

Introduction to R

Statistical Programming Language

12/8/2014



What is R?

- R is a computer language, an environment for statistical computing, graphics, and much more
- It is open source with great flexibility and power gained from contributions by many users
- It allows anyone using any operating system to reproduce your work from data to finished analysis
- It is script-based (text computer code) and not GUI-based (point and click with menus)



What is R

- An effective data handling and storage facility
- A large, integrated collection of tools for data analysis
- A large and highly flexible collection of graphing facilities for data display
- A well-developed and relatively simple programming language

History of R

- Ross Ihaka and Robert Gentleman created R in 1993, in University of Auckland, New Zealand.
- R language is the programming language that you write your commands and run in.
- R is the successor to the S, the statistics language from Bell Labs in 1976.
- Free and better than SPSS

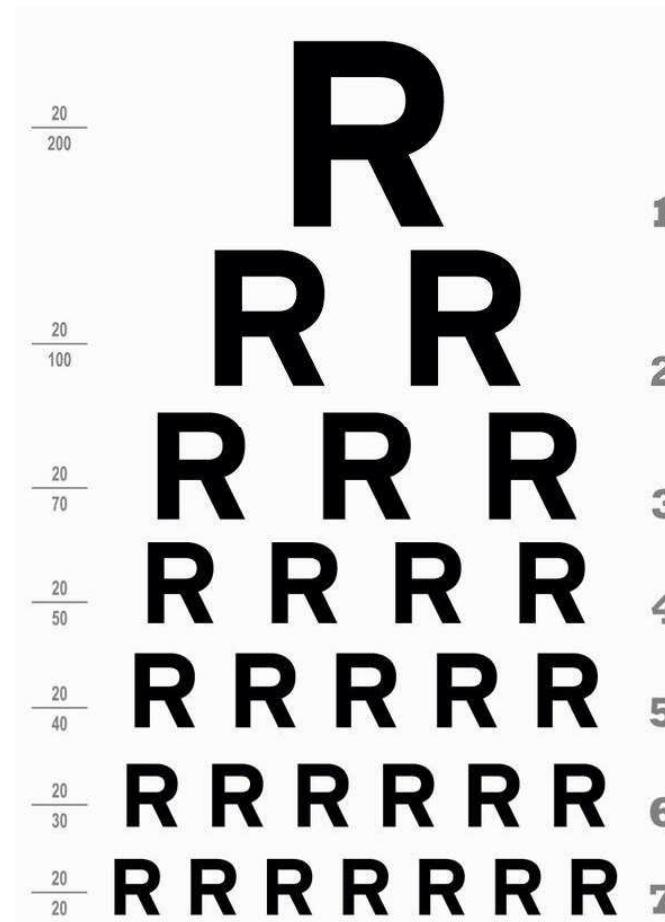
Proprietary and Open source software used for data analysis



What R is not

- It is not fast (C++ and Python are much faster)
- There is a limit to the size of data that can be processed
- There is a learning curve
- Debugging is difficult

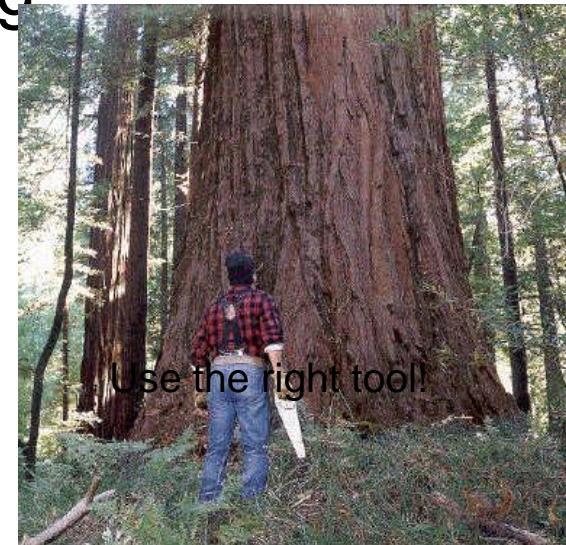
Pirate eye-chart



Posted by @utterben on Twitter

What about Excel?

- Excel allows quick prototyping
- Data manipulation is easy
- No concept of missing data
- Can see what is happening
- But: graphics are poor
- Looping is hard
- Limited statistical packages
- Inflexible
- Many things not possible in Excel

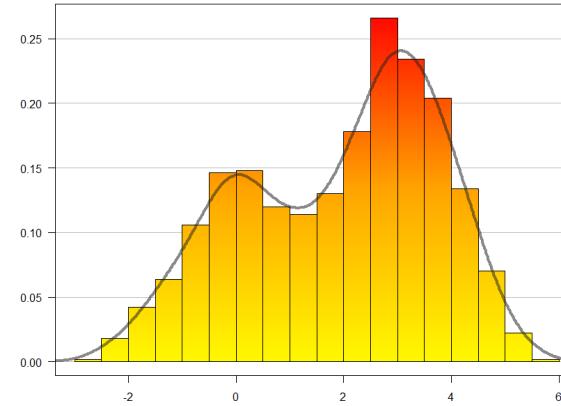


Strengths of R

- Comprehensive set of statistical analysis techniques
 - Classical statistical tests
 - Linear and nonlinear modeling
 - Time-series analysis
 - Classification and cluster analysis
 - Spatial statistics
 - Bayesian statistics
- Almost all statistical technique is either already in R, or in a user package

What are the strengths of R?

- Completely open-source
 - Users contribute and create new packages
 - Existing R functions can be edited and expanded
 - Free
 - Huge community of scientists using R
 - Easy to replicate your work from data to finished product
- Publication-quality graphics
 - Many default graphics
 - Full control of graphics
 - Make even rudimentary plots vibrant and exciting

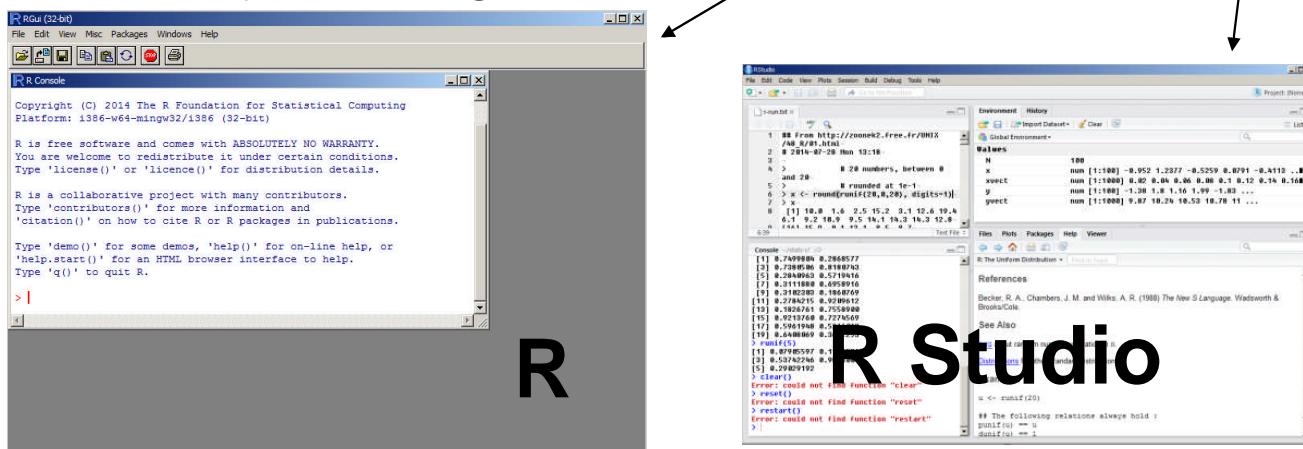


Learning R

- R is a programming language, the learning curve can be steep
- Very rewarding to become fluent: you can do more
- Be patient and creative.
- Try and practice, use Google.
- Lots of help files, online sources, books/

Installing R

- Google “*R stats windows download*” (32 and 64bits)
- Download R and R-Studio
- Right click>Run as Admin
- Install R and R-Studio
- Start R by clicking on it

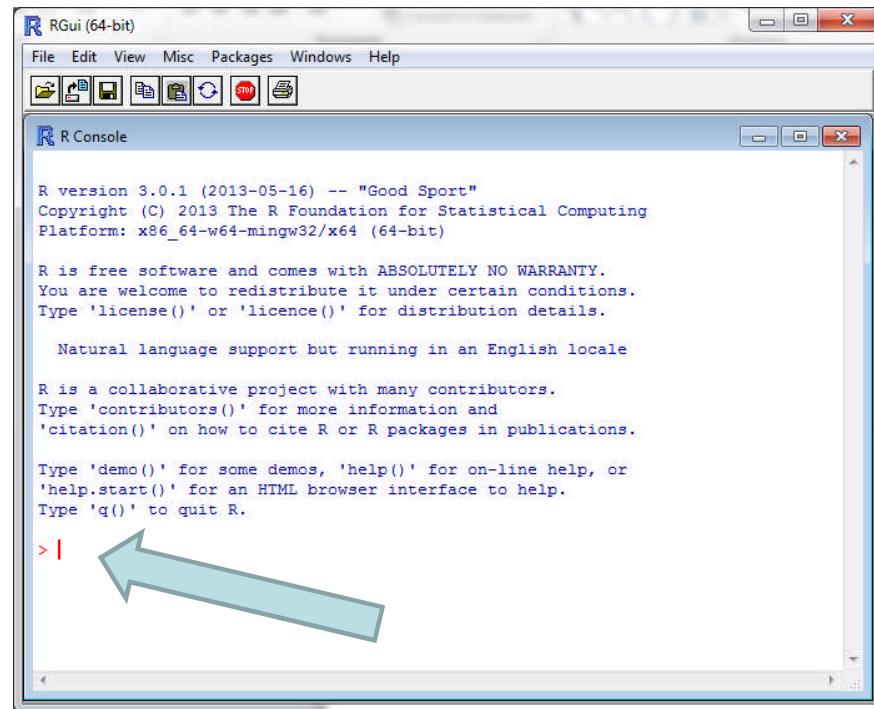


Workflow in R

1. Read Data into R
2. Analyze Data
3. Visualize Data
4. Make Conclusions from Data

Start R

- Find the standalone R program 
- Open it
- Enter commands at the > sign, e.g.
`> 2 + 4`
`> x <- 7`
`> x + 19`



R Command Prompt '>'

1. Start R by clicking on its icon

```
> # Comment lines are ignored by R  
> 2+2 # You type commands at the R prompt.  
[1] 4 # Result '4' printed by R, ignore [1].
```

```
> 1+1/2+1/3+2/3  
[1] 2.5
```

What is R doing?

```
> 2+4
```

The [1] means the first element of a vector
Even a single number in R is a vector, so "6" is a vector of size 1

```
[1] 6
```



```
> x <- 7
```

<- means "assign" in this case "assign the value 7 to the variable

```
> x + 19
```

Some use = for this purpose but it is frowned upon

```
[1] 26
```

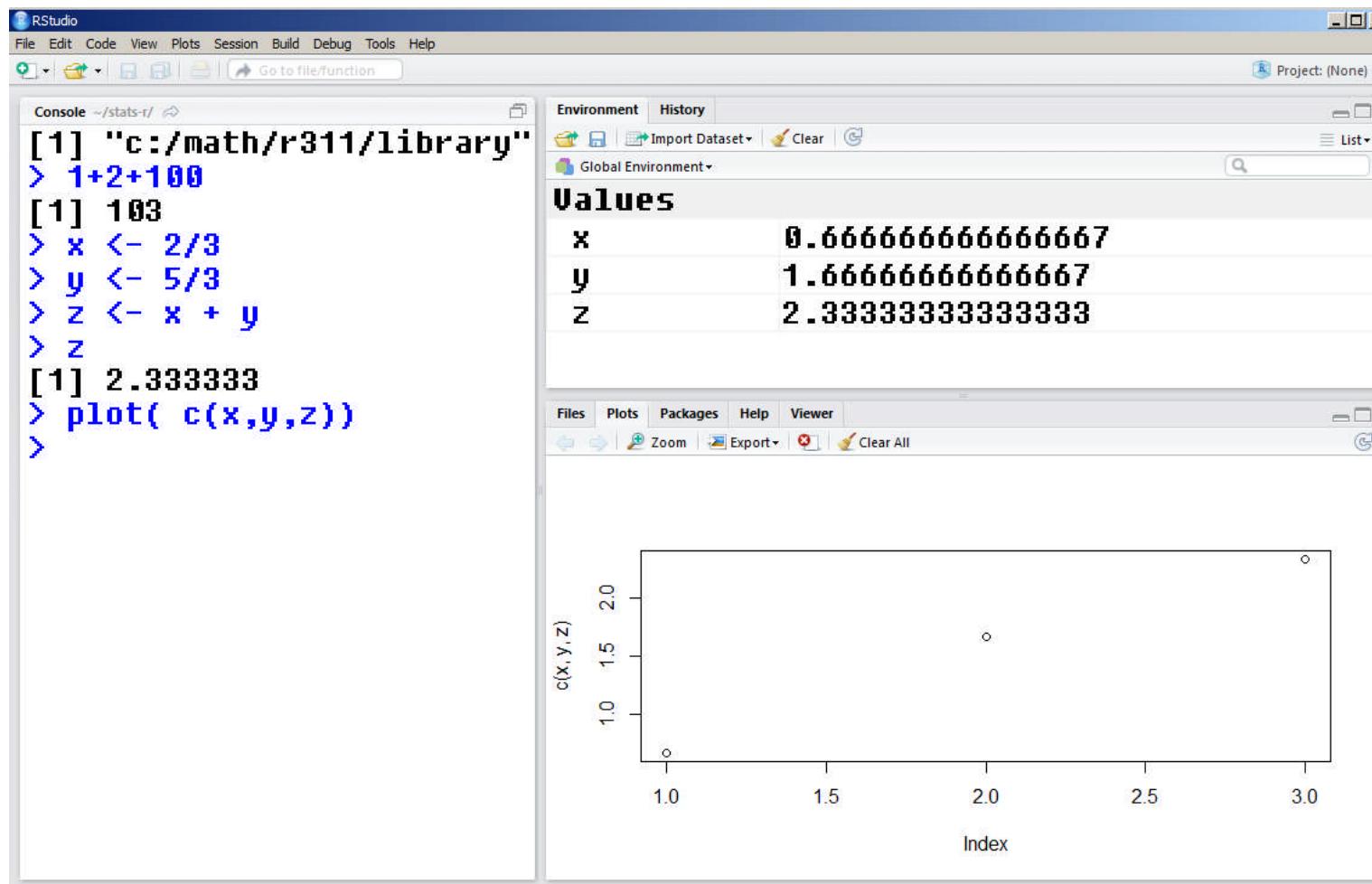
Adding 19 to x gives the expected value of 26


```
> X + 10
```

Error: object 'X' not found

X is not the same as x
R is **case sensitive**: upper case letters are different to lower case letters

R as a calculator



R as a Scientific calculator

```
> 1+1/2+1/3+2/3
```

```
[1] 2.5
```



```
> 1/0 # undefined to divide by zero
```

```
[1] Inf # Infinite
```

```
> 0/0
```

```
[1] NaN # undefined, not-a-number
```

```
> sqrt(2i) # Complex numbers
```

```
[1] 1+1i
```

Simple R commands

```
> 2+2
```

```
[1] 4
```

```
> 3^2
```

```
[1] 9
```

```
> sqrt(25)
```

```
[1] 5
```

```
> 2*(1+1)
```

```
[1] 4
```

```
> 2*1+1
```

```
[1] 3
```

```
> exp(1)
```

```
[1] 2.718282
```

```
> log(2.718282)
```

```
[1] 1
```

```
> log(10, base=10)
```

```
[1] 1
```

```
> log(10
```

```
+ , base = 10)
```

```
# + is prompt for more input.
```

```
[1] 1
```

Questions

1. Is R open source or closed source?
2. Is R copyright?
3. What is GPL software license?
4. What is BSD software license?
5. Does R run on windows / Unix ?
6. Who / when / Where was R invented?
7. R is a successor to which language?

Questions?

1. Is R compiled or a scripting language?
2. What's a command prompt?
3. How to write a comment in R language?
4. What is $2+2^3$ in R?
5. What is $2/3$ in R?
6. How does R treat these: $1/0$ and $0/0$?
7. How does R evaluate $1/3$?
8. How to save a number in R for later use?

In-class exercise 1

Use R to calculate the following.

1. $1 + 2(3 + 4)$
2. $\text{Log}_e(4^3+3^{2+1})$
3.
$$\sqrt{(4+3)(2+1)}$$
4.
$$\left(\frac{1+2}{3+4}\right)^2$$
5. $\cos^2(4)+\sin^2(4)$

References

1. *Intro to R*, by **Venables** and Smith,
<http://cran.r-project.org/doc/manuals/R-intro.pdf>
<http://cran.r-project.org/manuals.html>
1. *Basic Statistics tests in R*,
<http://www.statmethods.net/stats/index.html>
2. *Advanced Probability/Statistics in R*,
http://zoonek2.free.fr/UNIX/48_R/all.html
3. *More Statistics tests in R*,
<http://www.ats.ucla.edu/stat/r/whatstat/whatstat.htm>
4. *7 Lectures on Financial Trading with R*,
<http://www.rfortraders.com/>

R Studio

6/2/2016.

Why Rstudio?

- It is tedious to write R code in the command line
- Old style: create a text file (e.g. Notepad) and copy the code you want to run, to the command line
- Much better: use RStudio. Why?
 - Multiple files
 - View variable values, color coding
 - Built-in help
 - Quick running of code
 - Easy file handling
 - Easy package installation
 - Many other reasons

R Studio

The screenshot shows the R Studio interface with several overlaid text annotations:

- File viewer**: Located in the top-left corner of the code editor area.
- Console with Command Prompt to type Commands to R**: Located in the bottom-left corner of the console area.
- > 2 + 2 + 100**: A mathematical expression displayed in the console.
- 104**: A large orange number at the bottom left.
- Variables and values**: A blue text overlay on the right side of the environment pane.
- dts**: A blue text overlay on the right side of the plot area.
- Graphs plotted by R**: A blue text overlay on the right side of the plot area.

The R Studio interface includes:

- File**, **Edit**, **Code**, **View**, **Plots**, **Session**, **Build**, **Debug**, **Tools**, **Help** menu bar.
- forecast.R** file open in the code editor.
- Environment** and **History** tabs in the top-right pane.
- Data** section showing objects: **data** (An xts object), **data1** (An xts object), and **diwali** (122 obs. of 102 variables).
- Values** section showing **dates** (2004-01-01) and **dts** (chr [1:122, 1:4] "-0.387" "-0.386" ...).
- Files**, **Plots**, **Packages**, **Help**, **Viewer** tabs in the bottom navigation bar.
- Console** tab showing R commands and their output.
- Plots** tab showing a time series plot titled "dts" with peaks over time from 2004 to 2014.

RStudio

File Edit View Project Workspace Plots Tools Help

Scripts (files with R code)

```
1 library(ggplot2)
2
3 view(diamonds)
4 summary(diamonds)
5
6 summary(diamonds$price)
7 aveSize <- round(mean(diamonds$carat), 4)
8 clarity <- levels(diamonds$clarity)
9
10 p <- qplot(carat, price,
11             data=diamonds, color=clarity,
12             xlab="carat", ylab="price",
13             main="Diamond Pricing")
14
```

14:1 (Top Level) R Script

R console (results from running R code)

```
x
Min. : 0.000   Min. : 0.000   Min. : 0.000
1st Qu.: 2.910   Median : 3.530   Mean  : 3.539
Median : 3.530   3rd Qu.: 4.040   Max.  :31.800
3rd Qu.: 4.040
Max.  :31.800
```

> summary(diamonds\$price)
 Min. 1st Qu. Median Mean 3rd Qu. Max.
 326 950 2401 3933 5324 18820
> aveSize <- round(mean(diamonds\$carat), 4)
> clarity <- levels(diamonds\$clarity)
> p <- qplot(carat, price,
+ data=diamonds, color=clarity,
+ xlab="Carat", ylab="Price",
+ main="Diamond Pricing")
>
> format.plot(plot=p, size=23)
> |

Workspace History

Data

values

Functions

Objects you have created

Plots and help

Clarity

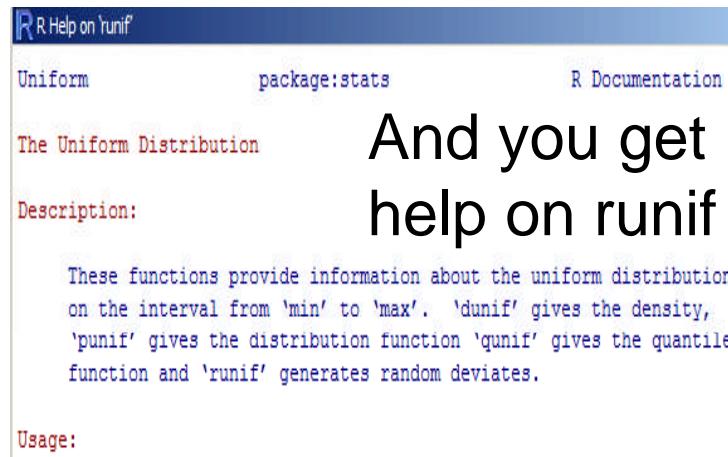
- I1
- SI2
- SI1
- VS2
- VS1
- VVS2
- VVS1
- IF

Files Plots Packages Help

Zoom Export Clear All

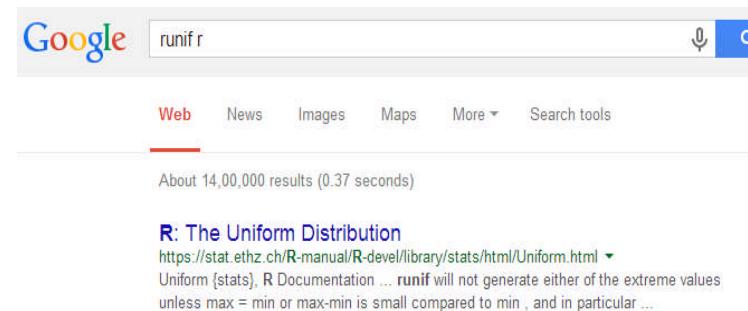
Help in R Studio

> ?runif



And you get
help on runif

Or google
R statistics runif



Help from within R

- Getting help for a function

```
> help("log")  
> ?log
```

- Searching across packages

```
> help.search("logarithm")
```

- Finding all functions of a particular type

```
> apropos("log")  
[7] "SSlogis" "as.data.frame.logical" "as.logical"  
     "as.logical.factor" "dlogis" "is.logical"  
[13] "log" "log10" "log1p" "log2" "logLik" "logb"  
[19] "logical" "loglin" "plogis" "print.logLik" "qlogis"  
     "rlogis"
```

R: Logarithms and Exponentials ▾ Find in Topic

log {base}

R Documentation

Logarithms and Exponentials

Description

What the function does in general terms

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

`log1p(x)` computes $\log(1+x)$ accurately also for $|x| \ll 1$ (and less accurately when x is approximately -1).

`exp` computes the exponential function.

`expm1(x)` computes $\exp(x) - 1$ accurately also for $|x| \ll 1$.

Usage

How to use the function

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
expm1(x)
```

Arguments

What does the function need

- `x` a numeric or complex vector.
- `base` a positive or complex number: the base with respect to which logarithms are computed.
Defaults to `e=exp(1)`.

Details

All except `logb` are generic functions: methods can be defined for them individually or via the [Math](#) group generic.

`log10` and `log2` are only convenience wrappers, but logs to bases 10 and 2 (whether computed via `log` or the wrappers) will be computed more efficiently and accurately where supported by the OS. Methods can be set for them individually (and otherwise methods for `log` will be used).

`logb` is a wrapper for `log` for compatibility with S. If (S3 or S4) methods are set for `log` they will be dispatched. Do not set S4 methods on `logb` itself.

All except `log` are [primitive](#) functions.

?log

R: Logarithms and Exponentials ▾ Find in Topic

Value

What does the function return

A vector of the same length as `x` containing the transformed values. `log(0)` gives `-Inf`, and `log(x)` for negative values of `x` is `NaN`. `exp(-Inf)` is 0.

For complex inputs to the log functions, the value is a complex number with imaginary part in the range $[-\pi, \pi]$: which end of the range is used might be platform-specific.

S4 methods

`exp`, `expm1`, `log`, `log10`, `log2` and `log1p` are S4 generic and are members of the [Math](#) group generic.

Note that this means that the S4 generic for `log` has a signature with only one argument, `x`, but that `base` can be passed to methods (but will not be used for method selection). On the other hand, if you only set a method for the [Math](#) group generic then `base` argument of `log` will be ignored for your class.

Source

`log1p` and `expm1` may be taken from the operating system, but if not available there are based on the Fortran subroutine `dlnrel` by W. Fullerton of Los Alamos Scientific Laboratory (see <http://www.netlib.org/slatec/fnlib/dlnrel.f> and (for small `x`) a single Newton step for the solution of `log1p(y) = x` respectively).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for `log`, `log10` and `exp`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (for `logb`.)

See Also

Discover other related functions

[Trig](#), [sqrt](#), [Arithmetic](#).

Examples

Sample code showing how it works

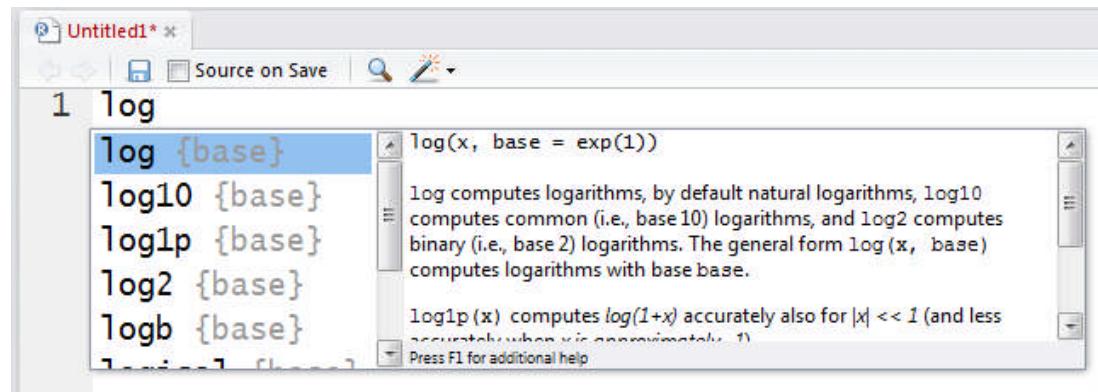
```
log(exp(3))
log10(1e7) # = 7

x <- 10^{-(1+2*1:9)}
cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

[Package `base` version 3.0.1 [Index](#)]

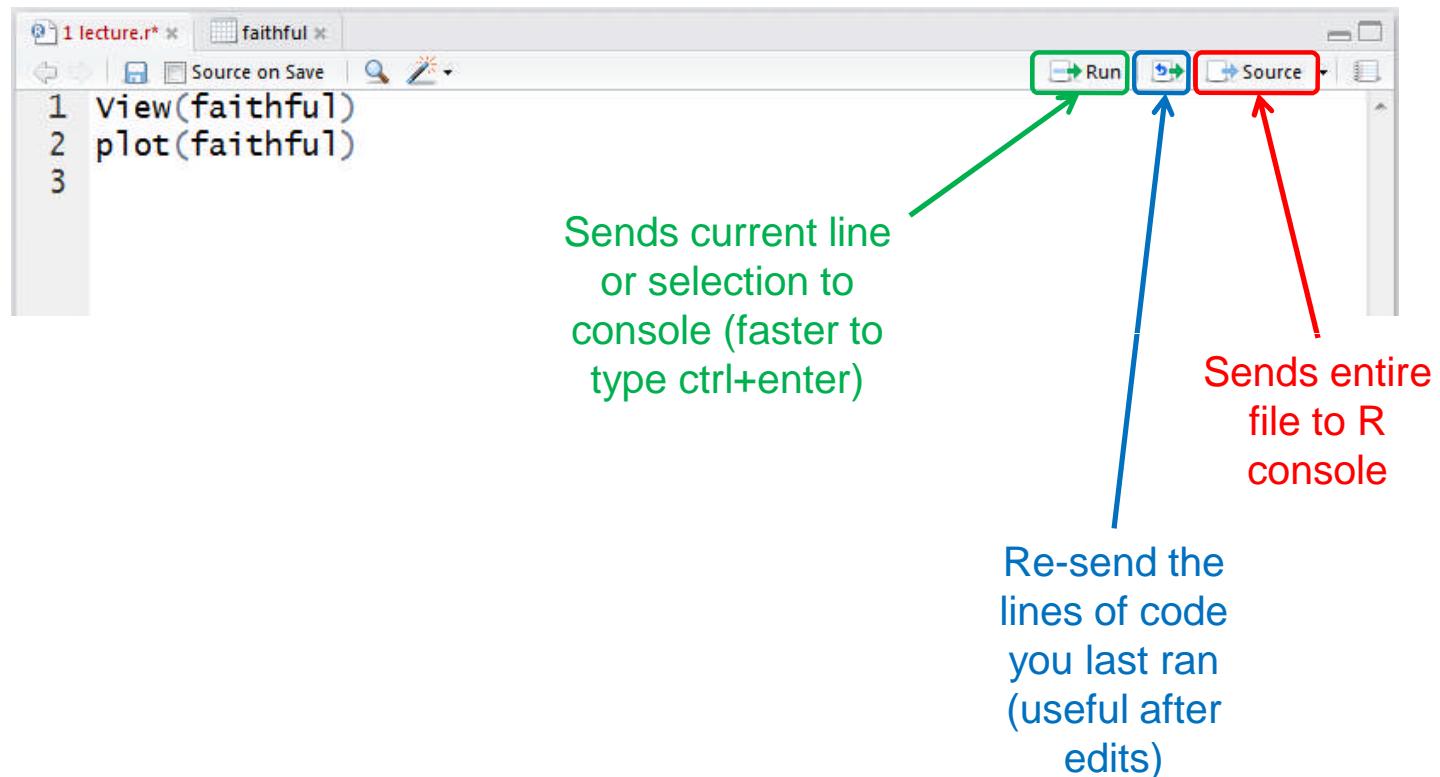
RStudio quick help

- Start typing in the Scripts window (top-left)
- Press <tab> and a list of available functions starting with those letters appears, plus help



Try typing `log(` and then pressing <tab>

RStudio tips



R scripts

- A text file (e.g. lab1.r) that contains all your R code
- Scientific method: complete record of your analyses
- Reproducible: rerunning your code is easy for you or someone else
- Easily modified and rerun
- In RStudio, select code and type <ctrl+enter> to run the code in the R console
- **SAVE YOUR SCRIPTS in your folder, disk, gmail, github**

Commenting your code (do it)

- Use “comments” to document the intention of your code
- Anything on a line after `#` is ignored by R

```
# Old Faithful geyser, Yellowstone NP  
plot(faithful)          RStudio: different color for  
                           comments
```

- Rules of thumb
 - Document the purpose of the code not how it works
 - Use good variable names
 - Assume you will remember nothing about the code when you look at it later (next week, year, decade)
 - R is very terse, so comments are essential.
 - Use git, cvs, perforce, svn for version/change tracking

R workspaces

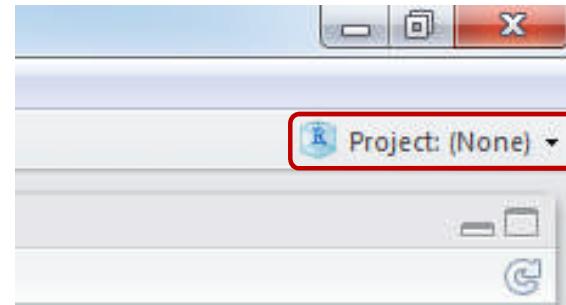
1. When you close your R session, you can save data and analyses in an R workspace
2. This saves everything run in your R console
3. Generally not recommended
 1. Exception: working with an enormous dataset
4. Better to start with a clean, empty workspace so that past analyses don't interfere with current analyses
5. `rm(list = ls())` clears out your workspace
6. Summary: save your R script, don't save your workspace

Projects in RStudio

- Getting R to figure out where the files are (directories) is key when reading in data from files
- RStudio has projects which do this in a very slick manner
- You can store different analyses in different projects and quickly switch between them
- Each project has a separate set of .r files that are open, and a separate R workspace (saved objects in console)

Using projects

- Click on top-right option
- Choose "Create Project" -> "Existing Directory" (or "New Project" if a directory does not exist)
- Select a directory, e.g. "Lectures" for me
- The project will be saved as a file in that directory called "Lectures.Rproj"
- Opening that will open the RStudio project
- Automatically sets the R working directory to that directory



References

1. Intro to R, by **Venables** and Smith,
<http://cran.r-project.org/doc/manuals/R-intro.pdf>
<http://cran.r-project.org/manuals.html>
1. Basic Statistics tests in R,
<http://www.statmethods.net/stats/index.html>
2. Advanced Probability/Statistics in R,
http://zoonek2.free.fr/UNIX/48_R/all.html
3. More Statistics tests in R,
<http://www.ats.ucla.edu/stat/r/whatstat/whatstat.htm>
4. 7 Lectures on Financial Trading with R,
<http://www.rfortraders.com/>

R Commander

Rcmdr

- 6/2/2016.

Installing Rcmdr

Google, Download and install R, R-Studio (32 or 64bit windows).

Start R-Studio and install Rcmdr

```
> install.packages("Rcmdr", dependencies=TRUE)
```

```
# Start R commander
```

```
> library(Rcmdr)
```



A screenshot of the R Commander interface within the RStudio environment. The R Commander window is open, showing its menu bar (File, Edit, Data, Statistics, Graphs, Models, Distributions, Tools, Help) and a status bar indicating '<No active dataset>'. Below the menu is a toolbar with various icons. The main area shows tabs for 'R Script' and 'R Markdown'. At the bottom of the screen, the R console output is displayed:

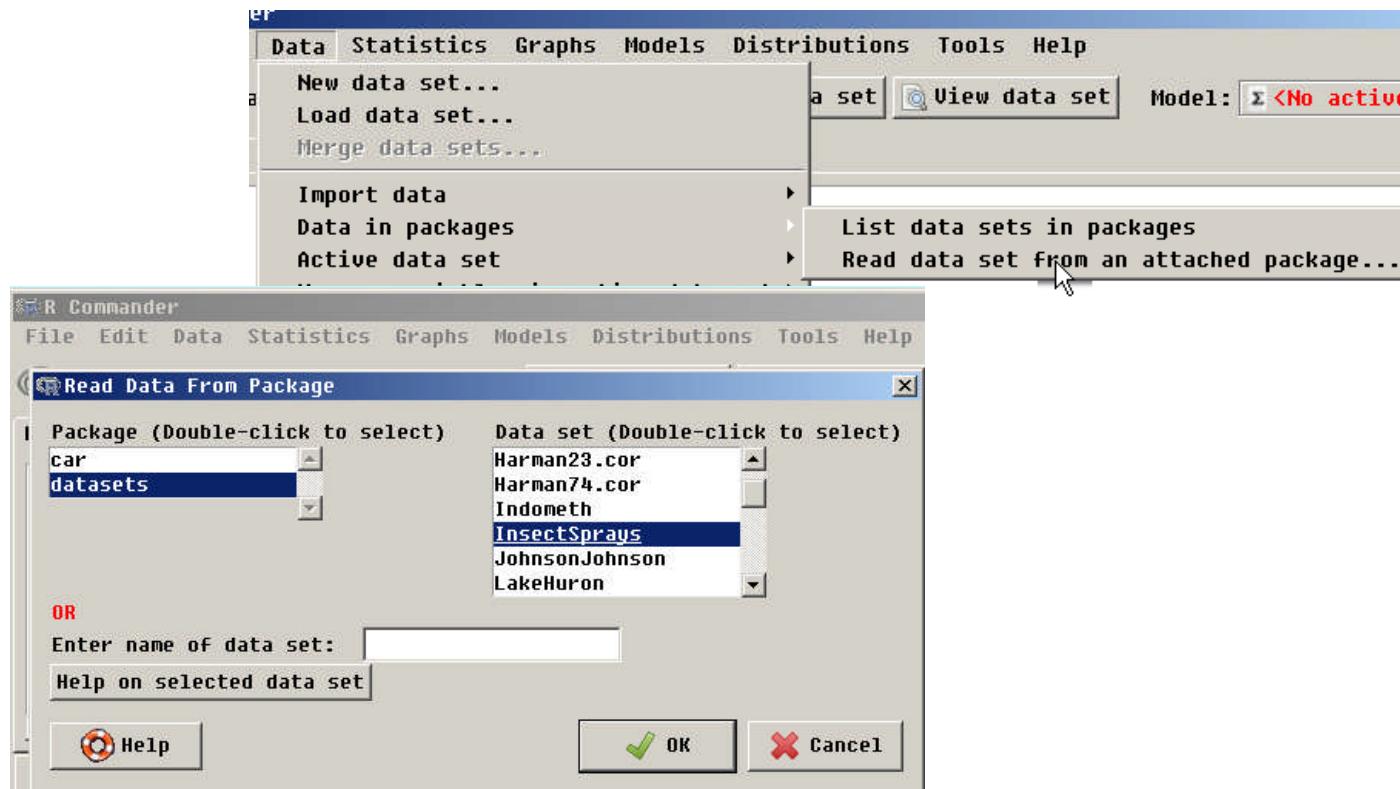
```
> library(Rcmdr)
Loading required package: spline
s
Loading required package: car
RcmdrMsg: [1] NOTE: R Commander
Version 2.0-4: Sat Feb 06 18:36:
45 2016
```

Clearing the R Workspace for new project

- R>Restart R
- R>Session>Clear Workspace
- Control-L to clear console window.

Rcmdr: Using builtin data

- library(Rcmdr)
- Data > Data in packages > Read data set..
- datasets > InsectSprays [OK]



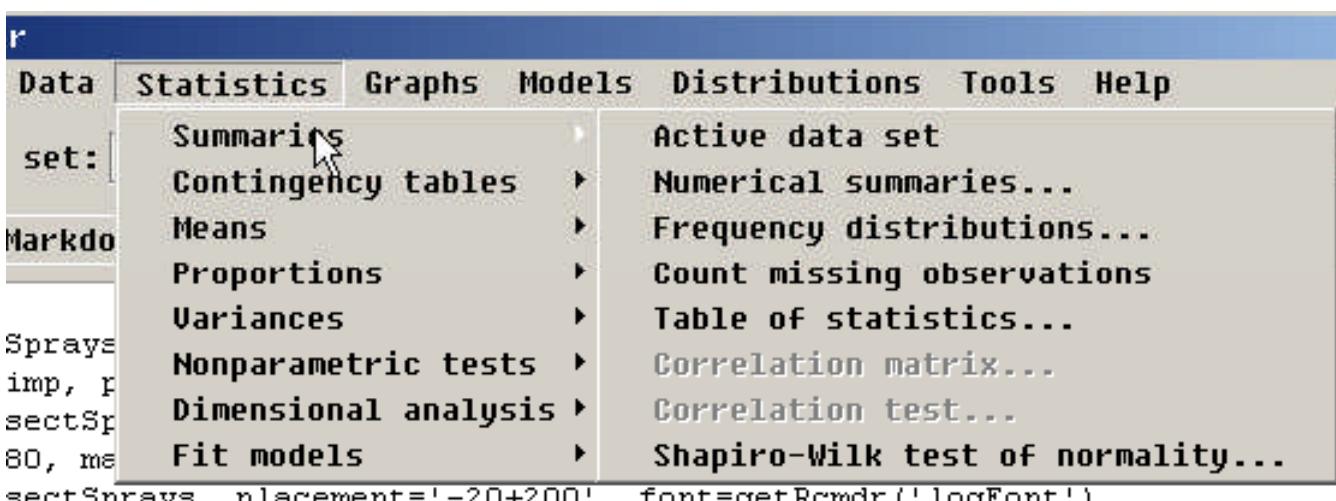
Rcmdr view data set

The screenshot shows the R Commander interface. The menu bar includes File, Edit, Data, Statistics, Graphs, Models, Distributions, Tools, and Help. The 'Data set:' dropdown is set to 'InsectSprays'. The 'Edit data set' and 'View data set' buttons are visible in the toolbar. Below the toolbar, there are tabs for 'R Script' and 'R Markdown', with 'R Script' currently selected. The main area contains R code for loading the dataset and displaying its first 10 rows. To the right, a preview window titled 'Inst... >' shows the first 10 rows of the 'InsectSprays' dataset.

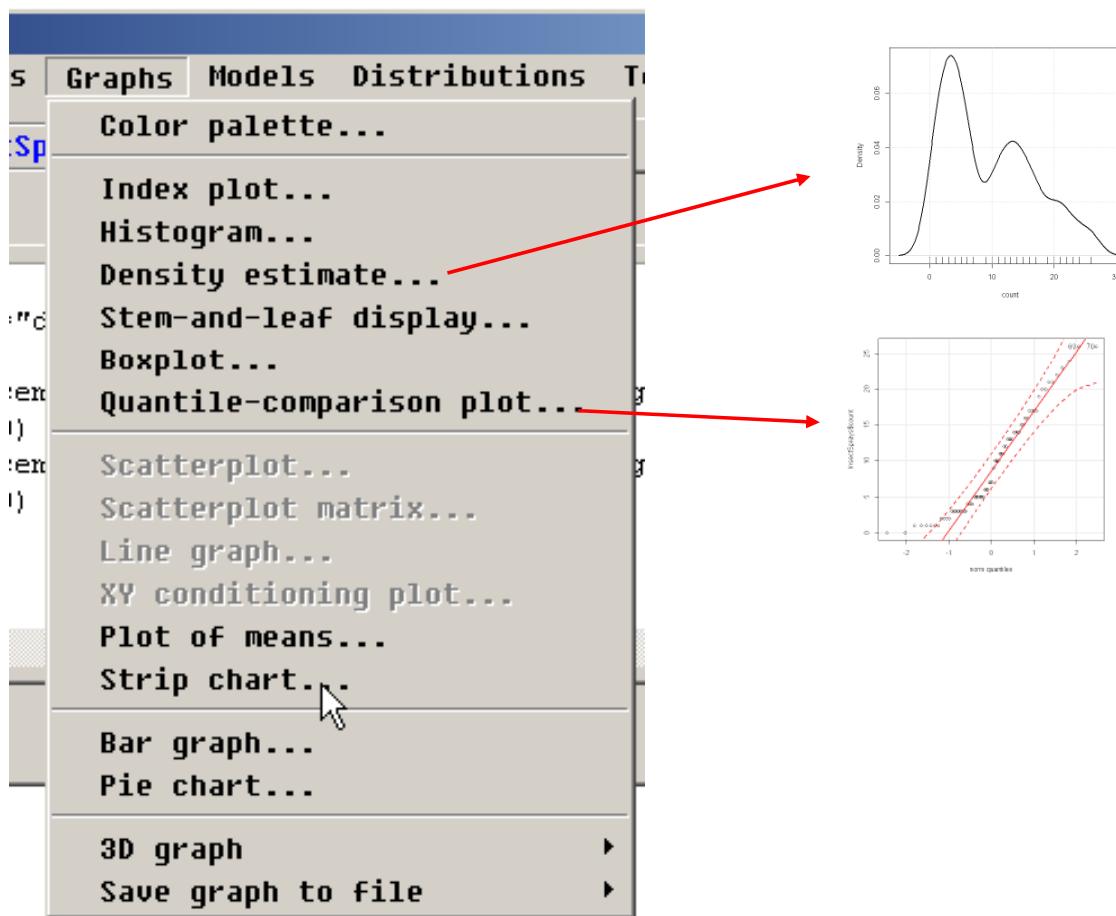
```
data(InsectSprays, package="datasets")
library(relimp, pos=4)
showData(InsectSprays, placement='<20+200', font
  maxwidth=80, maxheight=10)
showData(InsectSprays, placement='<20+200', font
  maxwidth=80, maxheight=10)
```

	count	spray
1	10	A
2	7	A
3	20	A
4	14	A
5	14	A
6	12	A
7	10	A
8	23	A
9	17	A
10	20	A

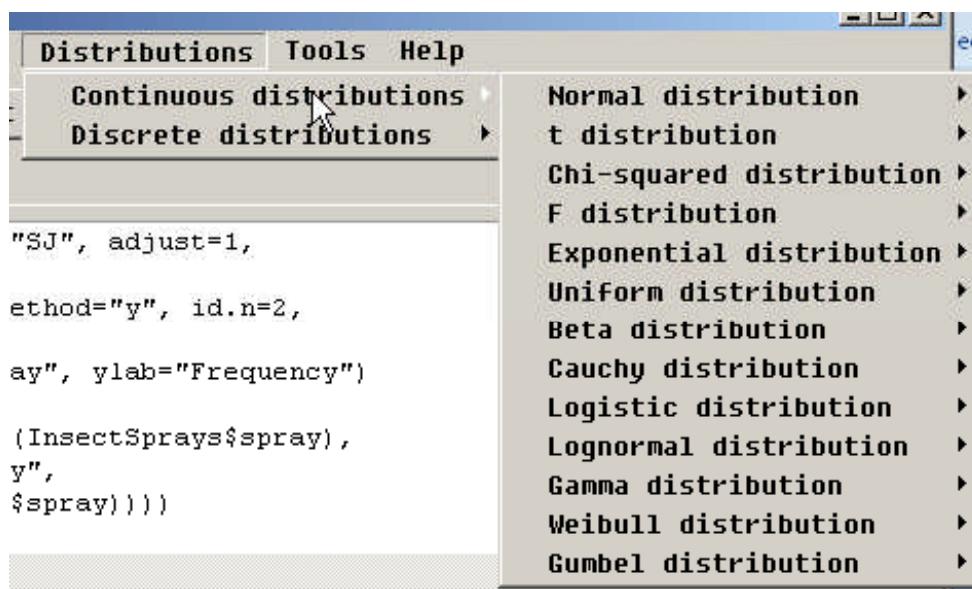
Rcmdr Statistics



Rcmdr graphs

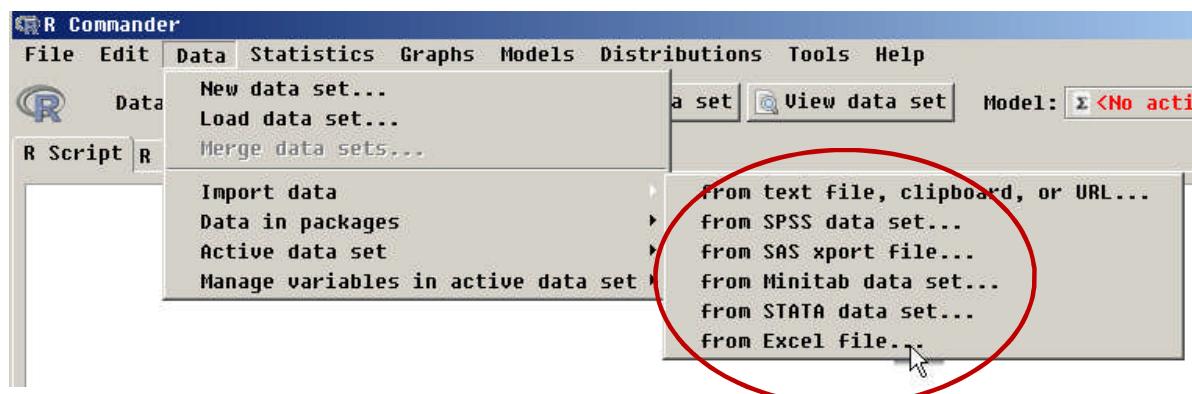


Rcmdr distributions



Read excel/spss data

- Rcmdr can load your excel / spss / text data into R, example:
- > Data > Import data > from Excel file



References

Details of R

Statistical Programming Language

4/2/2016

Vector of numbers

```
# Sequence of numbers from 1 : to 10
```

```
> 1:10
```

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

```
# Create 4 numbers and save the  
# vector in variable named u.
```

```
> u <- c(1, 4, 0, -2)
```

Sequence, vector

```
> 1:5                      # 1 2 3 4 5  
> c(1,2,3,4,5)            # 1 2 3 4 5  
> seq( 0, 4, len=3)       # 0 2 4  
> seq( 0, 4, by=2)        # 0 2 4  
  
> c(a=1, b=5, c=10) # Named vector  
a b c  
1 5 10
```

Functions on sequences

```
# Calculate: 1+2+3+4
```

```
> sum(1:4)
```

```
10
```

```
# Calculate: 9*99*999
```

```
> prod( c(9,99,999) )
```

```
890109
```

Transform a sequence

Q. How to generate $c \leftarrow (1, 1/2, 1/3)$?

A. Use `sapply(data, function)`

Example:

```
c <- sapply( 1:3, function(x) { 1/x } )
```

c is (1.0, 0.5, 0.33)

Questions

1. How would you create these vectors

$$x = (31, 41, 51, \dots, 91)$$

$$y = (-100, -99, \dots, +100)$$

2. How would you calculate this series?

$$1+2+3+4+\dots+100$$

3. Total odd numbers below 100: $1+3+5+99$

4. How would you compute $14!$ in R?

5. Compute e using $\text{sum}(n=1..12, 1/n!)$

$$e = 1+1/1!+1/2!+1/3!$$

Solutions

1. `x = seq(31,91,10) # 31,41,51,..,91`
2. `y = seq(-100,100,10) # -100,-99, ...,100`
3. `sum(1:100) # 1+2+3+..+100=5050`
4. `sum(seq(1,100,2)) # sum odds.. is 2500`
5. `prod(1:14) # 14! = 87178291200`
6. `# e = 1+1/1!+1/2!+1/3! = 2.718282`
`s<-1;for(i in 1:100){ s <- s+ 1/prod(1:i)}; s;`

Statistical functions

```
> u <- c(1, 4, 0, -2)
> mean(u)
  0.75
> sd(u); max(u); min(u); median(u); var(u)
  2.5,     4,     -2,      0.5,      6.25
> sum(u) ; length(u)
  3,     4,
```

Exercise

- Find average of 1,2,..100
- Find stddev of $1/1, 1/2, 1/3, \dots, 1/100$

Solutions

- Find average of 1,2,..100

```
> mean(1:100)
```

50.5

- Find stddev of $1/1, 1/2, 1/3, \dots, 1/100$

```
> sd(sapply(c(1:100),function(x){1/x}))
```

0.1174603

summary(data)

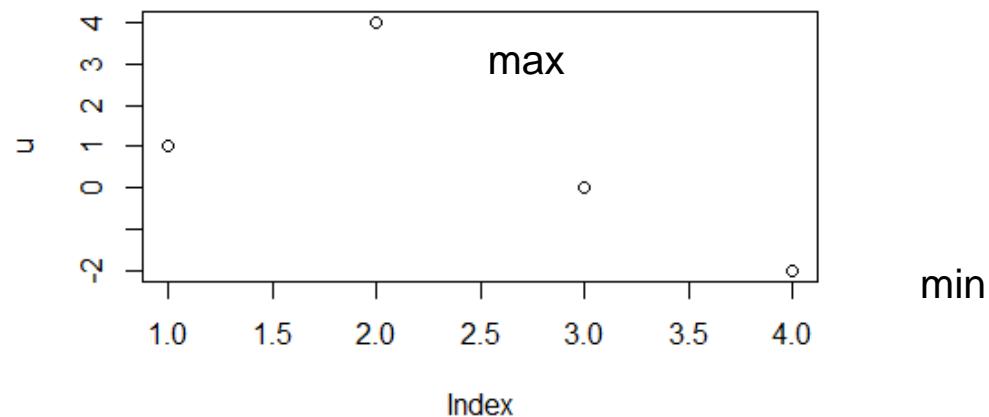
```
> u <- c(1, 4, 0, -2)
> summary(u)
Min. 1Q. Median Mean 3Q. Max.
-2.00 -0.50  0.50    0.75  1.75  4.00
```

quantile(data)

```
> u <- c(1, 4, 0, -2)  
> quantile(u)  
 0%   25%   50%   75% 100%  
-2.00 -0.50  0.50  1.75  4.00
```

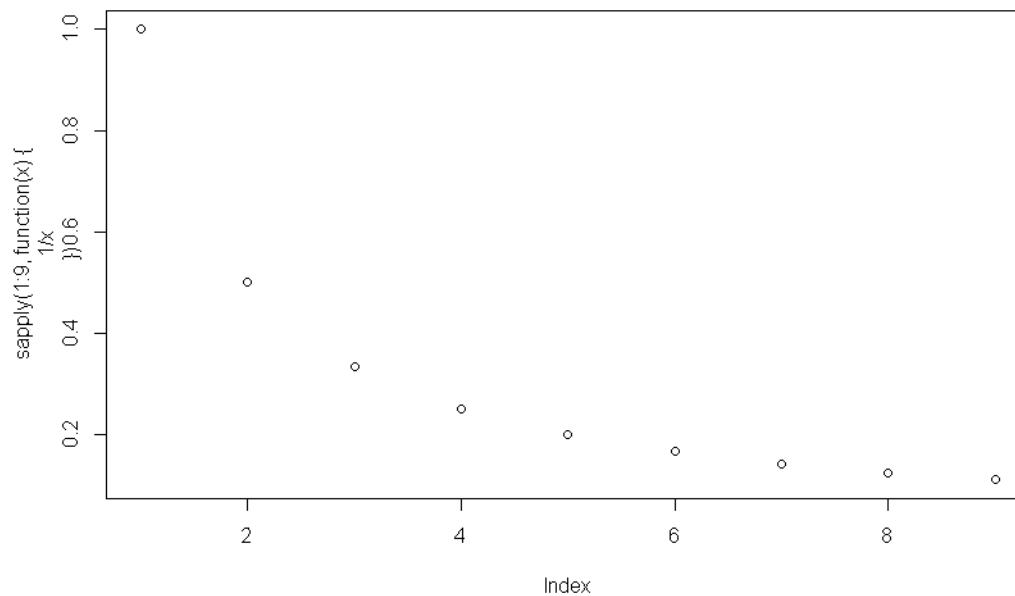
```
> quantile(u, c(0,.33,.66,1))  
 0%   33%   66% 100%  
-2.00 -0.02  0.98  4.00
```

```
plot(data)  
> u <- c(1, 4, 0, -2)  
> plot(u)
```



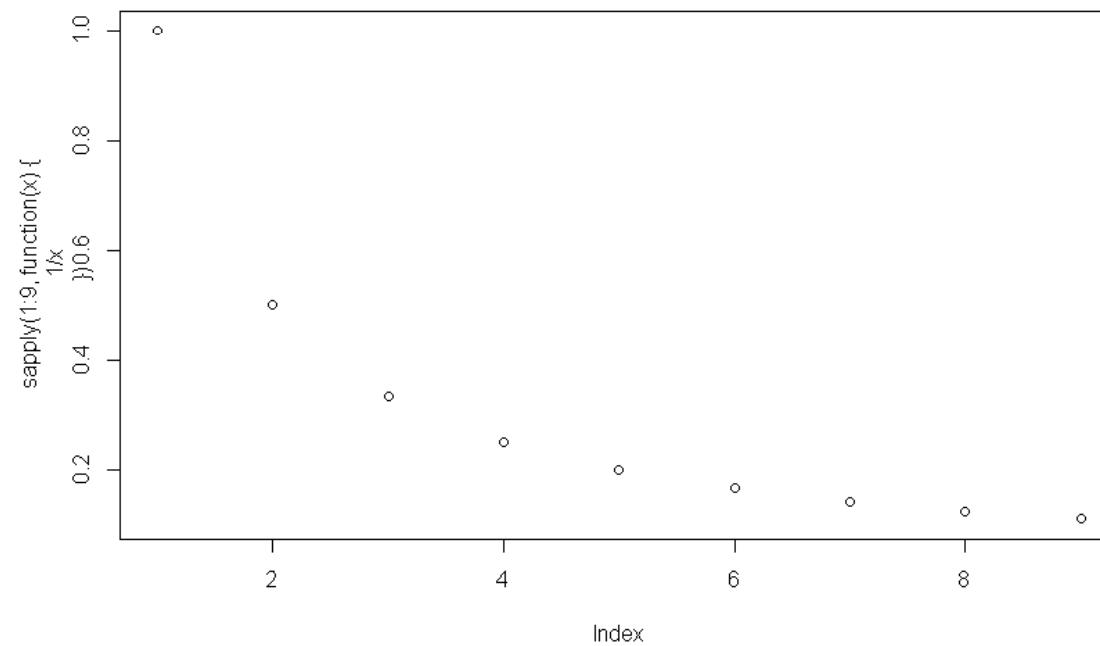
Exercise

- Plot $1, 1/2, 1/3, \dots, 1/9$



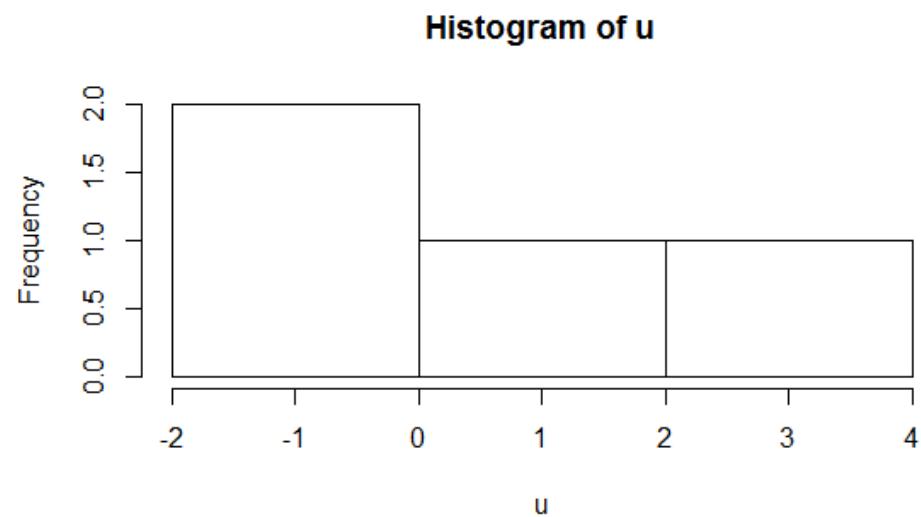
Exercise

```
# Plot 1,1/2,1/3,...1/9  
> plot( sapply(1:9, function(x){1/x}) )
```



histogram(data)

```
> u <- c(1, 4, 0, -2)  
> hist(u)
```



Not available: NA and NaN

```
> str( log( c(-1, 0, 1, 2, NA) ) )  
      NaN -Inf 0 0.693 NA
```

Warning .. NaNs produced

```
> is.finite( log(c(-1, 0, 1, 2, NA)) )  
      F F T T F
```

NA: Missing Values

```
# NA = Not Available
```

```
> x <- c(1,5,9,NA,2)
```

```
> x
```

```
1 5 9 NA 2
```

```
> is.na(x) # returns True/False
```

```
F F F T F
```

```
> x[!is.na(x)] # remove the NA from x.
```

```
1,5,9,2
```

Ignore NA in calculations

```
> x <- c(1,5,9,NA,2)  
> mean(x)  
NA      # Cannot compute mean of NA.  
> mean(x, na.rm=T) # Remove NA values  
4.25
```

Make some random numbers

```
# Make 3 random uniform numbers
```

```
> runif(3)
```

```
[1] 0.4285490 0.1428636 0.8774799
```

```
# Make 3 numbers between 5 to 10
```

```
> runif(3, 5,10)
```

```
[1] 6.749963 8.611054 8.108691
```

Random numbers

```
# Generate 3 random numbers in  
# the range 5 to 10,  
# round them to 1 decimal digit.
```

```
> round( runif(3, 5, 10), digits=1)  
[1] 5.5 9.7 9.5
```

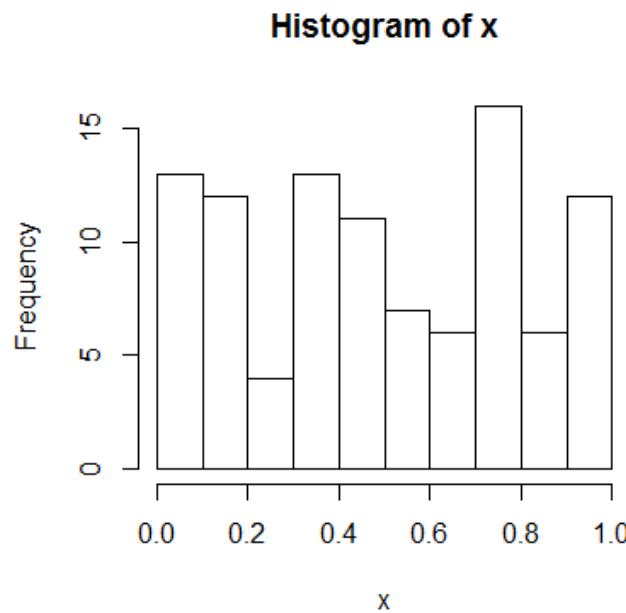
Save the numbers in variable y

```
# Save 3 random numbers in a variable y  
> y <- runif(3)  
# See what's in y  
> y  
[1] 0.1799650 0.3845684 0.1769475
```

Histogram

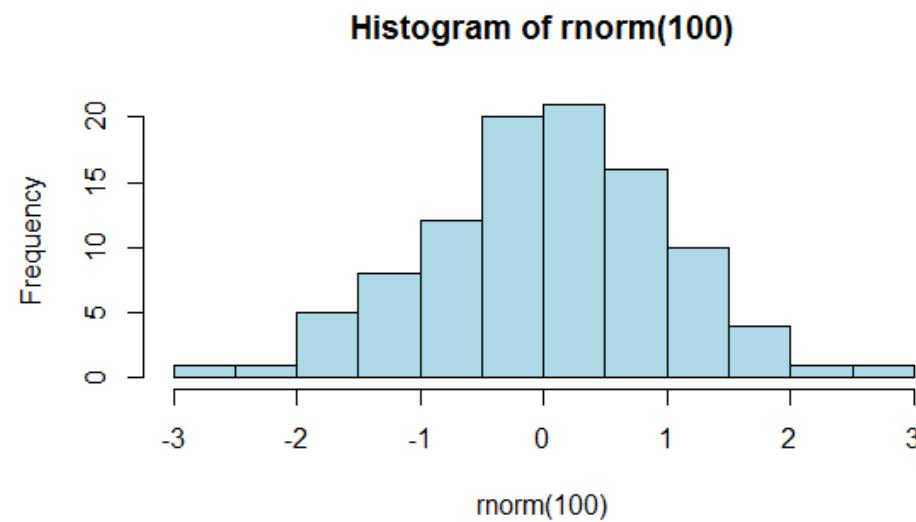
```
# Save 100 numbers in a  
# variable named x  
> x <- runif(100)
```

```
# Plot the histogram  
> hist( x )
```



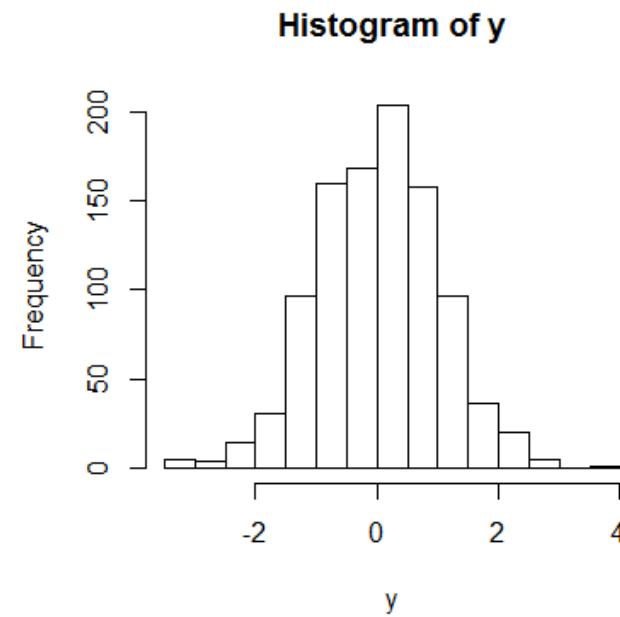
Histogram of normal random numbers in blue color

```
# 100 Normal distributed random numbers  
> hist( rnorm(100), col="light blue" )
```



Histogram of a normal distribution

```
# Generate 1000 normal distributed random numbers  
# save them in y, and make a histogram of y.  
> y <- rnorm(1000)  
> hist(y)
```



To roll a Dice (Die) 10 times.

```
> sample(1:6, 1) # one throw  
[1] 2
```



Throw it 10 times

```
> sample(1:6, 10, replace=T)  
[1] 5 6 3 2 5 5 3 4 1 6
```

Replace=T means, the same number can repeat.

Replace=F means, each number can appear only once.

Permutations (selection without replacement)

```
> sample(1:6, 6)
```

5 4 6 1 2 3

Toss a coin 10 times.

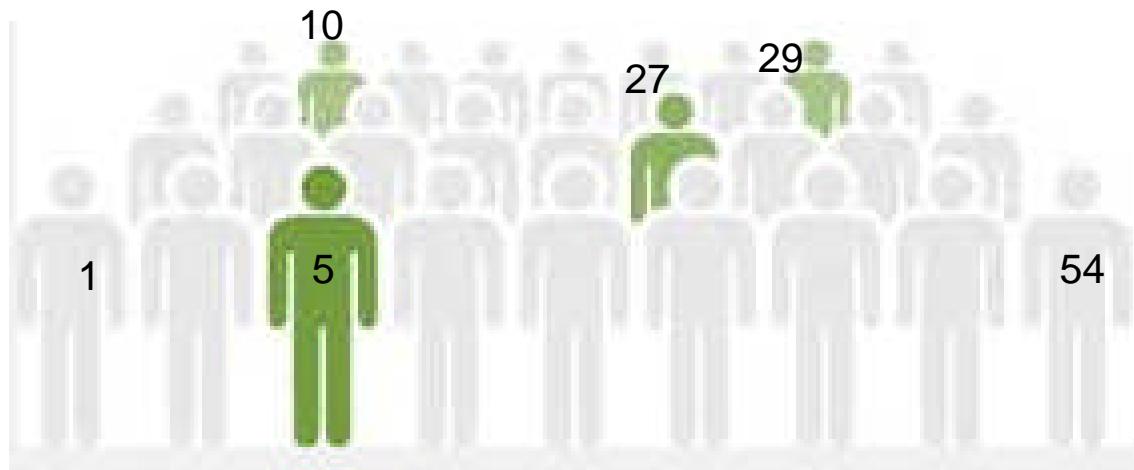
```
> sample( c("H","T"), 10, replace=TRUE)  
[1] "T" "H" "H" "H" "T" "H" "T" "H" "T" "T"
```



Select 4 different students from
a class of 54 students

```
> sample(1:54, 4) # default is no replacement
```

```
[1] 27 5 10 29
```



Create your own Functions

```
# Create a function with argument 'n'
```

```
> RollDie = function(n) sample(1:6, n, replace=T)
```

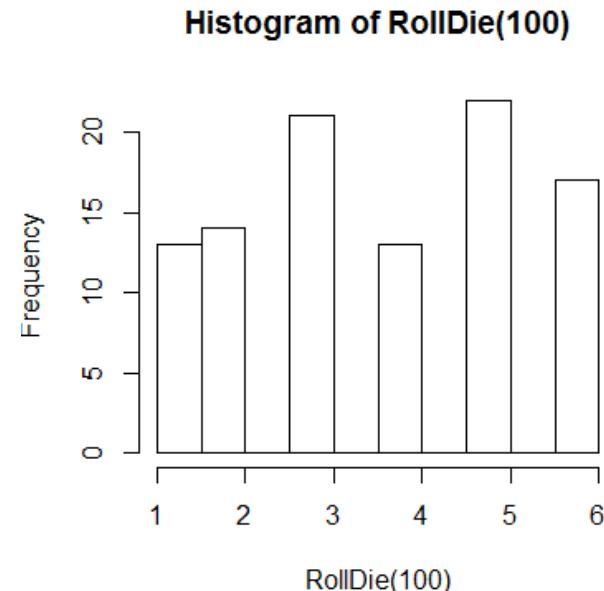
```
# Call it with n=4
```

```
> RollDie(4)
```

```
[1] 3 4 3 6
```

```
# Plot 100 rolls of the die
```

```
> hist( RollDie(100) )
```



Exercise

- Write a function to select m students from a set of n students.
- Sample usage would be:
> SelectStudents(5, 100)
[1] 54, 10, 16, 9, 8.
> SelectStudents(1, c('A','B','C'))
[1] 'B'.

Exercise: Solution to Select m/n

```
# A function to select students
```

```
# select m from n students
```

```
SelectStudents = function(m, n)
```

```
    sample(n, m, replace=F)
```

Exercise

- How to select 4 students from a set of (55,51,60) students in 3 sections?

Hint

- Step one: create a set with of $55+51+60$ students to sample from.
- Step two: sample the set.

Exercise Solution

- How to select 4 students from a set of (55,51,60) students in 3 sections?

```
> students <- c(  
  paste0('A', 1:55), # A1,A2..A55  
  paste0('B',1:51), # B1,B2..B51  
  paste0('C',1:60)) # C1,C2..C60  
> sample(students, 4)  
> "C52" "B12" "A5" "B2"
```

Exercise: Deck of card

- How to pick 5 cards from 52 cards?
- Step 1. create four suits
- Step 2. create card value: 1..13
- Step 3. combine the suit x values into deck
- Step 4. deal 5 cards from the deck

Cards: solution

```
> deck <- paste0(rep(c(2:10, "J", "Q", "K", "A"), 4),  
                  rep(c("♠", "♥", "♦", "♣"), each = 13))  
# Note these are unicode chars  
> sample(deck,5)  
[1] "2♥" "4♠" "6♣" "5♥" "3♦"
```

rep is replicate

paste0 is concatenate

```
> deck  
[1] "2♠"  "3♠"  "4♠"  "5♠"  "6♠"  "7♠"  "8♠"  "9♠"  "10♠"  
[10] "J♠"  "Q♠"  "K♠"  "A♠"  "2♥"  "3♥"  "4♥"  "5♥"  "6♥"  
[19] "7♥"  "8♥"  "9♥"  "10♥" "J♥"  "Q♥"  "K♥"  "A♥"  "2♦"  
[28] "3♦"  "4♦"  "5♦"  "6♦"  "7♦"  "8♦"  "9♦"  "10♦" "J♦"  
[37] "Q♦"  "K♦"  "A♦"  "2♣"  "3♣"  "4♣"  "5♣"  "6♣"  "7♣"  
[46] "8♣"  "9♣"  "10♣" "J♣"  "Q♣"  "K♣"  "A♣"
```

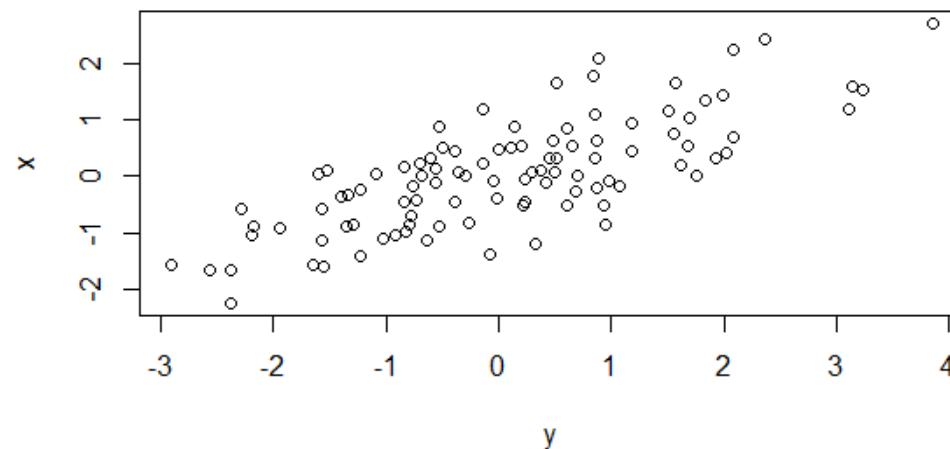
Unicode Characters for cards

```
> s <- c("♠", "♥", "♦", "♣")
> View(s)
# see https://en.wikipedia.org/wiki/List\_of\_Unicode\_characters
```

1	<U+2660>
2	<U+2665>
3	<U+2666>
4	<U+2663>

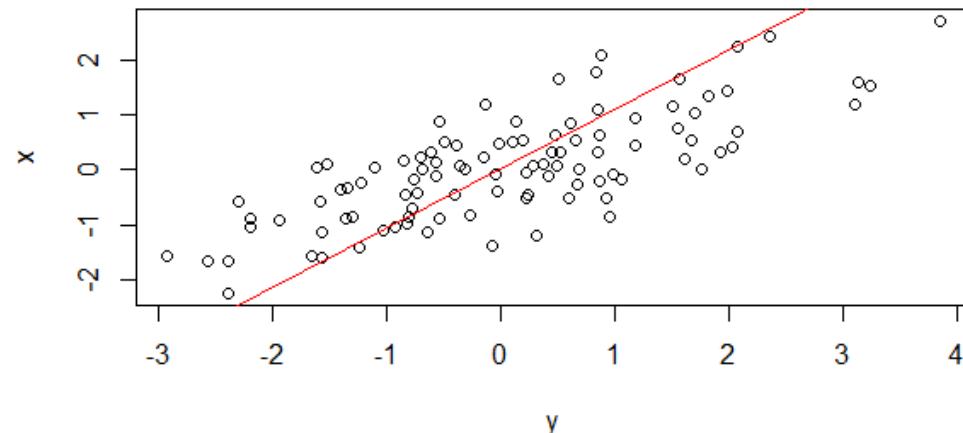
Scatter plot of two variables

```
> x <- rnorm(100)  
> y <- x + rnorm(100)  
> plot( x ~ y)
```



Add a "regression line"

```
> x <- rnorm(100)  
> y <- x + rnorm(100)  
> plot( x ~ y)  
> abline( lm(y ~ x), col = "red" )
```



Data Frames, Datasets, R/Excel

4/2/2016

Example of a R Data frame (like Excel sheet)

	A	B	C	D	E	F	G
1	Student	Test1	Test2	Test3	Test4	Test5	Average
2	Amy	80	80	75	84		80
3	Bob	85	85	88	82		85
4	Cathy	88	88	95	78		87
5	David	87	87	65	90		82
6	Edward	67	67	75	80		72
7	Frank	75	75	95	80		81
8	Ginny	25	45	95	80		61
9	Hank	99	99	78	88		91

Data frame example

```
# 3 columns: a, b c  
  
> x <- data.frame(a=1:3,  
+                   b=5:7,c=11:13)  
  
> View(x)  
  
> x  
  
      a  b  c  
1    1  5 11  
2    2  6 12  
3    3  7 13  
  
> x$a # Get column 'a' of x, same as x[['a']]  
1 2 3
```

	a	b	c
1	1	5	11
2	2	6	12
3	3	7	13

Columns of a data frame

```
> x$c <- NULL      # delete column c.  
> x$d <- 21:23     # add new column d.  
> View(x)
```

	a	b	d
1	1	5	21
2	2	6	22
3	3	7	23

cbind, combine two sheets (concat columns)

```
> y <- 31:33  
> cbind(x, y)
```

	a	b	d	y
1	1	5	21	31
2	2	6	22	32
3	3	7	23	33

Also see rbind (concat rows)

Merge two sheets (merge common columns)

```
a <- data.frame(a=1:3, b=101:103, c=201:203)
```

```
b <- data.frame(a=1:3, b=101:103, d=501:503)
```

```
> merge(a, b)
```

	a	b	c	d
1	1	101	201	501
2	2	102	202	502
3	3	103	203	503

```
> cbind(a, b)
```

	a	b	c	a	b	d
1	1	101	201	1	101	501
2	2	102	202	2	102	502
3	3	103	203	3	103	503

Merge two sheets by matching common column

```
g <- data.frame(a=1:3, b=21:23, c=201:203)
h <- data.frame(a=1:3, b=41:43, d=501:503)
m <- merge(x=g, y=h, by='a' )
# multi-columns merge using: by=c('a','b')
# vlookup in excel (limited), join in sql (4 types of joins in sql).
```

g	a	b	d
1	1	41	501
2	2	42	502
3	3	43	503

h	a	b	c
1	1	21	201
2	2	22	202
3	3	23	203

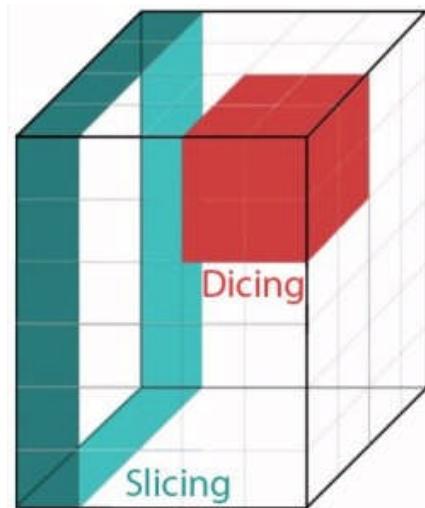
m	a	b.x	c	b.y	d
1	1	21	201	41	501
2	2	22	202	42	502
3	3	23	203	43	503

Omit NA data from Data Frame

```
> x <- c(1,2,NA,4)
> d <- data.frame(x, y=rev(x))
> d
  x  y
1 1  4
2 2  NA
3 NA  2
4 4  1

> na.omit(d)    # Remove rows with NA
  x  y
1 1  4
2 4  1
```

dplyr

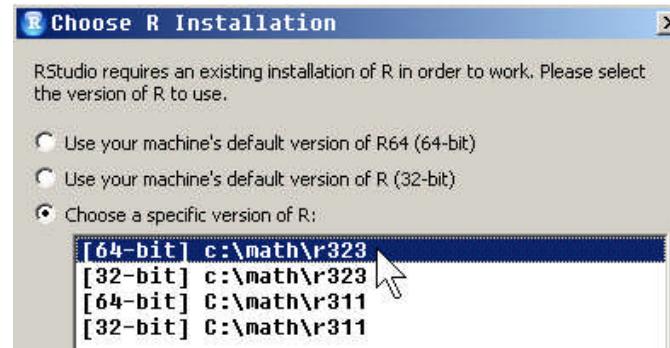


dplyr usage

Rstudio > tools >
> Global options > R
> Select R 3.2.3

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

Lots of dependencies all get installed.



library(dplyr)

dplyr verbs	Description
select()	select columns
filter()	filter rows
arrange()	re-order or arrange rows
mutate()	create new columns
summarise()	summarise values
group_by()	allows for group operations in the “split-apply-combine” concept

Select columns

```
# 3 columns: a, b c  
x <- data.frame(a=1:3,b=5:7,c=11:13)  
  
select(x, a, c) # only a and b.  
select(x, -b)  # remove b.  
select(x, b:c) # keep b to c.  
select(x, starts_with("a")) # only a.  
# Also: ends_with(s), contains(s),  
#       matches(regex), one_of(group)
```

	a	b	c
1	1	5	11
2	2	6	12
3	3	7	13

Filter rows

```
# 3 columns: a, b c  
x <- data.frame(a=1:3,b=5:7,c=11:13)  
  
filter(x, a<2) # only row 1.  
filter(x, a>2, b>6) # only row 3.  
filter(x, b %in% c(5,7) )
```

	a	b	c
1	1	5	11
2	2	6	12
3	3	7	13

Chaining/Pipe operator `%>%`

Show top of col a and c of x.

```
x %>% select(a,c) %>% head
```

Sort x by b, then descending a.

```
x %>% arrange(b, desc(a))
```

New columns

```
x %>% mutate( d = b+c ) %>%
  select(b:d) %>%
  summarize(mean(b), min(b),
  max(b))
```

	b	c	d
1	5	11	16
2	6	12	18
3	7	13	20
mean(b)	6	5	7
min(b)	5	11	16
max(b)	7	13	20

see <https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html>

group_by

```
x <- data.frame(a=1:3,  
                 b=c('m','f','m'),  
                 c=c(10,20,40))  
  
view(x)
```

	a	b	c
1	1	m	10
2	2	f	20
3	3	m	30

```
x %>% group_by(b) %>%  
  summarize(mean(c))
```

```
# A tibble: 2 x 2
  b    mean(c)
  <fctr>   <dbl>
1 f        20
2 m        25
```

see <https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html>

MATRIX

Matrix

```
> m <- matrix( c(1,2,3,4), nrow=2)
> View(m)
```

```
> m
```

```
      [,1] [,2]
[1,]   1   3    # Row 1
[2,]   2   4    # Row 2.
     Col1 Col2
```

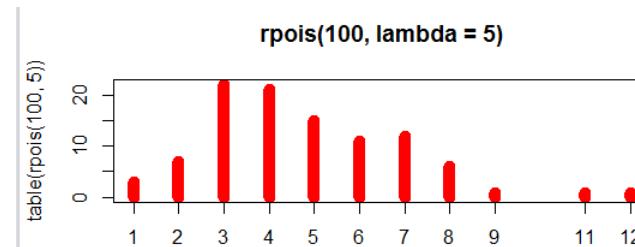
```
> det(m)  # determinant of m is -2
-2
```

	v1	v2
1	1	3
2	2	4

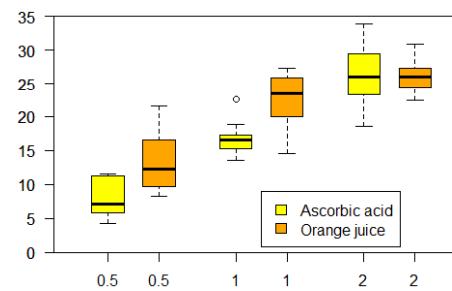
Built-in dataset

Try these examples in R

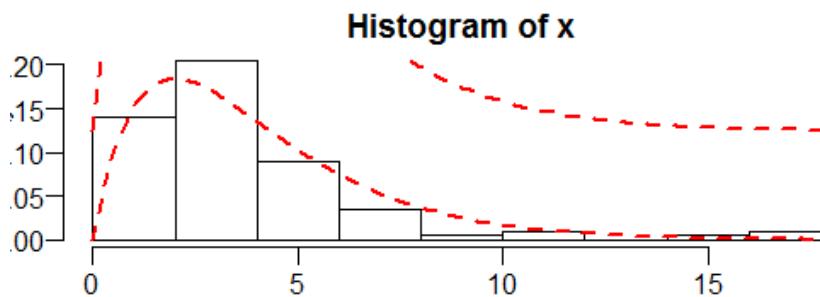
```
> example(plot)
```



```
> example(boxplot)
```



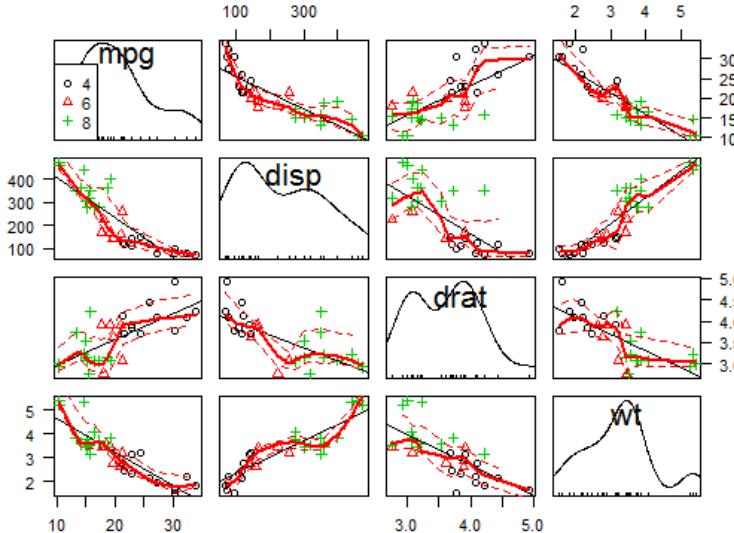
```
> example(hist)
```



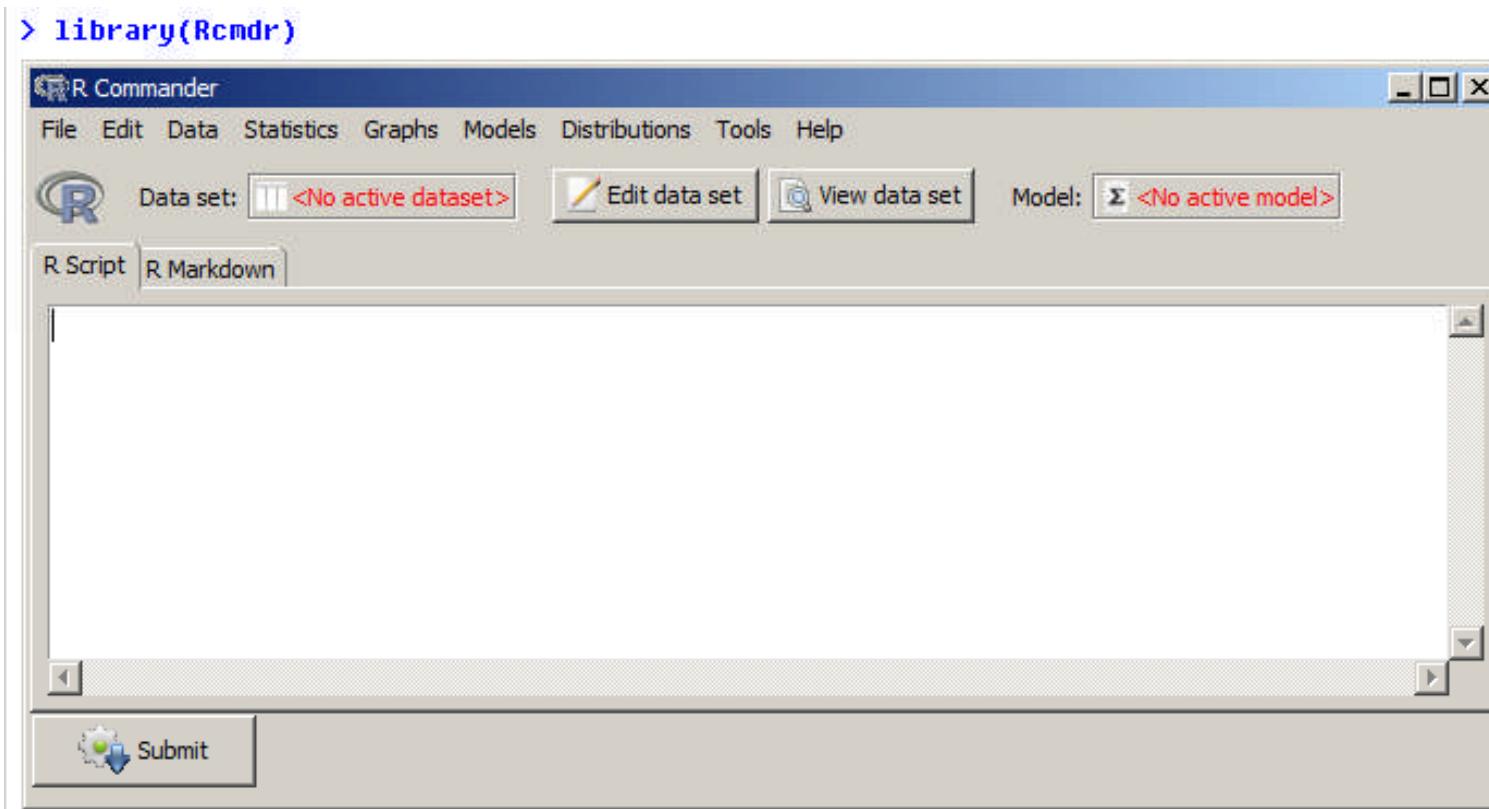
Playing with Builtin Data

Scatterplot Matrices from the car Package

```
> library(car)  
> scatterplotMatrix(~mpg +disp +drat +wt |cyl,  
+ data=mtcars)
```

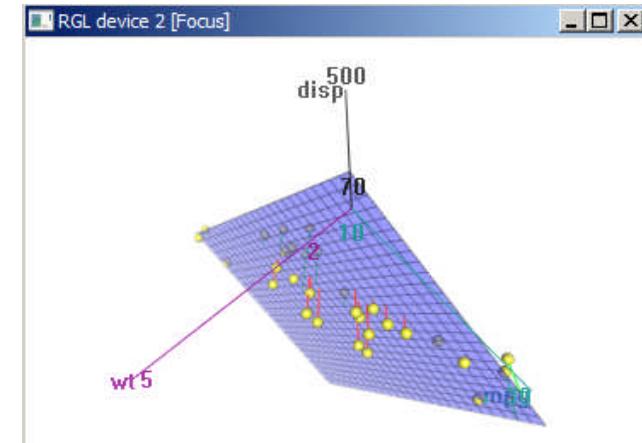


R Commander: Rcmdr



3D graphs

```
> library(Rcmdr)  
> attach(mtcars)          # see ?mtcars  
> scatter3d(wt, disp, mpg)
```

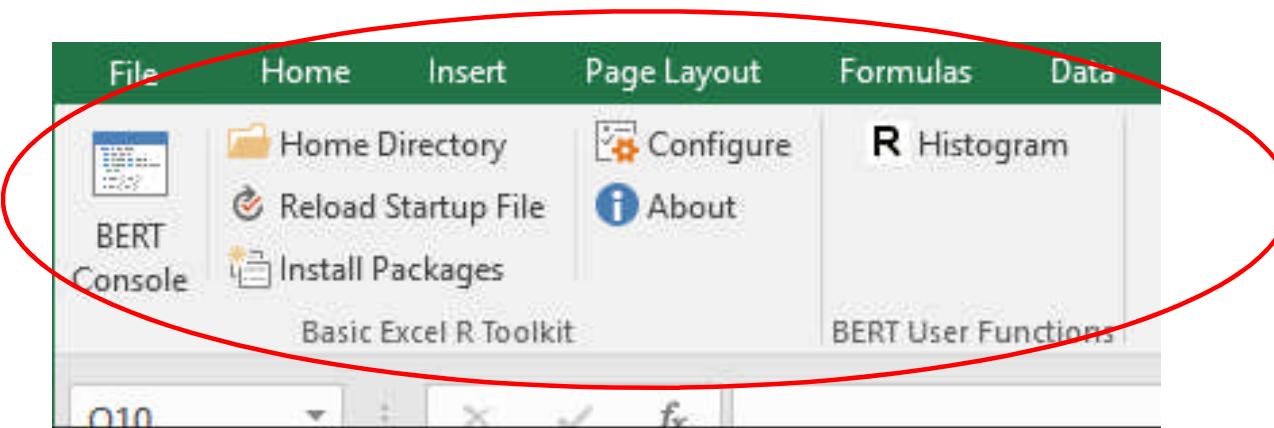


Data sharing with Excel, SPSS

```
# Reading excel csv, xls data files
sales <- read.csv( file.choose() )
prices <- read.csv("prices-2012.csv")
data <- read.xls("grades.xls")
# Load the foreign package
library(foreign)
# Import the spss data file
read.spss("newData.sav")
```

Excel with R functions

- Google "Excel + R + Bert"
- Download and Install *Bert* (free)
- R will appear inside Excel
- More later in lecture on *Bert*



R Style & Functions

Peter Kuriyama, Course:FISH 512

April 9th, 2014

Topics

1. R Style
2. Function
3. Anonymous Functions
4. Numerical Integration

Why R Style?

- Approaches that lead to clear, elegant code
- Easy to:
 - Pick up after years
 - Collaborate and share with others
 - Check code
 - Speed up code

R Style

- Code for People, Not Computers
 - We only remember a handful of things
 - Consistent, distinctive, and meaningful names
 - Consistent style and formatting
- Clear and Reproducible Code
 - Validate software like laboratory and field equipment

R Style

- Agree on common style up front
 - Sacrifice preferred aspects of style to work with others
- No style is superior to others
- Establish common "*R style*"
 - Always use
 - See Google Style Guide, and Hadley Wickham (see references, or google)

R Style

File Names

- Should end in .R and be meaningful
- Separate words with underscores
- Good:
predict_ad_revenue.R
- Bad:
foo.R

R Style

Identifiers - Objects

- Use lower case letters and periods
 - Good:
variable.name, avg.clicks
 - Bad:
VariableName , avg_Clicks

R Style

Identifiers – Functions

- Lowercase
- Words separated by “_”
- Make function names verbs
- Good:
function_name, calculate_avg_clicks
- Bad:
Func, Avg_Clicks, AvgClicks

R Style

Spacing

- Space around all binary operators
- Space around if statements
- Space after comma
- Good:
average <- mean(feet / 12 + inches, na.rm = TRUE)
- Bad:
average<- mean(feet/12+inches,na.rm=TRUE)

R Style

Curly Braces

- Opening brace should never be on its own line
- Closing brace should go on its own line
 - Unless followed by else

Good:

```
if (is.null(ylim)) {  
  ylim <- c(0, 0.06)  
}
```

Bad:

```
if (is.null(ylim)) {ylim <- c(0, 0.06)  
}
```

R Style

Curly Braces

- Opening brace should never be on its own line
- Closing brace should go on its own line
 - Unless followed by else

Good:

```
if (condition) {  
  ..  
} else {  
  ..  
}
```

Bad:

```
if (condition) {  
  ...  
}  
else {  
  ..  
}
```

R Style

Function Documentation

- Descriptive enough that caller can use and understand function without reading the code.
 - Roxygen: Package Development
- #Comments section immediately below function definition
- One-sentence description of the function
- List of function arguments denoted by Args: with description of each and data type
- Description of the return object

R Style

```
CalculateSampleCovariance <- function(x, y, verbose = TRUE) {  
  # Computes the sample covariance between two vectors.  
  # Args:  
  # x: One of two vectors whose sample covariance is to be calculated.  
  # y: The other vector. x and y must have the same length, >1.  
  # verbose: If TRUE, prints sample covariance; Default is TRUE.  
  # Returns:  
  #   The sample covariance between x and y.  
  n <- length(x)  
  covariance <- var(x, y)  
  if (verbose)  
    cat("Covariance = ", round(covariance, 4), ".\n", sep = "")  
  return(covariance)  
}
```

Style Guides

1. Hadley Wickham's Advanced R textbook

<http://adv-r.had.co.nz/>

2. Google's R Style Guide

<https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml>

Structuring Projects

Goal is Reproducibility

C://project/

C://project/**R**/

Contains function files – no code that runs

C://project/**data**/

Data treated as *read only* – usually .csv

C://project/**figs**/

Contains only generated figures

C://project/**output**/

Contains simulation output, processed datasets

C://project/**analysis.r**

Script that calls functions, reads data,
creates figures and outputs.

R Functionals

Peter Kuriyama, Course:FISH 512
April 9th, 2014

Functional Programming

- Functions as *First Class Objects*
- Treat functions as vectors
 - Assign to variables.
 - Store in lists.
 - Pass as arguments to functions (functions of functions).
 - Nested: Functions within functions

When to Program Functionally

Example: Replace -99 with NA

```
# Generate Sample Data
> set.seed(1014)
> df <- data.frame(replicate(6, sample(c(1:10, -99),
                                         10, rep = TRUE)))

> names(df) <- letters[1:6]
> head(df)
  a b c d e f
1 1 10 4 1 8 6
2 10 2 8 3 6 4
3 7 1 9 4 8 2
4 2 4 3 8 8 6
5 1 5 -99 6 2 6
6 6 3 9 5 9 9
```

What do you do?

When to Program Functionally

```
> head(df)
  a b c d e f
1 1 10 4 1 8 6
2 10 2 8 3 6 4
3 7 1 9 4 8 2
4 2 4 3 8 8 6
5 1 5 -99 6 2 6
6 6 3 9 5 9 9
```

One Solution:

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

- Easy to make mistakes - There are two
- Repetition leads to Inconsistencies
- Harder to change code
- If missing values are -999, have to change 6 lines of code

When to Program Functionally

Can write a function to fix values in single vector

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}
```

```
df$a <- fix_missing(df$a)  
df$b <- fix_missing(df$b)  
df$c <- fix_missing(df$c)  
df$d <- fix_missing(df$d)  
df$e <- fix_missing(df$e)  
df$f <- fix_missing(df$e)
```

- Doesn't eliminate errors
- Apply function to each column at a time

When to Program Functionally

```
# Can use a for loop to iterate over columns
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
for(ii in 1:ncol(df)) {
  df[, ii] <- fix_missing(df[, ii])
}
# For loops are SLOW
```

Apply any function

- **apply(X, MARGIN, FUN, ...)**
 1. X = array
 2. MARGIN: 1 = rows, 2 = columns
 3. FUN: an R function (can define yourself)
 4. ... : additional arguments to function FUN

Higher Order functions

- `apply()` is a functional: higher order function
- Multiple flavors of `apply()`
 1. `lapply()`
 2. `tapply()`
 3. `sapply()`
 4. `mapply()`

Functionals

```
# Fix missing values function
```

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x}
```

```
}      Matrix      Row or Column?      Function
```

```
df <- apply(X = df, MARGIN = 2, FUN = fix_missing)
```

```
List      Function  
df <- lapply(df, fix_missing)
```

Functionals

- Advantages of lapply()
 1. Compact
 2. If missing value changes, only need to change one place (can also make missing value an input to fix_missing function)
 3. No way for columns to be treated differently
 4. Code works regardless of number of columns
 5. Easy to generalise only a subset of columns

```
df[1:5] <- lapply(df[1:5], fix_missing)
```

Functionals Summary

- `apply()`: Evaluate function over row or column of matrix
- `lapply()`: Evaluate function over elements of a list. By column if data frame.
- By category?

Categorical Apply

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1        3.5         1.4        0.2   setosa
2          4.9        3.0         1.4        0.2   setosa
3          4.7        3.2         1.3        0.2   setosa
4          4.6        3.1         1.5        0.2   setosa
5          5.0        3.6         1.4        0.2   setosa
6          5.4        3.9         1.7        0.4   setosa
> unique(iris$Species)
[1] setosa    versicolor virginica
```

Calculate mean sepal lengths of each species

- tapply()
- plyr
- dplyr

Categorical Apply Functions

- **tapply()** : Confusing Syntax
 - http://www.ats.ucla.edu/stat/r/library/advanced_function_r.htm
- **plyr**: Easier Syntax, but slow
 - <http://www.seananderson.ca/2013/12/01/plyr.html>
- **dplyr**: Easier syntax and fast
 - Millions of rows
 - <http://blog.rstudio.org/2014/01/17/introducing-dplyr/>



Multiple Functions

What if you want the mean, sum, and standard deviation of each column?

```
> head(df)
```

	a	b	c	d	e	f
1	1	10		4	1	8
2	10	2		8	3	6
3	7	1		9	4	8
4	2	4		3	8	8
5	1	5	NA	6	2	6
6	6	3		9	5	9

Multiple Functions

One Solution:

```
mean(df$a)    # mean of column  
sum(df$a)     # sum of column  
sd(df$a)      # standard deviation of column
```

```
mean(df$b)  
sum(df$b)  
sd(df$b)
```

One function that does all three?

5 minutes

Multiple Functions

One solution is to write a `summary` function and `apply` to each column:

```
summary <- function(x) {  
  c(mean(x, na.rm = TRUE),  
    sum(x, na.rm = TRUE),  
    sd(x, na.rm = TRUE))  
}  
lapply(df, summary)
```

1. `x` and `na.rm` are repeated three times
2. Duplication => fragile code
3. Easy to introduce bugs, harder to adapt code

Apply a **list** of Functions

```
summary <- function(data.frame, funs) {  
  lapply(funs, function(f) f(data.frame, na.rm = T))  
}  
lapply(df, summary, funs = c(mean, median, sd))
```

1. Cleaner solution
2. Output requires some cleanup
3. **lapply** of lapply
4. Function of a function of a function

Store Functions in a List

```
summary <- function(data.frame, funs) {  
  lapply(funs, function(f) f(data.frame, na.rm = T))  
}  
  
sol <- lapply(df, summary, funs = c(mean, sum, sd))
```

```
fun <- c(mean, sum, sd)
```

```
> df  
  a b c d e f  
1 1 10 4 1 8 6  
2 10 2 8 3 6 4  
3 7 1 9 4 8 2  
4 2 4 3 8 8 6  
5 1 5 NA 6 2 6  
6 6 3 9 5 9 9  
7 6 5 1 8 9 3  
8 4 1 6 NA NA 8  
9 9 5 8 2 NA 1  
10 9 NA 8 3 5 4
```

```
> sol  
  mean sum   sd  
1 5.500000 55 3.374743  
2 4.000000 36 2.783882  
3 6.222222 56 2.905933  
4 4.444444 40 2.505549  
5 6.875000 55 2.416461  
6 4.900000 49 2.558211
```

Store Functions in Lists

```
summary <- function(data.frame, funs) {  
  lapply(funs, function(f) f(data.frame, na.rm = T))  
}  
  
sol <- lapply(df, summary, funs = c(mean, sum, sd))
```

```
fun <- c(mean, sum, sd)
```

```
> df  
  a b c d e f  
1 1 10 4 1 8 6  
2 10 2 8 3 6 4  
3 7 1 9 4 8 2  
4 2 4 3 8 8 6  
5 1 5 NA 6 2 6  
6 6 3 9 5 9 9  
7 6 5 1 8 9 3  
8 4 1 6 NA NA 8  
9 9 5 8 2 NA 1  
10 9 NA 8 3 5 4
```

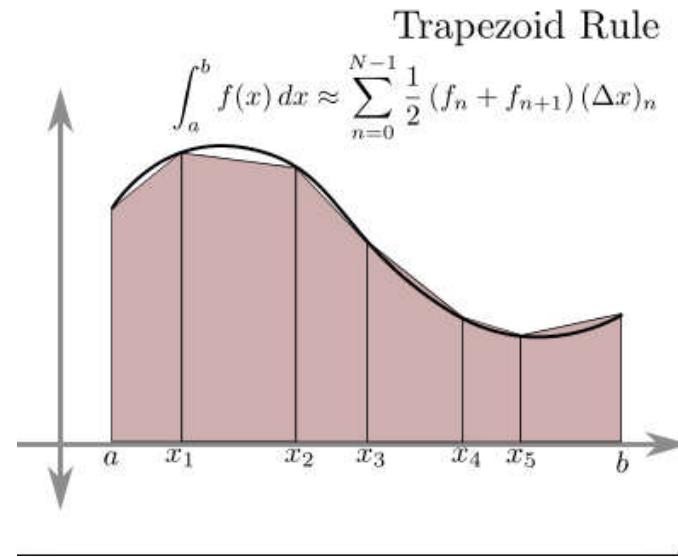
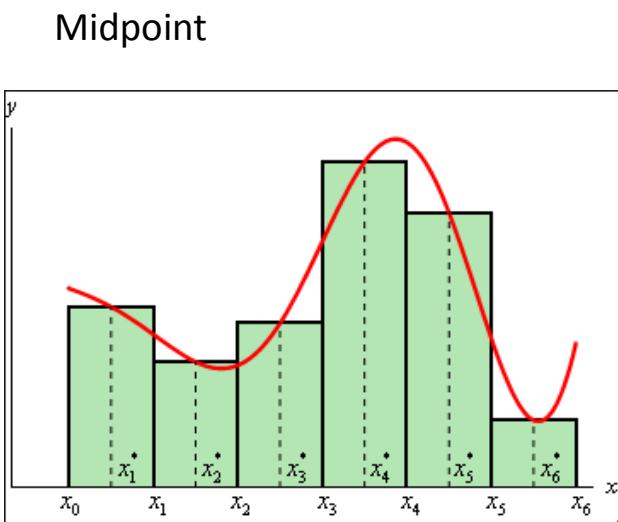
```
> sol  
  mean sum   sd  
1 5.500000 55 3.374743  
2 4.000000 36 2.783882  
3 6.222222 56 2.905933  
4 4.444444 40 2.505549  
5 6.875000 55 2.416461  
6 4.900000 49 2.558211
```

R Integration

Peter Kuriyama, Course:FISH 512

April 9th, 2014

Case Study: Numerical Integration



Case Study: Numerical Integration

- Goal is to find area under \sin curve from [0 to π]

Two Integration Methods

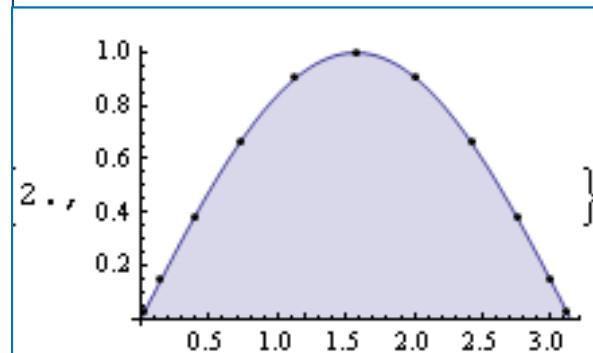
Method 1

```
midpoint <- function(f, a, b) {  
  (b - a) * f((a + b) / 2)  
}  
> midpoint(sin, 0, pi)  
[1] 3.141593
```

Method 2

```
trapezoid <- function(f, a, b) {  
  (b - a) / 2 * (f(a) + f(b))  
}  
> trapezoid(sin, 0, pi)  
[1] 1.923671e-16
```

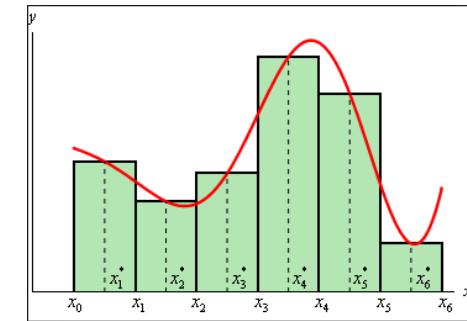
- True value is 2
- Neither method is good, need more pieces.



Case Study: Numerical Integration

```
# Break up range into smaller pieces and integrate with simple rules  
# Called composite integration
```

```
midpoint_composite <- function(f, a, b, n = 10) {  
    points <- seq(a, b, length = n + 1)  
    h <- (b - a) / n  
    area <- 0  
    for(i in seq_len(n)) {  
        area <- area + h * f((points[i] + points[i + 1]) / 2)  
    }  
    area  
}
```



Case Study: Numerical Integration

```
# Composite Integration  
> midpoint_composite(sin, 0, pi, 1)  
[1] 3.141593  
  
> midpoint_composite(sin, 0, pi, 5)  
[1] 2.033281  
  
> midpoint_composite(sin, 0, pi, 10)  
[1] 2.008248  
  
> midpoint_composite(sin, 0, pi, 100)  
[1] 2.000082
```

Integration Exercise

A general composite function that takes a *rule* as an argument: *Midpoint rule* or *trapezoid rule*

```
> composite(f = sin, a = 0, b = pi, n = 10,           rule = midpoint)  
[1] 2.008248
```

```
> composite(f = sin, a = 0, b = pi, n = 10,           rule = trapezoid)  
[1] 1.983524
```

Integration Exercise - Answer

General Composite Function, can use any method (rule).

```
composite <- function(f, a, b, n = 10, rule) {  
    points <- seq(a, b, length = n + 1)  
    area <- 0  
    for(i in seq_len(n)) {  
        area <- area + rule(f, points[i], points[i + 1])  
    }  
    area  
}
```

Case Study: Numerical Integration

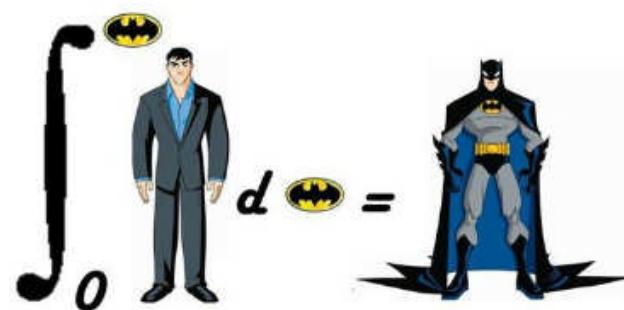
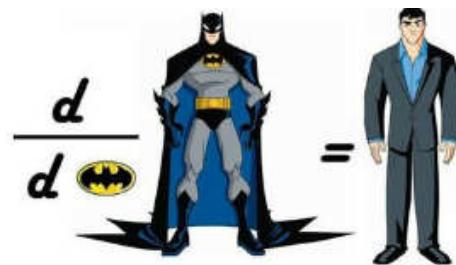
```
# Using functions we wrote already
```

```
> midpoint_composite(sin, 0, pi, n = 10)  
[1] 2.008248
```

```
> composite(sin, 0, pi, n = 10, rule = midpoint)  
[1] 2.008248
```

```
> composite(sin, 0, pi, n = 10, rule = trapezoid)  
[1] 1.983524
```

End



R inside Excel with Bert

11/3/2016

Bert

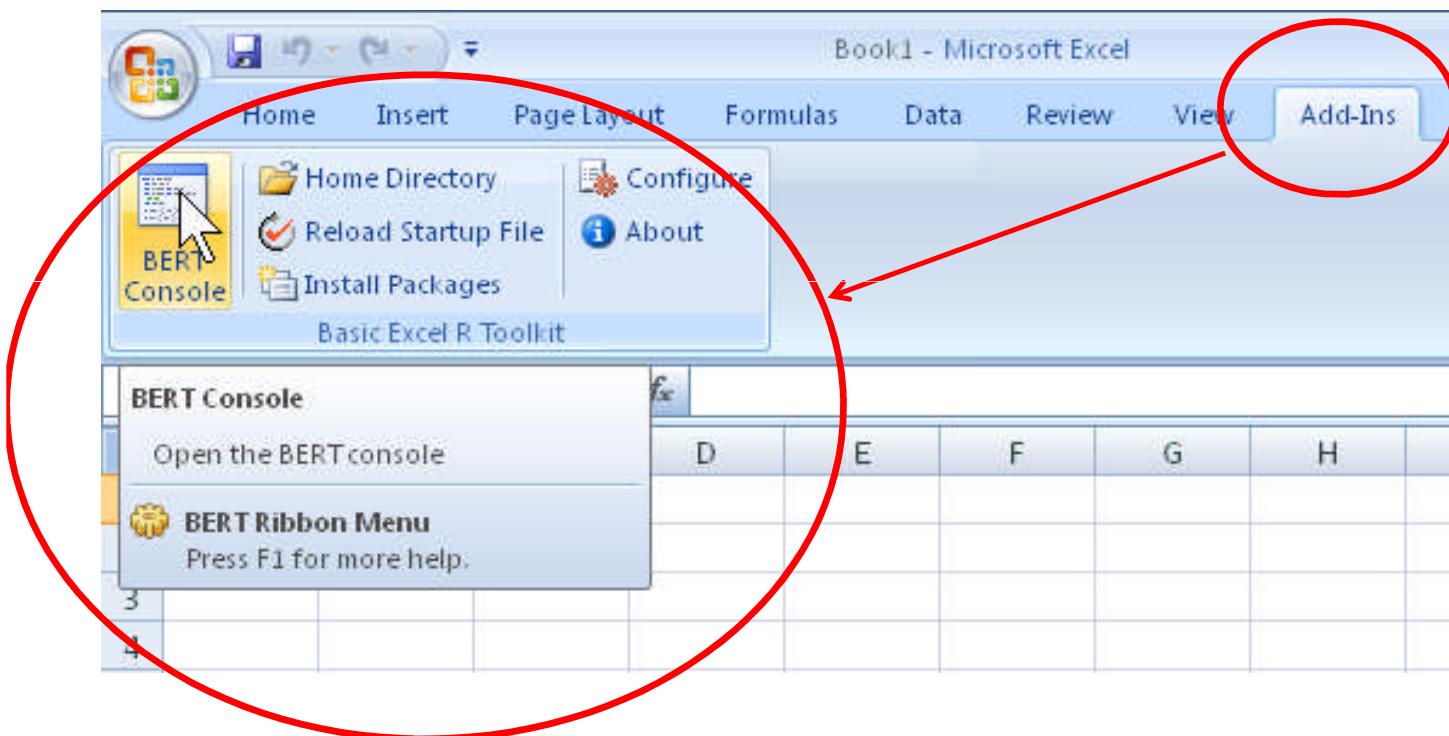
- BERT is a tool for connecting Excel with the statistics language R.
- It runs R functions from Excel spreadsheet cells.
- It is an Excel AddIn with **User Defined Functions**.
- All you have to do is write the R function.
- And use it inside Excel.

Install Bert

- Assumes you have
 - Windows Vista, 7, 8 or 10 with
 - Excel 2007, 2010, 2013 or 2016.
 - Both 32- and 64-bit
- Download BERT from <http://bert-toolkit.com/download-bert>,
- 80Mb, includes R.
- Install Bert in c:\tools\bert
- Start Excel

Excel > Add-Ins

You should see "**BERT Console**" in Add Ins



Calling a R function

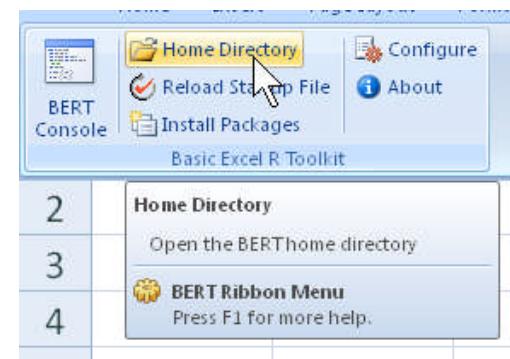
In Excel Cell A1, type =R.SUM(1,2,3)

A1	f _x	=R.SUM(1,2,3)
A	B	C
1	6	
2		

See <http://bert-toolkit.com/bert-quick-start>

Creating new R functions

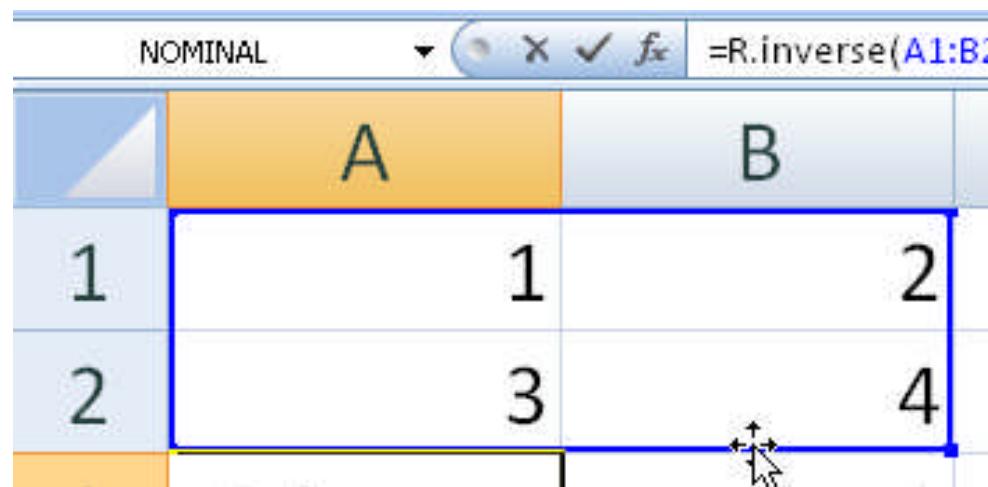
- Addins > Bert Home Directory
 - In File manager
 - Edit **functions.R** in notepad
 - Add this line to **functions.R**
- Inverse <- function(mat) { solve(mat) }



ExcelFunctions.R	11/4/2015	...	R File	14 KB
Functions.R	3/11/2016	...	R File	2 KB
README.md	6/18/2014	...	MD File	1 KB
Functions.R				R File

Using your R function in Excel

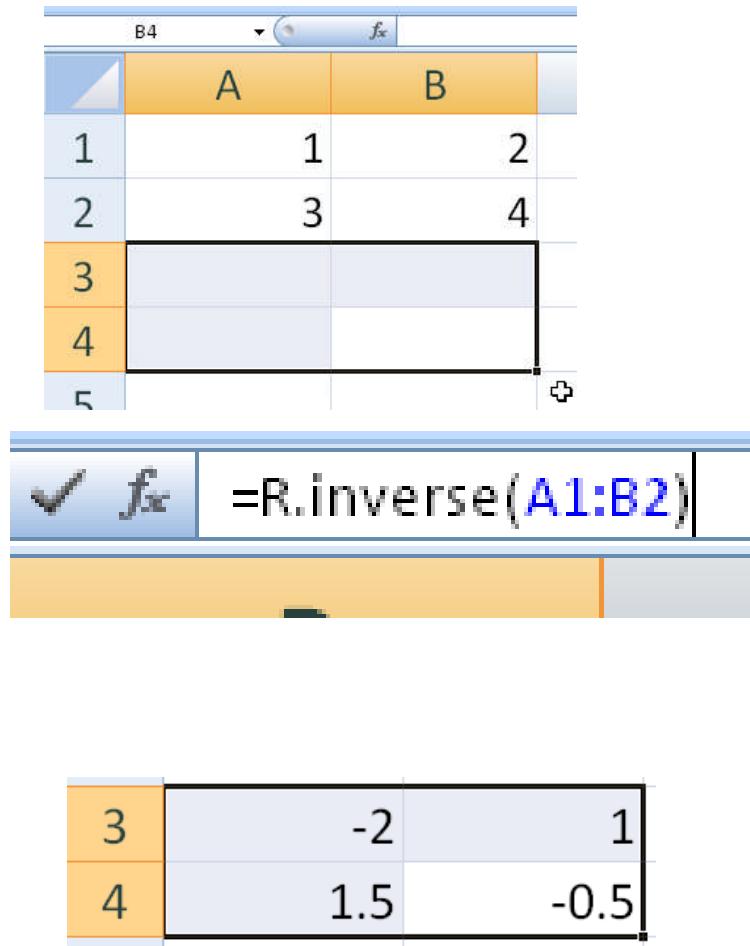
- Addins > Reload startup file
- To find inverse of a matrix by calling your **R.Inverse** function.
- Type 1,2,3,4 as shown in A1:B2



	A	B
1	1	2
2	3	4

Inverting a matrix using R

- Select A3:B4 for output
- In fx type
`=R.Inverse(A1:B2)`
- Press **Control-Shift-Enter** together (for matrix output)



The image shows a Microsoft Excel interface with two screenshots illustrating the process of inverting a matrix.

The top screenshot shows a 4x4 grid. The first two columns are labeled 'A' and 'B'. The first row contains values 1 and 2. The second row contains values 3 and 4. The third and fourth rows are highlighted in orange, representing the range A3:B4 selected for output.

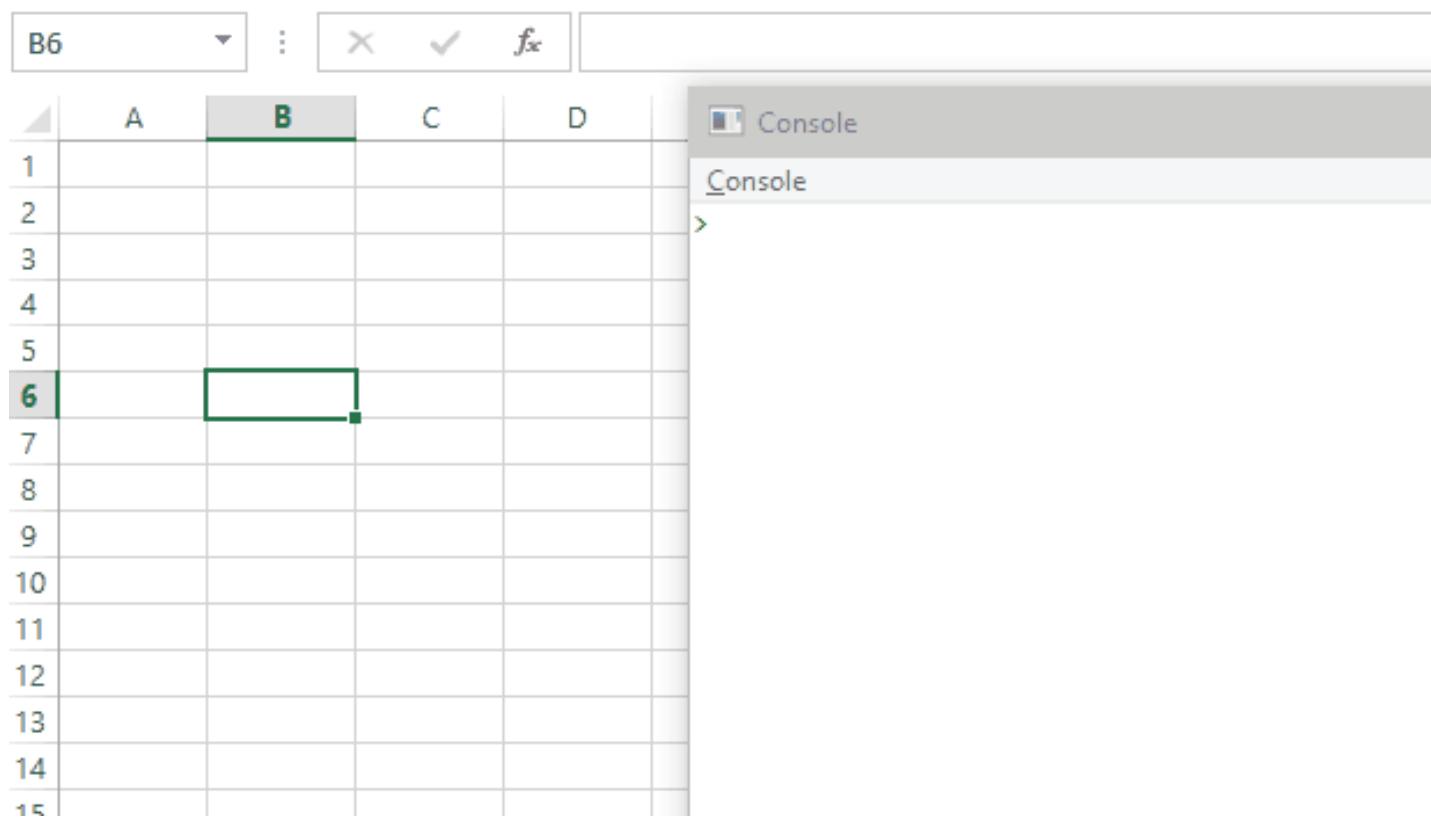
The bottom screenshot shows the formula bar with the formula `=R.Inverse(A1:B2)` entered. The formula has been evaluated, resulting in a 2x2 matrix output:

3	-2	1
4	1.5	-0.5

Result

B4			
	A	B	
1		1	2
2		3	4
3		-2	1
4		1.5	-0.5

Using R Console in Excel (animation)



Example of calling R.Cholesky (animation)

	A	B	C	D	E	F	G
1							
2		1.00	0.20	0.30			
3		0.20	1.00	0.10			
4		0.30	0.10	1.00			
5							
5							
7							
3							
9							
0							
1							
2							

References

1. <http://bert-toolkit.com/>

Introduction to Graphics in R

4/2/2016.

Goals for graphics

Two purposes for graphics:

Exploratory

need something quick that can produce graphs as fast as your mind can grasp them

Presentation

need flexibility to meet publication requirements

R graphics – Nice and Simple

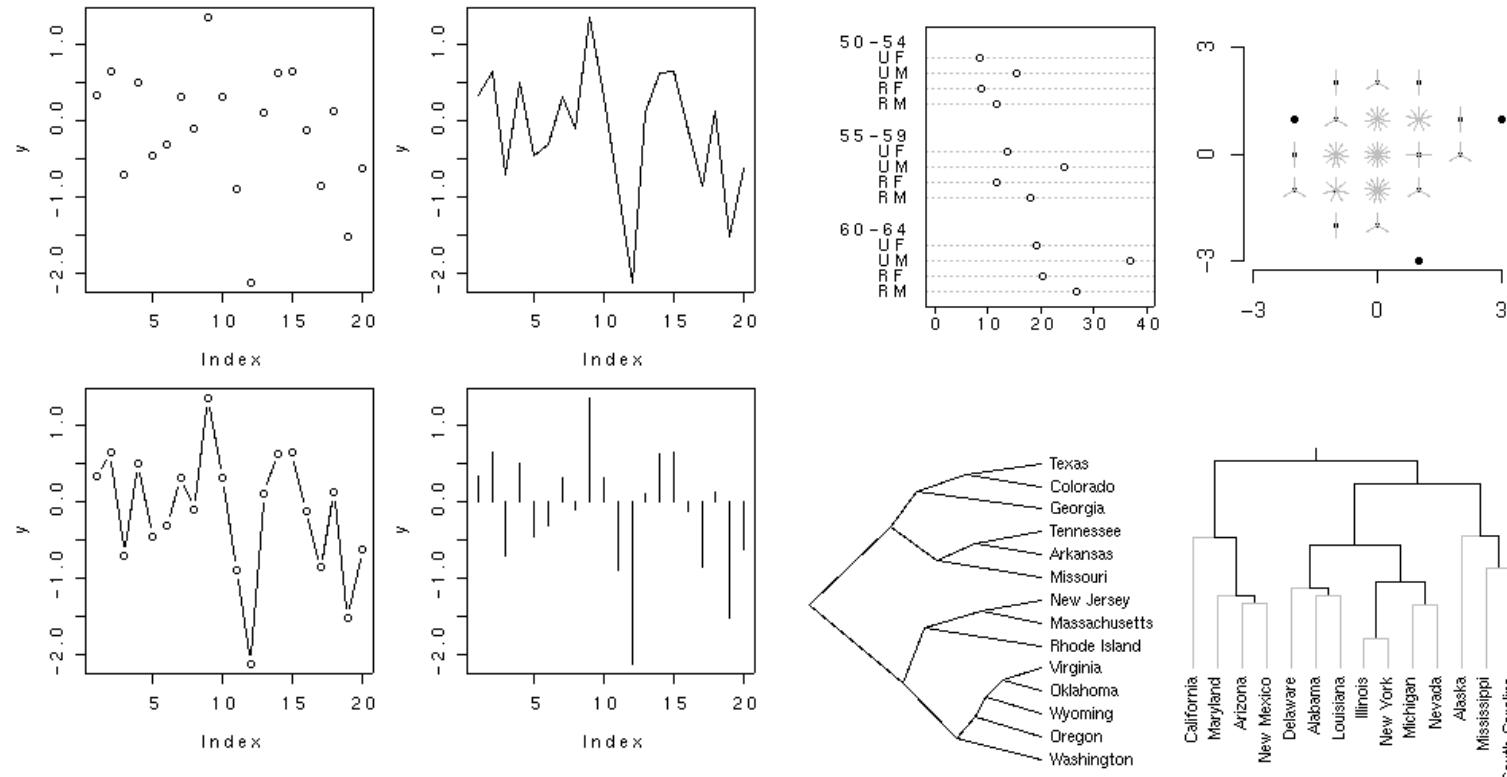
- R has powerful graphics facilities for the production of publication-quality diagrams and plots.
- Can produce traditional plots as well as grid graphics.
- Great reference: Murrell P., *R Graphics*

Graphing basics

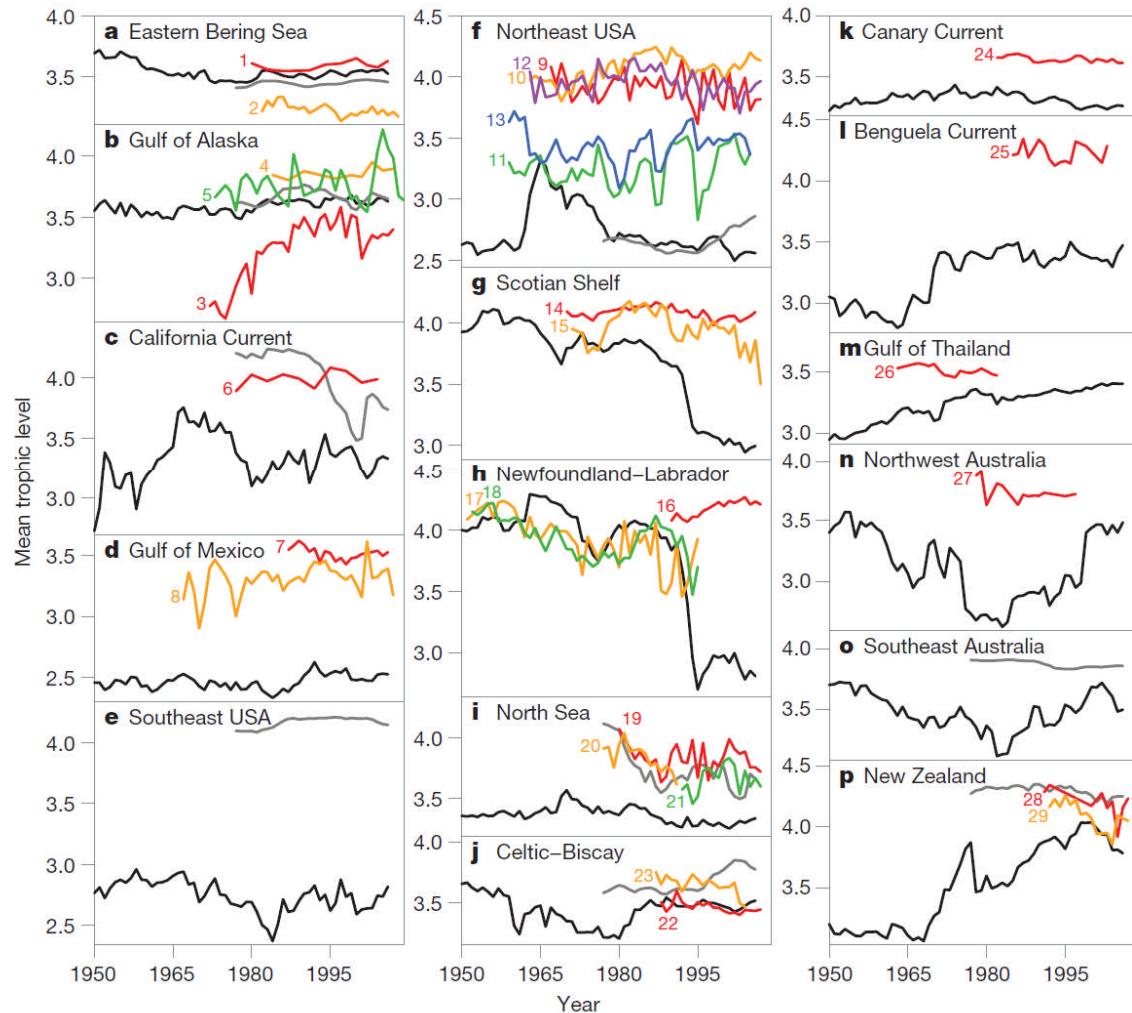
Plotting commands

1. High-level functions: Create a new plot on the graphics device
2. Low-level functions: Add more information to an already existing plot, such as extra points, lines, and labels

Example of R Graphics



There are many things Excel cannot do



Branch et al. (2010) Nature 468:431-435

Plot Types

- Univariate (one variable)
 - Categorical variables
 - Bar charts
 - Dot plots
 - Waffle plots
 - Continuous variables
 - Histogram
 - Box plot
 - Violin plots

Graphic Functions

Common high-level functions

- ***plot()***: A generic function that produces a type of plot that is dependent on the type of the first argument.
- ***hist()***: Creates a histogram of frequencies
- ***barplot()***: Creates a histogram of values
- ***boxplot()***: Creates a boxplot

Lower level graphical functions

plot type description: type= " "

p = points

l = lines

o = over plotted points and lines

b, c = points (empty if "c") joined by lines

s = stair steps

h = histogram-like vertical lines

n = does not produce any points or lines

Lower level graphical functions

- pch (plotting characters) = “ ” : character or numbers
- col (color) = “ ” : character or numbers
- lty (line type) = numbers
- lwd (line width) = numbers
- xlab (x label) = “string”, ylab (y label) = “string”
- main (heading) = “string”
- xlim (x limit) = c(lo,hi),
- ylim (y limit) = c(lo,hi)
- cex controls the symbol size in the plot, default is cex=1,

Lower-level graphing functions

- pch=0,square
- pch=1,circle
- pch=2,triangle point up
- pch=3,plus
- pch=4,cross
- pch=5,diamond
- pch=6,triangle point down
- pch=7,square cross
- pch=8,star
- pch=9,diamond plus
- pch=10,circle plus
- pch=11,triangles up and down
- pch=12,square plus
- pch=13,circle cross
- pch=14,square and triangle down
- pch=15, filled square blue
- pch=16, filled circle blue
- pch=17, filled triangle point up blue
- pch=18, filled diamond blue
- pch=19,solid circle blue
- pch=20,bullet (smaller circle)
- pch=21, filled circle red
- pch=22, filled square red
- pch=23, filled diamond red
- pch=24, filled triangle point up red
- pch=25, triangle point down red

Lower-level graphing functions

- **Adding text**

text()

text(x, y, “text”, options)

`points()` add some more points to the graph

`points(x, y, options)`

Basic graphics functions in R

Axis	Generic function to add an Axis to a Plot
abline	Add Straight Lines to a Plot
arrows	Add Arrows to a Plot
assocplot	Association Plots
axTicks	Compute Axis Tickmark Locations
axis	Add an Axis to a Plot
axis.POSIXct	Date and Date-time Plotting Functions
barplot	Bar Plots
box	Draw a Box around a Plot
boxplot	Box Plots
bxp	Draw Box Plots from Summaries
cdplot	Conditional Density Plots
contour	Display Contours
coplot	Conditioning Plots
curve	Draw Function Plots
dotchart	Cleveland Dot Plots
filled.contour	Level (Contour) Plots
fourfoldplot	Fourfold Plots
frame	Create / Start a New Plot Frame
graphics-package	The R Graphics Package
grid	Add Grid to a Plot
hist	Histograms
hist.POSIXt	Histogram of a Date or Date-Time Object

Basic graphics functions in R

identify

[Identify Points in a Scatter Plot](#)

image

[Display a Color Image](#)

layout

[Specifying Complex Plot Arrangements](#)

legend

[Add Legends to Plots](#)

lines

[Add Connected Line Segments to a Plot](#)

locator

[Graphical Input](#)

matplot

[Plot Columns of Matrices](#)

mosaicplot

[Mosaic Plots](#)

mtext

[Write Text into the Margins of a Plot](#)

pairs

[Scatterplot Matrices](#)

panel.smooth

[Simple Panel Plot](#)

par

[Set or Query Graphical Parameters](#)

persp

[Perspective Plots](#)

pie

[Pie Charts](#)

Basic graphics functions in R

Generic function ***plot*** and its methods:

plot

Generic X-Y Plotting

plot.data.frame

Plot Method for Data Frames

plot.default

The Default Scatterplot Function

plot.design

Plot Univariate Effects of a
'Design' or Model

plot.factor

Plotting Factor Variables

plot.formula

Formula Notation for Scatterplots

plot.histogram

Plot Histograms

plot.table

Plot Methods for 'table' Objects

plot.window

Set up World Coordinates for
Graphics Window

plot.xy

Basic Internal Plot Function

Basic graphics functions in R

points

polygon

rect

rug

screen
on a

segments

spineplot

stars
Diagrams

stem

stripchart

strwidth
and

sunflowerplot

symbols

text

title

xinch

Add Points to a Plot

Polygon Drawing

Draw One or More Rectangles

Add a Rug to a Plot

Creating and Controlling Multiple Screens

Single Device

Add Line Segments to a Plot

Spine Plots and Spinograms

Star (Spider/Radar) Plots and Segment

Stem-and-Leaf Plots

1-D Scatter Plots

Plotting Dimensions of Character Strings

Math Expressions

Produce a Sunflower Scatter Plot

Draw Symbols (Circles, Squares, Stars, Thermometers, Boxplots) on a Plot

Add Text to a Plot

Plot Annotation

Graphical Units

par - Change parameters using *par()*

1. A list of graphical parameters that define the default behavior of all plot functions.
2. Just like other R objects, par elements are similarly modifiable, with slightly different syntax.
 1. e.g. `par("bg"="green")`
 2. This would change the background color of all subsequent plots to green.
3. When par elements are modified directly (as above, this changes all subsequent plotting behavior.

Par examples modifiable from within plotting functions

1. bg – plot background color
2. lty – line type (e.g. dot, dash, solid)
3. lwd – line width
4. col – color
5. cex – text size inside plot
6. xlab, ylab – axes labels
7. Main – title
8. pch – plotting symbol
9. ... and many more (learn as you need them)

Plotting symbols for *pch*

Available symbols



21



22



23



24



25



16



17



18



19



20



11



12



13



14



15



6



7



8



9



10



1



2



3



4



5

Multiple plots

- The number of plots on a page, and their placement on the page, can be controlled using *par()* or *layout()*.
- The number of figure regions can be controlled using *mfrow* and *mfcol*.
e.g. `par(mfrow=c(3,2))` # Creates 6 figures arranged in 3 rows and 2 columns
- *Layout()* allows the creation of multiple figure regions of unequal sizes.
e.g. `layout(matrix(c(1,2)), heights=c(2,1))`

Colors

You can specify colors in R by index, name, hexadecimal, or RGB.

For example **col=1**,
col="white", and
col="#FFFFFF" are equivalent.

```
colors() #list of color  
names
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275
276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375
376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425
426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485
476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525
526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550
551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575
576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600
601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625
626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650
651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675

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".

Saving Graphs

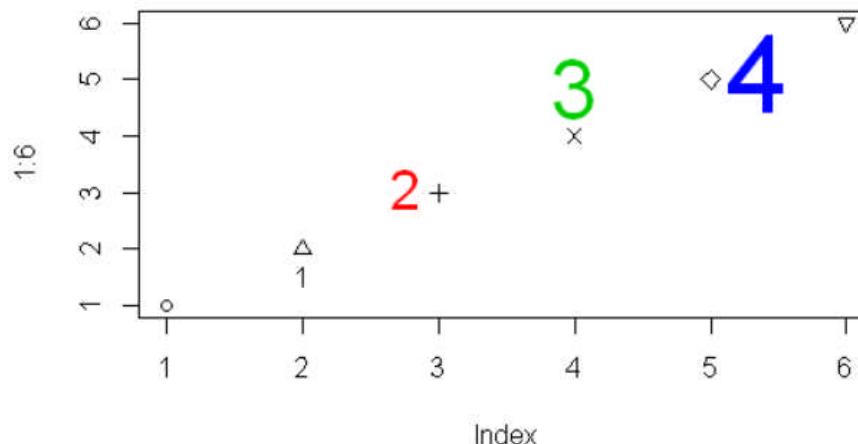
- `pdf("file.pdf")`
- `plot(...)`
- `dev.off()`

- `jpeg("file.jpg")`
- `plot(...)`
- `dev.off()`

- `win.metafile(file.wmf)`
- `plot(...)`
- `dev.off()`
 - Similar code for BMP, TIFF, PNG, POSTSCRIPT
 - PNG is usually recommended
 - The `dev.off()` function is used to close the graphical device

text() : Label size color position

```
# plot six symbols  
> plot(1:6,  
      pch=c(1:6) )  
  
# Label the 4 points  
# in 4 colors and  
# sizes  
> for(i in 1:4){  
text(  
  x=i+1,y=i+1, # point  
  i,      # label.  
  pos=i,  # position  
  col=i,  # color  
  cex=i   # size  
)  
}
```



References

1. *Introduction to R* by Venables

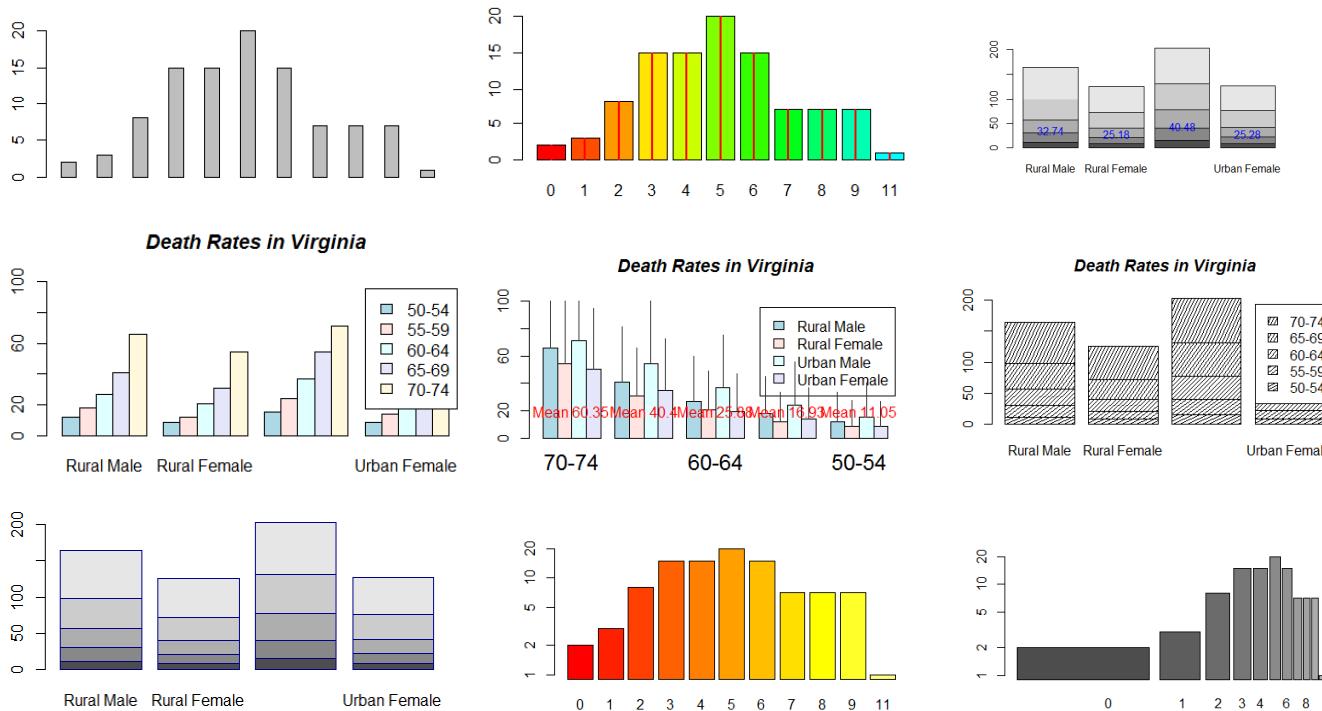
<http://cran.r-project.org/doc/manuals/R-intro.pdf>

Hands On Graphics in R

4/2/2016.

Demo: Bar plot examples

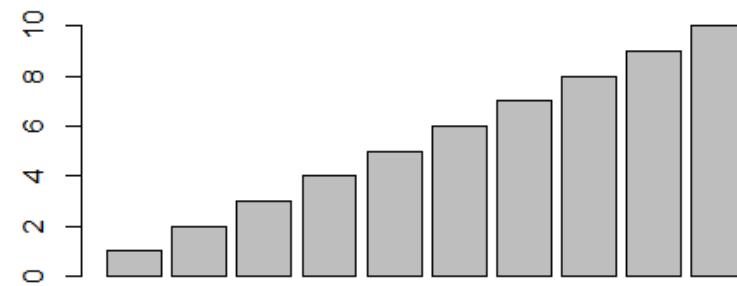
```
par(ask=TRUE)  
example(barplot)
```



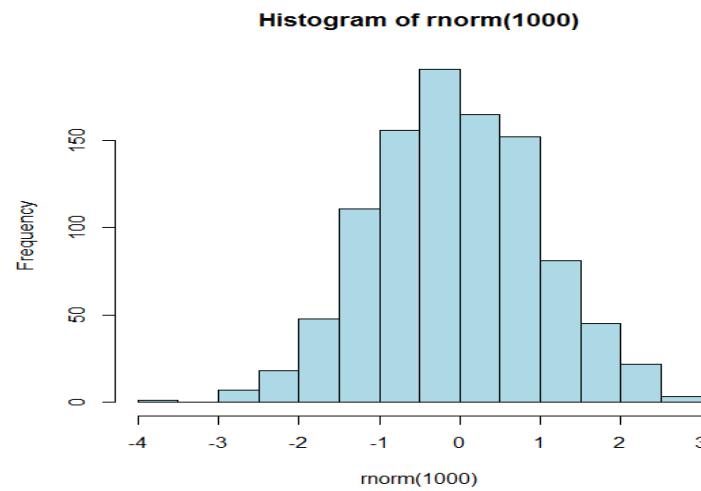
Bar plot and histogram

```
barplot(c(1:10))
```

```
barplot(1:10)
```



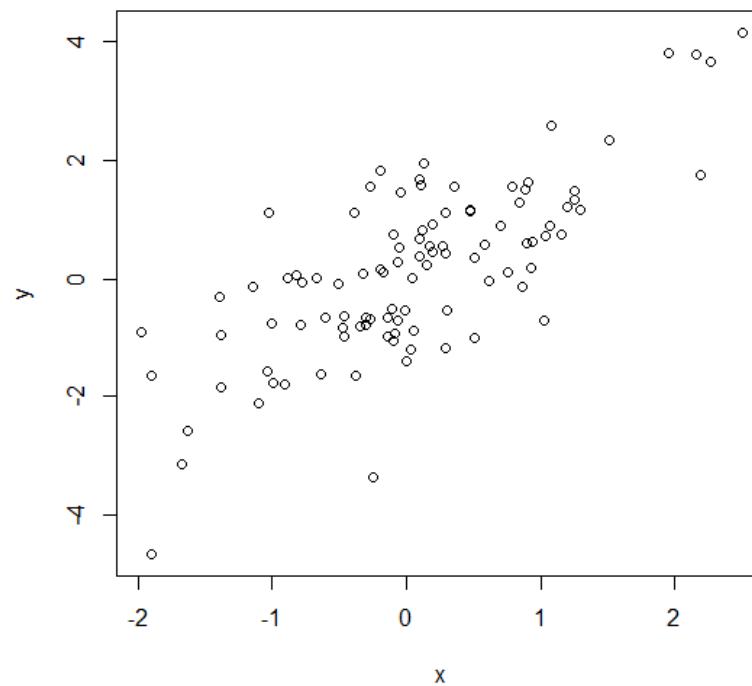
```
# normal randoms  
hist( rnorm(1000),  
      col="light blue")
```



Scatter plot

```
# Display a scatter  
plot of two  
variables
```

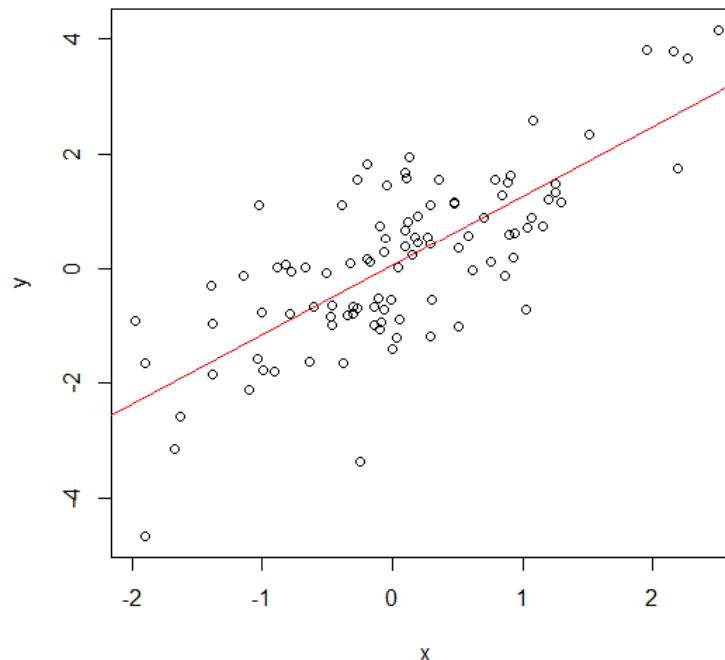
```
N <- 100  
x <- rnorm(N)  
y <- x + rnorm(N)  
plot(y ~ x)
```



Linear regression

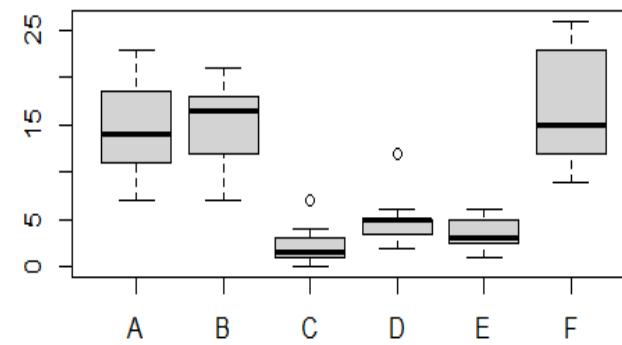
Add a "*regression line*" to that scatter plot.

```
N <- 100  
x <- rnorm(N)  
y <- x + rnorm(N)  
plot(y ~ x)  
abline(lm(y ~ x),  
       col="red" )
```

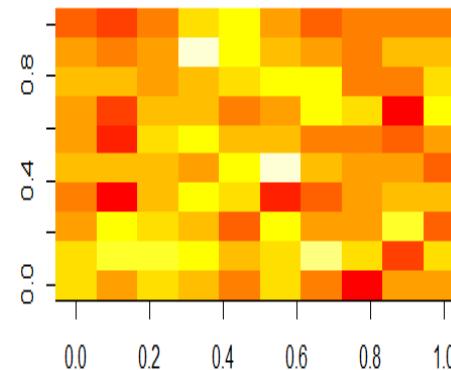


Demo: Box plots

```
boxplot( count ~  
spray, data =  
InsectSprays, col =  
"lightgray")
```

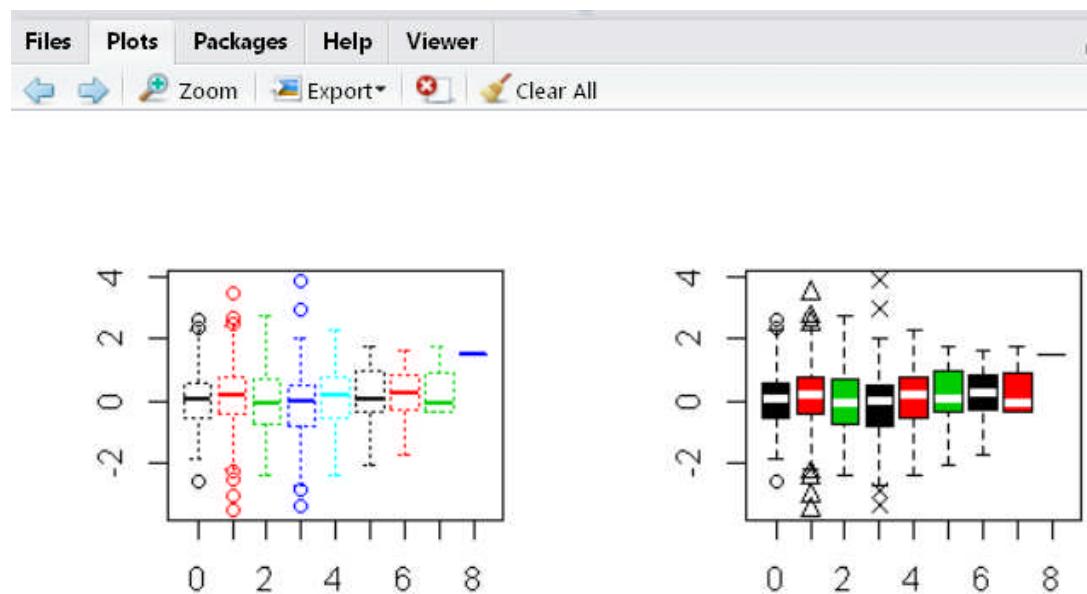


```
m <- matrix( rnorm(  
n=100, mean=0,  
sd=1), ncol=10 )  
image(m)  
text( 0:9/9, 0:9/9, 0:9)
```



Demo: boxplots

```
example(bxp) # Lots of graphs
```



Get list of built-in data sets

```
> ls("package:datasets")
```

```
Console ~/ 
> ls("package:datasets")
[1] "ability.csv"          "airmiles"
[3] "AirPassengers"        "airquality"
[5] "anscombe"              "attenu"
[7] "altitude"              "austres"
[9] "beaver1"                "beaver2"
[11] "BJSales"               "BJSales.lead"
[13] "BOD"                     "cars"
[15] "ChickWeight"            "chickwts"
[17] "CO2"                      "CO2"
```

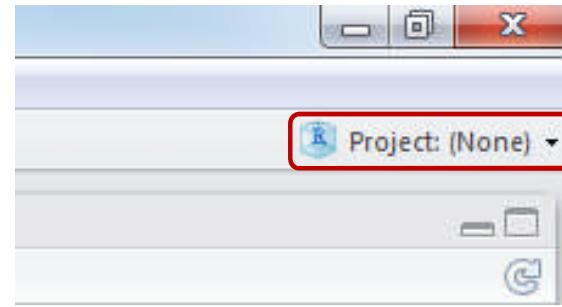
```
R data sets x | 
< < | <> > | >
> data()
> | T
Data sets in package 'datasets':
AirPassengers      Monthly Airline Passenger
Numbers             Numbers
                    1949-1960
BJSales             Sales Data with Leading
Indicator           Indicator
BJSales.lead (BJSales)   Sales Data with Leading
Indicator           Indicator
BOD                 Biochemical Oxygen Demand
CO2                 Carbon Dioxide Uptake in
```

Projects in RStudio

- Getting R to figure out where the files are (directories) is key when reading in data from files
- RStudio has projects which do this in a very slick manner
- You can store different analyses in different projects and quickly switch between them
- Each project has a separate set of .r files that are open, and a separate R workspace (saved objects in console)

Using projects

- Click on top-right option
- Choose "Create Project" -> "Existing Directory" (or "New Project" if a directory does not exist)
- Select a directory, e.g. "Lectures" for me
- The project will be saved as a file in that directory called "Lectures.Rproj"
- Opening that will open the RStudio project
- Automatically sets the R working directory to that directory



Barplots VADeaths

Barplots

- Data should be a vector or a table
- Row names and column names are used by default

> `VADeaths`

	Rural Male	Rural Female	Urban Male	Urban Female
50-54	11.7	8.7	15.4	8.4
55-59	18.1	11.7	24.3	13.6
60-64	26.9	20.3	37.0	19.3
65-69	41.0	30.9	54.6	35.1
70-74	66.0	54.3	71.1	50.0

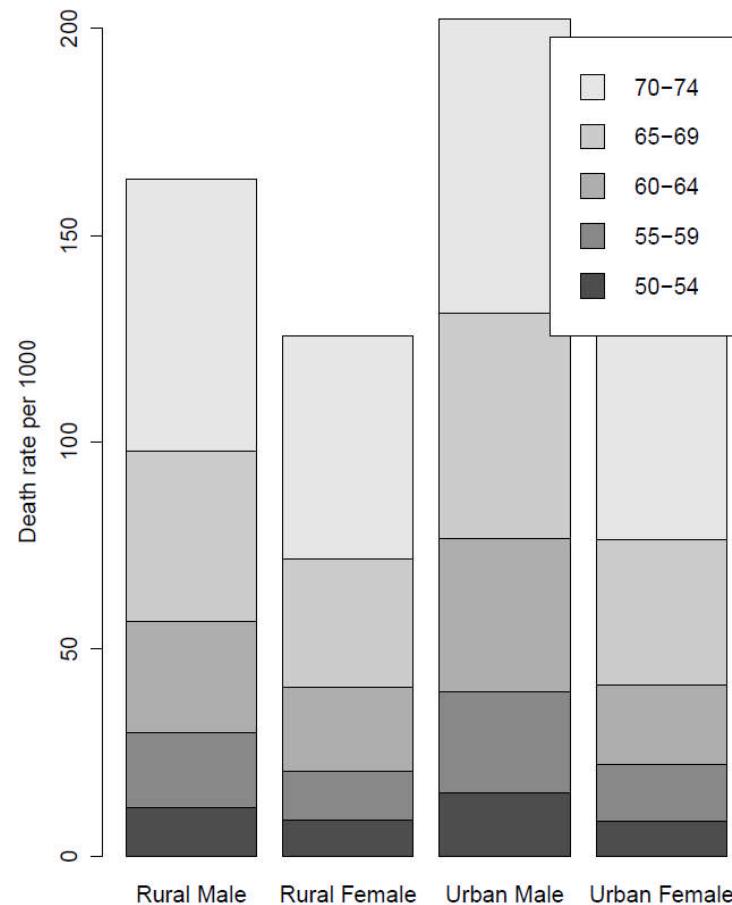
> `View (VADeaths)`

VADeaths *

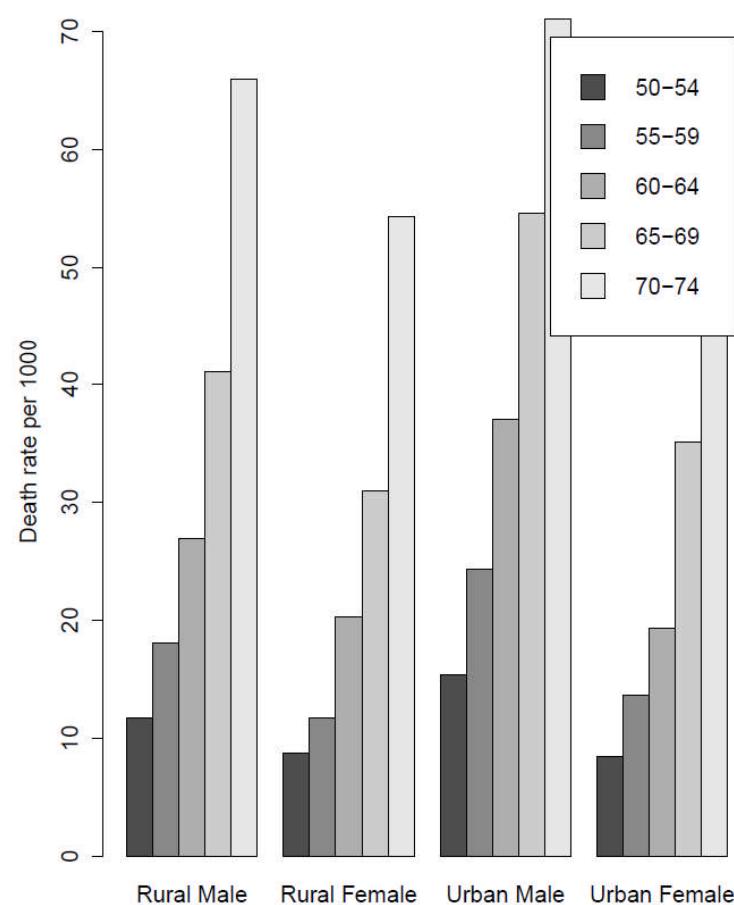
5 observations of 4 variables

	row.names	Rural Male	Rural Female	Urban Male	Urban Female
1	50-54	11.7	8.7	15.4	8.4
2	55-59	18.1	11.7	24.3	13.6
3	60-64	26.9	20.3	37.0	19.3
4	65-69	41.0	30.9	54.6	35.1
5	70-74	66.0	54.3	71.1	50.0

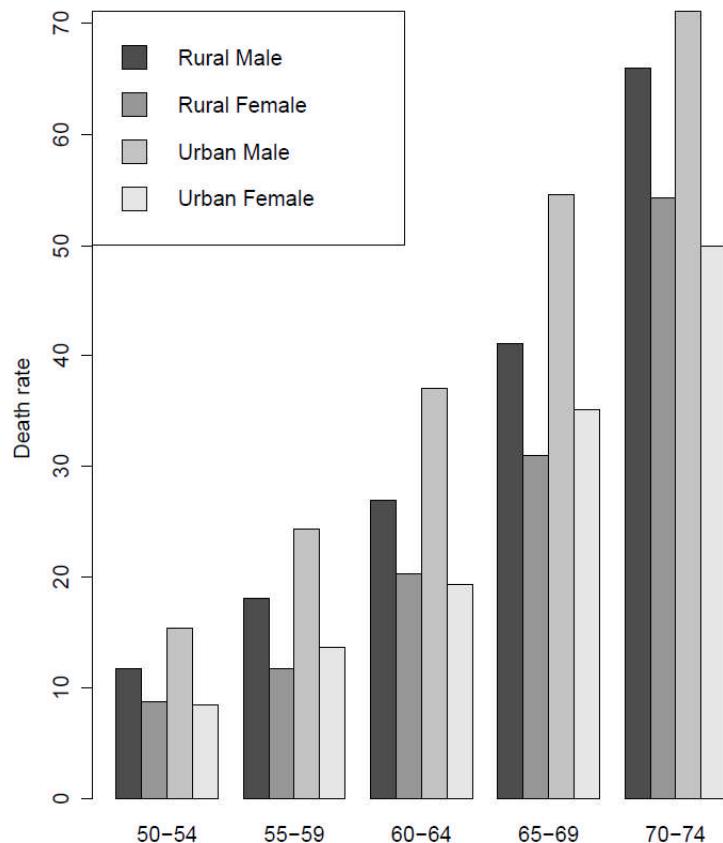
```
> barplot(VADeaths,  
Legend=TRUE, ylab="Death  
rate")
```



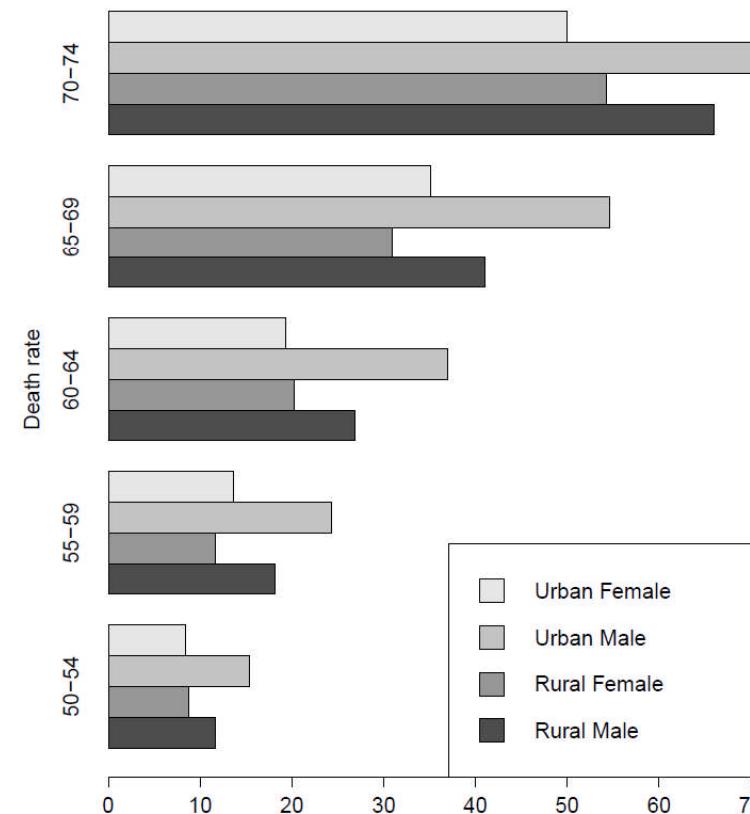
```
> barplot(VADeaths,  
Legend=TRUE, ylab="Death  
rate", beside=TRUE)
```



```
barplot(t(VADeaths), legend=TRUE,  
ylab="Death rate", beside=TRUE,  
args.legend=list(x="topleft"))
```



```
barplot(t(VADeaths), legend=TRUE,  
ylab="Death rate", beside=TRUE,  
args.legend=list(x="bottomright"),  
horiz=T)
```



Data Analysis Challenge

See <https://www.kaggle.com/datasets>

Plotting InsectSpray Data in R

5/2/2016

From <http://www.r-bloggers.com/exploratory-data-analysis-useful-r-functions-for-exploring-a-data-frame/>

The Data

We will use builtin data:

```
> View(InsectSprays)  
> summary(InsectSprays)
```

```
> summary(InsectSprays)  
    count      spray  
Min.   : 0.00  A:12  
1st Qu.: 3.00  B:12  
Median  : 7.00  C:12  
Mean   : 9.50  D:12  
3rd Qu.:14.25  E:12  
Max.   :26.00  F:12  
>
```

The screenshot shows the RStudio interface with the 'InsectSprays' data frame open in the Environment tab. The data frame contains 72 observations of 2 variables: 'count' and 'spray'. The 'count' variable ranges from 0.00 to 26.00, and the 'spray' variable has six levels: A, B, C, D, E, and F. The 'Console' tab at the bottom shows the command 'View(InsectSprays)'.

	count	spray
1	10	A
2	7	A
3	20	A
4	14	A
5	14	A
6	12	A
7	10	A
8	23	A
9	17	A

Examine the Data

Use [dim\(\)](#) to obtain the dimensions of the data frame (number of rows and number of columns).

The output is a vector of rows and columns.

```
> dim(InsectSprays)
```

```
[1] 72 2
```

Use [nrow\(\) and ncol\(\)](#) to get the number of rows and number of columns, respectively.

```
> nrow(InsectSprays)      # same as dim(InsectSprays)[1]
```

```
[1] 72
```

```
> ncol(InsectSprays)      # same as dim(InsectSprays)[2]
```

```
[1] 2
```

Head and tail of data

Use [head\(\)](#) to obtain the first n observations
and [tail\(\)](#) to obtain the last n observations.

```
> head(InsectSprays, n = 2)
```

S.No.	count	spray
-------	-------	-------

1	10	A
2	7	A

Levels

- To obtain all of the categories or levels of a categorical variable, use the [levels\(\)](#)function.

```
> levels(InsectSprays$spray)
```

```
[1] "A" "B" "C" "D" "E" "F"
```

Data header names() and str()

The [names\(\)](#) function will return the column headers.

- > names(InsectSprays)
[1] "count" "spray"

The [str\(\)](#) function returns many useful pieces of information, including the above useful outputs and the types of data for each column. In this example, “num” denotes that the variable “count” is numeric (continuous), and “Factor” denotes that the variable “spray” is categorical with 6 categories or levels.

```
> str(InsectSprays)
'data.frame': 72 obs. of 2 variables:
 $ count: num 10 7 20 14 14 12 10 23 17 20 ...
 $ spray: Factor w/ 6 levels "A","B","C","D",..: 1 1 1 1 1 1 1 1 ...
```

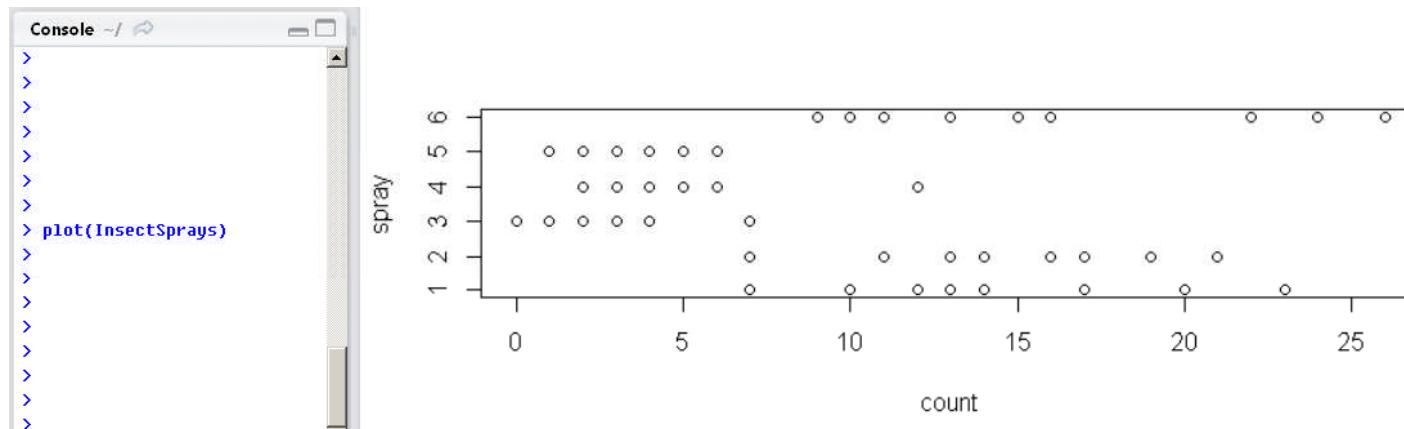
Base graphics: plot()

- Plot is the generic function for plotting R objects
 - points, lines, etc.

Three ways to plot

```
1> plot(x = InsectSprays$count,  
       y = InsectSprays$spray)  
  
2> plot(count ~ spray, data = InsectSprays)  
  
3> attach(InsectSprays)  
plot(x = count, y = spray)  
detach(InsectSprays)
```

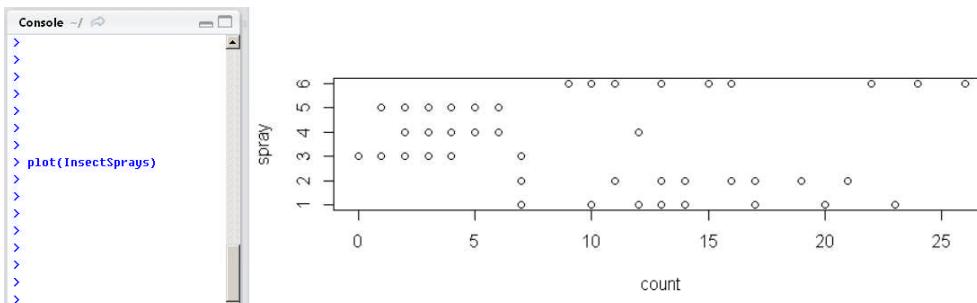
plot(InsectSprays)



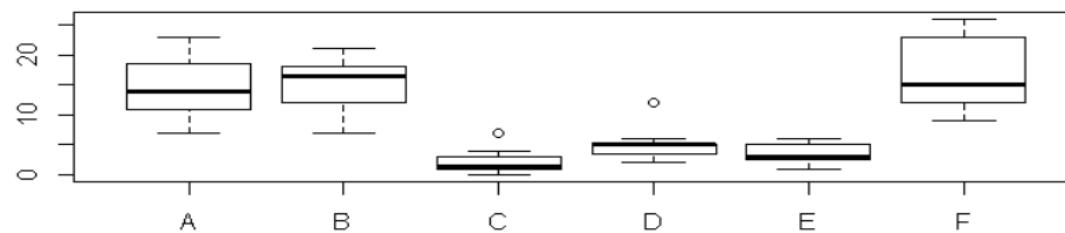
This plot is a little boring!

plot(InsectSprays)

```
# plot(InsectSprays)
plot(x = InsectSprays$count, y = InsectSprays$spray)
```



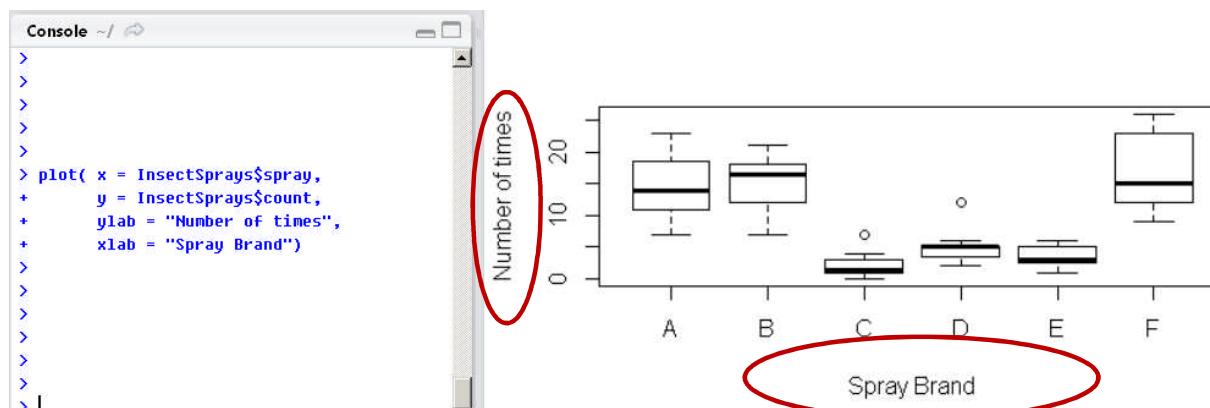
```
# Exchange X and Y
plot(y = InsectSprays$count, x = InsectSprays$spray)
plot(count ~ spray, data = InsectSprays)
```



Labels on the axes

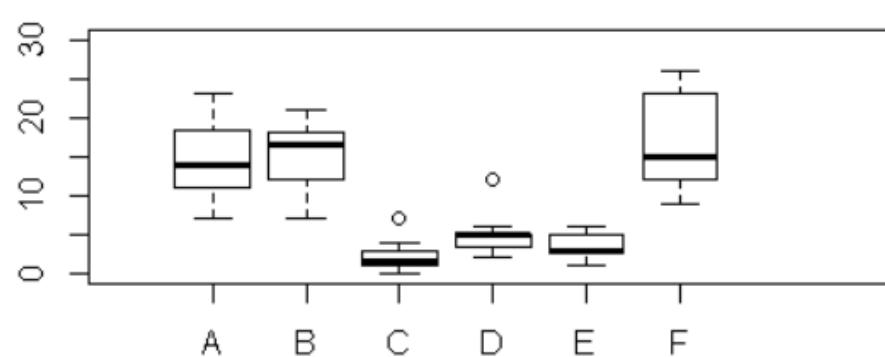
- By default R uses the name of the variables as axis labels
- Use the xlab and ylab options to change the labels

```
plot( x = InsectSprays$spray,  
      y = InsectSprays$count,  
      ylab = "Number of times",  
      xlab = "Spray Brand")
```



Limits of the axes

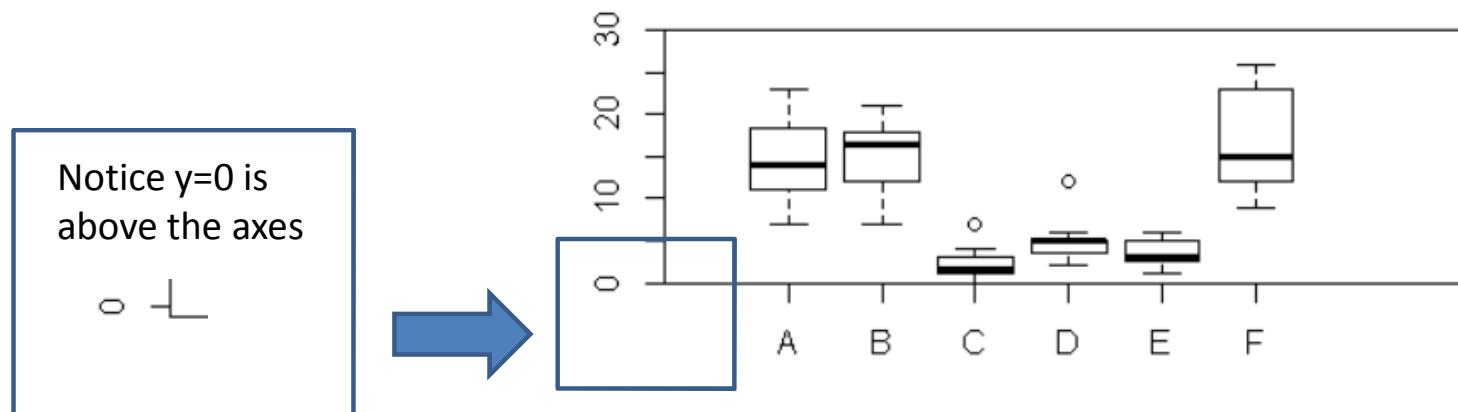
- By default, R chooses x and y limits that are just larger (4%) than the range of your data
 - May or may not include zero
 - To change the default x and y values use `xlim` and `ylim`
- > `plot(y = InsectSprays$count, ylim=c(0,30),
x = InsectSprays$spray, xlim=c(0,8))`



Remove space around zero

- But R adds space between the axis and 0
- This makes zero values look positive
- To remove this, use `xaxs="i"` and `yaxs="i"` together with `xlim` and `ylim`

```
> plot(y = InsectSprays$count, ylim=c(0,30), yaxs='i',
      x = InsectSprays$spray, xlim=c(0,8) , xaxs = 'i')
```

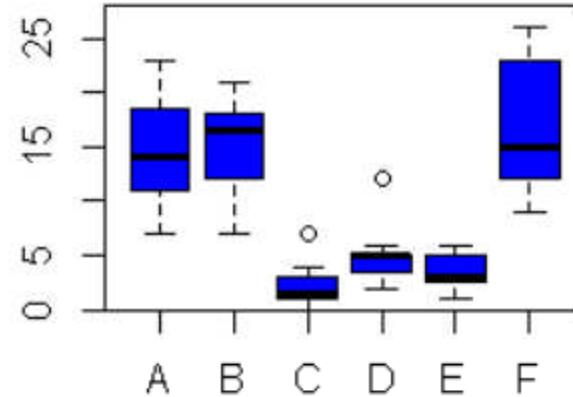


Using colors in R

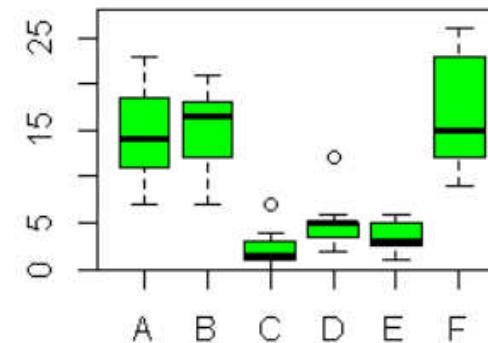
- Colors of points, lines, text etc. can all be specified
- Colors can be applied to various plot parts
 - `col` (default color)
 - `col.axis` (tick mark labels)
 - `col.lab` (x label and y label)
 - `col.main` (title of the plot)
- Colors can be specified as numbers or text strings
 - `col=1` or `col="red"`

Using colors in R

```
> plot(col="blue",
y = InsectSprays$count,
x = InsectSprays$spray,
xlim=c(0.5,6.5), ylim=c(0,28),
xaxs="i", yaxs="i")
```



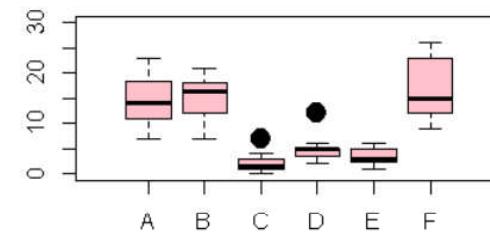
```
> plot(col="green",
y = InsectSprays$count,
x = InsectSprays$spray,
xlim=c(0.5,6.5), ylim=c(0,28),
xaxs="i", yaxs="i")
```



In-class exercise 1

- Copy and paste the following command into your R script

```
> plot(x = InsectSprays$spray, y = InsectSprays$count,  
       xlim = c(0,7), ylim = c(0,30),  
       cex = 2, pch = 19, col = "pink")
```



Experiment with different colors: e.g. col="red"

- Try different color numbers: col=1, col=2
- Try a vector of color numbers: col=c(2, 4)
- Try changing the values for cex and pch

Color naming

- There are 657 named colors
 - > `colors() # show the color names.`

```
Console ~/ ↗
>
> colors() # show the color names.
[1] "white"                      "aliceblue"
[3] "antiquewhite"                "antiquewhite1"
[5] "antiquewhite2"                "antiquewhite3"
[7] "antiquewhite4"                "aquamarine"
[9] "aquamarine1"                 "aquamarine2"
[11] "aquamarine3"                 "aquamarine4"
[13] "azure"                       "azure1"
[15] "azure2"                      "azure3"
[17] "azure4"                      "beige"
[19] "bisque"                      "bisque1"
[21] "bisque2"                     "bisque3"
```

R color chart: keep handy

<http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf>

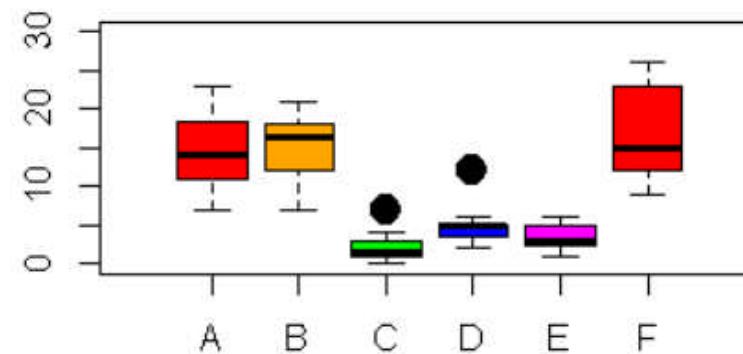
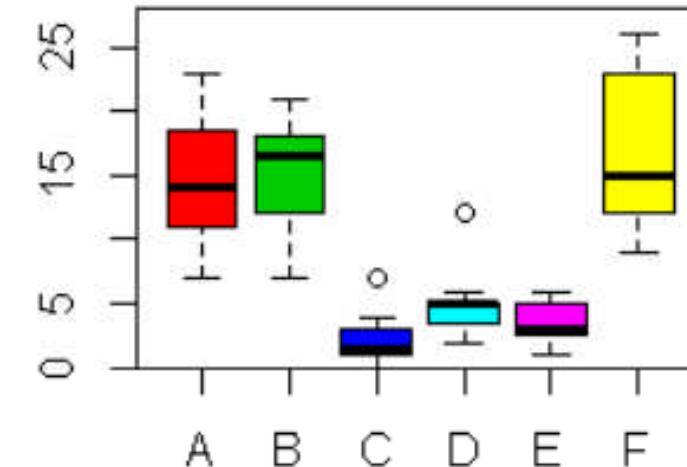
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250

1	white	#FFFFFF	255	255	255
2	aliceblue	#F0F8FF	240	248	255
3	antiquewhite	#FAEBD7	250	235	215
4	antiquewhite1	#FFEFD8	255	239	219
5	antiquewhite2	#EEDFCC	238	223	204
6	antiquewhite3	#CDC0B0	205	192	176
7	antiquewhite4	#BDBEAC	139	131	120

51	chartreuse4	#458B00	69	139	0
52	chocolate	#D2691E	210	105	30
53	chocolate1	#FF7F24	255	127	36
54	chocolate2	#EE7621	238	118	33
55	chocolate3	#CD661D	205	102	29
56	chocolate4	#8B4513	139	69	19
57	coral	#FF7F50	255	127	80

Using range of colors in R

```
> plot( col=c(2:8),  
y = InsectSprays$count,  
x = InsectSprays$spray,  
xlim=c(0.5,6.5), ylim=c(0,28),  
xaxs="i", yaxs="i" )  
  
#=====  
point.colors <- c("red",  
"orange", "green", "blue",  
"magenta")  
  
plot(  
x=InsectSprays$spray,  
y=InsectSprays$count,  
xlim=c(0,7),ylim=c(0,30),  
cex=2, pch=19,  
col=point.colors)
```



The point shape and size

- Default is an open circle (`pch=1`) of size 1 (`cex=1`)
 - `pch` is short for plotting character, 1..25, "+","*",etc.
 - `cex` is short for character expansion (size).

pch controls the type of symbol, either an integer between 1 and 25, or any single char within ""

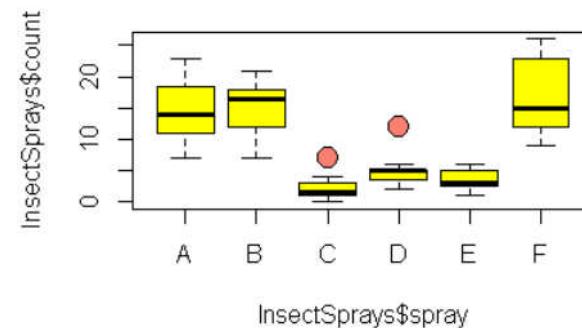
1 ○	2 △	3 +	4 ×	5 ◇	6 ▽	7 ☒	8 *
9 ◆	10 ⊕	11 ☂	12 ■	13 ☗	14 ☓	15 ■	
16 ●	17 ▲	18 ♦	19 ●	20 •	21 ○	22 □	23 ◇
24 ▲	25 ▽	* *	. .	X X	a a	? ?	

Source: R Reference Card 2.0

Useful options for points

- Useful code and options under ?points
- Use pch=21 for filled circles, for example:
 - Specify circle color with col
 - Specify fill color with bg

```
> plot(cex=2, pch=21, col="yellow", bg="salmon",
      InsectSprays$count ~ InsectSprays$spray)
```



Full list of plot parameters: par()

- If you ask for help on the `plot()` command using `?plot`, only a handful of commands are listed
- There are numerous extra commands listed under `?par` that can be added to **all** plotting commands, not just `plot()`
- Using `par()` by itself applies commands to multiple graphs

```
> par()
```

```
$xlog
```

```
[1] FALSE
```

```
...
```

Use `par()` to change global settings

avoid this whenever possible

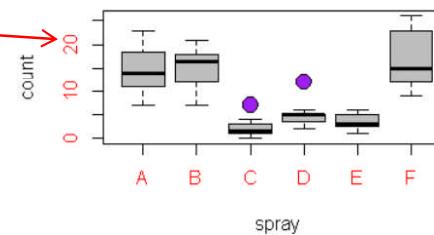
```
# Save default par values
```

```
old.par <- par( )
```

```
# Change to a new value
```

```
par(col.axis="red")
```

```
plot(count ~ spray, data = InsectSprays,  
     cex=2, pch=21, col="grey", bg="purple")
```

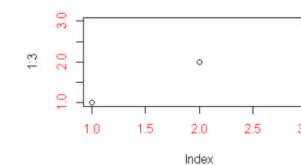
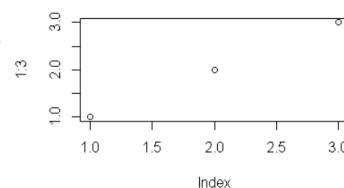


```
plot(1:3)
```

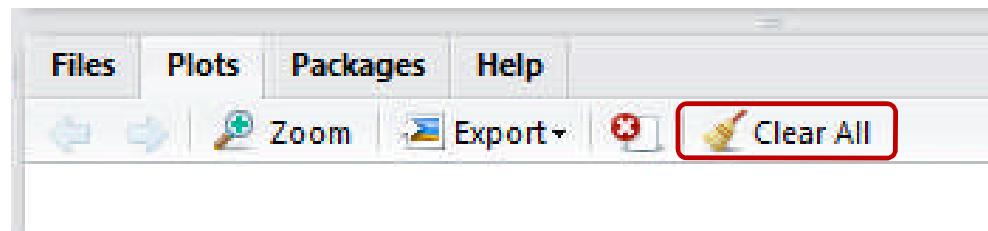
```
# Restore defaults
```

```
par(old.par)
```

```
plot(1:3)
```



To return to default plotting



Selecting the Clear All command in the plotting window resets all figures and sets `par()` to the default values. Use this option when you have gone too far and can't get back to a nice simple plotting screen.

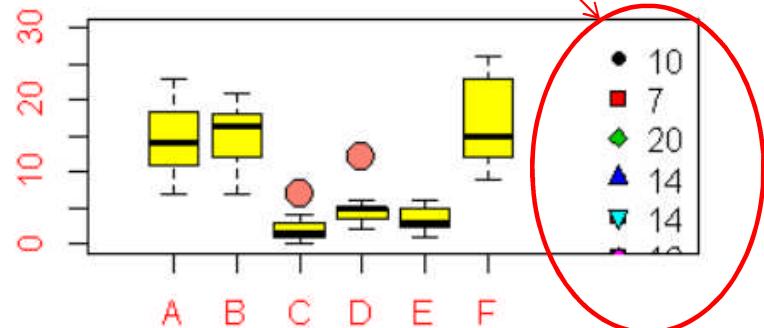
Vector options for plotting

- Many plotting options can handle vectors, each element applies to one point, are recycled
- Different point characters: `pch=1:5`
- Different point letters: `pch=c("a", "g", "t", "c")`
- Different colors: `col=1:5`
- Different sizes: `cex=1:5`

Adding *Legends*

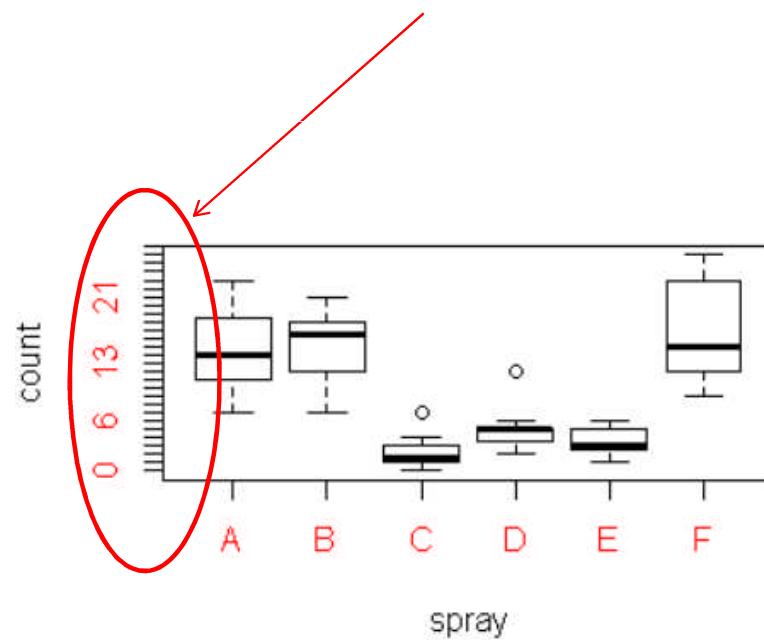
- Legends do not use anything in the plot
- Look up help on the legend function (`?legend`), note that most options in `par()` can be used too

```
legend(x="topright",      # also "bottomleft" etc., and can use x=100, y=100  
       Legend=InsectSprays[,1], # vector of numbers as text strings  
       pt.bg=1:5,              # background color of points  
       pch=21:25,              # vector of symbol type  
       bty="n")                # no box type
```



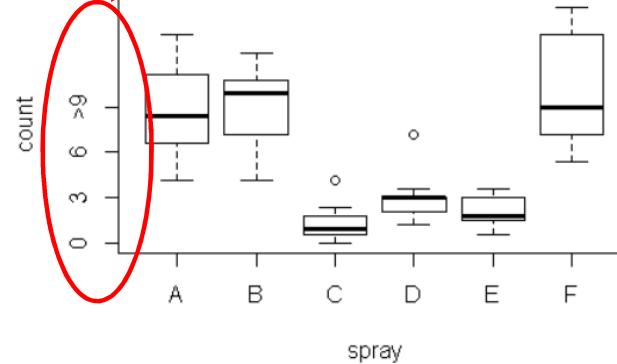
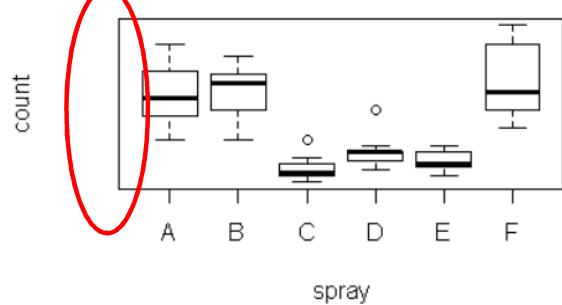
Axis properties

- Tick mark labeling using yaxp and xaxp
 - $c(\text{min}, \text{max}, \text{number of spaces between intervals})$
- > `plot(count ~ spray, data = InsectSprays, yaxp = c(0, 30, 30))`



More advanced axis properties

- For more control over axes, use the `axis()` function
- First create the plot but suppress the x or y axis using `xaxt="n"` and `yaxt="n"`
- Then add axes to whichever side they are needed
 - > `plot(count ~ spray, data = InsectSprays, yaxt = "n")`
 - > `axis(side=2, at=seq(0,15,5), labels=c(0,3,6,>"9"))`
- Note: Labels will only show if there is space to print



Adding text() using locator()

- Interactive function: click on the plot and it returns the x and y coordinates.

```
> locator(1)      Omit the 1 for multiple clicks, press <esc> to exit
```

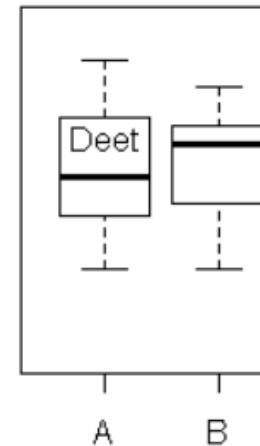
```
> locator( )
```

```
$x [1] 1.077401 2.196635 2.990929 4.110163 4.796144 6.186160
```

```
$y [1] 15.818076 14.873469 1.806414 4.167930 4.010496 18.337026
```

- Add text at those coordinates

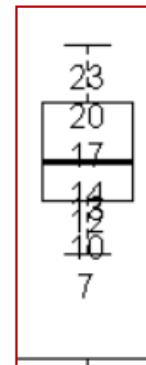
```
text(x=1, y=17, label="Deet")
```



Labeling points using text()

- Look up the help on ?text
- Can use vectors for x, y, and the text strings
- After creating the plot, call text()
 - pos=1 below
 - pos=2 to the left
 - pos=3 above
 - pos=4 to the right

```
text(x=InsectSprays$spray, y=InsectSprays$count,  
     labels=InsectSprays[,1], pos=1)
```



Interactive point labeling

- If you don't want to label all your points but there are a few outliers

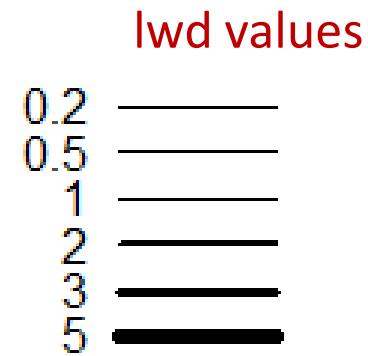
```
plot(count ~ spray, data = InsectSprays)
identify(x=InsectSprays$spray, y=InsectSprays$count,
         labels=InsectSprays[,1], n=5)
```

- Click near the 5 points to see their positions.

```
[1] 9 21 26 61 64
```

Lines widths and types

- `lwd` -- line widths



?lines shows values for

- `lty` -- line types

lty values

1	-----	solid
2	-----	dashed
3	dotted
4	dotdash
5	----	longdash
6	---	twodash

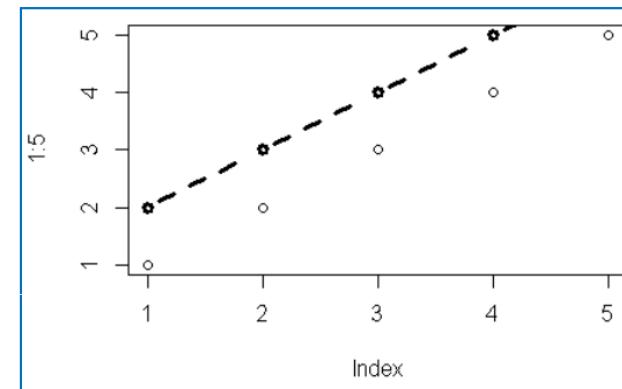
More plot types

- In the `plot()` command, `type` specifies the type of plot to be drawn
 - "p" points
 - "l" lines
 - "b" both lines and points
 - "c" lines part alone of "b"
 - "o" overplotted
 - "h" histogram-like (or high-density) vertical lines
 - "s" stair steps
 - "n" for no plotting

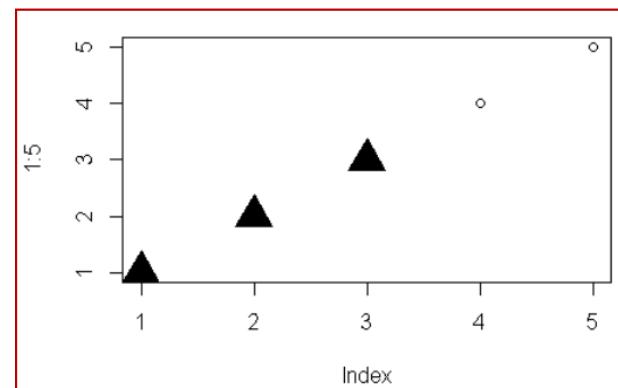
Adding points or lines

- You can add a series of points or lines to the current plot using `points()` and `lines()`

```
> plot(1:5)  
> lines(x=seq(1,5,1), y=c(2,3,4,5,6), +  
  type="b", lwd=3, lty=2)
```



```
> plot(1:5)  
> points(x=seq(1:3),  
  y=seq(4:6),cex=3, pch=17)
```



References

- <http://www.r-bloggers.com/exploratory-data-analysis-useful-r-functions-for-exploring-a-data-frame>

More Graphics in R

4/2/2016.

Table and Bins

```
> my.data=c(22,38,12,23,29,18,16,24)
> table(my.data)      # count of each value
my.data
12 16 18 22 23 24 28 29
  1   1   1   1   1   1   1   1
-----
> bins = c(10, 15, 20, 25, 30)
> temp.table = cut(my.data, bins)
> table(temp.table)
(10,15] (15,20] (20,25] (25,30]
    1         2         3         2
```

Stem Leaf Plot

```
# Note: my.data = c( 12 16 18 22 23 24 28 29 )
> stem(my.data)
```

```
The decimal point is 1 digit(s) to the right of the |
-----+
stem| leaves  # Actual numbers ={concat(stem, split(//,leaves))}

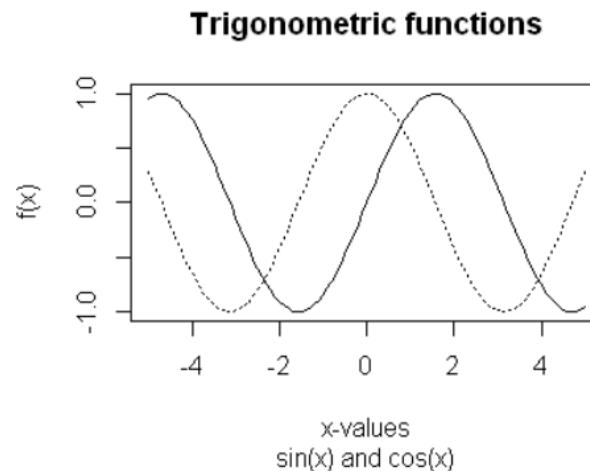
-----+
1 | 2      # => 12 [decimal]
1 | 68     # => 16,18  (1=stem, 6,8 are leaves).
2 | 234    # => 22,23,24
2 | 89     # => 28,29
-----+
```

Plot Sin, Cos, Lines, Title

```
xs <- seq(-5, 5, .1)
plot(xs, sin(xs),
      xlab = "x-values",
      ylab = "f(x)",
      type = "l")

lines(xs, cos(xs), lty = 3)

title(
  "Trigonometric functions",
  "sin(x) and cos(x)")
```



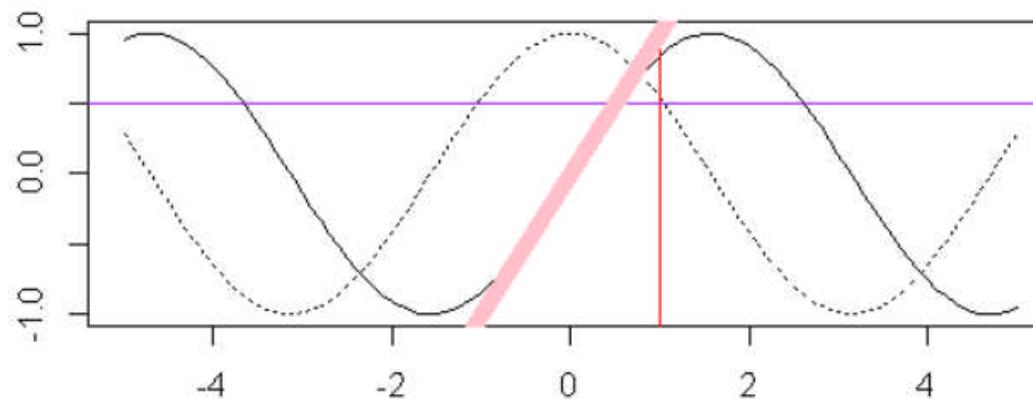
abline: draw lines on graph

- `abline(a=NULL, b=NULL, h=NULL, v=NULL, ...)`
a, b : single values specifying the **intercept** and the **slope** of the line
- **h** : the **y-value(s)** for horizontal line(s)
- **v** : the **x-value(s)** for vertical line(s)

```
abline(v=1,col='red')
```

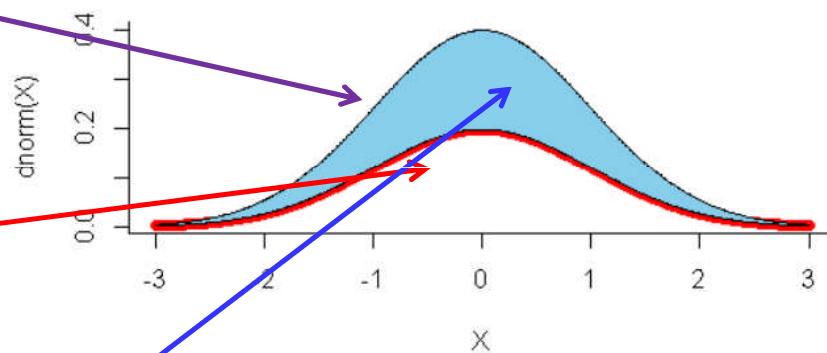
```
abline(h=0.5,col='purple')
```

```
abline(a=0,b=1,col='pink',lwd=8)
```



Shading

```
1. x <- seq(-3,3,0.01);  
2. y1 <- dnorm(x,0,1)  
3. plot(x,y1,type="l",  
       bty="L", xlab="X",  
       ylab="dnorm(X)" )  
4. y2 <- 0.5*dnorm(x,0,1)  
5. points(x,y2, type="p",  
           col="red")  
6. polygon( c(x,rev(x)),  
           c(y2,rev(y1)),  
           col="skyblue")
```



References

1. <http://www.alisonsinclair.ca/2011/03/shading-between-curves-in-r/>

Probability Distributions in R

4/2/2016.

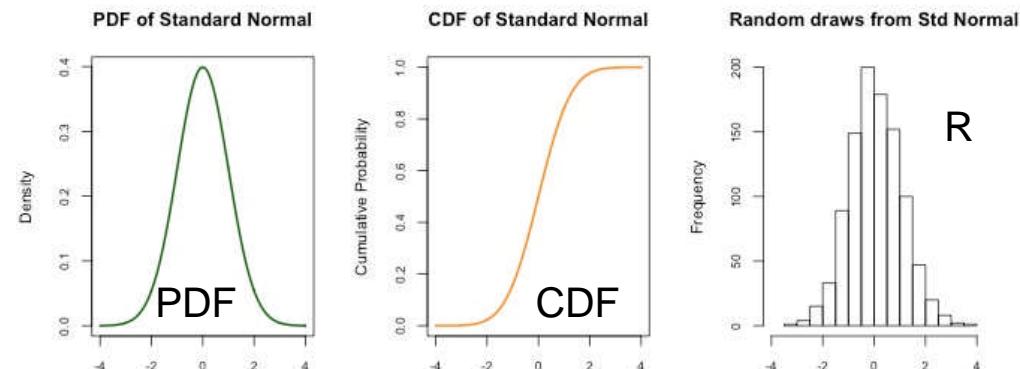
Builtin Distributions: r,d,p,q

> help(Distributions)

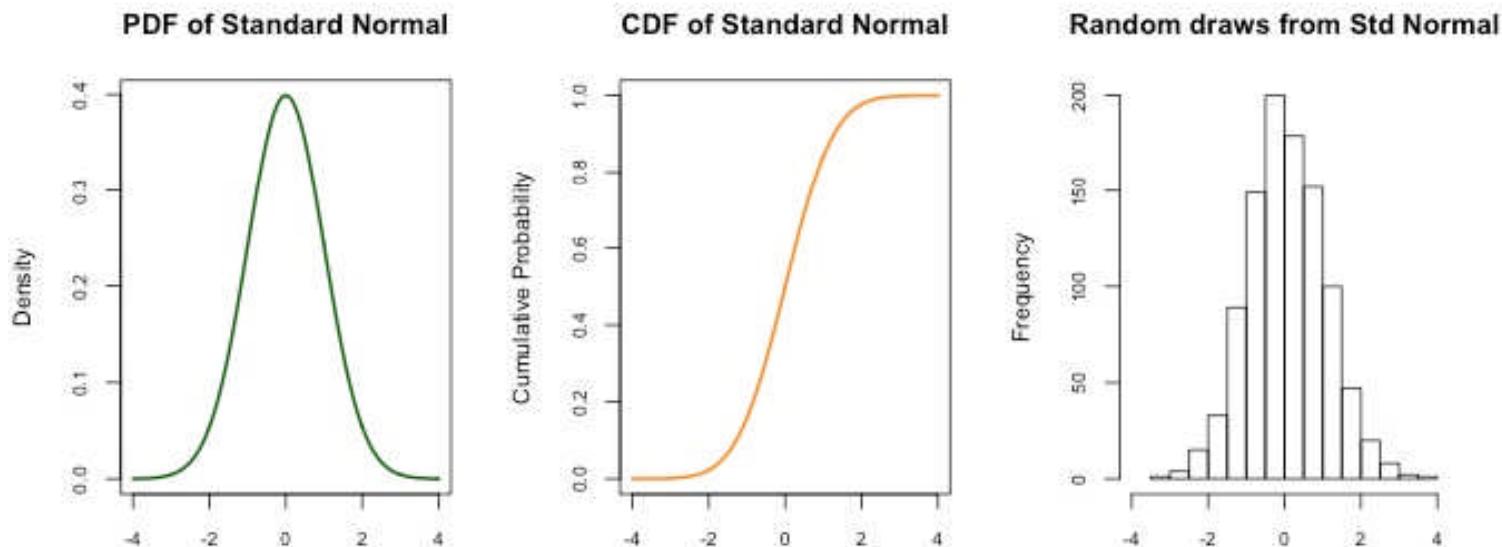
For every distribution there are four commands. The commands for each distribution are pre-pended with a letter to indicate the functionality:

1. “r” returns *randomly* generated numbers...
2. “d” returns the height of the *probability Density function* (PDF)
3. “p” returns the (\int PDF) *cumulative density function* (CDF)
4. “q” returns the (*Quantiles*) *inverse* of CDF.

See next slide for
diagram and examples.

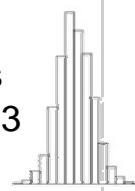


dnorm, pnorm, rnorm

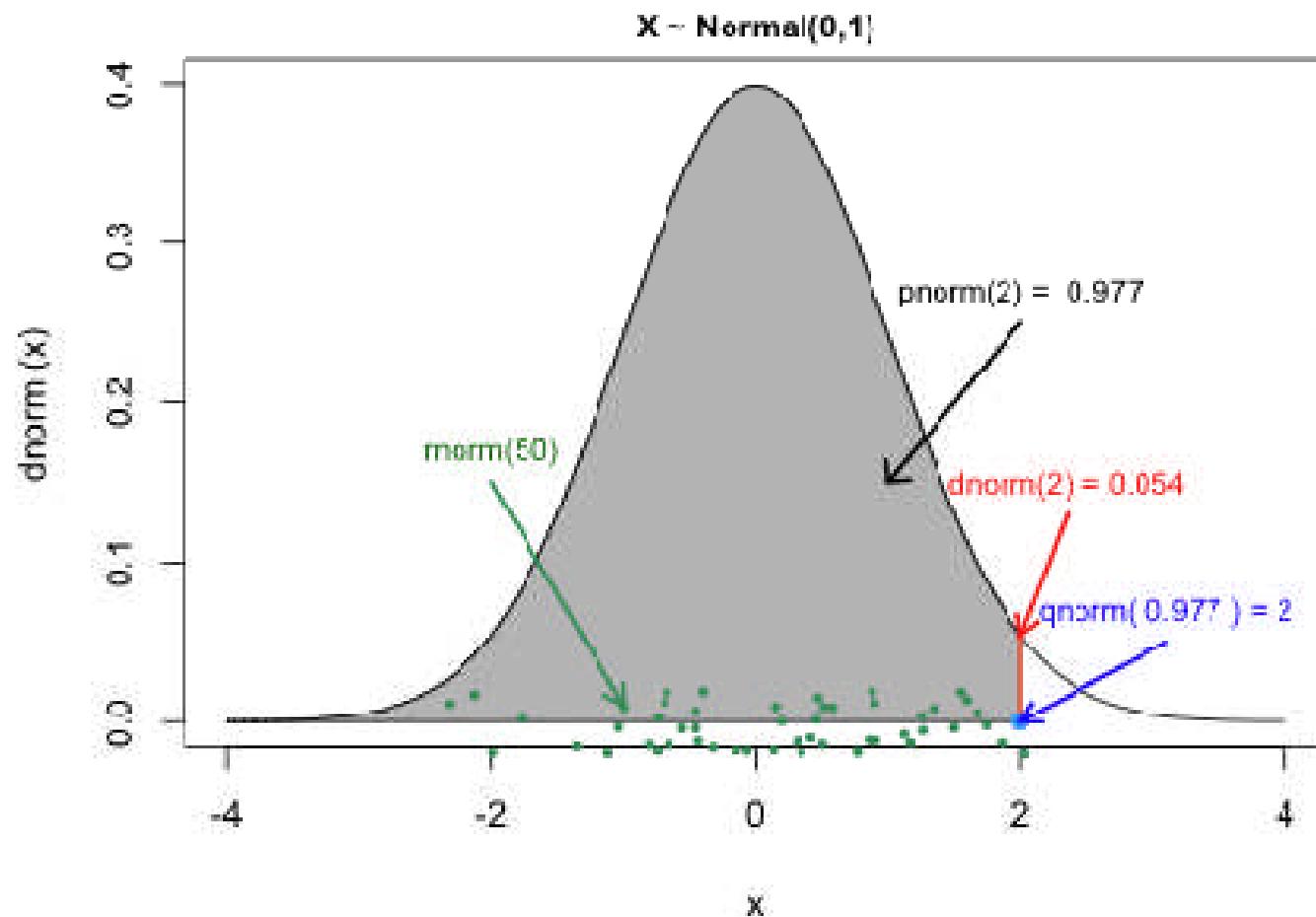


pnorm(x)
CDF is \int PDF

Normal prob distr functions

PURPOSE	SYNTAX	PROTOTYPE	EXAMPLE
RNORM	Generates random numbers from normal distribution	<code>rnorm(n, mean, sd)</code>	<code>rnorm(1000, 3, .25)</code> Generates 1000 numbers from a normal with mean 3 and sd=.25 
DNORM	Probability Density Function (PDF)	<code>dnorm(x, mean, sd)</code>	<code>dnorm(0, 0, .5)</code> Gives the density (height of the PDF) of the normal with mean=0 and sd=.5. 
PNORM	Cumulative Distribution Function (CDF)	<code>pnorm(q, mean, sd)</code>	<code>pnorm(1.96, 0, 1)</code> Gives the area under the standard normal curve to the left of 1.96, i.e. ~0.975 
QNORM	Quantile Function – inverse of pnorm	<code>qnorm(p, mean, sd)</code>	<code>qnorm(0.975, 0, 1)</code> Gives the value at which the CDF of the standard normal is .975, i.e. ~1.96

pnorm, dnorm, qnorm, rnorm



List of builtin distributions

from <http://www.statmethods.net/advgraphs/probability.html>

distribution	R name	distribution	R name
Beta	beta	Lognormal	lnorm
Binomial	binom	Negative Binomial	nbinom
Cauchy	cauchy	Normal	norm
Chisquare	chisq	Poisson	pois
Exponential	exp	Student t	t
F	f	Uniform	unif
Gamma	gamma	Tukey	tukey
Geometric	geom	Weibull	weib
Hypergeometric	hyper	Wilcoxon	wilcox
Logistic	logis		

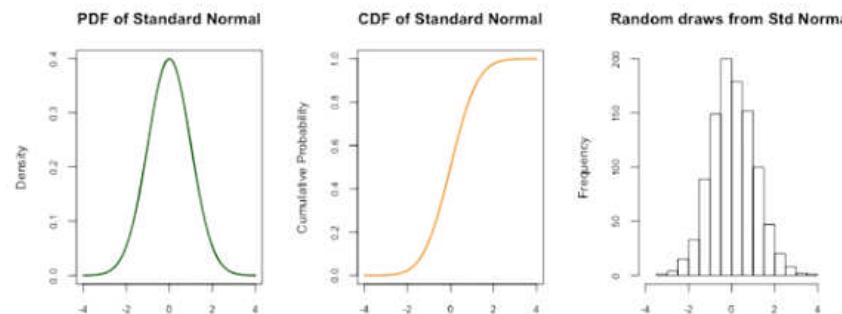
List of builtin distributions

<http://www.stat.umn.edu/geyer/old/5101/rlook.html>

Distribution	Functions			
Beta	pbeta	qbeta	dbeta	rbeta
Binomial	pbinom	qbinom	dbinom	rbinom
Cauchy	pcauchy	qcauchy	dcauchy	rcauchy
Chi-Square	pchisq	qchisq	dcchisq	rchisq
Exponential	pexp	qexp	dexp	rexp
F	pf	qf	df	rf
Gamma	pgamma	qgamma	dgamma	rgamma
Geometric	pgeom	qgeom	dgeom	rgeom
Hypergeometric	phyper	qhyper	dhyper	rhyper
Logistic	plogis	qlogis	dlogis	rlogis
Log Normal	plnorm	qlnorm	dlnorm	rlnorm
Negative Binomial	pnbino	qnbinom	dnbinom	rnbinom
Normal	pnorm	qnorm	dnorm	rnorm
Poisson	ppois	qpois	dpois	rpois
Student t	pt	qt	dt	rt
Studentized Range	ptukey	qtukey	dtukey	rtukey
Uniform	punif	qunif	dunif	runif
Weibull	pweibull	qweibull	dweibull	rweibull
Wilcoxon Rank Sum Statistic	pwilcox	qwilcox	dwilcox	rwilcox
Wilcoxon Signed Rank Statistic	psignrank	qsignrank	dsignrank	rsignrank

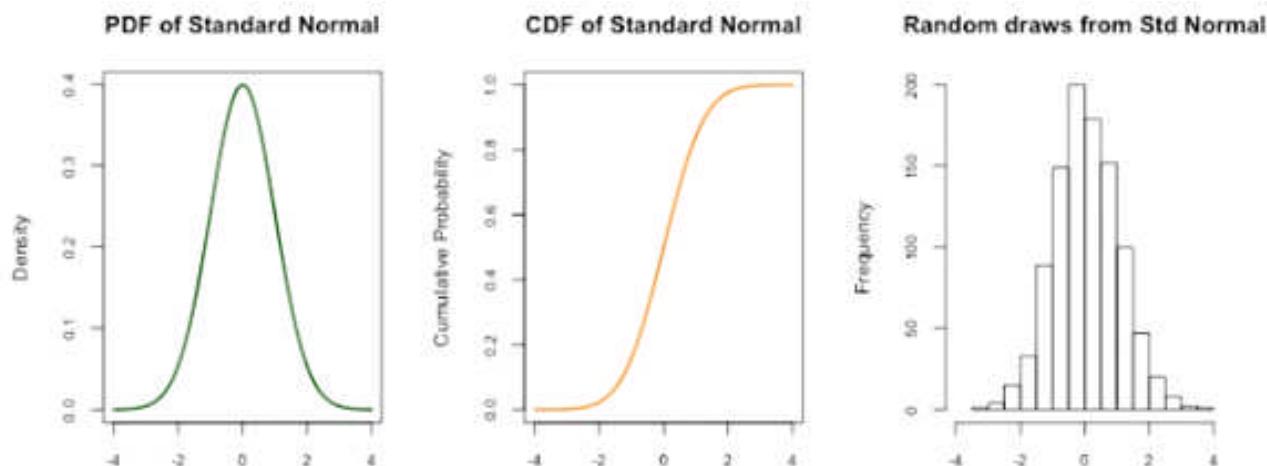
Plotting PDF, CDF, RNorm

```
set.seed(3000)
xseq<-seq(-4,4,.01)
1.densities <- dnorm(xseq,0,1) # PDF
2.cumulative <- pnorm(xseq,0,1) # CDF
3.randomdeviates <- rnorm(1000,0,1)
# Put 3 plots on a page
par(mfrow=c(1,3), mar=c(3,4,4,2))
```



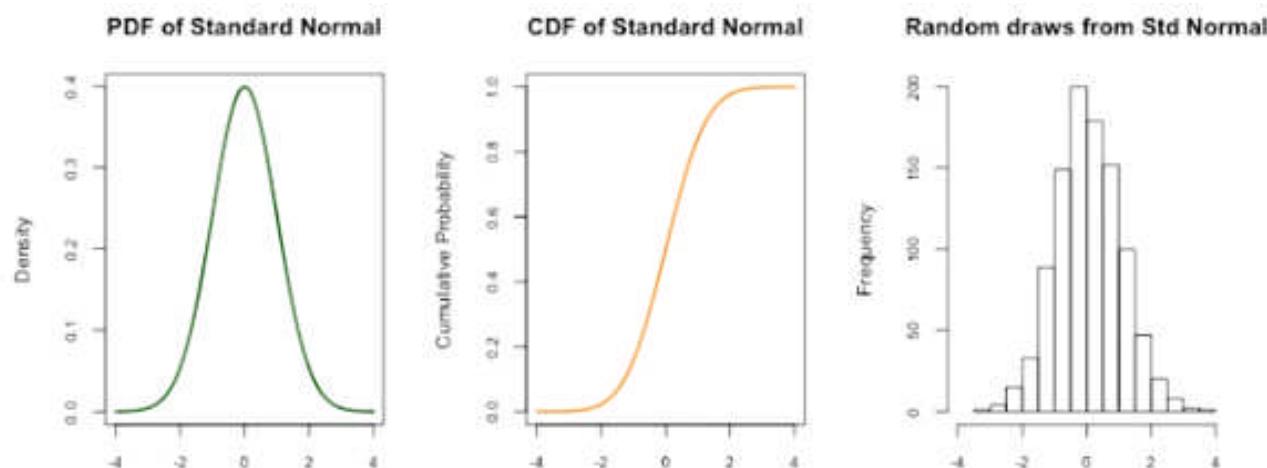
1. Plotting PDF, CDF, RNorm

```
1.plot(xseq, densities,  
       col="darkgreen",xlab="",  
       ylab="Density", type="l",lwd=2,  
       cex=2, main="PDF of Standard  
Normal", cex.axis=.8)
```



2. Plotting PDF, CDF, RNorm

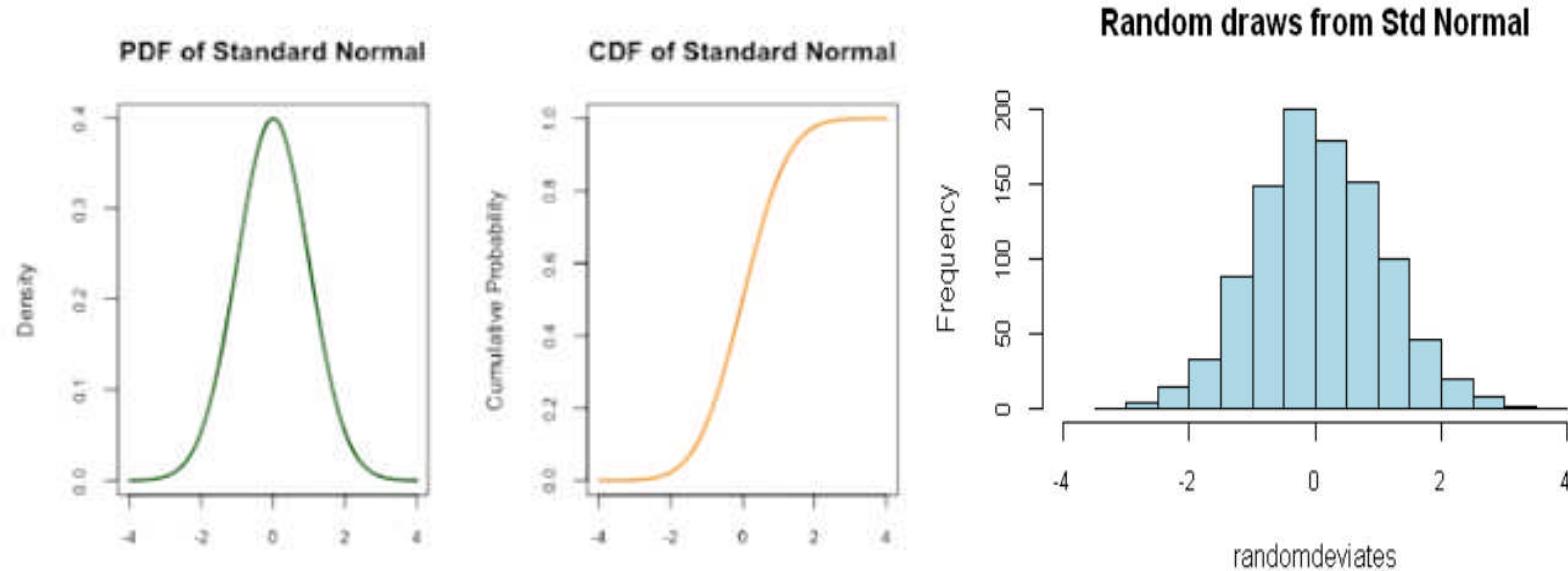
```
2. plot(xseq, cumulative,  
       col="darkorange", xlab="",  
       ylab="Cumulative  
Probability", type="l", lwd=2,  
       cex=2, main="CDF of Standard  
Normal", cex.axis=.8)
```



3. Plotting PDF, CDF, RNorm

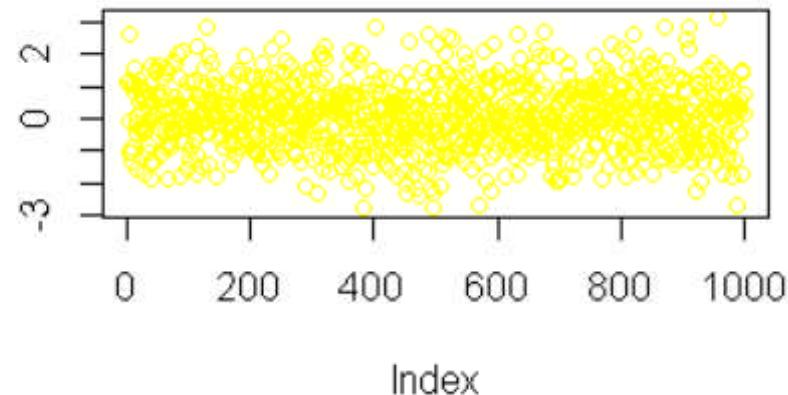
```
randomdeviates <- rnorm(1000,0,1)
```

```
hist(randomdeviates, main="Random  
draws from Std Normal",  
cex.axis=.8, xlim=c(-4,4),  
col="lightblue")
```



1000 Samples

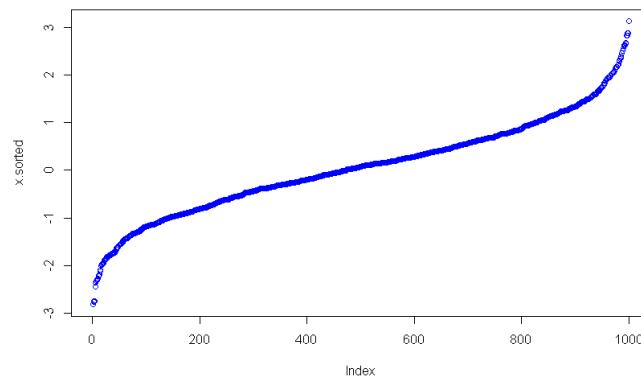
```
x <- rnorm(1000)  
plot(x,col="yellow")
```



See C:\doc3\stats-r\4norms and <http://casoilresource.lawr.ucdavis.edu/blog/rs-normal-distribution-functions-rnorm-and-pals/>

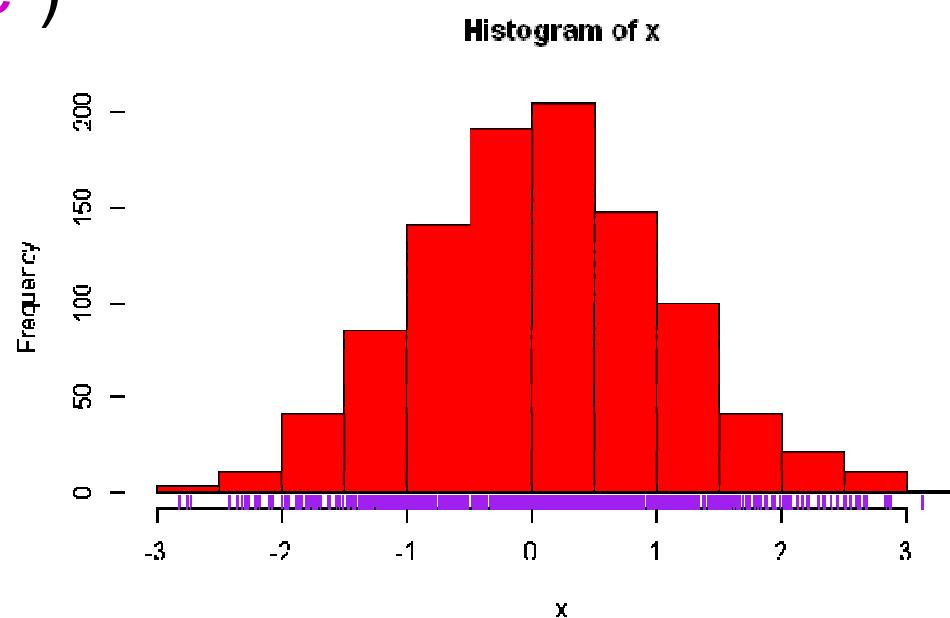
Plot sorted sample

```
s <- sort(x)  
plot(s, col="blue")
```



Sample histogram with rug

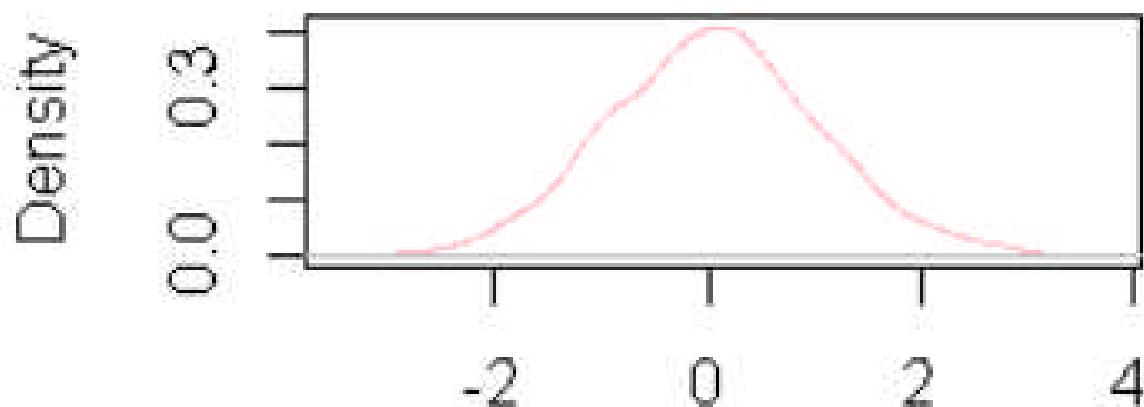
```
hist(x, col="red")  
rug(x, col="purple")
```



Sample Density plot

```
plot(density(x), col="pink")
```

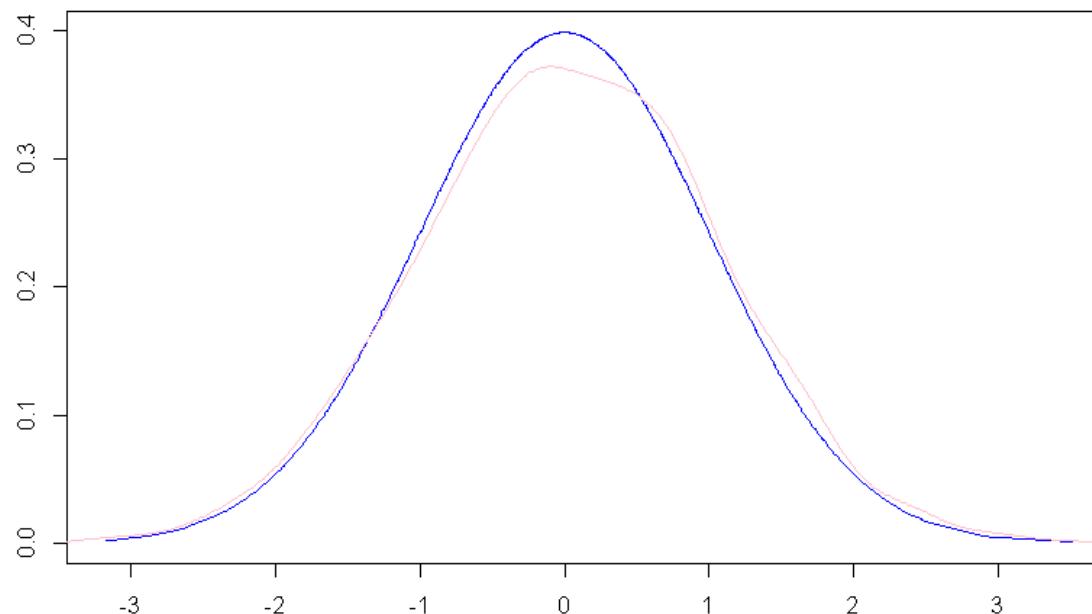
density.default(x = x)



N = 1000 Bandwidth = 0.2221

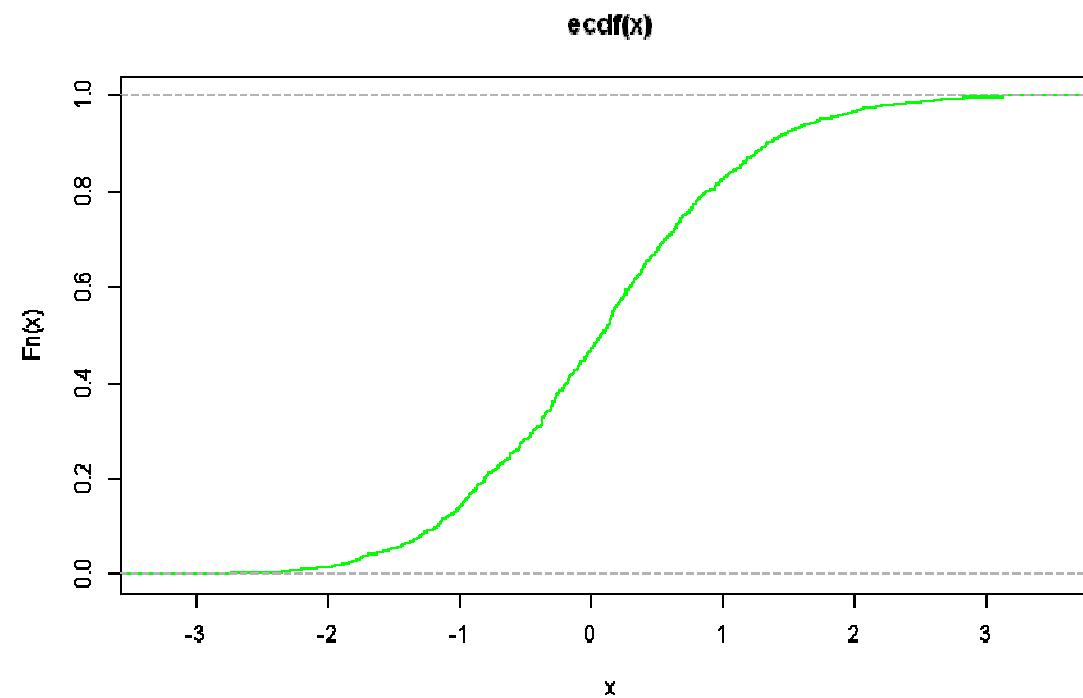
Compare Sample Density with Normal PDF (dnorm)

```
plot(density(x))  
lines(s,dnorm(s))
```



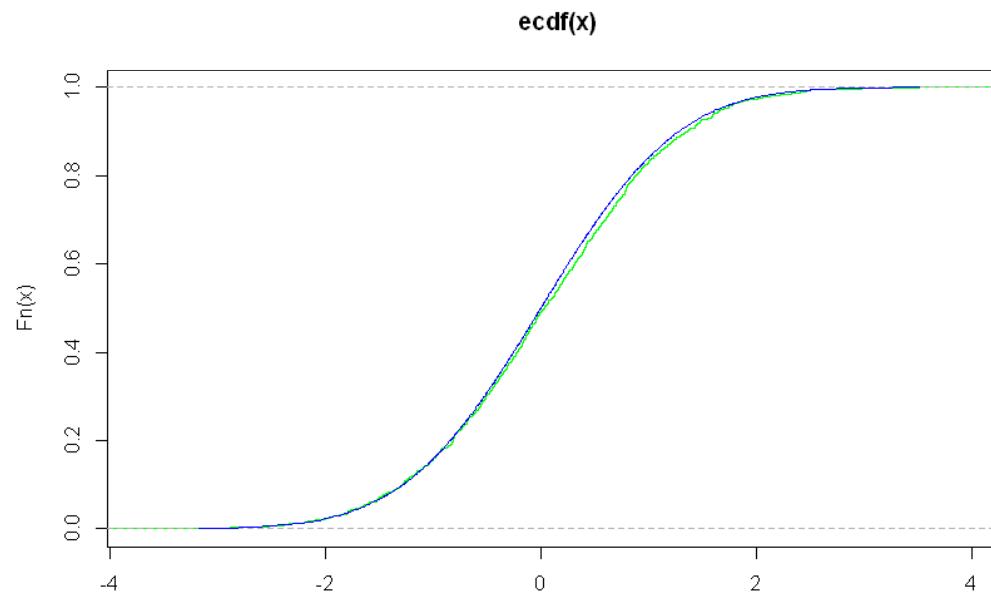
Sample to Empirical CDF

```
x.ecdf <- ecdf(x)  
plot(x.ecdf, col="green")
```



Compare Sample to ECDF with Normal CDF (pnorm)

```
x.ecdf <- ecdf(x)  
plot(x.ecdf, col="green")  
lines(s, pnorm(s), col="blue")
```



1. All plots together

```
# sample a normal distribution, with a mean of
# 5 and sd of 2, 100 times
x <- rnorm(100, mean=5, sd=2)

# sort in ascending order
x.sorted <- sort(x)

# compute the empirical cumulative
# distribution function
x.ecdf <- ecdf(x.sorted)

# plot the expected and actual probability
# density
plot(x.sorted, dnorm(x.sorted, mean=5, sd=2),
      type='l', ylim=c(0,1), ylab='Probability',
      xlab='Value', main='rnorm(), dnorm(),
      pnorm(), and qnorm()')
lines(density(x), col=1, lty=2)

# add the expected and actual cumulative
# probability
lines(x.sorted, pnorm(x.sorted, mean=5,
      sd=2), type='l', col=2)
lines(x.sorted, x.ecdf(x.sorted), type='l', col=2,
      lty=2)

# add the expected and actual p=0.5 (median)
# and p=0.95 quantiles
abline(v=qnorm(c(0.5, 0.95), mean=5, sd=2),
      col=3)
abline(v=quantile(x, probs=c(0.5, 0.95)),
      col=3, lty=2)

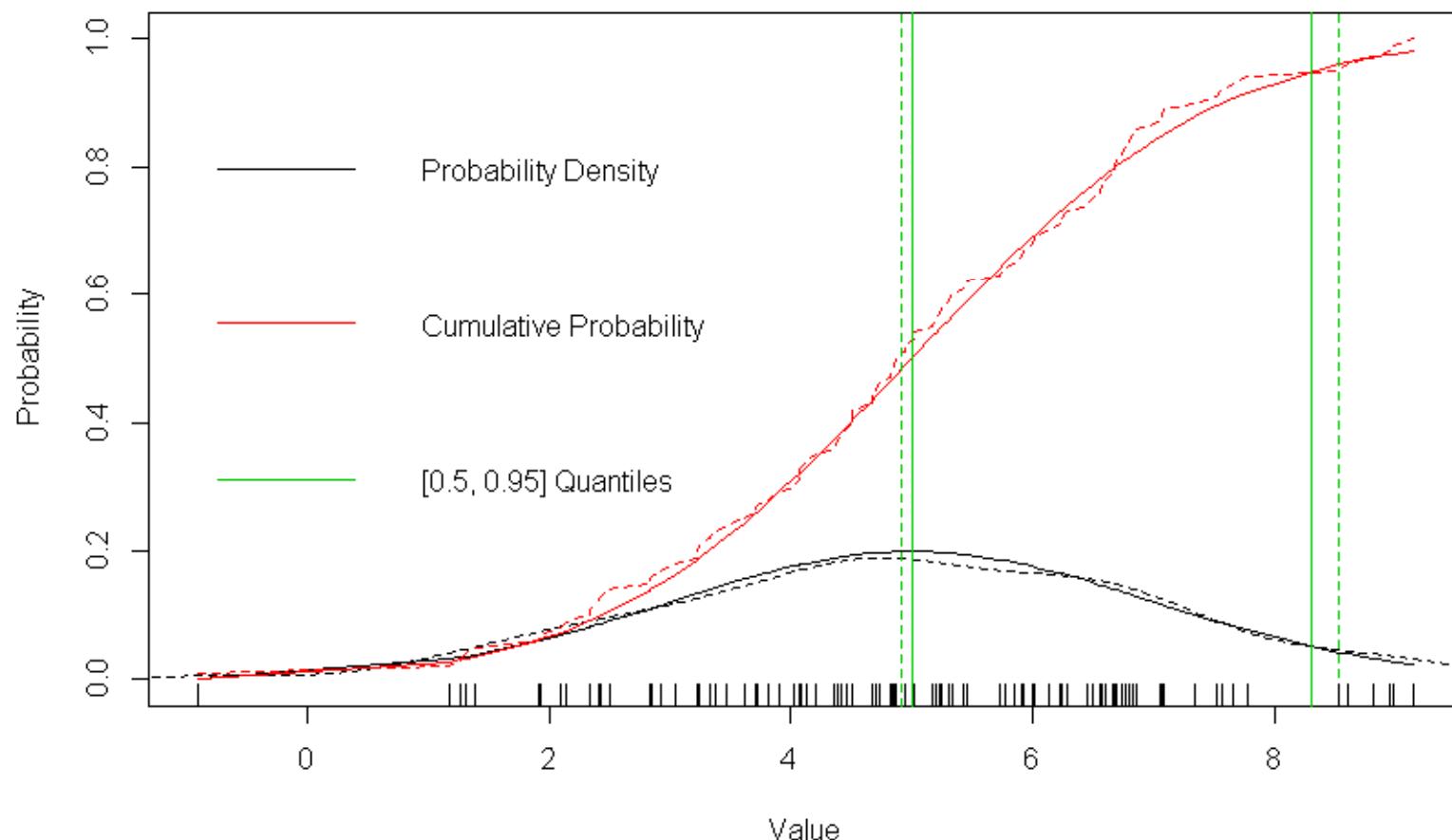
# add the original x values
rug(x)

# annotate
legend('topleft', legend=c('Probability
Density','Cumulative Probability',[0.5,
0.95] Quantiles'), lty=1, col=1:3, bty='n')
```

From <http://casoilresource.lawr.ucdavis.edu/blog/rs-normal-distribution-functions-rnorm-and-pals/>

2. All plots together

`rnorm()`, `dnorm()`, `pnorm()`, and `qnorm()`



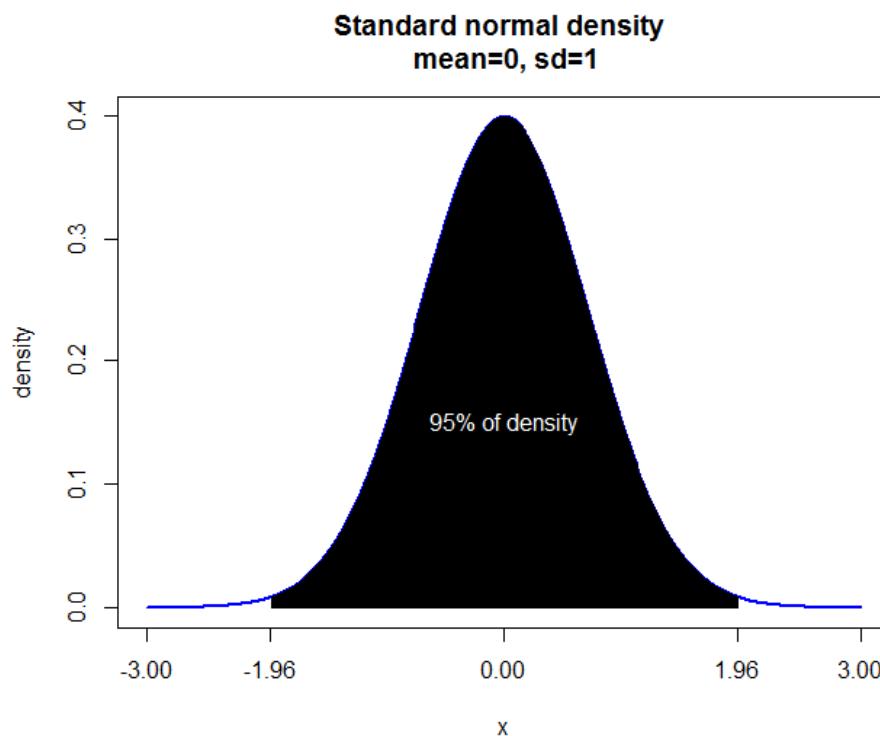
Exercise 3

The equation for the standard normal density is

$$\exp(-x^2)/\sqrt{2\pi}$$

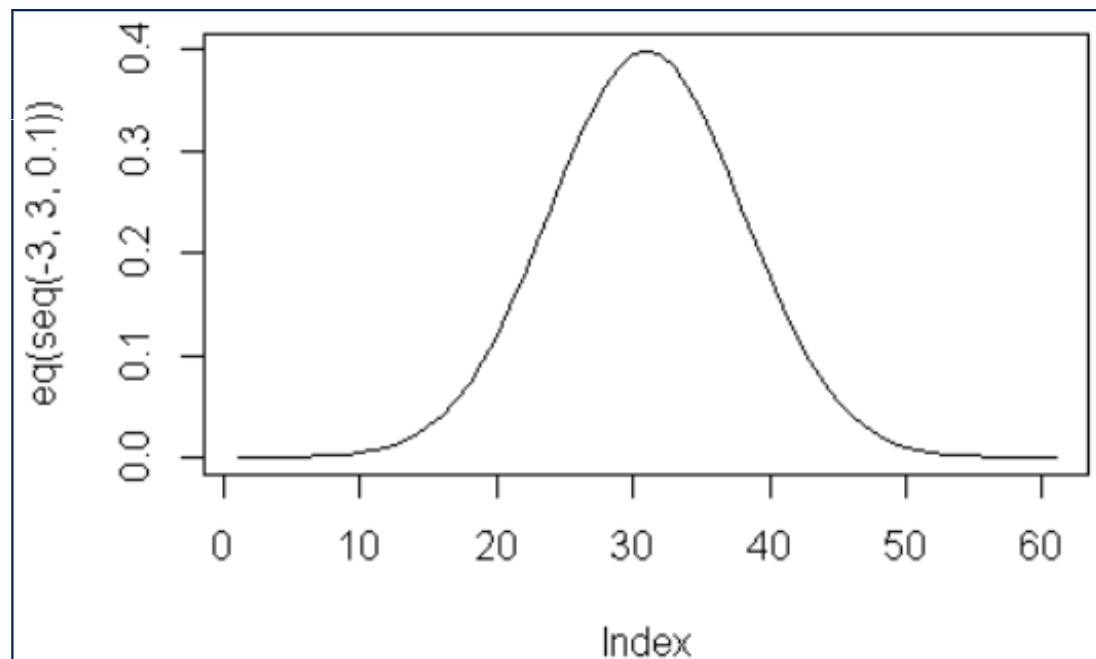
1. Create the plot on the right
 2. illustrate where 95% of the area falls, i.e.
- 1.96 ≤ x ≤ 1.96**

Hint: use `type` in two different ways.



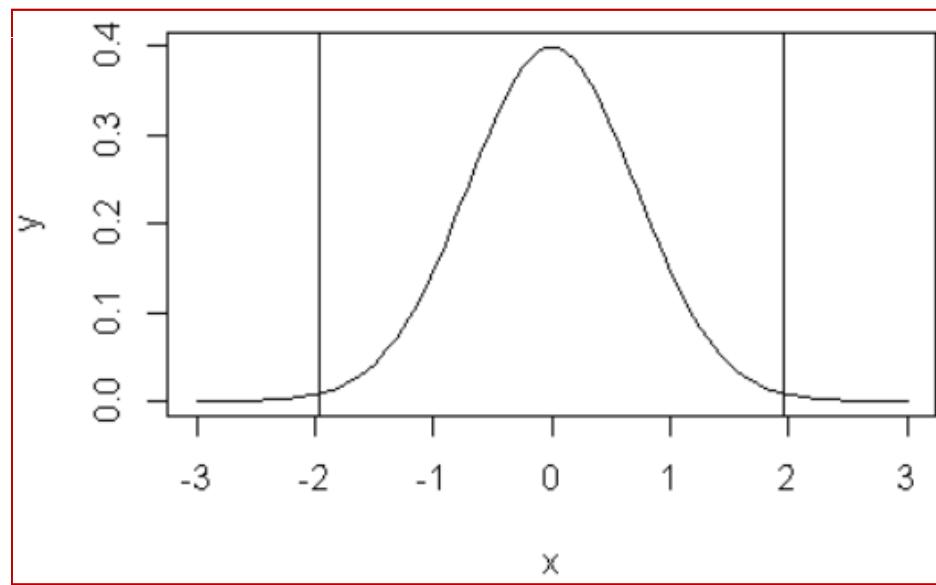
Solution: #1 plot

```
> eq = function(x){ exp(-x^2)/sqrt(2*pi) }  
> plot(eq(seq(-3,3,0.1)), type='l')
```



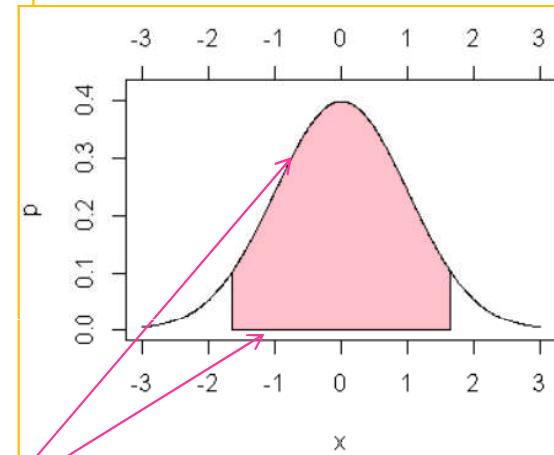
Solution: #2 Curve

```
> eq = function(x){ exp(-x^2)/sqrt(2*pi) }  
> curve(eq, from=-3, to=3, xlab="x", ylab="y")  
> abline(v=-1.96); abline(v=1.96);
```



Solution #3 Plot & Polygon

```
1. stdDev <- 1;  
2. x <- seq(-3,3,by=0.01)  
3. y <- dnorm(x, sd=stdDev)  
4. left <- qnorm(0.05, sd=stdDev)  
5. right <- qnorm(0.95, sd=stdDev)  
6. plot(x,y,type="l",  
7.       xaxt="n",ylab="p",  
8.       ylim=c(0,max(y)*1.05),  
9.       xlim=c(min(x),max(x)),)  
10.axis(1)  
11.axis(3)  
12.xReject <- seq(left,right,by=0.01)  
13.yReject <- dnorm(xReject, sd=stdDev)  
14.polygon(c(xReject,  
15.           xReject[length(xReject)],xReject[1]),  
16.           c(yReject,0, 0), col='pink')
```

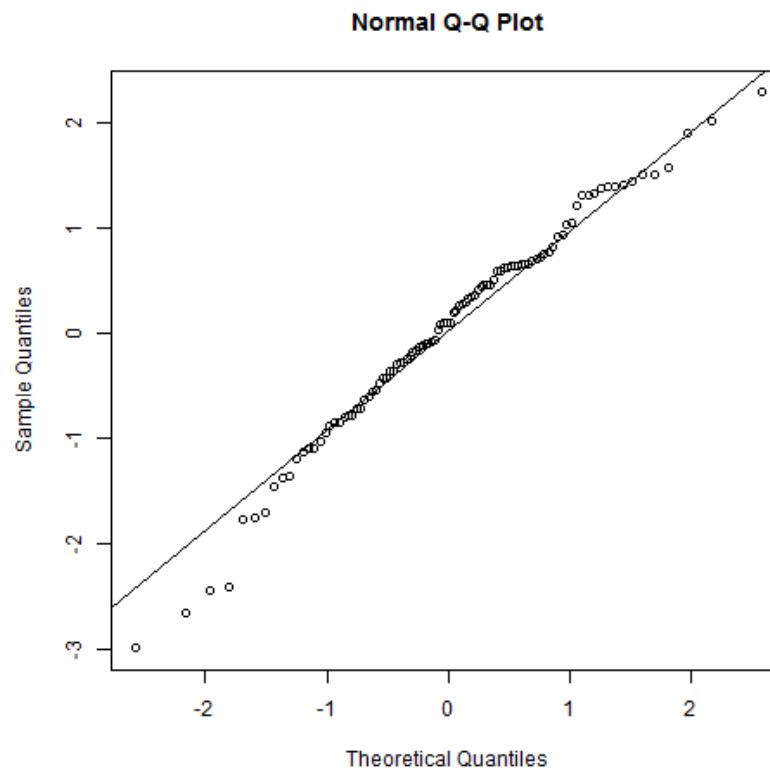


From <http://www.cyclismo.org/tutorial/R/intermediatePlotting.html>

QQPlot (Quantile Quantile Plot)

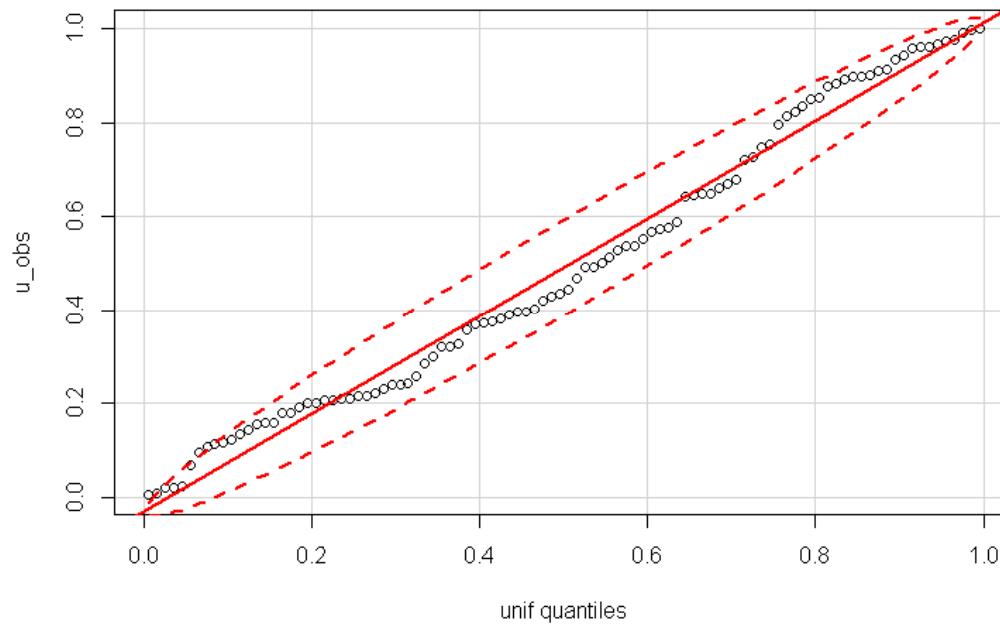
- A **q-q plot** is a plot of the quantiles of the first data set against the quantiles of the second data set.
- By a quantile, we mean the fraction (or percent) of points below the given value.

```
set.seed(42)
y <- rnorm(100)
qqnorm(y) // norm!
qqline(y)
```



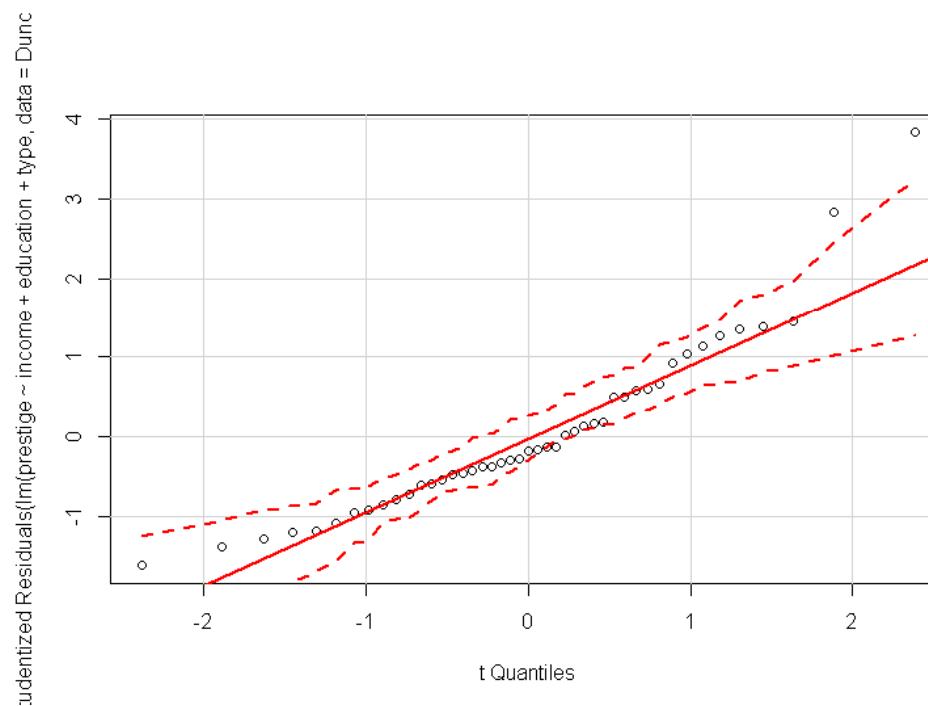
car.qqPlot of Uniform Ran#

```
> library(car)
> set.seed(9)
> u_obs <- runif(100)
> qqPlot(u_obs,
  distribution="unif",
  line="robust")
```



car.qqPlot & Regression

```
library(car)
qqPlot(
  lm(prestige ~ income
    + education + type,
  data=Duncan),
  envelope=.99)
```



References

1. <http://www.cyclismo.org/tutorial/R/probability.html>
2. <http://www.stat.umn.edu/geyer/old/5101/rlook.html>
3. <http://data.library.virginia.edu/understanding-q-q-plots/>

Regression in R

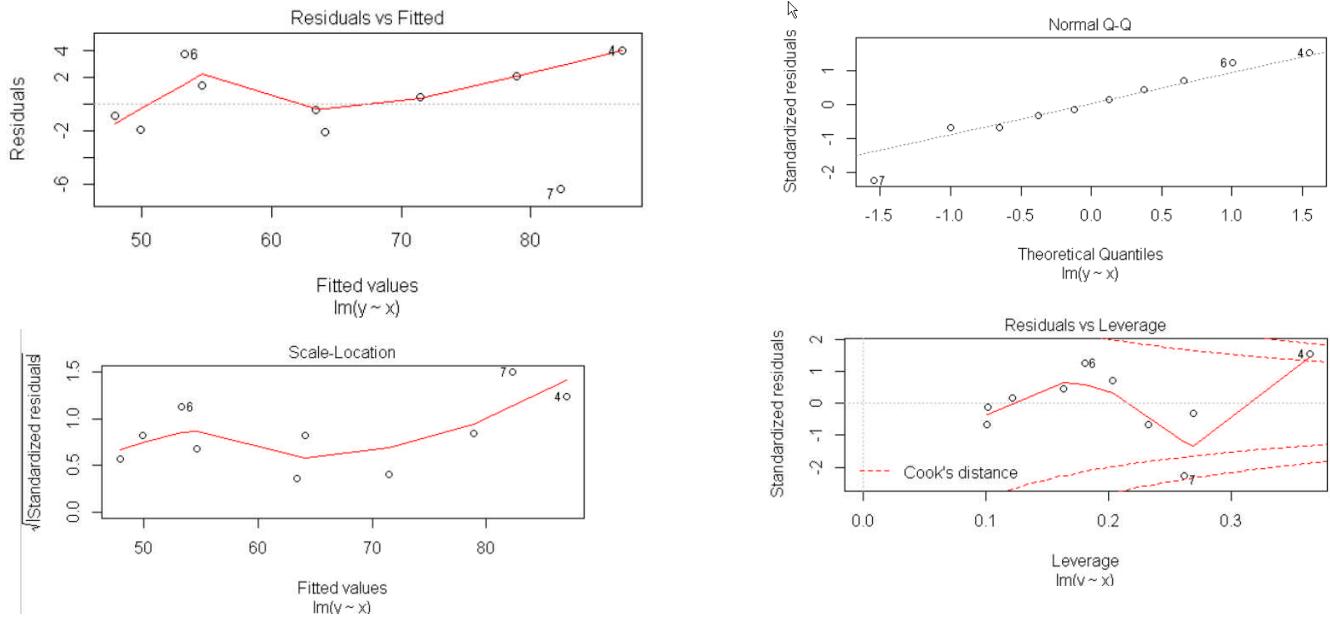
4/2/2016.

Regression: $\text{Im } y \sim x$

From http://www.tutorialspoint.com/r/r_linear_regression.htm

1m (dependent vars ~ independent vars)

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
m <- lm(y~x)
```



Predict using lm

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
rel <- lm(y~x)
# Find weight of a person with height 170.
a <- data.frame(x=170)
result <- predict(rel, a)
print(result)
```

1
76.22869

Continued next slide

Save image as 'predicted.jpg'

```
png(file = "predicted.jpg")
plot(y,x,col="blue",
      main="Ht vs Wt Regression",
      abline(lm(x~y)),
      cex = 1.3,pch=16,
      xlab="Weight in Kg",
      ylab="Height in cm")
dev.off()
```



Time Series and Finance in R

4/2/2016.

Quantmod to plot stocks

```
> install.packages("quantmod")
> library(quantmod)
> getSymbols(c("YHOO","GOOG","DEXJPUS","XPTUSD"))
> barChart(GOOG)
```



Many ways to plot

```
> candleChart(GOOG,multi.col=TRUE,theme="white")
```



MACD, BBands

```
> chartSeries(GOOG);  
> addMACD(); # Moving Average Convergence Divergence  
> addBBands(); # Bollinger Bands
```



References

1. **Interactive JS charts with dygraph for R3.1 only**
<https://rstudio.github.io/dygraphs/gallery-shaded-regions.html ..>
2. <http://www.quantmod.com/examples/intro/>

Statistical Tests with R

8/10/2014.

from http://zoonek2.free.fr/UNIX/48_R/08.html

List of tests

```
> library(help="stats")
```

```
> apropos(".test")
```

".valueClassTest"	"ansari.test"	"bartlett.test"	"binom.test"
"Box.test"	"catTestchisq"	"chisq.test"	"confint.htest"
"conTestkw"	"cor.test"	"coxph.wtest"	"file_test"
"fisher.test"	"fligner.test"	"formatTestStats"	"friedman.test"
"getLatestSource"	"KhmaladzeTest"	"kruskal.test"	"ks.test"
"mantelhaen.test"	"mauchly.test"	"mcnemar.test"	"mood.test"
"oneway.test"	"ordTestpo"	"pairwise.prop.test"	"pairwise.t.test"
"pairwise.wilcox.test"	"poisson.test"	"power.anova.test"	"power.prop.test"
"power.t.test"	"PP.test"	"print.t.test.cluster"	"prop.test"
"prop.trend.test"	"quade.test"	"rq.test.anowar"	"rq.test.rank"
"shapiro.test"	"simple.median.test"	"simple.z.test"	"spearman.test"
"survregDtest"	"t.test"	"t.test.cluster"	"var.test"
"wilcox.test"			

How to correctly perform a Test

0. Choose a confidence level, a test
1. Plot the data
2. Check the assumptions.
Change the test if needed.
3. Perform the test.
Check the confidence interval.

Assumptions

- Most of these tests are only valid for gaussian variables.
- Furthermore, if there are several samples, they often ask them to be independent and have the same variance.

Reading a test result

- One would expect those functions to yield a result as "*Null hypothesis rejected*" or "*Null hypothesis not rejected*" -- bad luck. The user has to know how to interpret the results, with a critical eye.
- The **result is mainly a number, the p-value**. It is the probability to get a result at least as extreme. If it is close to one, we do not reject the hypothesis, i.e., the test did not find anything statistically significant; if it close to zero, we can **reject the null hypothesis**.

Reading a test result

- More precisely, before performing the test, we choose a confidence level alpha (often 0.05; for human health, you will be more conservative and choose 0.01 or even less; if you want results with little data, if you do not mind that those results are not reliable, you can take 0.10):
- if $p < \alpha$, you reject the null hypothesis, if $p > \alpha$, you do not reject it.

Example of “T” test

```
> x <- rnorm(200)
```

```
> t.test(x)
```

One Sample t-test data: x

t = 3.1091, df = 199, **p-value = 0.002152**

alternative hypothesis: true mean is not equal to 0

95 percent confidence interval: 0.0785 0.351

sample estimates: mean of x=0.214

If we reject the null hypothesis (here: "the mean is zero"), we will be wrong with a probability **0.002152**, i.e., in 2 cases out of 1,000.

Is the data normally distributed?

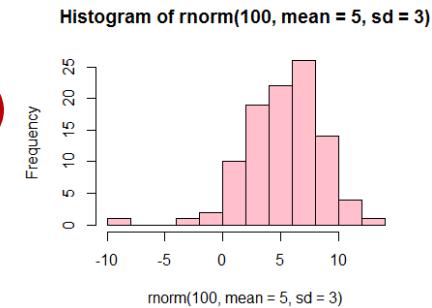
Use: Shapiro-Wilk normality test

- Normal Data.

```
> shapiro.test(rnorm(100, mean = 5, sd = 3))
```

... p-value = 0.6152 Likely to be normal.

```
> hist( rnorm(100, mean = 5, sd = 3), col='pink')
```

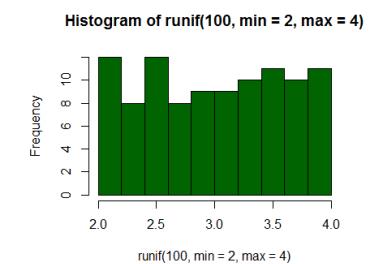


- Non normal data

```
> shapiro.test(runif(100, min = 2, max = 4))
```

.. p-value = 0.0001286 Unlikely to be normal

```
> hist(runif(100, min = 2, max = 4), col='dark green')
```



References

- *Estimators and Statistical Tests* by Zonnekyn from http://zoonek2.free.fr/UNIX/48_R/08.html
(URL blocked by Indian Govt, use google search instead)

Estimators and Statistical Tests with R

8/10/2014.
from http://zoonek2.free.fr/UNIX/48_R/08.html

Introduction

We want to answer a question of the kind

- "*Does tobacco increase the risk of cancer?*"
- "Does the proximity of a nuclear waste reprocessing plant increase the risk of leukemia?"
- "Is the mean of the population from which this sample was drawn zero, given that the sample mean is 0.02?"

Sampling

Let us detail the problem

- "Do the two samples have the same mean?"

It is a simplification of the problem

- "Do the two samples come from the same population?"

Sample

- Let us consider a first population, on which is defined a statistical variable (with a gaussian distribution), from which we get a sample.
- We do the same for a second population, with the same population mean.
- We can then consider the statistical variable: **sample mean in the first sample - sample mean in the second sample** and find its distribution.

P value

- If we measure a certain value of this difference, we can compute the probability of obtaining a difference at least as large.
- If $P(\text{difference} > \text{observed difference}) < \alpha$,
- (for a given value of α , say 0.05),
- we reject the hypothesis "the two means are equal", with a risk equal to α .
- Beware again, those tests are only valid under certain conditions (Gaussian variables, same variance, etc.).

Means are equal?

- If we really wish to be rigorous, we do not consider a single hypothesis, but two: for instance "the means are equal" and "the means are different"; or "the means are equal" and "the first mean is larger than the second".
- We would use the second formulation if we can a priori reject the fact that the first mean is lower than the second -- but this has to come from information independent from the samples at hand.

What can we PROVE?

- The statistical tests will **never tell "the hypothesis is true"**: they will merely reject or fail to reject the hypothesis stating **"there is nothing significant"**.
- This is very similar to the development of science as explained by Karl Popper:
- We never prove that something is true, we merely continuously try to prove it wrong and fail to do so.

H0 (null hypothesis)

- Let us consider two hypotheses: the null hypothesis H0, "**there is no noticeable effect**" and the alternative hypothesis H1, "**there is a noticeable effect**".
- H0 is sometimes called the "*conservative hypothesis*", because it is the hypothesis we keep if the results of the test are not conclusive.

H1 (alternative hypothesis)

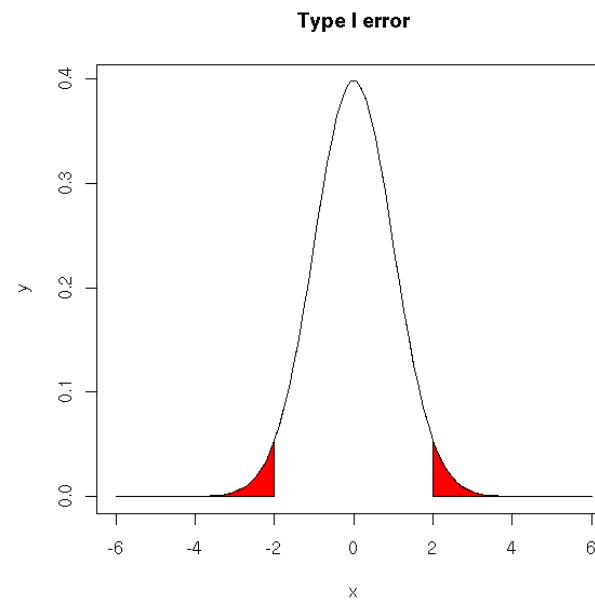
- The alternative hypothesis can be symmetric or not (asymmetric).
- To choose an asymmetric hypothesis means that we reject, a priori, half of the hypothesis: it can be a prejudice, so you should think carefully before choosing an asymmetric alternative hypothesis.

Type I error

- To wrongly reject the null hypothesis
- i.e. to wrongly conclude "there is an effect" or "there is a noticeable difference".

Type I error

- If we get extreme values, we shall reject, sometimes wrongly, the null hypothesis (that the mean is actually zero). The type I error corresponds to the red part in the following plot.

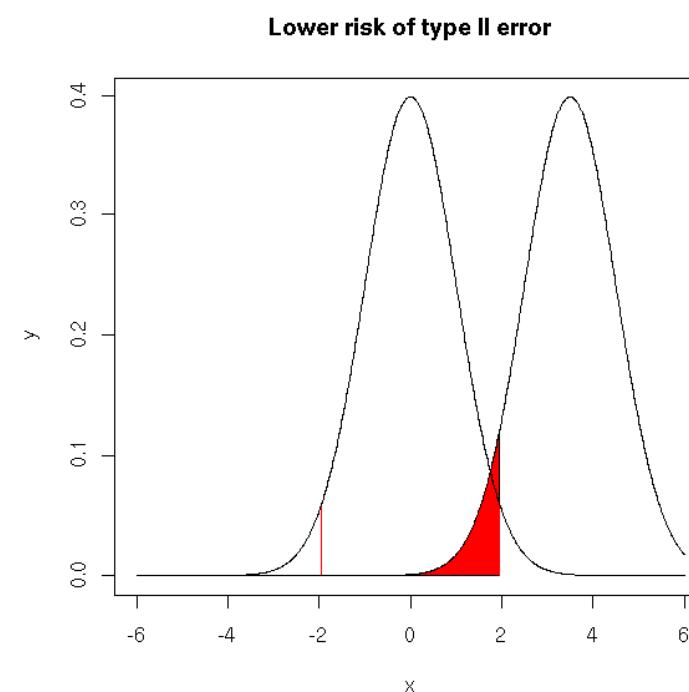
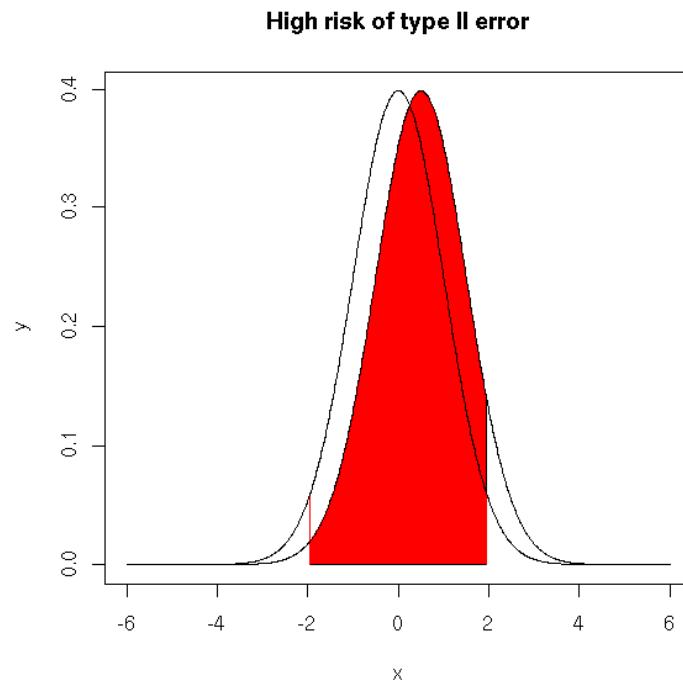


Type II error

- Wrongly accepting the null hypothesis (i.e., wrongly concluding "there is no statistically significant effect" or "**there is no difference**").
- In all rigor, it is not an error, because one never says " H_0 is true" but "we do not reject H_0 (yet)".
- It is not an error, but a missed opportunity.

Probability of Type II error

- If the two curves are sufficiently far apart (i.e., if the difference of means is more significant), the risk is much lower.



Power of a test

- power = $1 - P(\text{ type II error})$.
- the power is not a number but a function.
The null hypothesis is often of the form H_0 : " $\mu = \mu_0$ " and H_1 : " $\mu \neq \mu_0$ ".
- The power will depend on the actual value of μ : if μ is close to μ_0 , the type II error probability is high and the power is low; on the contrary, if μ and μ_0 are very different. the probability of type II error is lower and the power higher.

Use of power

- The power plays an important role when you design an experiment. Let us imagine that we want to know if the mean μ of a certain variable on a certain population equals μ_0 .
- We want to be able to detect a difference at least ϵ in at least 80% of the cases, with a type I error risk inferior to 5%: what should be the sample size?

Use of power

- In other words, we want that the power of the test $H_0: \mu = \mu_0$,
- $H_1: |\mu - \mu_0| > \epsilon$
- with a confidence level $\alpha=0.05$ be at least 0.80
- ("tradition" suggests a power equal to 0.80, and a confidence level 0.05).

power.t.test for Student's T-test.

```
> power.t.test(delta=.1, sd=1, sig.level=.05,  
    power=.80, type='one.sample')
```

One-sample t test power calculation

n = 786.8109, delta = 0.1,

sd = 1,

sig.level = 0.05

power = 0.8

alternative = two.sided

Confidence Interval

- Let us consider a random variable, whose distribution is not completely known: for instance, we know it is a gaussian distribution of variance 1, but the mean is unknown.
- If we estimate this mean as a single number, we are sure to be wrong: the actual mean might be close to our proposal, but there is no reason it should be exactly this one, up to the umpteenth decimal. Instead, we can give a confidence interval -- note the use of an indefinite article: there are many such intervals

Confidence Interval

- Here are two interpretations of this notion of "confidence interval".
 1. It is an interval in which we have a 95% probability of finding the sample mean.
 2. More naively, it is an interval that has a 95% probability of containing the population mean (i.e., the actual mean).
- Actually, these two interpretations are equivalent.

p-value and Confidence Interval

- From this, we get two interpretations of the p-value:
 - first, as the probability of getting results at least as extreme if H_0 is true (this is the definition);
 - second, it is the probability of being in "the" confidence interval.
- The p-value is NOT the probability that H_0 be true. To convince yourself of this, consider Gaussian random variable, of variance 1 and unknown mean.
- From a sample, we test H_0 : "the mean is 0" against H_1 : "the mean is not 0". We will get a certain p-value.

Parametric tests

- Most of the time, statistical tests assume that the random variables studied are Gaussian (and even, when there are several, that they have the same variance).
- Non-parametric tests do not make such assumptions -- we say that they are **more "robust"** -- but, as a counter part, they are less powerful.

Examples of parametric and non-parametric tests.

Aim	Parametric tests	Non-parametric tests
compare two means	Student's T test	Wilcoxon's U test
compare more than two means	Anova (analysis of variance)	Kruskal--Wallis test
Compare two variances	Fisher's F test	Ansari-Bradley or Mood test
Comparing more than 2 variances	Bartlett test	Fligner test

see http://www.cookbook-r.com/Statistical_analysis/Homogeneity_of_variance/

Resistance

- A statistic (mean, median, variance, trimmed mean, etc.) is resistant if it does not depend much on extreme values.
- For instance, the **mean is not resistant**, while the **median is resistant**: a single extreme value can drastically change the mean, while it will not change the median.
- Methods that remain efficient even in presence of outliers ("robust methods") do exist: use them!

Breaking point

- The breaking point is the proportion of observations you can tamper with without being able to make the estimator arbitrarily "large".
- This is a measure of robustness of an estimator.
- For instance, the breaking point of the mean is zero: by changing a single observation, you can make the mean arbitrary large.

Breaking point

- On the other hand, the breaking point of the median is 50%: if you change a single observation, the median will change, but its value will be bounded by the rest of the cloud of points -- to make the median arbitrarily large, you would have to move (say) the top half of the data points.
- The trimmed mean has a breaking point somewhere in between.

Decision Theory

- Among all the possible tests, we want one for which the risks of type I and type II errors are as low as possible.
- Among the tests that can not be improved (i.e., we cannot modify them to get one with the same type I error risk and a lower type II error risk, and conversely), there is no means of choosing THE best.

Decision Theory

- We can plot those tests in the (type I error risk)x(type II error risk) plane: we get a curve.
- However, decision theory allows everyone to choose, among those tests, the one that becomes best one's taste for risk.
- One crude way of proceeding is to choose an upper bound on the type I error risk (alpha) and minimize the type II error risk.
- Simon's French book, "Decision Theory: an introduction to the mathematics of rationality", Ellis Horwood series in mathematics and its applications, Halsted Press, 1988.

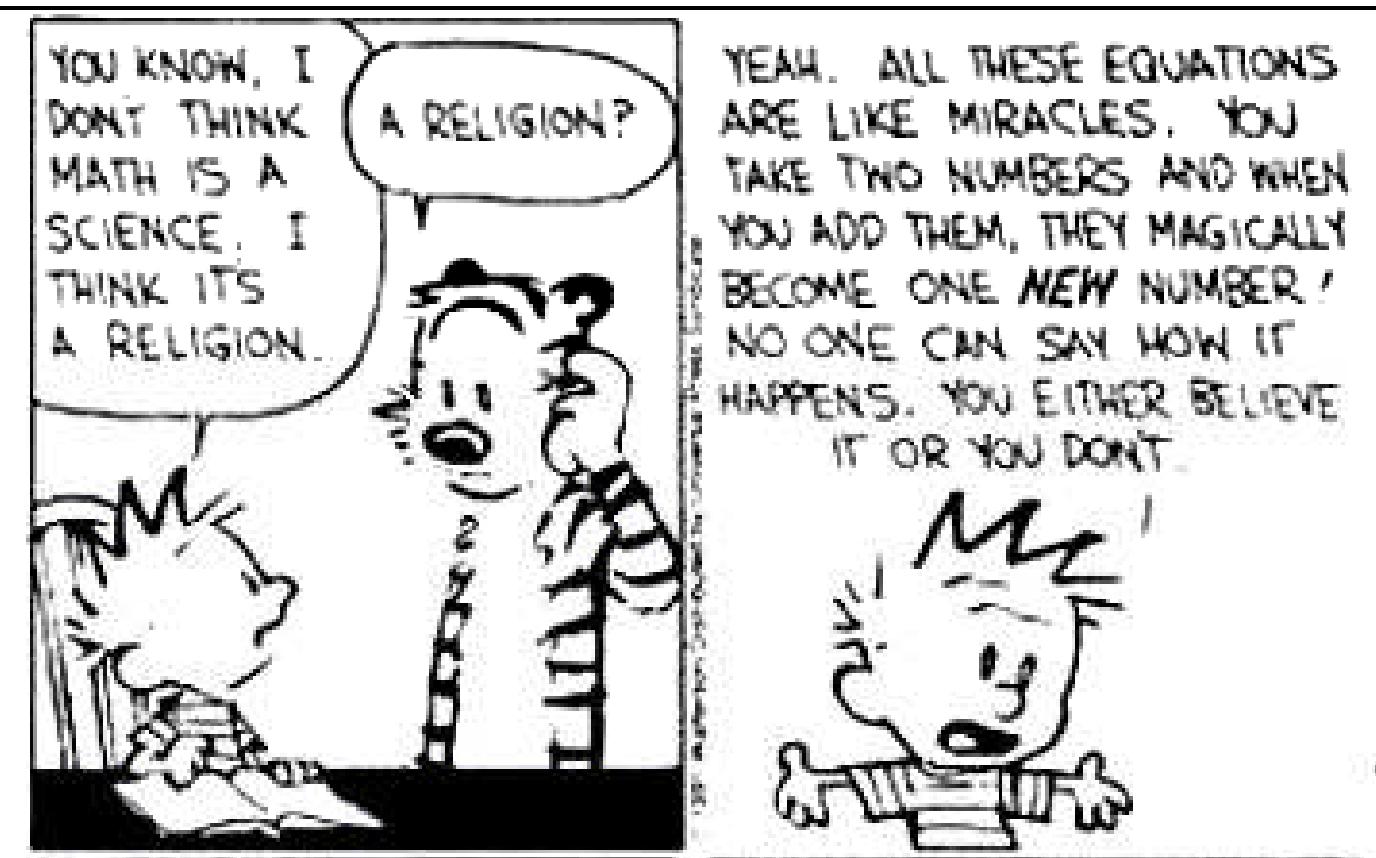
Robustness

- A (parametric) test is **robust** if its results are still valid when its assumptions are no longer satisfied (especially if the random variables studied are no longer gaussian).

References

- *Estimators and Statistical Tests* by Zoonekynd
- from http://zoonek2.free.fr/UNIX/48_R/08.html
URL blocked by Govt of India, use Google instead.
- http://www.cookbookr.com/Statistical_analysis/Homogeneity_of_variance/

Questions?



Graphics in R

data analysis and visualization

- 1. Dataset trees
- 2. Stem-leaf plot
- 3. Saving graphics
- 4. Reading data
- 5. Pie charts
- 6. Bar plots
- 7. Box plots
- 8. Histogram
- 9. Plot
- 10. Scatter plot
- 11. Abline
- 12. Multiple Graphs

From Katia Oleinik, koleinik@bu.edu, Scientific Computing and Visualization, Boston University <http://www.bu.edu/tech/research/training/tutorials/list>

Getting started

R comes along with some packages and data sets.

```
> # list all available libraries  
> library()  
  
> # load MASS package  
> library(MASS)  
  
> # list all datasets  
> data()  
  
> # load trees dataset into workspace  
> data(trees)
```

Exploring the data

First we need to explore the dataset. Often, it is too large to look at once.

```
> # view first few lines of the dataset.  
> head(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
```

Exploring the data

First we need to explore the dataset. Often, it is too large to look at once.

```
> # view first few lines of the dataset.  
> head(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  
  
> # get data dimensions:  
> dim(trees)  
[1] 31   3
```

Exploring the data

First we need to explore the dataset. Often, it is too large to look at it all at once.

```
> # column (variables) names:  
> names(trees)  
[1] "Girth"   "Height"  "Volume"
```

Exploring the data

```
> # column (variables) names:  
> names(trees)  
[1] "Girth"   "Height"  "Volume"  
> # description (help) for the dataset:  
> ?trees  
trees                  package:datasets          R  
Documentation  
  
Girth, Height and Volume for Black Cherry Trees  
  
Description:  
  
This data set provides measurements of the girth, height  
and  
volume of timber in 31 felled black cherry trees. Note  
that girth  
is the diameter of the tree (in inches) measured at 4 ft  
6 in  
above the ground.
```

Exploring the data

```
> # display an internal structure of an R object:  
> str(trees)  
'data.frame': 31 obs. of 3 variables:  
 $ Girth : num 8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...  
 $ Height: num 70 65 63 72 81 83 66 75 80 75 ...  
 $ Volume: num 10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9  
 ...
```

Exploring the data

Explore each variable in the dataset – its type, range, etc.

```
> # number of "observations"
> length(trees$Height)
[1] 31

> # type of variable
> mode(trees$Height)
[1] "numeric"

> # are there any missing data?
> length(trees$Height[is.na(trees$Height)])
[1] 0

> # explore some statistics
> mean(trees$Height)
[1] 76
```

Stemplot

Stem-and-leaf diagrams are very useful for quick analysis of small-size dataset.

```
> # find the name of stem-and-leaf utility
> apropos("stem")
[1] "R_system_version" "stem" "system" "system.file"
[5] "system.time" "system2"

> # get help for the function if necessary
> help("stem")
stem                  package:graphics      R Documentation
Stem-and-Leaf Plots
Usage:
  stem(x, scale = 1, width = 80, atom = 1e-08)
Arguments:
  x: a numeric vector.
  scale: This controls the plot length.
  width: The desired width of plot.
  atom: a tolerance.
```

Stemplot

The number on the left of the bar is the stem. The number on the right – the leaf.

```
> # stemplot for Girth variable
> stem(trees$Girth)
The decimal point is at the |

  8 | 368
 10 | 57800123447
 12 | 099378
 14 | 025
 16 | 03359
 18 | 00
 20 | 6
```

Stemplot

The number on the left of the bar is the stem. The number on the right – the leaf.

```
> # stemplot for Girth variable
> stem(trees$Volume)
The decimal point is 1 digit(s) to the right of the |

 1 | 00066899
 2 | 00111234567
 3 | 24568
 4 | 3
 5 | 12568
 6 |
 7 | 7
```

Stemplot

In some cases we might want to change the default settings of the function

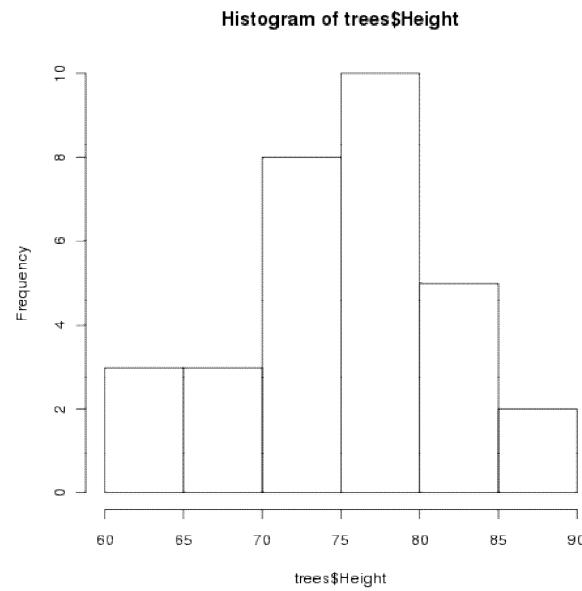
```
> # change the "scale" of the stemplot
> stem(trees$Volume, scale = .5)
The decimal point is 1 digit(s) to the right of the |

 0 | 00066899
 2 | 0011123456724568
 4 | 312568
 6 | 7
```

Graphics in R

R is famous for making powerful and very informative graphs. Lets first try to display something simple:

```
> # draw a simple histogram  
> hist(trees$Height)  
>
```



Displaying graphics

There might be a few choices for the output device.

```
> # list all available output devices
> dev.list()
X11cairo
2

> # check which device is current . If no device is active, returns 1 – a "null device"
> dev.cur()
X11cairo
2

> # switch between devices if necessary
> dev.set(2)
X11cairo
2
```

Saving graphics

Savings plots to a file. R supports a number of output formats, including
JPG, PNG, WMF, PDF, Postscript.

```
> # draw something to the screen
> hist(trees$Height)                      # plot
> dev.copy(png, "myHistogram.png")          # copy to device
> dev.off()                                # release the device
```

Saving graphics

Savings plots to a file. R supports a number of output formats, including BMP, JPG, PNG, WMF, PDF, TIFF, Postscript.

```
> # draw something to the screen  
> hist(trees$Height) # plot  
> dev.copy(png, "myHistogram.png") # copy to device  
> dev.off() # release the device
```

If we do not want to output anything to the screen, but would rather print it to the file, we can use another approach:

```
> png("myHistogram.png") # specify the device and format  
> hist(trees$Height) # plot to the device  
> dev.off() # release the device
```

Saving graphics

On katana you can view graphics files with

display file_name

or

gimp file_name

```
% display myHistogram.png
```

```
% gimp (myHistogram.png)
```

Reading in a dataset

Let's read in some data file and work with it.

```
> # read in the data from a spreadsheet:  
> pop <- read.csv("population.csv")  
> head(pop)  
Education South Sex Experience Union Wage Age Race Occupation Sector Married  
1 8 0 1 21 0 5.10 35 2 6 1 1  
2 9 0 1 42 0 4.95 57 3 6 1 1  
3 12 0 0 1 0 6.67 19 3 6 1 0  
4 12 0 0 4 0 4.00 22 3 6 0 0  
5 12 0 0 17 0 7.50 35 3 6 0 1  
6 13 0 0 9 1 13.07 28 3 6 0 0
```

Reading in a dataset

Education: Number of years of education.

South: 1=Person lives in South, 0=Person lives elsewhere.

Sex: 1=Female, 0=Male.

Experience: Number of years of work experience.

Union: 1=Union member, 0=Not union member.

Wage: dollars per hour.

Age: years.

Race: 1=Other, 2=Hispanic, 3=White, 4=African American, 5=Asian

Occupation: 1=Management, 2=Sales, 3=Clerical, 4=Service, 5=Professional, 6=Other.

Sector: 0=Other, 1=Manufacturing, 2=Construction.

Marriage: 0=Unmarried, 1=Married.

categorical variables

Sometimes we would like to reformat the input data. In our example it would be nice to have a word description for the "race" variable instead of numerical number.

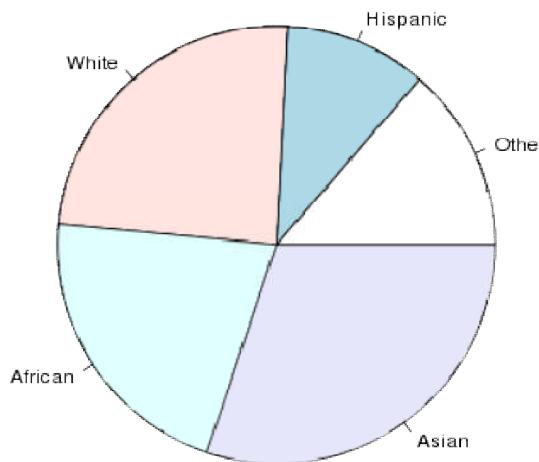
```
> # add a description to race variable
> pop$Race <- factor( pop$Race,
+   labels=c("Other","Hispanic","White", "African", "Asian"))
> table(pop$Race)

  Other Hispanic     White    African      Asian
    73        56       130       114       161
```

Pie charts

Statisticians generally regard pie charts as a poor method of displaying information, and they are uncommon in scientific literature.

```
> # draw a default pie chart  
> pie(table(pop$Race))
```



Pie charts

There are a few things that we might want to improve in this graph.

```
> # what parameters are available  
> ?pie  
Usage:  
  
 pie(x,  
       labels = names(x),  
       edges = 200, radius = 0.8,  
       clockwise = FALSE,  
       init.angle = if(clockwise) 90 else 0,  
       density = NULL, angle = 45,  
       col = NULL, border = NULL,  
       lty = NULL, main = NULL, ...)  
  
> # view a few examples  
> example(pie)
```

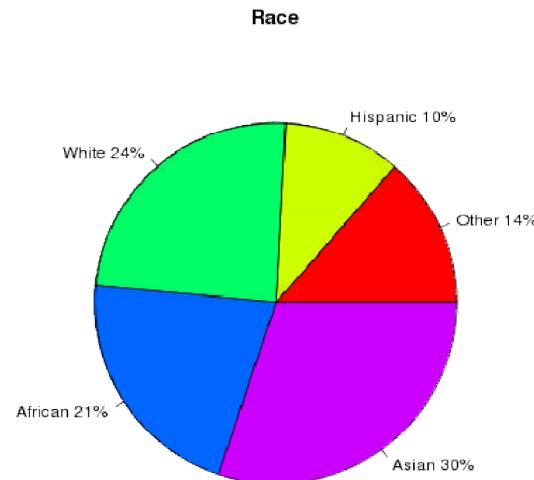
Pie charts

So lets give a title to the chart, change a color scheme and some labels

```
> # calculate percentage of each category  
> pct<-round(table(pop$Race)/sum(table(pop$Race))*100)  
  
> # make labels  
> lbls<-levels(pop$Race)  
  
> # add percentage value to the label  
> lbls<-paste(lbls,pct)  
  
> # add percentage sign to the label  
> lbls<-paste(lbls, "%", sep="")
```

Pie charts

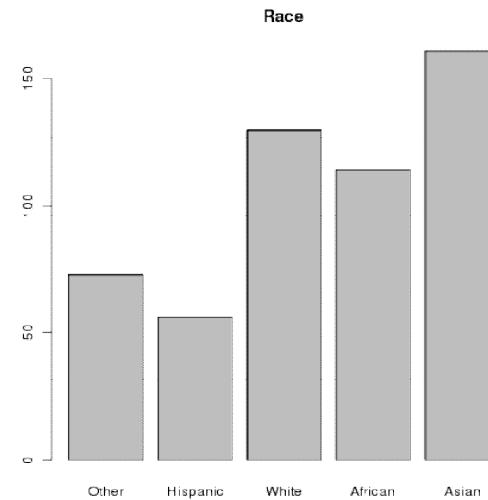
```
> # draw enhanced pie chart  
> pie(table(pop$Race),  
+      labels=lbls,  
+      col=rainbow(length(lbls)),  
+      main="Race")
```



Bar plots

Statisticians prefer bar plots over pie charts. For a human eye, it is much easier to compare heights than volumes.

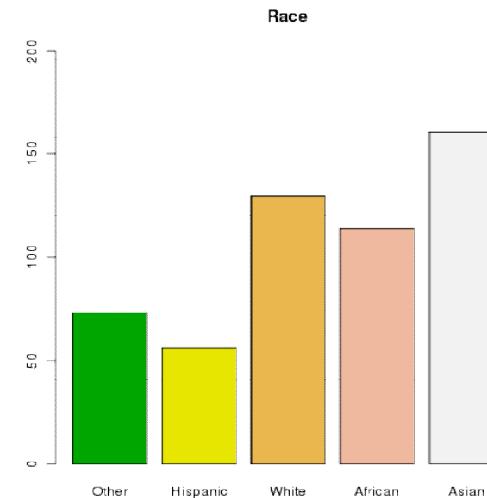
```
> # draw a simple barplot  
> barplot(table(pop$Race),  
+           main="Race")
```



Bar plots

We can further improve the graph, using optional variables.

```
> # draw barplot  
> b<-barplot(table(pop$Race),  
+             main="Race",  
+             ylim = c(0,200),  
+             col=terrain.colors(5))
```

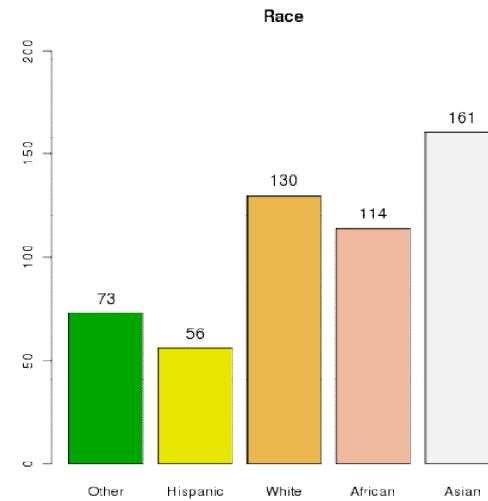


Bar plots

Some extra annotation can help make this plot easier to read.

```
> # draw barplot
> b<-barplot(table(pop$Race),
+             main="Race",
+             ylim = c(0,200),
+             col=terrain.colors(5))

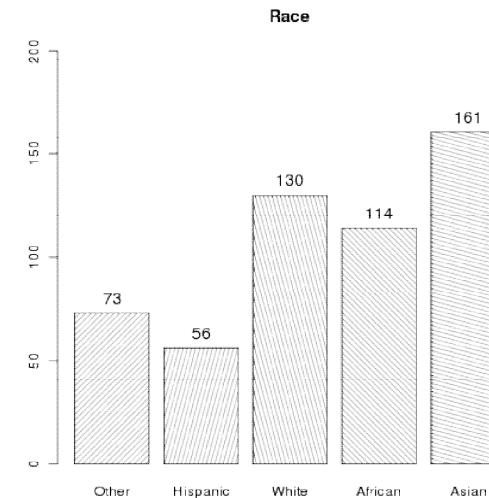
> # add text to the plot
> text(x = b,
+       y=table(pop$Race),
+       labels=table(pop$Race),
+       pos=3,
+       col="black",
+       cex=1.25)
>
```



Bar plots

For a graph in a publication it is better to use patterns or shades of grey instead of color.

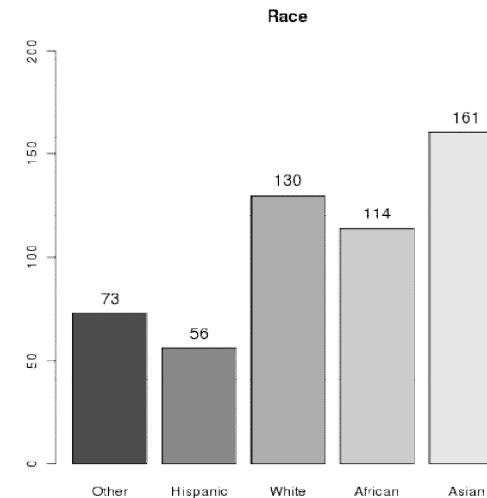
```
> # draw barplot  
> b<-barplot(table(pop$Race),  
+             main="Race",  
+             ylim = c(0,200),  
+             angle = 15 + 30*1:5,  
+             density = 20)
```



Bar plots

For a graph in a publication it is better to use patterns or shades of grey instead of color.

```
> # draw barplot  
> b<-barplot(table(pop$Race),  
+           main="Race",  
+           ylim = c(0,200),  
+           col=gray.colors(5))  
>
```



Bar plots

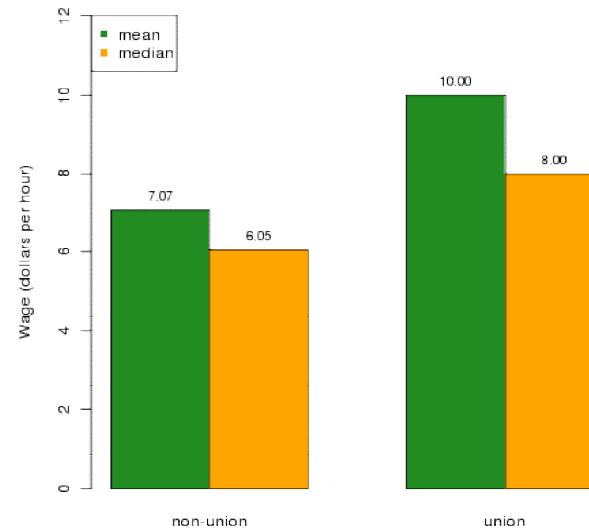
If we would like to display bar plots side-by-side

```
> # define 2 vectors
> m1 <- tapply(pop$Wage,pop$Union, mean)
> m2 <- tapply(pop$Wage,pop$Union, median)
> r  <- rbind(m1,m2)    # combine vectors by rows

> # draw the plot
> b<-barplot(r,
              col=c("forestgreen","orange"),
              ylim=c(0,12),
              beside=T,
              ylab="Wage (dollars per hour)",
              names.arg=c("non-union","union"))
```

Bar plots

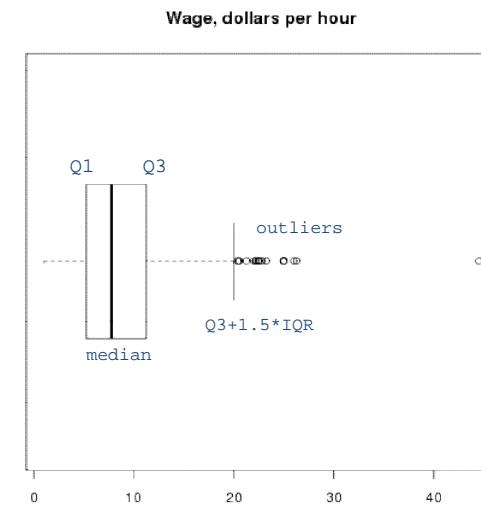
```
> # add a legend  
> legend("topleft",  
        c("mean","median"),  
        col=c("forestgreen","orange"),  
        pch=15) # use "square" symbol for the legend  
  
> # add text  
> text(b,  
      y = r,  
      labels=format(r,4),  
      pos=3,    # above of the spec. coordinates  
      cex=.75) # character size
```



Boxplots

Boxplots are a convenient way of graphically depicting groups of numerical data through their five-number summaries.

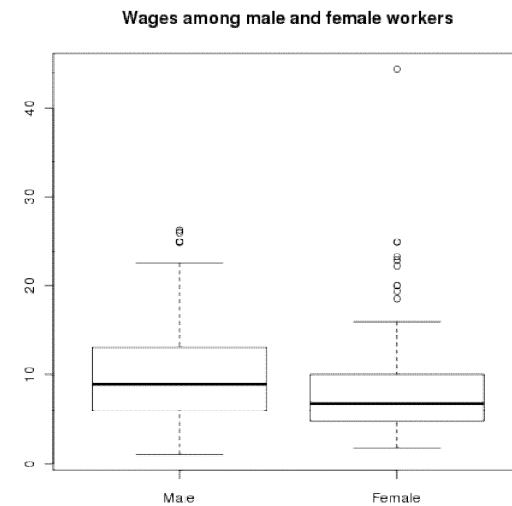
```
> # plot boxplot  
> boxplot(pop$Wage,  
main="Wage, dollars per hour",  
horizontal=TRUE)
```



Boxplots

To compare two subsets of the variables we can display boxplots side by side.

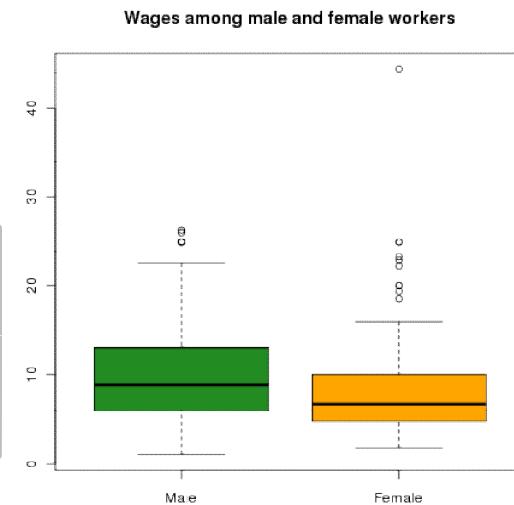
```
> # compare wages of male and female groups  
> boxplot(pop$Wage~pop$Sex  
+ main="Wages among male and female workers")
```



Boxplots

To compare two subsets of the variables we can display boxplots side by side.

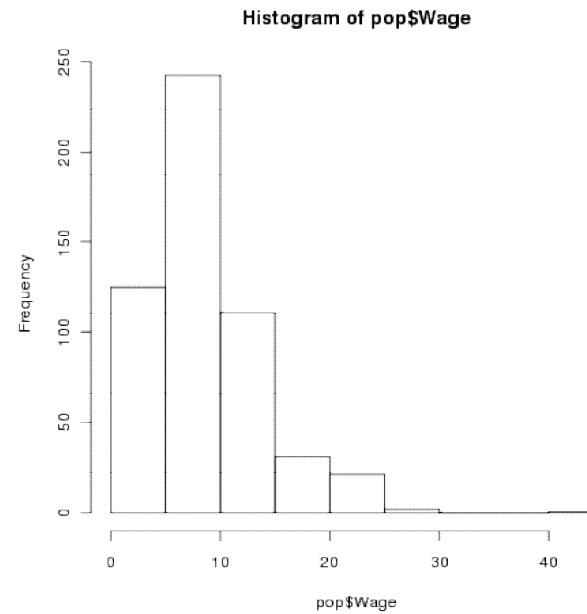
```
> # compare wages of male and female groups  
> boxplot(pop$Wage~pop$Sex  
+ main="Wages among male and female workers",  
+ col=c("forestgreen", "orange") )  
>
```



Histograms

A **histogram** is a graphical representation of the distribution of continuous variable .

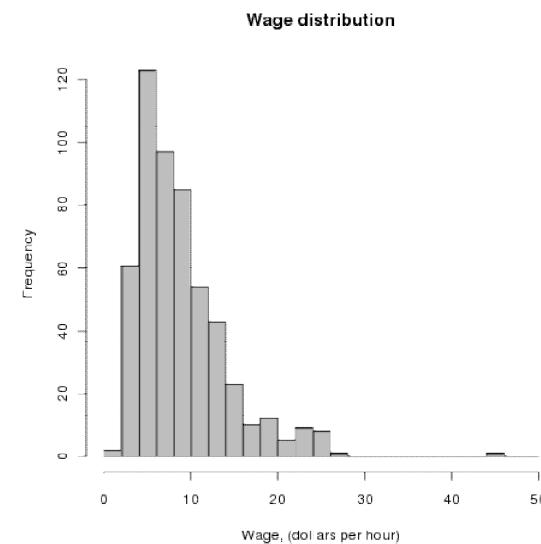
```
> # draw a default histogram  
> hist(pop$Wage)
```



Histograms

We can enhance this histogram from its plain default appearance.

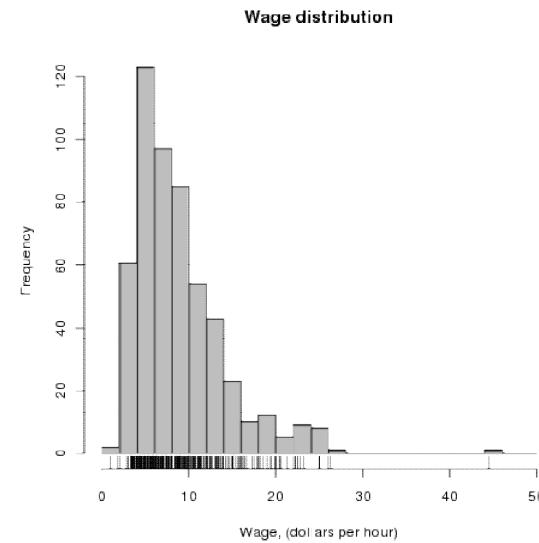
```
> # draw an enhanced histogram  
> hist(pop$Wage,  
+       col = "grey",  
+       border = "black",  
+       main = "Wage distribution",  
+       xlab = "Wage, (dollars per hour)",  
+       breaks = seq(0,50,by=2) )
```



Histograms

rug() function is an example of a graphics function that adds to an existing plot.

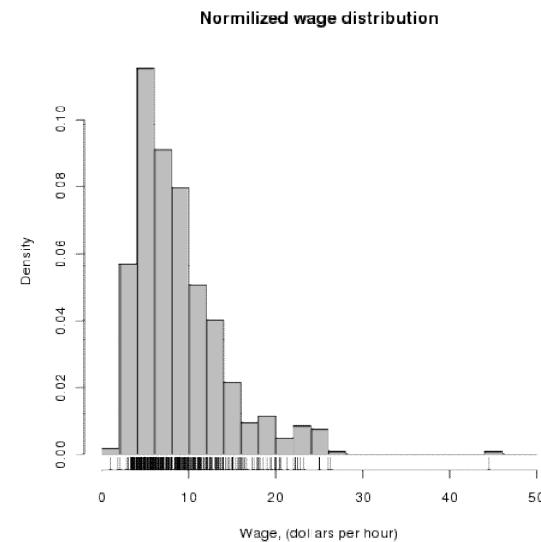
```
> # add actual observations to the plot  
> rug(pop$Wage)
```



Histograms

A histogram may also be normalized displaying probability density:

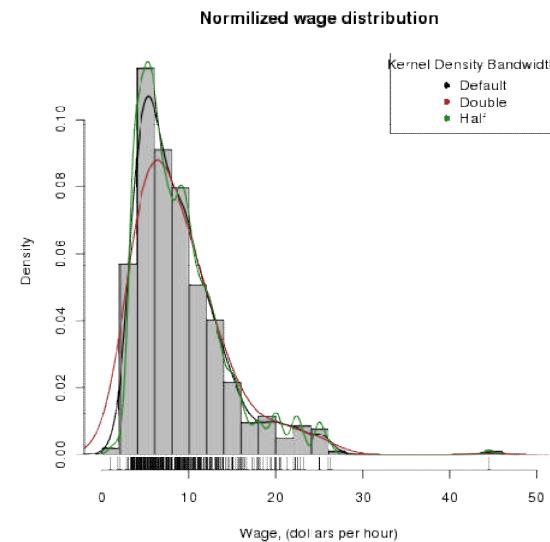
```
> # draw an enhanced histogram  
> hist(pop$Wage,  
+       col = "grey",  
+       border = "black",  
+       main = "Wage distribution",  
+       xlab = "Wage, (dollars per hour)",  
+       breaks = seq(0,50,by=2),  
+       freq = F )  
  
> rug(pop$Wage)
```



Histograms

We can now add some lines that display a kernel density and a legend:

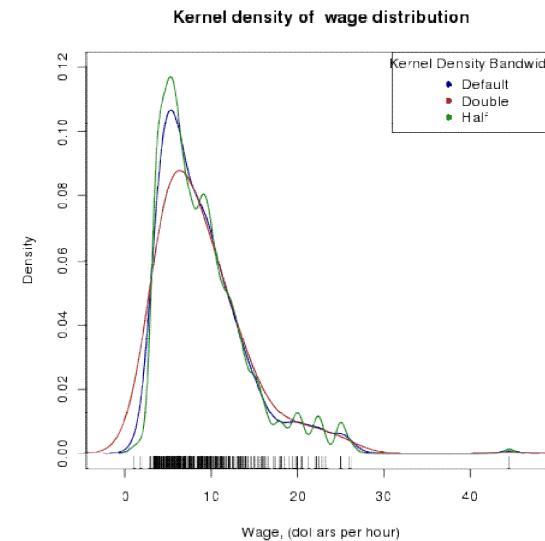
```
> # add kernel density lines  
> lines(density(pop$Wage), lwd=1.5)  
> lines(density(pop$Wage, adj=2),  
       lwd=1.5, # line width  
       col = "brown")  
> lines(density(pop$Wage, adj=0.5)),  
       lwd=1.5,  
       col = "forestgreen"))  
  
> legend("topright",  
       c("Default", "Double", "Half"),  
       col=c("black", "brown",  
             "forestgreen"),  
       pch = 16,  
       title="Kernel Density Bandwidth")
```



Plot

If we do not want to display the histogram now, but show only the lines, we have to call **plot()** first, since **lines()** function only adds to the existing plot.

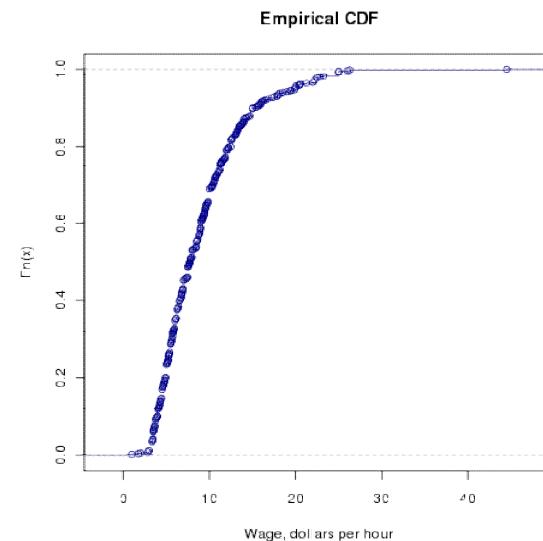
```
> # plot kernel density lines only  
> plot(density(pop$Wage),  
       col="darkblue",  
       main="Kernel ...",  
       xlab="Wage ...",  
       lwd=1.5,  
       ylim=c(0,0.12) )  
  
> # draw the other two lines, the rug plot and the legend  
> lines(...)
```



Plot

Similarly, we can display the empirical cumulative distribution.

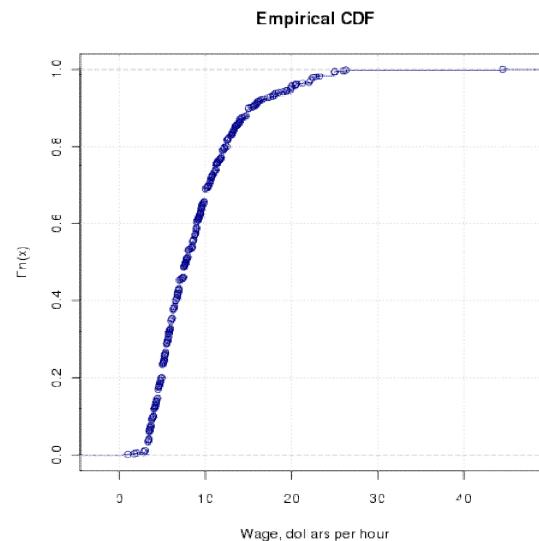
```
> # plot ecdf
> plot(ecdf(pop$Wage),
       pch=1,
       col="darkblue",
       xlab="Wage, dollars per hour",
       main="Empirical CDF")
```



Plot

Adding a grid will improve readability of the graph.

```
> # add grid  
> grid()
```



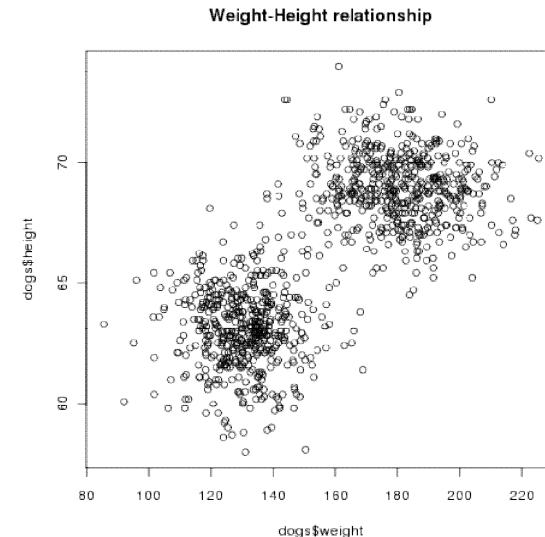
Scatterplots

Lets examine a relationship of 2 variables in **dogs** dataframe.

```
> # add grid  
> plot(dogs$height~dogs$weight,  
       main="Weight-Height relationship")
```

We can definitely improve this graph:

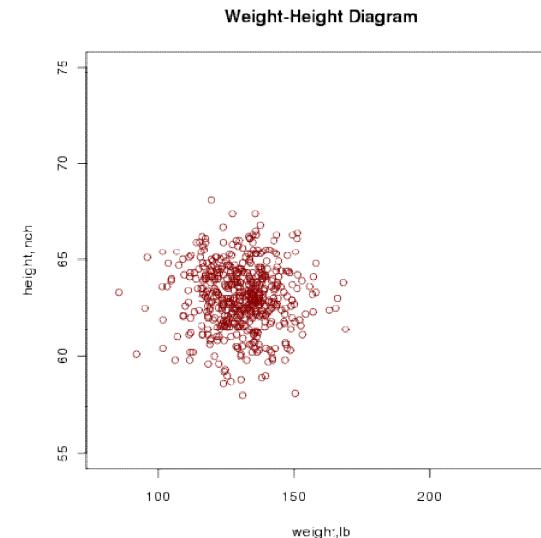
- change axes names
- change y-axis labels
- emphasize 2 clusters
- Legend
- add a grid
- Add some statistical analysis



Scatterplots

2 clusters can be emphasized using different colors and symbols.

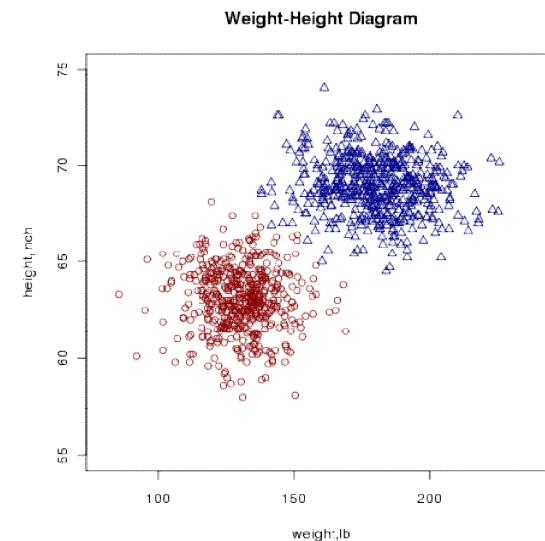
```
> # draw scatterplot  
> plot(dogs$height[dogs$sex=="F"] ~  
+       dogs$weight[dogs$sex=="F"] ,  
+       col="darkred",  
+       pch=1,  
+       main="Weight-Height Diagram",  
+       xlim=c(80,240),  
+       ylim=c(55,75),  
+       xlab="weight, lb",  
+       ylab="height, inch")  
>
```



Scatterplots

2 clusters can be emphasized using different colors and symbols.

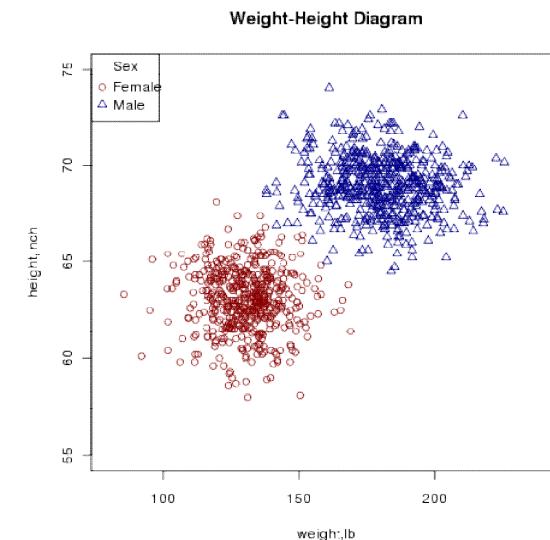
```
> # add points  
> points(dogs$height [dogs$sex=="M"] ~  
+          dogs$weight [dogs$sex=="M"] ,  
+          col="darkblue",  
+          pch=2)  
>
```



Scatterplots

2 clusters can be emphasized using different colors and symbols.

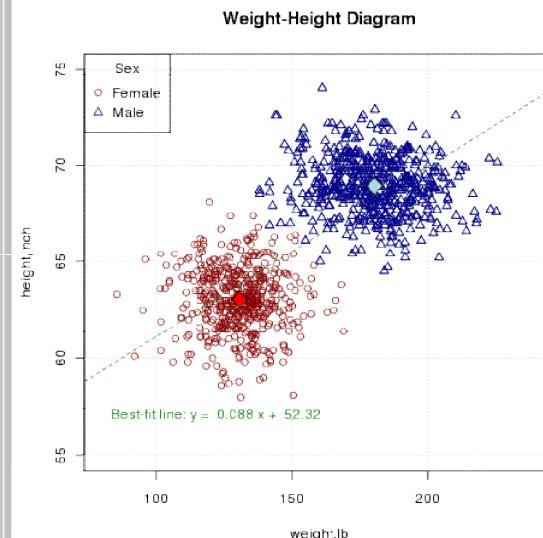
```
> # add legend  
> legend("topleft",  
+        c("Female", "Male"),  
+        col=c("darkred", "darkblue"),  
+        pch=c(1,2),  
+        title="Sex")  
>
```



abline

Add grid and best-fit line.

```
> # add best-fit line
> abline(lm(dogs$height~dogs$weight),
+         col="forestgreen",
+         lty=2)
> # add text
> text(80, 59, "Best-fit line: ...",
+       col="forestgreen",
+       pos=4)
> # plot mean point for the cluster of female dogs
> points(mean(dogs$weight[dogs$sex=="F"]),
+         mean(dogs$height[dogs$sex=="F"]),
+         col="black",
+         bg="red",
+         pch=23)
> # plot mean point for the cluster of male dogs
> points(mean(dogs$weight[dogs$sex=="M"]),
+         mean(dogs$height[dogs$sex=="M"]),
+         col="black",
+         bg="lightblue",
+         pch=23)
> # add grid
> grid()
```



abline

With abline() function we can easily add vertical and horizontal lines to the graph.

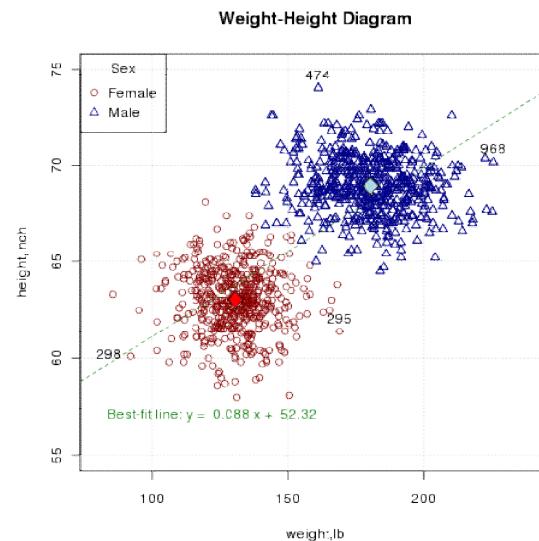
```
> # add horizontal line  
> abline( h = 150 )  
>  
> # add vertical line  
> abline( v = 65 )
```

interaction

R allows for *interaction*.

Left-click with the mouse on points to identify them,
Right-click to exit.

```
> # pick points  
> pts<- identify(dogs$weight,  
+                  dogs$height)  
>
```



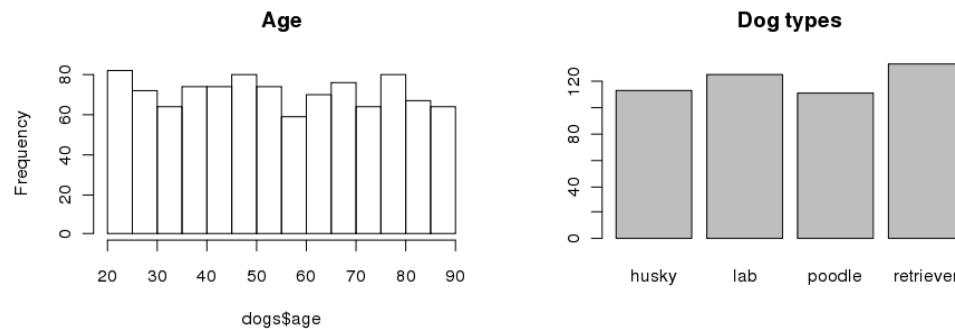
Multiple graphs

Often we need to place a few graphs together. Function **par()** allows to combine several graphs into one table.

```
> # specify number of rows and columns in the table:  
> # 3 rows and 2 columns  
> par( mfrow = c(3,2) )  
>
```

Multiple graphs

Often we need to place a few graphs together. Function **par()** allows to combine several graphs into one table.

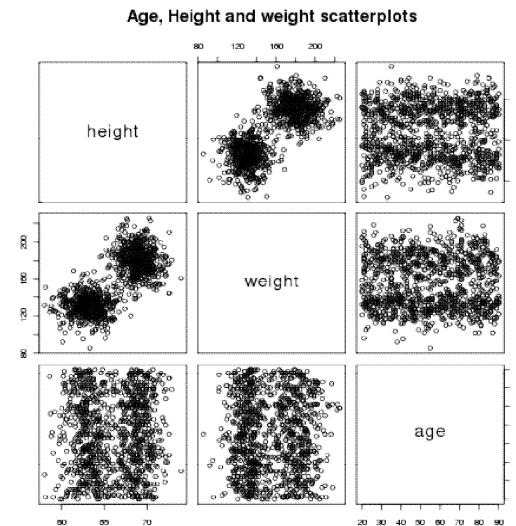


```
> # specify number of rows and columns in the table  
> par( mfrow = c(1,2) )  
> hist(dogs$age, main="Age" )  
> barplot(table( dogs$dog_type), main="Dog types" )  
> # return to normal display  
> par( mfrow = c(1,1) )
```

Matrix of scatterplots

To make scatterplots of all numeric variables in a dataset, use **pairs()**

```
> # matrix of scatterplots  
> pairs(~height+weight+age,  
+ data = dogs,  
+ main="Age, Height and weight scatterplots" )  
>
```



Hands On R Data

4/2/2016.

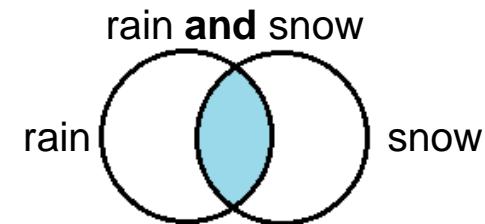
From Lecture 2, Working with data in R, by Trevor A. Branch,
Course: FISH 552 Introduction to R

Subsetting vectors

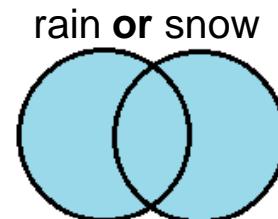
```
> x <- c(3,4,2,1,10,7)
> x[1] ←
[1] 3
> x[3]
[1] 2
> x[1:5] ←
[1] 3 4 2 1 10
Use a vector of indices to select
multiple items
> x[c(2,5)]
[1] 4 10
> x[-c(2,4)] ←
[1] 3 10 7
A negative index means exclude the items at
those index values, here exclude items 2 and 4
```

Boolean logic (T or F)

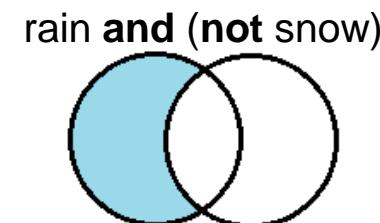
I carry an umbrella if it both rains
and snows on the same day



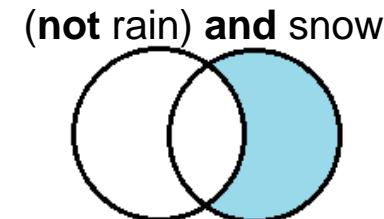
I carry an umbrella whenever it rains
or snows



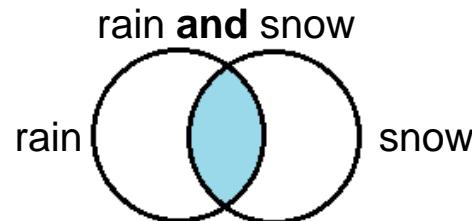
I carry an umbrella for rain but never
for snow



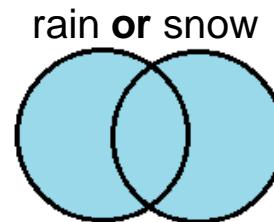
I never carry an umbrella for rain,
only for snow



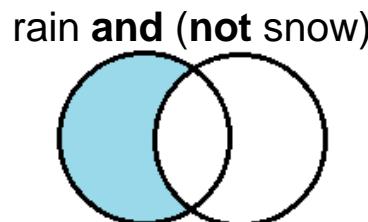
rain & snow



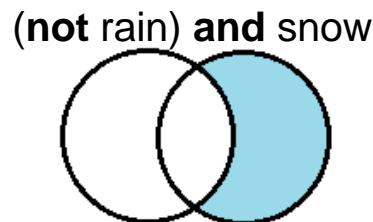
rain | snow



rain & !snow



!rain & snow



Boolean operators

- & and (element wise)
- | or (element wise)
- ! not
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equal to
- != not equal to

Boolean operators

`&&` and (first element of vector only)

`||` or (first element of vector only)

The ONLY place you use these is when you are asking **IF** something is true, in which case you need a single value and not an entire vector of T and F values

Boolean examples: single value

```
> x <- 3
```

```
> x == 3
```

```
[1] TRUE
```

```
> x < 10
```

```
[1] TRUE
```

```
> x < -1
```

```
[1] FALSE
```

```
> x > 0 & x < 10
```

```
[1] TRUE
```

Combine multiple conditions with AND (&) or OR

(|)

This is how you ask whether x is between 0
and 10

Boolean examples: vector of values

```
> x <- 1:5 Now x is a vector of values
```

```
> x == 3
```

```
[1] FALSE FALSE TRUE FALSE FALSE
```

```
> x < 10
```

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

```
> x > 2 & x <= 4
```

```
[1] FALSE FALSE TRUE TRUE FALSE
```

```
> x != 2
```

```
[1] TRUE FALSE TRUE TRUE TRUE
```

Umbrella logic

```
> day <- c("Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat")
> rain <- c("Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "No")
> snow <- c("No", "No", "No", "Yes", "No", "No", "No")
> rain == "Yes"
[1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE
> rain != "No"
[1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE
> snow == "Yes"
[1] FALSE FALSE FALSE TRUE FALSE FALSE FALSE
> rain=="Yes" & snow=="Yes"
[1] FALSE FALSE FALSE TRUE FALSE FALSE FALSE
> rain=="Yes" | snow=="Yes"
[1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE
```

Umbrella logic

Vectors can be subsetted according to logic

```
> day[rain=="Yes"]
[1] "Sun" "Mon" "Tues" "Wed" "Thurs" "Fri"
> day[snow=="Yes"]
[1] "wed"
```

I always carry an umbrella if it rains **and** snows

```
> day[rain=="Yes" & snow=="Yes"]
[1] "wed"
```

I always carry an umbrella if it rains **or** snows

```
> day[rain=="Yes" | snow=="Yes"]
[1] "Sun" "Mon" "Tues" "Wed" "Thurs" "Fri"
```

TRUE and FALSE

```
> rain <- c("Yes","Yes","Yes",
  "Yes","Yes","Yes","No")
```

How many days did it rain this week?

```
> sum(rain=="Yes")
[1] 6
```

Internal representation of TRUE and FALSE

```
> as.numeric(rain=="Yes")
[1] 1 1 1 1 1 1 0
```

TRUE == T == 1 and FALSE == F == 0
(long-standing programming convention)

Pro tip: In R, F is equivalent to FALSE and T is equivalent to TRUE. Most code uses T and F.

Other Boolean operators

```
> rain <- c("Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "No")
```

Which elements are TRUE?

```
> which(rain=="Yes")  
[1] 1 2 3 4 5 6
```

Are **any** elements true?

```
> any(rain=="Yes")  
[1] TRUE
```

Are **all** elements true?

```
> all(rain=="Yes")  
[1] FALSE
```

Hands-on exercise 1

```
y <- c(3,2,15,-1,22,1,9,17,5)
```

Complete the following using the vector y

1. Display the first and last values
2. Find the last value for a vector of any length
3. Display the values that are greater than the mean of y
4. Display the positions (indices) of the values greater than the mean
5. Are all the values positive?
6. Are any of the values equal to the mean?
7. Are any of the values equal to the median?

Data frames

- It is convenient to store data as a collection of variables

```
> nislnds <- length(islands)
> years <- seq(from=2013, length=nislnds)
> island.data <- data.frame(years, islands)
```

- `head()` is a quick way to view the first part of a data frame

```
> head(island.data)
```

	years	islands
Africa	2013	11506
Antarctica	2014	5500
Asia	2015	16988
Australia	2016	2968
Axel Heiberg	2017	16
Baffin	2018	184

Data frames

- Extract the names of a data frame

```
> names(island.data) [1] "years" "islands"
```

- Modify the names of a data frame

```
> names(island.data) <- c("years", "area")
```

```
> head(island.data, n=2)
```

	years	area
Africa	2013	11506
Antarctica	2014	5500

- Assign column names when creating a data frame

```
> island.data <- data.frame(years=years, area=islands)
```

Data frames

What is stored in your working directory?

```
> area  
Error: object 'area' not found  
> ls()  
[1] "island.data" "nislands" "years"  
> island.data$area  
[1] 11506 5500 16988 2968 16 184 23 280  
[9] 84 73 25 43 21 82 3745 840  
[17] 13 30 30 89 40 33 49 14  
[25] 42 227 16 36 29 15 306 44  
[33] 58 43 9390 32 13 29 6795 16  
[41] 15 183 14 26 19 13 12 82
```

Use the \$ operator to extract from data
frames

Extracting data from data frames

```
> tag <- c(2, 3, 5, 7, 8, 9, 15, 21, 23, 26)
> weight <- c(14.8, 21, 19.7, 23.2, 16, 16.1, 20,
29.3, 17.8, 21.2)
> condition <- c("good", "fair", "fair", "poor",
"fair", "good", "good", "fair", "fair", "poor")

> fishData <- data.frame(tag, weight, condition)

> head(fishData, n=2)
  tag weight condition
1   2    14.8      good
2   3    21.0      fair
```

Extracting columns by name

- Extract the column with the name `weight`

```
> fishData$weight
```

```
[1] 14.8 21.0 19.7 23.2 16.0 16.1 20.0 29.3 17.8  
21.2
```

- Note that changing the `weight` vector will
not change `fishData$weight`

```
> (weight <- rep(20,10))
```

```
[1] 20 20 20 20 20 20 20 20 20 20
```

```
> fishData$weight
```

```
[1] 14.8 21.0 19.7 23.2 16.0 16.1 20.0 29.3 17.8  
21.2
```

Extracting rows/columns by indices

- Specify the row index, column index or both
object[row, column]

- Extract column 2

```
> fishData[,2]
```

```
[1] 14.8 21.0 19.7 23.2 16.0 16.1 20.0 29.3 17.8  
21.2
```

- Exclude column 1, retain columns 2-3

```
> fishData[,-1]
```

weight condition

```
1 14.8 good
```

```
2 21.0 fair
```

```
3 19.7 fair
```

...

Extracting elements

```
> fishData[1,] Extract the first row  
  tag weight condition  
1   2    14.8      good  
> fishData[c(1,4),] Extract rows 1 and 4  
  tag weight condition  
1   2    14.8      good  
4   7    23.2      poor  
> fishData[1,2] Access element in row 1 and col 2  
[1] 14.8  
> fishData$weight[1]  
[1] 14.8
```

First get the weight column, then
find the first element of the resulting
vector

Methods for column extraction

```
> fishData[,2:3]  Extract columns 2 and 3  
    weight condition  
1   14.8      good  
2   21.0      fair  
3   19.7      fair  
...  
> fishData[,c("tag", "condition")]  
    tag condition  Extract columns by name (useful  
1   2      good  for big data frames where the  
2   3      fair  column indices are hard to find)  
3   5      fair  
...
```

Extracting elements logically

```
> fishData$weight Vector of weights  
[1] 14.8 21.0 19.7 23.2 16.0 16.1 20.0 29.3  
17.8 21.2  
> fishData$weight > 22 Vector of TRUE or FALSE  
[1] FALSE FALSE FALSE TRUE FALSE FALSE  
FALSE TRUE FALSE FALSE  
> fishData[fishData$weight > 22,]  
tag weight condition  
4    7    23.2    poor  
8   21    29.3    fair
```

Extract only the rows where the vector elements are TRUE, i.e. where weight > 22

Referencing the data frame TWICE is a key method for finding rows and columns

Combining conditions

```
> fishData[fishData$weight < 20 &  
           fishData$condition == "fair",]  
    tag weight condition  
 3   5    19.7      fair  
 5   8    16.0      fair  
 9  23    17.8      fair  
> fishData[fishData$weight < 15 |  
           fishData$weight > 25,]  
    tag weight condition  
 1   2    14.8     good  
 8  21    29.3      fair
```

Umbrella logic revisited

I always carry an umbrella if it rains **and** snows (vectors)

```
> day[rain=="Yes" & snow=="Yes"]  
[1] "wed"
```

Answered using a data frame

```
> weather <- data.frame(day, rain, snow)  
> weather[weather$rain=="Yes" &  
           weather$snow=="Yes",]  
    day rain snow  
4   wed  Yes  Yes
```

Dimensions of data frames

- `length` gives the number of elements in a vector
- For a data frame, `length` gives the number of columns

```
> length(fishData)
```

```
[1] 3
```

- Use `dim` for both rows and columns

```
> dim(fishData)
```

```
[1] 10 3
```

- Use `nrow` or `ncol` to get each individually

```
> nrow(fishData)
```

```
[1] 10
```

```
> ncol(fishData)
```

```
[1] 3
```

Sample data frame

Copy and paste this into your R code for the hands-on exercise

```
> patients <- data.frame(  
  id = c(31, 62, 50, 99, 53, 75, 54, 58, 4, 74),  
  age = c(12, 18, 20, 17, 14, 8, 12, 24, 24, 21),  
  sex = c("M", "F", "F", "M", "F", "M", "M", "F",  
"F", "M") )  
  
> head(patients, n=2)  
id age sex  
1 31 12 M  
2 62 18 F
```

Hands-on exercise 2 using patients

- Use a logical operator to display ages that are larger than 20
- Do the same as above but also display the corresponding id and sex
- Display only female observations
- Change the 7th age in patients from 12 to 21
- Calculate the proportion of subjects that are age 20 or greater
- Calculate the proportion of males that are greater than 20
- Permanently delete the 10th subject
- Permanently add two more subjects to this data frame (use rbind)

Missing values (NA)

```
> humidity <- c(63.33, NA, 64.63, 68.38,  
NA← 79.1, 77.46)                                  NA = not  
                                                          available
```

Many functions do not handle missing values by default

```
> mean(humidity)  
[1] NA  
> mean(humidity, na.rm=T)                        remove NAs  
                                                          before calculating  
[1] 70.58                                            mean
```

Missing values (NA)

- Omit missing values

```
> na.omit(humidity)
[1] 63.33 64.63 68.38 79.10 77.46
attr(,"na.action")
[1] 2 5
attr(,"class")
[1] "omit"
```

- Also see `na.pass()`, `na.fail()`, `na.exclude()`
- `!is.na()` is a slick way to handle missing values in vectors

```
> humidity[!is.na(humidity)]
[1] 63.33 64.63 68.38 79.10 77.46
```

More Hands On Data, in R: List, Matrix, Factors, Excel Files

From Lecture 3, Working with data in R II,
Trevor A. Branch, FISH 552 Introduction to R

Matrices

- Data frames: store objects of different types
- Matrices: store objects all of the same type

```
> x <- matrix(data=1:6, Numbers to put into matrix  
                nrow=3, ncol=2, Number of rows, number of columns  
                byrow=FALSE) Fill numbers in by column  
  
> x  
      [,1] [,2]  
[1,]    1    4  
[2,]    2    5  
[3,]    3    6
```

Number of elements provided
should be `nrow × ncol`

Creating matrices

- Matrix from the years visiting each island

```
> nislands <- length(islands) # 48
> years <- seq(2013, length.out=nislands)
> isl.mat <- matrix(c(years, islands), ncol=2, nrow=nislands)
> head(isl.mat, n=2)
      [,1] [,2]
[1,] 2013 11506
[2,] 2014  5500
```

- Fast way to create matrix using column bind (cbind)

```
> isl.mat <- cbind(years, islands)
```

```
> head(isl.mat, n=2)
```

	years	islands
Africa	2013	11506
Antarctica	2014	5500

Matrix functions

- Matrix formation by row binding

```
> isl.row.mat <- rbind(years, islands)
> head(isl.row.mat)
           Africa Antarctica Asia Australia...
years      2013        2014  2015    2016...
islands   11506        5500 16988   2968...
```

- Use `t()` to transpose (switch columns and rows)

```
> t(isl.row.mat)
           years  islands
Africa       2013    11506
Antarctica  2014     5500
...
...
```

Matrix dimensions

- Use the same functions to extract dimensions that were used for data frames

```
> dim(isl.mat)
[1] 48 2
> dim(isl.row.mat)
[1] 2 48
> nrow(isl.mat)
[1] 48
> ncol(isl.mat)
[1] 2
```

Arrays

- Matrices have 2 dimensions, arrays have N dimensions

```
> isl.array <- array(data=c(years, islands),  
                      dim=c(nislands,2))  
> head(isl.array, n=3)  
      [,1]  [,2]  
[1,] 2013 11506  
[2,] 2014  5500  
[3,] 2015 16988
```

Create an array of
dimension $n_{islands} \times 2$

- A matrix is a special case of an array, but a data frame is not

```
> is.array(isl.array)  
[1] TRUE  
> is.array(isl.mat)  
[1] TRUE
```

3-Dimensional array

- Keep adding dimensions to the `dim` argument

```
> array(1:24, dim=c(3,4,2))  
, , 1  
    [,1] [,2] [,3] [,4]  
[1,]    1    4    7   10  
[2,]    2    5    8   11  
[3,]    3    6    9   12
```

2 of the third dimension
4 columns
3 rows

```
, , 2  
    [,1] [,2] [,3] [,4]  
[1,]  13   16   19   22  
[2,]  14   17   20   23  
[3,]  15   18   21   24
```

Lists

1. Most flexible data structure
2. Each element can be varying length and mode

```
> description <- "Year of visit, island area (thousand  
sq miles)"  
> isl.list <- list(meta=description, nislands=nislands,  
data=isl.mat)  
> isl.list  
$meta  
[1] "Year of visit, island area (thousand sq miles)"  
$nislands  
[1] 48  
$data  
      years islands  
Africa      2013    11506  
Antarctica 2014     5500
```

Extracting elements from lists

- The \$ operator works for lists

```
> isl.list <- list(meta=description,  
nislands=nislands, data=isl.mat)
```

```
> isl.list$meta  
[1] "Year of visit, island area (thousand sq miles)"  
> isl.list$nislands  
[1] 48  
> isl.list$data[1:3,]  
          years islands  
Africa      2013    11506  
Antarctica 2014     5500  
Asia        2015    16988
```

Converting data frames to lists

```
> patients <- data.frame(  
+   id = c(31, 62, 50, 99, 53, 75, 54, 58, 4, 74),  
+   age = c(12, 18, 20, 17, 14, 8, 12, 24, 24, 21),  
+   sex = c("M", "F", "F", "M", "F", "M", "M", "M",  
+ "F", "M") )  
> pat.list <- as.list(patients)  
> pat.list  
$id  
[1] 31 62 50 99 53 75 54 58 4 74  
$age  
[1] 12 18 20 17 14 8 12 24 24 21  
$sex  
[1] M F F M F M M F F M  
Levels: F M
```

R coerced the variable `sex` into a categorical variable during the `data.frame` statement, which is only apparent now

Using the [[]] operator for lists

- [[]] extracts elements of lists (or \$ if list elements are named)

```
> pat.list[[1]]  
[1] 31 62 50 99 53 75 54 58 4 74
```

```
> pat.list$id  
[1] 31 62 50 99 53 75 54 58 4 74
```

```
> pat.list[[1]][1]  
[1] 31
```

```
> pat.list$id[1]  
[1] 31
```

Changing list elements

```
> pat.list$id <- matrix(pat.list$id, ncol=2)
> pat.list
$id
  [,1] [,2]
[1,]   31   75
[2,]   62   54
[3,]   50   58
[4,]   99     4
[5,]   53   74
$age
[1] 12 18 20 17 14 8 12 24 24 21
$sex
[1] M F F M F M M F F M
Levels: F M
```

Accessing elements of a matrix
inside a list

```
> pat.list[[1]][1,1]
[1] 31
> pat.list[[1]][1,]
[1] 31 75
```

Factors - Categorical variables in R

- A factor in R is a vector with discrete values assigned to individual elements, for example:
 - [Male, female]
 - [Democrat, Republican, Unaffiliated]
- Factors are used in basic data manipulation, plotting routines, and especially in statistical models
- R automatically specifies categorical variables as factors when
 - Creating data frames
 - Reading in data from files
 - Behind the scenes in many other applications

The `factor` function

- We can specify a categorical variable as a factor with the `factor()` command

```
> zone <- c("demersal", "pelagic", "reef",
  "demersal")
> is.factor(zone)
[1] FALSE
> zone.fac <- factor(zone)
> zone.fac
[1] demersal pelagic reef demersal
Levels: demersal pelagic reef
> is.factor(zone.fac)
[1] TRUE
```

Numbers to factors

- Often a categorical variable is coded numerically in a database; then the labels argument can be used:

```
> zone <- c(1, 1, 1, 2, 2, 2, 1, 2, 2, 1)
> zone.fac <- factor(zone, labels=c("demersal",
  "pelagic"))
> zone.fac
[1] demersal demersal demersal pelagic pelagic
pelagic demersal pelagic pelagic demersal
Levels: demersal pelagic
```

- To find the levels of a factor:

```
> levels(zone.fac)
[1] "demersal" "pelagic"
```

Hands-on exercise 1

- Create a 2×2 matrix `Amat` and a 2×3 matrix `Bmat`, each filled with unique numbers
- Combine `A` and `B` into a 2×5 matrix `Cmat` and a 5×2 matrix `Dmat`
- Create a factor `xfactor` from the following vector such that 1 is female and 2 is male
`sex <- c(1,1,2,1,2,2,2,1,1,1)`
- Create a list called `data` that contains matrices `Amat`, `Bmat` and `xfactor`
- Extract the first row of the matrix `Amat` from the list `data`
- Change to NA the value in row 1 and column 1 of matrix `Bmat` within `data`.

Reading in data

There are three oft-used functions

1. `scan()`
 - Most primitive, most flexible since it reads into a vector, and very fast, use for large or very messy data
2. `read.table()`
 - Easiest to use, reads into a data frame
3. `read.csv()`
 - Most useful for reading in Excel worksheets or other comma-separated data

Manual entry of data

Nearly always data are read in from a text file like .csv,
but data can be entered manually from the keyboard

```
> co2 <- scan()
1: 316
2: 316.91
3: 317.63
4: 318.46
5:
Read 4 items
> co2
[1] 316.00 316.91 317.63 318.46
```

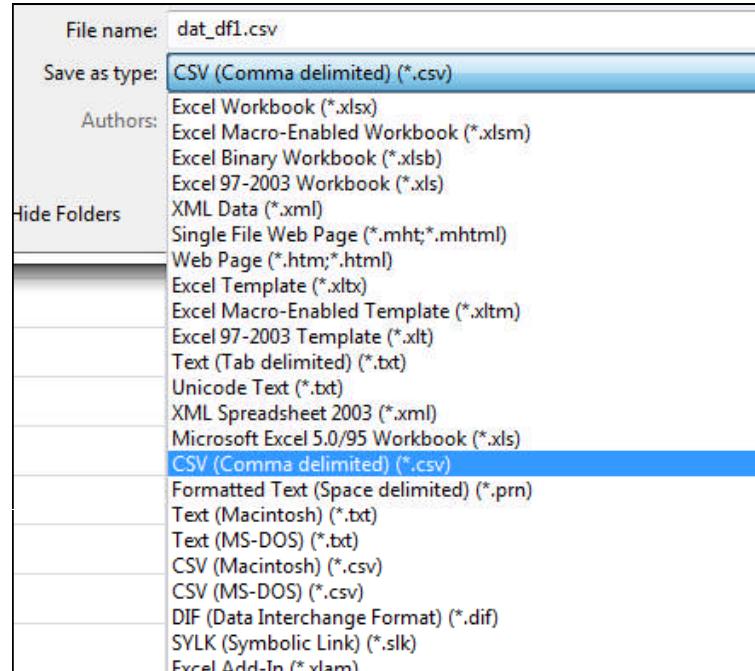
Reading in data from a file

- `read.table()` has a number of common options (useful defaults listed below)
 - `header=T` first row has names for columns
 - `sep=" "` how are entries separated (white space)
 - `na.strings=NA` which values are treated as NAs
 - `skip=0` the number of lines to skip before reading in data
 - `nrows=-1` number of lines of data to read (-1 means all)
 - `col.names=c("a", "b")` names for columns
- Download data files in varying formats from Canvas website under "files/data files"

Data from Excel

- Open fresh Excel workbook, single sheet, paste your data as values only, save as "Comma delimited (*.csv)"
- Read the data in using `read.csv()`

```
> read.csv( file="Data/dat_df1.csv", header=T)
  id age sex
1 31   12   M
2 62   18   F
3 50   20   F
```



Excel sheet problems

Data should start in the top left corner

	A	B	C	D	E
1	id	age	sex		
2	1	31	12	M	
3	2	62	18	F	
4	3	50	20	F	
5	4	99	17	M	
6	5	53	14	F	
7	6	75	8	M	
8	7	54	12	M	
9	8	58	24	F	
10	9	4	24	F	
11	10	74	21	M	
12	11				
13					
14					
15					
16					
17					

All columns to the right and rows below should be empty and always have been empty hence the need for an empty sheet

Or delete all rows below and all columns to the right in Excel before saving as .csv

References

- *An introduction to R* by Venables
 - <http://cran.r-project.org/doc/manuals/R-intro.pdf>
 - Chapters 4, 5.1-5.4, 5.9, 6.1-6.2, 7

Data Manipulation 3, in R

Lecture 5, Data manipulation, Trevor A. Branch,
FISH 552 Introduction to R

Outline for next three lectures

1. splitting data, applying functions across data sets, ordering data
2. merging data, dates, text manipulation, libraries in R
3. practical example

Quick review of data manipulation

- Many functions we have already seen in the class are useful tools in data manipulation
- Using **logical vectors** to subset data is often the quickest and easiest way to manipulate data: `&`, `|`, `!`,
`==`, `!=`
- Testing logical vector is also useful: `any`, `all`, `which`,
`which.max`
- What kind of data: `is.array`, `is.data.frame`,
`is.numeric`
- Is the data frame formatted correctly: `is.na`, `na.rm`

Testing logically

```
nums <- c(12,9,8,14,7,16,3,2,9)
```

- Are any numbers greater than 10?

```
> any(nums > 10)
```

```
[1] TRUE
```

- Are all the numbers greater than 10?

```
> all(nums > 10)
```

```
[1] FALSE
```

- Which numbers are greater than 10?

```
> which(nums > 10)
```

```
[1] 1 4 6
```

- These are equivalent statements

```
> nums[which(nums > 10)]
```

```
[1] 12 14 16
```

```
> nums[nums > 10]
```

```
[1] 12 14 16
```

Unique and duplicate observations

- A list of the unique observations in a vector

```
> plates <- c("WA", "WA", "OR", "RI", "WA", "WA", "CA",
  "WA", "WA")
> unique(plates)
[1] "WA" "OR" "RI" "CA"
```

- Which elements in a vector are duplicated

```
> duplicated(plates)
[1] FALSE TRUE FALSE FALSE TRUE TRUE FALSE TRUE TRUE
```

Testing modes

- You should **always** check that the data you are working with is behaving in the way you expect

randomly sample from normal distribution

```
> y <- rnorm(6)      sample with replacement from the vector
> x <- sample(factor(1:6))
> (xy <- data.frame(x,y))

  x         y
1 1 -0.39264361
2 2  1.05148538
3 3 -0.04623392
4 3  0.04002289
5 2 -0.32184645
6 1  0.41500848
```

Not at all obvious whether x is correctly coded as a factor

Testing attributes

- What kind of an object is x?

```
> is.numeric(xy$x)
```

```
[1] FALSE
```

```
> is.factor(xy$x)
```

```
[1] TRUE
```

- What are its attributes?

```
> attributes(xy$x)
```

```
$levels
```

```
[1] "1" "2" "3"
```

```
$class
```

```
[1] "factor"
```

Creating new factors (float to factor)

- Some data that are inherently continuous may need to be grouped categorically as a factor
- The `cut()` function is an elegant way to do this

```
> ages <- c(47,14,24,33,74)      ← Built-in constant "Infinity"
> cut(ages, breaks=c(0,18,65,Inf),
       labels=c("Kid", "Adult", "Senior"))
[1] Adult Kid Adult Adult Senior
Levels: Kid Adult Senior
> nums3 <- 1:100
> cut(nums3, breaks=3)
[1] (0.901,34] (0.901,34] (0.901,34] (0.901,34]
...
Levels: (0.901,34] (34,67] (67,100]
```

Useful built-in constants

```
> pi  
[1] 3.141593  
> letters  
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"  
"n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"  
> LETTERS  
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"  
"N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"  
> month.abb # Abbreviations  
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug"  
"Sep" "Oct" "Nov" "Dec"  
> month.name  
[1] "January" "February" "March" "April" "May" "June"  
"July" "August" "September" "October" "November"  
"December"
```

Built-in *undefined entities*

- `Inf`: positive infinity, e.g. `1/0`
- `-Inf`: negative infinity
- `NaN`: not a number, e.g. `sqrt(-1)`
- `NA`: is not-available/unknown/garbled/undefined
- `NULL`: completely blank, nothing there

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

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

Subsetting data

- We know how to subset a data frame or matrix, but when NAs are involved things may go awry

```
> x <- data.frame(a=c(11,13,12,15,17,20),  
                     b=c( 8,NA, 6, 4,NA,15))  
> x[x$b>5,]  
      a   b  
1    11   8  
NA   NA  NA  
3    12   6  
NA.1 NA  NA  
6    20  15
```

The subset() function

- Elegantly extracts portions of a data frame or matrix while handling NA values appropriately

- `subset(object, logical expression, variable selection)`

```
> x <- data.frame(a=c(11,13,12,15,17,20),  
                    b=c( 8,NA, 6, 4,NA,15))
```

```
> subset(x, b>5)
```

	a	b
1	11	8
3	12	6
6	20	15

```
> subset(x, b>7, a)
```

	a
1	11
6	20

Hands-on exercise 1

- Many functions used for data manipulation can be coded using logical expressions, but the tasks are common enough that R has built-in functions

```
temp <- c(1,2,3,1,2,3,1,2,6)
```

- Write code that is equivalent to `unique(temp)` using other operators or functions we have learned

```
x <- data.frame(a=c(11,13,12,15,17,20),  
                 b=c( 8,NA, 6, 4,NA,15))
```

- Perform the equivalent of `subset(x, b>5)` using other operators (hint: use `[]` but figure out how to handle NAs)
- Each example can be done in one line of code

The `apply()` function

- Amazingly flexible function
- `apply(X, MARGIN, FUN, ...)`
 - X is a matrix
 - MARGIN is 1 or 2: 1=rows, 2=columns
 - FUN is an R function (can be user-defined)
 - ... additional arguments to the function named in FUN

```
> m <- matrix(1:12, nrow=3)
> apply(X=m, MARGIN=2, FUN=mean)
[1] 2 5 8 11 # mean of each column
> apply(X=m, MARGIN=1, FUN=mean)
[1] 5.5 6.5 7.5 # mean of each row.
```

> m	[,1]	[,2]	[,3]	[,4]
[1,]	1	4	7	10
[2,]	2	5	8	11
[3,]	3	6	9	12

More on apply()

> m	[,1]	[,2]	[,3]	[,4]
	[1,]	1	4	7
	[2,]	2	5	8
	[3,]	3	6	9

```
> quantile(m[1,], c(0.05,0.5,0.95))  
 5% 50% 95%  
1.45 5.50 9.55
```

```
> apply(X=m, MARGIN=1, FUN=quantile,  
        c(0.05,0.5, 0.95))  
      [,1]   [,2]   [,3]  
5% 1.45 2.45 3.45  
50% 5.50 6.50 7.50  
95% 9.55 10.55 11.55
```

Add the extra parameters of quantile here

apply()

`round(x, 3)` rounds to the third decimal

```
> round(subj,3)
      Yr 1   Yr 2   Yr 3   Yr 4   Yr 5
Patient 1 -0.207 -0.353  0.240  0.213 -0.053
Patient 2  1.535  0.559 -0.189  0.783  0.657
Patient 3 -0.012 -1.213 -0.244 -0.789 -0.318
...
Patient 20 -1.307 -0.209 -0.440  0.465  0.114
```

```
> round(apply(subj, MARGIN=1, mean),3)
Patient 1 Patient 2 Patient 3 Patient 4 Patient 5
-0.032      0.669     -0.515     -0.789      1.660
Patient 6 Patient 7 Patient 8 Patient 9 Patient 10
-0.139     -0.654      0.012      0.131      0.213
...
```

```
> round(apply(subj, MARGIN=2, mean),3)
      Yr 1   Yr 2   Yr 3   Yr 4   Yr 5
-0.057 -0.172  0.153 -0.127 -0.143
```

To apply to items in a list, use `sapply()` and `tapply()`

rowMean() and colMean()

An easy way to get the means of the rows or columns

```
> round(rowMeans(subj),3)
```

```
Patient 1 Patient 2 Patient 3 Patient 4 Patient 5  
-0.032      0.669     -0.515     -0.789      1.660
```

```
Patient 6 Patient 7 Patient 8 Patient 9 Patient 10  
-0.139     -0.654      0.012      0.131      0.213
```

```
...
```

```
> round(colMeans(subj),3)
```

```
Yr 1    Yr 2    Yr 3    Yr 4    Yr 5  
-0.057 -0.172  0.153 -0.127 -0.143
```

tapply()

- Apply a function to a vector using a categorical variable

```
lengths <- sample(1:100, size=20, replace=T)
82 49 63 33 60 15 65 52 22 66 27 36 6 77 57 83 86 13 96
31 This could be the lengths of 20 fish
genders <- sample(c("Male","Female","Unknown"),
                  size=20, replace=T)
"Female" "Female" "Female" "Female" "Male" "Male"
"Unknown" "Female" "Unknown" "Male" "Female" "Unknown"
"Male" "Female" "Unknown" "Unknown" "Unknown" "Male"
"Female" "Male" The gender of the 20 fish
> tapply(X=lengths, INDEX=genders, FUN=mean)
Female      Male  Unknown
51.60000 40.11111 7.00000
```

More tapply()

```
> XX <- data.frame(lengths, genders)
> XX
  lengths genders
1      64     Male
2      50   Female
3      20   Female
...
> tapply(x=XX$lengths, INDEX=XX$genders, FUN=max)
Female  Male Unknown
      100     85       7
```

Sorting and ordering

- We often want to rearrange a data set by a single variable; in R `sort()` and `order()` do this
- `sort()` orders a vector in increasing order

```
> (cards <- sample(1:10))
[1] 6 5 3 10 7 9 1 8 4 2
> sort(cards)
[1] 1 2 3 4 5 6 7 8 9 10
> rev(sort(cards))      Reverse the order
[1] 10 9 8 7 6 5 4 3 2 1
```

Using `order()`

- `order()` returns the sort index of each element

```
> cards
```

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

```
> order(cards)
```

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

- In other words, if you take the 7th element, then the 10th element, then the 3rd element... the vector cards will be sorted. Or in R terms:

```
> cards[order(cards)] # Get a sorted list.
```

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

order() is best used for arrays

```
> YY[order(YY$ID),]  
# ID    dev  
3  1 -0.054  
9  2 -1.397  
7  3 -1.195  
1  4  0.808  
...  
.
```

data frame YY

	ID	dev
1	4	0.808
2	6	1.084
3	1	-0.054
4	7	0.907
5	9	0.380
6	10	0.423
7	3	-1.195
8	5	-0.340
9	2	-1.397
10	8	1.767

Sub-ordering

- Giving `order()` two or more vectors results in the order of vector 2 within vector 1

```
> zz[order(zz$laws, zz$year, zz$state), ]
```

	laws	year	state
10	1	2010	TX
8	1	2010	UT
7	1	2012	FL
3	2	2010	CA
2	2	2010	OR
9	2	2012	AZ
5	2	2012	NY
4	3	2010	VT
6	3	2012	RI
1	3	2012	WA

First order by laws,
then by year, then
by state

data frame ZZ

	laws	year	state
1	3	2012	WA
2	2	2010	OR
3	2	2010	CA
4	3	2010	VT
5	2	2012	NY
6	3	2012	RI
7	1	2012	FL
8	1	2010	UT
9	2	2012	AZ
10	1	2010	TX

Hands-on exercise 2

The `PlantGrowth` data frame is built into R, using this dataset:

- Examine the data thoroughly
- Calculate the mean and standard deviation for each treatment and the control
- Create a factor for small and large based on the weights
- Create a new data frame that just contains the control data
- Sort the weights
- Now display the entire data frame sorted by weight

References

- Data manipulation in R (Phil Spector, 2008)
 - <http://www.springerlink.com/content/t19776/>
 - Free download
 - Chapters 1, 5, 8

Data Manipulation: DateTime and Packages in R

Lecture 6 on Data manipulation II, by Trevor A. Branch,
in the course FISH 552, titled Introduction to R

Speed testing

- `system.time()` number of seconds to run a command

```
> temp <- sample(1:100, size=10000000, replace=T)
```

```
> system.time( unique(temp) )
```

user system elapsed 10,000,000-long vector of numbers

0.40 0.07 0.47

```
> system.time( temp[!duplicated(temp)] )
```

user system elapsed Slower than using unique

0.96 0.06 1.04

```
> system.time( temp[which(!duplicated(temp))] )
```

user system elapsed Should be slower, but actually is faster???

0.48 0.08 0.56

```
> system.time( as.numeric(levels(factor(temp))) )
```

user system elapsed MUCH slower

8.09 0.19 8.35

Combining data sources

- We often want to combine two source of data by **merging** a common variable or observation
 - e.g. Data measured by different instruments at overlapping times but on different time scales
 $\{0,5,10,15,\dots\}$ and $\{0,10,20,\dots\}$
- Such tasks can be difficult to code using only logical operations, `rbind()`, and `cbind()`. Instead, `merge()` combines data very effectively
- Explore the help function `?merge`

merge()

- Two sources of data, station1 and station2 with overlapping time measurements

```
> station1
```

	time1	data
[1,]	1	-0.069745093
[2,]	2	0.008806333
[3,]	3	-0.289268911
[4,]	4	0.944776136
...		
[98,]	98	-1.024886327
[99,]	99	-0.994975640
[100,]	100	-0.673815187

```
> station2
```

	time2	category
[1,]	0	1
[2,]	5	2
[3,]	10	2
[4,]	15	1
...		
[19,]	90	1
[20,]	95	3
[21,]	100	3

Merge using common time

- Create a single data set with both variables having common time observations

```
> merge(station1, station2, by.x="time1",
       by.y="time2")
```

	time1	data	category
1	5	1.58535005	2
2	10	1.00572430	2
3	15	1.12442383	1
4	20	-1.20569332	1
...			
18	90	-0.08973228	1
19	95	0.33275114	3
20	100	-0.67381519	3

Behind the scene of `merge()`

- The `merge()` function relies on other R functions that can be used for data manipulation
- Finding common elements in vectors: `intersect()`
`> intersect(1:10, 7:20)`
`[1] 7 8 9 10`
- Matching positions of common elements in vectors

```
match(x, table, nomatch=NA)  
> match(1:10, c(1,3,5,9))  
[1] 1 NA 2 NA 3 NA NA NA 4 NA
```

Hands-on exercise 1

- Write code that returns a TRUE/FALSE vector indicating whether the elements in one vector match (TRUE) any of the elements in the other vector
- Use the same two vectors `1:10` and `c(1, 3, 5, 9)`
 - Look at the help function for `match`. Seriously, look at it.
 - Hint: use the `match()` function. How can you change the results of `match` to TRUE/FALSE?

How to store and handle dates

- R has a built-in class "Date" to handle data entered as a date in various formats
- The `as.Date()` function allows a variety of input formats through the `format =` argument

```
> as.Date("2013/10/15")  
[1] "2013-10-15"
```

Formatting dates in R

- If your input dates are not in a standard format, you can add a format string, as follows:

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (two digits)
%Y	Year (four digits)

Using date formats

```
> as.Date('9/22/1983', format = '%m/%d/%Y')
[1] "1983-09-22"
> as.Date('September 22, 1983',
           format = '%B %d, %Y')
[1] "1983-09-22"
> as.Date('22SEP83', format = '%d%b%y')
[1] "1983-09-22"
> as.Date('22sep83', format = '%d%b%y')
[1] "1983-09-22"
```

Gracefully handles upper/lower case
R function toupper(x) converts to upper case

Extracting date components

- Components of dates can easily be extracted, provided the items are of class `PosIXt` or `Date`

```
> weekdays(as.Date("2013/10/15"))
[1] "Tuesday"
> months(as.Date("2013/10/15"))
[1] "October"
> quarters(as.Date("2013/10/15"))
[1] "Q4"
> julian(as.Date("2013/10/15"),
  origin=as.Date("2013/01/01"))      Number of days from the start
                                         of the year
[1] 287
attr(,"origin")
[1] "2013-01-01"
```

Sub-daily time scales

- Most dates read from instruments have a finer time scale than days (hours, minutes, seconds)
- For these, use the `POSIXct` class in R (and not the `Date` class)
- Default input format the `POSIXct` class consists of the year, month, day (separated by slashes or dashes), time values may be followed by white space and a time in the form hours:minutes:seconds or hours:minutes:
 - `1983/9/22 23:20:05`
 - If the dates are not in this format, see help on `strptime()`

Converting to POSIXt

```
> as.POSIXlt("1983-9-22 23:20:05")
[1] "1983-09-22 23:20:05"
> aDate <- as.POSIXlt("1983-9-22 23:20:05")
```

POSIXct includes time zone information

```
> as.POSIXct("1983-9-22 23:20:05")
[1] "1983-09-22 23:20:05 PDT"
> aDate <- as.POSIXct("1983-9-22 23:20:05")
```

Adding and averaging dates

- Many common functions can accept objects of Date class: `min()`, `mean()`, `max()`, ...
- The `difftime()` function computes the difference between two time dates

```
> mean(c(as.Date("2013/10/15"), as.Date("2010/06/14")))
[1] "2012-02-13"
> max(c(as.Date("2013/10/15"), as.Date("2010/06/15")))
[1] "2013-10-15"
> min(c(as.Date("2013/10/15"), as.Date("2010/06/15")))
[1] "2010-06-15"
> difftime(as.Date("2013/10/15"), as.Date("2010/06/14"))
Time difference of 1219 days
```

Converting dates to factors

- First create a vector of the days of the year

```
> everyday <- seq(from=as.Date("2013-01-01"),  
+                   to=as.Date("2013-12-31"), by="day")  
> everyday  
[1] "2013-01-01" "2013-01-02" "2013-01-03" ...
```
- Then convert to factors

```
> month <- months(everyday)  
> month <- factor(month, levels=unique(month),  
+ ordered=TRUE)  
> table(month)  
month  
January February March ...  
31          28      31 ...
```

Hands-on exercise 2

- Choose any two dates you like
- Save them as two objects in the year-month-date format
- Apply R functions to determine
 - the day of the week
 - the month
 - the difference between the two dates

Packages in R

- you can load the `MASS` package
 - This contains functions and data sets from Venables & Ripley “*Modern applied statistics with R*”
- Packages are a key feature in R, allowing users to benefit from other’s contributions
- Before performing any truly arduous programming task, ask yourself whether someone else is likely to have already done that
- Search for contributed packages that might already have the features you need

~/FISH552 Intro R/Lectures - RStudio

File Edit Code View Plots Session Project Build Tools Help

Workspace History

Values

everyday	Date[365]
month	ordered[365]
X	Date[1]
v	Date[1]

Files Plots Packages Help

Install Packages Check for Updates

bbmle	Tools for general maximum likelihood estimation	1.0.5.2
bitops	Bitwise Operations	1.0-5
boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3-9
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.14
class	Functions for Classification	7.3-7
cluster	Cluster Analysis Extended Rousseeuw et al.	1.14.4
codetools	Code Analysis Tools for R	0.2-8
colorspace	Color Space Manipulation	1.2-3
compiler	The R Compiler Package	3.0.1
<input checked="" type="checkbox"/> datasets	The R Datasets Package	3.0.1
devtools	Tools to make developing R code easier	1.2
dichromat	Color Schemes for Dichromats	2.0-0
digest	Create cryptographic hash digests of R objects	0.6.3
evaluate	Parsing and evaluation tools that provide more details than the default.	0.4.3
foreign	Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...	0.8-53
gdata	Various R programming tools for data manipulation	2.12.0.2
ggplot2	An implementation of the Grammar of Graphics	0.9.3.1

install a new package

already installed packages

Checking the box loads the package

Click on the name of the package to get a list of its functions

Installing and loading packages

- Once a package has been installed, it needs to be loaded into R
- This can be done by ticking the box next to the package
- In your R code, loading is done using one of
 - `library(package)` forces package to load every time
 - `require(package)` only loads package if not already loaded (I always use this version)
- R will return a warning if the package was compiled on a newer R version or if the version of R is incompatible with an older package

Finding packages

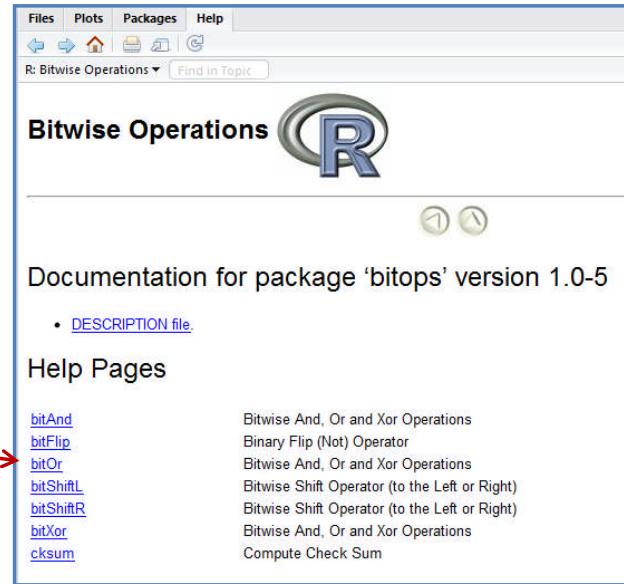
- Task views <http://cran.r-project.org/web/views/> that categorize packages into groups
- Ask other people
- Active community on Twitter use hashtag #Rstats
- Search engine (also try www.rseek.org)
- Scientific papers describing new methods, e.g. bathymetry plotting package `marmap`:
Pante E & Simon-Bouhet B (2013) PLOS ONE 8(9):e73051
- Start with the most popular 100 downloaded packages: <http://www.r-bloggers.com/top-100-r-packages-for-2013-jan-may/>

<http://www.r-bloggers.com/top-100-r-packages-for-2013-jan-may/>

	PACKAGE	TITLE	DOWNLOADS
1	plyr	Tools for splitting, applying and combining data	84049
2	digest	Create cryptographic hash digests of R objects	83192
3	ggplot2	An implementation of the Grammar of Graphics	82768
4	colorspace	Color Space Manipulation	81901
5	stringr	Make it easier to work with strings	77658
6	RColorBrewer	ColorBrewer palettes	66783
7	reshape2	Flexibly reshape data: a reboot of the reshape package	64911
8	zoo	S3 Infrastructure for Regular and Irregular Time Series (Z's ordered observations)	60844
9	proto	Prototype object-based programming	59043
10	scales	Scale functions for graphics	58369

Help on packages

- Click on the package for a list of functions
- You can also find vignettes online, for example
<http://cran.r-project.org/web/packages/survival/index.html>
- Many packages have an overview called a vignette that includes examples of the key functions
 - > `vignette(all=FALSE)` Lists vignettes for installed and attached packages
 - > `vignette(all=TRUE)` Lists vignettes for all installed packages
 - > `vignette("googlevis")` Open a particular vignette



References

- Data manipulation in R (Phil Spector, 2008)
 - <http://www.springerlink.com/content/t19776/>
 - Chapters 4.1, 8

Programming in R

Control Flow

Topics

1. If, else, switch,
2. loop, repeat, for,
3. functions, apply, source

Katia Oleinik, koleinik@bu.edu, Scientific Computing and Visualization, Boston University <http://www.bu.edu/tech/research/training/tutorials/list>

if

```
if (condition) {  
    command(s)  
} else {  
    command(s)  
}
```

Comparison operators:

<code>==</code>	equal
<code>!=</code>	not equal
<code>> (<)</code>	greater (less)
<code>>= (<=)</code>	greater (less) or equal

Logical operators:

<code>&</code>	and
<code> </code>	or
<code>!</code>	not

if

```
> # define x  
> x <- 7  
  
> # simple if statement  
> if ( x < 0 ) print("Negative")  
  
> # simple if-else statement  
> if ( x < 0 ) print("Negative") else print("Non-negative")  
[1] "Non-negative"  
  
> # if statement may be used inside other constructions  
> y <- if ( x < 0 ) -1 else 0  
> y  
[1] 0
```

if

```
> # multiline if - else statement
> if ( x < 0 ) {
+     x <- x+10
+     print("x is negative: subtract 10")
+ } else if ( x == 0 ) {
+     print("x is equal zero")
+ } else {
+     print("x is positive: add 10")
+ }
[1] positive
```

Note: For multiline if-statements **braces are necessary** even for single statement bodies. The left and right braces must be on the same line with else keyword (in interactive session).

~~ifelse~~

```
ifelse (test_condition, true_value, false_value)
```

```
> # ifelse statement
> y <- ifelse ( x < 0, -1, 0 )

> # nested ifelse statement
> y <- ifelse ( x < 0, -1, ifelse (x > 0, 1, 0) )
```

~~ifelse~~

Best of all – **ifelse** statement operates on vectors!

```
> # ifelse statement on a vector
> digits <- 0 : 9
> (odd <- ifelse( digits %% 2 > 0, TRUE, FALSE ))
[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
```

~~if else~~

Exercise:

- define a random vector ranging from -10 to 10:
`x<- as.integer(runif(10, -10, 10))`
- create vector y, such that its elements equal to absolute values of x

Note: normally, you would use **abs()** function to achieve this result

~~switch~~

```
switch (statement, list)
```

```
> # simple switch statement
> x <- 3
> switch( x, 2, 4, 6, 8 )
[1] 6

> switch( x, 2, 4 ) # returns NULL since there are only 2 elements in the list
```

~~switch~~

```
switch (statement, name1 = str1, name2 = str2, ... )
```

```
> # switch statement with named list
> day <- "Tue"
> switch( day, Sun = 0, Mon = 1, Tue = 2, Wed = 3, ... )
[1] 2

> # switch statement with a "default" value
> food <- "meet"
> switch( food, banana="fruit", carrot="veggie", "neither" )
[1] "neither"
```

loops

There are 3 statements that provide explicit looping:

- **repeat**
- **for**
- **while**

Built – in constructs to control the looping:

- **next**
- **break**

Note: Use explicit loops only if it is absolutely necessary. R has other functions for implicit looping, which will run much faster: **apply()**, **sapply()**, **tapply()**, and **lapply()**.

~~repeat~~

repeat { } statement causes repeated evaluation of the body until **break** is requested. Be careful – infinite loop may occur!

```
> # find the greatest odd divisor of an integer
> x <- 84
> repeat{
+   print(x)
+   if( x%%2 != 0) break
+   x <- x/2
+ }
[1] 84
[1] 42
[1] 21
>
```

for

```
for (object in sequence) {  
  command(s)  
}
```

```
> # print all words in a vector  
> names <- c("Sam", "Paul", "Michael")  
>  
> for( j in names ){  
+   print(paste("My name is" , j))  
+ }  
  
[1] "My name is Sam"  
[1] "My name is Paul"  
[1] "My name is Michael"  
  
>
```

for

```
for (object in sequence) {  
    command(s)  
  
    if (...) next      # return to the start of the loop  
  
    if (...) break     # exit from (innermost) loop  
}
```

while

```
while (test_statement) {  
    command(s)  
}
```

```
> # find the largest odd divisor of a given number  
> x <- 84  
> while (x %% 2 == 0){  
+     x <- x/2  
+ }  
> x  
[1] 21  
>
```

~~loops~~

Exercise:

- Using either loop statement print all the numbers from 0 to 30 divisible by 7.
- Use `%%` - modular arithmetic operator to check divisibility.

function

```
myFun <- function (ARG, OPT_ARGS ) {  
  statement(s)  
}
```

ARG: vector, matrix, list or a data frame
OPT_ARGS: optional arguments

Functions are a powerful R elements. They allows you to expand on existing functions by writing your own custom functions.

~~function~~

```
myFun <- function (ARG, OPT_ARGS ) {  
  statement(s)  
}
```

Naming:

Variable naming rules apply. Avoid usage of existing (built-in) functions

Arguments:

Argument list can be empty.

Some (or all) of the arguments can have a default value (arg1 = TRUE)

The argument '...' can be used to allow one function to pass on argument settings to another function.

Return value:

The value returned by the function is the last value computed, but you can also use return() statement.

function

```
> # simple function: calculate (x+1)2
> myFun <- function (x) {
+   x2 + 2*x + 1
+ }
> myFun( 3 )
[1] 16
>
```

function

```
> # function with optional arguments: calculate (x+a)2
> myFun <- function (x, a=1) {
+   x2 + 2*x*a + a2
+
+ }
> myFun( 3 )
[1] 16
> myFun( 3 , 2 )
[1] 25
>
> # arguments can be called using their names ( and out of order!!!)
> myFun( a = 2 , x = 1 )
[1] 9
```

function

```
> # Some optional arguments can be specified as '...' to pass them to another function
> myFun <- function (x, ...) {
+   plot (x, ...)
+ }
>
> # print all the words together in one sentence
> myFun <- function ( ...) {
+   print(paste ( ... ) )
+ }
> myFun("Hello", " R! ")
[1] "Hello R!"
```

function

Local and global variables:

All variables appearing inside a function are treated as local, except their initial value will be of that of the global (if such variable exists).

```
> # define a function
> myFun <- function (x) {
+   cat ("u=", u, "\n")    # this variable is local !
+   u<-u+1                 # this will not affect the value of variable outside f()
+   cat ("u=", u, "\n")
+ }
>
> u <- 2                  # define a variable
> myFun(5)                #execute the function
u= 2
u= 3
>
> cat ("u=", u, "\n")    # print the value of the variable
u= 2
```

function

Local and global variables:

If you want to access the global variable – you can use the super-assignment operator <<-. **You should avoid doing this!!!**

```
> # define a function
> myFun <- function (x) {
+   cat ("u=", u, "\n")      # this variable is local !
+   u <<- u+1      # this WILL affect the value of variable outside f()
+   cat ("u=", u, "\n")
+
>
> u <- 2          # define a variable
> myFun(u)          #execute the function
u= 2
u= 3
>
> cat ("u=", u, "\n")      # print the value of the variable
u= 3
>
```

function

Call vector variables:

Functions do not change their arguments.

```
> # define a function
> myFun <- function (x) {
+   x <- 2
+   print (x)
+
>
> x <- 3      # assign value to x
> y <- myFun(x)  # call the function
[1] 2
>
> print(x)    # print value of x
[1] 3
>
```

function

Call vector variables:

If you want to change the value of the function's argument, reassign the return value to the argument.

```
> # define a function
> myFun <- function (x) {
+   x <- 2
+   print (x)
+ }
>
> x <- 3      # assign value to x
> x <- myFun(x)  # call the function
[1] 2
>
> print(x)    # print value of x
[1] 2
>
```

function

Finding the source code:

You can find the source code for any R function by printing its name without parentheses.

```
> # get the source code of lm() function
> lm
function (formula, data, subset, weights, na.action, method = "qr",
model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
contrasts = NULL, offset, ...)
{
  ret.x <- x
  ret.y <- y
  cl <- match.call()

  . . .

z
}
<environment: namespace:stats>
>
```

function

Finding the source code:

For generic functions there are many methods depending on the type of the argument.

```
> # get the source code of mean() function
> mean
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
>
```

function

Finding the source code:

You can first explore different methods and then chose the one you need.

```
> # get the source code of mean() function
> methods( "mean" )
[1] mean.Date      mean.POSIXct     mean.POSIXlt     mean.data.frame
[5] mean.default   mean.difftime
>
> # get source code
> mean.default
function (x, trim = 0, na.rm = FALSE, ...)
{
  if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
    . . .
}
<environment: namespace:stats>
```

apply

```
apply (OBJECT, MARGIN, FUNCTION, ARGs )
```

object: vector, matrix or a data frame

margin: 1 – rows, 2 – columns, c(1,2) – both

function: function to apply

args: possible arguments

Description:

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix

apply

Example:

Create matrix and apply different functions to its rows and columns.

```
> # create 3x4 matrix
> x <- matrix( 1:12, nrow = 3, ncol = 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
>
```

apply

Example:

Create matrix and apply different functions to its rows and columns.

```
> # create 3x4 matrix
> x <- matrix( 1:12, nrow = 3, ncol = 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> # find median of each row
> apply (x, 1, median)
[1] 5.5 6.5 7.5
>
```

apply

Example:

Create matrix and apply different functions to its rows and columns.

```
> # create 3x4 matrix
> x <- matrix( 1:12, nrow = 3, ncol = 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> # find mean of each column
> apply (x, 2, mean)
[1] 2 5 8 11
>
```

apply

Example:

Create matrix and apply different functions to its rows and columns.

```
> # create 3x4 matrix
> x <- matrix( 1:12, nrow = 3, ncol = 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> # create a new matrix with values 0 or 1 for even and odd elements of x
> apply (x, c(1,2), function (x) x%%2)

     [,1] [,2] [,3] [,4]
[1,]    1    0    1    0
[2,]    0    1    0    1
[3,]    1    0    1    0
>
```

lapply

|**lapply()** function returns a list:

```
lapply(X, FUN, ...)
```

```
> # create a list
> x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE))

> # compute the list mean for each list element
> lapply (x, mean)
$a
[1] 5.5

$beta
[1] 4.535125

$logic
[1] 0.3333333
>
```

~~sapply~~

|sapply() function returns a vector or a matrix:

```
sapply(X, FUN, ... , simplify = TRUE, USE.NAMES = TRUE)
```

```
> # create a list
> x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE))

> # compute the list mean for each list element
> sapply (x, mean)
      a      beta    logic
5.5000000 4.5351252 0.3333333
>
```

~~code sourcing~~

```
source ( "file" , ... )
```

file: file with a source code to load (usually with extension .r)

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

code sourcing

```
katana:~ % emacs foo_source.r &
```

Linux prompt

```
# dummy function
foo <- function(x){
  x+1
}
```

Text editor

```
> # load foo.r source file
> source ("foo_source.r")

> # create a vector
> x <- c(3,5,7)

> # call function
> foo(x)
[1] 4 6 8
```

R session

code sourcing

```
> # load foo.r source file
> source ("foo_source.r", echo = TRUE)

> # dummy function
> foo <- function(x){
+   x+1;
+ }

> # create a vector
> x <- c(3,5,7)

> # call function
> foo(x)
[1] 4 6 8
```

code sourcing

Exercise:

- write a function that computes a logarithm of inverse of a number $\log(1/x)$
- save it in the file with .r extension
- load it into your workspace
- execute it
- try execute it with input vector $v=c(2, 1, 0, -1)$.

References

1. Intro to R, Venables and Smith, <http://cran.r-project.org/doc/manuals/R-intro.pdf>
<http://cran.r-project.org/manuals.html>
2. Basic Statistics tests in R,
<http://www.statmethods.net/stats/index.html>
3. Advanced Probability/Statistics in R,
http://zoonek2.free.fr/UNIX/48_R/all.html

Programming in R

debugging and optimizing

Topics

1. Debugging
2. Timing
3. Optimization
4. Measure speedup
5. Compiling
6. Profiling

Katia Oleinik, koleinik@bu.edu, Scientific Computing and Visualization, Boston University <http://www.bu.edu/tech/research/training/tutorials/list>

debugging

R package includes debugging tools.

cat () & **print** () – print out the values

browser () – pause the code execution and “browse” the code

debug (FUN) – execute function line by line

undebug (FUN) – stop debugging the function

debugging

```
# dummy function
inv_log <- function(x){
  y <- 1/x
  browser()
  y <- log(y)
}
```

inv_log.r

```
> # load foo.r source file
> source ("inv_log.r", echo = TRUE)
> # dummy function
> inv_log <- function(x){
+   y<-1/x;
+   browser();
+   y<-log(y);
+ }
> inv_log (x) # call function
Called from: inv_log(x)
Browse[1]> y      # check the values of local variables
[1] 0.3333333 0.5000000 1.0000000 Inf -1.0000000
```

debugging

<RET> Go to the next statement if the function is being debugged.
Continue execution if the browser was invoked.

c or cont Continue execution without single stepping.

n Execute the next statement in the function. This works from
the browser as well.

where Show the call stack.

Q Halt execution and jump to the top-level immediately.

To view the value of a variable whose name matches one of these
commands, use the print() function, e.g. print(n).

debugging

```
# dummy function  
inv_log <- function(x){  
  y <- 1/x  
  browser()  
  y <- log(y)  
}
```

inv_log.r

```
> # load foo.r source file  
> source ("inv_log.r", echo = TRUE)  
> # dummy function  
> inv_log <- function(x){  
+   y<-1/x;  
+   browser();  
+   y<-log(y);  
+ }  
> inv_log (x) # call function  
Called from: inv_log(x)  
Browse[1]> y  
[1] 0.3333333 0.5000000 1.0000000 Inf -1.0000000  
Browse[1]> n  
debug: y <- log(y)  
Browse[2]>  
Warning message:  
In log(y) : NaNs produced  
>
```

debugging

```
# dummy function
inv_log <- function(x){
  y <- 1/x
  y <- log(y)
}
```

inv_log.r

```
> # load foo.r source file
> source ("inv_log.r", echo = TRUE)
> # dummy function
> inv_log <- function(x){
+   y<-1/x;
+   y<-log(y);
+ }
> debug(inv_log) # debug mode
> inv_log (x) # call function
Called from: inv_log(x)
debugging in: inv_log(x)
debug: {
  y <- 1/x
  y <- log(y)
}
Browse[2]>
. . .
> undebug(inv_log) # exit debugging mode
```

timing

Use `system.time()` functions to measure the time of execution.

```
> # make a function
> g <- function(n) {
+   y = vector(length=n)
+   for (i in 1:n) y[i]=i/(i+1)
+   y
+ }
```

timing

Use `system.time()` functions to measure the time of execution.

```
> # make a function
> myFun <- function(x) {
+   y = vector(length=x)
+   for (i in 1:x) y[i]=i/(i+1)
+   y
+ }
```



```
> # execute the function, measuring the time of the execution
> system.time( myFun(100000) )
  user  system elapsed
 0.107  0.002  0.109
```

optimization

How to speed up the code?

- Use vectors !

```
> # using loops  
> g1 <- function(x) {  
+   y = vector(length=x)  
+   for (i in 1:x) y[i]=i/(i+1)  
+   y  
+ }
```

```
> # using vectors  
> x <- (1:100000)  
> g2 <- function(x) {  
+   x/(x+1)  
+ }  
>
```

optimization

Measure the speed

- Time the use of vectors !

```
> # using loops
> g1 <- function(x) {
+   y = vector(length=x)
+   for (i in 1:x) y[i]=i/(i+1)
+   y
+ }

> # execute the function
> system.time( g1(100000) )
  user  system elapsed
0.107    0.002   0.109
```

```
> # using vectors
> x <- (1:100000)
> g2 <- function(x) {
+   x/(x+1)
+ }

> # execute the function
> system.time( g2(x) )
  user  system elapsed
0.002    0.000   0.003
```

optimization

How to speed up the code?

- Avoid dynamically expanding arrays

```
> vec1<-NULL  
  
> # execute the command  
> system.time(  
+ for(i in 1:100000)  
+ vec1 <- c(vec1,mean(1:100)))  
    user   system elapsed  
58.181   0.193  58.417
```

```
> vec2 <- vector(  
+ mode="numeric",length=100000)  
  
> # execute the command  
> system.time(  
+ for(i in 1:100000)  
+ vec2[i] <- mean(1:100))  
    user   system elapsed  
2.324   0.063  2.388
```

optimization

How to speed up the code?

- Avoid dynamically expanding arrays

```
> f1<-function(x){  
+ vec1 <- NULL  
+ for(i in 1:100000)  
+   vec1 <- c(vec1,mean(1:10))  
+ }  
  
> # execute the command  
> system.time( f1(0) )  
    user  system elapsed  
 57.035   0.209  57.280
```

```
> f2<-function(x){  
+ vec2 <- vector(  
+ mode="numeric",length=100000)  
+ for(i in 1:100000)  
+   vec2[i] <- mean(1:10)  
+ }  
  
> # execute the command  
> system.time( f2(0) )  
    user  system elapsed  
 2.096   0.067   2.163
```

optimization

How to speed up the code?

- Use optimized R-functions, i.e.
rowSums(), rowMeans(), table(), etc.
- In some simple cases – it is worth it to write your own!

```
> matx <- matrix  
+   (rnorm(1000000),100000,10)  
  
> # execute the command  
> system.time(apply(matx,1,mean))  
    user   system elapsed  
2.686   0.057   2.748
```

```
> matx <- matrix  
+   (rnorm(1000000),100000,10)  
  
> # execute the command  
> system.time(rowMeans(matx))  
    user   system elapsed  
0.013   0.000   0.014
```

optimization

How to speed up the code?

- Use optimized R-functions, i.e.
rowSums(), rowMeans(), table(), etc.
- In some simple cases – it is worth it to write your own!

```
> system.time(  
+ for(i in 1:100000)mean(1:100))  
  
 user    system elapsed  
 1.862   0.052   1.914
```

```
> system.time(  
+ for(i in 1:100000)  
+ sum(1:100) / length(1:100) )  
  
 user    system elapsed  
 0.485   0.013   0.498
```

optimization

How to speed up the code?

- Use vectors
- Avoid dynamically expanding arrays
- Use optimized R-functions, i.e.
 rowSums(), rowMeans(), table(), etc.
- In some simple cases – it is worth it to write your own implementation!
- **Use R - compiler or C/C++ code**

compiling

Use **library(compiler)**:

- `cmpfun()` - compile existing function
- `cmpfile()` - compile source file
- `loadcmp()` - load compiled source file

compiling

```
# dummy function  
fsum <- function(x){  
  s <- 0  
  for ( n in x) s <- s+n  
  s  
}  
  
> # load compiler library  
> library (compiler)  
  
> # load function from a source file (if necessary)  
> source ("fsum.r")  
  
> # load function from a source file (if necessary)  
> fsumcomp <- cmpfun(fsum)
```

compiling

Using compiled functions decreases the time of computation.

```
> # run non-compiled version  
> system.time(fsum(1:100000))  
  
 user  system elapsed  
0.071   0.000   0.071
```

```
> # run compiled version  
> system.time(fsumcomp(1:100000))  
  
 user  system elapsed  
0.025   0.001   0.026
```

profiling

Profiling is a tool, which can be used to find out how much time is spent in each function. Code profiling can give a way to locate those parts of a program which will benefit most from optimization.

`Rprof()` – turn profiling on

`Rprof(NULL)` – turn profiling off

`summaryRprof("Rprof.out")` – Summarize the output of the `Rprof()` function to show the amount of time used by different R functions.

profiling

Brownian Motion simulation.

Input: x - initial position, steps - number of steps

bm.R

```
# slow version of BM function
bmslow <- function (x, steps){

  BM <- matrix(x, nrow=length(x))
  for (i in 1:steps){

    # sample from normal distribution
    z <- rnorm(2)

    # attach a new column to the output matrix
    BM <- cbind (BM,z)
  }
  return(BM)
}
```

profiling

Brownian Motion simulation.

Input: x - initial position, steps - number of steps

bm.R

```
# a faster version of BM function
bm <- function (x, steps){

  # allocate enough space to hold the output matrix
  BM <- matrix(nrow = length(x), ncol=steps+1)

  # add initial point to the matrix
  BM[,1] = x

  # sample from normal distribution (delX, delY)
  z <- matrix(rnorm(steps*length(x)), nrow=length(x))

  for (i in 1:steps) BM[,i+1] <- BM[,i] + z[,i]

  return(BM)
}
```

profiling

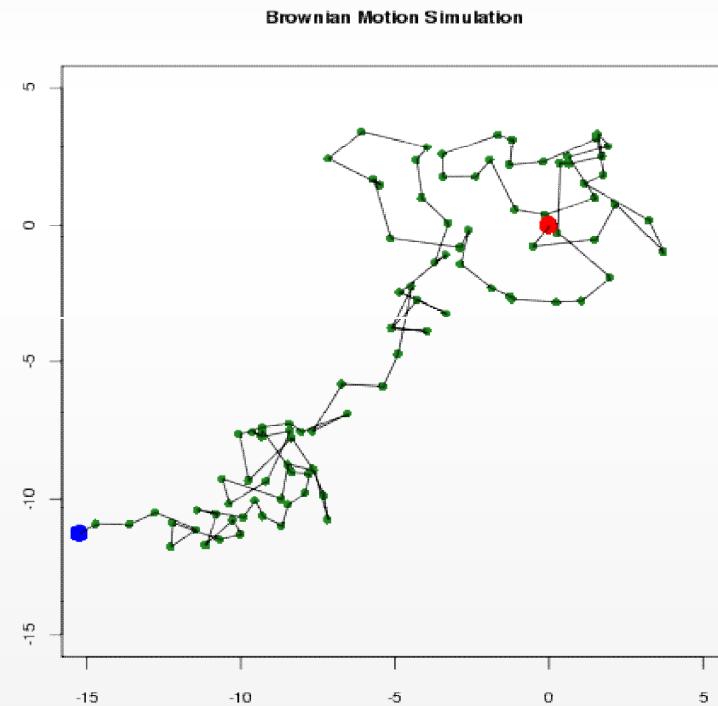
```
> # load compiler library (if you have not done it before)
> require (compiler)

> # compile function from a source file
> cmpfun ("bm.R")

> # load function from a compiled file
> loadcmp ("bm.Rc")
```

profiling

```
> # simulate 100 steps  
> BMsmall <- bm(c(0,0),100)  
  
> # plot the result  
> plot(BMsmall[1,],BMsmall[2,],...)
```



profiling

```
> # start profiling slow function  
> Rprof( "bmslow.out" ) # optional – provide output file name  
  
> # run function  
> BMS <- bmslow(c(0,0), 100000)  
  
> # finish profiling  
> Rprof(NULL)
```

profiling

```
> # start profiling faster function  
> Rprof( "bm.out" ) # optional – provide output file name  
  
> # run function  
> BM <- bm(c(0,0), 100000)  
  
> # finish profiling  
> Rprof(NULL)
```

profiling

```
> summaryRprof( "bmslow.out" )
$by.self
      self.time self.pct total.time total.pct
"cbind"     400.52    99.39     400.52    99.39
"rnorm"       1.70     0.42      1.70     0.42
"bmslow"      0.74     0.18     402.96   100.00
...
...
```

```
> summaryRprof( "bm.out" )
$by.self
      self.time self.pct total.time total.pct
"bm"        0.62    75.61      0.82   100.00
"rnorm"      0.08     9.76      0.08     9.76
"matrix"     0.04     4.88      0.12    14.63
"+"

" :"      0.04     4.88      0.04     4.88
...
...


```

Statistics in R

Lecture 9 Statistics in R, Trevor A. Branch FISH 552
Introduction to R

Outline

- Probability distributions
- Exploratory data analysis
- Comparison of two samples

Random variables (formal)

- “A random variable X is normally distributed”
- For a sample space S a **random variable** is any rule that associates a number with each outcome in S
- A random variable X is **continuous** if its set of possible values is an entire interval of numbers
- A random variable X is **discrete** if its set of possible values is a finite set or an infinite sequence
- Often we describe observations using discrete probability models (e.g. binomial, Poisson), or continuous probability models (e.g. normal, lognormal)

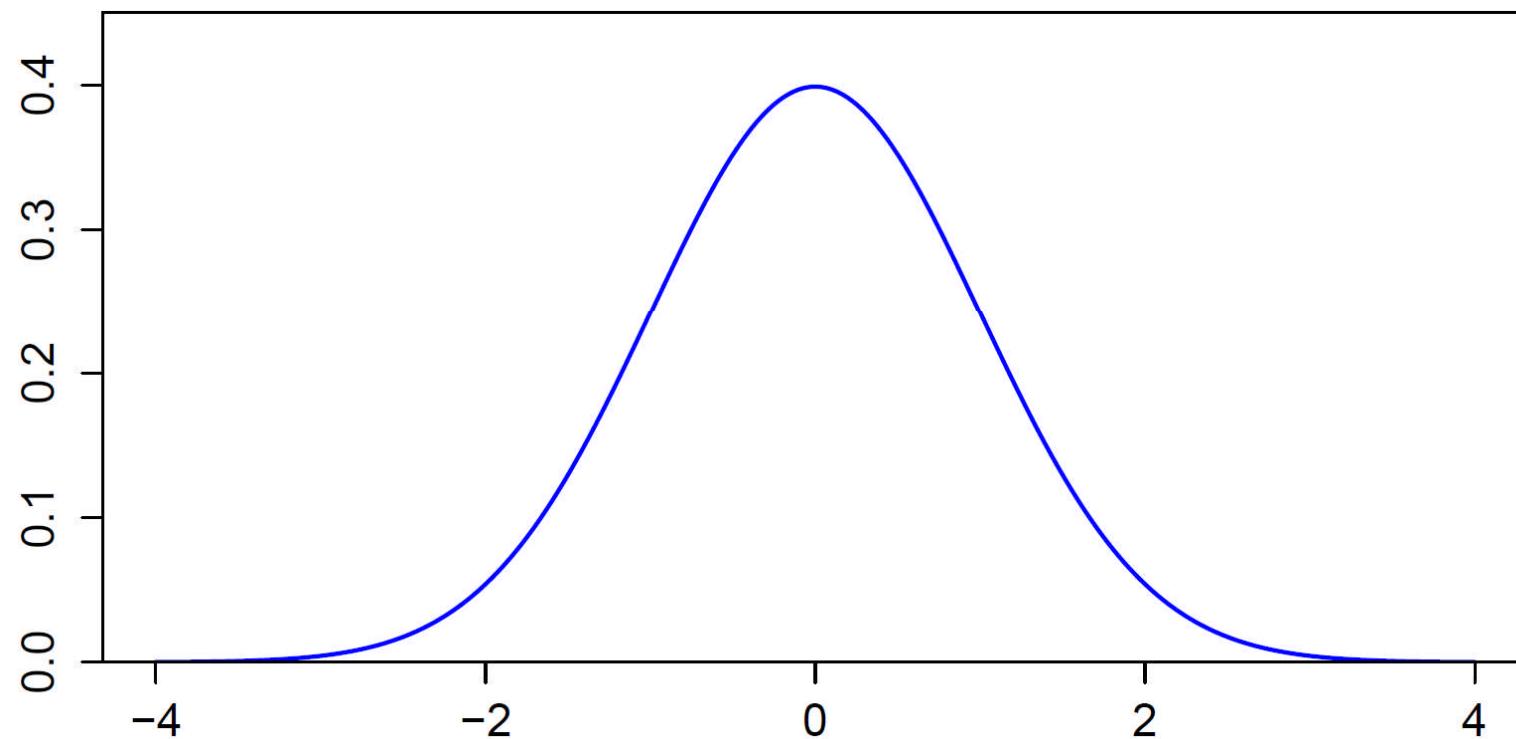
Probability distributions in R

- R includes a comprehensive set of probability distributions that can be used to simulate and model data
- If the function for the probability model is named `xxx`
 - `pxxx`: evaluate the cumulative distribution $P(X \leq x)$
 - `dxxx`: evaluate the probability mass/density function $f(x)$
 - `qxxx`: evaluate the quantile q , the smallest x such that $P(X \leq x) > q$
 - `rxxx`: generate a random variable from the model `xxx`

Probability distributions in R

Distribution	R name	Additional arguments
beta	<code>beta</code>	<code>shape1, shape2</code>
binomial	<code>binom</code>	<code>size, prob</code>
Cauchy	<code>cauchy</code>	<code>location, scale</code>
chi-squared	<code>chisq</code>	<code>df</code>
exponential	<code>exp</code>	<code>rate</code>
F	<code>f</code>	<code>df1, df2</code>
gamma	<code>gamma</code>	<code>shape, scale</code>
geometric	<code>geom</code>	<code>prob</code>
hypergeometric	<code>hyper</code>	<code>m, n, k</code>
lognormal	<code>lnorm</code>	<code>meanlog, sdlog</code>
logistic	<code>logis</code>	<code>location, scale</code>
negative binomial	<code>nbinom</code>	<code>size, prob</code>
normal	<code>norm</code>	<code>mean, sd</code>
Poisson	<code>pois</code>	<code>lambda</code>
Student's t	<code>t</code>	<code>df</code>
uniform	<code>unif</code>	<code>min, max</code>
Weibull	<code>weibull</code>	<code>shape, scale</code>
Wilcoxon	<code>wilcox</code>	<code>m, n</code>

Standard normal distribution



Functions for normal distribution

- Values of x for different quantiles

```
qnorm(p, mean=0, sd=1, lower.tail=TRUE, log.p=FALSE)
> quants <- qnorm(c(0.01,0.025,0.05,0.95,0.975,0.99))
> round(quants,2)
[1] -2.33 -1.96 -1.64 1.64 1.96 2.33
```

- Probability of observing value x or smaller

```
pnorm(q, mean=0, sd=1, lower.tail=TRUE, log.p=FALSE)
> pnorm(quants)
[1] 0.010 0.025 0.050 0.950 0.975 0.990
```

Standard normal

- Density

```
dnorm(x, mean = 0, sd = 1, log.p = FALSE)
```

```
> dnorm(quants) The height ("density") of the normal curve
```

```
[1] 0.02665214 0.05844507 0.10313564 0.10313564
```

```
[5] 0.05844507 0.02665214
```

- Generating random normal variables

```
rnorm(n, mean = 0, sd = 1)
```

```
> rnorm(n=10) Rerun and you get different values
```

```
[1] 1.91604284 0.41294905 -0.23959763 0.21590614
```

```
[5] 1.32797569 -0.19704848 -0.04746724 0.92903915
```

```
[9] 0.37813679 0.45441023
```

Random number generation

- Computers generate **pseudorandom** numbers using a sequence of specially chosen numbers and algorithms
- Each sequence of numbers starts at a **random seed** with values in `.Random.seed`
- By default the random sequence is initialized based on the start time of the program
- For repeatable pseudorandom sequences first call `set.seed(seed)` with `seed` = any integer between -2^{31} and $2^{31}-1$

Simulating data

- When simulating data or working with random numbers, it is often a good idea to use `set.seed()` and save the script detailing which number was used
- This ensures you can exactly repeat your results

```
> set.seed(100)
> rnorm(3)
[1] -0.50219235 0.13153117 -0.07891709
> rnorm(3)
[1] 0.8867848 0.1169713 0.3186301
> set.seed(100)
> rnorm(3)
[1] -0.50219235 0.13153117 -0.07891709
> rnorm(3)
[1] 0.8867848 0.1169713 0.3186301
```

The sample() function

- To generate random numbers from discrete sets of values
 - With or without replacement
 - Equal probability or weighted probability
- This is the function that underlies many modern statistical techniques
 - Resampling
 - Bootstrapping
 - Markov-chain Monte-Carlo (MCMC)

Using sample()

- Roll 10 dice

```
> sample(1:6, size=10, replace=T)
```

```
[1] 2 3 5 5 2 3 3 5 4 5
```

- Flip a coin 10 times

```
> sample(c("H","T"), size=10, replace=T)
```

```
[1] "H" "T" "T" "T" "H" "T" "T" "H" "H" "T"
```

- Pick 5 cards from a deck of cards

```
> cards <- paste(rep(c("A",2:10,"J","Q","K"),4),  
+ c("Club","Diamond","Heart","Spade"))
```

```
> sort(cards) #check that this worked!
```

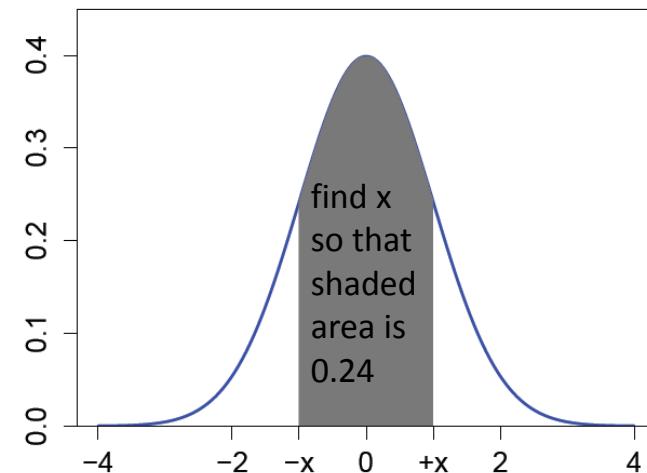
```
> sample(cards,5)
```

```
[1] "7 Club" "6 Diamond" "J Heart" "7 Spade"
```

```
[5] "K Heart"
```

Hands-on exercise 1

- Generate 100 random normal numbers with mean 80 and standard deviation 10. What proportion of these are ≥ 2 standard deviations from the mean?
- Select 6 numbers from a lottery containing 56 balls.
Go to <http://www.walottery.com/sections/WinningNumbers/>
 - Did you win?
- For a standard normal random variable, find the number x such that $P(-x \leq X \leq x) = 0.24$
(use symmetry, see diagram)



Exploratory data analysis

- Important starting point in any analysis
- Numerical summaries that quickly tell you things about your data
 - `summary()`
 - `boxplot()`
 - `fivenum()`
 - `sd()`
 - `range()`
- Visualizing your data is usually much more informative, find which built-in plot is more useful

The builtin *iris* data

```
> iris
```

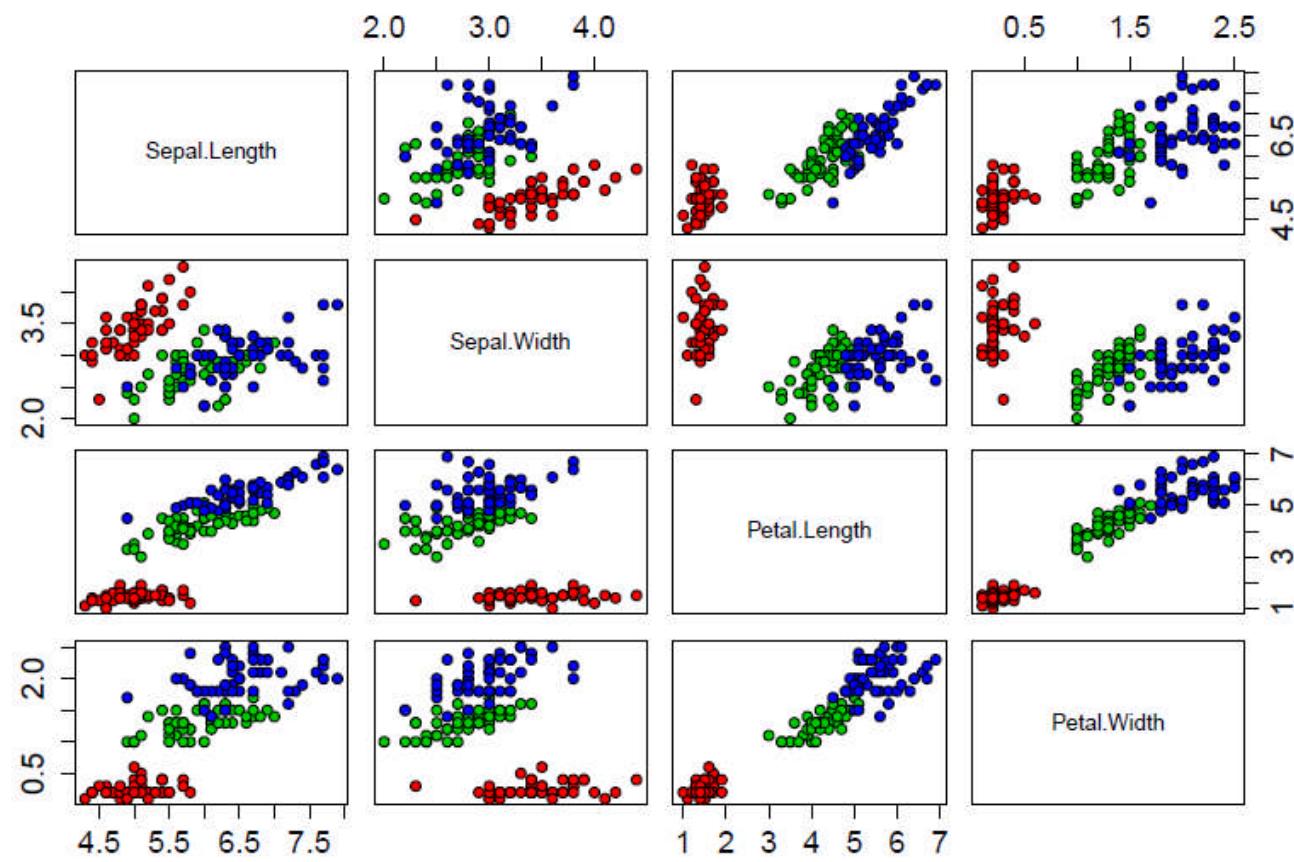
	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
...					
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
...					
101	6.3	3.3	6.0	2.5	virginica
102	5.8	2.7	5.1	1.9	virginica

pairs() function

- Quickly produces a matrix of scatterplots

```
> pairs(iris[,1:4])  
  
> pairs(iris[,1:4],  
        main = "Edgar Anderson's Iris Data",  
        pch = 21,  
        bg = rep(c("red", "green3", "blue"),  
                 table(iris$Species)))  
)
```

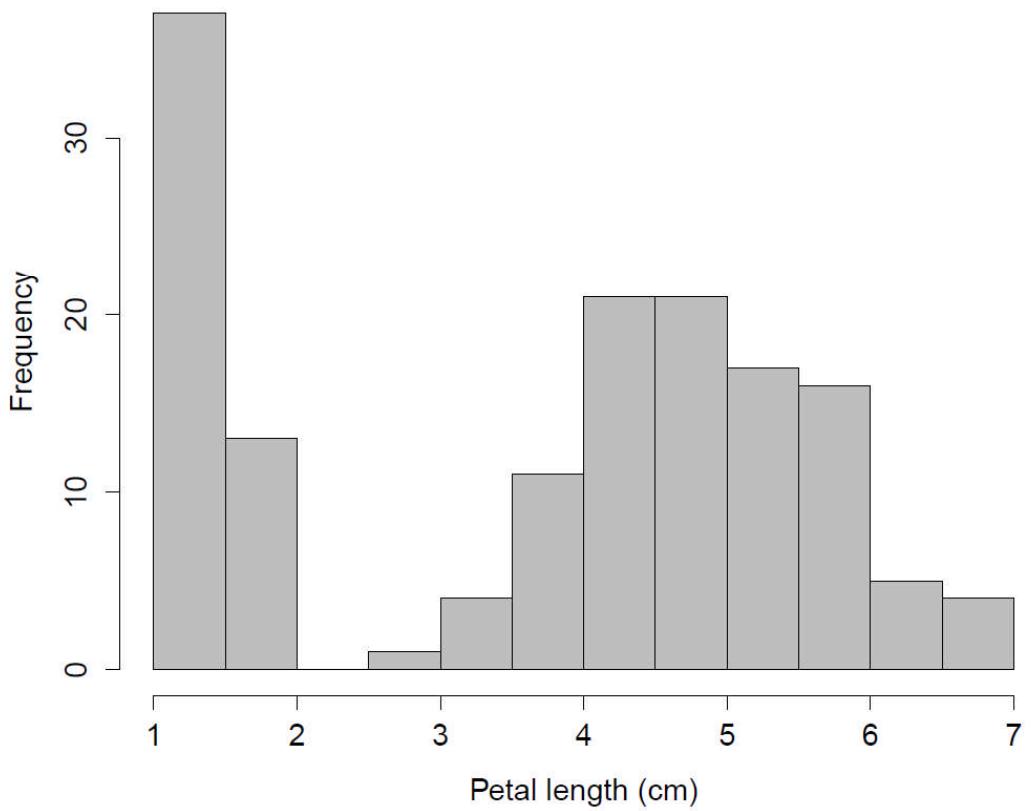
final two lines: repeats "red" 50 times, "green3" 50 times and
"blue" 50 times, for assigning these colors to the circles for each
species



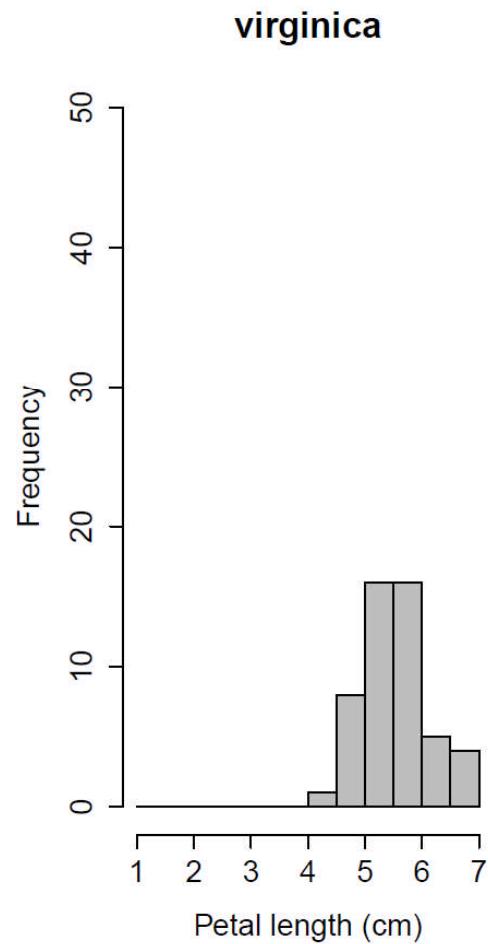
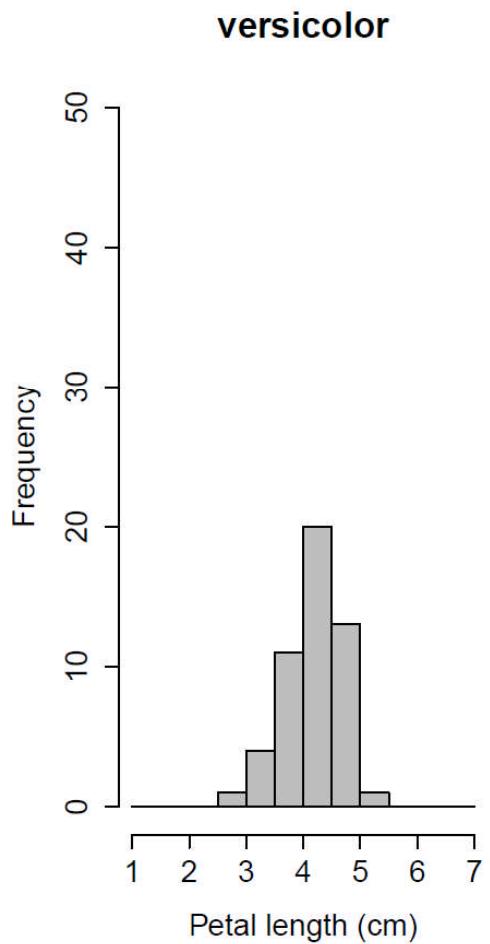
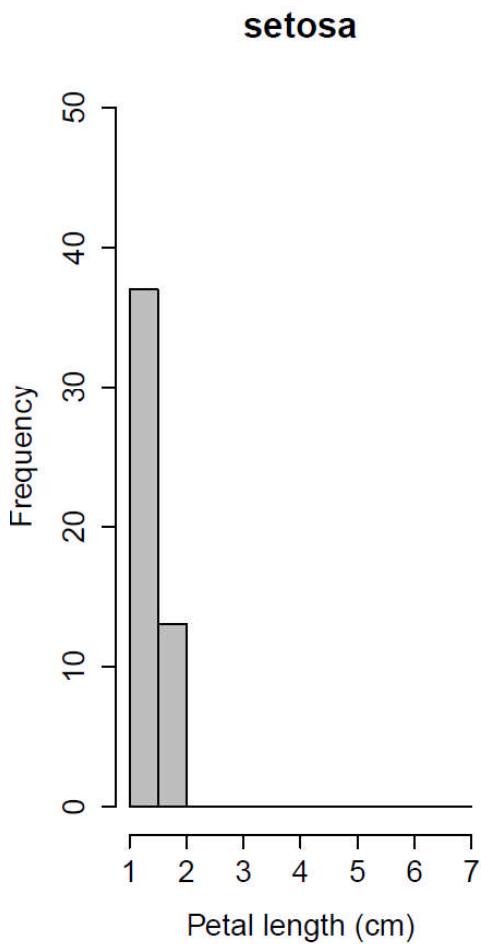
Histograms

- Quick and easy way of seeing data shape, center, skewness, and any outliers
- `?hist` (many options)
- When values are continuous, measurements are subdivided into intervals using breakpoints ("breaks")
- Many ways of specifying breakpoints
 - Vector of breaks
 - Number of breaks
 - Name of a general built-in algorithm

```
hist(iris$Petal.Length, main="", col="gray",
  xlab="Petal length (cm)")
```



Separate plots by species

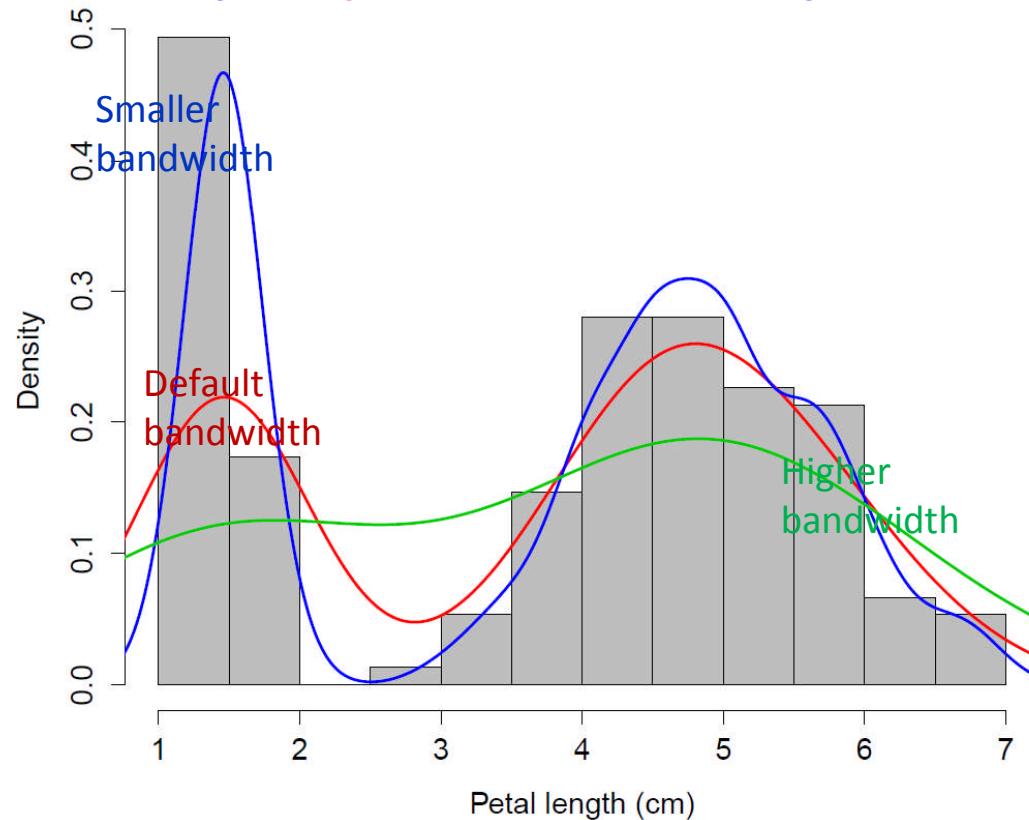


Adding density curves

- Kernel density estimation using the `density()` function is a non-parametric way of estimating the underlying probability function
- The higher the **bandwidth** the smoother the resulting curve
- Usually best to let R choose the optimal value for the bandwidth to avoid bias
- Many different kernel density methods, best to use the default for most purposes

Plotting Iris data

```
hist(iris$Petal.Length, main="", col="gray",
     xlab="Petal length (cm)", freq=F)
lines(density(iris$Petal.Length), lwd=2, col="red")
lines(density(iris$Petal.Length, adjust=0.4), lwd=2, col="blue")
lines(density(iris$Petal.Length, adjust=2), lwd=2, col="green3")
```

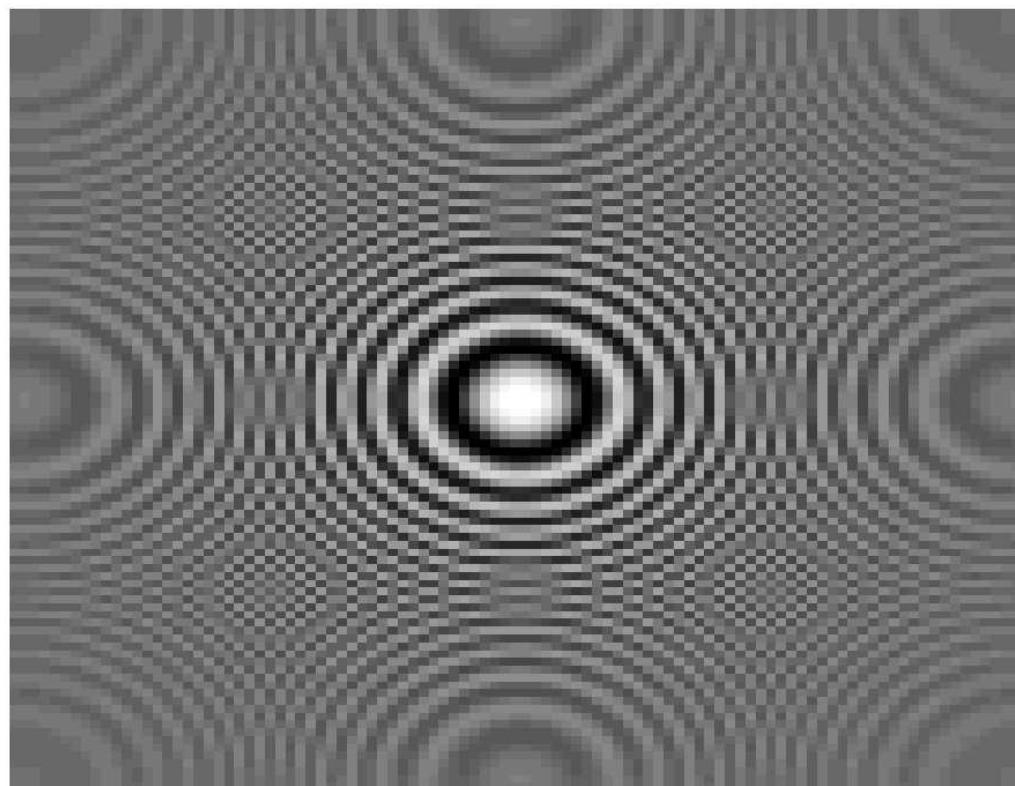


Other useful plots

Plot	R function
Barplot	<code>barplot()</code>
Contour lines of two-dimensional distribution	<code>contour()</code>
Plot of two variables conditioned on the other	<code>coplot()</code>
Represent 3-D using shaded squares	<code>image()</code>
Dotchart	<code>dotchart()</code>
Pie chart	<code>pie()</code>
3-dimensional surface plot	<code>persp()</code>
Quantile-quantile plot	<code>qqplot()</code>
Stripchart	<code>stripchart()</code>

Grey Shading image

Created using `image()`: squares each with a different gray shading



Basic statistical tests in R

- R has a bewildering array of built-in functions to perform classical statistical tests
 - Correlation `cor.test()`
 - Chi-squared `chisq.test()`
- In the next lecture
 - ANOVA `anova()`
 - Linear models `lm()`
- And many many more...

Comparison of two samples

- The t-test examines whether two population means, with unknown population variances, are significantly different from each other

$$H_0: \mu_1 - \mu_2 = 0$$

$$H_1: \mu_1 - \mu_2 \neq 0$$

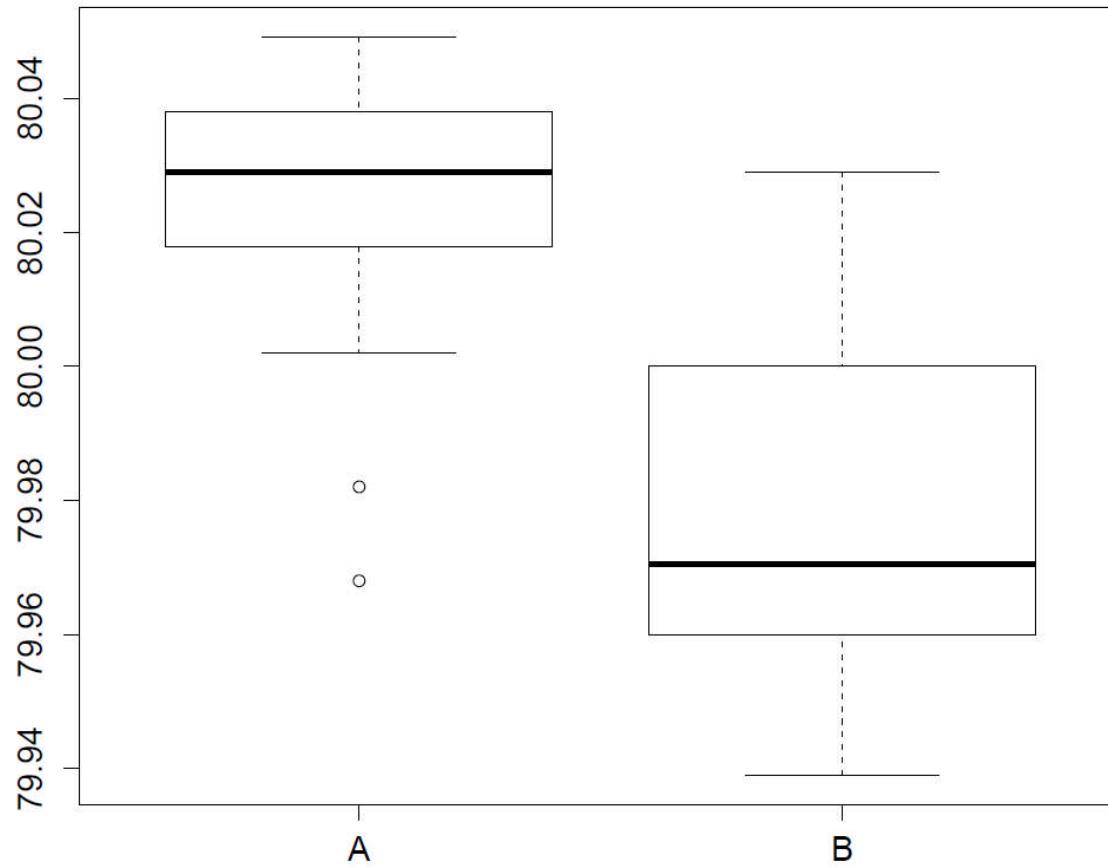
- The two-sample independent t-test assumes
 - Populations are normally distributed (not an issue for large sample sizes)
 - Equal variances (assumption can be relaxed)
 - Independent samples

Using `t.test()`

- The `t.test()` function in R can be used to perform many variants of the t-test
 - `?t.test`
 - Specify one- or two-tailed
 - Specify μ
 - Specify significance level α
- Example: two methods were used to determine the latent heat of ice. The investigator wants to find out how much (and if) the methods differed

```
methodA <- c(79.982, 80.041, 80.018, 80.041, 80.03, 80.029,  
80.038, 79.968, 80.049, 80.029, 80.019, 80.002, 80.022)  
methodB <- c(80.02, 79.939, 79.98, 79.971, 79.97, 80.029, 79.952,  
79.968)
```

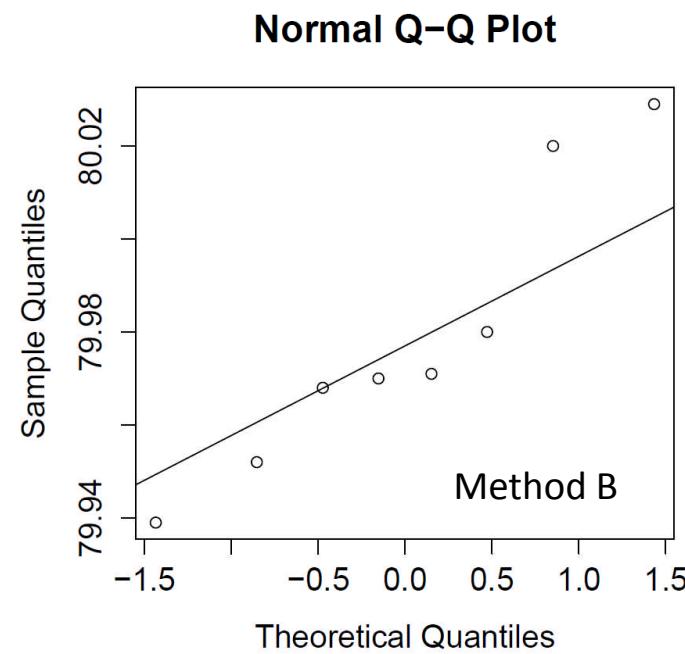
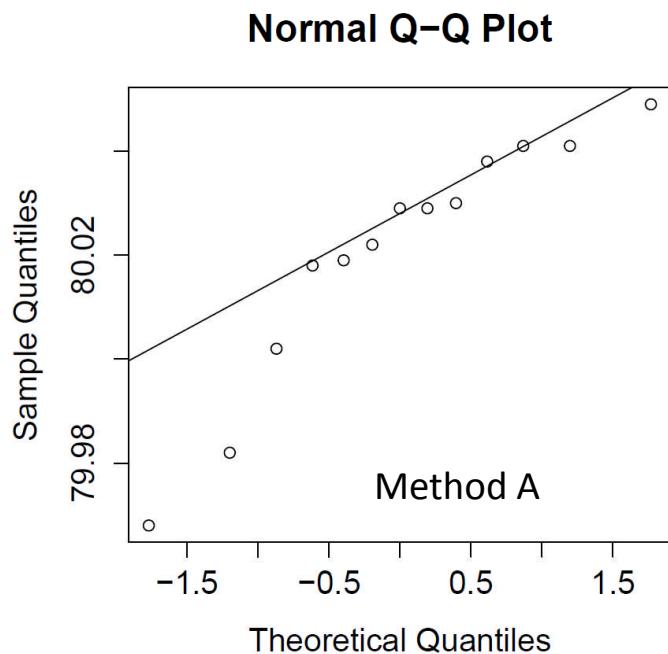
Rule 1: plot the data



Data normally distributed?

If normal, points should fall on the line of a `qqnorm()` plot. Here A has a strong left skew, B a right skew

`qqnorm(methodA); qqline(methodA)`



t-test

- The default in R is the Welch two-sample t-test which assumes unequal variances

```
> t.test(methodA, methodB)
Welch Two Sample t-test
data: methodA and methodB
t = 3.274, df = 12.03, p-value = 0.006633
alternative hypothesis: true difference in means is
not equal to 0
95 percent confidence interval:
0.01405393 0.06992684
sample estimates:
mean of x mean of y
80.02062 79.97862
```

Accessing output

- The results of a statistical test can be saved by assigning a name to the output and accessing individual elements
- The results are stored in a list (actually an object)

```
> resultsAB <- t.test(methodA, methodB)
> names(resultsAB)
[1] "statistic"    "parameter"    "p.value"
[4] "conf.int"     "estimate"     "null.value"
[7] "alternative" "method"       "data.name"
> resultsAB$p.value
[1] 0.006633411
```

Equal variances?

- The boxplot suggests that the variances of the two methods might be similar

```
> var(methodA)  
[1] 0.0005654231  
> var(methodB)  
[1] 0.0009679821
```

- More formally we can perform an F-test to check for equality in variances

F-test

- If we take two samples n_1 and n_2 from normally distributed populations, then the ratio of the variances is from an F-distribution with degrees of freedom $(n_1-1; n_2-1)$

$$H_0: \sigma^2_1 / \sigma^2_2 = 1$$

$$H_1: \sigma^2_1 / \sigma^2_2 \neq 1$$

Use var.test() in R

```
> var.test(methodA, methodB)
F test to compare two variances
data: methodA and methodB
F = 0.5841, num df = 12, denom df = 7, p-value =
0.3943
alternative hypothesis: true ratio of variances is
not equal to 1
95 percent confidence interval:
0.1251922 2.1066573
sample estimates:
ratio of variances
0.5841255
```

There is no evidence of a significant difference between the two variances

Redo t-test with equal variances

```
> t.test(methodA, methodB, var.equal = TRUE)
Two Sample t-test
data: methodA and methodB      (Previously p = 0.006633)
t = 3.4977, df = 19, p-value = 0.002408
alternative hypothesis: true difference in means is
not equal to 0
95 percent confidence interval:
 0.01686368 0.06711709
sample estimates:
mean of x mean of y
80.02062 79.97862
```

Non-parametric tests

- We showed earlier using `qqplot()` that the normality assumption is violated, particularly an issue since the sample sizes here were small
- The two-sample Wilcoxon (Mann-Whitney) test is a useful alternative when the assumptions of the t-test are not met
- Non-parametric tests rely on the rank order of the data instead of the numbers in the data itself

Mann-Whitney test

- How the test works
 - Arrange all the observations into a ranked (ordered) series
 - Sum the ranks from sample 1 (R_1) and sample 2 (R_2)
 - Calculate the test statistic U

$$U = \max \left\{ R_1 - \frac{n_1(n_1+1)}{2}, n_1n_2 - \left(R_1 - \frac{n_1(n_1+1)}{2} \right) \right\}$$

- The distribution under H_0 is found by enumerating all possible subsets of ranks (assuming each is equally likely), and comparing the test statistic to the probability of observing that rank
- This can be cumbersome for large sample sizes

Caveats

- When there are ties in the data, this method provides an approximate p-value
- If the sample size is less than 50 and there are no ties in observations, by default R will enumerate all possible combinations and produce an exact p-value
- When the sample size is greater than 50, a normal approximation is used
- The function to use is `wilcox.test()`
- `?wilcox.test`

Results of Mann-Whitney test

```
> wilcox.test(methodA, methodB)
wilcoxon rank sum test with continuity correction
data: methodA and methodB
W = 88.5, p-value = 0.008995
alternative hypothesis: true location shift is not
equal to 0
warning message:
In wilcox.test.default(methodA, methodB) :
  cannot compute exact p-value with ties
```

Once again we reach the same conclusion

Hands-on exercise 2

- Create 10 qqnorm plots (including qqline) by sampling 30 points from the following four distributions
 - normal, exponential, t, cauchy
- Make between 2 and 5 of the plots come from a normal distribution
- Have your partner guess which plots are actually from a normal distribution

Hypothesis Testing with R

7/10/2014.

from <http://www.math.csi.cuny.edu/Statistics/R/simpleR/stat011.html>

Introduction

- You make a hypothesis about the value of the unknown parameter and then calculate how likely it is that you observed the data or worse.

from <http://www.math.csi.cuny.edu/Statistics/R/simpleR/stat011.html>

Problem 1.1

- Consider a simple survey. You ask 100 people (randomly chosen) and 42 say “yes” to your question.
- Does this support the hypothesis that the true proportion is 50%.
- To answer this, we set up a *test of hypothesis*.

from <http://www.math.csi.cuny.edu/Statistics/R/simpleR/stat011.html>

Problem 1

- The *null hypothesis*, denoted H_0 is that $p=.5$,
- the *alternative hypothesis*, denoted H_A , in this example would be $p \neq 0.5$ (not).
- This is a so called *two-sided* alternative.
- To test the assumptions, we use the function *prop.test* as with the confidence interval calculation.

from <http://www.math.csi.cuny.edu/Statistics/R/simpleR/stat011.html>

Problem 1

```
> prop.test( 42,100, p=.5)
```

1-sample proportions test with continuity correction

data: 42 out of 100, null probability 0.5

X-squared = 2.25, df = 1, p-value = 0.1336

alternative hypothesis: true p is not equal to 0.5

95 percent confidence interval:

0.3233236 0.5228954

sample estimates: p 0.42

from <http://www.math.csi.cuny.edu/Statistics/R/simpleR/stat011.html>

P-value in problem 1

- The *p*-value reports how likely we are to see this data *or worse* assuming the *null hypothesis*.
- The notion of worse, is implied by the *alternative hypothesis*.
- In this example, the alternative is *two-sided* as too small a value or too large a value or the test statistic is consistent with H_A .
- In particular, the *p*-value is the probability of 42 or fewer *or* 58 or more answer *yes* when the chance a person will answer *yes* is fifty-fifty.

Problem 1.2

- Now, the p -value is not so small as to make an observation of 42 seem unreasonable in 100 samples assuming the null hypothesis. Thus, one would ‘accept’ the null hypothesis.
- Next, we repeat, only suppose we ask 1000 people and 420 say yes. Does this still support the null hypothesis that $p=0.5$?

Problem 1.2

```
> prop.test(420,1000,p=.5)
1-sample proportions test with continuity correction
data: 420 out of 1000, null probability 0.5
X-squared = 25.281, df = 1,
p-value = 4.956e-07
alternative hypothesis: true p is not equal to 0.5
95% confidence interval: 0.3892796 ... 0.4513427
sample estimates: p 0.42
```

p value in problem 1.2

Now the p -value is tiny (that's 0.0000004956) and the null hypothesis is not supported.

That is, we "*reject*" the null hypothesis. This illustrates the p value depends both on the ratio and also n .

In particular, it is because the standard error of the sample average gets smaller as n gets larger.

Problem 1.3

- Suppose a car manufacturer claims a model gets 25 mpg.
- A consumer group asks 10 owners of this model to calculate their mpg and the mean value was 22mpg with a standard deviation of 1.5
- Is the manufacturer's claim supported?

Problem 1.3

- In this case $H_0: \mu=25$ against the one-sided alternative hypothesis that $\mu<25$.
- To test using R we simply need to tell R about the type of test. (As well, we need to convince ourselves that the *t*-test is appropriate for the underlying parent population.)
- For this example, the builtin R function *t.test* isn't going to work -- the data is already summarized -- so we are on our own.

Compute the t-statistic

```
# assume mu=25 under H_0  
> xbar=22; s=1.5; n=10  
> t = (xbar-25)/(s/sqrt(n))  
# t is -6.324555  
## pt is the distribution function of t  
> pt( t, df = n-1 )  
6.846828e-05
```

Problem 1.3

- We calculate the test statistic and then find the p -value, using `pt` function.
- This is a small p -value (0.000068).
- The manufacturer's claim is suspicious.

Problem 1.4

- Suppose a study of cell-phone usage for a user gives the following lengths for the calls:
- 12.8 3.5 2.9 9.4 8.7 .7 .2 2.8
1.9 2.8 3.1 15.8
- What is an appropriate test for center?

Problem 1.4

- The distribution looks skewed with a possibly heavy tail.
- A t -test is ruled out.
- Instead, a test for the median is done.
- Suppose H_0 is that the median is 5, and the alternative is the median is bigger than 5.
- To test this with R we can use the `wilcox.test` as follows

Problem 1.4 median

```
> wilcox.test(x,mu=5,alt="greater")
```

Wilcoxon signed rank test with continuity correction

- data: x, $V = 39$, p-value = 0.5156
- alternative hypothesis: true mu is greater than 5
- Warning message: Cannot compute exact p-value with ties ...
- Note the p value is not small, so the null hypothesis is not rejected.

References

- Simple R, by Verzani from
<http://www.math.csi.cuny.edu/Statistics/R/simpleR/stat011.html>

Factor Analysis with R

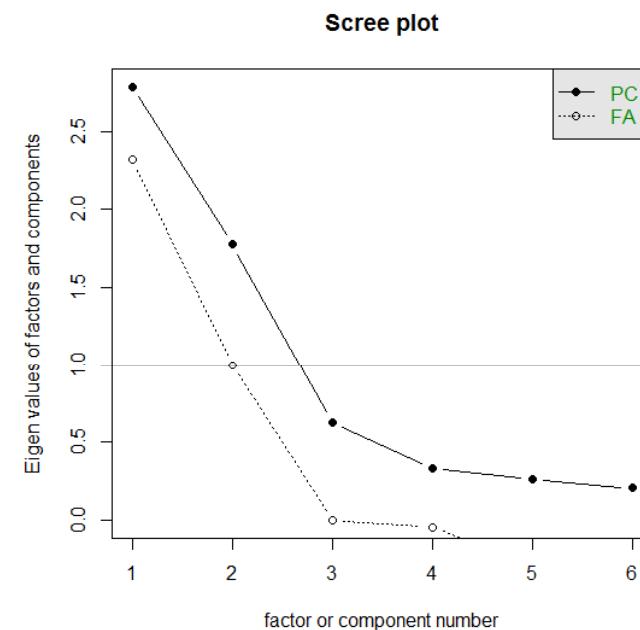
Load the packages

```
# packages for factor analysis.  
install.packages("psych")  
library(psych)
```

```
# Get help  
?scree  
?fa
```

Factor analysis

```
# data <- read.csv("5048-factor-r.csv")  
  
> data <- read.csv(  
+   file.choose())  
  
# Make sure you got  
# the right data.  
  
> head(data, n=2L)  
  
  BIO GEO CHEM ALG CALC STAT  
1  1    1    1    1    1    1  
2  4    4    3    4    4    4  
  
> scree(data)
```



"scree" means debris at
the bottom of a cliff

Method 1. fa output

```
cm = cor(data)
```

```
sol <- fa(r=cm); sol
```

```
Factor Analysis using method = minres
```

```
Call: fa(r = cm)
```

```
Standardized loadings (pattern matrix) based upon  
correlation matrix
```

	MR1	h2	u2	com
BIO	0.86	0.731	0.27	1
GEO	0.80	0.636	0.36	1
CHEM	0.83	0.693	0.31	1
ALG	0.20	0.039	0.96	1
CALC	0.28	0.079	0.92	1
STAT	0.28	0.080	0.92	1

Method 2. factanal output

```
> n.factors <- 2
> fit <- factanal(data, n.factors, scores=c("regression"), rotation="none")
> print(fit, digits=2, cutoff=.3, sort=TRUE)
Call: factanal(x = data, factors = n.factors,
               scores = c("regression"), rotation = "none")
```

Uniquenesses: BIO GEO CHEM ALG CALC STAT
0.25 0.37 0.25 0.37 0.05 0.71

Loadings:

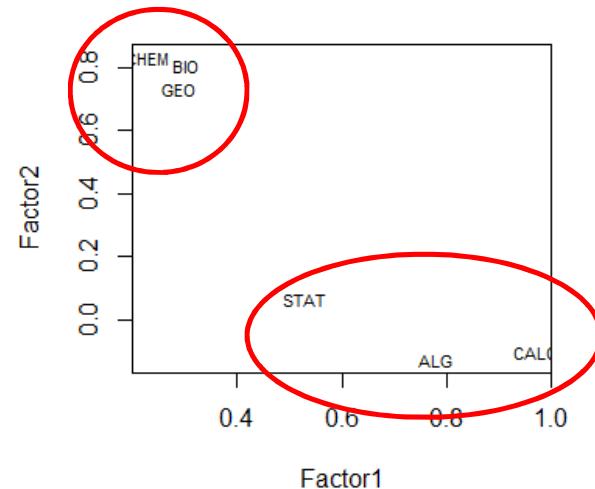
	Factor1	Factor2
ALG	0.78	
CALC	0.97	
STAT	0.53	
BIO	0.30	0.81
GEO		0.74
CHEM		0.84

	Factor1	Factor2
SS loadings	2.06	1.93
Proportion Var	0.34	0.32
Cumulative Var	0.34	0.66

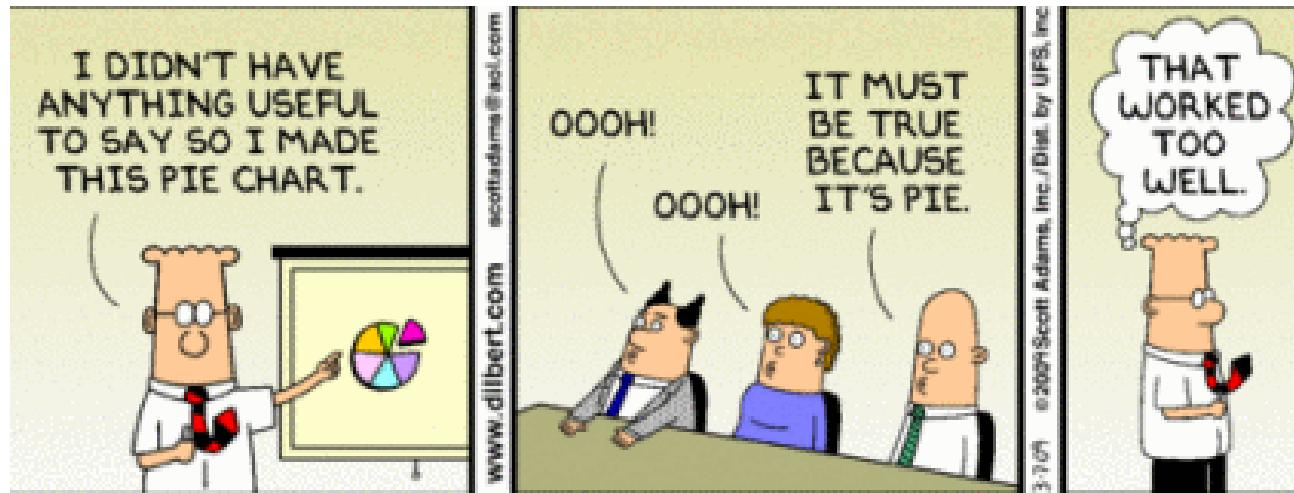
Test of the hypothesis that 2 factors are sufficient.
The chi square statistic is 2.94 on 4 degrees of freedom.
The p-value is 0.568

Plot factor 1 by factor 2

```
head(fit$scores) # see the scores  
load <- fit$loadings[,1:2]  
plot(load,type="n")  
text(load,labels=names(data),cex=.7)
```



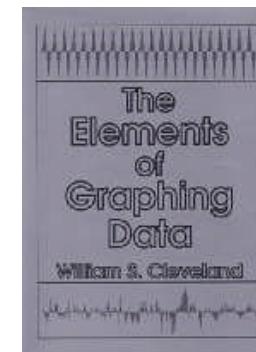
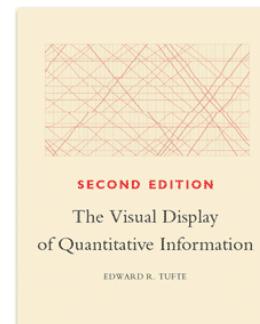
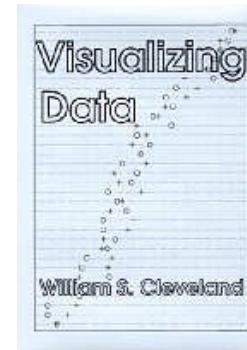
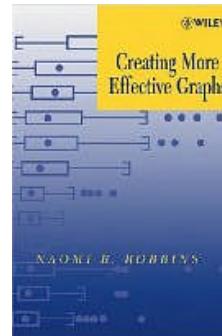
Good, Bad and Ugly Graphics



4/2/2016.

Book: Robbins

1. *Creating More Effective Graphics*
by Naomi Robbins.
2. William Cleveland
3. Tufte (2001) *The Visual Display of Quantitative Information.*

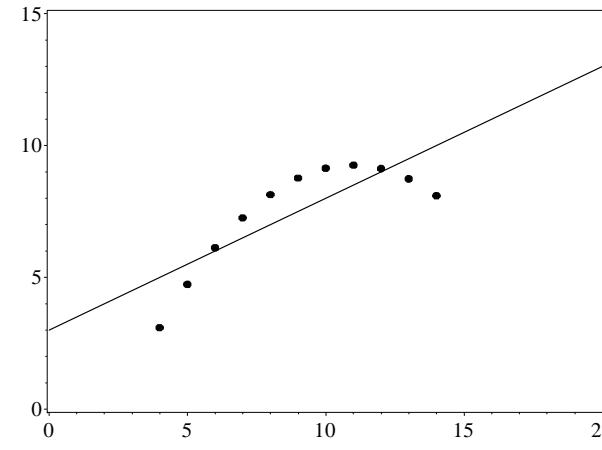
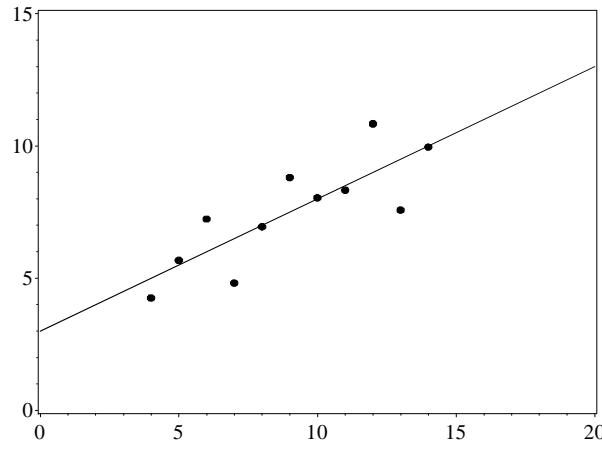


Why Do Data Visualization?

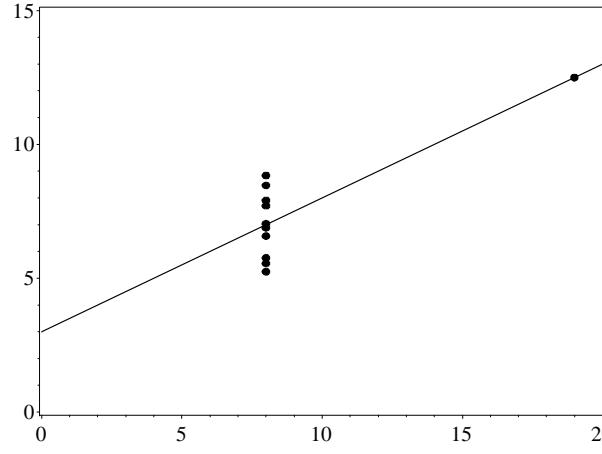
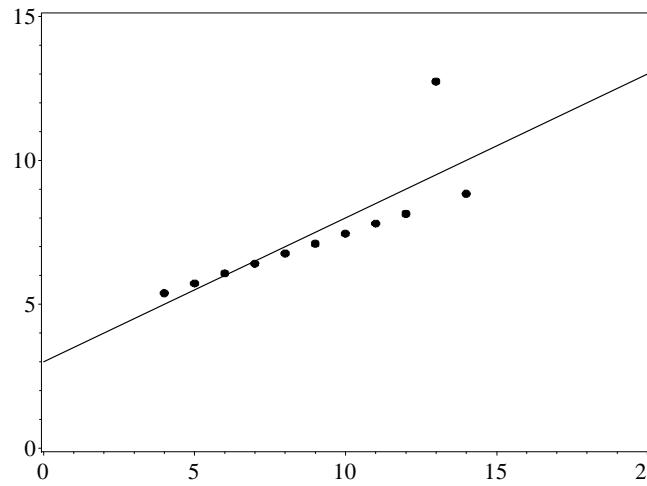
1. Well designed pictures will show you the details and the whole pattern in your data.
2. Numeric descriptions can easily hide important patterns.
3. Some patterns are hard to detect in tables.
 1. Whenever data is reported over time or locations, you need art.

You can learn A lot by just looking -Yogi Berra

Scatter Plot for Correlations



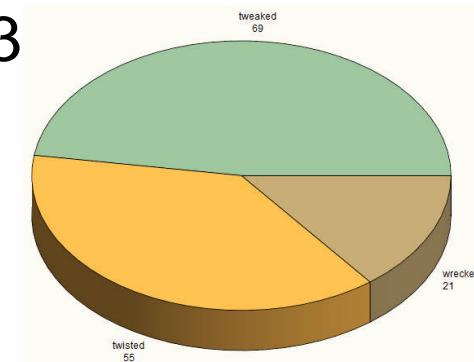
All have $r^2 = .67$



Anscombe 1973, Graphs in Statistical Analysis

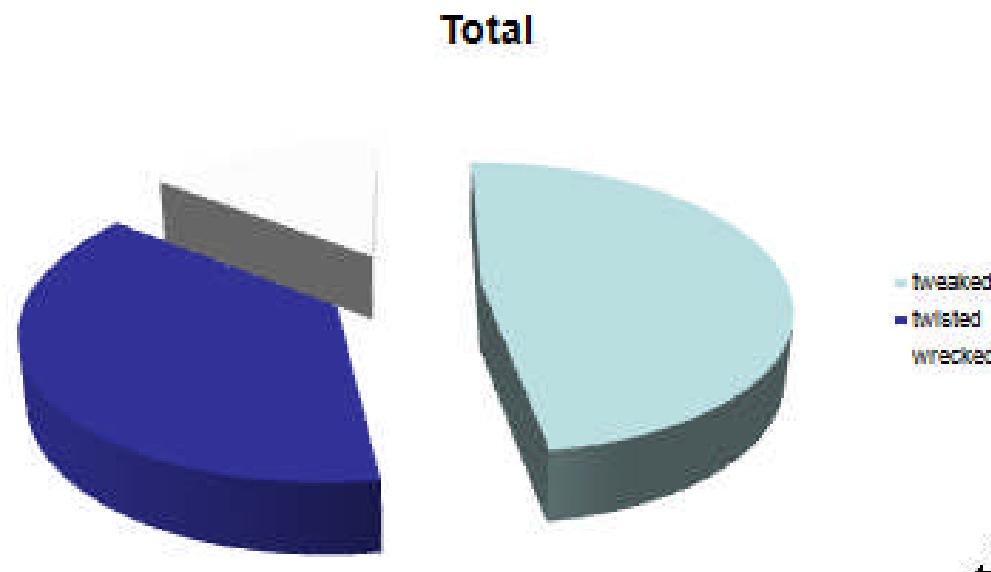
Pie is bad.

- Work by Cleveland (and experimental psychologists) suggests that:
 - people are bad at judging the relative magnitude of angles
 - if you twist the rotation of the pie you can cause people to systematically misjudge the size of the angles
 - a 3rd dimension makes judgment worse
- If you get a glossy handout with a 3 someone is lying to you.
- Don't use Pie charts.



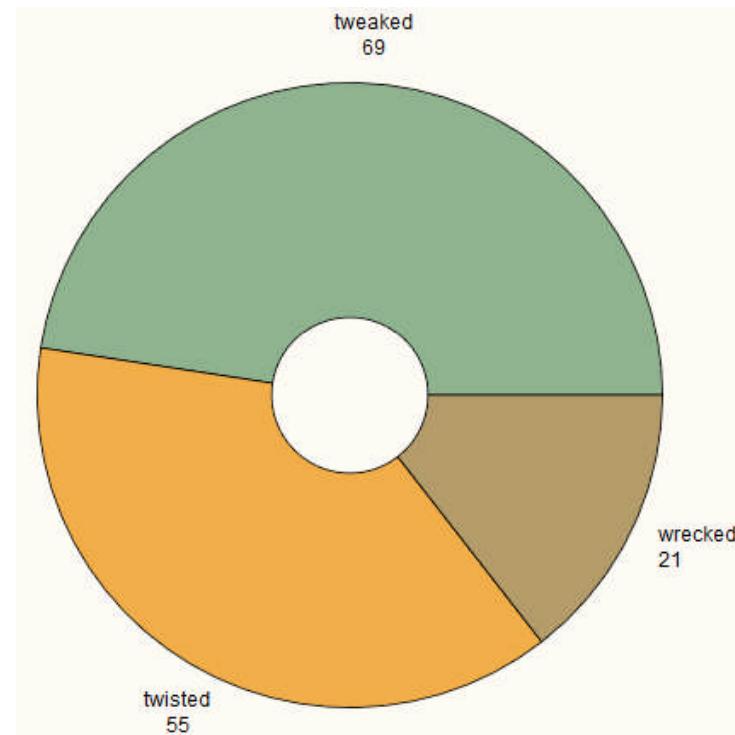
Don't Explode Pie Charts

- This exploded 3D pie (brought to you by Excel) is nearly useless for judging amounts.



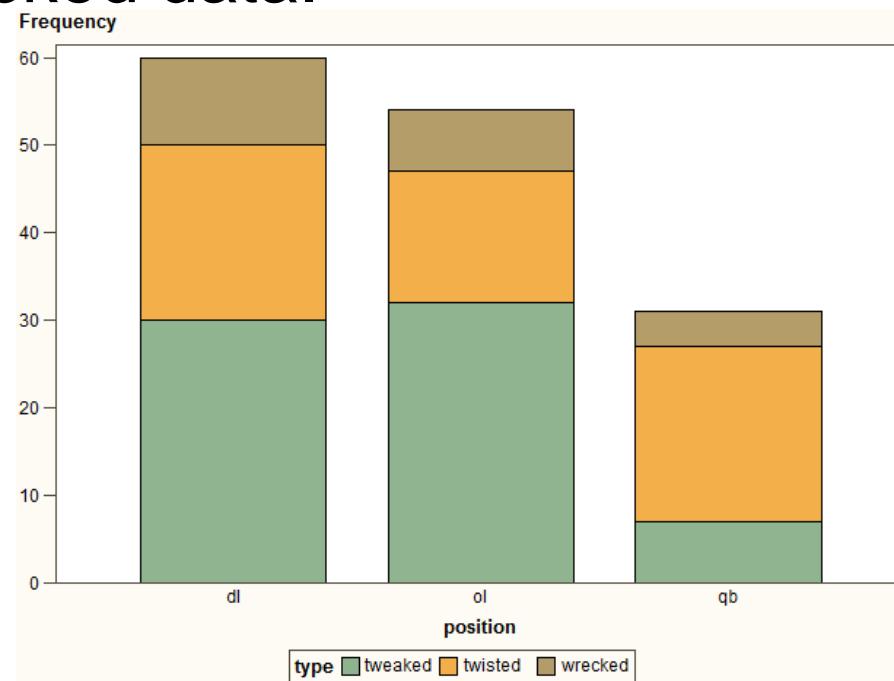
Forbidden Donut....

- Donut plots have the same problems as Pie charts (if not worse)



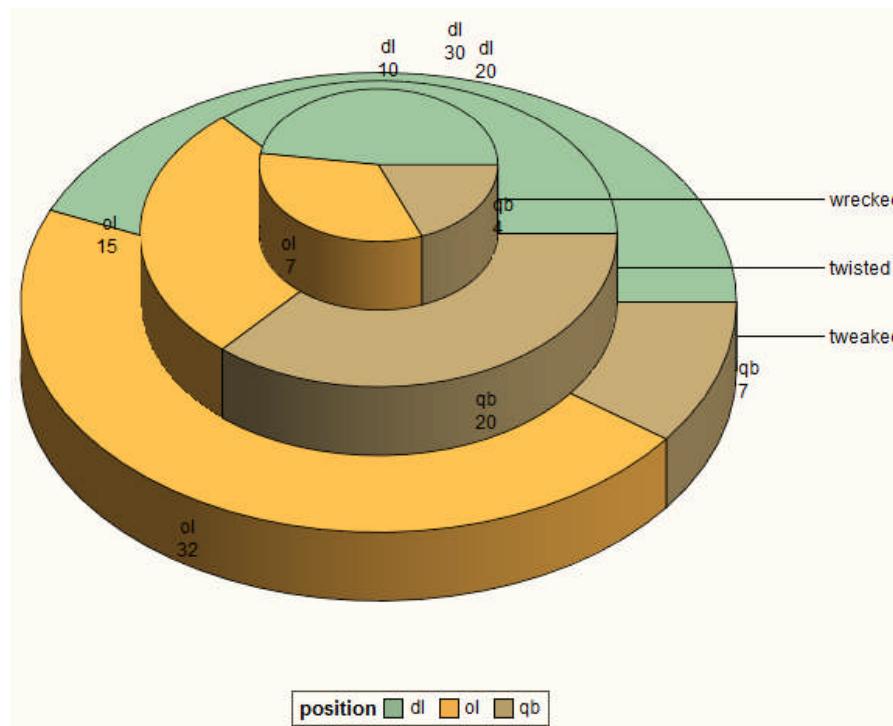
Stacking is Bad

- Cleveland also quantified the fact that people are bad at judging the relative height of stacked data.



Wow, a cinnamon roll plot!

- Good luck making rapid judgments using this stacked 3D pie.



What is a good graphic?

- Don't make your audience think unnecessarily!
- Minimize the amount of ink on the page.
 - This needs to be studied.
- Show the central tendency and the *variability*.
- Plot the quantity (inference) that you want people to notice.
- Be sure colorblind people can understand it.
 - Use a black and white photocopier and make sure you can distinguish all groups.

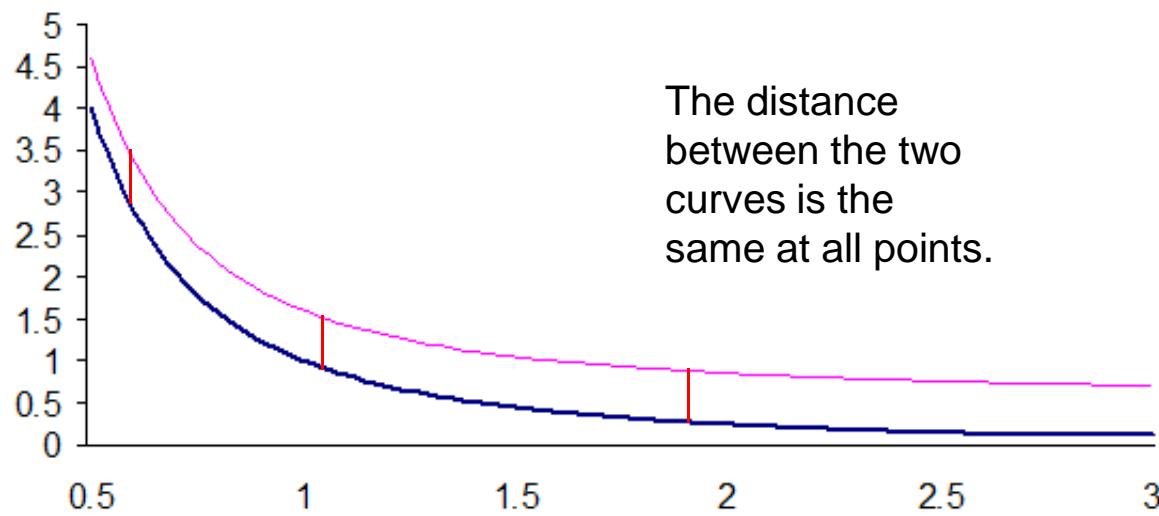
Avoid Thinking

- Put labels on the graphic directly instead of using a key.
- If you want people to compare the difference between two lines, plot the difference, not the two lines.

Bivariate Comparisons with Lines

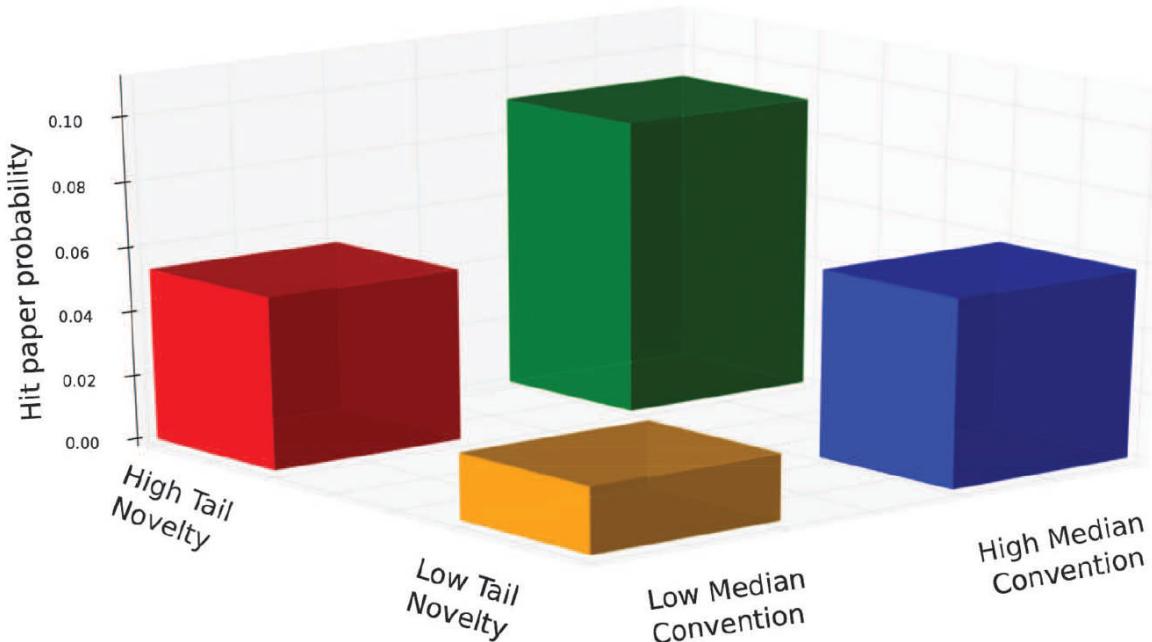
- People are extremely bad at judging the distance between two curves. Never ask people to judge up and down (vertical) distances between curves.

Based on:
Robbins
Creating More
Effective
Graphs, 2005



Data density index

The number of numbers plotted per cm² (Tufte)



DDI = 0.05 for
Fig. 2 of Uzzi
et al. (2013)

High numbers
are better

Commonly
ranges from
0.1 to >300

Figure: Uzzi et al. (2013) Atypical combinations and scientific impact. Science 342:468-472
Tufte ER (2001) The visual display of quantitative information

Data / Ink Ratio

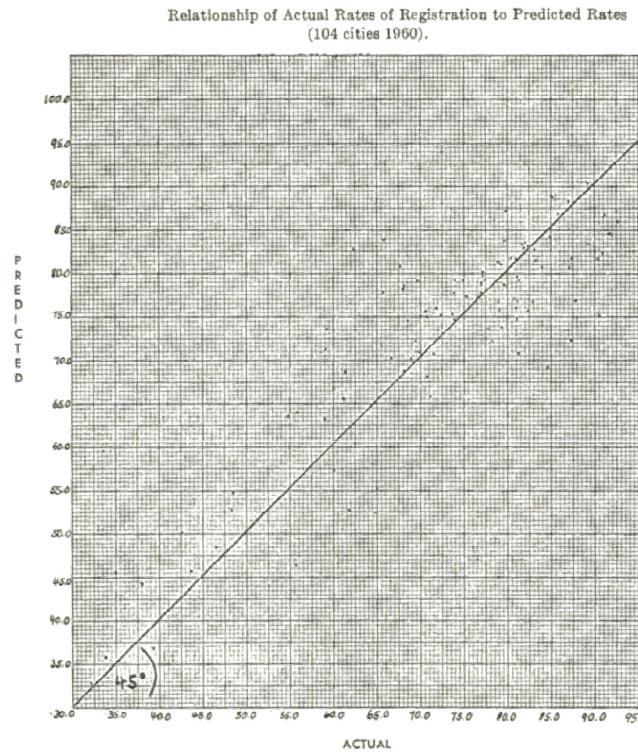
Data-ink ratio = $\frac{\text{data ink}}{\text{total ink used to print graphic}}$

= proportion of ink devoted to non-redundant
display of data-information

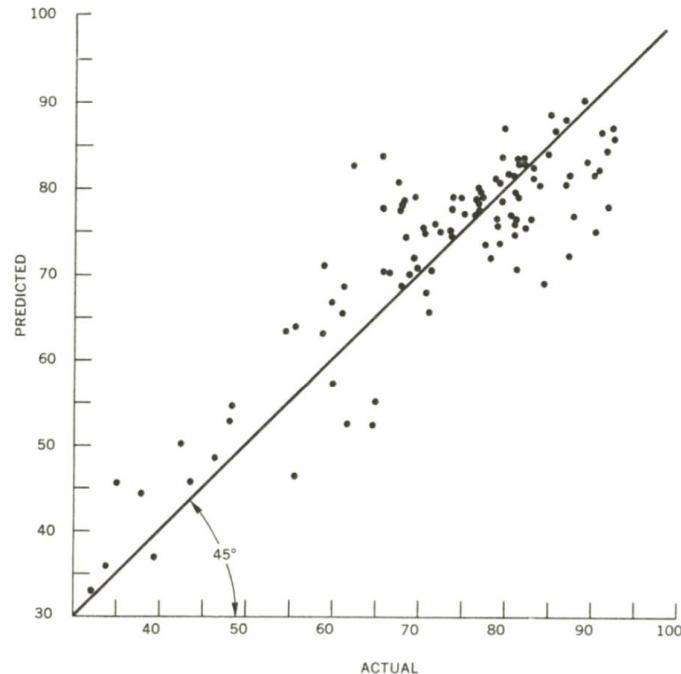
= 1.0 – proportion of graphic that can be erased
without loss of information

Tufte (2001) The visual display of quantitative information, p. 93-95

Which is easier to read?



Kelley & Bowen (1967) American Political Science Review, 61:371

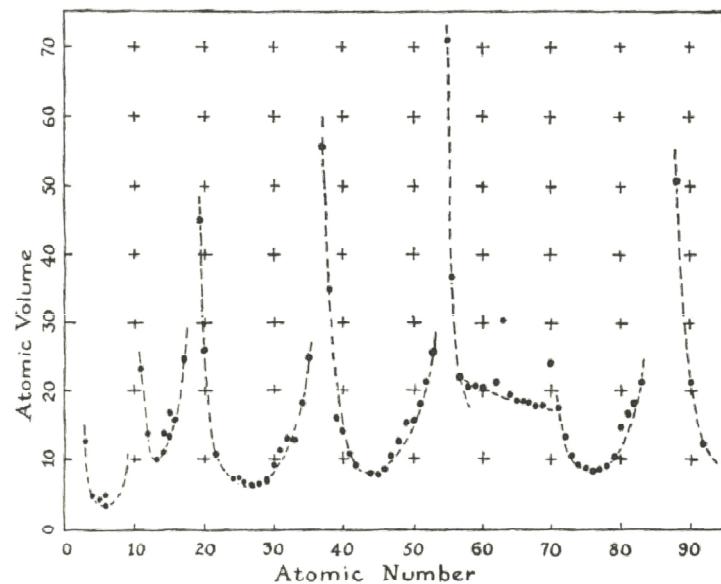


Relationship of Actual Rates of Registration to Predicted Rates (104 cities 1960).

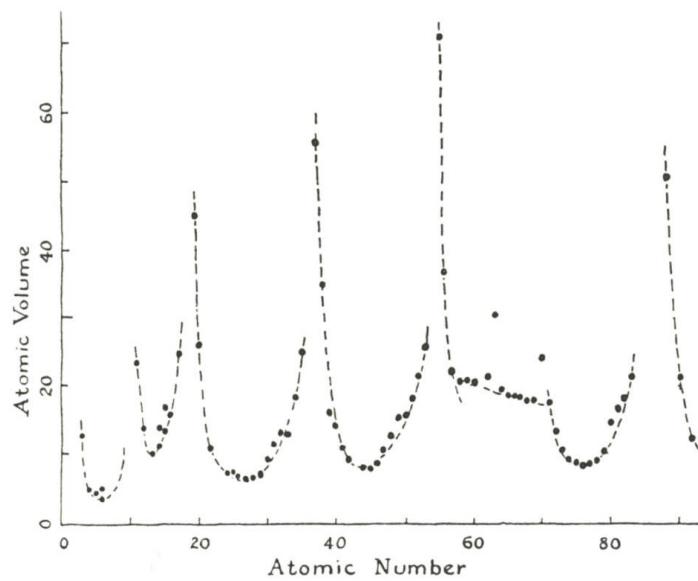
Tufte (2001) The visual display of quantitative information, p. 94-95

Data/Ink Ratio Examples

Data-ink ratio: < 0.6



Data-ink ratio: 0.9

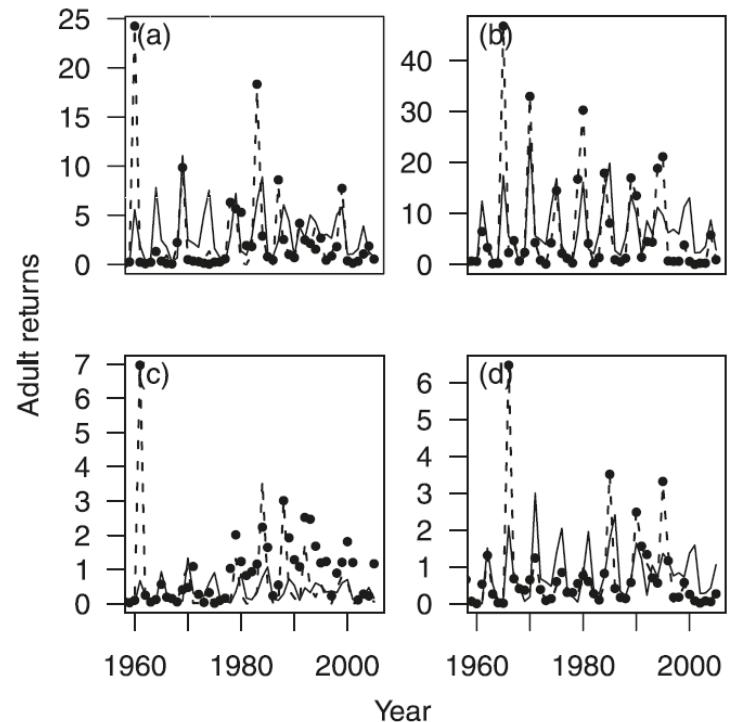


Pauling (1947) General chemistry, San Francisco, p. 64

Tufte (2001) The visual display of quantitative information, p. 102-105

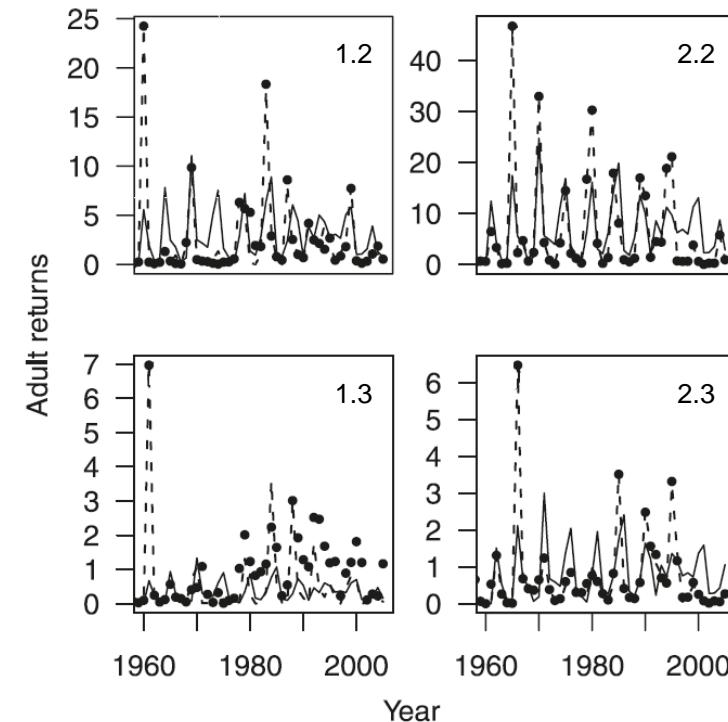
Legends hard to read, Label the Figures

Panels corresponds to adult returns for (fresh-water years.ocean years) (a) 1.2, (b) 2.2, (c) 1.3, and (d) 2.3.



CJFAS-mandated style

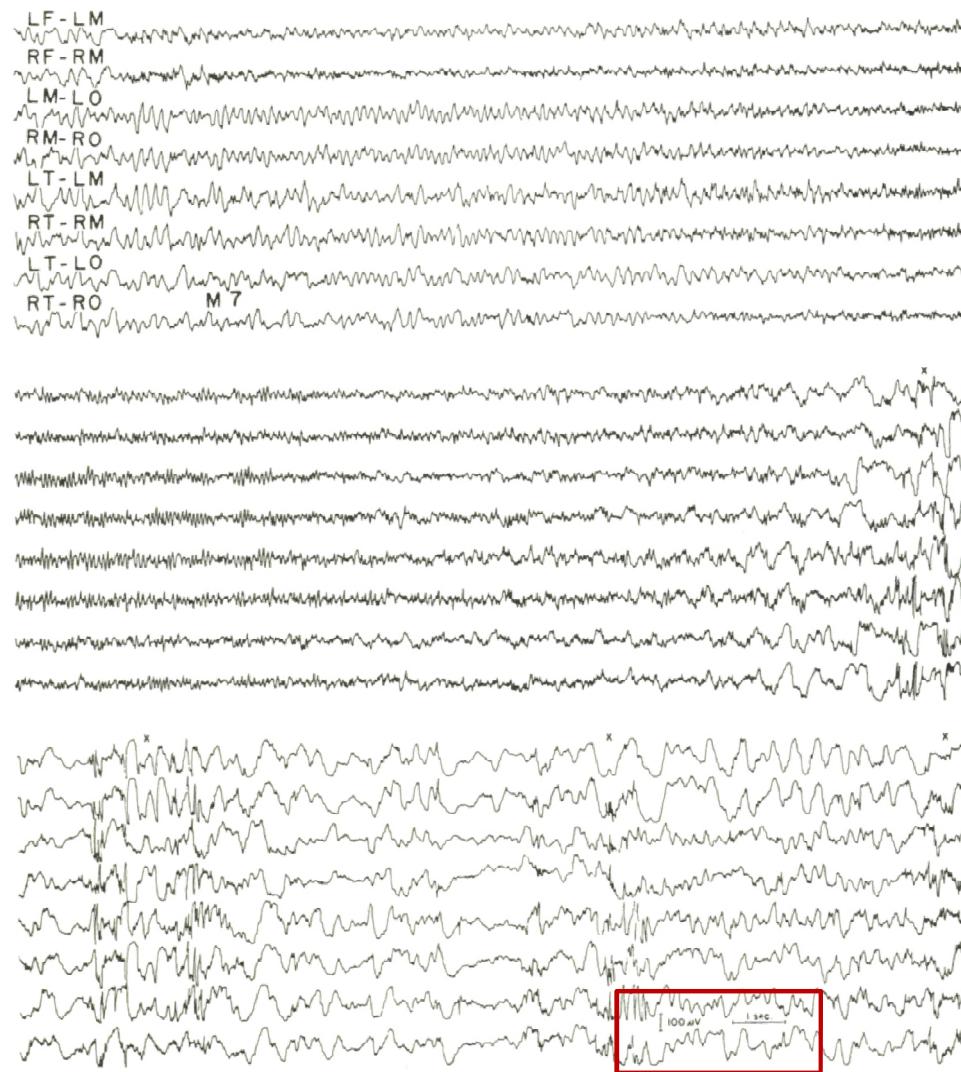
Panels corresponds to adult returns for (fresh-water years.ocean years).



Labels directly on subplots

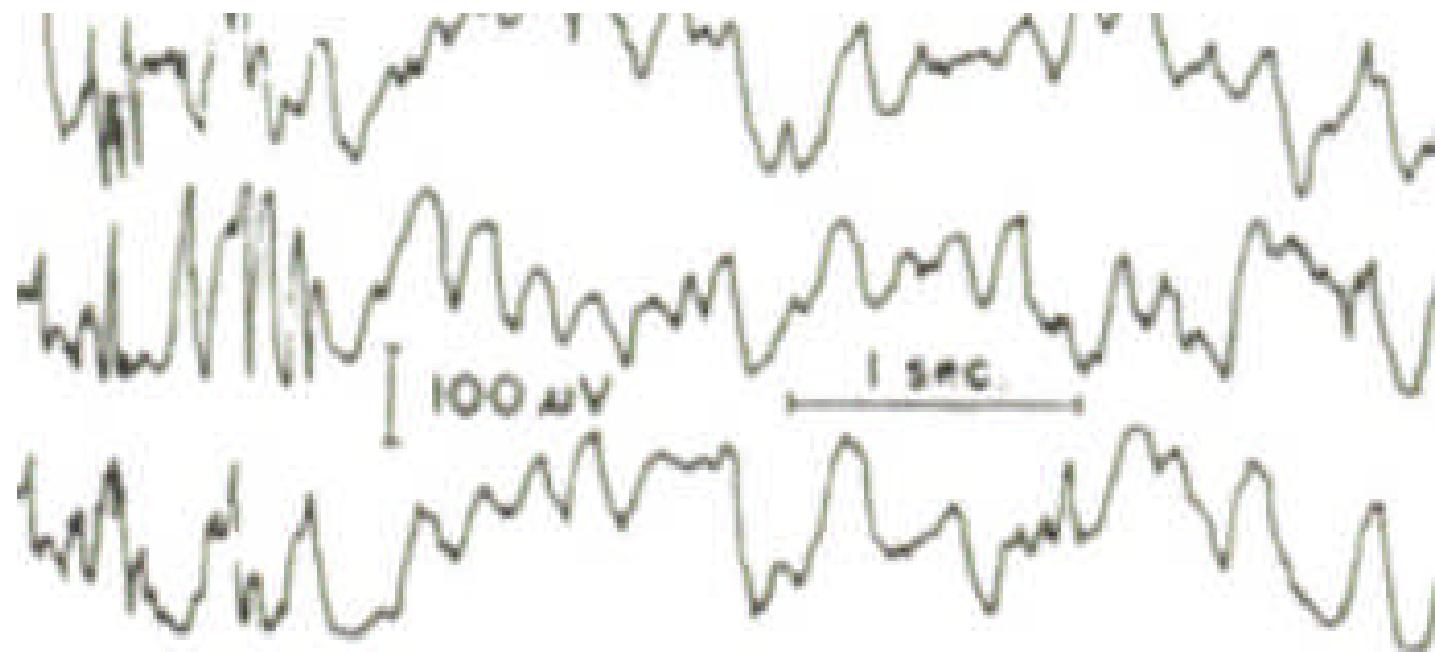
Principles of graphics design

- Above all else show the data
- Maximize the data-ink ratio
- Erase non-data-ink
- Erase redundant data-ink
- Revise and edit



from: Kooi (1971) Fundamentals of electroencephalography, New York, p. 110 Tufte (2001) The visual display of quantitative information, p. 93

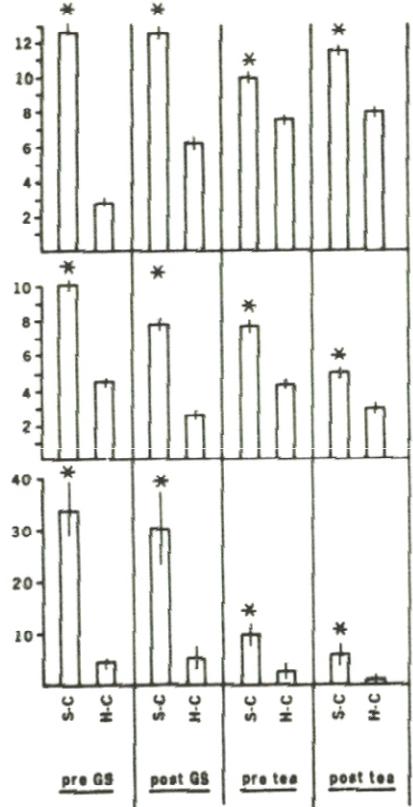
Heart Attack Waveform



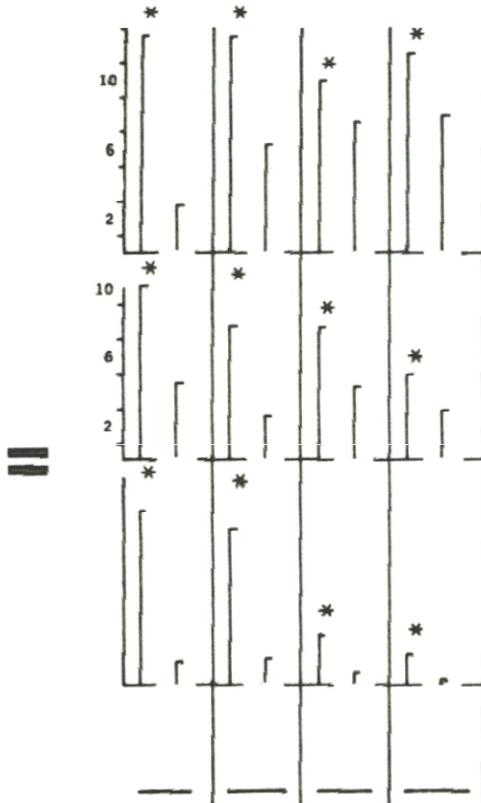
Principles of graphics design

- Above all else show the data
- Maximize the data-ink ratio
- Erase non-data-ink
- Erase redundant data-ink
- Revise and edit

Barplots 1



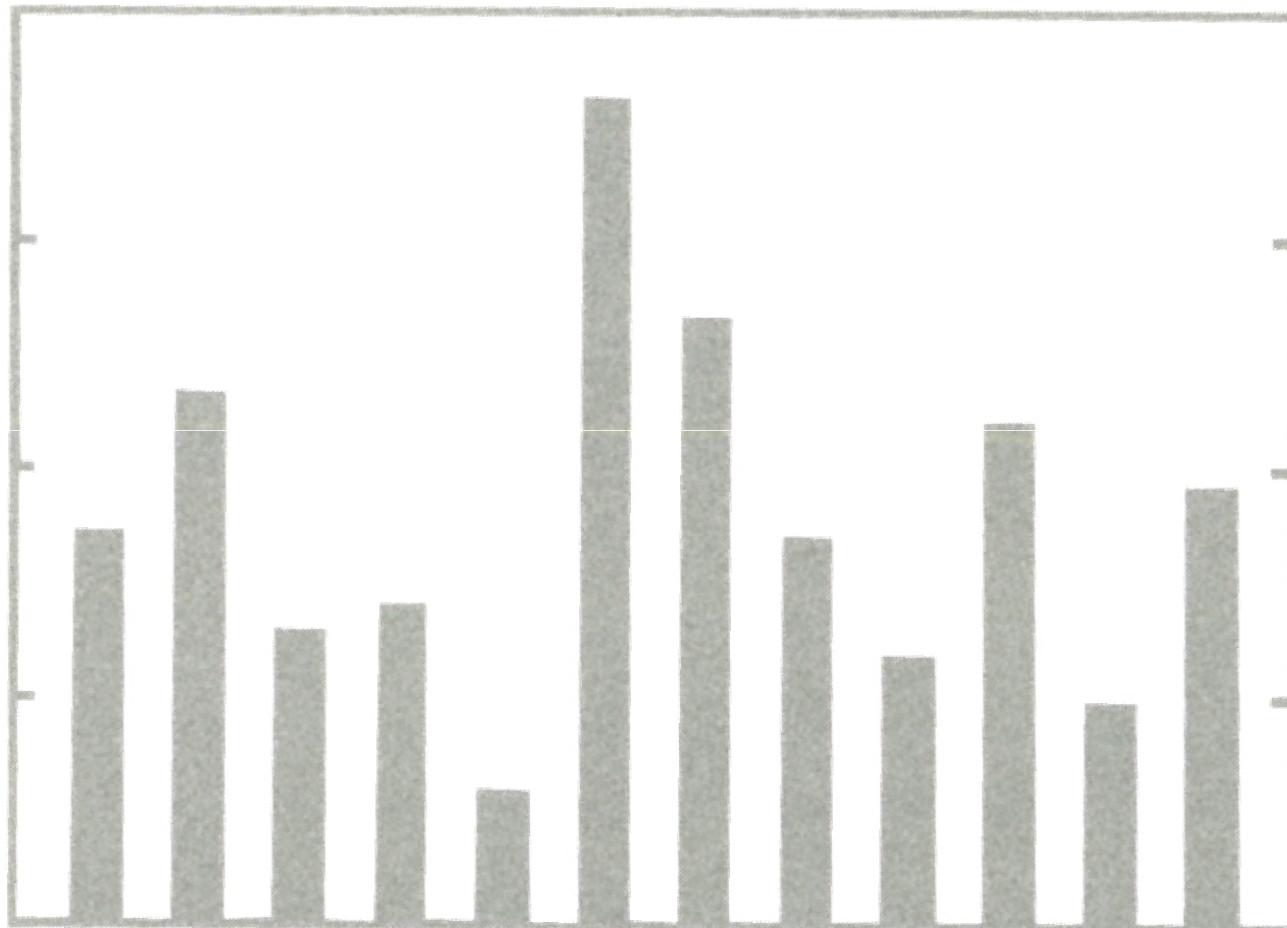
Kuznicki & McCutcheon (1979)
Journal of Experimental
Psychology: General, 108:76



Tufte (2001) The visual display of quantitative information, p. 102

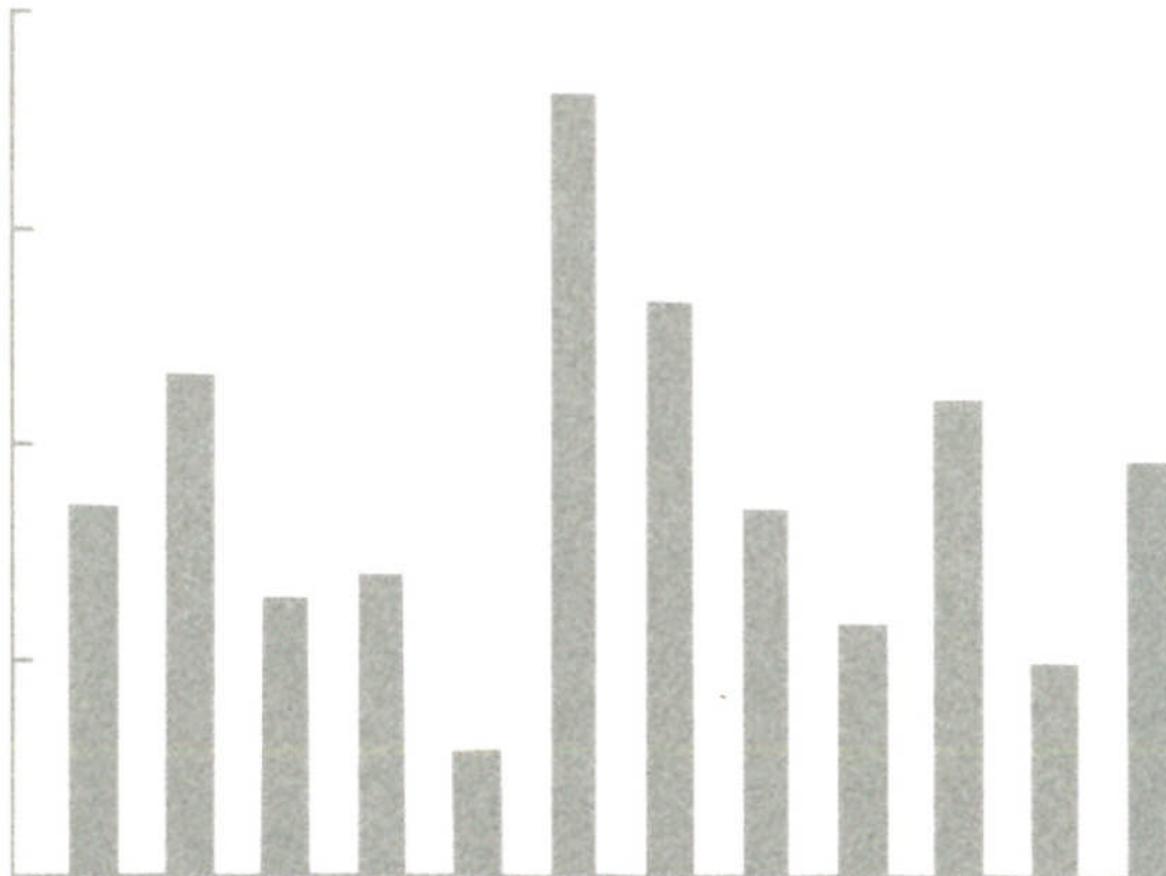


Barplots 2



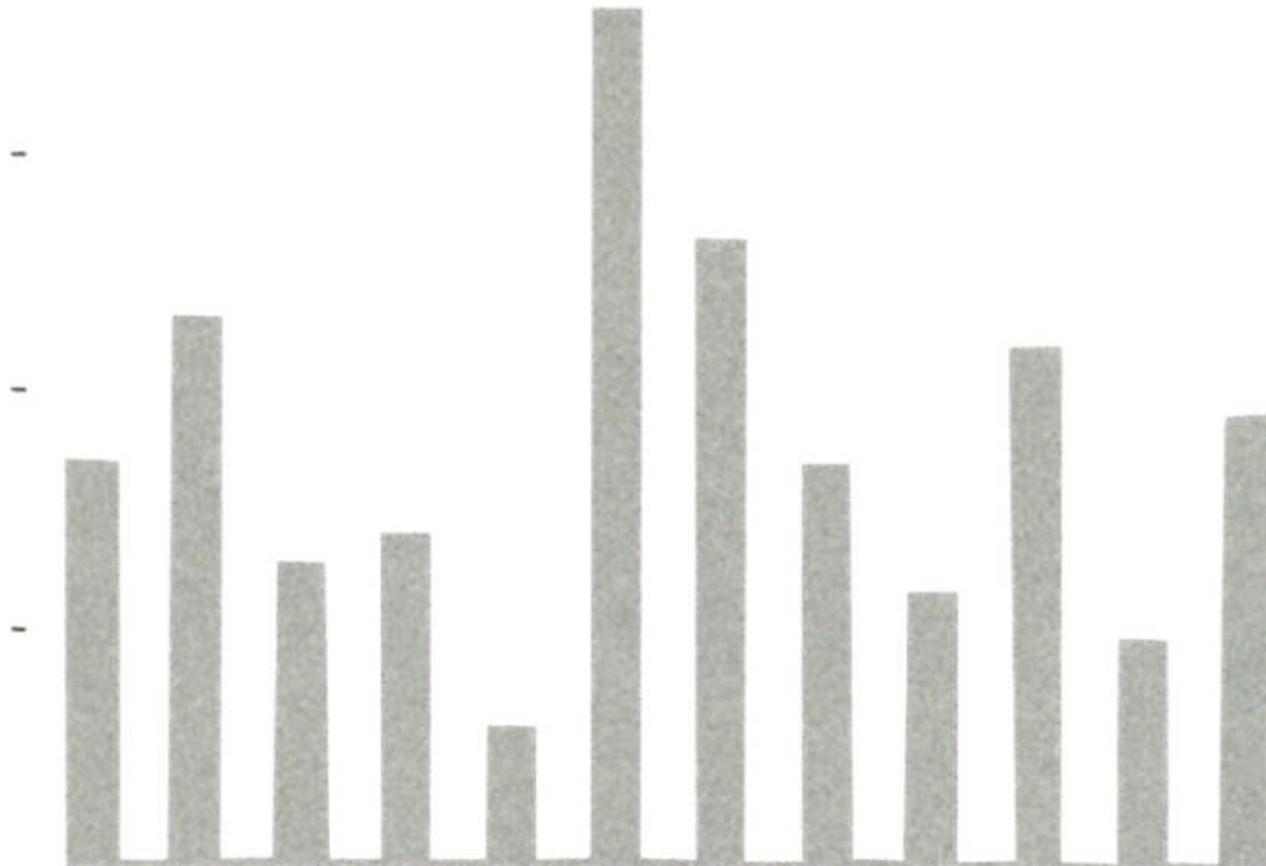
Tufte (2001) The visual display of quantitative information, p. 126-128

Erase the box



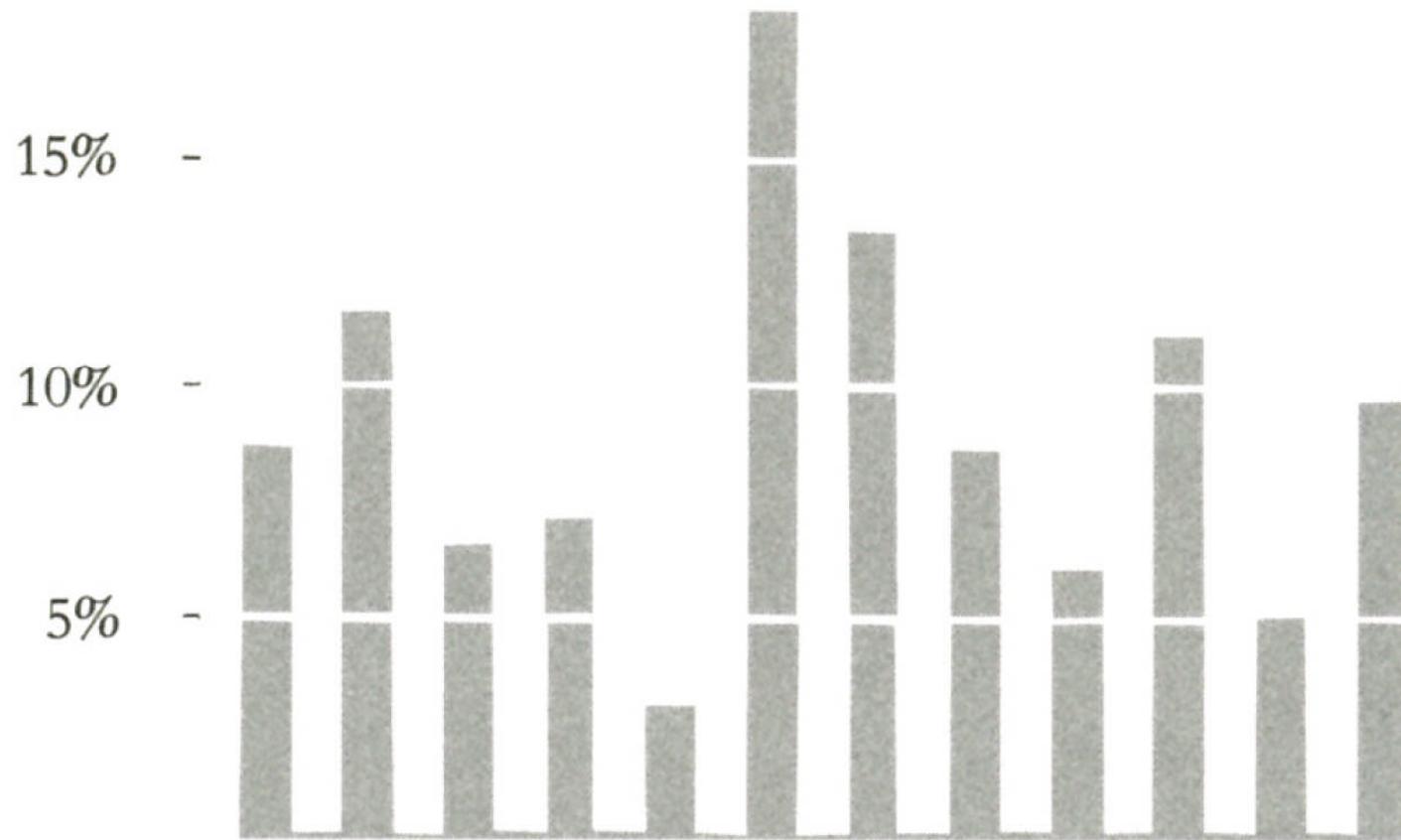
Tufte (2001) The visual display of quantitative information, p. 126-128

Leave only the tick marks



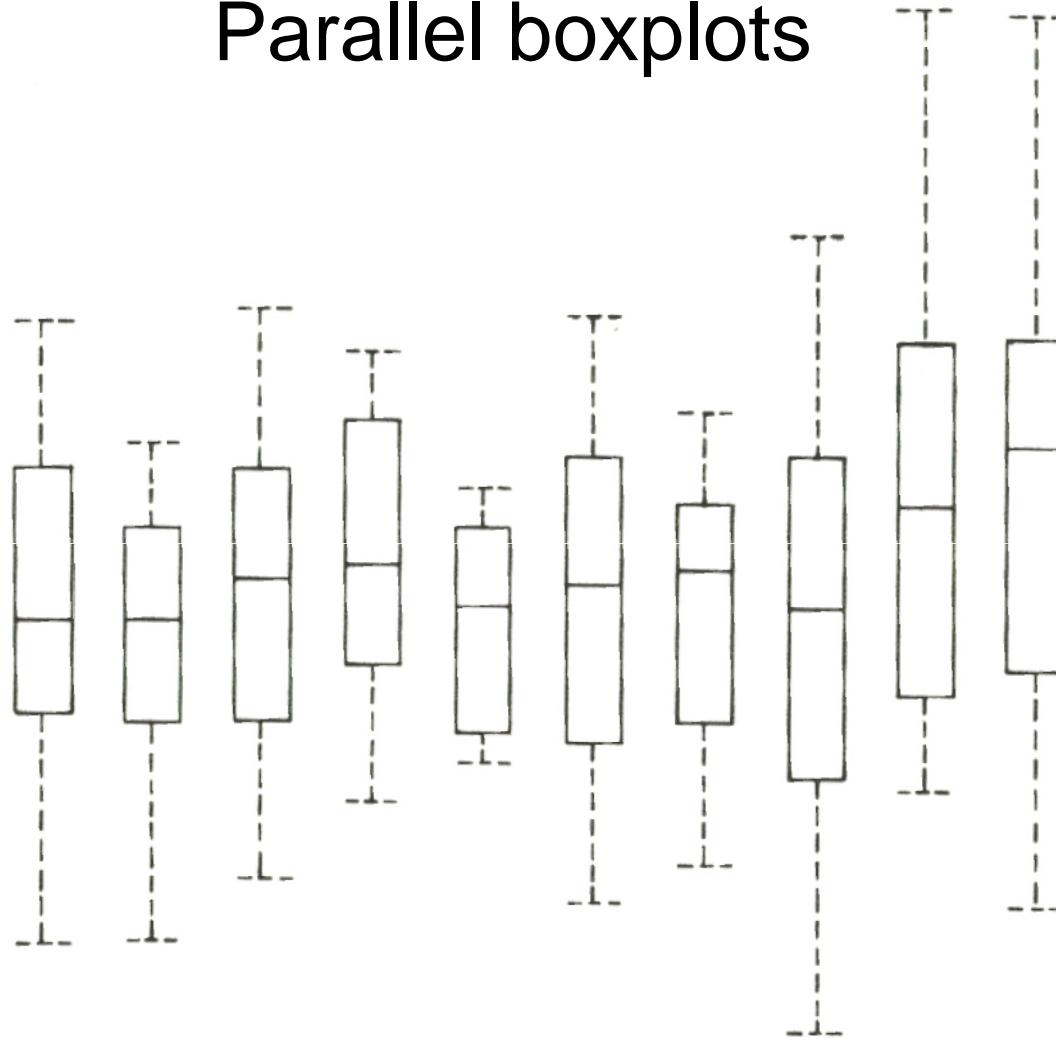
Tufte (2001) The visual display of quantitative information, p. 126-128

Erase some data to make a white grid



Tufte (2001) The visual display of quantitative information, p. 126-128

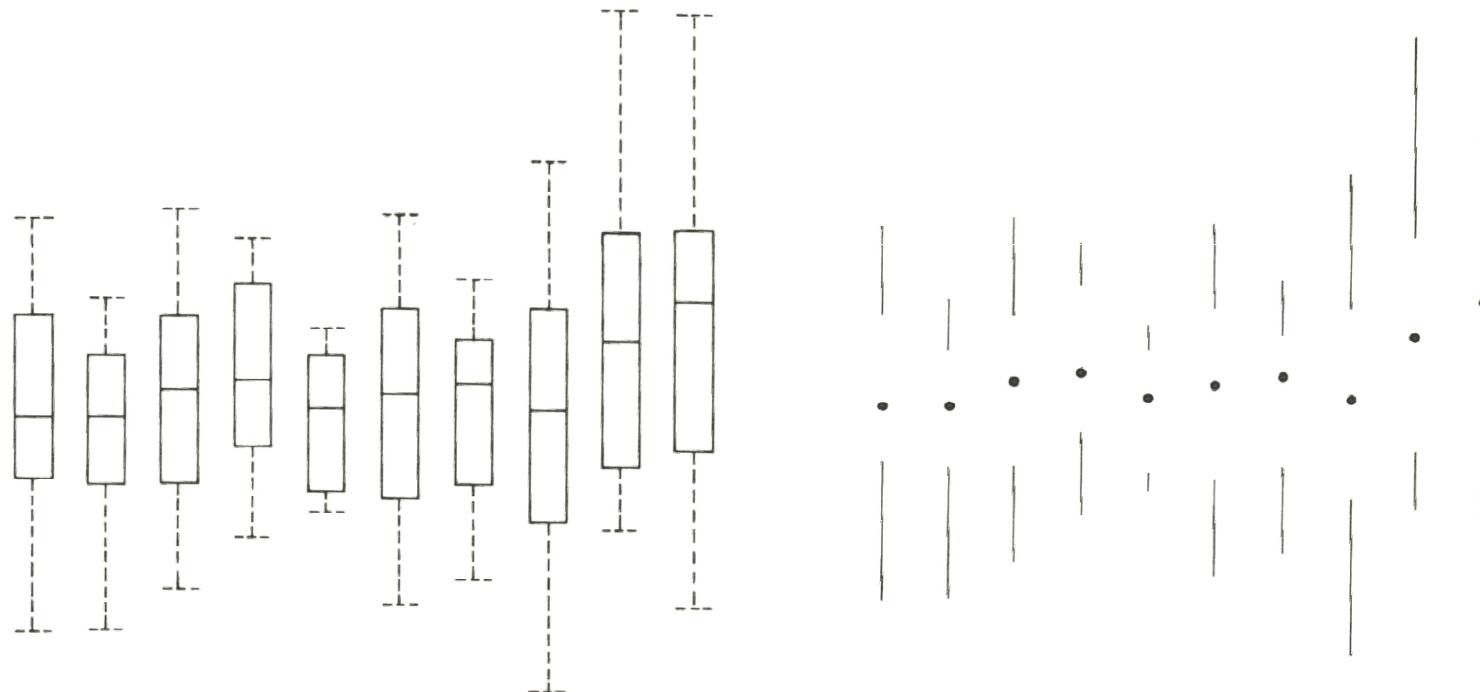
Parallel boxplots



Tufte (2001) The visual display of quantitative information, p. 125

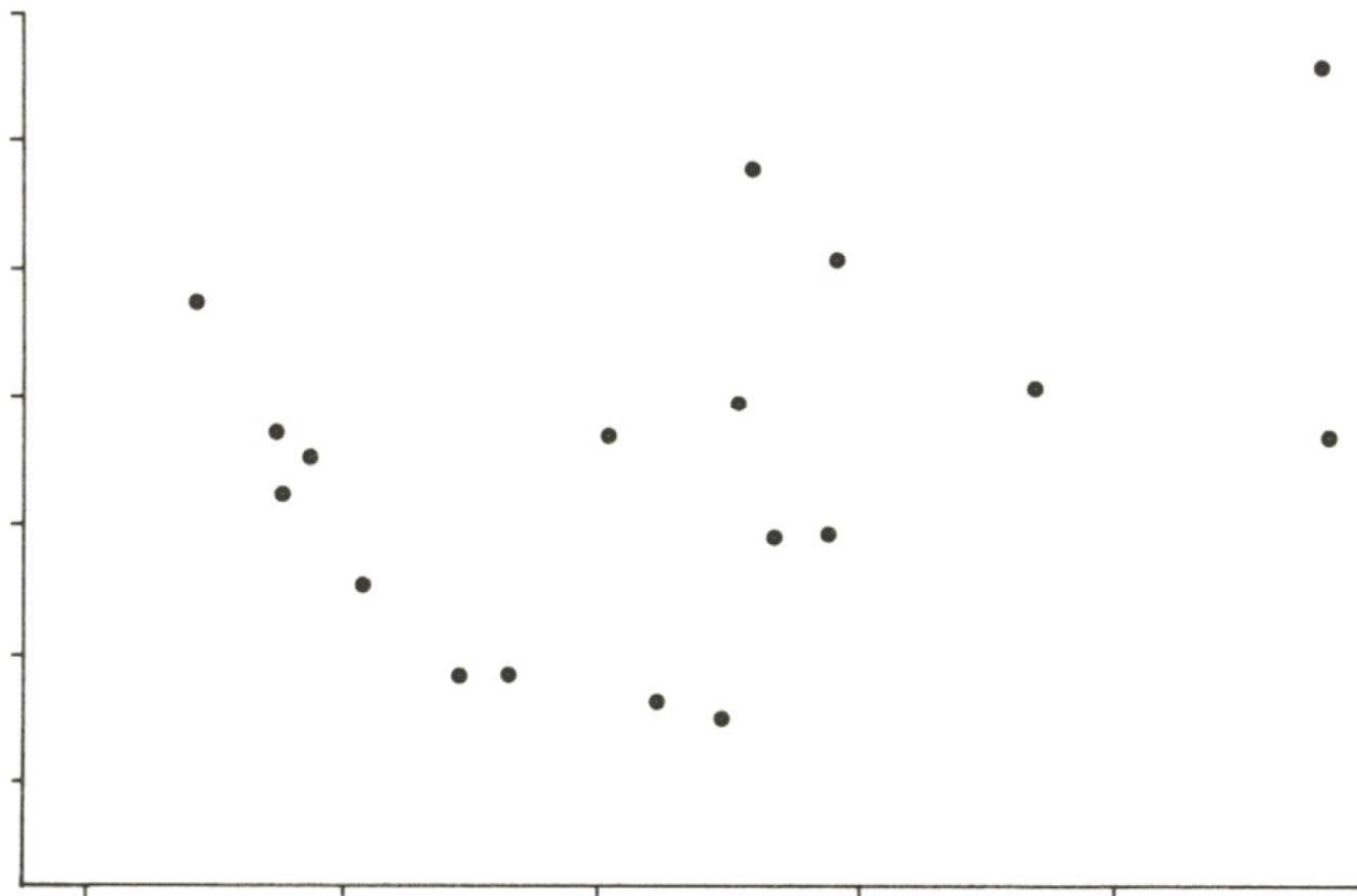
Parallel boxplots

“Original has 50 horizontals and 30 vertical lines,
revised needs only 10 verticals to show the same data”



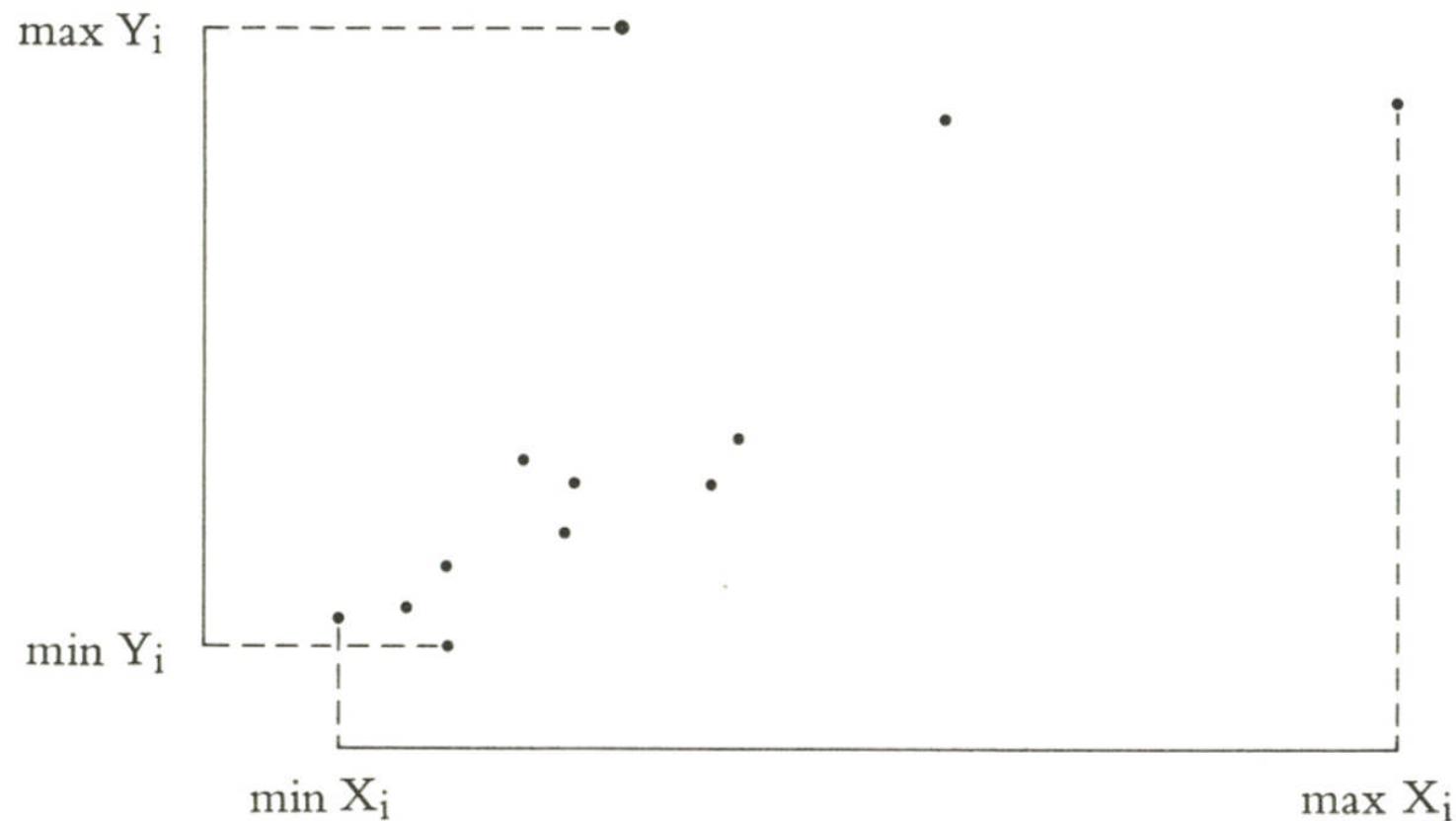
Tufte (2001) The visual display of quantitative information, p. 125

Scatterplots



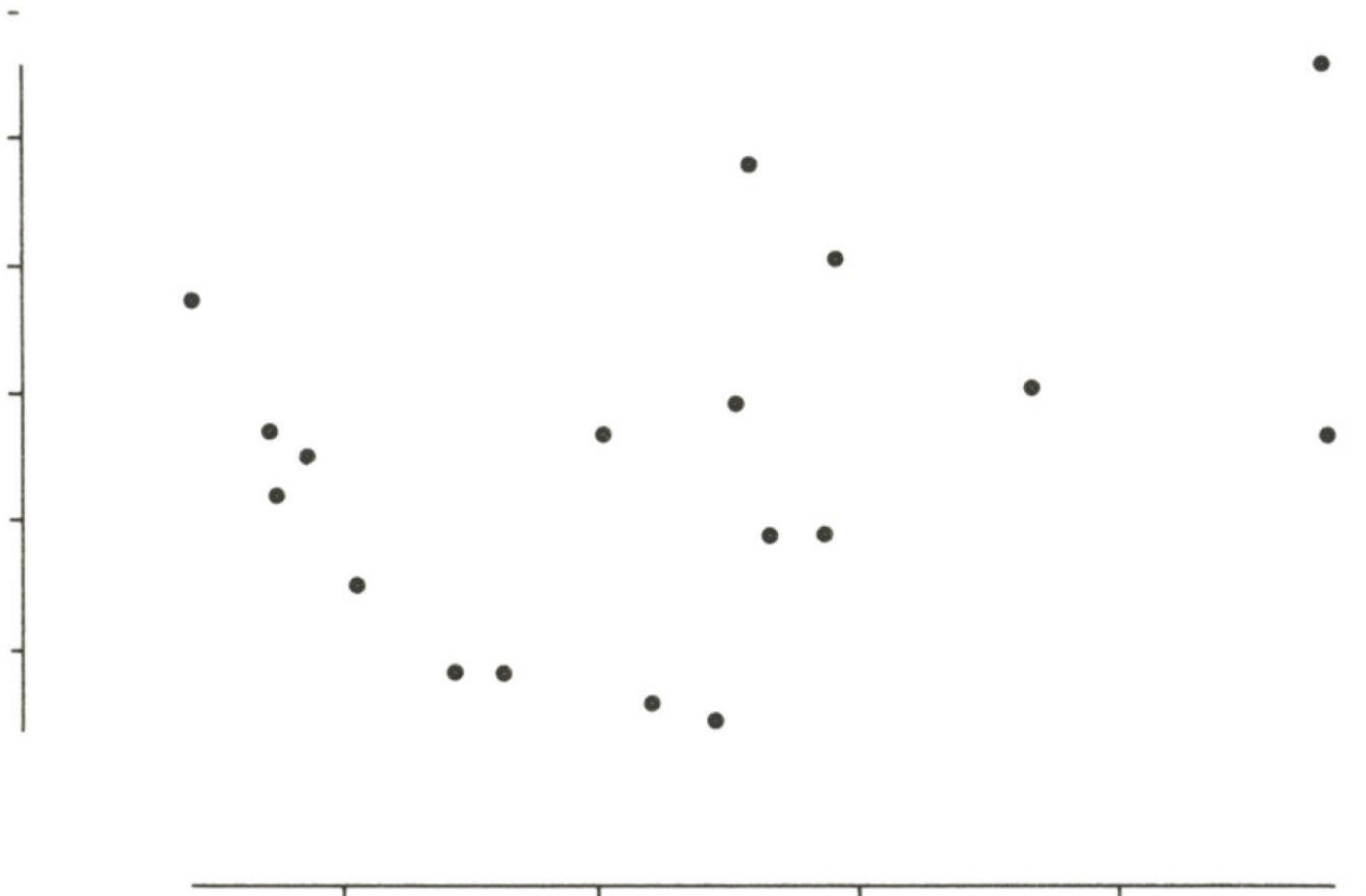
Tufte (2001) The visual display of quantitative information, p. 130-133

Scatterplots



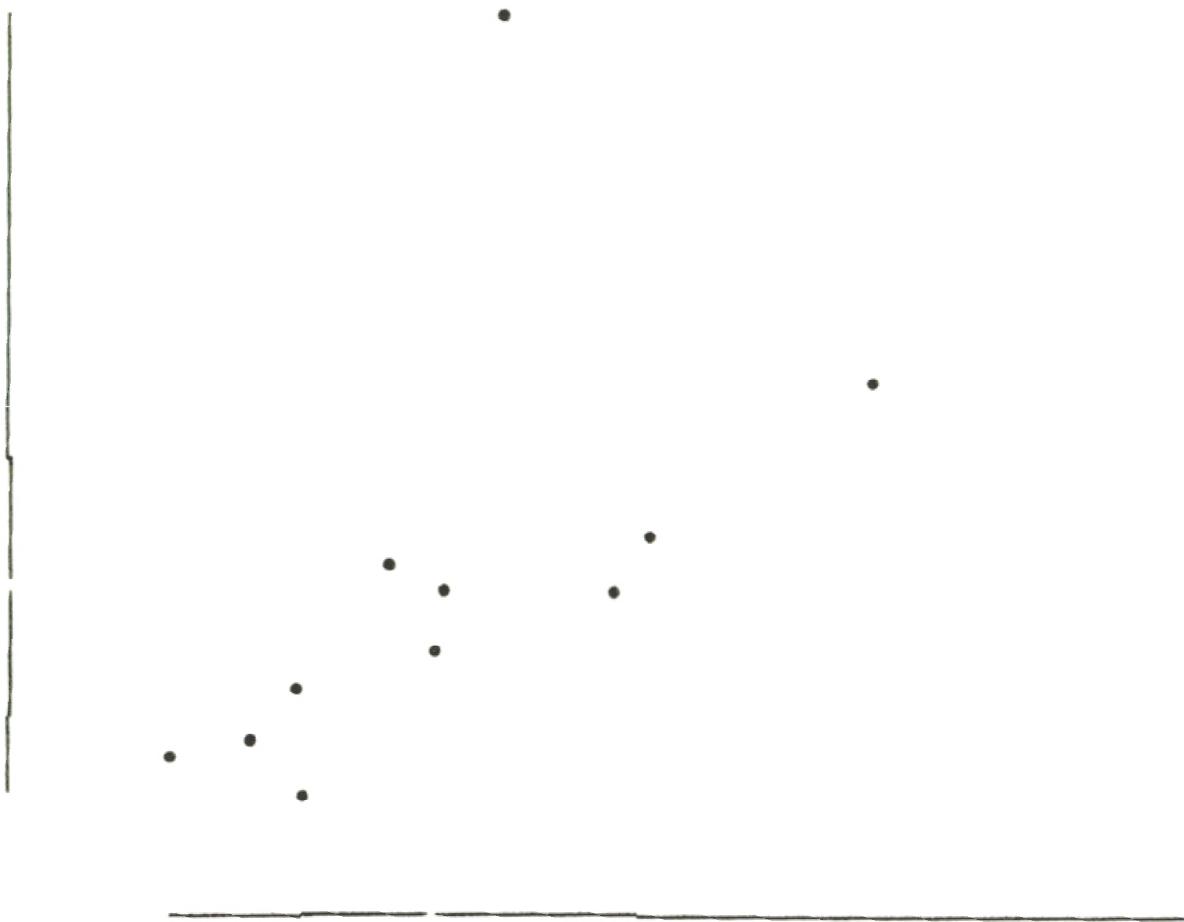
Tufte (2001) The visual display of quantitative information, p. 130-133

Scatterplots

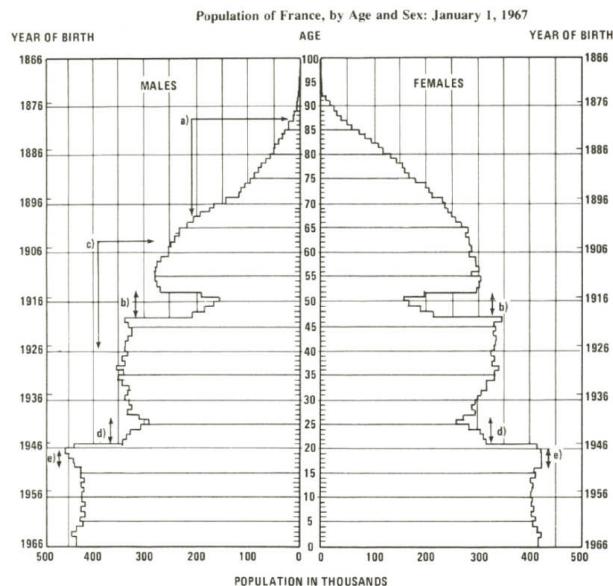


Tufte (2001) The visual display of quantitative information, p. 130-133

Turn axes into quartiles

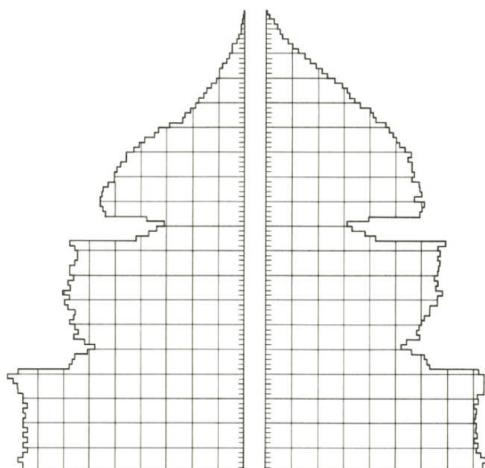


Tufte (2001) The visual display of quantitative information, p. 133



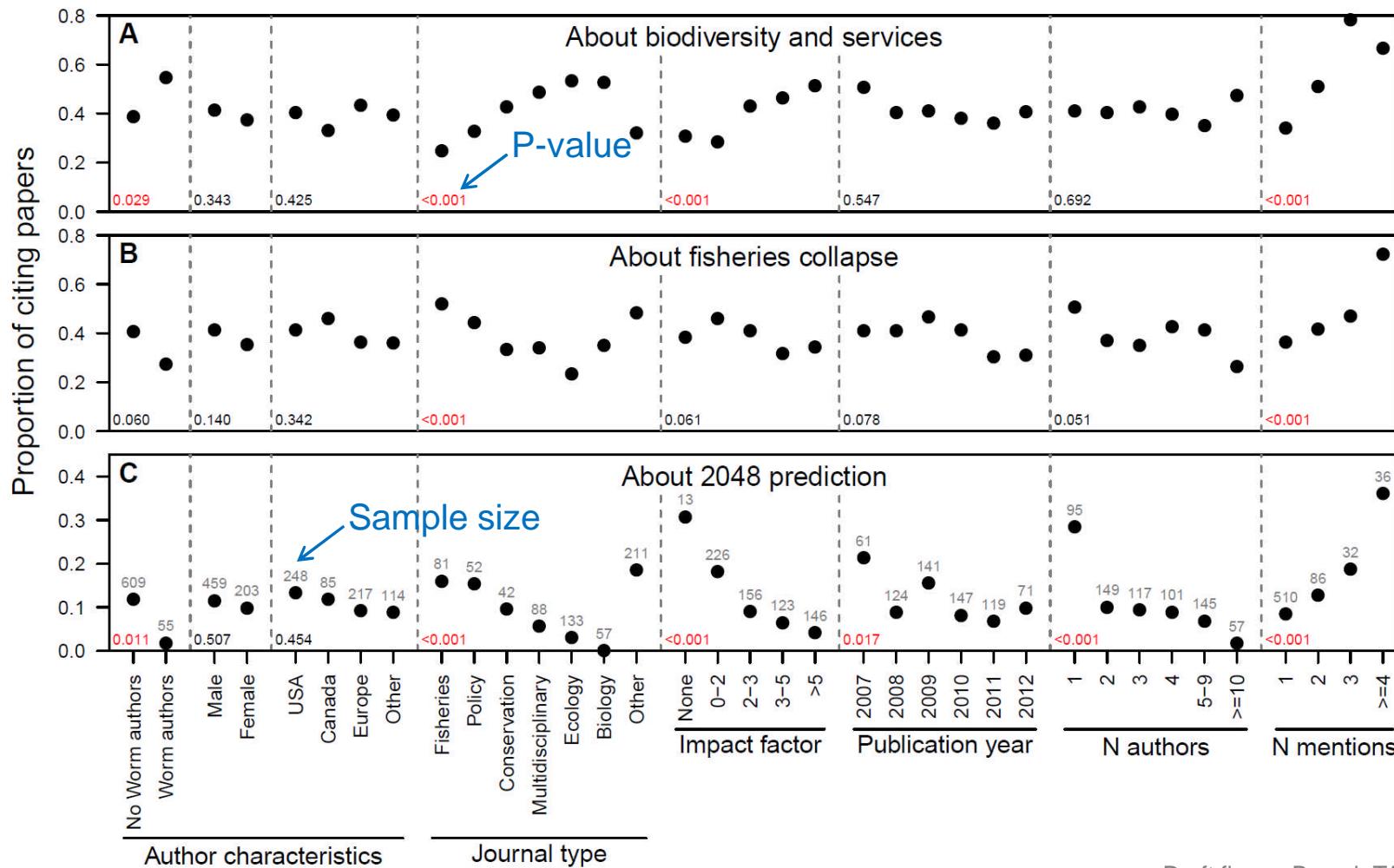
Age distribution of the population of France

A revision quiets the grid and gives emphasis to the data:



Tufte (2001) The visual display of quantitative information, p. 113

Graphs of "How do people cite Worm et al." (2006)?



Draft figure: Branch TA
Analysis of Worm et al. (2006) Science 314:787-790

How to lie with statistics

A Statistician's Lament

"I've been studying statistics for over 40 years and still don't understand it. The ease with which non-statisticians master it is staggering." —Stephen Senn (Prof. U. Glasgow)

On the Relation Between Statistics and Mathematics

"... Statistics is a science ... and it is no more a branch of mathematics than are physics, chemistry and economics; for if its methods fail the test of experience—not the test of logic—they are discarded" -- Tukey (1953), quoted by Brillinger (2002)

4/2/2016.

How to lie with statistics

- Reusing the same data several times to increase the statistical significance of an effect. (examples: choose H1 after looking at the data, use the data to choose the transformation to apply, etc.)
- Using a statistical test whose assumptions are not satisfied (it can either yield more significant results or less significant results)

How to lie with statistics

- Perform the same experiment several times and only publish the results that go in the direction you want.
- E.g. Adverts by tobacco companies claiming that passive smoking is harmless are done that way.

Misleading plots

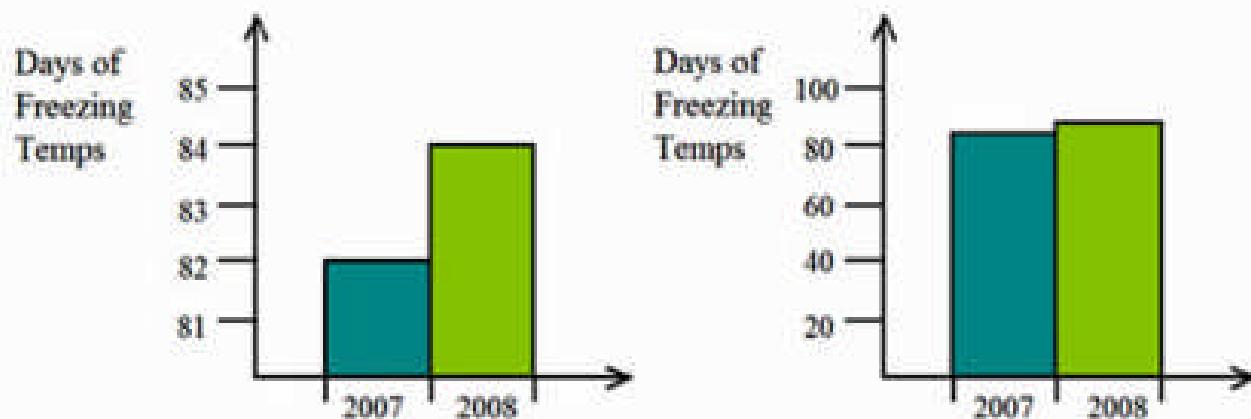
Adding a lot of unnecessary details 3D effects

The most striking is the 3-dimensional pie chart, with the part of interest at the forefront, so that it be deformed and larger)

You can also plot a quantity as a sphere (or any 3-dimensional object) of different scales:
for a quantity twice as large,
you can multiply all the dimensions by 2,
thereby getting a volume eight times as large.

Wrong scale 1

Compare the two graphs

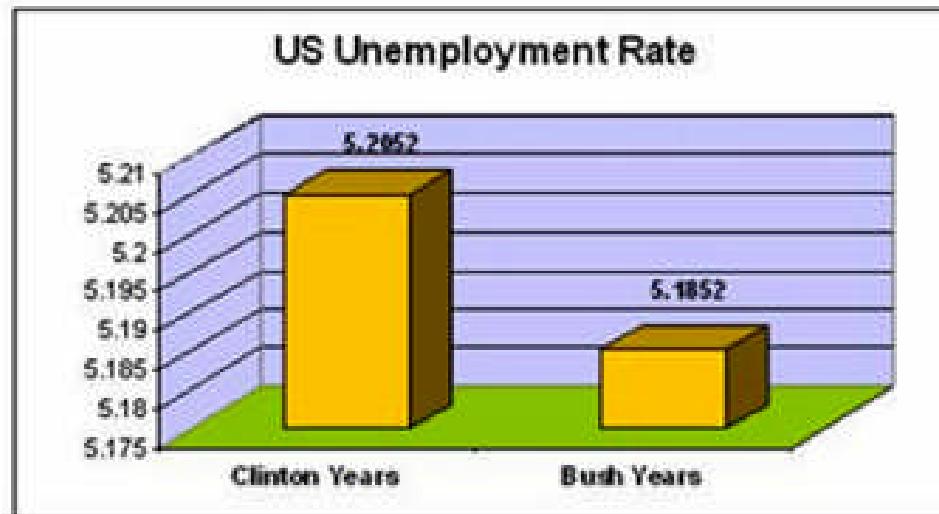


Both show exactly the same data. However, the graph on the left makes the change appear to be much larger than it really is because the numbers on the vertical axis do not start at 0. Each vertical mark on the left graph represents 1 and each mark on the right represents 20 (the scale changes).

Wrong scale 2

The most common "trick" with graphs is to not make the scale start at zero.

This makes small differences look much bigger.



3D Effect

In this Picture Graph, the year 2000 picture is about eight times bigger than the 1960 image. However when we check on the vertical scale, year 2000 is only two and a half times bigger. ($2.5 \times 100 = 250$).

