An Introduction to R for the Geosciences: R Basics & Plotting

Gavin Simpson

Institute of Environmental Change & Society
and
Department of Biology
University of Regina

30th April — 3rd May 2013

Outline

What is R

- The S statistical language was started at Bell Labs on May 5, 1976
- A system for general data analysis jobs that could replace the *ad hoc* creation of Fortran applications
- The S language was licensed by Insightful Corporation for use in their S-PLUS software
- In 2004 Insightful bought the S language from Lucent (formerly AT&T and before that Bell Labs)
- Robert Gentleman and Ross Ihaka designed a language that was compatible with S but which worked in a different way internally
- They called this language R
- There was a lot of interest in R and eventually it was made Open Source under the Gnu GPL-2
- R has drawn around it a group of dedicated stewards of the R software — R Core
- As well a large, vibrant community has developed around R and which contributes the vast number of R packages available on CRAN

Why R?

- Why use a complicated, command-line driven stats package like R?
- R is an Open Source
- Why is Open Source good? Freedom!
- But why R in particular?
  - Well, it is free!
  - R is the *lingua franca* of statistics — a lot of statisticians implement new methodologies and statistical techniques as R code
  - If something doesn't work the way you like, you can change it
  - As R is a programming language you can add your own functions
  - Also, you can use programming to manipulate data and fit a large number of models automatically
  - You can use R scripts and Sweave documents to perform reproducible research
- R works on Linux, Windows and MacOS plus others

R on the Web

- The R homepage is located at: `http://www.r-project.org`
- The download site is called CRAN — the Comprehensive R Archive Network
- CRAN is a series of mirrored web servers to spread the load of thousands of users downloading R and associated packages
- The CRAN master is at: `http://cran.r-project.org`
- The UK mirror is at: `http://cran.uk.r-project.org`

Starting R and other preliminaries

- You start R in a variety of ways depending on your OS
- R starts in a **working directory** where it looks for files and saves objects
- Best to run R in a new directory for each project or analysis task
- `getwd()` and `setwd()` get and set the working directory
- To exit R, the function `q()` is used
- You will be asked if you want to save your workspace; invariably you should answer `n` to this

```
> getwd()

[1] "/home/gavin/work/teaching/macmaster/day1-r-basics-and-plotting"

> setwd("~/work")
> getwd()

[1] "/home/gavin/work"
```

Getting help

- R comes with a lot of documentation
- To get help on functions or concepts within R, use the `"?"` operator
- For help on the `getwd()` function use: `?getwd`
- Function `help.search("foo")` will search through all packages installed for help pages with `"foo"` in them
- How the help is displayed is system dependent
- To search on-line, use `RSiteSearch()`; this opens results in your web browser and includes searching of the R-Help mailing list

```
> help.search("directory")
> RSiteSearch("directory")
```

Working with R; entering commands

- Type commands at prompt ">" and these are evaluated when you hit RETURN
- If a line is not syntactically complete, the prompt is changed to "+"
- If returned object not assigned, it is printed to console
- Assigning the results of a function call achieved by the assignment operator "<-"
- Whatever is on the right of "<-" is assigned to the object named on the left of "<-"
- Enter the name of an object and hit RETURN to print the contents
- `ls()` returns a list of objects currently in your workspace

```
> 5 * 3

[1] 15

> radius <- 5
> pi * radius^2

[1] 78.53982

> ans <- 5 * 3
> ans

[1] 15

> ans2 <- ans + 20
> ans2

[1] 35

> ls()

[1] "ans"    "ans2"   "radius"
```

Basic R object types

- R has several basic object types
  - **vectors** (character, numeric, factors, Date)
  - **matrices** (numeric or character)
  - **data** frames (matrix-like object with components [columns] of different types)
  - **lists** (arbitrary structures that form basis of many returned objects in R)
- Vectors and matrices contain elements of same basic type
- Data frames are more like Excel spreadsheets; each column can contain a different type of data
- But each column contains only a single type of object
- Data frames must also have components (columns) of the same length

Vectors

```
> vec <- c(1, 2, 2.5, 6.2, 4.8, 3.1)
> vec

[1] 1.0 2.0 2.5 6.2 4.8 3.1

> length(vec)

[1] 6

> chr.vec <- c("one", "two", "three")
> chr.vec

[1] "one"   "two"   "three"

> rnd <- rnorm(20)
> rnd

 [1] -1.20706575  0.27742924  1.08444118 -2.34569770  0.42912469
 [6]  0.50605589 -0.57473996 -0.54663186 -0.56445200 -0.89003783
[11] -0.47719270 -0.99838644 -0.77625389  0.06445882  0.95949406
[16] -0.11028549 -0.51100951 -0.91119542 -0.83717168  2.41583518
```

- A vector is a set of 0 or more elements of the same type
- Two main types; character and numeric
- A scalar is a vector of length 1
- When printed, R prepends `"[x]"` to each line - this tells you which element of the vector starts each line
- The `c()` function can be used to create vectors; short for combine or concatenate

Special vectors; factors, Dates

```
> fac <- c("red","blue","green","red","blue","red")
> fac <- factor(fac)
> fac

[1] red   blue  green red   blue  red
Levels: blue green red

> dates <- c("01/11/2007", "10/11/2007", "19/11/2007")
> dates <- as.Date(dates, format = "%d/%m/%Y")
> dates

[1] "2007-11-01" "2007-11-10" "2007-11-19"

> class(dates)

[1] "Date"
```

- Factors are special vectors, used when elements come from a set of possible choices: Male/Female
- R codes these numerically internally, but the labels are easy to read
- Factors can be **ordered**: `ordered()`
- Dates are a special data type and we convert from textual representations into something R understands using `as.Date()`

Matrices

```
> mat <- matrix(1:9, ncol = 3)
> mat

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> chr.mat <- matrix(letters[1:9], nrow = 3)
> chr.mat

     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"

> matrix(1:9, ncol = 3, byrow = TRUE)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> dim(mat)

[1] 3 3
```

- Matrices are vectors with dimensions; numeric or character matrices
- All elements of a matrix must be the same type
- By default R fills matrices by column; use argument `"byrow = TRUE"` to change this
- `dim()` returns the dimensions, rows first then cols

Data frames and lists

```
> (df <- data.frame(Var1 = 1:4, Var2 = letters[1:4], Var3 = factor(c("M","M","F","M")), Var5 = rnorm(4)))

  Var1 Var2 Var3       Var5
1    1    a    M  0.1340882
2    2    b    M -0.4906859
3    3    c    F -0.4405479
4    4    d    M  0.4595894

> (lst <- list(A = 1, B = c("Yes","No"), C = matrix(1:4, ncol = 2)))

$A
[1] 1

$B
[1] "Yes" "No"

$C
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- Data frames are the main object to handle your own data in R
- Like an Excel spreadsheet; each column can be a different type of data
- You can create data frames yourself using `data.frame()`
- Most likely they result from reading your data into R
- Lists generalize data frames; the components of a list can contain any R object

Subsetting

```
> vec[3]

[1] 2.5

> vec[2:5]

[1] 2.0 2.5 6.2 4.8

> vec[-4]

[1] 1.0 2.0 2.5 4.8 3.1

> mat[2, 3]

[1] 8

> df$Var2 #$

[1] a b c d
Levels: a b c d

> lst[["B"]]

[1] "Yes" "No"
```

- Subsetting usually done by `"[...]"`
- For vectors, select elements numerically within `"[...]"`
- Negative indices drop those elements
- Matrices and data frames have rows and columns, subsetting becomes `"[r, c]"`
- Components of data frames and list can be selected by use of $
- Lists also subset using `"[[...]]"`

Sequences and patterned vectors

```
> seq(from = 1, to = 10, by = 2)

[1] 1 3 5 7 9

> 1:5

[1] 1 2 3 4 5

> rep(1:3, each = 2)

[1] 1 1 2 2 3 3

> rep(1:3, times = 3:1)

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

- Sequences and patterned vectors are very useful in some circumstances
- `seq()` is a flexible function to produce sequences of numbers
- `rep()` creates repetitions of its first argument
- The `":"` operator is short hand for `seq(from = x, to = y, by = 1)`

Functions

```
> args(rnorm)

function (n, mean = 0, sd = 1)
NULL

> rnorm(n = 5, mean = 2, sd = 3)

[1] -0.08116074 -2.34461473  3.72426716 -1.07096717  1.95458510

> rnorm(5, 2, 3)

[1] -0.8078458  5.3068926  0.5732208 -0.1283201  0.4962258

> foo <- function(x, y) {(x + y) * 2}
> foo(3, 6)

[1] 18
```

- Functions are R objects that include one or more R function calls
- Encapsulate a set of operations on one or more arguments
- Arguments (options) to functions are entered within `"(...)"`
- Arguments are named and entered in `name = value` pairings
- Don't need to use argument names; but be careful, arguments matched by position

Reading in data from external files

```
> dat <- read.csv("test_file.csv", row.names = 1)
> dat

      Var1 Var2 Var3
Samp1    1    2    3
Samp2    5    6    7
Samp3    3    5    7
Samp4    3    7    9
Samp5    2    3    5

> class(dat)

[1] "data.frame"
```

- R can read from a wide range of file type, connections and databases
- Much of this is beyond the scope of today's workshop
- Easiest to produce spreadsheets in Excel/OpenOffice.Org and save each sheet off as a comma-separated file (*.csv)
- Any labels in row/column 1
- Read data from *.csv file using the `read.csv()` function
- Returns a data frame

Saving objects and writing data out of R

```
> save(dat, file = "test_data_object.rda")
> rm(dat)
> load(file = "test_data_object.rda")
> ls()

 [1] "ans"     "ans2"    "chr.mat" "chr.vec" "dat"     "dates"
 [7] "df"      "fac"     "foo"     "lst"     "mat"     "radius"
[13] "rnd"     "vec"

> write.csv(dat, file = "temp_file2.csv")
> read.csv("temp_file2.csv", row.names = 1)

      Var1 Var2 Var3
Samp1    1    2    3
Samp2    5    6    7
Samp3    3    5    7
Samp4    3    7    9
Samp5    2    3    5
```

- `save()` saves are R object in a compressed, portable format; useful for saving objects that are expensive to produce by don't change regularly
- `load()` used to load saved R objects
- `write.csv()` can be used to write matrix-like objects out as *.csv files

Handling temporal data in R

- R has a rich array of functions, classes and packages available for handling date and time data, *inter alia*
  - `"ts"` class; general object for handling regular time series
  - `"Date"` class; object to hold dates
  - `"POSIXlt"` & `"POSIXct"` classes; object handling calendar dates and times
  - chron package; general date time classes but no timezone or DST
  - zoo, xts packages; general time series packages for handling regular and irregular data
- Built-in time series functions generally assume a `"ts"` class object
- For other uses we need to know how to handle/manipulate data ourselves

The "ts" class

- The basic time series object in R is a "ts" object
- The ts() function is used to create a "ts" object
- We provide the data and some time series properties

```
> head(soi, n = 4)
    SOI
1 -0.62
2 -0.12
3 -0.62
4 -0.65
> ## convert to ts class - data start jan 1886, finish november 2006
> soi.ts <- with(soi, ts(SOI, start = c(1886, 1), end = c(2006, 11),
+                        frequency = 12))
> soi.ts
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
1886 -0.62 -0.12 -0.62 -0.65  0.04 -0.82 -0.34  0.36 -0.18  0.07  1.10 -0.16
1887  0.09 -0.01 -0.09  0.83  0.50 -0.48  0.44  0.34  0.12 -0.56 -0.65 -0.89
1888 -0.16 -0.34 -1.56  0.30 -1.34 -2.20 -0.40 -1.41 -1.23 -1.24 -1.49  0.52
1889 -1.90 -0.26 -0.59  2.12  1.40  1.53  1.42  0.94  0.12  0.85  0.56  0.42
....
```

The `"ts"` class

- `ts()` takes several arguments:
- `data`: the vector (or matrix) of observations
- `start`: the time of the first observation
- `end`: the time of the last observation
- `frequency`: the number of observation per unit time
- `deltat`: the fraction of a unit time between observations
- Annual data, `start = 1986` & `frequency = 1`
- Monthly data, `start = c(1986, 1)` & `frequency = 12` or `deltat = 1/12`
- In general, `start = c(YEAR, FRACTION)` & `frequency = FRACTIONS per YEAR` or `deltat = 1 / FRACTIONS per YEAR`

The "ts" class

Annual Series

```
> obs <- 1:24
> ## 24 years of observations from 1980
> ts1 <- ts(data = obs, start = 1980, frequency = 1)
> ts1
Time Series:
Start = 1980
End = 2003
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

Monthly series

```
> ts2 <- ts(data = obs, start = 1980, frequency = 12)
> ts2
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1980   1   2   3   4   5   6   7   8   9  10  11  12
1981  13  14  15  16  17  18  19  20  21  22  23  24
> ts3 <- ts(data = obs, start = 1980, deltat = 1/12)
> ts3
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1980   1   2   3   4   5   6   7   8   9  10  11  12
1981  13  14  15  16  17  18  19  20  21  22  23  24
```

The "ts" class

Monthly Series — specifying the start part

```
> ## starting in June 1980
> ts4 <- ts(data = obs, start = c(1980, 6), frequency = 12)
> ts4
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1980                       1   2   3   4   5   6   7
1981   8   9  10  11  12  13  14  15  16  17  18  19
1982  20  21  22  23  24
```

Weekly and Quarterly series

```
> (ts5 <- ts(data = obs, start = c(1980, 1), frequency = 52))
Time Series:
Start = c(1980, 1)
End = c(1980, 24)
Frequency = 52
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
> (ts6 <- ts(data = obs, start = c(1980, 1), frequency = 4))
     Qtr1 Qtr2 Qtr3 Qtr4
1980    1    2    3    4
1981    5    6    7    8
....
```

The R "Date" class

- The "Date" class is the basic in-built object for storing date data
- Takes a vector as input, plus other arguments. For "character" input need to supply a format
- format is a string describing how the parts of the input dates are structured
- format uses standard place holders in place of the actual date-parts
- Some commonly used place holders are
  - "%d": the day of the month as a decimal (01–31)
  - "%m": the month of the year as a decimal (01–12)
  - "%y": the year without century (00–99; 00–68 20XX; 69–99 19XX)
  - "%Y": the year with century
  - "%b": the abbreviated month name
  - "%B": the full month name
- Exactly what some of the place holders mean is locale dependent
- See ?strftime for details

The R "Date" class

"Date" examples

```
> ## Date class
> dates <- c("01/01/2012","02/01/2012","03/01/2012")
> c(Dates <- as.Date(dates, format = "%d/%m/%Y"))
[1] "2012-01-01" "2012-01-02" "2012-01-03"
> class(Dates)
[1] "Date"
>
> (dates2 <- paste(01:03, month.abb[1:3], 2012, sep = "-"))
[1] "1-Jan-2012" "2-Feb-2012" "3-Mar-2012"
> (Dates2 <- as.Date(dates2, format = "%d-%b-%Y"))
[1] "2012-01-01" "2012-02-02" "2012-03-03"
>
> (dates3 <- paste(11:13, 15:17, month.name[1:3], sep = "_"))
[1] "11_15_January"  "12_16_February" "13_17_March"
> (Dates3 <- as.Date(dates3, format = "%y_%d_%B"))
[1] "2011-01-15" "2012-02-16" "2013-03-17"
```

The R "Date" class

Working with "Date" objects

```
> ## Process the Dates3 Date object
> format(Dates3, format = "%B")
[1] "January"  "February" "March"
> format(Dates3, format = "%Y")
[1] "2011" "2012" "2013"
> ## as above but coerce to numeric
> as.numeric(format(Dates3, format = "%Y"))
[1] 2011 2012 2013
> format(Dates3, format = "%j") ## Julian day
[1] "015" "047" "076"
>
> ## Can use Date objects with math operators
> Dates
[1] "2012-01-01" "2012-01-02" "2012-01-03"
> Dates + 1
[1] "2012-01-02" "2012-01-03" "2012-01-04"
>
> ## seq() method for sequences of dates
> seq(from = Dates[1], by = "2 days", length = 4)
[1] "2012-01-01" "2012-01-03" "2012-01-05" "2012-01-07"
```

The R "POSIXlt" & "POSIXct" classes

- The "POSIXlt" & "POSIXct" classes are the basic in-built object for storing date-time data
- Takes a vector as input, plus other arguments. For "character" input need to supply a `format`
- `format` is a string describing how the parts of the input date-times are structured
- Some commonly used place holders are (in addition)
  - "%H": the hours as decimal (00–23)
  - "%M": the minutes as a decimal (00–59)
  - "%S": the seconds as a decimal (00–61, inc leap seconds)
  - "%I": the hours as a decimal (01-12)
  - "%p": AM/PM indicator for use with "%I"
- Exactly what some of the place holders mean is locale dependent
- See `?strftime` for details

The R "POSIXlt" & "POSIXct" classes

Working with "POSIXt" objects

```
> ## POSIXt classes
> (now <- Sys.time())
[1] "2012-01-23 17:58:55 GMT"
> (times <- now + 1:5)
[1] "2012-01-23 17:58:56 GMT" "2012-01-23 17:58:57 GMT"
[3] "2012-01-23 17:58:58 GMT" "2012-01-23 17:58:59 GMT"
[5] "2012-01-23 17:59:00 GMT"
> class(now)
[1] "POSIXct" "POSIXt"
> (input <- format(times, format = "%d/%m/%Y %H-%M-%S"))
[1] "23/01/2012 17-58-56" "23/01/2012 17-58-57"
[3] "23/01/2012 17-58-58" "23/01/2012 17-58-59"
[5] "23/01/2012 17-59-00"
> (times2 <- as.POSIXct(input, format = "%d/%m/%Y %H-%M-%S", tz = "GMT"))
[1] "2012-01-23 17:58:56 GMT" "2012-01-23 17:58:57 GMT"
[3] "2012-01-23 17:58:58 GMT" "2012-01-23 17:58:59 GMT"
[5] "2012-01-23 17:59:00 GMT"
> all.equal(times, times2) ## yes, all bar timezone attr
[1] "Attributes: < Length mismatch: comparison on first 1 components >"
```

The R "POSIXlt" & "POSIXct" classes

- "POSIXct": numeric vector containing the number of seconds since beginning of 1970 UTC
- "POSIXlt": a list of date-time components, each a vector, encoding the date time data

```
> str(unclass(times))
 num [1:5] 1.33e+09 1.33e+09 1.33e+09 1.33e+09 1.33e+09
> str(unclass(as.POSIXlt(times)))
List of 9
 $ sec  : num [1:5] 18 19 20 21 22
 $ min  : int [1:5] 39 39 39 39 39
 $ hour : int [1:5] 20 20 20 20 20
 $ mday : int [1:5] 23 23 23 23 23
 $ mon  : int [1:5] 0 0 0 0 0
 $ year : int [1:5] 112 112 112 112 112
 $ wday : int [1:5] 1 1 1 1 1
 $ yday : int [1:5] 22 22 22 22 22
 $ isdst: int [1:5] 0 0 0 0 0
 - attr(*, "tzone")= chr [1:3] "" "GMT" "BST"
```

The zoo package

- The zoo package was developed by Achim Zeileis and (later) Gabor Grothendeick
- Designed to provide a general time series class for ordered data in R that was independent of the index class used to define the ordering
- Allows conversion of `"zoo"` objects to other classes and packages (`"ts"`, `"its"`, `"irts"`, `"timeSeries"`)
- Handles irregular time series as easily as regular ones
- Key requirements are a vector or matrix of data and an indexing variable
- The xts package builds upon zoo adding extra functionality

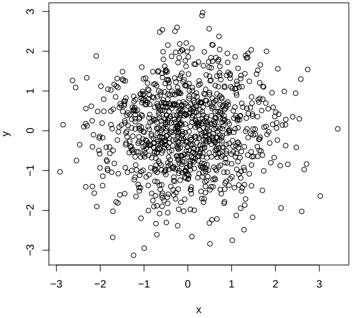
The zoo package

```
> require(zoo) ## load zoo package
> ## date sequence and a random walk
> dseq <- seq(Sys.Date(), by = "days", length = 40)
> set.seed(42)
> tdat <- cumsum(rnorm(40))
> tdat.z <- zoo(tdat, dseq) ## create a zoo object
> head(tdat.z)
2012-01-23 2012-01-24 2012-01-25 2012-01-26 2012-01-27 2012-01-28
 1.3709584  0.8062603  1.1693887  1.8022513  2.2065196  2.1003951
> plot(tdat.z)  ## plot method
> as.ts(tdat.z) ## can go between zoo and ts for example
Time Series:
Start = 15362
End = 15401
Frequency = 1
 [1]  1.3709584  0.8062603  1.1693887  1.8022513  2.2065196  2.1003951
 [7]  3.6119171  3.5172581  5.5356818  5.4729677  6.7778373  9.0644827
[13]  7.6756220  7.3968333  7.2635119  7.8994623  7.6152094  4.9587540
[19]  2.5182870  3.8384004  3.5317618  1.7504534  1.5785360  2.7932107
[25]  4.6884042  4.2579350  4.0006657  2.2375026  2.6975999  2.0576050
[31]  2.5130552  3.2178925  4.2529960  3.6440697  4.1490248  2.4320161
[37]  1.6475571  0.7966495 -1.6175582 -1.5814355
```

Graphics in R

```
> x <- rnorm(1000)
> y <- rnorm(1000)
> plot(x, y)
```

- Standard plotting command is `plot()`
- Takes one or two arguments of coordinates
- By default draws a scatterplot
- R's graphics are like drawing with pen on paper; once you draw anything that sheet of paper is no-longer pristine and you can't erase anything you have drawn
- Vast array of parameters to alter look of plots; see `?par`

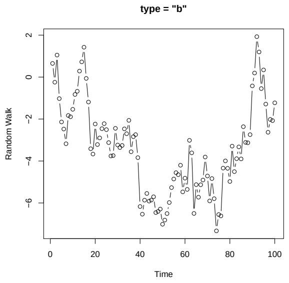


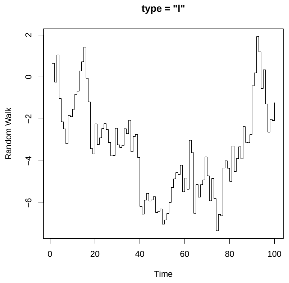
Graphics in R

```
> x <- 1:100
> y <- cumsum(rnorm(100))

> plot(x, y, type = "b", main = "type = \"b\"", xlab = "Time",
+      ylab = "Random Walk")
> plot(x, y, type = "s", main = "type = \"l\"", xlab = "Time",
+      ylab = "Random Walk")
```

- The "type" argument changes the type of plotting done
  - type = "p" draws points
  - type = "l" draws lines
  - type = "o" draws lines and points over-plotted
  - type = "b" draws lines and points
  - type = "h" draws histogram-like bars
  - type = "s" draws stepped lines
- "main" control the title of the plot
- "xlab" & "ylab" control axis labels



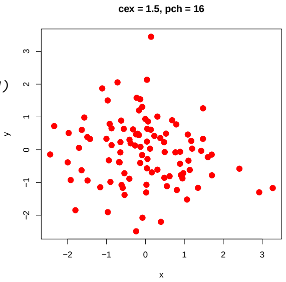


Graphics in R

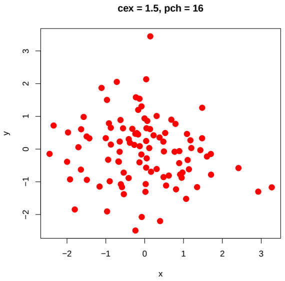
```
> x <- rnorm(100)
> y <- rnorm(100)

> plot(x, y, main = "cex = 1.5, pch = 16", cex = 1.5, pch = 19, col = "red")

> plot(x, y, cex = 1.5, pch = 19, col = "red", axes = FALSE, ann = FALSE)
> axis(side = 1)
> axis(side = 2)
> title(main = "cex = 1.5, pch = 16", xlab = "x", ylab = "y")
> box()
```



- "pch" controls the plotting character
- "cex" controls the size of the character
- "col" controls colour
- "axes" logical; should axes be drawn
- "ann" logical; should the plot be annotated
- axis(), title(), box() used to build up plotting
- Allows finer control



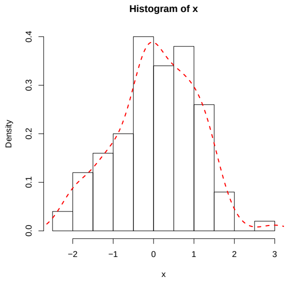
Graphics in R

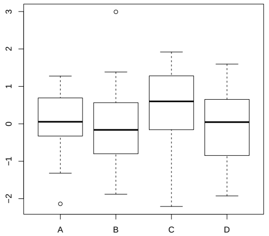
```
> x <- rnorm(100)
> grps <- factor(sample(LETTERS[1:4], 100, replace = TRUE))

> dens <-density(x)
> hist(x, freq = FALSE)
> lines(dens, col = "red", lwd = 2, lty = "dashed")

> boxplot(x ~ grps)
```

- `hist()` draws histograms
- `boxplot()` draws boxplots
- `"lwd"` controls the line width
- `"lty"` controls the line type
- `lines()` used to add lines to an existing plot
- Also `points()`





R Packages

```
> install.packages("vegan")
> update.packages()
> library("vegan")
> require("vegan")
```

- CRAN contains hundreds of packages of user-contributed code that you can install from an R session
- Package installation via function `install.packages()`
- Packages can be updated via function `updates.packages()`
- When installing or updating for the first time in a session, R will prompt you to choose a mirror to download from
- Once a package is installed you need to load it ready for use
- Load a package from your library using `library()` or `require()`
- Windows and MacOS have menu items to assist with these operations

R Package Management

- It is useful to create your own library for downloaded packages
- This library will not be overwritten when you install a new version of R
- To set a directory you have write permissions on as your user library, create a file named `".Renviron"` in your home directory
  - On Windows this is usually `C:\Documents and Settings\username\My Documents`
  - On Linux it is `/home/user/`
- To set your user library to stated directory, add following to your `".Renviron"`
  - On Windows if installed R to `C:\R` add: `R_LIBS=C:/R/myRlib`
  - On Linux, create directory `/home/user/R/libs` say and then add: `R_LIBS=/home/user/R/libs`