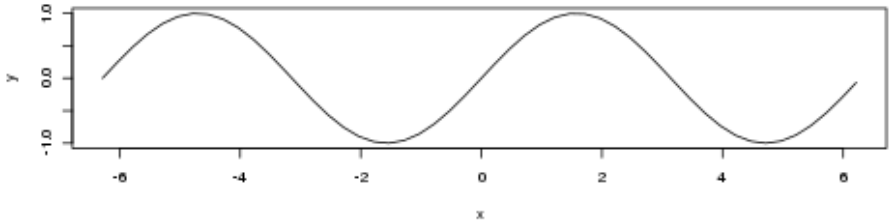
$> x = seq(-2 * pi, 2 * pi, 0.24)$

$> y = sin(x)$

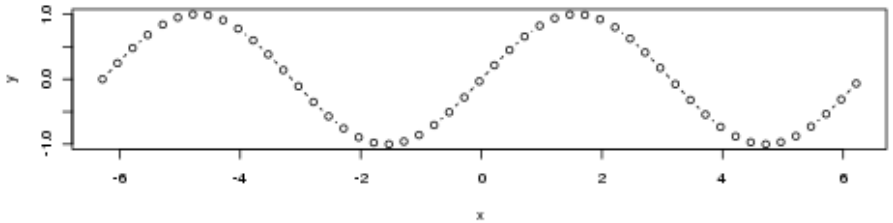
$> plot(x, y)$ **points**



$> plot(x, y, type = "l")$ **line**



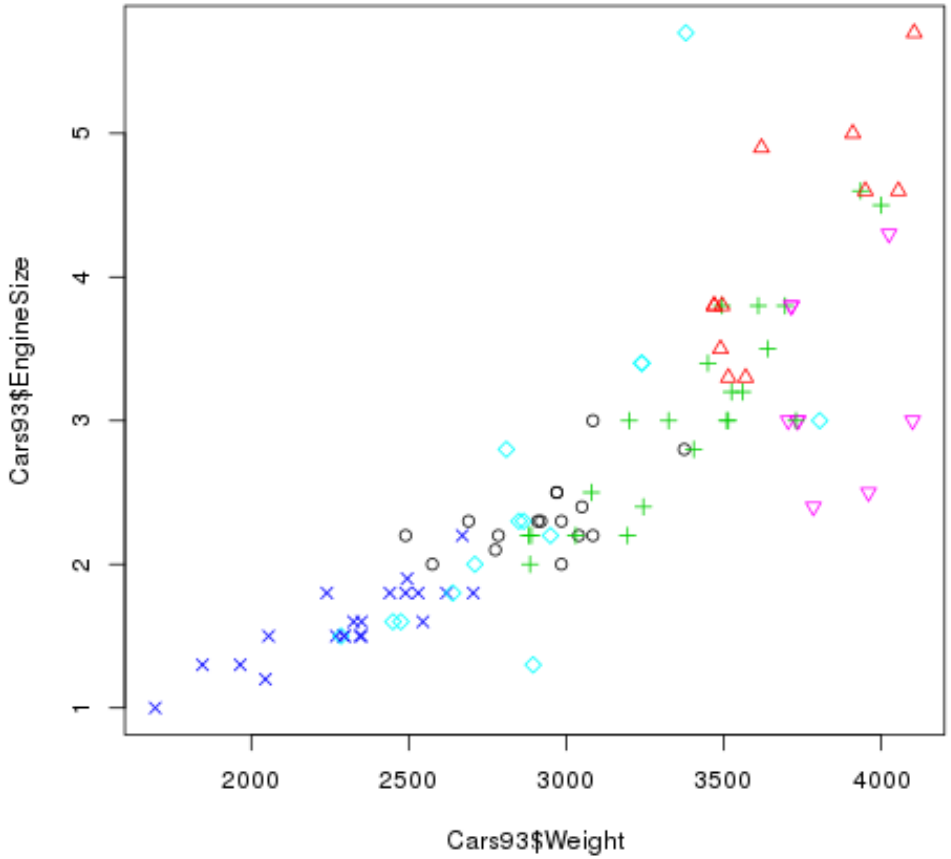
$> plot(x, y, type = "b")$ **points and line**



Function plot()

- A simple example using different colors and point types:

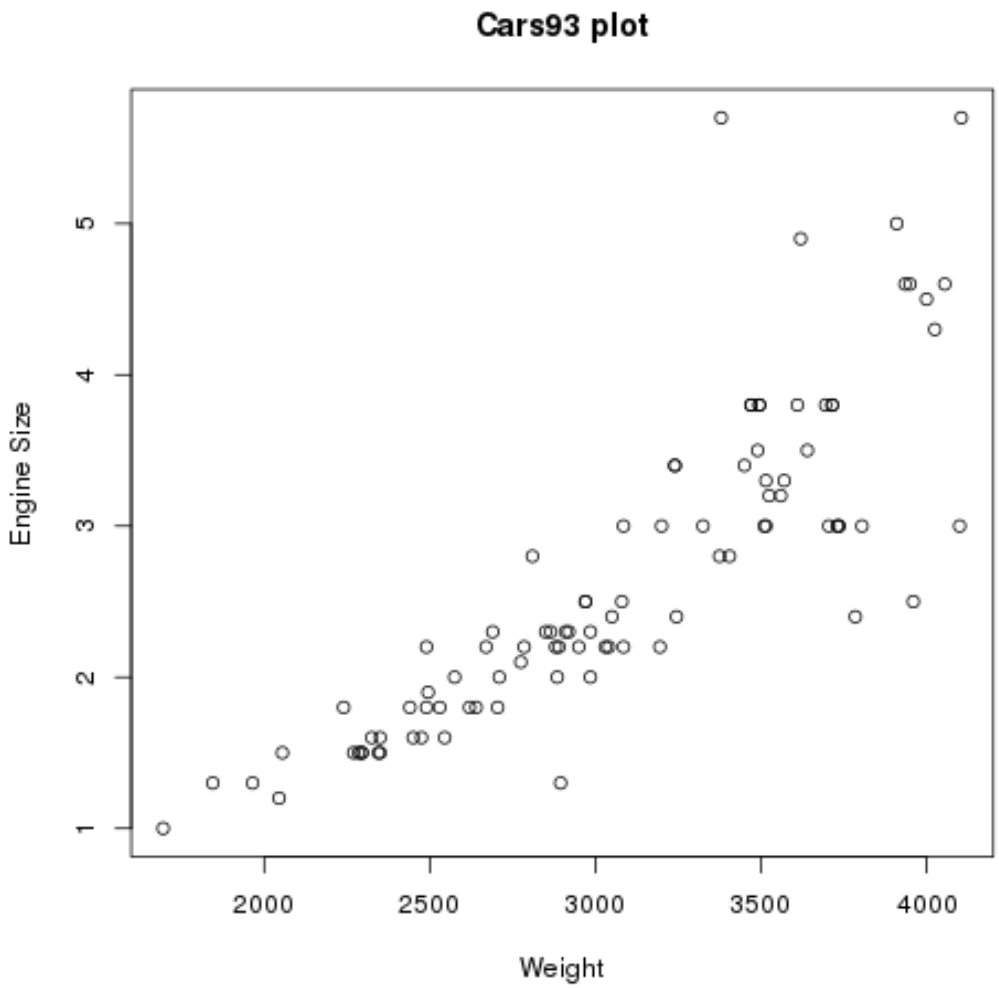
$> library(MASS)$
$> plot(Cars93\$Weight, Cars93\$EngineSize,$
$col = as.numeric(Cars93\$Type), pch = as.numeric(Cars93\$Type))$



Adding titles, lines, points ...

- To add x and y axes labels and a title.

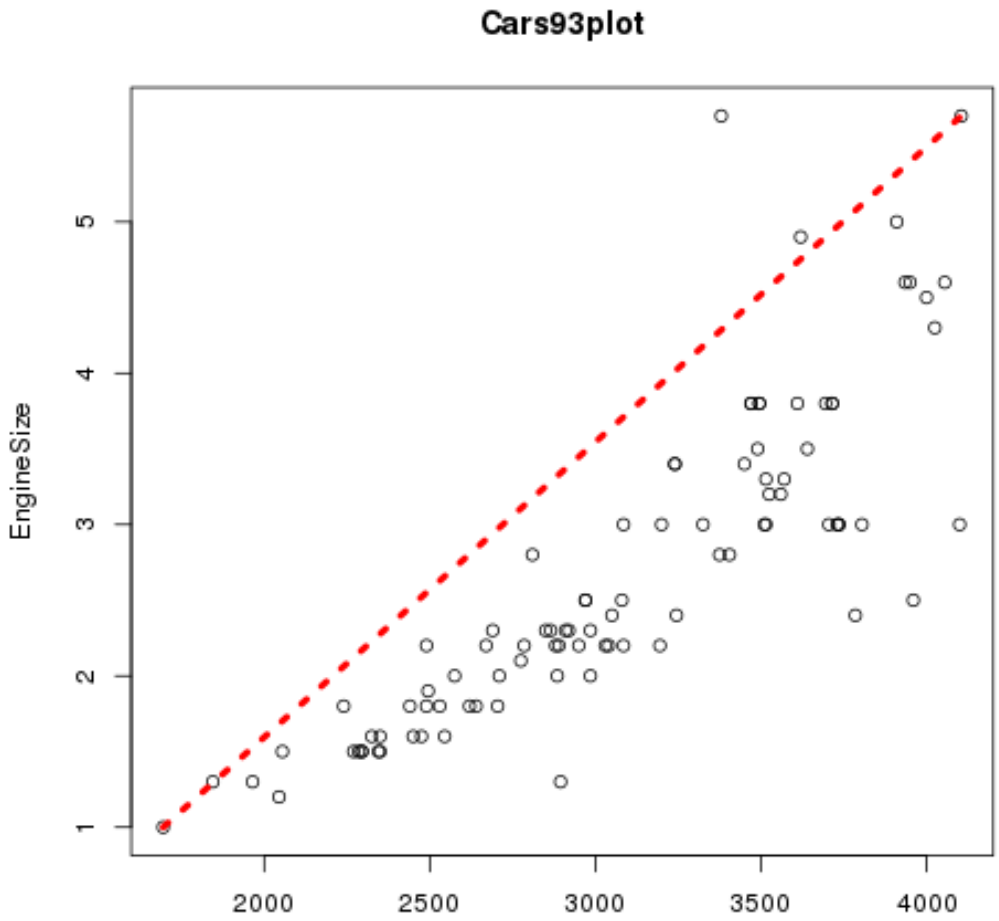
$> plot(Cars93\$Weight, Cars93\$EngineSize, ylab = "EngineSize", xlab = "Weight", main = "Cars93plot")$



Adding titles, lines, points ...

- To add a new line to the plot.

$$> lines(x = c(min(Cars93\$Weight), max(Cars93\$Weight)), y = c(min(Cars93\$EngineSize), max(Cars93\$EngineSize)), lwd = 4, lty = 3, col = "red")$$

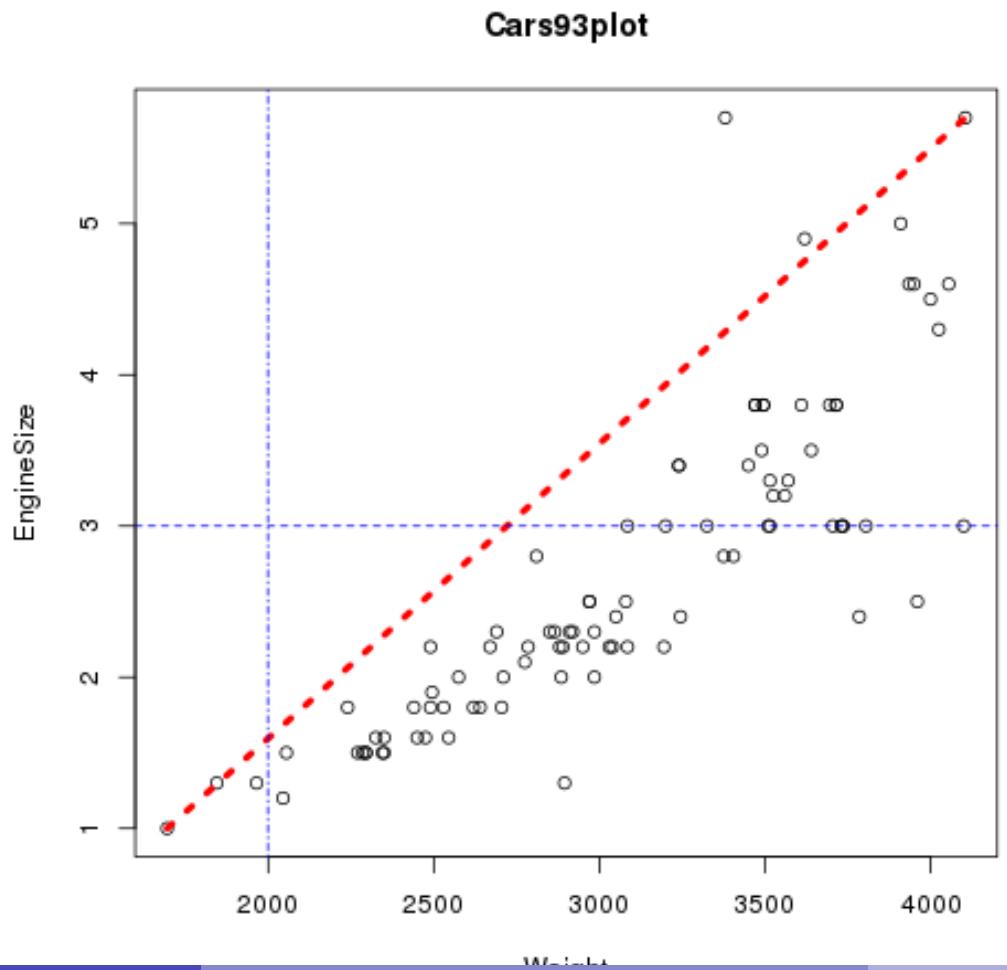


Adding titles, lines, points ...

- To add a new line to the plot.

$> abline(h = 3, lty = 2, col = "blue")$ **horizontal line.**

$> abline(v = 1999, lty = 4, col = "blue")$ **vertical line.**

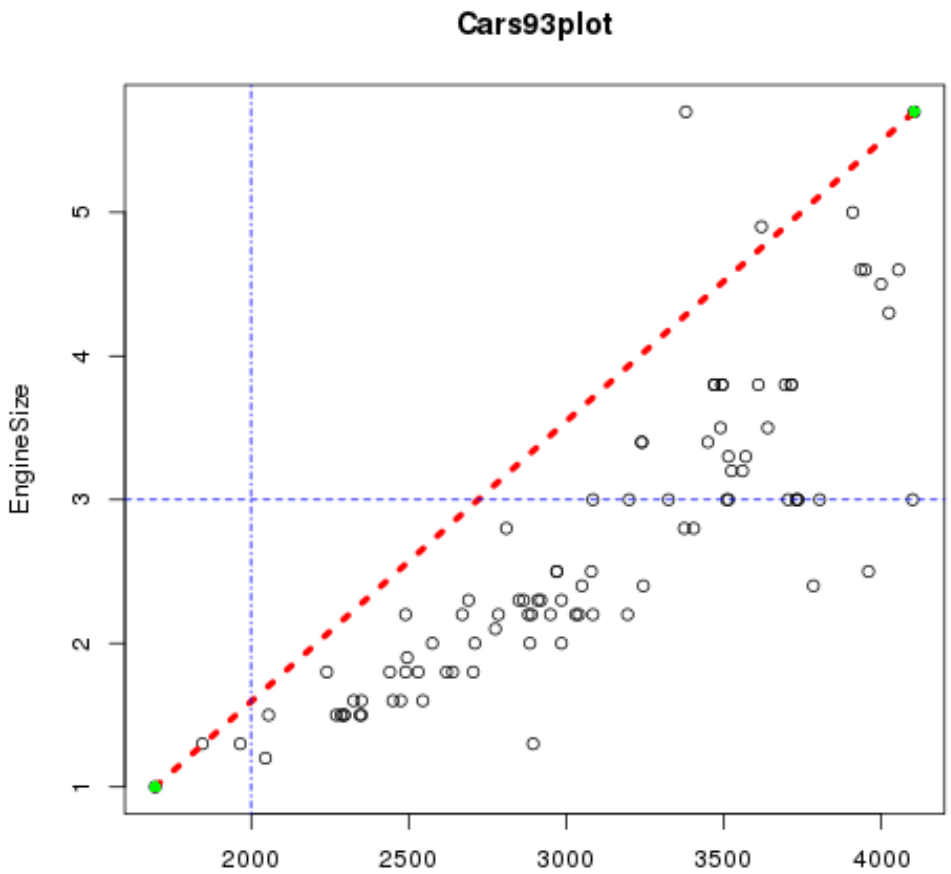


Adding titles, lines, points ...

- To add a new point to the plot.

$> points(x = min(Cars93\$Weight), y = min(Cars93\$EngineSize),$
$pch = 16, col = "green")$

$> points(x = max(Cars93\$Weight), y = max(Cars93\$EngineSize),$
$pch = 16, col = "green")$



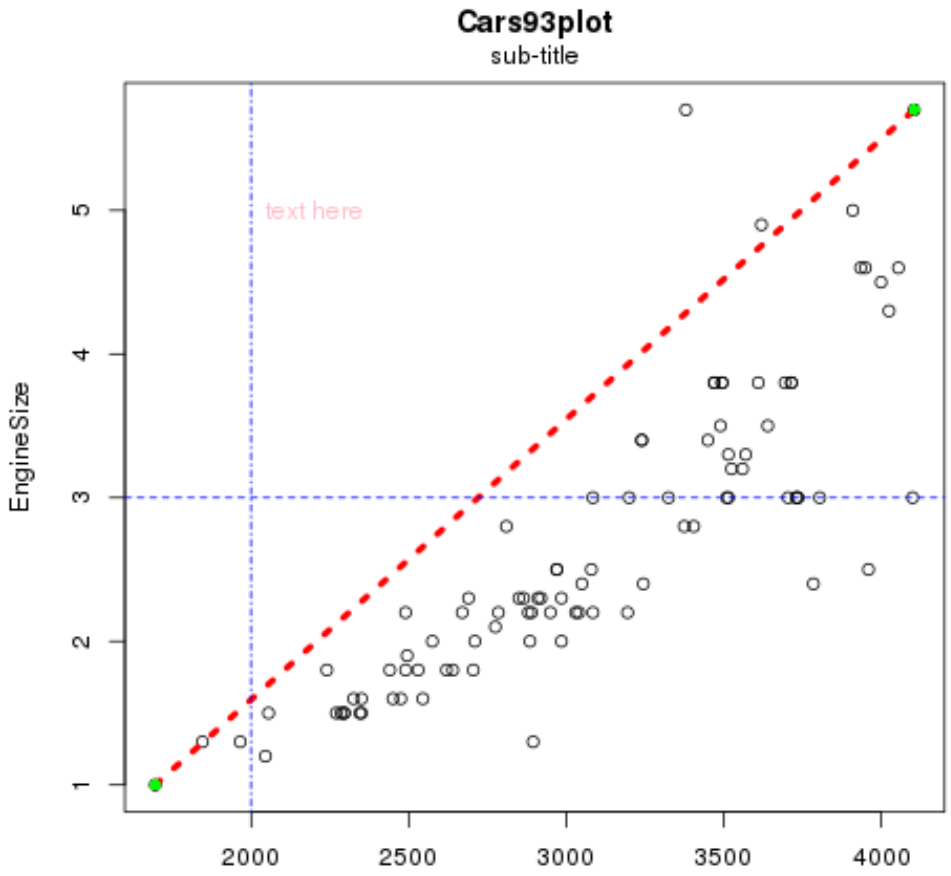
Adding titles, lines, points ...

- To add text to the plot.

  $> text(x = 2000, y = 5, "text\ here", col = "pink")$

- To add text under main title

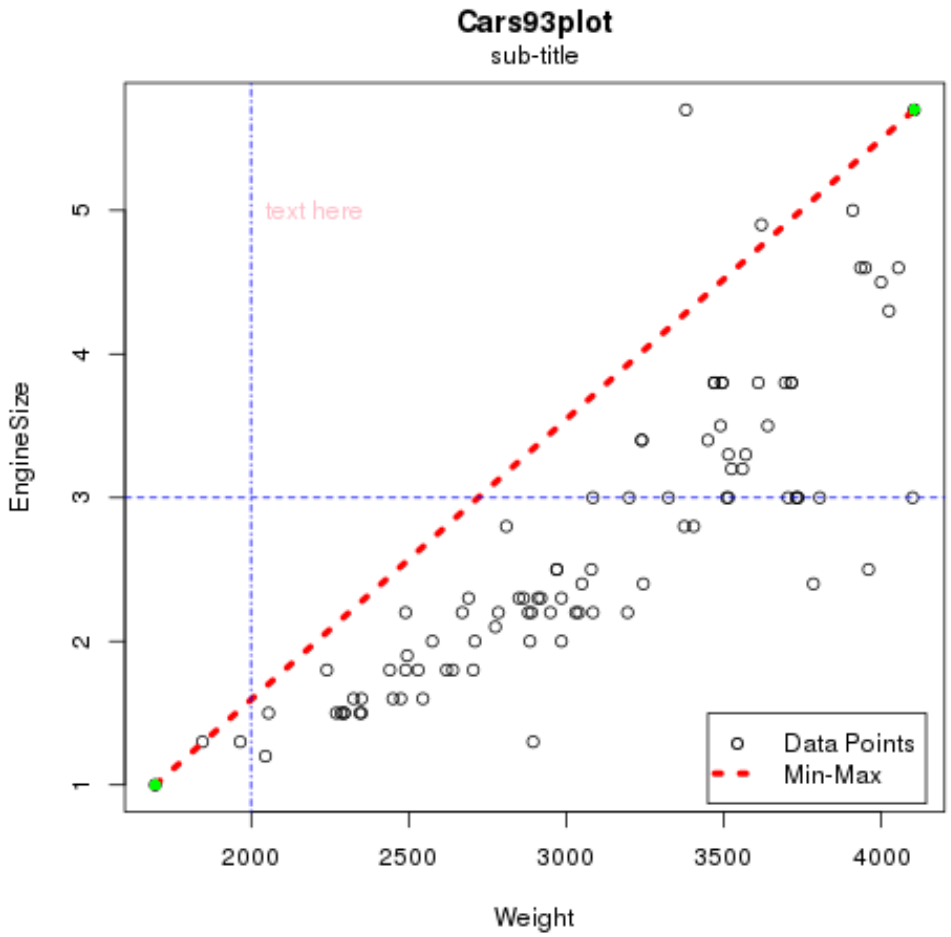
  $> mtext(side = 3, "subtitle", line = 0.45)$



Adding titles, lines, points ...

- To add legend to the plot.

$$> legend(x = 3450, y = 1.5, legend = c("DataPoints", "Min - Max"), pch = c("o", ""), lty = c(0, 3), lwd = c(0, 4), col = c("black", "red"))$$



Adding regression line in a plot

- Function lm() is used to fit linear models, then it can be used to carry out regression.

lm(formula, data, subset, ...)

formula : a symbolic description of the model to be fitted;
data : data frame, list, ... containing the variables in the model;
subset : an optional vector specifying a subset of observations to be used in the fitting process;

$> levels(Cars93\$Origin)$
[1]"*USA*" "*non-USA*"

We are going to generate a linear prediction with respect to Origin.

$> rgUSA < -lm(EngineSize \sim Weight, Cars93, subset = Origin == "USA")$

EngineSize is modelled by a linear predictor based on Weight.

$> rgOTHER < -lm(EngineSize \sim Weight, Cars93,$
$subset = Origin == "non\text{-}USA")$

Adding regression line in a plot

```
> summary(rgOTHER)

Call:
lm(formula = EngineSize ~ Weight, data = Cars93, subset = Origin ==
    "non-USA")

Residuals:
     Min       1Q   Median       3Q      Max
-0.89235 -0.14513  0.00823  0.13926  1.14337

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -8.580e-01  2.702e-01  -3.175  0.00277 **
Weight       1.054e-03  9.005e-05  11.701 5.96e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3547 on 43 degrees of freedom
Multiple R-squared:  0.761,     Adjusted R-squared:  0.7554
F-statistic: 136.9 on 1 and 43 DF,  p-value: 5.959e-15
```

Regression equation is $EngineSize = -8.580e^{-01} + 1.054e^{-03} * Weight$.
76.1% of date are described by the model
p-values are small then null hypothesis (the true coefficient is zero) is rejected.

Adding regression line in a plot

- To add regression line.

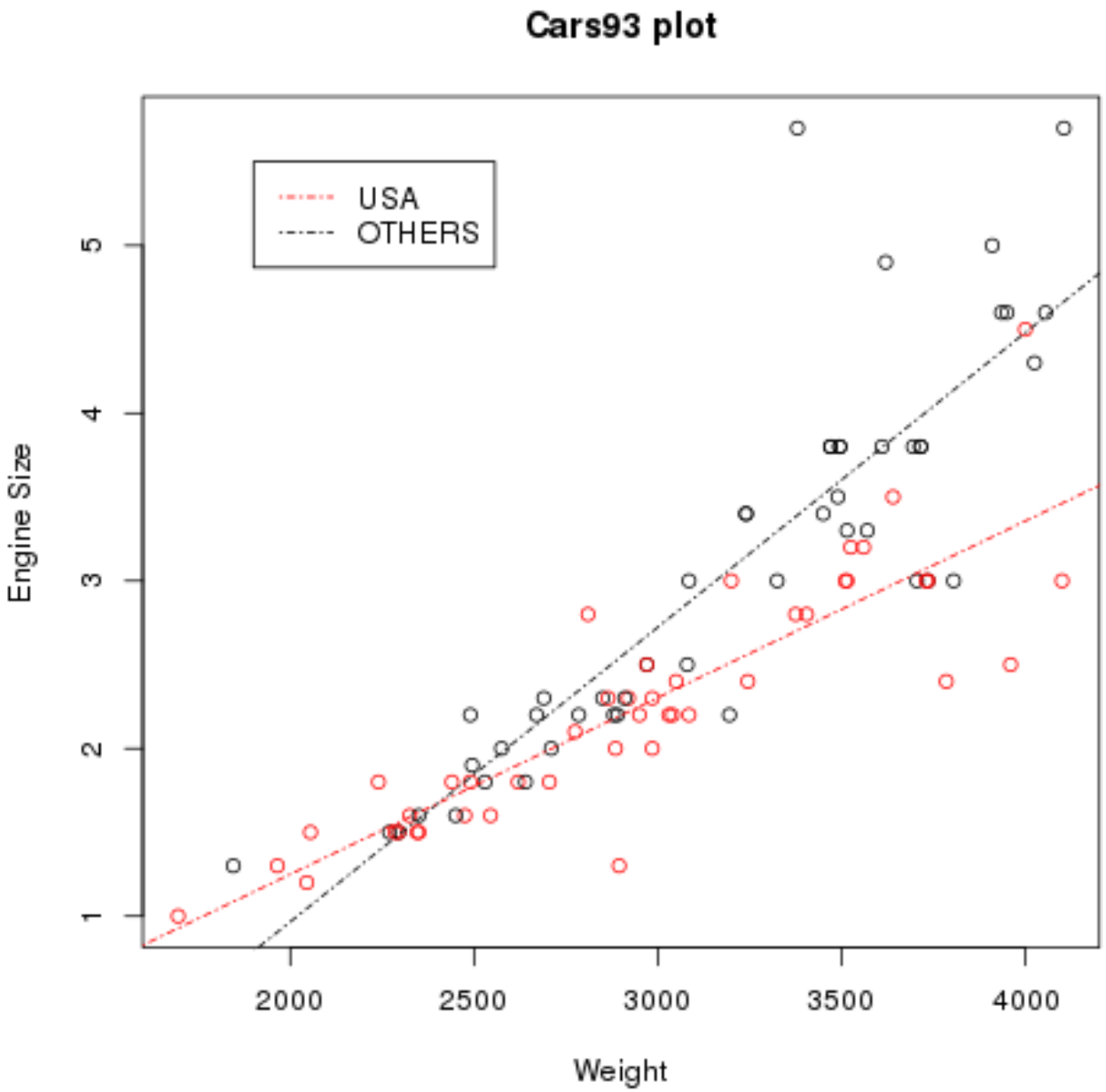
$> plot(Cars93\$Weight, Cars93\$EngineSize, ylab = \text{“}EngineSize\text{”},$
$xlab = \text{“}Weight\text{”}, main = \text{“}Cars93plot\text{”}, col = as.numeric(Cars93\$Origin))$

$> abline(coef(rgUSA), lty = 4, col = "red")$

$> abline(coef(rgOTHER), lty = 4, col = "black")$

$> legend(2000, 5, legend = c("USA", "OTHERS"), col =$
$c("red", "black"), lty = c(4, 4))$

Adding regression line in a plot



Multiple graphs in a plot

- To insert different graphs in a plot.

$> par(mfrow = c(2, 1))$

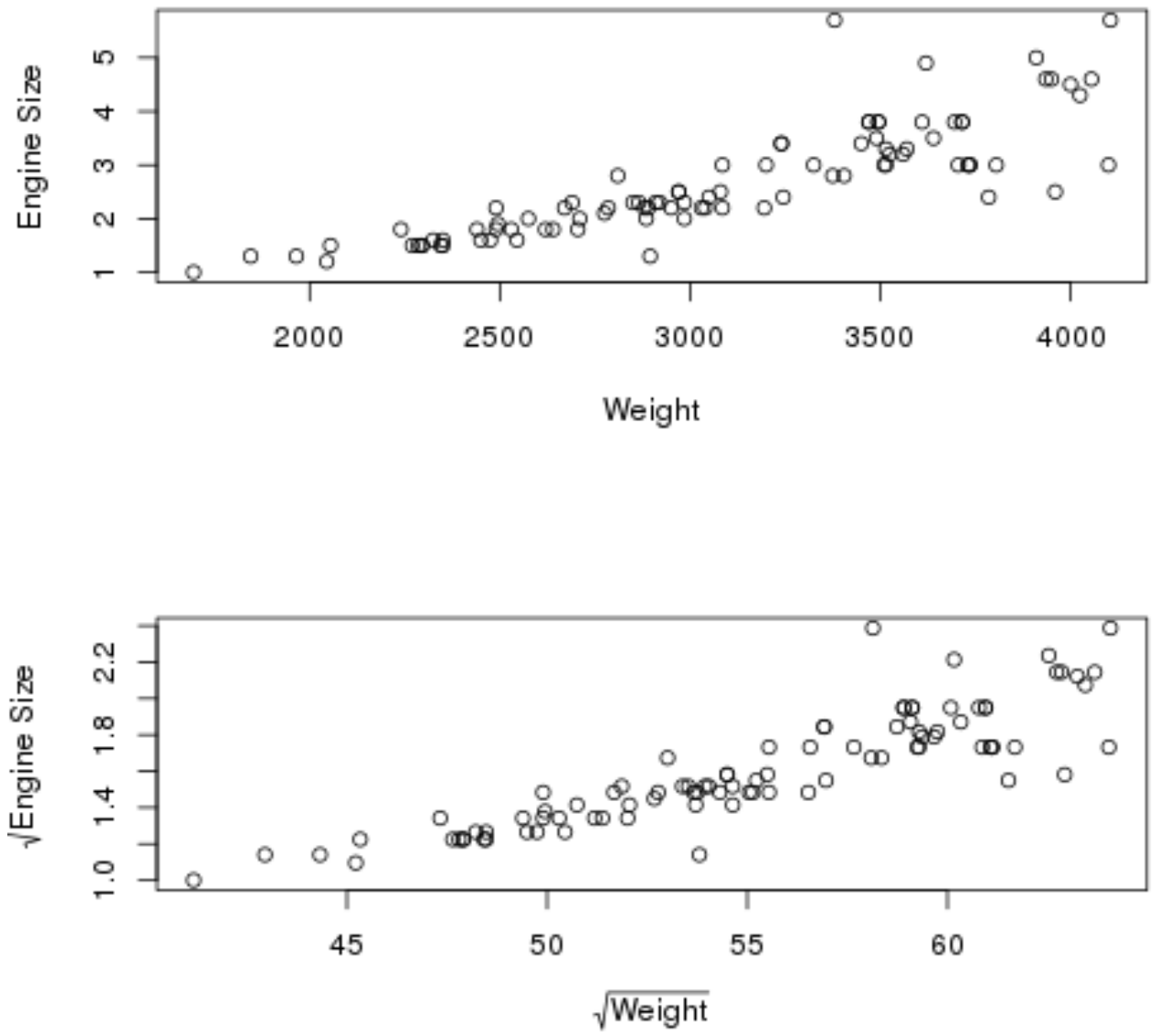
It will create 2 graphs in the same page (divided in a matrix 2×1)

$> plot(Cars93\$Weight, Cars93\$EngineSize,$
$xlab = "Weight", ylab = "EngineSize")$

$> plot(sqrt(Cars93\$Weight), sqrt(Cars93\$EngineSize),$
$xlab = expression(sqrt(Weight)), ylab = expression(sqrt(EngineSize)))$

The expression command plots mathematical symbols axes (see ?plotmath)

Multiple graphs in a plot



Multiple graphs in a plot

- To insert different graph in a plot.

$> par(mfrow = c(1, 2))$

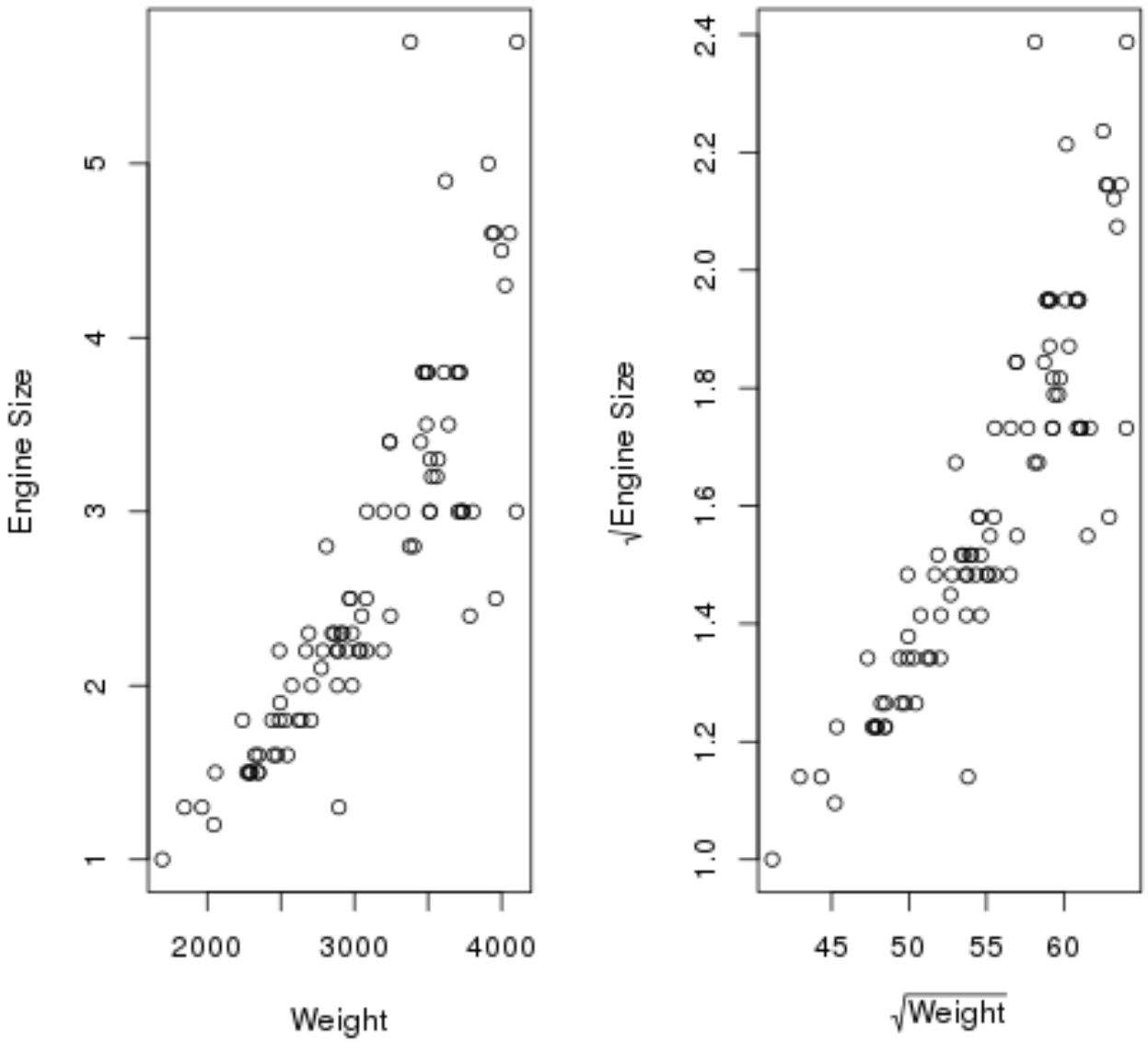
It will create 2 graphs in the same page (divided in a matrix 1×2)

$> plot(Cars93\$Weight, Cars93\$EngineSize,$
$xlab = "Weight", ylab = "EngineSize")$

$> plot(sqrt(Cars93\$Weight), sqrt(Cars93\$EngineSize),$
$xlab = expression(sqrt(Weight)), ylab = expression(sqrt("EngineSize")))$

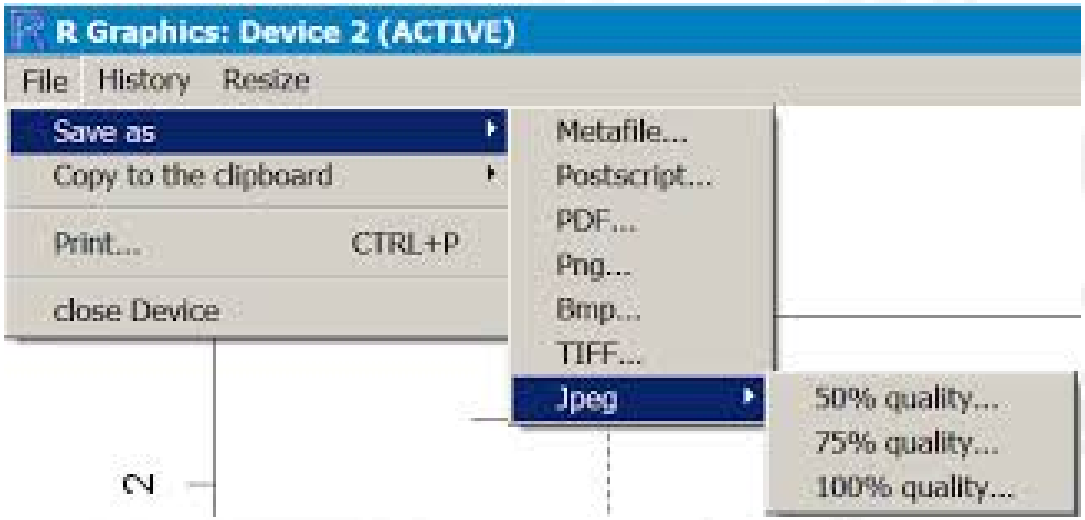
The expression command plots mathematical symbols axes (see ?plotmath)

Multiple graphs in a plot

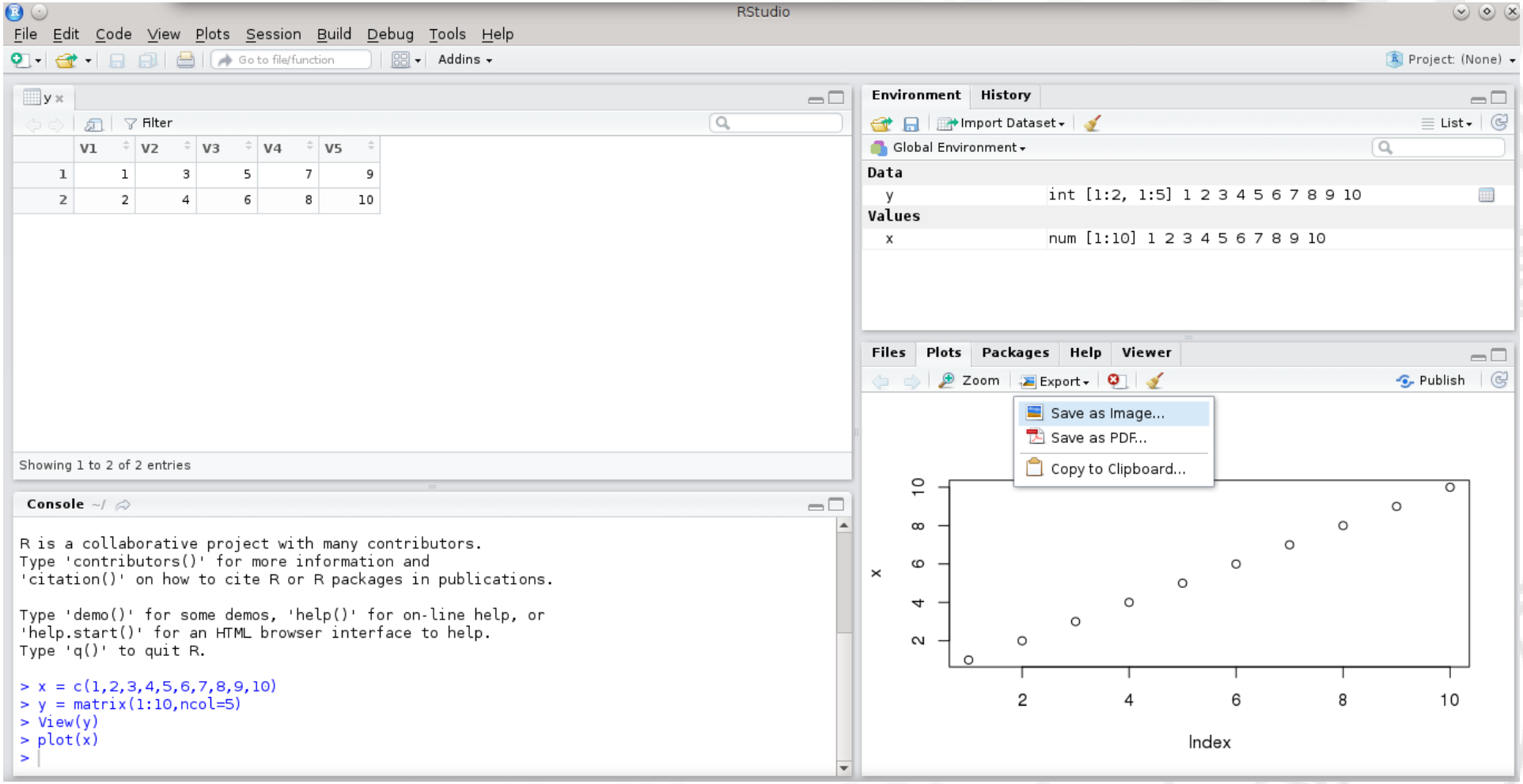


How to save a plot (by Window's GUI)

- Active graphic device by clicking on it;
- Then click File -> Save As -> ...



How to save a plot (by Rstudio)



How to save a plot (by console)

- Open the graphic devices for BMP, JPEG, PNG and TIFF format bitmap files.

  png(), bmp(), jpeg(), tiff() ...

  $> png("plot1.png")$ **saving a .png file.**

- Create the plot (it will be not visualized)

  $> plot(Cars93\$Weight, Cars93\$EngineSize, xlab = "Weight", ylab = "EngineSize")$

- Write the plot using command dev.off()

  $> dev.off()$

How to save a plot (by console)

- Open the graphics devices for BMP, JPEG, PNG and TIFF format bitmap files.

  png(), bmp(), jpeg(), tiff() ...

  $> jpeg("plot1.jpeg")$ **saving a .jpg file.**

- Create the plot (it will be not visualized)

  $> plot(Cars93\$Weight, Cars93\$EngineSize, xlab = "Weight", ylab = "EngineSize")$

- Write the plot using command dev.off()

  $> dev.off()$

Plotting a histogram

- Histograms can be created using the hist() command;

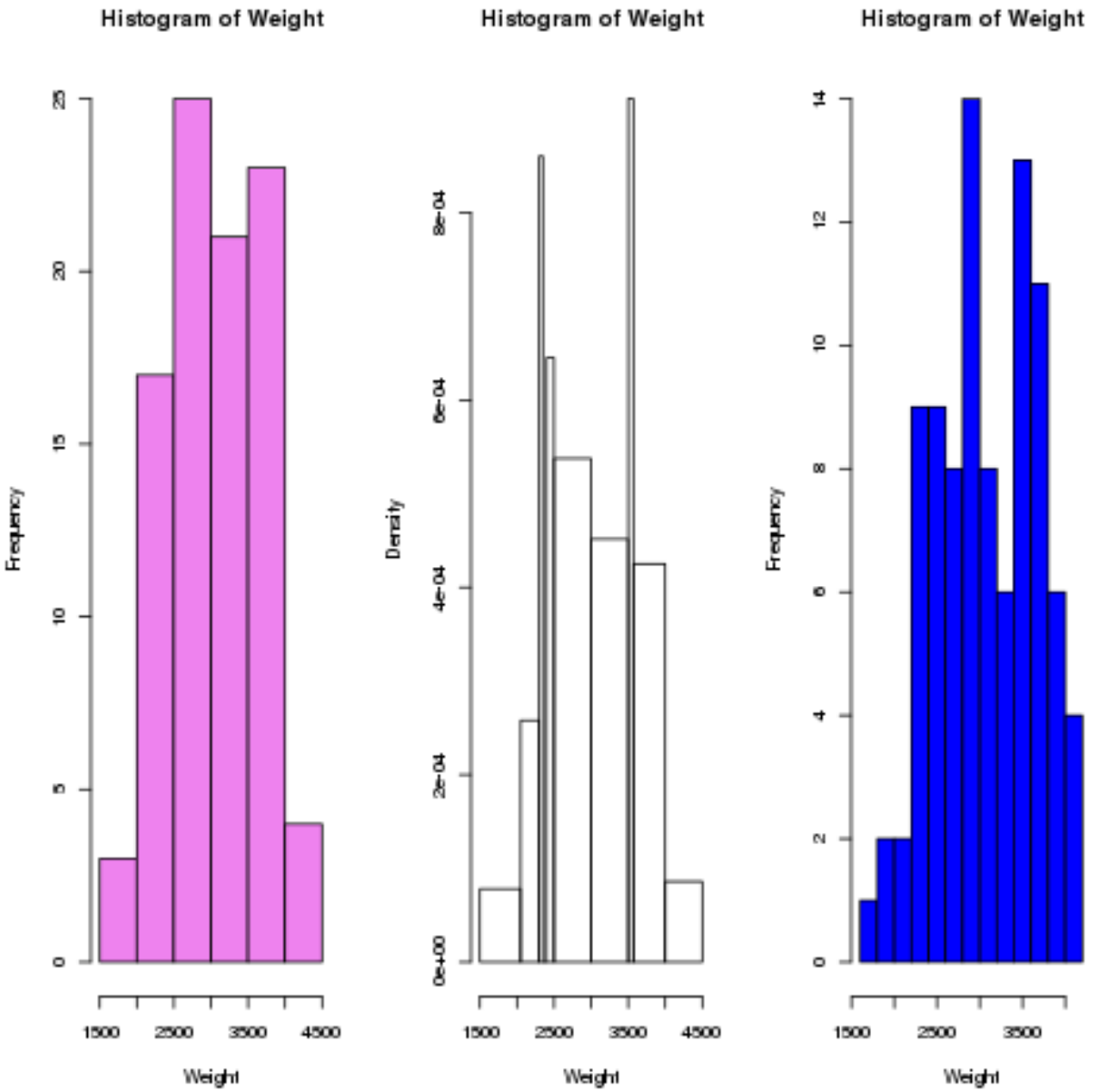
  $> hist(Cars93\$Weight, xlab = "Weight", main = "Histogram\ of\ \ Weight", col = "violet")$

- R automatically chooses the number and width of the bars.
  - User can also specify the location of the break points:

    $> hist(Cars93\$Weight, breaks = c(1500, 2050, 2300, 2350, 2400, 2500, 3000, 3500, 3570, 4000, 4500), xlab = "Weight", main = "HistogramofWeight")$
  - User can also specify the number of classes in which the data are split:

    $> hist(Cars93\$Weight, nclass = 10, xlab = "Weight", main = "HistogramofWeight", col = "blue")$

Plotting a histogram



Kernel Density plots in R

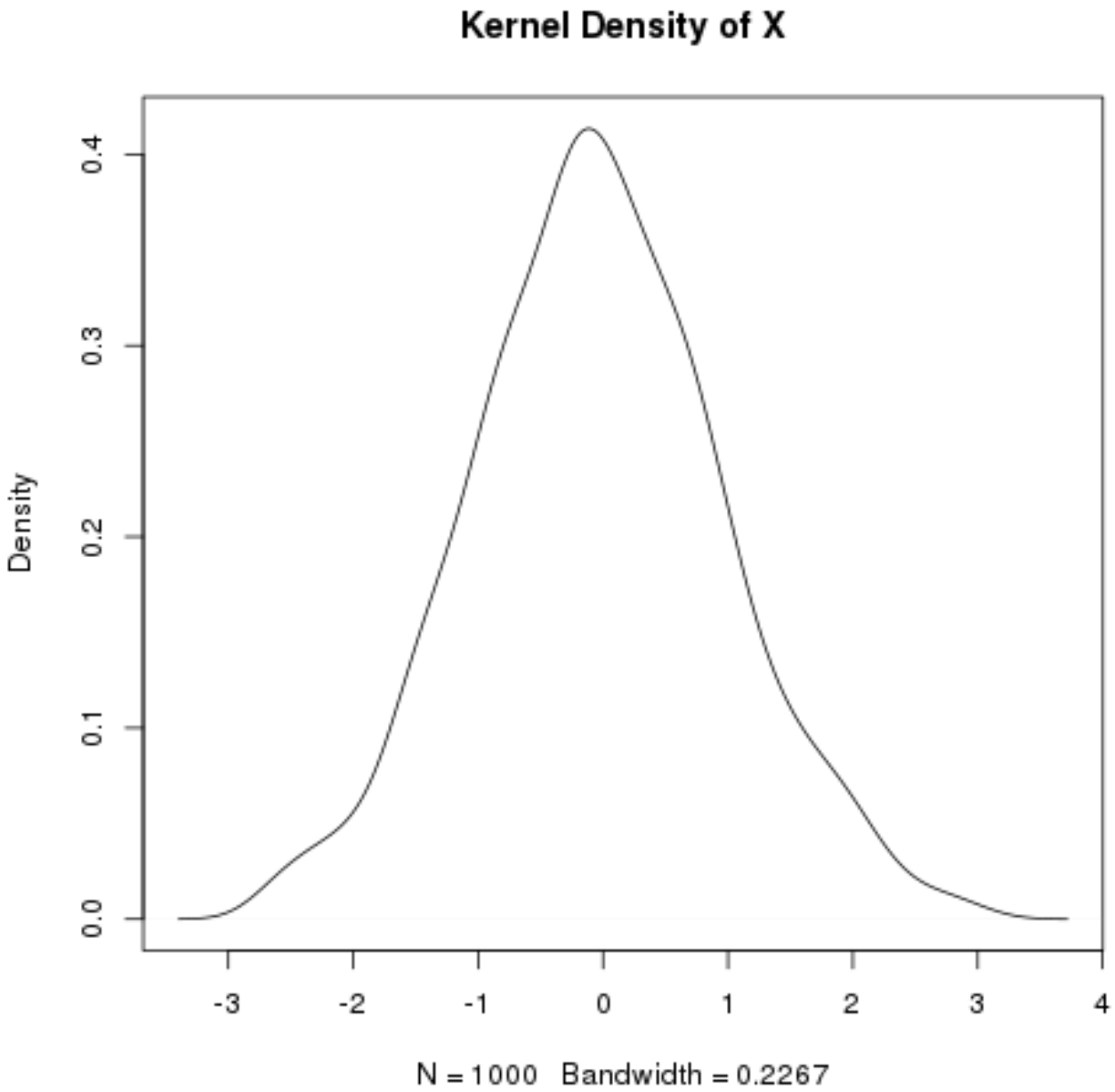
- Kernel density plots are usually a more effective way to view the distribution of a continuous variable;
- It requires to compute the kernel density estimation using function density()

$> x = rnorm(1000)$

$> dx = density(x)$

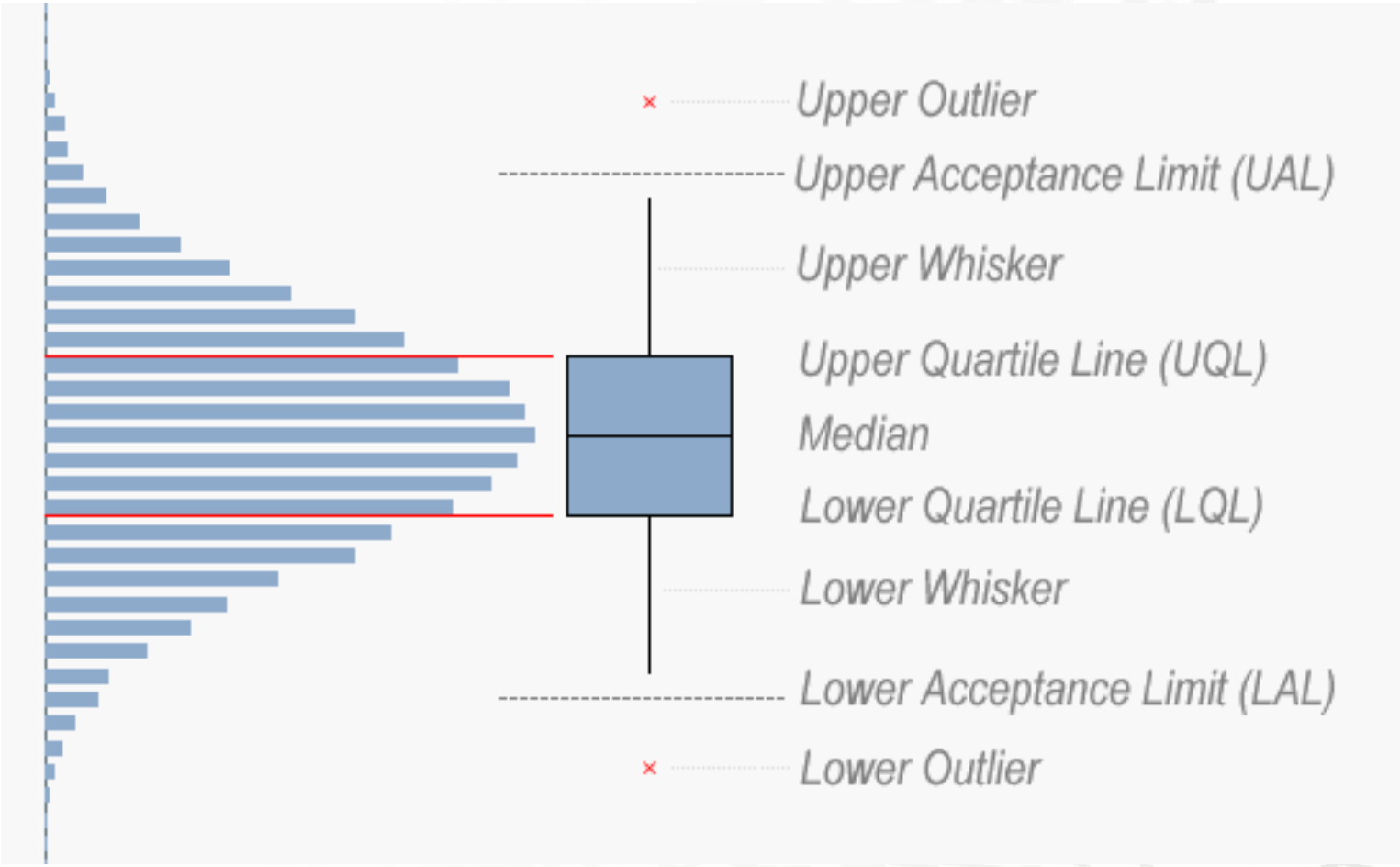
$> plot(dx, main = "Kernel\ Density\ of\ X")$

Kernel Density plots in R



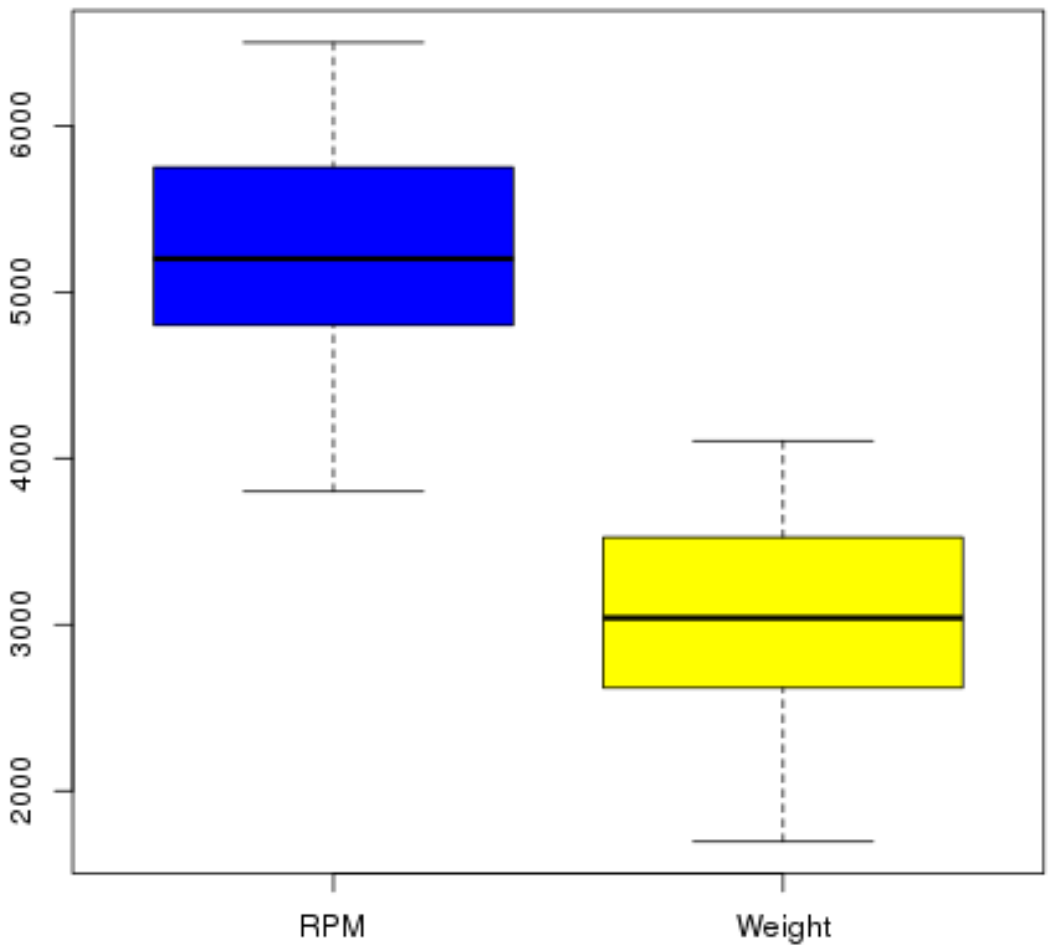
Box plots in R

- Box plot of a variable is a graphical representation based on its quartiles, as well as its smallest and largest values. It attempts to provide a visual shape of the data distribution.



Box plots in R

$> boxplot(Cars93\$RPM, Cars93\$Weight, col = c("blue", "yellow"), names = c("RPM", "Weight"))$

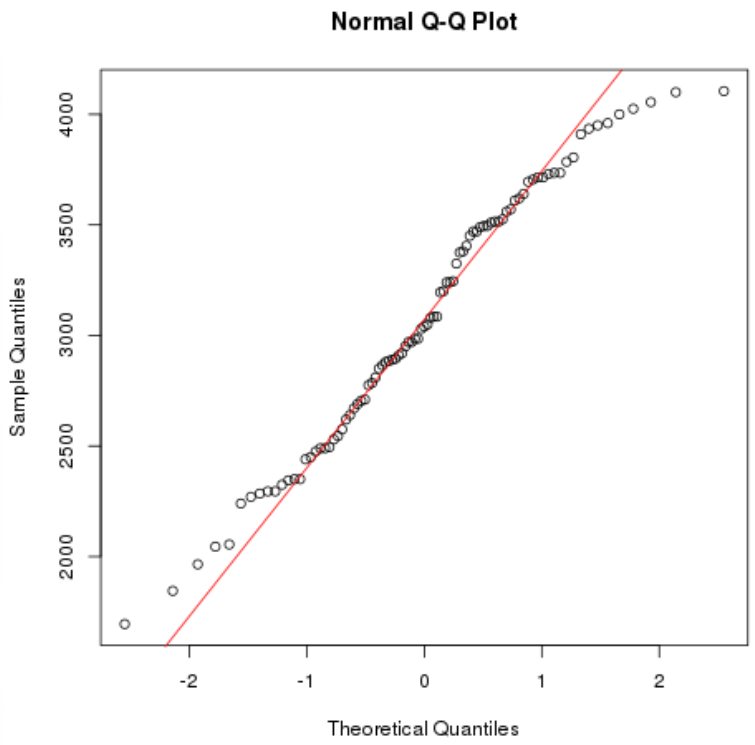


Q-Q plot in R

- Quantile-quantile (Q-Q) plot is a graphical technique for determining if two data sets come from populations with a common distribution.
- It plots the quantiles of the first data set against the quantiles of the second data set.

$> qqnorm(Cars93\$Weight)$ **Q-Q plot of the values in** *Cars93\$Weight* **with normal**

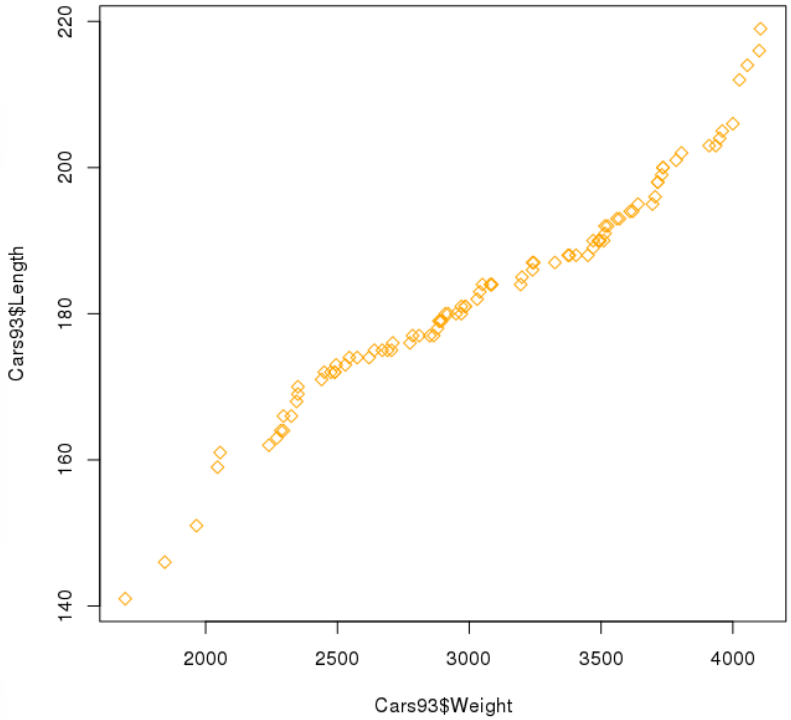
$> qqline(Cars93\$Weight, col = "red")$ **adds a line to a "theoretical", by default normal, q-q plot for** *Cars93\$Weight*



Q-Q plot in R

- Quantile-quantile (Q-Q) plot is a graphical technique for determining if two data sets come from populations with a common distribution.
- It plots the quantiles of the first data set against the quantiles of the second data set.

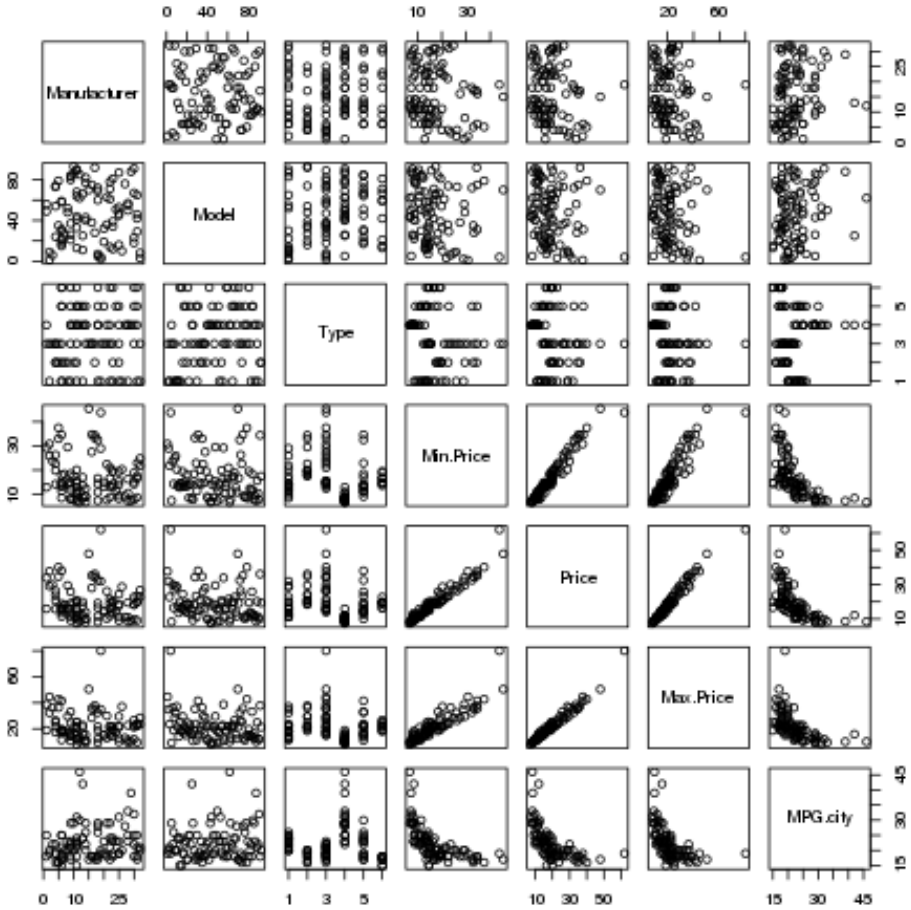
  $> qqplot(Cars93\$Weight, Cars93\$Length, col = "orange", pch = 5)$ **it produces a QQ plot of two datasets.**



Plotting multi-variate data in R

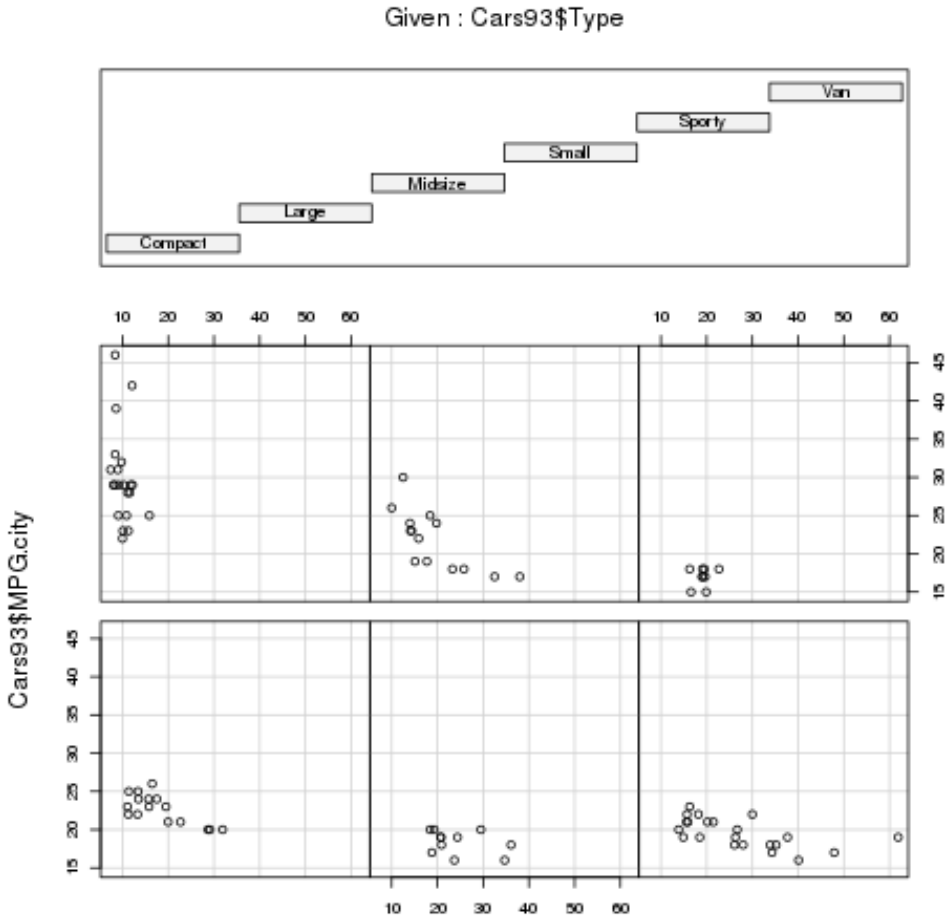
- If your data are stored in a data frame with several columns, pairs() command produces pairwise plots of the data in each column, i.e. the data in column 1 vs the data in column 2, column 1 vs column 3, and so on.

$> pairs(Cars93[, 1:7])$



Plotting multi-variate data in R

- function coplot() can be used to plot the values of a variable versus the values of another variable for every level of a third variable.
- For example: if a and b are numeric vectors and c is a numeric vector or factor, the command coplot($a \sim b|c$) produces plots of the values of a versus b for every level of c.
  $> coplot(Cars93\$MPG.city \sim Cars93\$Price|Cars93\$Type)$



Exercises on plots

1. Create a vector x of the values from 1 to 25;
2. Create a vector w = 1 + sqrt(x)/2;
3. Create a data frame called D, with columns x = x and y = x + rnorm(x)*w. To ensure we all get the same values, set the seed to 122345;
4. Create a histogram and a boxplot of y and plot them side-by-side on the same graphing region. Save the results as a png file;
5. Plot y versus x using an appropriate plotting command. Put a title on the graph, labels on the axes and a legend;
6. Enter the command f = lm(D$y ~ D$x, data=D) to fit a linear regression model. Add the estimated regression line to the current plot and make it in the colour blue;
7. Extract the values of the residuals using re = resid(f). Check that the residuals are normally distributed by creating a Q-Q plot.

Exercises on plots

- Create a vector x of the values from 1 to 25;
- Create a vector w = 1 + sqrt(x)/2;
- Create a data frame called D, with columns x = x and y = x + rnorm(x)*w. To ensure we all get the same values, set the seed to 12345;

$> x = 1 : 25$

$> w = 1 + sqrt(x)/2$

$> set.seed(12345)$
$> y = x + rnorm(x) * w$
$> D = data.frame(x, y)$

Exercises on plots

- Create a histogram and a boxplot of y and plot them side-by-side on the same graphing region. Save the results as a png file;

$> png("plot1.png")$

$> par(mfrow = c(1, 2))$

$> hist(y, col = "blue")$

$> boxplot(y)$

$> dev.off()$

Exercises on plots

- Plot y versus x using an appropriate plotting command. Put a title on the graph, labels on the axes and legend;

$> plot(D\$x, D\$y, type = "l", col = "blue", main = "X\ Vs\ Y", xlab = "X",$
$ylab = "Y")$
$> legend(10, 10, legend = "XvsY", col = "blue")$

Exercises on plots

- Enter the command f= lm(D$y ∼ D$x, data=D) to fit a linear regression model. Add the estimated regression line to the current plot and make it in the colour blue;

$$> f = lm(D\$y \sim D\$x, data = D)$$
$$> abline(coef(f), lty = 4, col = "blue")$$

Exercises on plots

- Extract the values of the residuals using re = resid(f). Check that the residuals are normally distributed by creating a Q-Q plot ;

$$> re = resid(f)$$
$$> qqnorm(re)$$
$$> qqline(re, col = "red")$$

Interactive functions

- locator(n) function reads *n* positions of the graphics cursor when the mouse button is pressed.
- locator() and text() functions can be combined together to print a text in a position specified by mouse pointer

  $> text(locator(1), "Critical\ Point")$

- identify(x) reads the position of the graphics pointer when the mouse button is pressed.

  $> x = 1 : 25$
  $> plot(x)$
  $> identify(x)$