DATA SCIENCE AND MACHINE LEARNING

Introduction to Basic Principles of R

Dimitris Fouskakis
Professor in Applied Statistics,
Department of Mathematics,
School of Applied Mathematical & Physical Sciences,
National Technical University of Athens
Email: fouskakis@math.ntua.gr

What is R?

- R is a dialect of the S language.

What is S?

- S is a language that was developed by John Chambers and others at Bell Labs.
- S was initiated in 1976 as an internal statistical analysis environment - originally implemented as Fortran libraries.
- Early versions of the language did not contain functions for statistical modeling.
- In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language).
- Version 4 of the S language was released in 1998 and is the version we use today.

Back to R

- 1991: Created in New Zealand by Ross Ihaka and Robert Gentleman. Their experience developing R is documented in a 1996 JCGS paper.
- 1993: First announcement of R to the public.
- 1995: Martin Machler convinces Ross and Robert to use the GNU General Public License to make R free software.
- 1996: A public mailing list is created (R-help and R-devel).
- 1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
- 2000: R version 1.0.0 is released.

Features of R

- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Runs on almost any standard computing platform/OS (even on the PlayStation 3).
- Frequent releases (annual + bugx releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages.
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists.
- It’s free!!!!!

Drawbacks of R

- Essentially based on 40 year old technology.
- Little built in support for dynamic or 3-D graphics (but things have improved greatly since the “old days").
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's your job! (Or you need to pay someone to do it).
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too.

Design of the R System

- The R system is divided into 2 conceptual parts:
  - The “base” R system that you download from CRAN.
  - Everything else.
- CRAN is the “Comprehensive R Archive Network”. It is a collection of sites which carry identical material, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries.
- R functionality is divided into a number of packages.
- The “base” R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.
- The other packages contained in the “base" system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.

Design of the R System

- And there are many other packages available.
- There are about 4000 packages (!!!) on CRAN that have been developed by users and programmers around the world.
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Some R Resources

- Available from CRAN (http://cran.r-project.org).
  - An Introduction to R
  - Writing R Extensions
  - R Data Import/Export
  - R Installation and Administration (mostly for building R from sources)

Books

- **Standard Texts:**
  - Adler (2010). R in a Nutshell, O’Reilly.
  - Albert (2007). Bayesian Computation with R, Springer.
  - Albert & Rizzo (2011). R by Example, Springer.
  - Chambers (2008). Software for Data Analysis: Programming with R, Springer.
  - Crawley (2007). The R book, Wiley.
  - Dalgaard (2002). Introductory Statistics with R, Springer – Verlag.
  - Everitt & Hothorn (2006). A Handbook of Statistical Analyses using R, Chapman & Hall/CRC Press.
  - Venables & Ripley (2002). Modern Applied Statistics with S, Springer.
  - Murrell (2005). R Graphics, Chapman & Hall/CRC Press.
  - Φουσκάκης (2013). Ανάλυση Δεδομένων με Χρήση της R. Εκδόσεις Τσότρας.

- **Other resources:**
  - Springer has a series of books called Use R!.
  - A longer list of books is at http://www.r-project.org/doc/bib/R-books.html.

Install R

- The home page for the R project, located at http://r-project.org, is the best starting place for information about the software. It includes links to CRAN, which features precompiled binaries as well as source code for R, add-on packages, documentation (including manuals, frequently asked questions, and the R newsletter) as well as general background information. Mirrored CRAN sites with identical copies of these files exist all around the world. New versions are regularly posted on CRAN, which must be downloaded and installed.

Install R

- Windows: More information on Windows-specific issues can be found in the CRAN R for Windows FAQ (http://cran.r-project.org/bin/windows/base/rw-FAQ.html).
- Mac OS: More information on Macintosh-specific issues can be found in the CRAN R for Mac OS X FAQ (http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html).
- Linux: Precompiled distributions of R binaries are available for the Debian, Redhat (Fedora), Suse and Ubuntu Linux, and detailed information on installation can be found at CRAN.
- In this course the windows version will be used.

R Commander

- If you wish you can download a more user-friendly graphical interface that works in concert with R called R Commander, created by John Fox. Users familiar with SPSS and other drop-down menu type programs will initially feel more comfortable using R Commander than R. The R environment is run with scripts, and in the long run it can be much more advantageous to have scripts that you can easily bring back up as you perform multiple calculations.
- Here we will use the default graphical interface.

Running R

- Once installation is complete, the recommended next step for a new user would be to start R by double click the R program.
- Then the R console shows up.
- The `>' character is the prompt, and commands are executed once the user presses the RETURN key.
- To terminate the program either press q(), or close the R main window (not the R console), or from the menu file choose exit. In all cases you will be asked if you wish to save the workspace image and all the objects you have created in your session.

R windows

- R – Console: You may write R commands there. This window also displays all the commands R has run, the results, and error report. This window appears when you launch R. To bring this window to the foreground, click on the window or go to the Windows pull-down menu and choose the R Console. You can type and run commands in this window line by line.

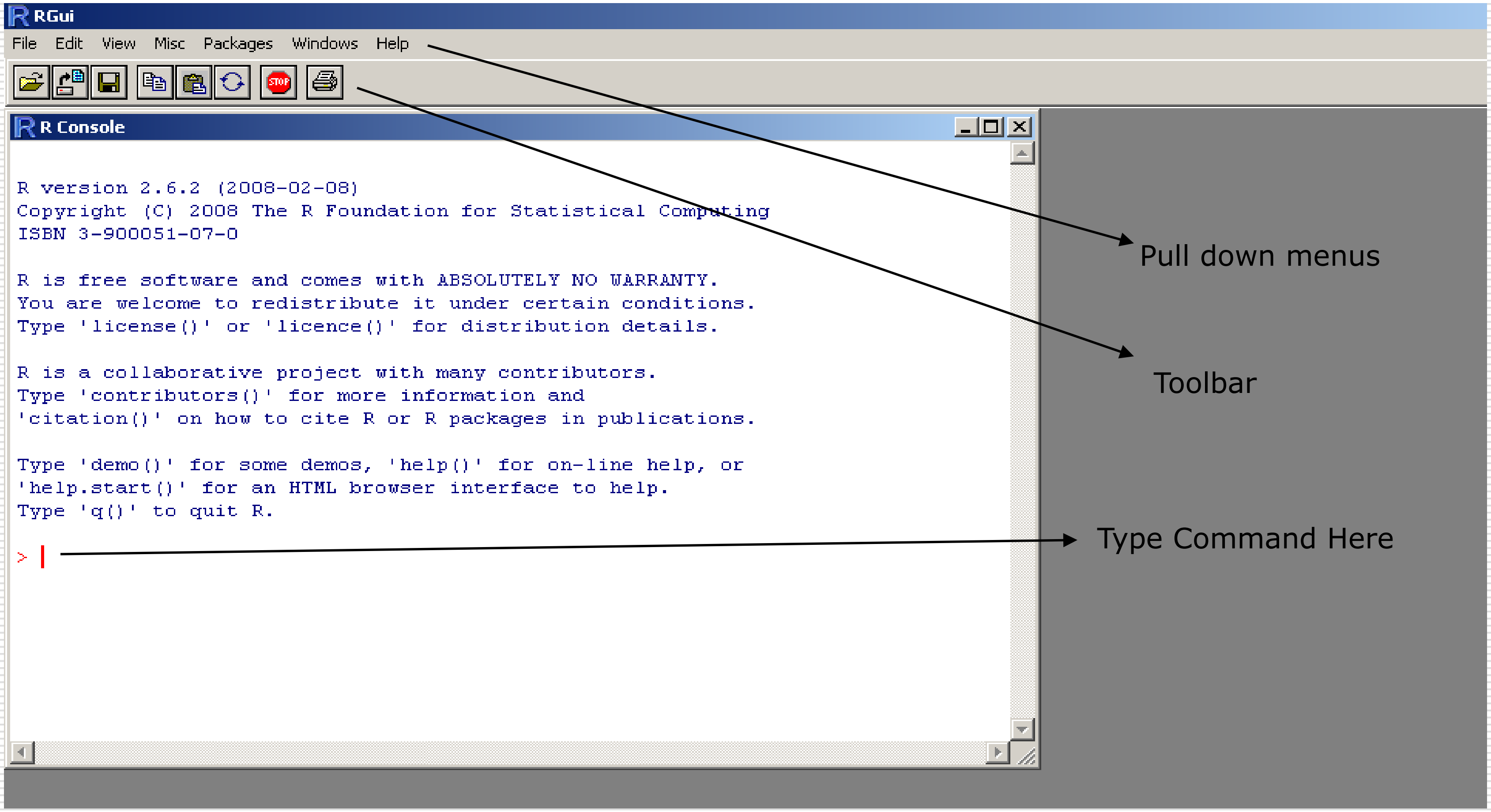
R windows

- R – Editor: Alternatively, you can write all your commands there and allow R to run part or all of the commands at once. You open this window by clicking the New Script option in the File menu. If you want to redo your analysis at a later time, or send your code in a file to someone else, you can save the scripts. You can also print the scripts by clicking the Print option from the File menu.

R windows

- R – Graphics: This window opens automatically when you create a graph. To bring a graph to the foreground, click on the graphics window or go to the Windows pull-down menu and choose the R Graphics window. Graphs can be saved in various formats, such as jpg, png, bmp, ps, pdf, emf, pictex, x_g and so on.

R Console

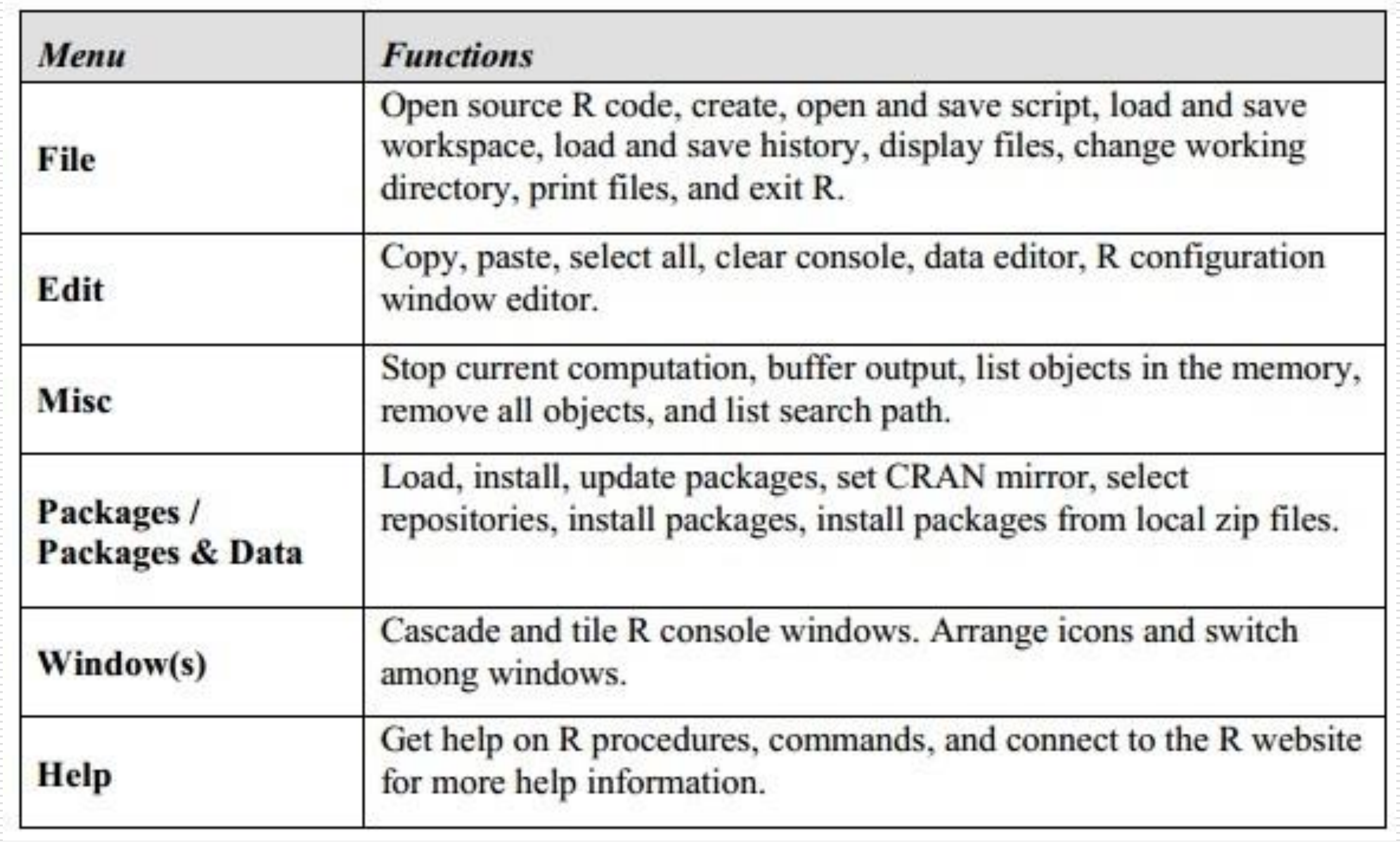


```
R version 2.6.2 (2008-02-08)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

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.

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.

> |
```

Toolbar & Menu Bar

- Like most window-based programs, R has a toolbar and a menu bar with pull-down menus that you can use to access many of the features of the program. The toolbar contains buttons for more commonly used procedures. To see what each button does, hold the mouse over the button for a moment and a description of what the button does will appear. The following is a summary of the main pull-down menus and their functions:

Toolbar & Menu Bar



Menu	*Functions*
File	Open source R code, create, open and save script, load and save workspace, load and save history, display files, change working directory, print files, and exit R.
Edit	Copy, paste, select all, clear console, data editor, R configuration window editor.
Misc	Stop current computation, buffer output, list objects in the memory, remove all objects, and list search path.
Packages / Packages & Data	Load, install, update packages, set CRAN mirror, select repositories, install packages, install packages from local zip files.
Window(s)	Cascade and tile R console windows. Arrange icons and switch among windows.
Help	Get help on R procedures, commands, and connect to the R website for more help information.

Getting Help

- R provides help files for all its functions. To access a help file, use the Help pull-down menu, or type ? followed by a function name in the R console. For example, to get help on the scan function, type:

```
> ?scan
```

Notation & Common R Operators

> Indicates the prompt at the start of each new line in the R console.

# The comment operator. Often called the “hash sign” or “pound sign”. Any type following the # on a line indicates a comment. R will ignore (not execute or attempt to interpret) anything written as a comment. Comments are a good way to indicate what you intend a line to do, or to describe a variable or command. They are useful to you when you read your code later or, and are useful to others with whom you share your code.

<- The assignment operator. This operator assigns the value on the right to the symbol on the left. Some pronounce this as “gets”, so the statement x <- 5 is read “x gets 5”. Once you have assigned 5 to x, R will fill in the value 5 in expressions using x. You can also use the = sign for assignment, but in R the <- sign is more conventional. The operator -> assigns the value on the left to the symbol on the right.

== Boolean equality operator. A double = sign is used in logical statements, assessing whether two quantities are equal.

```
x <- 5   # assigns the value 5 to x
x == 5  # returns TRUE
x = 6    # assigns the value 6 to x, overwriting the previous statement x <- 5.
           [Note that '=' accomplishes the same thing as '<-'] Now,
x == 5 # returns FALSE
```

Arithmetic Operators in R

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
% / %	Integer Division
% %	Modulus

Example

```
> 3+3  # this is my first command
[1] 6
> 9-2
[1] 7
> 17/3
[1] 5.666667
> 4*2
[1] 8
> 2^3
[1] 8
> 17%/%3
[1] 5
> 17%%3
[1] 2
> 7/0
[1] Inf
```

Special Numbers

- R supports 4 special numeric values: Inf, -Inf, NaN, NA and NULL. The first 2 are positive and negative infinity. NaN is short for “not-a-number” and means that our calculation either didn’t make mathematical sense or could not be performed properly. NA is short of “not available” and represents a missing value-a problem all too common in data analysis. NULL represents a null object in R.
- There a functions available to check for these special values (see examples in the next slides).

Special Numbers Example

```
> 7/0
[1] Inf
> -10^1000
[1] –Inf
> exp(-Inf)
[1] 0
> log(-2)
[1] NaN
> 0/0
[1] NaN
> x<-1
> names(x)
[1] NULL
> is.null(x)
[1] FALSE
> is.null(names(x))
[1] TRUE
> x<-c(0,Inf,-Inf, NaN, NA)
> is.finite(x)
[1] TRUE FALSE FALSE FALSE FALSE
> is.infinite(x)
[1] FALSE TRUE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE TRUE FALSE
> is.na(x)
[1] FALSE FALSE FALSE TRUE TRUE
```

Logical Operators

Operator	**Description**
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x|y	x OR y
x&y	x AND y
isTRUE(x)	Test if x is TRUE

Examples

```
> x<-56
> x
[1] 56
> 5*2->y
> y
[1] 10
> x>y
[1] TRUE
> y<4
[1] FALSE
```

```
> x>=56
[1] TRUE
> y<=9
[1] FALSE
> x==56
[1] TRUE
> y!=1
[1] TRUE
> !(5>3)
[1] FALSE
```

Examples

```
# An example
> x <- c(1:10)
> x[(x>8) | (x<5)]
# yields 1 2 3 4 9 10

# How it works
> x <- c(1:10)
> x
1 2 3 4 5 6 7 8 9 10
> x > 8
F F F F F F F F T T
> x < 5
T T T T F F F F F F
> x > 8 | x < 5
T T T T F F F F T T
> x[c(T,T,T,T,F,F,F,F,T,T)]
1 2 3 4 9 10
```

More on the Assignment Operator

- R είναι is case sensitive, i.e. x and X are different objects.
- When you use the assignment operator the result is not visible on screen. To see the value that your object took just type its name.
- Objects could also be vectors, matrices, data frames, or lists (more later).
- An object could be present in both parts of the assignment operator, but before we should give it a value.

```
> x<-5
> x<-x+3
> x
[1] 8
```

More on Logical Operators

- We usually use the symbols & (AND), | (OR) to combine two or more logical operators.

```
> (5>3) & (8>10)
[1] FALSE
> (5>3) | (8>10)
[1] TRUE
```

- Logical operators are frequently used in loops, e.g. if, for, etc. (more later).

Basic Mathematical Functions in R

Function	**Description**	**Function**	**Description**
sqrt()	Square root	asin()	Inverse sine
abs()	Absolute value	atan()	Inverse tangent
log()	Natural logarithm (ln)	gamma()	Gamma function
log2()	Logarithm base 2	lgamma()	Natural logarithm of gamma function
log10()	Logarithm base 10	beta()	Beta function
exp()	Exponential function	floor()	Previous integer
cos()	Cosine	ceiling()	Next integer
sin()	Sine	factorial()	Factorial
tan()	Tangent	choose()	Combinations
acos()	Inverse cosine	lchoose()	Natural logarithm of combinations

Examples

```
> sqrt(16)
[1] 4
> abs(-2)
[1] 2
> log(10)
[1] 2.302585
> log2(10)
[1] 3.321928
> log10(10)
[1] 1
> exp(3)
[1] 20.08554
> cos(pi)
[1] -1
> sin(2*pi)
[1] -2.449213e-16
> tan(pi/2)
[1] 1.633178e+16
```

```
> sin(pi/2)
[1] 1
> tan(0)
[1] 0
> acos(0.2)
[1] 1.369438
> atan(2)
[1] 1.107149
> asin(0)
[1] 0
> gamma(2)
[1] 1
> beta(2,3)
[1] 0.08333333
```

```
> lgamma(4)
[1] 1.791759
> floor(4.9)
[1] 4
> ceiling(4.1)
[1] 5
> factorial(5)
[1] 120
> choose(5,2)
[1] 10
> lchoose(5,2)
[1] 2.302585
```

General Functions in R

- **builtins():** built-in functions in R.
- **cat()** or **print():** Print on screen.
- **ls():** list all objects
- **rm():** remove objects.
- **getwd():** returns working directory.
- **setwd():** changes working directory.
- **list.files():** returns a list with all files in working directory.
- **date()** or **Sys.time():** Returns current day & time.

Classes of Objects in R

- R has five basic classes of objects:
  - Numeric (real number)
    ```
    > x<- 3
    ```
  - Complex
    ```
    > x<-complex(real=4, imaginary=3)
    > x
    [1] 4 +3i
    ```
  - Logical
    ```
    > x <-3
    > y <- x > 4
    > y
    [1] FALSE
    ```
  - Character
    ```
    > x <- "DIMITRIS"
    > x
     [1] "DIMITRIS"
    ```

- Use the function mode() or class() to check the objects class.

Data Types in R

- Vectors.
- Matrices.
- Arrays.
- Data frames.
- Lists.

- In the 3 first cases objects should be of the same type.
- A list allows you to gather a variety of (possibly unrelated) objects under one name.

Vectors

- Vectors in R are lists of objects of the same type.
  - Numerical Vectors.
  - Character Vectors.
  - Logical Vectors.
  - Factors.

Numerical Vectors

- To create a numerical vector use the function c().

```
> x<-c(1,2,3,4,5)
> x
[1] 1 2 3 4 5
```

- The same function can be used to concatenate already defined vectors

```
 > x<-c(1,2,3,4,5)
> y<-c(6,7)
> z<-c(10,x,y)
> z
[1] 10  1  2  3  4  5  6  7
```

Numerical Vectors

- Functions for numerical vectors:
  - **Length**:
    ```
    > x<-c(1,2,3,4,5)
    > length(x)
    [1] 5
    ```
  - **Min & Max**
    ```
    > min(x)
    [1] 1
    > max(x)
    [1] 5
    ```
  - **Sum & Product**
    ```
    > sum(x)
    [1] 15
    > prod(x)
    [1] 120
    ```

Numerical Vectors

- **Sort**

```
> x<-c(3,5,2,1,6)
> sort(x)
[1] 1 2 3 5 6
> sort(x,decreasing=T)
[1] 6 5 3 2 1
```

- **Rank**

```
> x<-c(3,5,2,1,6)
> rank(x)
[1] 3 4 2 1 5
```

Numerical Vectors

- **Rank & Ties**
  - *Average*: assigns each tied element the "average" rank

```
> x2<-c(3,5,2,1,6,3)
> rank(x2, ties.method="average")
[1] 3.5 5.0 2.0 1.0 6.0 3.5
```

  - *First*: lets the "earlier" entry "win", so the ranks are in numerical order

```
> rank(x2, ties.method="first")
[1] 3 5 2 1 6 4
```

  - *Random*: breaks ties randomly

```
> rank(x, ties.method="random")
[1] 4 5 2 1 6 3
> rank(x, ties.method="random")
[1] 3 5 2 1 6 4
```

  - *Min/Max*: assigns every tied element to the lowest/highest rank

```
> rank(x, ties.method="min")
[1] 3 5 2 1 6 3
> rank(x, ties.method="max")
[1] 4 5 2 1 6 4
```

Numerical Vectors

- **Order**

```
> x<-c(3,5,2,1,6)
> order(x)
[1] 4 3 1 2 5
```

(thus the smallest value is on the 4th position, the next one on the 3rd, etc).

Numerical Vectors

- **Assign names to vector members**

```
> weight<-c(70, 57, 68, 82)
> names(weight)
NULL
> names(weight)<-c("Mary", "Kelly", "Elena",
   "George")
> names(weight)
[1] "Mary"   "Kelly"  "Elena"  "George"
> weight
Mary  Kelly  Elena George
  70     57     68     82
```

Numerical Vectors

- **Missing Values**

```
> x3<-c(1,2,3,NA,9)
> x3
[1]  1  2  3 NA  9
> is.na(x3)
[1] FALSE FALSE FALSE  TRUE FALSE
```

Numerical Vectors

- **Colon Operator**
  - Using the syntax a:b you can generate a sequence of numbers from a to b with step 1 (or -1)

```
> x<-1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10
> x<--4:10
> x
[1] -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10
> x<-6:1
> x
[1] 6 5 4 3 2 1
> x<-3.3:10.3
> x
[1]  3.3  4.3  5.3  6.3  7.3  8.3  9.3 10.3
```

  - If (b-a) is not an integer, then R will stop before b

```
> x<-3.3:6.9
> x
[1] 3.3 4.3 5.3 6.3
```

Numerical Vectors

- If you want to use a different step then you should use the function seq(). You should define the first (from) & last (to) numbers of the sequence, the step, the length and the name of another vector (along) in order the generated sequence to have the same length as the specified vector. From all the above arguments you need to specify only 3; if the user specifies only 2 then by default R take the parameter by = 1.

```
> seq(from=1,to=9, by=2)
[1] 1 3 5 7 9
> seq(from=1,to=9, length=3)
[1] 1 5 9
> seq(to=9, length=3)
[1] 7 8 9
> seq(from=1,by=2,length=10)
[1]  1  3  5  7  9 11 13 15 17 19
> y<-1:10
> seq(from=1,by=2,along=y)
[1]  1  3  5  7  9 11 13 15 17 19
```

- If the ratio of the difference between the last and first number over the step is not an integer number then R will stop the sequence before the last term.

```
> seq(from=1,to=10,by=2)
[1] 1 3 5 7 9
```

Numerical Vectors

- **Replicate values or vectors.** With the function rep() you can replicate a value or a vector as many times as you want. The main arguments of the function are: (a) the value or the vector you wish to replicate and (b) the number of times to repeat the value or the whole vector (times) or the number of times to repeat each element of the vector (each).

```
> rep(2,5)
[1] 2 2 2 2 2
> x<-c(1,2,3)
> rep(x,5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
> rep(x,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Numerical Vectors

- **Arithmetic Operators & Vectors**. You can use the numeric operators between numerical vectors (they should be of the same length), or between scalars and vectors. Also the mathematical functions can be used in numerical vectors.

```
> x<-c(1,2,3)
> x*3
[1] 3 6 9
> x^2
[1] 1 4 9
> y<-c(4,5,6)
> y/x
[1] 4.0 2.5 2.0
```

Numerical Vectors

- By using [] you can easily choose specific elements of a vector. For example if you wish to extract the first element of the vector x you use the syntax x[1].

```
> x<-seq(from=1,to=9,by=2)
> x
[1] 1 3 5 7 9
> x[2]
[1] 3
> x[2:4]
[1] 3 5 7
> x[c(1,3)]
[1] 1 5
> x[-c(1,3)]
[1] 3 7 9
```

Character Vectors

```
> x<-c(“Statistics”, “Mathematics”)
> x
[1] “Statistics" “Mathematics"
```

Functions for Character Vectors

- character()

```
> character(length=2)
[1] "" ""
```

- as.character()

```
> x<-1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10
> as.character(x)
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

- as.numeric()

```
> x<-c("-0.1","2.7","B")
> as.numeric(x)
[1] -0.1  2.7   NA
Warning message:
NAs introduced by coercion
```

Functions for Character Vectors

- **is.character()**

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"  "6"
     "7"  "8"  "9"  "10"
> is.character(y)
[1] TRUE
> x<-c("-0.1","2.7","B")
> x
[1] "-0.1" "2.7"  "B"
> is.character(x)
[1] TRUE
> x<-as.numeric(x)
Warning message:
NAs introduced by coercion
> x
[1] -0.1  2.7   NA
> is.character(x)
[1] FALSE
```

- **noquote()**

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"
     "6"  "7"  "8"  "9"  "10"
> noquote(y)
[1] 1  2  3  4  5  6  7  8  9
     10
```

- **nchar()**

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"
     "6"  "7"  "8"  "9"  "10"
> nchar(y)
 [1] 1 1 1 1 1 1 1 1 1 2
```

Functions for Character Vectors

- paste()

```
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> paste(x)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"
     "9"  "10"
> paste(`Mathematical',`Statistics')
[1] "Mathematical Statistics"
> paste(`3',`5',`8', sep="+")
[1] "3+5+8"
> paste(paste(3,5, sep=` + '), 8, sep=`
     = ')
[1] "3 + 5 = 8"
> paste(`Chapter',2, sep=" ")
[1] "Chapter 2"
> paste("Today is", date())
[1] "Today is Mon Jun 03 20:42:41 2013"
> a<-c(`Kwstas', `Maria')
> b<-c(`Papadopoulos', `Kyriakou')
```

```
> paste(a,b)
[1] "Kwstas Papadopoulos" "Maria Kyriakou"
> paste(b,a, sep=`, ')
[1] "Papadopoulos, Kwstas" "Kyriakou, Maria"
> paste("Chapter", 1:2, sep=" ")
[1] "Chapter 1" "Chapter 2"
> a<-c(`Kwstas', `Maria')
> b<-c(`Papadopoulos', `Kyriakou', `Anagnostou')
> paste(a,b)
[1] "Kwstas Papadopoulos" "Maria Kyriakou"
     "Kwstas Anagnostou"
> a<-c(`Kwstas', `Maria')
> paste(a, collapse=",")
[1] "Kwstas,Maria"
> paste(1:10, collapse=`+')
[1] "1+2+3+4+5+6+7+8+9+10"
> b<-c(`Papadopoulos', `Kyriakou', `Anagnostou')
> paste(a, b, collapse=", ")
[1] "Kwstas Papadopoulos, Maria Kyriakou, Kwstas
Anagnostou"
```

Functions for Character Vectors

- strsplit()

```
> x<-c("Statistics", "Mathematics")
> strsplit(x,split="a")
[[1]]
[1] "St"      "tistics"
[[2]]
[1] "M"    "them" "tics"
> strsplit(x, split='')
[[1]]
 [1] "S" "t" "a" "t" "i" "s" "t" "i" "c" "s"
[[2]]
 [1] "M" "a" "t" "h" "e" "m" "a" "t" "i" "c"
        "s"
> strsplit(x, split="th")
[[1]]
[1] "Statistics"
[[2]]
[1] "Ma"      "ematics"
```

- substr()

```
> substr("abcdef",2,4)
[1] "bcd"
> x<-c("Statistics", "Mathematics")
> substr(x,2,4)
[1] "tat" "ath"
```

- grep()

```
> countries<-c("Greece", "United States",
"United Kingdom", "Italy",
"France", "United Arab Emirates")
> grep("United", countries)
[1] 2 3 6
> grep("United", countries, value=TRUE)
[1] "United States"        "United Kingdom"
"United Arab Emirates‟
> data[grep("United", data$country), ]
                   country        gdp  income continent
23  United Arab Emirates  21000   24200       AS
42        United Kingdom    28300   29400       EU
82         United States      35200   31200       NA
```

Functions for Character Vectors

- sub() & gsub()

```
> values<-c("1,700", "2,300")
> as.numeric(values)
[1] NA NA
Warning message:
NAs introduced by coercion
> as.numeric(gsub(",","",values))
[1] 1700 2300
> as.numeric(sub(",","",values))
[1] 1700 2300
> sub(",","",values)
[1] "1700" "2300"
> gsub(",","",values)
[1] "1700" "2300"
> values<-c("1,000,000", "2,000,000")
> sub(",","",values)
[1] "1000,000" "2000,000"
> gsub(",","",values)
[1] "1000000" "2000000"
```

- toupper () & tolower()

```
> x
[1] "Statistics"  "Mathematics"
> tolower(x)
[1] "statistics"  "mathematics"
> toupper(x)
[1] "STATISTICS"  "MATHEMATICS"
```

Logical Vectors

```
> logical(3)
[1] FALSE FALSE FALSE

> as.logical(c(0:10))
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
   TRUE  TRUE  TRUE  TRUE  TRUE
```

Factors

- Categorical Variables:

```
> gender<-c('Male', 'Female', 'Male', 'Male', 'Female')
> gender
[1] "Male"   "Female" "Male"   "Male"   "Female"
> factor(gender)
[1] Male   Female Male   Male   Female
Levels: Female Male
> levels(factor(gender))
[1] "Female" "Male"
```

- Ordinal Variables:

```
> opinion<-c('Low', 'Low', 'High', 'High', 'High', 'Medium')
> ordered(opinion, levels=c('Low', 'Medium', 'High'))
[1] Low    Low    High   High   High   Medium
Levels: Low < Medium < High
```

Matrices

```
> x<-1:10
> X<-matrix(x, ncol=2)
> X
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> X<-matrix(x, nrow=5)
```

```
> X
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> X<-matrix(x, nrow=5, byrow=T)
> X
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

Matrices

- Dimension

```
> dim(X)
[1] 5 2
```

- Specific elements

```
> X[3,2]
[1] 6
```

- Specific rows or columns

```
> X[3,]
[1] 5 6
> X[,2]
[1]  2  4  6  8 10
```

Matrices

- Create Matrices by combining vectors (cbind & rbind).

```
> x1<-1:5
> x2<-6:10
> cbind(x1,x2)
     x1 x2
[1,]  1  6
[2,]  2  7
[3,]  3  8
[4,]  4  9
[5,]  5 10
> rbind(x1,x2)
     [,1] [,2] [,3] [,4] [,5]
x1    1    2    3    4    5
x2    6    7    8    9   10
```

Matrices

- Diagonal Matrices

```
> diag(1:5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5
```

Matrices

- Identical Matrix

```
> diag(5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

Matrices

- **Arithmetic Operators & Matrices**. You can use the numeric operators between matrices (be careful with the dimensions) or between vectors & matrices (be careful with the dimensions) or between scalars & matrices. Also the mathematical functions can be used in matrices.

Matrices

Operator	Description
%*%	Multiplication of Matrices
t()	Transpose of a Matrix
solve()	Inverse of a Matrix

Matrices

```
> x<-matrix(c(1,2,3,4,5,6), ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(x)
[1] 3 2
> y<-matrix(c(0,1,1,1), ncol=2)
> y
     [,1] [,2]
[1,]    0    1
[2,]    1    1
```

```
> x%*%y
     [,1] [,2]
[1,]    4    5
[2,]    5    7
[3,]    6    9
> t(x)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> solve(y)
     [,1] [,2]
[1,]   -1    1
[2,]    1    0
```

Matrices

- The function apply().

```
> x<-matrix(c(1,2,3,4,5,6), ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(x,1,sum)
[1] 5 7 9
> apply(x,2,sum)
[1]  6 15
```

Compute row sum.

Compute column sum.

Function apply()

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

X is an array

MARGIN is an integer vector indicating which margins should be “retained”.

FUN is a function to be applied

... is for other arguments to be passed to FUN

Function apply()

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
[1] 0.04868268 0.35743615 -0.09104379
[4] -0.05381370 -0.16552070 -0.18192493
[7] 0.10285727 0.36519270 0.14898850
[10] 0.26767260
> apply(x, 1, sum)
[1] -1.94843314 2.60601195 1.51772391
[4] -2.80386816 3.73728682 -1.69371360
[7] 0.02359932 3.91874808 -2.39902859
[10] 0.48685925 -1.77576824 -3.34016277
[13] 4.04101009 0.46515429 1.83687755
[16] 4.36744690 2.21993789 2.60983764
[19] -1.48607630 3.58709251
```

$\rightarrow$ simulates from standard normal (more later)

col/row sums & means

- For sums and means of matrix dimensions, we have some shortcuts.
  - rowSums (x) = apply(x, 1, sum)
  - rowMeans (x) = apply(x, 1, mean)
  - colSums (x) = apply(x, 2, sum)
  - colMeans (x) = apply(x, 2, mean)
- The shortcut functions are much faster, but you won’t notice unless you’re using a large matrix.

Function apply()

- Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
           [,1]        [,2]       [,3]        [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686
75% 0.9258157 0.07065724 0.3050407 -0.06585436
           [,5]        [,6]       [,7]        [,8]
25% -0.05999553 -0.6588380 -0.653250 0.01749997
75% 0.52928743 0.3727449 1.255089 0.72318419
           [,9]        [,10]      [,11]       [,12]
25% -1.2467955 -0.8378429 -1.0488430 -0.7054902
75% 0.3352377 0.7297176 0.3113434 0.4581150
           [,13]       [,14]      [,15]       [,16]
25% -0.1895108 -0.5729407 -0.5968578 -0.9517069
75% 0.5326299 0.5064267 0.4933852 0.8868922
           [,17]       [,18]      [,19]       [,20]
25% -0.2502935 -0.7488003 -0.7190923 -0.638243
……………………………………………………………………………………..
```

Arrays

- Arrays are matrices with 3 or more dimensions. To create them we use the function array(). The dimension of the array is given by the parameter dim. For example if dim=c(2,3,4), we will have a 3 dimensional array of dimension $2\times3\times4$.

Arrays

```
> X<-
     array(c(1:12,36:48),dim=c(
     2,3,4))
> X
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   36   38   40
[2,]   37   39   41

, , 4

     [,1] [,2] [,3]
[1,]   42   44   46
[2,]   43   45   47
```

Function apply()

- Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
> rowMeans(a, dims = 2)
           [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
```

Data Frames

- Data frames are used for storing data tables. It is a list of vectors (not necessary of the same type) of equal length. In a data frame usually we store the observations we have from a sample.
- To create a data frame we use the function data.frame().

Data Frames

```
> Gender<-c('Male', 'Male', 'Male', 'Female')
> Gender<-factor(Gender)
> Gender
[1] Male   Male   Male   Female
Levels: Female Male
> Smoking<-c(T, T, F, F)
> Smoking<-factor(Smoking)
> Choresterol<-c(200, 220, 180, 172)
> Choresterol
[1] 200 220 180 172
> sample<-data.frame(Gender, Smoking, Choresterol)
> sample
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
```

Data Frames

- With the function as.data.frame() you can convert an R object into a data frame. With the parameter row.names () you can define names for the rows (observations) of the data frame. With the function names() you can give names into the columns (variables) of the data frame.

Data Frames

```
> x<-matrix(c(1,1,200,1,1,220, 1,0,180,0,0,172),
        ncol=3, byrow=T)
> x
     [,1] [,2] [,3]
[1,]    1    1  200
[2,]    1    1  220
[3,]    1    0  180
[4,]    0    0  172
> x<-as.data.frame(x)
> x
  V1 V2  V3
1  1  1 200
2  1  1 220
3  1  0 180
4  0  0 172
> names(x)
[1] "V1" "V2" "V3"
> names(x)<-c('Gender', 'Smoking', 'Choresterol')
> x
  Gender Smoking Choresterol
1      1       1         200
2      1       1         220
3      1       0         180
4      0       0         172

> x<-data.frame(x, row.names=c('obs1',
       'obs2', 'obs3', 'obs4') )
> x
     Gender Smoking Choresterol
obs1      1       1         200
obs2      1       1         220
obs3      1       0         180
obs4      0       0         172
```

Data Frames

- Similar functions used for matrices can be also used here.

```
> x
     Gender Smoking
      Choresterol
obs1      1        1          200
obs2      1        1          220
obs3      1        0          180
obs4      0        0          172
> dim(x)
[1] 4 3
> x[1,]
       Gender Smoking
      Choresterol
obs1      1        1          200
```

```
> x[1,2]
[1] 1
> x$Gender
[1] 1 1 1 0
> rbind(1,x)
    Gender Smoking Choresterol
1        1       1           1
obs1      1       1          200
obs2      1       1          220
obs3      1       0          180
obs4      0       0          172
> cbind(1,x)
         1 Gender Smoking Choresterol
obs1 1      1       1          200
obs2 1      1       1          220
obs3 1      1       0          180
obs4 1      0       0          172
```

Lists

- Lists are vectors with elements not necessarily of the same type or length. To create a list you can use the function list() giving as a main parameter the objects (members) you wish to be contained in your list, together with their names.

Lists

```
> Gender<-c('Male', 'Male', 'Male',
      'Female')
> Gender<-factor(Gender)
> Gender
[1] Male   Male   Male   Female
Levels: Female Male
> x<-1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> sample
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> y<-list(my_sample=sample, x=x,
      the_gender=Gender)
```

```
> y
$my_sample
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172

$x
 [1]  1  2  3  4  5  6  7  8  9 10

$the_gender
[1] Male   Male   Male   Female
Levels: Female Male
```

Lists

- With the symbol $ or [[ ]], we can see a member of the list.

```
> y$x
[1]  1  2  3  4  5  6  7  8  9 10
> y[[3]]
[1] Male   Male   Male   Female
Levels: Female Male
> y$x[1:3]
[1] 1 2 3
```

Reading Data

- There are a few principal functions reading data into R.
  - **read.table**, **read.csv**, for reading tabular data
  - **readLines**, for reading lines of a text file
  - **source**, for reading in R code files (inverse of dump)
  - **dget**, for reading in R code files (inverse of **dput**)
  - **load**, for reading in saved workspaces

Reading & Writing Data

- There are analogous functions for writing data to files
  - **write.table**
  - **writeLines**
  - **dump**
  - **dput**
  - **save**

The function read.table()

- The **read.table** function is one of the most commonly used functions for reading data. It has a few important arguments:
  - file: the name of a file, or a connection
  - header: logical indicating if the file has a header line
  - sep: a string indicating how the columns are separated
  - colClasses: a character vector indicating the class of each column in the dataset
  - nrows: the number of rows in the dataset
  - comment.char: a character string indicating the comment character
  - skip: the number of lines to skip from the beginning
  - stringsAsFactors: should character variables be coded as factors?

Reading & Writing Data

```
> Gender<-c("Male", "Male", "Male",
     "Female")
> write(Gender,file="g.txt", ncol=4)
> x
[1]  1  2  3  4  5  6  7  8  9 10
> write(x,file="x.txt", ncol=length(x))
> Smoking
  [1]  TRUE  TRUE FALSE FALSE
> write(Smoking, file="smoking.txt",
     ncol=4)
> X
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
[4,]    4   10
[5,]    5   11
[6,]    6   12
```

```
> write(t(X), "X.txt", ncol=2)
> sample
  Gender Smoking Cholesterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> write.table(sample, file="sample.txt")
> x<-scan("x.txt")
Read 10 items
> x
[1]  1  2  3  4  5  6  7  8  9 10
> X<-matrix(scan("XX.txt"), ncol=2, byrow=T)
Read 12 items
> X
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
[6,]   11   12
```

Reading & Writing Data

```
> zz<-read.table("sample.txt",
    header=T)
> zz
  Gender Smoking Cholesterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> zz<-read.table("sample.txt",
    header=T)
> zz
  Gender Smoking Cholesterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
```

```
> sapply(zz,mode)
     Gender     Smoking Cholesterol
  "numeric"   "logical"   "numeric"
> zz<-read.table("sample.txt",
    header=T, stringsAsFactors =
    FALSE)
> zz
  Gender Smoking Cholesterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> sapply(zz,mode)
     Gender     Smoking Cholesterol
"character"   "logical"   "numeric"
```

The function read.table()

- For small to moderately sized datasets, you can usually call **read.table** without specifying any other arguments

```
data <- read.table("foo.txt")
```

  R will automatically
  - skip lines that begin with a #
  - figure out how many rows there are (and how much memory needs to be allocated)
  - figure what type of variable is in each column of the table
- Telling R all these things directly makes R run faster and more efficiently.
- **read.csv** is identical to **read.table** except that the default separator is a comma.

The function read.table()

- With much larger datasets, doing the following things will make your life easier and will prevent R from choking.
  - Read the help page for **read.table**, which contains many hints.
  - Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
  - Set comment.char = "" if there are no commented lines in your file.

The function read.table()

- Use the colClasses argument. Specifying this option instead of using the default can make **read.table** run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are “numeric”, for example, then you can just set colClasses = "numeric". A quick an dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt", colClasses =
   classes)
```

- Set nrows. This doesn’t make R run faster but it helps with memory usage. A mild overestimate is okay.

Textual Format

- **dumping and dputing** are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or csv file, **dump and dput** preserve the metadata (sacrificing some readability), so that another user doesn't have to specify it all over again.
- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files.
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem.
- Textual formats adhere to the "Unix philosophy".
- Downside: The format is not very space-efficient.

dput() & dget()

- Another way to pass data around is by deparsing the R object with **dput** and reading it back in using dget.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)
structure(list(a = 1,
b = structure(1L, .Label = "a",
class = "factor")),
.Names = c("a", "b"), row.names = c(NA, -1L),
class = "data.frame")
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
  a b
1 1 a
```

dump()

- Multiple objects can be deparsed using the **dump** function and read back in using source.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
a b
1 1 a
> x
[1] "foo"
```

Package “foreign”

SPSS	read.spss()
S	read.S()
STATA	read.dta()
SAS	read.xport()
Epi Info	read.epiinfo()
Minitab	read.mtb()
Octave	read.octave()

Example: From Excel to R

- Suppose the first row contains variable names and we have data for 6 columns (variables). One of the variables is “value” where the comma is used as a thousand separator. Missing values are denoted by “:”.
- Convert data.xls to data.csv
  - Menu File, Save As… and choose csv as type.
  - Open data.csv with WordPad and replace all “;” with “,”.
- Run R from the same directory where data.csv is.
- Type in R

```
> data<-read.table(file="data.csv",header=TRUE,sep=",",
colClasses = c(rep("character",6)),na.strings=c(":"))
> data$value<-sub(",","",area$value)
> class(area$value)<-"numeric"
```

Example. From R to Excel

```
> write.csv(data, file="data.csv")
```

- Convert data.csv to data.xls
  - Open file and with the mouse choose first column
  - From menu Data choose Text to Columns…..
  - Click on "Delimited", and in the next window choose "Comma" as a delimiter.
  - In the next window click on Finish and from the menu File save the file (Save As…) as .xls.

Functions in R

```
> x
 [1]  46 104  94 114  35  70 120  29  19 135
   200 222  89 100  55 214  15  81 118 193
> range<-function(x){
  y<-max(x)-min(x)
  return(y)
  }
> range(x)
[1] 207
```

Functions in R

```
> calc<-function(a,b=2){
    y<-a^b
    return(y)
   }
> calc(4)
[1] 16
> calc(4,3)
[1] 64
```

b=2 default value.

α=4, b=3

Functions in R

Syntax	Description
if (A) B	Check if A is satisfied. If yes perform B
if (A) B1 else B2	Check if A is satisfied. If yes perform B1 else B2
ifelse(A, B1, B2)	Same as previous
break	Stops current loop
next	Stops current loop and starts next iteration
return(A)	Terminates a function and return A
while(A) B	repetitive execution of B is to be carried out till it meets the constraint condition (not A)
repeat A	Same as while
for(index in A) B	Performs B as long as index belongs to A

if else statement in R

```
if(A)
 {
  A1
 } else
 {
  A2
 }
```

$$h(x) = \begin{cases} x^2 & , x \le 0.05 \\ 0.25 & , x > 0.05 \end{cases}$$

```
> x<-0.10
> if(x<=0.05)
   {
    h<-x^2
   } else
   {
    h<-0.25
   }
```

if else statement in R

```
if(A)
 {
  B1
 } else if(C)
 {
  B2
 } else
 {
  B3
 }
```

$$h(x) = \begin{cases} x^2, & x \le 0.05 \\ 0.25, & 0.05 < x \le 1 \\ 1, & x > 1 \end{cases}$$

```
> x<-0.10
> if(x<=0.05)
   {
    h<-x^2
   } else if(x>0.25 & x<=1)
   {
    h<-0.25
   } else
   {
    h<-1
   }
```

for loop

for(index in A) B

```
> x<-c(3,6,2,7)
> n<-length(x)
> proda<-1
> summ<-0
> for(i in 1:n)
  {
   summ<-summ+x[i]
   proda<-proda*x[i]
  }
```

for loop

```
> A<-
    matrix(1:1000^2,ncol=1000,
    nrow=1000)
> summm<-0
> for(i in 1:1000)
  {
   for(j in 1:1000)
    {
     summ<-summ+A[i,j]
    }
  }
```

```
> system.time({summ<-0; for(i in
    1:1000){for(j in 1:1000){summ<-
    summ+A[i,j]}}})
  user  system elapsed
  1.94    0.00    1.96
>system.time({sum(as.numeric(apply(A,
    1,sum)))})
  user  system elapsed
  0.05    0.00    0.36
```

User cpu time

System cpu time

Real elapsed time

while & repeat

while(A) B

repeat(B; if(A) break)

Suppose we wish to apply Newton-Raphson method in order to find the solution of the equation $f(x) = x^3 + 2x^2 - 7 = 0$

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

$$\boxed{f'(x) = 3x^2 + 4x}$$

while & repeat

```
> x<-1
> tolerance<-0.000001
> f<-x^3+2*x^2-7
> f.prime<-3*x^2+4*x
> while(abs(f)>tolerance)
 {
  x<-x-f/f.prime
  f<-x^3+2*x^2-7
  f.prime<-3*x^2+4*x
 }
```

```
> x<-1
> tolerance<-0.000001
> f<-x^3+2*x^2-7
> f.prime<-3*x^2+4*x
> repeat
  {
   x<-x-f/f.prime
   f<-x^3+2*x^2-7
   f.prime<-3*x^2+4*x
   if(abs(f) <= tolerance) break
  }
```

Functions in R

- Example: Suppose we wish to create our own function to calculate x! where x is a natural number

```
fact1<-function(x){
y<-floor(x)
if (y!=x | x<0)
print("Your number is not natural")
else
{
f<-1
if (x<2) return(f)
for (i in 2:x) {
f<-f*i
}
 return(f)
}
}
```

```
> fact1(3)
[1] 6
> fact1(1)
[1] 1
> fact1(0)
[1] 1
> fact1(4)
[1] 24
> fact1(2.3)
[1] "Your number is not natural"
```

Functions in R

```
fact2<-function(x){
    y<-floor(x)
    if (y!=x | x<0)
    print("Your number is not natural")
    else
    {
    f<-1
    t<-x
    while(t>1){
    f<-f*t
    t<-t-1
    }
    return(f)
    }
    }
```

```
> fact2(3)
[1] 6
> fact2(1)
[1] 1
> fact2(0)
[1] 1
> fact2(4)
[1] 24
> fact2(2.3)
[1] "Your number is not natural"
```

Functions in R

```
fact3<-function(x){
      y<-floor(x)
      if (y!=x | x<0)
      print("Your number is not natural")
     else
      {
     f<-1
     t<-x
     repeat{
     if (t<2) break
     f<-f*t
     t<-t-1
      }
     return(f)
     }
     }
```

```
> fact3(3)
[1] 6
> fact3(1)
[1] 1
> fact3(0)
[1] 1
> fact3(4)
[1] 24
> fact3(2.3)
[1] "Your number is not natural"
```

Functions in R

- Loops in R could be time consuming. Therefore the following loop

```
for(i in 1:length(y)) {if(y[i]<0} y[i]<-0}
```

could be replaced by

```
y[y<0]<-0
```

Functions in R

- Additionally we could use the cumprod() function of R, that calculates the cumulative product, i.e.

```
> cumprod(c(1,2,4))
[1] 1 2 8
```

- Thus we could use the following function instead

```
fact4<-function(x){
      y<-floor(x)
      if (y!=x|x<0)
      print("Your number is not natural")
     else
      {
      return(max(cumprod(1:x)))
      }
      }
```

```
> fact4(3)
[1] 6
> fact4(1)
[1] 1
> fact4(0)
[1] 1
> fact4(4)
[1] 24
> fact4(2.3)
[1] "Your number is not natural"
```

Functions in R

- Finally we could use the gamma() function, since for natural numbers we have $x!=\Gamma(x+1)$ or of course the ready function factorial().

```
> gamma(4)
[1] 6
> gamma(2)
[1] 1
> gamma(1)
[1] 1
> factorial(3)
[1] 6
> factorial(1)
[1] 1
> factorial(0)
[1] 1
```

- Please note that the above functions give answers not necessarily for natural numbers.

Functions in R

- Overflow Problems:

$\binom{200}{100}$

```
> factorial(200)/(factorial(100)*factorial(100))
[1] NaN
Warning message:
In factorial(200) : value out of range in 'gammafn'
```

Functions in R

We have

$$\binom{200}{100} = \frac{101 \cdot \ldots \cdot 200}{1 \cdot \ldots \cdot 100}$$

Thus in R

```
> prod(101:200)/prod(1:100)
[1] 9.054851e+58
```

$\longrightarrow 9.05 \cdot 10^{58}$

or even better

```
> x<-1:100
> y<-101:200
> z<-y/x
> prod(z)
[1] 9.054851e+58
```

Functions in R

- Another way to avoid overflow problems is by using logs.

$$\binom{200}{100} = \exp\left[\log\left\{\binom{200}{100}\right\}\right] = \exp[\log(200!) - 2\log(100!)]$$

But

$$\log(n!) = \sum_{i=1}^{n} \log(i)$$

```
> exp(sum(log(1:200))-2*sum(log(1:100)))
[1] 9.054851e+58
```

Loop Functions

- Writing for & while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.
  - **lapply**: Loop over a list and evaluate a function on each element.
  - **sapply**: Same as lapply but try to simplify the result.
  - **apply**: Apply a function over the margins of an array.
  - **tapply**: Apply a function over subsets of a vector.
  - **mapply**: Multivariate version of lapply.

lapply()

- **lapply** takes three arguments: a list X, a function (or the name of a function) FUN, and other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list.

```
> lapply
function (X, FUN, ...)
{
FUN <- match.fun(FUN)
if (!is.vector(X) || is.object(X))
X <- as.list(X)
.Internal(lapply(X, FUN))
}
```

The actual looping is done internally in C code.

lapply()

- **lapply** always returns a list, regardless of the class of the input.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3
$b
[1] 0.0296824
```

lapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1),
               d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5
$b
[1] 0.06082667
$c
[1] 1.467083
$d
[1] 5.074749
```

sapply()

- **sapply** will try to simplify the result of **lapply** if possible.
- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned.

sapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1),
                d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5
$b
[1] 0.06082667
$c
[1] 1.467083
$d
[1] 5.074749
```

sapply()

```
> sapply(x, mean)
a b c d
2.50000000 0.06082667 1.46708277
    5.07474950
> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric
    or logical: returning NA
```

tapply()

- **tapply** is used to apply a function over subsets of a vector.

  ```
  > str(tapply)
  function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
  ```

  - X is a vector.
  - INDEX is a factor or a list of factors (or else they are coerced to factors).
  - FUN is a function to be applied.
  - ... contains other arguments to be passed FUN.
  - simplify, should we simplify the result?

tapply()

- Take group means.

simulates from uniform in (0,1) (more later)

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
1 2 3
0.1144464 0.5163468 1.2463678
```

Generate factors by specifying the pattern of their levels.

tapply()

- Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1144464
$`2`
[1] 0.5163468
$`3`
[1] 1.246368
```

tapply()

- Find group ranges.

```
> tapply(x, f, range)
$`1`
[1] -1.097309 2.694970
$`2`
[1] 0.09479023 0.79107293
$`3`
[1] 0.4717443 2.5887025
```

split()

- **split** takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
> str(split)
function (x, f, drop = FALSE, ...)
```

  - x is a vector (or list) or data frame.
  - f is a factor (or coerced to one) or a list of factors.
  - drop indicates whether empty factors levels should be dropped.

split()

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$‘1‘
[1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
[5] 0.2849881 0.9383361 -1.0973089 2.6949703
[9] 1.5976789 -0.1321970
$‘2‘
[1] 0.09479023 0.79107293 0.45857419 0.74849293
[5] 0.34936491 0.35842084 0.78541705 0.57732081
[9] 0.46817559 0.53183823
$‘3‘
[1] 0.6795651 0.9293171 1.0318103 0.4717443
…………………………………………………………………………………………
```

split()

- A common idiom is **split** followed by an **lapply**.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.1144464
$`2`
[1] 0.5163468
$`3`
[1] 1.246368
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
Ozone Solar.R Wind Temp Month Day
1 41 190 7.4 67 5 1
2 36 118 8.0 72 5 2
3 12 149 12.6 74 5 3
4 18 313 11.5 62 5 4
5 NA NA 14.3 56 5 5
6 28 NA 14.9 66 5 6
```

Splitting a Data Frame

```
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R",
    "Wind")]))
$`5`
Ozone Solar.R Wind
NA NA 11.62258
$`6`
Ozone Solar.R Wind
NA 190.16667 10.26667
$`7`
Ozone Solar.R Wind
NA 216.483871 8.941935
…………………………………………………….
```

Splitting a Data Frame

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R",
    "Wind")]))
5 6 7 8 9
Ozone NA NA NA NA NA
Solar.R NA 190.16667 216.483871 NA 167.4333
Wind 11.62258 10.26667 8.941935 8.793548 10.1800

> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R",
    "Wind")],
na.rm = TRUE))
5 6 7 8 9
Ozone 23.61538 29.44444 59.115385 59.961538 31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind 11.62258 10.26667 8.941935 8.793548 10.18000
```

Splitting on more that one level

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
[1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
[1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1, f2)
[1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4  2.4 1.5 2.5
```

$$\begin{tikzcd}
1 \arrow[d, leftrightarrow] & 1 \arrow[d, leftrightarrow] & 1 \arrow[d, leftrightarrow] & 1 \arrow[d, leftrightarrow] & 1 \arrow[d, leftrightarrow] & 2 \arrow[d, leftrightarrow] & 2 \arrow[d, leftrightarrow] & 2 \arrow[d, leftrightarrow] & 2 \arrow[d, leftrightarrow] & 2 \arrow[d, leftrightarrow] \\
1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 5
\end{tikzcd}$$

Empty levels

Splitting on more that one level

- Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))
List of 10
$ 1.1: num [1:2] -0.378 0.445
$ 2.1: num(0)
$ 1.2: num [1:2] 1.4066 0.0166
$ 2.2: num(0)
$ 1.3: num -0.355
$ 2.3: num 0.315
$ 1.4: num(0)
$ 2.4: num [1:2] -0.907 0.723
$ 1.5: num(0)
$ 2.5: num [1:2] 0.732 0.360
```

Splitting on more that one level

- Empty levels can be dropped.

```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
$ 1.1: num [1:2] -0.378 0.445
$ 1.2: num [1:2] 1.4066 0.0166
$ 1.3: num -0.355
$ 2.3: num 0.315
$ 2.4: num [1:2] -0.907 0.723
$ 2.5: num [1:2] 0.732 0.360
```

mapply()

- **mapply** is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY =
    TRUE, USE.NAMES = TRUE)
```

  - FUN is a function to apply
  - ... contains arguments to apply over
  - MoreArgs is a list of other arguments to FUN.
  - SIMPLIFY indicates whether the result should be simplified

mapply()

The following is tedious to type

```
> list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1
[[2]]
[1] 2 2 2
[[3]]
[1] 3 3
[[4]]
[1] 4
```

Vectorizing a Function

```
> noise <- function(n, mean, sd) {
+ rnorm(n, mean, sd)
+ }
> noise(5, 1, 2)
[1] 2.4831198 2.4790100 0.4855190 -
   1.2117759
[5] -0.2743532
> noise(1:5, 1:5, 2)
[1] -4.2128648 -0.3989266 4.2507057
   1.1572738
[5] 3.7413584
```

Vectorizing a Function

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658
[[2]]
[1] 0.7113482 2.7555797
[[3]]
[1] 2.769527 1.643568 4.597882
[[4]]
[1] 4.476741 5.658653 3.962813 1.204284
[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

The above is the same as:

```
> list(noise(1, 1, 2), noise(2, 2, 2), noise(3, 3, 2), noise(4, 4,
    2), noise(5, 5, 2))
```

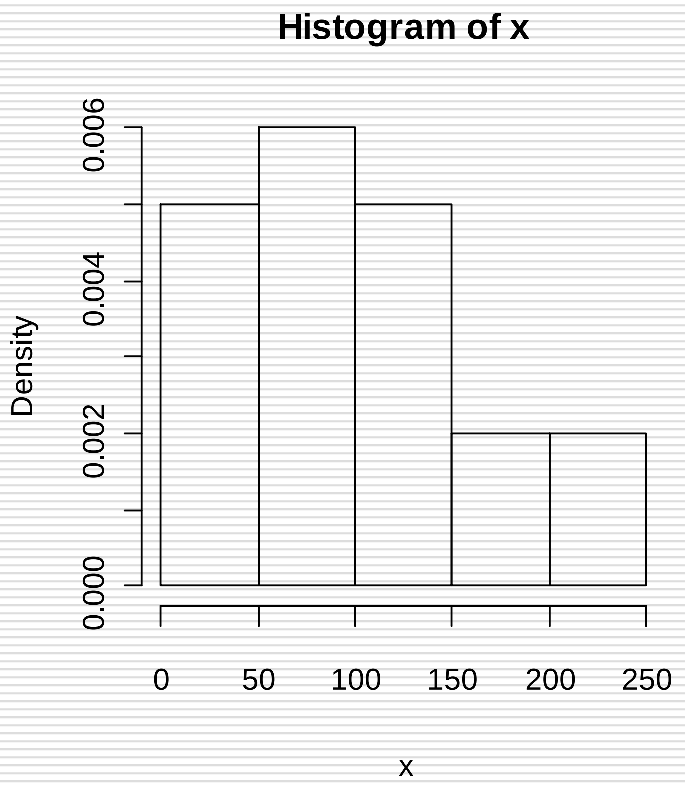
Numerical Measures – Quantitative Variables

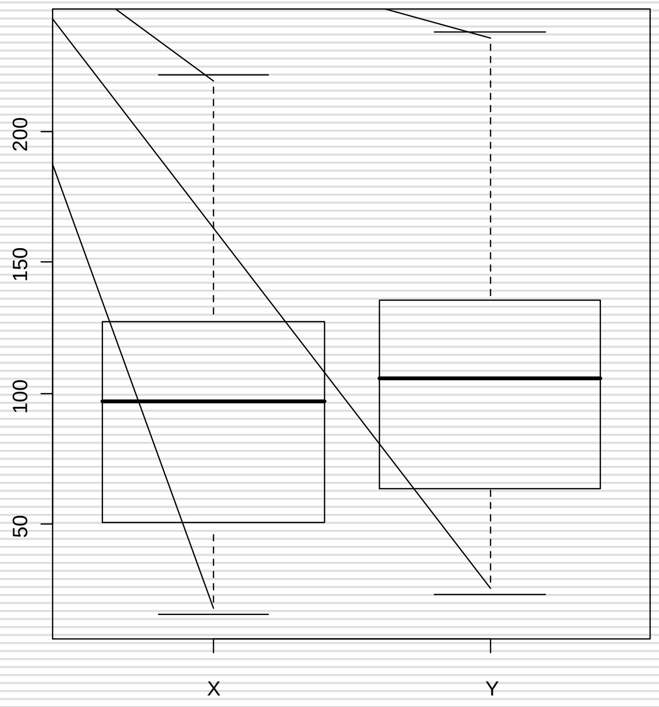
Function	**Description**
mean(x)	Sample Mean
min(x)	Minimum
max(x)	Maximum
median(x)	Median
var(x)	Variance
sd(x)	Standard Deviation
quantile(x,p)	Returns the p percentile. Για p=0.25 και p=0.75 we get the 1^o and 3^o quartile. Read the help in R for the different quantile algorithms (argument “types”)

Graphical Methods – Quantitative Variables

- Histogram:
  - hist(x)
  - hist(x, nclass=10)
  - hist(x, breaks=seq(from=0,to=240,by=30))
  - hist(x, probability=T)
- Boxplot:
  - boxplot(x)
  - boxplot(x,y, names=c("X", "Y"))

Graphical Methods – Quantitative Variables





Descriptive Measures - Categorical Variables

car=C,

metro=M,

bus=B

walk=F

males	C	C	B	M	M
	C	M	M	F	C
females	F	B	B	M	M
	C	C	C	M	C

Descriptive Measures - Categorical Variables

```
> Transportation<-c("C", "C", "B", "M", "M", "C", "M", "M", "F", "C",
     "F", "B", "B", "M", "M", "C", "C", "C", "M", "C")
> Transportation<-factor(Transportation)
> Gender<-c(rep("M",10), rep("F", 10))
> Gender
 [1] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "F" "F" "F" "F" "F" "F"
     "F" "F" "F" "F"
> Gender<-factor(Gender)
> table(Transportation)
A
B C F M
3 8 2 7
> prop.table(table(Transportation))
A
   B    C    F    M
0.15 0.40 0.10 0.35
```

Descriptive Measures - Categorical Variables

```
> mytable<-
      table(Transportation,Gender)
> mytable
   Gender
Transportation   F M
          B      2 1
          C      4 4
          F       1 1
          M       3 4
> margin.table(mytable, 1)
Trasportation
B C F M
3 8 2 7
> margin.table(mytable, 2)
Gender
 F  M
10 10
```

frequencies for transportation

frequencies for gender

```
> prop.table(mytable)
                   Gender
Transportation     F    M
  B              0.10 0.05
  C              0.20 0.20
  F              0.05 0.05
  M               0.15 0.20
```

relative frequencies for each cell

```
> prop.table(mytable, 1)
                  Gender
Transportation   F         M
  B          0.6666667 0.3333333
  C           0.5000000 0.5000000
  F           0.5000000 0.5000000
  M           0.4285714 0.5714286
```

relative frequencies rows

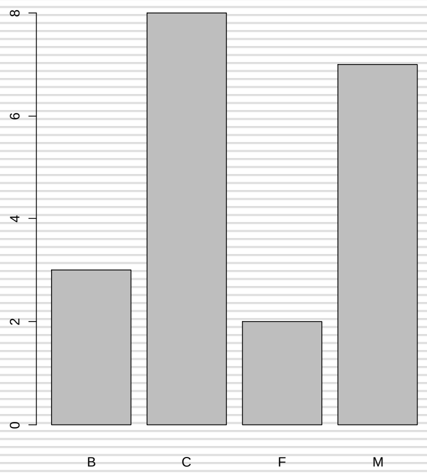
```
> prop.table(mytable, 2)
                  Gender
Transportation    F   M
  B              0.2 0.1
  C              0.4 0.4
  F               0.1 0.1
  M              0.3 0.4
```

relative frequency columns

Descriptive Measures - Categorical Variables

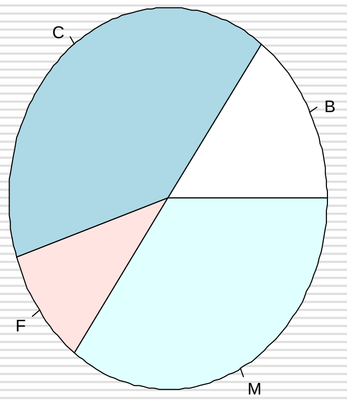
- Barplot

```
> AA<-table(A)
> AA
A
B C F M
3 8 2 7
> barplot(AA)
```



- Piechart

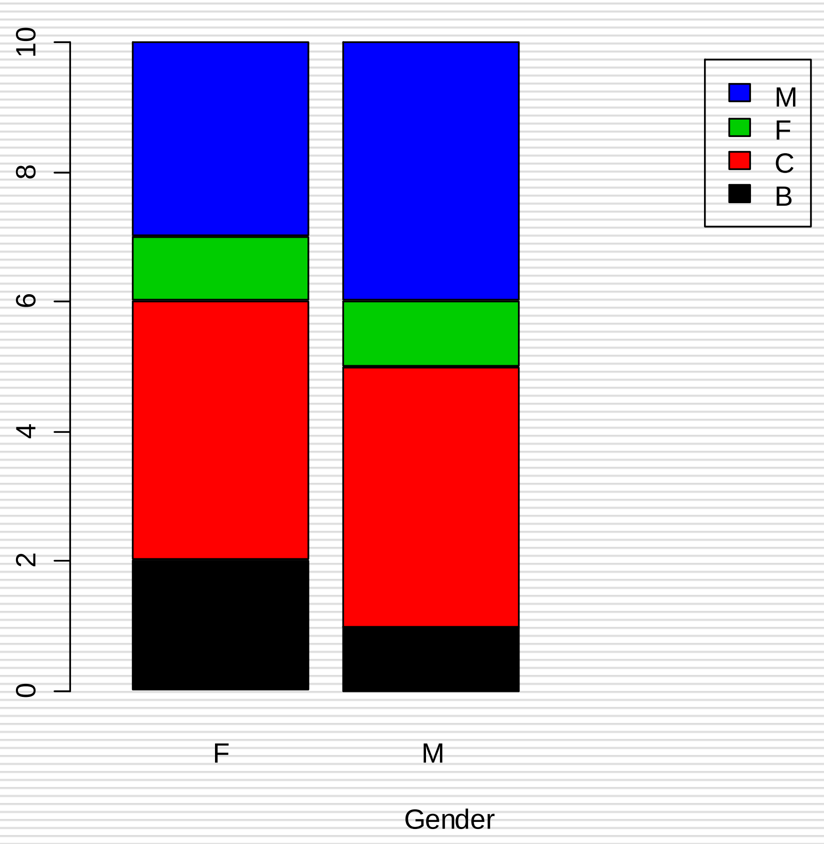
```
> pie(AA)
```

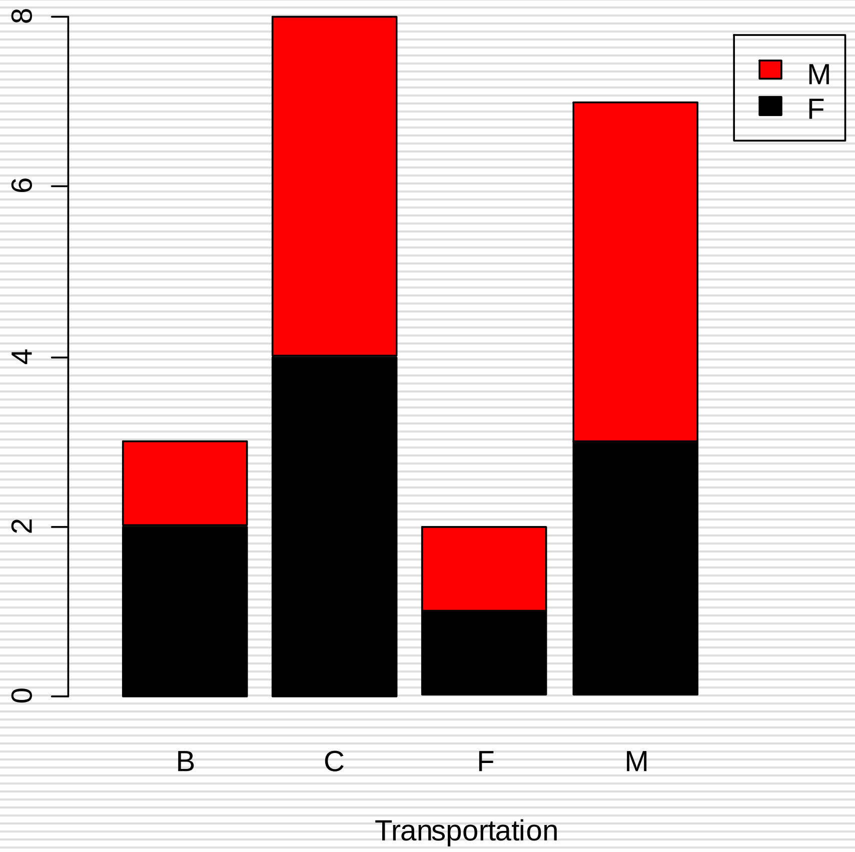


Stacked Barplots

```
> freq_table<-table(Transportation,Gender)
> barplot(freq_table, xlim=c(0,3),
   xlab="Gender",
   legend=levels(Transportation), col=1:4)
> freq_table<-table(Gender, Transportation)
> barplot(freq_table, width=0.85, xlim=c(0,5),
   xlab="Transportation",
   legend=levels(Gender), col=1:2)
```

Stacked Barplots

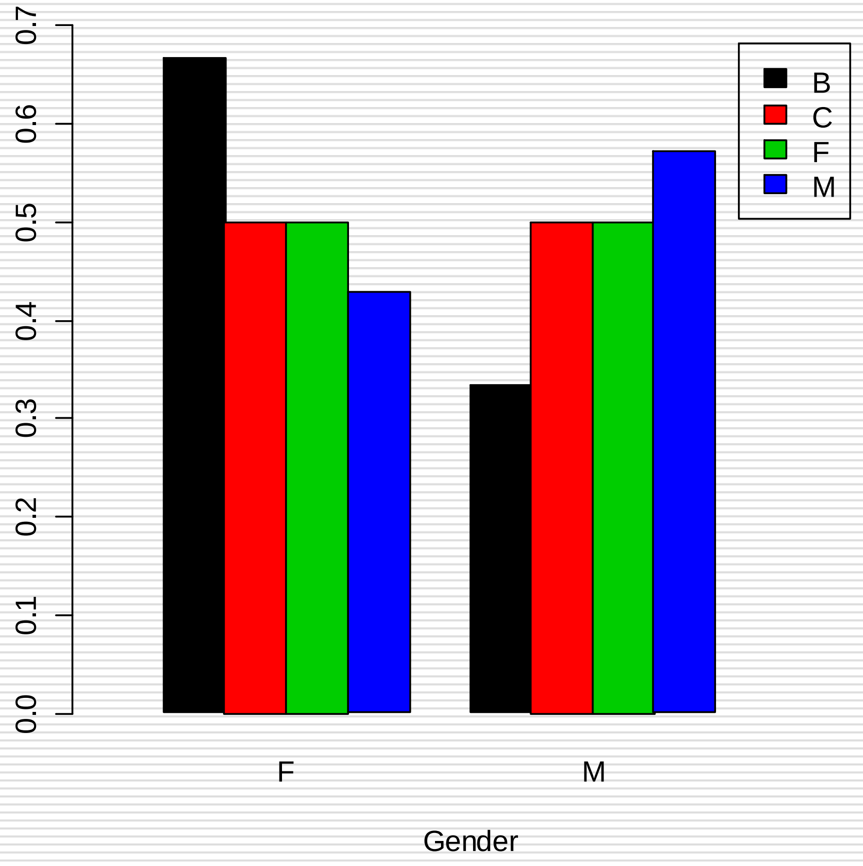


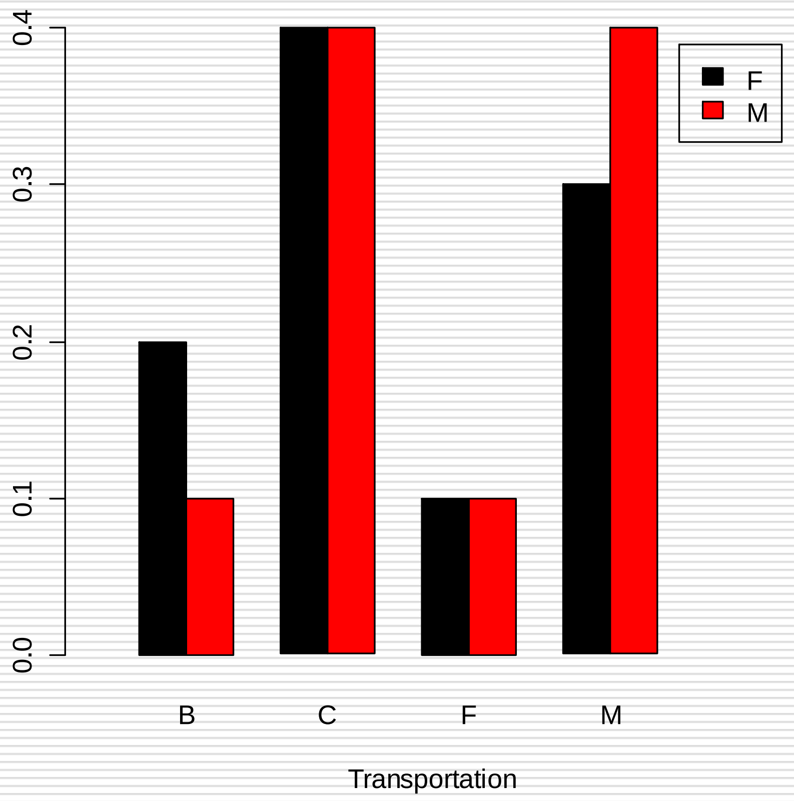


Grouped Barplot

```
> freq_table<-table(Transportation,Gender)
> barplot(prop.table(freq_table,1), width=0.25,
   xlim=c(0,3), ylim=c(0,0.7), xlab="Gender",
   legend=levels(Transportation), beside=T, col=1:4)
> freq_table<-table(Gender, Transportation)
> barplot(prop.table(freq_table,1), width=0.25,
   xlim=c(0,3.6), xlab="Transportation",
   legend=levels(Gender), beside=T, col=1:2)
```

Grouped Barplot





Distributions in R

- Probability distribution functions usually have four functions associated with them.
- The functions are prefixed with a:
  - **d** for density.
  - **r** for random number generation.
  - **p** for cumulative distribution.
  - **q** for quantile function.

Distributions in R

command	**distribution**	**command**	**distribution**
beta	Beta	hyper	Hypergeometric
norm	Normal	unif	Uniform
pois	Poisson	cauchy	Cauchy
nbinom	Neg. Binomial	weibull	Weibull
gamma	Gamma	chisq	X^2
t	Student	exp	Exponential
binom	Binomial	geom	Geometric
f	Snedecor	mvnorm	Multivariate Normal

Distributions in R

- Examples:

```
> pnorm(3,2,2)
[1] 0.6914625
```

Calculates the value of the cumulative distribution function of the Normal distribution with mean 2 and SD (**NOT VARIANCE**) 2 at x = 3.

```
> qgamma(0.3,1,1)
[1] 0.3566749
```

Finds the 0.3 percentile of the Gamma distribution with parameters 1 and 1.

```
> dt(2,3)
[1] 0.06750966
```

Calculates the value of the pdf of the Student distribution with 3 df at x = 2.

```
> runif(5,-2,2)
[1]  1.3448055 -0.4691324  1.2517269  1.5576504  0.9563447
```

Generates 5 uniform in (-2,2) variates.

Distributions in R

- Arguments of the previous functions could be vectors.

```
> dexp(1:5,2)
[1] 2.706706e-01 3.663128e-02 4.957504e-03 6.709253e-04
    9.079986e-05
```

Calculates the value of the pdf of the exponential distribution with parameter 2 for x = 1, 2, 3, 4, 5.

- Default Values in the Parameters: The command rnorm(50) generates 50 normal variates with $\mu = 0$ and $\sigma = 1$.

Distributions in R

- If $X \sim Student(10)$ then the $P(X \leq 2)$ is equal to

```
> pt(2,10)
[1] 0.963306
```

while the $P(X>2)$ is equal to

```
> pt(2,10, lower.tail=FALSE)
[1] 0.03669402
```

Distributions in R

- All functions (except the ones that are prefixed with r) could be in log (natural logarithm) scale

```
> pnorm(3,2,2)
[1] 0.6914625
> pnorm(3,2,2, log=T)
[1] -0.3689464
> dt(2,3)
[1] 0.06750966
> dt(2,3, log=T)
[1] -2.695485
```

Distributions in R

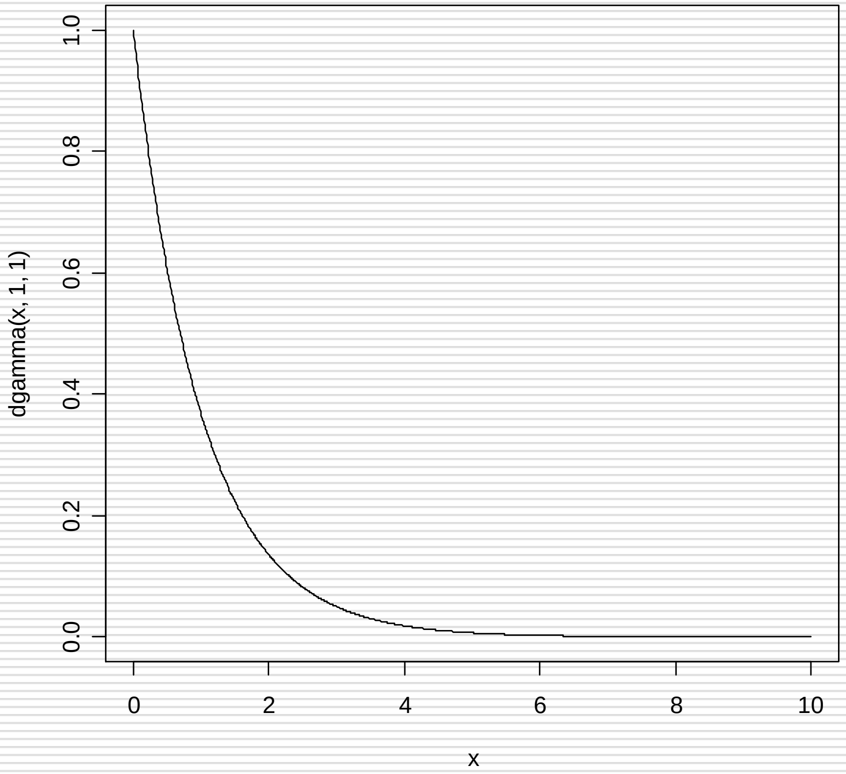
- Working with the Normal distributions requires using these four functions:
  - dnorm(x, mean = 0, sd = 1, log = FALSE).
  - pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE).
  - qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE).
  - rnorm(n, mean = 0, sd = 1).
- If Φ is the cumulative distribution function for a standard Normal distribution, then pnorm(q) = Φ(q) and qnorm(p) = Φ^{-1}(p).

Distributions in R

- Plot a pdf or pmf:

```
> x<-seq(0,10, 0.01)
> plot(x, dgamma(x,1,1), type='l')
```

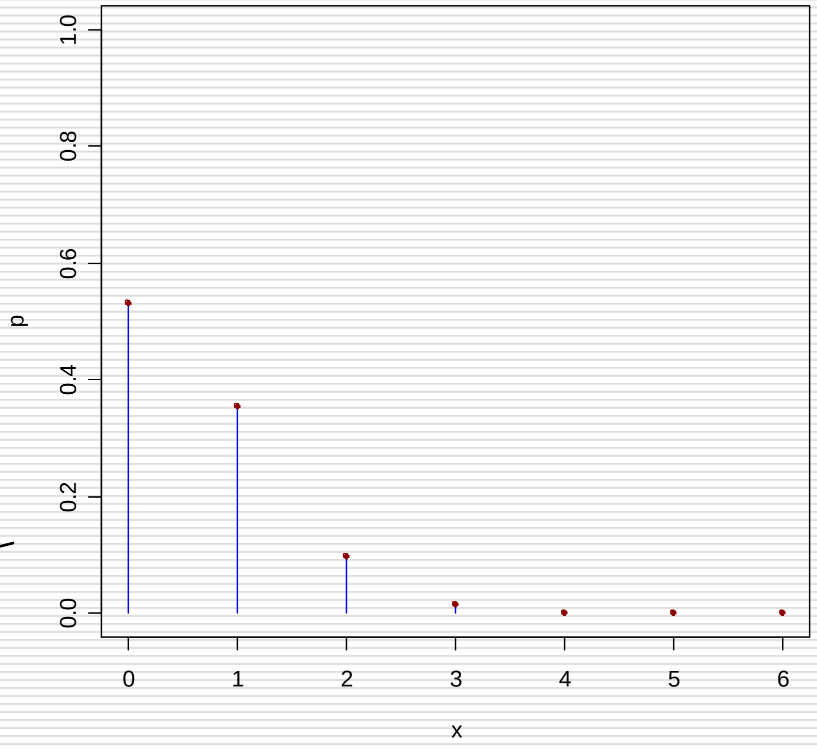
Gamma distribution with parameters (1,1)



Distributions in R

```
> n<-6
> p<-0.1
> x<-0:6
> pr<-dbinom(x,n,p)
>plot(x,pr,type="h",xlim=c(0,6),
    ylim=c(0,1),
    col="blue",ylab="p")
>points(x,pr,pch=20,col="dark
    red")
```

Binomial distribution with n=6 και p=0.1



Simulations

- Generating random Normal variates

```
> x <- rnorm(10)
> x
[1] 1.38380206 0.48772671 0.53403109 0.66721944
[5] 0.01585029 0.37945986 1.31096736 0.55330472
[9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
[1] 23.38812 20.16846 21.87999 20.73813 19.59020
[6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
Min. 1st Qu. Median Mean 3rd Qu. Max.
18.32 19.73 20.55 20.67 21.67 23.39
```

set.seed

- Setting the random number seed with set.seed ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808
[5] 0.3295078
> rnorm(5)
[1] -0.8204684 0.4874291 0.7383247 0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808
[5] 0.3295078
```

Simulations

- Generating Poisson data

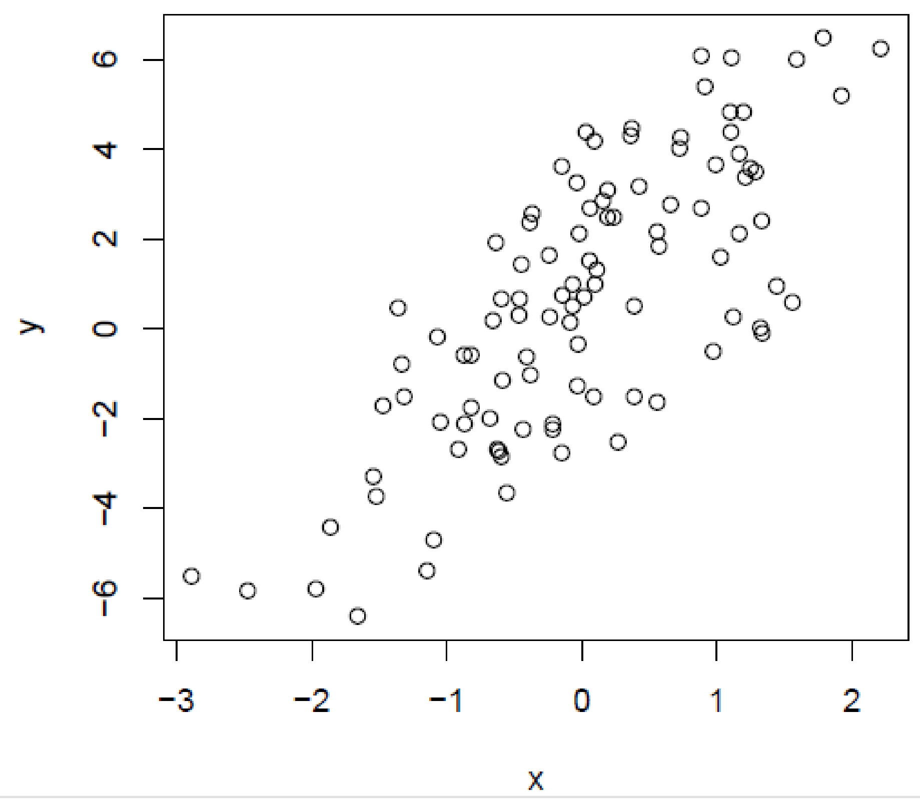
```
> rpois(10, 1)
[1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
[1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
[1] 20 11 21 20 20 21 17 15 24 20
> ppois(2, 2) ## Cumulative distribution
[1] 0.6766764 ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347 ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662 ## Pr(x <= 6)
```

Simulate from a Linear Model

- Suppose we want to simulate 100 values from the following linear model $y = \beta_0 + \beta_1 x + \varepsilon$, where $\varepsilon \sim N(0; 2^2)$. Assume $X \sim N(0; 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-6.4080 -1.5400 0.6789 0.6893 2.9300 6.5050
> plot(x, y)
```

Simulate from a Linear Model

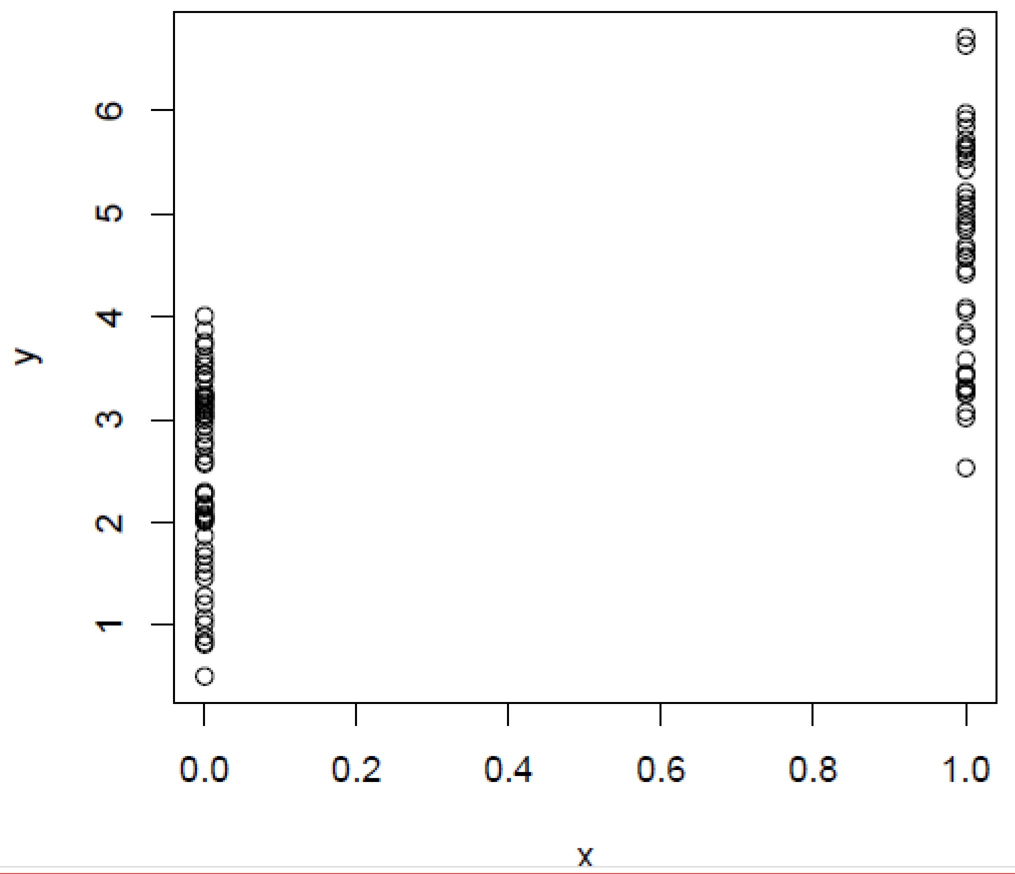


Simulate from a Linear Model

- What if X is binary?

```
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-3.4940 -0.1409 1.5770 1.4320 2.8400
   6.9410
> plot(x, y)
```

Simulate from a Linear Model

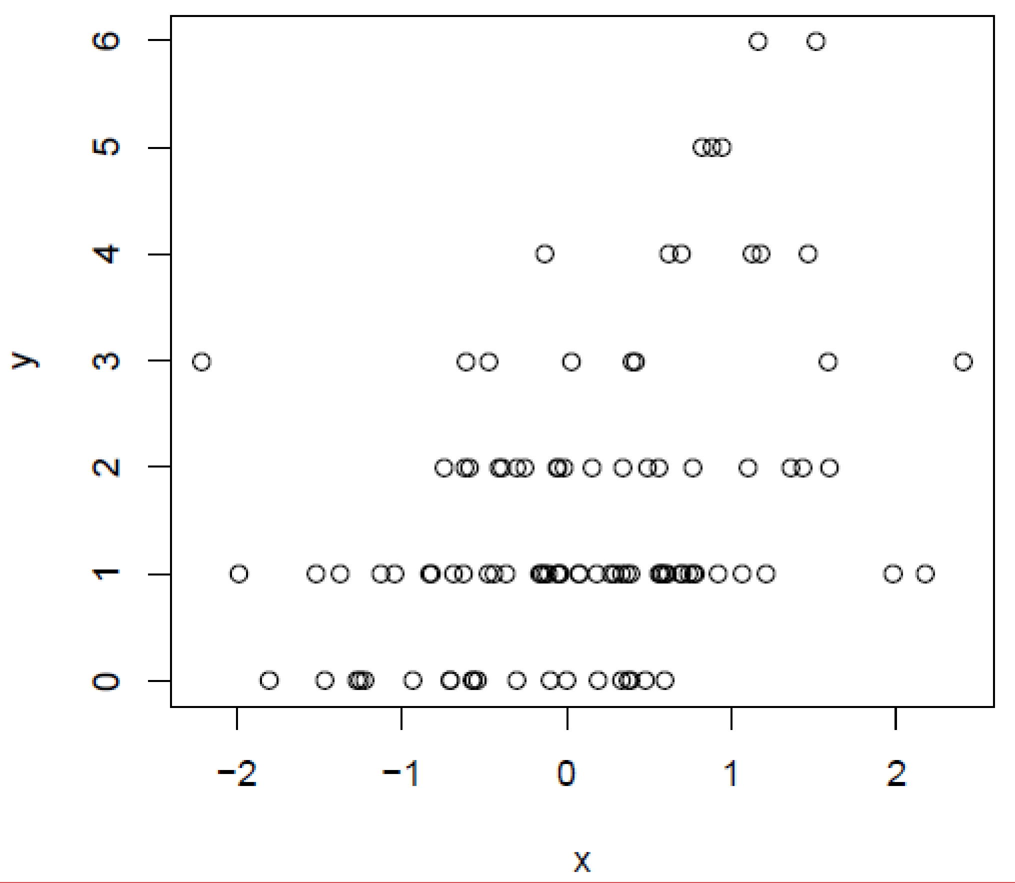


Simulate from a Poisson Model

- Suppose we want to simulate 100 values from a Poisson model where Y ~ Poisson(μ), log(μ) = $\beta_0 + \beta_1 x$. Assume X ~ N(0; 1^2) and $\beta_0 = 0.5$ and β_1 = 0:3.

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.00 1.00 1.00 1.55 2.00 6.00
> plot(x, y)
```

Simulate from a Poisson Model



Sampling

- The **sample** function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
> sample(1:10) ## permutation
[1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
[1] 2 3 4 1 9 5 10 8 6 7
> sample(1:10, replace = TRUE) ## Sample w/replacement
[1] 2 9 7 8 2 8 5 9 7 8
> sample(1:5, replace = TRUE, prob=c(0.3, 0.3, 0.4, 0,0)) ## Sample
    w/replacement and pre-specified probabilities
[1] 3 1 2 2 1
```

Weak Law of Large Numbers

- If $X_1,\ldots,X_n$ are idependent and identical distributed Random Variables with finite mean μ, then

$$\bar{X} = n^{-1}\sum_1^n X_i \to \mu \text{ by probability as } n \to \infty.$$

- Application: Let X_i~Bernoulli(p). Then $\mu=P(X_i=1)=p$, and therefore

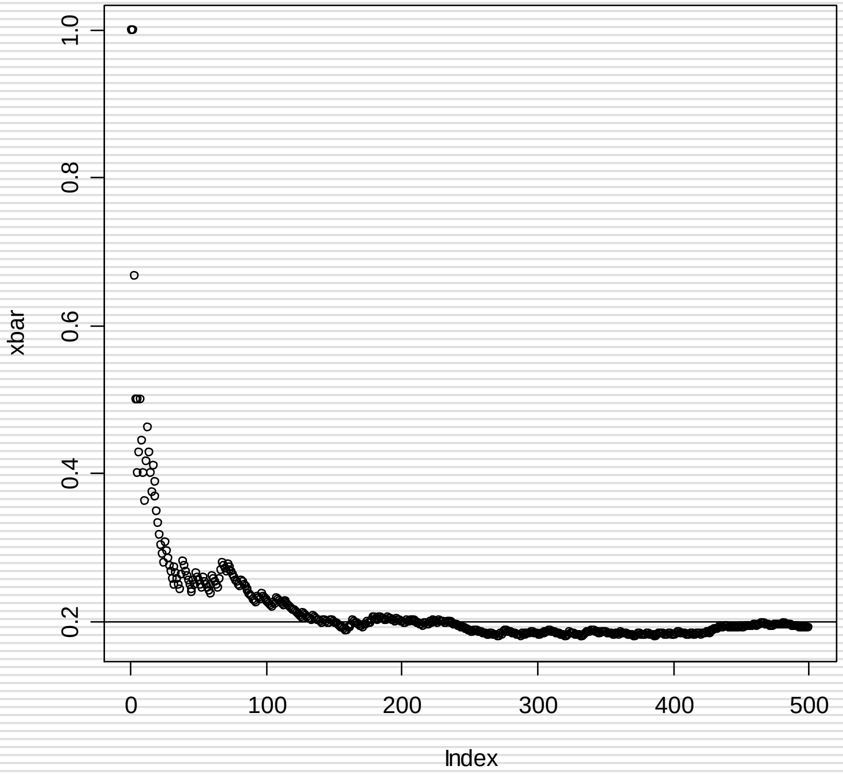
$$\bar{X} \to p \text{ by probability as } n \to \infty.$$

Weak Law of Large Numbers

- p=0.2, n=500:

```
> x<-rbinom(500,1,0.2)
> xbar<-cumsum(x)/(1:500)
> plot(xbar)
> abline(h=0.2)
```

Weak Law of Large Numbers

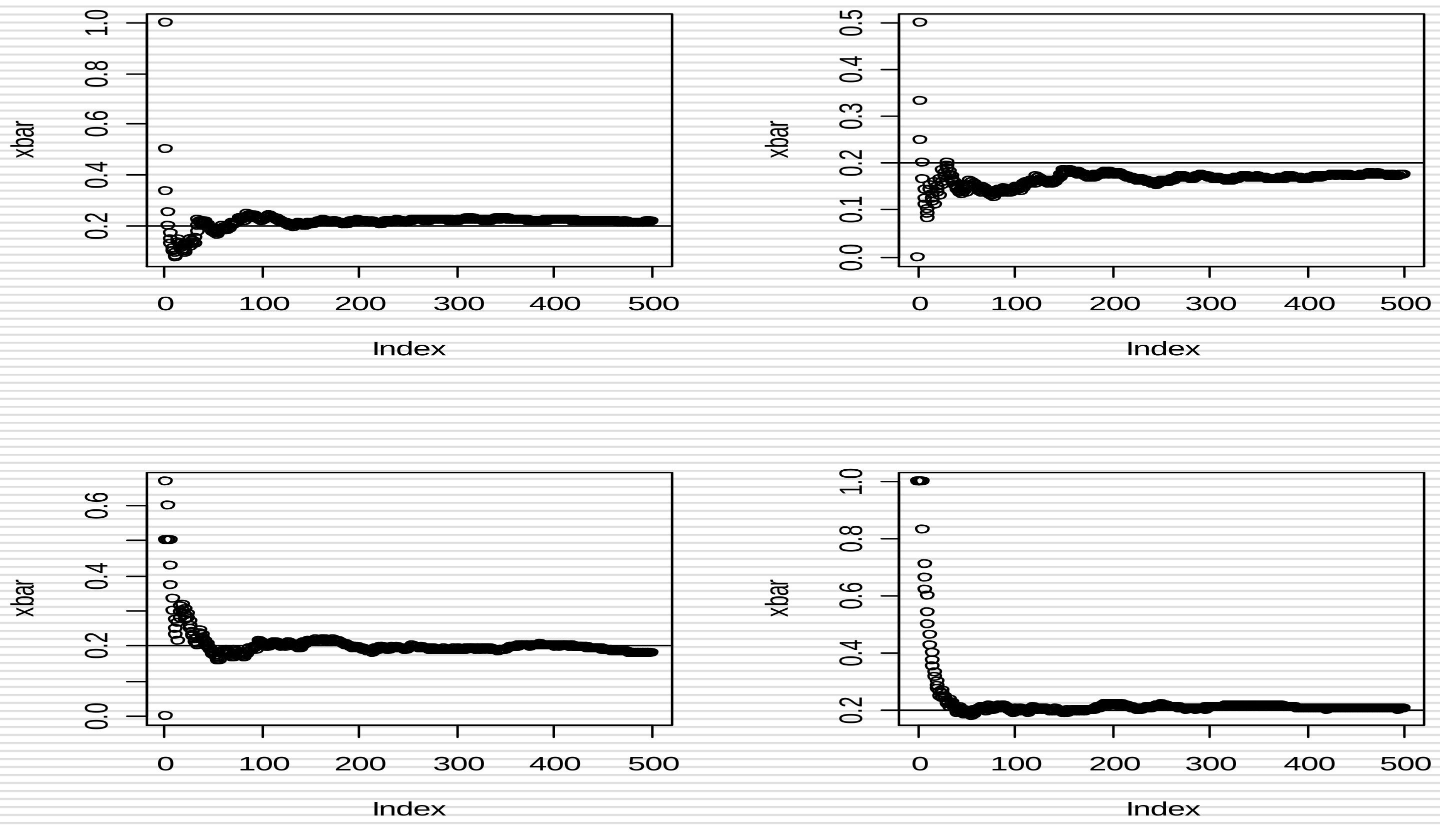


Weak Law of Large Numbers

- Repeat 4 times:

```
> par(mfrow=c(2,2))
> for(i in 1:4)
 {
 x<-rbinom(500,1,0.2)
 xbar<-cumsum(x)/(1:500)
 plot(xbar)
 abline(h=0.2)
 }
```

Weak Law of Large Numbers

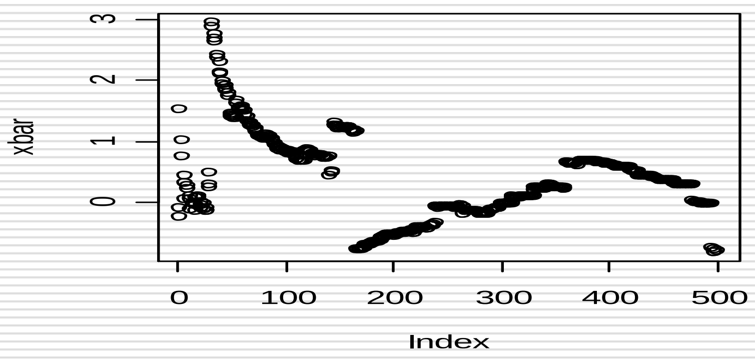


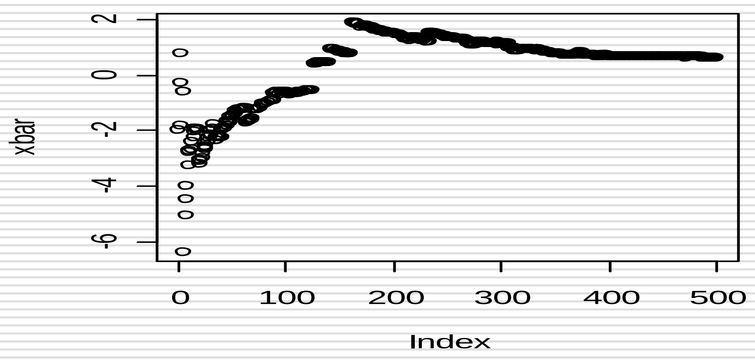
Weak Law of Large Numbers

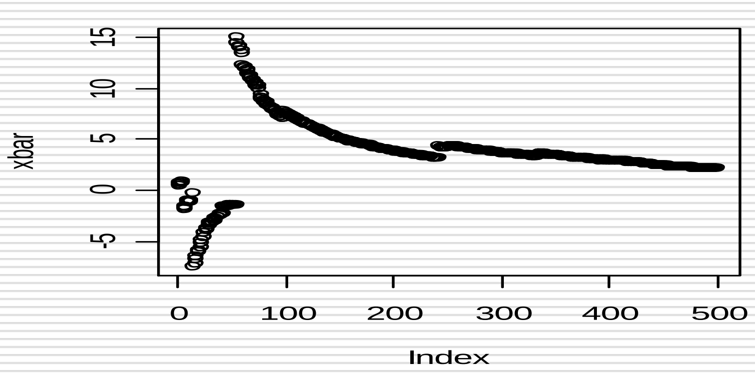
- Example of the Cauchy distribution (the mean is not finite).

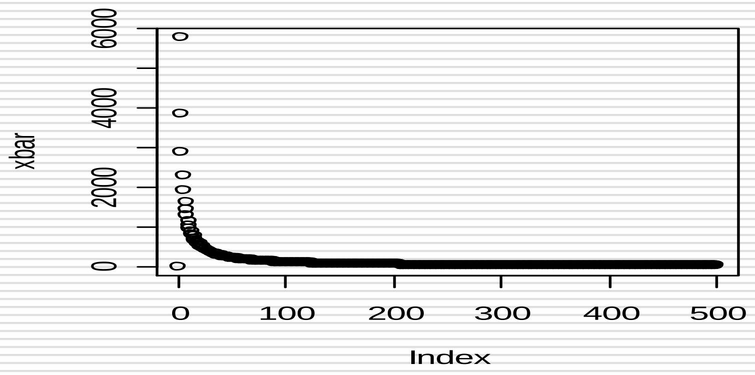
```
> par(mfrow=c(2,2))
> for(i in 1:4)
 {
 x<-rcauchy(500)
 xbar<-cumsum(x)/(1:500)
 plot(xbar)
 }
```

Weak Law of Large Numbers









Central Limit Theorem

- Let $X_1,\ldots,X_n$ be independent and identical distributed Random Variables with finite mean μ and variance σ². Then

$$S_n = \frac{\sum_1^n X_i - n\mu}{\sigma\sqrt{n}} \to Z \sim N(0,1) \text{ by law as } n \to \infty.$$

- Equivalently

$$\bar{X} \to Y \sim N(\mu, \sigma^2 / n) \text{ by law as } n \to \infty.$$

Central Limit Theorem

- Example: Let n = 150 from Poisson with $\lambda=2$ (then $\lambda=\mu=\sigma^2=2$).

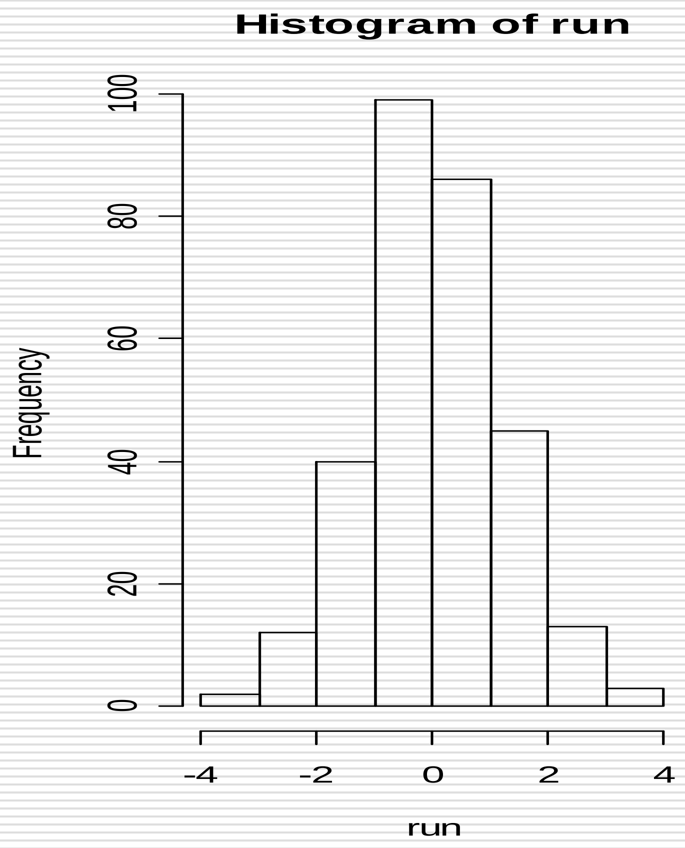
```
> poisson.clt<-function(k,n,l)
 {
 Sn<-rep(NA,k)
 for(i in 1:k)
 {
 x<-rpois(n,l)
 Sn[i]<-(sum(x)-n*l)/(sqrt(n*l))
 }
 return(Sn)
 }
```

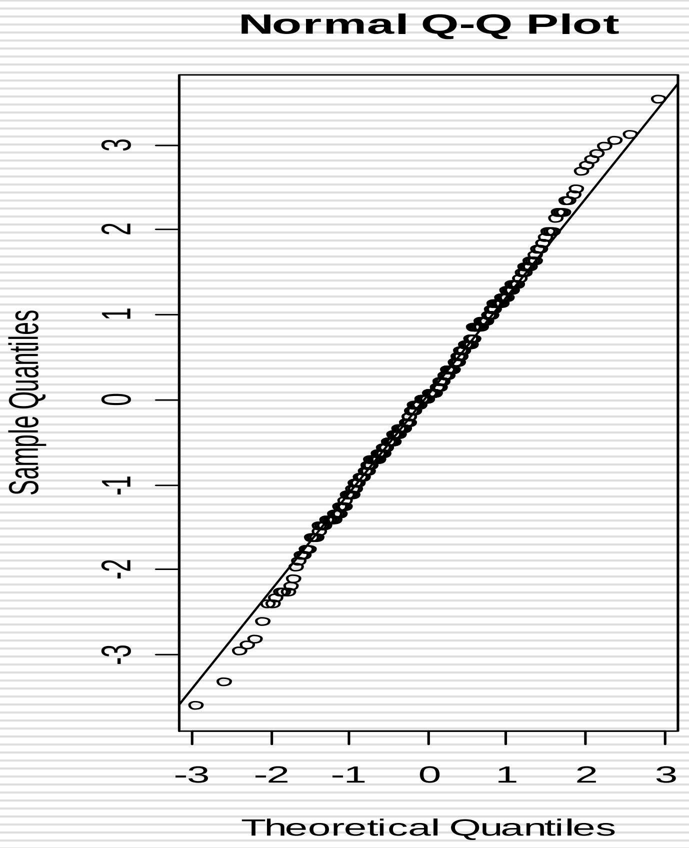
Central Limit Theorem

- We use k=300 replications.

```
> run<-poisson.clt(300,150,2)
> par(mfrow=c(1,2))
> hist(run)
> qqnorm(run)
> qqline(run)
```

Central Limit Theorem





Plotting

- The plotting and graphics engine in R is encapsulated in a few base and recommend packages:
  - **graphics**: contains plotting functions for the "base" graphing systems, including plot, hist, boxplot and many others.
  - **lattice**: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like xyplot, bwplot, levelplot.
  - **grid**: implements a different graphing system independent of the "base" system; the lattice package builds on top of grid; we seldom call functions from the grid package directly.
  - **grDevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

Base Graphics

- Base graphics are used most commonly and are a very powerful system for creating 2-D graphics.
  - Calling plot(x, y) or hist(x) will launch a graphics device (if one is not already open) and draw the plot on the device.
  - If the arguments to plot are not of some special class, then the default method for plot is called; this function has many arguments, letting you set the title, x axis lable, y axis label, etc.
  - The base graphics system has many parameters that can set and tweaked; these parameters are documented in ?par; it wouldn’t hurt to memorize this help page!

Main Parameters of a Graph

- **main**: Title of the graph.
- **sub**: Subtitle of the graph.
- **xlab & ylab**: Titles of the axes.
- **xlim & ylim**: Range of the axes.

Base Graphics Parameters

- The **par** function is used to specify global graphics parameters that affect all plots in an R session. These parameters can often be overridden as arguments to specific plotting functions.
  - **pch**: the plotting symbol (default is open circle).
  - **lty**: the line type (default is solid line), can be dashed, dotted, etc.
  - **lwd**: the line width, specified as an integer multiple.
  - **col**: the plotting color, specified as a number, string, or hex code; the colors function gives you a vector of colors by name.
  - **las**: the orientation of the axis labels on the plot.
  - **bg**: the background color.
  - **mar**: the margin size.
  - **oma**: the outer margin size (default is 0 for all sides).
  - **mfrow**: number of plots per row, column (plots are filled row-wise).
  - **mfcol**: number of plots per row, column (plots are filled column-wise).

Base Graphics Parameters

- Some default values:

```
> par("lty")
[1] "solid"
> par("lwd")
[1] 1
> par("col")
[1] "black"
> par("pch")
[1] 1
```

```
> par("bg")
[1] "transparent"
> par("mar")
[1] 5.1 4.1 4.1 2.1
> par("oma")
[1] 0 0 0 0
> par("mfrow")
[1] 1 1
> par("mfcol")
[1] 1 1
```

Some Important Base Plotting Functions

- **plot**: make a scatterplot, or other type of plot depending on the class of the object being plotted.
- **lines**: add lines to a plot, given a vector x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots.
- **points**: add points to a plot.
- **text**: add text labels to a plot using specified x, y coordinates.
- **title**: add annotations to x, y axis labels, title, subtitle, outer margin.
- **mtext**: add arbitrary text to the margins (inner or outer) of the plot.
- **axis**: adding axis ticks/labels.

Text & Symbol Size

- **cex:** number indicating the amount by which plotting text and symbols should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, etc.
- **cex.axis:** magnification of axis annotation relative to cex.
- **cex.lab:** magnification of x and y labels relative to cex.
- **cex.main:** magnification of titles relative to cex.
- **cex.sub:** magnification of subtitles relative to cex.

Axes

- You can create custom axes using the **axis( )** function.
  > axis(*side*, at=, labels=, pos=, lty=, col=, las=, tck=, ...)

  - **side:** an integer indicating the side of the graph to draw the axis (1=bottom, 2=left, 3=top, 4=right).
  - **at:** a numeric vector indicating where tic marks should be drawn.
  - **labels:** a character vector of labels to be placed at the tickmarks (if NULL, the *at* values will be used).
  - **pos:** the coordinate at which the axis line is to be drawn. (i.e., the value on the other axis where it crosses).
  - **lty:** line type.
  - **col:** the line and tick mark color.
  - **las:** labels are parallel (=0) or perpendicular(=2) to axis.
  - **tck:** length of tick mark as fraction of plotting region (negative number is outside graph, positive number is inside, 0 suppresses ticks, 1 creates gridlines) default is -0.01.

Axes

- The option **axes=FALSE** suppresses both x and y axes.
- **xaxt="n"** and **yaxt="n"** suppress the x and y axis respectively.
- **xlab=“”** and **ylab=“”** suppress the titles of the two axes.

Legend

- Add a legend with the **legend()** function.

  > legend(location, title, legend, ...)

  - **location:** There are several ways to indicate the location of the legend. You can give a **x,y coordinate** for the upper left hand corner of the legend. You can use **locator(1)**, in which case you use the mouse to indicate the location of the legend. You can also use the **keywords** "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "bottomright", or "center". If you use a keyword, you may want to use **inset=** to specify an amount to move the legend into the graph (as fraction of plot region).
  - **title:** A character string for the legend title (optional).
  - **legend:** A character vector with the labels.
  - **...Other options**. If the legend labels colored lines, specify **col=** and a vector of colors. If the legend labels point symbols, specify **pch=** and a vector of point symbols. If the legend labels line width or line style, use **lwd=** or **lty=** and a vector of widths or styles. To create colored boxes for the legend (common in bar, box, or pie charts), use **fill=** and a vector of colors.

Fonts

- **font:** Integer specifying font to use for text. 1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol.
- **font.axis:** font for axis annotation.
- **font.lab:** font for x and y labels.
- **font.main:** font for titles.
- **font.sub:** font for subtitles.
- **ps:** font point size (roughly 1/72 inch) text size=ps*cex.
- **family:** font family for drawing text. Standard values are "serif", "sans", "mono", "symbol". Mapping is device dependent. In windows, mono is mapped to "TT Courier New", serif is mapped to"TT Times New Roman", sans is mapped to "TT Arial", mono is mapped to "TT Courier New", and symbol is mapped to "TT Symbol" (TT=True Type). You can add your own mappings.

Margins & Graph Size

- **mar:** numerical vector indicating margin size c(bottom, left, top, right) in lines. default = c(5, 4, 4, 2) + 0.1.
- **mai:** numerical vector indicating margin size c(bottom, left, top, right) in inches.
- **pin:** plot dimensions (width, height) in inches.

Saving Graphs

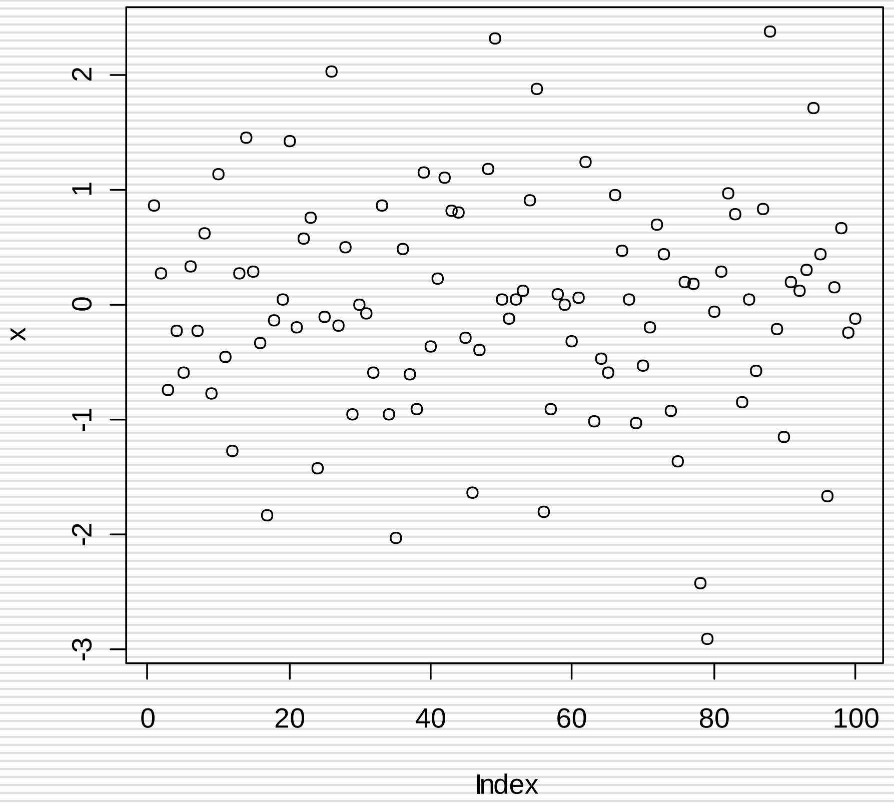
- You can save the graph in a variety of formats from the menu
  File -> Save As.
- You can also save the graph via code using one of the following functions.
  - **pdf("mygraph.pdf"):** pdf file.
  - **win.metafile("mygraph.wmf"):** windows metafile.
  - **png("mygraph.png"):** pngfile.
  - **jpeg("mygraph.jpg"):** jpeg file.
  - **bmp("mygraph.bmp"):** bmp file.
  - **postscript("mygraph.ps"):** postscript file.

Useful Graphics Devices

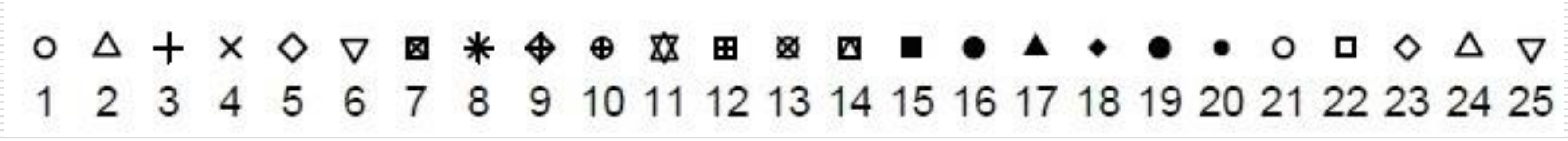
- The list of devices is found in ?Devices; there are also devices created by users on CRAN
  - **pdf**: useful for line-type graphics, vector format, resizes well, usually portable
  - **postscript**: older format, also vector format and resizes well, usually portable,can be used to create encapsulated postscript files, Windows systems often don’t have a postscript viewer.
  - **xfig**: good of you use Unix and want to edit a plot by hand.

Plot

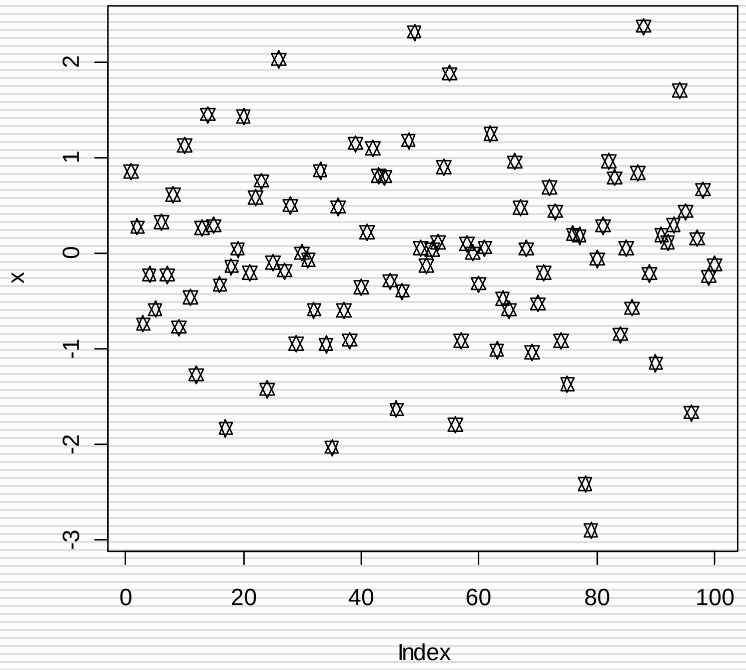
```
> plot(x)
```



Different plotting symbols

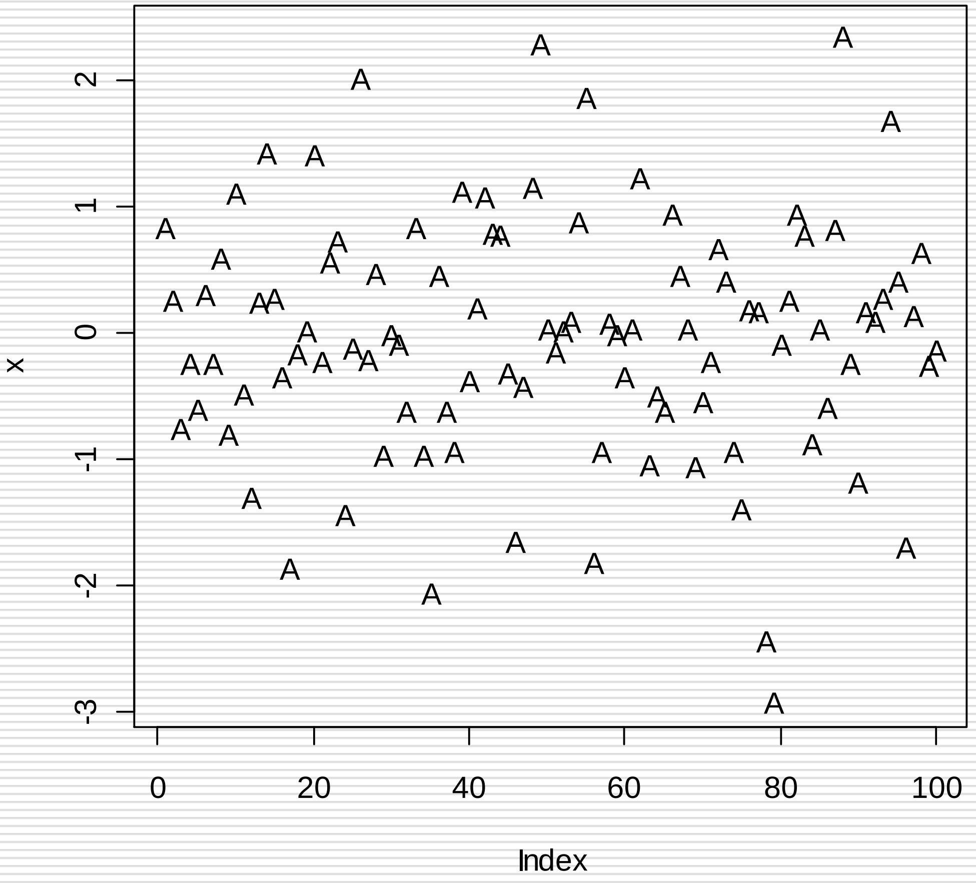


```
> plot(x, pch=11)
```



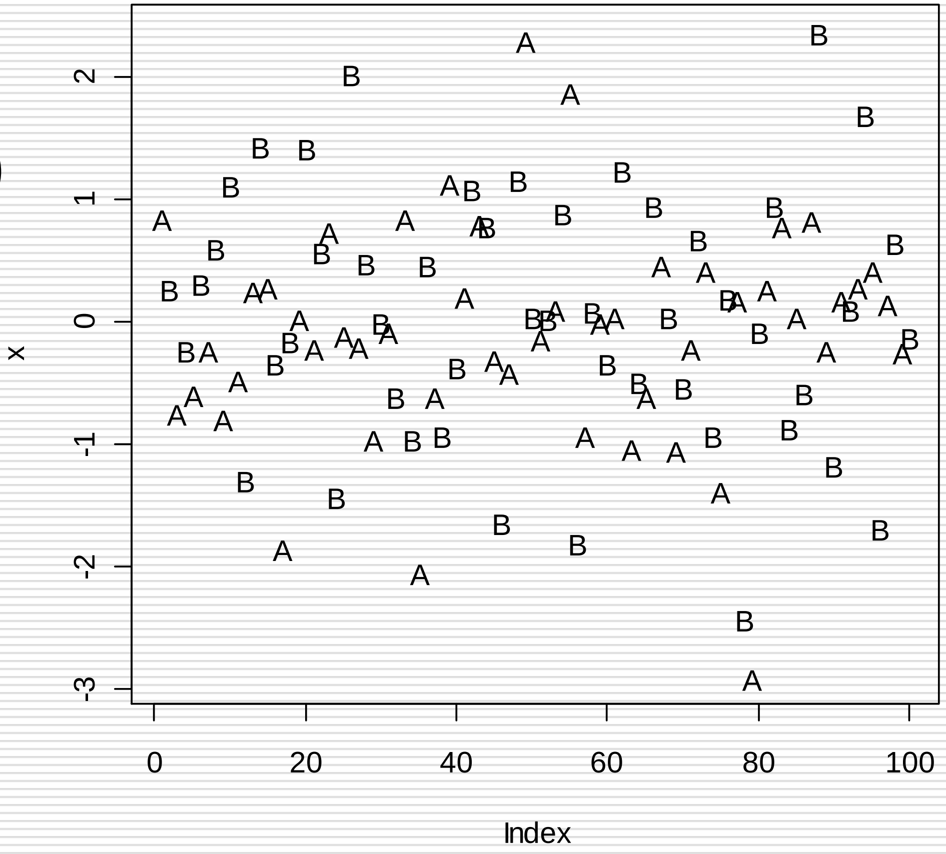
Different plotting symbols

```
> plot(x, pch='A')
```



Different plotting symbols

```
> plot(x, pch=c('A', 'B'))
```

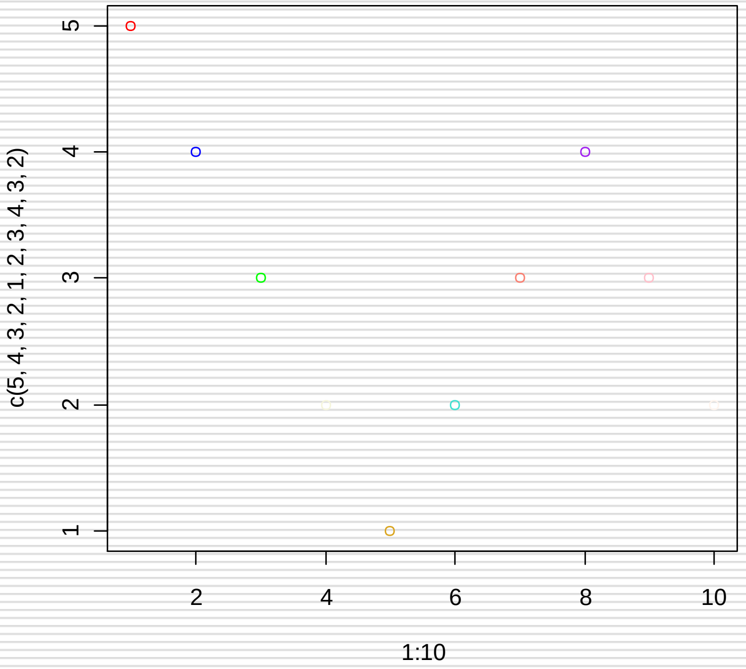


Colors

```
> colors()
  [1] "white"          "aliceblue"            "antiquewhite"
  [4] "antiquewhite1"  "antiquewhite2"        "antiquewhite3"
  [7] "antiquewhite4"  "aquamarine"           "aquamarine1"
 [10] "aquamarine2"    "aquamarine3"          "aquamarine4"
 [13] "azure"          "azure1"               "azure2"
 [16] "azure3"         "azure4"               "beige"
 [19] "bisque"         "bisque1"              "bisque2"
 [22] "bisque3"        "bisque4"              "black"
…………………………………………………………
…………………………………………………………
…………………………………………………………
```

Colors

```
> plot(1:10, c(5, 4, 3, 2, 1, 2, 3, 4, 3, 2), col=c("red",
    "blue", "green", "beige", "goldenrod", "turquoise",
              "salmon", "purple", "pink", "seashell"))
```



Colors

```
> palette()
[1] "black" "red" "green3" "blue" "cyan" "magenta" "yellow"
[8] "gray"
```

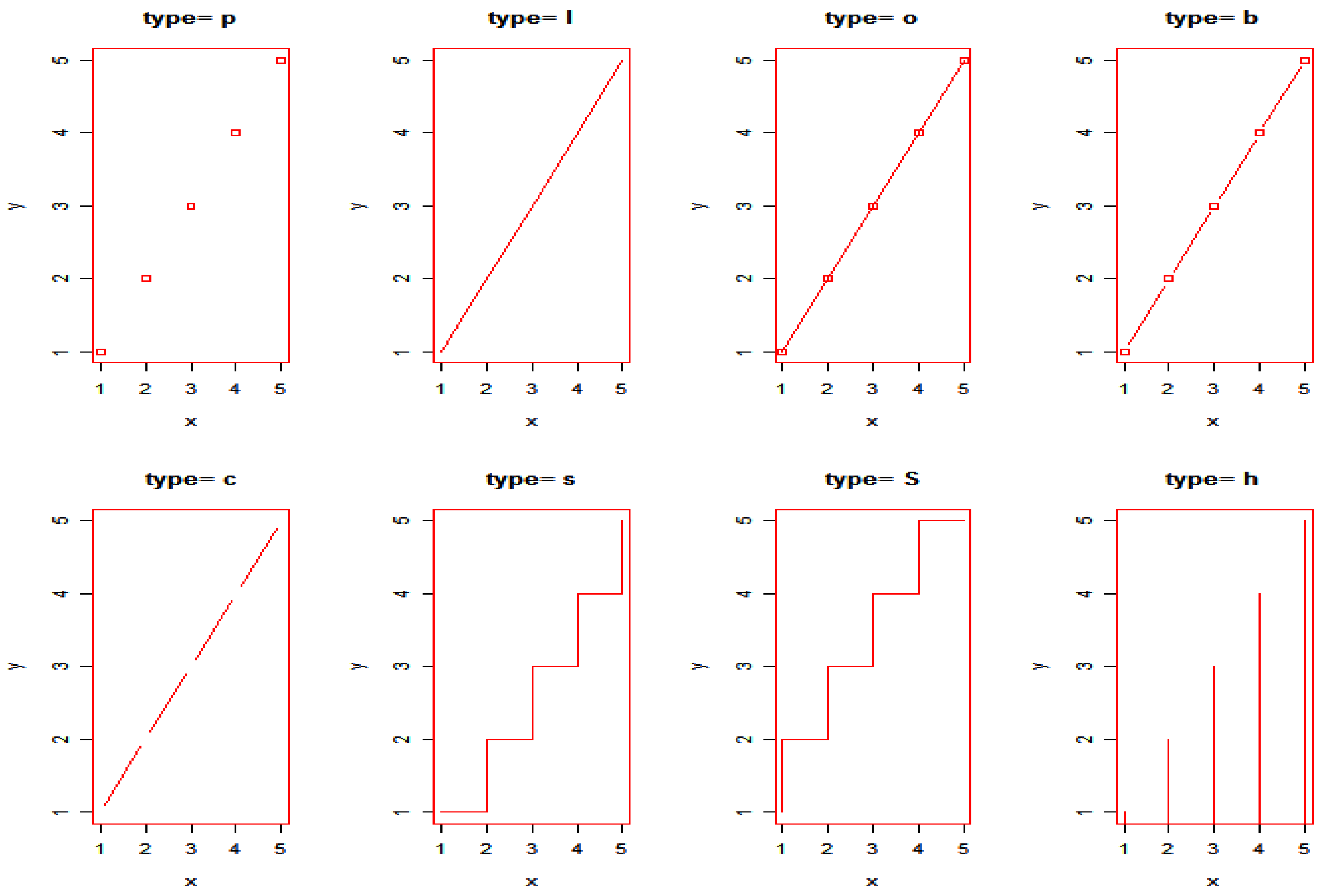
- col = 1 $\longleftrightarrow$ col = "black”
- col = 2 $\longleftrightarrow$ col = "red‟, etc

Type Arguments

- **"p":** Points
- **"l":** Lines
- **"b":** Both
- **"c":** The lines part alone of "b"
- **"o":** Both “overplotted”
- **"h":** Histogram like (or high-density) vertical lines
- **"n":** No plotting

Type Argument

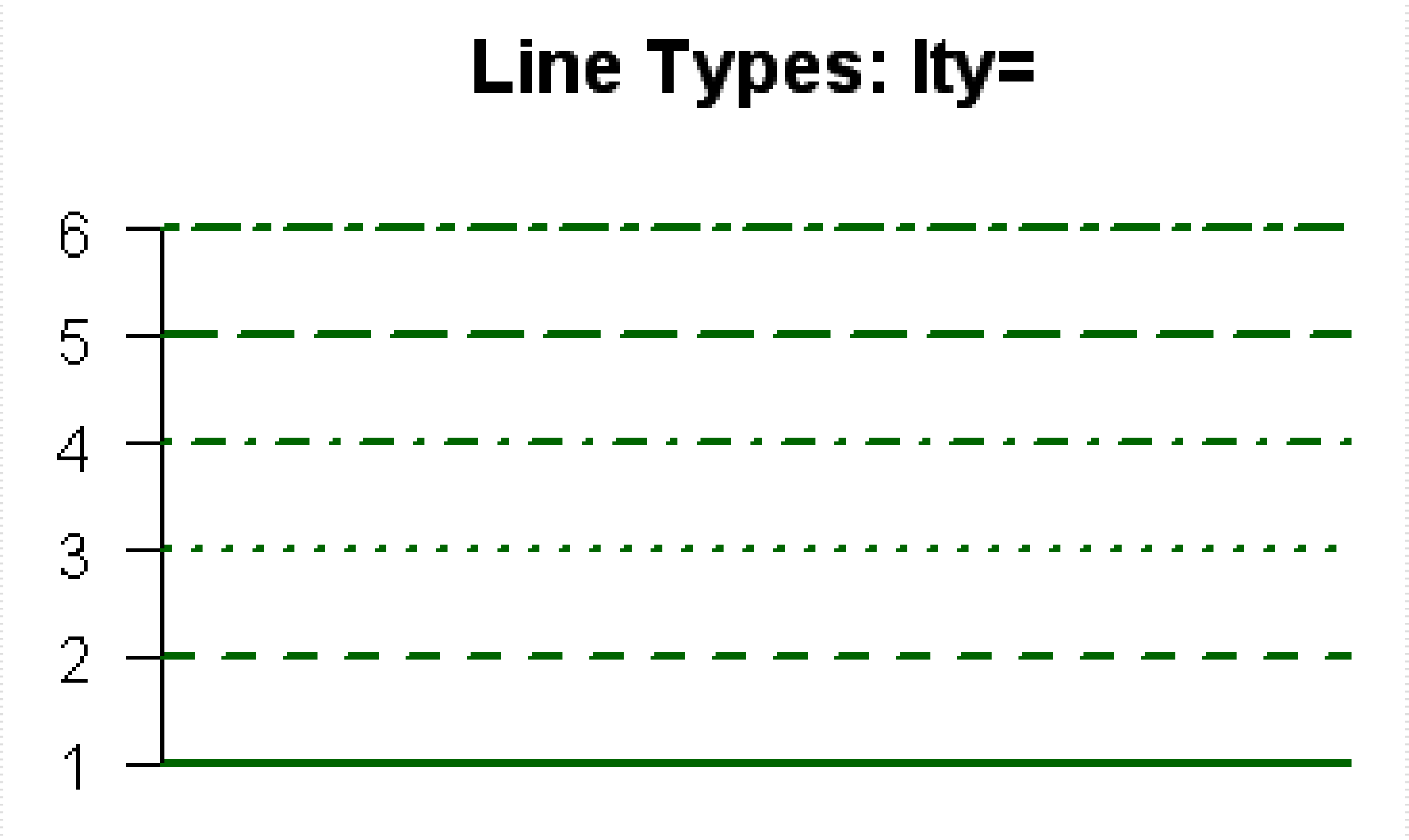
```
> x <- c(1:5); y <- x # create some data
  par(pch=22, col="red") # plotting symbol
  #and color
  par(mfrow=c(2,4)) # all plots on one page
  opts = c("p","l","o","b","c","s","S","h")
  for(i in 1:length(opts)){
    heading = paste("type=",opts[i])
    plot(x, y, type="n", main=heading)
    lines(x, y, type=opts[i])
  }
```


type= p
type= l
type= o
type= b
type= c
type= s
type= S
type= h
x
y
1
2
3
4
5

Add to a graph

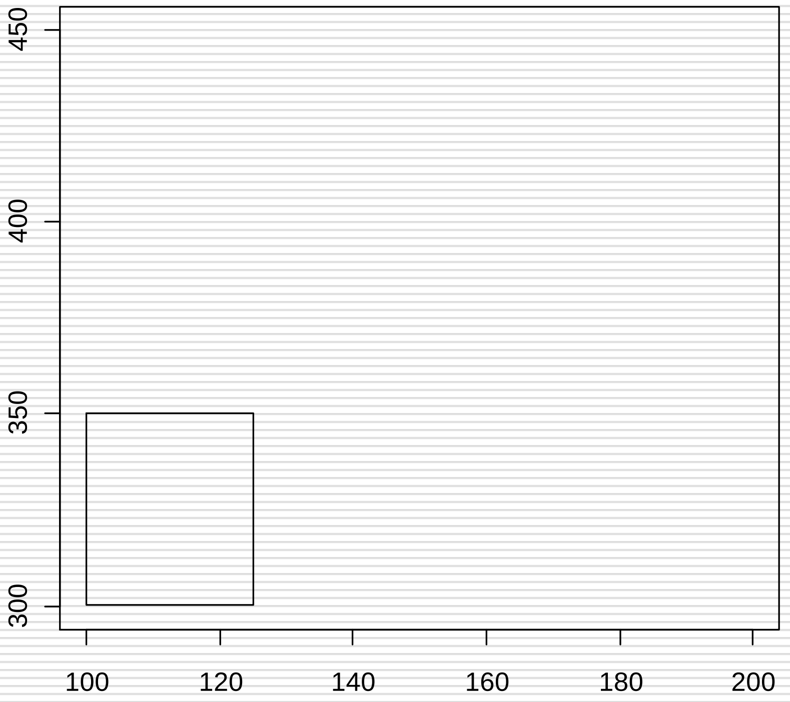
- **points(x,y):** Add points to an existing graph.
- **lines(x,y):** Add a line to an existing graph.
- **abline(a=2, b=4):** Add to the existing graph the line y = 2+4x.
- **abline(v=4):** Add to the existing graph the line x = 4.
- **abline(h=2):** Add to the existing graph the line y = 2.
- **abline(v=1:4):** Add to the existing graph the lines x = 1, x = 2, x = 3 & x = 4.
- **segments(x0, y0, x1,y1):** Draw line segments between pairs of points. (x0, y0): coordinates of points **from** which to draw & (x1, y1): coordinates of points **to** which to draw.
- **arrows(x0, y0, x1,y1):** Draw arrows between pairs of points. (x0, y0): coordinates of points **from** which to draw & (x1, y1): coordinates of points **to** which to draw. Additional arguments here are: (a) length: length of the edges of the arrow head (in inches), (b) angle: angle from the shaft of the arrow to the edge of the arrow head, (c) code: integer code, determining *kind* of arrows to be drawn.

Line Types



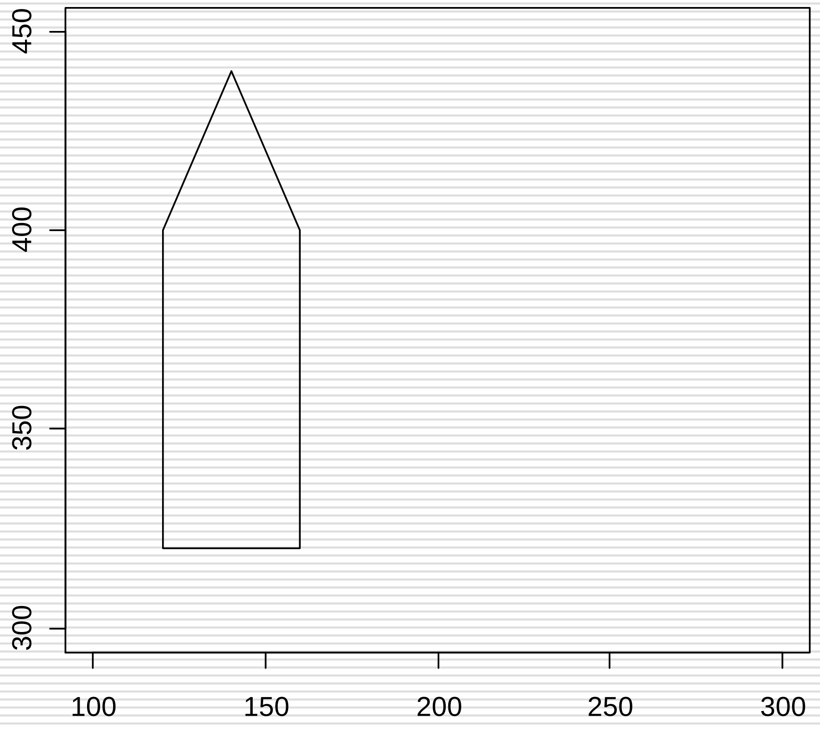
Rectangulars

```
> plot(c(100, 200), c(300, 450),
    type= "n", xlab="", ylab="")
> rect(100, 300, 125, 350)
```



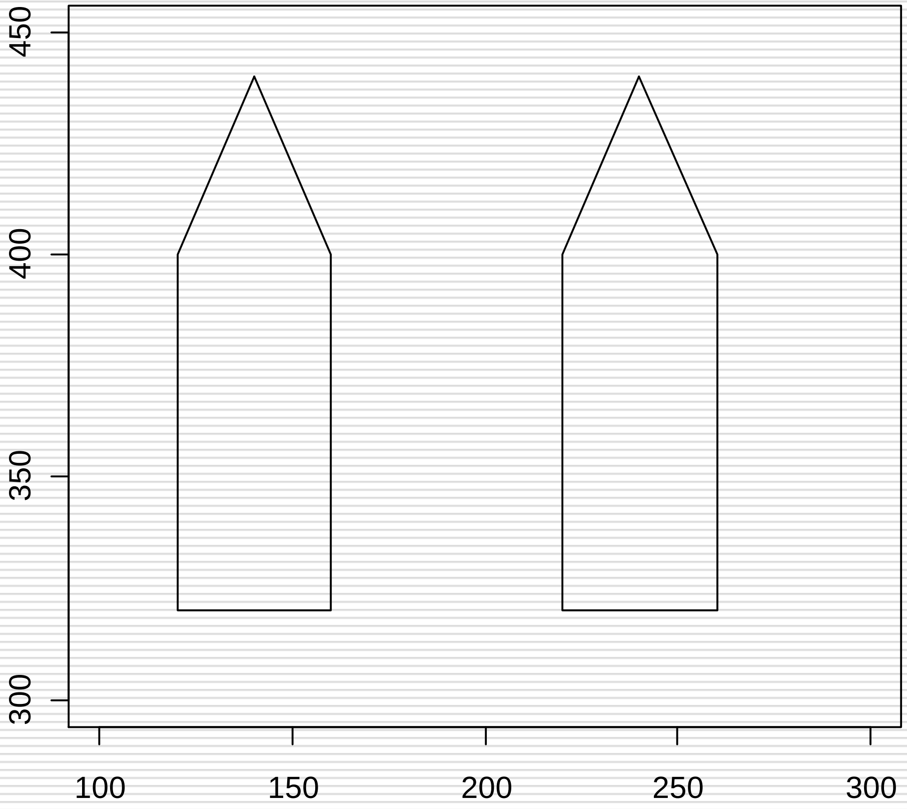
Polygons

```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> polygon(c(140, 120, 120, 160,
    160), c(440, 400, 320, 320,
    400))
```



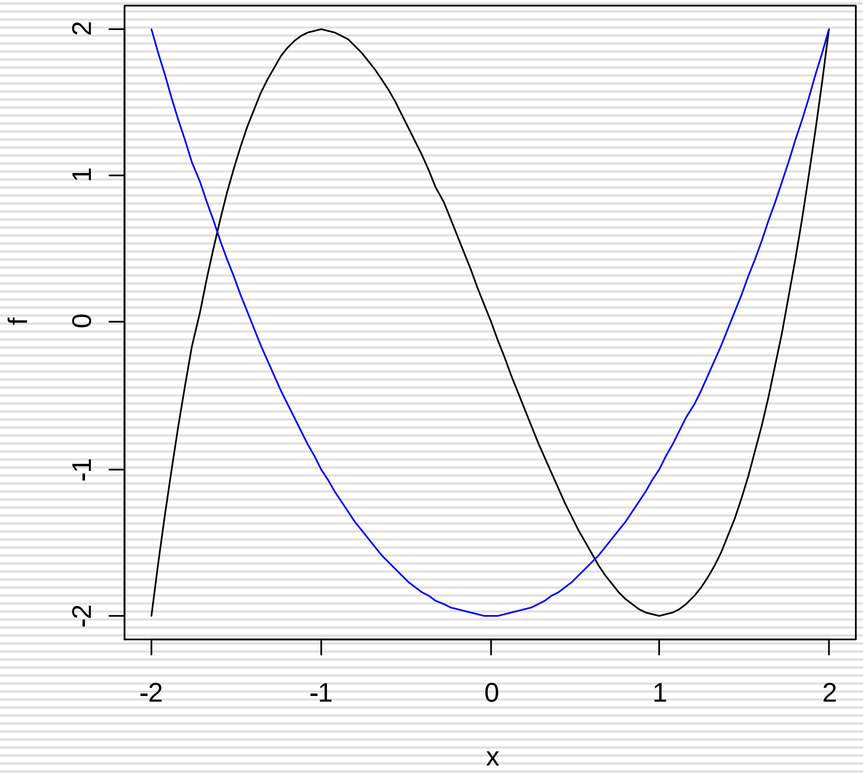
Polygons

```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> polygon(c(140, 120, 120, 160,
    160, NA, 240, 220, 220, 260,
    260),
  c(440, 400, 320, 320, 400, NA,
    440, 400, 320, 320, 400))
```



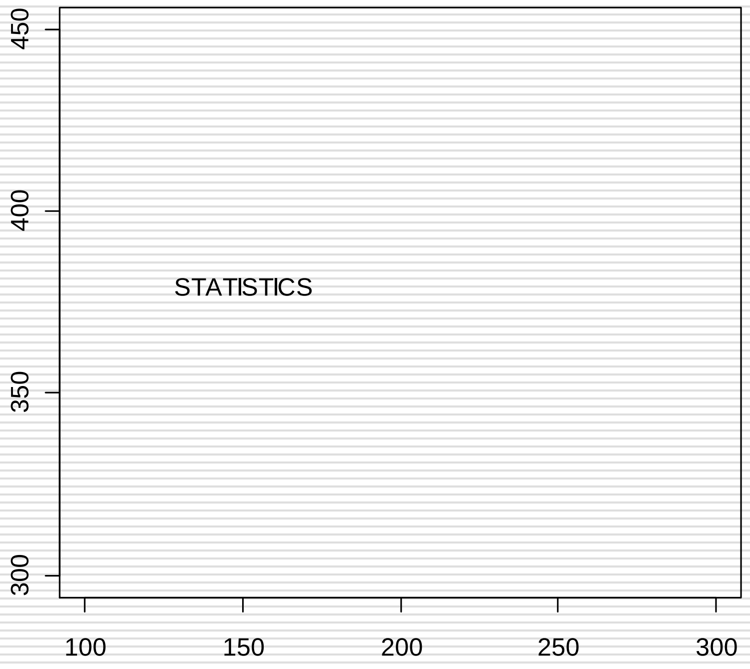
Curves

```
> curve(x^3 - 3*x, -2, 2,
    ylab="f")
> curve(x^2 - 2, add = TRUE, col
    = "blue")
```



Text

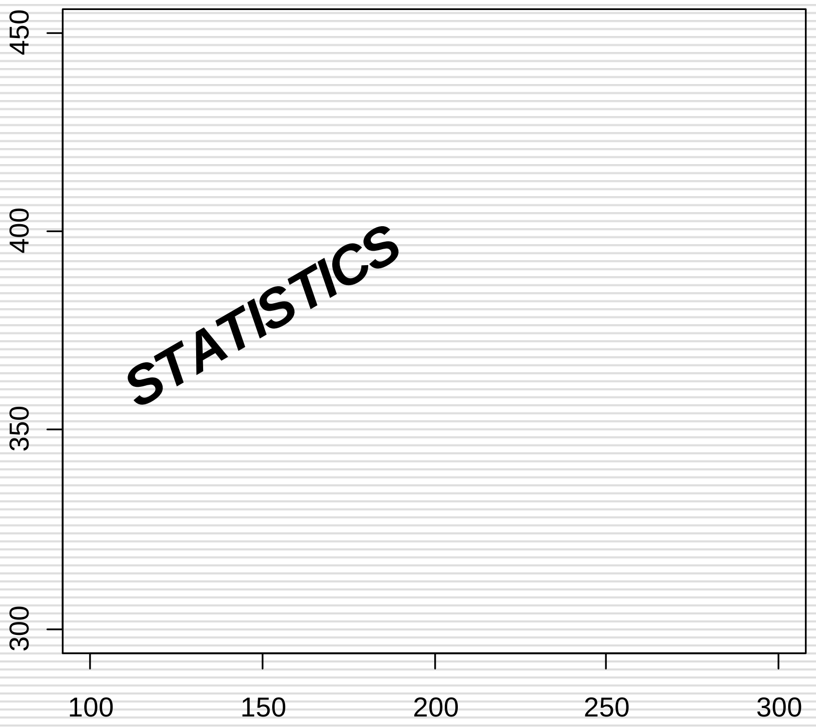
```
> plot(c(100, 300), c(300, 450),
     type= "n", xlab="", ylab="")
> text(150, 380, "STATISTICS")
```



Text

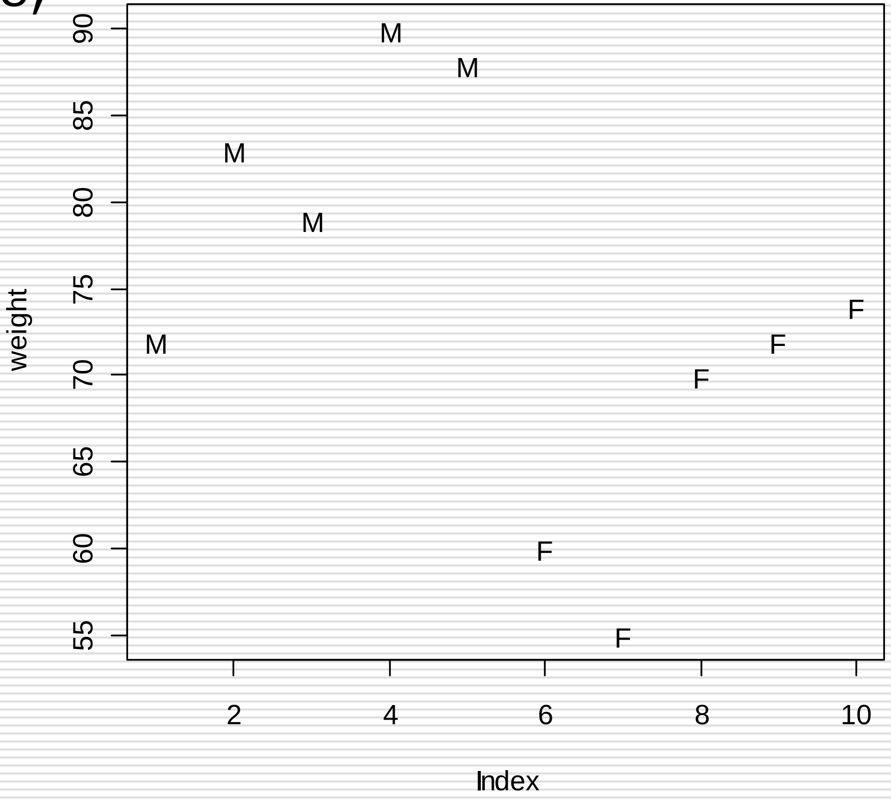
```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> text(150, 380, "STATISTICS",
    srt=30, cex=2, font=4)
```

rotation in degrees



Text

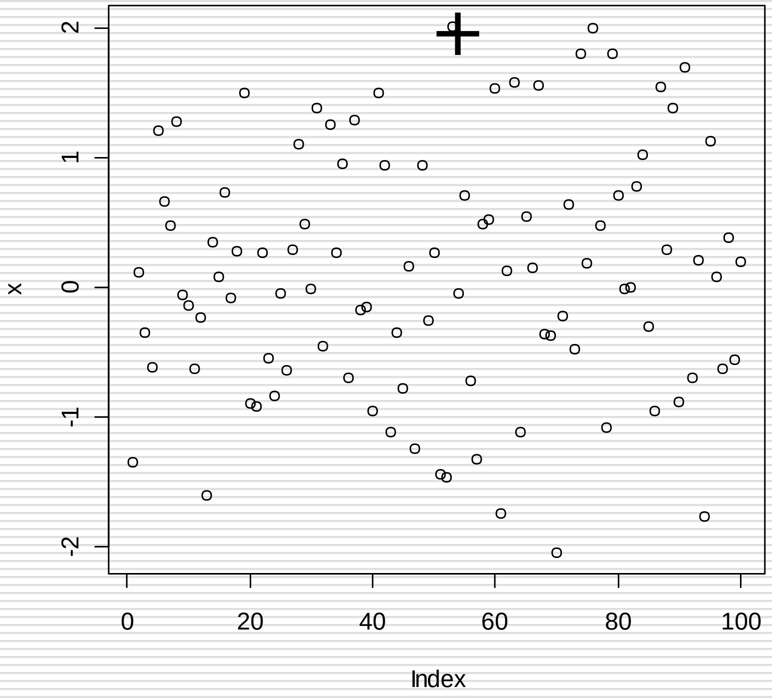
```
> weight<-c(72, 83, 79, 90, 88,
    60, 55, 70, 72, 74)
> gender<-rep(c("M", "F"),
    each=5)
> plot(weight, type="n")
> text(weight, label=gender)
```

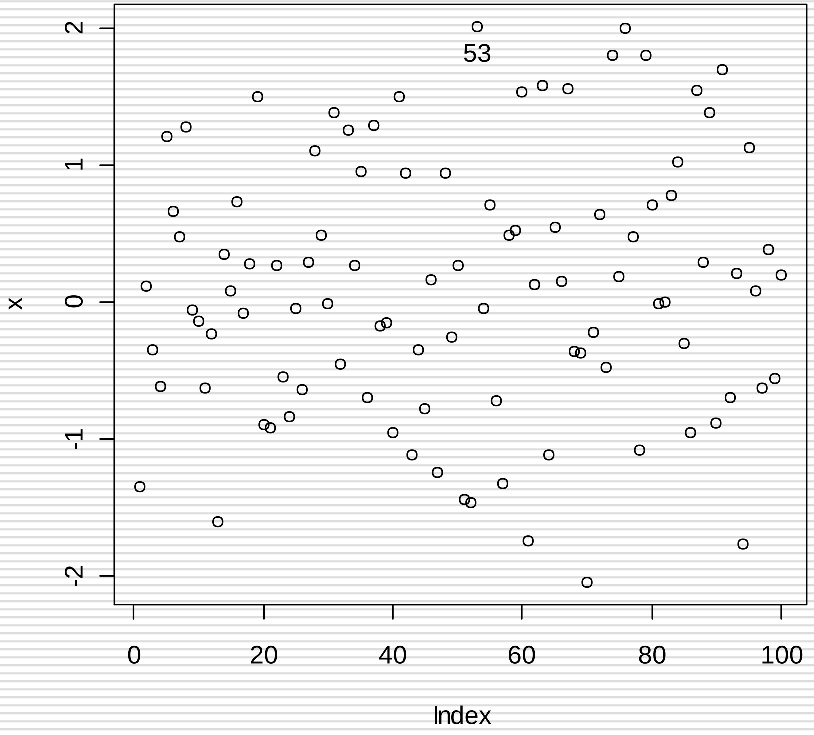


Identify Points in a Scatter plot

```
> plot(x)
> identify(x, n=1)
[1] 53
```

Identify Points in a Scatter plot





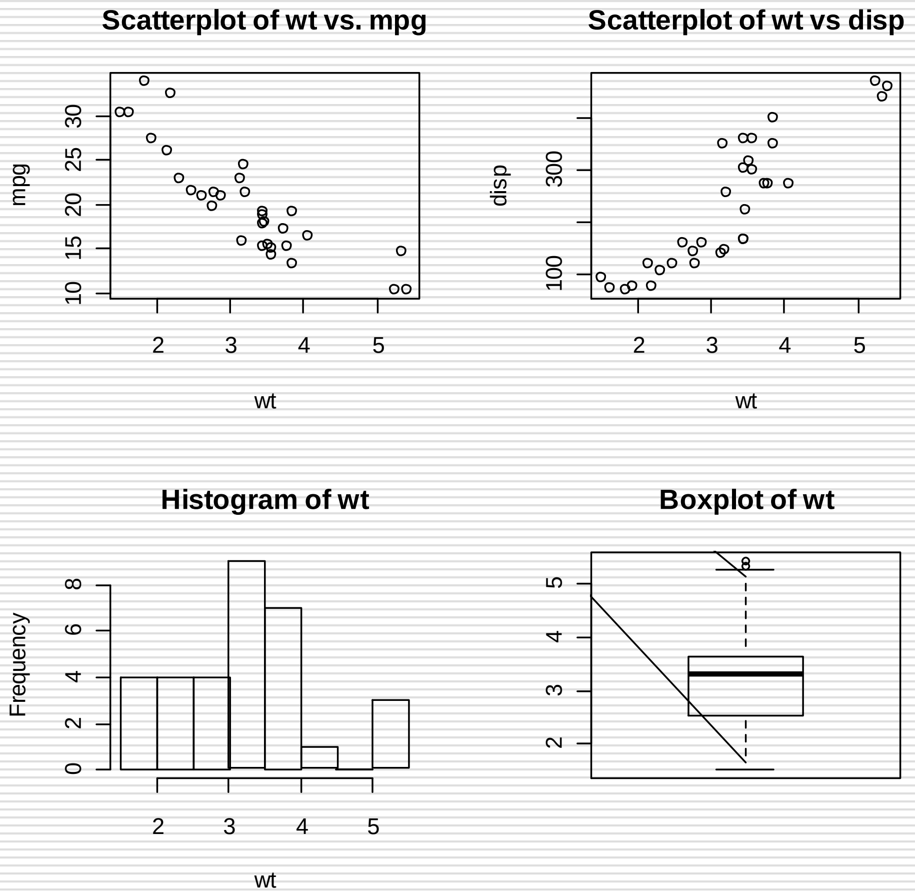
Multiple Graphs

- **R** makes it easy to combine multiple plots into one overall graph, using the **par( )** function.
- With the **par( )** function, you can include the option **mfrow=c(***nrows*, *ncols***)** to create a matrix of *nrows x ncols* plots that are filled in by row.
- **mfcol=c(***nrows*, *ncols***)** fills in the matrix by columns.

Multiple Graphs

```
# 4 figures arranged in 2 rows and 2 columns
> attach(mtcars) # attaches dataset mtcars
> par(mfrow=c(2,2))
> plot(wt,mpg, main="Scatterplot of wt vs. mpg")
> plot(wt,disp, main="Scatterplot of wt vs disp")
> hist(wt, main="Histogram of wt")
> boxplot(wt, main="Boxplot of wt")
```

Multiple Graphs



Graphs in Higher Dimensions

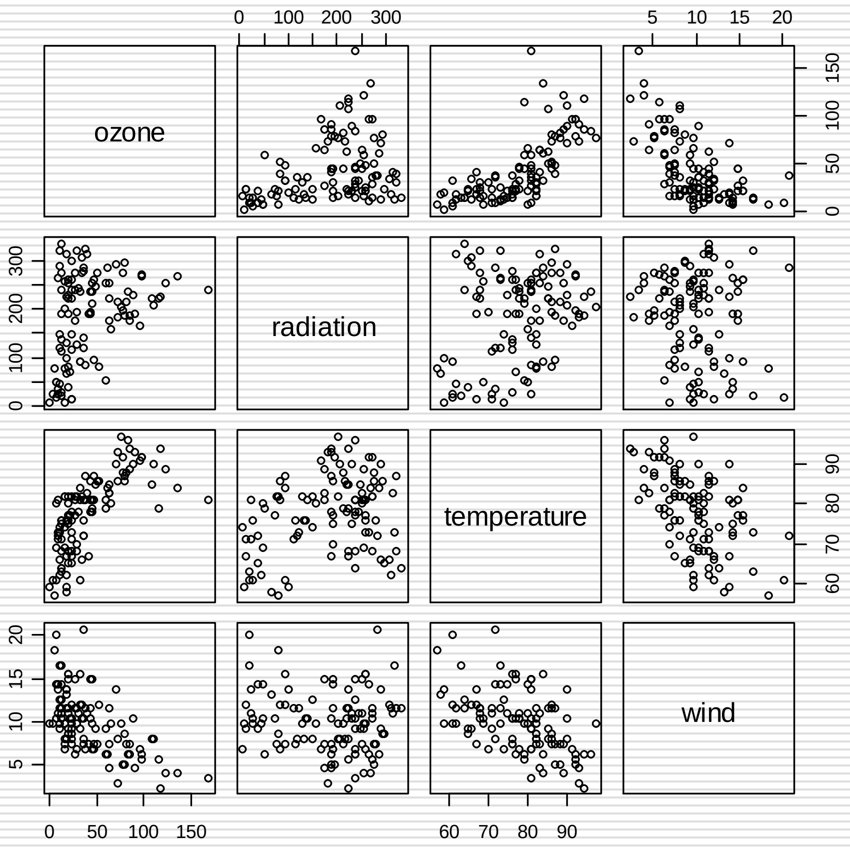
- Install and Download library ElemStatLearn
- From the Menu or Type:

```
> install.packages("ElemStatLearn”)
> library (ElemStatLearn)
```

- Data-set ozone:

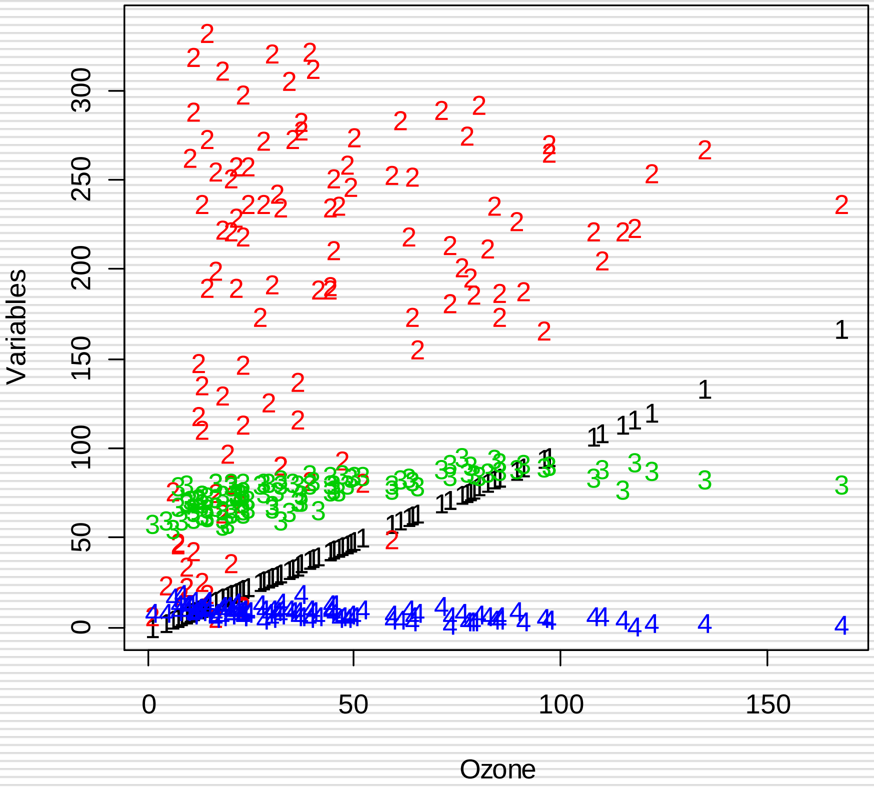
```
> data(ozone)
> pairs(ozone)
```

Scatterplot Matrices



Plot columns of matrices

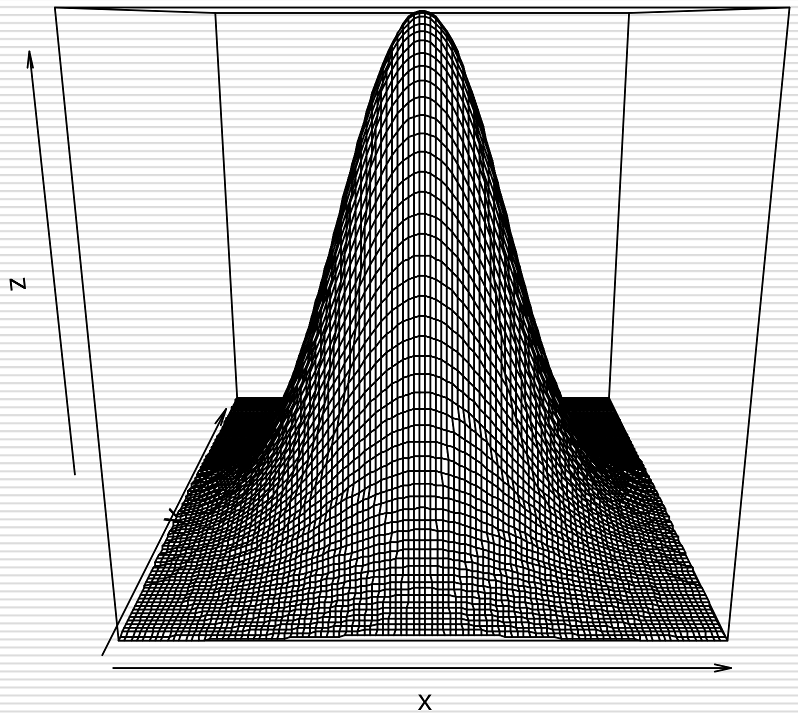
```
> matplot(ozone$ozone,
    ozone, xlab="Ozone",
    ylab="Variables")
```



Perspective Plot

```
> f <- function(x, y)   #the standard bivariate
    # normal density
   {
    z<-(1/(2*pi))*exp(-0.5*(x^2+y^2))
   }
> y <- seq(-3,3, length=100)
> x <-seq(-3,3, length=100)
> z<-outer(x,y,f)   #compute density for all x,
    y
> persp(x,y,z)
```

Perspective Plot

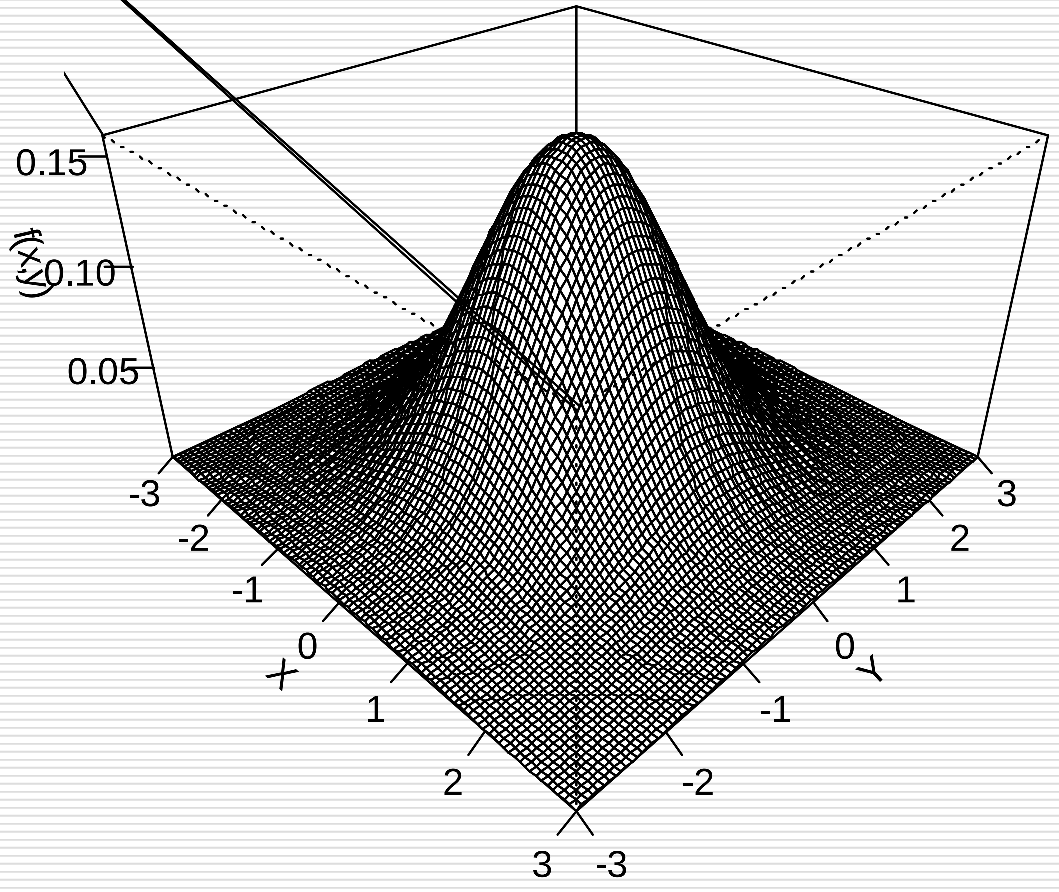


Perspective Plot

```
> persp(x,y,z, theta=45, phi=30,
    expand=0.6, ticktype="detailed", xlab="X",
    ylab="Y", zlab="f(x,y)")
```

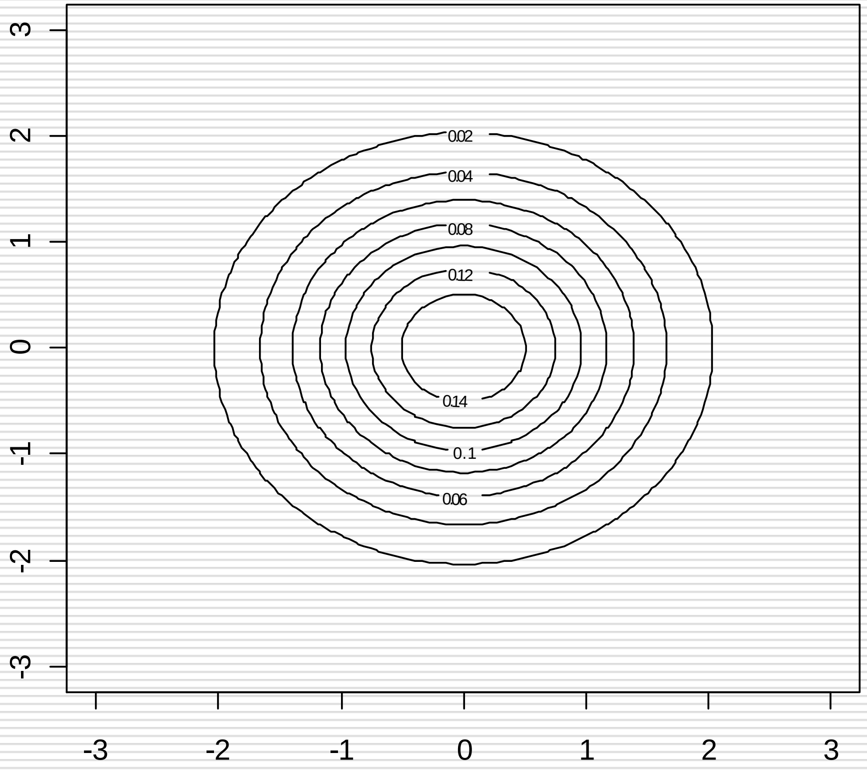
- **theta, Phi**: angles defining the viewing direction. **theta** gives the azimuthal direction and **phi** the colatitude.
- **expand**: a expansion factor applied to the z coordinates. Often used with 0 < expand < 1 to shrink the plotting box in the z direction.
- **ticktype**: character: "simple" draws just an arrow parallel to the axis to indicate direction of increase; "detailed" draws normal ticks as per 2D plots.

Perspective Plots



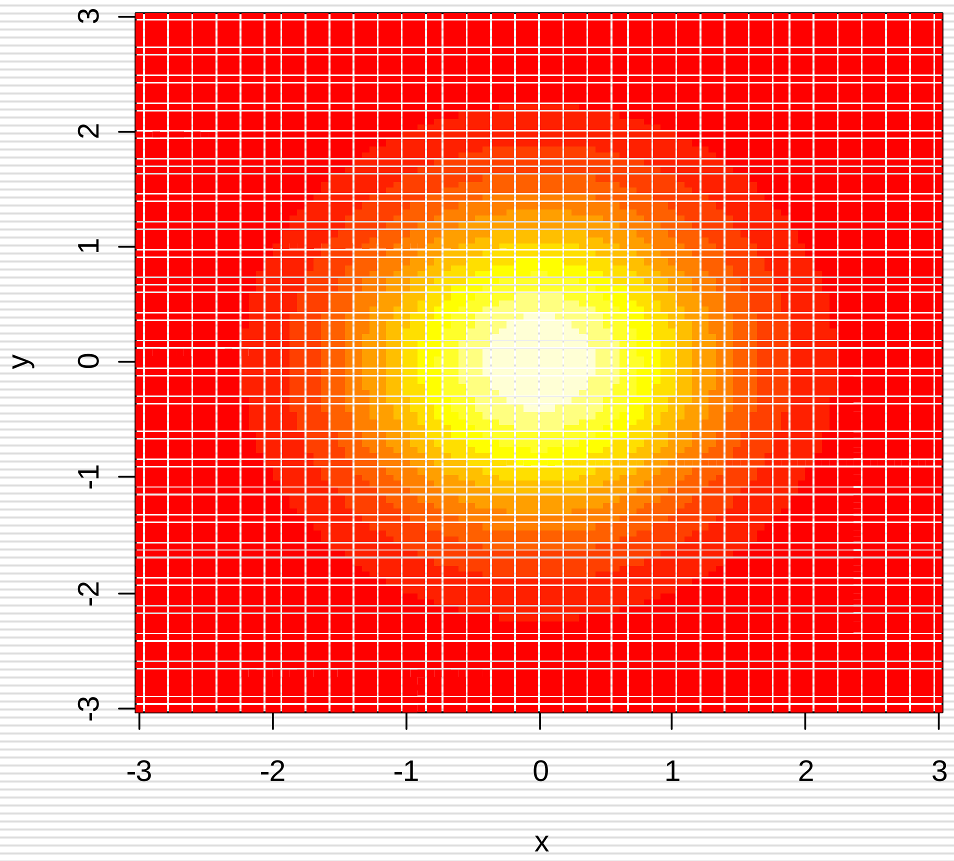
Contour Plots

```
> contour(x,y,z)
```



Color Image

```
> image(x,y,z)
```



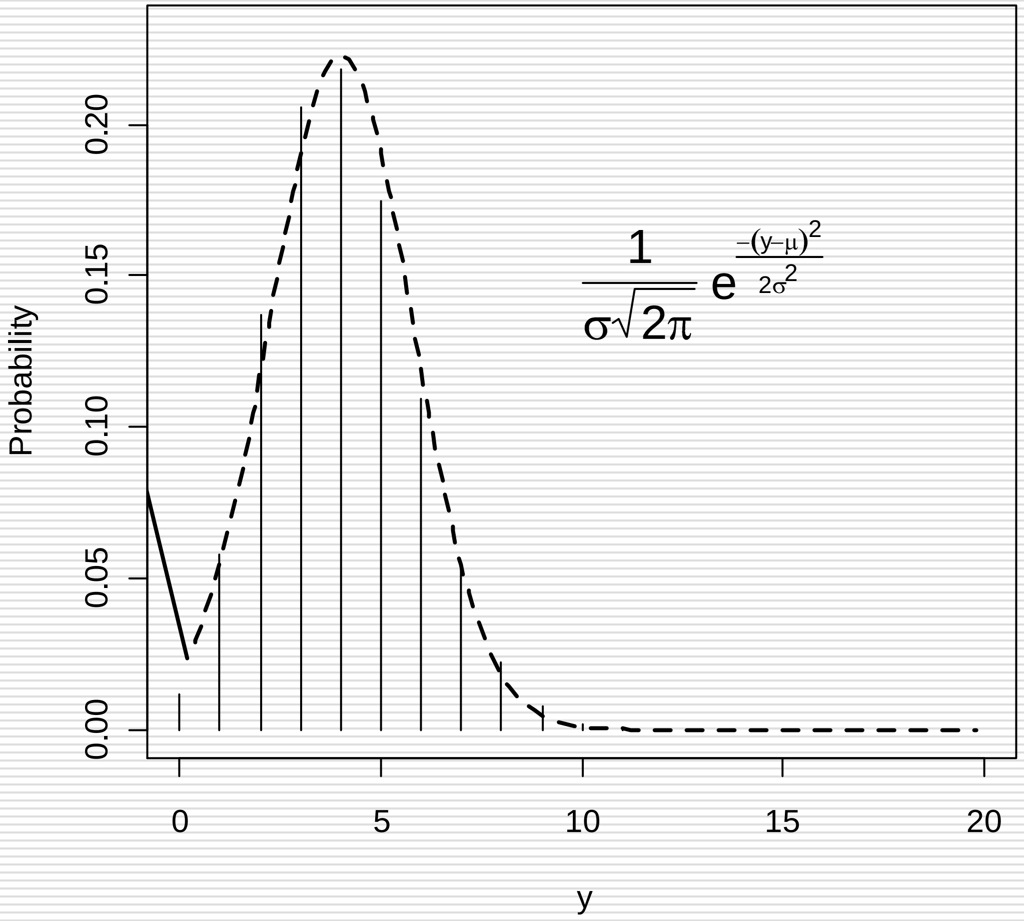
Mathematical Annotation

- If the text argument to one of the text-drawing functions (**text, mtext** & **axis**) in R is an expression, the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules. Expressions can also be used for titles, subtitles and x- and y-axis labels (but not for axis labels on persp plots).
- For more info type ?plotmath in R.

Mathematical Annotation

```
> n<-20
> p<-0.2
> y<-0:20
> pr<-dbinom(y,n,p)
> plot(y,pr,type="h", xlim=c(0,20), ylim=c(0,0.23),
   ylab="Probability")
> mu = n * p; sigma = sqrt(n * p * (1 - p))
> curve(dnorm(x, mu, sigma), add=TRUE, lwd=2, lty=2)
> text(13, 0.15, expression(paste(frac(1,
   sigma*sqrt(2*pi)), " ", e^{frac(-(y-mu)^2,
   2*sigma^2)})), cex = 1.5)
```

Mathematical Annotation



Conclusion

useR!